Fault Injection Techniques (Bare-Metal) for Airbus — i.MX RT1050 EVKB + ARINC 429 via ADK-8582

Audience: Embedded/avionics engineers building safety-critical software on NXP i.MX RT1050 (ARM Cortex-M7) and integrating **ARINC 429 (Aeronautical Radio, Incorporated 429)** data links via **ADK-8582** over **UART (Universal Asynchronous Receiver/Transmitter)**.

Prereqs: C, basic MCU bring-up, familiarity with MCUXpresso SDK for **EVKB-IMXRT1050**. No prior fault injection experience required.

---

1) Why Fault Injection?

In avionics, we must demonstrate that software detects, contains, and recovers from faults with a rigor consistent with **DO-178C (Software Considerations in Airborne Systems and Equipment Certification)** and aligned with **ARP4761 (Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment)** and **ARP4754A (Guidelines for Development of Civil Aircraft and Systems)**. Fault injection is a structured way to *provoke* misbehaviour—bit flips, timing overruns, bus protocol violations—so we can verify monitors, *Built-In Tests (BIT)*, graceful degradation, and annunciation paths.

This module covers **software-implemented fault injection (SWIFI)** you can run directly on the **EVKB-IMXRT1050** board and **interface-level injection** for **ARINC 429** using the **ADK-8582** instrument via **UART** commands.

---

2) Clear Definitions (first use expanded)

- **Fault:** The cause of an error (e.g., flipped SRAM bit, corrupted pointer, malformed **ARINC (Aeronautical Radio, Incorporated)** word).
- **Error:** The system state deviating from correct state (e.g., wrong altitude value stored).
- **Failure:** External deviation from required service (e.g., incorrect display to pilot).
- **FI (Fault Injection):** Techniques to deliberately introduce faults to observe error detection, containment, and recovery.
- **SWIFI (Software Implemented Fault Injection):** Using code to corrupt memory, stall timing, trigger exceptions.
- **MPU (Memory Protection Unit):** ARM core feature creating memory regions with access permissions to provoke **MemManage** faults on violations.
- **NVIC (Nested Vectored Interrupt Controller):** ARM exception controller, routes **HardFault**, **BusFault**, **UsageFault**, **MemManage**.
- **SCB (System Control Block):** ARM system registers providing Fault Status registers (**CFSR**, **HFSR**, **BFAR**, **MMFAR**).

- **WDOG (Window Watchdog)** and **RTWDOG (Real-Time Watchdog):** i.MX RT watchdog peripherals that reset or interrupt on missed service.
- **CMSIS (Cortex Microcontroller Software Interface Standard):** HAL/headers used by the MCUXpresso SDK.
- **SSM (Sign/Status Matrix):** ARINC 429 bits [31:30] encoding status (Normal, No Computed Data, Functional Test, Failure).
- **SDI (Source/Destination Identifier):** ARINC 429 bits [10:9] identifying source or destination channel.

---

3) Platform map: EVKB-IMXRT1050 (ARM Cortex-M7)

- Core: ARM Cortex-M7 with **HardFault**, **BusFault**, **MemManage**, **UsageFault**, **NMI (Non-Maskable Interrupt)**, **Systick**.
- On-chip memory: OCRAM (On-Chip RAM). Execute-in-place from external QSPI via **FlexSPI**.
- Relevant peripherals in SDK: **LPUART** (for ADK-8582), **WDOG/RTWDOG**, **EDMA (Enhanced DMA)**, **TRNG (True Random Number Generator)**, **GPIO**, **FLEXPWM** (for the SDK's PWM fault demo).

SDK references (paths in your uploaded package): - `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/` - `boards/evkbimxrt1050/driver_examples/wdog/` and `.../rtwdog/` - `boards/evkbimxrt1050/driver_examples/edma/` - `boards/evkbimxrt1050/driver_examples/trng/random/` - `boards/evkbimxrt1050/demo_apps/pwm_fault/` (peripheral fault input demo)

---

4) Taxonomy of Fault Injection Covered Here

1. **Memory/Data corruption**
   Flip bits in SRAM buffers; over-/under-write; CRC mismatch; stale data reuse.
2. **Protection violations & exceptions**
   Use **MPU** to mark a region as **read-only** then attempt a write → **MemManage**; dereference invalid addresses → **BusFault**; divide by zero → **UsageFault**; escalate to **HardFault**.
3. **Timing/scheduling faults**
   Inject jitter/delay; starve service routines; miss **WDOG** window to force system reset; stress **NVIC** priority inversions.
4. **Interface-level faults (ARINC 429)**
   Bad parity (ARINC 429 is *odd parity*), illegal **SSM**, inconsistent **SDI**, wrong word rates/bursts (at instrument level), label mismatches.
5. **DMA/Peripheral faults**
   Mis-configure **EDMA** to write unexpected sizes; starve UART RX to simulate dropped words.
6. **Security-motivated fault tests**
   Brief note on **FIH (Fault-Injection Hardening)** patterns (e.g., double-checked states as seen in MCUboot), useful to guard safety monitors.

---

5) A generic worked example (end-to-end)

Scenario: A simple "sensor pipeline" reads a 32-byte buffer, checks a **CRC (Cyclic Redundancy Check)**, and publishes a value. We inject a single-bit flip and verify the CRC monitor catches it, then make sure the system *does not* crash (graceful handling + annunciation).

1. **Normal path:** Buffer → CRC32 → OK → publish.
2. **Injected fault:** Flip bit `k` in buffer using SWIFI utility → CRC32 fails → publish is inhibited; a **BIT (Built-In Test)** counter is incremented; an **ECAM (Electronic Centralized Aircraft Monitor)**/ maintenance log line is printed via `PRINTF`.
3. **Recovery:** After 3 consecutive CRC failures, **WDOG** reset is triggered deliberately to emulate a "go-safe" policy.

We provide **SDK-based code** for the buffer corruption utility and CRC in §10.

---

6) A strong avionics use case

Use case: *IRU (Inertial Reference Unit) air data fusion sanity monitoring on a Display Computer.* The **Display Computer (DC)** receives **ARINC 429** words for pressure altitude and indicated airspeed from an **ADC (Air Data Computer)**. The DC independently computes a consistency check using an internal inertial solution. When we inject:

- **Wrong parity** on altitude words → DC shall reject the word and maintain last-known-good for that label; increment a *word-level error counter*; flag **SSM = Failure** when generating outputs to downstream busses.
- **SSM = "Failure"** while the numeric field looks valid → DC shall trust **SSM** and discard the numeric value; annunciate *ADC A FAIL*.
- **Label mismatch** (airspeed word sent under an altitude label) → DC's label filter shall reject it and record *label discipline violation*.
- **Timing burst** (no inter-word gap) → DC's input driver shall detect overrun and report *ARINC input saturation*.

We will implement these tests using the **ADK-8582** (driving ARINC 429) with commands issued over **LPUART**. The i.MX RT1050 executes monitors that count, log, and react. Where the ADK-8582 command strings differ, adapt the `adk429_send_*()` wrappers accordingly.

---

7) i.MX RT1050 exception handling primer (Cortex-M7)

- **MemManage Fault:** MPU access violation (bad permissions or execute-never).
- **BusFault:** Precise/imprecise AXI/AHB bus error (e.g., invalid address).
- **UsageFault:** Undefined instruction, divide by zero (if enabled), unaligned access (if trapped).
- **HardFault:** Escalation or system error not handled by the above.
- **SCB registers:** `SCB->CFSR` (Configurable Fault Status), `SCB->HFSR` (HardFault Status), `SCB->BFAR`, `SCB->MMFAR`.

- **Recommendation:** Always implement verbose handlers that **log then reset** (or halt in lab mode). See §10 code.

---

8) Lab topology and toolchain

- **Board:** EVKB-IMXRT1050 via USB debug (CMSIS-DAP/J-Link) and UART (virtual or external).
- **Instrument:** ADK-8582 ARINC 429 generator/analyzer connected to DUT ARINC 429 RX; command/control via **LPUART** from the EVKB.
- **SDK:** Use the uploaded SDK. Start from `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/` for UART and `.../wdog/` for watchdog patterns.
- **Build:** MCUXpresso IDE or equivalent; include `board.c`, `clock_config.c`, `pin_mux.c`, and enable `PRINTF` through `fsl_debug_console.h`.

  Safety note: Perform all ARINC 429 injections on a bench harness, never on aircraft wiring. Use opto-isolated transceivers where applicable. Ensure the watchdog's final reset does not drive unintended IO states.

---

9) Step-by-step labs (with solutions)

Lab 1 — Memory bit-flip and CRC detection (SWIFI)

Goal: Demonstrate data integrity monitors catch single-bit errors.

Steps: 1. Create a new project by copying `boards/evkbimxrt1050/driver_examples/trng/random/` as a template (it already initializes board/clock/pins).
2. Add `fi_crc32.c/.h` and `fi_mutate.c/.h` from §10 to your project.
3. Call `FI_FlipRandomBit()` on a 32-byte buffer seeded by **TRNG (True Random Number Generator)**; compute CRC using `FI_CRC32()`.
4. If CRC mismatches, log via `PRINTF`, increment a *BIT* counter, and deliberately *do not* publish the buffer.

Success criteria: CRC failures are detected deterministically; no exceptions or crashes; counters/logs increment.

Lab 2 — MPU-guarded region and MemManage fault

Goal: Use the **MPU (Memory Protection Unit)** to catch stray writes.

Steps: 1. Add `fi_mpu.c/.h` from §10, which uses CMSIS (`core_cm7.h` and `m-profile/armv7m_mpu.h`).
2. Call `FI_MPU_EnableReadOnlyRegion(buf, 32U)`; then attempt a write to `buf[0]`.
3. Observe **MemManage** fault; see the **SCB** logs printed by `HardFault_Handler()`/`MemManage_Handler()`.

Success criteria: Violation triggers handler; status registers printed; system resets cleanly after log.

Lab 3 — UsageFault via divide-by-zero and recovery

Goal: Show **UsageFault** handling and graceful recovery path.

Steps: 1. Enable divide-by-zero trapping: `SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;`.
2. Deliberately divide by zero in a lab function guarded by `#if LAB_MODE`.
3. Handler logs the event and returns to a safe main loop.

Success criteria: No board lock-up; fault count increments; watchdog remains serviced.

Lab 4 — Missed WDOG service → autonomous reset

Goal: Show that the **WDOG** policy resets the system on missed refresh.

Steps: 1. Start from `boards/evkbimxrt1050/driver_examples/wdog/`.
2. Configure an interrupt time (`config.interruptTimeValue`) and timeout short enough for lab (e.g., ~2 s).
3. Intentionally skip `WDOG_Refresh()`; on interrupt, log urgent context (e.g., last error codes), then allow timeout reset.

Success criteria: Board resets; boot log shows reset reason as *WDOG timeout*.

Lab 5 — ARINC 429 parity/SSM injection via ADK-8582

Goal: Verify ARINC input filters reject malformed words.

Steps: 1. Start from `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/` for UART to ADK-8582. Wire EVKB LPUARTx TX/RX to ADK-8582 control UART.
2. Add `adk429_uart.c/.h` and `arinc429_word.c/.h` from §10.
3. Send a valid altitude word `ARINC429_BuildWord(...)` → ADK-8582 pass-through to DUT ARINC; your DUT monitor should accept it.
4. Resend with `parityOverride = EVEN` (ARINC requires odd parity).
5. Resend with `SSM = Failure` but plausible numeric field.
6. Resend with label set to a different parameter while keeping data field from the first (label discipline violation).

Success criteria: Each injected fault increments the right counters; invalid words are dropped; normal operation resumes after valid traffic.

Lab 6 — UART burst/overrun simulation (timing fault on interface)

Goal: Emulate bus saturation; ensure driver handles overrun.

Steps: 1. Modify `adk429_uart_send_burst()` to stream back-to-back commands with minimal inter-byte gap (use EDMA or a tight loop).
2. Verify LPUART RX overrun flags are detected (`kLPUART_RxOverrunFlag`), counters increment, and recovery occurs.

Success criteria: Overruns logged, no deadlocks; flow control (if used) is honoured.

---

10) Code (built on your SDK)

> **Conventions:** All examples include `board.h`, `clock_config.h`, `pin_mux.h`, and `fsl_debug_console.h`. UART/WDOG APIs follow the SDK style you can see in the examples listed earlier. Handlers print SCB status then either reset or loop in lab mode.

10.1 `fi_crc32.h / fi_crc32.c` — lightweight CRC32 & utilities

```
// fi_crc32.h
#pragma once
#include <stdint.h>
#include <stddef.h>

uint32_t FI_CRC32(const void *data, size_t len);
```

```
// fi_crc32.c
#include "fi_crc32.h"

uint32_t FI_CRC32(const void *data, size_t len)
{
    const uint8_t *p = (const uint8_t*)data;
    uint32_t crc = 0xFFFFFFFFu;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int k = 0; k < 8; ++k) {
            uint32_t mask = -(crc & 1u);
            crc = (crc >> 1) ^ (0xEDB88320u & mask);
        }
    }
    return ~crc;
}
```

10.2 `fi_mutate.h / fi_mutate.c` — bit flip & random delay

```c
// fi_mutate.h
#pragma once
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

void FI_FlipBit(uint8_t *buf, size_t len, uint32_t bitIndex);
void FI_FlipRandomBit(uint8_t *buf, size_t len);
void FI_RandomDelayUs(uint32_t minUs, uint32_t maxUs);
```

```c
// fi_mutate.c
#include "fi_mutate.h"
#include "fsl_trng.h"
#include "fsl_common.h"

extern TRNG_Type * const TRNG0; // provided by SDK device headers

static uint32_t rng32(void)
{
    uint32_t w;
    TRNG_GetRandomData(TRNG0, &w, sizeof(w));
    return w;
}

void FI_FlipBit(uint8_t *buf, size_t len, uint32_t bitIndex)
{
    uint32_t byte = bitIndex / 8u;
    uint8_t  mask = 1u << (bitIndex % 8u);
    if (byte < len) buf[byte] ^= mask;
}

void FI_FlipRandomBit(uint8_t *buf, size_t len)
{
    uint32_t totalBits = (uint32_t)len * 8u;
    FI_FlipBit(buf, len, rng32() % totalBits);
}

void FI_RandomDelayUs(uint32_t minUs, uint32_t maxUs)
{
    uint32_t span = (maxUs > minUs) ? (maxUs - minUs) : 0u;
    uint32_t delay = minUs + (span ? (rng32() % span) : 0u);
    /* Busy wait: for lab only. Use DWT cycle counter if enabled. */
```

```
    for (volatile uint32_t t = 0; t < delay * 100; ++t) { __NOP(); }
}
```

Hook: Initialize TRNG as shown in `boards/evkbimxrt1050/driver_examples/trng/random/trng_random.c` before using these utilities.

10.3 Exception/MPU: `fi_faults.c` — enable traps and verbose handlers

```
// fi_faults.c (compile as part of your project)
#include "fsl_debug_console.h"
#include "fsl_common.h"
#include "board.h"
#include "core_cm7.h"
#include "m-profile/armv7m_mpu.h"

/* Enable divide-by-zero trap for UsageFault tests */
void FI_EnableCoreTraps(void)
{
    SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;     // trap divide-by-zero
    SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk | SCB_SHCSR_BUSFAULTENA_Msk |
SCB_SHCSR_USGFAULTENA_Msk;
}

/* Create a read-only MPU region over [addr, addr+len) to trigger MemManage on
write */
void FI_MPU_EnableReadOnlyRegion(void *addr, uint32_t len)
{
    uint32_t base = (uint32_t)addr;
    /* region size must be power-of-two and aligned; round up to next power */
    uint32_t size = 32; while (size < len) size <<= 1;
    uint32_t region = 3; // pick a free region index for lab

    __DMB();
    ARM_MPU_Disable();
    MPU->RNR  = region;
    MPU->RBAR = (base & 0xFFFFFFE0u) | MPU_RBAR_VALID_Msk |
region; // 32B align minimal; adjust for your size
    /* AP=110 (RO for priv/unpriv), TEX/S/C/B=0, SRD=0, XN=0, SIZE=log2(size)-1
*/
    uint32_t rasr = (6u << MPU_RASR_AP_Pos) | (((31 - __CLZ(size)) - 1) <<
MPU_RASR_SIZE_Pos) | MPU_RASR_ENABLE_Msk;
    MPU->RASR = rasr;
    ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk);
    __DSB(); __ISB();
}
```

```
/* Minimal verbose fault handlers (lab mode). Avoid PRINTF in production ISRs.
*/
void HardFault_Handler(void)
{
    PRINTF("\n[HardFault] HFSR=0x%08lx CFSR=0x%08lx BFAR=0x%08lx
MMFAR=0x%08lx\r\n",
           SCB->HFSR, SCB->CFSR, SCB->BFAR, SCB->MMFAR);
    NVIC_SystemReset();
}
void MemManage_Handler(void)
{
    PRINTF("\n[MemManage] CFSR=0x%08lx MMFAR=0x%08lx\r\n", SCB->CFSR, SCB-
>MMFAR);
    NVIC_SystemReset();
}
void BusFault_Handler(void)
{
    PRINTF("\n[BusFault] CFSR=0x%08lx BFAR=0x%08lx\r\n", SCB->CFSR, SCB->BFAR);
    NVIC_SystemReset();
}
void UsageFault_Handler(void)
{
    PRINTF("\n[UsageFault] CFSR=0x%08lx\r\n", SCB->CFSR);
    NVIC_SystemReset();
}
```

Note: The exact **MPU** helper macros are from CMSIS included in your SDK under `CMSIS/Core/Include/`. The code above uses raw register fields and `ARM_MPU_Enable/Disable` from CMSIS for clarity.

10.4 WDOG policy snippet (based on SDK example)

```
#include "fsl_wdog.h"

static WDOG_Type * const s_wdog = WDOG1; // per board

void FI_WDOG_InitAndArm(void)
{
    wdog_config_t config;
    WDOG_GetDefaultConfig(&config);
    config.timeoutValue = 0x400u;          // short for lab
    config.interruptTimeValue = 0x200u;    // early warning
    WDOG_Init(s_wdog, &config);
}

void FI_WDOG_Service(void)
```

```
{
    WDOG_Refresh(s_wdog);
}
```

Compare with `boards/evkbimxrt1050/driver_examples/wdog/wdog.c` for interrupts and reset cause printing.

10.5 ARINC 429 word builder and ADK-8582 UART wrappers

```
// arinc429_word.h
#pragma once
#include <stdint.h>
#include <stdbool.h>

typedef enum { ARINC429_SSM_NORMAL=0, ARINC429_SSM_NCD=1, ARINC429_SSM_FT=2,
ARINC429_SSM_FAIL=3 } arinc429_ssm_t;

uint32_t ARINC429_BuildWord(uint8_t label, uint8_t sdi, uint32_t data21,
arinc429_ssm_t ssm, bool forceEvenParity);
```

```
// arinc429_word.c
#include "arinc429_word.h"

static inline uint8_t parity_odd32(uint32_t w)
{
    /* Returns 1 if odd number of ones in bits 0..30 */
    w ^= w >> 16; w ^= w >> 8; w ^= w >> 4; w ^= w >> 2; w ^= w >> 1;
    return (uint8_t)(~w & 1u);
}

uint32_t ARINC429_BuildWord(uint8_t label, uint8_t sdi, uint32_t data21,
arinc429_ssm_t ssm, bool forceEvenParity)
{
    data21 &= 0x1FFFFFu;  // 21 bits
    sdi    &= 0x3u;       // 2 bits
    uint32_t word = 0;
    /* Bit allocation (logical view): [31:30]=SSM, [29:11]=DATA(21), [10:9]=SDI,
[8:1]=LABEL, [32]=PARITY] */
    word |= ((uint32_t)ssm & 0x3u) << 30;
    word |= (data21 & 0x1FFFFFu) << 11;
    word |= ((uint32_t)sdi & 0x3u) << 9;
    word |= (uint32_t)label; // 8 bits
    /* Compute odd parity over bits 0..30. If forceEvenParity, flip it to be
wrong. */
    uint8_t odd = parity_odd32(word);
```

```
    uint8_t parity = forceEvenParity ? (odd ^ 1u) : odd;
    if (parity) word |= (1u << 31); // place in MSB logically; instrument
handles wire order
    return word;
}
```

```
// adk429_uart.h
#pragma once
#include <stdint.h>
#include <stdbool.h>
#include "fsl_lpuart.h"

status_t ADK429_InitUart(LPUART_Type *base, uint32_t baudrate);
status_t ADK429_SendWord(LPUART_Type *base, uint8_t label, uint8_t sdi,
uint32_t data21, int ssm, bool badParity);
status_t ADK429_SendBurst(LPUART_Type *base, const uint32_t *words, uint32_t
count);
```

```
// adk429_uart.c  (illustrative; adapt to your ADK-8582 command set)
#include "adk429_uart.h"
#include "arinc429_word.h"
#include <stdio.h>

status_t ADK429_InitUart(LPUART_Type *base, uint32_t baudrate)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baudrate;
    cfg.enableRx = true; cfg.enableTx = true;
    return LPUART_Init(base, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));
}

static status_t send_ascii(LPUART_Type *base, const char *s)
{
    return LPUART_WriteBlocking(base, (const uint8_t*)s, (size_t)strlen(s));
}

status_t ADK429_SendWord(LPUART_Type *base, uint8_t label, uint8_t sdi,
uint32_t data21, int ssm, bool badParity)
{
    uint32_t w = ARINC429_BuildWord(label, sdi, data21, (arinc429_ssm_t)ssm,
badParity);
    /* Example ASCII command — replace with actual ADK-8582 format: "TX
<hexword>\r\n" */
    char line[32];
```

```
    (void)snprintf(line, sizeof(line), "TX %08lX\r\n", (unsigned long)w);
    return send_ascii(base, line);
}

status_t ADK429_SendBurst(LPUART_Type *base, const uint32_t *words, uint32_t
count)
{
    for (uint32_t i = 0; i < count; ++i) {
        char line[32];
        (void)snprintf(line, sizeof(line), "TX %08lX\r\n", (unsigned
long)words[i]);
        status_t s = send_ascii(base, line);
        if (s) return s;
    }
    return kStatus_Success;
}
```

> **Important:** The `TX` ASCII is a placeholder. Replace with the actual **ADK-8582** command set (word rate, channel select, loop, etc.). The `LPUART_WriteBlocking` usage follows the style in `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/lpuart_interrupt_transfer.c`.

10.6 Putting it together — `main.c` (sketch)

```
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"
#include "fsl_lpuart.h"
#include "fi_crc32.h"
#include "fi_mutate.h"

#define LAB_MODE 1

static uint8_t g_buf[32];
static volatile uint32_t g_crcOkCnt = 0, g_crcBadCnt = 0, g_faultCnt = 0;

int main(void)
{
    BOARD_ConfigMPU();     // generated by MCUXpresso config tool; keeps caches
coherent
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    PRINTF("\r\nFault Injection Lab start\r\n");
```

```
    FI_EnableCoreTraps();
    FI_WDOG_InitAndArm();

    /* Fill buffer and compute CRC */
    for (int i=0;i<32;i++) g_buf[i]=(uint8_t)i;
    uint32_t crc = FI_CRC32(g_buf, sizeof g_buf);

    /* Inject a random bit flip */
    FI_FlipRandomBit(g_buf, sizeof g_buf);
    uint32_t crc2 = FI_CRC32(g_buf, sizeof g_buf);
    if (crc2 != crc) {
        ++g_crcBadCnt;
        PRINTF("CRC mismatch as expected after bit flip (good=%08x bad=%08x)
\r\n", crc, crc2);
    } else {
        ++g_crcOkCnt; // extremely unlikely
    }

#if LAB_MODE
    /* Provoke divide-by-zero UsageFault */
    volatile int z = 0; volatile int x = 10 / z; (void)x;
#endif

    while (1) {
        FI_WDOG_Service();
        // Your ARINC test sequences here, e.g., ADK429_SendWord(...)
    }
}
```

---

11) Test design patterns to maximise coverage

- **One-fault-at-a-time:** Isolate fault types to attribute detection to a single mechanism.
- **Seeded randomness:** When using TRNG for bit selection, also log a *software seed* so the sequence can be reproduced (or capture the flipped index).
- **State checkpointing:** On fault, dump minimal state (active label, last counters, `SCB` fault registers) into a reserved SRAM log (guard with MPU **execute-never** to avoid code execution from logs).
- **Watchdog policy table:** Define when to *continue*, when to *re-initialise a subsystem*, and when to *reset the whole system*.
- **Graceful degradation:** For ARINC reception, drop to *last-known-good* with SSM propagated downstream.
- **Fault campaigns:** Automate sequences: parity→SSM→label→burst→timing to prove monitors don't interfere with each other.

---

12) Best practices (avionics-grade)

1. **Traceability to requirements:** Each injected fault maps to a low-level requirement (LLR) and a safety objective from the **SSA (System Safety Assessment)**.
2. **Segregation:** Keep FI utilities out of production by `#if LAB_MODE` and separate libraries/binaries.
3. **Deterministic logging:** Use ring buffers; don't `PRINTF` inside tight ISRs in production (acceptable in lab).
4. **Re-entrancy & priority:** Be mindful that faults often occur at awkward times; test with interrupts enabled and realistic **NVIC** priorities.
5. **MPU everywhere:** Protect stacks, constant tables (RO), and a *poison* region at address 0 to trap null dereferences.
6. **Data guards:** CRC or **HMAC (Hash-Based Message Authentication Code)** on critical tables; dual-compute then compare for **FIH**-style robustness of safety monitors.
7. **Watchdog reason codes:** On boot, read/reset reason and surface it to maintenance logs (compare WDOG examples).
8. **Interface discipline:** For ARINC 429, enforce label allow-lists, SDI checks, SSM semantics, and rate policing before values reach the application.
9. **Bench safety:** Current-limit supplies, use IO clamps; have a physical kill-switch; isolate ARINC wiring.
10. **Certification evidence:** Keep *procedures*, *objective evidence* (test results, logs), and *independence* per **DO-178C**. If tools drive fault scripts, assess under **DO-330 (Software Tool Qualification Considerations)**.

---

13) What to collect as evidence

- Test procedure IDs linked to LLRs.
- The exact fault parameters (e.g., bit index, label, SSM, parity flag) and timestamps.
- SCB register dumps for exception faults.
- WDOG reset cause and reboot times.
- Before/after counters for ARINC input filters.
- Pass/fail verdict and rationale.

---

14) Troubleshooting

- **No MemManage fault when writing RO region?** Ensure region size/alignment follow MPU rules and `SCB->SHCSR` enables MemManage.
- **PRINTF not visible in handlers?** Use SWO/ITM or buffer the log to SRAM and print on next boot.
- **ADK-8582 not reacting?** Verify UART baud, newline, command syntax, and instrument channel enable.
- **Board resets unexpectedly during FI runs?** Likely WDOG timeout—tune `timeoutValue` or ensure service calls in your loops.

---

15) Wrap-up

You now have a progressive set of **SWIFI** and **ARINC 429** interface-level injections on i.MX RT1050, built directly on the **MCUXpresso SDK** you provided. The labs aim to exercise the **exception architecture (HardFault/MemManage/BusFault/UsageFault)**, **MPU protections**, **WDOG policy**, and **ARINC input discipline**. With these in place, you can build automated *fault campaigns* and gather certification-ready evidence.

---

16) Appendix — Mapping to SDK examples (for quick copy-start)

- **LPUART (for ADK-8582 control):** `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/`
- **WDOG/RTWDOG:** `boards/evkbimxrt1050/driver_examples/wdog/` and `.../rtwdog/`
- **TRNG (for randomized bit flips):** `boards/evkbimxrt1050/driver_examples/trng/random/`
- **EDMA (for burst stress):** `boards/evkbimxrt1050/driver_examples/edma/`
- **PWM Fault demo (peripheral fault concept):** `boards/evkbimxrt1050/demo_apps/pwm_fault/`

Keep the **acronym expansion policy** in your documents: expand at first use (as done here).