Key-Value Pool

Purpose

The `kv_pool` module implements a **fixed-key key-value store backed by a custom memory pool**.

It is designed for systems where:

- keys are known as integer indices in a fixed range
- values are variable-sized blobs of bytes
- dynamic allocation from the general heap is undesirable or unavailable
- memory must come from a caller-provided region
- multiple threads may access the store concurrently

The module combines two things:

- a **lookup table** that maps keys to stored values
- an internal **heap allocator** that manages the memory used by those values

The result is a storage system where each key corresponds to one slot, and each valid slot points to a block allocated from the pool's private heap.

> This is not a dictionary in the desktop-software sense. Keys are not hashed, compared, or discovered dynamically. A key is just an **index into a preallocated slot table**.

What problem this solves

This module exists to store variable-sized values in a memory-constrained system without relying on the standard heap.

It solves these problems:

- fixed set of logical keys, but variable-sized data per key
- need to provide memory externally
- need to safely read and update values across threads
- need to reclaim storage when a key is removed
- need predictable ownership of all stored data

>

> Instead of calling `malloc()` and `free()` from the general runtime allocator, the module manages a private heap inside caller-provided memory.

That gives the application explicit control over:

- where memory lives
- how large the pool is
- how many keys exist
- whether metadata and data live together or in different memory regions

High-Level Design

The module consists of two main parts.

Lookup table

Each key corresponds to one `kv_slot`.

A slot contains:

- a per-slot lock
- a pointer to the stored data block
- the size of that block
- a validity flag

If a slot is valid, the key currently has stored data.

If a slot is invalid, the key is empty.

Internal heap

Actual data bytes are stored inside a custom heap managed by the module.

This heap:

- lives in caller-provided memory
- uses a free-list allocator
- allocates variable-sized blocks
- coalesces adjacent free blocks when freeing

This means the slot table stores metadata only. The actual value bytes are stored elsewhere in the pool heap.

Memory model

The pool manages two logical memory regions:

- **lookup table memory**
- **data heap memory**

These can be provided in two ways:

Contiguous mode

One big memory block is given to `kv_pool_init()`. The module splits it into:

1. lookup table
2. data heap

Fragmented mode

Separate memory regions are given to `kv_pool_init_fragmented()`. This allows the lookup table and data heap to live in different memory banks.

That can be useful when, for example:

- metadata should live in fast SRAM
- bulk value storage should live in larger but slower RAM

Public API overview

The public API consists of:

- data structures:
  - `kv_slot`
  - `kv_header`
  - `kv_pool`
- macros:
  - `KV_ALIGNMENT`
  - `ALIGN(x)`
  - `MINIMUM_BLOCK_SIZE`
  - `LOOKUP_TABLE_SIZE(x)`
- initialization:
  - `kv_pool_init_fragmented()`
  - `kv_pool_init()`

- key operations:
  - `kv_pool_get()`
  - `kv_pool_write()`
  - `kv_pool_insert()`
  - `kv_pool_remove()`
  - `kv_pool_is_index_valid()`
- allocator functions:
  - `kv_pool_allocate()`
  - `kv_pool_free()`

The last two are currently exposed in the header, although they behave more like internal allocator primitives than ordinary user-facing API.

Data structures

`kv_slot`

```
typedef struct {
  atomic_flag slot_lock;
  void *data_ptr;
  size_t data_size;
  bool is_valid;
} kv_slot;
```

Purpose

Represents the metadata for one key.

Fields

`slot_lock`

A spinlock protecting this slot’s metadata and associated data access.

Used to synchronize operations on a single key.

`data_ptr`

Pointer to the allocated data block in the internal heap.

The caller must not free or reallocate this pointer directly.

`data_size`

Size in bytes of the stored value.

`is_valid`

Whether this slot currently contains valid data.

If false, the key is considered empty.

Important note

> The key itself is not stored in the slot. The key is simply the slot's index in the lookup table.

`kv_header`

```
typedef struct kv_header {
  size_t size;
  union {
    struct kv_header *next_free;
    char data[1];
  } as;
} kv_header;
```

Purpose

Header for blocks in the internal heap.

Role

When a block is free, `as.next_free` links it into the free list.

When a block is allocated, `as.data` is the start of the user-visible payload.

Meaning of `size`

`size` is the total size of the block, including the header and payload region.

This is important for:

- pointer arithmetic
- block splitting
- coalescing adjacent free blocks

---

kv_pool

```
typedef struct {
  void *pool_start;
  size_t pool_size;

  atomic_flag heap_lock;
  void (*delay)(void);
  kv_header *free_list_head;

  size_t max_keys;
  kv_slot *lookup_table;
} kv_pool;
```

Purpose

Represents the entire key-value pool.

Fields

`pool_start`

Start address of the data heap region.

`pool_size`

Size of the data heap region in bytes.

`heap_lock`

Spinlock protecting heap allocator operations.

`delay`

Callback invoked while waiting for a lock.

This is used during busy-waiting to avoid a pure tight spin.

`free_list_head`

Head of the free-list allocator.

`max_keys`

Maximum number of keys supported by this pool.

Valid keys are:

```
0 <= key < max_keys
```

`lookup_table`

Pointer to the slot array.

Alignment and size macros

`KV_ALIGNMENT`

```
#define KV_ALIGNMENT 16
```

All allocations are aligned to this boundary.

This affects:

- payload placement
- allocator block sizes
- pointer arithmetic

If this value changes, allocator behavior changes with it.

`ALIGN(x)`

```
#define ALIGN(x) (((x) + (KV_ALIGNMENT - 1)) & ~(KV_ALIGNMENT - 1))
```

Rounds a size up to the next `KV_ALIGNMENT` boundary.

Used by the allocator.

`MINIMUM_BLOCK_SIZE`

```
#define MINIMUM_BLOCK_SIZE sizeof(kv_header) + 1
```

Minimum block size allowed in the heap.

Used to decide whether a free block can be split.

The allocator will not create a leftover free fragment smaller than this.

`LOOKUP_TABLE_SIZE(x)`

```
#define LOOKUP_TABLE_SIZE(x) (sizeof(kv_slot) * (x))
```

Returns the number of bytes required for a lookup table supporting `x` keys.

Used during initialization and memory size validation.

Initialization APIs

`kv_pool_init_fragmented()`

```
result_t kv_pool_init_fragmented(void *lookup_table,
                                 size_t lookup_table_size,
                                 size_t max_keys,
                                 void *pool_data,
                                 size_t pool_size,
                                 kv_pool *pool,
                                 void (*delay)(void));
```

Purpose

Initializes a pool from two separate memory regions:

- one for slot metadata
- one for heap storage

Parameters

`lookup_table`

Memory for the `kv_slot` array.

`lookup_table_size`

Size of the lookup table memory region in bytes.

`max_keys`

Maximum number of keys.

`pool_data`

Memory for the heap region.

`pool_size`

Size of the heap region in bytes.

`pool`

Output pool structure to initialize.

`delay`

Function called while waiting for spinlocks.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if required pointers are null or `max_keys == 0`
- `RESULT_ERR_BUFFER_TOO_SMALL` if:
  - heap is too small
  - lookup table region is too small

What it does

1. validates inputs
2. clears both memory regions with `memset`
3. sets up the lookup table
4. sets up the free-list heap as one large free block
5. clears locks
6. marks all slots invalid

Important lifetime rule

> The memory regions provided to this function must remain valid for the entire lifetime of the pool.

The module does not copy them.

`kv_pool_init()`

```
result_t kv_pool_init(void *data,
                      size_t data_size,
                      size_t max_keys,
                      kv_pool *pool,
                      void (*delay)(void));
```

Purpose

Initializes a pool from one contiguous memory block.

Parameters

`data`

Start of the full memory region.

`data_size`

Total size of the region.

`max_keys`

Number of keys.

`pool`

Output pool structure.

`delay`

Lock wait callback.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_BUFFER_TOO_SMALL` if total memory is insufficient
- any error returned by `kv_pool_init_fragmented()`

What it does

It computes:

- lookup table size = `LOOKUP_TABLE_SIZE(max_keys)`
- heap start = `data + LOOKUP_TABLE_SIZE(max_keys)`
- heap size = remaining bytes

Then it delegates to `kv_pool_init_fragmented()`.

When to use it

Use this function when you want simple setup from one static buffer.

Use `kv_pool_init_fragmented()` when memory placement matters.

Key operations

`kv_pool_get()`

```
result_t kv_pool_get(kv_pool *pool, int key, void *buffer, size_t *buffer_size);
```

Purpose

Copies the value for a key into a caller-provided buffer.

Parameters

`pool`

Initialized pool.

`key`

Key index to read.

`buffer`

Destination buffer for copied data.

`buffer_size`

Input/output parameter.

- on input: capacity of `buffer`
- on output:
  - actual copied size on success
  - required size if buffer is too small

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL` or `buffer_size == NULL`
- `RESULT_ERR_NOT_FOUND` if key is out of range or not valid
- `RESULT_ERR_BUFFER_TOO_SMALL` if destination buffer is too small

Behavior

The function:

1. validates inputs
2. checks key bounds
3. locks the slot
4. verifies the slot is valid
5. checks whether the provided buffer is large enough
6. copies the stored data with `memcpy`
7. unlocks the slot

Important note

The function does **not** require `buffer` to be non-null when the buffer is too small path is taken first, but in practice a null `buffer` with a large enough `buffer_size` would lead to invalid `memcpy`. So callers should always provide a valid buffer unless they intentionally use this as a size query pattern and know what they are doing.

> The implementation does not explicitly validate `buffer != NULL`. Why? No clue.

`kv_pool_write()`

```
result_t kv_pool_write(kv_pool *pool, int key, void *buffer, size_t buffer_size);
```

Purpose

Overwrites the existing value for a valid key without reallocating memory.

Parameters

`pool`

Initialized pool.

`key`

Key index to overwrite.

`buffer`

Source data to copy from.

`buffer_size`

Size of new data.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL`, `buffer == NULL`, or size mismatches existing allocation
- `RESULT_ERR_NOT_FOUND` if key is invalid or not currently in use

Behavior

The function:

1. validates arguments
2. checks that the key refers to a valid existing slot
3. locks the slot
4. checks that `buffer_size` exactly matches the stored size
5. copies new bytes over existing allocation
6. unlocks the slot

Important constraint

> This function does **not** resize.

The size must exactly match the existing allocation.

If the caller wants to store a different-sized value, they must:

- remove the key
- insert again with the new size

or implement a resize API in the future.

`kv_pool_insert()`

```
result_t kv_pool_insert(kv_pool *pool, int key, void *data, size_t data_size);
```

Purpose

Allocates heap space and stores new data for a key.

Parameters

`pool`

Initialized pool.

`key`

Key index to populate.

`data`

Source bytes to copy into pool storage.

`data_size`

Size of the data to store.

Returns

Actual implementation returns:

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if:
  - `pool == NULL`
  - `data == NULL`
  - key is out of range
- `RESULT_ERR_NO_MEM` if allocation fails

Inserting on an already allocated slot

> If `kv_pool_insert()` is called on a slot that is already valid, the old allocation is orphaned and leaked from the pool heap.

So the safe usage rule today is:

- only call `kv_pool_insert()` on an empty key
- remove existing keys first before reinserting

The implementation does not enforce that, but callers must.

Internal behavior

The function:

1. validates inputs
2. locks the slot
3. marks the slot invalid and clears metadata
4. allocates a heap block
5. copies data into the new block
6. marks the slot valid
7. unlocks and returns

`kv_pool_remove()`

```
result_t kv_pool_remove(kv_pool *pool, int key);
```

Purpose

Removes a key and frees its allocated data block.

Parameters

`pool`

Initialized pool.

`key`

Key index to remove.

Returns

Actual implementation returns:

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL`
- `RESULT_ERR_NOT_FOUND` if key is out of bounds or not valid

Behavior

The function:

1. validates the pool pointer
2. verifies the key is valid with `kv_pool_is_index_valid()`
3. calls `kv_pool_free()` on the stored pointer

`kv_pool_is_index_valid()`

```
result_t kv_pool_is_index_valid(kv_pool *pool, int key);
```

Purpose

Checks whether a key currently contains valid data.

Parameters

`pool`

Initialized pool.

`key`

Key index to check.

Returns

- `RESULT_OK` if key is in range and valid
- `RESULT_ERR_INVALID_ARG` if pool is null or key is out of range
- `RESULT_ERR_NOT_FOUND` if slot is currently invalid

Behavior

The function:

1. validates arguments
2. locks the slot
3. checks `is_valid`
4. unlocks and returns a snapshot result

Important concurrency note

> This is only a snapshot.

> A slot that is valid at the time of the check may become invalid immediately afterward.

So callers must not do:

1. `kv_pool_is_index_valid()`
2. assume later access is now guaranteed safe forever

They must still handle failure from the actual operation.

Allocator APIs

These are declared in the header and can be called externally, but they behave like internal heap primitives.

Using them directly requires understanding the allocator and slot ownership rules.

`kv_pool_allocate()`

```
result_t kv_pool_allocate(kv_pool *pool, size_t size, void **out_ptr);
```

Purpose

Allocates a block from the pool heap.

Parameters

`pool`

Initialized pool.

`size`

Payload size requested.

`out_ptr`

Output pointer for the allocated payload address.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` for null pointers or zero size
- `RESULT_ERR_NO_MEM` if no free block can satisfy the request

Allocation strategy

The allocator uses **first fit** on the free list.

It scans from the head until it finds the first block large enough.

Block sizing

The allocator computes:

- header offset to payload
- total required size including header
- alignment rounding
- minimum block size enforcement

Splitting

If the selected block is larger than needed and the remainder is big enough, it splits the block and leaves the remainder on the free list.

Otherwise it consumes the whole block.

`kv_pool_free()`

```
result_t kv_pool_free(kv_pool *pool, void *ptr);
```

Purpose

Returns a previously allocated block to the pool heap.

Parameters

`pool`

Initialized pool.

`ptr`

Pointer previously returned by `kv_pool_allocate()` or stored in a slot.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if arguments are null

Behavior

The function:

1. derives the block header from the payload pointer
2. inserts the block back into the sorted free list
3. coalesces with adjacent free neighbors when possible
4. scans the slot table for a matching `data_ptr`
5. if found, clears that slot's metadata

Important consequence

`kv_pool_free()` does not merely free heap memory. It also tries to invalidate any slot pointing at that memory.

That means allocator state and key-value metadata are coupled.

Safe usage implication

External callers should not casually use `kv_pool_allocate()` and `kv_pool_free()` unless they understand this coupling.

If you free a pointer that a slot still references, the function will clear that slot.

> If you allocate memory manually and never attach it to a slot, the allocator still works, but you are now using the pool partly as a raw allocator and partly as a key-value store, which increases maintenance complexity.

Role of `delay()`

> The caller supplies the delay function during pool initialization.

This allows board or environment-specific waiting behavior, such as:

- yielding
- sleeping
- short pause loop
- RTOS task delay
- platform-specific backoff

The pool does not define how delay behaves. That is the caller's responsibility.

Example usage

Contiguous initialization

```
static uint8_t kv_memory[2048];
static kv_pool pool;

static void pool_delay(void) {
  /* platform-specific wait/yield */
}

result_t app_kv_init(void) {
  return kv_pool_init(kv_memory, sizeof(kv_memory), 16, &pool, pool_delay);
}
```

This creates a pool with:

- 16 keys
- lookup table at the start of `kv_memory`
- heap in the remaining bytes

Insert value

```
uint32_t value = 1234;
result_t res = kv_pool_insert(&pool, 3, &value, sizeof(value));
```

This stores 4 bytes at key `3`.

Read value

```
uint32_t value = 0;
size_t size = sizeof(value);

result_t res = kv_pool_get(&pool, 3, &value, &size);
```

On success:

- `res == RESULT_OK`
- `value` contains the stored bytes
- `size == sizeof(uint32_t)`

Handle too-small buffer

```
uint8_t small_buf[4];
size_t size = sizeof(small_buf);

result_t res = kv_pool_get(&pool, key, small_buf, &size);
if (res == RESULT_ERR_BUFFER_TOO_SMALL) {
  /* size now contains required size */
}
```

This is the intended size negotiation pattern.

Overwrite same-sized value

```
uint32_t new_value = 5678;
result_t res = kv_pool_write(&pool, 3, &new_value, sizeof(new_value));
```

This succeeds only if key `3` already exists and has exactly 4 bytes allocated.

Remove key

```
result_t res = kv_pool_remove(&pool, 3);
```

This frees the associated heap block and invalidates the slot.

Backend and platform independence

> The API is largely independent of where memory comes from and how waiting is implemented.

The caller provides:

- raw memory regions
- a delay function
- the `kv_pool` object itself

That means different boards or environments can use the same API with different backing strategies, for example:

- contiguous static RAM region on one board
- split metadata/data regions across different RAM banks on another
- RTOS yield in the delay callback on one target
- busy-wait pause or test hook in host-side simulation

> The implementation is therefore **memory-placement agnostic** and **wait-strategy agnostic**, even though the current source file provides one specific allocator and lock implementation.

This is useful for portability, provided each target respects the same concurrency and memory lifetime contract.

Recommended usage rules for current code

Given the implementation as it exists today, these rules are the safest:

1. Initialize once before concurrent use.
2. Provide memory that remains valid for the full pool lifetime.
3. Use `insert()` only on empty keys.
4. Use `write()` only when new data size exactly matches old size.
5. Do not rely on `is_index_valid()` as a guarantee for later access.
6. Treat direct allocator calls as advanced/internal use.
7. Do not assume allocator concurrency is fully correct (there is definitely a bug or two in there).
8. Always pass a valid destination buffer to `get()` when copying data.

Revision #4
Created 2026-04-14 15:42:39 UTC by Nikolaos Diamantopoulos
Updated 2026-04-15 08:19:02 UTC by Nikolaos Diamantopoulos