

# Goal

Design and implement a practical **protocol analysis and debug environment** for **ARINC 429 (Aeronautical Radio, Incorporated 429)** using equipment you already have:

- **Two ADK-8582** units to act as a *Talker* and *Listener* for end-to-end functional testing.
- **One additional ADK-8582** configured as a *Sniffer* (passive monitor) that converts observed ARINC 429 traffic to **UART** and forwards it to the **NXP i.MX RT1050-EVKB** board for decoding, filtering, time-stamping, logging, and visualization.
- **Hantek 6022BL PC oscilloscope + logic analyzer** for physical-layer bring-up, timing checks, and sanity verification.

The end result is a self-contained, avionics-grade lab environment that lets engineers **set up, configure, and debug ARINC 429 links** without a dedicated commercial protocol analyzer.

---

## 1) ARINC 429 fundamentals

**ARINC 429** is a **simplex, self-clocking, differential** data bus widely used in transport aircraft. Each physical link is one-way (Talker → Listener). A system is formed by multiple unidirectional links. The electrical signaling is **bipolar return-to-zero (BRZ)** over a twisted pair with a third state (return to zero) between symbols. Two standardized bit rates exist: **High-Speed (HS) 100 kbps** and **Low-Speed (LS) 12.5 kbps**.

A single **ARINC 429 word** is **32 bits** and is transmitted most-significant word first with a mandatory **inter-word gap  $\geq 4$  bit times**. The fields are conventional in almost every ICD (Interface Control Document):

- **Bits 1–8: LABEL** (8 bits). By convention shown and referenced in **octal**.
- **Bits 9–10: SDI** (Source/Destination Identifier). Identifies source or intended receiver channel.
- **Bits 11–29: DATA** (19 bits). Format depends on the label definition: **BNR** (Binary Number Representation, 2's complement) or **BCD** (Binary-Coded Decimal), sometimes discrete bits.
- **Bits 30–31: SSM** (Sign/Status Matrix). Encodes sign and validity states (e.g., **Normal Operation (NO)**, **Functional Test (FT)**, **No Computed Data (NCD)**, **Failure Warning (FW)**—the exact mapping is per your ICD).
- **Bit 32: PARITY**. **Odd parity** over bits 1–31 (i.e., the total number of 1s in bits 1–32 is odd).

A healthy HS link can deliver up to **~2,777 words/s** ( $100,000 \text{ bps} \div (32 \text{ data bits} + \geq 4 \text{ gap bits})$ ). LS yields **~347 words/s**.

Why this matters for our analyzer design: - Your UART bandwidth and firmware buffering must sustain worst-case HS traffic, including a safety margin. - Time-stamping resolution must be better than one word time ( $\leq 36 * 10 \mu\text{s} = 360 \mu\text{s}$  at HS). We target **10–50  $\mu\text{s}$**  resolution using the i.MX RT1050 general-purpose timer. - Parity, SSM, and label/SDI awareness are the minimum required to call this a “protocol analyzer.”

---

## 2) Proposed lab architecture (three ADK-8582 + i.MX RT1050 + Hantek 6022BL)

### Physical connections

1. **Talker path:** ADK-8582-T (Talker)  $\rightarrow$  ARINC pair  $\rightarrow$  ADK-8582-L (Listener). Configure both for the same **HS or LS** rate.
2. **Sniffer:** ADK-8582-S (Sniffer) is wired **in parallel** with the same ARINC pair (passive receive). Its **UART TX** exports each observed 32-bit word.
3. **i.MX RT1050-EVKB:** Receives the Sniffer’s UART stream on a free **LPUARTx RX** pin (3.3 V TTL or RS-232 via a **MAX3232** level shifter if the Sniffer’s UART is  $\pm 12 \text{ V}$ ; check the ADK-8582 electrical levels and use a level shifter if needed). The EVKB’s on-board **LPUART1** remains the debug console (115200 bps) while **LPUART3** (or another available instance) is dedicated to the Sniffer input at a higher rate (e.g., **460800 bps**).
4. **Hantek 6022BL:** Use **both analog channels** across the ARINC differential pair to observe BRZ signaling. If the PC software supports math, view **CH1–CH2** to approximate a differential probe. Never clip scope ground to one side of the floating differential pair—reference properly via the hardware’s designed test points. For digital timing checks, the Hantek’s logic analyzer can sample the **UART TX** from the Sniffer or the EVKB’s **LPUART RX** line.

### Data flow

ARINC wire  $\rightarrow$  ADK-8582-S (receives)  $\rightarrow$  **UART framing**  $\rightarrow$  i.MX RT1050 (DMA-backed LPUART)  $\rightarrow$  **decode + time-stamp + filter**  $\rightarrow$  console and optional log file (SD-card)  $\rightarrow$  optional PC viewer.

### Assumption on Sniffer UART framing

Different ADK-8582 firmwares output different UART formats. The EVKB firmware supports **both** common cases without re-flashing the Sniffer: - **Binary mode:** each ARINC word is sent as a fixed 5- or 6-byte record (e.g., speed flag + 4 data bytes + optional checksum). - **ASCII-HEX mode:** one 8-hex-digit word per line, **CR/LF** terminated; optional prefix/speed tag.

A simple run-time setting in the EVKB CLI selects the mode. If neither format matches your ADK-8582, the parser's framing state machine is small and easily adapted.

---

### 3) What “Protocol Analysis & Debug” means in this setup

#### Feature set implemented on the i.MX RT1050

- **Live decode:** Label (shown in octal), SDI, DATA (BNR or BCD rendering), SSM text, parity OK/FAIL.
- **Time-stamping:** microsecond-resolution stamps from a free-running **GPT** (General Purpose Timer) so you can measure inter-word and inter-label timing accurately.
- **Filters:** Include/exclude by Label (octal), by SDI, and by HS/LS. Useful when multiple streams are on the wire.
- **Statistics:** per-label counts, parity errors, SSM state counts, min/avg/max inter-arrival times.
- **Logging:** stream to console; optional **SD-card CSV** (straight reuse of the SDK’s FatFS example) for long captures.
- **Throughput safety: LPUART + EDMA** RX ring buffer sized to absorb HS bursts without overrun.

#### How the Hantek 6022BL fits

- **Electrical sanity:** confirm BRZ levels, symmetry, and 10 µs (HS) or 80 µs (LS) bit periods.
  - **Inter-word gap:** measure that the gap is  $\geq$  4 bit times; this also validates that the Sniffer’s UART isn’t rate-limiting the source.
  - **UART timing:** verify the Sniffer→EVKB UART actually runs at the configured baud (e.g., 460800 bps) and has the expected frame (8-N-1).
- 

### 4) Step-by-step bring-up

#### Step A — Wire & power

- Power all ADK-8582 boards from a clean 5 V or as specified by the vendor.
- Wire the Talker’s ARINC **A/B** to Listener **A/B**. Wire the Sniffer’s ARINC **A/B** *in parallel* (receive-only). Keep twisted pairs short and tidy.
- Connect Sniffer **UART TX** → EVKB **LPUARTx RX** (3.3 V). If the Sniffer outputs  $\pm 12$  V RS-232, insert a **MAX3232** board.
- Connect EVKB to PC via USB for the debug console.

#### Step B — Hantek checks (10 minutes)

- HS: set timebase  $\sim 2 \mu\text{s}/\text{div}$ ; LS:  $\sim 20 \mu\text{s}/\text{div}$ . Trigger on CH1 rising edge.
- Verify BRZ shape: each bit returns to zero mid-bit; measure the **10  $\mu\text{s}$**  (HS) or **80  $\mu\text{s}$**  (LS) bit time.
- Check the **inter-word gap**  $\geq 4$  bit times.

### Step C — EVKB firmware

- Flash the analyzer firmware (provided below) built on the **i.MX RT1050 SDK (EVKB)**. The debug console (115200 bps) offers a CLI to select **binary/ascii** input, set **HS/LS**, and define label filters.
- Start capture; observe decoded output with parity and SSM.

### Step D — Validation loop

- Send a known label from the Talker (e.g., a BNR value increasing by +1 every 50 ms). Verify the EVKB shows an octal label, SDI, incrementing data, SSM=Normal, parity OK, and roughly 50 ms inter-arrival.
  - Flip the Sniffer and Talker between HS and LS; ensure the decoder matches and timing scales accordingly.
- 

## 5) Realistic avionics use case to exercise the analyzer

**Scenario:** Air Data Computer (ADC) publishes Baro-Corrected Altitude and Indicated Airspeed to a Flight Warning Computer (FWC) over HS ARINC 429. Your aircraft ICD specifies:

- ADC transmits **Altitude** (BNR, feet), **I.A.S.** (Indicated Airspeed, BNR, knots), and **Static Air Temperature** (BCD,  $^{\circ}\text{C}$ ) at 10 Hz each on dedicated labels. SDI differentiates left/right sides.
- During an ADC internal fault, **SSM switches to Failure Warning (FW)**; during missing sensor, **SSM=No Computed Data (NCD)**.

### What you test with this bench

1. **Rate and freshness:** Confirm all three labels arrive at  $10 \text{ Hz} \pm \text{tolerance}$ ; your analyzer's per-label timing stats should show this.
2. **SSM reaction:** Use the Talker ADK to inject test/invalid data. Verify that your analyzer flags SSM transitions and that the Listener ADK reacts (e.g., ignores NCD).
3. **Cross-side isolation:** If you transmit the same label on SDI=0 and SDI=1, verify your filter can isolate one side, and that the Listener configured for SDI=0 ignores SDI=1.
4. **Parity error handling:** Intentionally flip a bit in the Talker (if the ADK supports error injection) or temporarily noise the line (gentle capacitive probe) to provoke parity errors; observe that parity FAILs are counted and the underlying word is still logged as raw for post-mortem.

This is the kind of failure-directed testing Airbus teams expect during **HIL** (**Hardware-In-the-Loop**) and **V&V (Verification & Validation)** campaigns.

---

## 6) Throughput math and buffer sizing (so nothing drops at HS)

At HS worst case, ~2,777 words/s. If the Sniffer emits **binary 6-byte records**, the input stream  $\approx 16.7 \text{ kB/s}$ . A 4 KB RX ring buffer gives ~240 ms of burst headroom; **16 KB** is comfortable. The i.MX RT1050 at 600+ MHz can easily service this via **LPUART + EDMA**. If ASCII-HEX (~11–14 bytes/word with CR/LF), expect  $\sim 30\text{--}40 \text{ kB/s}$ ; pick **32–64 KB** total buffering (EDMA + software ring) or increase the Sniffer→EVKB UART baud (e.g., 460800 or 921600 bps).

---

## 7) Firmware design on i.MX RT1050-EVKB (based on NXP SDK)

### Key SDK components reused

- **LPUART driver** (`fsl_lpuart.h`) for the debug console.
- **LPUART + EDMA** (`fsl_lpuart_edma.h`, `fsl_edma.h`, `fsl_dmamux.h`) for high-rate RX from the Sniffer.
- **GPT timer** (`fsl_gpt.h`) for microsecond time-stamps.
- **Board glue**: `board.h`, `clock_config.h`, `pin_mux.h` generated from the SDK's pins & clocks tools.
- **Optional logging: FatFS** SD-card example (EVKB has an SD slot) if you want long captures.

### Runtime blocks

- **`uart_sniffer_rx.c`** — sets up LPUARTx + EDMA into a circular DMA buffer; exposes a non-blocking `sniffer_read()` that returns framed words to the parser.
- **`arinc429_decode.c/.h`** — pure decode: label (octal), SDI, SSM, BNR/BCD helpers, parity check.
- **`label_filter.c/.h`** — include/exclude filters; keeps stats per label/SDI.
- **`cli.c`** — text menu on the debug console (115200 bps) to configure everything at run-time.

### Pin note (important)

EVKB pinouts differ by LPUART instance and header used. In your `pin_mux.c`, assign a **free LPUART** (e.g., LPUART3) to a convenient header (Arduino-style Jxx or J5/J9) using the **MCUXpresso Pins Tool**. Ensure **RX** is 3.3 V tolerant. If in doubt, keep the Sniffer UART at **3.3 V TTL** and avoid RS-232 levels on the EVKB.

---

## 8) Core decode: ARINC word anatomy and helpers

Implementation note on bit positions: in memory we'll treat bit **0** as the **least-significant bit** of the 32-bit value and bit **31** as the most-significant bit.

Under that convention:

- **Label** = bits **0..7**
- **SDI** = bits **8..9**
- **DATA** = bits **10..28** (19 bits)
- **SSM** = bits **29..30**
- **PARITY** = bit **31** (odd)

This aligns with many UART-delivered sniffer formats. If your ADK-8582 delivers a different bit ordering, flip in the parser (a one-line table lookup or bit-reversal).

```
arinc429_decode.h
#ifndef ARINC429_DECODE_H
#define ARINC429_DECODE_H
#include <stdint.h>
#include <stdbool.h>

#ifndef __cplusplus
extern "C" {
#endif

typedef enum {
    ARINC_SSM_NO  = 0,      // Normal Operation (example mapping; confirm with
    ICD)
    ARINC_SSM_FT  = 1,      // Functional Test
    ARINC_SSM_NCD = 2,      // No Computed Data
    ARINC_SSM_FW  = 3,      // Failure Warning
} arinc_ssm_t;

typedef struct {
    uint8_t  label;        // octal when printed
    uint8_t  sdi;          // 0..3
    uint32_t data_raw;     // 19-bit field right-aligned
    arinc_ssm_t ssm;       // 0..3
    bool     parity_ok;    // true if odd parity satisfied
} arinc429_t;

// Odd parity over bits 0..30; bit 31 is parity bit.
bool arinc429_check_parity(uint32_t word);

// Extract fields per the mapping above.
void arinc429_extract(uint32_t word, arinc429_t *out);
```

```

// Convert 19-bit BNR (two's complement) to double with given LSB weight.
// Example: altitude feet label might use 0.25 ft/LSB (confirm in ICD).
double arinc429_bnr_to_double(uint32_t data19, double lsb_weight, bool *neg);

// Convert 19-bit packed BCD to double (invalid nibbles become NAN).
double arinc429_bcd_to_double(uint32_t data19);

const char* arinc429_ssm_str(arinc_ssm_t ssm);

#ifdef __cplusplus
}
#endif

#endif // ARINC429_DECODE_H

arinc429_decode.c
#include "arinc429_decode.h"
#include <math.h>

static inline uint32_t popcount32(uint32_t v) {
    // builtin on many compilers, fallback if needed
    #ifdef __GNUC__
    return __builtin_popcount(v);
    #else
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    return (((v + (v >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
    #endif
}

bool arinc429_check_parity(uint32_t word) {
    // Odd parity over bits 0..31 (inclusive). If popcount is odd => OK.
    return (popcount32(word) & 1u) == 1u;
}

void arinc429_extract(uint32_t word, arinc429_t *out) {
    out->label      = (uint8_t)(word & 0xFFu);
    out->sdi       = (uint8_t)((word >> 8) & 0x3u);
    out->data_raw  = (uint32_t)((word >> 10) & 0x7FFFu); // 19 bits
    out->ssm       = (arinc_ssm_t)((word >> 29) & 0x3u);
    out->parity_ok = arinc429_check_parity(word);
}

double arinc429_bnr_to_double(uint32_t data19, double lsb_weight, bool *neg)
{
    // 19-bit two's complement; bit18 is sign
    int32_t s = (int32_t)(data19 & 0x7FFFFu);

```

```

if (s & 0x40000) { // negative
    s |= ~0xFFFF; // sign-extend to 32 bits
    if (neg) *neg = true;
} else if (neg) {
    *neg = false;
}
return (double)s * lsb_weight;
}

double arinc429_bcd_to_double(uint32_t data19) {
    // Bits [18:0] contain up to 5 nibbles + 3 bits; many labels define
    subsets.
    // We decode as D4 D3 D2 D1 D0 with D0 the Least significant nibble.
    int digits[5];
    for (int i = 0; i < 5; ++i) {
        digits[i] = (data19 >> (i*4)) & 0xF;
        if (digits[i] > 9) return NAN; // invalid BCD
    }
    double value = 0.0;
    double scale = 1.0;
    for (int i = 0; i < 5; ++i) {
        value += digits[i] * scale;
        scale *= 10.0;
    }
    return value; // unit per ICD; caller applies sign/decimal if defined
}

const char* arinc429_ssmToStr(arinc_ssm_t ssm) {
    switch (ssm) {
        case ARINC_SSM_NO: return "Normal";
        case ARINC_SSM_FT: return "FunctionalTest";
        case ARINC_SSM_NCD: return "NoComputedData";
        case ARINC_SSM_FW: return "FailureWarning";
        default: return "Unknown";
    }
}

```

---

## 9) Sniffer UART ingest (EDMA ring) and CLI (SDK-style)

Below is a **single-file example** of the EVKB **main.c** that:

- Initializes **clocks**, **pins**, **debug console** using standard EVKB SDK glue.
- Configures **LPUARTx** (choose instance) for **EDMA RX** into a circular buffer.
- Parses **ASCII-HEX** or fixed **binary 6-byte** frames from the Sniffer.
- Time-stamps each word with **GPT** in microseconds.
- Applies simple **label filters** and prints a decoded line.

Integrate this into an MCUXpresso SDK project (start from lpuart\_edma\_rb\_transfer or lpuart\_interrupt example for EVKB-IMXRT1050 and replace main.c), then add the two small decoder files above.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_common.h"
#include "fsl_iomuxc.h"
#include "fsl_lpuart.h"
#include "fsl_edma.h"
#include "fsl_dmamux.h"
#include "fsl_gpt.h"
#include "arinc429_decode.h"

/***************** USER CONFIG *****************/
#define SNIFFER_LPUART      LPUART3          // choose a free instance
#define SNIFFER_LPUART_IRQn  LPUART3_IRQn
#define SNIFFER_DMA_BASE    DMA0
#define SNIFFER_Dmamux     DMAMUX
#define SNIFFER_LPUART_RX_DMA_CHANNEL 0
#define SNIFFER_LPUART_TX_DMA_CHANNEL 1
#define SNIFFER_LPUART_BAUD 460800U
#define DEBUG_BAUD          115200U
#define RX_DMA_BUF_SZ       (16384U)        // 16 KB DMA ring

static edma_handle_t s_lpuartRxEdmaHandle;
static lpuart_edma_handle_t s_lpuartEdmaHandle;
static edma_handle_t s_lpuartTxEdmaHandle;
static uint8_t s_rxDmaBuf[RX_DMA_BUF_SZ];

// Simple filter: pass all if label_mask == 0; otherwise allow labels in set
static uint8_t g_labelAllow[256];
static bool g_filterEnabled = false;

// Parser mode
static enum { MODE_ASCII = 0, MODE_BINARY = 1 } g_mode = MODE_ASCII;

/***************** Time base: GPT for us ****************/
static inline void timebase_init(void) {
    gpt_config_t cfg; GPT_GetDefaultConfig(&cfg); cfg.clockSource =
kGPT_ClockSource_Pерiph; cfg.enableFreeRun = true;
    GPT_Init(GPT1, &cfg); GPT_SetClockDivider(GPT1, 24); // 24 MHz / 24 = 1
MHz => 1 us ticks
    GPT_StartTimer(GPT1);
```

```

}

static inline uint32_t time_us(void) { return GPT_GetCurrentTimerCount(GPT1);
}

/****************** UART + EDMA init *****/
static void sniffer_uart_init(void) {
    // Enable clocks as configured by pins tool (placeholder; use your
    generated code)
    lpuart_config_t cfg; LPUART_GetDefaultConfig(&cfg); cfg.baudRate_Bps =
SNIFFER_LPUART_BAUD; cfg.enableTx = true; cfg.enableRx = true;
    LPUART_Init(SNIFFER_LPUART, &cfg, CLOCK_GetFreq(kCLOCK_OscClk));

    DMAMUX_Init(SNIFFER_Dmamux);
    DMAMUX_SetSource(SNIFFER_Dmamux, SNIFFER_LPUART_RX_DMA_CHANNEL,
kDmaRequestMuxLPUART3Receive);
    DMAMUX_EnableChannel(SNIFFER_Dmamux, SNIFFER_LPUART_RX_DMA_CHANNEL);
    DMAMUX_SetSource(SNIFFER_Dmamux, SNIFFER_LPUART_TX_DMA_CHANNEL,
kDmaRequestMuxLPUART3Transmit);
    DMAMUX_EnableChannel(SNIFFER_Dmamux, SNIFFER_LPUART_TX_DMA_CHANNEL);

    EDMA_CreateHandle(&s_lpuartRxEdmaHandle, SNIFFER_DMA_BASE,
SNIFFER_LPUART_RX_DMA_CHANNEL);
    EDMA_CreateHandle(&s_lpuartTxEdmaHandle, SNIFFER_DMA_BASE,
SNIFFER_LPUART_TX_DMA_CHANNEL);

    LPUART_TransferCreateHandleEDMA(SNIFFER_LPUART, &s_lpuartEdmaHandle,
NULL, NULL,
                           &s_lpuartRxEdmaHandle,
&s_lpuartTxEdmaHandle);

    lpuart_transfer_t xfer = { .data = s_rxDmaBuf, .dataSize = RX_DMA_BUF_SZ
};
    LPUART_TransferReceiveEDMA(SNIFFER_LPUART, &s_lpuartEdmaHandle, &xfer);
}

/****************** Tiny CLI over debug console *****/
static void print_help(void) {
    printf("\r\nCommands:\n");
    printf(" mode ascii|bin           - select sniffer UART framing\n");
    printf(" allow <oct_label>        - allow only this octal label (repeat to
add)\n");
    printf(" allow *                  - disable filter (allow all)\n");
    printf(" clear                   - clear all label filters\n");
    printf(" stat                    - show per-label counts\n");
}

static void filter_clear(void) { memset(g_labelAllow, 0, sizeof
g_labelAllow); g_filterEnabled=false; }
static void filter_add(uint8_t label) { g_labelAllow[label]=1;
}

```

```

g_filterEnabled=true; }
static bool filter_pass(uint8_t label) { return !g_filterEnabled ||
g_labelAllow[label]; }

static uint32_t g_labelCounts[256];

/********** ASCII and binary parsers *****/
static bool parse_ascii_line(const uint8_t *buf, size_t len, uint32_t
*word_out) {
    // Expect 8 hex digits in the line; ignore Leading/trailing spaces
    char tmp[16]; size_t j=0;
    for (size_t i=0;i<len && j<8;i++) {
        char c = (char)buf[i];
        if ((c>='0'&&c<='9')||(c>='A'&&c<='F')||(c>='a'&&c<='f')) tmp[j++]=c;
    }
    if (j!=8) return false; tmp[8]='\0';
    *word_out = (uint32_t)strtoul(tmp,NULL,16);
    return true;
}

static bool parse_binary_record(const uint8_t *buf, size_t len, uint32_t
*word_out) {
    // Expect 6 bytes: [0xA5][spd][w3][w2][w1][w0] (example). We only use
    // the 4 bytes for now.
    if (len < 6) return false;
    if (buf[0] != 0xA5) return false; // sync
    *word_out =
((uint32_t)buf[2]<<24)|((uint32_t)buf[3]<<16)|((uint32_t)buf[4]<<8)|((uint32_
t)buf[5]);
    return true;
}

/********** DMA ring consumer *****/
static size_t s_consumeIdx = 0;
static inline size_t dma_produce_idx(void) {
    // For LPUART EDMA circular mode, the TCD current address maps to
    // producer index.
    // Simplify by reading CITER/CSR or poll SADDR; SDK abstracts this
    // differently per version.
    // If your SDK lacks a direct API, convert to an interrupt-driven
    // half/full buffer callback.
    return (RX_DMA_BUF_SZ - EDMA_GetRemainingBytes(SNIFFER_DMA_BASE,
SNIFFER_LPUART_RX_DMA_CHANNEL)) % RX_DMA_BUF_SZ;
}

static int get_line_from_dma(uint8_t *line, size_t maxlen) {
    // Gather until CR/LF for ASCII; for binary, gather fixed 6 bytes.
    if (g_mode == MODE_BINARY) {
        size_t need = 6; size_t got = 0;

```

```

        while (got < need) {
            size_t prod = dma_produce_idx();
            if (s_consumeIdx == prod) return 0; // no new data
            line[got++] = s_rxDmaBuf[s_consumeIdx++]; if
(s_consumeIdx==RX_DMA_BUF_SZ) s_consumeIdx=0;
        }
        return (int)got;
    } else {
        size_t got = 0; bool seenCR=false;
        while (got < maxLen-1) {
            size_t prod = dma_produce_idx();
            if (s_consumeIdx == prod) break; // no new data yet
            uint8_t c = s_rxDmaBuf[s_consumeIdx++]; if
(s_consumeIdx==RX_DMA_BUF_SZ) s_consumeIdx=0;
            if (c=='\n') { line[got]='\0'; return (int)got; }
            if (c=='\r') { seenCR=true; continue; }
            line[got++] = c;
        }
        return 0; // incomplete
    }
}

***** Octal print helper *****
static void print_label_octal(uint8_t label) {
    // Print as 3-digit octal with leading zeros, the conventional ARINC
format
    printf("%03o", label);
}

***** MAIN *****
int main(void) {
    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    printf("\r\nARINC 429 Analyzer (EVKB-IMXRT1050)\n");
    printf("Debug console ready @ %u bps\n", (unsigned)DEBUG_BAUD);

    timebase_init();
    sniffer_uart_init();

    filter_clear();
    print_help();

    uint8_t line[128];
    while (1) {
        int n = get_line_from_dma(line, sizeof line);
        if (n > 0) {

```

```

        uint32_t word = 0; bool ok=false;
        if (g_mode == MODE_BINARY) ok = parse_binary_record(line,
(size_t)n, &word);
        else ok = parse_ascii_line(line, (size_t)n, &word);
        if (!ok) continue;

        uint32_t t = time_us();
        arinc429_t d; arinc429_extract(word, &d);
        if (!filter_pass(d.label)) continue;
        g_labelCounts[d.label]++;

        // Example: try BNR decode with 0.25 unit/LSB; if NAN, also try
BCD.
        bool neg=false; double bnr = arinc429_bnr_to_double(d.data_raw,
0.25, &neg);
        double bcd = arinc429_bcd_to_double(d.data_raw);

        printf("T=%8u us  HSLS=?  LBL=", (unsigned)t);
        print_label_octal(d.label);
        printf(" SDI=%u SSM=%s PAR=%s DATAraw=0x%05lX  BNR=%g
BCD=%g\r\n",
d.sdi, arinc429_ssm_str(d.ssm), d.parity_ok?"OK":"FAIL",
(unsigned long)d.data_raw, bnr, bcd);
    }

    // Non-blocking read of CLI from debug console (left as exercise; use
SDK getchar/puts)
    int c = GETCHAR();
    if (c > 0) {
        if (c=='h') print_help();
        // parse minimal commands from stdin (omitted for brevity)
    }
}
}

```

Notes: 1) Replace the **DMAMUX source constants** with the ones matching your chosen LPUART instance (the example uses LPUART3). MCUXpresso's **Peripherals** tool will emit the correct enums. 2) If your SDK version doesn't expose EDMA\_GetRemainingBytes(), switch to the SDK's **EDMA ring-buffer pattern** (half/full callbacks). Start from the example lpuart\_edma\_rb\_transfer and move the parsing logic into the callback. 3) The printf uses the Debug Console retargeting included by BOARD\_InitDebugConsole().

## 10) Optional: CSV logging to SD-card

For long captures, copy the **EVKB sdcard\_fatfs example** into your project and, in the word-processing loop, write:

T(us),Rate,Label(oct),SDI,SSM,Parity,Data(HEX),BNR,BCD\n

Make each row one decoded word. This creates files the certification and V&V teams can ingest directly.

---

## 11) Using the Hantek 6022BL effectively

Even though it isn't a high-end avionics scope, the 6022BL is sufficient for timing and level checks:

- **Bit timing:** place cursors on consecutive transitions; HS should read ~10 µs, LS ~80 µs.
  - **BRZ recognition:** each bit has a pulse in the first half of the bit cell returning to zero in the second half.
  - **Differential view:** if the software supports math, display CH1–CH2; otherwise, observe each side separately for symmetry.
  - **UART check:** point the logic analyzer at Sniffer UART TX → verify baud, start/stop bits, and no framing errors.
- 

## 12) Best practices (avionics-oriented)

- **Treat the ICD as law:** your analyzer decodes generically, but **label meaning, scaling (BNR LSB), and SSM mapping** must come from the aircraft program's ICD. Bake an ICD-specific **label dictionary CSV** you can load at run-time to render engineering units correctly.
  - **Always log raw + decoded:** store the 32-bit word and your interpretation side-by-side. During audits you must demonstrate that decoded values are traceable to raw.
  - **Parity is advisory, not a gate:** many analyzers count and flag parity but still record the word. Follow that practice here.
  - **Timebase calibration:** periodically cross-check the GPT tick against a known PPS or a laboratory timer to ensure µs accuracy.
  - **Non-intrusive sniffing:** the Sniffer must be **receive-only** with high input impedance so you never load the ARINC pair. Do not back-drive the bus.
  - **Isolate lab supplies:** noisy or floating grounds can masquerade as parity errors. Keep supplies clean and grounds short.
  - **Rate margin:** if you use ASCII-HEX framing from the Sniffer, raise the UART baud to ≥460800 bps at HS to prevent overruns.
  - **Version-control your analyzer:** commit firmware + label dictionaries + ICD references; tag builds used in test reports.
-

## 13) Troubleshooting guide

- **Nothing decodes:** check Sniffer UART voltage levels (3.3 V vs  $\pm 12$  V). Wrong levels require a MAX3232. Verify LPUART instance/pins in `pin_mux.c`.
  - **Garbage characters** on the console: console baud mismatch. Keep debug console at 115200 bps, Sniffer link independent at 460800 bps.
  - **Parity FAIL for all words:** ADK-8582 bit ordering differs from the assumed mapping. Add a **bit-reverse** step before `arinc429_extract()` and re-test.
  - **Overruns** at HS: increase RX DMA buffer to 32–64 KB, avoid heavy `printf`, switch to binary mode, or raise Sniffer→EVKB baud.
  - **No HS/LS distinction:** either parse a speed flag from the Sniffer framing or provide two serial profiles and let the user select.
- 

## 14) Acceptance criteria (what “done” looks like)

1. With HS selected and the Talker emitting three known labels at 10 Hz, the EVKB console shows:
    - Correct **octal labels** and **SDI, parity OK, SSM=Normal**.
    - **Inter-arrival times** clustered around 100 ms with  $<\pm 5$  ms jitter.
  2. When SSM is forced to **NCD/FW** at the Talker, the analyzer shows the change within one word and increments per-label SSM counters.
  3. With the Hantek on the ARINC pair, measured **bit time** is 10  $\mu$ s and **inter-word gap**  $\geq 4$  bit times.
  4. No RX overruns for a 10-minute HS run with ASCII-HEX; no drops for a 30-minute HS run with binary framing.
- 

## 15) What to adapt for your specific ADK-8582

Because “ADK-8582” implementations vary, confirm the following and tweak one small function if needed:

- **UART frame shape** (binary vs ASCII). If binary, adjust `parse_binary_record()` to your exact header and byte order. If ASCII, confirm the width (8 hex chars) and terminator (CR/LF).
- **Bit ordering** in the 32-bit word. If the ADK reports bits with label in the top byte instead of the low byte, add `word = ((word & 0xFF)<<24) | ((word & 0xFF00)<<8) | ((word & 0xFF0000)>>8) | ((word>>24)&0xFF);` or similar.
- **Speed flag:** If the Sniffer embeds HS/LS in a header byte, parse and print it.

None of these change the analyzer core; they are tiny adapters.

---

## 16) (Optional) PC-side viewer

If you prefer a richer UI, stream the EVKB's decoded CSV to a PC and visualize with a small Python script or a simple Qt app. The EVKB remains the authoritative decoder and time-stamper; the PC only filters and plots. This separation keeps lab wiring simple and avoids USB driver work on the EVKB.

---

## 17) Summary

With two ADK-8582 units forming a Talker/Listener link, a third ADK-8582 as a passive Sniffer, an **i.MX RT1050-EVKB** running the firmware above, and the **Hantek 6022BL** to verify the physical layer, you have a complete **protocol analysis and debug environment for ARINC 429**. It handles **setup, configuration, live decode, timing, filtering, and logging**, while staying faithful to avionics practices (ICD-driven meaning, SSM awareness, parity audit, and raw/decoded traceability). This is sufficient for day-to-day integration tasks and supports realistic avionics use cases without a commercial analyzer.