Module: Middleware Frameworks and Stack Overview

What “middleware” means in embedded avionics

Middleware is the software that sits **between** your application and the raw hardware drivers/RTOS. It is not the peripheral register layer and not the business logic. It is the reusable, testable functionality that gives you higher-level services such as message framing, transport abstraction, file systems, cryptographic services, and protocol stacks. A good mental model is:

Hardware → **BSP** (Board Support Package) → **HAL** (Hardware Abstraction Layer) and **Drivers** → **OSAL** (Operating System Abstraction Layer) / **RTOS** (Real-Time Operating System) → **Middleware** (protocols, services, stacks) → *Application logic*.

In avionics, middleware must be designed to meet deterministic execution budgets and to be testable in isolation. It should avoid hidden dynamic memory, uncontrolled threads, or non-deterministic I/O. It should also support architectural patterns such as **IMA** (Integrated Modular Avionics) partitioning and must integrate with system health monitoring and logging.

Mapping the stack on EVKB-i.MXRT1050

On this specific platform, the layers and their concrete artifacts are visible in the SDK you have. Knowing **where** in the tree each layer lives is invaluable when you read code, file an issue, or prepare evidence for certification audits.

BSP and startup are provided by the MCUXpresso SDK start-up code, clock trees, pin mux, and memory scripts that live under `boards/evkbimxrt1050/*` and `devices/MIMXRT1052/*`. For example, `boards/evkbimxrt1050/…/clock_config.c` and `…/pin_mux.c` define the clocks and pads used by each demo. The chip header and feature descriptions are under `devices/MIMXRT1052/`.

HAL and drivers are the `fsl_*` drivers under `devices/MIMXRT1052/drivers/`. For example, UART is implemented in `fsl_lpuart.c` with DMA and FreeRTOS helpers in `fsl_lpuart_edma.c` and `fsl_lpuart_freertos.c`. These are thin, deterministic wrappers over the hardware, exposing configuration structures and non-blocking APIs.

OSAL (Operating System Abstraction Layer) is provided in `components/osa/`. You will find `fsl_os_abstraction_bm.c` for bare-metal (no RTOS) builds and `fsl_os_abstraction_free_rtos.c` for **FreeRTOS** builds. The OSAL lets the same middleware compile in both worlds with the same API.

RTOS choices in this SDK include **FreeRTOS** under `rtos/freertos/`. Many examples have a `…/freertos/` variant with `FreeRTOSConfig.h` generated for the board.

Middleware libraries appear under `middleware/` and `components/` and are integrated by example apps in `boards/evkbimxrt1050/*`. Notable items you can use as building blocks:

- `middleware/lwip` for TCP/IP; examples live under `boards/evkbimxrt1050/lwip_examples/*`.
- `middleware/mbedtls` for cryptography and TLS.
- `middleware/usb` for device/host/OTG.
- `middleware/fatfs` and LittleFS examples under `boards/evkbimxrt1050/littlefs_examples/*` for file systems.
- `components/serial_manager` and `components/log` for structured I/O and logging.
- `boards/evkbimxrt1050/component_examples/log/*` show a minimal logging stack on both bare-metal and FreeRTOS.

With those pieces, you can assemble a stack appropriate for a particular Line Replaceable Unit (LRU) function: for instance, an ARINC 429 acquisition module that forwards selected labels over Ethernet and stores snapshots to flash.

Two running modes you must master

On i.MX RT1050, the same middleware may be compiled for two execution models.

Bare-metal (no RTOS) uses `fsl_os_abstraction_bm.c` plus interrupt handlers and finite state machines. This is preferred for small, highly deterministic subsystems, initial boot stages, or when certification objectives penalize an RTOS.

FreeRTOS introduces tasks, queues, and timers with `fsl_lpuart_freertos.h`, `fsl_os_abstraction_free_rtos.c`, and specific `FreeRTOSConfig.h`. This is appropriate when you need clear separation between periodic activities (acquisition, processing, storage) and non-critical tasks, as long as you budget and bound every call.

For Airbus-style design, you should be comfortable offering both builds from the same codebase. OSAL and strict layering make that possible.

A generic middleware example: a framed UART transport

Imagine a “sensor hub” that reads binary frames from a co-processor and exposes them to the application as validated messages. The hardware driver (LPUART) gives you bytes. The middleware transforms bytes into messages, handles timeouts, and retries.

Design. At the bottom, you use `fsl_lpuart.c` to configure a UART with DMA or interrupts. In the middle, you implement a ring buffer and a packetizer that searches for a header, length, checksum, and tail. At the top, your application subscribes to messages via a callback or a lock-free queue. None of this middleware knows the board’s pin numbers or clock registers; those are a HAL concern. None of it knows how messages will be interpreted; that is application logic.

Benefit. Your packetizer becomes a reusable component you can drop into another project by only changing the UART instance and callbacks.

An avionics-specific middleware example: an ARINC 429 word broker over UART

In our training hardware, the EVKB-i.MXRT1050 connects to an **ADK-8582** ARINC 429 adapter through UART. The adapter does the line-level TX/RX and exposes a UART command protocol. Your middleware—the "ARINC 429 word broker"—is responsible for three things: (1) assembling ARINC 429 words inside the microcontroller from label, SDI (Source/Destination Identifier), payload, SSM (Sign/Status Matrix) and parity, (2) converting words to the adapter's UART command frames and pushing them to LPUART, and (3) decoding received frames back to ARINC 429 words with integrity checks and time tagging.

Because the physical ARINC 429 layer is off-chip, your safety case focuses on the UART link being bounded and correct, and on the logical formation of ARINC 429 words. The middleware surface area is well defined and easily unit-tested.

How to read the SDK tree for this module

Before we write code, open these locations in your SDK to understand the moving parts you will use:

- `devices/MIMXRT1052/drivers/fsl_lpuart.h` and `fsl_lpuart.c` for low-level serial.
- `devices/MIMXRT1052/drivers/fsl_lpuart_freertos.h` for an RTOS-aware wrapper.
- `components/serial_manager/*` if you prefer a higher-level I/O abstraction and unified logging routing.
- `boards/evkbimxrt1050/driver_examples/lpuart/*` for pin mux and clocks that already work on the EVKB.
- `boards/evkbimxrt1050/component_examples/log/*` for a minimal middleware stack that prints through the debug console.
- `boards/evkbimxrt1050/littlefs_examples/*` for a ready-made file system layer you can reuse for logging.

Keep these open while you work; we will reference concrete files and functions.

Hands-on exercise 1 — walking the stack with a minimal logging service (bare-metal)

Your objective is to bring up a tiny middleware service that hides the debug console behind a logging API. This lets you demonstrate BSP, HAL, and middleware separation in a single program.

Outcome. A `log_info()` function callable from any module that prints timestamped messages through the existing debug console, running on bare-metal with no RTOS.

Starting point. Import the example at `boards/evkbimxrt1050/component_examples/log/bm/`. Build it once unchanged to confirm the toolchain and clocks. Then replace the example `log_main.c` with your own `main.c` that introduces a thin middleware `log.c` / `log.h` pair. The debug console is already configured in `board.c`, `clock_config.c`, and `pin_mux.c` for LPUART1, so you avoid HAL work here.

Key idea. `LOG` is middleware because it hides transport, adds a time source, and can be redeployed on another output by changing only a backend.

Suggested API.

```
// log.h
#ifndef LOG_H
#define LOG_H
#include <stdint.h>
void log_init(void);
void log_info(const char *fmt, ...);
void log_error(const char *fmt, ...);
#endif
```

Implementation sketch. Use `PRINTF` from `fsl_debug_console.h` inside your `log_*` functions, prefixing with a tick counter read from the `SysTick` or GPT. Keep the formatter in middleware, not in the application. Ensure `log_init()` is called after `BOARD_InitBootPins()` and `BOARD_BootClockRUN()`.

Acceptance test. The console prints a boot banner such as `T=000123 ms : system init OK` without the main program knowing about LPUART registers.

Hands-on exercise 2 — a framed UART transport middleware on LPUART3 (bare-metal)

This exercise builds a generic, reusable framed transport over UART on **LPUART3**, leaving LPUART1 free for the debug console. We will reuse the board-proven pin mux from an existing example.

Hardware routing. The EVKB pin mux for LPUART3 is demonstrated in `boards/evkbimxrt1050/driver_examples/lpuart/hardware_flow_control/pin_mux.c`. TX is on `GPIO_AD_B1_06`, RX on `GPIO_AD_B1_07`, with optional CTS/RTS if your adapter supports flow control. Cable these pins to the UART side of your ADK-8582.

Driver configuration. Create a minimal HAL wrapper in `uart_hal.c`:

```
// uart_hal.c — HAL wrapper using SDK drivers
#include "fsl_common.h"
#include "fsl_lpuart.h"
#include "fsl_iomuxc.h"
#include "board.h"

#define ADK_UART              LPUART3
#define ADK_UART_CLK_SRC      kCLOCK_UartMux
#define ADK_UART_CLK_FREQ     CLOCK_GetFreq(kCLOCK_UartMux) // Board clock
tree must feed UartMux

static void UART3_InitPins(void)
{
    CLOCK_EnableClock(kCLOCK_Iomuxc);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0U);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0x10B0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0x10B0U);
}

void adk_uart_init(uint32_t baud)
{
    lpuart_config_t cfg;
    UART3_InitPins();
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baud;
    cfg.enableTx = true;
    cfg.enableRx = true;
    LPUART_Init(ADK_UART, &cfg, ADK_UART_CLK_FREQ);
}

size_t adk_uart_write(const uint8_t *data, size_t len)
{
    LPUART_WriteBlocking(ADK_UART, data, len);
    return len;
}

size_t adk_uart_read(uint8_t *data, size_t len)
{
    return LPUART_ReadBlocking(ADK_UART, data, len);
}
```

This HAL wrapper is intentionally tiny and keeps `fsl_lpuart.*` out of your middleware.

Middleware: framed link. Now create `framed_link.c` that packetizes your UART stream. The frame format is simple and robust for lab work: `[0xA5][LEN][PAYLOAD …][CRC8]`. The middleware owns a ring buffer and exposes `fl_send()` and an event-driven `fl_poll()` that should be called from the main loop or from a periodic interrupt.

```
// framed_link.h
#ifndef FRAMED_LINK_H
#define FRAMED_LINK_H
#include <stddef.h>
#include <stdint.h>
void fl_init(void (*on_frame)(const uint8_t*, size_t));
int  fl_send(const uint8_t *payload, size_t len);
void fl_poll(void); // call periodically to pump RX
#endif
```

Your `fl_poll()` reads bytes via `adk_uart_read()` if available, hunts for the 0xA5 preamble, checks length bounds, computes CRC-8, and, on success, calls the `on_frame` callback with the reassembled payload. Because the UART function pointers live behind `adk_uart_*`, you can swap in DMA or RTOS variants later without touching `framed_link.c`.

Acceptance test. Loop back TX to RX, inject a payload, and verify the callback fires with the original bytes. Log events with the middleware from Exercise 1.

Hands-on exercise 3 — ARINC 429 word broker middleware (bare-metal)

We now build a thin ARINC 429 broker on top of `framed_link`. The broker knows how to compose and parse ARINC 429 words and how to issue adapter commands over UART. It does not know which labels the application cares about; that routing is part of the application.

ARINC 429 word formation. An ARINC 429 word is 32 bits: **label** bits 1–8, **SDI** bits 9–10, **data** bits 11–29, **SSM** bits 30–31, and **parity** bit 32 (odd parity). In code, we will produce the 32-bit value in a controller-friendly layout and then serialize as the adapter requires.

```
// arinc429.h
#ifndef ARINC429_H
#define ARINC429_H
#include <stdint.h>
static inline uint8_t arinc429_odd_parity32(uint32_t w)
{
    w ^= w >> 16; w ^= w >> 8; w ^= w >> 4; w ^= w >> 2; w ^= w >> 1; //
parity of 31 bits
    return (uint8_t)(~w) & 1U; // odd parity bit
}
```

```c
static inline uint32_t arinc429_make_word(uint8_t label, uint8_t sdi,
uint32_t data19, uint8_t ssm)
{
    uint32_t w = 0;
    w |= ((uint32_t)(label) & 0xFFU) << 0;          // bits 1..8
    w |= ((uint32_t)(sdi)   & 0x03U) << 8;          // bits 9..10
    w |= ((uint32_t)(data19)& 0x7FFFFU) << 10;      // bits 11..29
    w |= ((uint32_t)(ssm)   & 0x03U) << 29;         // bits 30..31
    uint32_t p = arinc429_odd_parity32(w);          // bit 32
    w |= p << 31;
    return w;
}
#endif
```

Adapter frames over UART. You will adapt the frame details to your **ADK-8582** documentation. For lab purposes, we assume two commands: `0x54` (ASCII ‘T’) to transmit one 32-bit word, and `0x52` (‘R’) in a response frame that brings received words. We send little-endian words here, but you must confirm endianness with the adapter manual.

```c
// arinc429_broker.c
#include "framed_link.h"
#include "arinc429.h"
#include <string.h>

static void (*s_on_rx_word)(uint32_t w);

static void on_fl_frame(const uint8_t *p, size_t n)
{
    if (n >= 5 && p[0] == 0x52) { // 'R'
        uint32_t w;
        memcpy(&w, &p[1], sizeof(w));
        if (s_on_rx_word) s_on_rx_word(w);
    }
}

void a429_init(void (*on_rx_word)(uint32_t))
{
    s_on_rx_word = on_rx_word;
    fl_init(on_fl_frame);
}

int a429_send(uint32_t word)
{
    uint8_t frame[1+4];
    frame[0] = 0x54; // 'T'
    memcpy(&frame[1], &word, sizeof(word));
    return fl_send(frame, sizeof(frame));
}
```

System integration. In `main.c`, initialize the board, bring up the debug console, initialize the `adk_uart` at the baud rate required by your ADK (start with 115200 8-N-1), then `fl_init()` and `a429_init()`. Compose a known word such as Label `0x10` (Pressure), SDI 0, Data `0x12345`, SSM `0b00`, and send it once per second. In the RX callback, log the timestamp and value. For hardware bring-up without the ARINC link wired, keep TX and RX looped back at the UART pins to validate the framing first.

Acceptance criteria. You can see the transmit frames leave the controller on a logic analyzer, and the broker is able to parse a simulated `R` frame and call your callback.

Hands-on exercise 4 — the same ARINC 429 broker under FreeRTOS

When your application has multiple activities, FreeRTOS helps isolate them with clear priorities and budgets. You will now run the ARINC 429 broker in its own task and communicate with other tasks via queues.

Driver choice. Switch your HAL to the RTOS-aware driver `fsl_lpuart_freertos.h`. Create a UART task that blocks on `UART_RTOS_Receive` and posts frames into a queue. A separate broker task packetizes and routes messages. The logging task handles formatting and I/O in the background.

```
// adk_uart_rtos.c — HAL using RTOS driver
#include "fsl_lpuart_freertos.h"
#include "fsl_lpuart.h"
#include "board.h"

static lpuart_rtos_handle_t s_handle;
static struct _lpuart_handle t_handle;
static uint8_t rx_buf[256];

void adk_uart_rtos_init(uint32_t baud)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baud;
    cfg.enableTx = true; cfg.enableRx = true;
    lpuart_rtos_config_t rtos_cfg = {
        .baudrate = baud,
        .parity = kLPUART_ParityDisabled,
        .stopbits = kLPUART_OneStopBit,
        .buffer = rx_buf,
        .buffer_size = sizeof(rx_buf),
        .srcclk = CLOCK_GetFreq(kCLOCK_UartMux),
        .base = LPUART3,
    };
    LPUART_RTOS_Init(&s_handle, &t_handle, &rtos_cfg);
}
```

```c
int adk_uart_rtos_send(const uint8_t *data, size_t len)
{
    return LPUART_RTOS_Send(&s_handle, (uint8_t*)data, len);
}

int adk_uart_rtos_recv(uint8_t *data, size_t *len, uint32_t timeout_ms)
{
    return LPUART_RTOS_Receive(&s_handle, data, len, timeout_ms);
}
```

Task structure. Create `TaskA429` (priority high) to translate messages to/from ARINC words, `TaskLogger` (low) to print to the console, and—optionally—`TaskStorage` to flush selected labels to flash using LittleFS. Each task has a bounded message queue; all allocations happen during init.

Acceptance criteria. Under induced UART noise (e.g., toggling RX line), the system remains responsive, the broker rejects malformed frames, and no task starves.

Hands-on exercise 5 — persistence middleware with LittleFS (optional)

This exercise shows how a file system layer becomes middleware in your stack. Starting from `boards/evkbimxrt1050/littlefs_examples/littlefs_shell/`, extract the block device and LittleFS initialization code into a `storage` module. The middleware exposes a simple API: `storage_append(label, word)`. Your ARINC broker can call this for a whitelist of labels to create on-board evidence logs. Because file I/O is isolated, you can switch between on-board QSPI flash and an SD card without changing the broker.

Acceptance criteria. After a power cycle, the log remains intact, and you can dump it via the console shell.

Middleware selection and integration strategy for avionics

When selecting a middleware component, you must think beyond features. In DO-178C programs you care about **verifiability**, **configuration control**, **determinism**, and **traceability**. A small stack with fewer states and with unit tests is preferable to a feature-rich black box. For example, for on-board data logging, **LittleFS** is attractive because it tolerates unexpected resets and uses bounded RAM; however, you must freeze its configuration (block size, wear leveling) early and prove timing under worst-case patterns. For network telemetry, **LwIP** can be used on a maintenance port but is not a drop-in replacement for **AFDX** (Avionics Full-Duplex Switched Ethernet) requirements; treat it as ground support functionality, not flight controls.

For cryptography with **MBedTLS**, disable any algorithm you do not need and lock down entropy sources. Any middleware carrying hidden threads or background daemons must be scrutinized; in many cases, a synchronous, polled design is easier to verify.

Best practices that scale to certification

Build your middleware like a library you will have to audit in a year.

Design each layer to have a **single, testable responsibility** and a **narrow API**. The UART HAL only knows pins and bytes. The framed link only knows start/length/checksum. The ARINC broker only knows how to map labels, SDI, data and SSM to 32-bit words with correct odd parity. Each module must be unit-tested with simulated I/O and must run in both bare-metal and FreeRTOS builds.

Keep **memory deterministic**. No dynamic allocation after initialization. Use fixed-size ring buffers and queues sized from a worst-case budget. For file systems, pre-allocate and pre-erase regions during maintenance, not in flight. Avoid recursion; ensure every function has a clear upper bound on CPU cycles.

Treat **time as a first-class requirement**. Write down budgets per layer: UART ISR < 10 µs, framed-link parse < 50 µs per frame, ARINC broker encode < 5 µs per word at 600 kbps peak. Link these to task priorities and to ISR nesting rules. Every non-blocking API must have a timeout and an error code path exercised in tests.

Ensure **traceability**. For every public API, maintain a requirement ID and a test case ID (e.g., REQ-A429-TX-001 → TEST-A429-TX-001). Keep these in your version control along with tool versions. Freeze third-party middleware versions and hash them.

Practice **strong defensive coding**. Validate all inputs at boundaries; never trust a UART length field. Saturate when values exceed range. Verify checksums. Include guards in `fl_send()` to reject overly long frames. In the ARINC broker, re-compute and verify parity on all received words.

Align with **MISRA-C** and **CERT-C** guidelines. Use static analysis and enforce zero compiler warnings at the strictest level supported by your toolchain.

Separate **build configurations**. Provide `bm_debug`, `bm_release`, `frtos_debug`, `frtos_release` configurations. Keep identical APIs across them; the difference must be compile-time only, resolved by OSAL and by a small HAL shim.

Instrument with **health monitoring**. Add a low-frequency “I’m alive” counter per middleware task. Wire a watchdog fed only by successful completion of critical loops, not by idle hooks.

Document **interfaces as contracts**. Every middleware module ships with a short markdown contract describing inputs, outputs, error codes, time budgets, and state diagrams. This becomes audit-ready evidence.

A realistic avionics scenario tying it together

You are integrating a **Cabin Pressure Controller (CPC)** LRU into an aircraft's **ATA 21** (Air Conditioning and Pressurization) domain. The CPC must transmit ARINC 429 labels 210 (Cabin Pressure), 212 (Differential Pressure), and 217 (Rate of Change) at 10 Hz, while receiving label 204 (Baro Set) from the **FGP** (Flight Guidance Panel). The CPC must also serve a maintenance Ethernet port for ground diagnostics and log every arbitration event to flash for post-flight analysis.

On the EVKB, you deploy the following stack. At the bottom, `fsl_lpuart` drives LPUART3 to the ADK-8582. Above that, `framed_link` packetizes frames. The `arinc429_broker` encodes the outflow labels using BNR (Binary Number Representation) scaling and parity and decodes inflow labels, applying plausibility checks and time stamps from a GPT timer. A `storage` middleware built on LittleFS keeps a circular log of the latest 60 minutes. In parallel, a low-priority LwIP task exposes a read-only HTTP status page for maintenance. The application logic sets label 210 from the control algorithm and samples label 204 to update the barometric correction. The entire chain is testable on the bench with the ADK-8582 looped to a second ARINC node; each middleware layer has a unit test harness that runs on the host with UART simulated.

What to measure and verify

For each layer, define quantitative acceptance targets. For UART at 115200 bps, the framed link must sustain > 90% link utilization with zero drops under 1% random byte errors; the broker must reject all malformed frames with bounded CPU use. Under a 10 Hz ARINC stream of three labels, total CPU load on a 600 MHz i.MX RT1050 should stay well below 1%; you will still budget aggressively and measure with the DWT cycle counter. File system writes are deferred to maintenance power states; in flight, only a ring buffer in RAM is used with periodic background flush in non-critical windows.

Common pitfalls and how to avoid them

Do not let middleware silently spawn threads or timers you did not account for. Avoid "helpful" background features that violate determinism. Do not give middleware direct access to the debug console; route it through a logging facade to control formatting and rate. Never hide blocking I/O inside a `printf`-like call on a high-priority task. Resist the temptation to use large general-purpose stacks for tiny needs; write focused, testable modules instead.

Appendix — file paths you will touch most often

This shortlist helps you navigate the SDK:

- UART: `devices/MIMXRT1052/drivers/fsl_lpuart.h`, `devices/MIMXRT1052/drivers/fsl_lpuart.c`, `devices/MIMXRT1052/drivers/fsl_lpuart_freertos.h`.

- Board: boards/evkbimxrt1050/*/clock_config.c, boards/evkbimxrt1050/*/pin_mux.c, boards/evkbimxrt1050/*/board.c.
- Examples for UART3 pins: boards/evkbimxrt1050/driver_examples/lpuart/hardware_flow_control/pin_mux.c.
- OSAL: components/osa/*.
- Logging: boards/evkbimxrt1050/component_examples/log/bm/* and /freertos/*.
- File systems: boards/evkbimxrt1050/littlefs_examples/*.
- Networking: boards/evkbimxrt1050/lwip_examples/* and middleware/lwip/*.
- Crypto: middleware/mbedtls/*.