# Minimizing Interrupt Latency and Jitter (i.MX RT1050 EVKB, ARM Cortex-M7)

---

## 1) Why latency and jitter matter (with precise definitions)

**Interrupt latency** is the elapsed time from when a hardware event asserts an interrupt line until the **first instruction** of the corresponding Interrupt Service Routine (**ISR**) executes. **Interrupt jitter** is the variation of that latency from one occurrence to the next (i.e., the spread of the distribution, not its mean).

In avionics, we need **predictability** as much as speed. Accurate sampling, time-aligned data fusion, and bus timestamping must meet **worst-case** budgets for certification under **RTCA DO-178C** (*Software Considerations in Airborne Systems and Equipment Certification*). Jitter is often more damaging than average latency because it creates rare but dangerous outliers that break assumptions in control loops and data correlation.

---

## 2) Cortex-M7 latency path (i.MX RT1050)

From event to first ISR instruction, the path is:

1. **Peripheral → NVIC** (Nested Vectored Interrupt Controller): peripheral asserts an interrupt; request traverses on-chip interconnect.
2. **Masking and priority checks**: `PRIMASK` (global mask), `BASEPRI` (mask at/under threshold), and `FAULTMASK` (masks even faults) can delay service. If not masked and of higher priority than the current context, preemption proceeds.
3. **Automatic stacking**: core saves R0–R3, R12, LR, PC, xPSR; optionally Floating-Point Unit (**FPU**) state (depending on lazy stacking policy).
4. **Vector fetch & instruction fetch**: handler address is read from the vector table (`SCB->VTOR`), then the first instruction fetch occurs.

Micro-architectural helpers when priorities are set well: - **Tail-chaining**: switches directly between back-to-back ISRs, cutting return/entry overhead. - **Late arrival**: a higher-priority interrupt can preempt during the entry of a lower-priority ISR, protecting the latency of the more important event.

One cannot delete these steps, but one can **remove variability** from them.

---

## 3) Primary sources of latency **jitter** on i.MX RT1050

- **Execute-In-Place (XIP) from external flash** via FlexSPI and the AXI (Advanced eXtensible Interface) fabric: I-cache misses and bus contention make vector fetch and ISR fetch time variable.
- **Data cache and DMA (Direct Memory Access) interactions**: without explicit clean/invalidate discipline, buffer sharing between CPU and EDMA (Enhanced DMA) introduces unpredictable maintenance points and stale reads/writes.
- **Lazy FPU stacking**: the first floating-point instruction in an ISR triggers FP state save/restore, creating sporadic long entries.
- **Masking strategy**: long `__disable_irq()` (global mask) windows inflate jitter across the system; `BASEPRI` misuse can also block a lifeline ISR if priorities are mis-assigned.
- **Long or branchy ISRs**: medium-priority ISRs that run long block higher-priority ISRs; heavy branching increases path-length variance on Cortex-M7.
- **Clock and power management**: late enable of clocks or bus gating adds wake-up variability.

---

## 4) Design rules that consistently reduce latency and jitter

### 4.1 Use priorities and masking deliberately

- Assign the truly timing-critical ISR a **numerically small** priority (e.g., 1; 0 is the absolute highest). Document a priority map.
- Avoid global masking (`PRIMASK`); guard short critical sections with **`BASEPRI`** so the lifeline ISR can still preempt instantly.

### 4.2 Make memory deterministic

- Place the **vector table** and timing-critical ISRs in **ITCM** (Instruction Tightly Coupled Memory); place their hot data in **DTCM** (Data TCM). This removes XIP/cache/AXI variability.
- If executing from flash is unavoidable, keep ISRs tiny and straight-line so they stay I-cache hot (still not as deterministic as ITCM).

### 4.3 Keep ISRs tiny, constant-time, and branch-light

- Acknowledge the device, capture timestamp/counter, enqueue a compact record, and exit. Push all parsing/logging to a bottom half.

### 4.4 Floating-point policy

- Prefer **no float in ISRs**. If float is required, disable lazy stacking to pay a constant entry cost every time. Do not allow rare first-use penalties to become outliers.

### 4.5 Cache discipline with DMA

- Easiest deterministic choice: place DMA rings in **non-cacheable** memory.
- If buffers must be cacheable, enforce a documented **clean/invalidate + barrier** protocol at handoff points (producer cleans before DMA-out; consumer invalidates after DMA-in).

### 4.6 Strictly periodic scheduling

- Program `next_deadline = previous_deadline + period`, not `now + period`, to prevent drift and convert occasional delays into non-accumulating phase errors.

### 4.7 Stabilize clocks

- Keep the GPT (General Purpose Timer) / PIT (Periodic Interrupt Timer) and relevant interconnect clocks enabled at known divisors during critical operation.

---

## 5) Measuring what matters (before optimizing)

Use **two timebases**: - **DWT** (Data Watchpoint and Trace) `CYCCNT` for core-cycle timing. - **GPT/PIT** as a peripheral domain clock to timestamp ISR entry relative to scheduled compare events.

Also toggle a **GPIO** (e.g., EVKB user LED on `GPIO1_IO09`) at ISR entry/exit and view on a logic analyzer. Collect **min/mean/max** and **p99.9** under idle and **worst-case contention** (UART floods, DMA bursts). If you cannot measure the worst case, you do not own the worst case.

---

## 6) Generic worked example: a rock-steady 10 kHz timestamp ISR

**Goal**: ≤ 2 µs worst-case entry, ≤ 0.5 µs jitter.

**Concept**: GPT1 free-runs; schedule strictly periodic compares. The ISR (in ITCM, high priority) sets a probe GPIO high, reads GPT, computes `latency = now – expected`, schedules `expected + period`, clears the flag, sets GPIO low, exits. No float, no logging, no branches beyond the essential.

**Why it works**: vector and ISR fetch are single-cycle (ITCM), data hits DTCM, preemption is guaranteed by priority and `BASEPRI` discipline, and the code path is constant-time.

**SDK-style snippet (compiles on EVKB-IMXRT1050)**

```
#include "fsl_gpt.h"
#include "fsl_gpio.h"
#include "fsl_clock.h"
#include "fsl_common.h"
#include "board.h"

static inline void DWT_EnableCycleCounter(void) {
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL  |= DWT_CTRL_CYCCNTENA_Msk;
}

volatile uint32_t g_last_target;
volatile uint32_t g_latency_ticks;
volatile uint32_t g_isr_count;

#define GPTx            GPT1
#define GPTx_IRQn       GPT1_IRQn
#define PERIOD_US       (100U)
static uint32_t gpt_ticks_per_us;

#ifndef BOARD_USER_LED_GPIO
#define BOARD_USER_LED_GPIO     GPIO1
#define BOARD_USER_LED_GPIO_PIN 9U
#endif

__attribute__((section(".itcm_func")))
void GPT1_IRQHandler(void)
{
    GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, 1);
    uint32_t now    = GPT_GetCurrentTimerCount(GPTx);
    uint32_t expect = g_last_target;
    g_latency_ticks = now - expect; /* natural wrap */
    g_isr_count++;

    g_last_target = expect + (PERIOD_US * gpt_ticks_per_us);
    GPT_SetOutputCompareValue(GPTx, kGPT_OutputCompare_Channel1,
g_last_target);

    GPT_ClearStatusFlags(GPTx, kGPT_OutputCompare1Flag);
    GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, 0);
    __DSB(); __ISB();
}

static void LatencyDemo_Init(void)
```

```
{
    BOARD_BootClockRUN();
    gpio_pin_config_t out = { kGPIO_DigitalOutput, 0, kGPIO_NoIntmode };
    GPIO_PinInit(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, &out);

    gpt_config_t cfg; GPT_GetDefaultConfig(&cfg); cfg.enableFreeRun = true;
    GPT_Init(GPTx, &cfg);
    GPT_SetClockDivider(GPTx, 1);
    uint32_t gpt_freq = CLOCK_GetFreq(kCLOCK_PerClk);
    gpt_ticks_per_us  = gpt_freq / 1000000U;

    GPT_StartTimer(GPTx);
    uint32_t now = GPT_GetCurrentTimerCount(GPTx);
    g_last_target = now + 10 * gpt_ticks_per_us;
    GPT_SetOutputCompareValue(GPTx, kGPT_OutputCompare_Channel1,
g_last_target);
    GPT_EnableInterrupts(GPTx, kGPT_OutputCompare1InterruptEnable);

    NVIC_SetPriority(GPTx_IRQn, 1U);
    NVIC_EnableIRQ(GPTx_IRQn);

    DWT_EnableCycleCounter();
}
```

**Linker/startup note**: add a section `.itcm_func` mapped to ITCM; relocate the vector table (`SCB->VTOR` → ITCM) with `__DSB(); __ISB();` after the write. Place hot ISR data in DTCM similarly.

---

## 7) Avionics worked example: ADK-8582 (ARINC 429) → UART with tight timestamps

**ARINC 429** (Aeronautical Radio, Incorporated 429) transmits 32-bit words at **100 kbps** (10 µs/bit) or **12.5 kbps** (80 µs/bit). The **ADK-8582** board decodes ARINC 429 and streams decoded words over **LPUART** (Low-Power Universal Asynchronous Receiver/Transmitter). We must avoid RX overruns during bursts and timestamp labels within, say, **≤ 5 µs** at 100 kbps.

**Deterministic architecture**: - **Timestamp is sovereign**: GPT ISR in ITCM at priority 1; it only captures time and marks boundaries. - **Bulk RX via EDMA**: LPUART uses EDMA into **non-cacheable** rings; the EDMA completion ISR (priority 3–5) advances pointers and stores a concise “buffer-ready @ time T” record. - **Parsing in thread context**: label decoding and parity checks happen outside ISRs; short `BASEPRI` regions protect pointer swaps without globally masking.

**SDK-style skeleton**

```
#include "fsl_lpuart.h"
#include "fsl_dmamux.h"
#include "fsl_edma.h"
#include "fsl_cache.h"
#include "board.h"

AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t s_rxRing[2048], 16);
static edma_handle_t           s_edmaHandle;
static lpuart_edma_handle_t    s_lpuartEdmaHandle;

static void UART_EDMA_UserCallback(LPUART_Type *base,
                                   lpuart_edma_handle_t *handle,
                                   status_t status,
                                   void *userData)
{
    (void)base; (void)handle; (void)userData;
    if (status == kStatus_LPUART_RxIdle || status ==
kStatus_LPUART_IdleLineDetected)
    {
        uint32_t tstamp = GPT_GetCurrentTimerCount(GPT1); /* minimal
timestamp */
        /* record {buffer_index, tstamp}; do not parse here */
    }
}

static void UART_Rx_EDMA_Init(void)
{
    CLOCK_EnableClock(kCLOCK_Dma0);
    CLOCK_EnableClock(kCLOCK_Dmamux0);
    DMAMUX_Init(DMAMUX0);
    EDMA_Init(DMA0);

    lpuart_config_t cfg; LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = 2000000U; cfg.enableTx = false; cfg.enableRx = true;
    LPUART_Init(LPUART1, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));

    EDMA_CreateHandle(&s_edmaHandle, DMA0, 0);
    LPUART_TransferCreateHandleEDMA(LPUART1, &s_lpuartEdmaHandle,
                                    UART_EDMA_UserCallback, NULL,
                                    &s_edmaHandle, NULL);

    lpuart_transfer_t xfer = { .data = s_rxRing, .dataSize = sizeof(s_rxRing)
};
    LPUART_TransferReceiveEDMA(LPUART1, &s_lpuartEdmaHandle, &xfer);
```

```
    NVIC_SetPriority(DMA0_0_IRQn, 3U); /* below GPT, above UART */
    NVIC_SetPriority(LPUART1_IRQn, 5U);
}
```

**Why this minimizes jitter**: the timestamp ISR always preempts because the system uses `BASEPRI`, not global masking; its code/data run from TCM; UART traffic does not generate per-byte interrupts; DMA buffers are non-cacheable, avoiding unpredictable cache maintenance in ISRs.

---

## 8) Hands-on labs (with MCUXpresso SDK paths)

**A. Baseline latency & jitter with GPT**
Start from `boards/evkbimxrt1050/driver_examples/gpt/timer/`. Add the 10 kHz ISR above, place it and vectors in ITCM, and toggle GPIO1_IO09 on entry/exit. Measure min/mean/max and p99.9 under idle.

**B. Stress with UART floods (interrupt mode)**
Start from `boards/evkbimxrt1050/driver_examples/lpuart/interrupt/`. Flood RX at 1–2 Mbps. Compare jitter when background code uses **`BASEPRI`** vs. global masking (`__disable_irq()`): the latter inflates jitter.

**C. Switch UART RX to EDMA**
Start from `boards/evkbimxrt1050/cmsis_driver_examples/lpuart/edma_transfer/`. Use `AT_NONCACHEABLE_SECTION_ALIGN` for RX rings; keep the EDMA callback minimal and timestamp it with GPT. Jitter improves due to lower interrupt pressure and short ISRs.

**D. ARINC 429 end-to-end with ADK-8582**
Connect ADK-8582 to LPUART. Keep GPT ISR at priority 1 (ITCM), EDMA at 3–5, UART at 5–7. Demonstrate **zero RX overrun** during a scripted 100 kbps burst and timestamp error ≤ 5 µs.

---

## 9) Best practices (with reasons)

- **Tiny, constant-time ISRs**: acknowledge, timestamp, queue, exit. Parsing/logging in bottom halves only.
- **Prefer `BASEPRI` to `PRIMASK`**: allow lifeline ISR preemption; jitter collapses when you stop pausing the world.
- **ITCM/DTCM for critical paths**: removes XIP/cache/AXI variability.
- **Non-cacheable DMA rings**: the simplest deterministic coherency model.

- **No `printf` or allocation in ISRs**: unbounded latency and cache churn.
- **Written priority map**: avoid accidental inversions; treat priorities as architecture, not a tweak.
- **Program periodic timers from previous deadline**: prevents phase drift and jitter accumulation.
- **Instrument and keep distributions**: worst-case and p99.9 are design-closing numbers; means are not.

---

## 10) Acceptance criteria and verification

- **Latency**: GPT compare ISR worst-case entry ≤ **2 µs** (example budget) at 600 MHz core; measured over ≥ 10^6 events with UART/DMA load present.
- **Jitter**: p99.9 of entry latency ≤ **0.5 µs**.
- **No data loss**: 0 LPUART RX overruns during **N** minutes with worst-case ARINC 429 bursts.
- **Regression guard**: testbench fails build if worst-case widens by >10% from baseline.

---

## 11) Troubleshooting guide

- **Rare long outlier on first ISR run** → FPU lazy stacking. **Fix**: ban floats in ISRs or disable lazy stacking to pay a constant cost.
- **Heavy tail only with XIP** → I-cache/AXI variance. **Fix**: move vectors/ISR to ITCM; keep hot data in DTCM.
- **Timing drift over minutes** → scheduled from `now + period`. **Fix**: schedule `previous + period`.
- **UART overruns under load** → per-byte RX interrupts or cacheable DMA. **Fix**: EDMA + non-cacheable rings.
- **Stale or duplicated DMA data** → missing cache maintenance. **Fix**: non-cacheable buffers or explicit clean/invalidate + barriers.

---

## 12) Reusable snippets (SDK-compatible)

**Enable DWT `CYCCNT`**

```c
static inline void DWT_Init(void) {
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    (void)DWT->CYCCNT; /* probe presence */
    DWT->CYCCNT = 0;
    DWT->CTRL  |= DWT_CTRL_CYCCNTENA_Msk;
}
```

**`BASEPRI` helpers**

```c
static inline uint32_t irq_lock_basepri(uint32_t prio) {
    uint32_t old = __get_BASEPRI();
    __set_BASEPRI(prio << (8 - __NVIC_PRIO_BITS));
    __DSB(); __ISB();
    return old;
}
static inline void irq_unlock_basepri(uint32_t old) {
    __set_BASEPRI(old);
    __DSB(); __ISB();
}
```

**Non-cacheable DMA buffers**

```c
AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t s_dmaRing[2048], 16);
```

**Relocate vector table to ITCM**

```c
extern uint32_t __isr_vector_start__;
SCB->VTOR = ((uint32_t)&__isr_vector_start__) & SCB_VTOR_TBLOFF_Msk;
__DSB(); __ISB();
```

---

## 13) Closing thought

Four choices usually get you to avionics-friendly determinism on i.MX RT1050: **(1)** place vectors and critical ISRs in **ITCM** with data in **DTCM**, **(2)** use **`BASEPRI`** instead of global masking, **(3)** keep ISRs tiny and defer work, and **(4)** handle heavy I/O with **EDMA** into **non-cacheable** rings. The labs above, built on your EVKB SDK, let you measure and *prove* both latency and jitter budgets for DO-178C evidence.