Discrete I/O Fundamentals: GPIO configuration and electrical characteristics (EVKB-i.MX RT1050)

---

1) What “discrete I/O” means in avionics

A **discrete** is a binary signal—physically an electrical line that is asserted (logic “1” / true) or de-asserted (logic “0” / false). In aircraft systems, discretes announce states such as **WOW (Weight-On-Wheels)**, **FLT/GRD (Flight/Ground)**, **FIRE DETECT**, **DOOR CLOSED**, or **APU (Auxiliary Power Unit) READY**. Software interacts with discretes via **GPIO (General-Purpose Input/Output)** pins. GPIOs can be configured as inputs or outputs with features like internal pull-ups/pull-downs, Schmitt triggers (hysteresis), open-drain mode, drive strength, and slew rate control.

On the NXP i.MX RT1050, each physical **pad** is routed by a **pin-mux** to one of several peripheral functions (GPIO, UART, SPI, etc.). After selecting **GPIO** in the mux, you configure pad **electrical characteristics** (pulls, slew, drive strength, hysteresis) so logic levels are reliable, immune to noise, and compatible with surrounding hardware.

For safety-critical avionics, **DAL (Design Assurance Level)** software (per DO-178C) and hardware (per DO-254) expect discretes to be engineered for determinism, diagnosability (**BIT—Built-In Test**), and tolerance to **EMI/EMC (Electromagnetic Interference/Compatibility)** and **ESD (Electrostatic Discharge)** per DO-160.

---

2) Terminology and definitions (first use expanded)

- **GPIO (General-Purpose Input/Output):** Configurable digital pin used as a binary input or output.
- **VIH / VIL (Input High/Low Voltage):** Minimum voltage recognized as logic “1” (VIH) and maximum recognized as logic “0” (VIL).
- **VOH / VOL (Output High/Low Voltage):** Guaranteed output voltages while driving high/low under load.
- **Drive strength (mA):** Sourcing/sinking capability at valid logic levels.
- **Slew rate:** Edge speed; slower edges reduce EMI.
- **Schmitt trigger (hysteresis):** Adds separate rising/falling thresholds to reject noise and slow edges.
- **Pull-up / Pull-down:** Weak resistors to rails that define a default state for otherwise floating signals.
- **Push-pull:** Output stage can drive both high and low.

- **Open-drain:** Output can only pull low; a pull-up defines the high level—ideal for wired-OR and level translation.
- **Pad / Mux:** “Pad” is the electrical cell at the pin; “mux” selects which internal function connects to it.
- **Debounce:** Filtering that removes mechanical chatter and narrow spikes.
- **Wired-OR / wired-AND:** Multiple open-drain outputs share a line with a pull-up; any device pulling low dominates.

---

3) Electrical characteristics to get right

Logic levels & supplies

The EVKB user I/O domain is **3.3 V** (LVCMOS). Typical thresholds (always confirm in the data sheet) are ~0.6–0.7×VDDIO for **VIH** and ~0.3–0.4×VDDIO for **VIL**. Never drive GPIOs above their bank supply or below ground (beyond allowed diode/leakage currents) and observe absolute maximum ratings.

Inputs

- **Pull-ups/pull-downs:** Prevent floating inputs. Use pull-up for **active-low** signals; pull-down for **active-high**.
- **Schmitt trigger:** Essential for long harnesses or RC-filtered discretes.
- **Glitch filtering:** In software (sampling/debounce) or hardware where available.

Outputs

- **Push-pull** for on-board logic at same voltage.
- **Open-drain** when sharing a line, implementing fail-safe wired-OR, or level shifting (with proper external circuitry).
- **Drive & slew:** Use the **lowest** drive and **slowest** slew that still meets timing.

Protection & aircraft interfacing

Aircraft systems are often **28 V** nominal. Never connect these directly to MCU pins. Use resistor dividers, **TVS (Transient Voltage Suppressor)**, RC filters, and isolation (**optocoupler** or digital isolator) as required. For outputs, use open-drain transistor/MOSFET stages into aircraft discrete networks and add EMI/ESD components per DO-160.

---

4) EVKB-IMXRT1050 SDK building blocks you’ll use

- **Pin mux & pad control:** `fsl_iomuxc.h` → `IOMUXC_SetPinMux()` and `IOMUXC_SetPinConfig()` (or tool-generated `BOARD_InitPins()` in `pin_mux.c`).

- **GPIO driver:** `fsl_gpio.h` →
  - `gpio_pin_config_t` (direction, default value, interrupt mode),
  - `GPIO_PinInit(GPIOx, pin, &cfg)`, `GPIO_PinWrite()`, `GPIO_PinRead()`,
  - `GPIO_PortSetInterruptConfig()`, `GPIO_PortClearInterruptFlags()`.
- **Board helpers:** `board.c/h`, `fsl_debug_console.h` for `PRINTF()`, `BOARD_InitBootPins()`, `BOARD_InitBootClocks()`.

Exact LED & Button macros from your SDK's `boards/evkbimxrt1050/.../board.h`:

```
#define LOGIC_LED_ON  (0U)
#define LOGIC_LED_OFF (1U)
#define BOARD_USER_LED_GPIO      GPIO1
#define BOARD_USER_LED_GPIO_PIN  9U           // Pad: GPIO_AD_B0_09 →
GPIO1_IO09

#define BOARD_USER_BUTTON_GPIO         GPIO5  // Pad: SNVS_WAKEUP →
GPIO5_IO00
#define BOARD_USER_BUTTON_GPIO_PIN     (0U)
#define BOARD_USER_BUTTON_IRQ          GPIO5_Combined_0_15_IRQn
#define BOARD_USER_BUTTON_IRQ_HANDLER  GPIO5_Combined_0_15_IRQHandler
#define BOARD_USER_BUTTON_NAME         "SW8"
```

> The pad names above matter for **electrical** settings (pulls, hysteresis, open-drain). `BOARD_InitBootPins()` usually configures these for you; when you need to override, use the pad constants shown in Section 8 (Lab 4).

---

5) Generic example—requirements → pad configuration → GPIO code

Requirement: "Provide a debounced active-low input with noise immunity and a push-pull output to drive an on-board LED."

Hardware choices: Input uses **pull-up** + **Schmitt trigger**; software debounces. Output uses **push-pull** at the lowest drive/slowest slew that still meets timing.

Software path: 1) Call `BOARD_InitBootPins()` (and clocks/console) to set mux & baseline pad config. 2) Initialize GPIO pins via `GPIO_PinInit()`. 3) If using interrupts on the input, configure `GPIO_PortSetInterruptConfig()` and enable its NVIC IRQ. 4) Read inputs (poll or ISR) and write outputs.

---

6) Realistic avionics use case—WOW (Weight-On-Wheels) with diagnostics

Scenario: An LRU (Line-Replaceable Unit) ingests a **WOW** discrete via an isolated interface, filters bounce/spikes, **votes** samples for truth, timestamps edges for

maintenance logs to the **CMC (Central Maintenance Computer)**, and asserts a **FLT/GRD (Flight/Ground)** status discrete on an open-drain output so multiple LRUs can share the line (**wired-OR**). Power-up and periodic **BIT (Built-In Test)** detect stuck-at faults.

Engineering choices: Input with internal pull-up + hysteresis + 5 ms debounce; output as open-drain (external pull-up on the backplane). Software records both raw and filtered states with timestamps.

---

7) Hands-on labs on EVKB-IMXRT1050 (using SDK macros from `board.h`)

> All labs assume a bare-metal project with `BOARD_InitBootPins()`, `BOARD_InitBootClocks()`, and `BOARD_InitDebugConsole()` available. The exact LED and button pins/macros are taken from the SDK, so you **do not** hardcode GPIO banks/pins.

Lab 0 — Project skeleton

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

int main(void) {
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();
    PRINTF("Discrete I/O lab start\r\n");

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

Lab 1 — Output: blink the user LED with SDK macros

Because `board.h` defines the LED logic and location, you can use either the convenience macros or the GPIO driver directly. Here's a clean pattern using the driver **and** the logic constants:

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

static void led_init(void) {
    gpio_pin_config_t led_cfg = {
        .direction = kGPIO_DigitalOutput,
        .outputLogic = LOGIC_LED_OFF,
        .interruptMode = kGPIO_NoIntmode
    };
```

```
    GPIO_PinInit(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, &led_cfg);
}

static inline void led_set(int on) {
    GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, on ?
LOGIC_LED_ON : LOGIC_LED_OFF);
}

static inline void led_toggle(void) {
    GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN,
                  GPIO_PinRead(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN)
^ 0x1);
}

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

    led_init();
    PRINTF("LED blink using board.h macros (active-%s)\r\n",
           LOGIC_LED_ON ? "high" : "low");

    while (1) {
        led_toggle();
        SDK_DelayAtLeastUs(250000U, SystemCoreClock); // 250 ms
    }
}
```

Electrical reasoning: the LED is local and shares 3.3 V, so **push-pull** is fine. If edges are noisy in EMI testing, reduce drive strength and set slow slew in the pad config (see Lab 4).

Lab 2 — Input: read **SW8** with pull-up, Schmitt trigger, interrupt, and debounce

We'll use the **exact** button macros from `board.h`:

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

static volatile uint8_t s_buttonStable; // 1=not pressed (since SW8 is
active-low), 0=pressed

static void button_init(void) {
    gpio_pin_config_t in_cfg = {
        .direction = kGPIO_DigitalInput,
```

```
        .outputLogic = 0U,
        .interruptMode = kGPIO_IntRisingOrFallingEdge
    };
    GPIO_PinInit(BOARD_USER_BUTTON_GPIO, BOARD_USER_BUTTON_GPIO_PIN,
&in_cfg);
    GPIO_PortClearInterruptFlags(BOARD_USER_BUTTON_GPIO, 1UL <<
BOARD_USER_BUTTON_GPIO_PIN);
    GPIO_PortSetInterruptConfig(BOARD_USER_BUTTON_GPIO,
BOARD_USER_BUTTON_GPIO_PIN,
                                kGPIO_IntRisingOrFallingEdge);
    NVIC_ClearPendingIRQ(BOARD_USER_BUTTON_IRQ);
    NVIC_EnableIRQ(BOARD_USER_BUTTON_IRQ);
}

void BOARD_USER_BUTTON_IRQ_HANDLER(void) {
    GPIO_PortClearInterruptFlags(BOARD_USER_BUTTON_GPIO, 1UL <<
BOARD_USER_BUTTON_GPIO_PIN);
    // Simple time-based debounce: re-sample after 5 ms
    SDK_DelayAtLeastUs(5000U, SystemCoreClock);
    uint8_t raw = GPIO_PinRead(BOARD_USER_BUTTON_GPIO,
BOARD_USER_BUTTON_GPIO_PIN); // 0 when pressed
    s_buttonStable = raw ? 1U : 0U;
    __DSB(); __ISB();
}

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

    // Ensure SNVS mux clock is on if you override pin mux manually elsewhere
    // CLOCK_EnableClock(kCLOCK_IomuxcSnvs);

    // LED for feedback
    gpio_pin_config_t led_cfg = { kGPIO_DigitalOutput, LOGIC_LED_OFF,
kGPIO_NoIntmode };
    GPIO_PinInit(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, &led_cfg);

    button_init();
    PRINTF("SW8 interrupt+debounce: press to toggle LED\r\n");

    uint8_t last = 1U;
    while (1) {
        if (s_buttonStable != last) {
            last = s_buttonStable;
            if (s_buttonStable == 0U) { // pressed
                GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN,
                              GPIO_PinRead(BOARD_USER_LED_GPIO,
BOARD_USER_LED_GPIO_PIN) ^ 0x1);
```

```
            }
        }
    }
}
```

Electrical reasoning: the input pad should have **pull-up** and **hysteresis** enabled (see Lab 4 for explicit pad settings). Software adds a deterministic 5 ms debounce.

Lab 3 — Toward WOW: edge timestamps + 3-of-5 voting (using SW8 as a stand-in)

This skeleton shows timestamping edges and producing a robust "truth" via majority voting. Replace SW8 with your isolated WOW input in a real LRU.

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

static volatile uint32_t s_edgeTimeUs;
static volatile uint8_t  s_wowTruth;

static void timers_init_1MHz(void) {
    CLOCK_EnableClock(kCLOCK_Gpt1);
    gpt_config_t cfg;
    GPT_GetDefaultConfig(&cfg);
    cfg.clockSource = kGPT_ClockSource_Periph;
    cfg.divider = CLOCK_GetFreq(kCLOCK_PerClk) / 1000000U; // 1 MHz
    cfg.enableFreeRun = true;
    GPT_Init(GPT1, &cfg);
    GPT_StartTimer(GPT1);
}

static inline uint32_t micros(void) { return GPT_GetCurrentTimerCount(GPT1);
}

// Use board macros for SW8
void BOARD_USER_BUTTON_IRQ_HANDLER(void) {
    GPIO_PortClearInterruptFlags(BOARD_USER_BUTTON_GPIO, 1UL <<
BOARD_USER_BUTTON_GPIO_PIN);
    s_edgeTimeUs = micros();

    // 3-of-5 voting over the last 5 ms samples
    static uint8_t window[5] = {1,1,1,1,1};
    static uint8_t idx = 0;

    // SW8 is active-low: 0 = asserted → treat as WOW=1 when pressed
    uint8_t raw = GPIO_PinRead(BOARD_USER_BUTTON_GPIO,
```

```
BOARD_USER_BUTTON_GPIO_PIN);
    uint8_t logical = (raw == 0U) ? 1U : 0U;

    window[idx] = logical; idx = (idx + 1) % 5;
    uint8_t sum = window[0] + window[1] + window[2] + window[3] + window[4];
    s_wowTruth = (sum >= 3U) ? 1U : 0U;

    __DSB(); __ISB();
}

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

    timers_init_1MHz();

    // Configure LED and SW8
    gpio_pin_config_t led_cfg = { kGPIO_DigitalOutput, LOGIC_LED_OFF,
kGPIO_NoIntmode };
    GPIO_PinInit(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, &led_cfg);

    gpio_pin_config_t in_cfg = { kGPIO_DigitalInput, 0U,
kGPIO_IntRisingOrFallingEdge };
    GPIO_PinInit(BOARD_USER_BUTTON_GPIO, BOARD_USER_BUTTON_GPIO_PIN,
&in_cfg);
    GPIO_PortClearInterruptFlags(BOARD_USER_BUTTON_GPIO, 1UL <<
BOARD_USER_BUTTON_GPIO_PIN);
    GPIO_PortSetInterruptConfig(BOARD_USER_BUTTON_GPIO,
BOARD_USER_BUTTON_GPIO_PIN,
                                kGPIO_IntRisingOrFallingEdge);
    NVIC_EnableIRQ(BOARD_USER_BUTTON_IRQ);

    PRINTF("WOW voting demo (press %s): timestamps in us will print on
edges)\r\n",
           BOARD_USER_BUTTON_NAME);

    uint32_t lastEdge = 0;
    while (1) {
        if (s_edgeTimeUs != lastEdge) {
            lastEdge = s_edgeTimeUs;
            PRINTF("Edge @ %u us, WOW truth=%u\r\n", (unsigned)lastEdge,
(unsigned)s_wowTruth);
            // Visualize truth on LED
            GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN,
                          s_wowTruth ? LOGIC_LED_ON : LOGIC_LED_OFF);
        }
    }
}
```

Lab 4 — **Electrical** pad settings you should know (pulls, hysteresis, drive, slew, open-drain)

When you must override or audit the electrical configuration, use the correct pad constants:

- **LED (GPIO1_IO09)** pad is `GPIO_AD_B0_09` → `IOMUXC_GPIO_AD_B0_09_GPIO1_IO09`.
- **SW8 (GPIO5_IO00)** pad is `SNVS_WAKEUP` → `IOMUXC_SNVS_WAKEUP_GPIO5_IO00`.

```
#include "fsl_iomuxc.h"

static void pad_config_audit(void) {
    // Ensure mux clocks are on if not handled by BOARD_InitBootPins
    CLOCK_EnableClock(kCLOCK_Iomuxc);
    CLOCK_EnableClock(kCLOCK_IomuxcSnvs);

    // LED: push-pull output, slow slew, modest drive
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09,
        IOMUXC_SW_PAD_CTL_PAD_PKE_MASK |              // pull/keeper enable
        IOMUXC_SW_PAD_CTL_PAD_PUE_MASK |              // pull selected (vs
keeper)
        IOMUXC_SW_PAD_CTL_PAD_PUS(2U)  |              // pull-up ~100k
(device-specific)
        IOMUXC_SW_PAD_CTL_PAD_SPEED(2U) |             // medium speed
        IOMUXC_SW_PAD_CTL_PAD_DSE(2U)   |             // moderate drive
        0U /* ODE=0 push-pull, HYS not needed on output */);

    // SW8 input: enable pull-up + hysteresis; slow slew is fine
    IOMUXC_SetPinMux(IOMUXC_SNVS_WAKEUP_GPIO5_IO00, 0U);
    IOMUXC_SetPinConfig(IOMUXC_SNVS_WAKEUP_GPIO5_IO00,
        IOMUXC_SW_PAD_CTL_PAD_PKE_MASK |
        IOMUXC_SW_PAD_CTL_PAD_PUE_MASK |
        IOMUXC_SW_PAD_CTL_PAD_PUS(2U)  |              // pull-up
        IOMUXC_SW_PAD_CTL_PAD_HYS_MASK |              // Schmitt trigger
        IOMUXC_SW_PAD_CTL_PAD_DSE(2U));               // modest drive
(applies if used as output)
}
```

> The exact numeric values for `PUS`, `DSE`, and `SPEED` are device/SDK-version specific; the pattern above is what to audit: **pull + hysteresis for inputs**, **slow slew/low drive** unless timing demands otherwise, and **open-drain (ODE)** if you configure a shared aircraft discrete output.

8) Strong avionics-grade scenarios to simulate

1) **WOW with transient spikes:** Inject brief pulses; show Schmitt + voting reject them. Log edge timestamps and report "Noise event ignored; no state change."
2) **Wired-OR FLT/GRD line:** Tie two open-drain outputs with a single pull-up. Show either controller can assert ground. Demonstrate fault containment when one MCU resets.
3) **Stuck-at BIT:** Short the WOW input to ground; detect stuck-low during maintenance stimulus and report to the **CMC (Central Maintenance Computer)** (log to console for the lab).
4) **EMI mitigation tradeoff:** Observe ringing at high toggle rates; then reduce drive/slow slew via pad config and measure improvement.

---

9) Best practices (hardware + software)

- **Match electrical domains:** Never expose 28 V aircraft discretes directly to MCU pins. Use dividers, TVS, isolation.
- **Fail-safe states:** Choose pulls and output types so a broken wire or power-down yields the least hazardous interpretation.
- **Enable hysteresis:** Turn on Schmitt trigger for external discretes.
- **Respect pull strengths:** Disable MCU pulls if a strong external pull exists to avoid undesired dividers.
- **Debounce deterministically:** Use a stable time base (SysTick or GPT). Record raw and filtered states.
- **BIT for stuck-at/open:** Exercise inputs/outputs at power-up and periodically; use loopbacks when possible.
- **Tame EMI:** Use the lowest drive and slowest slew that meet timing; add small series resistors if needed.
- **ISRs stay short:** Clear flags first; defer work to main loop/task.
- **Document bank supplies & mux:** Keep a one-pager mapping pads → banks → rails; control `pin_mux.c` under configuration management.
- **Traceability:** Requirements → pad config → code → tests; keep `pin_mux.c` settings under revision control.

---

10) Assessment checklist

- Asserted polarity and fail-safe state documented?
- Pad configs explicitly set pulls, hysteresis, slew, drive, and open-drain (as needed)?
- Debounce/filter rationale stated (e.g., 5 ms)?
- BIT covers stuck-high/low and open-circuit?

- Edge timestamps logged for maintenance?
- EMI/ESD robustness considered (lab + DO-160 assumptions)?

---

11) Notes for later ARINC 429 sessions

ARINC (Aeronautical Radio, Incorporated) 429 is a differential, unidirectional serial bus. While not a discrete itself, you frequently manage **enables**, **health discretes**, and **label-present** lines as GPIOs alongside the ARINC transceiver (e.g., on an ADK-8582 interfaced over UART). The GPIO concepts above—pad config, open-drain, debounce, BIT—apply directly to those companion lines.

Implementation guidance

- Prefer the **board macros** shown here instead of hardcoding banks/pins. This matches the SDK and reduces porting effort.
- Use the **MCUXpresso Pins Tool** to generate `pin_mux.c` and **explicitly** include the electrical options you require (pulls, hysteresis, drive, slew, open-drain). Keep the generated file under configuration control.