Compile-time and Runtime Fault Injection on i.MX RT1050

Target board: NXP i.MX RT1050 EVKB (MIMXRT1050, Arm Cortex-M7)

SDK baseline: MCUXpresso SDK for EVKB-IMXRT1050 (use the provided archive). All hands-on code in this module is derived from the SDK's reference projects (e.g., `boards/evkbimxrt1050/demo_apps/hello_world` and `boards/evkbimxrt1050/driver_examples/lpuart/*`).

Avionics interface for this module's ARINC-429 exercises: i.MX RT1050 EVKB ↔ **ADK-8582** via **UART**. We will build a UART shim that lets us intentionally corrupt ARINC-429 words before they are handed to an external adapter. (You can adapt the UART framing to your converter's documented command set.)

---

1) Why Fault Injection (FI) in avionics?

Fault Injection (FI) is the deliberate introduction of errors to evaluate the robustness of software and hardware. In civil avionics, FI supports objectives related to **robustness, error detection, and fault containment** in systems developed under **DO-178C** (Software Considerations in Airborne Systems and Equipment Certification) and **DO-254** (Design Assurance Guidance for Airborne Electronic Hardware). FI does **not** replace verification required by standards; instead, it strengthens confidence that implemented mitigations (timeouts, retries, range checks, mode fallbacks) behave as intended when things go wrong.

Key outcomes you should be able to demonstrate by the end of this module: - Provoke errors at **compile time** (by instrumenting the build) and at **runtime** (by controlled triggers) without permanently modifying production code paths. - Show that the system **detects**, **contains**, and **recovers** from injected faults, or enters a **defined safe state**. - Make injection **deterministic** and **reproducible** with seeds, feature flags, and precise triggers.

---

2) Core definitions (used consistently throughout)

- **MCU:** Microcontroller Unit. Here, NXP i.MX RT1050.
- **HAL:** Hardware Abstraction Layer. In this module, NXP's MCUXpresso drivers (e.g., `fsl_lpuart.h`).
- **UART:** Universal Asynchronous Receiver/Transmitter. On RT1050, Low-Power UART (**LPUART**).
- **DMA:** Direct Memory Access.
- **ISR:** Interrupt Service Routine.
- **MPU:** Memory Protection Unit (Armv7-M, used for generating precise **MemManage** faults).
- **FI:** Fault Injection.
- **SWIFI:** Software-Implemented Fault Injection (runtime injection via software).

---

3) Taxonomy of fault injection methods covered here

We structure FI methods into two families you will implement on the EVKB:

3.1 Compile-time fault injection

Compile-time methods modify the **built artifact** so the fault mechanism is present in the image, gated by macros or weak symbols: 1. **Preprocessor gating:** Insert hooks like `FI_POINT(id)` that compile to no-ops in production but to injections in FI builds (`-DFI_ENABLE`). 2. **Link-time interposition (weak symbols):** Provide `__attribute__((weak))` hooks and override them in a test build to change function returns (e.g., force `LPUART_WriteBlocking()` to time out). 3. **Linker wrapping:** Use the GNU linker's `--wrap=symbol` to route calls through a wrapper that can corrupt parameters or change return codes before calling the real function. 4. **Aspect-style instrumentation:** Centralize instrumentation macros so the *call sites* remain readable while the injection policy is swapped per build configuration.

3.2 Runtime fault injection (SWIFI)

Runtime methods introduce faults while the system is running: 1. **Time-based triggers:** Inject after *N* milliseconds or *M* loop iterations using `SysTick`. 2. **Event-based triggers:** Inject only when a predicate is true (e.g., "next UART TX of ARINC label 0x123"). 3. **Randomized triggers:** Inject based on a pseudorandom threshold with a fixed seed for reproducibility. 4. **Exception injection:** Deliberately cause **HardFault**, **BusFault**, **UsageFault**, or **MemManage** faults and verify handler behavior/logging. 5. **Memory corruption:** Flip bits in SRAM buffers, DMA descriptors, or control structures and verify detection (e.g., CRC/guards). 6. **Peripheral behavior emulation:** Force HAL calls to report transient/busy/error and confirm retry/backoff logic.

---

4) Generic example (non-avionics): robust UART send with compile-time & runtime FI

Scenario: An application sends telemetry over UART using the MCUXpresso LPUART driver (`fsl_lpuart.h`). We want to verify: - The app retries on `kStatus_LPUART_TxBusy`. - The app times out and escalates to a safe mode if the driver never drains. - The app logs corrupted bytes and refuses to transmit when a CRC check fails.

We will 1) instrument the build with **compile-time hooks** and 2) add a **runtime trigger** so that 1 in 20 transmissions experiences a forced "busy" or byte corruption.

---

5) Avionics example: ARINC-429 word corruption via UART shim

ARINC-429 (Aeronautical Radio, Incorporated Specification 429) is a unidirectional data bus transmitting **32-bit words** at **12.5 kbps** or **100 kbps**. A word is structured as: **Label (bits 1–8)**, **Source/Destination Identifier – SDI (bits 9–10)**, **Data (bits 11–29)**, **Sign/Status Matrix – SSM (bits 30–31)**, and **Parity (bit 32, odd parity)**.

Use case: The EVKB prepares ARINC-429 words and forwards them via **UART** to an **ADK-8582** adapter that drives the physical ARINC line. We will inject: - **Parity faults:** Toggle bit-32 (odd parity) to produce invalid words. - **Label faults:** Substitute unexpected **Label** values to verify filter behavior. - **Burst faults:** Corrupt consecutive bytes to emulate noise. - **Timing faults:** Delay or duplicate words to test rate/sequence monitors.

Your UART framing towards the adapter can be binary or ASCII command oriented; the shim below assumes a simple binary frame `[SOF 0xA5][32-bit word LSB..MSB][CRC-8]`, which you can replace with the ADK-8582's actual protocol without changing the FI hooks.

---

6) Minimal fault-injection framework (drop-in for SDK projects)

Create four files next to an SDK sample (e.g., copy from `hello_world` and add these sources):

6.1 `fi_config.h` — build-time switches and probabilities

```
#ifndef FI_CONFIG_H
#define FI_CONFIG_H

// Global enable (compile-time). Add -DFI_ENABLE in your FI build configuration.
#ifndef FI_ENABLE
#  define FI_ENABLE 0
#endif

// Default probability in percent for probabilistic injections.
#ifndef FI_DEFAULT_PROB_PCT
#  define FI_DEFAULT_PROB_PCT 5  // 5% by default
#endif

// Seed to make randomized injections reproducible.
#ifndef FI_SEED
#  define FI_SEED 0x4D534543u  // "MSEC"
#endif

// Optional: narrow down to a specific feature under test.
#ifndef FI_FEATURE_MASK
#  define FI_FEATURE_MASK 0xFFFFFFFFu
#endif

// Feature bits (you can add more as needed)
#define FI_F_UART_TX_BUSY   (1u << 0)
#define FI_F_UART_CORRUPT   (1u << 1)
#define FI_F_ARINC_PARITY   (1u << 2)
#define FI_F_ARINC_LABEL    (1u << 3)
#define FI_F_MEM_BITFLIP    (1u << 4)
```

```
#define FI_F_EXCEPTIONS     (1u << 5)

#endif // FI_CONFIG_H
```

6.2 `fi.h` — lightweight API and instrumentation macros

```
#ifndef FI_H
#define FI_H
#include <stdint.h>
#include <stdbool.h>
#include "fi_config.h"

#ifdef __cplusplus
extern "C" {
#endif

// Initialize the runtime injector (seed PRNG, start SysTick trigger if
desired).
void FI_Init(void);

// Enable/disable at runtime.
void FI_SetEnabled(bool en);
bool FI_IsEnabled(void);

// Configure feature mask and probability (0..100%).
void FI_SetMask(uint32_t mask);
void FI_SetProbability(uint8_t pct);

// Deterministic pseudo-random in [0, 100).
uint8_t FI_RandPct(void);

// Return true if the injection for 'feature' should fire *now*.
bool FI_ShouldFire(uint32_t feature);

// Memory corruption helpers.
void FI_BitFlip8(uint8_t *p, uint8_t mask);
void FI_BitFlipRange(void *buf, uint32_t len, uint32_t everyN);

// Exception injection helpers (UsageFault, BusFault, MemManage, HardFault)
void FI_Inject_DivByZero(void);
void FI_Inject_UnalignedAccess(void);
void FI_Inject_MemFault(void);

// Macros to keep call sites tidy.
#if FI_ENABLE
#  define FI_POINT(feature, code_block) do { if (FI_ShouldFire((feature)))
```

```
{ code_block; } } while(0)
#else
#  define FI_POINT(feature, code_block) do { (void)(feature); } while(0)
#endif

#ifdef __cplusplus
}
#endif

#endif // FI_H
```

6.3 `fi.c` — implementation (uses CMSIS/SDK facilities only)

```
#include "fi.h"
#include "fsl_device_registers.h"  // for SCB, SysTick
#include <string.h>

static volatile uint32_t fi_mask = FI_FEATURE_MASK;
static volatile uint8_t  fi_prob_pct = FI_DEFAULT_PROB_PCT;
static volatile bool     fi_enabled = (FI_ENABLE != 0);

static uint32_t lcg_state = FI_SEED;
static inline uint32_t lcg_next(void) {
    // 32-bit LCG (Numerical Recipes constants)
    lcg_state = lcg_state * 1664525u + 1013904223u;
    return lcg_state;
}

void FI_Init(void) {
    lcg_state = (FI_SEED ^ 0xA5A5A5A5u) + 1u;
}

void FI_SetEnabled(bool en) { fi_enabled = en; }
bool FI_IsEnabled(void) { return fi_enabled; }

void FI_SetMask(uint32_t mask) { fi_mask = mask; }
void FI_SetProbability(uint8_t pct) { fi_prob_pct = pct; }

uint8_t FI_RandPct(void) {
    return (uint8_t)(lcg_next() % 100u);
}

bool FI_ShouldFire(uint32_t feature) {
    if (!fi_enabled) return false;
    if ((fi_mask & feature) == 0u) return false;
    return (FI_RandPct() < fi_prob_pct);
```

```
}

void FI_BitFlip8(uint8_t *p, uint8_t mask) { *p ^= mask; }

void FI_BitFlipRange(void *buf, uint32_t len, uint32_t everyN) {
    if (!fi_enabled || everyN == 0u) return;
    uint8_t *b = (uint8_t *)buf;
    for (uint32_t i=0; i<len; ++i) {
        if ((i % everyN) == 0u) { b[i] ^= (uint8_t)(1u << (lcg_next() & 7u)); }
    }
}

// === Exception injections ===
void FI_Inject_DivByZero(void) {
    // Trap divide-by-zero to UsageFault
    SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
    volatile int x = 1;
    volatile int y = 0;
    volatile int z = x / y; (void)z;
}

void FI_Inject_UnalignedAccess(void) {
    SCB->CCR |= SCB_CCR_UNALIGN_TRP_Msk;
    uint32_t __attribute__((aligned(4))) word = 0x12345678u;
    uint8_t *pb = (uint8_t *)&word;
    // Force an unaligned 32-bit read
    volatile uint32_t bad = *(uint32_t *)(pb + 1);
    (void)bad;
}

void FI_Inject_MemFault(void) {
    // Configure MPU to forbid access to a small SRAM subregion and then touch
it.
    // Region setup is platform-specific; ensure MPU is present and enabled.
    // Minimal example left to lab #3 where we program a no-access region.
    extern void FI_MPU_SetupDenyRegion(void);
    FI_MPU_SetupDenyRegion();
    volatile uint32_t *forbidden = (uint32_t *)0x20200000u; // adjust to the
deny region you program
    volatile uint32_t v = *forbidden; (void)v;
}
```

Note: We keep `FI_Inject_MemFault()` as a placeholder that calls a function you implement in Lab 3 to program a deny region with the **MPU** (Memory Protection Unit). The i.MX RT1050's Arm Cortex-M7 MPU supports 8 regions with sub-regions.

6.4 `uart_shim.c` — interpose LPUART transmit paths

Two strategies are provided: **call-site shim** and **linker wrap**.

6.4.1 Call-site shim (simpler; no linker flags required)

Replace direct calls to `LPUART_WriteBlocking()` in your app with `UART_FI_WriteBlocking()` during FI builds.

```
#include "fi.h"
#include "fsl_lpuart.h"

status_t UART_FI_WriteBlocking(LPUART_Type *base, const uint8_t *data, size_t
length) {
    // 1) Occasionally report TX busy (compile-time + runtime controlled)
    FI_POINT(FI_F_UART_TX_BUSY, return kStatus_LPUART_TxBusy;);

    // 2) Occasionally corrupt payload bytes
    FI_POINT(FI_F_UART_CORRUPT, FI_BitFlipRange((void *)data, (uint32_t)length,
7););

    return LPUART_WriteBlocking(base, data, length);
}
```

6.4.2 Linker wrapping (advanced; keeps app code untouched)

If you prefer not to modify call sites, the GNU linker can redirect all calls to `LPUART_WriteBlocking` through a wrapper.

Add to **linker flags** of your FI build: `-Wl,--wrap=LPUART_WriteBlocking`

Provide this wrapper:

```
#include "fi.h"
#include "fsl_lpuart.h"

extern status_t __real_LPUART_WriteBlocking(LPUART_Type *base, const uint8_t
*data, size_t length);
status_t __wrap_LPUART_WriteBlocking(LPUART_Type *base, const uint8_t *data,
size_t length) {
    FI_POINT(FI_F_UART_TX_BUSY, return kStatus_LPUART_TxBusy;);
    FI_POINT(FI_F_UART_CORRUPT, FI_BitFlipRange((void *)data, (uint32_t)length,
11););
```

```
    return __real_LPUART_WriteBlocking(base, data, length);
}
```

---

7) ARINC-429 helpers and FI points

Create an encoder/forwarder that builds a 32-bit ARINC-429 word and sends it over UART in a simple binary frame. The FI points let you toggle parity, label, and burst faults.

7.1 `arinc429.h`

```
#ifndef ARINC429_H
#define ARINC429_H
#include <stdint.h>
#include <stdbool.h>
#include "fi.h"

typedef struct {
    uint8_t  label;  // bits 1..8
    uint8_t  sdi;    // bits 9..10 (0..3)
    uint32_t data;   // bits 11..29 (19-bit)
    uint8_t  ssm;    // bits 30..31 (0..3)
} arinc429_word_t;

uint32_t ARINC429_Pack(const arinc429_word_t *w, bool force_bad_parity);
status_t ARINC429_SendWord(LPUART_Type *base, const arinc429_word_t *w);

#endif
```

7.2 `arinc429.c`

```
#include "arinc429.h"
#include "fsl_lpuart.h"

static uint8_t crc8(const uint8_t *d, uint32_t n) {
    uint8_t c = 0x00u;
    for (uint32_t i=0;i<n;i++) {
        uint8_t b = d[i];
        for (int k=0;k<8;k++) {
            uint8_t mix = (c ^ b) & 0x01u;
            c >>= 1; if (mix) c ^=
0x8Cu;  // CRC-8 polynomial x^8 + x^5 + x^4 + 1
            b >>= 1;
        }
```

```
    }
    return c;
}

static uint8_t odd_parity_31(uint32_t v31) {
    // Return 1 if parity of 31 LSBs is even (to make overall odd), else 0.
    uint32_t x = v31;
    x ^= x >> 16; x ^= x >> 8; x ^= x >> 4; x &= 0xFu; // popcount mod 2 via
nibble parity
    uint8_t parity = (0x6996u >> x) & 1u; // parity of low nibble
    return (parity == 0u) ? 1u : 0u;
}

uint32_t ARINC429_Pack(const arinc429_word_t *w, bool force_bad_parity) {
    // Pack fields into 31 LSBs (bit 0 = Label bit1)
    uint32_t v = 0u;
    v |= ((uint32_t)(w->label) & 0xFFu) << 0;     // bits 0..7 (1..8)
    v |= ((uint32_t)(w->sdi)   & 0x03u) << 8;     // bits 8..9  (9..10)
    v |= ((uint32_t)(w->data)  & 0x7FFFFu) << 10; // bits 10..28 (11..29)
    v |= ((uint32_t)(w->ssm)   & 0x03u) << 29;    // bits 29..30 (30..31)

    uint8_t p = odd_parity_31(v);
    if (force_bad_parity) p ^= 1u;
    v |= ((uint32_t)p) << 31;                     // bit 31 (32)
    return v;
}

status_t ARINC429_SendWord(LPUART_Type *base, const arinc429_word_t *w) {
    bool bad_parity = false;
    arinc429_word_t temp = *w;

    // FI #1: label substitution
    FI_POINT(FI_F_ARINC_LABEL, temp.label ^= 0x1Fu);

    // FI #2: parity toggle
    FI_POINT(FI_F_ARINC_PARITY, bad_parity = true);

    uint32_t word = ARINC429_Pack(&temp, bad_parity);

    // Frame: [0xA5][word 32b LSB..MSB][CRC-8] (you can replace this to match
your adapter)
    uint8_t frame[1 + 4 + 1];
    frame[0] = 0xA5u;
    frame[1] = (uint8_t)(word & 0xFFu);
    frame[2] = (uint8_t)((word >> 8) & 0xFFu);
    frame[3] = (uint8_t)((word >> 16) & 0xFFu);
    frame[4] = (uint8_t)((word >> 24) & 0xFFu);
```

```
    frame[5] = crc8(&frame[1], 4);

    // Optional additional corruption for burst noise
    FI_POINT(FI_F_UART_CORRUPT, FI_BitFlipRange(frame, sizeof(frame), 2));

    return LPUART_WriteBlocking(base, frame, sizeof(frame));
}
```

---

8) Putting it together: main program (based on SDK `hello_world`)

Start from `boards/evkbimxrt1050/demo_apps/hello_world` and replace `hello_world.c` with the following `main.c`. This reuses the board init from the SDK (clock, pins, debug console).

```
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "board.h"
#include "app.h"
#include "fi.h"
#include "arinc429.h"
#include "fsl_lpuart.h"

#define ADAPTER_UART LPUART1  // adjust to the UART instance wired to ADK-8582

static void UART1_Init_115200(void) {
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = 115200U;
    cfg.enableTx = true;
    cfg.enableRx = true;
    LPUART_Init(ADAPTER_UART, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));
}

int main(void) {
    BOARD_InitHardware();      // from SDK project
    UART1_Init_115200();

    PRINTF("FI demo start\r\n");

    FI_Init();
#if FI_ENABLE
    FI_SetEnabled(true);
    FI_SetMask(FI_F_UART_TX_BUSY | FI_F_UART_CORRUPT | FI_F_ARINC_PARITY |
FI_F_ARINC_LABEL);
    FI_SetProbability(5);  // 5% injection rate
```

```
#else
    FI_SetEnabled(false);
#endif

    // Example ARINC word (Label 0x12, SDI 1, Data 0x12345, SSM 2)
    arinc429_word_t w = { .label=0x12u, .sdi=1u, .data=0x12345u, .ssm=2u };

    while (1) {
        status_t st = ARINC429_SendWord(ADAPTER_UART, &w);
        if (st == kStatus_LPUART_TxBusy) {
            PRINTF("UART busy, retrying...\r\n");
            continue; // minimal retry policy for demo purposes
        } else if (st != kStatus_Success) {
            PRINTF("UART error %d\r\n", (int)st);
        }

        // Periodically exercise exception injections (runtime)
        FI_POINT(FI_F_EXCEPTIONS, {
            PRINTF("Injecting divide-by-zero\r\n");
            FI_Inject_DivByZero();
        });

        SDK_DelayAtLeastUs(10000u, CLOCK_GetFreq(kCLOCK_CpuClk)); // 10 ms
    }
}
```

> **Board routing:** Check your EVKB schematic to choose the correct **LPUARTx** connected to your adapter interface. The SDK exposes helpers like `BOARD_InitPins()` and `BOARD_InitDebugConsole()` which configure the default console on `LPUART3` (Virtual COM). Keep your adapter on a *different* LPUART instance (e.g., `LPUART1` / `LPUART2`) to avoid console interference.

---

9) Hands-on labs

Lab 1 — Compile-time FI via preprocessor & weak hooks

Goal: Add FI points to driver calls and prove that your application survives sporadic busy/error returns.

Steps: 1. Copy `hello_world` into a new project **hello_world_fi**. 2. Add `fi_config.h`, `fi.h`, `fi.c`, and either `uart_shim.c` or the linker-wrap method. 3. Create a new **build configuration** named **FI** and add `-DFI_ENABLE=1` to compiler defines. Optionally add `-Wl,--wrap=LPUART_WriteBlocking` to linker flags. 4. Flash the image. Observe on the debug console: periodic “UART busy, retrying...”. Verify no system hang.

Success criteria: - With FI enabled, the app logs occasional injected conditions but continues running. With FI disabled, no spurious errors occur.

Lab 2 — Runtime FI policy: time-based and event-based firing

Goal: Make injections fire under precise conditions to reproduce missed-edge bugs.

Steps: 1. Extend `fi.c` to count transmitted ARINC words. Inject only on the 10th, then again after 100 ms windows (use `SysTick->VAL` or `SDK_DelayAtLeastUs`). 2. Add a CLI on the **debug console** (uses `GETCHAR()` / `PUTCHAR()` from `fsl_debug_console.h`) to set mask, probability, and seed at runtime: - `en 1` → enable FI - `msk FFFFFFFF` → set mask - `pct 1..100` → set probability 3. Demonstrate deterministic reproduction by setting the same seed and showing identical injection pattern across runs.

Success criteria: - Repeatability with the same seed. Clear evidence that policy changes alter observed behavior without reflashing.

Lab 3 — Exception injection with MPU (MemManage), UsageFault, BusFault

Goal: Prove fault handlers are installed, log context cleanly, and system reboots to a defined state.

Steps: 1. Implement `FI_MPU_SetupDenyRegion()` to configure one MPU region as **no-access** inside OCRAM, then perform a read to trigger **MemManage**. 2. Add application **HardFault_Handler**, **MemManage_Handler**, **BusFault_Handler**, **UsageFault_Handler** that print a short signature and reset via the **Watchdog** after logging. 3. For **UsageFault**, call `FI_Inject_DivByZero()` and `FI_Inject_UnalignedAccess()`.

Reference snippet (handlers):

```
void HardFault_Handler(void)      { PRINTF("HardFault\\r\\n");
NVIC_SystemReset(); }
void MemManage_Handler(void)      { PRINTF("MemManage\\r\\n");
NVIC_SystemReset(); }
void BusFault_Handler(void)       { PRINTF("BusFault\\r\\n");
NVIC_SystemReset(); }
void UsageFault_Handler(void)     { PRINTF("UsageFault\\r\\n");
NVIC_SystemReset(); }
```

Success criteria: - Each injected exception is caught by the correct handler; the system resets deterministically.

Lab 4 — ARINC-429 corruption toward ADK-8582 via UART

Goal: Validate ARINC input filtering and parity checking on the receiving side by injecting realistic faults.

Steps: 1. Wire EVKB LPUARTx TX/RX to ADK-8582 UART (level-compatible) per the adapter's integration guide. 2. Replace the simple `[0xA5][word][CRC8]` framing in `ARINC429_SendWord()` with the adapter's documented UART command (e.g., a "raw word TX" command). Keep the **FI points** as written. 3. Send a known stream of words (e.g., airspeed, baro). Enable FI bits `FI_F_ARINC_PARITY | FI_F_ARINC_LABEL | FI_F_UART_CORRUPT` with `FI_SetProbability(10)`. 4. On the receiver/analyzer, note rejected frames for bad parity, label filter hits, and CRC mismatches.

Success criteria: - Receiver reports parity/label/timing faults at the expected injection rate. Your transmitter logs when injections were performed.

---

10) Advanced compile-time techniques (optional for further study)

- **Mutation toggles:** Provide switchable variations of boundary checks (e.g., `<` vs `<=`) behind `#if FI_ENABLE` to verify tests catch off-by-one issues.
- **Link-time layering:** Wrap not only LPUART but also `fsl_i2c.h`, `fsl_spi.h`, and networking (if using LWIP) to emulate peripheral timeouts.
- **Deterministic DMA interference:** Under FI, preload a DMA channel with a benign transfer overlapping a buffer to test data coherency management.

---

11) Best practices for FI in airborne systems development

1. **Strict build separation:** Maintain **Release (flight)**, **Debug**, and **FI/Test** configurations. Enforce that FI symbols cannot be enabled in Release through CI checks.
2. **Reproducibility:** Always log the **seed**, **feature mask**, **probability**, and a **monotonic counter** of injections.
3. **Tight scope:** Inject at **well-defined boundaries** (API shims, message packers) rather than arbitrary scattering that is hard to reason about.
4. **Safe-state assurance:** Before enabling FI, ensure reset/recovery paths and **Watchdog** (WDOG) are active so any deadlock is recovered.
5. **Telemetry:** Print a short tag when an injection fires, e.g., `FI[ARINC_PARITY]#123`. Keep logs rate-limited.
6. **Deterministic time base:** Use `SysTick` or a hardware timer to timestamp injections when timing is relevant.
7. **Hardware protection:** Use **MPU** to confine corruptions; never write outside test buffers. Never run FI while connected to **live aircraft buses**.
8. **Documentation:** Capture FI configuration with each test run so results are traceable to a precise software state (commit hash, build config, seed).

---

12) Assessment checklist (what to demonstrate)

- Show compile-time FI hooks do not change Release behavior (binary diff acceptable within build reproducibility constraints).
- Show runtime FI produces controlled errors (busy returns, corrupted frames) and that your app's **error handling** responds correctly.
- Demonstrate **exception handlers** capture injected faults and that the system resets cleanly.
- In the ARINC-429 lab, demonstrate that receiving equipment rejects bad parity/labels and that send-side logs match the receiver's fault counts.

---

13) Appendix — notes on mapping to MCUXpresso SDK examples

- `hello_world`: Provides `BOARD_InitHardware()`, `BOARD_InitDebugConsole()`, and a working console on `LPUART3`.
- `driver_examples/lpuart/*`: Use these as reference for LPUART configuration and interrupt/EDMA patterns if you need higher throughput for the ARINC stream.
- Header locations (from the SDK): `devices/MIMXRT1052/drivers/fsl_lpuart.h`, `devices/MIMXRT1052/utilities/debug_console/fsl_debug_console.h`.

  Keep your FI additions in separate files (`fi.*`, `arinc429.*`, `uart_shim.*`) so production code can exclude them via the build system.

End of module