CPU Load Profiling & Bottleneck Analysis

Board & tools assumed: NXP i.MX RT1050 EVKB (Cortex-M7 @ up to 600 MHz), MCUXpresso SDK (EVKB-IMXRT1050), console over **LPUART1** (`PRINTF`), no RTOS (bare-metal). For ARINC 429 topics we interface the EVKB to **ADK-8582** (ARINC 429 adapter kit) via **UART (Universal Asynchronous Receiver/Transmitter)**.

> This module goes from fundamentals to advanced techniques. Every section includes: plain-language explanation, a generic example, a realistic **avionics** use case, and **hands-on** exercises based on the EVKB MCUXpresso SDK you uploaded (`/boards/evkbimxrt1050/...`).

---

1) Why CPU load profiling matters

CPU load is the fraction of available core time consumed by useful work (and overhead) over an interval. On a real-time avionics controller, meeting deadlines is more important than raw throughput. Load tells you **how much margin** remains before deadlines are missed.

- **Instantaneous load**: utilization at a specific moment (often noisy).
- **Averaged load**: utilization integrated over a window (e.g., 10 ms, 1 s).
- **Headroom**: 100% – average load (safety margin). In avionics, headroom is sized to worst-case execution time (WCET) and **DAL (Design Assurance Level)** objectives under **DO-178C (RTCA Document 178C – Software Considerations in Airborne Systems and Equipment Certification)**.

Bottleneck analysis locates the code path or hardware resource that limits performance (e.g., Flash XIP latency, cache misses, unbounded ISR work, UART burst handling, memory copies). You always **measure first**, then narrow the hot spot, then iterate: *measure → change one thing → measure again*.

---

2) i.MX RT1050 features that help profiling

- **Cortex-M7 DWT (Data Watchpoint and Trace)**: `CYCCNT` cycle counter + event counters (`CPICNT`, `EXCCNT`, `SLEEPCNT`, `LSUCNT`, `FOLDCNT`).
- **ITM (Instrumentation Trace Macrocell)** / SWO: high-speed event logging (optional; UART `PRINTF` is simpler but intrusive).
- **SysTick**: 24-bit down counter; convenient 1 kHz timebase.
- **PIT/GPT (Periodic/GPIO Timers)**: long-period, low-overhead timekeeping or input capture.
- **GPIO**: scope/LA-visible strobes to time code or ISR sections.
- **Memory hierarchy**: **ITCM/DTCM** (tightly-coupled, deterministic), **OCRAM**, and **FlexSPI XIP** from external QSPI Flash with cache and prefetch. Many bottlenecks are actually **XIP wait states** or **cache misses**.

---

3) Load and time measurement models

1. **Idle-time method (system-wide load)**
   Let `idle_ticks` be work done only when the system is idle. Sample it periodically with SysTick or PIT.

$$\text{CPU Load \%} = 100 \times \left(1 - \frac{\Delta idle\_ticks}{\Delta idle\_ticks\_baseline}\right)$$

   The **baseline** is the increment when the CPU does nothing but idle.

2. **Cycle counting (section-level)**
   Enclose code with DWT `CYCCNT` reads: cycles → time using `CLOCK_GetFreq(kCLOCK_CpuClk)`.

3. **GPIO timing (ISR/latency)**
   Raise a pin on entry and lower on exit; measure pulse width on logic analyzer. Zero software overhead for timing, but needs equipment.

4. **Event counters**
   `EXCCNT` shows exception/ISR activity, `SLEEPCNT` shows time in `WFI` (Wait-For-Interrupt), `LSUCNT` hints at load/store stalls (e.g., unaligned, cache).

---

4) Generic walk-through: from "how busy" to "what's slow"

1) Implement an **idle loop meter** with SysTick (1 kHz).
2) Use **DWT** to bracket suspected hot functions (serialization, parsing, filtering).
3) If time spikes occur in interrupts, add a **GPIO strobe** or `CYCCNT` inside the ISR.
4) If section time is stable but slow, check **where code executes** (XIP vs ITCM), data placement (DTCM vs OCRAM), and **cache** effects.
5) Move only the hot path (e.g., a parser) to **ITCM/DTCM**; verify improvement.
6) For bursty peripherals (UART from ARINC 429 adapter), convert blocking processing to **interrupt + ring buffer** and defer heavy work to the main loop; confirm reduced ISR time and smoother load.

---

5) Avionics scenario: ARINC 429 message burst

ARINC 429 (Aeronautical Radio, Incorporated 429) provides 32-bit words at **100 kbps** (high-speed) or **12.5 kbps** (low-speed) on a unidirectional, self-clocking bus. Many aircraft subsystems (e.g., **ADC – Air Data Computer**, **AHRS – Attitude and Heading Reference System**) broadcast labels that your controller must parse, scale, sanity-check, and publish over other links.

Setup: The **ADK-8582** converts an ARINC 429 channel to a byte stream over a UART. During mode changes (e.g., sensor self-test), the adapter can burst several hundred words quickly. If the EVKB parses each byte inside the UART ISR, the ISR monopolizes the core, starving other deadlines (e.g., a control loop at 1 kHz). The symptom is **sporadic deadline misses**; the cause is an **ISR bottleneck**.

Fix path: Measure ISR time with DWT/GPIO, prove it's too long, then switch to: quick ISR → push to ring buffer → parse in main loop slices. Move the hot parser to **ITCM** and verify the load drop. Keep the UART at a baud rate with headroom (e.g., 460 800 Bd) and set a FIFO watermark to reduce IRQ rate.

---

6) Hands-on 1 — System-wide CPU load via SysTick + idle loop (bare-metal)

Goal: Compute average CPU load every 100 ms using the **idle-time method**. Uses the EVKB SDK: `board.h`, `clock_config.h`, `fsl_debug_console.h`.

Where in SDK: Start from `boards/evkbimxrt1050/project_template/`. Add the source below to `source/cpu_load.c` (or your main file) and enable the debug console.

```
#include "fsl_common.h"
#include "fsl_debug_console.h"
#include "board.h"
#include "clock_config.h"
#include "pin_mux.h"

#ifndef SYSTICK_HZ
#define SYSTICK_HZ (1000U)
#endif

static volatile uint32_t idle_counter = 0;          // increments only in idle
static volatile uint32_t idle_baseline = 1;         // measured at startup
static volatile uint32_t ms_accum = 0;              // elapsed ms in current
window
static volatile uint32_t last_idle_sample = 0;      // last idle snapshot

static inline void idle_touch(void) { idle_counter++; __NOP(); }

static void measure_idle_baseline(uint32_t ms)
{
    uint32_t start = idle_counter;
    uint32_t target = ms;
    ms_accum = 0;
    while (ms_accum < target) { idle_touch(); }
    idle_baseline = idle_counter -
start; // idle increments for 'ms' when fully idle
    if (idle_baseline == 0) idle_baseline = 1;
}
```

```
void SysTick_Handler(void)
{
    ms_accum++;
}

static void setup_systick(uint32_t cpu_hz)
{
    SysTick->LOAD  = (cpu_hz / SYSTICK_HZ) - 1U;
    SysTick->VAL   = 0;
    SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk |
SysTick_CTRL_ENABLE_Msk;
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    const uint32_t cpu_hz = CLOCK_GetFreq(kCLOCK_CpuClk);
    setup_systick(cpu_hz);

    PRINTF("\r\nCPU=%lu Hz\r\n", (unsigned long)cpu_hz);

    /* Establish baseline: how fast idle_counter increments when *only* idling
for 100 ms */
    measure_idle_baseline(100U);
    PRINTF("Idle baseline=%lu counts/100ms\r\n", (unsigned long)idle_baseline);

    uint32_t window_ms = 100U;  // report period
    uint32_t last_ms = 0;

    for (;;)
    {
        /* ---- Application work would go here (timers, polling, etc.) ---- */

        /* Idle touch marks spare cycles. Keep it cheap. */
        idle_touch();

        /* Periodic reporting from main loop (non-ISR). */
        uint32_t now_ms = ms_accum;
        if ((now_ms - last_ms) >= window_ms)
        {
            uint32_t idle_now = idle_counter;
            uint32_t idle_delta = idle_now - last_idle_sample;
            last_idle_sample = idle_now;
            last_ms = now_ms;

            /* Scale idle delta to baseline window */
            uint32_t scaled_idle = (idle_delta * 100U) / (idle_baseline == 0 ?
1U : idle_baseline);
            uint32_t load = (scaled_idle >= 100U) ? 0U : (100U - scaled_idle);
            PRINTF("Load=%u %%  (idle_delta=%lu)\r\n", (unsigned)load,
(unsigned long)idle_delta);
        }
    }
}
```

What to observe - With no extra work, `Load≈0%` (some non-zero due to interrupts).
- Add a busy loop inside `for(;;)` (e.g., a CRC or `for(volatile int i=0;i<20000;i++);`)—load should increase.
- Change `window_ms` to 10 ms and see noisier readings (shorter window).

Avionics mapping
Idle budget reflects the headroom for deterministic loops like **FCC (Flight Control Computer)** health monitoring or **BIT (Built-In Test)**. Document the minimum observed headroom.

---

7) Hands-on 2 — Fine-grained timing with DWT `CYCCNT`

Goal: Time code sections with **cycle precision** and compute µs. Uses CMSIS registers directly; MCUXpresso SDK already enables clock helpers.

```
#include "fsl_common.h"
#include "fsl_debug_console.h"
#include "board.h"
#include "clock_config.h"

static inline void dwt_init(void)
{
    /* Enable DWT & CYCCNT */
#if (__CM_CMSIS_VERSION_MAIN >= 6U)
    DCB->DEMCR |= DCB_DEMCR_TRCENA_Msk;
#else
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
#endif
    DWT->CYCCNT = 0; // reset
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

static inline uint32_t dwt_cycles(void) { return DWT->CYCCNT; }
static uint32_t dummy_crc32(const uint8_t *buf, size_t len)
{
    uint32_t crc = 0xFFFFFFFFu;
    for (size_t i = 0; i < len; ++i)
    {
        crc ^= buf[i];
        for (int b = 0; b < 8; ++b)
            crc = (crc >> 1) ^ (0xEDB88320u & (-(int)(crc & 1)));
    }
    return ~crc;
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();
    dwt_init();

    const uint32_t cpu_hz = CLOCK_GetFreq(kCLOCK_CpuClk);

    uint8_t buf[512];
    for (size_t i = 0; i < sizeof buf; ++i) buf[i] = (uint8_t)i;

    uint32_t c0 = dwt_cycles();
    (void)dummy_crc32(buf, sizeof buf);
    uint32_t c1 = dwt_cycles();

    uint32_t cycles = c1 - c0;
    uint32_t us = (uint32_t)((uint64_t)cycles * 1000000ull / cpu_hz);
    PRINTF("CRC32: %lu cycles, ~%lu us\r\n", (unsigned long)cycles, (unsigned
long)us);

    for(;;) {}
}
```

What to observe
- Re-run with the function placed in **ITCM** (see Hands-on 4) and compare cycles.
- Run the same code from **XIP** Flash (default). Expect more cycles due to Flash latency mitigated by cache/prefetch, but still slower than ITCM.

Avionics mapping
Use this to time **label parsing** and **sanity checks** for ARINC 429 words (e.g., label 203 Airspeed (BNR – Binary Number Representation), label 204 Magnetic Heading (BCD – Binary-Coded Decimal)).

8) Hands-on 3 — ISR profiling: UART burst (proxy for ARINC 429 via ADK-8582)

Goal: Convert a potentially heavy UART ISR into a short ISR that pushes bytes to a **ring buffer** and measure **ISR time**. Replace `LPUARTx`/pins with the port connected to ADK-8582.

> The EVKB debug console uses **LPUART1**. For the ARINC adapter, pick another instance (e.g., LPUART3/4) and route pins in `pin_mux.c`. The SDK driver is `devices/MIMXRT1052/drivers/fsl_lpuart.h`.

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

#define ARINC_UART       LPUART3
#define ARINC_UART_IRQn  LPUART3_IRQn
#define ARINC_UART_BAUD  460800U

#define RBUF_SIZE 1024
static volatile uint8_t rbuf[RBUF_SIZE];
static volatile uint16_t rhead = 0, rtail = 0;

/* Simple GPIO strobe on USER LED pin to scope ISR duration (active-low LED) */
static inline void isr_strobe_hi(void){ GPIO_PortSet(GPIO1, 1u << 9); }
static inline void isr_strobe_lo(void){ GPIO_PortClear(GPIO1, 1u << 9); }

static inline void rbuf_push(uint8_t b)
{
    uint16_t n = (uint16_t)(rhead + 1u);
    if (n == RBUF_SIZE) n = 0;
    if (n != rtail) { rbuf[rhead] = b; rhead = n; } // drop if full
}

void LPUART3_IRQHandler(void)
{
    isr_strobe_hi();
    uint32_t status = LPUART_GetStatusFlags(ARINC_UART);
    if (status & kLPUART_RxDataRegFullFlag)
    {
        uint8_t b = LPUART_ReadByte(ARINC_UART);
        rbuf_push(b);
    }
    LPUART_ClearStatusFlags(ARINC_UART, kLPUART_RxOverrunFlag);
    SDK_ISR_EXIT_BARRIER; // MCUXpresso SDK macro
    isr_strobe_lo();
}

static void uart_init(void)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = ARINC_UART_BAUD;
    cfg.enableTx = true;
    cfg.enableRx = true;
    LPUART_Init(ARINC_UART, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));
    LPUART_EnableInterrupts(ARINC_UART, kLPUART_RxDataRegFullInterruptEnable |
kLPUART_RxOverrunInterruptEnable);
    EnableIRQ(ARINC_UART_IRQn);
}

/* In main(): call BOARD_InitBootPins/Clocks/DebugConsole; mux pins for LPUART3;
then uart_init().
   In the main loop, drain rbuf and run the ARINC word parser there.
*/
```

Measurements
- On a scope/logic analyzer, probe `GPIO1_IO09` (USER LED) to see ISR pulse width under steady and burst traffic.
- With DWT, bracket the **parser** in the main loop and confirm ISR time drops while total throughput remains.

Avionics mapping
At **100 kbps** ARINC 429, worst-case bursts from an **AHRS** (Attitude & Heading Reference System) can cluster labels; the ring buffer prevents ISR monopolization, making control-loop jitter acceptable.

---

9) Hands-on 4 — Moving hot code to ITCM / hot data to DTCM

Goal: Demonstrate that placing hot paths in **tightly-coupled memory** reduces cycles and jitter.

1. **Place function in ITCM**
   In GCC/MCUXpresso, add a section and attribute:

```
__attribute__((section(".itcm_func")))
static uint32_t parse_label_203(const uint8_t *w)
{
    /* convert ARINC429 BNR Airspeed word into fixed-point knots, range checks,
etc. */
```

```
    /* ... */
}
```

1. **Linker file**: ensure `.itcm_func` is mapped to ITCM region. MCUXpresso EVKB linker scripts already define ITCM/DTCM; add a new section mapping.

2. **Measure** with the DWT code from Hands-on 2 before and after relocation.

Avionics mapping
Pin the **label decode + sanity check** path in ITCM, keep bulk buffers in OCRAM. Demonstrate reduced worst-case latency during UART bursts.

---

10) Advanced counters: DWT event insight

You can reset and read additional DWT counters to understand *why* it's slow:

```
static inline void dwt_counters_reset(void)
{
    DWT->CYCCNT = 0; DWT->CPICNT = 0; DWT->EXCCNT = 0; DWT->SLEEPCNT = 0; DWT-
>LSUCNT = 0; DWT->FOLDCNT = 0;
}

static inline void dwt_counters_read(uint32_t *cyc, uint8_t *cpi, uint8_t *exc,
uint8_t *slp, uint8_t *lsu, uint8_t *fld)
{
    *cyc = DWT->CYCCNT; *cpi = DWT->CPICNT; *exc = DWT->EXCCNT; *slp = DWT-
>SLEEPCNT; *lsu = DWT->LSUCNT; *fld = DWT->FOLDCNT;
}
```

- **High** `EXCCNT` → too many interrupts; increase FIFO watermark or coalesce work.
- **High** `SLEEPCNT` with low load → good headroom (lots of `WFI`).
- **High** `LSUCNT` → memory stalls; move to ITCM/DTCM or align data.

---

11) Bottleneck patterns on i.MX RT1050 (Cortex-M7)

1. **XIP from QSPI Flash**: Even with cache/prefetch, random branches or large hot sets exceed cache → extra wait states. **Symptom:** sections fast when placed in ITCM but not in Flash.
2. **Large/long ISRs**: Byte-wise parsing or `PRINTF` inside ISRs. **Fix:** buffer in ISR, process in main; never print from ISR.
3. **Busy-wait polling** of peripherals: Replace with IRQ + DMA where available.
4. **Unaligned or 8-bit accesses in tight loops**: Use 32-bit aligned buffers; copy once, process aligned.

5. **Cache management** with DMA: If later using DMA, maintain cache coherence (clean/invalidate); bad coherence appears as random spikes.
6. **FPU (Floating-Point Unit) context**: Heavy float in ISR causes lazy stacking or long save/restore; prefer fixed-point in ISR.

---

12) Best practices checklist

- **Measure with minimal perturbation**: logging over UART changes timing; prefer cycle counts or GPIO strobes for critical sections.
- **Stabilize the clock**: Know `CLOCK_GetFreq(kCLOCK_CpuClk)` and keep it constant during runs.
- **One change at a time**: Keep diffs small and attributable.
- **Pin hot code/data**: ITCM for code, DTCM for working sets; leave bulk buffers in OCRAM.
- **Keep ISRs short**: Move parsing/validation to main loop or deferred handlers. Avoid dynamic allocation and `PRINTF` in ISRs.
- **Watch heap/stack**: Overruns cause unpredictable slowdowns and faults.
- **Document WCET**: Record worst-case cycles and the test that produced them; keep artifacts for audit (aligns with DO-178C evidence needs).
- **Use SDK helpers**: `SDK_ISR_EXIT_BARRIER` avoids re-entry glitches on fast cores vs peripheral clocks.

---

13) Capstone avionics lab — ARINC 429 pipeline profiling (EVKB + ADK-8582 over UART)

Objective: Prove end-to-end timing budget from UART byte arrival to parsed/validated word publish within 1 ms deadline at 100 kbps ARINC 429 bursts.

Steps 1. Wire ADK-8582 UART to EVKB chosen LPUART (e.g., LPUART3), pins via `pin_mux.c` (Arduino header or J24 per your schematic).
2. Implement **Hands-on 3** ISR ring buffer; set UART baud (e.g., 460 800 Bd) and RX FIFO watermark to reduce IRQ rate.
3. In main loop, periodically drain the buffer, reassemble 32-bit ARINC words, verify **parity**, decode known **labels** (203/204/364 etc.), and run **range/sanity checks**.
4. Use **DWT** to time each phase: (A) ISR, (B) buffer→word assembly, (C) parse+scale, (D) publish.
5. Toggle a **GPIO** around the whole pipeline for external measurement.
6. Move phases (B) and (C) to **ITCM**; re-measure.
7. Increase adapter's output rate to emulate bursts; confirm margin remains (e.g., <40% average load, no deadlines missed).

Deliverables
- Table of cycles/µs for each phase (min/avg/max over 1000 words).
- Scope captures of ISR and pipeline GPIO pulses under nominal and burst traffic.
- Notes explaining any cache/ITCM effects observed.

---

14) Troubleshooting tips

- **Load reads >100% or negative**: Baseline recalibration needed; ensure no extra work runs during baseline.
- **No DWT counts**: Ensure `TRCENA` set and `DWT_CTRL_CYCCNTENA` enabled **after** clocks.
- **UART overrun**: Raise baud, increase RX FIFO watermark, process faster in main loop, or enlarge ring buffer.
- **Jittery times**: Disable other noisy sources temporarily; confirm caches/branch predictor enabled (board files do this by default).

---

15) What to keep for certification evidence (DO-178C)

- Exact test binaries and linker maps showing ITCM/DTCM placement.
- Measured **WCET** of each safety-relevant function and the test setup (clock, tools, stimulus).
- Rationale that residual load/headroom meets system-level timing budgets with assumed fault hypotheses.

Appendix A — Pin, clock, and macros (from SDK)

- **USER LED** is `GPIO1_IO09` on EVKB. In `board.h`:
- `USER_LED_INIT(LOGIC_LED_OFF)`, `USER_LED_TOGGLE()`.
- **Debug console** defaults to **LPUART1 @ 115200** (`BOARD_InitDebugConsole()`).
- **Useful SDK bits**: `CLOCK_GetFreq(kCLOCK_CpuClk)`, `SDK_ISR_EXIT_BARRIER`, `LPUART_*` API.

  All exercises above follow the MCUXpresso SDK style and include headers you have in `boards/evkbimxrt1050/...` and `devices/MIMXRT1052/drivers/`.

Appendix B — Minimal ARINC 429 word framing (main-loop side)

Sketch only; adapt framing to your ADK-8582 UART protocol.

```
static bool try_pop_byte(uint8_t *out)
{
    uint16_t t = rtail; if (t == rhead) return false; *out = rbuf[t]; rtail =
(uint16_t)((t + 1u) % RBUF_SIZE); return true;
}

static inline bool arinc_parity_ok(uint32_t w)
{
    /* ARINC 429 odd parity over bits 1..31; bit 32 is parity bit */
    uint32_t data = w & 0x7FFFFFFFu; // 31 bits
    uint32_t p = __builtin_popcount(data);
    uint32_t parity_bit = (w >> 31) & 1u;
    return ((p + parity_bit) & 1u) == 1u; // odd
}

static bool try_get_word(uint32_t *w)
{
    /* Example: adapter sends 4 bytes MSB first per word */
    static uint8_t acc[4]; static int n = 0; uint8_t b;
    while (try_pop_byte(&b))
    {
        acc[n++] = b; if (n == 4) { *w = ((uint32_t)acc[0]<<24)|
((uint32_t)acc[1]<<16)|((uint32_t)acc[2]<<8)|acc[3]; n = 0; return true; }
    }
    return false;
}

static void process_words_slice(void)
{
    uint32_t w; while (try_get_word(&w)) { if (!arinc_parity_ok(w))
continue; /* decode labels, scale, publish */ }
}
```

Run `process_words_slice()` periodically in the main loop between other duties; measure with DWT.

Wrap-up

You now have: (1) a **system-wide** load meter, (2) **micro-timing** for any section, (3) **ISR-safe** UART handling, and (4) a repeatable workflow to **find and fix bottlenecks**—all on the i.MX RT1050 EVKB with SDK-compliant code. Apply the same pattern to other peripherals (SPI sensors, I^{2}C ADCs, Ethernet) and maintain WCET evidence for your avionics safety case.