Middleware Abstraction & Service Layers on i.MX RT1050 (EVKB)

---

1) Why middleware and service layers even on bare metal?

Bare-metal means no operating system scheduler: you own the reset vector, startup, interrupts, and the main control loop. That does *not* imply firmware must be a tangle of register pokes. A disciplined **layered architecture** keeps hardware churn contained, enables unit testing, and supports design assurance (DO-178C) with clear interfaces and traceability.

We will work toward this stack:

```
Application (mission logic: data acquisition, command/response, health)
└─ Service Layer (logging, time, parameter store, command router, data-bus services)
   └─ Middleware Abstractions (serial port, framed link, checksum, packetizer)
      └─ Drivers/HAL (NXP SDK: fsl_lpuart, fsl_gpio, fsl_iomuxc, clocks)
         └─ BSP (pins, clocks, boot, debug console) + Startup/Linker
```

Goals - **Isolation:** Hardware-specific code lives behind stable C APIs. - **Determinism:** No hidden dynamic allocation or unbounded latency. - **Testability:** Middleware/services test on PC with fakes; only driver shim is board-specific. - **Traceability:** Each requirement maps to a function or interface in a designated layer.

---

2) Core definitions (first use expanded)

- **BSP (Board Support Package):** Board pins, clocks, memory bring-up, debug console init.
- **HAL (Hardware Abstraction Layer):** Thin wrappers around registers or vendor drivers, e.g., NXP's `fsl_lpuart`.
- **Middleware Abstraction:** Hardware-agnostic utilities that *compose* HAL features (e.g., serial framing with CRC16) and present a clean interface.
- **Service Layer:** Domain-oriented components (e.g., ARINC 429 word encode/decode, health monitoring, event logging) that applications call.
- **LRU (Line Replaceable Unit):** Aircraft hardware module; our firmware should be portable across LRUs that use i.MX RT.

- **FCC (Flight Control Computer), FMS (Flight Management System), ADC (Air Data Computer), EICAS (Engine-Indication and Crew-Alerting System):** Typical avionics participants we will reference in scenarios.

---

3) The EVKB-IMXRT1050 SDK as the foundation

We base all hardware interactions on the EVKB SDK you provided. The UART driver API we use is from `fsl_lpuart.h` (e.g., `LPUART_Init`, `LPUART_TransferStartRingBuffer`, `LPUART_TransferSendNonBlocking`, `LPUART_TransferReceiveNonBlocking`). Board utilities provide `BOARD_InitBootPins`, `BOARD_InitBootClocks`, and `BOARD_InitDebugConsole`. Using these APIs keeps our code aligned with NXP support and example projects under `boards/evkbimxrt1050/driver_examples/lpuart/...` in the SDK.

Note on instances: We will use the board's debug LPUART (for logs) and a second LPUART instance for the ARINC converter (ADK-8582). Pin selection is left in the BSP so the same middleware compiles for different EVKs or carrier cards.

---

4) File-tree blueprint (drop straight into an SDK project)

```
src/
  bsp/
    pins.c, pins.h            # IOMUX (fsl_iomuxc), board pin maps
    clocks.c, clocks.h        # BOARD_InitBootClocks(), sources
    board_init.c              # BOARD_InitBootPins/Clocks/DebugConsole
  drivers/
    uart_hw.c, uart_hw.h      # Thin wrapper over fsl_lpuart (HAL)
  middleware/
    serial_port.c, serial_port.h          # Generic serial interface over HAL
    ringbuf.c, ringbuf.h                  # Lockless single-producer/single-
consumer
    crc16_ccitt.c, crc16_ccitt.h          # 0x1021 polynomial
    frame_link.c, frame_link.h            # SLIP-like or custom
header+length+CRC
  services/
    log.c, log.h                          # Timestamped records to debug UART
    timebase.c, timebase.h                # SysTick or GPT based ticks
    params.c, params.h                    # Read-only build-time parameters
    arinc429.c, arinc429.h                # Pack/unpack 32-bit A429 words
    a429_service.c, a429_service.h        # High-level API using UART to
ADK-8582
  app/
    main.c                                # Use services; no direct HAL calls
```

Each folder compiles as part of your MCUXpresso project; only `drivers/` can include `fsl_*` headers. Everything above relies solely on headers inside our project to enforce separation.

---

5) HAL: a minimal UART shim over `fsl_lpuart`

Intent: Make a tiny, stable surface that our middleware uses. If we move to a different MCU, only this file changes.

```
// drivers/uart_hw.h
#pragma once
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_lpuart.h"

typedef struct {
    LPUART_Type *base;
    lpuart_handle_t handle;           // NXP async handle
    uint8_t *rxRing;                  // Provided by caller (middleware)
    size_t rxRingSize;
    volatile bool txBusy;
} uart_hw_t;

// Construction & configuration (no dynamic alloc)
void UART_HW_Init(uart_hw_t *dev, LPUART_Type *base, uint32_t srcClkHz,
uint32_t baud);
void UART_HW_AttachRing(uart_hw_t *dev, uint8_t *ring, size_t size);

// Non-blocking send/receive using NXP transfer APIs
status_t UART_HW_Send(uart_hw_t *dev, const uint8_t *data, size_t len);
status_t UART_HW_Receive(uart_hw_t *dev, uint8_t *dst, size_t len, size_t
*got);

// Pump from ISR
void UART_HW_IrqHandler(uart_hw_t *dev);

// drivers/uart_hw.c
#include "uart_hw.h"

static void uart_cb(LPUART_Type *base, lpuart_handle_t *handle, status_t
status, void *userData) {
    uart_hw_t *dev = (uart_hw_t*)userData;
    if (status == kStatus_LPUART_TxIdle) dev->txBusy = false;
}

void UART_HW_Init(uart_hw_t *dev, LPUART_Type *base, uint32_t srcClkHz,
```

```
uint32_t baud) {
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baud;
    cfg.enableTx = true;
    cfg.enableRx = true;
    dev->base = base;
    dev->txBusy = false;
    LPUART_Init(base, &cfg, srcClkHz);
    LPUART_TransferCreateHandle(base, &dev->handle, uart_cb, dev);
}

void UART_HW_AttachRing(uart_hw_t *dev, uint8_t *ring, size_t size) {
    dev->rxRing = ring; dev->rxRingSize = size;
    LPUART_TransferStartRingBuffer(dev->base, &dev->handle, ring, size);
}

status_t UART_HW_Send(uart_hw_t *dev, const uint8_t *data, size_t len) {
    lpuart_transfer_t x = { .data = (uint8_t*)data, .dataSize = len };
    dev->txBusy = true;
    status_t s = LPUART_TransferSendNonBlocking(dev->base, &dev->handle, &x);
    return s;
}

status_t UART_HW_Receive(uart_hw_t *dev, uint8_t *dst, size_t len, size_t
*got) {
    lpuart_transfer_t x = { .rxData = dst, .dataSize = len };
    size_t rec = 0;
    status_t s = LPUART_TransferReceiveNonBlocking(dev->base, &dev->handle,
&x, &rec);
    if (got) *got = rec;
    return s; // kStatus_Success even if rec < len (non-blocking)
}

void UART_HW_IrqHandler(uart_hw_t *dev) {
    LPUART_TransferHandleIRQ(dev->base, &dev->handle);
}
```

> **BSP binding:** In your `app.h`, tie the appropriate IRQ to `UART_HW_IrqHandler()` and provide `BOARD_DebugConsoleSrcFreq()` for the selected LPUART instance, following the style in SDK examples (`boards/evkbimxrt1050/driver_examples/lpuart/interrupt/app.h`).

---

6) Middleware #1 — A generic `SerialPort` facade

Problem: Upper layers should not depend on NXP types. **Solution:** Provide a tiny façade with read/write and a `poll()` hook; implement it over `uart_hw_t`.

```c
// middleware/serial_port.h
#pragma once
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct serial_port serial_port_t;

typedef struct {
    size_t (*read)(serial_port_t*, uint8_t *dst, size_t max); // non-blocking
    size_t (*write)(serial_port_t*, const uint8_t *src, size_t len); //
schedule async TX
    void   (*poll)(serial_port_t*); // progress TX queues if needed
} serial_vtbl_t;

struct serial_port { const serial_vtbl_t *v; void *impl; };

// Factory over HAL (implementation in .c)
void SerialPort_fromUart(serial_port_t *sp, void *uart_hw_ptr);

// middleware/serial_port.c
#include "serial_port.h"
#include "uart_hw.h"

static size_t sp_read(serial_port_t* s, uint8_t *dst, size_t max) {
    uart_hw_t *u = (uart_hw_t*)s->impl; size_t got = 0;
(void)UART_HW_Receive(u, dst, max, &got); return got;
}
static size_t sp_write(serial_port_t* s, const uint8_t *src, size_t len) {
    uart_hw_t *u = (uart_hw_t*)s->impl; if (UART_HW_Send(u, src, len) !=
kStatus_Success) return 0; return len;
}
static void sp_poll(serial_port_t* s) { (void)s; /* nothing; async IRQ-driven
*/ }

static const serial_vtbl_t V = { sp_read, sp_write, sp_poll };

void SerialPort_fromUart(serial_port_t *sp, void *uart_hw_ptr) { sp->v = &V;
sp->impl = uart_hw_ptr; }
```

---

7) Middleware #2 — Byte ring buffer (for parsing)

Even though the NXP driver offers an RX ring, we keep a light **SPSC (single-producer, single-consumer)** ring for protocol parsing so the framing layer never sees partial multibyte fields.

```c
// middleware/ringbuf.h
#pragma once
#include <stdint.h>
#include <stddef.h>

typedef struct { uint16_t head, tail, cap; uint8_t *buf; } ring8_t;
void rb_init(ring8_t *r, uint8_t *backing, uint16_t cap);
uint16_t rb_avail(const ring8_t *r);
uint16_t rb_space(const ring8_t *r);
uint16_t rb_push(ring8_t *r, const uint8_t *src, uint16_t n);
uint16_t rb_pop(ring8_t *r, uint8_t *dst, uint16_t n);
int rb_peek(const ring8_t *r, uint16_t idx); // -1 if out of range
```

```c
// middleware/ringbuf.c
#include "ringbuf.h"
void rb_init(ring8_t *r, uint8_t *b, uint16_t c){ r->head=r->tail=0; r-
>cap=c; r->buf=b; }
uint16_t rb_avail(const ring8_t *r){ return (r->head - r->tail) & (r->cap-1);
}
uint16_t rb_space(const ring8_t *r){ return (r->cap-1) - rb_avail(r); }
uint16_t rb_push(ring8_t *r, const uint8_t *s, uint16_t n){ uint16_t k=0;
while(k<n && rb_space(r)){ r->buf[r->head++ & (r->cap-1)] = s[k++]; } return
k; }
uint16_t rb_pop(ring8_t *r, uint8_t *d, uint16_t n){ uint16_t k=0; while(k<n
&& rb_avail(r)){ d[k++] = r->buf[r->tail++ & (r->cap-1)]; } return k; }
int rb_peek(const ring8_t *r, uint16_t i){ if(i>=rb_avail(r)) return -1;
return r->buf[(r->tail+i) & (r->cap-1)]; }
```

> **Determinism:** Capacity is a power of two; arithmetic is modulo with masking; functions are O(n) and bounded.

---

8) Middleware #3 — CRC16-CCITT and framed link

We will use a simple **Header + Length + Payload + CRC16** protocol over UART. This is robust on noisy lines and easy to test.

Frame format

```
Sync:   0xA5 0x5A
Len:    16-bit little-endian payload length (N)
Body:   N bytes
CRC16:  CCITT (poly 0x1021, init 0xFFFF) over Len+Body
```

```c
// middleware/crc16_ccitt.h
#pragma once
#include <stddef.h>
#include <stdint.h>
uint16_t crc16_ccitt(const void *data, size_t len, uint16_t init);
```

```c
// middleware/crc16_ccitt.c
#include "crc16_ccitt.h"
uint16_t crc16_ccitt(const void *data, size_t len, uint16_t init){
    const uint8_t *p = (const uint8_t*)data; uint16_t crc = init;
    for(size_t i=0;i<len;i++){ crc ^= (uint16_t)p[i] << 8; for(int
b=0;b<8;b++){ crc = (crc & 0x8000)? (crc<<1)^0x1021 : (crc<<1); } }
    return crc;
}

// middleware/frame_link.h
#pragma once
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "serial_port.h"
#include "ringbuf.h"

typedef struct {
    serial_port_t *sp;
    ring8_t *rx;
} frame_link_t;

void frame_init(frame_link_t *L, serial_port_t *sp, ring8_t *rxbuf);
bool frame_tx(frame_link_t *L, const uint8_t *payload, uint16_t n);
// returns payload length if a valid frame is found; 0 otherwise
(non-blocking)
uint16_t frame_rx_poll(frame_link_t *L, uint8_t *out, uint16_t max);

// middleware/frame_link.c
#include "frame_link.h"
#include "crc16_ccitt.h"

void frame_init(frame_link_t *L, serial_port_t *sp, ring8_t *rx) { L->sp=sp;
L->rx=rx; }

bool frame_tx(frame_link_t *L, const uint8_t *p, uint16_t n){
    uint8_t hdr[4] = {0xA5,0x5A,(uint8_t)(n&0xFF),(uint8_t)(n>>8)};
    uint16_t crc = crc16_ccitt(&hdr[2], 2, 0xFFFF); crc = crc16_ccitt(p, n,
crc);
    (void)L->sp->v->write(L->sp, hdr, 4);
    (void)L->sp->v->write(L->sp, p, n);
    uint8_t c[2] = {(uint8_t)(crc & 0xFF), (uint8_t)(crc>>8)};
    (void)L->sp->v->write(L->sp, c, 2);
    return true;
}

static int scan_for_sync(const ring8_t *r){
    uint16_t a = rb_avail(r); for(uint16_t i=0;i+1<a;i++){ int
b0=rb_peek(r,i); int b1=rb_peek(r,i+1); if(b0==0xA5 && b1==0x5A) return i; }
    return -1;
```

```
}

uint16_t frame_rx_poll(frame_link_t *L, uint8_t *out, uint16_t max){
    // Ingest any new bytes from serial into the parsing ring
    uint8_t tmp[64]; size_t got = L->sp->v->read(L->sp, tmp, sizeof tmp);
    if(got){ (void)rb_push(L->rx, tmp, (uint16_t)got); }

    int sync = scan_for_sync(L->rx); if(sync < 0) return 0;
    // Drop junk before sync
    if(sync>0){ uint8_t drop[64]; while(sync){ uint16_t d=(sync>sizeof
drop)?sizeof drop:sync; rb_pop(L->rx, drop, d); sync-=d; } }

    if(rb_avail(L->rx) < 4) return 0; // need header+len
    uint8_t header[4]; (void)rb_peek(L->rx,0); // ensure contiguous read
    (void)rb_pop(L->rx, header, 4);
    uint16_t n = (uint16_t)header[2] | ((uint16_t)header[3]<<8);
    if(n > max) { // oversize payload -> resync
        uint8_t dump[2]; rb_pop(L->rx, dump, 2); return 0;
    }
    if(rb_avail(L->rx) < n+2) { // wait for body+CRC
        // Put header back (simple strategy: prepend back)
        // For brevity, not implemented; in practice use a cursor.
        return 0;
    }
    (void)rb_pop(L->rx, out, n);
    uint8_t crcB[2]; (void)rb_pop(L->rx, crcB, 2);
    uint16_t rxCrc = (uint16_t)crcB[0] | ((uint16_t)crcB[1]<<8);
    uint16_t crc = crc16_ccitt(&header[2], 2, 0xFFFF); crc = crc16_ccitt(out,
n, crc);
    if(crc != rxCrc) return 0; // bad frame
    return n;
}
```

---

9) Service #1 — Logging and timebase

Minimal time service using SysTick (or GPT timer if higher resolution is required). Logs go to the debug UART through the same `serial_port_t` API so the app never touches `fsl_debug_console`.

```
// services/timebase.h
#pragma once
#include <stdint.h>
void timebase_init(uint32_t cpuHz); // config SysTick to 1 kHz
uint32_t time_ms(void);

// services/timebase.c
#include "timebase.h"
#include "fsl_common.h"
```

```c
#include "fsl_device_registers.h"

static volatile uint32_t g_ms;
void SysTick_Handler(void){ g_ms++; }
void timebase_init(uint32_t cpuHz){ SysTick->LOAD = (cpuHz/1000u) - 1u;
SysTick->VAL=0; SysTick->CTRL =
SysTick_CTRL_CLKSOURCE_Msk|SysTick_CTRL_TICKINT_Msk|SysTick_CTRL_ENABLE_Msk;
}
uint32_t time_ms(void){ return g_ms; }
```

```c
// services/log.h
#pragma once
#include <stdarg.h>
#include "serial_port.h"
void log_bind(serial_port_t *sp);
void log_printf(const char *fmt, ...);
```

```c
// services/log.c
#include "log.h"
#include <stdio.h>
#include <string.h>
#include "timebase.h"
static serial_port_t *g_log;
void log_bind(serial_port_t *sp){ g_log = sp; }
void log_printf(const char *fmt, ...){ if(!g_log) return; char buf[160];
va_list ap; va_start(ap,fmt); int n=vsnprintf(buf,sizeof buf,fmt,ap);
va_end(ap); if(n<0) return; char line[192]; int m=snprintf(line,sizeof
line,"[%6lu ms] %.*s\r\n", (unsigned long)time_ms(), n, buf); (void)g_log->v-
>write(g_log,(const uint8_t*)line,(size_t)m); }
```

---

10) Service #2 — ARINC 429 word utilities

ARINC 429 encodes aviation parameters in a 32-bit word over a unidirectional, self-clocking bus. We will not implement the physical layer here (that's the ADK-8582), but we *will* encode/decode words:

- **Label**: 8 bits (bit numbers 1–8 in ARINC notation; customarily written in octal).
- **SDI (Source/Destination Identifier)**: 2 bits (bits 9–10).
- **Data**: 19 bits (bits 11–29), interpretation BNR (Binary Number Representation) or BCD (Binary-Coded Decimal).
- **SSM (Sign/Status Matrix)**: 2 bits (bits 30–31).
- **Parity**: 1 bit (bit 32), odd parity over bits 1–31.

```c
// services/arinc429.h
#pragma once
#include <stdint.h>
```

```h
typedef struct { uint8_t label; uint8_t sdi; uint32_t data; uint8_t ssm; }
a429_fields_t;
uint32_t a429_make_word(a429_fields_t f);  // returns 32-bit word (bit 0 =
LSB)
int a429_parse_word(uint32_t word, a429_fields_t *out);
uint32_t a429_bnr_from_float(float value, float lsb, unsigned width
/*<=19*/);
float    a429_bnr_to_float(uint32_t data, float lsb, unsigned width);

// services/arinc429.c
#include "arinc429.h"

static uint8_t odd_parity32(uint32_t v){ // parity over 31 bits (exclude bit
31: parity)
    v ^= v>>16; v ^= v>>8; v ^= v>>4; v ^= v>>2; v ^= v>>1; return
(uint8_t)(~v & 1u);
}

uint32_t a429_make_word(a429_fields_t f){
    uint32_t word = 0;
    word |= ((uint32_t)(f.label & 0xFF)) << 0;       // bits 0..7
    word |= ((uint32_t)(f.sdi   & 0x3))  << 8;       // bits 8..9
    word |= ((uint32_t)(f.data  & 0x7FFFF)) << 10;   // bits 10..28
    word |= ((uint32_t)(f.ssm   & 0x3))  << 29;      // bits 29..30
    // bit 31 is parity (odd over bits 0..30)
    uint32_t p = odd_parity32(word);
    word |= p << 31;
    return word;
}

int a429_parse_word(uint32_t w, a429_fields_t *o){
    uint8_t p = (uint8_t)((w>>31)&1u); uint8_t op = odd_parity32(w &
0x7FFFFFFFu);
    if(p != op) return -1; // parity error
    o->label = (uint8_t)((w>>0)&0xFF);
    o->sdi   = (uint8_t)((w>>8)&0x3);
    o->data  = (uint32_t)((w>>10)&0x7FFFF);
    o->ssm   = (uint8_t)((w>>29)&0x3);
    return 0;
}

uint32_t a429_bnr_from_float(float value, float lsb, unsigned width){
    int32_t q = (int32_t)(value/lsb);
    uint32_t mask = (1u<<width)-1u; return (uint32_t)q & mask;
}
float a429_bnr_to_float(uint32_t data, float lsb, unsigned width){
    // Sign-extend from width bits
    if(width<32){ uint32_t sign=1u<<(width-1u); data = (data ^ sign) - sign;
}
```

```
    return (float)((int32_t)data) * lsb;
}
```

> **Note:** Bit numbering here is least-significant bit = 0 in C. This maps cleanly to ARINC 429 bits 1..32 when you remember that ARINC's bit 1 is the LSB of the label field.

---

11) Service #3 — ARINC 429 over UART to ADK-8582

The ADK-8582 provides physical ARINC 429 TX/RX. We assume a UART command channel to instruct TX and receive RX captures. We encode our *own* framed protocol (Section 8) that carries ARINC words so the solution is portable across external adapters.

Command payloads - `CMD_TX` (0x01): `{ label(1) sdi(1) data(3) ssm(1) repeat(1) gap_ms(2) }` - `CMD_RX_SUB` (0x02): subscribe to words by label: `{ count(1) labels[count] }` - `EV_RX` (0x81): device→host indication: `{ label(1) sdi(1) data(3) ssm(1) }`

```
// services/a429_service.h
#pragma once
#include <stdint.h>
#include <stdbool.h>
#include "frame_link.h"
#include "arinc429.h"

typedef struct { frame_link_t *link; } a429_service_t;
void a429_service_init(a429_service_t *S, frame_link_t *L);
bool a429_tx(a429_service_t *S, a429_fields_t f, uint8_t repeat, uint16_t
gap_ms);
int  a429_poll(a429_service_t *S, a429_fields_t *out); // returns 1 if a word
received
```

```
// services/a429_service.c
#include "a429_service.h"
#include <string.h>

void a429_service_init(a429_service_t *S, frame_link_t *L){ S->link=L; }

bool a429_tx(a429_service_t *S, a429_fields_t f, uint8_t repeat, uint16_t
gap_ms){
    uint8_t p[1+1+1+3+1+1+2]; size_t i=0;
    p[i++]=0x01; // CMD_TX
    p[i++]=f.label; p[i++]=f.sdi; p[i++]=(uint8_t)(f.data&0xFF);
p[i++]=(uint8_t)((f.data>>8)&0xFF); p[i++]=(uint8_t)((f.data>>16)&0x7F);
    p[i++]=f.ssm; p[i++]=repeat; p[i++]=(uint8_t)(gap_ms&0xFF);
p[i++]=(uint8_t)(gap_ms>>8);
    return frame_tx(S->link, p, (uint16_t)i);
}
```

```
int a429_poll(a429_service_t *S, a429_fields_t *out){
    uint8_t b[16]; uint16_t n = frame_rx_poll(S->link, b, sizeof b);
    if(n>=1 && b[0]==0x81 && n>=1+1+1+3+1){
        out->label=b[1]; out->sdi=b[2]; out-
>data=(uint32_t)b[3]|((uint32_t)b[4]<<8)|((uint32_t)(b[5]&0x7F)<<16); out-
>ssm=b[6]; return 1;
    }
    return 0;
}
```

---

12) Application example A — **Generic** framed link: ping/pong

Scenario: During integration, you connect the EVKB to a PC or a second EVKB via UART. You want a deterministic, framed echo test with CRC. This validates the middleware stack without ARINC hardware.

```
// app/main.c (generic demo)
#include "board.h"
#include "clock_config.h"
#include "pin_mux.h"
#include "uart_hw.h"
#include "serial_port.h"
#include "ringbuf.h"
#include "frame_link.h"
#include "timebase.h"
#include "log.h"

#define RING_CAP 256
static uint8_t rx_storage[RING_CAP]; static ring8_t rx;
static uart_hw_t uart_debug_hw; static serial_port_t sp_debug;
static frame_link_t link;

int main(void){
    BOARD_InitBootPins(); BOARD_InitBootClocks();
    // Debug console clock for chosen LPUART instance
    uint32_t uartClk = BOARD_DebugConsoleSrcFreq();

    UART_HW_Init(&uart_debug_hw, LPUART1, uartClk, 115200);
    UART_HW_AttachRing(&uart_debug_hw, rx_storage, RING_CAP);
    SerialPort_fromUart(&sp_debug, &uart_debug_hw);

    rb_init(&rx, rx_storage, RING_CAP); frame_init(&link, &sp_debug, &rx);
    timebase_init(SystemCoreClock); log_bind(&sp_debug);
    log_printf("Booted framed-link ping demo");

    const uint8_t ping[] = { 'P','I','N','G' };
```

```
    uint32_t t0 = time_ms();
    while(1){
        // send a ping every 1000 ms
        if((time_ms()-t0)>=1000){ frame_tx(&link, ping, sizeof ping); t0 =
time_ms(); }
        // if a frame arrives, echo it back
        uint8_t buf[64]; uint16_t n = frame_rx_poll(&link, buf, sizeof buf);
        if(n){ log_printf("RX %u bytes", (unsigned)n); frame_tx(&link, buf,
n); }
    }
}
```

Expected behavior: A PC tool or second board receives a ping frame every second and echoes back. CRC mismatches are dropped silently, demonstrating robustness to noise and framing loss.

---

13) Application example B — **Avionics** ARINC 429 via UART (ADK-8582)

Scenario: A Display Unit (DU) needs **Indicated Airspeed (IAS)** from an Air Data Computer (ADC) on ARINC 429 at 100 kHz. Our EVKB acts as a gateway: it receives IAS words from the ADK-8582 (as UART events) and republishes them to another LRUs' maintenance port while also periodically transmitting a **Baro-Corrected Altitude** word.

Data: - IAS label commonly 0x174 (octal 174, decimal 124). We will use label `0174` (octal) → binary `0x74`. - Altitude (feet) can be encoded as BNR with LSB = 1 ft.

```
// app/main.c (ARINC service demo)
#include "..." // same includes as generic demo + a429_service.h, arinc429.h

static uint8_t a429_rx_store[512]; static ring8_t a429_rx_ring;
static uart_hw_t uart_a429_hw; static serial_port_t sp_a429;
static frame_link_t a429_link; static a429_service_t a429;

int main(void){
    BOARD_InitBootPins(); BOARD_InitBootClocks();
    uint32_t clk = BOARD_DebugConsoleSrcFreq();

    // Debug UART for logs
    UART_HW_Init(&uart_debug_hw, LPUART1, clk, 115200);
UART_HW_AttachRing(&uart_debug_hw, rx_storage, RING_CAP);
SerialPort_fromUart(&sp_debug, &uart_debug_hw); log_bind(&sp_debug);

    // Second UART connected to ADK-8582
    UART_HW_Init(&uart_a429_hw, LPUART3 /*example*/, clk, 115200);
UART_HW_AttachRing(&uart_a429_hw, a429_rx_store, sizeof a429_rx_store);
SerialPort_fromUart(&sp_a429, &uart_a429_hw);
```

```
    rb_init(&a429_rx_ring, a429_rx_store, sizeof a429_rx_store);
frame_init(&a429_link, &sp_a429, &a429_rx_ring); a429_service_init(&a429,
&a429_link);

    timebase_init(SystemCoreClock);
    log_printf("ARINC 429 gateway starting");

    // Periodically transmit baro altitude (example: 12345 ft)
    a429_fields_t tx; tx.label = 0xA1; /* example label for altitude */
tx.sdi=0; tx.ssm=0;
    tx.data = a429_bnr_from_float(12345.0f, 1.0f, 19);

    uint32_t tTx = time_ms();
    while(1){
        if(time_ms()-tTx >= 100){ a429_tx(&a429, tx, /*repeat*/1,
/*gap_ms*/0); tTx = time_ms(); }
        a429_fields_t rx;
        if(a429_poll(&a429, &rx)){
            if(rx.label == 0174 /* IAS */){
                float ias = a429_bnr_to_float(rx.data, /*LSB*/0.5f,
/*width*/19); // example scale
                log_printf("IAS=%.1f kt (SDI=%u, SSM=%u)", ias, rx.sdi,
rx.ssm);
            }
        }
    }
}
```

What this demonstrates: - The **service layer** offers a business-level API (`a429_tx`, `a429_poll`). No driver types or registers leak upward. - Swapping UART instance or baudrate is a BSP concern; the service and middleware remain unchanged.

> **Integration note:** Adapt labels/SSM/BNR scaling to your aircraft ICD (Interface Control Document). The ADK-8582's exact UART protocol can be fit behind the `a429_service.c` payload rules without changing application code.

---

14) Hands-on lab sequence (with solutions)

Lab 1 — Bring-up and serial abstraction

1. Start from the SDK's `lpuart/interrupt` example. Create a new project and copy `drivers/uart_hw.*` and `middleware/serial_port.*`.
2. In `main.c`, initialize LPUART1 with `BOARD_DebugConsoleSrcFreq()` and bind a `serial_port_t`.
3. Verify you can `sp_debug.v->write()` a banner and see it on a terminal at 115200-8-N-1.

Solution highlight: `LPUART_GetDefaultConfig`, `LPUART_Init`, `LPUART_TransferCreateHandle`, `LPUART_TransferStartRingBuffer` are used exactly as in the SDK examples.

Lab 2 — CRC16 and framed link

1. Add `crc16_ccitt.*` and `frame_link.*`. Use the generic demo.
2. With a PC tool, inject intentionally corrupted frames; verify they are discarded (no crash, no echo).

Solution highlight: Frame verification compares computed CRC to the received trailer before accepting payload.

Lab 3 — ARINC 429 utilities

1. Add `arinc429.*`; unit-test `a429_make_word` and `a429_parse_word` on the host (compile with `-DUNIT_TEST` replacing `main`).
2. Verify parity errors are detected. Confirm BNR scaling round-trips for a few values.

Solution highlight: Parity is odd over bits 0..30 in the C representation.

Lab 4 — A429 service over UART

1. Add `a429_service.*` and connect the EVKB UART to the ADK-8582.
2. Subscribe to a label (extend `a429_service.c` with a `CMD_RX_SUB`).
3. Log received words; format the label in octal to match avionics ICDs.

Solution highlight: Only `a429_service.c` knows the UART payload schema; app logic stays invariant.

Lab 5 — Flight-realistic scenario

Scenario: The FCC (Flight Control Computer) requires IAS at 50 Hz and altitude at 10 Hz. Implement two periodic jobs in the main loop driven by `time_ms()`. On RX of IAS, compute *trend* (ΔIAS/Δt) and log an amber caution if trend > 10 kt/s while flaps are extended (simulate flap input via GPIO).

Solution approach: Two timers, a simple FIR filter for the trend, GPIO read via `fsl_gpio`. No dynamic memory.

Lab 6 — Robustness and recovery

- Force UART overflow (momentarily stop servicing RX). Confirm the frame parser resynchronizes.
- Toggle the ADK-8582 power: detect link loss (no frames for 200 ms), raise an SSM = Failure for published parameters.

Solution highlight: Add a watchdog timer and a link-timeout finite-state machine in the service.

15) Best practices (avionics-aimed)

1. **Layer contracts in headers:** Every `*.h` carries a brief contract and constraints (blocking behavior, maximum sizes, timing guarantees). This feeds DO-178C low-level requirements.
2. **Immutable configuration:** Keep baud, pin maps, and label tables in `const` structs placed in a dedicated linker section; version them in the build.
3. **No hidden allocation:** All buffers are static; sizes are ICD-driven and justified. Provide watermarks and asserted capacities.
4. **Time-bounded operations:** Any loop that waits on hardware has a timeout or is driven by interrupts. Document worst-case latencies.
5. **MISRA-C compliance:** Stay within MISRA-C:2012; document deviations (e.g., use of `uint8_t` unions) with rationale.
6. **Unit tests on middleware/services:** Build the same code for host (`-DHOST`) with fakes for `serial_port_t` to exercise framing and A429 packing.
7. **Error taxonomy:** Convert driver `status_t` to project-wide error codes; never leak vendor types across layers.
8. **Traceability:** Map each safety requirement to a function; embed requirement IDs as comments that tools can scrape.
9. **Diagnostics:** Provide a structured health page over the framed link: firmware version, uptime, RX/TX counters, last CRC error timestamp.
10. **Watchdog and brownout:** Enable IWDT or software watchdog; on brownout events, fail safe and log SSM = Failure.
11. **Interrupt hygiene:** Keep ISRs short; defer parsing to the main loop; guard shared state with atomic flags or disable/enable IRQ only around single writes.
12. **Deterministic logging:** Rate-limit logs; avoid printf floating formats in tight ISRs; prefer preformatted integers.

16) Traceability matrix snapshot (example)

Requirement	Layer	Artifact	Verification
R-UART-001: System shall transmit framed packets with CRC	Middleware	`frame_link.c`	Unit test: inject bit errors; observe discard
R-A429-002: System shall detect parity errors	Service	`arinc429.c`	Unit test with flipped parity bit
R-LINK-003: System shall resync within	Middleware	`frame_rx_poll`	HIL test: inject noise via PC tool

Requirement	Layer	Artifact	Verification
200 ms after noise burst			

---

17) What to adapt for your aircraft program

- **Labels & scaling:** Replace example labels (e.g., IAS `0174`) and BNR LSBs per your ICD. Provide a `const` lookup to scale and SSM semantics.
- **Pin muxing:** Select LPUART instance for ADK-8582 in `pins.c` only; keep services middleware-pure.
- **Rates:** If you must guarantee, e.g., 100 Hz TX with 100 µs jitter, move `frame_tx` onto EDMA-driven UART TX and gate by a hardware timer ISR; the service API stays unchanged.

---

18) Checklist before flight-test

- Builds are reproducible (compiler version, linker script, `-fno-common`, `-fstack-protector` as applicable).
- Stack usage measured worst-case (map file + test harness); buffer sizes justified.
- Every ISR execution time measured and logged under worst case.
- Fault injection performed: CRC errors, UART overflow, parity errors, cable disconnects.
- All acronyms expanded at first use in the documentation; ICD references embedded in code comments.

---

19) Appendix — Minimal BSP snippets

```c
// bsp/board_init.c
#include "board.h"
void BOARD_InitHardware(void){ BOARD_InitBootPins(); BOARD_InitBootClocks();
BOARD_InitDebugConsole(); }
```

```c
// bsp/pins.c (sketch)
#include "fsl_iomuxc.h"
void BOARD_InitPins(void){
    // Configure pins for LPUART3 RX/TX as needed for ADK-8582 connection
    // IOMUXC_SetPinMux(...);
    // IOMUXC_SetPinConfig(...);
}
```

Keep all IOMUX and clock specifics in BSP. Higher layers remain untouched if the UART instance or pins change.

---

20) Summary

We now have a layered, cert-friendly firmware template: **drivers → middleware → services → application**, demonstrated with a robust **framed UART** link and a realistic **ARINC 429** gateway scenario using the EVKB-IMXRT1050 SDK APIs. Extend the service layer to add label filtering, SSM rules, and maintenance commands without touching drivers or application logic.