Discrete I/O Driver Design Using Polling

---

1. What is a “Discrete I/O” and why polling?

A **discrete I/O** is a digital signal with two logical states (asserted/deasserted, high/low, true/false). In aircraft systems these lines are used for: - **State annunciation** (e.g., “Weight-On-Wheels (WOW)”, “Gear handle UP/DOWN”, “Door closed”). - **Hard enables/inhibits** (e.g., “Hydraulic pump enable”, “Thrust reverser interlock”).

Polling means firmware periodically reads inputs and updates outputs in a timed loop, instead of using interrupts. For safety-critical functions, polling offers: - **Determinism:** fixed sampling cadence; easier to prove timing for RTCA **DO-178C** (Software Considerations in Airborne Systems) objectives. - **Simplicity:** fewer concurrency hazards; straightforward verification. - **Control:** easy to implement debouncing, majority voting, and cross-channel checks inside one loop.

When not to use polling: very high-rate edges, very tight latency requirements, or when CPU loading is constrained (then consider interrupts or DMA). For most aircraft discretes (human-scale switches, relays, proximity sensors), 1–10 ms polling is more than sufficient.

---

2. Key terms (expanded on first use)

- **IOMUXC** – I/O Multiplexer Controller that binds a physical pad to a peripheral function (GPIO, LPUART, etc.) and configures pad electrical features (pull, drive, slew).
- **GPIO** – General-Purpose Input/Output peripheral bank (GPIO1…GPIO5 on i.MX RT1050).
- **Active-high / Active-low** – whether “asserted” corresponds to logic 1 or logic 0 at the MCU pin.
- **Debouncing** – algorithm that filters bounces/glitches so a state change is only accepted after the new level is stable for a specified time.
- **Fail-safe default** – output default that places the controlled function in the safest state on boot or fault.
- **Built-in Test (BIT)** – self-checks such as stuck-at-high/low detection at startup (**PBIT**: Power-on Built-in Test).

3. i.MX RT1050 EVKB specifics you will use

- **User LED:** pad **GPIO_AD_B0_09 → GPIO1_IO09** (on EVKB this is the green user LED).
- **User push-button SW8:** pad **SNVS_WAKEUP → GPIO5_IO00**.
- **Voltage:** 3.3 V logic on the MCU. (Aircraft interfaces commonly need 28 V discrete conditioning off-board; see Best Practices).

MCUXpresso SDK already provides the drivers you will use: - `fsl_iomuxc.h` → `IOMUXC_SetPinMux`, `IOMUXC_SetPinConfig` - `fsl_gpio.h` → `GPIO_PinInit`, `GPIO_PinWrite`, `GPIO_PinRead`, `GPIO_PortSet/Clear/Toggle` - Board helpers in `board.h`, `pin_mux.h`, `clock_config.h`

> We will stick to these SDK APIs so your code integrates cleanly with the provided SDK.

---

4. Architecture of a polling-based discrete driver

Goal: Provide a tiny reusable component that abstracts pins, debounces inputs, detects edges, and drives outputs deterministically.

4.1 Driver layering

1. **HAL (hardware abstraction)** – IOMUXC + GPIO set-up using SDK calls.
2. **Discrete channel model** – a `struct` per line describing port, pin, polarity (active-high/low), debounce time, and current state.
3. **Poller** – periodic function (called from the main loop) that samples all inputs, runs debouncing, emits events, and updates derived outputs.

4.2 Timing model

- Choose a **poll period** (T_p). Typical: 5 ms.
- For a **debounce time** (T_d) (e.g., 20 ms), compute **debounce counts** (N = T_d / T_p).
- Each line keeps a counter that increments while the sampled level differs from the debounced state; once the counter reaches (N), the debounced state flips.

4.3 Edge/event model

For each input line, track **debounced state** and generate **events**: `RISING`, `FALLING`, or `NONE`. This allows the application to react only to transitions (e.g., toggle on press).

4.4 Output model

Outputs can be configured as active-high or active-low. The driver accepts a logical command (true = asserted) and writes the correct physical level.

---

5. Generic example: button mirrors to LED (debounced, polled)

Behavior: Sample SW8 every 5 ms. When the debounced state is **pressed**, turn the user LED **ON**; when released, turn it **OFF**. Also print transitions.

Pins: - **Input:** SW8 → `GPIO5` pin `0` (active-low on EVKB; pressing pulls low).
- **Output:** LED → `GPIO1` pin 9 (active-high for LED logic).

Electrical/Pad set-up: - `IOMUXC_SetPinMux(IOMUXC_SNVS_WAKEUP_GPIO5_IO00, 0U)` → bind pad to `GPIO5_IO00`.
- `IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0U)` → bind LED pad.
- Use `IOMUXC_SetPinConfig(...)` to enable keeper/pulls as needed (examples below).

We will provide full code in §8 (Exercises & Solutions), ready to drop into the SDK tree.

---

6. Avionics use-case example: WOW-gated pump command (polled discretes)

Scenario: The **Hydraulic Pump A** may only energize when the aircraft is airborne (i.e., **WOW** not asserted). Also, to avoid nuisance transitions, inputs must be stable for **30 ms**. Additionally, we detect **stuck-at** faults during **PBIT (Power-on BIT)**.

Inputs (discretes): - **WOW (Weight-On-Wheels):** asserted on ground (active-low at the conditioned MCU pin).
- **MAINT_DOOR_CLOSED:** asserted when the maintenance door is closed (active-high).

Output (discrete): - **PUMP_A_CMD:** asserted enables the pump driver (active-high). On EVKB we will map it to the user LED for demonstration.

Rules: 1. Poll every **5 ms**.
2. Debounce time **30 ms** → **6 samples**.
3. **Command logic:** `PUMP_A_CMD = (NOT WOW) AND MAINT_DOOR_CLOSED`.
4. **PBIT:** During the first 200 ms of polling, verify each input toggles at least once if commanded by the procedure (on bench, you can press the button to emulate). If not, log

stuck-at.
5. **Fail-safe:** On reset and on detected fault, **deassert** `PUMP_A_CMD`.

We implement this with the same driver from §7 and wire two logical inputs; on the EVKB demonstration `WOW` will map to SW8 (active-low) and `MAINT_DOOR_CLOSED` can be tied to a constant (or another GPIO if you connect a jumper).

---

7. Implementation details

7.1 Pin muxing & pad control (based on SDK)

Below is a minimal `pin_mux.c` fragment that sets up UART, LED, and SW8. In your project, either generate this via MCUXpresso Config Tools or merge into your existing `BOARD_InitPins()`.

```
#include "fsl_iomuxc.h"

void BOARD_InitPins(void)
{
    // LED: GPIO_AD_B0_09 → GPIO1_IO09
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0U);
    // Optional: keeper/pull, slow slew, medium drive
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0x10B0U);

    // UART (console) – leave as in SDK examples if needed
    // IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_12_LPUART1_TXD, 0U);
    // IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_13_LPUART1_RXD, 0U);
    // IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_12_LPUART1_TXD, 0x10B0U);
    // IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_13_LPUART1_RXD, 0x10B0U);

    // Button SW8: SNVS_WAKEUP → GPIO5_IO00
    IOMUXC_SetPinMux(IOMUXC_SNVS_WAKEUP_GPIO5_IO00, 0U);
    // Pad config for input w/ keeper/pull; use default or set explicitly if
desired
    // (SNVS_WAKEUP pin may have limited configurable options; defaults are
fine for lab.)
}
```

7.2 GPIO initialization (SDK `fsl_gpio.h`)

```
#include "fsl_gpio.h"

// LED output (active-high logical)
static void LED_Init(void)
{
    gpio_pin_config_t led = {
        .direction = kGPIO_DigitalOutput,
        .outputLogic = 0U,   // start OFF (fail-safe)
```

```
        .interruptMode = kGPIO_NoIntmode
    };
    GPIO_PinInit(GPIO1, 9U, &led); // GPIO1_IO09
}

// SW8 input (active-low physical level when pressed)
static void BTN_Init(void)
{
    gpio_pin_config_t btn = {
        .direction = kGPIO_DigitalInput,
        .outputLogic = 0U,
        .interruptMode = kGPIO_NoIntmode
    };
    GPIO_PinInit(GPIO5, 0U, &btn); // GPIO5_IO00
}
```

7.3 Discrete driver (polling + debounce + edge detection)

Header (io_discrete_poll.h)

```
#pragma once
#include <stdint.h>
#include <stdbool.h>
#include "fsl_gpio.h"

typedef enum { IO_EDGE_NONE=0, IO_EDGE_RISE, IO_EDGE_FALL } io_edge_t;

typedef struct {
    GPIO_Type *port;       // GPIO1..GPIO5
    uint32_t   pin;        // 0..31
    bool       activeHigh; // true if logic-1 = asserted
    uint8_t    debounceN;  // samples required to accept a change
    // Runtime state
    bool       debounced;  // current debounced asserted/not
    uint8_t    cnt;        // counter toward debounceN
} io_input_t;

typedef struct {
    GPIO_Type *port;
    uint32_t   pin;
    bool       activeHigh;
} io_output_t;

// Initialize a GPIO as input/output (assumes pin mux already configured)
void io_input_init(io_input_t *in);
void io_output_init(io_output_t *out, bool initialAsserted);

// Poll one input once (call every T_p). Returns edge for debounced state.
io_edge_t io_input_poll(io_input_t *in);
```

```
// Output control (logical asserted/deasserted)
void io_output_set(io_output_t *out, bool asserted);
bool io_output_get(const io_output_t *out);
```

Source (io_discrete_poll.c)

```
#include "io_discrete_poll.h"

static inline uint32_t read_level(GPIO_Type *port, uint32_t pin)
{
    return GPIO_PinRead(port, pin) ? 1U : 0U;
}

static inline void write_level(GPIO_Type *port, uint32_t pin, uint32_t level)
{
    GPIO_PinWrite(port, pin, level & 1U);
}

void io_input_init(io_input_t *in)
{
    gpio_pin_config_t cfg = { .direction = kGPIO_DigitalInput,
                              .outputLogic = 0U,
                              .interruptMode = kGPIO_NoIntmode };
    GPIO_PinInit(in->port, in->pin, &cfg);
    // Seed debounced state from current physical level
    uint32_t raw = read_level(in->port, in->pin);
    bool asserted = in->activeHigh ? (raw!=0U) : (raw==0U);
    in->debounced = asserted;
    in->cnt = 0U;
}

void io_output_init(io_output_t *out, bool initialAsserted)
{
    gpio_pin_config_t cfg = { .direction = kGPIO_DigitalOutput,
                              .outputLogic = 0U,
                              .interruptMode = kGPIO_NoIntmode };
    GPIO_PinInit(out->port, out->pin, &cfg);
    io_output_set(out, initialAsserted);
}

io_edge_t io_input_poll(io_input_t *in)
{
    // Sample physical level and map to logical asserted
    uint32_t raw = read_level(in->port, in->pin);
    bool sampleAsserted = in->activeHigh ? (raw!=0U) : (raw==0U);

    if (sampleAsserted != in->debounced) {
        if (++in->cnt >= in->debounceN) {
```

```
            // Accept change
            bool old = in->debounced;
            in->debounced = sampleAsserted;
            in->cnt = 0U;
            if (!old && in->debounced) return IO_EDGE_RISE;
            if (old && !in->debounced) return IO_EDGE_FALL;
        }
    } else {
        in->cnt = 0U; // stable again, reset counter
    }
    return IO_EDGE_NONE;
}

void io_output_set(io_output_t *out, bool asserted)
{
    uint32_t level = out->activeHigh ? (asserted ? 1U : 0U)
                                     : (asserted ? 0U : 1U);
    write_level(out->port, out->pin, level);
}

bool io_output_get(const io_output_t *out)
{
    // Read back physical and map to logical asserted state
    uint32_t raw = read_level(out->port, out->pin);
    if (out->activeHigh) return raw != 0U; else return raw == 0U;
}
```

7.4 Main application (generic example)

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "io_discrete_poll.h"
#include "fsl_common.h" // SDK_DelayAtLeastUs

#define POLL_PERIOD_US 5000U // 5 ms

static io_input_t btn_sw8 = {
    .port = GPIO5, .pin = 0U, .activeHigh = false, .debounceN = 4 // 20 ms
};
static io_output_t led = { .port = GPIO1, .pin = 9U, .activeHigh = true };

void BOARD_InitHardware(void);

int main(void)
{
    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
```

```
    BOARD_InitDebugConsole();

    BTN_Init();   // if you keep these helpers; or just io_input_init()
directly
    LED_Init();

    io_input_init(&btn_sw8);
    io_output_init(&led, false);

    PRINTF("Discrete I/O (polled) demo. Press SW8 to light LED.\r\n");

    for (;;) {
        io_edge_t e = io_input_poll(&btn_sw8);
        // Mirror debounced level to LED (pressed→asserted→ON)
        io_output_set(&led, btn_sw8.debounced);

        if (e == IO_EDGE_RISE)  PRINTF("SW8 debounced: PRESSED\r\n");
        if (e == IO_EDGE_FALL)  PRINTF("SW8 debounced: RELEASED\r\n");

        SDK_DelayAtLeastUs(POLL_PERIOD_US, CLOCK_GetFreq(kCLOCK_CpuClk));
    }
}
```

The code above uses only SDK APIs (`fsl_gpio.h`, `fsl_iomuxc.h`, `fsl_common.h`) and the board scaffolding that ships in the SDK examples.

---

8. Hands-on exercises (with SDK-based solutions)

Exercise 1 – Single input with debounce (polling)

Task: Implement a polled, debounced read of SW8 and print “PRESSED/RELEASED”. LED mirrors the debounced state.

Steps: 1. Start from `boards/evkbimxrt1050/driver_examples/gpio/led_output` example as a template (keeps UART + LED already working).
2. Merge the `BOARD_InitPins()` changes from §7.1 to add SW8 muxing.
3. Add `io_discrete_poll.[ch]` from §7.3 to your project.
4. Replace `gpio_led_output.c` `main()` with §7.4’s `main()`.

Solution: exactly as shown in §7, with `POLL_PERIOD_US=5000` and `debounceN=4` (≈20 ms debounce).

Exercise 2 – Multi-channel discrete bank with events

Task: Create a table of **four** input discretes and a periodic poller that: - Updates each line’s debounced state. - Generates `RISING/FALLING` events into a small ring buffer (depth 16). - Mirrors line 0 to the LED; line 1 toggles the LED on a press (edge), lines 2–3 only print.

Hints: - Represent each line with `io_input_t`; store them in an array. - Design a tiny event record `{index, edge, tick}`; push when `io_input_poll()` returns a non-NONE edge. - Use `SDK_DelayAtLeastUs()` to keep the poll cadence deterministic.

Solution (sketch):

```
#define MAX_LINES 4
static io_input_t lines[MAX_LINES] = {
  {GPIO5,0,false,4}, /* SW8 */
  {GPIO1,9,true,4},  /* reuse LED pad wired as input if you move the jumper;
else choose another free pad */
  {GPIO1,3,true,4},  /* example placeholder */
  {GPIO1,2,true,4},
};

typedef struct { uint8_t idx; io_edge_t edge; uint32_t tick; } evt_t;
static evt_t q[16]; static uint8_t qh, qt; static uint32_t tick;
static void qpush(uint8_t i, io_edge_t e) { q[qh] = (evt_t){i,e,tick};
qh=(qh+1)&15; if (qh==qt) qt=(qt+1)&15; }

for (int i=0;i<MAX_LINES;i++) io_input_init(&lines[i]);
for (;;) {
  tick++;
  for (int i=0;i<MAX_LINES;i++) {
    io_edge_t e = io_input_poll(&lines[i]);
    if (e) qpush(i,e);
  }
  // Consume events
  while (qt!=qh) {
     evt_t ev=q[qt]; qt=(qt+1)&15;
     PRINTF("L%u %s at t=%lu\r\n", ev.idx,
ev.edge==IO_EDGE_RISE?"RISE":"FALL", ev.tick);
     if (ev.idx==1 && ev.edge==IO_EDGE_RISE) { // toggle LED on press
        bool cur = io_output_get(&led);
        io_output_set(&led, !cur);
     }
  }
  SDK_DelayAtLeastUs(POLL_PERIOD_US, CLOCK_GetFreq(kCLOCK_CpuClk));
}
```

Note: replace placeholder pins with actual free pads if you wire extra switches on the EVKB headers.

Exercise 3 – WOW-gated pump command (avionics scenario)

Task: Implement the logic in §6 with 5 ms polls and 30 ms debounce. Use the LED as `PUMP_A_CMD` indication.

Acceptance criteria: - LED **never** lights while WOW is asserted (simulate by pressing SW8 since it is active-low).

- When released (airborne), LED lights only if `MAINT_DOOR_CLOSED` is logically true (define a constant or add a second input). - On power-up, LED is off until both inputs are valid (after debounce). - If an input remains fixed for >1 s while the app expects motion during PBIT, print `STUCK_AT_HIGH/LOW`.

Solution (core):

```
static io_input_t wow = {GPIO5,0,false,6};              // active-low
static bool maint_door_closed_const = true;             // emulate constant
asserted
static io_output_t pump_cmd = {GPIO1,9,true};           // LED

io_input_init(&wow);
io_output_init(&pump_cmd, false);

uint32_t pbit_timer_ms=0; bool pbit_wow_toggled=false; bool pbit_done=false;

for (;;) {
  io_edge_t e = io_input_poll(&wow);
  if (e) pbit_wow_toggled = true;

  bool airborne = !wow.debounced; // NOT WOW
  bool pump_ok = airborne && maint_door_closed_const;
  io_output_set(&pump_cmd, pump_ok);

  if (!pbit_done) {
    pbit_timer_ms += 5;
    if (pbit_timer_ms >= 1000U) {
      if (!pbit_wow_toggled) PRINTF("PBIT: WOW stuck-at (no toggle)\r\n");
      pbit_done = true;
    }
  }
  SDK_DelayAtLeastUs(5000U, CLOCK_GetFreq(kCLOCK_CpuClk));
}
```

Exercise 4 – CPU-time measurement of the poller (optional advanced)

Use the ARMv7-M **DWT** cycle counter to measure how long the poll loop takes and log worst-case. This is useful evidence for **DO-178C** timing objectives.

```
static void dwt_init(void){ CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
DWT->CYCCNT = 0; DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk; }
static uint32_t dwt_cycles(void){ return DWT->CYCCNT; }

// Around your poll section:
uint32_t t0=dwt_cycles();
// poll lines and process
uint32_t dt=dwt_cycles()-t0;
PRINTF("poll dt = %lu cycles\r\n", dt);
```

9. Verification plan (bench)

1. **Power-on BIT:** After reset, confirm LED is OFF and console prints the banner.

2. **Debounce check:** Press and release SW8 rapidly. LED should change only once per deliberate act; console prints single PRESS/RELEASE lines.

3. **Timing check:** With Exercise 4, ensure `dt` << (poll period × CPU frequency). On a 600 MHz i.MX RT1050 and a small channel set, `dt` will be in the few hundred cycles.
4. **Fault injection:** Hold SW8 pressed at reset to emulate WOW asserted. Ensure `PUMP_A_CMD` (LED) remains OFF until release.

---

10. Best practices for airborne discrete I/O

- **Conditioning to MCU levels:** Aircraft "28 V discrete" lines must be conditioned (opto-isolation, resistor dividers, filters) to 3.3 V logic. Avoid direct wiring to MCU. Ensure **ESD (Electrostatic Discharge)** and **EMI (Electromagnetic Interference)** protection as per **RTCA DO-160** environmental tests.
- **Polarity clarity:** Document **logical** meaning (asserted/deasserted) separately from **electrical** level (high/low). The `activeHigh` flag formalizes this.
- **Debounce ≥ 20 ms** for mechanical switches. For proximity sensors or relay contacts, confirm with hardware specs.
- **Fail-safe defaults:** Outputs should start **deasserted**. On internal faults (watchdog reset, over-temperature), fall back to safe state.
- **No busy waits in production:** While `SDK_DelayAtLeastUs()` is adequate for labs, prefer a tick-driven scheduler (e.g., SysTick) to guarantee poll cadence without burning CPU.
- **Bounded complexity:** Keep per-poll work O(N) with small constants; avoid heap allocations inside the poller.
- **Traceability:** Map each requirement (e.g., "30 ms debounce") to exact constants in code for **DO-178C** traceability.
- **Unit isolation:** If multiple discrete banks exist, design **channelized** instances (A/B) with no shared state for **DAL (Design Assurance Level)** partitioning.
- **Regression logs:** Print event logs with time tags in bench builds; compile out in flight builds.

---

11. Where in the SDK these pieces come from (for reference)

- **GPIO LED example:** `boards/evkbimxrt1050/driver_examples/gpio/led_output/`

    - `pin_mux.c`: shows `IOMUXC_GPIO_AD_B0_09_GPIO1_IO09` and UART muxing.

- **GPIO input example:**
  `boards/evkbimxrt1050/driver_examples/gpio/input_interrupt/`
    - `pin_mux.c`: shows `IOMUXC_SNVS_WAKEUP_GPIO5_IO00` setup for SW8 (we use polling, but mux is identical).

- **APIs used:** `fsl_iomuxc.h`, `fsl_gpio.h`, `fsl_common.h`, `fsl_debug_console.h`.

The solutions provided above are written to those SDK APIs so you can drop them into any new SDK example project. They avoid vendor-independent register twiddling for clarity and maintainability.

---

12. Next steps

- Extend the driver with **input voting** (2-out-of-3) and **plausibility checks** for redundant sensors.

- Add a simple **scheduler tick** (SysTick) so the poller runs from a precise timebase without active delays.

- Wrap the driver with **unit tests** on the host using a simulated GPIO layer to verify edge cases (bounce patterns, stuck-at conditions).

Appendix A – Minimal combined `BOARD_InitHardware()` (as used in many SDK examples)

```
void BOARD_InitHardware(void)
{
    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();
}
```

Appendix B – Pad config hint (`0x10B0U`)

This common value in SDK examples configures a reasonable keeper/pull and slew/drive for many pads. For production, size these per your signal integrity and EMC analysis.