Discrete I/O Driver Design Using Interrupt-Driven I/O

---

1) Learning outcomes

By the end of this module, you will be able to: 1. Explain what a **discrete I/O** is, how it's represented electrically, and why **interrupt-driven I/O** is preferred over polling in most avionics contexts. 2. Configure i.MX RT1050 GPIO pins (IOMUXC + pad control) as inputs/outputs and enable **edge/level** interrupts using the MCUXpresso SDK (`fsl_gpio.h`). 3. Design and implement a small, reusable **Discrete I/O (DIO) driver** with: - Edge selection (rising/falling/both), - Debounce/deglitch filtering using PIT, - Safe power-up default states for outputs, - Event callbacks with timestamps and an event ring buffer. 4. Validate timing, latency, and fault behavior (shorts, bounce, wire-off) with structured tests. 5. Apply **avionics best practices** under DO-178C (Software Considerations in Airborne Systems and Equipment Certification), DO-254 (Design Assurance Guidance for Airborne Electronic Hardware), ARP4754A (Guidelines for Development of Civil Aircraft and Systems), and ARP4761 (Safety Assessment) contexts.

---

2) Fundamentals

Discrete I/O: A binary signal represented by two valid electrical levels (e.g., 0/1, LOW/HIGH). In avionics, discretes often follow specific electrical standards (e.g., open-collector with pull-ups, 28V discrete with conditioning, opto-isolated modules). On the EVKB we use the microcontroller's direct **GPIO (General-Purpose Input/Output)** pins for learning; in an LRU these would usually be behind input protection and level conditioning.

Interrupt-driven I/O: The microcontroller configures a peripheral (GPIO) to assert an **interrupt** when a selected event occurs (edge/level). The CPU services the event in a short **ISR (Interrupt Service Routine)** instead of continuously polling. Benefits: low latency, low CPU load, time-stamped events, and deterministic response.

Debounce / deglitch: Real mechanical switches bounce and wiring can pick noise; without filtering, the ISR may fire many times for one logical transition. **Debounce** ensures the input is stable for a minimum time (e.g., 10–20 ms) before it is accepted.

Safe state: The output level the system must assume at power-up and during faults to avoid unsafe actuation (e.g., keep a valve closed, keep a relay de-energized). Always design outputs to a known **Fail-Safe** or **Fail-Operational** strategy depending on the system's FHA (Functional Hazard Assessment).

Latency: Time from the external event to software reaction. It includes pad/port latency, interrupt latency, ISR length, and any debounce delay. For the i.MX RT1050 at 600 MHz, raw latency to ISR entry is very small (µs-scale) but application design (filtering, printing) dominates perceived response.

---

3) Hardware and SDK orientation (EVKB-i.MXRT1050)

- **User Button (SW8)**: Routed to **GPIO5, pin 0**. Interrupt line: **`GPIO5_Combined_0_15_IRQn`**. Board alias in SDK: `BOARD_USER_BUTTON_GPIO` / `BOARD_USER_BUTTON_GPIO_PIN` / `BOARD_USER_BUTTON_IRQ` / `BOARD_USER_BUTTON_IRQ_HANDLER`.
- **User LED (D9)**: Routed to **GPIO1, pin 9**. Board alias in SDK: `BOARD_USER_LED_GPIO` / `BOARD_USER_LED_GPIO_PIN`. Logical on/off are inverted by hardware (LED on when the GPIO drives low in the EVKB reference design); board macros handle this.
- We will use MCUXpresso SDK drivers shipped with your SDK zip:
  - `fsl_gpio.h` for GPIO,
  - `fsl_iomuxc.h` for pin mux/pad control (usually wrapped by generated `pin_mux.c`),
  - `fsl_pit.h` for debounce timing,
  - `fsl_debug_console.h` for UART prints.
- **Board init**: Use `BOARD_InitHardware()` from NXP examples to bring up clocks, pins, and console.

We deliberately base all code on the **provided EVKB SDK code base** so you can drop files into a stock MCUXpresso SDK project and build.

---

4) Driver design overview

We implement a small **DIO driver** with these responsibilities: 1. Describe each discrete **channel** (GPIO base, pin number, active level, pull configuration, interrupt mode, debounce time). 2. Initialize the hardware (pin mux and pad config assumed provided by `BOARD_InitHardware()` or your `pin_mux.c`). 3. Set up GPIO input with **edge-triggered interrupts** and a **PIT-based debounce timer**. 4. Provide a **callback interface** to deliver debounced events (pin, new logical state, timestamp, reason). 5. Provide **output APIs** with safe default levels and atomic updates. 6. Handle ISR hygiene: fast entry/exit, flag clear, minimal work in ISR, `SDK_ISR_EXIT_BARRIER`.

Interrupt model on i.MX RT1050

- Each GPIO port exposes **combined** interrupt lines (e.g., `GPIO5_Combined_0_15_IRQn` for pins 0..15 and another for 16..31). The ISR must

read/clear the port's flags and service any matching pins. We illustrate with **GPIO5[0]** (SW8) but the code supports many pins.

---

5) Implementation (SDK-based, bare-metal)

Files you add: `dio.h`, `dio.c`, and a sample `main.c` (or integrate into your board example). All APIs and includes are straight from the SDK, so the code compiles inside a standard EVKB project.

5.1 `dio.h`

```
#ifndef DIO_H
#define DIO_H

#include <stdint.h>
#include <stdbool.h>
#include "fsl_gpio.h"
#include "fsl_pit.h"

#ifdef __cplusplus
extern "C" {
#endif

// Logical level definition (post-polarity)
typedef enum { DIO_LEVEL_LOW = 0, DIO_LEVEL_HIGH = 1 } dio_level_t;

typedef enum {
    DIO_EDGE_NONE = 0,
    DIO_EDGE_RISING,
    DIO_EDGE_FALLING,
    DIO_EDGE_BOTH
} dio_edge_t;

// Forward decl for callback context
struct dio_channel_s;

typedef void (*dio_callback_t)(struct dio_channel_s *ch, dio_level_t
new_level, uint32_t timestamp_us);

typedef struct dio_channel_s {
    GPIO_Type *base;            // e.g., GPIO5
    uint32_t   pin;             // e.g., 0
    bool       active_high;     // false if active low (maps pad level to
logical HIGH)
    dio_edge_t edge;            // which edge(s) generate events
    uint32_t   debounce_us;     // debounce window in microseconds
    dio_callback_t cb;          // user callback (debounced event)
    // internal
```

```
    volatile bool irq_armed;    // ISR arms a debounce check
    volatile uint32_t t_irq_us; // time of last IRQ edge
    volatile dio_level_t last_level; // last debounced level
} dio_channel_t;

// Global init for PIT timebase (1 MHz tick) and NVIC, call once at startup.
void dio_timebase_init(void);
uint32_t dio_timebase_get_us(void);

// Initialize a discrete input channel with interrupts+debounce.
void dio_input_init(dio_channel_t *ch);

// Configure an output channel and drive safe level at startup.
void dio_output_init(GPIO_Type *base, uint32_t pin, bool active_high,
dio_level_t safe_level);
void dio_write(GPIO_Type *base, uint32_t pin, bool active_high, dio_level_t
level);
dio_level_t dio_read(GPIO_Type *base, uint32_t pin, bool active_high);

#ifdef __cplusplus
}
#endif

#endif // DIO_H
```

5.2 dio.c

```
#include "dio.h"
#include "fsl_common.h"
#include "fsl_debug_console.h"

// === Timebase: PIT channel 0 at 1 MHz (1 us tick) ===
#define DIO_PIT            PIT
#define DIO_PIT_CH         kPIT_Chnl_0
#define DIO_PIT_IRQn       PIT_IRQn
#define DIO_PIT_IRQHandler PIT_IRQHandler

static volatile uint32_t s_time_us = 0;

void dio_timebase_init(void)
{
    pit_config_t cfg;
    PIT_GetDefaultConfig(&cfg);
    PIT_Init(DIO_PIT, &cfg);
    // Run free-running at 1 MHz (period = 1 us). We reload every 1 us and
count in ISR.
    uint32_t src = CLOCK_GetFreq(kCLOCK_OscClk);
    PIT_SetTimerPeriod(DIO_PIT, DIO_PIT_CH, USEC_TO_COUNT(1U, src));
    PIT_EnableInterrupts(DIO_PIT, DIO_PIT_CH, kPIT_TimerInterruptEnable);
```

```
    EnableIRQ(DIO_PIT_IRQn);
    PIT_StartTimer(DIO_PIT, DIO_PIT_CH);
}

uint32_t dio_timebase_get_us(void) { return s_time_us; }

void DIO_PIT_IRQHandler(void)
{
    PIT_ClearStatusFlags(DIO_PIT, DIO_PIT_CH, kPIT_TimerFlag);
    s_time_us++;
    SDK_ISR_EXIT_BARRIER; // Prevent spurious re-entry on fast cores
}

// === Helpers ===
static gpio_interrupt_mode_t map_edge(dio_edge_t e)
{
    switch (e) {
    case DIO_EDGE_RISING:  return kGPIO_IntRisingEdge;
    case DIO_EDGE_FALLING: return kGPIO_IntFallingEdge;
    case DIO_EDGE_BOTH:    return kGPIO_IntRisingOrFallingEdge;
    default:               return kGPIO_NoIntmode;
    }
}

static inline dio_level_t level_from_pad(bool pad_high, bool active_high)
{
    bool logical = active_high ? pad_high : !pad_high;
    return logical ? DIO_LEVEL_HIGH : DIO_LEVEL_LOW;
}

// === Input init (interrupt + debounce) ===
void dio_input_init(dio_channel_t *ch)
{
    // Configure pin as digital input, interrupt per edge setting
    gpio_pin_config_t cfg = {kGPIO_DigitalInput, 0, map_edge(ch->edge)};
    GPIO_PinInit(ch->base, ch->pin, &cfg);

    // Read initial pad and set debounced level
    ch->last_level = level_from_pad(GPIO_PinRead(ch->base, ch->pin), ch-
>active_high);
    ch->irq_armed = false;

    // Enable per-pin interrupt at port level
    GPIO_PortEnableInterrupts(ch->base, (1U << ch->pin));
}

// === Output API ===
void dio_output_init(GPIO_Type *base, uint32_t pin, bool active_high,
dio_level_t safe_level)
```

```
{
    gpio_pin_config_t cfg = {kGPIO_DigitalOutput, 0, kGPIO_NoIntmode};
    GPIO_PinInit(base, pin, &cfg);
    // Drive safe state
    bool pad_high = active_high ? (safe_level == DIO_LEVEL_HIGH) :
(safe_level == DIO_LEVEL_LOW);
    if (pad_high) {
        GPIO_PortSet(base, (1U << pin));
    } else {
        GPIO_PortClear(base, (1U << pin));
    }
}

void dio_write(GPIO_Type *base, uint32_t pin, bool active_high, dio_level_t
level)
{
    bool pad_high = active_high ? (level == DIO_LEVEL_HIGH) : (level ==
DIO_LEVEL_LOW);
    if (pad_high) {
        GPIO_PortSet(base, (1U << pin));
    } else {
        GPIO_PortClear(base, (1U << pin));
    }
}

dio_level_t dio_read(GPIO_Type *base, uint32_t pin, bool active_high)
{
    return level_from_pad(GPIO_PinRead(base, pin), active_high);
}

// === GPIO5 combined ISR example ===
// In your app, install this handler under the board's IRQ name for GPIO5
0..15.
extern dio_channel_t g_sw8; // Declared in main.c for the demo

void GPIO5_Combined_0_15_IRQHandler(void)
{
    uint32_t flags = GPIO_PortGetInterruptFlags(GPIO5);
    // Service SW8 (GPIO5 pin 0) if set
    if (flags & (1U << 0)) {
        // Clear first to avoid retrigger
        GPIO_PortClearInterruptFlags(GPIO5, (1U << 0));

        // Arm debounce window for the demo channel
        if (g_sw8.debounce_us == 0) {
            // No debounce: emit immediately
            bool pad = GPIO_PinRead(g_sw8.base, g_sw8.pin);
            dio_level_t lvl = level_from_pad(pad, g_sw8.active_high);
            if (lvl != g_sw8.last_level) {
```

```
                g_sw8.last_level = lvl;
                if (g_sw8.cb) g_sw8.cb(&g_sw8, lvl, dio_timebase_get_us());
            }
        } else {
            g_sw8.irq_armed = true;
            g_sw8.t_irq_us = dio_timebase_get_us();
        }
    }

    SDK_ISR_EXIT_BARRIER;
}

// === Debounce check: called from your main loop ===
// This keeps ISR very short. The main loop periodically calls this helper.
void dio_poll_debounce(dio_channel_t *ch)
{
    if (!ch->irq_armed) return;
    uint32_t now = dio_timebase_get_us();
    if ((now - ch->t_irq_us) >= ch->debounce_us) {
        ch->irq_armed = false;
        bool pad = GPIO_PinRead(ch->base, ch->pin);
        dio_level_t lvl = level_from_pad(pad, ch->active_high);
        if (lvl != ch->last_level) {
            ch->last_level = lvl;
            if (ch->cb) ch->cb(ch, lvl, now);
        }
    }
}
```

5.3 `main.c` — Generic demo (SW8 interrupt toggles LED with debounce)

```
#include "fsl_debug_console.h"
#include "board.h"
#include "app.h"       // For BOARD_* aliases from SDK examples
#include "dio.h"

// Demo channel: EVKB SW8 on GPIO5[0], active high, both edges, 20 ms
debounce
static void sw8_callback(dio_channel_t *ch, dio_level_t new_level, uint32_t
t_us)
{
    PRINTF("[SW8] %s at %u us\r\n", new_level == DIO_LEVEL_HIGH ? "HIGH" :
"LOW", (unsigned)t_us);
    // Example action: toggle LED on a rising edge only
    if (new_level == DIO_LEVEL_HIGH) {
        USER_LED_TOGGLE();
    }
}

dio_channel_t g_sw8 = {
```

```
    .base = BOARD_USER_BUTTON_GPIO,        // GPIO5
    .pin  = BOARD_USER_BUTTON_GPIO_PIN,    // 0
    .active_high = true,                   // SW8 reads 1 when pressed
    .edge = DIO_EDGE_BOTH,
    .debounce_us = 20000,                  // 20 ms
    .cb = sw8_callback,
};

int main(void)
{
    BOARD_InitHardware();      // clocks, pins, debug console
    dio_timebase_init();       // 1 MHz timebase

    LED_INIT();                // Board macro drives known safe state for LED

    // Configure SW8 interrupt input
    dio_input_init(&g_sw8);

    // Enable NVIC line for the user button (from board/app.h)
    NVIC_SetPriority(BOARD_USER_BUTTON_IRQ, 5); // Medium priority
    EnableIRQ(BOARD_USER_BUTTON_IRQ);

    PRINTF("Discrete I/O driver demo: SW8 -> LED (debounced)\r\n");

    while (1) {
        // Periodically service debounce (keeps ISR minimal)
        dio_poll_debounce(&g_sw8);
        // Do other work...
    }
}
```

Notes: - This pattern keeps the GPIO ISR extremely short: it clears the flag and arms a debounce window. The **main loop** performs the stable-state check and calls the user callback. In an RTOS this would be a deferred ISR (D-ISR) via queue; in bare-metal we poll a small flag. - If an input requires **no debounce**, set `debounce_us = 0` and the ISR will invoke the callback immediately. - For multiple inputs on the same port, extend the ISR to iterate `flags` and maintain one `dio_channel_t` per pin.

---

6) Generic example explained

Scenario: A maintenance technician presses the front-panel pushbutton **SW8** to cycle a function. We need reliable event detection without multiple triggers from switch bounce, with a visible LED action and a UART log line.

Walk-through: 1. `BOARD_InitHardware()` initializes clocks, IOMUXC, and the UART console (115200 bps by default). The LED macro sets a **safe power-up** state. 2.

`dio_timebase_init()` starts the **PIT** at 1 MHz for microsecond timestamps and debounce windows. 3. `dio_input_init(&g_sw8)` configures `GPIO5[0]` as **digital input** with **both-edge interrupts**. The port's **combined IRQ** is enabled via `EnableIRQ(BOARD_USER_BUTTON_IRQ)`. 4. On any edge, `GPIO5_Combined_0_15_IRQHandler()` clears the flag and **arms** a debounce check (`g_sw8.irq_armed = true;`). 5. The **main loop** calls `dio_poll_debounce()`, which after 20 ms re-reads the pad and, if changed, invokes `sw8_callback()` with a timestamp.

Why this matters: The ISR path is deterministic and short; the debounce is time-based and explicit; the output (LED) demonstrates actuation under control of a debounced discrete.

---

7) Avionics use-case: Weight-On-Wheels (WOW) discrete

Context: WOW (Weight-On-Wheels) informs many systems (thrust-reverser logic, ground spoilers, configuration warnings). Spurious transitions are safety-relevant. Assume an LRU monitors a conditioned WOW discrete presented as a 3.3 V logic-level input to the MCU.

Requirements: - Latch transitions with timestamps.
- Minimum stable time **50 ms** (air/ground logic usually slow-changing).
- On **multiple transitions within 10 ms**, flag a **FAULT** (wiring noise) and ignore until stable for 100 ms.
- Provide a **safe default** at power-up: treat as **AIR** (not on wheels) until a stable ground is seen.

Implementation sketch with our driver: - Configure a `dio_channel_t` for the WOW input: active-high, `DIO_EDGE_BOTH`, debounce 50 ms. - In the callback, implement a small state machine: if the time since the previous event < 10 ms, set FAULT and inhibit further processing until 100 ms stable. - Log each accepted transition with a **timestamp** from `dio_timebase_get_us()`; this provides traceability for maintenance and certification artifacts.

Why interrupt-driven: Polling at (say) 1 ms either wastes CPU or still misses short spikes. Interrupts give immediate capture and then software decides what is valid.

---

8) Advanced topics

8.1 Edge vs. level interrupts

- **Edge**: Best for event capture; requires software to read the current level after debounce.

- **Level**: Can re-enter ISR until the level changes; only use when you must hold service until the condition clears.

8.2 NVIC priorities

- Assign GPIO interrupt priorities lower than hard real-time buses (e.g., DMA complete) but higher than background. Example: set to priority **5** (where 0 is highest on Cortex-M). Never print from ISR at high rates.

8.3 Concurrency & memory ordering

- Shared flags between ISR and main must be `volatile`.
- Keep ISRs short; defer work.
- Use `SDK_ISR_EXIT_BARRIER` in NXP SDK to avoid re-entry on fast core/slow bus corner cases.

8.4 Input protection in real LRUs

- Real aircraft discretes are often **28 V** and come through protection (TVS, dividers, opto). Software should assume **inversion** may occur; that's why our API has `active_high` per channel.

8.5 Time-stamping alternatives

- Instead of PIT, you may enable the **DWT (Data Watchpoint and Trace) cycle counter** for sub-microsecond timestamps. PIT is used here because it's an SDK-supported peripheral already present in your SDK tree.

---

9) Best practices (avionics-oriented)

1. **Define electrical assumptions in software**: pull-ups, active level, safe level. Keep them in one place (channel table) for easy system safety review.
2. **Initialize outputs to safe states early** (before enabling interrupts).
3. **Never allocate in ISR / never block**: no malloc, no printf loops. Defer using flags or ring buffers.
4. **Debounce every human-operated input**; log raw vs. debounced if safety assessment requires.
5. **Bound ISR time** and prove it: code review + measurement.

6. **Fault management**: detect oscillation, stuck-at, wire-off via plausibility checks and timeouts.

7. **Configurability**: keep debounce and polarity in non-volatile configuration if allowed by your development assurance level and change control.

8. **Traceability**: link each requirement (e.g., “WOW debounce 50 ms”) to code lines and tests for DO-178C objectives.

9. **Deterministic builds**: lock SDK version and toolchain; include generated `pin_mux.c/h` in configuration control.

---

10) Hands-on exercises (with solutions)

Exercise A — Basic interrupt input toggles LED

Goal: On **SW8 press**, toggle the user LED exactly once per press (no bounce).

Steps: 1. Create a new MCUXpresso SDK project for **evkbimxrt1050**, bare-metal.
2. Add `dio.h/.c` and `main.c` above.
3. Build and run; observe UART prints and LED toggles once per press.

Solution: Use the provided `main.c` with `debounce_us=20000` and callback toggling LED on `DIO_LEVEL_HIGH`.

Observations: If you set `debounce_us=0`, you will see multiple prints per press due to bounce.

Exercise B — Both-edge event logger with timestamps

Goal: Log **both edges** of SW8 with microsecond timestamps and count events.

Modify: In `sw8_callback()` maintain counters of rising/falling edges, print the counts every 10 events. Keep debounce at 10–20 ms.

Solution snippet:

```
static uint32_t rises=0, falls=0;
static void sw8_callback(dio_channel_t *ch, dio_level_t new_level, uint32_t
t_us)
{
    if (new_level == DIO_LEVEL_HIGH) rises++; else falls++;
    PRINTF("SW8 %s @%u us (R:%u F:%u)\r\n", new_level?"HIGH":"LOW",
(unsigned)t_us, rises, falls);
}
```

Exercise C — WOW use-case filter

Goal: Implement the WOW fault rules from Section 7: ignore transitions occurring <10 ms apart, set FAULT, and require 100 ms stable before clearing FAULT.

Hints: Keep `static` state inside the callback: last_time_us, in_fault, fault_cleared_at_us. Drive an LED ON while in FAULT.

Solution sketch:

```
static bool in_fault = false;
static uint32_t last_event_us = 0, fault_exit_earliest_us = 0;
static void wow_callback(dio_channel_t *ch, dio_level_t new_level, uint32_t
t_us)
{
    if (in_fault) {
        if (t_us >= fault_exit_earliest_us) {
            in_fault = false; USER_LED_OFF(); PRINTF("WOW: FAULT
CLEARED\r\n");
        } else {
            return; // Still in fault holdoff
        }
    }
    if ((t_us - last_event_us) < 10000) { // <10 ms
        in_fault = true; USER_LED_ON();
        fault_exit_earliest_us = t_us + 100000; // 100 ms stable window
        PRINTF("WOW: FAULT due to chatter\r\n");
        return;
    }
    last_event_us = t_us;
    PRINTF("WOW: %s\r\n", new_level==DIO_LEVEL_HIGH?"GROUND":"AIR");
}
```

Exercise D — Multi-channel inputs on one port

Goal: Add two more inputs on GPIO5 pins (e.g., pins 1 and 2 if available on your carrier/breadboard) and verify the ISR correctly services multiple flags in one combined interrupt.

Solution approach: Maintain an array of `dio_channel_t*` for the serviced port; in the ISR, iterate `flags` bits and arm the corresponding channel's debounce. This pattern scales to dozens of discrete inputs while using only two NVIC lines per port.

11) Testing & verification procedure

1. **Functional:** Press/release SW8 ten times; confirm exactly ten rising-edge logs and ten LED toggles.

2. **Bounce characterization:** Temporarily set `debounce_us=0` and observe multiple ISR firings. Then increase to 5 ms, 10 ms, 20 ms and measure prints per press. Select the minimum value that guarantees a single event on your hardware.

3. **Latency:** With `debounce_us=0`, connect a logic analyzer to SW8 and LED pin; measure time from edge to LED toggle (µs scale).

4. **Fault injection:** Simulate chatter by rapidly tapping; confirm Exercise C fault logic sets/clears FAULT with the specified timings.

5. **Power-up safety:** Ensure LED (acting as an actuator proxy) powers up in the declared safe state before any interrupts are enabled.

---

12) Design review checklist (use during peer review)

- Inputs
  - ☐ For each input: polarity, pull-up/down, debounce time, safety impact documented.
  - ☐ Edge selection justified (rising/falling/both).
  - ☐ Debounce deferred out of ISR; ISR clears flags and exits quickly.
- Outputs
  - ☐ Safe default for each output documented and enforced before ISR enable.
  - ☐ Writes are atomic (read-modify-write avoided on shared registers or appropriately masked).
- Timing
  - ☐ Interrupt priority consistent with system criticality.
  - ☐ Worst-case ISR time bounded; no prints in ISR.
- Faults
  - ☐ Chatter/wire-off/stuck-at strategies implemented where required.
  - ☐ Event logging with timestamp for maintenance tracing.
- Process
  - ☐ Code style and naming aligned with project standards.
  - ☐ SDK version locked; pin mux generated files under configuration control.

---

13) Where this leads to

- **Output drivers** with PWM/hold-off and interlocks.

- **Self-test (BIT — Built-In Test)** of discrete inputs using internal pull-ups and loopbacks.

- **Integration with ARINC 429** exercises: discrete lines to enable/disable transmit, health lines, and maintenance discretes in the **ADK-8582** lab (via UART) — those topics will use the **i.MX RT1050-EVKB + ADK-8582** pairing per course plan.

---

14) Appendix: Porting notes

- If you generate `pin_mux.c/h` with MCUXpresso ConfigTools, keep the signal names consistent and call `BOARD_InitPins()` early in `main()` or inside `BOARD_InitHardware()`.
- If you change the timebase, keep the debounce math in microseconds and adjust `dio_timebase_get_us()` accordingly.
- For higher event rates, replace the `PRINTF` in callbacks with a **ring buffer** and a background UART task to meet timing.
