Discrete I/O Fundamentals — Software Debouncing (i.MX RT1050 EVKB)

Why this matters in avionics

A discrete input is a binary signal (logical **0** or **1**) typically sourced by a switch, relay, proximity sensor, or an opto-isolated line. In aircraft systems, discrete lines drive flight-phase logic, interlocks, inhibits, configuration bits, and maintenance modes. Examples include **WOW** (Weight-On-Wheels), cargo-door latch microswitches, fire-handle positions, and air/ground relays. These sources are mechanical or relay-based and **they bounce**—their contacts chatter between 0 and 1 for microseconds to tens of milliseconds after a transition. If your software samples the raw line, the system can misinterpret a single actuation as multiple events. In a safety-critical context governed by **DO-178C** (Software Considerations in Airborne Systems and Equipment Certification) and **ARP4754A** (Guidelines for Development of Civil Aircraft and Systems), this is unacceptable.

Software debouncing converts a noisy, fast-changing raw input into a stable logical state and well-defined edges with guaranteed timing and bounded latency.

Physical intuition: what is "bounce"?

When a mechanical contact closes or opens, the metal surfaces approach with elasticity and microscopic asperities. They hit, rebound, and scrape until settling. The electrical resistance toggles rapidly, producing an apparent square-wave burst. Environmental extremes (temperature, vibration), supply noise, EMI/EMC conditions, and aging all affect bounce duration. For typical panel switches and microswitches, design for **5–20 ms** worst-case bounce; for relays or harsh environments, budget **up to 50 ms** unless component data justifies less.

Terminology (expanded at first use)

- **GPIO** — *General-Purpose Input/Output*. i.MX RT1050 exposes GPIO1..GPIO5 ports with per-pin direction and state.
- **PIT** — *Periodic Interrupt Timer*. Provides periodic interrupts at programmable rates; ideal for fixed-rate sampling.
- **ISR** — *Interrupt Service Routine*. Code executed in response to an interrupt.
- **WOW** — *Weight-On-Wheels* discrete indicating aircraft on ground.

- **EICAS** — *Engine Indication and Crew Alerting System* (or ECAM on Airbus: *Electronic Centralized Aircraft Monitor*). Used here as a representative alerting/monitoring consumer of discretes.

Design goals for a debouncer in airborne systems

1. **Determinism**: fixed sampling period, bounded processing time, bounded edge-to-decision latency.
2. **Monotonicity**: one real actuation → at most one debounced rising edge; one real release → at most one debounced falling edge.
3. **Configurability**: thresholds per input to match electrical characteristics and safety requirements.
4. **Traceability**: requirements↔design↔code↔tests under DO-178C; configuration captured and reviewable.
5. **Robustness**: works across temperature, electrical noise, and timing jitter.
6. **Observability**: counters and status exposed for BIT (Built-In Test) and maintenance.

Core software debouncing strategies

1) Time-based lockout (monostable)

On any detected change, start a timer (e.g., 10 ms). Ignore further changes until the timer expires; then re-sample to confirm. This is simple and low-CPU, but its **lockout** can delay recognition of a quick legitimate reversal.

2) Saturating up/down counter (a.k.a. "integrator") — **Recommended**

Sample at a fixed rate (e.g., 1 kHz). Maintain an N-bit counter per input. If the raw sample is 1, increment (up to max); if 0, decrement (down to 0). Declare the debounced state = 1 only when the counter reaches max; declare 0 only when it reaches 0. This behaves like a discrete low-pass filter, offers excellent immunity to sporadic glitches, and yields tunable latency via the counter size and sample period.

3) Majority vote over a sliding window

Keep the last *W* samples; the debounced state is the majority bit. Useful when noise is symmetric but has higher memory and shift overhead.

4) Edge-qualified state machine

Trigger on a detected edge, then verify that the input remains stable for **T_confirm** (e.g., 10 ms) before committing. Combines fast detection with a stability check; good for interrupt-first designs.

For i.MX RT1050 bare-metal training, we implement strategy (2) with a 1 ms PIT tick and show how to parameterize it to emulate (1) and (3).

Choosing sampling rate and thresholds

Let **T_s** be the sampling period and **N** the counter range. The worst-case additional latency to recognize a true edge is roughly **N × T_s**. If you budget **T_bounce = 10 ms** and want lock-solid immunity with ≤5 ms added latency, choose **T_s = 1 ms** and **N = 5** (≈5 ms). If measurements show occasional 15 ms bursts, push **N** to 15 or reduce **T_s** (at cost of CPU load). Always validate under environmental extremes.

Generic example (non-avionics): panel pushbutton toggles a status LED

A front-panel pushbutton wired to EVKB-i.MXRT1050 **SW8** (GPIO5_IO00) should toggle the **USER LED** (GPIO1_IO09) exactly once per press. Raw bounce should not create multiple toggles. We sample at 1 kHz and use a 5-count integrator.

Avionics use case example: Weight-On-Wheels (WOW) gating ground spoilers

A WOW discrete feeds a Flight Control Computer. On touchdown, ground spoilers should deploy **only** after WOW = 1 has been stable for **≥ 20 ms** to avoid spurious deployment from bounce or runway joints. In air (WOW = 0) the spoilers must be inhibited. For this lab:

- Map the EVKB **SW8** to WOW (pressed = on-ground = 1; released = in-air = 0).
- Drive a software “spoiler deploy command” when a **debounced rising edge** occurs and the state remains 1.
- Light the USER LED solid while WOW is 1. Flash at 2 Hz during the 20 ms stabilization window to show the qualifier timing.
- Log events over the debug console for observability (time-stamp, raw, filtered, edges).

This exercise demonstrates time qualification, edge extraction, and how a consumer subsystem (spoiler logic) must rely **only on debounced edges**, never raw input.

Implementation on EVKB-i.MXRT1050 (bare metal)

We build on the NXP MCUXpresso SDK drivers in this package: - `fsl_gpio.h` / `fsl_gpio.c` — GPIO read/write and configuration. - `fsl_pit.h` / `fsl_pit.c` — Periodic Interrupt Timer configuration and ISR support. - Board support: `board.h`, `pin_mux.h`, `clock_config.h`, and the example pattern `BOARD_InitHardware()` from `hardware_init.c` to enable clocks and the debug console.

> **Pins used** (from EVKB board definitions): - `BOARD_USER_BUTTON_GPIO = GPIO5`, `BOARD_USER_BUTTON_GPIO_PIN = 0` (SW8) - `BOARD_USER_LED_GPIO = GPIO1`, `BOARD_USER_LED_GPIO_PIN = 9` (User LED)

Debouncer data structure (integrator)

```
/* debouncer.h */
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    uint8_t count;          /* current integrator value (0..max) */
    uint8_t max;            /* threshold to accept a new state */
    uint8_t state;          /* current debounced state (0 or 1) */
    uint8_t last_state;     /* previous debounced state */
    bool    rose;           /* latched rising edge flag */
    bool    fell;           /* latched falling edge flag */
} debounce_t;

static inline void debounce_init(debounce_t *d, uint8_t max, uint8_t initial)
{
    d->count = initial ? max : 0;
    d->max = max;
    d->state = initial;
    d->last_state = initial;
    d->rose = false;
    d->fell = false;
}

/* Call at fixed rate (e.g., every 1 ms) with the RAW pin value */
static inline void debounce_update(debounce_t *d, uint8_t raw)
{
    /* Saturating up/down integrator */
    if (raw) {
        if (d->count < d->max) d->count++;
    } else {
        if (d->count > 0) d->count--;
    }

    /* Commit new debounced state only at the extremes */
    uint8_t new_state = d->state;
    if (d->count == d->max) new_state = 1U;
    else if (d->count == 0U) new_state = 0U;

    d->rose = (d->last_state == 0U && new_state == 1U);
    d->fell = (d->last_state == 1U && new_state == 0U);
    d->last_state = d->state = new_state;
}
```

Board + timer + GPIO glue (uses SDK drivers)

```
/* debounce_demo.c : EVKB-i.MXRT1050 + MCUXpresso SDK */
#include "fsl_debug_console.h"
#include "fsl_gpio.h"
#include "fsl_pit.h"
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"

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

/* PIT configuration (mirrors SDK example style) */
#ifndef DEMO_PIT_BASEADDR
#define DEMO_PIT_BASEADDR PIT
#endif
#ifndef DEMO_PIT_CHANNEL
#define DEMO_PIT_CHANNEL kPIT_Chnl_0
#endif
#ifndef PIT_IRQ_ID
#define PIT_IRQ_ID PIT_IRQn
#endif
#ifndef PIT_SOURCE_CLOCK
#define PIT_SOURCE_CLOCK CLOCK_GetFreq(kCLOCK_OscClk)
#endif

/* Make the button/LED symbols portable via board.h */
#define BTN_GPIO        BOARD_USER_BUTTON_GPIO
#define BTN_PIN         BOARD_USER_BUTTON_GPIO_PIN
#define LED_GPIO        BOARD_USER_LED_GPIO
#define LED_PIN         BOARD_USER_LED_GPIO_PIN

/* 1 kHz sampling → 1 ms per tick; integrator max=5  ⇒ ~5 ms qualification */
#define DEBOUNCE_TICK_HZ   1000U
#define INTEGRATOR_MAX     5U

static volatile bool s_tick = false;
static debounce_t s_wow; /* our single debounced input */

/* Simple time base for prints and LED blinking */
static volatile uint32_t s_ms = 0;

void PIT_IRQHandler(void)
```

```c
{
    /* Clear interrupt flag */
    PIT_ClearStatusFlags(DEMO_PIT_BASEADDR, DEMO_PIT_CHANNEL,
kPIT_TimerFlag);

    s_tick = true;
    s_ms++;
    SDK_ISR_EXIT_BARRIER;
}

static void gpio_init(void)
{
    /* Ensure GPIO clocks are on (GPIO1 for LED, GPIO5 for SW8) */
    CLOCK_EnableClock(kCLOCK_Gpio1);
    CLOCK_EnableClock(kCLOCK_Gpio5);

    /* LED as output */
    gpio_pin_config_t led_cfg = {kGPIO_DigitalOutput, 1}; /* inactive (OFF)
*/
    GPIO_PinInit(LED_GPIO, LED_PIN, &led_cfg);

    /* Button as input with pull-up disabled here - rely on board pin_mux */
    gpio_pin_config_t btn_cfg = {kGPIO_DigitalInput, 0};
    GPIO_PinInit(BTN_GPIO, BTN_PIN, &btn_cfg);
}

static void pit_init(void)
{
    pit_config_t pitCfg;
    PIT_GetDefaultConfig(&pitCfg);
    PIT_Init(DEMO_PIT_BASEADDR, &pitCfg);

    /* Set period for 1 kHz */
    uint32_t period_counts = PIT_SOURCE_CLOCK / DEBOUNCE_TICK_HZ;
    PIT_SetTimerPeriod(DEMO_PIT_BASEADDR, DEMO_PIT_CHANNEL, period_counts);

    PIT_EnableInterrupts(DEMO_PIT_BASEADDR, DEMO_PIT_CHANNEL,
kPIT_TimerInterruptEnable);
    EnableIRQ(PIT_IRQ_ID);
    PIT_StartTimer(DEMO_PIT_BASEADDR, DEMO_PIT_CHANNEL);
}

int main(void)
{
    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();
    gpio_init();
    pit_init();

    PRINTF("\r\n[Debounce demo] 1kHz, integrator=%u (≈%u ms)\r\n",
INTEGRATOR_MAX, INTEGRATOR_MAX);

    debounce_init(&s_wow, INTEGRATOR_MAX, 0);

    uint32_t last_print = 0;
    uint8_t last_led = 1; /* OFF */

    while (1) {
        if (s_tick) {
            s_tick = false;

            /* RAW sample: on EVKB SW8 the pressed level is logic 0 or 1
depending on routing; read and normalize */
            uint8_t raw = (uint8_t)GPIO_PinRead(BTN_GPIO, BTN_PIN);
            /* If your hardware inverts the sense, flip here: raw ^= 1; */

            debounce_update(&s_wow, raw);

            /* Drive LED with debounced state for visibility */
            if (s_wow.state != last_led) {
                last_led = s_wow.state;
                if (last_led) GPIO_PortClear(LED_GPIO, 1U << LED_PIN); /* LED
ON (active low) */
                else          GPIO_PortSet(LED_GPIO,   1U << LED_PIN); /* LED
OFF */
            }

            /* Use the edge latches exactly once */
            if (s_wow.rose) {
                PRINTF("%lu ms : DEBOUNCED RISE\r\n", (unsigned long)s_ms);
                s_wow.rose = false;
            }
            if (s_wow.fell) {
                PRINTF("%lu ms : DEBOUNCED FALL\r\n", (unsigned long)s_ms);
                s_wow.fell = false;
            }

            /* Periodic status */
            if ((s_ms - last_print) >= 500U) {
                last_print = s_ms;
                PRINTF("%lu ms : raw=%u count=%u state=%u\r\n",
                       (unsigned long)s_ms, raw, s_wow.count, s_wow.state);
            }
        }
    }
}
```

Notes: - The EVKB user LED is active-low (logic 0 turns it on). We explicitly use `GPIO_PortClear`/`GPIO_PortSet` accordingly. - `BOARD_InitBootPins()` relies on `pin_mux.c` generated for your SDK project. Ensure SW8 is configured as GPIO5_IO00 with the correct pull state; if unsure, add an external pull-up/down on your breadboard switch. - If your particular board wiring yields an inverted sense (pressed = 0), flip `raw` as shown.

Mapping this to the avionics WOW use case

To enforce a **≥20 ms** stability requirement before arming ground spoilers:

1. Set `INTEGRATOR_MAX = 20` (with a 1 kHz tick → 20 ms).
2. Gate any spoiler logic on the **debounced rising edge** (`s_wow.rose`) and the current debounced state (`s_wow.state == 1`).
3. Add a small blinking indicator during the qualification window by reading `s_wow.count` (0..20). When `0 < count < 20`, toggle the LED every 250 ms to indicate “validating”; once it hits 20, turn solid.

Because your consumer logic subscribes **only to debounced edges**, a 5–15 ms contact chatter will not deploy spoilers prematurely.

Parameterizing the debouncer to emulate other strategies

- **Time-based lockout**: after an edge, ignore updates by forcing the counter to stay at 0 or max for *L* ticks before allowing it to move again.
- **Majority vote**: store a bitfield of the last *W* `raw` samples; every tick, popcount and compare to `W/2` to set `state`.

The provided `debounce_update()` is the best general default for discrete inputs subject to both chatter and occasional single-sample spikes.

Verification on target

1. Build the project with the MCUXpresso SDK, enabling modules: **GPIO**, **PIT**, **DEBUG CONSOLE**.
2. Start from the SDK **PIT driver example** (`boards/evkbimxrt1050/driver_examples/pit`) or **GPIO input_interrupt example** and replace `main()` with the demo above, keeping `board.c`, `hardware_init.c`, `pin_mux.c`, and `clock_config.c` from the example.
3. Open a terminal on the EVKB’s debug UART at the baud set by `BOARD_InitDebugConsole()` (typically 115200 bps). Press and hold SW8; observe

one **DEBOUNCED RISE** print, solid LED, and stable `state=1` even if you tap or wiggle the switch. Release SW8; observe one **DEBOUNCED FALL**.
4. Stress: quick double-presses faster than 5 ms should produce **one** edge at most. Increase `INTEGRATOR_MAX` to 20 and verify the designed 20 ms latency.

For formal verification, capture the raw SW8 pin on a logic analyzer in parallel with a GPIO-toggled test point on each debounced edge, and measure the edge-to-decision latency under temperature extremes.

Best practices (avionics-oriented)

- Treat debouncing as **requirements-driven**. Write an HLR (High-Level Requirement): “The WOW discrete shall be recognized only after 20 ms of continuous assertion; deassertion shall be recognized only after 20 ms of continuous negation. The edge-to-decision latency shall be ≤ 25 ms.” Trace this to LLR, code, and tests.
- **Synchronize to the CPU clock domain**. External discretes can violate setup/hold at the GPIO sampling flop. Use the fixed-rate PIT sampling (software synchronization). If you ever use GPIO interrupts directly, add a two-flop synchronizer in an FPGA or qualify in software before acting.
- **Measure your switch** population. Do not assume 5 ms; verify the worst case with your exact hardware and harness.
- **Bound CPU usage**. Keep the ISR constant-time; do only sampling and debouncer updates there. Push logging/printing to the main loop.
- **Avoid long ISRs**. Printing (`PRINTF`) in ISRs can stall the system; never do this in certification-bound code.
- **Fail-safe defaults**. On startup, seed the debouncer to the physically safe state (e.g., WOW=air) until proven otherwise for N ticks.
- **Configuration control**. Make the thresholds part of a configuration database with checksums and version control.
- **Diagnostics**. Expose `count`, `state`, `rose/fell` via a maintenance/health page; include a BIT that detects stuck-at lines.
- **EMI/EMC**. Software debouncing complements, not replaces, proper input conditioning (RC filters, Schmitt buffers, opto-isolators, surge protection to RTCA/DO-160).

Optional extension: driving an ARINC 429 stub for lab realism

If you have the **ADK-8582** ARINC 429 interface board connected over **UART** to the EVKB, extend the demo so that each **debounced** WOW edge transmits an ARINC 429 label representing air/ground state (e.g., label 053 octal, a common discrete label in some installations). The EVKB sends a simple UART command to the ADK-8582 to put the label into its transmit queue. This shows how **only debounced edges** should trigger avionics bus outputs. (The ADK-8582 specifics are board-vendor dependent; keep the UART framing and label mapping in a separate module.)

What you hand in (for the training check-off)

1. Your `debouncer.h` and `debounce_demo.c` compiled and running on EVKB-i.MXRT1050.
2. A short capture of the debug log demonstrating single edges per actuation at integrator values 5 and 20.
3. A photo or scope trace showing raw bounce vs. debounced edge for at least one transition.
4. If attempting the ARINC 429 extension, the UART log of label transmissions aligned to debounced edges.

Appendix: tuning cheat-sheet

- Start with **1 kHz** sampling; set `INTEGRATOR_MAX = T\_qual_ms`.
- If the environment is harsher, increase `INTEGRATOR_MAX` first; only then consider faster sampling to keep latency down.
- For actions that must be **fast on release**, you can use asymmetric thresholds: separate `max_assert` and `max_deassert` counts.
- When multiple discretes arrive together (e.g., gear down + WOW), de-correlate sampling jitter by reading all raw inputs in the **same tick**.