Hardware Fault Injection & Signal-Glitch–Induced Resets

Platform focus: NXP i.MX RT1050-EVKB (Cortex-M7) • **I/O context for avionics labs:** EVKB interfaced to Holt **ADK-8582** (HI-8582/HI-8583) ARINC 429 evaluation board via UART console bridge.

---

1) Why this module exists (learning outcomes)

By the end of this module, you will be able to: 1. Distinguish power, clock, reset-pin and electromagnetic/transient fault mechanisms and explain how they precipitate **system resets**. 2. Read, decode and persist **reset cause** on the i.MX RT1050 using the **System Reset Controller (SRC)**; maintain a monotonic **boot counter** across resets. 3. Configure **RTWDOG (windowed watchdog)** safely; prove, with experiments, that watchdog and reset sources are correctly attributed. 4. Design board- and firmware-level **glitch resilience** (reset supervision, power integrity, EMI/ESD countermeasures, software fallbacks). 5. In an **avionics** context (ARINC 429), build a fault-tolerant data path; recover from microcontroller resets without corrupting bus timing or labels.

---

2) Plain-language definitions (no jargon left behind)

Hardware fault injection (HFI): Any deliberate or accidental physical perturbation—voltage, clock, reset, electromagnetic, or environmental—that changes the microcontroller's intended behavior. In safety-critical work our goal is *resilience* (avoid unsafe states), not bypassing protections.

Signal glitch: A short-duration, unintended pulse or transient on a digital or power signal. On reset lines it can cause spurious **POR/warm** resets; on power rails it can cause **brown-outs**; on clocks it can cause **loss-of-lock** or instruction fetch faults.

Reset taxonomy (as seen by i.MX RT1050 SRC): - **POR (Power-On Reset):** Triggered by the POR_B circuit when power ramps. - **External reset pin events:** Short pulses on **IPP_RESET_B/IPP_USER_RESET_B** qualified by on-chip logic. - **Watchdog resets:** From **RTWDOG/WDOG3** or **WDOG1/WDOG2** timeouts. - **Software/CPU lockup resets:** Via **SYSRESETREQ** or lockup. - **JTAG-initiated resets:** System, software, or certain IR code selections.

Glitch sources you'll meet in the lab: - **Power droop/brown-out:** Supply impedance + step load → rails dip → POR logic asserts. - **Reset-pin bounce:** Mechanical button bounce, crosstalk, or radiated bursts coupled onto the trace. - **Clock disturbance:** RF/EMI near the crystal/PLL or bad decoupling. - **EMI/EFT (Electro-Magnetic Interference / Electrical Fast Transient):** Cable injection (e.g., ARINC 429 cabling near static discharges) coupling into MCU nets.

> **Important:** In all exercises we stay strictly within component absolute-maximum ratings. We never attempt security bypass; we explore reliability and recovery only.

3) The i.MX RT1050 reset architecture in practice

3.1 What the SRC (System Reset Controller) tells you

On each boot, the SRC exposes **sticky flags** indicating the last reset source. You will: - Read **SRC_GetResetStatusFlags()** (MCUXpresso SDK) at the start of `main()`. - Map bits such as **Watchdog**, **WDOG3 (RTWDOG)**, **JTAG**, **IPP_RESET_B**, **IPP_USER_RESET_B**, **CPU lockup**, etc., to human-readable strings. - Clear the flags you've consumed so the next reset is unambiguous.

3.2 GPRs survive resets

Use **SRC GPRx** (General Purpose Registers) as tiny, always-on "scratchpads" across resets to keep a **boot counter**, last reset cause, and a sequence number for post-mortem correlation.

3.3 Board-level reset and supervision

On the EVKB the pushbutton asserts reset via the board's reset path. In production avionics hardware, add an **external reset supervisor IC** to qualify POR_B and guarantee **minimum reset pulse width** and **release timing** relative to power rails. Pair this with clean routing (short reset trace, strong pull-up, optional RC for switch bounce), and robust decoupling near VDD pins.

---

4) How signal glitches become resets (failure physics)

1. **Supply droop** → POR comparator trips → SOC goes through cold or warm reset. Typical trigger: motor/relay actuation, RF PA burst, or ARINC 429 line drivers starting up.
2. **Reset-pin transients** → Threshold crossing for just long enough to qualify as reset. Typical trigger: ESD, cable hot-plug, or long reset trace as an antenna.
3. **Clock/PLL disturbance** → Instruction fetch or bus faults → software triggers system reset or watchdog times out.
4. **EMI/EFT bursts** on I/O → Internal protection diodes inject current → local ground bounce → unintended resets.

---

5) Generic example (non-avionics)

A handheld industrial scanner intermittently resets when a high-current barcode LED strobe fires. Root cause: shared 3.3 V rail with insufficient bulk capacitance; 1.8 V LDO momentarily falls out of regulation; POR asserts. Fixes: add 22 µF bulk + 0.1 µF local ceramics per IC, move LED driver to its own buck, add a 180 ms reset supervisor with proper release sequencing, and implement windowed watchdog for graceful recovery.

---

6) Specific avionics use case (strong, realistic scenario)

Platform: A **Remote Data Concentrator (RDC)** collecting discrete inputs and forwarding status over **ARINC 429 (Aeronautical Radio, Incorporated 429)** to a **Flight Control Computer (FCC)**. The RDC is built on i.MX RT1050; ARINC 429 Tx/Rx are handled by a Holt HI-8582 on the **ADK-8582** evaluation board. A cabin maintenance event produces **EFT/ESD bursts** on a harness near the RDC bay.

Observed symptom: Sporadic resets every few hours, each causing a 300 ms gap in label transmission. Downstream LRUs (Line-Replaceable Units) flag “intermittent source” faults when three consecutive words are missed.

Root-cause chain: Harness-coupled burst → transient on MCU reset net (long, unterminated trace) → SRC shows external **IPP_RESET_B** resets; boot counter increments with no watchdog flags.

Remediation deployed: - Hardware: Add reset supervisor IC (≥140 ms), route reset trace away from high-dv/dt lines; tighten VDD decoupling around ARINC line driver; add TVS at connector; lift pull-up to 10 kΩ and add 100 nF RC to button path only (not POR_B); improve chassis bonding. - Firmware: Early-boot SRC logging, **windowed RTWDOG** (no feeding during boot stalls), **ARINC 429 stream state machine** that resumes at label boundaries after reset; **UART heartbeat** to the ADK-8582 bridge to re-negotiate stream parameters on reboot. - Operations: DO-160 (Environmental Conditions and Test Procedures for Airborne Equipment) radiated and conducted susceptibility sweeps repeated; no further resets.

---

7) Hands-on exercises (EVKB + MCUXpresso SDK 25.06.00)

> **Project base:** Start from the EVKB `hello_world` (XIP or SDRAM) example in the SDK you provided. Add the files below and enable `fsl_debug_console` for logging over OpenSDA UART.

Exercise 1 — “Know your resets”: decode and persist reset causes

Goal: On every boot, print the last reset cause, increment a boot counter in SRC GPR1, and clear the flags.

Key ideas: `SRC_GetResetStatusFlags()`, `SRC_ClearResetStatusFlags()`, `SRC_{Get,Set}GeneralPurposeRegister()`.

```
/* File: reset_diag.c  (add to your app and call ResetDiag_RunEarly()) */
#include "fsl_src.h"
#include "fsl_debug_console.h"

static const char *flag_to_str(uint32_t f)
{
    switch (f) {
    case kSRC_Wdog3ResetFlag:          return "RTWDOG/WDOG3 timeout";
    case kSRC_WatchdogResetFlag:       return "WDOG1/WDOG2 timeout";
```

```
    case kSRC_IppResetPinFlag:         return "IPP_RESET_B (power-up sequence)";
    case kSRC_IppUserResetFlag:        return "IPP_USER_RESET_B (external reset
pin)";
    case kSRC_JTAGSystemResetFlag:     return "JTAG system reset";
    case kSRC_JTAGSoftwareResetFlag:   return "JTAG software reset";
    case kSRC_JTAGGeneratedResetFlag:  return "JTAG generated reset (EXTEST/
HIGHZ)";
    case kSRC_LockupSysResetFlag:      return "CPU lockup / SYSRESETREQ";
    case kSRC_TemperatureSensorResetFlag: return "On-chip temperature sensor
reset";
    default: return "Unknown/Multiple";
    }
}

void ResetDiag_RunEarly(void)
{
    uint32_t flags = SRC_GetResetStatusFlags(SRC);
    uint32_t bootCount = SRC_GetGeneralPurposeRegister(SRC, 0); /* GPR1 index 0
*/
    bootCount++;
    SRC_SetGeneralPurposeRegister(SRC, 0, bootCount);

    PRINTF("\r\n=== Reset diagnostic ===\r\n");
    PRINTF("Boot count (GPR1): %lu\r\n", (unsigned long)bootCount);

    if (flags == 0) {
        PRINTF("Reset flags: none (cold power-up or flags already cleared)
\r\n");
    } else {
        PRINTF("Reset flags raw: 0x%08lx\r\n", (unsigned long)flags);
        /* Print all flags that are set (may be more than one). */
        if (flags & kSRC_Wdog3ResetFlag)         PRINTF(" - %s\r\n",
flag_to_str(kSRC_Wdog3ResetFlag));
        if (flags & kSRC_WatchdogResetFlag)      PRINTF(" - %s\r\n",
flag_to_str(kSRC_WatchdogResetFlag));
        if (flags & kSRC_IppResetPinFlag)        PRINTF(" - %s\r\n",
flag_to_str(kSRC_IppResetPinFlag));
        if (flags & kSRC_IppUserResetFlag)       PRINTF(" - %s\r\n",
flag_to_str(kSRC_IppUserResetFlag));
        if (flags & kSRC_JTAGSystemResetFlag)    PRINTF(" - %s\r\n",
flag_to_str(kSRC_JTAGSystemResetFlag));
        if (flags & kSRC_JTAGSoftwareResetFlag)  PRINTF(" - %s\r\n",
flag_to_str(kSRC_JTAGSoftwareResetFlag));
        if (flags & kSRC_JTAGGeneratedResetFlag) PRINTF(" - %s\r\n",
flag_to_str(kSRC_JTAGGeneratedResetFlag));
        if (flags & kSRC_LockupSysResetFlag)     PRINTF(" - %s\r\n",
flag_to_str(kSRC_LockupSysResetFlag));
        if (flags & kSRC_TemperatureSensorResetFlag) PRINTF(" - %s\r\n",
```

```
flag_to_str(kSRC_TemperatureSensorResetFlag));
    }

    /* Clear all observed flags so the next reset cause is unambiguous. */
    SRC_ClearResetStatusFlags(SRC, flags);
}
```

Integration: Call `ResetDiag_RunEarly();` at the very top of `main()` right after clocks and console init, e.g.:

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

    ResetDiag_RunEarly();

    PRINTF("System up.\r\n");
    while (1) { __NOP(); }
}
```

> **What you should see:** Press the EVKB reset button → `IPP_USER_RESET_B` appears. If you purposely cause a watchdog timeout in Exercise 2, you'll see `RTWDOG/WDOG3 timeout` flagged.

Exercise 2 — Configure a windowed watchdog and verify attribution

Goal: Configure **RTWDOG (aka WDOG3)** in *windowed* mode. Demonstrate three behaviors: (a) correct periodic refresh; (b) early refresh → reset; (c) missed refresh → reset. Verify SRC flags each time.

```
/* File: wdog_window.c */
#include "fsl_rtwdog.h"
#include "fsl_debug_console.h"
#include "board.h"

static void busy_delay(volatile uint32_t loops) { while (loops--) __NOP(); }

void Wdog_Demo_Run(void)
{
    rtwdog_config_t cfg;
    RTWDOG_GetDefaultConfig(&cfg);
    cfg.enableUpdate   =
```

```
true;                   /* allow reconfig without reset */
    cfg.enableInterrupt = false;                /* simple reset behavior */
    cfg.enableWindowMode = true;                /* enable windowing */
    cfg.windowValue     = 0x4000;               /* earliest allowed refresh */
    cfg.timeoutValue    = 0x9000;               /* reset if not refreshed by
then */
    cfg.prescaler       = kRTWDOG_ClockPrescalerDivide1;

    PRINTF("Configuring RTWDOG (window:0x%X, timeout:0x%X)\r\n",
cfg.windowValue, cfg.timeoutValue);
    RTWDOG_Init(RTWDOG, &cfg);

    /* Case (a): good refresh cadence (between window and timeout) */
    for (int i = 0; i < 5; ++i) {
        while (RTWDOG_GetCounterValue(RTWDOG) > cfg.windowValue) { /* wait to
enter window */ }
        RTWDOG_Refresh(RTWDOG);
        PRINTF("Refreshed in window, iteration %d\r\n", i);
    }

    /* Case (b): early refresh → intentional violation */
    PRINTF("Refreshing too early to trigger window violation reset...\r\n");
    RTWDOG_Refresh(RTWDOG); /* out-of-window refresh → reset */

    /* You will not reach here on violation; after reset, Exercise 1 prints
WDOG3 flag. */
}
```

Integration: Call `Wdog_Demo_Run();` from `main()` after `ResetDiag_RunEarly()`. To demonstrate **missed refresh** (case c), comment out the refresh loop so the timeout expires.

Safety note: RTWDOG uses its own low-frequency clock domain. Do not disable it globally in SystemInit if you want reliable reset attribution.

Exercise 3 — Reset-pin pulse qualification (safe, controlled)

Goal: Empirically determine what **minimum reset pulse width** on the EVKB's external reset input produces a reset, and demonstrate that adding a supervisor or RC qualification raises the immunity margin.

Method (safe lab pattern): - Use a lab **function generator** configured for 3.3 V logic levels (or lower with a buffer) and **AC-coupled via a series resistor (≥10 kΩ)** into the board's *reset test point* (not the POR_B pin). Start with **clean 5–10 ms pulses** at 0.2 Hz and progressively shorten to 1 ms/500 µs/100 µs. - Observe console output after each pulse. Exercise 1 prints `IPP_USER_RESET_B` when the pulse is long enough to qualify. - If your hardware has a reset supervisor mod, repeat the sweep; you should see the threshold lengthen (i.e., short pulses are ignored), which is the intended effect.

Boundaries: Never inject negative voltages, never exceed VDDIO, and never connect a low-impedance source directly to the reset pin. You are testing *qualification*, not abusing the net.

Exercise 4 — ARINC 429 continuity across MCU resets (EVKB ↔ ADK-8582 via UART bridge)

Goal: Maintain bus integrity and rapid recovery of ARINC 429 transmissions when the EVKB resets, without corrupting labels or data on the 429 bus.

Topology: EVKB UART (115 200 bps, 8-N-1) ↔ simple UART bridge (on the ADK-8582 kit or a small intermediary MCU if required) ↔ HI-8582 parallel interface driving the ARINC 429 line driver. The UART side accepts framed commands like `TX:<LABEL>:<SDI>:<DATA>:<SCHED>` and returns health/ack frames.

Firmware tasks on EVKB: 1. **Handshake:** On boot, send `HELLO <bootCount> <lastResetCause>`. Expect `READY` from the bridge. 2. **Stream restart:** Resume label schedule only after `READY`; this guarantees label boundaries after a reset. 3. **Heartbeat:** Send `PING n` every second; if 3 consecutive timeouts occur, pause and re-handshake. 4. **Logging:** Store last 32 events (reset, handshake, timeouts) in OCRAM ring for post-flight analysis.

Skeleton UART framing (EVKB side):

```
/* Pseudocode sketch for clarity; use SDK LPUART driver on EVKB */
void Arinc_Bridge_OnBoot(uint32_t bootCount, const char *cause)
{
    printf("HELLO %lu %s\r\n", (unsigned long)bootCount, cause);
    if (!WaitForReply("READY", 100 /*ms*/)) {
        /* Retry/backoff; do not emit ARINC traffic until READY received */
    }
}
```

Validation: While streaming labels, press the EVKB reset button (Exercise 1 & 2 active). You should see a short gap on the ARINC bus (the line driver keeps its last good state), then automatic resynchronization after `READY`.

---

8) Best-practices checklist (avionics-grade)

Architecture & requirements - Allocate reset behavior in the system safety assessment (ARP4754A / ARP4761): define maximum permitted **data gap** on ARINC 429 for your function and ensure watchdog/reset strategy cannot produce a hazardous latent fault. - Distinguish **degraded mode** (reduced label set) from **fail-silent** (no transmit) on recovery.

Hardware - Add an **external reset supervisor** with ≥140 ms hold-time; route reset short, isolated from high-dv/dt nets; use Schmitt-trigger where a manual button is present. - Power integrity: low-ESR bulk + local 0.1 µF decouplers per VDD pin; split noisy loads (ARINC line drivers) from MCU core rails; keep ground returns tight. - EMC: TVS on I/O at the connector, common-mode chokes on long runs, proper shielding/grounding, cable clamp bonding to chassis.

Firmware - Read and clear **SRC reset flags** *first thing*; persist **boot counter** in **SRC GPR1**. - Use **windowed RTWDOG** with an independent clock; refresh from a healthy-system task only (not a bare SysTick) so stalled systems time out. - Implement **boot-loop detection**: if `bootCount` increased N times within T seconds, enter 'safe mode' (minimal label set, heavy logging) and report via maintenance channel. - Make drivers **idempotent**: all I/O (UART/ARINC bridge) must re-initialize cleanly on any reset cause.

Verification - Reproduce faults with controlled pulses and DO-160 style susceptibility tests; capture **before/after** immunity margins. - Include reset-storm tests in CI hardware loops: random watchdog violations, external reset pulses, power rail brown-out *simulations* using programmable supplies (within ratings).

---

9) Troubleshooting guide (what you'll actually see)

- **"Unknown/Multiple" reset causes:** You forgot to clear flags after reading, or more than one source asserted. Clear and repeat.
- **WDOG resets when debugging:** You halted the core without disabling watchdog in debug mode. Set `workMode.enableDebug = true` or disable RTWDOG during interactive sessions.
- **Resets when ARINC bursts start:** Check line-driver inrush → beef up local decoupling; add soft-start if available.
- **Resets only with long UART cables:** Common-mode noise → add CM choke and improve bonding; ensure UART ground reference is solid.

---

10) Wrap-up (what "good" looks like)

- Every reset is **attributed** and **logged** with a monotonic counter.
- Watchdog is **windowed** and feeds only when the system is actually healthy.
- Short reset pulses and moderate EMI no longer produce spurious resets thanks to supervision and routing.
- ARINC 429 streams **resume deterministically** after EVKB resets without corrupting label timing.

Appendix A — Turn-key file list (drop-in to your SDK tree)

- `source/reset_diag.c` (Exercise 1)
- `source/wdog_window.c` (Exercise 2)
- `source/arinc_bridge.c` (Exercise 4, UART framing)

- Add includes: `#include "fsl_src.h"`, `#include "fsl_rtwdog.h"`, `#include "fsl_debug_console.h"` in `main.c` and link the corresponding drivers in your MCUXpresso project settings.

Appendix B — Acronyms (expanded at first use in text)

- **ARINC 429** — *Aeronautical Radio, Incorporated 429 serial data bus standard*
- **EMI** — *Electromagnetic Interference*
- **EFT** — *Electrical Fast Transient*
- **POR** — *Power-On Reset*
- **RDC** — *Remote Data Concentrator*
- **FCC** — *Flight Control Computer*
- **LRU** — *Line-Replaceable Unit*
- **SRC** — *System Reset Controller*
- **RTWDOG/WDOG3** — *Real-Time Watchdog Timer*
- **GPR** — *General Purpose Register (in SRC)*
- **DO-160** — *Environmental Conditions and Test Procedures for Airborne Equipment (RTCA)*
- **ARP4754A / ARP4761** — *Guidelines for development of civil aircraft and systems / Safety assessment*

Notes on kit connectivity

The Holt **ADK-8582** kit exposes the HI-8582/HI-8583 ARINC 429 device. Where a native UART bridge is unavailable, insert a small microcontroller bridge that accepts UART frames from EVKB and services the HI-8582's parallel interface; the exercise remains unchanged from the EVKB perspective.