Memory Footprint Reduction Techniques (with i.MX RT1050 EVKB & Airbus-style Avionics Scenarios)

Why memory footprint matters in avionics

In airborne embedded systems, the pressure to reduce code (Flash) and data (RAM) footprint is not just about cost: it directly affects reliability, safety certification, timing determinism, and long-term maintainability. Smaller binaries typically mean tighter control of dependencies, fewer hidden allocations, and simpler worst-case analysis—key inputs to **DO-178C** (Software Considerations in Airborne Systems and Equipment Certification) compliance, especially at higher **DAL** (Design Assurance Level). Memory optimization also improves boot time, reduces pressure on instruction/data caches, and can materially shrink the attack surface for security assessments.

From an operational standpoint, Airbus LRUs (Line Replaceable Units) often run multiple partitions and functions on shared SoCs. Conservatively budgeted RAM/Flash keeps integration risks manageable, avoids re-architecting under schedule pressure, and makes incremental feature certification easier.

Essential definitions you will use precisely

Before diving into techniques, we synchronize on terminology as used in the **NXP i.MX RT1050** environment and typical GCC/FreeRTOS toolchains.

- **Footprint**: The total non-volatile (Flash/XIP) and volatile (RAM: stack/heap/BSS/data) memory used by a program.
- **Text**: The executable machine code placed in Flash or **XIP** (Execute-In-Place) from external QSPI NOR, and sometimes in **ITCM** (Instruction Tightly Coupled Memory) for hot paths.
- **ROData**: Read-only constant data stored in Flash/XIP.
- **Data**: Initialized globals that occupy RAM at run time; their initial values are copied from Flash at boot.
- **BSS**: Zero-initialized globals that occupy RAM but no Flash storage for initializers.
- **Heap**: Dynamically allocated memory via `malloc` / `free` or RTOS allocators.
- **Stack**: Per-thread (task) automatic storage growing and shrinking with function calls.
- **TCM**: **Tightly Coupled Memory**. On M7 cores this includes **ITCM** (instruction) and **DTCM** (data). Very low-latency but capacity-limited.
- **OCRAM**: On-Chip RAM blocks (slower than TCM, faster than external SDRAM; cacheable). In the SDK you will often see memory names like `SRAM_ITC`, `SRAM_DTC`, and `SRAM_OC` in linker fragments.
- **MPU**: **Memory Protection Unit**, used on Cortex-M for region-based protections; helpful to bound stacks/heaps.
- **DMA/EDMA**: **Direct Memory Access** / **Enhanced DMA** engines that read/write buffers without CPU intervention. Buffers used by DMA must obey cache-coherency and alignment rules.

The NXP MCUXpresso SDK for EVKB-IMXRT1050 provides convenience macros to steer placement:

- `AT_NONCACHEABLE_SECTION(...)` and `AT_NONCACHEABLE_SECTION_ALIGN(var, align)` to place DMA-visible buffers into non-cacheable RAM regions.
- `AT_QUICKACCESS_SECTION_CODE(func)` / `AT_QUICKACCESS_SECTION_DATA(var)` to place hot code/data into quick-access regions (e.g., ITCM/DTCM) as defined by the project linker script.

These are available via `#include <fsl_common_arm.h>` in the SDK device drivers.

A quick mental model of i.MX RT1050 memory map (what matters for footprint)

On the EVKB-IMXRT1050, typical applications execute from external QSPI Flash via **FlexSPI XIP** while using on-chip RAM (OCRAM + TCM) for data and hot code. In MCUXpresso projects, you will find linker fragments that name regions such as `SRAM_ITC`, `SRAM_DTC`, and `SRAM_OC`. You do not need to memorize absolute addresses to reduce footprint; you *do* need to make sure that rarely used code lives in XIP Flash, DMA buffers live in non-cacheable OCRAM, and a tiny subset of latency-critical functions (only if justified) lives in ITCM.

For avionics, we strongly prefer **static allocation** with compile-time sizes, explicit section placement for critical paths, and a constrained set of toolchain switches that guarantee dead-code elimination and compact libraries.

Build-time techniques that almost always pay off

Effective footprint reduction starts with the toolchain. The following settings are safe defaults for production on GCC/arm-none-eabi and are compatible with the i.MX RT1050 SDK projects (armgcc or CMake).

1) Optimize *for size* and enable section-level GC

Use these compiler and linker flags in *Release* builds:

```
# CFLAGS
-Os -ffunction-sections -fdata-sections -fno-exceptions -fno-unwind-tables -fno-asynchronous-unwind-tables
-fno-builtin -fstrict-aliasing -fno-common

# LDFLAGS
-Wl,--gc-sections -Wl,--cref
```

- `-Os` prefers smaller code than `-O2` with minimal speed tradeoffs for M7.

- `-ffunction-sections -fdata-sections` emits each function/object in its own section so the linker can discard unreferenced ones.
- `--gc-sections` removes unused code/data pulled in by libraries.
- If you use C++, add `-fno-rtti -fno-exceptions`; but most avionics code here is C.

2) Use `newlib-nano` and avoid floating-point `printf`

Link against the smaller C library:

```
-specs=nano.specs
```

Only pull float formatting if truly required; otherwise the default will avoid huge formatters. If you *must* print floats, link explicitly to opt-in:

```
-u _printf_float
```

3) Consider Link-Time Optimization (LTO) for whole-program shrink

`-flto` lets the linker inline/truncate across compilation units, often saving 5–15% Flash. Validate that your build system and debug settings handle LTO correctly, then record the before/after map files for certification artifacts.

4) Disable or down-level the SDK debug console in production

The NXP debug console pulls in formatting code. In production builds set:

```
#define SDK_DEBUGCONSOLE DEBUGCONSOLE_DISABLE
```

and gate all `PRINTF()` calls with a compile-time macro, or replace with the *lite* console if minimal logs are still required. This one change can save tens of kilobytes.

5) Trim libraries and middleware aggressively

Start from the SDK example closest to what you need and delete entire middleware stacks you do not use (e.g., FATFS/USB/BT). Section GC helps but cannot remove code referenced by init paths or weak symbols—removing the module from the build is more effective.

Code-level techniques that reduce RAM and Flash

Compiler flags are only half the story. The bigger wins usually come from how data and control flow are structured.

Make all large read-only tables truly `const`

If a lookup table never changes, declare it `const`. That moves it into RODatain XIP Flash and removes its RAM copy.

```
/* Bad: ends up in .data (occupies RAM + Flash init) */
uint16_t crc_table[256] = { /* ... */ };

/* Good: ends up in .rodata (Flash only) */
const uint16_t crc_table[256] = { /* ... */ };
```

For unusually large tables used sparsely, consider compressing them (e.g., run-length encoding) or generating them at boot only if you have a boot-time budget and the RAM to hold the decompressed form.

Replace dynamic allocation with fixed pools and ring buffers

In safety-critical code avoid long-lived `malloc`. Pre-allocate pools with compile-time sizes and expose deterministic `alloc/free` functions. This not only reduces fragmentation risk and peak heap demand but also shrinks code because you can delete general-purpose allocators.

```
typedef struct {
    uint32_t word;   /* ARINC 429 32-bit word */
    uint8_t  ssm;    /* Sign/Status Matrix */
    uint8_t  parity; /* stored for diagnostics */
} arinc429_msg_t;

#define MSG_POOL_CAP 64
static arinc429_msg_t g_msg_pool[MSG_POOL_CAP];
static uint16_t g_free_ix_head, g_free_ix_tail; /* 16-bit indexes are enough */
/* …a tiny free-list or bitmap; no heap involved… */
```

Use the smallest sufficient integer types and pack consciously

Storing ARINC 429 label (8 bits), SDI (2 bits), SSM (2 bits), and parity (1 bit) inside a 32-bit word is standard. Where space matters, pack multiple flags into a single byte. Prefer manual masks/shifts over C bit-fields for portability and code-size predictability.

```
/* Pack label (8), sdi (2), ssm (2), rate_id (4) in one byte pair */
typedef struct {
    uint8_t label;   /* 8 bits */
    uint8_t meta;    /* [7:6] SSM, [5:4] SDI, [3:0] rate id */
} a429_key_t;
```

Control inlining explicitly

`static inline` can *increase* code size if used indiscriminately across many call sites. Mark only the truly tiny hot paths as `static inline`; for everything else, allow the compiler to keep one out-of-line copy. Conversely, if needed, add `__attribute__((noinline))` to cold diagnostic helpers.

Avoid recursion and large VLAs (variable length arrays)

Recursion and VLAs make worst-case stack sizing hard. Replace with iterative algorithms and bounded static work buffers.

Replace `printf`-style logs with compact, binary events

A 16-byte binary record with a module ID and event code is smaller in RAM and Flash than format strings and variable arguments. Emit them to a small circular buffer only when needed.

Data placement, cache coherency, and linker control on i.MX RT1050

This SoC gives you multiple levers to tailor footprint and performance.

1. **DMA buffers must be non-cacheable and aligned**. Use the SDK macros so that the linker places them into a pre-defined non-cacheable OCRAM region, aligned to cache line size.

```
#include "fsl_common_arm.h"  /* for AT_NONCACHEABLE_SECTION macros */

#define UART_RX_BUF_SZ 128
AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t s_uartRx[UART_RX_BUF_SZ], 32);
```

1. **Quick access for truly hot paths**. If a function is on the critical path (e.g., ARINC429 parity check used in a tight ISR), place it in quick access/ITCM using:

```
#include "fsl_common_arm.h"
AT_QUICKACCESS_SECTION_CODE(static inline uint32_t a429_parity(uint32_t w))
{
    /* compute odd parity in ~6 instructions */
    w ^= w >> 16; w ^= w >> 8; w ^= w >> 4; w &= 0xF;
    static const uint8_t ptab[16] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};
    return ptab[w] ^ 1U; /* odd */
}
```

1. **Keep rarely used, large code in XIP Flash**. Bootloaders, self-test, or maintenance command handlers belong in `.text`/`.rodata` that live in external Flash. Use `.noinit` for buffers you

don't want initialized to zero (e.g., crash traces), which reduces boot-time copy/clear costs and sometimes Flash for initializers.

2. **Inspect and tune the linker script**. MCUXpresso projects use template fragments (e.g., `bss.ldt`, `data.ldt`, `main_text.ldt`) that group sections into `SRAM_ITC`, `SRAM_DTC`, and `SRAM_OC`. Confirm that only the *minimal* set is mapped to TCM.

FreeRTOS tuning that saves both Flash and RAM (when an RTOS is needed)

Many avionics functions on i.MX RT are perfectly fine bare-metal, but when you do use **RTOS** (Real-Time Operating System), configure it for the minimum viable feature set and bounded memory:

1. **Disable features you do not use**: in `FreeRTOSConfig.h` set `configUSE_TIMERS`, `configUSE_CO_ROUTINES`, `configUSE_RECURSIVE_MUTEXES`, and the trace/stats options to 0 unless your design *requires* them. Remove the corresponding source files from the build to help the linker drop dead code.
2. **Prefer static allocation**: set `configSUPPORT_STATIC_ALLOCATION` to 1, provide `vApplicationGetIdleTaskMemory()` and `vApplicationGetTimerTaskMemory()`, and use `xTaskCreateStatic`. That collapses heap requirements and variance.
3. **Right-size stacks from measurements**: use `uxTaskGetStackHighWaterMark()` during stress tests to derive certified margins. Do not over-provision by "feeling".
4. **Choose the simplest heap scheme**: `heap_1.c` (no free) or `heap_4.c` (coalescing) depending on lifecycle. In many DAL-B/C systems, dynamic allocation is only allowed during init, after which all tasks/queues exist and the allocator is not called.

A generic worked example: shrinking a UART echo app (Flash and RAM)

Start from the SDK example `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer` and measure.

1. **Baseline build**. Compile as provided (Debug). Record `size` and map file.
2. **Switch to Release flags** with `-Os -ffunction-sections -fdata-sections` and `--gc-sections`.
3. **Disable the debug console** by defining `SDK_DEBUGCONSOLE=DEBUGCONSOLE_DISABLE` in your project (or `app.h`). Replace `PRINTF` uses with minimal UART writes via `LPUART_WriteBlocking()`.
4. **Make buffers** `const` wherever possible. If you have fixed banner strings or tables, declare them `const` so they remain in Flash.
5. **Rebuild and compare**. Expect double-digit % reductions in Flash, and measurable RAM savings when `.data` shrinks.

Illustrative replacement for log prints in that example:

```
#include "fsl_lpuart.h"
#include "board.h"

static void put_str(const char *s)
{
    size_t n = 0U; while (s[n] != '\0') n++; /* tiny strlen */
    LPUART_WriteBlocking(DEMO_LPUART, (const uint8_t*)s, n);
}
```

Avionics use case: ARINC 429 (*Aeronautical Radio, Incorporated* 429) receive path with a strict memory budget

Scenario: an Airbus cabin systems controller must ingest ARINC 429 status words for several subsystems (e.g., potable water, door status) and forward a subset over a maintenance port. The hardware interface is an external ARINC 429 transceiver board (ADK-8582) connected to EVKB-IMXRT1050 via **UART**. The certification goal prohibits dynamic allocation after initialization and sets a RAM ceiling of 32 KiB for the receiver function.

Design for compactness and determinism:

- Define a **fixed-size ring buffer** of decoded 32-bit words plus a tiny metadata byte. Use 16-bit head/tail indices to keep the structure small while allowing power-of-two capacity for mask wrapping.
- Use a small **label filter** table in `const` Flash that allows only the needed labels (e.g., landing gear, doors, potable water). Store each entry as 1 byte label + 1 byte meta mask.
- Implement the UART RX ISR in interrupt mode (no DMA) with a 128-byte **non-cacheable** staging buffer—small enough to avoid cache maintenance, large enough to tolerate bursty traffic from the ADK-8582.
- Keep parity and SSM checks in a tiny inline routine tagged with `AT_QUICKACCESS_SECTION_CODE` only if the ISR budget needs it; otherwise, leave in Flash.

Key fragments (SDK-based):

```
#include "board.h"
#include "fsl_common_arm.h"
#include "fsl_lpuart.h"
#include "pin_mux.h"
#include "clock_config.h"

#define UART_BASE       LPUART1
#define UART_IRQn_      LPUART1_IRQn
#define UART_CLK_SRC    kCLOCK_UartClkRoot
```

```
/* Staging buffer for ISR; non-cacheable prevents stale reads if later moved to
DMA. */
#define UART_RX_BUF_SZ  128u
AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t s_rxStaging[UART_RX_BUF_SZ], 32);

/* Fixed pool for decoded messages; no heap used at runtime */
typedef struct {
    uint32_t word; /* 32-bit ARINC 429 word */
    uint8_t  meta; /* [7:6] SSM, [5:4] SDI, [3:0] rate id */
} a429_msg_t;

#define RX_CAPACITY 64u
static a429_msg_t g_rxRing[RX_CAPACITY];
static volatile uint16_t g_rxHead = 0, g_rxTail = 0; /* mask with RX_CAPACITY-1
*/

static inline uint16_t rb_mask(uint16_t v) { return (uint16_t)(v & (RX_CAPACITY
- 1u)); }

/* Label allow-list in Flash; not copied to RAM */
static const uint8_t s_allowedLabels[] = { 0x20, 0x21, 0x34, 0x5A, 0xAF };

AT_QUICKACCESS_SECTION_CODE(static inline uint32_t a429_parity(uint32_t w))
{
    w ^= w >> 16; w ^= w >> 8; w ^= w >> 4; w &= 0xFu;
    static const uint8_t ptab[16] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};
    return ptab[w] ^ 1u; /* odd parity */
}

static bool label_allowed(uint8_t lbl)
{
    for (size_t i = 0; i < sizeof(s_allowedLabels); ++i) if (s_allowedLabels[i]
== lbl) return true;
    return false;
}

void LPUART1_IRQHandler(void)
{
    uint32_t status = LPUART_GetStatusFlags(UART_BASE);
    if (status & kLPUART_RxDataRegFullFlag)
    {
        /* Minimal framed protocol: ADK-8582 presents 4 data bytes per word. */
        size_t got = 0;
        while ((status & kLPUART_RxDataRegFullFlag) && got < UART_RX_BUF_SZ)
        {
            s_rxStaging[got++] = LPUART_ReadByte(UART_BASE);
```

```
            status = LPUART_GetStatusFlags(UART_BASE);
        }
        /* Parse complete 4-byte words in place (little endian for example). */
        for (size_t i = 0; i + 3u < got; i += 4u)
        {
            uint32_t w = (uint32_t)s_rxStaging[i] | ((uint32_t)s_rxStaging[i+1]
<< 8) |
                         ((uint32_t)s_rxStaging[i+2] << 16) |
((uint32_t)s_rxStaging[i+3] << 24);
            uint8_t label =
(uint8_t)w; /* bits 1..8 are label in A429 LSB/8-bit label convention */
            if (!label_allowed(label)) continue;
            if (a429_parity(w) == 0u)   continue; /* drop bad parity */
            uint16_t next = rb_mask(g_rxHead + 1u);
            if (next == g_rxTail)       continue; /* overflow: drop and count
(counter omitted here) */
            g_rxRing[g_rxHead].word = w;
            g_rxRing[g_rxHead].meta = (uint8_t)(((w >> 29) & 0x3u) /* SSM */ |
(((w >> 8) & 0x3u) << 2));
            g_rxHead = next;
        }
    }
    SDK_ISR_EXIT_BARRIER;
}

static void uart_init(void)
{
    const lpuart_config_t cfg = {
        .baudRate_Bps = 115200U,
        .parityMode   = kLPUART_ParityDisabled,
        .stopBitCount = kLPUART_OneStopBit,
        .txFifoWatermark = 0,
        .rxFifoWatermark = 1,
        .enableRx = true,
        .enableTx = true
    };
    CLOCK_SetMux(kCLOCK_UartMux, 0); /* depends on board clocks */
    CLOCK_SetDiv(kCLOCK_UartDiv, 1);
    LPUART_Init(UART_BASE, &cfg, CLOCK_GetRootClockFreq(UART_CLK_SRC));
    LPUART_EnableInterrupts(UART_BASE, kLPUART_RxDataRegFullInterruptEnable);
    EnableIRQ(UART_IRQn_);
}
```

This implementation uses only statically sized arrays, places the ISR staging buffer in non-cacheable RAM, avoids `malloc`, and keeps filters in Flash. On a typical build, it consumes a few kilobytes of RAM for the ring and less than a kilobyte of Flash for the filter and ISR logic.

How to *measure* footprint correctly (repeatable, cert-friendly)

Footprint reduction is engineering, not folklore. Always capture *before/after* evidence.

1. **Use** `size` on the final ELF:

```
arm-none-eabi-size -A -d build/yourapp.elf
```

1. **Generate a symbol-sorted view** of large consumers:

```
arm-none-eabi-nm --size-sort -C -r build/yourapp.elf | tail -n 50
```

1. **Inspect the map file** produced by the linker to verify section placement (`.text` vs `.rodata` vs `.data` / `.bss`) and that `NonCacheable` buffers ended where you expect.

2. **Record configuration deltas**: exact compiler flags, `FreeRTOSConfig.h` changes, and `SDK_DEBUGCONSOLE` level. Attach these to your DO-178C Problem Reports or Software Accomplishment Summary as objective evidence.

Practical exercises on EVKB-IMXRT1050 (hands-on)

Each lab starts from an SDK example and uses only code and macros present in the provided SDK.

Lab 1 — Strip the debug console and shrink a driver example

Start: `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer`

1. Build the example as-is (Debug). Record Flash/RAM via `arm-none-eabi-size` and save the map file.
2. Create a `RELEASE` configuration with `-Os -ffunction-sections -fdata-sections` and link with `--gc-sections`.
3. Add `#define SDK_DEBUGCONSOLE DEBUGCONSOLE_DISABLE` to `app.h` or the compiler command line.
4. Replace `PRINTF` calls with the small `put_str()` shown earlier using `LPUART_WriteBlocking()`.
5. Rebuild and capture `size` output again. Explain the change: where did RODataand `.data` shrink?

Deliverable: a short note showing code/flag diffs and the numeric footprint delta.

Lab 2 — Make DMA-safe buffers without wasting cache lines

Start: any EDMA-based SDK example (e.g., `cmsis_driver_examples/lpi2c/edma_b2b_transfer`).

1. Identify buffers already declared with `AT_NONCACHEABLE_SECTION` in the example.
2. Change declarations to the aligned variant `AT_NONCACHEABLE_SECTION_ALIGN(buf, 32)` and confirm the map file shows placement into `NonCacheable`.
3. Reduce each buffer to the minimum that still meets throughput needs. Show the bound with a simple timing counter.

Deliverable: the modified source and a paragraph explaining why the chosen sizes are sufficient under worst-case ARINC 429 burst assumptions.

Lab 3 — FreeRTOS: static tasks and right-sized stacks

Start: `boards/evkbimxrt1050/freertos_examples/freertos_hello`

1. Set `configSUPPORT_STATIC_ALLOCATION` to 1 and implement `vApplicationGetIdleTaskMemory()` / `vApplicationGetTimerTaskMemory()`.
2. Change task creation to `xTaskCreateStatic` and delete all heap use after init.
3. Add periodic calls to `uxTaskGetStackHighWaterMark(NULL)` and log the minimum water mark through a binary event ring (not `printf`).
4. Reduce `configMINIMAL_STACK_SIZE` and the task stack arrays until you have a 30% headroom under peak load.

Deliverable: diff of `FreeRTOSConfig.h`, the static allocation code, and measurements.

Lab 4 — ARINC 429 UART bridge with a fixed message pool

Start: the UART ISR skeleton provided in this document.

1. Add a tiny driver that reads from the fixed ring and forwards a subset to a second interface (e.g., `LPUART3`) at a controlled rate.
2. Implement a compile-time filter for labels and demonstrate that changing the filter only changes ROData size, not RAM.
3. Verify parity and SSM filtering on representative data from the ADK-8582.

Deliverable: the complete source file(s), a one-page timing/footprint report, and a photo/screenshot of the bridge in operation.

Best practices (what consistently works on Airbus-style programs)

The following practices emerge repeatedly from flight-worthy code bases on Cortex-M7 and are compatible with the provided SDK:

- **Design for determinism first, then optimize**. Define fixed pool sizes and message rates early. You will save more RAM by eliminating unbounded behaviors than by clever micro-optimizations later.
- **Keep features behind `#if` switches**. Make debugging, verbose logs, self-tests, and traces compile-time optional so production builds exclude them entirely—this lets `--gc-sections` do its job.
- **Prefer *init-time only* dynamic allocation**. If you must use `malloc`, call it during boot, then lock out allocators before entering operational mode. Document this in your safety case.
- **Use the SDK's section macros** for DMA and TCM placement instead of handwritten linker scripts. It is clearer to reviewers and more robust across tool updates.
- **Continuously track stack** in integration tests. A single `snprintf` can dwarf your stack overnight. Enforce limits with MPU guards and watchdog resets on overflow.
- **Review the map file in every release**. Make it a checklist item. Confirm that nothing unexpectedly moved from Flash to RAM and that libc variants stayed on `nano`.
- **Document every toolchain flag** with rationale. Certification audits expect traceability from requirements to configuration.

Appendix A — Minimal FreeRTOS static allocation hooks

These are directly compatible with SDK FreeRTOS examples and avoid heap use for system tasks.

```
static StaticTask_t s_idleTaskTCB;
static StackType_t  s_idleStack[configMINIMAL_STACK_SIZE];

void vApplicationGetIdleTaskMemory( StaticTask_t **ppxIdleTaskTCBBuffer,
                                    StackType_t  **ppxIdleTaskStackBuffer,
                                    uint32_t     *pulIdleTaskStackSize )
{
    *ppxIdleTaskTCBBuffer   = &s_idleTaskTCB;
    *ppxIdleTaskStackBuffer = &s_idleStack[0];
    *pulIdleTaskStackSize   = (uint32_t)configMINIMAL_STACK_SIZE;
}

#if (configUSE_TIMERS)
static StaticTask_t s_timerTaskTCB;
static StackType_t  s_timerStack[configTIMER_TASK_STACK_DEPTH];
void vApplicationGetTimerTaskMemory( StaticTask_t **ppxTimerTaskTCBBuffer,
                                     StackType_t  **ppxTimerTaskStackBuffer,
                                     uint32_t     *pulTimerTaskStackSize )
{
```

```
    *ppxTimerTaskTCBBuffer   = &s_timerTaskTCB;
    *ppxTimerTaskStackBuffer = &s_timerStack[0];
    *pulTimerTaskStackSize   = (uint32_t)configTIMER_TASK_STACK_DEPTH;
}
#endif
```

Appendix B — Common *small* but helpful compiler switches

- `-fno-printf-return-value` (when supported) if you do not use the return value; enables slightly smaller code generated around `printf` if it remains.
- `-fno-strict-overflow` where appropriate to let the compiler fold some arithmetic without inserting overflow checks (verify safety impact).
- `-mpoke-function-name` **off** in production to avoid embedding function names in code for hard-fault dumps.

What to avoid (footprint anti-patterns)

- Pulling in large middleware for a one-off utility (e.g., using FATFS for a diagnostic file when a fixed-format UART dump works).
- Leaving the debug console enabled in production; it drags in formatters and I/O stubs.
- Over-using inlining and templates (in C++)—these easily bloat Flash.
- Storing mutable configuration in RAM when it can be read from Flash on demand or cached once.

Closing

Memory footprint reduction is about intentional design choices visible in your map file: tiny, bounded buffers; `const` lookup tables; size-oriented compilation; and section-aware placement for the i.MX RT memory hierarchy. The SDK you have already includes the needed macros and examples. The exercises above will leave you with repeatable techniques and evidence suitable for a DO-178C safety case.