AXILOG

SRS-001-DETERMINISTIC-HASH

Deterministic Hash Table

Final v0.1 L1 [D1]

Last updated 20 January 2026
Compliance MISRA-C:2012 · ISO 26262
Changelog 1 entry

Contents

1. Purpose 3
2. Requirements 3
   2.1 Hash Function 3
   2.2 Collision Resolution 4
   2.3 Iteration Order 4
   2.4 Memory Management 5
   2.5 Capacity and Bounds 5
3. Verification Criteria 6
4. Implementation 7
5. Data Structures 7
   5.1 Result Codes 7
   5.2 Entry Structure 8
   5.3 Table Structure 8
6. API Specification 8
   6.1 Initialization 8
   6.2 Insert 9
   6.3 Get 9
   6.4 Iterate 9
7. Design Rationale 10
   7.1 Why FNV-1a? 10
   7.2 Why Linear Probing? 10
   7.3 Why Insertion-Order Iteration? 10
8. Test Mapping 11

9. References 11
10. Revision History 11

Component: Containers **Compliance:** MISRA-C:2012 · ISO 26262 **Applicability:** All certifiable-* components requiring key-value storage

---

1. Purpose

The system shall provide a hash table implementation with deterministic behavior, including:

1. Platform-independent hash computation
2. Deterministic collision resolution
3. Insertion-order iteration (not hash-order)
4. Static memory allocation (no malloc/free)

---

2. Requirements

2.1 Hash Function

SRS-001.1: The system shall use the FNV-1a hash algorithm for all key hashing operations.

Rationale: FNV-1a produces identical results across all platforms using only integer arithmetic. No floating-point or platform-specific instructions are required.

Mathematical Definition:

```
hash ← 2166136261 (FNV offset basis, 32-bit)
FOR each byte c in key:
    hash ← hash XOR c
    hash ← hash × 16777619 (FNV prime)
RETURN hash
```

Verification: Unit tests comparing hash output against known test vectors.

SRS-001.2: The hash function shall produce identical output regardless of:

- Target architecture (x86, ARM, RISC-V)
- Compiler vendor (GCC, Clang, MSVC)

- Optimization level (-O0 through -Ofast)
- Endianness (little-endian, big-endian)

Rationale: Deterministic hashing is foundational for reproducible model weight storage and lookup.

Verification: Cross-platform testing with checksum comparison.

2.2 Collision Resolution

SRS-001.3: The system shall use linear probing for collision resolution.

Mathematical Definition:

```
probe_index(hash, attempt, capacity) = (hash + attempt) mod capacity
```

Where `attempt` increments from 0 until an empty slot is found.

Rationale: Linear probing is deterministic (probe sequence depends only on hash value), cache-friendly, and simple to verify.

Verification: Unit tests with keys that produce hash collisions.

SRS-001.4: Duplicate key insertion shall return `D_TABLE_KEY_EXISTS` without modifying the existing entry.

Rationale: Predictable behavior on duplicate keys prevents silent data corruption.

Verification: Unit test `test_duplicate_key`.

2.3 Iteration Order

SRS-001.5: Iteration shall occur in insertion order, not hash order.

Rationale: Hash order varies with table capacity and load factor. Insertion order is deterministic and reproducible.

Implementation: Maintain a separate `insertion_order[]` array tracking the order of insertions.

Verification: Unit test `test_deterministic_iteration_order`.

SRS-001.6: Two tables with identical insertion sequences shall produce identical iteration sequences.

Mathematical Definition:

$$\forall \text{ sequences } S_1, S_2: \quad S_1 = S_2 \Rightarrow \text{iterate}(\text{table}_1) = \text{iterate}(\text{table}_2)$$

Verification: Unit test `test_hash_consistency` with memory state comparison.

2.4 Memory Management

SRS-001.7: The hash table shall use caller-provided buffers with no dynamic allocation.

Rationale: Dynamic allocation introduces non-deterministic timing and potential fragmentation. Safety-critical systems require bounded, predictable memory usage.

API Pattern:

```
uint8_t buffer[REQUIRED_SIZE];
d_table_t table;
d_table_init(&table, buffer, sizeof(buffer));
```

Verification: Static analysis confirming no malloc/free calls.

SRS-001.8: Initialization shall zero all memory to prevent uninitialized data from affecting behavior.

Rationale: Uninitialized memory can contain address-dependent garbage, breaking determinism.

Verification: Unit test `test_hash_consistency` comparing memory states across multiple runs.

2.5 Capacity and Bounds

SRS-001.9: The table shall enforce a maximum capacity specified at initialization.

Rationale: Unbounded growth requires dynamic allocation. Fixed capacity enables static memory analysis.

Verification: Unit test `test_capacity_limit`.

SRS-001.10: Insertion into a full table shall return `D_TABLE_FULL` without modifying any state.

Rationale: Predictable failure behavior enables proper error handling.

Verification: Unit test `test_capacity_limit`.

---

3. Verification Criteria

V-001.1: Cross-Platform Parity

Unit tests shall verify identical hash output on at least two CPU architectures.

Test Matrix:

- x86_64 (Ubuntu 24.04, GCC 13)
- ARM64 (Raspberry Pi, GCC 13)
- RISC-V (QEMU, GCC 13) — optional

Pass Criteria: Byte-identical memory states after identical insertion sequences.

V-001.2: Memory State Comparison

The `test_hash_consistency` suite shall:

1. Execute identical workloads N times (N ≥ 3)
2. Compare byte-for-byte memory states between runs
3. Verify all comparisons are identical

Pass Criteria: Zero bytes differ between any pair of runs.

V-001.3: No Floating-Point Instructions

Static analysis or disassembly shall verify that `deterministic_hash.c` contains no floating-point instructions.

Method: `objdump -d` inspection for FPU mnemonics.

Pass Criteria: Zero floating-point instructions in compiled binary.

V-001.4: Iteration Order Stability

Unit tests shall verify that iteration order depends only on insertion order, not on:

- Table capacity
- Hash function output
- Memory addresses

Pass Criteria: Identical iteration sequences for identical insertion sequences.

---

4. Implementation

Files:

- `include/deterministic_hash.h` — API specification
- `src/containers/deterministic_hash.c` — Implementation
- `tests/unit/test_hash_basic.c` — Functional tests
- `tests/unit/test_hash_consistency.c` — Determinism tests

Traceability:

- Code: `@traceability SRS-001-DETERMINISM`
- Tests: Link to this document in header

---

5. Data Structures

5.1 Result Codes

```
typedef enum {
    D_TABLE_OK = 0,         /* Success */
    D_TABLE_KEY_EXISTS,     /* Insert failed: key exists */
    D_TABLE_NOT_FOUND,      /* Get failed: key not found */
    D_TABLE_FULL,           /* Insert failed: at capacity */
    D_TABLE_ERROR           /* Generic error */
} d_table_res_t;
```

5.2 Entry Structure

```
typedef struct {
    char key[D_TABLE_MAX_KEY_LEN];  /* Null-terminated key */
    int32_t value;                   /* Associated value */
    bool occupied;                   /* Slot contains valid data */
} d_table_entry_t;
```

5.3 Table Structure

```
typedef struct {
    d_table_entry_t *entries;       /* Hash slots (caller-owned) */
    uint32_t *insertion_order;      /* Iteration order tracking */
    uint32_t capacity;              /* Maximum entries */
    uint32_t count;                 /* Current entry count */
} d_table_t;
```

---

6. API Specification

6.1 Initialization

```
d_table_res_t d_table_init(d_table_t *table, uint8_t *buffer, size_t
buffer_size);
```

Preconditions:

- `table ≠ NULL`
- `buffer ≠ NULL`
- `buffer_size ⩾ D_TABLE_MIN_BUFFER_SIZE`

Postconditions:

- `table→count == 0`
- `table→capacity > 0`
- All entries marked as unoccupied
- Buffer memory zeroed

6.2 Insert

```
d_table_res_t d_table_insert(d_table_t *table, const char *key, int32_t value);
```

Preconditions:

- `table` is initialized
- `key ≠ NULL`
- `strlen(key) < D_TABLE_MAX_KEY_LEN`

Postconditions (on success):

- `table→count` incremented by 1
- Key-value pair stored
- Insertion order recorded

6.3 Get

```
d_table_res_t d_table_get(const d_table_t *table, const char *key, int32_t
*value);
```

Preconditions:

- `table` is initialized
- `key ≠ NULL`
- `value ≠ NULL`

Postconditions (on success):

- `*value` contains the stored value

6.4 Iterate

```
void d_table_iterate(const d_table_t *table, d_table_iter_fn callback);
```

Preconditions:

- `table` is initialized
- `callback ≠ NULL`

Postconditions:

- Callback invoked for each entry in insertion order

---

7. Design Rationale

7.1 Why FNV-1a?

Alternatives considered:

- Jenkins: Good distribution but more complex
- MurmurHash: Uses multiplication that may vary with compiler
- CRC32: Hardware acceleration varies by platform
- SHA-256: Overkill for hash table indexing

Decision: FNV-1a is simple, fast, deterministic, and has no platform-specific behavior.

7.2 Why Linear Probing?

Alternatives considered:

- Quadratic probing: Probe sequence depends on attempt², may skip slots
- Double hashing: Requires second hash function
- Chaining: Requires dynamic allocation for linked lists

Decision: Linear probing is simplest to implement correctly and verify.

7.3 Why Insertion-Order Iteration?

Problem: Standard hash table iteration visits entries in hash-bucket order, which:

- Varies with table capacity
- Varies with insertion order due to collision chains
- Is effectively non-deterministic for verification purposes

Solution: Track insertion order separately. O(1) space overhead per entry.

8. Test Mapping

Requirement	Test Function	Test File
SRS-001.1	(implicit in all tests)	—
SRS-001.2	test_hash_consistency	test_hash_consistency.c
SRS-001.3	test_insert_and_get	test_hash_basic.c
SRS-001.4	test_duplicate_key	test_hash_basic.c
SRS-001.5	test_deterministic_iteration_order	test_hash_basic.c
SRS-001.6	test_hash_consistency	test_hash_consistency.c
SRS-001.7	(static analysis)	—
SRS-001.8	test_hash_consistency	test_hash_consistency.c
SRS-001.9	test_capacity_limit	test_hash_basic.c
SRS-001.10	test_capacity_limit	test_hash_basic.c

9. References

- **MISRA-C:2012** — Rule 21.3 (No dynamic memory allocation)
- **ISO 26262-6:2018** — Software unit design
- **Fowler-Noll-Vo Hash** — http://www.isthe.com/chongo/tech/comp/fnv/

10. Revision History

Version	Date	Author	Changes
1.0	2026-01-20	William Murray	Full specification (expanded from stub)

Document Classification: Technical Specification
Approval Status: Approved for Implementation

Next Review: 2026-04-20
