

# LIBXS

LIBXS is a portable C library providing building blocks for memory operations, numerics, synchronization, and more -- with a focus on performance and minimal dependencies. Targets x86-64, AArch64, and RISC-V; requires only a C89 compiler. Originally developed as part of LIBXSMM.

## Functionality

Domain	Header	Description
Permutation	<code>libxs_perm.h</code>	Co-prime shuffling, smooth row permutations
Histogram	<code>libxs_hist.h</code>	Thread-safe histogram with running statistics
Registry	<code>libxs_reg.h</code>	Thread-safe key-value store with per-thread caching
Hashing	<code>libxs_hash.h</code>	CRC32-based hashing, Adler-32, string hashing
Predict	<code>libxs_predict.h</code>	Fingerprint-guided parameter prediction with model persistence
Malloc	<code>libxs_malloc.h</code>	Pool-based allocator (steady-state, no system calls)
Memory	<code>libxs_mem.h</code>	Byte comparison, matrix copy/transpose, alignment queries
String	<code>libxs_str.h</code>	Edit distance, substring search, word similarity, formatting
Timer	<code>libxs_timer.h</code>	High-resolution timing via calibrated TSC
CPUID	<code>libxs_cpuid.h</code>	CPU feature detection (SSE to AVX-512, AArch64, RISC-V)
Utils	<code>libxs_utils.h</code>	ISA feature gates, bit-scan, SIMD helpers
Sync	<code>libxs_sync.h</code>	Portable atomics, locks, TLS, and file locking
GEMM	<code>libxs_gemm.h</code>	Batched dense GEMM (strided, pointer-array, grouped)
Math	<code>libxs_math.h</code>	Matrix comparison, GCD/LCM, coprime, BF16 conversion
MHD	<code>libxs_mhd.h</code>	Read/write MetaImage (MHD/MHA) files
RNG	<code>libxs_rng.h</code>	Thread-safe pseudo-random number generation (SplitMix64)

See also: Fortran Interface, Scripts.

## Build

```
make GNU=1
```

The library is compiled for SSE4.2 by default but dynamically dispatches to the best ISA available at runtime (up to AVX-512). Use `SSE=0` to compile natively for the build host.

Variable	Default	Description
GNU	0	Use GNU GCC-compatible compiler
DBG	0	Debug build
SYM	0	Include debug symbols (-g)
SSE	1	x86 baseline: 0=native, 1=SSE4.2 (portable)

CMake is also supported (header-only or library):

```
cmake -S . -B build -DLIBXS_HEADER_ONLY=ON
cmake --build build
```

## Installation

Install into a chosen prefix:

```
make GNU=1 -j $(nproc) install PREFIX=$HOME/libxs
```

This installs headers, the Fortran module, the static and shared libraries, and the header-only source tree under `PREFIX`.

Out-of-tree builds are also supported:

```
mkdir /tmp/libxs-build && cd /tmp/libxs-build
make -j $(nproc) -f /path/to/libxs/Makefile
```

Usage

**Library** -- link against libxs.a (or .so) and include the desired headers:

```
#include <libxs/libxs_mem.h>
#include <libxs/libxs_timer.h>
```

**Header-only** (explicit) -- include libxs\_source.h (no separate library needed). Safe to include from multiple translation units:

```
#include <libxs/libxs_source.h>
```

**Header-only** (implicit) -- compile with -DLIBXS\_SOURCE and any LIBXS public header automatically includes the implementation. No special include order is required. When used through LIBXSTREAM without a pre-built library (-DLIBXSTREAM\_SOURCE), LIBXS\_SOURCE is implied automatically.

**Fortran** -- use the provided module (documentation):

```
USE :: libxs, ONLY: libxs_memcmp
```

Samples

Sample	Description
registry	Registry dispatch microbenchmark
stratify	Space-filling 3D-to-2D stratification for dense volumes
rosetta	Hierarchical type discovery on opaque binary data
predict	Train a prediction model from CSV and save it
shuffle	Shuffling strategies comparison
setdiff	Deterministic set-difference tolerance experiments
scratch	Pool allocator vs system malloc
memory	Benchmarks for comparison, matrix copy, and transpose
fprint	Foeppel fingerprint experiments for structure and geometry
ozaki	Ozaki-scheme low-precision GEMM with intercepted BLAS
gemm	Batched DGEMM (strided, pointer-array, grouped) with OMP
syrk	Symmetric rank-k/2k update (SYRK/SYR2K) with validation
sync	Lock implementation microbenchmarks

License

BSD 3-Clause

LIBXS Domains

CPU Identification

Header: libxs\_cpuid.h

Portable CPU feature detection for x86-64, AArch64, and RISC-V targets. Returns an ISA level that can be compared numerically -- higher values indicate more capable instruction sets. x86 levels use thermometer ordering: higher numeric value implies all features of lower levels. AVX10/256 sits below AVX512 because it lacks 512-bit vectors.

ISA Constants

Constant	Value	Architecture
LIBXS_TARGET_ARCH_UNKNOWN	0	Unknown / unsupported
LIBXS_TARGET_ARCH_GENERIC	1	Portable scalar baseline
LIBXS_X86_GENERIC	1002	x86-64 baseline
LIBXS_X86_SSE3	1003	SSE3
LIBXS_X86_SSE42	1004	SSE4.2
LIBXS_X86_AVX	1005	AVX
LIBXS_X86_AVX2	1006	AVX2 + FMA
LIBXS_X86_AVX10_256	1030	AVX10.1/256: all features, 256-bit max (Sierra Forest)
LIBXS_X86_AVX512	1100	AVX-512 (F + CD + DQ + BW + VL + VNNI)
LIBXS_X86_AVX512_AMX	1105	AVX-512 + AMX (TILE + INT8 + BF16); Sapphire/Granite Rapids
LIBXS_X86_AVX512_INT8	1110	AVX-512 + AVX-VNNI-INT8 (VPDPBUUD/BSSD)
LIBXS_X86_AVX10_512	1200	AVX10.1/512 + AMX + INT8: full feature set
LIBXS_AARCH64	2001	ARMv8.1 baseline
LIBXS_AARCH64_SVE128	2201	SVE 128-bit
LIBXS_AARCH64_SVE256	2301	SVE 256-bit
LIBXS_AARCH64_SVE512	2401	SVE 512-bit
LIBXS_RV64_MVL128	3001	RISC-V RVV 128-bit
LIBXS_RV64_MVL256	3002	RISC-V RVV 256-bit
LIBXS_RV64_MVL128_LMUL	3003	RISC-V RVV 128-bit (non-unit LMUL)
LIBXS_RV64_MVL256_LMUL	3004	RISC-V RVV 256-bit (non-unit LMUL)
LIBXS_X86_ALLFEAT	1999	x86 sentinel (all features)
LIBXS_AARCH64_ALLFEAT	2999	AArch64 sentinel (all features)
LIBXS_RV64_ALLFEAT	3999	RISC-V sentinel (all features)

Types

```
typedef struct libxs_cpuid_t {
    char model[256];    /* CPU model name from OS */
    int constant_tsc;   /* non-zero if TSC is invariant */
} libxs_cpuid_t;
```

Functions

```
int libxs_cpuid(libxs_cpuid_t* info);
```

Detect the ISA level of the current platform. Optionally fills `info` with the CPU model name and TSC capability. Returns a constant from the table above.

```
const char* libxs_cpuid_name(int id);
```

Returns a human-readable string for the given ISA constant, e.g., "avx2".

```
int libxs_cpuid_id(const char* name);
```

Translates a name (e.g., "avx512") to the corresponding ISA constant.

```
int libxs_cpuid_vlen(int id);
```

Returns the SIMD vector length in bytes for the given ISA level (0 for scalar-only targets).

```
int libxs_cpuid_amx_enable(void);
```

Request AMX tile state (XTILE\_DATA) from the OS. Must be called before any AMX tile instruction. Returns `EXIT_SUCCESS` if tiles are ready (or were already enabled), `EXIT_FAILURE` on unsupported platform or OS refusal. Not called automatically by `libxs_cpuid` to avoid the per-thread XSAVE overhead for non-AMX workloads.

## GEMM: Matrix Multiplication

Header: `libxs_gemm.h` Fortran: `USE LIBXS (libxs/libxs.f)`

Batched general matrix-matrix multiplication (GEMM) and symmetric rank-k/2k updates. Operations are expressed as  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where `op()` is an optional transpose. Kernels are dispatched via MKL JIT, LIBXSMM, or BLAS; a built-in default kernel (auto-vectorized) is used as fallback.

### Types

```
typedef struct libxs_gemm_shape_t {
    libxs_data_t datatype;
    int transa, transb;
    int m, n, k, lda, ldb, ldc;
    double alpha, beta;
} libxs_gemm_shape_t;
```

GEMM shape: problem geometry, transpose flags, and scalar coefficients. Serves as registry key when caching dispatched configurations.

```
typedef struct libxs_gemm_config_t {
    libxs_gemm_dblas_t dgemm_blas;
    libxs_gemm_sblas_t sgemm_blas;
    libxs_gemm_djit_t dgemm_jit;
    libxs_gemm_sjit_t sgemm_jit;
    libxs_gemm_xfn_t xgemm;
    void* jitter;
    libxs_gemm_flags_t flags;
    libxs_gemm_shape_t shape;
} libxs_gemm_config_t;
```

Configuration holding dispatched GEMM kernels. Kernel priority:

1. JIT kernel (`dgemm__jit/sgemm__jit + jitter`),
2. XGEMM kernel (`xgemm`),
3. BLAS kernel (`dgemm__blas/sgemm__blas`) -- always non-NULL after dispatch (falls back to built-in auto-vectorized C code).

```
typedef enum libxs_gemm_flags_t {
    LIBXS_GEMM_FLAGS_DEFAULT = 0,
    LIBXS_GEMM_FLAG_NOLOCK = 1
} libxs_gemm_flags_t;
```

Flags controlling batch synchronization. Set `LIBXS_GEMM_FLAG_NOLOCK` when no duplicate C pointers exist across the batch.

### Dispatch

```
libxs_gemm_config_t* libxs_gemm_dispatch(
    libxs_data_t datatype, char transa, char transb,
    int m, int n, int k, int lda, int ldb, int ldc,
    const void* alpha, const void* beta,
    void* registry /* = NULL */);
```

Inline function that selects the backend at compile time:

- MKL JIT (if `mkl.h` is included before `libxs_gemm.h`),
- LIBXSMM (if `libxsmm.h` is included),
- BLAS `dgemm/sgemm` (if `__BLAS`, `__MKL`, or `MKL_H` is defined),
- built-in default otherwise.

On registry hit, returns a pointer to the cached config (hash probe only). On miss, dispatches a new kernel and stores it. If registry is NULL, an internal registry is used.

```
typedef struct libxs_gemm_backend_t {
    libxs_jit_create_dgemm_t  jit_create_dgemm;
    libxs_jit_get_dgemm_t     jit_get_dgemm;
    libxs_jit_create_sgemm_t  jit_create_sgemm;
    libxs_jit_get_sgemm_t     jit_get_sgemm;
    libxs_xgemm_dispatch_t    xgemm_dispatch;
    libxs_gemm_dblas_t        dgemm_blas;
    libxs_gemm_sblas_t        sgemm_blas;
} libxs_gemm_backend_t;
```

```
libxs_gemm_config_t* libxs_gemm_dispatch_rt(
    const libxs_gemm_shape_t* shape,
    const libxs_gemm_shape_t* kernel_shape,
    const libxs_gemm_backend_t* backend,
    void* registry);
```

Non-inline function that accepts backend and shape structs. shape: full problem shape (registry key, stored in config). kernel\_shape: actual kernel dimensions (may differ, e.g., tight ldc for scratch). NULL means same as shape. If kernel\_shape differs from shape, the kernel is looked up under kernel\_shape first (double-dispatch), avoiding redundant code generation. backend: function pointers for backends. NULL means built-in default only. Same registry semantics as above. LIBXS\_GEMM\_BACKEND can restrict the starting point of the runtime fallback chain: 0 = automatic/default, 1 = MKL JIT, 2 = LIBXSMM, 3 = BLAS/MKL, 4 = built-in fallback. Choices 1-3 still fall through to lower-priority backends if the requested backend is not supplied or cannot dispatch the shape.

Backend callback signatures (MKL-compatible):

```
jit_create_dgemm: int(void** jitter, int layout, int transa,
                    int transb, int m, int n, int k,
                    double alpha, int lda, int ldb,
                    double beta, int ldc)
    Return: MKL_JIT_SUCCESS (0) or
            MKL_NO_JIT (1) on success,
            MKL_JIT_ERROR (2) on failure.

jit_get_dgemm:     void*(void* jitter)
    Return kernel function pointer.

xgemm_dispatch:    libxs_gemm_xfn_t(int datatype, int flags,
                    int m, int n, int k,
                    int lda, int ldb, int ldc)
    flags: bit 0 = transa, bit 1 = transb,
           bit 2 = beta==0.
```

```
USE LIBXS_JIT
rc = libxs_gemm_dispatch(config, LIBXS_DATATYPE_F64,
    & 'N', 'N', m, n, k, lda, ldb, ldc, alpha, beta)
```

The LIBXS\_JIT adapter module provides the same dispatch API but fills in BLAS (and MKL JIT when compiled with \_\_MKL) automatically. Requires BLAS at link time.

```
USE LIBXS
rc = libxs_gemm_dispatch(config, datatype, transa, transb,
    & m, n, k, lda, ldb, ldc, alpha, beta,
    & jit_create_dgemm=..., jit_get_dgemm=...,
    & dgemm_blas=..., registry=...)
```

All backend arguments are OPTIONAL C\_FUNPTR (named arguments). Returns nonzero on success (dispatch produced a callable config). The config is populated from the registry-owned copy.

## Single-Kernel Call

```
void libxs_gemm_call(  
    const libxs_gemm_config_t* config,  
    const void* a, const void* b, void* c);
```

Call the dispatched GEMM kernel (JIT > XGEMM > BLAS fallback). The caller must ensure config is non-NULL (dispatch succeeded).

## Release

```
void libxs_gemm_release(const libxs_gemm_config_t* config);
```

Release resources (e.g., MKL jitter handle) held by config. In debug builds (NDEBUG not defined), zeros the config to poison use-after-release.

```
void libxs_gemm_release_registry(libxs_registry_t* registry);
```

Release all configs in a registry, then destroy the registry.

## Pointer-Array Batch

```
void libxs_gemm_batch(  
    const void* a_array[], const void* b_array[], void* c_array[],  
    int batchsize, const libxs_gemm_config_t* config);
```

```
void libxs_gemm_batch_task(  
    const void* a_array[], const void* b_array[], void* c_array[],  
    int batchsize, const libxs_gemm_config_t* config,  
    int tid, int ntasks);
```

Batch of GEMMs from pointer arrays. The `_task` variant splits work across ntasks threads (tid = 0..ntasks-1).

## Index/Strided Batch

```
void libxs_gemm_index(  
    const void* a, const int stride_a[],  
    const void* b, const int stride_b[],  
    void* c, const int stride_c[],  
    int index_stride, int index_base,  
    int batchsize, const libxs_gemm_config_t* config);
```

```
void libxs_gemm_index_task(  
    const void* a, const int stride_a[],  
    const void* b, const int stride_b[],  
    void* c, const int stride_c[],  
    int index_stride, int index_base,  
    int batchsize, const libxs_gemm_config_t* config,  
    int tid, int ntasks);
```

Batch of GEMMs from element-offset index arrays into contiguous buffers. `index_base`: 0 (C) or 1 (Fortran). `index_stride`: byte stride between consecutive index entries (sizeof(int) for packed arrays, 0 for constant-stride mode).

## SYR2K / SYRK

Symmetric rank-2k and rank-k updates built on top of GEMM dispatch.

For small problems ( $n \leq \text{LIBXS\_GEMM\_BLOCK\_M}$  and  $k \leq \text{LIBXS\_GEMM\_BLOCK\_K}$ ), the dispatched kernel handles the full GEMM in one call. For larger problems, the implementation tiles the output into blocks and accumulates along K. Full-size tiles use the dispatched JIT kernel; remainder tiles fall back to BLAS (if available) or the built-in default. Scratch memory is thread-local.

```
libxs_gemm_config_t* libxs_syr2k_dispatch(  
    libxs_data_t datatype, int n, int k, int lda, int ldb, int ldc,  
    const libxs_gemm_backend_t* backend /* = NULL */,  
    void* registry /* = NULL */);  
  
libxs_gemm_config_t* libxs_syrk_dispatch(  
    libxs_data_t datatype, int n, int k, int lda, int ldc,  
    const libxs_gemm_backend_t* backend /* = NULL */,  
    void* registry /* = NULL */);
```

Dispatch a GEMM config for SYR2K/SYRK. The shape (n, k, lda, ldb, ldc) is stored in the config and used by the call functions. backend supplies JIT and BLAS function pointers (NULL = built-in default). Returns NULL on failure.

Fortran variants accept OPTIONAL backend function pointers:

```
ptr = libxs_syrk_dispatch(LIBXS_DATATYPE_F64, n, k, lda, ldc,  
    & jit_create_dgemm=C_FUNLOC(mkl_cblas_jit_create_dgemm),  
    & jit_get_dgemm=C_FUNLOC(mkl_jit_get_dgemm_ptr),  
    & dgemm_blas=C_FUNLOC(DGEMM))
```

```
void libxs_syr2k(  
    const libxs_gemm_config_t* config, char uplo,  
    double alpha, double beta,  
    const void* a, const void* b, void* c);
```

```
void libxs_syr2k_task(  
    const libxs_gemm_config_t* config, char uplo,  
    double alpha, double beta,  
    const void* a, const void* b, void* c,  
    int tid, int ntasks);
```

$C := \alpha(AB^T + BA^T) + \beta C$ . Only the triangle specified by uplo ('U' or 'L') is written. All dimensions and leading dimensions come from the dispatched config.

```
void libxs_syrk(  
    const libxs_gemm_config_t* config, char uplo,  
    double alpha, double beta,  
    const void* a, void* c);
```

```
void libxs_syrk_task(  
    const libxs_gemm_config_t* config, char uplo,  
    double alpha, double beta,  
    const void* a, void* c,  
    int tid, int ntasks);
```

$C := \alpha AA^T + \beta C$ . Only the triangle specified by uplo ('U' or 'L') is written.

The `_task` variants partition work across `ntasks` threads. Each thread operates on independent output blocks; no locking is required. Thread-local scratch buffers are used internally.

## Compile-Time Tuning

The block sizes used for tiled SYRK/SYR2K can be overridden at compile time via preprocessor defines:

```
LIBXS_GEMM_BLOCK_M    Row block size      (default: 32)
LIBXS_GEMM_BLOCK_N    Column block size (default: BLOCK_M)
LIBXS_GEMM_BLOCK_K    K-direction block (default: 128)
```

Problems fitting within these limits use a single specialized kernel call (MKL JIT or LIBXSMM when available).

## Environment Variables

```
LIBXS_GEMM_PRINT=N    Print dispatch info every N-th call
                      to stderr (compile-time gate).
LIBXS_GEMM_BACKEND=N  Select runtime backend chain start:
                      0 auto/default, 1 MKL JIT, 2 LIBXSMM,
                      3 BLAS/MKL, 4 built-in fallback.
```

## Example (C)

```
#include <libxs/libxs_gemm.h>

libxs_registry_t* reg = libxs_registry_create();

/* dispatch once per unique shape */
const libxs_gemm_config_t* cfg = libxs_syr2k_dispatch(
    LIBXS_DATATYPE_F64, n, k, lda, ldb, ldc, reg);

/* call many times (registry hit = hash probe only) */
for (batch = 0; batch < nbatches; ++batch) {
    libxs_syr2k(cfg, 'U', 0.5, 0.0, a[batch], b[batch], c[batch]);
}

libxs_gemm_release_registry(reg);
```

## Example (Fortran)

```
USE :: LIBXS

TYPE(libxs_gemm_config_t) :: config
INTEGER(C_INT) :: rc

rc = libxs_gemm_dispatch(config, LIBXS_DATATYPE_F64,
    & 'N', 'N', m, n, k, lda, ldb, ldc, alpha, beta,
    & dgemm_blas=C_FUNLOC(DGEMM))

IF (0 /= rc) THEN
    CALL libxs_gemm_call(config, C_LOC(a), C_LOC(b), C_LOC(c))
END IF
! rc /= 0 guarantees libxs_gemm_call will succeed
! (JIT, XGEMM, or BLAS/built-in fallback)
```



## Hashing and Checksums

Header: `libxs_hash.h`

Hash functions for buffers, fixed-width keys, and strings. The default hash uses hardware-accelerated CRC32-C (SSE4.2) with a software fallback.

### Buffer Hashing

```
unsigned int libxs_hash(const void* data, unsigned int size,
    unsigned int seed);
```

Produce a 32-bit hash from a byte buffer of the given size. NULL buffer is accepted (returns seed). Uses CRC32-C when SSE4.2 is available, otherwise a software table-based CRC32.

### Fixed-Width Hashing

```
unsigned int libxs_hash8(unsigned int data);
unsigned int libxs_hash16(unsigned int data);
unsigned int libxs_hash32(unsigned long long data);
```

Hash an 8-, 16-, or 32-bit value. The input is split into a data part and an implicit seed derived from the remaining bits. The result is masked to the corresponding width (8, 16, or 32 bits).

### String Hashing

```
unsigned long long libxs_hash_string(const char string[]);
```

64-bit hash of a character string. Short strings (up to 8 bytes) are stored directly; longer strings use two CRC32 passes combined into a 64-bit result. NULL-string is accepted (returns zero).

### CRC32 (ISO 3309)

```
unsigned int libxs_hash_iso3309(const void* data,
    unsigned int size, unsigned int seed);
```

CRC-32 using the ISO 3309 polynomial (used by PNG, gzip, and similar formats). Software-only, no hardware acceleration. Pre-condition and post-XOR with 0xFFFFFFFF are the caller's responsibility. NULL buffer is accepted.

### Adler-32

```
unsigned int libxs_adler32(const void* data,
    unsigned int size, unsigned int seed);
```

Adler-32 checksum (the variant used by zlib). The standard initial seed is 1. NULL buffer is accepted.

# Histogram

Header: `libxs_hist.h`

Thread-safe histogram with running statistics. Buckets by `vals[0]`; additional values per entry track user-defined statistics.

## Types

```
typedef struct libxs_hist_t libxs_hist_t;  /* opaque */

typedef void (*libxs_hist_update_t)(double* dst, const double* src, int count);

typedef struct libxs_hist_info_t {
    const int* buckets;    /* per-bucket counts [nbuckets] */
    const double* vals;    /* per-bucket values [nbuckets * nvals] */
    double range[2];       /* [min, max] of vals[0] */
    int nbuckets, nvals, nsamples;
} libxs_hist_info_t;
```

## Functions

```
libxs_hist_t* libxs_hist_create(int nbuckets, int nvals,
    const libxs_hist_update_t update[]);
void libxs_hist_destroy(libxs_hist_t* hist);
```

Create/destroy. `update[nvals]` specifies per-slot accumulation (NULL defaults to avg). Destroy accepts NULL.

```
void libxs_hist_push(libxs_lock_t* lock,
    libxs_hist_t* hist, const double vals[]);
```

Insert one sample. Re-bins automatically if `vals[0]` exceeds range.

```
void libxs_hist_query(libxs_lock_t* lock,
    const libxs_hist_t* hist, libxs_hist_info_t* info);
```

Query statistics. Lazy-commits queued items on first call.

```
void libxs_hist_query_percentile(libxs_lock_t* lock,
    const libxs_hist_t* hist, double vals[], double percentile);
void libxs_hist_query_median(libxs_lock_t* lock,
    const libxs_hist_t* hist, double vals[]);
```

Interpolated values at percentile `[0..1]` or median (0.5).

```
void libxs_hist_print(FILE* ostream, const libxs_hist_t* hist,
    const int prec[], const char fmt[], ...);
```

Print to stream. `prec[k]` controls precision; negative skips output. NULL ostream/fmt accepted.

## Update Functions

```
void libxs_hist_update_avg(double* dst, const double* src, int count);
void libxs_hist_update_add(double* dst, const double* src, int count);
void libxs_hist_update_min(double* dst, const double* src, int count);
void libxs_hist_update_max(double* dst, const double* src, int count);
```

- avg -- Welford online mean: `*dst += (*src - *dst) / count`
- add -- sum: `*dst += *src`
- min -- `*dst = min(*dst, *src)`
- max -- `*dst = max(*dst, *src)`

Custom callbacks use the same signature; `count` enables online algorithms without external state.

## Memory Allocation

Header: `libxs_malloc.h`

Pool-based memory allocator designed for steady-state performance: after an initial warm-up phase, allocations are served from a recycled pool without system calls.

## Types

```
typedef struct libxs_malloc_info_t {
    size_t size; /* allocated size in bytes */
} libxs_malloc_info_t;

typedef struct libxs_malloc_pool_hist_t {
    size_t count; /* allocations in this bucket */
    size_t nreuses; /* chunk reused without reallocation */
    size_t ngrows; /* chunk reallocated (too small) */
    size_t nevicts_age; /* evictions triggered by age */
    size_t nevicts_limit; /* evictions triggered by memory limit */
} libxs_malloc_pool_hist_t;

typedef struct libxs_malloc_pool_info_t {
    size_t used; /* memory currently in use (sum of requested sizes) */
    size_t size; /* total allocated memory (sum of actual chunk sizes) */
    size_t peak; /* peak memory consumption */
    size_t nactive; /* pending (not yet freed) allocations */
    size_t nmallocs; /* total allocation count */
    libxs_malloc_pool_hist_t hist[6]; /* per-bucket histogram */
} libxs_malloc_pool_info_t;
```

The histogram uses 6 logarithmic buckets keyed on requested size: <1M, 1-4M, 4-16M, 16-64M, 64-256M, >=256M. Each bucket tracks how allocations in that size class were served (reuse vs. grow) and how many were evicted by the pool's memory-pressure heuristics.

```
typedef struct libxs_malloc_pool_t libxs_malloc_pool_t; /* opaque */

typedef void* (*libxs_malloc_fn)(size_t size);
typedef void (*libxs_free_fn)(void* pointer);

typedef void* (*libxs_malloc_xfn)(size_t size, const void* extra);
typedef void (*libxs_free_xfn)(void* pointer, const void* extra);
```

## Pool Management

```
libxs_malloc_pool_t* libxs_malloc_pool(libxs_malloc_fn malloc_fn, libxs_free_fn free_fn);
```

Create a memory pool. If `malloc_fn` and `free_fn` are both `NULL`, the standard `malloc/free` are used. Both must be `NULL` or both must be non-`NULL`.

```
libxs_malloc_pool_t* libxs_malloc_xpool(libxs_malloc_xfn malloc_fn, libxs_free_xfn free_fn,
    int max_nthreads);
```

Create a memory pool with extended allocator functions that receive a per-thread extra argument (see `libxs_malloc_arg`). The `max_nthreads` parameter determines the size of the internal per-thread argument table (indexed by `libxs_tid`). Both function pointers must be non-`NULL` and `max_nthreads` must be positive; otherwise `NULL` is returned.

```
void libxs_malloc_arg(libxs_malloc_pool_t* pool, const void* extra);
```

Set the per-thread extra argument for an extended pool (created by `libxs_malloc_xpool`). The pointer is stored at the calling thread's slot (`libxs_tid() % max_nthreads`) and passed to the registered `malloc_xfn/free_xfn` on subsequent allocations and frees from this thread. No-op for standard pools. Access is lock-free since each thread writes only its own slot.

```
void libxs_free_pool(libxs_malloc_pool_t* pool);
```

Destroy the pool and release all associated memory. Accepts `NULL`.

```
int libxs_malloc_pool_info(const libxs_malloc_pool_t* pool, libxs_malloc_pool_info_t* info);
```

Query aggregate pool statistics including the per-bucket histogram.

```
void libxs_malloc_pool_print(FILE* ostream, const char prefix[],
    const libxs_malloc_pool_t* pool);
```

Print pool statistics and per-bucket histogram to `ostream`. Each non-empty bucket is printed on its own line showing count, reuse, grow, and eviction numbers. The `prefix` string (may be `NULL`) is prepended to each line. No output is produced if the pool has zero allocations or `ostream` is `NULL`.

```
libxs_malloc_pool_t* libxs_default_pool(void);
```

Access the default host memory pool (created at `libxs_init`). Printed automatically at `libxs_finalize` when `LIBXS_VERBOSE`  $\geq 4$ .

## General Allocation

```
void* libxs_malloc(libxs_malloc_pool_t* pool, size_t size, int alignment);
```

Allocate `size` bytes from the given `pool`. `LIBXS_MALLOC_AUTO` uses automatic alignment with inline metadata. `LIBXS_MALLOC_NATIVE` preserves the allocator's native pointer (out-of-band metadata via registry). Values greater than 1 are interpreted as explicit alignment in Bytes (inline metadata). Returns `NULL` on failure or if `pool` is `NULL`.

```
void libxs_free(void* pointer);
```

Return memory to its originating pool (derived internally). Accepts `NULL`.

```
int libxs_malloc_info(const void* pointer, libxs_malloc_info_t* info);
```

Query the size of an allocation made with `libxs_malloc`. The pool is derived internally.

## Fixed-Size Pool

A lightweight fixed-size pool for scenarios where the element size is known at initialization time.

```
void libxs_pmalloc_init(size_t size, size_t* num,
    void* pool[], void* storage);
```

Partition storage into \*num chunks of size bytes and register them in pool.

```
void* libxs_pmalloc(void* pool[], size_t* num);
void* libxs_pmalloc_lock(void* pool[], size_t* num,
    libxs_lock_t* lock);
```

Pop one chunk from the pool. The \_lock variant uses a caller-provided lock; the plain variant uses an internal lock.

```
void libxs_pfree(void* pointer, void* pool[], size_t* num);
void libxs_pfree_lock(void* pointer, void* pool[],
    size_t* num, libxs_lock_t* lock);
```

Push a chunk back into the pool.

## Math Utilities

Header: libxs\_math.h

## Matrix Difference

```
typedef struct libxs_matdiff_t {
    double norm1_abs, norm1_rel; /* one-norm */
    double normi_abs, normi_rel; /* infinity-norm */
    double normf_rel; /* Frobenius-norm (relative) */
    double linf_abs, linf_rel; /* max difference (abs/rel at same element) */
    double l2_abs, l2_rel, rsq; /* L2-norm and R-squared */
    double l1_ref, min_ref, max_ref, avg_ref, var_ref; /* reference stats */
    double l1_tst, min_tst, max_tst, avg_tst, var_tst; /* test stats */
    double diag_min_ref, diag_max_ref; /* diagonal min/max (reference) */
    double diag_min_tst, diag_max_tst; /* diagonal min/max (test) */
    double v_ref, v_tst; /* values at max-diff location */
    double w; /* cumulative weight for online mean */
    int m, n, i, r; /* location and reduction count */
} libxs_matdiff_t;
```

The fields linf\_abs, linf\_rel, and v\_ref/v\_tst always refer to the same element. For complex types, statistics use modulus; differences use complex absolute error.

The w field accumulates element count (m\*n) across reductions. libxs\_matdiff\_reduce uses weighted online mean (West/Welford) for avg\_ref/avg\_tst: collapses to plain Welford when all matrices are the same size; gives the exact element-weighted grand mean otherwise.

```
int libxs_matdiff(libxs_matdiff_t* info,
    libxs_data_t datatype, int m, int n,
    const void* ref, const void* tst,
    const int* ldref, const int* ldtst);
```

Compute scalar differences between two matrices. Supports all real and integer libxs\_data\_t types, plus LIBXS\_DATATYPE\_C64 and LIBXS\_DATATYPE\_C32 (interleaved complex; dimensions refer to complex elements).

```
double libxs_matdiff_epsilon(const libxs_matdiff_t* input);
```

Combined error margin from absolute and relative norms. Optionally logs to file via LIBXS\_MATDIFF env var.

```
int libxs_matdiff_combine(libxs_matdiff_t* output,
    const libxs_matdiff_t* input);
```

Combine two single-matrix infos (`ref=NULL`) into a meta-diff. Per-side statistics are exact; `linf_abs` is the mean shift, `l2_abs` a statistical bound.

```
void libxs_matdiff_reduce(libxs_matdiff_t* output,
    const libxs_matdiff_t* input);
void libxs_matdiff_clear(libxs_matdiff_t* info);
```

Worst-case reduction of matdiff results. Initialize with `libxs_matdiff_clear`.

```
double libxs_matdiff_posdef(const libxs_matdiff_t* info);  /* inline */
```

Returns the smallest test-side diagonal element (positive = necessary condition for positive definiteness met). Zero if no diagonal data.

## Multiset Distance

Order-independent distance between two vectors treated as multisets. Counts elements that cannot be matched 1-to-1 within the given tolerance. The result satisfies metric properties: symmetry, non-negativity, and identity of indiscernibles.

For real and integer types, both arrays are sorted and matched via a two-pointer merge. For complex types (C64, C32), a 2D k-d tree is built on one array and nearest-neighbor queries with consumption (used-flags) provide 1-to-1 matching by Euclidean distance. The tolerance threshold is inclusive (less-than-or-equal).

```
int libxs_setdiff(libxs_data_t datatype,
    const void* a, int na,
    const void* b, int nb, double tol);
```

Supports all `libxs_data_t` types. Returns the number of unmatched elements, or -1 for unsupported types. The distance is at least  $\text{abs}(na - nb)$  when the vector lengths differ.

```
int libxs_setdiff_min(libxs_data_t datatype,
    const void* a, int na,
    const void* b, int nb, double* tol);
```

Minimizes the multiset distance over all tolerances using Golden Section Search (`libxs_gss_min`). Returns the minimum unmatched count. The pointer `tol` (may be `NULL`) receives the smallest tolerance that achieves this minimum. Supported types: F64, F32, C64, C32 only (integer types are not meaningful here; returns  $\max(na, nb)$  with `tol` set to zero).

The distance as a function of tolerance is monotonically non-increasing (a step function with discrete drops), which makes it unimodal -- the prerequisite for Golden Section Search.

## Foepl Polynomial Fingerprint

Structural fingerprint for n-dimensional data based on Foepl (Macaulay) bracket polynomials. For 1-D data, the fingerprint records per-derivative-order norms of the forward finite differences, normalized to the unit interval [0,1]. Higher dimensions are handled hierarchically: the innermost dimension is fingerprinted first, each child fingerprint is collapsed to a Sobolev self-norm scalar, and those scalars form the 1-D input for the next outer dimension.

Because the fingerprint is normalized to the unit interval, datasets of different lengths (or shapes) produce directly comparable fingerprints. The derivative-order decomposition captures structural features at multiple scales: order 0 measures value magnitude, order 1 measures slope/trend, order 2 measures curvature, and so on.

```
#define LIBXS_FPRINT_MAXORDER 8
```

```
typedef struct libxs_fprint_t {
    double l2[LIBXS_FPRINT_MAXORDER + 1];
    double l1[LIBXS_FPRINT_MAXORDER + 1];
    double linf[LIBXS_FPRINT_MAXORDER + 1];
    double mean[LIBXS_FPRINT_MAXORDER + 1];
    int order, n;
    libxs_data_t datatype;
} libxs_fprint_t;
```

Three norm families and a signed mean are computed per derivative order  $k = 0..order$ :

Field	Description
l2[k]	L2 norm of the k-th finite difference
l1[k]	Mean absolute value (total variation)
linf[k]	Maximum absolute value (worst-case decay)
mean[k]	Signed mean (sum / count, preserves sign/phase)

All norms are normalized to the unit interval ( $h = 1/(n-1)$ ). The signed mean preserves the dominant direction of the  $k$ -th derivative and breaks pseudometric collisions that pure norms cannot distinguish (negation, reflection).

The `order` field records how many derivative orders were used; `n` is the extent of the fingerprinted dimension. The `datatype` field records the element type -- either the type passed by the caller, or the type discovered by probing when the caller passes `LIBXS_DATATYPE_UNKNOWN` (see "Type Discovery" below).

To recover raw (unnormalized) finite-difference magnitudes, use `libxs_fprint_raw` (see Generalized Binomial and Distance section).

```
double libxs_fprint_decay(const libxs_fprint_t* info);
```

Geometric-mean per-order decay ratio:

$$r = (l2[K] / l2[0])^{(1/K)} / (n - 1)$$

The  $1/(n-1)$  factor removes the unit-interval scaling so that decay ratios are comparable across different element counts (without it, fewer elements always appears to have lower decay).

Interpretation of the return value:

```
r << 1    Structured data: forward differences shrink with
           derivative order. Data is smooth and compressible
           under Newton truncation.

r ~= 1    Noise / random data: forward differences grow at the
           rate expected for uncorrelated sequences (~2x per
           order before normalization).

r >> 1    Should not occur for well-formed data; indicates
           divergent finite differences (numerical overflow or
           pathological input).

1e30      Returned when no valid ratio can be computed (e.g.,
           order == 0 or l2[0] == 0 or n < 2).
```

The decay ratio is the key discriminator for type discovery (see below) and for the compressibility diagnostic described in the Foeppl paper.

The L2 norms and signed means serve comparison via the Sobolev distance (`libxs_fprint_diff`). The Linf norms serve as a decay diagnostic: decaying `linf[k]` indicates structurally smooth data (compressible under Newton truncation), while growing `linf[k]` indicates unstructured data (no exploitable smoothness). The L1 norms provide a noise-robust middle ground between L2 and Linf.

```
int libxs_fprint(libxs_fprint_t* info,
    libxs_data_t datatype, const void* data,
    int ndims, const size_t shape[], const size_t stride[],
    int order, int axis);
```

Build a fingerprint from data described by shape and stride arrays. For 1-D data, pass `ndims=1`, `shape={n}`, `stride=NULL`. For higher dimensions, `shape[0]` is the innermost extent and `shape[ndims-1]` the outermost. When `stride` is `NULL`, contiguous storage is assumed (`stride[0]=1`, `stride[k]=product of shape[0..k-1]`). The requested order is clamped to `min(order, extent-1, LIBXS_FPRINT_MAXORDER)`.

The `axis` parameter controls how multi-dimensional data is handled:

axis < 0 Hierarchical mode (default). Sweeps the innermost dimension first and collapses each level into the next outer dimension via Sobolev self-norms.

axis >= 0 Per-axis mode. Differentiates along dimension 'axis' only and takes the per-order maximum of each norm (linf, l2, l1) across all positions in the remaining dimensions.

For 1-D data, axis is ignored (both modes are equivalent).

Supported types: F64, F32, and all integer libxs\_data\_t types (I64, I32, U32, I16, U16, I8, U8 -- promoted to double internally).

When datatype is LIBXS\_DATATYPE\_UNKNOWN, the function performs automatic type discovery (Level 0 of the hierarchical analysis described below). In this mode, shape[0] is the byte count of the opaque buffer. The function probes all concrete types whose element width divides the byte count, fingerprints each, and selects the interpretation with the smallest decay ratio (libxs\_fprint\_decay). The discovered type is written to info->datatype. If no candidate has  $r < 1$ , the function returns EXIT\_FAILURE. Ties are broken by preferring the smallest element width. Float candidates whose maximum absolute value is below  $1e-37$  (subnormals) are rejected.

```
double libxs_fprint_diff(
    const libxs_fprint_t* a, const libxs_fprint_t* b,
    const double* weights);
```

Weighted Sobolev distance between two fingerprints:  $d = \sqrt{\sum_k \text{weights}[k] * ((a \rightarrow l2[k] - b \rightarrow l2[k])^2)}$

- $(a \rightarrow \text{mean}[k] - b \rightarrow \text{mean}[k])^2$  ). The number of orders compared is  $\min(a \rightarrow \text{order}, b \rightarrow \text{order}) + 1$ . If weights is NULL, default weights  $w(k) = 1/k!$  are used, which naturally dampens higher-order (noisier) derivatives.

The inclusion of the signed mean difference breaks collisions from negation (identical L2 norms but opposite means) and reflection (identical L2 but opposite first-derivative means). The distance is a metric in fingerprint space: symmetric, non-negative, zero if and only if the fingerprints are identical. The same function serves both as a distance measure and as a fingerprint comparator since the fingerprint is the decomposed form of the Sobolev norm.

Type Discovery for Opaque Data

When data arrives as a raw byte stream with no type metadata, the LIBXS toolkit can be composed into a hierarchical analysis that discovers the element type and structure. Each level uses a different tool, exploiting the fact that each tool is sensitive to a different axis of structure while being invariant to others.

Tool	Invariant to	Sensitive to
fprint (decay)	element count and scaling	consecutive-element correlation (local)
setdiff	element order	value identity (global)
matdiff	(positional)	magnitude, spread, extrema
sort_smooth	row order	row-to-row proximity
shuffle	--	provides controlled reordering

The hierarchical procedure:

Level 0 -- Decay Screening (libxs\_fprint + libxs\_fprint\_decay)

```
Interpret the byte stream under every candidate type whose
element width divides the stream length. Compute the decay
ratio r for each. Discard r >= 1 (noise). Short-list the
best few candidates (within a factor 2 of the minimum r).

This level is built into libxs_fprint when the caller passes
LIBXS_DATATYPE_UNKNOWN. The discovered type is reported in
the datatype field of the returned libxs_fprint_t.
```

Level 1 -- Self-Consistency via Setdiff (libxs\_setdiff\_min)

```
Split the stream into two halves (A, B) under each surviving
```



candidate type. Compute  $d^* = \text{libxs\_setdiff\_min}(A, B)$ . The ratio  $d^* / \max(|A|, |B|)$  is a "novelty fraction": how many values in one half have no counterpart in the other. A correct type produces low novelty (values repeat or cluster); a wrong type scatters values across the range (high novelty).

Synergy: `fprint` measures local correlation (adjacent elements); `setdiff` measures global value identity (ignoring order). A candidate that passes both has independent evidence on two orthogonal axes.

#### Level 2 -- Smoothness-Sort Separability (`libxs_sort_smooth`)

Reshape the stream as a matrix under each surviving candidate. Compute a GREEDY row permutation that minimizes row-to-row Euclidean distance. Fingerprint the permuted matrix along the row axis and measure the improvement:

$$\text{delta} = r_{\text{before}} - r_{\text{after}}$$

The candidate with the largest delta has the most exploitable row structure.

Synergy: the GREEDY sort is  $O(m^2 n)$  -- too expensive for the initial sweep but affordable for 2-3 survivors. Its sensitivity (row-to-row proximity) is orthogonal to 1-D decay and to order-independent `setdiff`.

#### Level 3 -- Shuffle Stability (`libxs_shuffle` + `libxs_fprint_decay`)

Shuffle the interpreted elements via `libxs_shuffle` (coprime permutation). Fingerprint the shuffled data. If the decay ratio increases substantially, the original element order carries genuine structure. If it stays the same, the apparent decay was accidental.

Synergy: this is the only test that probes whether the ORDER of elements matters. Decay (Level 0) measures local correlation but cannot distinguish true order from coincidence. `Setdiff` (Level 1) is explicitly order-invariant. The shuffle test closes this gap.

#### Level 4 -- Statistical Plausibility (`libxs_matdiff`)

Compute single-matrix statistics (`ref=NULL`) for each surviving candidate: mean, variance, min, max, diagonal range. Reject float interpretations with NaN, infinity, or subnormals. Flag integer interpretations whose value range spans less than 1% of the type range. Compare two candidates via `libxs_matdiff_combine` (mean-shift and variance bound).

#### Stride Sweep -- Record Layout Discovery

After discovering the atomic element width  $w$ , sweep candidate strides  $s$  that are multiples of  $w$ . Reshape the stream as an  $(L/s) \times (s/w)$  matrix. Repeat Levels 0-2 on the outer dimension (across "records"). The stride with the smallest decay and largest sort-improvement likely corresponds to the record boundary.

Why the composition is stronger than any single tool:

- Decay alone is fooled by reinterpretation artifacts (e.g., integer data producing subnormal floats that look smooth).
- `Setdiff` alone cannot detect structure -- it only counts matches, and with large enough tolerance any data matches.
- Sort-smooth alone is too expensive for a brute-force sweep and cannot discriminate types without a baseline decay.
- Shuffle stability alone cannot distinguish true order from accidental order in short sequences.
- Statistics alone reject only pathological cases (NaN, subnormals) and cannot resolve two plausible interpretations.

Each level eliminates a different class of false positives. A candidate that survives all five has demonstrated structure along five orthogonal axes: local smoothness, global value consistency, row separability, order sensitivity, and statistical plausibility. A false positive would require a simultaneous coincidence on all five, which is combinatorially unlikely.

Limitations: the analysis tests fixed-width candidates and cannot discover variable-length encodings, compression, or encryption. For data that is genuinely random at every granularity, all candidates are rejected at Level 0 and the framework reports "no structure found." The stride sweep discovers fixed-size records but not nested or recursive structures.

## Golden Section Search

```
double libxs_gss_min(  
    double (*fn)(double x, const void* data),  
    const void* data,  
    double x0, double x1, double* xmin, int maxiter,  
    int flags, double ftol,  
    libxs_gss_info_t* info);
```

Minimizes a unimodal function `fn` on the interval `[x0, x1]`. The callback receives `x` and an opaque context pointer. Returns `f(x*)` where `x*` is the minimizer; `xmin` (may be `NULL`) receives `x*`.

The bracket shrinks by factor  $\phi = (\sqrt{5}-1)/2$  per iteration, reusing one evaluation from the previous step. Convergence is reached when the bracket collapses to machine precision or `maxiter` iterations are exhausted. Flags can enable endpoint evaluation. For step-like objectives such as tolerance thresholds, where plain GSS may discard the side containing the first point of the minimum plateau, use `libxs_bisect_min` directly.

The optional `info` pointer (may be `NULL`) receives diagnostics: status flags, iteration and evaluation counts, the final `x/f` pair, the final bracket, and a unimodality score in `[0, 1]`. The score indicates how consistent the sampled points are with a single valley: 1.0 means all samples are monotonically decreasing before the minimum and increasing after; lower values indicate irregularity or multimodality. Endpoint results are reported through status flags rather than through the unimodality score.

```
double libxs_bisect_min(  
    double (*fn)(double x, const void* data),  
    const void* data,  
    double x0, double x1, double fmin, double* xmin, int maxiter,  
    double ftol, libxs_gss_info_t* info);
```

Bisects the interval `[x0, x1]` to find the left edge of a known minimum level `fmin`. The right endpoint is expected to reach `fmin`; otherwise `info->status` includes `LIBXS_GSS_STATUS_NO_BRACKET`. The function is useful as a standalone primitive for monotone threshold or plateau-edge searches.

## Generalized Binomial and Distance

```
double libxs_binom(double t, int k);
```

Generalized binomial coefficient  $C(t, k)$  for real-valued `t`:  $C(t, k) = t * (t-1) * \dots * (t-k+1) / k!$ . Evaluates the Newton basis polynomial at position `t`. For integer `t >= k >= 0`, this equals the standard binomial coefficient.

```
double libxs_dist2(const double* a, const double* b, int n);
```

Squared Euclidean distance between two `n`-dimensional vectors. Returns the sum of squared component differences.

```
double libxs_fprint_raw(const libxs_fprint_t* info,  
    int k, double value);
```

Recover unnormalized finite-difference magnitude by dividing by  $(n-1)^k$ , undoing the unit-interval scaling. For `k == 0`, the value is returned unchanged.

## Number Theory

```
size_t libxs_gcd(size_t a, size_t b);
size_t libxs_lcm(size_t a, size_t b);
```

GCD (returns 1 for gcd(0,0)) and LCM.

```
int libxs_primes_u32(unsigned int num,
    unsigned int num_factors_n32[], int num_factors_max);
```

Prime factors of num in ascending order. Returns factor count.

```
size_t libxs_coprime(size_t n, size_t minco);
size_t libxs_coprime_bias(size_t n, double bias);
size_t libxs_coprime2(size_t n); /* inline: coprime_bias(n, 0) */
```

libxs\_coprime: co-prime of n not exceeding minco.

libxs\_coprime\_bias: co-prime of n selected by bias in [-1, +1] via a logarithmic mapping (target =  $n^{((1+bias)/2)}$ ). bias=-1 selects the smallest non-trivial coprime (maximum displacement), bias=0 selects near sqrt(n) (balanced, same as libxs\_coprime2), bias=+1 selects near n/2 (near-alternation). Monotonic: larger bias always yields a larger (or equal) coprime.

libxs\_coprime2: inline convenience for libxs\_coprime\_bias(n, 0).

```
unsigned int libxs_remainder(unsigned int a, unsigned int b,
    const unsigned int* limit, const unsigned int* remainder);
```

Smallest multiple of b minimizing remainder mod a.

```
unsigned int libxs_product_limit(unsigned int product,
    unsigned int limit, int is_lower);
```

Sub-product of prime factors within limit (0/1-Knapsack).

## Scalar Utilities

```
double libxs_kahan_sum(double value, double* accumulator,
    double* compensation);
```

Kahan compensated summation.

```
unsigned int libxs_isqrt_u64(unsigned long long x);
unsigned int libxs_isqrt_u32(unsigned int x);
unsigned int libxs_isqrt2_u32(unsigned int x);
```

Integer square root. The isqrt2 variant returns a factor of x.

```
double libxs_pow2(int n);
```

$2^n$  via IEEE-754 exponent. Valid for n in [-1022, 1023]. Returns 0.0 for underflow (subnormals flushed to zero) and +Inf for overflow.

## Modular Arithmetic

```
unsigned int libxs_mod_inverse_u32(unsigned int a, unsigned int m);
size_t libxs_mod_inverse(size_t a, size_t m);
```

Modular inverse via extended Euclidean algorithm:  $a^{-1} \bmod m$ . Requires  $\gcd(a, m) = 1$  and  $0 < a, 1 < m$ . The `size_t` variant supports counting down to  $-1$ .

```
unsigned int libxs_barrett_rcp(unsigned int p);
unsigned int libxs_barrett_pow18(unsigned int p);
unsigned int libxs_barrett_pow36(unsigned int p);
```

Barrett reduction constants for modulus `p`.

```
unsigned int libxs_mod_u32(uint32_t x, unsigned int p,
    unsigned int rcp); /* inline */
unsigned int libxs_mod_u64(uint64_t x, unsigned int p,
    unsigned int rcp, unsigned int pow18,
    unsigned int pow36); /* inline */
```

Fast modular reduction via Barrett. The 64-bit variant uses a radix-2<sup>18</sup> split.

## BF16 Conversion

```
typedef uint16_t libxs_bf16_t;
libxs_bf16_t libxs_round_bf16(double x); /* inline */
double libxs_bf16_to_f64(libxs_bf16_t v); /* inline */
```

Round to BF16 (round-to-nearest-even) and expand to double. Uses native `__bf16` when available (`LIBXS_BF16`), otherwise portable bit manipulation.

## Memory Operations

Header: `libxs_mem.h`

Byte-level memory macros, pointer alignment queries, buffer comparison, and matrix copy/transposition.

### Memory Macros

```
LIBXS_MEMSET(DST, SRC, SIZE)
```

Set `SIZE` bytes at `DST` to the value `SRC`, unrolled at compile-time.

```
LIBXS_MEMZERO(DST)
```

Zero all bytes of `*DST` (size derived from `sizeof(*DST)`).

```
LIBXS_MEMCPY(DST, SRC, SIZE)
```

Copy `SIZE` bytes from `SRC` to `DST`, unrolled at compile time.

```
LIBXS_ASSIGN(DST, SRC)
```

Copy `sizeof(*SRC)` bytes from `SRC` to `DST`. Asserts `sizeof(*SRC) <= sizeof(*DST)`.

```
LIBXS_MEMSWP(PTR_A, PTR_B, SIZE)
```

Swap `SIZE` bytes between `PTR_A` and `PTR_B`. Asserts `PTR_A != PTR_B`.

```
LIBXS_VALUE_ASSIGN(DST, SRC)
LIBXS_VALUE_SWAP(A, B)
```

Operate on L-values directly (address-of is taken internally). `VALUE_ASSIGN` can cast away const qualifiers. `VALUE_SWAP` asserts equal size.

## Offset and Alignment

```
size_t libxs_offset(size_t ndims, const size_t offset[],
    const size_t shape[], size_t* size);
```

Compute the linear offset of an n-dimensional coordinate within a shape. Optionally returns the total linear size of the shape.

```
int libxs_aligned(const void* ptr, const size_t* inc,
    int* alignment);
```

Check whether `ptr` is SIMD-aligned (optionally considering the next access at `ptr + *inc`). Optionally writes the pointer alignment in bytes to `*alignment`.

## Comparison

```
unsigned char libxs_diff(const void* a, const void* b,
    unsigned char size);
```

Compare two short buffers. Returns zero when equal, non-zero otherwise. Uses SSE/AVX2/AVX-512 when available.

```
unsigned int libxs_diff_n(const void* a, const void* bn,
    unsigned char elemsize, unsigned char stride,
    unsigned int hint, unsigned int count);
```

Compare `a` against count elements in `bn`. Returns the first index that matches `a`, or count if none match. `hint` supplies the initial search position.

```
int libxs_memcmp(const void* a, const void* b, size_t size);
```

Boolean variant of the C `memcmp`. Returns zero when equal. Uses SSE/AVX2/AVX-512 when available.

## Matrix Copy and Transposition

```
void libxs_matcopy(void* out, const void* in,
    unsigned int typesize, int m, int n, int ldi, int ldo);
void libxs_matcopy_task(void* out, const void* in,
    unsigned int typesize, int m, int n, int ldi, int ldo,
    int tid, int ntasks);
```

Copy an m-by-n matrix from `in` to `out` with leading dimensions `ldi` and `ldo`. If `in` is NULL the destination is zeroed. The `_task` variant distributes work across `ntasks` threads (caller passes its `tid`).

```
void libxs_otrans(void* out, const void* in,
    unsigned int typesize, int m, int n, int ldi, int ldo);
void libxs_otrans_task(void* out, const void* in,
    unsigned int typesize, int m, int n, int ldi, int ldo,
    int tid, int ntasks);
```

Out-of-place matrix transposition. The `_task` variant distributes work across `ntasks` threads.

```
void libxs_itrans(void* inout, unsigned int typesize,
    int m, int n, int ldi, int ldo, void* scratch);
void libxs_itrans_task(void* inout, unsigned int typesize,
    int m, int n, int ldi, int ldo, void* scratch,
    int tid, int ntasks);
```

In-place matrix transposition (square or via scratch buffer). `scratch` can be NULL (auto-allocated).

```
void libxs_itrans_batch(void* inout, unsigned int typesize,
    int m, int n, int ldi, int ldo,
    int index_base, int index_stride,
    const int stride[], int batchsize,
    int tid, int ntasks);
```

Batch of in-place matrix transpositions (per-thread form).

## MetaImage I/O

Header: `libxs_mhd.h`

Read and write image data in the MetaImage (MHD/MHA) format. Files are compatible with ITK, ITK-SNAP, 3D Slicer, ParaView, and similar tools.

### Types

```
typedef enum libxs_mhd_element_conversion_hint_t {
    LIBXS_MHD_ELEMENT_CONVERSION_DEFAULT,
    LIBXS_MHD_ELEMENT_CONVERSION_MODULUS
} libxs_mhd_element_conversion_hint_t;
```

Hint controlling element conversion behaviour (default clamping vs. modulus mapping).

```
typedef struct libxs_mhd_element_handler_info_t {
    libxs_data_t type;
    libxs_mhd_element_conversion_hint_t hint;
} libxs_mhd_element_handler_info_t;
```

Destination type and conversion hint, passed to built-in or custom element handlers.

```
typedef int (*libxs_mhd_element_handler_t)(void* dst,
    const libxs_mhd_element_handler_info_t* dst_info,
    libxs_data_t src_type,
    const void* src, const void* src_min, const void* src_max,
    size_t index, void* context);
```

Per-element callback for reading or converting image data. The value range (`src_min`, `src_max`) is available for scaling. `index` is the logical element index in write/read traversal order, and `context` is user data supplied by the caller.

```
typedef struct libxs_mhd_info_t {
    size_t ndims;           /* dimensionality */
    size_t ncomponents;     /* interleaved image channels */
    libxs_data_t type;      /* pixel element type */
    size_t header_size;     /* header size in bytes (LOCAL data) */
} libxs_mhd_info_t;
```

Image descriptor populated by `libxs_mhd_read_header` and consumed by read/write.

```
typedef struct libxs_mhd_write_info_t {
    const libxs_mhd_element_handler_info_t* handler_info;
    libxs_mhd_element_handler_t handler;
    void* handler_context;
    const char* extension_header;
    const void* extension;
    size_t extension_size;
} libxs_mhd_write_info_t;
```

Optional write parameters (conversion, custom handler, extension data). NULL is accepted for all-defaults.

## Element Helpers

```
int libxs_mhd_element_conversion(void* dst,
    const libxs_mhd_element_handler_info_t* dst_info,
    libxs_data_t src_type,
    const void* src, const void* src_min, const void* src_max,
    size_t index, void* context);
```

Built-in element conversion. Scales values when `src_min` and `src_max` are non-NULL; otherwise clamps to the destination type.

```
int libxs_mhd_element_comparison(void* dst,
    const libxs_mhd_element_handler_info_t* dst_info,
    libxs_data_t src_type,
    const void* src, const void* src_min, const void* src_max,
    size_t index, void* context);
```

Check an in-memory buffer against file content (element-wise comparison with optional type conversion).

## Type Queries

```
const char* libxs_mhd_typename(libxs_data_t type,
    const char** ctypename);
```

Return the MHD element-type name (e.g. "MET\_FLOAT") for the given `libxs_data_t`. Optionally writes the C type name. Returns NULL for unknown types.

```
libxs_data_t libxs_mhd_typeinfo(const char elemname[]);
```

Return the `libxs_data_t` for a given MHD element-type string.

## Read

```
int libxs_mhd_read_header(const char header_filename[],
    size_t filename_max_length, char filename[],
    libxs_mhd_info_t* info, size_t size[],
    char extension[], size_t* extension_size);
```

Parse an MHD header file. `info->ndims` is an in/out parameter (maximum on input, actual on output). `filename` receives the data-file path (may differ from the header when data is stored separately). Extension data is optional.

```
int libxs_mhd_read(const char filename[],
    const size_t offset[], const size_t size[],
    const size_t pitch[], const libxs_mhd_info_t* info,
    void* data,
    const libxs_mhd_element_handler_info_t* handler_info,
    libxs_mhd_element_handler_t handler,
    void* handler_context);
```

Read image data into `data`. Optional `handler_info` triggers type conversion; optional `handler` supplies a custom per-element callback (can also serve as a comparison function without allocating a buffer).

## Write

```
int libxs_mhd_write(const char filename[],
    const size_t offset[], const size_t size[],
    const size_t pitch[], libxs_mhd_info_t* info,
    const void* data,
    const libxs_mhd_write_info_t* write_info);
```

Write image data in the extended MHD format. `info->header_size` is updated on output. Pass NULL for `write_info` when no conversion or extension is needed.

When the environment variable `LIBXS_MHD_PNG` is set and the image is two-dimensional, an uncompressed PNG file is written alongside the MHD output (same base name, `.png` extension). Supported channel counts are 1 (grayscale), 3 (RGB), and 4 (RGBA) at 8-bit depth. Source data of any element type is converted to unsigned 8-bit using the built-in element conversion with automatic min/max scaling.

Visit [LIBXSTREAM](#)

[Start Presentation](#)

## Permutation and Sorting

Header: `libxs_perm.h`

Permutation-based data reordering: deterministic co-prime shuffling (in-place and out-of-place with optional SIMD gather) and smoothness-optimized row permutations for matrices.

### Shuffling

The shuffle functions use a co-prime stride `C` to produce a fixed, deterministic permutation of count elements. The mapping is affine:

```
dst[k] = src[(N-1) - ((C*k + offset) mod N)]
```

The stride must be co-prime to count; passing NULL selects `libxs_coprime2(count)`. The offset parameter extends the basic co-prime permutation into an affine family, expanding the number of available permutations from  $\sim\phi(N)/2$  to  $\sim N*\phi(N)/2$ . Pass `offset=0` for the standard (non-affine) shuffle.

```
int libxs_shuffle(void* inout, size_t elemsize, size_t count,
    const size_t* shuffle, size_t offset,
    const size_t* nrepeat);
```

In-place shuffle of count elements of `elemsize` bytes each. Uses cycle following with a bit vector ( $N/8$  bytes auxiliary). `nrepeat` controls the number of successive permutation applications (NULL or pointing to 1 means one pass). Returns `EXIT_SUCCESS` or `EXIT_FAILURE`.

```
int libxs_shuffle2(void* dst, const void* src, size_t elemsize,
    size_t count, const size_t* shuffle, size_t offset,
    const size_t* nrepeat);
```

Out-of-place shuffle from `src` to `dst`. `dst` and `src` must not overlap. If `*nrepeat` is zero an ordinary copy is performed. Uses AVX2/AVX-512 gather instructions when available for 4- and 8-byte elements (`offset=0` path).

```
size_t libxs_unshuffle(size_t count, const size_t* shuffle);
```

Return the number of `libxs_shuffle2` applications needed to restore the original element order for the given count and co-prime stride. The cycle length depends only on `C` and `N`, not on the offset.

```
int libxs_unshuffle2(void* dst, const void* src, size_t elemsize,
    size_t count, const size_t* shuffle, size_t offset,
    const size_t* nrepeat);
```



Single-pass inverse of `libxs_shuffle2`. Computes the modular inverse  $C_{\text{inv}} = C^{-1} \bmod N$  via the extended Euclidean algorithm, then gathers elements using the inverse permutation:

```
dst[m] = src[C_inv * (N-1-offset-m) mod N]
```

This restores the original order in one  $O(N)$  pass rather than the  $R-1$  repeated applications that `libxs_unshuffle` would require. The loop structure (sequential write, strided read) is identical to the forward shuffle and amenable to the same SIMD gather optimizations. Supports multi-pass inversion ( $nrepeat > 1$ ) by iterating  $f^{-1}$ .

## General-Purpose Sort

```
typedef int (*libxs_sort_cmp_t)(
    const void* a, const void* b, void* ctx);

void libxs_sort(void* base, int n, size_t size,
    libxs_sort_cmp_t cmp, void* ctx);
```

Sort  $n$  elements of the given byte size. The comparator returns negative, zero, or positive (tristate, like `qsort_r`). The `ctx` pointer is forwarded to the comparator unchanged.

Built-in comparators:

```
int libxs_cmp_f64(const void* a, const void* b, void* ctx);
int libxs_cmp_f32(const void* a, const void* b, void* ctx);
int libxs_cmp_i32(const void* a, const void* b, void* ctx);
int libxs_cmp_u32(const void* a, const void* b, void* ctx);
```

Built-in comparators enable an  $O(n)$  radix sort fast path. Custom comparators use  $O(n \log n)$  heap sort.

With a built-in comparator, `ctx` controls the mode:

`ctx = NULL` Sort base in-place. `ctx != NULL` Read from `ctx`, write sorted result to base (out-of-place, source unchanged).

With a custom comparator, `ctx` is user state passed to `cmp` and base is sorted in-place.

Examples:

```
libxs_sort(data, n, sizeof(double), libxs_cmp_f64, NULL); libxs_sort(dst, n, sizeof(double), libxs_cmp_f64, src);
libxs_sort(perm, n, sizeof(int), my_indirect_cmp, keys);
```

## Space-Filling Curves

```
uint64_t libxs_hilbert(const unsigned int coords[], int ndims);
void libxs_hilbert_decode(uint64_t code, unsigned int coords[], int ndims);
```

$N$ -dimensional Hilbert curve index. Maps  $ndims$  coordinates to a 64-bit key with strong locality guarantees. Each coordinate is quantized to  $\text{floor}(64/ndims)$  bits. `coords[k]` must be in  $[0, 2^{\text{bits\_per\_dim}})$ . The decode routine maps such a key back to  $ndims$  coordinates with the same bit budget.

```
uint64_t libxs_morton(const unsigned int coords[], int ndims);
void libxs_morton_decode(uint64_t code, unsigned int coords[], int ndims);
```

$N$ -dimensional Morton code (Z-order curve). Bit-interleaves  $ndims$  coordinates into a 64-bit key. Each coordinate is quantized to  $\text{floor}(64/ndims)$  bits. The decode routine deinterleaves the key back to  $ndims$  coordinates.

```
int libxs_stratify_morton(
    const unsigned int src_coords[], int src_ndims,
    unsigned int dst_coords[], int dst_ndims);
int libxs_stratify_hilbert(
    const unsigned int src_coords[], int src_ndims,
    unsigned int dst_coords[], int dst_ndims);
```

Stratify higher-dimensional coordinates into a lower-dimensional layout without slicing. The source coordinates are encoded using the selected source-dimensional space-filling curve, and the resulting key is decoded as target-dimensional coordinates using the same curve family. This gives deterministic rank-preserving embeddings such as 3D to 2D:

```
code = curve(src_coords, 3)
dst_coords = inverse_curve(code, 2)
```

The target dimensionality must be smaller than the source dimensionality. The functions return EXIT\_SUCCESS or EXIT\_FAILURE.

## k-d Tree

```
typedef int (*libxs_kdtree_split_t)(
    int* dim, int* pos, const double* pts, int* idx,
    int count, int depth, int nleaves, void* ctx);

typedef struct libxs_kdtree_config_t {
    int min_leaf;
    libxs_kdtree_split_t split;
    void* ctx;
} libxs_kdtree_config_t;
```

Split callback for custom tree construction. Writes chosen split dimension into \*dim and left-child count into \*pos. The callback may reorder idx[0..count-1] to place left entries first; if it does, the partition function skips its internal quickselect. nleaves gives the current number of leaves already assigned. Return 0 for a valid split, nonzero to force a leaf.

Config struct: min\_leaf controls minimum node size (0 = split down to 1 element). split=NULL uses round-robin median splits.

```
void libxs_kdtree_build(const double* pts, int* idx, int n,
    int ndims, int stride, const libxs_kdtree_config_t* config);
```

Build a k-d tree in-place. Points are stored row-major: pts[i\*stride + k] is coordinate k of point i. The index array idx[0..n-1] is rearranged into implicit tree structure. config may be NULL (round-robin median, leaf at 1 element).

```
int libxs_kdtree_partition(const double* pts, int* idx, int n,
    int ndims, int stride, int* assignments,
    const libxs_kdtree_config_t* config);
```

Partition n points into leaves and write leaf IDs into assignments[0..n-1]. Returns the number of leaves. Same tree logic as build, but produces cluster assignments instead of a query-tree index.

```
int libxs_kdtree_nearest(const double* pts, const int* idx,
    const unsigned char* used, int n, int ndims, int stride,
    const double* query, double max_dist2);
```

Find the nearest point to query[0..ndims-1] within squared Euclidean distance max\_dist2. Returns the point index or -1. The used array (may be NULL) marks consumed points.

Convenience wrappers for 2D (interleaved x,y layout):

```
void libxs_kdtree2d_build(double* pts, int* idx, int n);
int libxs_kdtree2d_nearest(const double* pts, const int* idx,
    const unsigned char* used, int n,
    double x, double y, double max_dist2);
```

## Smooth Sorting

```
typedef enum libxs_sort_t {
    LIBXS_SORT_NONE      = 0,
    LIBXS_SORT_IDENTITY  = 1,
    LIBXS_SORT_NORM      = 2,
    LIBXS_SORT_MEAN      = 3,
    LIBXS_SORT_GREEDY    = 4,
    LIBXS_SORT_MORTON    = 5,
    LIBXS_SORT_HILBERT   = 6
} libxs_sort_t;

int libxs_sort_smooth(libxs_sort_t method, int m, int n,
    const void* mat, int ld, libxs_data_t datatype, int* perm);
```

Compute a row permutation that reorders the rows of an m-by-n column-major matrix for smoothness (decaying forward differences between consecutive rows).

Parameters:

method Sorting strategy (see constants above). m, n Matrix dimensions (m rows, n columns). mat Column-major matrix data (read-only). ld Leading dimension (ld >= m). datatype Element type (F64, F32, I32, U32, I16, U16, I8, U8). perm Output array of m ints; perm[i] = source row for position i after reordering.

Methods:

NONE No permutation (early return). IDENTITY Identity permutation (perm[i] = i). NORM Sort rows by ascending L1-norm. MEAN Sort rows by ascending column mean. GREEDY Nearest-neighbor chain: starting from row 0, each step picks the unvisited row with the smallest Euclidean distance to the current row. MORTON Sort rows by Morton (Z-order) key computed from quantized column values. HILBERT Sort rows by Hilbert curve key computed from quantized column values.

Returns EXIT\_SUCCESS or EXIT\_FAILURE.

## Parameter Prediction

Header: libxs\_predict.h

Predict output parameters from input parameters using fingerprint-guided polynomial interpolation and kNN classification. Supports incremental training, model persistence, confidence-gated evaluation, output transforms, and thread-safe operation.

## Mode Flags

```
typedef enum libxs_predict_mode_t {
    LIBXS_PREDICT_AUTO      = 0,
    LIBXS_PREDICT_INTERPOLATE = 1,
    LIBXS_PREDICT_CLASSIFY   = 2,
    LIBXS_PREDICT_TEMPORAL   = 4
} libxs_predict_mode_t;
```

ORable flags controlling prediction behavior:

- AUTO (0): fingerprint decides per-output (default).
- INTERPOLATE: force polynomial for all outputs.
- CLASSIFY: force kNN vote for all outputs.
- TEMPORAL: timeseries mode -- enables recency weighting (recent neighbors preferred), continuous output (no snap-to-nearest), and local coherence smoothing across horizon steps. Only effective when set\_series was called (nseries > 0). For timeseries models without this flag, temporal heuristics auto-enable only when the query falls outside the training bounding box.

Example: LIBXS\_PREDICT\_CLASSIFY | LIBXS\_PREDICT\_TEMPORAL forces kNN-weighted projection forward with temporal heuristics.

## Input Decomposition and Feature Selection

```
typedef enum libxs_predict_decompose_t {  
    LIBXS_PREDICT_RAW      = 0,  
    LIBXS_PREDICT_SPREAD  = 1,  
    LIBXS_PREDICT_PCA      = 2,  
    LIBXS_PREDICT_SETDIFF  = 3,  
    LIBXS_PREDICT_FISHER   = 4,  
    LIBXS_PREDICT_RF       = 5,  
    LIBXS_PREDICT_HKNN     = 6  
} libxs_predict_decompose_t;
```

Controls input processing and prediction strategy:

- RAW (0): no decomposition, standard kNN (default).
- SPREAD: sum/diff modes for anti-correlated pairs (2 series). Forward:  $\text{sum} = (A+B)/2$ ,  $\text{diff} = (A-B)/2$ . Inverse:  $A = \text{sum} + \text{diff}$ ,  $B = \text{sum} - \text{diff}$ . Only effective when `nseries == 2`.
- PCA: principal component rotation of the input space. At build time, computes eigenvectors of the input covariance matrix (Jacobi iteration), stores the rotation matrix, and transforms all entries into PC space. Weights are auto-set to zero out PCs below the 95% cumulative variance threshold (dimensionality reduction). Applied via `libxs_gemm`.
- SETDIFF: automatic feature selection using `libxs_setdiff`. At build time, scores each input dimension by class separability (per-class-pair setdiff with 5% tolerance, range-normalized). Zeroes weights for features below the median score. Supervised (uses output labels).
- FISHER: automatic feature selection using Fisher's discriminant ratio (between-class / within-class variance). At build time, computes per-feature Fisher score, applies sqrt weighting above median, zeroes below. Supervised. Generally best for kNN classification.
- RF: Random Forest classification. At build time, trains an ensemble of 100 decision trees per output (bootstrap sampling,  $\sqrt{\text{ninputs}}$  random features per split, Gini impurity). At eval time, returns majority vote across trees. Per-output confidence = vote fraction. Excels on high-dimensional classification (37+ features). Persisted in save/load (compact on-disk format: 15 bytes/node).
- HKNN: hierarchical kNN. At build time, partitions data using a Fisher-guided kd-tree (output-aware dimension selection, adaptive balance penalty) followed by Lloyd refinement. Uses Gini impurity when a single categorical output is detected. A soft cap based on the Fisher/Gini score suppresses marginal splits near target cluster count. Inference uses standard kNN within each partition. Best for structured parameter prediction where output-aware spatial partitioning outperforms geometric k-means.

PCA/SPREAD decomposition is applied at build time to stored entries and at eval time to user queries. Inverse prediction returns inputs in raw (user) space. Feature selection modes (SETDIFF, FISHER) only set weights at build time. RF replaces the kNN eval path entirely. HKNN replaces only the clustering step (kNN inference is unchanged).

## Output Transforms

```
typedef enum libxs_predict_transform_t {  
    LIBXS_PREDICT_IDENTITY = 0,  
    LIBXS_PREDICT_LOG      = 1,  
    LIBXS_PREDICT_SQRT     = 2  
} libxs_predict_transform_t;
```

Per-output transforms applied transparently:

- IDENTITY (0): no transform (default).
- LOG: forward =  $\log(x+1)$ , inverse =  $\exp(x)-1$ .

- SQRT: forward =  $\sqrt{x}$ , inverse =  $x \cdot x$ .

The forward transform is applied during push, the inverse during eval. The fingerprint and kNN operate in transformed space where heavy-tailed data is smoother.

## Create and Destroy

```
libxs_predict_t* libxs_predict_create(int ninputs, int noutputs);
void libxs_predict_destroy(libxs_predict_t* model);
libxs_lock_t* libxs_predict_lock(libxs_predict_t* model);
```

Create a model for the given input/output dimensionality. Returns NULL on invalid arguments. Destroy accepts NULL. The lock accessor returns a pointer to the model's internal lock for use with push/eval (NULL if model is NULL).

## Configuration

```
void libxs_predict_set_mode(libxs_predict_t* model, int mode);
```

Set prediction mode (ORable flags from `libxs__predict__mode_t`). Default is `LIBXS_PREDICT_AUTO`.

```
void libxs_predict_set_weights(libxs_predict_t* model,
    const double weights[]);
```

Set per-dimension input weights for distance computation. Larger weight means the dimension contributes more. Pass NULL to reset to uniform weighting. Must be called before `libxs__predict__build`.

```
void libxs_predict_set_transform(libxs_predict_t* model,
    int output, int transform);
```

Set per-output transform. The output parameter is 0-based, or -1 to set all outputs. Must be called before pushing entries. Transforms are persisted in save/load.

```
void libxs_predict_set_refine(libxs_predict_t* model,
    int iterations);
```

Set the number of forward-inverse-forward refinement iterations applied during eval. Default (0): iterate automatically when per-output confidence drops below 0.9. When set to >0, always perform the given number of iterations regardless of confidence.

## Timeseries Configuration

```
void libxs_predict_set_series(libxs_predict_t* model,
    int nseries, int window);
void libxs_predict_set_target(libxs_predict_t* model,
    int target);
void libxs_predict_set_decompose(libxs_predict_t* model,
    int decompose);
void libxs_predict_set_diff(libxs_predict_t* model,
    int order);
```

Declare timeseries structure for automatic sliding-window construction. The model's `ninputs` must equal `nseries * window`; `noutputs` is the forecast horizon.

When `set_series` has been called, `push(lock, model, values, NULL)` accumulates one timestep (`nseries` values per call). The build step then constructs all valid sliding windows internally: input = concatenated windows across all series, output = the next horizon values of the target series.

`set_target` selects which series to predict (0-based, default 0). `set_decompose` selects input processing mode (see above).

set\_diff enables auto-differencing for non-stationary series:

- set\_diff(model, 0): auto-detect order from fingerprint decay.
- set\_diff(model, d): explicit order d (1 = linear detrend).
- Default is disabled (-1).

Single-series example (sunspots):

```
model = libxs_predict_create(WINDOW, HORIZON);
libxs_predict_set_series(model, 1, WINDOW);
for (t = 0; t < n; ++t)
    libxs_predict_push(NULL, model, &series[t], NULL);
libxs_predict_build(model, 0, 2, 0);
```

Multi-series example (anti-correlated pair):

```
model = libxs_predict_create(WINDOW * 2, HORIZON);
libxs_predict_set_series(model, 2, WINDOW);
libxs_predict_set_target(model, 0);
libxs_predict_set_decompose(model, LIBXS_PREDICT_SPREAD);
for (t = 0; t < n; ++t) {
    double vals[2] = {A[t], B[t]};
    libxs_predict_push(NULL, model, vals, NULL);
}
libxs_predict_build(model, 0, 2, 0);
```

Non-stationary example (trending stock prices):

```
model = libxs_predict_create(WINDOW * 2, HORIZON);
libxs_predict_set_mode(model, LIBXS_PREDICT_TEMPORAL);
libxs_predict_set_series(model, 2, WINDOW);
libxs_predict_set_target(model, 0);
libxs_predict_set_decompose(model, LIBXS_PREDICT_SPREAD);
libxs_predict_set_diff(model, 0);
for (t = 0; t < n; ++t) {
    double vals[2] = {price_A[t], price_B[t]};
    libxs_predict_push(NULL, model, vals, NULL);
}
libxs_predict_build(model, 0, 2, 0);
```

## Training

```
int libxs_predict_push(libxs_lock_t* lock,
    libxs_predict_t* model,
    const double inputs[], const double outputs[]);
```

Push one training entry. When set\_series was called and outputs is NULL, inputs holds nseries values representing one timestep; sliding windows are constructed at build time. Returns EXIT\_SUCCESS or EXIT\_FAILURE.

```
int libxs_predict_build(libxs_predict_t* model,
    int nclusters, int order, double quality);
```

Build the model from pushed entries. nclusters=0 selects sqrt(n) clusters automatically. order>0 uses at most that order, order=0 auto-optimizes, order<0 scans up to |order|.

The quality parameter (0..1) controls model compression:

- quality=0: no compression (default, all entries retained).
- quality>0: leave-one-out pass removes entries that are perfectly predictable by their neighbors.

```
int libxs_predict_build_task(libxs_lock_t* lock,
    libxs_predict_t* model, int nclusters, int order,
    double quality, int tid, int ntasks);
```

Per-thread collective form. All threads call with same parameters. tid=0 performs the build, others spin-wait.

## Evaluation

```
void libxs_predict_eval(libxs_lock_t* lock,
    const libxs_predict_t* model,
    const double inputs[], double outputs[],
    libxs_predict_info_t* info, int nblend);
```

Predict outputs for given inputs. nblend controls multi-cluster blending: 1=nearest only, 0=auto. The info pointer (optional) receives per-output confidence, variance, error bounds, and mode flags.

```
void libxs_predict_inverse(libxs_lock_t* lock,
    const libxs_predict_t* model,
    const double target_outputs[], double inputs[],
    libxs_predict_info_t* info);
```

Inverse prediction: find inputs that produce desired outputs.

```
void libxs_predict_eval_batch(
    const libxs_predict_t* model,
    const double inputs_batch[], double outputs_batch[],
    int count, int nblend);
```

```
void libxs_predict_eval_batch_task(
    const libxs_predict_t* model,
    const double inputs_batch[], double outputs_batch[],
    int count, int nblend, int tid, int ntasks);
```

Batch prediction. The task variant distributes queries across threads by slice.

## Query and Access

```
void libxs_predict_query(const libxs_predict_t* model,
    libxs_predict_query_t* info);
```

```
void libxs_predict_get(const libxs_predict_t* model,
    int index, double inputs[], double outputs[]);
```

query fills a statistics struct (cluster count, entry count, compression ratio, polynomial order, diff\_order). get retrieves the i-th pushed entry (0-based).

## Persistence

```
int libxs_predict_save(const libxs_predict_t* model,
    void* buffer, size_t* size);
libxs_predict_t* libxs_predict_load(
    const void* buffer, size_t size);
```

Save: pass buffer=NULL to query required size, then allocate and call again. Load: returns a ready-to-eval model or NULL.

## CSV Import

```
int libxs_predict_load_csv(libxs_predict_t* model,
    const char filename[], const char delims[],
    const char* inputs[], int ninputs,
    const char* outputs[], int noutputs);
```

Load delimited text and push entries. delims=NULL auto-detects separator. Returns number of entries pushed, or -1 on error.

Structures

```
typedef struct libxs_predict_info_t {
    const double* values;
    const double* error;
    const double* confidence;
    const double* variance;
    const int* interpolated;
    int noutputs;
    int cluster;
    double distance;
} libxs_predict_info_t;
```

Populated by eval when info is non-NULL. confidence holds per-output kNN vote fraction (0..1). variance holds per-output neighbor disagreement. cluster gives the assigned cluster index (-1 if blended). distance gives normalized distance to nearest centroid.

```
typedef struct libxs_predict_query_t {
    double compression;
    int order;
    int nclusters;
    int nentries;
    int iterations;
    int diff_order;
} libxs_predict_query_t;
```

Populated by query. compression = raw/model size ratio. order = polynomial order used. diff\_order = differencing order (0 if disabled).

Registry

Header: libxs\_reg.h

Thread-safe key-value store backed by a hash table with per-thread caching.

Configuration Macros

Macro	Default	Description
LIBXS_REGKEY_MAXSIZE	64	Maximum key size in bytes
LIBXS_REGISTRY_NBUCKETS	64	Initial hash-table buckets (power of two); registries grow dynamically
LIBXS_REGCACHE_NENTRIES	16	Thread-local cache entries per thread (power of two; 0 disables caching)

Types

```
typedef struct libxs_registry_t libxs_registry_t;    /* opaque */

typedef struct libxs_registry_info_t {
    size_t capacity, size, nbytes;
} libxs_registry_info_t;
```



## Lifetime

```
libxs_registry_t* libxs_registry_create(void);
```

Create a new registry. Returns `NULL` in case of an error.

```
void libxs_registry_destroy(libxs_registry_t* registry);
```

Destroy a registry and release all entries.

```
libxs_lock_t* libxs_registry_lock(libxs_registry_t* registry);
```

Return a pointer to the registry's internal lock. The returned lock can be passed as the `lock` argument to other registry functions (e.g., to hold the lock across multiple operations).

## Iteration

```
void* libxs_registry_begin(libxs_registry_t* registry,  
    const void** key, size_t* cursor);  
void* libxs_registry_next(libxs_registry_t* registry,  
    const void** key, size_t* cursor);
```

Enumerate entries. Initialize `*cursor` to 0 before the first `begin` call. Each call returns the value pointer and writes the key pointer to `*key`, or returns `NULL` when iteration is complete.

## Access

```
void* libxs_registry_set(libxs_registry_t* registry,  
    const void* key, size_t key_size,  
    const void* value_init, size_t value_size,  
    libxs_lock_t* lock /* = NULL */);
```

Insert or update a key-value pair. The key must be binary-reproducible (zero-initialize padding). `value_init` may be `NULL` to defer initialization. Re-registering an existing key reallocates if the new value is larger. When `lock` is `NULL` the registry's internal lock is used. Returns a pointer to the stored value.

```
void* libxs_registry_get(const libxs_registry_t* registry,  
    const void* key, size_t key_size,  
    libxs_lock_t* lock /* = NULL */);
```

Look up a value by key. Returns `NULL` if not found.

```
int libxs_registry_get_copy(const libxs_registry_t* registry,  
    const void* key, size_t key_size,  
    void* value_out, size_t value_size,  
    libxs_lock_t* lock /* = NULL */);
```

Thread-safe query: copies up to `value_size` bytes of the stored value into `value_out` under the lock. Unlike `libxs_registry_get`, the caller never receives a raw pointer into the registry's internal storage, making this safe under concurrent modifications. Returns non-zero if the key was found, zero otherwise.

```
int libxs_registry_has(libxs_registry_t* registry,  
    const void* key, size_t key_size,  
    libxs_lock_t* lock /* = NULL */);
```

Check whether a key exists. Non-zero if found.

```
size_t libxs_registry_value_size(libxs_registry_t* registry,  
    const void* key, size_t key_size,  
    libxs_lock_t* lock /* = NULL */);
```

Query the stored value size in bytes. Returns 0 if not found.

```
void libxs_registry_remove(libxs_registry_t* registry,
    const void* key, size_t key_size,
    libxs_lock_t* lock /* = NULL */);
```

Remove a key-value pair and release its memory.

```
int libxs_registry_extract(libxs_registry_t* registry,
    const void* key, size_t key_size,
    void* value_out, size_t value_size,
    libxs_lock_t* lock /* = NULL */);
```

Atomically retrieve and remove a key-value pair. Copies up to `value_size` bytes of the stored value into `value_out` (may be NULL to discard the value), then removes the entry and releases its memory. The lookup, copy, and removal are performed under a single lock hold, eliminating the race that exists when calling `libxs_registry_get` followed by `libxs_registry_remove` separately. Returns non-zero if the key was found, zero otherwise.

## Status

```
int libxs_registry_info(libxs_registry_t* registry,
    libxs_registry_info_t* info);
```

Query registry capacity, number of entries, and total bytes stored.

## Random Number Generator

Header: `libxs_rng.h`

Thread-safe pseudo-random number generation (SplitMix64 with per-thread TLS state).

## Matrix Initialization Macros

```
LIBXS_MATRNG(INT_TYPE, REAL_TYPE, ESPAN, DST, NROWS, NCOLS, LD, SCALE)
LIBXS_MATRNG_SEQ(INT_TYPE, REAL_TYPE, ESPAN, DST, NROWS, NCOLS, LD, SCALE)
LIBXS_MATRNG_OMP(INT_TYPE, REAL_TYPE, ESPAN, DST, NROWS, NCOLS, LD, SCALE)
```

Fill a column-major matrix (NROWS x NCOLS, leading dimension LD) with deterministic values. The `_OMP` variant parallelizes with OpenMP; `_SEQ` is serial.

ESPAN (exponent span) selects the initialization mode:

ESPAN == 0 Shuffle mode. A coprime permutation maps each linear index to a unique value in  $[-|\text{SCALE}|, +|\text{SCALE}|]$ . Covers the full LD\*NCOLS range (including padding).

ESPAN != 0 Adversarial exponent-span mode for stress-testing floating-point emulation (Ozaki scheme, etc.). Base values are shuffled in  $[1, 2)$ , then each column  $j$  is scaled by

$$2^{(\text{sign}(\text{ESPAN}) \setminus \text{floor}(|\text{ESPAN}| \setminus j / (\text{NCOLS}-1)))}$$

so column 0 has exponent 0 and the last column has exponent  $\pm|\text{ESPAN}|$ . Padding rows  $\setminus [\text{NROWS}, \text{LD})$  are zero-filled.

Use `+ESPAN` for matrix A and `-ESPAN` for matrix B so that  $A \setminus B$  remains well-conditioned while each operand individually has adversarial exponent range.

Example -- adversarial DGEMM test with exponent span 512:

```
int espan = 512;
LIBXS_MATRNG(int, double, +espan, a, m, k, lda, 1.0);
LIBXS_MATRNG(int, double, -espan, b, k, n, ldb, 1.0);
```

## Functions

```
void libxs_rng_set_seed(unsigned int seed);
```

Set the seed of the calling thread's PRNG state. Each thread maintains independent state via TLS. Unseeded threads start with a deterministic default (seed = 1).

```
unsigned int libxs_rng_u32(unsigned int n);
```

Return a pseudo-random value in  $[0, n)$  with uniform distribution (Lemire's nearly-divisionless method). Thread-safe.

```
double libxs_rng_f64(void);
```

Return a double-precision value in  $[0, 1)$  with full 53-bit mantissa resolution. Thread-safe.

```
void libxs_rng_seq(void* data, size_t nbytes);
```

Fill a buffer with pseudo-random bytes. Thread-safe.

## String Utilities

Header: `libxs_str.h`

Case-insensitive string search, edit distance, word-level similarity scoring, and value formatting.

### Substring Search

```
const char* libxs_stristrn(const char a[], const char b[],
    size_t maxlen);
const char* libxs_stristr(const char a[], const char b[]);
```

Case-insensitive substring search. `stristrn` limits the match length to `maxlen` characters of `b`. Returns a pointer to the first match in `a`, or `NULL`.

### Edit Distance

```
int libxs_stridist(const char a[], const char b[]);
```

Case-insensitive character-level edit distance (Levenshtein) between two strings. Returns the minimum number of single-character insertions, deletions, or substitutions to transform `a` into `b` (ignoring case). Returns -1 for `NULL` input.

### Word Matching

```
int libxs_strimatch(const char a[], const char b[],
    const char delims[], int* count);
```

Word-level fuzzy matching. Counts how many words in `a` (or `b`) appear in the other string (case-insensitive, symmetric). Optional `delims` define word separators (default: space, tab, semicolon, comma, colon, dash). Optional `count` receives the total word count. Returns -1 for invalid input.

## Word Similarity

```
typedef enum libxs_strisimilar_t {
    LIBXS_STRISIMILAR_GREEDY,
    LIBXS_STRISIMILAR_TWOOPT,
    LIBXS_STRISIMILAR_DEFAULT = LIBXS_STRISIMILAR_GREEDY
} libxs_strisimilar_t;

int libxs_strisimilar(const char a[], const char b[],
    const char delims[], libxs_strisimilar_t kind, int* order);
```

Word-level similarity score combining edit distance and word-order analysis.

Strings are split into words using the same delimiters as `libxs_strimatch`. Each word in `a` is matched to a word in `b` via minimum-cost bipartite matching, where the cost of a pair is the character-level Levenshtein distance (case-insensitive). Unmatched words (when the strings have different word counts) contribute their full length.

The matching strategy is selected by `kind`:

**GREEDY** Picks the globally cheapest pair first. **TWOOPT** Refines the greedy result by iteratively swapping pairs whenever a swap reduces total cost.

The optional `order` output receives the number of pairwise inversions among matched words (Kendall tau distance), measuring how much the word order differs (0 = same order).

Returns the total edit distance (0 for identical word sets in any order), or -1 for invalid input.

## Value Formatting

```
size_t libxs_format_value(char buffer[], int buffer_size,
    size_t nbytes, const char scale[], const char* unit, int base);
```

Format a scalar value with SI-style scaling. Example:

```
libxs_format_value(buf, sizeof(buf), nbytes, "KMGT", "B", 10)
```

produces a human-readable byte count such as "128 KB". Returns the value in the selected unit so the caller can decide whether to print the buffer.

## Synchronization

Header: `libxs_sync.h`

Thread-local storage, atomic operations, lock abstractions (spin/mutex/rwlock/atomic), file locking, and stdio synchronization.

## Thread-Local Storage

`LIBXS_TLS`

Storage-class qualifier for thread-local variables. Maps to `__thread`, `__declspec(thread)`, or `thread_local` depending on the compiler. Defined as empty when TLS is unavailable or disabled (`LIBXS_NO_TLS`).

Atomic Operations

The header provides a suite of macros for atomic loads, stores, compare-and-swap, and arithmetic. The implementation selects among GCC builtins (`__atomic_*`), legacy GCC sync builtins, Windows Interlocked intrinsics, or no-op fallbacks depending on the compiler and `LIBXS_SYNC` setting.

Macro	Description
<code>LIBXS_ATOMIC_LOAD(SRC, KIND)</code>	Atomic load
<code>LIBXS_ATOMIC_STORE(DST, VALUE, KIND)</code>	Atomic store
<code>LIBXS_ATOMIC_STORE_ZERO(DST, KIND)</code>	Atomic store of zero
<code>LIBXS_ATOMIC_CMPSWP(DST, OLDVAL, NEWVAL, KIND)</code>	Compare-and-swap
<code>LIBXS_ATOMIC_FETCH_OR(DST, VALUE, KIND)</code>	Fetch-and-or
<code>LIBXS_ATOMIC_FETCH_ADD(DST, VALUE, KIND)</code>	Fetch-and-add
<code>LIBXS_ATOMIC_FETCH_SUB(DST, VALUE, KIND)</code>	Fetch-and-subtract
<code>LIBXS_ATOMIC_ADD_FETCH(DST, VALUE, KIND)</code>	Add-and-fetch
<code>LIBXS_ATOMIC_SUB_FETCH(DST, VALUE, KIND)</code>	Subtract-and-fetch
<code>LIBXS_ATOMIC_TRYLOCK(DST, KIND)</code>	Try-lock (returns acquired state)
<code>LIBXS_ATOMIC_ACQUIRE(DST, NPAUSE, KIND)</code>	Spin-acquire with backoff
<code>LIBXS_ATOMIC_RELEASE(DST, KIND)</code>	Release (store zero)
<code>LIBXS_ATOMIC_SYNC(KIND)</code>	Full memory fence

The `KIND` parameter selects the memory order but is ignored on most backends (sequential consistency is always used).

Lock Abstraction

```
LIBXS_LOCK_TYPE(KIND)
LIBXS_LOCK_INIT(KIND, LOCK, ATTR)
LIBXS_LOCK_DESTROY(KIND, LOCK)
LIBXS_LOCK_ACQUIRE(KIND, LOCK)
LIBXS_LOCK_TRYLOCK(KIND, LOCK)
LIBXS_LOCK_RELEASE(KIND, LOCK)
```

Generic lock interface parameterized by `KIND`:

KIND	Backend
<code>LIBXS_LOCK_SPINLOCK</code>	Atomic spin-lock
<code>LIBXS_LOCK_MUTEX</code>	Pthreads / Windows <code>CRITICAL_SECTION</code>
<code>LIBXS_LOCK_RWLOCK</code>	Pthreads reader-writer lock
<code>LIBXS_LOCK_ATOMIC</code>	Lightweight atomic lock

The default lock kind used by the library is `LIBXS_LOCK` (resolves to one of the above based on build configuration).

```
typedef LIBXS_LOCK_TYPE(LIBXS_LOCK) libxs_lock_t;
```

General-purpose lock type. Instances of `libxs_lock_t` are used by the registry and other library components; users may also create their own.

Reader-writer variants are available for `LIBXS_LOCK_RWLOCK`:

```
LIBXS_LOCK_ACQREAD(KIND, LOCK)
LIBXS_LOCK_RELREAD(KIND, LOCK)
LIBXS_LOCK_TRYREAD(KIND, LOCK)
```

## File Locking

```
LIBXS_FLOCK(FILE)
LIBXS_FUNLOCK(FILE)
```

Per-file locking for thread-safe I/O. Maps to `flockfile/funlockfile` on POSIX, `_lock_file/_unlock_file` on Windows, or no-ops when synchronization is disabled.

## Functions

```
unsigned int libxs_nrank(void);
```

Return the number of MPI ranks.

```
unsigned int libxs_nrank(void);
```

Return the MPI rank of the calling process.

```
unsigned int libxs_rid(void);
```

Return a rank ID of the calling process.

```
unsigned int libxs_pid(void);
```

Return the process ID of the calling process.

```
unsigned int libxs_tid(void);
```

Return a zero-based, consecutive thread ID for the calling thread. TID = 0 does not necessarily correspond to the main thread.

```
void libxs_stdio_acquire(void);
void libxs_stdio_release(void);
```

Acquire/release a global lock around console output. The macros `LIBXS_STDIO_ACQUIRE()` and `LIBXS_STDIO_RELEASE()` expand to these calls.

## Timer

Header: `libxs_timer.h`

High-resolution timing via a calibrated TSC when available, with an OS real-time clock fallback.

## Types

```
typedef unsigned long long libxs_timer_tick_t;
```

Integer type representing a single timer sample.

```
typedef struct libxs_timer_info_t {
    int tsc;    /* non-zero if a calibrated TSC is used */
} libxs_timer_info_t;
```

Functions

```
int libxs_timer_info(libxs_timer_info_t* info);
```

Query timer properties. `info->tsc` is non-zero when RDTSC (x86), CNTVCT\_EL0 (AArch64), or MFTB (PPC64) is available. Triggers lazy library initialization if necessary.

```
libxs_timer_tick_t libxs_timer_tick(void);
```

Sample the current tick of the monotonic timer. The tick unit is platform-specific; convert with `libxs_timer_duration` or `libxs_timer_ncycles`.

```
libxs_timer_tick_t libxs_timer_ncycles(
    libxs_timer_tick_t tick0, libxs_timer_tick_t tick1);    /* inline */
```

Unsigned difference `tick1 - tick0`, handling wrap-around safely. Implemented via `LIBXS_DELTA`.

```
double libxs_timer_duration(
    libxs_timer_tick_t tick0, libxs_timer_tick_t tick1);
```

Convert a pair of ticks to elapsed wall-clock time in seconds.

Low-Level Utilities and SIMD

Header: `libxs_utils.h`

Target-attribute machinery, ISA feature gates, intrinsic fix-ups, bit-scan operations, and AVX-512 inline helpers.

Target Attributes

```
LIBXS_INTRINSICS(TARGET)
```

Expands to a `__attribute__((target(...)))` clause for the given ISA level. Used to compile individual functions for a higher ISA than the baseline without changing compiler flags. The macro is a no-op when the baseline already covers the requested level or when the compiler does not support multi-versioning.

```
LIBXS_ATTRIBUTE_TARGET(TARGET)
```

Lower-level macro that resolves `TARGET` (an integer ISA constant such as `LIBXS_X86_AVX512`) to the corresponding target string, e.g. `target("avx2,fma,avx512f,...")`.

ISA Feature Gates

Preprocessor symbols defined when the compiler can generate code for a given ISA:

Macro	ISA level
LIBXS_INTRINSICS_X86	x86 baseline (SSE2)
LIBXS_INTRINSICS_SSE3	SSE3
LIBXS_INTRINSICS_SSE42	SSE4.2
LIBXS_INTRINSICS_AVX	AVX
LIBXS_INTRINSICS_AVX2	AVX2 + FMA
LIBXS_INTRINSICS_AVX512	AVX-512 (F + CD + DQ + BW + VL + VNNI)

LIBXS\_STATIC\_TARGET\_ARCH records the highest ISA provided by the compiler flags. LIBXS\_MAX\_STATIC\_TARGET\_ARCH records the highest ISA reachable via target attributes. Target attributes (LIBXS\_ATTRIBUTE\_TARGET\_\*) are defined up to LIBXS\_X86\_AVX10\_512 depending on compiler version.

Intrinsic Fix-ups

Portability wrappers for intrinsics whose signatures differ across compilers:

Macro	Purpose
LIBXS_INTRINSICS_LOADU_SI128	_mm_loadu_si128
LIBXS_INTRINSICS_MM512_LOAD_PS/PD	Unaligned 512-bit float/double load
LIBXS_INTRINSICS_MM512_STREAM_*	Non-temporal 512-bit stores
LIBXS_INTRINSICS_MM256_STORE_EPI32	256-bit integer store
LIBXS_INTRINSICS_MM512_SET_EPI16	Portable 512-bit set from 32 int16 values
LIBXS_INTRINSICS_MM512_UNDEFINED_*	Undefined-value initializers (zero in debug)
LIBXS_INTRINSICS_MM512_EXTRACTI64X4_EPI64	256-bit extract from 512-bit
LIBXS_INTRINSICS_MM512_ABS_PS	Absolute value of packed floats
LIBXS_INTRINSICS_MM512_MASK_I32GATHER_EPI32	Masked gather
LIBXS_INTRINSICS_MM512_STORE/LOAD_MASK*	Mask register I/O (8/16/32/64-bit)
LIBXS_INTRINSICS_MM512_CVTU32_MASK*	Convert uint32 to mask register

Bit-Scan Operations

```
LIBXS_INTRINSICS_BITSCANFWD32(N)    /* count trailing zeros, 32-bit */
LIBXS_INTRINSICS_BITSCANFWD64(N)    /* count trailing zeros, 64-bit */
LIBXS_INTRINSICS_BITSCANBWD32(N)    /* bit index of highest set bit, 32-bit */
LIBXS_INTRINSICS_BITSCANBWD64(N)    /* bit index of highest set bit, 64-bit */
```

Use GCC \_\_builtin\_ctz/\_\_builtin\_clz when available, Windows \_BitScanForward/\_BitScanReverse intrinsics on MSVC, or portable software fallbacks (\*\_SW variants). All return 0 when N is 0.

Derived Macros

```
LIBXS_NBITS(N)          /* minimum bits to represent N */
LIBXS_ISQRT2(N)         /* fast power-of-two approximation of sqrt(N) */

unsigned int LIBXS_ILOG2(unsigned long long n); /* ceil(log2(n)) */
```

LIBXS\_ILOG2 is a function (not a macro) and handles the n = 0 case.



## MXCSR (FP Environment)

Macros for preserving and restoring the x86 MXCSR register (floating-point control/status). On non-x86 platforms these are no-ops.

Macro	Description
LIBXS_MXCSR_FTZ	Flush-to-zero flag (0x8000 on x86, 0 otherwise)
LIBXS_MXCSR_DAZ	Denormals-are-zero flag (0x0040 on x86, 0 otherwise)
LIBXS_MXCSR_GET()	Read the MXCSR register ( <code>_mm_getcsr()</code> on x86, 0 otherwise)
LIBXS_MXCSR_SET(VALUE)	Write the MXCSR register ( <code>_mm_setcsr()</code> on x86, no-op otherwise)

## AVX-512 Inline Functions

Available only when `LIBXS_INTRINSICS_AVX512` is defined.

```
__m512i libxs_mulhi_epu32(__m512i a, __m512i b);    /* inline */
```

Unsigned 32-bit high-multiply for 16 lanes:  $\text{floor}(a * b / 2^{32})$ . Emulates the missing `_mm512_mulhi_epu32` via even/odd `_mm512_mul_epu32`.

```
__m512i libxs_mod_u32x16(__m512i x,
    unsigned int p, unsigned int rcp);                /* inline */
```

Vectorized Barrett reduction:  $x \bmod p$  for 16 uint32 lanes. `rcp` is the Barrett reciprocal from `libxs_barrett_rcp()`. This is the SIMD counterpart of the scalar `libxs_mod_u32`.

## Appendix

### Scripts

#### Build Support

- `tool_source.sh` -- Generate `libxs_source.h` (header-only amalgamation). Called by `make headers`.
- `tool_version.sh` -- Extract version from Git tags. Generates `libxs_version.h`.
- `libxs.pc.in` -- Template for the installed `.pc` file.

#### Code Quality

- `tool_analyze.sh` -- Run `cppcheck` on `src/`.
- `tool_checkabi.sh` -- Compare exported symbols against a reference.
- `tool_clangformat.sh` -- Run `clang-format` and `shellcheck`.
- `tool_normalize.sh` -- Strip trailing whitespace, fix line endings.

## Documentation

**md2pptx.py** -- Convert Markdown to PowerPoint. Splits on --- rules or ## headings (auto-detected). Requires python-pptx and lxml.

## Development

**tool\_cpufreq.sh** -- Print CPU topology (sockets, cores, threads).

**tool\_getenvvars.sh** -- List environment variables used in the source tree.

**tool\_pexec.sh** -- Parallel command execution with CPU affinity (-h for options).

**tool\_gitaddforks.sh** -- Add GitHub forks as Git remotes.

**tool\_gitprune.sh** -- Aggressive Git housekeeping (gc, fsck, repack).