## 1 | Introduction

For this project I took a deep dive into Thumb-2 assembly on an ARM Cortex-M4, re-creating half a dozen bread-and-butter C library calls—`malloc`, `free`, `bzero`, `strncpy`, `signal`, and `alarm`.
At the centre of it all sits a kernel-mode **buddy allocator**. User code requests memory in un-privileged mode; a tiny system-call layer (SVC) flips the CPU into supervisor mode, where the allocator does the heavy lifting. Building this from scratch forced me to wrestle with system-call plumbing, interrupt handling, stack juggling, and dynamic memory, all in bare-metal assembly—equal parts brutal and fun.

## 2 | System overview

### Memory map

| Region | Address range | Size | Purpose |
|---|---|---|---|
| **Heap** | `0x2000 1000 – 0x2000 4FFF` | 16 KB | User allocations |
| **MCB** | `0x2000 6800 – 0x2000 6BFF` | 1 KB | 512 MCB entries (2 B each) |
| **Stacks** | `0x2000 5800 – 0x2000 5FFF` (handler) / `0x2000 5000 – 0x2000 57FF` (thread) | 2 KB each | Separate stacks for kernel & user |
| **Sys-call table** | `0x2000 7B00 – 0x2000 7B7F` | 128 B | Function pointers for SVC dispatcher |
| **Timer / signal vars** | `0x2000 7B80 – 0x2000 7BFF` | 128 B | Countdown + user handler ptr |

### What happens at run-time

1. **Reset** – the start-up code sets both stacks, zeroes the heap MCB, builds the system-call table and arms the timer.
2. **User code** (driver in C) calls the assembly wrappers in `stdlib.s`.
3. If a call needs privilege (`malloc`, `free`, `signal`, `alarm`) it executes `SVC`.
4. **`SVC_Handler`** looks up the right kernel routine (in `heap.s` or `timer.s`) and jumps there.
5. The **buddy allocator** manages every byte of heap—user code never touches raw memory directly.

## 3 | Implementation details

### Buddy allocator in a nutshell

- **MCB layout** – 512 entries × 2 bytes
    - Bits 15-4 = block size (bytes)
    - Bit 0 = 1 if in-use, 0 if free
- **Heap geometry** – 16 KB split in powers-of-two down to 32 B.

| Phase | What happens |
|---|---|
| **Alloc (`_kalloc`/`_ralloc`)** | Round size ↑ to next power-of-2 → recursively split blocks until the fit is exact → tag header as *in-use*. |
| **Free (`_kfree`/`_rfree`)** | Clear *in-use* flag → if buddy is also free, merge upward (coalesce) → repeat until no more merges. |
| **Init (`_kinit`)** | MCB[0] = `0x4000` (16 KB free); rest zero. |

### System-call plumbing

- Each privileged wrapper sets **R7 = call ID** then triggers **`SVC #0`**.
- The dispatch table holds addresses of `_timer_start`, `_signal_handler`, `_kalloc`, `_kfree`.
- Everything inside the handler runs with MSP in privileged mode; user code stays on PSP.

### Timer & signal

- `alarm(seconds)` stores the value at `SECOND_LEFT`, reloads SysTick, and enables the interrupt.
- `signal(SIGALRM, fn)` just drops `fn` into `USR_HANDLER`.
- Every SysTick interrupt decrements the counter; when it hits 0 the kernel disables the timer and `BLX`-calls the user handler.

## 4 | Testing & results

Using **`driver_keil.c`**:

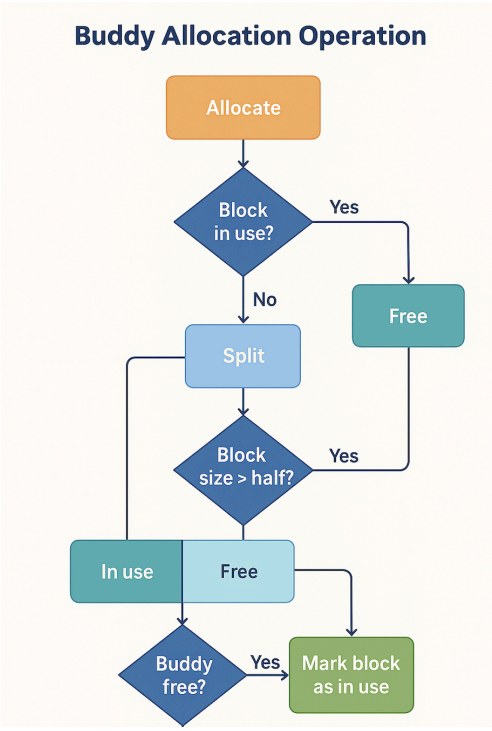
- **String ops** – `bzero`/`strncpy` work totally in user space.
- **Allocator torture-test** – allocating and freeing 0.5 KB – 8 KB chunks proves the split/merge logic (watching the MCB bytes change in the Keil debugger is oddly satisfying).
- **Alarm / signal** – two back-to-back alarms flip the `alarmed` variable from 1 → 2 → 3 exactly as expected.

## 5 | Challenges & lessons learned

- **Recursive assembly** – writing a reentrant allocator without trashing registers takes real discipline.

- **MCB bookkeeping** – one wrong off-by-one in the merge logic and the heap collapses.
- **APCS calling convention** – respecting *every* callee-saved register was non-negotiable; the debugger caught me each time I got lazy.
- **SVC debugging** – early on the dispatcher happily jumped into the weeds when R7 got clobbered, teaching me the value of a good prologue.

## 6 | Buddy allocator diagram




## 7 | Conclusion

Building a mini-runtime from bare metal was a great way to peel back the OS curtain. I now have a much clearer picture of how memory managers, system calls, and interrupt controllers cooperate on ARM hardware—and a brand-new respect for anyone who writes production firmware in assembly!