THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
EUROPE

Demystifying memory: A Practical Tutorial on Managing & Optimizing memory in Zephyr

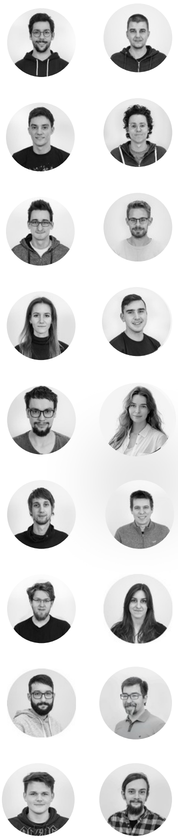
Marko Sagadin, IRNAS

About speaker

- Electrical (electronics) engineer by education
- Embedded systems engineer by profession
- PCB designer / assembly line engineer -> firmware engineer
- Consumer electronics, medical devices, IoT systems, TinyML...
- Secondary areas: tooling, developer environment, CI, processes, etc.

Who is IRNAS?

- Cross-disciplinary team of 18 experts in connected product design and production
- 10 years of development of connected hardware products
- Full in-house infrastructure for innovating and manufacturing new products within days



Motivation

- An IoT device sending data via Wi-Fi
- PCB would contain nRF53 SOC, nRF7002 Wi-Fi companion IC and external flash chip.
- Only UI are an eInk display and two buttons
- On cloud side the Azure's IoT Hub service (application protocol built on top of MQTT)
  - Data is sent in strings, representing JSON objects
- Provisioning of Wi-Fi credentials will use BLE
- LVGL is used to render graphics
- External flash chip used to store assets and second image slot
- Code relocation and execute in place (XIP) were expected

Motivation

- Due to the number of features the higher memory usage was expected
- But how much?
- First good approximation was analysis of the *`nrf/samples/lib/azure_iot_hub`* sample.

- 61 KB FLASH and 69 KB RAM available
- Second MCUboot partition was already on external flash
- LTO was already enabled

```
MCUboot:
Memory region         Used Size  Region Size  %age Used
           FLASH:       46288 B        64 KB     70.63%
             RAM:       21328 B        48 KB     43.39%
        IDT_LIST:          0 GB        32 KB      0.00%

TF-M:
Memory region         Used Size  Region Size  %age Used
           FLASH:      129976 B     130560 B     99.55%
             RAM:       38996 B        48 KB     79.34%

Azure IoT Hub sample:
Memory region         Used Size  Region Size  %age Used
           FLASH:      702060 B       768 KB     89.27%
             RAM:      338376 B       400 KB     82.61%
        IDT_LIST:          0 GB        32 KB      0.00%
```

Flash and RAM

- Memory regions, not specific memory device technologies!
- FLASH
  - Read-only memory
  - Non-volatile — retains data after power loss
  - Stores program code (.text), read-only data (.rodata), and initial values for initialized data (.data)
- RAM
  - Read-write memory
  - Volatile — loses contents on power loss
  - Holds initialized data (.data), zero-initialized data (.bss), dynamic memory (heap), and call stacks (stack)

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* Copied from FLASH to RAM at startup */
/* (Initialized global variable stored in .data section) */
uint8_t data_section = 42;

/* Stored in FLASH (read-only memory) */
/* (Constant global variable in .rodata section) */
const uint8_t rodata_section = 42;

/* Zero-initialized global variable stored in RAM */
/* (Uninitialized globals go to .bss section) */
uint8_t bss_section;

int main(int argc, char *argv[])
{
    /* Allocated on stack at runtime */
    /* (Local automatic variable inside main's stack frame) */
    uint8_t stack_var = 42;

    /* Allocated dynamically on heap at runtime */
    uint8_t *heap_section = malloc(sizeof(uint8_t));

    /* String literal stored in FLASH (.rodata section) */
    printf("Hello World\n");

    return 0;
}
```

Memory managment in Zephyr

- `k_heap_*` API
  - Thread safe heap allocator
  - Timeout arguments
  - Built on top of `sys_heap_*`
- `k_mem_slab_* API`
  - Fixed-sized block allocator
- System Heap
  - heap allocator with libc-like API (`k_malloc`, `k_free`, etc.)
  - Wrapper around k_heap
  - It's size is set with `CONFIG_HEAP_MEM_POOL_SIZE`

```c
/* zephyr/kernel/mempool.c, line 62 */
K_HEAP_DEFINE(_system_heap, K_HEAP_MEM_POOL_SIZE);

/* zephyr/subsys/net/ip/tcp.c, line 76 */
K_MEM_SLAB_DEFINE_STATIC(tcp_conns_slab, sizeof(struct tcp),
                         CONFIG_NET_MAX_CONTEXTS, 4);
```

Memory managment in Zephyr

- Both macros are expanded to an array and a structure that references it.
- Prefixes: `kheap__*`, `_k_mem_slab_buf_*`
- The size of arrays is determined at compile time, and the reserved space **is included in the RAM usage report**.
- There are no “hidden” consumers, that are unaccounted for.
- Caveat: where libc’s malloc allocates from depends on the choice of libc.

```c
/* zephyr/kernel/mempool.c, line 62 */
char kheap__system_heap[K_HEAP_MEM_POOL_SIZE];

struct k_heap _system_heap = {
    .heap = {
        .init_mem = kheap__system_heap,
        .init_bytes = K_HEAP_MEM_POOL_SIZE,
    },
};

/* zephyr/subsys/net/ip/tcp.c, line 76 */
size_t size = CONFIG_NET_MAX_CONTEXTS * sizeof(struct tcp);
static char _k_mem_slab_buf_tcp_conns_slab[size];

static struct k_mem_slab tcp_conns_slab = {
    .buffer = _k_mem_slab_buf_tcp_conns_slab,
    /* Other struct members... */
};
```

Very simplified code after pre-processor macro expansion

Useful tools and approaches for memory analysis

ROM and RAM report

- `west build -t rom_report`
- `west build -t ram_report`
- Sysbuild: add `-d build/<app dir>`

```
├── wpa_supp_event_info          136    0.03%  0x000b0b98 rodata
├── wpa_supp_ops.lto_priv.0      148    0.03%  0x2000ce4c datas
├── wpa_supp_ready_sem            24    0.00%  0x2000d96c k_sem_area
├── wpa_supp_status_work          48    0.01%  0x2000c4c0 datas
├── wpa_supplicant_mutex          20    0.00%  0x2000d87c k_mutex_area
├── wpas_api_ctrl                 56    0.01%  0x20012c8c bss
├── z_arm_tls_ptr                  4    0.00%  0x20012e74 bss
├── z_idle_stacks                320    0.06%  0x2003a810 noinit
├── z_idle_threads               224    0.04%  0x2000de58 bss
├── z_interrupt_stacks          2048    0.38%  0x20036038 noinit
├── z_main_stack                6144    1.15%  0x2003a950 noinit
├── z_main_thread                224    0.04%  0x20010058 bss
├── z_malloc_heap                 12    0.00%  0x20012e54 bss
├── z_malloc_heap_mutex           20    0.00%  0x2000cb30 datas
├── z_sys_post_kernel              1    0.00%  0x20015ade bss
├── z_thread_monitor_lock          4    0.00%  0x2001256c bss
├── zep_wifi_bh_q                256    0.05%  0x20012438 bss
├── zep_wifi_intr_q              256    0.05%  0x20012338 bss
├── zep_work_item               3200    0.60%  0x20013f58 bss
```

Binutils

- `nm` and `addr2line`
- They work most of the time

```
# Usecase: find type and location of a symbol by its name
arm-zephyr-eabi-nm -l build/<app>/zephyr/zephyr.elf | grep <symbol>

# Usecase: find location of a symbol by its address
arm-zephyr-eabi-addr2line -e build/<app>/zephyr/zephyr.elf <address>
```

Puncover

- Creates report in Web GUI
  - Code size
  - Variable size
  - Worst case statck analysis
- `west build -t puncover`
  - If using Sysbuild add `-d build/<app dir>`
  - `CONFIG_STACK_USAGE`
- What do the headers mean?
  - Remarks:
    - `x-module` – function is called from another file
    - `float` – function eventually calls a fp function
  - Code: `.text` section
  - Static: `.data`, `.rodata`, `.bss`

home / user / workdir / zephyr

Name	Remarks	Code	Static
arch		4,340	89
drivers		30,300	88,062
include		4,380	4
kernel		24,778	53,910
lib		18,946	3,408
modules		14,604	99,741
soc		594	68
subsys		118,402	58,296
∑ over all (8 folders, 0 files)		216,344	303,578

home / user / workdir / zephyr / kernel / mempool.c

Name	Remarks	Stack	Code	Static
k_aligned_alloc		8	108	
k_free	x-module	0	14	
k_malloc	x-module	0	8	
k_thread_system_pool_assign	x-module	0	12	
z_heap_aligned_alloc		32	104	
z_thread_aligned_alloc	x-module	16	48	
∑ 6 functions			294	
_system_heap				24
kheap__system_heap				40,144
∑ 2 variables				40,168
∑ over all (6 functions, 2 variables)			294	40,168

Thread analyzer

- Shows maximum stack usage for each thread at runtime
- Very versatile and easy to setup

```
CONFIG_THREAD_ANALYZER=y
CONFIG_THREAD_ANALYZER_USE_LOG=y
CONFIG_THREAD_ANALYZER_AUTO=y
CONFIG_THREAD_ANALYZER_AUTO_INTERVAL=5
CONFIG_THREAD_NAME=y
```

```
tcp_work                 : STACK: unused 864 usage 160 / 1024 (15 %); CPU: 0 %
                         : Total CPU cycles used: 0
shell_uart               : STACK: unused 5472 usage 672 / 6144 (10 %); CPU: 0 %
                         : Total CPU cycles used: 10426
sysworkq                 : STACK: unused 3864 usage 232 / 4096 (5 %); CPU: 0 %
                         : Total CPU cycles used: 36
nrf70_intr_wq            : STACK: unused 1448 usage 600 / 2048 (29 %); CPU: 0 %
                         : Total CPU cycles used: 510
nrf70_bh_wq              : STACK: unused 824 usage 1224 / 2048 (59 %); CPU: 0 %
                         : Total CPU cycles used: 90
logging                  : STACK: unused 1544 usage 504 / 2048 (24 %); CPU: 0 %
                         : Total CPU cycles used: 3790
idle                     : STACK: unused 248 usage 72 / 320 (22 %); CPU: 99 %
                         : Total CPU cycles used: 3265937
main                     : STACK: unused 4792 usage 1352 / 6144 (22 %); CPU: 0 %
                         : Total CPU cycles used: 1713
```

Heap and memory slab analysis

- System heap run time statistics
  - `CONFIG_SYS_HEAP_RUNTIME_STATS`
  - `sys_heap_runtime_stats_get`
  - Can also be used with `k_heap`
  - Shell command: `kernel heap`
- Memory slab's `info` member
  - `num_blocks, block_size, num_use, max_used`
  - `CONFIG_MEM_SLAB_TRACE_MAX_UTILIZATION`

```c
#include <zephyr/kernel.h>
#include <zephyr/sys/sys_heap.h>

extern struct k_heap heap;
extern struct k_mem_slab slab;

void print_usage(void)
{
    k_spinlock_key_t key;
    struct sys_memory_stats stats;
    struct k_mem_slab_info info;

    key = k_spin_lock(&heap->lock);
    sys_heap_runtime_stats_get(&heap->heap, &stats);
    k_spin_unlock(&heap->lock, key);

    key = k_spin_lock(&slab->lock);
    info = slab.info;
    k_spin_unlock(&slab->lock, key);

    printk("Heap stats\n"
        "\t%zu free, %zu allocated, %zu max allocated",
        stats.free_bytes,
        stats.allocated_bytes,
        stats.max_allocated_bytes);

    printk("Slab stats\n"
        "%u blocks, %zu block size, %u used, %u max used",
        info.num_blocks,
        info.block_size,
        info.num_used,
        info.max_used);
}
```

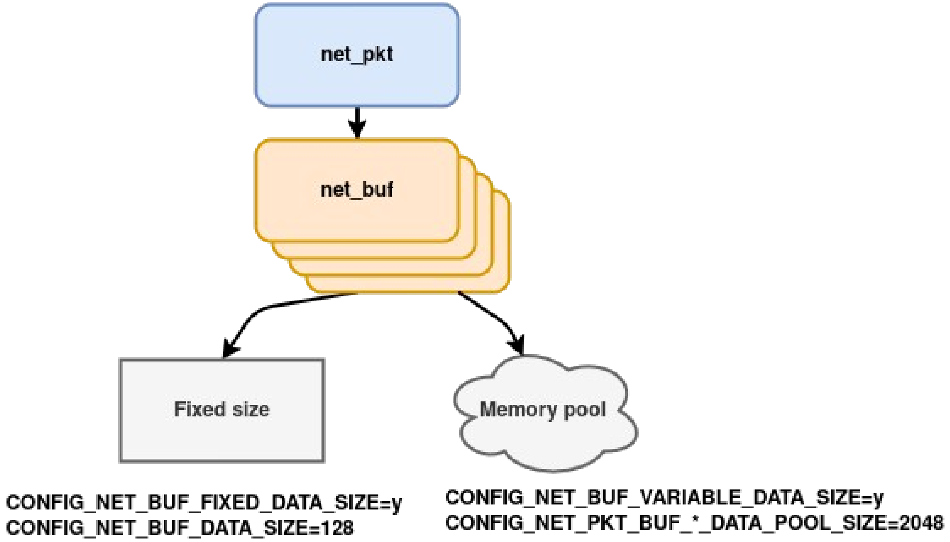
Mbed TLS heap analysis

- Allocates from a static buffer.
- `CONFIG_MBEDTLS_HEAP_SIZE` controls size.
- Uses it's own memory allocator with it's own usage tracking logic.
- Depending on the type of Zephyr application you will need to configure different options.

```
# Zephyr project
CONFIG_MBEDTLS_SHELL=y
CONFIG_MBEDTLS_DEBUG=y
CONFIG_MBEDTLS_DEBUG_C=y
CONFIG_MBEDTLS_MEMORY_DEBUG=y

# For Zephyr project with TF-M also add:
CONFIG_TFM_CMAKE_BUILD_TYPE_RELWITHDEBINFO=y
# or
CONFIG_DEBUG=y

# NCS project with TF-M
# Compile ${ZEPHYR_BASE}/modules/mbedtls/shell.c
# in nrf/subsys/nrf_security/src/zephyr/CMakeLists.txt

# Shell
uart:~$ mbedtls heap
Maximum (peak): 52340 bytes, 111 blocks
Current: 0 bytes, 0 blocks
```

Network stack and nRF70 Wi-Fi

- Many “knobs” to turn, all affect memory consumption and throughput in different ways.
- RX and TX path are separately configurable
- `net_pkt` holds one or more `net_bufs`
- Driver for nRF7002 uses variable size.
- Interesting heaps to watch:
  - `net_buf_mem_pool_rx_bufs`
  - `net_buf_mem_pool_tx_bufs`
  - `wifi_drv_ctrl_mem_pool`
  - `wifi_drv_dat_mem_pool`



Optimization strategy

- Locate big spenders with the static tools.
  - Determine symbols that enable or tune them.
  - Determine if you even need them.
  - Tackle them first.
- Many subsystem have constrained variants
- Decide what you can miss and what is mandatory to have.

- Define what are your device workflows and possible code paths.
- Determining correct sizes of threads, heaps and mem slabs:
- 1st estimation: Run your code through all possible code paths and then check report tools for maximum values.
- 2nd estimation: keep device running for several days.
- 3rd estimation: run several devices in the field and have them report back statistics.

Memory analysis demo

https://github.com/MarkoSagadin/demistifying-memory-project

Azure IoT Hub memory results

Test conditions:

- Sending 1kB of data (hard-coded string) 10 times every 20 sec,
- Received 1kB data sporadically, CJSON parses it
- Disconnection (sudden removal of the access point)
- No FOTA, no direct methods, little CJSON use.

Build variant	**Used Flash**	**Max Flash**	**Free Flash**	**Used RAM**	**Max RAM**	**Free RAM**
Original	707 KB	768 KB	61 KB	331 KB	400 KB	69 KB
Optimized 1	684 KB	768 KB	84 KB	226 KB	400 KB	174 KB
Optimized 2	668 KB	768 KB	100 KB	226 KB	408 KB	182 KB

What I learned?

- How to find things in the Zephyr.
- General rule: FLASH usage is “hard” feature bound , but RAM is “soft” feature bound (more tunable).
- External flash and execute in place (XIP) are essentially required for nRF7002 (STA applications). Nordic documents well how this can be done.
- If you can, keep heaps separate. Although they can waste RAM space, they make it mach easier to reason about the system.

IRNAS's resources

- Found at: https://github.com/irnas
- To speed up our own development we made many different project templates, reusable scripts, documentation. They are public and free to use. Search for "`irnas-`" prefix.
- Contents:
  - Docker images
  - CI infrastructure
  - Zephyr project template
  - Custom East tool
  - Project and Development guidelines