Middleware Standardized Application Programming Interfaces (APIs)

1. Why "standardized APIs" matter in avionics

A **standardized Application Programming Interface (API)** is a contract that describes *what* a service does—independent of *how* a particular vendor or board implements it. In safety-critical avionics we pursue standardization for four reasons:

1) **Portability and supplier flexibility.** When an application speaks a standard API (for example **POSIX—Portable Operating System Interface** threads or **BSD—Berkeley Software Distribution** sockets), you can swap the underlying **RTOS—Real-Time Operating System** or network stack with minimal code churn.

2) **Determinism and test reuse.** Stable APIs let us freeze interface behavior and reuse conformance tests and qualification evidence across programs—vital under **DO-178C—Software Considerations in Airborne Systems and Equipment Certification** and **ARP4754A—Guidelines for Development of Civil Aircraft and Systems**.

3) **Architectural partitioning.** Clear API seams support **IMA—Integrated Modular Avionics** concepts and **ARINC 653—Avionics Application Executive** style partitioning, even when you are not running a full ARINC 653 OS on a microcontroller.

4) **Cybersecurity posture.** Standard crypto and key-store APIs (for example **PKCS#11—Public-Key Cryptography Standards #11**) make it easier to harden designs consistently and to audit them under **DO-326A—Airworthiness Security** and **ED-202A—Airworthiness Security** guidance.

On the EVKB-i.MX RT1050, the NXP SDK already ships with multiple standardized APIs:

- **CMSIS—Cortex Microcontroller Software Interface Standard** *Driver* and *Core* APIs (for UART, SPI, I^{2}C, ENET, etc.). See `boards/evkbimxrt1050/cmsis_driver_examples/`… in the SDK.
- **BSD sockets** via **lwIP—lightweight IP** for TCP/UDP networking. See `boards/evkbimxrt1050/lwip_examples/`….
- **USB class APIs** (CDC-ACM, HID, MSC). See `boards/evkbimxrt1050/usb_examples/`….
- **FatFs** file system API. See `boards/evkbimxrt1050/fatfs_examples/`….

- **mbedTLS—Transport Layer Security** and **PKCS#11** for crypto and key management. See `middleware/mbedtls` and `rtos/freertos/corepkcs11`.

We will start from fundamentals and build up to advanced patterns that are directly applicable to Airbus embedded/avionics use cases.

---

2. The layering model you will use in this training

Think of your software stack as four strata:

```
Application (mission logic, protocols, data models)
        ↑        | portable interface boundary
Standardized API (POSIX/CMSIS/BSD sockets/FatFs/USB class/PKCS#11)
        ↑        | adapter boundary
Adapter (thin shim mapping standard calls to vendor SDK or to a device)
        ↑
Vendor SDK drivers & RTOS (fsl_* drivers, FreeRTOS, lwIP, USB, etc.)
```

The **Application** only knows the standardized API. The **Adapter** is where you translate that API to the exact board implementation (for example mapping `recv()` to lwIP, or `ARM_USART_xxx` to NXP LPUART). This is also the seam where you perform fault containment, logging, and timing measurements needed for **DAL—Design Assurance Level** evidence under DO-178C.

---

3. Survey of standardized APIs in the EVKB-i.MX RT1050 SDK

3.1 CMSIS-Driver (UART as the running example)

CMSIS-Driver unifies peripheral drivers behind ARM-defined interfaces like `Driver_USART`, `Driver_I2C`, and `Driver_SPI`. In the SDK you will find a working example at:

`boards/evkbimxrt1050/cmsis_driver_examples/lpuart/interrupt_transfer/`

The pattern is always:

```c
/* app.h from the SDK example maps to Driver_USART1 */
#define DEMO_USART Driver_USART1

/* Initialize, power on, configure baud rate using the CMSIS API */
DEMO_USART.Initialize(USART_SignalEvent_t);
DEMO_USART.PowerControl(ARM_POWER_FULL);
DEMO_USART.Control(ARM_USART_MODE_ASYNCHRONOUS, BOARD_DEBUG_UART_BAUDRATE);
DEMO_USART.Send(txBuf, len);
DEMO_USART.Receive(rxBuf, len);
```

Your **Adapter** is a tiny file (for example `uart_port_cmsis.c`) exposing an *application-owned* interface:

```
// uart_port.h — your app's UART abstraction (stable across boards)
typedef struct {
    int  (*open)(void);
    int  (*write)(const uint8_t *buf, size_t len, uint32_t timeout_ms);
    int  (*read)(uint8_t *buf, size_t len, uint32_t timeout_ms);
    void (*close)(void);
} uart_port_api_t;
```

and mapping that to CMSIS calls in `uart_port_cmsis.c`. If later you decide to run without CMSIS, you can supply `uart_port_fslrtos.c` that maps to `LPUART_RTOS_*` instead—no higher-layer code changes.

3.2 POSIX/BSD sockets via lwIP

Networking code written to the **BSD sockets** API (`socket()`, `bind()`, `recvfrom()`, `sendto()`) ports between network stacks readily. On the EVKB, sockets are provided by **lwIP**. You can see the idiom in:

`boards/evkbimxrt1050/lwip_examples/lwip_ipv4_ipv6_echo/freertos/socket_task.c`

A minimal UDP telemetry sender looks like this (your Adapter would wrap this):

```
#include "lwip/sockets.h"

int udp_send_telemetry(const char *ip, uint16_t port,
                       const void *payload, size_t len)
{
    int s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0) return -1;
    struct sockaddr_in a = {0};
    a.sin_family = AF_INET;
    a.sin_port   = PP_HTONS(port);
    a.sin_addr.s_addr = ipaddr_addr(ip);
    if (sendto(s, payload, len, 0, (struct sockaddr *)&a, sizeof(a)) < 0) {
        closesocket(s);
        return -2;
    }
    closesocket(s);
    return 0;
}
```

3.3 USB class APIs (CDC-ACM as a virtual COM port)

The NXP USB device stack exposes standardized **USB—Universal Serial Bus** class interfaces (CDC, HID, MSC). The example:

`boards/evkbimxrt1050/usb_examples/usb_device_cdc_vcom/`

presents a `virtual com` device that any operating system can talk to using its class driver. You will reuse this class API unchanged across boards while swapping only the low-level DCD (Device Controller Driver).

3.4 File system API (FatFs)

The **FatFs** API (`f_mount`, `f_open`, `f_write`, `f_read`) is industry-standard for microcontrollers. Try the FreeRTOS version in:

```
boards/evkbimxrt1050/fatfs_examples/fatfs_ramdisk_freertos/
```

FatFs lets you log and retrieve time-series data using the same code whether the medium is SD, eMMC, QSPI flash, or RAM disk.

3.5 Cryptography APIs (mbedTLS and PKCS#11)

For hashing, TLS handshakes, and X.509 parsing, use **mbedTLS** (`middleware/mbedtls` or `middleware/mbedtls3x`). When you need a portable key store abstraction, use **PKCS#11** (see `rtos/freertos/corepkcs11`). Example (SHA-256—Secure Hash Algorithm 256-bit) that compiles against the SDK:

```c
#include "mbedtls/sha256.h"

void sha256_example(const uint8_t *data, size_t len, uint8_t out[32])
{
    mbedtls_sha256_context ctx;
    mbedtls_sha256_init(&ctx);
    mbedtls_sha256_starts(&ctx, 0 /* 0 = SHA-256, 1 = SHA-224 */);
    mbedtls_sha256_update(&ctx, data, len);
    mbedtls_sha256_finish(&ctx, out);
    mbedtls_sha256_free(&ctx);
}
```

---

4. Generic example: "Swap the transport, not the application"

Imagine you are streaming health data from a sensor module. The application offers three services: `telemetry_publish()`, `maintenance_log()` and `command_rx()`. At first flight test, the stream must go over **UART—Universal Asynchronous Receiver/Transmitter**; later, the same packet must travel via **UDP—User Datagram Protocol** to a ground test rack; eventually, an **ARINC 429—Aeronautical Radio, Incorporated 429** channel will be required.

You design a **Comms API** that never leaks implementation details:

```c
// comms.h – stable, testable interface
typedef struct {
    int  (*open)(void);
    int  (*send)(const void *buf, size_t len, uint32_t timeout_ms);
    int  (*recv)(void *buf, size_t len, uint32_t timeout_ms);
    void (*close)(void);
} comms_api_t;

extern const comms_api_t g_uart_backend;  // LPUART via CMSIS or LPUART_RTOS
extern const comms_api_t g_udp_backend;   // lwIP sockets
```

The **Application** binds to const `comms_api_t *active = &g_udp_backend;` and does not change when you move from UART to UDP or to an ARINC 429 UART bridge; only the **Adapter** changes. You can qualify the application once and reuse most of that evidence.

---

5. Avionics use case: ARINC 429 publishing via a UART-attached ARINC converter (ADK-8582)

ARINC 429 messages are 32-bit words with fields: **Label** (8 bits, least significant byte on the wire), **SDI—Source/Destination Identifier** (2 bits), **Data** (19 bits), **SSM—Sign/Status Matrix** (2 bits), and **Parity** (1 bit, odd). In many airframes, microcontrollers don't drive the 10k/100kbit ARINC 429 differential bus directly; instead they feed an external converter over UART or SPI. In this training we use an **ADK-8582** ARINC 429 interface connected by UART to the EVKB-i.MX RT1050.

We standardize an **ARINC 429 Portable Service API** that is independent of the specific converter:

```c
// arinc429_psa.h — portable service API
typedef enum { ARINC429_SPEED_12K5, ARINC429_SPEED_100K } arinc429_speed_t;

typedef struct {
    uint8_t  label;   // 8-bit label (LSB transmitted first on the wire)
    uint8_t  sdi;     // 2-bit SDI (0..3)
    uint32_t data;    // 19-bit data field, aligned to bits 0..18
    uint8_t  ssm;     // 2-bit SSM
} arinc429_word_t;

uint32_t arinc429_pack(const arinc429_word_t *w);
int arinc429_tx_word(uint8_t channel, const arinc429_word_t *w, uint32_t
timeout_ms);
int arinc429_rx_word(uint8_t channel, arinc429_word_t *w, uint32_t
timeout_ms);
int arinc429_set_speed(uint8_t channel, arinc429_speed_t s);
```

And we map this to a concrete **UART** protocol spoken by the converter in `arinc429_adk8582_uart.c`. The UART driver itself uses the SDK's RTOS driver for determinism:

```c
#include "fsl_lpuart_freertos.h"
#include "fsl_lpuart.h"
#include "app.h"    // DEMO_LPUART from SDK freertos_lpuart example

static lpuart_rtos_handle_t s_uart;
static struct _lpuart_handle t_handle;

int arinc_uart_open(void)
{
    lpuart_rtos_config_t cfg = {0};
    cfg.base    = DEMO_LPUART;              // from SDK: LPUART1 on EVKB
    cfg.srcclk = DEMO_LPUART_CLK_FREQ;      // from SDK board clock
    cfg.baudrate = 115200;
    cfg.parity   = kLPUART_ParityDisabled;
    cfg.stopbits = kLPUART_OneStopBit;
    cfg.buffer   = s_rxBackground;          // static RX ring per SDK pattern
    cfg.buffer_size = sizeof(s_rxBackground);
    return (LPUART_RTOS_Init(&s_uart, &t_handle, &cfg) == kStatus_Success) ?
0 : -1;
}

static uint32_t arinc429_add_parity(uint32_t word)
{
    // Odd parity across bits 0..30; parity in bit 31 (MSB)
    uint32_t p = word;
    p ^= p >> 16; p ^= p >> 8; p ^= p >> 4; p ^= p >> 2; p ^= p >> 1;
    uint32_t parity = (~p) & 1U; // odd parity
    return (word & 0x7FFFFFFFU) | (parity << 31);
}

uint32_t arinc429_pack(const arinc429_word_t *w)
{
    uint32_t raw = 0;
    raw |= ((uint32_t)w->label) & 0xFFU;                // bits 0..7
    raw |= ((uint32_t)(w->sdi & 0x3U)) << 8;            // bits 8..9
    raw |= ((uint32_t)(w->data & 0x7FFFFU)) << 10;      // bits 10..28
    raw |= ((uint32_t)(w->ssm & 0x3U)) << 29;           // bits 29..30
    return arinc429_add_parity(raw);                    // bit 31
}

int arinc429_tx_word(uint8_t ch, const arinc429_word_t *w, uint32_t to_ms)
{
    uint8_t frame[8];
    uint32_t word = arinc429_pack(w);
    // Example UART framing: [0xA5, ch, 0x54 /*'T'*/, WORD3, WORD2, WORD1,
WORD0, CRC]
    // If your ADK-8582 uses a different frame, adjust here only.
    frame[0]=0xA5; frame[1]=ch; frame[2]=0x54;
    frame[3]=(uint8_t)(word>>24); frame[4]=(uint8_t)(word>>16);
    frame[5]=(uint8_t)(word>>8);  frame[6]=(uint8_t)word;
    frame[7]=frame[0]^frame[1]^frame[2]^frame[3]^frame[4]^frame[5]^frame[6];
    size_t n = sizeof(frame);
    return (LPUART_RTOS_Send(&s_uart, frame, n) == kStatus_Success) ? 0 : -1;
}
```

The **Application** never sees UART details nor vendor-specific commands—it sees only `arinc429_tx_word()` and friends. If, in the future, the converter changes or moves to SPI, only `arinc429_adk8582_spi.c` changes.

---

6. Airbus-style realistic scenarios

Scenario A — Engine parameter broadcast during ground test. During power-on at a maintenance hangar, a module publishes engine-related parameters over ARINC 429 for a ground test set while simultaneously forwarding the same parameters over UDP to a **GSE—Ground Support Equipment** laptop. The API seam is the Comms API and the ARINC 429 Portable Service API; one application binary satisfies both flows.

Scenario B — Brake temperature trend logging. A **BCU—Brake Control Unit** variant logs wheel-end brake temperatures at 10 Hz to a FatFs volume for trending and pushes out summaries over a USB CDC virtual COM port when maintenance is connected. When the USB cable is absent, the API calls simply fail with a clean status and the application continues.

Scenario C — Avionics security readiness. A communication module uses PKCS#11 to store a TLS client certificate that authenticates to a fleet-maintenance server. The underlying cipher suite implementation (mbedTLS vs. a hardware cryptographic accelerator) can change underneath the same `C_EncryptInit`/`C_Encrypt` API with no change to application code.

---

7. Hands-on exercises on EVKB-i.MX RT1050

Exercise 1 — Run three SDK examples that embody standardized APIs

1) **CMSIS-Driver USART echo.** Import and build `boards/evkbimxrt1050/cmsis_driver_examples/lpuart/interrupt_transfer/`. Observe that your application calls only `Driver_USARTx` and never touches `fsl_lpuart.h` directly.

2) **BSD sockets echo.** Import and build `boards/evkbimxrt1050/lwip_examples/lwip_ipv4_ipv6_echo/freertos/`. Use a PC to `telnet` to the board's IP; note that the code uses `socket()/accept()/recv()/send()` from lwIP's sockets API.

3) **USB CDC VCOM.** Import and build `boards/evkbimxrt1050/usb_examples/usb_device_cdc_vcom/`. The host enumerates a virtual COM port using the USB CDC class driver—your device-side code is stable across NXP microcontrollers.

Capture timing for each using GPIO toggles around the API calls (helpful later for worst-case execution time evidence under DO-178C).

Exercise 2 — Build a portable Comms API with two backends (UART and UDP)

Create `comms.h/.c` and provide two backends:

- `comms_uart_freertos.c` uses the SDK RTOS UART driver (`LPUART_RTOS_*`) and the exact configuration idiom from `freertos_driver_examples/freertos_lpuart/` (look at `app.h` for `DEMO_LPUART`).
- `comms_udp_lwip.c` uses `socket()/sendto()` from lwIP.

Then write a single test task that can switch backends at runtime:

```c
#include "comms.h"

static const comms_api_t *active;

static void tx_task(void *arg)
{
    const char *msg = "HELLO AIRBUS\r\n";
    for (;;) {
        active->send(msg, strlen(msg), 100);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int app_main(void)
{
    // Choose the backend based on a GPIO strap or compile-time flag
#ifdef USE_UART
    active = &g_comms_uart_backend;
#else
    active = &g_comms_udp_backend;
#endif
    xTaskCreate(tx_task, "tx", 1024, NULL, tskIDLE_PRIORITY+1, NULL);
    vTaskStartScheduler();
}
```

Deliverables: one set of application sources, two small adapter files, and a README explaining how to flip transports without touching `tx_task()`.

Exercise 3 — ARINC 429 Portable Service API over ADK-8582 (UART)

Wire the ADK-8582 UART port to `LPUART1` on the EVKB headers. Implement `arinc429_psa.h/.c` and `arinc429_adk8582_uart.c` as shown earlier. Write a task `arinc_publisher_task` that publishes a rotating set of labels at 10 Hz:

```
static void arinc_publisher_task(void *arg)
{
    arinc429_word_t w = { .label = 0xAF, .sdi = 0, .data = 0, .ssm = 0 };
    for (uint32_t n = 0; ; ++n) {
        w.data = n & 0x7FFFFU; // demo counter in 19 bits
        arinc429_tx_word(0 /*ch0*/, &w, 10);
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}
```

Create a companion `arinc_monitor_task` that receives and parses words back into fields, and prints via `fsl_debug_console` for inspection. Fault-inject by flipping SSM or parity to verify your receiver rejects invalid words.

Exercise 4 — Gateway: forward selected ARINC 429 words over UDP

Add a small router task that subscribes to received ARINC 429 words and forwards only configured labels to a maintenance laptop using your UDP backend. Because both sides are driven by standardized APIs (`arinc429_psa` and BSD sockets), the exercise is mostly glue: no vendor driver calls appear in the application.

Exercise 5 — File a maintenance log using FatFs, then serve it over USB CDC

Mount a RAM disk with the SDK FatFs example as a template. Append a CSV line per received word. When a host opens the USB CDC COM port and sends the command `DUMP`, transmit the last 100 lines and keep streaming new entries at 1 Hz. Again the file I/O is purely via the standard FatFs API and the host link is a standard USB CDC class device.

---

8. Advanced guidance for certification-friendly design

- **Keep your API contracts stable and small.** Every function signature becomes part of your compliance surface. Favor simple types and linear execution over callbacks that alter control flow in ways that are hard to analyze for timing and stack use.

- **Make adapters the only place that touches hardware registers or vendor-specific headers.** This isolates change and allows static analysis and **MISRA C—Motor Industry Software Reliability Association C** checking of application code separately from low-level drivers.

- **Deterministic memory policy.** Avoid dynamic allocation in DAL-A/B partitions. FreeRTOS, lwIP, and USB stacks can be configured to use static pools. In the SDK look for `lwipopts.h` and `FreeRTOSConfig.h` to select static allocations and fixed-size pools.
- **Time-bound every call.** All of the APIs we used have timeouts (`LPUART_RTOS_Receive`, non-blocking sockets with `select()`/timeouts, FatFs with retry loops). Propagate timeouts in your own portable APIs and log exceedances.
- **Instrument at the seam.** Add GPIO strobes, counters, and error codes inside adapters. This is where you capture Worst-Case Execution Time (WCET) and fault statistics for DO-178C objectives.
- **Conformance & regression.** Maintain a small suite of **conformance tests** for each standardized API you adopt (for example socket open/bind/error paths, UART parity error handling). Run them on every board variant to protect portability.
- **Security.** Prefer standardized cryptographic APIs (mbedTLS, PKCS#11). Store secrets in a dedicated key object and never expose raw key bytes above the adapter seam.

---

9. Mapping guide: where to look in the SDK you have

- CMSIS-Driver UART examples: `boards/evkbimxrt1050/cmsis_driver_examples/lpuart/…`
- FreeRTOS UART (RTOS API): `boards/evkbimxrt1050/freertos_driver_examples/freertos_lpuart/`
- lwIP sockets examples: `boards/evkbimxrt1050/lwip_examples/lwip_ipv4_ipv6_echo/`
- USB device (CDC/HID/MSC): `boards/evkbimxrt1050/usb_examples/`
- FatFs with FreeRTOS: `boards/evkbimxrt1050/fatfs_examples/fatfs_ramdisk_freertos/`
- Crypto libraries: `middleware/mbedtls`, `middleware/mbedtls3x`, and `rtos/freertos/corepkcs11`

These exact paths exist in your `SDK_25_06_00_EVKB-IMXRT1050.zip` and provide buildable, reference-quality code you can copy into adapters without guessing the right initialization idioms.

---

10. Best practices checklist (to apply while you implement the exercises)

Write these directly into your project's `README` and tick them off during code reviews:

- Every new module exposes a **portable API header**; all vendor or board specifics live only in an `*_port_*.c` adapter.
- Every blocking call has a **timeout** and returns a **defined error code**; the application never loops forever on I/O.
- All **buffers are static** with compile-time sizes; document stack and heap budgets per task.
- **No direct register access** north of the adapter seam; no `fsl_*` includes in application files.
- **Unit tests** for your portable APIs run under a host build (stubbed drivers) and on target (real adapters).
- **Acronyms are expanded** on first use in code comments and documentation (as in this document) to eliminate tribal knowledge.

---

11. What you will submit after completing the module

1) `comms.h/.c` plus `comms_uart_freertos.c` and `comms_udp_lwip.c`.
2) `arinc429_psa.h/.c` plus `arinc429_adk8582_uart.c` (or `spi.c` if that is how your lab hardware is wired).
3) A FreeRTOS demo application with tasks `tx_task`, `arinc_publisher_task`, `arinc_monitor_task`, and the UDP gateway task.
4) A short report with timing, stack usage per task, and an explanation of how you would qualify the adapters for DO-178C DAL-B re-use.

Appendix A — Reference snippets from the SDK you were given

- **FreeRTOS LPUART idiom** (see `freertos_driver_examples/freertos_lpuart/freertos_lpuart.c`): uses `LPUART_RTOS_Init→LPUART_RTOS_SetRxTimeout→LPUART_RTOS_Send/Receive`.
- **lwIP sockets idiom** (see `lwip_examples/lwip_ipv4_ipv6_echo/freertos/socket_task.c`): `socket()→bind()→listen()/accept()` or `sendto()/recvfrom()`.
- **FatFs idiom** (see `fatfs_examples/fatfs_ramdisk_freertos/fatfs_ramdisk_freertos.c`): `f_mount()→f_open()→f_write()/f_read()`.

This appendix is here so you always have a concrete, buildable reference while you implement your adapters.