Training Module: Software Fault Injection — Data Corruption & API Failures

Target hardware: NXP i.MX RT1050 EVKB (EVKB-IMXRT1050)
SDK baseline: SDK_25_06_00_EVKB-IMXRT1050
Avionics link for labs: EVKB-IMXRT1050 LPUART ↔ ADK-8582 (ARINC 429 development kit) over UART

---

1) Why this module exists

Software Fault Injection (SFI) is the deliberate introduction of faults into a running program to validate **Fault Detection, Isolation and Recovery (FDIR)** mechanisms and to harden error-handling paths that rarely execute in nominal testing. In avionics, this supports objectives typically derived from **ARP4754A** (Guidelines for Development of Civil Aircraft and Systems) and **DO-178C** (Software Considerations in Airborne Systems and Equipment Certification), by demonstrating that the software behaves safely when inputs are corrupted or when lower-layer **Application Programming Interface (API)** calls fail.

This module builds a practical SFI harness you can embed into bare-metal firmware on the **i.MX RT1050**. You will exercise two fault classes end-to-end:

1. **Data corruption**: bit-flips and malformed frames on internal buffers and on the UART receive path.
2. **API failures**: synthetic error returns, timeouts, and partial operations injected at driver boundaries.

The labs culminate in an avionics scenario where ARINC 429 words from an external **ARINC 429** (Aeronautical Radio, Incorporated) link are fed to the EVKB through a UART bridge (ADK-8582), and software corruption/failure is injected to verify monitors and fail-safe behaviors.

---

2) Foundations and precise definitions

Before we inject anything, align on terminology used in safety engineering:

- **Fault**: the hypothesized root cause introduced in the system (e.g., a flipped bit in a buffer, or an API returning a failure code).
- **Error**: the part of the system state that becomes incorrect because of a fault (e.g., a corrupted ARINC 429 word in the receive ring buffer).
- **Failure**: an externally observable deviation of service (e.g., the application outputs an invalid attitude to the bus, or crashes).

SFI (Software Fault Injection) is the act of adding code that creates *controlled* faults so that you can observe whether your detection and recovery logic prevents those errors from escaping as failures.

API (Application Programming Interface) failure here means the *software* boundary between your application and a driver or service (e.g., NXP LPUART driver). We synthetically return error codes, truncate transfers, or delay/tamper with completion notifications to exercise retry/timeout paths.

ARINC 429 is a unidirectional, self-clocking, two-wire databus standard widely used in transport aircraft. Each 32-bit word contains an **8-bit Label** (bits 1..8), **Source/Destination Identifier (SDI)** (bits 9..10), **19-bit Data** (bits 11..29), **Sign/Status Matrix (SSM)** (bits 30..31), and an odd **parity** bit (bit 32). In this module we will *not* generate electrical ARINC 429 directly from the EVKB; instead, an external ADK-8582 provides ARINC I/O and exposes a simple UART framing toward the EVKB, which we exercise and corrupt.

---

3) Fault models we will implement on EVKB-IMXRT1050

We keep the models simple, reproducible, and relevant to airborne systems:

3.1 Data corruption models

- **Single-bit flip**: invert one bit at a configured offset within a buffer (typical Single Event Upset).
- **Multi-bit burst**: invert N random bits within a packet or ring-buffer window.
- **Stuck-at**: force a bit to 0 or 1 regardless of written value.
- **Field-aware tamper**: for ARINC 429 frames: mis-label (change Label), corrupt **Sign/Status Matrix (SSM)**, or flip **parity**.

3.2 API failure models

- **Synchronous error returns**: driver function returns `kStatus_Fail`, `kStatus_InvalidArgument`, or a synthetic code while doing nothing.
- **Busy/partial operations**: nonblocking send reports `kStatus_LPUART_TxBusy` or only transfers part of the buffer.
- **Timeouts**: completion IRQ or DMA callback is intentionally suppressed or delayed beyond application watchdog thresholds.

All models are **deterministically controllable** by a small runtime harness with a seedable pseudo-random number generator (PRNG). You can run campaigns that are *repeatable* for debugging and certification evidence.

---

4) Architecture of the SFI harness

The harness is a tiny library (`fi.h`/`fi.c`) compiled into the firmware with `#define SFI_ENABLED 1`. A **Fault Site** is a named point in code where a fault *might* be injected. Each site has a trigger policy (every Nth hit, probability p, one-shot after count K, or active within a window), an optional parameter (e.g., how many bits to flip), and a per-site PRNG state so that sites are independent.

The harness provides three primitives:

1. `bool FI_ShouldFault(fi_site_t *s)`: increments a site counter and decides whether to fire.
2. `void FI_CorruptBytes(uint8_t *buf, size_t len, uint32_t nbits)`: flips `nbits` in place using the site PRNG.
3. **Driver wrappers**: thin functions/macros that *either* call into SDK drivers *or* synthesize an error/partial behavior when `FI_ShouldFault` fires.

Every injection emits an **event record** into a small ring buffer (site name, monotonic counter, argument, first few bytes of the affected buffer). A shell prompt on the debug UART dumps this ring on demand so you retain evidence of what was injected and when.

4.1 Observability on the EVKB

- **Console**: the EVKB's default debug UART (LPUART1 in the SDK examples) prints injection events and accepts simple commands (e.g., `fi arm RX_BURST every 100 bits=3`).
- **LED**: the board user LED (GPIO1 pin 9 via `BOARD_USER_LED_GPIO/BOARD_USER_LED_PIN`) is toggled on each injection and set solid if the application detects data corruption or an API error.
- **Watchdog**: use RTWDOG or WDOG1 to demonstrate that unhandled failure paths lead to a controlled reset.

---

5) Implementation: the fault injector library (fi.h / fi.c)

> **Note:** Code below is written to the NXP SDK style and builds directly inside an SDK example application. Headers from the SDK used here: `fsl_common.h`, `fsl_gpio.h`, `fsl_lpuart.h`, and the board layer (`board.h`).

5.1 `fi.h`

```c
#ifndef FI_H
#define FI_H

#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifndef SFI_ENABLED
#define SFI_ENABLED 1
#endif

/* Trigger modes */
```

```c
typedef enum {
    FI_MODE_NEVER = 0,          /* disabled */
    FI_MODE_EVERY_N,            /* fire every Nth hit */
    FI_MODE_PROB_1_IN_N,        /* fire with probability 1/N */
    FI_MODE_AFTER_K_THEN_ONCE,  /* fire once after K hits */
    FI_MODE_WINDOW              /* fire when K1 <= hit < K2 */
} fi_mode_t;

/* Per-site descriptor */
typedef struct {
    const char *name;
    fi_mode_t mode;
    uint32_t paramN;     /* N for EVERY_N/PROB; K for AFTER_K; K1 for WINDOW */
    uint32_t paramM;     /* unused except: WINDOW end K2; or bits to flip when
used for data sites */
    uint32_t hits;       /* incremented on every probe */
    uint32_t rng;        /* xorshift32 state (non-zero) */
} fi_site_t;

/* Event log entry */
typedef struct {
    const char *name;
    uint32_t hits;
    uint32_t arg0;
    uint32_t arg1;
    uint8_t  peek[8];
} fi_event_t;

void FI_Init(uint32_t global_seed);
void FI_Arm(fi_site_t *s, const char *name, fi_mode_t mode, uint32_t pN,
uint32_t pM, uint32_t seed);
bool FI_ShouldFault(fi_site_t *s);
void FI_CorruptBytes(fi_site_t *s, uint8_t *buf, size_t len, uint32_t nbits);

/* Optional helpers for driver/API sites */
bool FI_ShouldFail_API(fi_site_t *s, int32_t *out_synthetic_status);

/* Event logging */
void FI_Log(const char *name, uint32_t hits, uint32_t a0, uint32_t a1, const
uint8_t *peek, size_t peek_len);
size_t FI_Dump(char *out, size_t maxlen); /* returns bytes written into out */

#ifdef __cplusplus
}
#endif

#endif /* FI_H */
```

5.2 fi.c

```
#include "fi.h"
#include <string.h>

#ifndef FI_MAX_EVENTS
#define FI_MAX_EVENTS 128u
#endif

static fi_event_t s_events[FI_MAX_EVENTS];
static volatile uint32_t s_evt_head;
static uint32_t s_global_seed = 0xA5A5A5A5u;

static inline uint32_t xorshift32(uint32_t *state)
{
    uint32_t x = (*state) ? *state : 0x6D2B79F5u; /* avoid zero */
    x ^= x << 13; x ^= x >> 17; x ^= x << 5; *state = x; return x;
}

void FI_Init(uint32_t global_seed)
{
    s_global_seed = (global_seed) ? global_seed : 0x1u;
    s_evt_head = 0u;
    memset((void*)s_events, 0, sizeof(s_events));
}

void FI_Arm(fi_site_t *s, const char *name, fi_mode_t mode, uint32_t pN,
uint32_t pM, uint32_t seed)
{
    s->name = name; s->mode = mode; s->paramN = pN; s->paramM = pM; s->hits =
0u; s->rng = seed ? seed : (name ? (uint32_t)name : s_global_seed);
    if (!s->rng) s->rng = 0xCAFEBABEu;
}

bool FI_ShouldFault(fi_site_t *s)
{
    if (!SFI_ENABLED || !s) return false;
    uint32_t h = ++(s->hits);
    switch (s->mode) {
        case FI_MODE_NEVER: return false;
        case FI_MODE_EVERY_N: return (s->paramN && (h % s->paramN) == 0u);
        case FI_MODE_PROB_1_IN_N: return (s->paramN && (xorshift32(&s->rng) % s-
>paramN) == 0u);
        case FI_MODE_AFTER_K_THEN_ONCE: return (h == s->paramN);
        case FI_MODE_WINDOW: return (h >= s->paramN && h < s->paramM);
        default: return false;
    }
```

```
}

void FI_CorruptBytes(fi_site_t *s, uint8_t *buf, size_t len, uint32_t nbits)
{
    if (!SFI_ENABLED || !buf || !len || !nbits) return;
    for (uint32_t i = 0; i < nbits; ++i)
    {
        uint32_t r = xorshift32(&s->rng);
        size_t idx = r % len; uint8_t bit = (1u << (r % 8u));
        buf[idx] ^= bit;
    }
}

bool FI_ShouldFail_API(fi_site_t *s, int32_t *out_synthetic_status)
{
    if (!SFI_ENABLED) return false;
    if (FI_ShouldFault(s)) { if (out_synthetic_status) *out_synthetic_status =
-1; return true; }
    return false;
}

void FI_Log(const char *name, uint32_t hits, uint32_t a0, uint32_t a1, const
uint8_t *peek, size_t peek_len)
{
    uint32_t i = s_evt_head++ % FI_MAX_EVENTS;
    s_events[i].name = name; s_events[i].hits = hits; s_events[i].arg0 = a0;
s_events[i].arg1 = a1;
    if (peek && peek_len) {
        size_t n = (peek_len > sizeof(s_events[i].peek)) ?
sizeof(s_events[i].peek) : peek_len;
        memcpy(s_events[i].peek, peek, n);
    } else {
        memset(s_events[i].peek, 0, sizeof(s_events[i].peek));
    }
}

size_t FI_Dump(char *out, size_t maxlen)
{
    size_t w = 0; uint32_t start = (s_evt_head > FI_MAX_EVENTS) ? (s_evt_head -
FI_MAX_EVENTS) : 0;
    for (uint32_t k = start; k < s_evt_head; ++k) {
        uint32_t i = k % FI_MAX_EVENTS; const fi_event_t *e = &s_events[i];
        if (!e->name) continue;
        int n = snprintf(out ? out + w : NULL, out ? (int)(maxlen - w) : 0,
                         "[%06u] %-16s hits=%lu a0=%lu a1=%lu peek=%02X %02X
%02X %02X\r\n",
                         (unsigned)k, e->name, (unsigned long)e->hits,
                         (unsigned long)e->arg0, (unsigned long)e->arg1,
```

```
                         e->peek[0], e->peek[1], e->peek[2], e->peek[3]);
        if (n < 0) break; w += (size_t)n; if (out && w >= maxlen) break;
    }
    return w;
}
```

---

6) Integrating with the NXP LPUART examples from the SDK

We start from the SDK example `boards/evkbimxrt1050/driver_examples/lpuart/interrupt/` which sets up `DEMO_LPUART` (LPUART1) and an RX ring buffer in its `DEMO_LPUART_IRQHandler`. We will add two **fault sites**:

- `RX_CORRUPT`: corrupt bytes arriving from the ADK-8582 UART before storing into the ring.
- `TX_API_FAIL`: make `LPUART_TransferSendNonBlocking` return a synthetic error or behave as if busy.

Below is a trimmed patch you can apply to the example's ISR and transmit path.

6.1 Header and site declarations (add to `lpuart_interrupt.c`)

```
#include "fi.h"

static fi_site_t s_rx_corrupt;
static fi_site_t s_tx_api_fail;

static void FI_Bringup(void)
{
    /* Seed with a constant for reproducibility; change at runtime from console
*/
    FI_Init(0x1050A429u);
    /* Corrupt 3 bits every 200 received bytes */
    FI_Arm(&s_rx_corrupt, "RX_CORRUPT", FI_MODE_EVERY_N, 200, 3, 0);
    /* Fail every 100th API call */
    FI_Arm(&s_tx_api_fail, "TX_API_FAIL", FI_MODE_EVERY_N, 100, 0, 0);
}
```

Call `FI_Bringup();` once in `main()` after board init and before entering the transmit/receive loop.

6.2 Injecting data corruption in the RX interrupt

Locate in `DEMO_LPUART_IRQHandler` the standard code that reads a byte and pushes it into the ring buffer:

```
if ((kLPUART_RxDataRegFullFlag)&LPUART_GetStatusFlags(DEMO_LPUART))
{
    data = LPUART_ReadByte(DEMO_LPUART);
    /* If ring buffer not full, add data to ring */
    if (((tmprxIndex + 1) % DEMO_RING_BUFFER_SIZE) != tmptxIndex)
    {
        demoRingBuffer[rxIndex] = data;
        rxIndex++;
        rxIndex %= DEMO_RING_BUFFER_SIZE;
    }
}
```

Replace the assignment with a corrupt-on-trigger block:

```
    if (((tmprxIndex + 1) % DEMO_RING_BUFFER_SIZE) != tmptxIndex)
    {
        uint8_t tmp = data;
        if (FI_ShouldFault(&s_rx_corrupt)) {
            FI_CorruptBytes(&s_rx_corrupt, &tmp, 1u, s_rx_corrupt.paramM /
*bits*/);
            FI_Log("RX_CORRUPT", s_rx_corrupt.hits, (uint32_t)tmp, 0, &tmp, 1);
            USER_LED_TOGGLE();
        }
        demoRingBuffer[rxIndex] = tmp;
        rxIndex = (rxIndex + 1u) % DEMO_RING_BUFFER_SIZE;
    }
```

6.3 Injecting API failure on transmit path

If your app uses `LPUART_TransferSendNonBlocking()`, wrap it as follows; otherwise, for the basic example that uses `LPUART_WriteByte()`, we simulate a *drop* (no transmit) on trigger.

```
static status_t LPUART_Send_WithFI(LPUART_Type *base, lpuart_handle_t *handle,
lpuart_transfer_t *xfer)
{
    int32_t syn = 0;
    if (FI_ShouldFail_API(&s_tx_api_fail, &syn)) {
        /* Synthetic failure: pretend TX is busy; nothing sent */
        FI_Log("TX_API_FAIL", s_tx_api_fail.hits, (uint32_t)(xfer ? xfer-
>dataSize : 0), (uint32_t)syn, xfer ? xfer->data : NULL, xfer ? (xfer-
>dataSize>4?4:xfer->dataSize):0);
        USER_LED_TOGGLE();
        return kStatus_LPUART_TxBusy; /* or kStatus_Fail */
    }
```

```
    return LPUART_TransferSendNonBlocking(base, handle, xfer);
}
```

Now route your application's transmit calls through `LPUART_Send_WithFI()` instead of calling the SDK directly.

---

7) A generic worked example: corrupting a CRC-protected message

Imagine two tasks on the EVKB exchanging fixed-size messages over a queue, each message ending with a **CRC (Cyclic Redundancy Check)**. Insert a **Fault Site** on the producer's memcpy into the queue. On every 50th hit, flip 2 bits. The consumer's CRC check must detect the corruption, increment an error counter, and *not* act on the message.

Expected observation: console shows `RX_CORRUPT` events at deterministic intervals; error counter increments; system stays alive and responsive; watchdog does not fire. If you temporarily disable the CRC check, the corrupted payload escapes and you will see the consumer misbehave—this demonstrates the purpose of the check.

---

8) Avionics use case: ARINC 429 word intake over UART from ADK-8582

8.1 Framing and decode on the EVKB UART

For the lab we assume the ADK-8582 emits ARINC 429 words into UART frames like:

```
[0xA5][LEN=4][W0][W1][W2][W3][CHK]
```

Where `W0..W3` is the 32-bit ARINC word (LSB first or MSB first per your adapter; pick one and keep it consistent), and `CHK` is a simple 8-bit sum of all payload bytes. On the EVKB we parse the frame into a 32-bit `word`, verify `CHK`, then verify **odd parity** on the word itself. Utilities below help construct/decode and intentionally corrupt fields.

```
/* arinc429.h */
static inline uint8_t arinc_label(uint32_t w)  { return (uint8_t)(w & 0xFFu); }
static inline uint8_t arinc_sdi(uint32_t w)    { return (uint8_t)((w >> 8) &
0x3u); }
static inline uint32_t arinc_data(uint32_t w)  { return (w >> 10) & 0x7FFFFu; }
static inline uint8_t arinc_ssm(uint32_t w)    { return (uint8_t)((w >> 29) &
0x3u); }
static inline uint8_t arinc_parity(uint32_t w) { return (uint8_t)((w >> 31) &
0x1u); }
```

```c
static inline uint8_t arinc_parity_odd(uint32_t w)
{
    uint32_t v = w & 0x7FFFFFFFu; /* drop existing parity */
    v ^= v >> 16; v ^= v >> 8; v ^= v >> 4; v &= 0xFu; /* 1-bit parity of 31
bits */
    uint8_t p = (0x6996u >> v) & 1u; /* even parity of 4 bits */
    return (uint8_t)(p ^ 1u); /* odd parity */
}

static inline uint32_t arinc_make(uint8_t label, uint8_t sdi, uint32_t data,
uint8_t ssm)
{
    uint32_t w = 0u;
    w |= (uint32_t)label;
    w |= ((uint32_t)(sdi & 0x3u))  << 8;
    w |= ((uint32_t)(data & 0x7FFFFu)) << 10;
    w |= ((uint32_t)(ssm & 0x3u)) << 29;
    w |= ((uint32_t)arinc_parity_odd(w)) << 31; /* set odd parity */
    return w;
}
```

8.2 Injected faults specific to ARINC 429

We place two **Fault Sites** around the point where the 32-bit `word` is accepted from the UART frame:

- `A429_LABEL_TAMPER`: when triggered, invert bit 0 of the **Label**.
- `A429_PARITY_TAMPER`: when triggered, invert the **parity** bit.

The receiver performs these checks in order: (1) UART frame checksum `CHK`, (2) ARINC odd parity, (3) label/value plausibility (e.g., only accept a configured set of labels/SDI, and enforce monotonicity or rate limits on numeric data). On any failure, the software increments a health counter, sets the LED, and **does not propagate** the value to the application.

Expected observation: all three detection stages catch the different injected errors, and your application maintains safe outputs (e.g., freezes to last-known-good value with SSM = Failure).

---

9) API failure injection pattern at driver boundary

The i.MX RT1050 SDK uses `status_t` returns for most drivers. Your application must treat them as contracts. The FI wrappers replace calls to functions such as `LPUART_TransferSendNonBlocking()` or `LPUART_TransferReceiveNonBlocking()` with behavior that returns error codes or withholds callbacks.

Exercise behavior:

- Every 100th transmit request returns `kStatus_LPUART_TxBusy`. The application should back off and retry, bounded by a deadline.
- A window between hits 500 and 520 suppresses the TX empty interrupt, simulating an ISR lost event. Your application must detect the stall via a **PIT (Periodic Interrupt Timer)** tick and recover by re-priming the transfer.

Implementation follows the wrapper in §6.3 and a small modification in the TX Empty ISR path: guard the call to `LPUART_WriteByte()` with `if (!FI_ShouldFault(&s_tx_api_fail))`.

---

10) Minimal console to control the fault campaign

Add a thin console on the debug UART so you can adjust sites without rebuilding. Two commands are enough for the labs:

- `fi dump` — prints the last N injection events using `FI_Dump()`.
- `fi arm <site> every <N> [bits=<M>]` — re-arms a site at runtime. For example: `fi arm RX_CORRUPT every 50 bits=2`.

A simple parser can reuse the SDK's `lpuart_polling` example for I/O.

---

11) Step-by-step labs (hands-on)

Lab 1 — Data corruption on the UART receive path

Objective. Prove that your input validation detects corrupted bytes before they can be decoded as valid ARINC 429 data.

Setup.

1. Start from `boards/evkbimxrt1050/driver_examples/lpuart/interrupt/`.
2. Add `fi.h`/`fi.c` to the project; call `FI_Bringup()` in `main()`.
3. Connect the ADK-8582 UART to EVKB LPUART1 (default debug UART) or a second LPUART instance as available. Configure baud/format to match the adapter (typical 115200-8-N-1 for bridging frames).
4. Implement the simple frame parser and ARINC checks from §8.1. On successful decode, increment `rx_ok`; on failure, increment `rx_bad` and **do not** forward the value.

Injection. Arm `s_rx_corrupt` with `EVERY_N=200` and `bits=3`.

Success criteria. `rx_bad` increases at the expected cadence; console shows `RX_CORRUPT` events; no invalid ARINC words pass parity + plausibility; LED toggles on each injection. The system remains responsive for >10 minutes without WDT reset.

Lab 2 — API failure on transmit

Objective. Demonstrate that the application's UART transmit code retries on `kStatus_LPUART_TxBusy` and does not deadlock when TX empty interrupts are suppressed for a bounded window.

Setup.

1. Start from the same example, but switch transmit to `LPUART_TransferSendNonBlocking()` with a handle and callback (see SDK driver examples under `driver_examples/lpuart/*` for reference).
2. Route all sends through `LPUART_Send_WithFI()`.
3. Add a **PIT** tick at 10 ms; in the tick, check a TX progress counter and if stalled for >100 ms, cancel and re-prime the send (recovery path under test).

Injection. Arm `s_tx_api_fail` to `EVERY_N=100` (synthetic busy), and once to `WINDOW [500,520)` (suppress TX empty ISR).

Success criteria. Application retries successfully; progress counter never stalls >100 ms; event log shows both types of injections.

Lab 3 — ARINC 429 field-aware corruption

Objective. Catch label and parity tampering and enforce safe output behavior.

Setup.

1. In your UART frame handler, after assembling `uint32_t word`, check `arinc_parity_odd(word)` equals the parity bit. Then decode `label`, `sdi`, `data`, `ssm`.
2. Maintain an allow-list of labels/SDIs and a per-label rate limit (e.g., do not accept more than 50 Hz unless expected).
3. On plausibility failure, set an internal health flag and hold last-known-good value.

Injection. Add two new sites `A429_LABEL_TAMPER` and `A429_PARITY_TAMPER`; arm them with `EVERY_N=37` and `EVERY_N=53` respectively. Implement the mutation as:

```
if (FI_ShouldFault(&s_label)) { word ^= 0x01u; FI_Log("A429_LABEL_TAMPER",
s_label.hits, word, 0, (uint8_t*)&word, 4); }
if (FI_ShouldFault(&s_par))   { word ^= (1u << 31);
FI_Log("A429_PARITY_TAMPER", s_par.hits, word, 0, (uint8_t*)&word, 4); }
```

Success criteria. Neither tampered frame updates the output value; parity errors rejected; mis-label rejected by allow-list; LED set and health counters reflect events; system continues nominal operation.

---

12) Best practices tailored for avionics projects

1. **Trace every injection to a requirement or hazard.** Each site should map to a low-level requirement (LLR) derived from safety analysis (e.g., "The receiver shall reject frames that fail parity or plausibility checks").
2. **Make injections deterministic and reportable.** Fix seeds and record every event with counters and site names so that a failing campaign can be reproduced exactly.
3. **Isolate SFI from flight builds.** Compile with `SFI_ENABLED=0` and remove the console in production configurations. A visible banner on startup should indicate SFI builds.
4. **Exercise realistic magnitudes.** For data corruption, inject *single-bit* and *burst* patterns aligned with expected failure modes (e.g., SEU). For API failures, choose codes actually returned by the SDK (`kStatus_LPUART_TxBusy`, `kStatus_Fail`).
5. **Measure recovery latencies.** Use PIT timestamps to demonstrate that the system meets detection/recovery deadlines consistent with aircraft function timing budgets.
6. **Use independent observation.** When possible, monitor outputs on a second UART or GPIO pin to confirm the application did not mis-actuate during injected faults.
7. **Review and lock down fault sites.** The set of sites is configuration-controlled; do not add ad-hoc sites without analysis and review.
8. **Keep the harness tiny and audited.** Simplicity reduces the risk that SFI code itself becomes a source of defects.

---

13) What to submit (lab deliverables)

- **Code diffs** against the NXP SDK examples showing `fi.*` integration and driver wrappers.
- **Event logs** from `FI_Dump()` correlating with your test plan.
- **Pass/fail matrix** per lab showing detection points and recovery paths.
- **Short narrative** (1–2 pages) explaining what failed when you intentionally disabled a check (to demonstrate the value of each barrier).

---

14) Appendix — SDK integration notes

- The EVKB user LED is exposed via `BOARD_USER_LED_GPIO` and `BOARD_USER_LED_PIN` (GPIO1/9 in the SDK demo). Macros `USER_LED_ON/OFF/TOGGLE` exist in `board.h` under the LED demos.
- The LPUART interrupt example macro-configures the peripheral as:

```
#define DEMO_LPUART             LPUART1
#define DEMO_LPUART_CLK_FREQ    BOARD_DebugConsoleSrcFreq()
#define DEMO_LPUART_IRQn        LPUART1_IRQn
#define DEMO_LPUART_IRQHandler  LPUART1_IRQHandler
```

Keep this mapping unless your hardware wiring requires a different instance. * Keep ISR code bounded: the corruption logic above only flips a few bits and logs a small event; the heavy console output is deferred to the main loop to avoid ISR floods.

15) Frequently asked "what if?"

- **Can we inject timing jitter?** Yes. Add a `FI_MODE_WINDOW` site around a busy-wait or PIT delay to stretch it and verify timeouts.
- **Can we flip DMA descriptors?** Yes, but start with byte-level corruption of the payload; descriptor corruption can crash the SoC and is better done later with a debugger halt/resume experiment.
- **Can we use ARM Debug (DAP) to patch memory at runtime?** On i.MX RT1050, yes with an external probe; but for this module we keep SFI self-contained in firmware.

End of module