# ARINC 429 Traffic & Error Analysis

---

## 1) Fundamentals you must know first

### 1.1 What ARINC 429 is and is not

ARINC 429 (often called **Mark 33 Digital Information Transfer System (DITS)**) is a unidirectional, differential, self-clocking bus used in transport-class aircraft. One transmitter (a **source** Line Replaceable Unit, **LRU**) can feed up to 20 receivers (**sinks**). Each physical channel is simplex: there is no arbitration, no address field, and no acknowledgments. Instead, every word contains a **Label** that says *what* the data is; all interested receivers simply listen and use it.

**Data rate.** Two standard bitrates exist and cannot mix on the same wire: **100 kbit/s (High-Speed)** and **12.5 kbit/s (Low-Speed)**.

**Modulation.** Electrical signaling is tri-state bipolar **Return-to-Zero (RZ)** on a twisted, shielded pair (Line A / Line B). Words are separated by a minimum inter-word gap of **≥ 4 bit-times**.

**Error model.** ARINC 429 provides **error detection only** (no forward error correction). A single **odd parity** bit covers the entire 32-bit word.

### 1.2 The 32-bit word layout and bit order

Each ARINC 429 **word** is 32 bits long:

- **Bits 1–8 — Label** (8 bits). Identifies the parameter (typically written in **octal**, e.g., label `260`).
- **Bits 9–10 — SDI (Source/Destination Identifier)** (2 bits). Distinguishes redundant sources or addressed receivers.
- **Bits 11–29 — DATA** (19 bits). Encoded as **BNR (Binary, two's complement)**, **BCD (Binary-Coded Decimal)** or **Discrete** per the label's definition.
- **Bits 30–31 — SSM (Sign/Status Matrix)** (2 bits). Indicates sign or health/validity depending on the data type.
- **Bit 32 — PARITY** (1 bit). Odd parity over **all 32 bits**.

**Transmission order gotcha.** Words are *transmitted* starting with **Bit 1** and ending with **Bit 32**. Within that, the **Label field is sent MSB-first**, while the other fields are **LSB-first**. Many tools therefore show the word diagram "reversed" (Bit 32 at the left). Your software should not assume how a vendor formats hex printouts for the Label — some publish it with bits reversed. We'll make that configurable in code.

### 1.3 SSM (Sign/Status Matrix) quick semantics

SSM meanings depend on the data representation for that label:

- **BNR (two's complement):** Bit 29 of the DATA field is the sign; SSM conveys **status** codes such as **Normal Operation (NO)**, **No Computed Data (NCD)**, **Functional Test (FT)** and **Failure Warning (FW)**.
- **BCD:** SSM often conveys **sign or direction** (e.g., "Plus/North/East/Right/To/Above") *and/or* the same **status** codes (NO/NCD/FT/FW).
- **Discrete:** SSM conveys status only.

You will implement checks that flag **NCD bursts**, **FW**, and **illegal SSM** combinations for selected labels.

---

## 2) What "traffic analysis" means on an Airbus program

A practical analyzer should answer, in real time:

1. **Which labels are present and at what rates?** (Words/second per label and per SDI; bus load estimate.)
2. **Is the traffic healthy?** (Parity error rate, SSM health, SDI consistency by source, duplicates.)
3. **Are timing constraints met?** (No missing labels beyond a timeout; no unexpected rate changes.)
4. **Is the content reasonable?** (Optional domain checks: ranges, monotonicity, deadband — implemented per label.)

The output must be **deterministic**, **non-intrusive** (the analyzer never drives the 429 bus), and **traceable** (logged and time-stamped).

---

## 3) Generic worked example (end-to-end)

**Scenario.** The ADK-8582 is connected to a live 429 High-Speed (100 kbit/s) bus carrying three labels: an angle (BNR), a rate (BNR), and a status bitfield (Discrete). The adapter decodes physical 429 electrical signaling and streams **one line per word over UART** to the EVKB:

`<HEX32> <OPTIONAL_STATUS>\r\n`

Examples: `18F9C3A1 OK`, `18F943A1 PERR`, `3501A001`
(Your exact adapter format may differ. We make the parser pluggable so you can adapt it to the ADK-8582 manual in minutes.)

**What the EVKB does.** For each 32-bit word it: 1) **Checks odd parity** over all 32 bits; 2) **Extracts Label/SDI/DATA/SSM**; 3) **Optionally reverses Label bits** if your upstream prints them reversed; 4) **Updates per-label statistics**; 5) **Prints a 1-second health summary** to the debug console (LPUART1/USB-CDC) and raises alarms on SSM=NCD or FW bursts.

---

## 4) Specific avionics use case (realistic Airbus-style)

**Use case.** Monitor **Air Data and Inertial Reference System (ADIRS)** outputs during aircraft power-up and taxi-out. Requirements:

- **Validity gating.** While on the ground with wheels-speed < threshold, some labels are permitted to be **NCD**. After engine start, all flight-critical labels must transition to **NO (Normal Operation)** within **5 s**.
- **Redundant sources.** Two independent ADIRS sources publish the same label with **SDI=00** and **SDI=01**. The analyzer must: (a) verify that both sources are present at the expected rate (e.g., 20 Hz), (b) flag **SDI flips** (source swapping) and (c) compare SSM across sources.
- **Anomaly detection.** Any **FW (Failure Warning)** on specified labels is latched and reported. A parity error burst > **5 per second** is flagged as **bus integrity issue**. Missing label timeout is **500 ms** beyond nominal period.

**Expected behavior during a taxi-out trial.** On initial power, many labels read **NCD**; within seconds most flip to **NO**. If one ADIRS drops out, your console shows rising **missing** and **NCD** counts for that Label/SDI, while the other source continues clean. If wiring is marginal or shielding is disturbed, you'll observe a spike in **parity errors** concentrated in time.

---

## 5) Hardware on the bench

**Boards.** i.MX RT1050-EVKB (MIMXRT1050-EVKB) + ADK-8582.

**UART topology.** - **EVKB LPUART3** ↔ **ADK-8582 UART** for inbound ARINC stream (3.3 V TTL levels).
EVKB pins (IOMUXC): `GPIO_AD_B1_06` (**LPUART3_TXD**) and `GPIO_AD_B1_07` (**LPUART3_RXD**).
Connect **EVKB RX (AD_B1_07)** to **ADK TX**, **EVKB TX (AD_B1_06)** to **ADK RX** (if needed for commands), and **GND ↔ GND**. - **EVKB LPUART1** to your PC over the on-board OpenSDA (USB) for the debug console prints.

**Bitrate.** Start with **115200 8-N-1** on the EVKB⇄ADK UART unless the ADK-8582 specifies a different rate.

**Safety & EMI.** Keep twisted-pair 429 wiring away from USB cables. Use short jumpers and a common ground. Avoid hot-plugging the 429 side while powered.

---

## 6) Firmware architecture (SDK-based, bare-metal)

We build on the MCUXpresso SDK **LPUART** driver examples for the EVKB:

- **Non-blocking RX with ring buffer:** Based on `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_rb_transfer`.
- **Pin mux:** We enable **LPUART3** on pins **AD_B1_06/AD_B1_07** using IOMUXC calls (pattern from SDK pin_mux files).

**Key modules.** 1. **uart_adk.c** — Initializes LPUART3 and streams RX bytes into a ring buffer; optional TX to send simple commands to the ADK. 2. **adk_frame.c** — Tiny pluggable parser for two common formats: **ASCII-HEX** (8 hex chars per word) and **Binary-5** (4 bytes raw + 1 status). 3. **arinc429.c** — Pure-C utilities: parity check, field extraction, label bit-reverse option, SSM decode helpers. 4. **analyzer.c** — Label/SDI tables, per-second aggregation, timeouts, console reporting through LPUART1.

All code uses only the SDK's public headers (`fsl_lpuart.h`, `fsl_common.h`, `board.h`, `clock_config.h`, `pin_mux.h`).

---

## 7) Implementation — source code (drop-in for your SDK tree)

Below is a compact, single-file variant you can paste into a fresh **driver_examples**-style project to validate the pipeline end-to-end. In production, split into modules as listed above.

> **Build tip.** Start from the SDK example **lpuart/interrupt_rb_transfer** (EVKB), then replace its `main.c` with the code below, and add the extra pin mux calls for **LPUART3**.

```
// File: arinc429_analyzer.c  (bare-metal, MCUXpresso SDK on EVKB-i.MX RT1050)
// Purpose: Receive ARINC 429 words from ADK-8582 over LPUART3, analyze, and
print a 1 Hz summary to debug console.

#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_common.h"
#include "fsl_lpuart.h"
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
```

```
/************* User configuration *************/
#define ADK_UART                LPUART3
#define ADK_UART_IRQn           LPUART3_IRQn
#define ADK_UART_BAUD           (115200U)
#define ADK_UART_CLK_FREQ       CLOCK_GetFreq(kCLOCK_UartClk)

#define DBG_UART                LPUART1   // already set up by
BOARD_InitDebugConsole()
#define DBG_UART_CLK_FREQ       BOARD_DebugConsoleSrcFreq()

#define PARSER_ASCII_HEX        (1)       // 1: expect "XXXXXXXX\r\n" ; 0:
expect Binary-5 (4B raw + 1B status)
#define LABEL_REVERSE_BITS      (1)       // 1: reverse label bits (to handle
MSB-first printouts)

#define RX_RING_SIZE            (512U)
#define MAX_LINE                (64U)

/************* Globals *************/
static lpuart_handle_t s_adkHandle;
static uint8_t         s_rxRing[RX_RING_SIZE];
static volatile bool   s_rxIdle = false;  // set by callback on idle events

static uint8_t         s_lineBuf[MAX_LINE];
static uint32_t        s_lineLen = 0;

// Per-label stats
typedef struct {
    uint32_t ok_words;
    uint32_t parity_err;
    uint32_t ssm_ncd;
    uint32_t ssm_fw;
    uint32_t last_seen_ms;     // for missing timeout
    uint16_t rate_hz;          // simple EMA of rate
} label_stats_t;
static label_stats_t g_label[256][4];  // [label][sdi]

/************* Time base (SysTick @ 1 kHz) *************/
static volatile uint32_t s_ms = 0;
void SysTick_Handler(void) { s_ms++; }
static inline uint32_t now_ms(void){ return s_ms; }

/************* Utilities *************/
static inline uint8_t reverse8(uint8_t x){
    x = (x >> 4) | (x << 4);
    x = ((x & 0xCCu) >> 2) | ((x & 0x33u) << 2);
    x = ((x & 0xAAu) >> 1) | ((x & 0x55u) << 1);
```

```
    return x;
}
static inline bool arinc_parity_ok(uint32_t raw){
    // Odd parity over all 32 bits
    uint32_t ones = __builtin_popcount(raw);
    return (ones & 1u) == 1u;
}
static inline uint8_t arinc_label(uint32_t raw){
    uint8_t lab = (uint8_t)(raw & 0xFFu);
#if LABEL_REVERSE_BITS
    lab = reverse8(lab);
#endif
    return lab;
}
static inline uint8_t arinc_sdi(uint32_t raw){ return (uint8_t)((raw >> 8) &
0x3u); }
static inline uint32_t arinc_data19(uint32_t raw){ return (raw >> 10) &
0x7FFFFu; }
static inline uint8_t arinc_ssm(uint32_t raw){ return (uint8_t)((raw >> 29) &
0x3u); }

static const char* ssm_to_str(uint8_t ssm){
    switch(ssm){
        case 0: return "FW";   // Failure Warning
        case 1: return "NCD";  // No Computed Data
        case 2: return "FT";   // Functional Test
        case 3: return "NO";   // Normal Operation (or sign in BCD)
        default:return "?";
    }
}

/************* ADK UART callback *************/
static void adk_cb(LPUART_Type *base, lpuart_handle_t *handle, status_t
status, void *userData){
    (void)base; (void)handle; (void)userData;
    if (status == kStatus_LPUART_RxIdle){ s_rxIdle = true; }
}

/************* Minimal console print *************/
static void dbg_write(const char *s){ LPUART_WriteBlocking(DBG_UART, (const
uint8_t*)s, (uint32_t)strlen(s)); }
static void dbg_puthex8(uint8_t v){ const char* d="0123456789ABCDEF"; char
b[3]={d[v>>4],d[v&0xF],0}; dbg_write(b);}
static void dbg_putu32(uint32_t v){ char b[12]; int i=0;
if(!v){dbg_write("0");return;} char tmp[12]; while(v){ tmp[i++]= '0'+ (v%10);
v/=10;} while(i--) *b++=tmp[i]; *(--b)=0; /* not used, kept simple */ }

/************* Parser(s) *************/
static bool parse_ascii_hex_line(const uint8_t *buf, uint32_t len, uint32_t
```

```
*outRaw){
    // Expect exactly 8 hex characters at start; ignore trailing
whitespace/tokens.
    if (len < 8) return false;
    uint32_t v=0; int n=0;
    for (uint32_t i=0; i<8; ++i){
        uint8_t c = buf[i];
        uint8_t d = (c>='0'&&c<='9') ? (c-'0') : (c>='A'&&c<='F') ? (10+c-
'A') : (c>='a'&&c<='f') ? (10+c-'a') : 0xFF;
        if (d==0xFF) return false; v = (v<<4)|d; n++;
    }
    *outRaw = v; return true;
}

static bool parse_binary5_stream(uint8_t byte, uint32_t *state, uint32_t
*outRaw){
    // Collect 4 bytes (MSB first) into a 32-bit word; ignore 5th status byte
here.
    static uint8_t buf[5]; static uint8_t idx=0; (void)state;
    buf[idx++] = byte;
    if (idx==5){ uint32_t w =
((uint32_t)buf[0]<<24)|((uint32_t)buf[1]<<16)|((uint32_t)buf[2]<<8)|buf[3];
*outRaw=w; idx=0; return true; }
    return false;
}

/************* EVKB pin mux for LPUART3 RX/TX *************/
static void BOARD_EnableLpuart3Pins(void){
    // Pattern from SDK pin_mux.c files: map AD_B1_06/07 to LPUART3 TX/RX
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0U);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0U);
    // Optional: default pad config is fine for UART; add SetPinConfig if
needed
}

/************* Main *************/
int main(void)
{
    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole(); // Debug console on LPUART1

    BOARD_EnableLpuart3Pins();

    // SysTick 1 kHz for timing
    SysTick_Config(SystemCoreClock / 1000U);

    // Init ADK UART (LPUART3) non-blocking with RX ring buffer
```

```
    lpuart_config_t cfg; LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = ADK_UART_BAUD; cfg.enableTx = true; cfg.enableRx =
true;
    LPUART_Init(ADK_UART, &cfg, ADK_UART_CLK_FREQ);
    LPUART_TransferCreateHandle(ADK_UART, &s_adkHandle, adk_cb, NULL);
    LPUART_TransferStartRingBuffer(ADK_UART, &s_adkHandle, s_rxRing,
sizeof(s_rxRing));

    // Kick off an initial non-blocking receive to trigger idle events
    lpuart_transfer_t rxXfer = { .data = s_rxRing, .dataSize = 0 };
    LPUART_TransferReceiveNonBlocking(ADK_UART, &s_adkHandle, &rxXfer, NULL);

    dbg_write("\r\nARINC429 Analyzer ready. Expecting ");
#if PARSER_ASCII_HEX
    dbg_write("ASCII-HEX lines.\r\n");
#else
    dbg_write("Binary-5 frames.\r\n");
#endif

    uint32_t lastPrint = now_ms();
    uint32_t state = 0; // parser state for Binary-5 if used

    while (1)
    {
        // Drain RX ring into line buffer or binary assembler
        uint8_t ch;
        while (kStatus_Success == LPUART_ReadByte(ADK_UART, &ch))
        {
#if PARSER_ASCII_HEX
            if (ch=='\n' || ch=='\r'){
                if (s_lineLen){
                    uint32_t raw;
                    if (parse_ascii_hex_line(s_lineBuf, s_lineLen, &raw)){
                        // Analyze
                        bool ok = arinc_parity_ok(raw);
                        uint8_t lab = arinc_label(raw);
                        uint8_t sdi = arinc_sdi(raw);
                        uint8_t ssm = arinc_ssm(raw);
                        label_stats_t *st = &g_label[lab][sdi];
                        if (ok) st->ok_words++; else st->parity_err++;
                        if (ssm==1) st->ssm_ncd++; else if (ssm==0) st-
>ssm_fw++;
                        st->last_seen_ms = now_ms();
                    }
                    s_lineLen = 0;
                }
            } else if (s_lineLen < MAX_LINE-1){ s_lineBuf[s_lineLen++] = ch;
}
#else
```

```
            uint32_t raw;
            if (parse_binary5_stream(ch, &state, &raw)){
                bool ok = arinc_parity_ok(raw);
                uint8_t lab = arinc_label(raw);
                uint8_t sdi = arinc_sdi(raw);
                uint8_t ssm = arinc_ssm(raw);
                label_stats_t *st = &g_label[lab][sdi];
                if (ok) st->ok_words++; else st->parity_err++;
                if (ssm==1) st->ssm_ncd++; else if (ssm==0) st->ssm_fw++;
                st->last_seen_ms = now_ms();
            }
#endif
        }

        // 1 Hz summary on debug console
        if ((now_ms() - lastPrint) >= 1000U){
            lastPrint += 1000U;
            dbg_write("\r\nLabel SDI  OK    PERR  NCD   FW\r\n");
            for (uint32_t lab=0; lab<256; ++lab){
                for (uint32_t sdi=0; sdi<4; ++sdi){
                    label_stats_t *st = &g_label[lab][sdi];
                    if (st->ok_words || st->parity_err || st->ssm_ncd || st-
>ssm_fw){
                        dbg_puthex8((uint8_t)lab); dbg_write("   ");
                        dbg_puthex8((uint8_t)sdi); dbg_write("   ");
                        dbg_putu32(st->ok_words); dbg_write("   ");
                        dbg_putu32(st->parity_err); dbg_write("   ");
                        dbg_putu32(st->ssm_ncd); dbg_write("   ");
                        dbg_putu32(st->ssm_fw); dbg_write("\r\n");
                    }
                }
            }
        }
    }
}
```

**Notes.** - The code uses SDK APIs you already have: `LPUART_TransferCreateHandle`, `LPUART_TransferStartRingBuffer`, `LPUART_ReadByte`, and `LPUART_WriteBlocking` for the console. - The helper `LABEL_REVERSE_BITS` accommodates the frequent vendor choice to print the Label byte with bit order as physically transmitted. - The SSM mapping here uses a conservative convention common in test gear: `0=FW`, `1=NCD`, `2=FT`, `3=NO`. If your program needs tighter conformance for **BCD sign** interpretation, extend `ssm_to_str()` per label.

**Pin mux addition.** If your project's generated `pin_mux.c` does not already expose LPUART3, add:

```
// In BOARD_InitBootPins() or a helper function
IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_06_LPUART3_TXD, 0U);
IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_07_LPUART3_RXD, 0U);
```

## 8) How to run the lab (step-by-step)

1. **Create the project.** In MCUXpresso, import **`boards/evkbimxrt1050/driver_examples/lpuart/interrupt_rb_transfer`** from your SDK. Build & run once to verify the UARTs.
2. **Wire the hardware.** Connect ADK-8582 UART TX → EVKB **AD_B1_07/LPUART3_RXD**, ADK RX ← **AD_B1_06/LPUART3_TXD** (optional), and **GND ↔ GND**. Connect the EVKB's USB debug port to your PC for the console.
3. **Replace `main.c`.** Drop in `arinc429_analyzer.c` above. Ensure `clock_config.c` and `pin_mux.c` are present (they come with the example).
4. **Set UART rate.** If the ADK uses a different UART rate, change `ADK_UART_BAUD`.
5. **Choose parser.** If your ADK outputs 8-hex-digit lines, keep `PARSER_ASCII_HEX=1`. If it streams binary words with a trailing status byte, set to 0.
6. **Build & run.** Open a terminal on the **debug** COM port (LPUART1) at 115200 8-N-1. You should see a rolling 1-Hz table as traffic appears.
7. **Sanity test.** If no ARINC source is available, you can temporarily loop the ADK's **TX to RX** and type 8-hex-digit words (with valid odd parity) into the ADK side to exercise the pipeline.

---

## 9) Error conditions you will detect (and how)

1. **Parity errors (PERR).** Recompute odd parity over the 32-bit word and count mismatches. Isolated PERR suggests noise; clustered PERR suggests wiring/shielding or bitrate mismatch upstream.
2. **SSM health.** Count **NCD** and **FW** occurrences per Label/SDI. Bursts or persistent values are actionable.
3. **Missing/late labels.** Track `last_seen_ms` and compare against expected periods (configure per label). Missing data in flight phases escalates severity.
4. **SDI flips / duplication.** If the same label arrives with alternating SDI unexpectedly, suspect source selection logic.
5. **Illegal SSM for data type.** (Advanced) For chosen labels, verify that SSM is a legal combination for BNR/BCD, and that **BCD sign** via SSM agrees with payload sign if that label encodes sign.
6. **Bus load (estimation).** Sum observed word rates × 32 / bitrate to estimate % utilization. (Use the known 100 kbit/s or 12.5 kbit/s.)

---

## 10) Best practices (engineering & certification mindset)

- **Keep the analyzer passive.** Never drive the 429 pair. UART connection is to the adapter only.

- **Time-stamp everything.** Use a monotonic millisecond counter. For certification evidence, store CSV logs to SD or stream to a ground station.
- **Treat SSM seriously.** Don't just decode numbers; gate downstream logic on **NO/NCD/FW/FT**.
- **Understand Label definitions.** SSM meaning and scaling depend on the label. Maintain a controlled label database (CSV or compiled table) with ranges, units, and expected update rates. Review before flight test.
- **Plan for redundancy.** Aggregate by **Label+SDI**. For critical parameters, compare redundant sources and vote or flag disagreement.
- **Watch for phase-of-flight.** Validity expectations change (e.g., NCD may be acceptable on ground but not after takeoff). Implement phase-dependent rules.
- **Noise diagnostics.** If parity errors spike: check shields/grounds, cable length, stubs, and whether HS/LS bitrate is mismatched upstream. Use the ADK (or a second receiver) to cross-check.
- **Determinism.** Avoid dynamic allocation and unbounded loops in interrupt context. Use ring buffers sized from worst-case traffic.
- **Traceability.** Keep source control of label definitions and test reports. Tag logs with software build ID.

---

## 11) Advanced extensions (for the ambitious)

- **Label dictionary & scaling.** Add a const table with label→type (BNR/BCD/Discrete), scale, units, min/max, and expected rate. Compute engineering values and reasonableness.
- **Burst detectors.** Sliding-window detection of PERR or NCD bursts → assert GPIO/LED or send a discrete to a test rig.
- **PC host tool.** Mirror the 1-Hz summaries over the debug UART to a Python script that plots rates and errors for quick triage.
- **Command channel.** If the ADK supports commands, implement simple `GET RATE`, `SET INJECT PERR`, or `FILTER LABEL` messages from the EVKB TX.

---

## 12) Verification checklist (what "done" looks like)

- ☐ With a live 100 kbit/s source, the analyzer prints at least 5 distinct labels with growing **OK** counts.
- ☐ For an injected bad word (toggle one bit), **PERR** increments for that label.
- ☐ Pull the ARINC source: within one second, missing label timeouts are reported in the console.
- ☐ Force an **NCD** condition upstream: **SSM=NCD** increments. Latching log shows the time and label.
- ☐ SDI flip test: alternate two sources; analyzer separates stats `SDI=0` vs `SDI=1`.

## 13) Appendix A — ARINC 429 mechanics (quick reference)

- **Word:** 32 bits: `Parity(32) | SSM(31:30) | Data(29:11) | SDI(10:9) | Label(8:1)`.
- **Parity:** Odd across **all 32 bits**.
- **Bit order on the wire:** Bit 1 is sent first; **Label sent MSB-first**, other fields **LSB-first**.
- **Inter-word gap:** ≥ 4 bit-times.
- **Bit rates:** 12.5 kbit/s or 100 kbit/s (not mixed).

---

## 14) Appendix B — Mapping this to your SDK

**Where this code style comes from in your SDK:** - `boards/evkbimxrt1050/driver_examples/lpuart/interrupt_rb_transfer/*` — non-blocking RX with ring buffer - `boards/evkbimxrt1050/driver_examples/lpuart/hardware_flow_control/pin_mux.c` — example of enabling **LPUART3** pins `AD_B1_06/07` - `devices/MIMXRT1052/drivers/fsl_lpuart.h` — driver APIs used above

**Porting to FreeRTOS (optional):** Replace the polling in the main loop with a task that blocks on a queue fed by the UART ISR callback.

---