Fault Injection Techniques for Bare-Metal Avionics

Preface and scope

Fault injection is the deliberate introduction of faults into a system to observe whether safety mechanisms detect them and whether the system transitions to a defined safe state. In airborne systems, we do this to provide objective evidence for **Software Considerations in Airborne Systems and Equipment Certification (DO-178C)** and **Design Assurance Guidance for Airborne Electronic Hardware (DO-254)**, and to validate the safety claims that originate from the **Guidelines for Development of Civil Aircraft and Systems (ARP4754A)** and the **Guidelines and Methods for Conducting the Safety Assessment (ARP4761)**. Throughout this module we work on bare-metal targets—microcontrollers without an operating system—to keep the mechanisms visible and controllable.

This material is intended for internal engineering training and lab use only. Do not perform any physical fault injection on operational aircraft equipment. Perform all labs on development boards in a controlled environment, using code you own. When we mention physical techniques (e.g., voltage or clock glitching) we restrict ourselves to high-level descriptions for awareness; all hands-on exercises use software-implemented approaches.

Fundamental definitions

A **fault** is the hypothesized cause of an error. An **error** is the part of the system state that may lead to failure. A **failure** is the deviation of the delivered service from correct service. For example, a single-event upset (SEU) due to radiation in static random-access memory (SRAM) is a fault, a flipped bit in a stored sensor value is an error, and an out-of-range command sent to an actuator is a failure.

A **fault model** formalizes what can go wrong. Common fault models for avionics include value faults (bit-flips, stuck-at-0/1), timing faults (jitter, deadline misses), control-flow faults (unexpected exceptions), interface/protocol faults (bus contention, malformed frames), and environmental transients such as **single-event transient (SET)** and **single-event upset (SEU)**. Faults can be **transient** (vanish after some time), **intermittent** (recur sporadically), or **permanent**.

Why inject faults in avionics software

The primary objective is to verify that safety requirements and architectural mitigations behave as specified. In software this includes input validation, range and rate-of-change checks, timeouts, **watchdog timer (WDT)** handling, **cyclic redundancy check (CRC)** verification, and graceful degradation strategies. In hardware it includes **error-correcting code (ECC)** for memories, parity on buses, and, in some platforms, **triple modular redundancy (TMR)**. For DO-178C Level A/B software we also need to support **modified condition/decision coverage (MC/DC)** by constructing tests that exercise detection and recovery branches. Fault injection is also a way to substantiate **failure modes and effects analysis (FMEA)** claims, to evaluate diagnostic coverage, and to practice incident logging that supports **continuing airworthiness**.

Taxonomy of fault injection techniques

We divide techniques into four families that are useful on bare-metal targets:

1. **Software-Implemented Fault Injection (SFI)**. We instrument code to perturb variables, corrupt buffers, alter control-flow, or delay computation. This is the workhorse of our labs because it is repeatable and automatable.
2. **Interface/Input Perturbation.** We introduce malformed, delayed, or missing data at software boundaries: serial ports, **controller area network (CAN)** frames, **time-triggered Ethernet (TTE)** frames, **serial peripheral interface (SPI)** transactions, or sensor drivers. On bare metal we typically emulate the device rather than the bus.
3. **Exception and Memory Faults.** We provoke **HardFault**, **BusFault**, and **MemManage** exceptions on an **ARM Cortex-M** to validate handlers, logging, and reset paths. We also flip bits in data structures to emulate ECC-detectable single-bit errors.
4. **Timing Perturbation.** We add jitter, hold off interrupts, or stretch critical sections to simulate load spikes and observe deadline monitors.

Physical techniques—voltage/clock glitches, electromagnetic (EM) pulses, laser injection, and radiation—are acknowledged here only for completeness. Use vendor labs and certified facilities if you ever need hardware evidence; do not improvise.

Target and tools for the labs

Any ARM Cortex-M4 or Cortex-M7 development board with a user LED and a hardware timer will work (for example, an STM32F4-class board), a **serial wire debug (SWD)** probe, and a UART-to-USB cable. We write in C, with no vendor libraries, to emphasize fundamentals. Tracing is optional but useful; an **Instrumentation Trace Macrocell (ITM)** with **single wire output (SWO)** is ideal if your board supports it.

We will build a small, testable application: a periodic sensor acquisition loop with range checking, CRC verification, and a WDT-guarded control output. We then apply the four technique families above. Each technique is illustrated with (a) a generic embedded example and (b) a specific, realistic avionics scenario.

Technique 1 — Software-Implemented Fault Injection (SFI)

Concept

SFI uses compile-time or run-time hooks to perturb program variables, buffers, and control flow. A common pattern is to define a fault registry and injectors that can be dynamically enabled via a test control channel. This allows you to run an automated campaign where each unique fault is introduced at precise times and locations, while you collect detection/response metrics.

Minimal SFI harness (C, Cortex-M)

```
// sfi.h — single-file injectable harness for bare-metal labs
#include <stdint.h>
#include <stdbool.h>

// Test control port (replace with UART/ITM as needed)
static volatile uint32_t sfi_tick;

// Fault identifiers
typedef enum {
    FI_NONE = 0,
    FI_SENSOR_SAMPLE_BITFLIP,
    FI_LENGTH_FIELD_CORRUPT,
    FI_DELAY_HANDLER,
    FI_STUCK_INPUT,
} fi_id_t;

// Global configuration written by the test script
static volatile fi_id_t g_fi_active = FI_NONE;
static volatile uint32_t g_fi_param  = 0;   // e.g., bit index, delay cycles

// Helper: pseudo-random LFSR for noise
static uint32_t lfsr32(uint32_t *state){
    uint32_t x = *state; x ^= x<<13; x ^= x>>17; x ^= x<<5; return *state = x; }

// Injection sites use these macros for clarity
#define FI_ACTIVE(id)      (g_fi_active == (id))
#define FI_PARAM()         (g_fi_param)

// Example 1: flip a bit in a 16-bit sensor sample
static inline uint16_t fi_maybe_flip_u16(uint16_t v){
    if (FI_ACTIVE(FI_SENSOR_SAMPLE_BITFLIP)) {
        uint32_t bit = (FI_PARAM() & 0xF); // 0..15
        v ^= (1u << bit);
    }
    return v;
}

// Example 2: corrupt a length field in a frame-like buffer
static inline void fi_maybe_corrupt_len(uint16_t *len){
    if (FI_ACTIVE(FI_LENGTH_FIELD_CORRUPT)) {
        *len = (uint16_t)(*len + (FI_PARAM() ? FI_PARAM() : 1));
    }
}

// Example 3: inject a delay in a handler (busy wait)
```

```
static inline void fi_maybe_delay(void){
    if (FI_ACTIVE(FI_DELAY_HANDLER)) {
        for (volatile uint32_t i = 0; i < FI_PARAM(); ++i) __asm__("nop");
    }
}

// Example 4: force a stuck input value
static inline int16_t fi_stuck_value(int16_t current){
    if (FI_ACTIVE(FI_STUCK_INPUT)) return (int16_t)FI_PARAM();
    return current;
}
```

Generic embedded example: protecting a sensor processing chain

We read a 16-bit sample every 10 ms, check range and rate-of-change, verify a CRC on a simple frame, and pass the value to a control function. We insert SFI hooks at the sample acquisition and frame assembly points. The objective is to observe that each injected corruption is detected and that the control path degrades gracefully.

```
// app.c — skeleton application with SFI hooks
#include <stdint.h>
#include <stdbool.h>
#include "sfi.h"

#define PERIOD_TICKS  (SystemCoreClock/
100) // 10 ms at 1 kHz SysTick, adjust as needed

// Simplified CRC-16 (polynomial 0x1021)
static uint16_t crc16_ccitt(const uint8_t* data, uint16_t len){
    uint16_t crc = 0xFFFF;
    for (uint16_t i=0;i<len;++i){
        crc ^= (uint16_t)data[i] << 8;
        for (int b=0;b<8;++b) crc = (crc & 0x8000) ? (crc<<1) ^ 0x1021 :
(crc<<1);
    }
    return crc;
}

// Mock sensor read (replace with ADC/driver)
static uint16_t read_sensor_raw(void){
    static uint32_t prng = 1;
    uint16_t v = (uint16_t)(lfsr32(&prng) & 0x0FFF); // 12-bit ADC-like range
    return fi_maybe_flip_u16(v); // SFI site: bit-flip sample
}

// Range and slew-rate checks
```

```
static bool plausibility_ok(uint16_t v){
    static uint16_t prev;
    const uint16_t MIN=50, MAX=3950, MAX_DELTA=200;
    bool in_range = (v>=MIN && v<=MAX);
    bool rate_ok  = (v > prev) ? ((v-prev) <= MAX_DELTA) : ((prev - v) <=
MAX_DELTA);
    prev = v;
    return in_range && rate_ok;
}

// Frame the sample and CRC it
typedef struct { uint16_t len; uint16_t sample; uint16_t crc; }
__attribute__((packed)) frame_t;

static frame_t make_frame(uint16_t sample){
    frame_t f; f.len = sizeof(frame_t);
    fi_maybe_corrupt_len(&f.len); // SFI site: corrupt length
    f.sample = sample;
    f.crc = crc16_ccitt((uint8_t*)&f, sizeof(frame_t)-2);
    return f;
}

// Control output guarded by plausibility; returns command
static uint16_t control_law(uint16_t sample){
    if (!plausibility_ok(sample)) return 0; // safe default
    return sample / 2; // toy transform
}

// Periodic task entry point
void tick_10ms(void){
    fi_maybe_delay(); // SFI site: timing perturbation
    uint16_t s = read_sensor_raw();
    s = (uint16_t)fi_stuck_value((int16_t)s);
    frame_t f = make_frame(s);
    uint16_t expected = crc16_ccitt((uint8_t*)&f, sizeof(frame_t)-2);
    bool crc_ok = (expected == f.crc) && (f.len == sizeof(frame_t));
    uint16_t cmd = crc_ok ? control_law(f.sample) : 0; // safe default on CRC
fail
    (void)cmd; // route to actuator driver in real code
}
```

Run a campaign by setting `g_fi_active` and `g_fi_param` at different ticks and confirming that each mechanism catches the corruption and returns to a defined safe output.

Avionics use case example: Air Data Inertial Reference Unit (ADIRU) pressure path

In an **Air Data Inertial Reference Unit (ADIRU)**, pressure transducers feed an **Analog-to-Digital Converter (ADC)**, then software applies polynomial linearization, temperature compensation, and plausibility checks before supplying the **Flight Control Computer (FCC)**. We emulate a pressure channel and introduce a transient bit-flip on the raw ADC sample and a stuck-at fault at the driver boundary. The expected behavior is: range/rate checks detect anomalies; the ADIRU channel suppresses the bad sample, uses the previous valid value for a bounded interval, and flags the channel as “degraded” on the maintenance bus while the output to the FCC remains within safe limits. Our SFI hooks correspond exactly to these points and allow demonstration of both detection and bounded-time masking.

Technique 2 — Interface/Input Perturbation

Concept

Most avionics faults propagate through interfaces. On a bare-metal target, we create a controllable software stub that stands in for the device or bus. This lets us inject malformed frames, missing bytes, out-of-order sequences, and timing gaps without external equipment. We can also fuzz parsers deterministically to support repeatable campaigns and debugging.

Generic embedded example: UART line-discipline faults

We replace the real **universal asynchronous receiver-transmitter (UART)** driver with a stub that sometimes drops bytes, duplicates them, or delays a terminator. The application must detect framing errors, timeouts, and checksum mismatches.

```
// uart_stub.c — deterministic perturbation of a byte stream
#include <stdint.h>
#include <stdbool.h>
#include "sfi.h"

static const uint8_t golden_msg[] = { 0xAA,0x55,0x03,0x10,0x20,0x30,0x00 }; //
last two bytes CRC16

// Deliver exactly one test frame per call with faults injected
int uart_read_frame(uint8_t *buf, uint16_t *len){
    uint16_t L = sizeof(golden_msg);
    for (uint16_t i=0;i<L;i++) buf[i] = golden_msg[i];

    if (FI_ACTIVE(FI_LENGTH_FIELD_CORRUPT)) {
        // simulate truncation by reducing available length
        L = (uint16_t)(L - (FI_PARAM() ? FI_PARAM() : 1));
    }
    *len = L;
    return 0; // 0 ok
}
```

The application frames input with a timeout and validates a CRC before accepting it. The test toggles between correct and corrupted frames to prove that the parser never consumes a malformed message as valid input.

Avionics use case example: Remote Sensor Interface and Frame Robustness

Consider an **Air Data Computer (ADC)** speaking over a point-to-point **serial peripheral interface (SPI)** to a remote pressure module. The module frames measurements as `{SOF, LEN, PAYLOAD, CRC}`. We implement the module as a stub in software and inject length/CRC inconsistency and delayed end-of-frame. The ADC software must timeout, increment a **Built-In Test (BIT)** counter, and hold last-valid data for at most N frames before declaring channel invalid to the consuming **Flight Warning Computer (FWC)**. Our perturbation stub allows repeatable validation of these behaviors without oscilloscopes or hardware pattern generators.

Technique 3 — Exception and Memory Faults on ARM Cortex-M

Concept

ARM Cortex-M cores provide separate exceptions for memory protection and bus errors. Validating these handlers is a certification expectation: they must record enough context to support root cause analysis, avoid unbounded recursion, and drive the system to a safe state—often a controlled reset under WDT supervision after persisting fault logs to non-volatile memory.

Basic setup: vector table and handlers

```
// fault.c — minimal HardFault/BusFault/MemManage handlers
#include <stdint.h>
#include <stdbool.h>

// Volatile storage for post-mortem (keep tiny in examples)
volatile uint32_t g_fault_sig;
volatile uint32_t g_fault_addr;

void HardFault_Handler(void){ g_fault_sig = 0xHARDFAU1; for(;;){} }
void BusFault_Handler(void){ g_fault_sig = 0xBUSBAD00; for(;;){} }
void MemManage_Handler(void){ g_fault_sig = 0xMEMBAD00; for(;;){} }
```

Provoking controlled faults

We can attempt a write to an unmapped address to trigger **BusFault**, or enable divide-by-zero trapping to provoke **UsageFault**. The key is to do this behind a test gate and to verify that the handler does not compromise the WDT service path.

```
// provoke.c — gated provocations
#include <stdint.h>
#include "sfi.h"

#define BAD_PTR ((volatile uint32_t*)0xFFFFFFF0u)

void provoke_busfault_if_enabled(void){
    if (FI_ACTIVE(FI_LENGTH_FIELD_CORRUPT) && FI_PARAM()==0xDEAD) {
        *BAD_PTR = 0x12345678u; // will raise BusFault on most MCUs
    }
}
```

Generic embedded example: proving safe reset with logging

We enable the provocation and confirm that the **watchdog timer (WDT)** continues to tick from a low-priority context, that the fault handler sets a signature, and that after reset we can read back a small ring buffer in backup registers noting the exception type and program counter. This demonstrates a bounded-time path to a known safe state with maintenance evidence preserved.

Avionics use case example: Flight Control Computer (FCC) sensor driver containment

In a **Flight Control Computer (FCC)** with multiple channels, a single faulty sensor driver must not crash the channel or, worse, propagate corrupt data to control laws. We inject a **BusFault** from inside a mock sensor driver while an independent WDT service task continues to run. The expected behavior is: the fault is trapped; the WDT expires because the driver starves low-priority service; the channel resets cleanly; on warm boot the maintenance log shows a BusFault in the driver with context; voting logic in a tri-channel **triple modular redundancy (TMR)** scheme masks the rebooting lane until it rejoins.

Technique 4 — Timing Perturbation

Concept

Many hazards arise from missed deadlines and jitter. On bare metal, the scheduler is typically a timer interrupt and a cooperative main loop. We can inflate execution time at specific points, create bursty interrupt load, and delay the clearing of interrupt flags. The objective is to prove that deadline monitors detect the condition and that outputs revert to a safe, bounded behavior.

Generic embedded example: deadline monitor around a 10 ms control task

We wrap the control task with a cycle counter based on the **Data Watchpoint and Trace (DWT)** unit. If the task exceeds its 10 ms budget, we command a safe output and raise a diagnostic.

```c
// timing.c — simple deadline monitor using DWT_CYCCNT
#include <stdint.h>
#include <stdbool.h>
#include "sfi.h"

#define CYCLES_PER_TICK (SystemCoreClock/100) // if SysTick gives 10 ms ticks

static inline void dwt_init(void){
    *(volatile uint32_t*)0xE0001000 |= 1; // DEMCR.TRCENA
    *(volatile uint32_t*)0xE0001004 = 0;  // DWT_CYCCNT = 0
    *(volatile uint32_t*)0xE0001000 |= 1; // enable again
}

bool deadline_ok(void (*task)(void)){
    uint32_t *CYCCNT = (uint32_t*)0xE0001004;
    uint32_t start = *CYCCNT;
    task();
    uint32_t elapsed = *CYCCNT - start;
    return elapsed < (CYCLES_PER_TICK - 1000); // some margin
}
```

With `fi_maybe_delay()` active, we verify that the monitor detects an overrun and the output path enters a safe hold-last-value mode.

Avionics use case example: Fly-By-Wire (FBW) inner-loop jitter

Fly-By-Wire (FBW) control laws often run at 1 kHz (1 ms), with strict jitter limits. We inject extra cycles into the inner loop, simulating cache miss storms or DMA bus contention. The monitor detects overruns and the control channel temporarily freezes the actuator command within certified bounds until timing is re-established. A tri-channel voter masks the degraded channel if it cannot recover within N cycles.

Planning a fault campaign

A good campaign is traceable to safety artifacts. Start from your **system safety assessment (SSA)** and **FMEA**. For each failure mode that relies on software detection/mitigation, identify the software points where evidence must be produced. Define a set of injectors (like the ones above) and a schedule (which tick, which parameter) that traverses the relevant states. For each injection you predefine the expected detection signal, reaction time, and steady-state behavior. Instrumentation collects timestamps so that you can compute detection latency and recovery time. Stop criteria are based on functional coverage (every mitigation executed), structural coverage (e.g., MC/DC on safety-critical branches), and diagnostic coverage (every detection mechanism exercised under realistic conditions).

Where hardware supports it, correlate SFI with architectural features: enable ECC single-bit correction; deliberately write a word with an incorrect ECC syndrome via a vendor-provided error-injection register if available; then confirm that the ECC handler increments the correct counter and that double-bit errors lead to an exception and safe shutdown. If your microcontroller does not expose error-inject registers, keep to software-level flips as shown earlier.

Evidence and observability on bare metal

On minimal targets, observability is the challenge. A simple UART log at 115200 bps with fixed-size binary records is often sufficient. If available, use **Instrumentation Trace Macrocell (ITM)** with **single wire output (SWO)** to stream timestamped events with negligible CPU burden. For time correlation, use the **Data Watchpoint and Trace (DWT)** cycle counter. Persist a small ring buffer of the last N events into battery-backed registers or a reserved flash page during exception processing so that you can recover context after an automatic reset.

An example record format might include: 32-bit timestamp (DWT_CYCCNT), injector ID, parameter, detection flag, reaction taken, and a brief context code. A companion Python script on the host can render timelines and compute detection latency statistics. While the present module focuses on C and firmware, always plan for data reduction and automated report generation as part of your evidence package.

Hands-on Lab 1 — Build and run the SFI harness

Objective

Construct the SFI harness, integrate it into a periodic sensor loop, and demonstrate detection of bit-flip, length corruption, delay, and stuck-at faults. Capture evidence of detection and safe output.

What you will build

You will compile `sfi.h` and `app.c` on your Cortex-M target, map `tick_10ms()` to your SysTick or hardware timer, and expose `g_fi_active`/`g_fi_param` through a simple control interface (e.g., memory-mapped variables written by a debugger or by a UART command).

Procedure

1. Bring up a 10 ms tick using SysTick. Inside the ISR, call `tick_10ms()`.
2. Implement a UART RX routine that accepts ASCII commands like `SET 1 7` (activate `FI_SENSOR_SAMPLE_BITFLIP`, bit index 7) and `CLR` (deactivate). Parsing can be minimal; this is a lab tool.
3. Run a baseline for 5 seconds: log per-tick CRC validity and plausibility flags.

4. Activate each injector in turn. For each, capture: time of activation, time of detection, output command value, and the diagnostic flag set by the software.
5. Produce a short conclusion: which mechanisms detected which fault types, any false positives, and maximum detection latency.

Success criteria

Each injection is detected deterministically, output never exceeds defined limits, and the system returns to nominal behavior within N ticks after deactivation.

Avionics tie-in

Explain how the same hooks correspond to an ADIRU pressure path and how the lab evidence would trace to safety requirements such as “The system shall detect and suppress erroneous sensor samples within 20 ms.”

Hands-on Lab 2 — Validate exception handling and safe reset

Objective

Demonstrate that a deliberate **BusFault** or **UsageFault** is trapped, minimally logged, and leads to a controlled WDT-mediated reset with preservation of a tiny, durable log.

What you will build

A gated provocation such as `provoke_busfault_if_enabled()`, a minimal non-blocking WDT service routine, and exception handlers that write a signature and the faulting address to a known RAM location. A warm-boot routine copies that info into a small wear-leveled flash log and clears the RAM signature.

Procedure

1. Configure the WDT for a 250 ms timeout and service it in a low-priority periodic function.
2. Arm the provocation and call it from a driver context that deliberately omits WDT service.
3. Confirm that the exception handler executes, the WDT expires, and the system resets.
4. On reboot, print the captured signature and fault address over UART and append it to flash.

Success criteria

No unbounded loops in the handler, WDT always wins within a bounded time, and the rebooted system reports the exception type and a valid program counter or faulting address.

Avionics tie-in

Relate this to an FCC channel where a driver fault should not jeopardize command continuity thanks to TMR masking and defined recovery behavior.

Hands-on Lab 3 — Timing fault injection and deadline monitoring

Objective

Show that adding artificial delays at specified points leads to deadline violation detection and activation of a safe output policy.

What you will build

A DWT-based cycle counter wrapper around the 10 ms task, a configurable delay injector, and a simple “hold-last-good-command” strategy with a maximum hold time after which outputs are forced to zero.

Procedure

1. Enable the DWT cycle counter and calibrate cycles per 10 ms.
2. Activate `FI_DELAY_HANDLER` with increasing `g_fi_param` until you exceed the budget.
3. Record the detected overruns and the moments at which the safe policy activates and deactivates.

Success criteria

Overruns are detected with low false positives; safe policy caps output within specified bounds; normal operation resumes automatically when timing returns to nominal.

Avionics tie-in

Map this to a Fly-By-Wire inner loop with tight timing constraints and show how a deadline monitor prevents unstable transients.

Campaign metrics and reporting

For each campaign, report:

- **Coverage of mitigations.** Which detection branches executed at least once (trace to requirements IDs).
- **Detection latency.** Median/95th percentile ticks from injection to detection.
- **Recovery time.** Time to return to nominal output.
- **Residual risk.** Scenarios where detection did not occur (with rationale).

Even though avionics standards do not mandate automotive-style **single point fault metric (SPFM)**, be explicit about diagnostic coverage so that safety engineers can relate software evidence to the hardware safety case when relevant.

Best practices for fault injection in avionics

Faults must be traceable to safety analyses. Begin with ARP4761 artifacts and DO-178C safety requirements, not with an improvised list of experiments. Keep all injection mechanisms behind explicit test gates; it must be impossible to compile production firmware with injectors active. Maintain a crisp separation between production code and test harness through compile-time flags and linker sections, and subject both to peer review. Always define safe output policies in measurable, bounded terms—“set command to zero within 20 ms and hold for 100 ms unless cleared”—and verify them under injection. Prefer deterministic, scriptable campaigns with fixed seeds so that the results are reproducible. Treat observability as a first-class design concern: timestamp events, keep logs bounded, and protect log writes against power loss.

When exercising exceptions, aim for minimal handlers that set a signature, capture essential context, and let the WDT enforce reset—complex exception code often creates secondary failures. For timing perturbation, keep the perturbation localized (a single delay loop or intentional cache flush if present) to avoid masking root causes. When emulating ECC events in software, be explicit about the limits of emulation and avoid over-claiming coverage. Finally, never perform physical glitching or radiation experiments outside certified labs, and never on operational avionics hardware; awareness of such techniques is useful for robustness but belongs in specialized facilities with safety controls.

Appendix A — Putting it together: a tiny reference application

The reference application combines the SFI harness, a UART control mini-shell, a SysTick-driven 10 ms loop, and minimal logging. Use it as a seed for your team’s internal framework. Ensure the entire test harness is compiled only with a `#define TEST_ONLY` guard and is excluded from production binaries during configuration audits under DO-178C.

Appendix B — Acronym expansion index

ADIRU: Air Data Inertial Reference Unit ADC: Analog-to-Digital Converter ADC (computer): Air Data Computer ARP4754A: Guidelines for Development of Civil Aircraft and Systems ARP4761: Guidelines and Methods for Conducting the Safety Assessment BIT: Built-In Test CBIT: Continuous Built-In Test CRC: Cyclic Redundancy Check DAL: Development Assurance Level DO-178C: Software Considerations in Airborne Systems and Equipment Certification DO-254: Design Assurance Guidance for Airborne Electronic Hardware DWT: Data Watchpoint and Trace ECC: Error-Correcting Code EM: Electromagnetic FCC: Flight Control Computer FBW: Fly-By-Wire FMEA: Failure Modes and Effects Analysis FWC: Flight Warning Computer HIL: Hardware-in-the-Loop ITM: Instrumentation Trace Macrocell JTAG: Joint Test Action Group LFSR: Linear Feedback Shift Register MC/DC: Modified Condition/Decision Coverage PBIT: Power-On Built-In Test PIL: Processor-in-the-Loop SEU: Single Event Upset SET: Single Event Transient SPI: Serial Peripheral Interface SWD: Serial Wire Debug SWO: Single Wire Output TMR: Triple Modular Redundancy TTE: Time-Triggered Ethernet UART: Universal Asynchronous Receiver-Transmitter WDT: Watchdog Timer

---

Closing note

This module intentionally emphasizes techniques that are realistic on a bench with a low-cost development board but map cleanly to Airbus avionics scenarios. Integrate these patterns into your team’s internal test frameworks, and ensure every experiment ties back to a requirement, a safety claim, and a clear definition of “safe output.”