Task Registration & Scheduling — Application Task Lifecycle (i.MX RT1050-EVKB)

1) Why this topic matters (avionics context)

In avionics, especially for **DAL** (Design Assurance Level) A/B functions under **DO-178C** (Software Considerations in Airborne Systems and Equipment Certification), every application task must have a *predictable* lifecycle. From power-up to shutdown, we need to know *when* a task starts, *how often* it runs, *how long* it takes (its **WCET** — Worst-Case Execution Time), and *what happens if it overruns*. On the **i.MX RT1050-EVKB** (NXP i.MX RT1052, Arm® Cortex-M7), we can build a robust, traceable task lifecycle with bare-metal code using the **MCUXpresso SDK** and a thin cooperative scheduler. This module walks you from fundamentals to advanced concepts with realistic avionics use cases, including **ARINC 429** (Aeronautical Radio, Incorporated 429) receive/transmit flows via **ADK-8582** interfaced over **UART** (Universal Asynchronous Receiver/Transmitter).

---

2) Essential definitions (expanded on first use)

- **Task**: A unit of work with a defined purpose, period, priority, inputs, outputs, and timing budget. In bare metal, a task is typically a C function plus metadata (period, state, etc.).
- **ISR** (Interrupt Service Routine): A short, bounded-time routine that handles asynchronous events (e.g., timer tick, UART RX). ISRs should defer non-urgent work to tasks.
- **Scheduler**: Code that decides *which* task runs *when*. We will implement a **TTC** (Time-Triggered Cooperative) scheduler; you will also learn how to extend it toward fixed-priority preemption later.
- **Period/Frequency**: How often a periodic task is released (e.g., 10 ms period = 100 Hz).
- **Release**: The instant a task becomes eligible to run. In TTC, tasks run when released and the CPU is free.
- **Deadline**: Latest acceptable completion time for a task instance.
- **WCET** (Worst-Case Execution Time): Upper bound on the task's execution time.
- **Jitter**: Variation in task start/finish time around the nominal schedule.
- **Sporadic task**: Runs on demand, with a minimum inter-arrival time (e.g., fault reporter).
- **Aperiodic task**: Runs when an event occurs without a guaranteed spacing (e.g., maintenance command).
- **Overrun**: When a task has not finished before the next release time.

3) Platform specifics you will use

- **CPU**: Arm Cortex-M7 @ up to 600 MHz (i.MX RT1050).
- **Tick source**: **SysTick** timer (1 ms tick) from Arm core.
- **SDK**: MCUXpresso SDK (file provided: `SDK_25_06_00_EVKB-IMXRT1050.zip`). We will reuse board init, clock init, pin mux, and the debug console.
- **I/O for labs**:
    - **Debug UART** via `BOARD_InitDebugConsole()` (LPUART1) — for logs (`PRINTF`).
    - **User LED** `BOARD_USER_LED_GPIO/BOARD_USER_LED_GPIO_PIN` — for timing visibility.
    - **ADK-8582** on a *separate* LPUART instance (e.g., LPUART3) — for ARINC 429 bridging exercises. You will map pins in `pin_mux.c` with MCUXpresso ConfigTools.

  Tip: All pin/clock/console macros are already present in the SDK's `board.h` within each example; we will follow the SDK patterns to stay aligned with your codebase.

---

4) Application Task Lifecycle (concept & state model)

A robust lifecycle enforces *determinism* and *traceability*:

States 1. **UNREGISTERED** → not known to the scheduler. 2. **REGISTERED** → in scheduler table with metadata. 3. **INITIALIZED** → `init()` ran once; resources allocated. 4. **READY** → released by time or event, waiting to run. 5. **RUNNING** → `run()` executing to completion. 6. **SUSPENDED** → temporarily disabled (e.g., degraded mode, missing dependency). 7. **ERROR** → faulted; requires recovery policy.

Transitions - Power-up: `UNREGISTERED → REGISTERED → INITIALIZED` (during system bring-up). - Scheduler tick release: `INITIALIZED/READY → READY` when `now >= next_release`. - Dispatcher: `READY → RUNNING → READY` (on success) or `→ ERROR` (on failure/overrun policy). - Mode control: `READY/RUNNING → SUSPENDED` (crew command/degraded mode) and back.

Artefacts per task - Name, `init(void*)`, `run(void*)`, context pointer, period (ms), priority (for tie-breakers), next release time, counters (overruns/executions), and state.

---

5) Registration patterns on bare metal

5.1 Static table (DAL-friendly)

Define tasks at compile time in a constant table. Benefits: traceability, no heap use, deterministic footprint, easy certification evidence.

5.2 Dynamic registration (careful)

Allow (de)registration at runtime from a bounded static pool. You *must* prove boundedness: fixed pool size, strict error handling if pool is full, time-bounded list operations, no heap.

In this course, we implement both. The production default is **static table**; dynamic is used only for mode switching or optional features.

---

6) Scheduling models (what we implement & why)

- **Super-loop**: simplest; no explicit timing — not sufficient for avionics periodicity.
- **TTC** (Time-Triggered Cooperative): periodic scheduler driven by SysTick; tasks run to completion and must be short. Very deterministic, certification-friendly. **We implement this.**
- **Fixed-priority preemptive** (e.g., Rate Monotonic — **RM**): tighter latency for high-rate tasks but added complexity (context switch, stack per task). Optional extension is outlined for later.
- **Time partitioning** (ARINC 653 style): generally done with an RTOS/hypervisor; out of scope for this bare-metal module, but concepts (windows, budgets) are mirrored in our TTC parameters.

---

7) Avionics use case (realistic scenario)

Scenario: A **Data Concentrator Unit (DCU)** ingests **ARINC 429** high-speed (100 kbps) words from multiple sensors via an **ADK-8582** bridge on UART, performs integrity checks, and republishes selected parameters. Typical labels include *Airspeed* (label 203), *Altitude* (label 204), *Angle of Attack* (label 102). The DCU must:

- **Receive** ARINC 429 via ADK-8582 UART stream, parse to structured words, time-stamp, and write to a shared database.
- **Run periodic monitors** (e.g., **BIT** — Built-In Test; **HM** — Health Monitoring) every 100 ms.
- **Republish** computed data at 20 Hz (every 50 ms) to a secondary link or log.
- **Degrade** gracefully: if the UART link is quiet for >500 ms, suspend the ARINC tasks and raise a caution.

Mapped tasks (periods chosen to match typical update rates): - `arinc429_rx_task` — 1 ms period (budget 50 µs): pull bytes from UART ring buffer, frame words, enqueue. - `arinc429_tx_task` — 50 ms period: package selected labels for transmission (or echo). - `bit_100ms_task` — 100 ms: BIT/HM metrics. - `telemetry_task` — 200 ms: log counters via debug console. - `led_hb_task` — 1000 ms: heartbeat LED.

---

8) Hands-on — build a TTC scheduler on i.MX RT1050

> All code below follows the MCUXpresso SDK patterns. Start from the SDK **hello_world** or **led_blinky** example to inherit `board.c`, `board.h`, `pin_mux.c`, `clock_config.c`, and `BOARD_InitDebugConsole()`.

8.1 File: `scheduler.h`

```c
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include <stdint.h>
#include <stdbool.h>

typedef enum {
    TASK_UNREGISTERED = 0,
    TASK_REGISTERED,
    TASK_INITIALIZED,
    TASK_READY,
    TASK_RUNNING,
    TASK_SUSPENDED,
    TASK_ERROR
} task_state_t;

typedef struct task_tag {
    const char *name;
    void (*init)(void *ctx);
    void (*run)(void *ctx);
    void *ctx;
    uint32_t period_ms;      // 0 for aperiodic; use event to release
    uint32_t next_release_ms;
    uint8_t priority;        // 0 = highest
    volatile task_state_t state;
    volatile uint32_t executions;
    volatile uint32_t overruns;
} task_t;

void scheduler_init(uint32_t tick_hz);
bool task_register(task_t *t);
void task_suspend(task_t *t);
void task_resume(task_t *t, uint32_t delay_ms);
```

```
void scheduler_run(void);
void scheduler_tick_isr(void); // call from SysTick_Handler
uint32_t scheduler_now_ms(void);

#endif // SCHEDULER_H
```

8.2 File: scheduler.c

```
#include "scheduler.h"
#include "fsl_common.h"
#include "fsl_debug_console.h"
#include <string.h>

#ifndef MAX_TASKS
#define MAX_TASKS 16
#endif

static volatile uint32_t g_ms_ticks = 0;
static task_t *g_tasks[MAX_TASKS];
static uint32_t g_task_count = 0;

static inline int cmp_prio(const task_t *a, const task_t *b) {
    return (a->priority < b->priority) ? -1 : (a->priority > b->priority);
}

void scheduler_init(uint32_t tick_hz)
{
    SystemCoreClockUpdate();
    (void)tick_hz; // we use fixed 1 ms below for simplicity
    if (SysTick_Config(SystemCoreClock / 1000U)) {
        while (1) { /* error */ }
    }

    // Enable DWT cycle counter for timing (optional but useful)
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

bool task_register(task_t *t)
{
    if ((t == NULL) || (t->run == NULL)) return false;
    if (g_task_count >= MAX_TASKS) return false;

    t->state = TASK_REGISTERED;
    t->executions = 0;
    t->overruns = 0;
    t->next_release_ms = g_ms_ticks + t->period_ms; // first release after
one period
```

```
    if (t->init) {
        t->init(t->ctx);
    }
    t->state = TASK_INITIALIZED;

    g_tasks[g_task_count++] = t;
    return true;
}

void task_suspend(task_t *t)
{
    if (t) t->state = TASK_SUSPENDED;
}

void task_resume(task_t *t, uint32_t delay_ms)
{
    if (!t) return;
    t->next_release_ms = g_ms_ticks + delay_ms;
    t->state = TASK_INITIALIZED; // will move to READY when released
}

void scheduler_tick_isr(void)
{
    g_ms_ticks++;
}

uint32_t scheduler_now_ms(void)
{
    return g_ms_ticks;
}

void scheduler_run(void)
{
    for (;;) {
        uint32_t now = g_ms_ticks;
        task_t *best = NULL;

        // Pick the highest priority READY task whose release time has
arrived
        for (uint32_t i = 0; i < g_task_count; i++) {
            task_t *t = g_tasks[i];
            if (!t) continue;
            if (t->state == TASK_SUSPENDED || t->state == TASK_ERROR)
continue;

            if (t->period_ms == 0U) {
                // aperiodic: only runs when someone sets next_release_ms ==
now
```

```
            }

            if (now >= t->next_release_ms) {
                // Make it READY implicitly for cooperative run
                if (best == NULL) {
                    best = t;
                } else {
                    // choose by priority, then by earliest next_release
                    if (cmp_prio(t, best) < 0 ||
                        (t->priority == best->priority && t->next_release_ms
< best->next_release_ms)) {
                        best = t;
                    }
                }
            }
        }

        if (best) {
            // Dispatch
            uint32_t start = DWT->CYCCNT;
            best->state = TASK_RUNNING;
            best->run(best->ctx);
            best->executions++;
            best->state = TASK_READY;

            // Schedule next release
            uint32_t expected = best->next_release_ms + best->period_ms;
            best->next_release_ms = expected;

            // Overrun accounting: if we are already past next release, fix
up
            if (g_ms_ticks > expected) {
                best->overruns++;
                // catch-up: push to the next boundary after now
                uint32_t late = g_ms_ticks - expected;
                uint32_t periods_missed = 1U + late / (best->period_ms ?
best->period_ms : 1U);
                best->next_release_ms = expected + periods_missed * best-
>period_ms;
            }

            (void)start; // available for timing prints if needed
        } else {
            __WFI(); // save power; wake on next tick/interrupt
        }
    }
}
```

8.3 File: `app_tasks.c` — example tasks (LED, telemetry, button)

```c
#include "board.h"
#include "fsl_debug_console.h"
#include "fsl_gpio.h"
#include "scheduler.h"

// --- LED heartbeat (1 Hz) ---
static void led_init(void *ctx)
{
    (void)ctx;
    USER_LED_INIT(LOGIC_LED_OFF);
}

static void led_run(void *ctx)
{
    (void)ctx;
    USER_LED_TOGGLE();
}

// --- Telemetry: print counters every 200 ms ---
extern task_t t_led, t_telemetry; // declared in main.c or app_tasks.h

static void telemetry_init(void *ctx) { (void)ctx; }

static void telemetry_run(void *ctx)
{
    (void)ctx;
    static uint32_t hb = 0;
    PRINTF("[telemetry] tick=%lu led.exec=%lu led.ovr=%lu\r\n",
           (unsigned long)scheduler_now_ms(),
           (unsigned long)t_led.executions,
           (unsigned long)t_led.overruns);
    if (++hb % 5U == 0U) {
        PRINTF("[telemetry] alive\r\n");
    }
}

// --- Button sampler (10 ms) ---
#ifndef BOARD_USER_BUTTON_GPIO
#define BOARD_USER_BUTTON_GPIO GPIO5
#define BOARD_USER_BUTTON_GPIO_PIN 0U
#endif

static void button_init(void *ctx)
{
    (void)ctx;
    // Input by default via pin mux; ensure clock to GPIO enabled in board
init
```

```
}

static void button_run(void *ctx)
{
    (void)ctx;
    static uint8_t stable = 1U;
    static uint8_t last = 1U;
    uint8_t now = (uint8_t)GPIO_PinRead(BOARD_USER_BUTTON_GPIO,
BOARD_USER_BUTTON_GPIO_PIN);
    if (now == last) {
        if (stable != now) {
            stable = now;
            PRINTF("[button] state=%u at %lu ms\r\n", (unsigned)stable,
(unsigned long)scheduler_now_ms());
        }
    }
    last = now;
}

// Public task descriptors
#include <stddef.h>

task_t t_led       = { .name = "led",       .init = led_init,       .run =
led_run,       .ctx = NULL, .period_ms = 1000U, .priority = 3 };
task_t t_telemetry = { .name = "telemetry", .init = telemetry_init, .run =
telemetry_run, .ctx = NULL, .period_ms = 200U,  .priority = 4 };
task_t t_button    = { .name = "button",    .init = button_init,    .run =
button_run,    .ctx = NULL, .period_ms = 10U,   .priority = 2 };
```

8.4 File: `main.c` — bring-up & run scheduler

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

// Declarations from app_tasks.c
extern task_t t_led, t_telemetry, t_button;

void SysTick_Handler(void)
{
    scheduler_tick_isr();
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole(); // LPUART1
```

```
    PRINTF("\r\nIMXRT1050 TTC scheduler demo\r\n");

    scheduler_init(1000U); // 1 kHz tick

    (void)task_register(&t_button);
    (void)task_register(&t_led);
    (void)task_register(&t_telemetry);

    scheduler_run(); // never returns
}
```

What you should see - LED toggles once per second. - UART (115200 8N1 on LPUART1) prints telemetry every 200 ms and button events when pressed.

> All macros such as `USER_LED_*`, `BOARD_InitBoot*`, and debug console are provided by the SDK's `board.h` in the EVKB examples you started from.

---

9) ARINC 429 integration via ADK-8582 (UART) — tasks & skeleton

> The ADK-8582 is attached to a **separate** UART (e.g., LPUART3) so we do not interfere with the debug console on LPUART1. Configure TX/RX pins for LPUART3 in `pin_mux.c` using MCUXpresso ConfigTools. Keep DMA optional; interrupt-driven ring buffer is sufficient at 115200 bps.

9.1 UART driver glue (ring buffer) — `adk_uart.c`

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

#ifndef ADK_LPUART
#define ADK_LPUART LPUART3
#endif

#ifndef ADK_LPUART_CLK_FREQ
#define ADK_LPUART_CLK_FREQ CLOCK_GetFreq(kCLOCK_UartClk) // or
BOARD_DebugConsoleSrcFreq() if routed similarly
#endif

#ifndef ADK_BAUD
#define ADK_BAUD 115200U
#endif

#define RX_RING_BUFFER_SIZE 256U

static lpuart_handle_t s_handle;
static uint8_t s_rx_ring[RX_RING_BUFFER_SIZE];
```

```
void ADK_UART_Init(void)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = ADK_BAUD;
    cfg.enableRx = true;
    cfg.enableTx = true;
    LPUART_Init(ADK_LPUART, &cfg, ADK_LPUART_CLK_FREQ);
    LPUART_TransferCreateHandle(ADK_LPUART, &s_handle, NULL, NULL);
    LPUART_TransferStartRingBuffer(ADK_LPUART, &s_handle, s_rx_ring,
RX_RING_BUFFER_SIZE);
}

size_t ADK_UART_Read(uint8_t *dst, size_t maxLen)
{
    size_t n = 0;
    (void)LPUART_TransferReceiveNonBlocking(ADK_LPUART, &s_handle, dst,
maxLen, &n);
    return n; // bytes actually copied from ring buffer
}

void ADK_UART_Write(const uint8_t *src, size_t len)
{
    lpuart_transfer_t xfer = { .data = (uint8_t *)src, .dataSize = len };
    (void)LPUART_TransferSendNonBlocking(ADK_LPUART, &s_handle, &xfer);
}
```

9.2 ARINC 429 tasks — `arinc_tasks.c`

The ADK typically emits bytes that can be framed into ARINC 429 words (parity, label, SDI, data, SSM). Your exact protocol may vary; here we sketch a minimal parser shell.

```
#include "scheduler.h"
#include "fsl_debug_console.h"
#include <string.h>

#define QUIET_TIMEOUT_MS 500U

typedef struct {
    uint32_t words_rx;
    uint32_t words_tx;
    uint32_t last_rx_ms;
    uint8_t  link_ok;
} arinc_ctx_t;

static arinc_ctx_t g_arinc;

// --- RX (1 ms) ---
```

```c
static void arinc_rx_init(void *ctx)
{
    (void)ctx; g_arinc.link_ok = 0U; g_arinc.last_rx_ms = scheduler_now_ms();
}

static void arinc_rx_run(void *ctx)
{
    (void)ctx;
    uint8_t buf[64];
    size_t n = ADK_UART_Read(buf, sizeof(buf));
    if (n > 0U) {
        g_arinc.last_rx_ms = scheduler_now_ms();
        g_arinc.link_ok = 1U;
        // TODO: parse bytes into ARINC 429 word(s); demo increments counter
        g_arinc.words_rx += (uint32_t)n; // placeholder for demo
    } else {
        if ((scheduler_now_ms() - g_arinc.last_rx_ms) > QUIET_TIMEOUT_MS) {
            g_arinc.link_ok = 0U; // quiet — declare link down
        }
    }
}

// --- TX (50 ms) ---
static void arinc_tx_init(void *ctx) { (void)ctx; }

static void arinc_tx_run(void *ctx)
{
    (void)ctx;
    if (g_arinc.link_ok) {
        const char msg[] = "TX: label203 mock\r\n"; // replace with real
framing for ADK-8582
        ADK_UART_Write((const uint8_t*)msg, sizeof(msg)-1U);
        g_arinc.words_tx++;
    }
}

// --- BIT/HM (100 ms) ---
static void bit_init(void *ctx) { (void)ctx; }

static void bit_run(void *ctx)
{
    (void)ctx;
    PRINTF("[BIT] rx=%lu tx=%lu link=%u\r\n",
           (unsigned long)g_arinc.words_rx,
           (unsigned long)g_arinc.words_tx,
           (unsigned)g_arinc.link_ok);
}

// Expose task descriptors
```

```
#include <stddef.h>

task_t t_arinc_rx = { .name = "arinc_rx", .init = arinc_rx_init, .run =
arinc_rx_run, .ctx = NULL, .period_ms = 1U,   .priority = 0 };
task_t t_arinc_tx = { .name = "arinc_tx", .init = arinc_tx_init, .run =
arinc_tx_run, .ctx = NULL, .period_ms = 50U,  .priority = 1 };
task_t t_bit      = { .name = "bit_100ms",.init = bit_init,      .run =
bit_run,      .ctx = NULL, .period_ms = 100U, .priority = 1 };
```

9.3 Add ARINC tasks to `main.c`

```
extern task_t t_arinc_rx, t_arinc_tx, t_bit;
void ADK_UART_Init(void);

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

    ADK_UART_Init(); // LPUART3; configure pins in pin_mux.c

    scheduler_init(1000U);

    (void)task_register(&t_arinc_rx);
    (void)task_register(&t_arinc_tx);
    (void)task_register(&t_bit);
    (void)task_register(&t_led);
    (void)task_register(&t_button);
    (void)task_register(&t_telemetry);

    scheduler_run();
}
```

Validation steps 1. Connect ADK-8582 UART to EVKB LPUART3 (3.3 V levels). Configure pins with ConfigTools. 2. Power up; observe `[BIT]` lines every 100 ms and `link=0` until data arrives. 3. Drive ADK to emit ARINC words; `rx` counter increases; `link` flips to 1; `TX` messages appear every 50 ms. 4. Disconnect cable; within ~500 ms `link` returns to 0; tasks continue safely.

> For certification-style testing, record timestamps to compute jitter and compare to budgets.

---

10) From cooperative to preemptive (optional extension)

- Assign **BASEPRI** mask for critical sections; keep ISRs short.

- Use **SysTick** for periodic tick and **PendSV** for context switch if you later implement fixed-priority preemption. Each task would then need its own stack. This adds complexity (context save/restore) and increases verification scope under **DO-178C**; use only if latency demands justify it. For many DCU-style functions, **TTC** suffices with proper WCET control.

---

11) Best practices (Airbus-grade)

1. **Deterministic design**: Prefer **static tables** and **bounded** operations. Avoid heap; allocate all buffers at compile time.
2. **Tight ISRs**: Do the *minimum* in interrupts (e.g., push byte into ring buffer), schedule work to tasks.
3. **Time budgets**: Define WCET per task (measured with DWT cycle counter) and margin; document in a timing sheet.
4. **Jitter control**: Use release-time computation `next += period` (not `now + period`) to avoid drift; handle catch-up on overrun.
5. **Critical sections**: Use `__disable_irq()`/`__enable_irq()` sparingly or `__set_BASEPRI()` for fine-grained masking; keep sections very short.
6. **Memory placement**: Put hot code/data in **ITCM/DTCM** when necessary; be aware of caches on Cortex-M7 (enable/flush/invalidate correctly around DMA).
7. **Interfaces**: Separate drivers (`adk_uart.c`) from tasks (`arinc_tasks.c`) to keep clear abstraction boundaries for reviews.
8. **Diagnostics**: Count executions, overruns, last error. Telemetry at a low rate (e.g., 5 Hz) helps flight test without flooding links.
9. **MISRA-C:2012**: Follow rule set (no dynamic allocation, bounded loops, explicit casts, no hidden side effects). Keep a deviation log.
10. **Safety nets**: Implement `TASK_ERROR` policy — e.g., suspend and raise a maintenance message; never spin forever in an ISR.

---

12) Generic example vs. avionics use case summary

- **Generic**: LED, button, and telemetry tasks mapped to TTC with 1 ms tick. Demonstrates registration, release, dispatch, and overrun handling.
- **Avionics**: ARINC 429 via ADK-8582 over UART; RX (1 ms), TX (50 ms), BIT/HM (100 ms), heartbeat (1 s), telemetry (200 ms), with quiet-link detection and suspend/resume policy.

---

13) Exercise checklist

1. Build the TTC kernel (`scheduler.[ch]`), integrate with SysTick, and verify LED heartbeat.
2. Add telemetry and button; confirm periods and jitter visually.
3. Instrument WCET using DWT cycle counter; log max execution time per task for 60 s.
4. Wire ADK-8582 to LPUART3; enable ring buffer; verify RX/TX counters and quiet-link behavior.
5. Demonstrate **task_suspend/task_resume** by simulating a mode change (e.g., maintenance mode suspends TX).

Keep your project based on the SDK example tree so all board/clock/console code remains identical to NXP's reference, easing integration and certification evidence collection.
