Discrete I/O Driver Design on i.MX RT1050-EVKB — Integrating I/O into a Bare-Metal Scheduler (Airbus Training)

Learning goals

By the end of this module you will be able to: - Explain what “discrete I/O” means in avionics and why timing/robustness matter. - Configure i.MX RT1050 pads and GPIO blocks (IOMUXC + GPIO) for deterministic discrete inputs/outputs. - Build a minimal, deterministic bare-metal scheduler driven by SysTick and integrate periodic and event-driven I/O tasks. - Implement production-grade debouncing, edge/level detection, latching, and event queuing. - Produce safe, fail-silent outputs with known power-up states and interlocks. - (Extension) Publish discrete state over UART to an external ARINC 429 adapter (ADK-8582) as a stepping stone toward label generation.

> **Hardware & SDK used throughout**
> • NXP i.MX RT1050-EVKB MCU board (Cortex-M7)
> • NXP MCUXpresso SDK for EVKB-i.MXRT1050 (provided)
> • On-board USER LED (via `BOARD_USER_LED_*` macros) and USER BUTTON SW8 (via `BOARD_USER_BUTTON_*` macros).
> • For the ARINC 429 extension: EVKB-i.MXRT1050 connected to **ADK-8582** via LPUART1 (3-wire UART).

---

1. Fundamentals

1.1 What is discrete I/O?

Discrete input: a binary signal representing the logical state of an external condition (e.g., door closed, landing gear down, weight-on-wheels).
Discrete output: a binary drive provided by the MCU to signal or command external hardware (e.g., lamp on, relay coil drive, inhibit line).

In avionics, discretes are often **active-low**, electrically conditioned off-board, and **safety-relevant**. Software must account for polarity, pull-ups, filtering, and deterministic timing.

1.2 Determinism & certification vocabulary (first use expansions)

- **LRU (Line Replaceable Unit)**: field-replaceable avionics box.
- **CDD (Component Development Data)** and **SRS (Software Requirements Specification)**: artifacts under **DO-178C (Software Considerations in Airborne**

Systems and Equipment Certification).

- **DAL (Design Assurance Level)**: A through E criticality classification.
  We are not implementing a certified design here; however, we adopt patterns (deterministic timing, traceability, defensive coding) that align with DO-178C expectations.

1.3 i.MX RT1050 pin path overview

- **IOMUXC (I/O Multiplexer Controller)** selects the peripheral function for a physical pad and sets pad electrical properties (pull-up/down, keeper, hysteresis, drive strength, slew rate).

- **GPIO** peripheral provides direction control, data registers, and interrupts (edge or level) per pin.
  Implication: Correct pad config is as important as the GPIO mode. A "floating" input without pull/keeper will cause chattering and false edges.

---

2. Architecture: Discrete I/O driver inside a bare-metal system

2.1 Layered view

1. **HAL (Hardware Abstraction Layer)**: IOMUXC pad config + GPIO init (SDK drivers).

2. **Driver (DISCRETE)**: Debounce, polarity, edge/level detect, latching, timestamps, event queue.

3. **Scheduler**: Runs periodic tasks (e.g., every 5 ms for inputs, 10 ms for outputs) and dispatches events from ISRs.

4. **Application**: Implements system logic and avionics use cases.

2.2 Scheduler requirements for I/O

- **Stable sampling period** (e.g., 5 ms for de-bounce integrators).

- **Bounded ISR time**: ISRs raise events or snapshot timestamps; heavy work runs in tasks.

- **Monotonic timebase**: SysTick at 1 ms is adequate for most cockpit-rate discretes.

---

3. Configuring pads and GPIO (SDK-centric)

We always use the SDK's `fsl_iomuxc.h` and `fsl_gpio.h` plus the board layer macros.
Typical steps: 1. **Enable pads & mux** with `IOMUXC_SetPinMux(PAD, 0U);` and set pad electricals with `IOMUXC_SetPinConfig(PAD, configVal);`
2. **Initialize GPIO** with `gpio_pin_config_t` and `GPIO_PinInit()` (direction, default logic, interrupt mode).
3. For inputs, prefer keeper/pull and hysteresis enabled; for outputs, set drive strength and slew appropriately.

> **Board convenience macros**:
> - `BOARD_USER_LED_GPIO`, `BOARD_USER_LED_GPIO_PIN` (LED on GPIO1_IO09; active-low on EVKB).
> - `BOARD_USER_BUTTON_GPIO`, `BOARD_USER_BUTTON_GPIO_PIN`, `BOARD_USER_BUTTON_IRQ` (SW8 on GPIO5_IO00; interrupt on `GPIO5_Combined_0_15_IRQn`).
> Use these macros instead of hard-coding pins to survive board revisions.

---

4. Bare-metal scheduler design

We implement a cooperative, tick-driven scheduler using SysTick at 1 ms.

- **Tick source**: `SysTick_Config(SystemCoreClock / 1000U);`
- **Task table**: each task has a period, next-release time, and a function pointer.
- **Determinism**: no blocking inside tasks; each task finishes within its budget.

Task classes used here: - `DiscreteIn_Task_5ms()`: samples & debounces inputs.
- `DiscreteOut_Task_10ms()`: applies requested outputs, handles lamp-test overrides.
- `App_Task_10ms()`: avionics logic.
- `EventPump_Task_1ms()`: empties the ISR-to-task event queue.

---

5. Discrete input driver design

5.1 Features

- Configurable **polarity** (active-low or active-high).
- **Debounce integrator** (up/down saturating counter) with a settable time constant in samples.

- **Edge detect** with millisecond timestamps (rise/fall).
- **Latching** and **sticky faults** (until explicitly cleared).
- ISR support for hardware edges, but **work deferred** to the 5 ms task.

5.2 Debounce algorithm (integrator)

For each input we maintain an unsigned counter. On each sample: - If raw signal == TRUE, counter++ up to `COUNT_MAX`; else counter– down to 0.
- Debounced state = TRUE when `counter >= THRESHOLD`; FALSE when `counter == 0`.
This avoids time-window race conditions of "N consecutive samples."

5.3 Event model

- **EVENT_RISE / EVENT_FALL**: Posted when debounced state changes.
- **EVENT_GLITCH** (optional): Posted when the ISR detects an edge but the debounced state doesn't change within a window (diagnostic).

---

6. Discrete output driver design

6.1 Features

- **Polarity**: active-low or active-high mapping.
- **Failsafe defaults** at power-up (e.g., de-energize relays).
- **Interlocks**: optional requires-true discretes before enabling (e.g., weight-on-wheels before deploy).
- **Lamp-test**: global override forcing outputs on for ground checks.

6.2 Timing

Outputs are **requested** by the application at any time; the 10 ms output task applies them in a deterministic batch, minimizing pin wiggle and accounting for interlocks.

---

7. Generic example (board-level): Button toggles LED with robust debouncing

Goal: SW8 toggles USER LED. LED is active-low, so OFF=logic '1', ON=logic '0'. We implement: - 5 ms debounce, edge detection, event queue.
- Press = toggle LED state; long press (>1 s) = lamp-test mode.

---

8. Avionics use-case examples

8.1 Weight-On-Wheels (WOW) discrete consolidation

- **Context**: Two independent WOW microswitches feed a Single Board Computer (this EVKB) that shares a consolidated WOW status with adjacent LRUs (Line Replaceable Units).

- **Requirement**: assert `WOW_TRUE` if **both** channels are TRUE for ≥40 ms; de-assert on **either** channel FALSE for ≥40 ms. Timestamp each transition.

- **Usage**: Application inhibits thrust-reverser test in flight, and sends WOW status to a 429 transmitter LRU.

8.2 Fire handle lamp drive & latch

- **Context**: An Engine Fire Handle discrete output should illuminate when EICAS (Engine Indication and Crew Alerting System) signals FIRE, and **remain latched** until maintenance clears it, unless the lamp-test override is active.

- **Requirement**: enforce safe default (lamp off) at power-up; respect lamp-test.

8.3 Doors closed monitoring with disagree detection

- **Context**: Two "Door Closed" discretes from independent sensors.

- **Requirement**: present a single "Doors Closed" discrete to another LRU; raise a sticky fault if sensors disagree for more than 500 ms.

Each example maps cleanly onto our input/output drivers and scheduler periods.

---

9. Best practices (summary)

1. **Configure pads, not just GPIO**: enable keeper/pull and hysteresis for inputs; set drive and slew for outputs.

2. **Deterministic timing**: periodic sampling (5 ms) and bounded ISR time.
3. **Polarity & defaults**: encode at the driver boundary; never scatter active-low knowledge across the app.
4. **Debounce with integrators**: avoids edge races and handles noise.
5. **Separate request vs. apply** for outputs to allow atomic, rate-limited updates.
6. **Event queues** crossing ISR→task boundaries; never do heavy work in ISRs.
7. **Self-test & lamp-test hooks** wired through the driver, not ad-hoc.
8. **Measure**: verify periods and jitter with GPIO scope pins and logic analyzer.
9. **Document assumptions**: voltage levels, external pull-ups, active polarity, time constants, safety defaults.

---

10. Hands-on Exercises (with SDK-based solutions)

> **Project tip**: Start from `boards/evkbimxrt1050/driver_examples/gpio/led_output` in the SDK. Replace the `main.c` with the files below or add new source files to a clean MCUXpresso project that already includes `board.c/.h`, `pin_mux.c/.h`, and `clock_config.c/.h`.

Exercise 1 — Minimal bare-metal scheduler & precise LED blink

Task: Implement a 1 ms SysTick, schedule a 100 ms LED toggle task; measure blink period accuracy.
Acceptance: Average period 200 ms (on/off), jitter < ±2 ms when idle.

Solution (files)

scheduler.h

```
#ifndef SCHEDULER_H
#define SCHEDULER_H
#include <stdint.h>

typedef void (*task_fn_t)(void);

typedef struct {
    task_fn_t fn;
    uint32_t period_ms;
    uint32_t next_ms;
```

```
} sched_task_t;

void SCHED_Init(uint32_t tick_hz);
void SCHED_AddTask(task_fn_t fn, uint32_t period_ms);
void SCHED_RunOnce(void);
uint32_t SCHED_Millis(void);

#endif
```

scheduler.c

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

#define MAX_TASKS 8
static volatile uint32_t g_ms;
static sched_task_t g_tasks[MAX_TASKS];
static uint32_t g_task_count;

void SysTick_Handler(void) { g_ms++; }

void SCHED_Init(uint32_t tick_hz)
{
    g_ms = 0; g_task_count = 0;
    (void)SysTick_Config(SystemCoreClock / tick_hz);
}

uint32_t SCHED_Millis(void) { return g_ms; }

void SCHED_AddTask(task_fn_t fn, uint32_t period_ms)
{
    if (g_task_count < MAX_TASKS) {
        g_tasks[g_task_count].fn = fn;
        g_tasks[g_task_count].period_ms = period_ms;
        g_tasks[g_task_count].next_ms = SCHED_Millis() + period_ms;
        g_task_count++;
    }
}

void SCHED_RunOnce(void)
{
    uint32_t now = SCHED_Millis();
    for (uint32_t i = 0; i < g_task_count; i++) {
        if ((int32_t)(now - g_tasks[i].next_ms) >= 0) {
            g_tasks[i].fn();
            g_tasks[i].next_ms += g_tasks[i].period_ms; // catch-up,
fixed-period
        }
```

```
    }
}
```

main.c (Exercise 1)

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

static void led_task_100ms(void)
{
    // USER_LED_TOGGLE uses board macros (active-low LED)
    USER_LED_TOGGLE();
}

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

    // LED pin is already configured by board/pin_mux in the SDK example
    USER_LED_OFF(); // set to a known state

    SCHED_Init(1000U); // 1 ms tick
    SCHED_AddTask(led_task_100ms, 100U);

    while (1) {
        SCHED_RunOnce();
        __NOP();
    }
}
```

Exercise 2 — Discrete input driver with debouncing and event queue (SW8 → LED)

Task: Sample SW8 every 5 ms, debounce with an integrator, post PRESS/RELEASE events; toggle LED on PRESS. Hold >1 s to enable a lamp-test state that forces LED on.

Solution (files)

discrete.h

```
#ifndef DISCRETE_H
#define DISCRETE_H
#include <stdbool.h>
#include <stdint.h>
```

```h
typedef enum { DISCRETE_ACTIVE_LOW=0, DISCRETE_ACTIVE_HIGH=1 }
discrete_polarity_t;

typedef struct {
    GPIO_Type *base;
    uint32_t pin;
    discrete_polarity_t pol;
    uint8_t count;      // integrator
    uint8_t thresh;     // threshold to declare TRUE
    uint8_t count_max;  // saturation
    bool debounced;
    bool latched;       // optional sticky
    uint32_t last_change_ms;
} discrete_in_t;

void DISCRETE_IN_Init(discrete_in_t *d, GPIO_Type *base, uint32_t pin,
                      discrete_polarity_t pol, uint8_t thresh, uint8_t
count_max);
void DISCRETE_IN_Sample(discrete_in_t *d, uint32_t now_ms);
bool DISCRETE_IN_Get(const discrete_in_t *d);

// Simple lock-free ring buffer for ISR→task events
typedef enum { EV_NONE=0, EV_PRESS, EV_RELEASE } io_event_t;
void EV_Post(io_event_t ev, uint32_t t_ms);
bool EV_Pop(io_event_t *ev, uint32_t *t_ms);

#endif
```

discrete.c

```c
#include "discrete.h"
#include "fsl_gpio.h"

#define EVQ_SZ 16
static volatile io_event_t ev_q[EVQ_SZ];
static volatile uint32_t ev_t[EVQ_SZ];
static volatile uint8_t ev_head, ev_tail;

static bool raw_read(GPIO_Type *base, uint32_t pin)
{
    // SDK returns pin level as 0/1
    return GPIO_PinRead(base, pin) ? true : false;
}

static bool apply_polarity(bool raw, discrete_polarity_t pol)
{
    return (pol == DISCRETE_ACTIVE_LOW) ? (!raw) : raw;
}
```

```
void DISCRETE_IN_Init(discrete_in_t *d, GPIO_Type *base, uint32_t pin,
                      discrete_polarity_t pol, uint8_t thresh, uint8_t
count_max)
{
    d->base = base; d->pin = pin; d->pol = pol;
    d->count = 0; d->thresh = thresh; d->count_max = count_max;
    d->debounced = false; d->latched = false; d->last_change_ms = 0U;
}

void DISCRETE_IN_Sample(discrete_in_t *d, uint32_t now_ms)
{
    bool raw = raw_read(d->base, d->pin);
    bool val = apply_polarity(raw, d->pol);

    if (val) { if (d->count < d->count_max) d->count++; }
    else { if (d->count > 0) d->count--; }

    bool prev = d->debounced;
    if (!prev && d->count >= d->thresh) {
        d->debounced = true; d->last_change_ms = now_ms; EV_Post(EV_PRESS,
now_ms);
    } else if (prev && d->count == 0) {
        d->debounced = false; d->last_change_ms = now_ms; EV_Post(EV_RELEASE,
now_ms);
    }
}

bool DISCRETE_IN_Get(const discrete_in_t *d) { return d->debounced; }

void EV_Post(io_event_t ev, uint32_t t_ms)
{
    uint8_t next = (uint8_t)((ev_head + 1U) % EVQ_SZ);
    if (next != ev_tail) { ev_q[ev_head] = ev; ev_t[ev_head] = t_ms; ev_head
= next; }
}

bool EV_Pop(io_event_t *ev, uint32_t *t_ms)
{
    if (ev_tail == ev_head) return false;
    *ev = ev_q[ev_tail]; *t_ms = ev_t[ev_tail];
    ev_tail = (uint8_t)((ev_tail + 1U) % EVQ_SZ); return true;
}
```

main.c (Exercise 2)

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_gpio.h"
#include "fsl_iomuxc.h"
```

```c
#include "scheduler.h"
#include "discrete.h"

static discrete_in_t sw8;  // SW8 on BOARD_USER_BUTTON_*
static bool lamp_test;
static uint32_t lamp_test_start;

static void io_init(void)
{
    // Button pin already muxed in board/pin_mux for the SDK examples.
    // Ensure it is input with interrupt disabled (we'll poll in 5 ms task
and optionally enable IRQ later)
    gpio_pin_config_t inCfg = { kGPIO_DigitalInput, 0U, kGPIO_NoIntmode };
    GPIO_PinInit(BOARD_USER_BUTTON_GPIO, BOARD_USER_BUTTON_GPIO_PIN, &inCfg);

    // LED: ensure known state
    USER_LED_OFF(); // logical off (drives high on active-low hardware)

    // Our logical polarity: SW8 is active-low on EVKB → DISCRETE_ACTIVE_LOW
    DISCRETE_IN_Init(&sw8, BOARD_USER_BUTTON_GPIO,
BOARD_USER_BUTTON_GPIO_PIN,
                     DISCRETE_ACTIVE_LOW, /*thresh*/3, /*count_max*/5);
}

static void task_inputs_5ms(void)
{
    DISCRETE_IN_Sample(&sw8, SCHED_Millis());
}

static void task_events_1ms(void)
{
    io_event_t ev; uint32_t t;
    while (EV_Pop(&ev, &t)) {
        if (ev == EV_PRESS) {
            USER_LED_TOGGLE();
            lamp_test = true; lamp_test_start = t; // start long-press timer
        } else if (ev == EV_RELEASE) {
            lamp_test = false;
        }
    }
}

static void task_outputs_10ms(void)
{
    if (lamp_test && (SCHED_Millis() - lamp_test_start) > 1000U) {
        // Long press → force LED ON regardless of app state
        GPIO_PortClear(BOARD_USER_LED_GPIO, 1U << BOARD_USER_LED_GPIO_PIN);
    }
}
```

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

    io_init();

    SCHED_Init(1000U);
    SCHED_AddTask(task_events_1ms, 1U);
    SCHED_AddTask(task_inputs_5ms, 5U);
    SCHED_AddTask(task_outputs_10ms, 10U);

    while (1) { SCHED_RunOnce(); __NOP(); }
}
```

> **Test**: Observe stable debouncing (no chatter). Verify long-press lamp-test behavior.

Exercise 3 — Avionics WOW consolidator with UART publish to ARINC 429 adapter (ADK-8582)

Task: Model two WOW inputs (here we use SW8 as channel A and a second jumper input as channel B). Consolidate with 40 ms debounce on each channel and **logical AND** for the final `WOW_TRUE`. On any change, publish a short ASCII status frame over **LPUART1** to the external ADK-8582 board (e.g., `"WOW:1\n"` or `"WOW:0\n"`).

> **Note**: The exact ADK-8582 UART command protocol may differ. We send a simple placeholder token; adjust the command format per your ADK-8582 documentation to drive a specific ARINC 429 label/SDI as needed.

Solution (delta files)

wow.h

```c
#ifndef WOW_H
#define WOW_H
#include <stdbool.h>
#include <stdint.h>
#include "discrete.h"

typedef struct {
    discrete_in_t chA;
    discrete_in_t chB;
    bool wow;              // consolidated
    uint32_t last_change_ms;
} wow_t;
```

```
void WOW_Init(wow_t *w, GPIO_Type *a_base, uint32_t a_pin,
              GPIO_Type *b_base, uint32_t b_pin, discrete_polarity_t pol);
void WOW_Task_5ms(wow_t *w, uint32_t now_ms);

#endif
```

wow.c

```
#include "wow.h"
#include "scheduler.h"
#include "fsl_gpio.h"

void WOW_Init(wow_t *w, GPIO_Type *a_base, uint32_t a_pin,
              GPIO_Type *b_base, uint32_t b_pin, discrete_polarity_t pol)
{
    DISCRETE_IN_Init(&w->chA, a_base, a_pin, pol, 8, 10); // ~40 ms at 5 ms
period
    DISCRETE_IN_Init(&w->chB, b_base, b_pin, pol, 8, 10);
    w->wow = false; w->last_change_ms = 0U;
}

void WOW_Task_5ms(wow_t *w, uint32_t now_ms)
{
    DISCRETE_IN_Sample(&w->chA, now_ms);
    DISCRETE_IN_Sample(&w->chB, now_ms);
    bool newWow = DISCRETE_IN_Get(&w->chA) && DISCRETE_IN_Get(&w->chB);
    if (newWow != w->wow) { w->wow = newWow; w->last_change_ms = now_ms;
EV_Post(newWow ? EV_PRESS : EV_RELEASE, now_ms); }
}
```

uart_publish.c (LPUART1 send helper)

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

void UART_Init_115200(void)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = 115200U;
    cfg.enableTx = true; cfg.enableRx = true;
    LPUART_Init(LPUART1, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));
}

static void uart_send(const char *s)
{
    while (*s) { LPUART_WriteBlocking(LPUART1, (const uint8_t*)s, 1U); s++; }
}

void UART_SendWOW(bool wow)
```

```
{
    uart_send(wow ? "WOW:1\n" : "WOW:0\n");
}
```

main.c (Exercise 3)

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_gpio.h"
#include "scheduler.h"
#include "discrete.h"
#include "wow.h"

void UART_Init_115200(void);
void UART_SendWOW(bool wow);

static wow_t wow;
static bool last_pub;

static void io_init(void)
{
    // Configure two inputs. We reuse SW8 as chA and choose another free GPIO
pin for chB.
    gpio_pin_config_t inCfg = { kGPIO_DigitalInput, 0U, kGPIO_NoIntmode };

    GPIO_PinInit(BOARD_USER_BUTTON_GPIO, BOARD_USER_BUTTON_GPIO_PIN, &inCfg);
// chA

    // Example chB: use GPIO1_IO10 (adjacent pad to LED on many EVKBs) —
update to an actual pin available on your board.
    // Ensure this pin is routed as GPIO in pin_mux.c using MCUXpresso Config
Tools.
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_10_GPIO1_IO10, 0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_10_GPIO1_IO10, 0x10B0U);
    GPIO_PinInit(GPIO1, 10U, &inCfg);

    WOW_Init(&wow,
             BOARD_USER_BUTTON_GPIO, BOARD_USER_BUTTON_GPIO_PIN,
             GPIO1, 10U,
             DISCRETE_ACTIVE_LOW);

    USER_LED_OFF();
}

static void task_inputs_5ms(void)
{
    WOW_Task_5ms(&wow, SCHED_Millis());
}
```

```
static void task_events_1ms(void)
{
    io_event_t ev; uint32_t t;
    while (EV_Pop(&ev, &t)) {
        if (ev == EV_PRESS || ev == EV_RELEASE) {
            bool now = (ev == EV_PRESS);
            if (now != last_pub) { last_pub = now; UART_SendWOW(now); }
            // Mirror consolidated WOW onto LED (LED ON when WOW TRUE)
            if (now) { GPIO_PortClear(BOARD_USER_LED_GPIO, 1U <<
BOARD_USER_LED_GPIO_PIN); }
            else { GPIO_PortSet(BOARD_USER_LED_GPIO, 1U <<
BOARD_USER_LED_GPIO_PIN); }
        }
    }
}

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

    UART_Init_115200();
    io_init();

    SCHED_Init(1000U);
    SCHED_AddTask(task_events_1ms, 1U);
    SCHED_AddTask(task_inputs_5ms, 5U);

    while (1) { SCHED_RunOnce(); __NOP(); }
}
```

Bench validation 1. Tie `GPIO1_IO10` high/low with a jumper to emulate WOW channel B.
2. Toggle SW8 for channel A.
3. Observe UART frames on LPUART1 TX (pins per `pin_mux.c`) to the ADK-8582; LED mirrors consolidated WOW.

---

11. Advanced topics (for further study)

- **GPIO interrupts vs. polling**: For very slow discretes, polling at 5 ms is sufficient; for time stamping, enable edge interrupts (`GPIO_PinSetInterruptConfig`) and only capture timestamps in the ISR. Mask/unmask in the driver when lamp-test or maintenance modes run.
- **Glitch capture**: Maintain a short-window glitch counter when ISRs report edges that do not survive debouncing; use for harness diagnostics.

- **Output rate limiting**: If outputs drive external loads, enforce minimum on/off times to avoid relay chatter.
- **Time-deterministic logging**: Push events to a pre-allocated buffer; flush in a low-priority task to avoid interfering with I/O periods.

---

12. Traceability checklist (mapping to avionics-style requirements)

- **D-I-001**: The driver shall provide configurable input polarity and a deterministic debounce algorithm.
- **D-I-002**: The driver shall generate events on debounced transitions with timestamps.
- **D-O-001**: The driver shall guarantee a defined power-up output state.
- **S-K-001**: The scheduler shall provide a 1 ms system tick and run tasks at configured periods.
- **APP-WOW-001**: WOW consolidation shall require both channels TRUE ≥40 ms and de-assert on either FALSE ≥40 ms.

---

13. Common pitfalls and how to avoid them

- **Floating inputs**: Always configure pull/keeper and hysteresis; otherwise you will see random events.
- **Polarity confusion**: Encode active-low at the driver; expose only logical TRUE/FALSE to the app.
- **Doing work in ISRs**: Keep ISRs short; defer to tasks via event queues.
- **Unbounded tasks**: A busy task will starve 5 ms sampling and break debounce guarantees. Budget and measure.
- **Not measuring**: Use a spare GPIO as a scope pin toggled at the start/end of tasks to visualize jitter.

---

14. What to hand-in (lab deliverables)

- Source tree with `scheduler.*`, `discrete.*`, `wow.*`, and modified `main.c`, `pin_mux.c`.
- Logic analyzer capture showing: 1 ms tick timing, 5 ms input sampling, and LED response.
- UART capture to ADK-8582 of `WOW:0/1` frames on transitions.

---

15. Appendix: Pad configuration cheat-sheet (EVKB typical)

- `IOMUXC_SetPinMux(<PAD>, 0U);`
- `IOMUXC_SetPinConfig(<PAD>, 0x10B0U);` → typical keeper+pull, 100 MHz speed, medium drive; adjust per harness.
- Inputs: prefer `HYS=ON`, pull-up/down according to external wiring.
- Outputs: set `DSE` (drive) and `SRE` (slew) to meet EMC without over-driving.
