ARINC 429 Protocol Fundamentals — Parity and Error Detection

---

1) What “parity” means on ARINC 429

ARINC (Aeronautical Radio, Incorporated) **429**—also called the *Mark 33 Digital Information Transfer System (DITS)*—moves information as fixed **32-bit “words”** over a unidirectional, two-wire, differential bus using **bipolar return-to-zero (BPRZ)** signalling. The 32nd bit of every word is the **parity bit**. ARINC 429 uses **odd parity**:

> Count the number of logic ‘1’ bits in bits **1–31**. Set bit **32** so that the **total** number of ’1’s in **all 32 bits** is **odd**.

If the receiver counts an even number of ’1’s across bits 1–32, it flags a **parity error** and—per the system Interface Control Document (**ICD**)—must **discard or de-weight** the word and usually set a **status** (for example, “No Computed Data”).

1.1 ARINC 429 word layout (conventional bit numbering)

- **Bits 1–8 — Label (8 bits):** Identifies what the word means (e.g., airspeed, altitude). **On the wire, the label’s LSB is transmitted first.**

- **Bits 9–10 — SDI (Source/Destination Identifier, 2 bits):** Multiplexing cue (which source or which destination the word is intended for).

- **Bits 11–29 — DATA (19 bits):** The payload, most commonly Binary Number Representation (**BNR**) or Binary-Coded Decimal (**BCD**).

- **Bits 30–31 — SSM (Sign/Status Matrix, 2 bits):** Sign and validity/status coding (e.g., *Normal*, *Functional Test*, *No Computed Data*, *Failure*).

- **Bit 32 — Parity (1 bit):** Odd parity across bits 1–31.

Implementation tip: It is common to store an ARINC word in a `uint32_t` where **bit 0 ↔ ARINC bit 1** and **bit 31 ↔ ARINC bit 32**. This makes masking/parity math simple even though the **label’s on-the-wire bit order is reversed**.

---

2) Why parity matters (and what it cannot do)

Parity is **simple, fast, and cheap** to check in hardware or software, and it **guarantees detection of any odd number of bit flips** in a 32-bit word (single-bit errors are the dominant random error on a clean bus). It **cannot** detect errors that flip an **even** number of bits within the same word. Therefore ARINC 429 robustness relies on **more than parity**:

- **SSM checks** (e.g., word marked *Failure* or *No Computed Data*).
- **Reasonableness/Range checks** (value within expected physical limits).
- **Rate-of-change monitors** (e.g., airspeed doesn't jump 100 kt in 10 ms).
- **Redundancy/Voting** across multiple sources (e.g., dual Air Data Computers).
- **Sequence/Refresh timing** monitors (label arrives at required rate).

For certification, combine parity with those higher-level monitors to meet **RTCA DO-178C** (Software Considerations in Airborne Systems and Equipment Certification) and **RTCA DO-254** (Design Assurance Guidance for Airborne Electronic Hardware).

---

3) Worked example — compute the parity bit (generic)

Assume we want to transmit an ARINC word with:

- **Label** = `0xAF` (binary `1010 1111`)
- **SDI** = `01b`
- **DATA (BNR)** = `0x12345` masked into 19 bits
- **SSM** = `00b` (Normal)

Pack bits 1–31 first (leave bit 32 = 0 while calculating). Count the number of '1's in bits 1–31. If that count is **even**, set bit 32 = 1. If it's **odd**, set bit 32 = 0. The total 1-bit count across 32 bits will then be **odd**.

A compact software way is to XOR-reduce (population count & 1) and invert as needed. See `arinc429_parity_odd_bit()` in the lab code below.

---

4) Avionics use-case scenario — detecting a corrupted airspeed word

A **Primary Flight Display (PFD)** receives **Indicated Airspeed** from an **Air Data Computer (ADC)** on a high-speed (100 kbps) ARINC 429 bus. The aircraft is in cruise at **280 kt**. Due to a transient disturbance on the wiring harness, **bit 17** of the airspeed word flips during transmission. Because only a single bit flipped, the parity at the receiver becomes **even**, the PFD's ARINC interface **flags a parity error**, discards that word, and **holds the last valid value**. If subsequent words also fail parity or the **SSM** transitions to *No Computed Data*, the PFD annunciates **"AIRSPEED DATA"** caution and transitions to **reversionary/standby indications**, per the system safety analysis.

If, instead, **two bits** flipped in the same word, the parity check alone might **not catch it**; this is where **range and rate monitors** (e.g., "airspeed change must be ≤ 10 kt per 40 ms") and **multi-source cross-checks** prevent hazardous effects.

---

5) Lab platform: IMXRT1050-EVKB + ADK-8582 (UART bridge)

For training, we treat **ADK-8582** as a UART-controlled ARINC 429 bridge/transceiver connected to the **NXP i.MX RT1050 EVKB**. The EVKB side uses **LPUART1** (pins `GPIO_AD_B0_12` TX and `GPIO_AD_B0_13` RX). We compute parity in software on the EVKB, then deliver **packed 32-bit words** to the ADK-8582 over UART for transmission on the ARINC bus. On receive, we read 32-bit words from the ADK-8582 and **verify parity** before passing data up to application logic.

> **Note:** Different ARINC-UART bridges package words differently (raw 4-byte binary, big-endian, little-endian, ASCII frames, checksums). The UART **framing** helpers below isolate that detail—adjust just those few lines to match your specific ADK-8582 command set/manual while keeping the **parity logic and SDK driver usage unchanged**.

5.1 Minimal, SDK-based UART bring-up (EVKB)

The snippet follows the EVKB SDK style (same APIs as the EVKB `lpuart_polling` example). It shows:

- LPUART1 init on EVKB.
- ARINC word packing with **odd parity**.
- Send/receive helpers that you can adapt to your ADK-8582 protocol.

```c
// File: arinc429_parity_demo.c
#include "board.h"
#include "app.h"                // DEMO_LPUART, DEMO_LPUART_CLK_FREQ
#include "fsl_lpuart.h"
```

```
#include "fsl_common.h"
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

// -------------------- ARINC 429 bitfield helpers --------------------
#define ARINC_LABEL_MASK   (0xFFu)       // bits 1..8
#define ARINC_SDI_MASK     (0x3u)        // bits 9..10
#define ARINC_DATA_MASK    (0x7FFFFu)    // bits 11..29 (19 bits)
#define ARINC_SSM_MASK     (0x3u)        // bits 30..31

// Bit positions in a uint32_t where bit0 == ARINC bit1, bit31 == ARINC bit32
#define ARINC_LABEL_POS    (0u)
#define ARINC_SDI_POS      (8u)
#define ARINC_DATA_POS     (10u)
#define ARINC_SSM_POS      (29u)
#define ARINC_PARITY_POS   (31u)

// Compute odd parity for bits 0..30 and return 0/1 to place into bit31.
// This produces bit31 such that the total number of 1s in all 32 bits is
odd.
static inline uint32_t arinc429_parity_odd_bit(uint32_t w31)
{
    // XOR-reduce over bits 0..30.
    // If popcount is even => xor = 0 => parity bit must be 1 (to make total
odd).
    // If popcount is odd  => xor = 1 => parity bit must be 0.
    // So: parity_bit = (~xor) & 1
    uint32_t x = w31 ^ (w31 >> 16);
    x ^= (x >> 8);
    x ^= (x >> 4);
    x ^= (x >> 2);
    x ^= (x >> 1);
    uint32_t xor1 = x & 1u;
    return (xor1 ^ 1u); // invert to get odd parity
}

// Pack bits 1..31, compute and insert bit32 (odd parity)
static inline uint32_t arinc429_pack(uint8_t label, uint8_t sdi, uint32_t
data19, uint8_t ssm)
{
    uint32_t w = 0;
    w |= ((uint32_t)(label & ARINC_LABEL_MASK) << ARINC_LABEL_POS);
    w |= ((uint32_t)(sdi   & ARINC_SDI_MASK)   << ARINC_SDI_POS);
    w |= ((uint32_t)(data19 & ARINC_DATA_MASK) << ARINC_DATA_POS);
    w |= ((uint32_t)(ssm   & ARINC_SSM_MASK)   << ARINC_SSM_POS);
    // compute parity on bits 0..30
    uint32_t p = arinc429_parity_odd_bit(w);
    w |= (p << ARINC_PARITY_POS);
    return w;
}

// Verify odd parity: return true if word passes parity
static inline bool arinc429_check_parity(uint32_t word)
{
    // Total 1s across all 32 bits must be odd => XOR-reduce must be 1
    uint32_t x = word ^ (word >> 16);
    x ^= (x >> 8);
    x ^= (x >> 4);
    x ^= (x >> 2);
    x ^= (x >> 1);
    return (x & 1u) == 1u;
}

// -------------------- ADK-8582 UART framing (adjust to your board) --------
------------
// Option A: raw 4-byte big-endian word
static status_t adk_send_word_raw_be(uint32_t w)
{
    uint8_t frame[4] = { (uint8_t)(w >> 24), (uint8_t)(w >> 16), (uint8_t)(w
>> 8), (uint8_t)(w) };
    LPUART_WriteBlocking(DEMO_LPUART, frame, sizeof(frame));
    return kStatus_Success;
}

static status_t adk_read_word_raw_be(uint32_t *out)
{
    uint8_t buf[4];
    LPUART_ReadBlocking(DEMO_LPUART, buf, 4);
    *out = ((uint32_t)buf[0] << 24) | ((uint32_t)buf[1] << 16) |
((uint32_t)buf[2] << 8) | buf[3];
    return kStatus_Success;
}

// -------------------- EVKB LPUART setup --------------------
static void uart_init_115200_8N1(void)
{
    lpuart_config_t config;
    LPUART_GetDefaultConfig(&config);
    config.baudRate_Bps = 115200;
    config.enableTx = true;
    config.enableRx = true;
    LPUART_Init(DEMO_LPUART, &config, DEMO_LPUART_CLK_FREQ);
}

// Example: transmit one word and then read one word back and check parity
void arinc_parity_demo(void)
```

```
{
    uint8_t  label = 0xAF;       // example label (ICD will define the real
one)
    uint8_t  sdi   = 0x1;
    uint32_t data  = 0x12345;    // 19-bit application data
    uint8_t  ssm   = 0x0;        // Normal
    uint32_t wtx   = arinc429_pack(label, sdi, data, ssm);

    adk_send_word_raw_be(wtx);

    // Receive a word and validate parity
    uint32_t wrx = 0;
    adk_read_word_raw_be(&wrx);
    bool ok = arinc429_check_parity(wrx);

    // In a real app, do not printf on the flight CPU path; use queues/state.
    const char *msg = ok ? "PARITY OK\r\n" : "PARITY ERROR\r\n";
    LPUART_WriteBlocking(DEMO_LPUART, (const uint8_t*)msg,
(uint32_t)strlen(msg));
}

int main(void)
{
    BOARD_InitHardware();   // From SDK template; includes pins/clock.
    uart_init_115200_8N1();
    while (1) {
        arinc_parity_demo();
        // Simple pacing; replace with proper scheduler/RTOS tick in real
code
        for (volatile uint32_t i = 0; i < 12000000; ++i) { __NOP(); }
    }
}
```

Why this matches the EVKB SDK: It uses the standard driver layer (`fsl_lpuart.h`), the `LPUART_GetDefaultConfig` / `LPUART_Init` / `*_WriteBlocking` / `*_ReadBlocking` calls, and assumes the EVKB LPUART instance/pins used by the official examples. Replace the two adk_* functions with the exact frame your ADK-8582 requires (binary vs ASCII, endianness, checksums).

5.2 Bench wiring and configuration checklist

- **Levels:** Ensure EVKB UART is 3.3 V TTL and **matches** the ADK-8582 UART levels.
- **Ground:** Common ground between EVKB and ADK-8582.
- **UART settings:** Start with **115200 8-N-1**; adjust to the ADK-8582's documented default.

- **ARINC port rate:** Configure ADK-8582 channel as **Low-speed (12.5 kbps)** or **High-speed (100 kbps)** per test.
- **Byte order:** The example sends **big-endian** (network order). Change if your bridge expects little-endian.
- **Label bit order:** Do **not** reverse label bits in memory—only the physical line order is reversed; parity math remains unchanged.

---

6) Hands-on exercises (with solutions)

Exercise A — Parity generator/validator (software)

Goal: Write two functions on the EVKB: `arinc429_pack()` and `arinc429_check_parity()`.

Pass criteria: For 1000 random words, packing + checking always passes; flipping any **single** bit in bits 1–32 always **fails** parity.

Solution scaffold: Functions provided above. To test, add a loop that flips each bit b:

```
for (uint32_t b = 0; b < 32; ++b) {
    uint32_t w = arinc429_pack(0x22, 0x0, 0x12345, 0x1);
    uint32_t wbad = w ^ (1u << b);
    assert(arinc429_check_parity(w) == true);
    assert(arinc429_check_parity(wbad) == false);
}
```

Exercise B — Inject and detect an error end-to-end

Goal: Demonstrate a **single-bit flip** causes a parity failure at the receiver.

1) On the EVKB, compute a good word.
2) Before sending to the ADK-8582, deliberately flip one randomly chosen bit among **bits 1–31** (don't flip the parity bit; we want an odd error count).
3) Transmit.
4) On the return path (EVKB receiving from ADK-8582 or from a loopback), call `arinc429_check_parity()` and increment a counter on failure.

Snippet:

```
uint32_t inject_single_bit_error(uint32_t w)
{
    uint32_t b = (SysTick->VAL % 31u);    // toy pseudo-random bit index 0..30
    return w ^ (1u << b);
}
```

Expected: Every injected frame fails parity.

Exercise C — SSM behavior on parity error (avionics use case)

Goal: Emulate a **Primary Flight Display (PFD)** input monitor.

- If **parity OK** and data **in range**, forward the value with `SSM = Normal`.
- If **parity error**, **discard** the word, **hold last good**, and set a **downstream status** to "**NCD (No Computed Data)**" after *N* consecutive parity failures.
- If your ADK-8582 lets you **force SSM**, send the same label with `SSM = Failure` and verify the receiver logic treats it as invalid **even if parity is OK**.

This shows **parity is necessary but not sufficient**; SSM and reasonableness logic complete the safety case.

---

7) Practical details that often trip people up

- **Label bit order vs. storage:** On the wire, the label's **LSB goes first**; in memory, keep the label as an 8-bit value and **don't reverse bits** unless your bridge specifically requires it. Parity calculation is unaffected by wire order—always compute across **bits 1–31 as stored**.
- **Endianness over UART:** EVKB is little-endian; many bridges expect **big-endian** words. The example explicitly builds a big-endian frame—change if needed.
- **Parity over what?** Only **bits 1–31**. **Never include bit 32** in the parity calculation.
- **Don't "fix up" bad words:** If a received word fails parity, **discard it**; do not try to guess the flipped bit.
- **Refresh/rates:** Even perfect-parity words that **stop arriving at the expected refresh rate** must be treated as invalid by higher-level logic.
- **Determinism:** In RTOS builds, do parity checks in the **driver ISR or DMA completion** path so applications only see validated words.

---

8) Best practices (Airbus-style rigor)

1. **Compute parity where the word is formed** and **verify parity where the word is consumed**. Do not depend on "someone upstream checked it."
2. **Instrument error counters** (per label and per source). Trend them and expose via built-in test (**BIT**) to catch intermittent wiring faults.
3. **Validate SSM semantics** end-to-end during integration. Parity-OK + `SSM = Failure` must be treated as **invalid data**.
4. **Rate/range/sanity checks:** Implement label-specific monitors (e.g., airspeed ∈ [0, 450] kt, Δv/Δt bounded).
5. **Keep parity logic constant-time** and side-effect free. The XOR-reduction compiles to a handful of shifts/XORs with no branches.
6. **Lock down a UART-to-ARINC ICD** early (byte order, escapes, checksums). Only the adk_* helpers should change when you update the ICD; parity and SDK usage stay fixed.
7. **Handle noise bursts:** Use **glitch counters** and **timeout-based de-weighting** rather than instantaneous failover to prevent nuisance reconfigurations.
8. **Certification evidence:** Tie your unit tests (including the "flip any single bit" test) to **DO-178C** verification objectives and store logs as artifacts.
9. **Clocking and pins:** Verify **LPUART1 clock source** and **IO mux** match EVKB defaults; mismatches lead to subtle framing errors you may misinterpret as parity faults.
10. **Telemetry for integration:** Provide a debug counter of `parity_errors_per_label_per_minute` to aid flight-line troubleshooting.

---

9) Quick reference: pack/unpack masks

- **Pack:**
  `word = (label<<0) | (sdi<<8) | (data19<<10) | (ssm<<29);` then set bit 31 using `arinc429_parity_odd_bit(word)`.
- **Check:**
  `bool ok = arinc429_check_parity(word);`

10) Resources

- **Source file:** `arinc429_parity_demo.c` (drop into an EVKB SDK application skeleton; it uses the same `app.h`/`BOARD_InitHardware()` pattern as the SDK examples).

- **Pinout:** EVKB **LPUART1** on `GPIO_AD_B0_12` (TX) / `GPIO_AD_B0_13` (RX) to ADK-8582 UART.

- **One-page ICD:** Define the **UART frame** for an ARINC word (binary or ASCII, byte order, and any checksum). Only the two `adk_*` helpers change when you update the ICD; the **parity logic and SDK usage stay fixed**.