ARINC 429 Protocol Fundamentals: Word Structure and Timing Requirements

---

0) Big picture: what ARINC 429 is and why it matters

ARINC 429, formally *Mark 33 Digital Information Transfer System (DITS)*, is a **simplex**, **self-clocking** avionics bus used extensively in transport aircraft. A single **transmitter** fans out over a shielded, twisted pair to multiple **receivers** (point-to-multipoint). There are two standardized bit rates: **12.5 kbps** ("low-speed") and **100 kbps** ("high-speed"). Signaling is **Bipolar Return-to-Zero (BPRZ)**, so the line carries both data and timing; no separate clock wire exists.

Why focus on **word structure** and **timing**?

- If you place the wrong bits into the wrong fields, downstream **Line-Replaceable Units (LRUs)** will either drop your data or misinterpret it.
- If you violate bit timing, rise/fall limits, or the mandatory **NULL gap** between words, receivers cannot reliably frame or decode your traffic.

---

1) The 32-bit ARINC word — fields, numbering, and transmission order

Every ARINC message is a fixed-length **32-bit word** subdivided into five fields by **bit number** (Bit 1 … Bit 32). The canonical layout is:

```
Bit#:   1        8 9  10       11                               29 30 31 32
        | Label  |SDI|            DATA (19 bits)                  | SSM | P |
        +--------+---+--------------------------------------------+-----+---+
```

Fields: - **Label (Bits 1–8):** An 8-bit identifier, traditionally written in **octal** (e.g., label 203_8). The label tells any LRU *what* the word is (e.g., baro altitude), independent of who sent it. - **SDI — Source/Destination Identifier (Bits 9–10):** Often used to differentiate sources (e.g., Air Data Computer 1 vs 2) or to select a destination on legacy receivers. When unused, it may be repurposed as data per the program's Interface Control Document (ICD). - **DATA (Bits 11–29):** A 19-bit payload. Common encodings are **BNR — Binary Number Representation** (two's-complement fixed-point), **BCD — Binary-Coded Decimal** (packed digits), or **discrete** flags. - **SSM — Sign/Status Matrix (Bits 30–31):** Conveys sign and/or validity, typically including codes like *Normal Operation (NO)*, *Functional Test (FT)*, *Failure* or *No Computed Data (NCD)*. Meaning depends on the data type and label family. - **Parity (Bit 32): Odd parity** over Bits 1–31. Set Bit 32 so the total number of '1' bits in the entire 32-bit word is **odd**.

1.1 Bit numbering vs. bit *transmission* order — the classic gotcha

ARINC's bit numbering (1..32) is a **logical** view. On the wire, **the Label field (Bits 1–8) is transmitted first and inside that byte the most-significant bit (MSB) is sent first**. The remainder of the word (Bits 9–32) is then transmitted **least-significant-bit first**. Many host libraries or analyzers hide this by flipping the label byte for you; others expect you to do it. Always check your adapter's API.

Practical rule: In your software, build words in the *bit-numbered* layout above. Before handing a word to a device that expects an on-the-wire bitstream, ensure the **Label byte is bit-reversed** if required by that device.

1.2 Computing odd parity (Bit 32)

1) Assemble Bits 1–31 (Label, SDI, DATA, SSM).
2) Count the number of '1' bits.
3) If the count is **even**, set Bit 32 to 1. If it's **odd**, set Bit 32 to 0. After this, the total count of '1's across 32 bits is odd.

Worked parity example: Suppose Bits 1–31 currently contain 17 one-bits (odd). Then Bit 32 must be 0 (leave odd alone). If Bits 1–31 contain 18 one-bits (even), set Bit 32 to 1 to make 19 (odd).

1.3 Encodings inside DATA (Bits 11–29)

- **BNR (Binary Number Representation):** two's-complement fixed-point. Choose **LSB** (least significant bit) scale to fit the range (e.g., LSB = 1/128 °/s). Negative values are represented by two's-complement. Sign is conveyed by the numeric value; status lives in SSM.
- **BCD (Binary-Coded Decimal):** decimal digits packed into nibbles; often used for times, headings, altitudes, or settings where decimal readability is important. Sign may live in SSM or an agreed bit.
- **Discrete:** individual bits are on/off flags (e.g., gear down, spoilers deployed). SSM still carries validity/state.

SDI and SSM policy are ICD-defined. Decide and document, for example, "SDI 00 = ADC #1, 01 = ADC #2" and "SSM 00 = Normal Operation, 01 = Functional Test, 10 = Failure, 11 = No Computed Data," or whatever your program specifies.

---

2) Timing requirements — bit times, waveform, and gaps

ARINC 429 transmitters must meet three major timing domains:

1) **Bit rate accuracy and stability.**
    - High-speed: **100 kbps** (bit time **10 µs**).

    - Low-speed: **12.5 kbps** nominal (bit time **80 µs**). Bit-time and half-bit windows include tight tolerances for accuracy and jitter.

2) **Waveform shape — Bipolar Return-to-Zero (BPRZ).**
    - Logical ‘1’: +10 V differential for the **first half** of the bit time, then return to 0 V (NULL) for the second half.
    - Logical ‘0’: −10 V differential for the **first half** of the bit time, then NULL.
    - **Rise/fall times** (10–90 %) are bounded so receivers can reconstruct timing without excessive ISI (inter-symbol interference). High-speed rise/fall is microseconds; low-speed is an order of magnitude slower.
3) **Inter-word gap (NULL spacing).**
    - Between two words there must be **≥ 4 bit-times** of continuous NULL (0 V differential).
    - Therefore, the absolute minimum “time budget” per word is **36 bit-times** (32 bits + 4 bits gap). At 100 kbps that’s **360 µs** per word; at 12.5 kbps it’s **2.88 ms** per word.

Throughput rule of thumb: Max words/s ≈ bit_rate / 36. At 100 kbps ⇒ ≈ 2777 words/s. Real systems add margin for bus loading, flight phase transients, and adapter FIFOs. Keeping steady-state channel load **≤ 50 %** is common practice.

Scheduling implication: Because the bus is self-clocking, receivers rely on clean NULL gaps to frame words. Bursting at the absolute minimum gap repeatedly can stress older LRUs. Prefer **6–8 bit-time gaps** when possible; your adapter’s hardware will enforce the on-line gap once you set it.

---

3) Worked examples (clear and realistic)

3.1 Generic example — BNR packing

We must transmit body-X angular rate **+3.25 °/s** as BNR with **LSB = 1/128 °/s**.

1) Scaled integer = 3.25 × 128 = **416** (0x01A0).
2) Place into DATA (Bits 11–29). SDI = 0. SSM = Normal Operation.
3) Compute **odd parity** over Bits 1–31; set Bit 32 accordingly.
4) If the adapter expects the on-the-wire bitstream, **bit-reverse the Label byte** before transmit.

3.2 Real avionics scenario — ADIRU to PFD/FMS stream

An **ADIRU — Air Data/Inertial Reference Unit** streams on a **100 kbps** channel to both the **PFD — Primary Flight Display** and **FMS — Flight Management System**:

- Airspeed (label A) at **50 Hz**

- Baro altitude (label B) at **50 Hz**
- Body rates p, q, r (labels C, D, E) at **100 Hz** each

Words/s = 2×50 + 3×100 = **400 words/s**. Capacity headroom at 100 kbps (max ≈ 2777 words/s) is ample. Target an average gap of **≥ 6 bit-times** to be courteous to legacy receivers. Document SDI mapping (e.g., SDI 00 = ADIRU #1, 01 = ADIRU #2) and SSM policy in the ICD.

3.3 Real avionics scenario — Flight Controls timing budget

A **Flight Control Computer (FCC)** multicast channel carries:

- Elevator servo command (label F) at **200 Hz** (control loop)
- Surface position feedback (label G) at **200 Hz**
- Monitoring discretes (label H) at **25 Hz**
- Mode/status (label J) at **10 Hz**

Total words/s = 200 + 200 + 25 + 10 = **435 words/s**. At a conservative **8 bit-time** average gap, each word consumes 32 + 8 = 40 bit-times ⇒ 100 kbps / 40 = **2500 words/s** capacity. Bus load ≈ 435 / 2500 = **17.4 %** — healthy. Ensure rising-edge slew and stub lengths meet installation limits; test worst-case cold-soak (slow edges) and hot-soak (timing drift).

---

4) Hands-on labs — i.MX RT1050-EVKB driving Holt ADK-8582 over UART

Hardware - **EVKB-IMXRT1050** board (MIMXRT1052 MCU — Microcontroller Unit). - **ADK-8582** evaluation kit for the HI-8582/HI-8583 ARINC-429 interface ICs. The kit's on-board MCU presents a **UART — Universal Asynchronous Receiver/Transmitter** control/telemetry console; the ARINC line driver/receiver on the kit enforces BPRZ waveform and line levels on the twisted pair.

Connectivity - Choose a free EVKB **LPUART — Low-Power UART** instance (e.g., LPUART3). Route its TX/RX to a header using the Pins tool. Keep LPUART1 for the debug console. - Wire EVKB TX → ADK RX, EVKB RX → ADK TX, and GND ↔ GND. Level is 3.3 V UART logic. - The ADK's ARINC output goes to your analyzer or loopback as desired.

Software prerequisites - MCUXpresso SDK for **EVKB-IMXRT1050** (provided). Driver headers you'll use: `fsl_lpuart.h`, `fsl_pit.h`, plus board init (`board.h`, `clock_config.h`, `pin_mux.h`). - Start from an SDK **lpuart** driver example (`boards/evkbimxrt1050/driver_examples/lpuart/*`) and convert it to an application task.

Lab 1 — Build an ARINC word packer (BNR) with odd parity

Goal: Create a reusable packer that assembles **Label**, **SDI**, **DATA**, **SSM**, and computes **Bit 32 odd parity**. Keep the API agnostic to whether the adapter expects you to bit-reverse the Label byte.

```
#include <stdint.h>
#include <stdbool.h>

/* Reverse bit order within the 8-bit Label byte: b7..b0 → b0..b7 */
static inline uint8_t arinc429_reverse_label_bits(uint8_t label)
{
    uint8_t x = label;
    x = (uint8_t)(((x * 0x0802U & 0x22110U) | (x * 0x8020U & 0x88440U)) *
0x10101U >> 16);
    return x;
}

/* 32-bit population count */
static inline uint8_t popcount32(uint32_t v)
{
    v = v - ((v >> 1) & 0x55555555U);
    v = (v & 0x33333333U) + ((v >> 2) & 0x33333333U);
    return (uint8_t)((((v + (v >> 4)) & 0x0F0F0F0FU) * 0x01010101U) >> 24);
}

/* Pack an ARINC429 word in bit-number order (Bit 1 at LSB). */
uint32_t arinc429_pack(uint8_t label /*octal*/, uint8_t sdi /*0..3*/,
uint32_t data19 /*19 bits*/,
                       uint8_t ssm /*0..3*/, bool reverseLabelBitsForTx)
{
    uint32_t w = 0;
    uint8_t lab = reverseLabelBitsForTx ? arinc429_reverse_label_bits(label)
: label;

    w |= (uint32_t)(lab & 0xFFu);              /* Bits 1..8   */
    w |= (uint32_t)(sdi & 0x3u) << 8;          /* Bits 9..10 */
    w |= (uint32_t)(data19 & 0x1FFFFu) << 10; /* Bits 11..29 */
    w |= (uint32_t)(ssm & 0x3u) << 29;         /* Bits 30..31 */

    /* Odd parity over bits 1..31 ⇒ set Bit 32 so total ones (1..32) is odd
*/
    uint8_t ones = popcount32(w);
    if ((ones & 1u) == 0u) { w |= 0x80000000u; } /* set MSB (bit 32) */
    return w;
}

/* Helper: BNR two's-complement fixed-point for 19-bit field */
uint32_t arinc429_make_bnr_data19(float value, float lsb)
```

```c
{
    const float scale = 1.0f / lsb;
    int32_t i = (int32_t)(value * scale + (value >= 0 ? 0.5f : -0.5f));
    if (i >  0x3FFFF) i =  0x3FFFF;            /*  19-bit max  */
    if (i < -0x40000) i = -0x40000;            /* -19-bit min  */
    return (uint32_t)i & 0x1FFFFu;
}
```

Unit-test idea: Pack label **203_8**, SDI = 0, SSM = Normal, with **+3.25 °/s** and LSB = 1/128. Confirm the total '1' bits across 32 bits is odd.

Lab 2 — Transmit a word to ADK-8582 over LPUART

Goal: Bring up an EVKB **LPUART** (not the debug console) and send ASCII/HEX commands that the ADK's demo firmware understands to transmit an ARINC word on the line.

Key SDK pieces: `fsl_lpuart.h`, `board.h`, `clock_config.h`, `pin_mux.h`. Use the MCUXpresso *Pins* tool to route your chosen LPUART instance (e.g., **LPUART3**) to a header.

```c
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_lpuart.h"

#define ADK_UART                   LPUART3
#define ADK_UART_CLK_FREQ          CLOCK_GetFreq(kCLOCK_UartClk)
#define ADK_UART_BAUD              115200U   /* Confirm with ADK guide */
#define ADK_CMD_BUF_MAX            64

static void ADK_UART_Init(void)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps  = ADK_UART_BAUD;
    cfg.parityMode    = kLPUART_ParityDisabled;
    cfg.dataBitsCount = kLPUART_EightDataBits;
    cfg.isMsb         = false;
    LPUART_Init(ADK_UART, &cfg, ADK_UART_CLK_FREQ);
}

static void ADK_SendString(const char *s)
{
    LPUART_WriteBlocking(ADK_UART, (const uint8_t*)s, (size_t)strlen(s));
}

static void ADK_SendWord(uint32_t word)
{
    /* Many ADK firmwares accept: "TX <8 hex chars>\r\n". Adjust to your
firmware. */
```

```
    char buf[ADK_CMD_BUF_MAX];
    (void)snprintf(buf, sizeof(buf), "TX %08lX\r\n", (unsigned long)word);
    ADK_SendString(buf);
}

int main(void)
{
    BOARD_InitBootPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole(); /* LPUART1 stays for printf */

    ADK_UART_Init();

    const uint8_t  label203 = 0203; /* octal literal */
    const uint8_t  sdi = 0, ssmNO = 0b00;
    const bool reverseLabel = true;  /* Set per ADK expectations */

    while (1) {
        float value = 3.25f; /* example value */
        uint32_t data19 = arinc429_make_bnr_data19(value, 1.0f/128.0f);
        uint32_t w = arinc429_pack(label203, sdi, data19, ssmNO,
reverseLabel);
        ADK_SendWord(w);
        SDK_DelayAtLeastUs(20000, SystemCoreClock); /* 50 Hz pacing */
    }
}
```

Pin-mux note: The EVKB schematics and SDK `pin_mux.c` templates show which pins can serve LPUART3/5/6 TX/RX. Avoid conflicting with the debug console.

Lab 3 — Enforce update rates and inter-word gaps with PIT

Goal: Use the EVKB **PIT — Periodic Interrupt Timer** to schedule label-specific update rates (e.g., 50 Hz, 100 Hz) and avoid adapter FIFO overflows.

Steps: 1) Initialize PIT with a known source clock (e.g., Bus clock). 2) Configure a channel to interrupt at your desired period. 3) In the ISR, build words and call `ADK_SendWord()` for that label. Use separate PIT channels or a simple software scheduler for multiple rates.

Skeleton:

```
#include "fsl_pit.h"

static void PIT_Init_50Hz(void)
{
    pit_config_t pitCfg;
    PIT_GetDefaultConfig(&pitCfg);
    PIT_Init(PIT, &pitCfg);
    uint32_t srcClk = CLOCK_GetFreq(kCLOCK_BusClk);
    PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(20000U, srcClk)); /*
20 ms */
    PIT_EnableInterrupts(PIT, kPIT_Chnl_0, kPIT_TimerInterruptEnable);
    EnableIRQ(PIT_IRQn);
    PIT_StartTimer(PIT, kPIT_Chnl_0);
}

void PIT_IRQHandler(void)
{
    if (PIT_GetStatusFlags(PIT, kPIT_Chnl_0) & kPIT_TimerFlag) {
        PIT_ClearStatusFlags(PIT, kPIT_Chnl_0, kPIT_TimerFlag);
        /* Build and send one or more words at 50 Hz */
        /* ... call ADK_SendWord(w) ... */
    }
}
```

Inter-word gap: Many ARINC adapters allow configuring the **gap** they emit on the ARINC side. If your ADK exposes that setting via UART, choose **≥ 6–8 bit-times**. If not, pace host commands so the adapter's FIFO never underflows into minimum-gap bursts.

Lab 4 — Receive, validate, and decode returned words

Goal: Enable the ADK's loopback or connect an ARINC source, then read back received words over UART and verify **parity**, **Label**, **SDI**, **DATA**, and **SSM**.

Simple UART line reader:

```
static int ADK_ReadLine(char *dst, size_t maxLen, uint32_t timeoutMs)
{
    size_t n = 0; uint8_t ch; uint32_t t0 = SDK_Millis();
    while (n + 1 < maxLen) {
        if (LPUART_ReadByte(ADK_UART, &ch) == kStatus_Success) {
            if (ch == '\n') break;
            if (ch != '\r') dst[n++] = (char)ch;
        }
        if ((SDK_Millis() - t0) > timeoutMs) break;
    }
    dst[n] = '\0';
    return (int)n;
}
```

Parsing: If the ADK emits lines like `RX 12345678`, parse the 8 hex digits to a `uint32_t`, recompute parity with `popcount32()`, and then extract fields with masks/shifts. Apply Label bit reversal if the adapter presents raw on-the-wire ordering.

Lab 5 (optional) — Mix BNR and BCD, exercise SSM states

- Transmit a BCD-encoded altitude and deliberately toggle SSM among Normal, Functional Test, and No Computed Data. Confirm that downstream tools show the expected validity/state.

- Inject an odd-parity error (flip Bit 32) and observe that receivers reject the word.

Build & run checklist

- **Clocks:** `BOARD_BootClockRUN()` must set UART/PIT source clocks to the expected frequencies.
- **Pins:** Verify LPUART instance routing in `pin_mux.c` and conflict-free with debug UART.
- **Baud:** Match the ADK console baud (likely 115200-8-N-1).
- **ARINC rate:** Select 100 kbps or 12.5 kbps in the ADK command set before transmitting.
- **Verification:** With a scope or ARINC analyzer, confirm:
    - Bit rate within tolerance (10 µs per bit at 100 kbps; 80 µs per bit at 12.5 kbps).
    - BPRZ pulse width ≈ half-bit time, proper polarity for '1' and '0'.
    - Inter-word NULL gap ≥ **4 bit-times**; target **6–8**.
    - Parity is odd across full 32 bits.

---

5) Best practices

1) **Treat Labels as octal** everywhere: code, comments, and ICDs. This avoids mis-matches with standard tables.
2) **Centralize packing/unpacking and scaling** so parity, Label bit reversal, and BNR/BCD handling are uniform.
3) **Document SDI/SSM policy** in the ICD and enforce it in code reviews. Never leave SSM ambiguous.
4) **Budget channel load** and avoid minimum gaps in steady state; leave margin for maintenance traffic and phase-of-flight bursts.
5) **Verify on real hardware**: rise/fall times, pulse widths, gaps, and parity. Vary temperature and supply.
6) **Grounding and cabling**: use 78 Ω shielded twisted pair; respect stub length limits and shield termination practice.
7) **Error handling**: on RX, check parity and SSM; discard or flag words with errors or invalid status. On TX, fail safe (e.g., transmit NCD) when data becomes stale.

---

6) Common pitfalls (and how to avoid them)

- **Label bit order misunderstood** → PFD shows nonsense. *Fix:* always confirm whether your adapter expects the label byte bit-reversed.
- **Parity computed over wrong bits** → receivers drop words. *Fix:* compute over Bits 1–31 only; set Bit 32 afterwards.

- **Over-aggressive bursting** → violates effective gaps. *Fix:* use PIT-based pacing and set adapter gap to ≥ 6–8 bit-times.
- **Assuming SSM semantics** → downstream misinterpretation. *Fix:* follow your ICD for each label family.
- **Ignoring installation effects** → ringing, slow edges. *Fix:* scope at the line under worst-case load and temperature.

---

7) Summary: what to remember

- ARINC 429 uses fixed **32-bit words** with **odd parity**; **Label is Bits 1–8** and is **transmitted first (MSB-first)** while Bits 9–32 go **LSB-first**.
- Two rates: **100 kbps** (10 µs/bit) and **12.5 kbps** (80 µs/bit). Signaling is **BPRZ** with a **mandatory ≥ 4 bit-time** inter-word NULL gap.
- Prefer **scheduled transmissions** with **PIT** and keep average gaps ≥ 6–8 bit-times to be friendly to legacy receivers.
- Encode DATA using **BNR**, **BCD**, or **discretes** as your ICD specifies; use **SSM** to convey validity/state.

Appendix A — Notation quick-ref

- **ADIRU:** Air Data/Inertial Reference Unit
- **ADC:** Air Data Computer
- **BNR:** Binary Number Representation (two's-complement fixed-point)
- **BCD:** Binary-Coded Decimal
- **BPRZ:** Bipolar Return-to-Zero
- **DITS:** Digital Information Transfer System (ARINC 429 formal name)
- **FCC:** Flight Control Computer
- **FMS:** Flight Management System
- **ICD:** Interface Control Document
- **ISR:** Interrupt Service Routine
- **LRU:** Line-Replaceable Unit
- **LSB:** Least Significant Bit (also least-significant *bit weight* in fixed-point)
- **LPUART:** Low-Power Universal Asynchronous Receiver/Transmitter
- **NCD:** No Computed Data
- **NO:** Normal Operation
- **PFD:** Primary Flight Display
- **PIT:** Periodic Interrupt Timer
- **RZ:** Return-to-Zero signaling
- **SDI:** Source/Destination Identifier

- **SSM:** Sign/Status Matrix

Appendix B — Alternate (portable) parity helper

```
static inline uint32_t arinc429_apply_odd_parity(uint32_t w31)
{
    /* w31: Bits 1..31 populated; Bit 32 clear */
    uint32_t v = w31;
    v ^= v >> 16; v ^= v >> 8; v ^= v >> 4; v &= 0xFu;
    /* parity LUT for 0..15 nibble popcount parity */
    static const uint8_t lut[16] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};
    if (lut[v]) { /* odd already */ return w31; }
    else { return w31 | 0x80000000u; }
}
```

Appendix C — Field extraction masks (for RX)

```
#define A429_LABEL_MASK   0x000000FFu
#define A429_SDI_MASK     0x00000300u
#define A429_DATA_MASK    0x0FFFFC00u
#define A429_SSM_MASK     0x60000000u
#define A429_PARITY_MASK  0x80000000u

#define A429_LABEL(x)     ((uint8_t)((x) & A429_LABEL_MASK))
#define A429_SDI(x)       ((uint8_t)(((x) & A429_SDI_MASK) >> 8))
#define A429_DATA19(x)    ((uint32_t)(((x) & A429_DATA_MASK) >> 10))
#define A429_SSM(x)       ((uint8_t)(((x) & A429_SSM_MASK) >> 29))
#define A429_PARITY(x)    (((x) & A429_PARITY_MASK) ? 1u : 0u)
```
