



Protocol

English version. In case of doubt, the English text shall be binding.

Deutsche Version anzeigen

Automatic Stations at the ARDF Transmitting Facilities (Foxes)

A fox regularly broadcasts its identifier and telemetry data. This is done, whenever possible, at times when the fox is not transmitting on the 80m or 2m bands. The purpose of this is, on the one hand, the time synchronization of adjacent nodes, and on the other hand, the regular official announcement of the call sign. Usually, the call sign of the organizer or host of the direction-finding competition is used.

Beacon

Composed based on the modified AX.25, APRS, and MeshCom protocols as follows:

Pseudo-Byte-Stream:

```
{0E3TKT-03CpnnnnxRSqxRSqxRSq. .
```

Structure

Byte / Offset	Field / Format	Description
Byte 0	"{" 0x7B	Data Type Identifier (DTI): Derived from the APRS & MeshCom protocol specification (see APRS Protocol Reference). Reasoning: Currently marked as [Unused] in the specification; character visually resembles a station icon.
Bytes 1–6	"0E3TKT" (6 Bytes ASCII)	International Call Sign: Formatted as a 1–2 character prefix, followed by one or more digits, and a 1–3 letter suffix. Strictly 6 characters wide, right-padded with trailing spaces if shorter.
Byte 7	"-" 0x2D or "~" 0x7E	Node Type Separator: Hyphen for standard nodes, tilde for backup/hot-standby nodes. Serves as a visual delimiter and structural anchor for the parser.
Bytes 8–9	"03", or "S6" (2 Bytes ASCII)	Station SSID Identifier: Sub-string format representing the node ID. Supports numeric IDs (0–99) or extended notation (e.g., "S6") i.e. any letters as described in aprs11/SSIDs.txt (e.g., 0E3TKT-S6 for any special 6th station).
Byte 10	Own Battery Level (unsigned char)	Value: Capacity in percent (0 to 100%). Values above 100 indicate charging or mains power.
Byte 11	Protocol / Net Flags (unsigned char)	Network Flags / Padding: Reserved for mesh network flags and hop control. Ensures precise 4-byte memory alignment for the subsequent 32-bit timestamp.
Bytes 12–15	nnnn / Timestamp (int32_t, Little-Endian)	Timestamp: Signed 32-bit integer representing the number of milliseconds since the start of the competition day. Formatted as Little-Endian (LSB first) and aligned to a 4-byte boundary for optimized, native ESP32 pointer access. All official ARDF contest times and logs must be recorded in Coordinated Universal Time (UTC). To convert <i>[Local Time Name]</i> to UTC, subtract/add <i>[X]</i>

		hours (UTC = $[Local\ Time] \pm [X]h$). Please pay close attention to date changes for events crossing midnight UTC.
Bytes 15+	Heard Neighbor Nodes (xRSqxRSqxRSq...)	Multi-byte blocks containing mesh neighbor details (SSID, link quality, battery status). See detailed breakdown below.

Heard Neighbor Nodes Block Breakdown (Bytes 15+)

Byte Offset	Field / Format	Description
Bytes 15–16	Neighbor SSID (2 Bytes ASCII)	Two ASCII characters representing the neighbor's station ID/SSID (e.g., "03" or "56"). This matches the exact sub-string format from Bytes 8–9 of the source node for easy string matching.
Byte 17	RSSI Value (R) (unsigned char)	Stored as RSSI + 164 (e.g., 0x3A = 58 -> 58 - 164 = -106 dBm).
Byte 18	SNR Value (S) (signed char)	Stored as SNR multiplied by 4 (e.g., 0x42 = 66 -> 66 / 4 = +16.5 dB).
Byte 19	Status Byte (q) (unsigned char)	Combined Quality and Battery Status Byte. (See Bit-mapping below).

Byte 19 (q) Bit-mapping:

Bits	Value	Definition
Bits 7–6	00	Normal (Battery OK / Powered)
Bits 7–6	01	Low (Battery low, intervention recommended)
Bits 7–6	10	Critical (Battery critical, node might shut down soon)
Bits 7–6	11	Reserved
Bits 5–0	0 to 63	Moving average counter of how many times this station was heard within the last 30 minutes. If ≥ 3 , the link is considered stable. Dropped if 0 within 30 minutes.

Checkpoint Messages (Posten)

As soon as a participant checks in at a checkpoint using their RFID tag or identifier, they are registered at that node. The runner is assigned a timestamp and their ID (runner number). The mapping between the runner number and the RFID code is stored within the system. During the next available time slot, a checkpoint message is transmitted to the destination with QoS 2 (guaranteed packet delivery, duplicate detection, and filtering).

Messages are sent to the destination and optionally to the start node simultaneously. Both nodes must have an SSID < 15, as the remaining bits in the address byte are reserved for protocol flags.

This message packet is designed to be extremely short and efficient. It contains the node number, runner number, capture timestamp, and the current hop count.

Pseudo-Byte-Stream:

(ppMLzzzzRR

Structure (Checkpoint Message)

Byte / Offset	Field / Format	Description
---------------	----------------	-------------

Byte 0	(0x40)	Data Type Identifier (DTI): Special identifier designated for FoxMash checkpoint messages.
Bytes 1–2	Own SSID (pp) (2 Bytes ASCII)	Two ASCII characters representing the transmitting checkpoint node.
Byte 3	Routing & Flags (M) (unsigned char)	Combined Target SSID, Hop Count, and ACK Flags. (See Bit-mapping below).
Bytes 4–7	zzzz (Timestamp) (int32_t, Little-Endian)	Timestamp: Signed 32-bit integer representing the number of milliseconds since the start of the competition day. Formatted as Little-Endian (LSB first) and aligned to a 4-byte boundary for direct, high-performance ESP32 pointer access.
Bytes 8–9	Runner Number (L) (uint16_t)	Unique participant identification / runner number (Supports IDs from 0 to 65,535).
Bytes 10	Hop Count / Flags (unsigned char)	Current network hop count and remaining mesh routing flags.

Byte 3 (M) Bit-mapping:

Bits	Flag / Field	Definition
Bits 7–5	Hop Count	Dynamic routing counter. 0 if destination is seen directly with good quality. Incremented by 1 if no ACK within 5 minutes. Discarded if it exceeds max hop value.
Bit 4	ACK Flag	0 for a standard data message, 1 for an acknowledgment packet (ACK).
Bits 3–0	Target SSID	The SSID of the destination node (Supports IDs from 0 to 15).

Acknowledgments and Retries

The maximum number of retries is configurable (default: 4). Every checkpoint packet must be acknowledged. The acknowledgment uses the exact same format (vppMLzzzzRR), but with **Bit 4 in Byte 3 (M) set to 1**. The return routing follows the identical path in the reverse direction based on the route recorded in the packet (source routing). Remit nodes detect and discard duplicate packets by matching the runner's unique timestamp.

Protocol Optimization Analysis

Hardware-Optimized Protocol Design for ESP32 and LoRa

When designing low-power, high-performance mesh networks using ESP32 microcontrollers over LoRa modulation, data structure layout critically impacts both transmission time (Airtime) and CPU efficiency. The updated FoxMash protocol layout restructures fields to achieve an optimal balance between RF efficiency and hardware execution.

Why this Layout is Perfect

1. **Native Word Alignment (No Performance Penalties):** The Tensilica Xtensa and RISC-V architectures used in ESP32 chips are natively 32-bit Little-Endian processors. Accessing a 4-byte integer from a memory address that is not a multiple of 4 (*unaligned access*) causes the hardware to execute multi-cycle fixes, shifting bytes internally. In worst-case scenarios or specific compiler configurations, this causes a `LoadStoreAlignmentCause` hardware crash (Core Dump). By placing the 32-bit timestamp at precisely **Byte 4** (Checkpoint) and **Byte 12** (Beacon), the integer aligns perfectly with the processor's natural word boundaries.

2. **Zero-Copy Pointer Casting:** Because fields are naturally aligned, the firmware does not need to waste clock cycles using `memcpy()` or bit-shifting operators (`<<`, `|`) to reconstruct integers from individual bytes. The application layer can execute direct pointer casting directly out of the LoRa hardware receive buffer.
3. **Zero Airtime Overhead:** Traditional alignment techniques insert dummy padding bytes (`0x00`) to force alignment, which inflates the packet size. In LoRa networks, every additional byte directly increases the Time-on-Air (ToA), resulting in higher energy consumption and a higher probability of packet collisions. This optimized layout achieves perfect alignment solely by **reordering existing essential fields**, keeping the frame size strictly minimal.
4. **Human-Readable Delimiters Maintained:** Keeping the standard hyphen ("-") or tilde("~") delimiter between the international call sign and the SSID preserves compatibility with legacy APRS/MeshCom parsers and keeps raw hexadecimal logs intuitive during serial debugging.

High-Performance C++ Implementation Example

The following code snippet demonstrates how to leverage this memory-aligned layout within an ESP32 application utilizing standard Arduino or ESP-IDF LoRa libraries.

```

#include <Arduino.h>

// Ensure the compiler doesn't add structural padding within our structs
#pragma pack(push, 1)

struct BeaconPacket {
    char dti;                // Offset 0 (0x7B '{')
    char callsign[6];        // Offset 1 (e.g., "0E3TKT")
    char separator;          // Offset 7 (e.g., '-')
    char ssid[2];            // Offset 8 (e.g., "03")
    uint8_t battery;         // Offset 10 (0-100%)
    uint8_t net_flags;       // Offset 11 (Routing control / Padding)
    int32_t timestamp_ms;    // Offset 12 -> PERFECTLY 4-BYTE ALIGNED!
    uint8_t neighbor_block[]; // Offset 16+ (Dynamic mesh payload)
};

struct CheckpointPacket {
    char dti;                // Offset 0 (0x28 '(')
    uint16_t own_ssid;        // Offset 1
    uint8_t routing_flags;    // Offset 3 (Target / Control Flags)
    int32_t timestamp_ms;    // Offset 4 -> PERFECTLY 4-BYTE ALIGNED!
    uint16_t runner_id;      // Offset 8
    uint8_t hop_count;       // Offset 10
};

#pragma pack(pop)

// Example parser function processing incoming raw LoRa bytes
void processIncomingLoRa(uint8_t* rxBuffer, size_t packetLength) {
    if (packetLength < 1) return;

    // Check Data Type Identifier (DTI)
    char dti = rxBuffer[0];

    if (dti == '(') {
        // Zero-copy direct cast pointer access
        CheckpointPacket* cp = (CheckpointPacket*)rxBuffer;

        // Fast native access directly from memory
        int32_t time = cp->timestamp_ms;
        uint16_t runner = cp->runner_id;

        Serial.printf("Runner %d checked in. Local Timestamp: %d ms\n", runner, time);
    }
    else if (dti == '{') {
        BeaconPacket* beacon = (BeaconPacket*)rxBuffer;

        // Fast native access directly from memory
        int32_t time = beacon->timestamp_ms;

        Serial.printf("Beacon received. Network Timestamp: %d ms\n", time);
    }
}

```

Document maintained and compiled by [OE3TKT, Tom]

Typeset beautifully with Typst ♥