1. Why abstraction matters in avionics bare-metal systems

A modern avionics Line Replaceable Unit (LRU) rarely survives first contact with reality if its software is coupled directly to registers. Hardware spins, component obsolescence, qualification testing, and airworthiness updates require software that is testable and portable. Two layers make this possible:

- **Low-Level Device Driver (LLDD) Abstraction** isolates silicon specifics (register maps, clock trees, pin multiplexers, DMA engines) behind a stable API. A correct LLDD makes the rest of the system oblivious to which timer, UART, or SPI block is in use.
- **Middleware Abstraction** sits above device drivers and exposes protocol or service semantics (file systems, communications stacks, sensor fusion). It depends only on the LLDD API—not on registers—and therefore ports cleanly across boards.

In certification terms, this partitioning assists with **DO-178C (Software Considerations in Airborne Systems and Equipment Certification)** objectives such as verifiability, traceability, and structural coverage, and helps segregate change-prone hardware-dependent logic from mission logic.

---

2. Mapping the concepts to the i.MX RT1050 EVKB SDK

The **NXP MCUXpresso SDK** you have for the **EVKB-IMXRT1050** already embodies these layers.

- **LLDD equivalents** live under `devices/MIMXRT1052/drivers/` (for example `fsl_lpuart.h/.c`, `fsl_gpio.h/.c`, `fsl_edma.h/.c`). These are thin, silicon-aware drivers with stable C APIs.
- **Board Support** (clocks, pins, board macros) is under `boards/evkbimxrt1050/project_template/` and per-example `pin_mux.c`, `clock_config.c`.
- **Middleware** is under `middleware/` (for example **FatFS** for file systems, **lwIP** for IP networking, **USB**, **mbedTLS**). These components depend on adapters that call into LLDD.
- **Component adapters** appear under `components/` (for example `fsl_adapter_lpuart.c`). They demonstrate a portable façade over one or more LLDDs.

We will deliberately build our own small but rigorous abstractions so that the structure is obvious and the approach can be repeated for other peripherals.

---

3. Vocabulary and scope

- **HAL (Hardware Abstraction Layer)** is often used as a synonym for LLDD, but we'll be more precise: the LLDD is a portability layer **to this SoC family**; a project-specific **HAL** is your façade on top of LLDDs that may normalize naming, error models, timing, and concurrency policies.
- **Middleware** refers to software that implements a protocol or service (for example **ARINC 429 (Aeronautical Radio, Incorporated 429) word framing and parsing**, a log storage service built on **FatFS (FAT File System)**, or an **NVM (Non-Volatile Memory)** key store).
- **OSA (Operating System Abstraction)** is omitted here because our thrust is **bare-metal**; when you later add an RTOS (Real-Time Operating System), the same abstractions remain valid.

---

4. A reference layering for this course

```
Application (mission logic)
└── Middleware (protocols & services; e.g., ARINC 429 parser, file logger)
    └── Project HAL (serial, timing, storage façades with strict APIs)
        └── NXP LLDDs (fsl_lpuart, fsl_gpio, fsl_edma, fsl_flexspi, ...)
            └── Registers & silicon (IOMUXC, CCGR clocks, DMA MUX, ...)
```

Our goal is to make each boundary contractual and testable.

---

5. Generic example: a portable serial façade over LPUART

We shall create a **Serial** interface whose header exposes no NXP types. The EVKB implementation will call **LPUART (Low-Power UART)** driver APIs from `fsl_lpuart.h`. Two transports are provided:

1) An **interrupt-driven ring buffer** suitable for command/control and moderate throughput.
2) An **EDMA (Enhanced Direct Memory Access)** variant for sustained high-rate receptions.

Both share the same `serial_if_t` API so middleware can swap them at link time.

5.1 Public header (project HAL)

```
// serial_if.h — project HAL (no SoC headers allowed here)
#pragma once
#include <stdint.h>
#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef enum {
    SERIAL_OK = 0,
    SERIAL_EINVAL,
    SERIAL_EBUSY,
    SERIAL_EIO,
    SERIAL_ETIMEOUT
} serial_status_t;

typedef void (*serial_rx_cb_t)(const uint8_t *data, size_t len, void *user);

typedef struct serial_if serial_if_t; // Opaque handle

// Open/close and configuration
serial_if_t *serial_open(uint32_t baud, serial_rx_cb_t cb, void *user);
void serial_close(serial_if_t *s);

// Blocking TX with bounded time; returns bytes written
int serial_write(serial_if_t *s, const uint8_t *data, size_t len, uint32_t
timeout_ms);

// Optional nonblocking read into caller buffer; returns bytes copied
int serial_read(serial_if_t *s, uint8_t *out, size_t cap);

#ifdef __cplusplus
}
#endif
```

5.2 EVKB LPUART1/LPUART3 interrupt implementation (LLDD usage)

```
// serial_lpuart.c — SoC-specific implementation (depends on fsl_lpuart.h)
#include "serial_if.h"
#include "fsl_lpuart.h"
#include "fsl_iomuxc.h"
#include "fsl_clock.h"
#include "board.h"

#ifndef SERIAL_LPUART_INSTANCE
#define SERIAL_LPUART_INSTANCE 3 // Use LPUART3 to avoid conflict with the
```

```
debug console
#endif

#define RX_RING_SZ 256

struct serial_if {
    LPUART_Type *base;
    volatile uint8_t rx_ring[RX_RING_SZ];
    volatile size_t  rx_w, rx_r;
    serial_rx_cb_t   cb;
    void            *cb_user;
};

static struct serial_if s_ctx;

static void serial_pinmux_lpuart3(void)
{
    // GPIO_AD_B1_06 -> LPUART3_TXD, GPIO_AD_B1_07 -> LPUART3_RXD
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0U);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0U);
    // Basic 100K pullup, fast slew
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0x10B0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0x10B0U);
}

serial_if_t *serial_open(uint32_t baud, serial_rx_cb_t cb, void *user)
{
    s_ctx.base    = (SERIAL_LPUART_INSTANCE == 1) ? LPUART1 : LPUART3;
    s_ctx.rx_w    = s_ctx.rx_r = 0;
    s_ctx.cb      = cb;
    s_ctx.cb_user = user;

    BOARD_InitBootClocks();
    serial_pinmux_lpuart3();

    const uint32_t srcClk = CLOCK_GetFreq(kCLOCK_OscClk); // Or
BOARD_DebugConsoleSrcFreq() if using LPUART1

    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baud;
    cfg.enableTx     = true;
    cfg.enableRx     = true;

    LPUART_Init(s_ctx.base, &cfg, srcClk);

    LPUART_EnableInterrupts(s_ctx.base, kLPUART_RxDataRegFullInterruptEnable
| kLPUART_RxOverrunInterruptEnable);
```

```
    EnableIRQ((SERIAL_LPUART_INSTANCE == 1) ? LPUART1_IRQn : LPUART3_IRQn);

    return &s_ctx;
}

void serial_close(serial_if_t *s)
{
    if (!s) return;
    DisableIRQ((SERIAL_LPUART_INSTANCE == 1) ? LPUART1_IRQn : LPUART3_IRQn);
    LPUART_Deinit(s->base);
}

int serial_write(serial_if_t *s, const uint8_t *data, size_t len, uint32_t
timeout_ms)
{
    if (!s || !data) return -SERIAL_EINVAL;
    (void)timeout_ms; // For simplicity, use blocking API; you may extend
with a timer
    LPUART_WriteBlocking(s->base, data, len);
    return (int)len;
}

int serial_read(serial_if_t *s, uint8_t *out, size_t cap)
{
    if (!s || !out || cap == 0) return 0;
    size_t n = 0;
    while ((s->rx_r != s->rx_w) && (n < cap)) {
        out[n++] = s->rx_ring[s->rx_r++ & (RX_RING_SZ-1)];
    }
    return (int)n;
}

void LPUART3_IRQHandler(void)
{
    const uint32_t flags = LPUART_GetStatusFlags(s_ctx.base);
    if (flags & kLPUART_RxDataRegFullFlag) {
        uint8_t c = LPUART_ReadByte(s_ctx.base);
        s_ctx.rx_ring[s_ctx.rx_w++ & (RX_RING_SZ-1)] = c;
        if (s_ctx.cb) {
            size_t w = s_ctx.rx_w, r = s_ctx.rx_r;
            if ((w - r) >= 8) { // simple watermark
                size_t len = w - r;
                uint8_t buf[8];
                size_t take = (len > sizeof buf) ? sizeof buf : len;
                for (size_t i = 0; i < take; ++i) {
                    buf[i] = s_ctx.rx_ring[s_ctx.rx_r++ & (RX_RING_SZ-1)];
                }
                s_ctx.cb(buf, take, s_ctx.cb_user);
            }
```

```
        }
    }
    if (flags & kLPUART_RxOverrunFlag) {
        LPUART_ClearStatusFlags(s_ctx.base, kLPUART_RxOverrunFlag);
    }
    __DSB();
}
```

This implementation is intentionally small. It uses **LPUART_WriteBlocking**, **LPUART_ReadByte**, and the standard **IRQ handler pattern** from the SDK so that the rest of the course code compiles against your SDK without private hacks.

5.3 EDMA receive variant for sustained throughput (optional)

For high-rate streams, replace the ISR ring buffer with an **EDMA** circular transfer (`fsl_lpuart_edma.h` + `fsl_edma.h`). The façade API does not change; only the backend does. You will implement this in Lab 4.

---

6. Specific avionics example: ARINC 429 over a UART-attached adapter (ADK-8582)

ARINC 429 (Aeronautical Radio, Incorporated 429) transmits 32-bit words on a twisted pair at 12.5 or 100 kbps. In this track, the **ADK-8582** acts as a UART bridge that exposes raw 32-bit words. We will treat the UART as a byte stream and build middleware that understands ARINC 429 framing.

6.1 ARINC 429 word parsing middleware

We create a tiny, deterministic parser that knows nothing about LPUART. It accepts 32-bit words in big-endian or little-endian form and exposes fields: **Label (8 bits)**, **SDI (Source/Destination Identifier, 2 bits)**, **Data (19 bits, BNR/BCD as configured)**, **SSM (Sign/Status Matrix, 2 bits)**, **Parity (1 bit)**. It also includes a parity check (odd parity per ARINC 429).

```
// arinc429.h — middleware API (no silicon includes)
#pragma once
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    uint8_t  label;  // bits 1..8 (LSB first in serial transmission; we keep
numeric value)
    uint8_t  sdi;    // bits 9..10
    uint32_t data;   // bits 11..29
    uint8_t  ssm;    // bits 30..31
```

```
    uint8_t  parity; // bit 32
    bool     parity_ok;
} arinc429_word_t;

// Parse a raw 32-bit word as delivered by the adapter (MSB=bit32)
static inline arinc429_word_t arinc429_parse(uint32_t w)
{
    arinc429_word_t out;
    out.label     = (uint8_t)(w & 0xFFu);
    out.sdi       = (uint8_t)((w >> 8) & 0x3u);
    out.data      = (uint32_t)((w >> 10) & 0x7FFFFu);
    out.ssm       = (uint8_t)((w >> 29) & 0x3u);
    out.parity    = (uint8_t)((w >> 31) & 0x1u);
    // Odd parity over bits 1..31 must equal bit 32
    uint32_t bits = w & 0x7FFFFFFFu; // 31 bits
    bits ^= bits >> 16; bits ^= bits >> 8; bits ^= bits >> 4; bits &= 0xF; //
Hamming weight mod 2 (partial)
    bool odd = (0x6996u >> bits) & 1u; // parity of low nibble
    out.parity_ok = (odd == out.parity);
    return out;
}
```

Notice how the middleware has no knowledge of interrupts, DMA, pin mux, or clock trees. That is the essence of **middleware abstraction**.

6.2 A transport shim that frames UART bytes into 32-bit ARINC words

```
// arinc429_transport_uart.c — binds serial_if_t to the ARINC 429 parser
#include "serial_if.h"
#include "arinc429.h"
#include <string.h>

typedef void (*arinc_rx_callback_t)(const arinc429_word_t *w, void *user);

typedef struct {
    serial_if_t *ser;
    uint8_t      buf[4];
    size_t       fill;
    arinc_rx_callback_t cb;
    void        *user;
} arinc_uart_ctx_t;

static void on_serial_bytes(const uint8_t *data, size_t len, void *user)
{
    arinc_uart_ctx_t *ctx = (arinc_uart_ctx_t*)user;
    for (size_t i = 0; i < len; ++i) {
        ctx->buf[ctx->fill++] = data[i];
        if (ctx->fill == 4U) {
            uint32_t w = (uint32_t)ctx->buf[0] << 24 |
                         (uint32_t)ctx->buf[1] << 16 |
```

```
                        (uint32_t)ctx->buf[2] << 8  |
                        (uint32_t)ctx->buf[3];
            arinc429_word_t parsed = arinc429_parse(w);
            if (ctx->cb) ctx->cb(&parsed, ctx->user);
            ctx->fill = 0U;
        }
    }
}

void arinc_uart_open(arinc_uart_ctx_t *ctx, uint32_t baud,
arinc_rx_callback_t cb, void *user)
{
    memset(ctx, 0, sizeof *ctx);
    ctx->cb   = cb;
    ctx->user = user;
    ctx->ser  = serial_open(baud, on_serial_bytes, ctx);
}
```

Again, the transport depends only on the **Serial** façade, not on LPUART itself. If the adapter ever moves to SPI or USB CDC, only the serial implementation changes.

---

7. Lab sequence (theory + hands-on)

Each lab includes a **goal**, **what to read in the SDK**, **step-by-step work**, and **checkpoints**. Code uses the SDK include files exactly as shipped in your archive so it builds in MCUXpresso or your preferred IDE.

Lab 1 — Identify LLDD vs middleware in the SDK tree

Goal. Be able to point to LLDD, board files, component adapters, and middleware in the supplied SDK.

Work. Explore these folders in your archive: `devices/MIMXRT1052/drivers/`, `boards/evkbimxrt1050/project_template/`, `components/` (look for `fsl_adapter_*`), and `middleware/`.

Checkpoint. You can name three LLDDs you will use this week (for example `fsl_lpuart`, `fsl_gpio`, `fsl_edma`) and one middleware (for example `fatfs`).

Lab 2 — Build a minimal bare-metal project template

Goal. Bring up clocks and pins on EVKB and print a banner on the debug console.

Work. Start from `boards/evkbimxrt1050/project_template/`. Create `main.c` and call `BOARD_InitBootPins(); BOARD_InitBootClocks();`. Initialize the SDK debug console or,

if you will use **LPUART1** for the ADK-8582, keep the console off and use **LPUART3** for the adapter.

Checkpoint. You can toggle the user LED and print one line of text.

Lab 3 — Implement the Serial façade on top of LPUART (interrupt mode)

Goal. Expose a UART as a portable, testable API without leaking NXP types.

Work. Add `serial_if.h` and `serial_lpuart.c` from Section 5. Compile and verify you can send and receive bytes when shorting TX/RX or when attaching the ADK-8582.

Checkpoint. A loopback echo application works at 115200 bps and survives bursts.

Lab 4 — Swap in EDMA receive to demonstrate abstraction power

Goal. Sustain continuous reception with minimal ISR time using the same `serial_if_t` API.

Work. Create `serial_lpuart_edma.c`. Use `EDMA_CreateHandle`, `LPUART_TransferCreateHandleEDMA`, and a circular `LPUART_TransferReceiveEDMA` into a power-of-two buffer. Keep the public header unchanged.

Checkpoint. Your middleware and application build without changes while CPU usage drops measurably during high-rate tests.

Lab 5 — Build ARINC 429 middleware and connect it to the serial façade

Goal. Parse 32-bit ARINC words arriving over UART from the ADK-8582 and surface clean events to the application.

Work. Add `arinc429.h` and `arinc429_transport_uart.c` from Section 6. Configure the ADK-8582 to emit raw words (endianness as per its user guide). Log each parsed word to the console or to an in-memory ring.

Checkpoint. You can print `Label`, `SDI`, `Data`, `SSM`, and `Parity OK` for a live stream.

Lab 6 — Middleware using FatFS: persistent ARINC log

Goal. Demonstrate a classic middleware-over-HAL stack by logging parsed words to an SD card.

Work. Enable `middleware/fatfs` and the SDMMC LLDD for the EVKB. Use a single writer task substitute (bare-metal state machine) that formats CSV. The only functions it calls outside the middleware are `serial_*` and FatFS `f_*` functions.

Checkpoint. After running for five minutes, your CSV has only complete lines and timestamps are monotonic.

8. An avionics-realistic scenario

Scenario. You are integrating a **Remote Data Concentrator (RDC)** for an **Airspeed/Attitude/Heading Reference System (AAHRS)** upgrade. The RDC must ingest multiple **ARINC 429** labels, filter them down to a subset, and log suspect parity to non-volatile storage for maintenance download.

Constraints. Change rates are high during engine start; bus has a mixture of 12.5 and 100 kbps. The aircraft power bus brown-outs can occur during taxi.

Design using our abstractions.

- The **application** installs a label filter and an alarm policy (for example, three successive parity failures on label `0x20` (Pressure Altitude) triggers `MAINT` annunciation).
- The **ARINC middleware** parses words, checks parity, and timestamps frames with a monotonic counter sourced from the SysTick LLDD.
- The **Serial façade** is swapped from interrupt to EDMA during flight test by flipping one source file, with no changes to middleware or application.
- The **FatFS middleware** uses a pre-allocated file to avoid fragmentation and depends only on the storage HAL; the storage HAL depends on **fsl_sdmmc** LLDD.

This separation lets you stress each part independently for **DO-178C** verification: inject synthetic words into the middleware without touching UART; or DRAM-emulate SD card failures without touching the ARINC parser.

9. Application skeleton that ties it together (compiles against your SDK)

```
// main.c — EVKB-IMXRT1050
#include <stdio.h>
#include "board.h"
#include "serial_if.h"
#include "arinc429.h"

static void on_arinc(const arinc429_word_t *w, void *user)
{
    (void)user;
    printf("LBL=%03o SDI=%u DATA=0x%05X SSM=%u PAR=%u %s\r\n",
           w->label, w->sdi, (unsigned)w->data, w->ssm, w->parity,
           w->parity_ok ? "OK" : "BAD");
```

```
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    // Optionally: BOARD_InitDebugConsole();  // avoid if using LPUART1 for
ADK

    // Open ARINC over UART (ADK-8582) using LPUART3 at 115200 bps
    extern void arinc_uart_open(void *ctx, uint32_t baud, void *cb, void
*user);
    typedef struct { void *opaque[8]; } arinc_uart_ctx_t; // match transport
size
    static arinc_uart_ctx_t ctx;
    arinc_uart_open(&ctx, 115200, (void*)on_arinc, NULL);

    // Idle forever; callbacks deliver events
    for(;;) {
        __WFI(); // Wait-For-Interrupt for low jitter
    }
}
```

This skeleton uses only SDK board init and the LLDDs indirectly via the serial implementation. It will compile inside a standard EVKB project once you add the two small modules above and enable the pin-mux for LPUART3.

---

10. Best practices checklist for Airbus-grade code

Determinism and timing. Keep ISR work minimal; defer heavy parsing to non-ISR context. Bound all blocking calls. Use a single timebase for all middleware timestamps.

No dynamic allocation in flight code. Allocate buffers statically; prove margins with worst-case ARINC rates.

Stable error model. Normalize LLDD error returns into a project enum; never leak vendor status codes past the HAL boundary.

Configuration discipline. Collect all board-specifics (pins, clock roots, DMA channels) into one module per peripheral. Do not scatter IOMUXC writes across the codebase.

Reentrancy and concurrency. For bare-metal, make middleware reentrant for one producer/one consumer at minimum. For multi-context access, document locking at the API.

Logging. Separate flight log from maintenance log. Avoid printing in ISRs. Use loss-tolerant ring buffers and watermarking.

Testability. Provide stubbed serial implementations for unit tests that feed known ARINC sequences, including parity faults and label bursts. Tie test vectors to requirements for traceability.

MISRA-C (Motor Industry Software Reliability Association C) alignment. Follow MISRA-C:2012 guidelines. Wrap vendor headers if you must deviate; document all justifications.

Portability beyond EVKB. Keep all `fsl_*` includes in SoC-specific files only. HAL headers include only `<stdint.h>` and project headers.

Versioning. Record the exact SDK version and driver versions (for example `FSL_LPUART_DRIVER_VERSION`) in the build banner.

---

11. Assessment and extension

To demonstrate mastery, you will:

1) Replace the interrupt serial backend with an EDMA backend without touching middleware or the application.
2) Add a second transport (for example SPI) to show that the ARINC middleware is truly link-agnostic.
3) Log ARINC label `203 octal` rate and create a maintenance report by replaying CSV into a Python script on the ground.

---

12. Troubleshooting notes specific to EVKB-IMXRT1050

- The on-board debug console often uses **LPUART1**. If you attach the ADK-8582 there, disable the console or move the adapter to **LPUART3** (pins `GPIO_AD_B1_06` TX, `GPIO_AD_B1_07` RX).
- Ensure `BOARD_InitBootClocks()` selects a clock root that meets your UART baud tolerance; 24 MHz OSC is fine for 115200 bps. High-rate ARINC gateways benefit from PLL-derived roots.
- If characters drop: check for FIFO configuration (enable RX FIFO in `LPUART_EnableRx` path) and confirm ISR priorities do not preempt critical timing.

---

13. What to hand in for this module

- Your `serial_if.h`, one of `serial_lpuart.c` (ISR) or `serial_lpuart_edma.c` (EDMA), and the two ARINC files.
- A short README describing which UART instance and pins you used on the EVKB, and how the ADK-8582 is wired.
- A five-minute capture file (CSV) with parsed ARINC fields and parity status, produced by your middleware.
