API design principles for bare-metal avionics on EVKB-IMXRT1050

Why API design matters in bare-metal avionics

An API is a **contract**: a set of types, functions, rules, and guarantees that let one module use another without knowing its internals. In **bare-metal** systems (no operating system, or a minimal scheduler), an API also encodes *timing*, *resource*, and *safety* expectations. It is how your application software "shakes hands" with silicon: registers, clocks, interrupts, Direct Memory Access (**DMA**), non-volatile memory (**NVM**), and pins.

Designing APIs for avionics adds constraints: determinism, traceability, and certifiability (for example, **DO-178C**—Software Considerations in Airborne Systems and Equipment Certification—and system-level guidance like **ARP4754A**). Even in a classroom environment, design as though certification could follow later: explicit preconditions, bounded execution time, predictable resource usage, disciplined error handling, and excellent documentation.

> In embedded work we often think "the driver is the product." In avionics, the **API quality is the product** because it controls how safely, clearly, and testably the flight application uses the hardware.

Core principles (with embedded context)

1) Explicit contracts: preconditions, postconditions, invariants

State what must be true before entry (preconditions), what is guaranteed on return (postconditions), and what remains true across calls (invariants). In C, enforce via documentation and defensive checks. For example, a UART "send" API must state whether it blocks, which buffers it touches, and whether it is safe from an **ISR (Interrupt Service Routine)**.

2) Determinism and bounded execution time

For safety-critical or time-sensitive paths, keep calls **O(1)** with known worst-case latency. Avoid hidden loops, dynamic allocation, and unbounded retries. If a call *can* block, say how long and under what conditions.

3) Separation of concerns and layers

Keep clear layers: - **HAL (Hardware Abstraction Layer)** / drivers: thin, register-level, vendor-supplied (`fsl_lpuart_*`, `fsl_pit_*`, `fsl_gpio_*`). - **Service**: device-independent capabilities built on drivers (e.g., serial framed transport, timers, debounced inputs). - **Application**: domain logic (e.g., an **ARINC 429** word scheduler).

4) Consistent naming, types, and error codes

Follow MCUXpresso conventions. Use `status_t` (`kStatus_Success`, `kStatus_Fail`, `kStatus_InvalidArgument`) from `fsl_common.h`. Keep function names verb-first and explicit: `UartService_Init`, `UartService_SendFrame`, `Arinc429Bridge_SendWord`.

5) Ownership and lifetime

Be explicit about who owns TX/RX buffers and for how long. Caller-owned TX buffers with "safe to reuse after callback" semantics are common; RX often uses a fixed driver-owned ring.

6) Concurrency and reentrancy

Concurrency is typically **main loop + ISRs**. Decide what is callable from ISRs; document and enforce it. Keep ISRs short; push work to main loop or DMA. Protect shared state with short critical sections.

7) Configuration surface: compile-time vs run-time

Clock trees, pin mux, and instance selection belong at **compile time** (`mcux_config.h`, `pin_mux.c/h`, `clock_config.c/h`). Runtime tuning (baud rate, parity, timeouts) lives in config structs.

8) Telemetry, diagnostics, and **BITE (Built-In Test Equipment)**

Design obvious places to insert counters, last-error codes, loopback tests, and timestamped events. Deterministic diagnostics are priceless during flight test.

9) Versioning and compatibility

Expose `MAJOR.MINOR.PATCH` and avoid breaking source compatibility in a patch. Place version macros in headers and provide `GetVersion()`.

10) Documentation that doubles as verification

Use **Doxygen** comments with `@pre`, `@post`, *timing*, *calling context*, and *error codes*. Ensure examples compile. If a reviewer can "execute" your docs mentally, your API is on track.

A minimal, generic example: digital output service on top of `fsl_gpio`

This simple **Digital Output** service demonstrates explicit contracts, compile-time pin selection, and clear error handling. It is intentionally straightforward and testable.

digital_out.h

```c
/**
 * @file digital_out.h
 * @brief Simple digital output API on top of NXP fsl_gpio.
 * @version 1.0.0
 *
 * @pre BOARD_InitHardware() has configured clocks and pin muxing for the
target pin.
 * @note This API is not ISR-safe (calls may touch registers that stall
briefly).
 */
#ifndef DIGITAL_OUT_H
#define DIGITAL_OUT_H

#include <stdint.h>
#include <stdbool.h>
#include "fsl_gpio.h"
#include "fsl_common.h"

typedef struct
{
    GPIO_Type   *base;      /**< GPIO base (e.g., GPIO1) */
    uint32_t     pin;       /**< Pin number within the port */
    uint32_t     pinMask;   /**< (1u << pin) cached mask */
    bool         activeHigh;/**< TRUE: logic 1 drives high; FALSE: logic 1
drives low */
} DigitalOut;

/** Initialize a DigitalOut. Idempotent for the same instance.
 *  @pre out != NULL; base != NULL; pin <= 31
 *  @post Output is driven low (logical 0).
 */
status_t DigitalOut_Init(DigitalOut *out, GPIO_Type *base, uint32_t pin, bool
activeHigh);

/** Drive logical value (0/1) to the output. */
static inline void DigitalOut_Write(const DigitalOut *out, bool value)
{
    if ((value && out->activeHigh) || (!value && !out->activeHigh))
    {
        GPIO_PinWrite(out->base, out->pin, 1U);
    }
    else
    {
        GPIO_PinWrite(out->base, out->pin, 0U);
    }
}

/** Toggle output state (purely physical toggle). */
static inline void DigitalOut_Toggle(const DigitalOut *out)
{
    GPIO_PortToggle(out->base, out->pinMask);
}

#endif /* DIGITAL_OUT_H */
```

digital_out.c

```
#include "digital_out.h"

status_t DigitalOut_Init(DigitalOut *out, GPIO_Type *base, uint32_t pin, bool
activeHigh)
{
    if (!out || !base || pin > 31U)
    {
        return kStatus_InvalidArgument;
    }
    out->base       = base;
    out->pin        = pin;
    out->pinMask    = (1UL << pin);
    out->activeHigh = activeHigh;

    /* Configure as output, default level 0. */
    gpio_pin_config_t cfg = {
        .direction = kGPIO_DigitalOutput,
        .outputLogic = 0U,
        .interruptMode = kGPIO_NoIntmode
    };
    GPIO_PinInit(base, pin, &cfg);
    return kStatus_Success;
}
```

Realistic avionics use case: ARINC 429 over a UART bridge (EVKB-IMXRT1050 ↔ ADK-8582)

ARINC 429 is a unidirectional, self-clocking, two-wire differential serial bus widely used in transport aircraft. Each **word** is 32 bits: **Label** (bits 1–8), **SDI (Source/Destination Identifier)** (bits 9–10), **Data** (bits 11–29), **SSM (Sign/Status Matrix)** (bits 30–31), and even-parity bit (bit 32). Data rates are **12.5 kbps** (low-speed) and **100 kbps** (high-speed). Hardware interfaces convert between ARINC 429 signaling and a host interface.

In this lab, an **ADK-8582** transceiver/adapter provides ARINC 429 TX/RX and exposes its control/data plane over **UART**. We will design a small service (**UartService**) and a domain layer (**Arinc429Bridge**) that:

1. Frames and sends an ARINC 429 word to the module for transmission on a bus.
2. Receives framed ARINC 429 words from the module and hands them to the application.
3. Schedules periodic transmission of specific labels with deterministic timing.

The design is layered and testable: the UART service knows nothing about ARINC 429, and the ARINC 429 bridge knows nothing about `LPUART_Type` registers.

Service layer: framed UART transport on top of `fsl_lpuart`

Why a service layer? Because every UART-speaking module is different but initialization, non-blocking send/receive, ring buffers, and callbacks are largely the same. Capture it once and reuse it.

Contract (guarantees): - **Non-blocking** TX using the SDK handle (`lpuart_handle_t`) and interrupts; RX via a driver-owned ring and byte polling. - **Service-owned TX buffer** that persists until the transfer completes (safe for non-blocking DMA/interrupt transfers). - **Driver-owned RX ring** (fixed capacity, overwrite policy documented). - **Frame boundaries** recognized by Start-Length-Payload-Checksum. - **ISR-safe internals; application callbacks fire from the main loop** via `UartService_Poll()`.

Frame format: `0x55 | LEN | PAYLOAD[LEN] | CHECKSUM` where `CHECKSUM` is the 8-bit two's-complement of the sum of `LEN` and `PAYLOAD` (sum + checksum == 0).

`uart_service.h`

```
/**
 * @file uart_service.h
 * @brief Framed UART service built on NXP fsl_lpuart for EVKB-IMXRT1050.
 * @version 1.0.0
 *
 * Frame format: 0x55 | LEN | PAYLOAD[LEN] | CHECKSUM
 * CHECKSUM = 8-bit sum of LEN and PAYLOAD, two's complement
(sum+checksum==0)
 *
 * @note All callbacks fire from UartService_Poll(), not from the ISR.
 */
#ifndef UART_SERVICE_H
#define UART_SERVICE_H

#include <stdint.h>
#include <stdbool.h>
#include "fsl_lpuart.h"
```

```c
#include "fsl_common.h"

#ifndef UART_SERVICE_RX_RING_BYTES
#define UART_SERVICE_RX_RING_BYTES (256u)
#endif

#define UART_SERVICE_MAX_PAYLOAD   (255u)

typedef struct
{
    LPUART_Type       *base;
    lpuart_handle_t    handle;

    /* RX ring */
    uint8_t            rxRing[UART_SERVICE_RX_RING_BYTES];
    volatile uint16_t  rxHead;
    volatile uint16_t  rxTail;

    /* TX state (service-owned buffer to ensure persistence during non-
blocking TX) */
    uint8_t            txBuf[1 + 1 + UART_SERVICE_MAX_PAYLOAD + 1];
    uint8_t            txLen;
    volatile bool      txBusy;
    volatile bool      txDonePending;

    /* App callbacks */
    void (*onFrame)(const uint8_t *payload, uint8_t len, void *cookie);
    void (*onTxDone)(void *cookie);
    void *cookie;

    /* Diagnostics */
    uint32_t           checksumErrors;
} UartService;

/** Initialize LPUART and the service. Non-blocking TX; RX via byte polling.
 *  @pre BOARD_InitBootPins(), BOARD_InitBootClocks(), pin mux for selected
LPUART instance.
 *  @pre config != NULL; base != NULL
 *  @post LPUART enabled; internal buffers empty; callbacks cleared.
 */
status_t UartService_Init(UartService *svc, LPUART_Type *base, const
lpuart_config_t *config);

/** Queue a framed payload for transmit (non-blocking). Returns kStatus_Busy
if TX in progress. */
status_t UartService_Send(UartService *svc, const uint8_t *payload, uint8_t
len);
```

```
/** Polls RX, parses frames, and dispatches callbacks. Call from the main
loop at ~1 kHz or faster. */
void UartService_Poll(UartService *svc);

/** Optional: change callbacks at runtime. */
static inline void UartService_SetCallbacks(UartService *svc,
                                            void (*onFrame)(const uint8_t*,
uint8_t, void*),
                                            void (*onTxDone)(void*),
                                            void *cookie)
{
    svc->onFrame = onFrame;
    svc->onTxDone = onTxDone;
    svc->cookie = cookie;
}

#endif /* UART_SERVICE_H */
```

`uart_service.c` (C-only; no C++ constructs)

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

#define START_BYTE  (0x55u)

static void LpuartUserCallback(LPUART_Type *base, lpuart_handle_t *handle,
status_t status, void *userData)
{
    (void)base; (void)handle;
    UartService *svc = (UartService*)userData;
    if (status == kStatus_LPUART_TxIdle)
    {
        svc->txBusy = false;
        svc->txDonePending = true; /* Defer app callback to Poll() */
    }
}

status_t UartService_Init(UartService *svc, LPUART_Type *base, const
lpuart_config_t *config)
{
    if (!svc || !base || !config) return kStatus_InvalidArgument;

    memset(svc, 0, sizeof(*svc));
    svc->base = base;

    /* Use the UART clock selected by your clock_config.c; adjust if needed
for your board setup. */
    LPUART_Init(base, config, CLOCK_GetFreq(kCLOCK_UartClk));
    LPUART_TransferCreateHandle(base, &svc->handle, LpuartUserCallback, svc);
```

```c
    return kStatus_Success;
}

static uint8_t checksum8(const uint8_t *data, uint8_t len)
{
    uint16_t sum = 0;
    for (uint8_t i = 0; i < len; ++i) sum = (uint16_t)(sum + data[i]);
    return (uint8_t)(0u - (uint8_t)sum);
}

status_t UartService_Send(UartService *svc, const uint8_t *payload, uint8_t
len)
{
    if (!svc || !payload || len == 0u || len > UART_SERVICE_MAX_PAYLOAD)
return kStatus_InvalidArgument;
    if (svc->txBusy) return kStatus_Busy;

    /* Build the frame into the service-owned buffer so it persists during
non-blocking TX. */
    svc->txBuf[0] = START_BYTE;
    svc->txBuf[1] = len;
    (void)memcpy(&svc->txBuf[2], payload, len);
    svc->txBuf[2u + len] = checksum8(&svc->txBuf[1], (uint8_t)(1u + len));
    svc->txLen = (uint8_t)(len + 3u);

    lpuart_transfer_t xfer = {
        .data = svc->txBuf,
        .dataSize = svc->txLen
    };
    svc->txBusy = true;
    svc->txDonePending = false;
    status_t st = LPUART_TransferSendNonBlocking(svc->base, &svc->handle,
&xfer);
    if (st != kStatus_Success) { svc->txBusy = false; return st; }
    return kStatus_Success;
}

void UartService_Poll(UartService *svc)
{
    if (!svc) return;

    /* RX: Drain any available bytes into the ring */
    while ((LPUART_GetStatusFlags(svc->base) &
(uint32_t)kLPUART_RxDataRegFullFlag) != 0u)
    {
        uint8_t b = (uint8_t)LPUART_ReadByte(svc->base);
        uint16_t next = (uint16_t)((svc->rxHead + 1u) %
UART_SERVICE_RX_RING_BYTES);
```

```
        if (next != svc->rxTail) { svc->rxRing[svc->rxHead] = b; svc->rxHead
= next; }
        else { /* overflow policy: drop oldest */
            svc->rxTail = (uint16_t)((svc->rxTail + 1u) %
UART_SERVICE_RX_RING_BYTES);
            svc->rxRing[svc->rxHead] = b; svc->rxHead = next;
        }
    }

    /* TX completion notification */
    if (svc->txDonePending)
    {
        svc->txDonePending = false;
        if (svc->onTxDone) { svc->onTxDone(svc->cookie); }
    }

    /* Frame parser: search for START, then expect LEN, PAYLOAD, CHK */
    for (;;)
    {
        /* Find START byte */
        uint16_t t = svc->rxTail;
        bool found = false;
        while (t != svc->rxHead)
        {
            if (svc->rxRing[t] == START_BYTE) { found = true; break; }
            t = (uint16_t)((t + 1u) % UART_SERVICE_RX_RING_BYTES);
            svc->rxTail = t; /* discard until START */
        }
        if (!found) break; /* need more data */

        /* We have START at rxTail. Check if LEN is present */
        uint16_t lenIndex = (uint16_t)((svc->rxTail + 1u) %
UART_SERVICE_RX_RING_BYTES);
        if (lenIndex == svc->rxHead) break; /* not enough data for LEN */
        uint8_t len = svc->rxRing[lenIndex];

        uint16_t needed = (uint16_t)(len + 3u); /* START + LEN + PAYLOAD +
CHK */
        /* Compute bytes available in ring from tail to head (modulo) */
        uint16_t avail = (svc->rxHead >= svc->rxTail)
                       ? (uint16_t)(svc->rxHead - svc->rxTail)
                       : (uint16_t)(UART_SERVICE_RX_RING_BYTES - svc->rxTail
+ svc->rxHead);
        if (avail < needed) break; /* wait for full frame */

        /* Copy frame into a temp buffer for checksum and callback */
        uint8_t frame[1 + 1 + UART_SERVICE_MAX_PAYLOAD + 1];
        for (uint16_t i = 0u; i < needed; ++i)
        {
            frame[i] = svc->rxRing[svc->rxTail];
            svc->rxTail = (uint16_t)((svc->rxTail + 1u) %
UART_SERVICE_RX_RING_BYTES);
        }

        /* Validate checksum over LEN+PAYLOAD */
        uint8_t chk = frame[needed - 1u];
        if ((uint8_t)(checksum8(&frame[1], (uint8_t)(needed - 2u)) + chk) ==
0u)
        {
            if (svc->onFrame) { svc->onFrame(&frame[2], frame[1], svc-
>cookie); }
        }
        else
        {
            svc->checksumErrors++;
        }
    }
}
```

Domain layer: ARINC 429 bridge on the UART service

The **Arinc429Bridge** takes **words** as 32-bit values (with parity provided or computed) and packages them into UART frames the module understands (for training we assume a command `0xA1` “TXWORD” and `0xB1` “RXWORD”). It also implements a tiny scheduler to transmit certain labels periodically using a millisecond tick advanced from the main loop.

`arinc429_bridge.h`

```
/**
 * @file arinc429_bridge.h
 * @brief ARINC 429 bridge over UART service (EVKB-IMXRT1050 + ADK-8582).
 * @version 1.0.0
 *
 * @note The on-wire UART command identifiers (0xA1/0xB1) are training
placeholders.
 *       Adapt to the ADK-8582 specification you use in the lab.
 */
#ifndef ARINC429_BRIDGE_H
#define ARINC429_BRIDGE_H

#include <stdint.h>
#include <stdbool.h>
#include "fsl_common.h"
#include "uart_service.h"

typedef void (*ArincRxCallback)(uint32_t word, void *cookie);
```

```c
typedef struct
{
    UartService     *uart;
    ArincRxCallback  onWord;
    void            *cookie;
    /* simple periodic scheduler state */
    uint32_t         msNow;
    uint32_t         nextTxMs;
    uint32_t         periodMs;
    uint32_t         scheduledWord;
    bool             scheduleActive;
} Arinc429Bridge;

status_t Arinc429Bridge_Init(Arinc429Bridge *b, UartService *svc,
ArincRxCallback onWord, void *cookie);
status_t Arinc429Bridge_SendWord(Arinc429Bridge *b, uint32_t word, bool
computeParity);
void     Arinc429Bridge_Schedule(Arinc429Bridge *b, uint32_t word, uint32_t
periodMs, bool computeParity);
void     Arinc429Bridge_Poll(Arinc429Bridge *b, uint32_t elapsedMs);

#endif /* ARINC429_BRIDGE_H */
```

arinc429_bridge.c

```c
#include "arinc429_bridge.h"

static uint32_t arinc_set_even_parity(uint32_t word)
{
    /* Bits 1..31 form the parity computation; treat bit 31 as the parity bit
location in a 0..31 packing. */
    uint32_t w = word & 0x7FFFFFFFu; /* clear parity bit */
    uint32_t ones = 0u;
    for (uint8_t i = 0; i < 31; ++i) { ones += (w >> i) & 1u; }
    uint32_t parity = (ones & 1u) ? 1u : 0u; /* even parity: set bit if ones
is odd */
    return (w | (parity << 31));
}

static void on_uart_frame(const uint8_t *payload, uint8_t len, void *cookie)
{
    Arinc429Bridge *b = (Arinc429Bridge*)cookie;
    /* Assume 0xB1 RXWORD response: [0xB1][W0][W1][W2][W3] (big endian) */
    if (len == 5u && payload[0] == 0xB1u)
    {
        uint32_t word = ((uint32_t)payload[1] << 24) |
                        ((uint32_t)payload[2] << 16) |
                        ((uint32_t)payload[3] << 8)  |
                        ((uint32_t)payload[4]);
        if (b->onWord) b->onWord(word, b->cookie);
    }
}

status_t Arinc429Bridge_Init(Arinc429Bridge *b, UartService *svc,
ArincRxCallback onWord, void *cookie)
{
    if (!b || !svc) return kStatus_InvalidArgument;
    b->uart = svc;
    b->onWord = onWord;
    b->cookie = cookie;
    b->msNow = 0u;
    b->nextTxMs = 0u;
    b->periodMs = 0u;
    b->scheduledWord = 0u;
    b->scheduleActive = false;

    UartService_SetCallbacks(svc, on_uart_frame, NULL, b);
    return kStatus_Success;
}

status_t Arinc429Bridge_SendWord(Arinc429Bridge *b, uint32_t word, bool
computeParity)
{
    if (!b) return kStatus_InvalidArgument;
    uint32_t w = computeParity ? arinc_set_even_parity(word) : word;
    uint8_t frame[1 + 4];
    frame[0] = 0xA1u; /* TXWORD */
    frame[1] = (uint8_t)(w >> 24);
    frame[2] = (uint8_t)(w >> 16);
    frame[3] = (uint8_t)(w >> 8);
    frame[4] = (uint8_t)(w);
    return UartService_Send(b->uart, frame, (uint8_t)sizeof(frame));
}

void Arinc429Bridge_Schedule(Arinc429Bridge *b, uint32_t word, uint32_t
periodMs, bool computeParity)
{
    if (!b) return;
    b->scheduledWord = computeParity ? arinc_set_even_parity(word) : word;
    b->periodMs = periodMs;
    b->msNow = 0u;
    b->nextTxMs = 0u;
    b->scheduleActive = (periodMs != 0u);
}

void Arinc429Bridge_Poll(Arinc429Bridge *b, uint32_t elapsedMs)
{
    if (!b) return;
    b->msNow += elapsedMs;

    /* Always poll the UART service first */
    UartService_Poll(b->uart);

    if (b->scheduleActive && (b->msNow >= b->nextTxMs))
    {
        (void)Arinc429Bridge_SendWord(b, b->scheduledWord, false);
        b->nextTxMs += b->periodMs;
    }
}
```

Application wiring on EVKB-IMXRT1050

A minimal `main.c` illustrates the EVKB SDK pattern. In your project, select the LPUART instance and pins connected to the ADK-8582.

```
#include "board.h"
#include "clock_config.h"
#include "pin_mux.h"
#include "fsl_common.h"
#include "fsl_lpuart.h"
#include "uart_service.h"
#include "arinc429_bridge.h"

static void on_arinc_rx(uint32_t word, void *cookie)
{
    (void)cookie;
    /* TODO: decode Label, SDI, DATA, SSM and route to application */
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole(); /* Optional: PRINTF via LPUART */

    /* Configure the LPUART instance used for ADK-8582 connection (e.g.,
LPUART1).
       Ensure pin mux matches your hardware. */
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = 115200U;
    cfg.enableTx = true;
    cfg.enableRx = true;

    UartService svc;
    (void)UartService_Init(&svc, LPUART1, &cfg);

    Arinc429Bridge bridge;
    (void)Arinc429Bridge_Init(&bridge, &svc, on_arinc_rx, NULL);

    /* Example word: set your packing convention and compute parity in
SendWord/Schedule. */
    uint32_t word = 0u;
    /* word |= ... pack Label, SDI, Data, SSM ... */

    Arinc429Bridge_Schedule(&bridge, word, 100u, true); /* 100 ms */

    /* Coarse 1 ms tick using delay (replace with PIT-based tick in Exercise
1). */
    while (1)
    {
        SDK_DelayAtLeastUs(1000U, SystemCoreClock);
        Arinc429Bridge_Poll(&bridge, 1u);
    }
}
```

Replace the crude `SDK_DelayAtLeastUs` loop with a **PIT (Periodic Interrupt Timer)** or **SysTick** tick for precise scheduling (Exercise 1).

Best practices (avionics-focused)

1. **Define bit packing unambiguously.** ARINC 429 literature varies on “bit 1” orientation. In your header, define the canonical packing and stick to it. Provide pack/unpack and parity helpers that match your convention.
2. **Avoid hidden dynamic memory.** Use fixed-size buffers with explicit capacities. If you must reserve memory, do it at initialization and expose capacity.
3. **Fail early, fail loud.** Return `kStatus_InvalidArgument` for null pointers and out-of-range values. In test builds, use `assert()`; in production, return explicit errors and bump counters.
4. **Callbacks from main loop, not ISRs** (unless documented). Keep ISRs short and predictable.
5. **One job per API.** `*_Init()` should only initialize.
6. **Time and memory budgets are part of the contract.** Document “≤ 3 µs at 600 MHz” or “uses 512 B RX ring.” These anchor verification.
7. **Version every header.** Include version macros and a `GetVersion()`; maintain a short changelog.
8. **MISRA-C friendliness.** Avoid tricky macros, side effects, and implicit conversions; ease static analysis.

9. **Deterministic diagnostics.** Count checksum errors, buffer overflows, and last error codes; expose getters.
10. **Pin/clock/DMA live outside APIs.** Keep pin mux and clock dependencies in `pin_mux.c/h` and `clock_config.c/h`.

Hands-on exercises (EVKB-IMXRT1050 + ADK-8582)

Each exercise states a goal, step-by-step guidance, and a reference solution using the EVKB SDK drivers.

Exercise 1 — PIT-based tick driving the ARINC 429 scheduler

Goal. Replace the delay loop with a **PIT**-driven millisecond tick and show that scheduled ARINC 429 transmissions maintain **±1 ms jitter** while the main loop does unrelated work.

Steps. 1. Enable the PIT clock and NVIC interrupt; include `fsl_pit.h`. 2. Configure Channel 0 for a 1 ms period. 3. In the PIT ISR, increment a `volatile uint32_t g_msTick` and set a flag. 4. In `main`, consume the `g_msTick` delta and call `Arinc429Bridge_Poll(&bridge, deltaMs)`. 5. Add a background CPU load (e.g., checksum over a buffer) to demonstrate robust timing.

Reference solution (core files).

`tick.h`

```c
#ifndef TICK_H
#define TICK_H
#include <stdint.h>
void     Tick_Init(void);
uint32_t Tick_ConsumeDeltaMs(void); /* returns elapsed ms since last call */
#endif
```

`tick.c`

```c
#include "tick.h"
#include "fsl_pit.h"
#include "fsl_common.h"

static volatile uint32_t s_msTick = 0u;
static uint32_t s_last = 0u;

void PIT0_IRQHandler(void)
{
    PIT_ClearStatusFlags(PIT, kPIT_Chnl_0, kPIT_TimerFlag);
    s_msTick++;
    SDK_ISR_EXIT_BARRIER;
}
```

```
void Tick_Init(void)
{
    pit_config_t cfg;
    PIT_GetDefaultConfig(&cfg);
    PIT_Init(PIT, &cfg);
    uint32_t srcClk = CLOCK_GetFreq(kCLOCK_BusClk);
    PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000U, srcClk)); /* 1
ms */
    PIT_EnableInterrupts(PIT, kPIT_Chnl_0, kPIT_TimerInterruptEnable);
    EnableIRQ(PIT_IRQn);
    PIT_StartTimer(PIT, kPIT_Chnl_0);
}

uint32_t Tick_ConsumeDeltaMs(void)
{
    uint32_t now = s_msTick;
    uint32_t d = now - s_last;
    s_last = now;
    return d;
}
```

Integrate in `main`:

```
Tick_Init();
for (;;)
{
    uint32_t d = Tick_ConsumeDeltaMs();
    if (d) { Arinc429Bridge_Poll(&bridge, d); }
    /* Background load to show jitter tolerance */
    static uint8_t buf[256]; for (int i=1;i<256;i++) buf[0]^=buf[i];
}
```

What you learn. Separation of the time base (PIT) from the domain logic (ARINC 429 scheduler), ISR hygiene, and a deterministic polling pattern common in certified systems.

Exercise 2 — Flow control and back-pressure in the UART service

Goal. Extend `UartService` to support **CTS/RTS (Clear To Send/Request To Send)** hardware flow control using LPUART settings, and add **back-pressure** so callers see `kStatus_Busy` or automatic queuing without data loss.

Steps. 1. In pin mux, enable RTS/CTS pins for your chosen LPUART instance. 2. In `UartService_Init`, enable flow control: c `lpuart_config_t c = *config; c.enableTxCTS = true; c.enableRxRTS = true; LPUART_Init(base, &c, CLOCK_GetFreq(kCLOCK_UartClk));` 3. Add a small single-frame TX queue: if `txBusy==true`, copy the frame into a spare buffer and mark `txQueued=true`. 4. When the ISR sets `txBusy=false`, if `txQueued` is set, immediately start the queued transfer and keep onTxDone pending until both are done. 5. Add getters for metrics such as dropped frames and checksum errors.

Reference additions (illustrative snippets).

```
/* In UartService struct */
uint8_t  txSpare[1+1+UART_SERVICE_MAX_PAYLOAD+1];
uint8_t  txSpareLen;
bool     txQueued;
uint32_t droppedFrames;

/* In Send(): */
if (svc->txBusy)
{
    if (!svc->txQueued)
    {
        (void)memcpy(svc->txSpare, svc->txBuf, (size_t)svc->txLen);
        svc->txSpareLen = svc->txLen;
        svc->txQueued = true;
        return kStatus_Success; /* buffered */
    }
    svc->droppedFrames++;
    return kStatus_Busy;
}

/* In Poll(), after handling txDonePending: */
if (!svc->txBusy && svc->txQueued)
{
    lpuart_transfer_t xfer = { .data = svc->txSpare, .dataSize = svc-
>txSpareLen };
    svc->txBusy = true; svc->txQueued = false; svc->txDonePending = false;
    (void)LPUART_TransferSendNonBlocking(svc->base, &svc->handle, &xfer);
}
```

What you learn. How requirements (“no data loss”, “honor flow control”) become crisp API contracts with simple, testable behavior.

A second avionics scenario: Air Data Computer (ADC) label fan-out

Scenario. The EVKB receives **pressure altitude** and **indicated airspeed** from an upstream **ADC (Air Data Computer)** over ARINC 429. The mission computer must: (1) validate parity and **SSM (Sign/Status Matrix)** for labels 203 (Pressure Altitude) and 204 (Indicated Airspeed); (2) convert raw engineering units (e.g., **BNR (Binary Number Representation)** or **BCD (Binary-Coded Decimal)**) to internal units; and (3) redistribute those values to another ARINC 429 channel and log periodically to a maintenance UART.

Clean application API:

```
typedef struct {
    int32_t  pressureAlt_ft; /* scaled integer for determinism */
    uint16_t ias_knots;
    bool     valid;
} AirData;

void AirData_OnArincWord(AirData *ad, uint32_t word);
```

Lessons mirror earlier guidance: keep conversions pure and side-effect-free, make validity explicit, and let wiring handle UART and scheduling. This makes unit testing easy on a desktop with the same headers.

Wrap-up: design checklist

- Do functions document preconditions and postconditions?
- Are timing characteristics explicit and bounded?
- Is the concurrency model clear (who can call from ISR, who cannot)?
- Is buffer ownership and lifetime unambiguous?
- Does naming match the SDK and stay consistent?
- Can the physical transport change (e.g., SPI→UART) without touching the application layer?
- Do examples compile, and would a peer understand them without a meeting?

What to do on your EVKB bench (step-by-step)

1. Start from the EVKB **project template** (or the `lpuart/interrupt_transfer` SDK example). Add `uart_service.*` and `arinc429_bridge.*`.
2. Confirm TX/RX pins and, if used, **RTS/CTS** pins in `pin_mux.c/h`.
3. Bring up UART with a USB-to-TTL dongle first; echo frames to validate framing and checksum.
4. Connect the **ADK-8582** and adapt command identifiers (`0xA1`, `0xB1`) and payloads to its specification.
5. Add the **PIT** tick and measure schedule jitter with a logic analyzer.
6. Add deterministic diagnostics (counters, last error) and expose them via getters rather than `PRINTF` in the hot path.

Notes about the MCUXpresso SDK

- The code above uses only standard MCUXpresso drivers (`fsl_*`) available in the EVKB-IMXRT1050 SDK archive you provided.

- Clock source selectors (`CLOCK_GetFreq(kCLOCK_UartClk)` and `kCLOCK_BusClk`) should match your `clock_config.c`. If your project template exposes a helper like `BOARD_DebugConsoleSrcFreq()`, prefer that for the specific LPUART instance.
- Replace placeholders (LPUART instance number, pins, and the ADK-8582 serial command IDs) with your lab hardware's actual values.