U4BProtocol Specification

w/references to WsprEncoded C/Python Libraries

Current versions

U4BProtocol Specification v1.1

WsprEncoded Library v4.3.2

Table of Contents

Intent

QRPLabs for some time has sold a tracker https://www.qrp-labs.com/u4b.html that has a behavior for transmitting WSPR packets to convey information from a balloon in flight over planet Earth. This info is received by worldwide WSPR receivers, and the data uploaded to accessible databases like wsprlive and wsprnet.

A number of trackers have been produced, some open-source and available on github.com that have behavior compatible with the QRPLabs U4B “protocol”. Additionally, websites have been created, or other software, that interprets the WSPR data and creates various analysis or visualizations.

The behavior of this protocol has changed over time, and various software may or may not support what is considered the “current definition”. That is fine. There is a discussion process at https://groups.io/g/picoballoon that attempts to keep all interested parties in alignment over time, with the benefit of shared software or website use, and to avoid conflict..i.e. one person’s tracker garbling correct reception of information from another person’s tracker.

QRPLabs has graciously allowed this compatible reverse-engineering. Hans G0UPL, over time, has released details of the U4B behaviors, which can be considered “U4B Protocol”.

QRPLabs is under no agreement to comply in the future with anything here, and this spec does not cover all behaviors of a QRPLabs U4B tracker.

The goal is to serve a common interest for all people in the u4b or u4b-compatible world, and create more fun!

Part of the fun is innovation. Change over time should be slow to minimize chaos in the U4B-compatible world, but hopefully it continues to happen. Important changes will be highlighted in the Updates section.

Some specification detail available on https://www.qrp-labs.com may be out of date and not reflect the true current behavior of U4B trackers.

This spec should be considered the current community consensus. Issues or Discussion can be posted at

https://github.com/traquito/WsprEncoded

https://github.com/traquito/WsprEncodedPython

The intent is easy understanding by non-English speakers. It may take some time to get to a good spec, so revisions may happen more quickly during 2025.

Updates

V1.1 Initial release at the current repo: https://github.com/knormoyle/U4BProtocol

History

U3B and U3S WSPR on balloons appeared around 2015, with U4B development after that by Hans G0UPL and Dave VE3KCL.

Some other trackers reverse-engineered U4B protocol, but Doug KD2KDD did extensive work in that area starting in 2023. This was called the Traquito project.

Doug KD2KDD organized a collaborative effort to extend U4B protocol in 12/24 and made the extensions (Extended Telemetry) available under AGPL.

The original U4B protocol before 12/24 is considered to be Basic Telemetry. The extensions are considered to be Extended Telemery. Hans G0UPL aligned QRPLabs U4B firmware to the Extended Telemetry spec around 7/25.

The supporting library and clarity of definition here, are all because of Doug’s work, and the community thanks him for his time and commitment to the idea.

License

This specification and the WsprEncoded library is under AGPL license.

This document provides specification for Basic and Extended U4B protocol, definition of channels to isolate telemetry from different transmitters, and a description of the WsprEncoded library available at https://github.com/traquito/WsprEncoded to simplify implementation of the protocol.

Contributions, modifications and extensions are encouraged and merges will be accepted via https://github.com/knormoyle/U4BProtocol repo.

U4BProtocol includes Basic Telemetry and Extended Telemetry. QRPLabs has some alternative transmissions with its legacy TELE instruction that are not part of this spec. The QRPLabs TELEX command meets the Extended Telemetry spec.

Basic Telemetry

Overview

The Basic Telemetry API provides a standardized method for encoding common tracker measurements into WSPR Type 1 messages. This system enables transmission of GPS location, environmental sensors, and system status information through the WSPR protocol's 50-bit data capacity.

Note that there can be bad GPS data on altitude, latitude and longitude, and voltage measurement can be bad also.

For example, if the tracker is not getting 3.3v to an rp2040 and rp2040 voltage measurement is only "right" if supply voltage to the rp2040 and the temperature ADC is getting a correct 3.3v compare voltage. A tracker may implement additional filtering or clamping to deal with those issues.

The language below governs what result happens on a website if no such filtering happens, i.e. what values are reported from particular encoded values.

Note there is no such thing as Nan or Invalid encodings. So it’s important to understand that Basic Telemetry will always decode to something. You always get an answer. The validity of that answer depends on an understanding of all of U4B Protocol.

Decoding of random unused callsign space spots from wsprlive will show that everything can decode to something (even it's out of the valid range of the intermediate "big number" for U4B protocol, if there is no checking for that).

The current basic telemetry doesn't fully occupy the space allowed by the wspr big-number encoding process. So just being able to decode something doesn’t validate that it’s U4B Protocol information.

It’s possible that unused big number space could be used for "anything" like NaN or None. Place holder for the future.

A common case is accidentally decoding WB8ELK protocol telemetry. It will sometimes have a intermediate big number range that’s illegal relative to U4B protocol telemetry. A website could detect that. It can be discovered by looking at the big number being out-of-expected-range, or having a remainder left over after decoding all known U4B protocol subfields from the big number.

Message Format

Basic Telemetry messages use the WSPR Type 1 format with specific field allocations:

```
<callsign> <grid> <power>
```

Field Mapping:

- **Callsign**: Encodes Grid5, Grid6, and Altitude
- **Grid**: Encodes Temperature, Voltage, Speed, GPS validity, and telemetry type
- **Power**: Part of the encoding scheme

Telemetry Fields

Field Specifications

Field Name	Unit	Min Value	Max Value	Step Size	Possible Values
Grid5	Character	0	23	1	24
Grid6	Character	0	23	1	24
Altitude	Meters	0	21,340	20	1,068
Temperature	Celsius	-50	39	1	90
Voltage	Volts	2.0	3.95	0.05	40
Speed	Knots	0	82	2	42
IsGpsValid	Boolean	0	1	1	2
HdrTelemetryType	Enum	0	1	1	2

Grid Position Enhancement (Grid5 & Grid6)

Purpose

Extends the 4-character WSPR grid to 6 characters for improved location precision.

Resolution Improvement

- **4-character grid**: 70 × 140 miles resolution
- **6-character grid**: 3 × 3 miles resolution

Encoding

```
// Grid5 and Grid6 encode characters A-X (values 0-23)
char grid5_char = 'A' + grid5_value;  // grid5_value: 0-23
char grid6_char = 'A' + grid6_value;  // grid6_value: 0-23

// Complete 6-character grid formation
std::string full_grid = wspr_grid_4char + grid5_char + grid6_char;
```

Example

```
// WSPR Type 1 grid: "FN31"
// Grid5 = 12 (M), Grid6 = 7 (H)
// Complete grid: "FN31MH"
```

Altitude Measurement

Specification

- **Source**: GPS-derived elevation
- **Range**: 0 to 21,340 meters
- **Resolution**: 20-meter steps
- **Encoding**: Linear quantization with rollover support

Usage

```
uint16_t encodeAltitude(uint16_t altitude_meters) {
    // Rollover at 21340m.
```

```
    // Trackers may optionally clamp to min/max range
    // Negative numbers should also be filtered, in case of bad gps data
    altitude_meters = altitude_meters % 21340;

    // Quantize to 20m steps
    return altitude_meters / 20;
}

uint16_t decodeAltitude(uint16_t encoded_value) {
    return encoded_value * 20;  // Convert back to meters
}
```

Temperature Measurement

Specification

- **Source**: Environmental sensor or onboard temperature sensor
- **Range**: -50°C to +39°C
- **Resolution**: 1°C steps
- **Encoding**: Offset binary with rollover

Rollover Behavior

```
int16_t encodeTemperature(int16_t temp_celsius) {
    // Apply offset and rollover.
    // Trackers should filter for negative results and do something (clamp?)
    return (temp_celsius - (-50)) % 90;
}

int16_t decodeTemperature(uint16_t encoded_value) {
    // Note: Rollover means ambiguous decoding
    return encoded_value + (-50);
}
```

Rollover Example

```
// Both -45°C and +45°C encode to the same value (5)
int16_t temp1 = -45;  // encodes to: (-45 - (-50)) % 90 = 5
int16_t temp2 = 45;   // encodes to: (45 - (-50)) % 90 = 5
```

Voltage Measurement

Specification

- **Source**: System input voltage monitoring
- **Range**: 3.0 to 4.95v is considered the standard range, although a tracker can do continual wraparound, so there is no range limits, or pick a different 1.95v range
- **Resolution**: 0.05V steps
- **Purpose**: Monitor solar panel or battery performance

Standard Rollover Ranges

The voltage field can support multiple 1.95V ranges. There is no standard for how to decide what range a tracker is using though. 3.0 to 4.95v is considered the standard range. A tracker may clamp to min/max values for the range or not. Website viewers typically do not have a way of visualizing voltages correctly outside of the 3.0v to 4.95v range.

It's possible website developers could have a configurable "base voltage" so that balloons with different known voltage ranges could be supported and visualized easily. But 3-4.95v is the consensus, non-configurable range that websites should report.

Possible examples:

- 2.0V to 3.95V
- 4.0V to 5.95V
- 6.0V to 7.95V

Encoding

```
uint16_t encodeVoltage(float voltage) {
    uint16_t range = (uint16_t)((voltage - 2.0) / 2.0);
    float range_offset = voltage - (2.0 + range * 2.0);

    // Quantize within range
    uint16_t step = (uint16_t)(range_offset / 0.05);

    return step % 40;  // Rollover within 40 possible values
}
```

Alternative Encoding example for only 3.0 to 4.95v with clamping

```
uint16_t encodeVoltage(float voltage) {
    // voltage encodings:
    // 20 to 39, 0 to 19 for 3.00 to 4.95V with a resolution of 0.05V
    // 0 to 39 encodings
    if (voltage > 4.95) voltage = 4.95;
    else if (voltage < 3.00) voltage = 3.00;
    // should only be 3 to 4.95
    uint16_t voltageNum = (int)(round ((voltage - 3.00) / .05) + 20) % 40;
    return voltageNum

}
```

Decoding voltage assuming it represents 3.0 to 4.95v range at original measurement

```
uint16_t decodedVoltage(float encoded_value) {
    voltage = round((encoded_value * 0.05) + 2.00, 2)
    return voltage
}
```

Speed Measurement

Specification

- **Source**: GPS-derived ground speed
- **Range**: 0 to 82 knots
- **Resolution**: 2-knot steps
- **Encoding**: Linear quantization with rollover

Usage

```
uint16_t encodeSpeed(uint16_t speed_knots) {
    return (speed_knots / 2) % 42;  // Rollover support
```

```
}

uint16_t decodeSpeed(uint16_t encoded_value) {
    return encoded_value * 2;  // Convert back to knots
}
```

GPS Validity Flag

Purpose

Indicates whether GPS-derived measurements (altitude, speed, grid position) are valid. Some trackers may only send callsign+telemetry when GPS is valid, so this may always be 1. QRPLabs u4b can send callsign+telemetry when GPS is not valid. Old values are used for telemetry then, although it's possible some data should be ignored if GPS is not valid.

Values

- **0 (false)**: GPS lock not available, position/speed data invalid
- **1 (true)**: GPS lock acquired, position/speed data valid

Usage

```
bool gps_valid = hasGpsLock();
uint8_t gps_flag = gps_valid ? 1 : 0;
```

Telemetry Type Header

Purpose

Identifies the telemetry format version.

Values

- **0**: Reserved
- **1**: Standard Basic Telemetry format

Usage

```
const uint8_t TELEMETRY_TYPE_STANDARD = 1;
```

Encoding examples

`encodeBasicTelemetry()`

Encodes all basic telemetry fields into a WSPR message.

```
struct BasicTelemetryData {
    uint8_t grid5;              // 0-23
    uint8_t grid6;              // 0-23
    uint16_t altitude_meters;   // 0-21340
    int16_t temperature_c;      // -50 to 39
    float voltage_v;            // 2.0-3.95 (or rollover ranges)
    uint16_t speed_knots;       // 0-82
    bool gps_valid;             // true/false
    uint8_t telemetry_type;     // 1 for standard
};

bool encodeBasicTelemetry(
```

```
    const BasicTelemetryData& data,
    WSPRMessage& message
);
```

`decodeBasicTelemetry()`

Decodes basic telemetry from a WSPR message.

```
bool decodeBasicTelemetry(
    const WSPRMessage& message,
    BasicTelemetryData& data
);
```

Complete Example

```
#include <WsprEncoded/BasicTelemetry.h>

void transmitBasicTelemetry() {
    // Collect sensor data
    BasicTelemetryData telemetry = {
        .grid5 = 12,           // 'M'
        .grid6 = 7,            // 'H'
        .altitude_meters = 1200,
        .temperature_c = 25,
        .voltage_v = 3.7,
        .speed_knots = 0,
        .gps_valid = true,
        .telemetry_type = 1
    };

    // Encode to WSPR message
    WSPRMessage message;
    if (encodeBasicTelemetry(telemetry, message)) {
        // Transmit WSPR message
        printf("Transmitting: %s %s %d\n",
               message.callsign, message.grid, message.power);

        // Send via radio...
        transmitWSPR(message);
    }
}

void receiveBasicTelemetry(const WSPRMessage& received) {
    BasicTelemetryData telemetry;

    if (decodeBasicTelemetry(received, telemetry)) {
        printf("Decoded telemetry:\n");
        printf("  Location: %c%c extension\n",
               'A' + telemetry.grid5, 'A' + telemetry.grid6);
        printf("  Altitude: %d meters\n", telemetry.altitude_meters);
        printf("  Temperature: %d°C\n", telemetry.temperature_c);
        printf("  Voltage: %.2fV\n", telemetry.voltage_v);
        printf("  Speed: %d knots\n", telemetry.speed_knots);
        printf("  GPS Valid: %s\n", telemetry.gps_valid ? "Yes" : "No");
    }
}
```

Encoding Architecture

Two-Stage Encoding Process

Basic Telemetry uses a two-stage encoding process to fit all fields into WSPR message components:

Stage 1: Callsign Encoding

```
// Encode Grid5, Grid6, and Altitude into callsign
BigNumber callsign_data = 0;
callsign_data = callsign_data * 24 + grid5;
callsign_data = callsign_data * 24 + grid6;
callsign_data = callsign_data * 1068 + altitude_index;
```

Stage 2: Grid+Power Encoding

```
// Encode remaining fields into grid and power
BigNumber grid_power_data = 0;
grid_power_data = grid_power_data * 90 + temperature_index;
grid_power_data = grid_power_data * 40 + voltage_index;
grid_power_data = grid_power_data * 42 + speed_index;
grid_power_data = grid_power_data * 2 + gps_valid;
grid_power_data = grid_power_data * 2 + telemetry_type;
```

Rollover Considerations

Advantages

- Measurements not strictly limited to defined ranges
- Handles sensor readings outside nominal ranges
- Continuous operation during extreme conditions

Disadvantages

- **Ambiguous decoding**: Multiple input values map to same encoded value
- **Requires context**: Additional information needed for correct interpretation
- **Data loss**: Precision lost due to quantization and rollover

Best Practices

```
// Implement bounds checking before encoding
float clampVoltage(float voltage, float min_v, float max_v) {
    if (voltage < min_v) return min_v;
    if (voltage > max_v) return max_v;
    return voltage;
}

// Use application-specific knowledge for decoding
int16_t decodeTemperatureWithContext(uint16_t encoded, int16_t expected_range) {
    int16_t base_temp = encoded - 50;

    // Apply context-based correction
    while (abs(base_temp - expected_range) > 45) {
        if (base_temp < expected_range) {
            base_temp += 90;  // Next rollover range
        } else {
            base_temp -= 90;  // Previous rollover range
        }
```

```
    }

    return base_temp;
}
```

Implementation-Specific Behavior

Example:

- **Temperature**: Uses onboard RP2040 temperature sensor
- **Voltage**: Samples during high-load TX conditions for worst-case readings
- **GPS Validity**: Always true (GPS lock required for transmission)
- **Rollover**: Not implemented - values are clamped to ranges

Voltage Range Restriction

```
// Traquito, for example, uses restricted voltage range
const float VOLTAGE_MIN = 3.0;
const float VOLTAGE_MAX = 4.95;

float clampVoltage(float voltage) {
    if (voltage < VOLTAGE_MIN) return VOLTAGE_MIN;
    if (voltage > VOLTAGE_MAX) return VOLTAGE_MAX;
    return voltage;
}
```

Behavior Differences

```
// Standard implementation with rollover
int16_t encodeTemperatureStandard(int16_t temp) {
    return (temp - (-50)) % 90;  // Rollover enabled
}

// Implementation with clamping
int16_t encodeTemperatureClamp(int16_t temp) {
    if (temp < -50) temp = -50;   // Clamp to minimum
    if (temp > 39) temp = 39;     // Clamp to maximum
    return temp - (-50);          // No rollover
}
```

Transmission Scheduling

Channel Selection

Basic Telemetry transmission timing is coordinated with the U4B ChannelMap system. The channel selection implies a starting minute for the callsign transmission. There are 5 possible starting minutes within each 10 minute interval.

```
// Check if current time slot is appropriate for Basic Telemetry
bool canTransmitBasicTelemetry(uint8_t current_minute, uint8_t channel) {
    return isBasicTelemetrySlot(current_minute, channel);
}
```

Integration with Channel Map

```
#include <WsprEncoded/ChannelMap.h>
#include <WsprEncoded/BasicTelemetry.h>

void scheduleBasicTelemetry() {
    uint8_t current_minute = getCurrentMinute();

    // Find appropriate channel for Basic Telemetry
    ChannelMap::ChannelInfo channel;
    if (ChannelMap::findBasicTelemetryChannel(20, current_minute, channel)) {
        // Encode and transmit
        BasicTelemetryData data = collectSensorData();
        WSPRMessage message;

        if (encodeBasicTelemetry(data, message)) {
            transmitWSPR(message, channel.frequency_hz);
        }
    }
}
```

Error Handling

Validation Functions

```
bool validateBasicTelemetry(const BasicTelemetryData& data) {
    // Check field ranges
    if (data.grid5 > 23 || data.grid6 > 23) return false;
    if (data.altitude_meters > 21340) return false;
    if (data.temperature_c < -50 || data.temperature_c > 39) return false;
    if (data.voltage_v < 2.0 || data.voltage_v > 3.95) return false;
    if (data.speed_knots > 82) return false;
    if (data.telemetry_type != 1) return false;

    return true;
}
```

Error Codes

```
enum class BasicTelemetryError {
    SUCCESS = 0,
    INVALID_GRID_VALUES,
    ALTITUDE_OUT_OF_RANGE,
    TEMPERATURE_OUT_OF_RANGE,
    VOLTAGE_OUT_OF_RANGE,
    SPEED_OUT_OF_RANGE,
    INVALID_TELEMETRY_TYPE,
    ENCODING_FAILED,
    DECODING_FAILED
};
```

Performance Characteristics

Encoding Efficiency

- **Total fields**: 8 telemetry values
- **Bit utilization**: ~48 of 50 available WSPR bits

Precision Trade-offs

Field	Original Precision	Encoded Precision	Efficiency
Altitude	1m	20m	95% bit utilization
Temperature	0.1°C	1°C	90% bit utilization
Voltage	0.001V	0.05V	98% bit utilization
Speed	0.1 knots	2 knots	95% bit utilization

ChannelMap Specification

Overview

The ChannelMap specification enables identification and association of Telemetry messages with their corresponding Regular Type 1 messages within repeating 10-minute transmission windows. This specification defines the data structures and identification mechanisms used to locate and correlate WSPR telemetry transmissions.

Core Concepts

Message Types

- **Regular Type 1 Messages**: Standard WSPR messages transmitted at the start of each 10-minute window
- **Telemetry Messages**: Encoded data transmissions that must be associated with their corresponding Regular messages

Channel Identification

Channels provide a systematic approach to locate Telemetry messages by specifying:

- Unique identifier encoding (`id13`)
- Transmission time slots
- Target frequencies

Data Structures

Channel Object

```
{
  "channel_number": 248,
```

```
  "id13": "12",
  "time_slot": 4,
  "frequency": 14095600,
  "band": "20m"
}
```

Telemetry Message Structure

```
{
  "callsign": "1X2XXX",
  "grid": "AA00",
  "power": 37,
  "frequency": 14095598,
  "timestamp": "2025-01-15T12:04:00Z",
  "id13_char1": "1",
  "id13_char3": "2"
}
```

Identification Methods

1. ID13 Encoding

The `id13` value uniquely identifies channels and is encoded into Telemetry message callsigns:

- **Format**: Two-character identifier (00-Q9)
- **Encoding**:
  - Character 1 → Callsign position 1
  - Character 2 → Callsign position 3
- **Purpose**: Differentiates Telemetry messages sharing frequency or time slots

Example

```
Channel 248: id13 = "12"
Blank callsign: _ _ _ _ _ _
After id13 encoding: 1 _ 2 _ _ _
After full encoding: 1 X 2 X X X
```

2. Time Slot Identification

- **Definition**: Specific minute within the 10-minute window when Telemetry is transmitted
- **Purpose**: Differentiates Telemetry messages sharing the same `id13` value
- **Range**: 0-9 minutes within each transmission window

3. Frequency Matching

Target Frequency

- Specified in Channel Map for each channel
- Used as baseline for transmission and reception

Frequency Fingerprinting, Frequency binning or other mechanisms

Due to receiver calibration issues, implement fingerprinting for accurate association:

1. **Locate Regular Message**: Find Regular Type 1 message for target callsign

2. **Extract Reported Frequency**: Use actual received frequency (not target frequency)
3. **Match Telemetry**: Search for Telemetry messages at the reported frequency
4. **Validate Association**: Confirm `id13` and time slot match

Implementation Notes

Receiver Calibration Challenges

Many WSPR receivers have poorly calibrated frequency references, leading to:

- Inaccurate frequency reports
- Difficulty in message association
- Need for fingerprinting techniques, frequency binning, frequency binning with rx frequency error correction, or possibly even no filtering based on frequency (can have no errors, if no balloon rx spots cause conflicts on channels solely differentiated by frequency.

Fingerprinting Algorithm

1. **Baseline Establishment**: Use Regular message reported frequency as reference
2. **Frequency Clustering**: Group Telemetry messages by similar frequency deviations
3. **Temporal Correlation**: Validate time slot alignment
4. **ID13 Verification**: Confirm identifier encoding matches channel specification

Performance Considerations

- Cache Channel Map data for frequent lookups
- Implement frequency tolerance ranges for matching
- Use time-based indexing for efficient searches
- Consider batch processing for large datasets

Channel Map Integration

This API integrates with the Channel Map system to provide:

- Channel number to frequency mapping
- Time slot scheduling information
- ID13 identifier assignments
- Band allocation details

Refer to the Channel Map Help documentation for detailed mapping information and scheduling specifics.

Extended Telemetry Specification

Overview

Extended Telemetry is an enhanced protocol that extends the Basic Telemetry scheme while maintaining full backward compatibility. It provides a flexible framework for transmitting structured telemetry data with improved encoding capabilities and extensible message types.

Key Features

- **16 message types** including user-defined and vendor-defined types
- **Up to 5 messages per 10-minute window** with flexible scheduling
- **Backward compatibility** with Basic Telemetry and Regular Type 1 messages
- **Enhanced encoding** supporting up to 29.180 bits (608,212,404 values) per field
- **Collision-free transmission** with sender identification
- **Extensible message structure** for future growth

Message Architecture

Message Structure

All Extended Telemetry messages consist of two parts:

1. **Header Fields** - Common structure across all message types
2. **Message Fields** - Type-specific data payload

Header Fields

Every Extended Telemetry message includes these header fields (listed in unpacking order):

Field Name	Type	Range	Step	Values	Description
`HdrTelemetryType`	Enum	0-1	1	2	Always 0 for Extended Telemetry
`HdrRESERVED`	Enum	0-3	1	4	Reserved for future use (must be 0)
`HdrType`	Enum	0-15	1	16	Message type identifier
`HdrSlot`	Enum	0-4	1	5	Time slot identifier

Header Field Details

HdrTelemetryType

- Must be set to `0` to identify Extended Telemetry messages
- Used to distinguish from other telemetry types

HdrRESERVED

- Must be set to `0b00`
- Reserved for future protocol extensions
- Receivers must ignore messages with non-zero values

HdrType

- Identifies the specific Extended Telemetry message type
- Default value: `0b0000`
- See Message Types for valid values

HdrSlot

- Identifies the sender and time slot (0-4)
- Maps to 2-minute intervals within a 10-minute window
- Default value: `0b00`

Message Types

Extended Telemetry supports 16 different message types:

HdrType	Type	Description
0	User-Defined	Custom structure for testing and experimentation
1-14	Enumerated	Standardized message types (to be defined)
15	Vendor-Defined	Vendor-specific structure and scheduling

Planned Enumerated Types (Examples)

HdrType	Type	Purpose
1	Basic Telemetry 2	Extended ranges and higher resolution
2	GPS Stats	GPS behavior and satellite information
3-14	TBD	Future standardized types

Time Slot Management

10-Minute Window Structure

Extended Telemetry operates within established 10-minute transmission windows, divided into 5 time slots:

Slot	Timing	Usage
0	Start minute	Primary slot
1	Start + 2 min	Secondary slot
2	Start + 4 min	Extended slot
3	Start + 6 min	Extended slot
4	Start + 8 min	Extended slot

Transmission Patterns

Extended Telemetry provides flexible transmission patterns:

Backward Compatible

```
Slot 0: Regular Type 1
Slot 1: Basic Telemetry
Slots 2-4: [Available for Extended Telemetry]
```

Extended Only

```
Slots 0-4: [Extended Telemetry in any combination]
```

Mixed Mode

```
Slot 0: Regular Type 1
Slot 1: Extended Telemetry (replacing Basic)
Slots 2-4: Extended Telemetry
```

Encoding Specification

Data Encoding

Extended Telemetry uses a unified encoding algorithm that:

- Supports field values up to 29.180 bits (608,212,404 values)
- Allows fields to span the entire WSPR message space
- Implements defined clamping and rounding behaviors

Packing Order

Fields are packed into the big number in **reverse order** from their definition:

1. Message fields (last defined → first defined)
2. Header fields (HdrSlot → HdrType → HdrRESERVED → HdrTelemetryType)

Value Processing

Clamping Behavior

- No rollover occurs
- All values are clamped to their defined ranges before encoding

Rounding Behavior

- Field values are rounded to the closest multiple of step size within range
- Rounding occurs during encoding process

Example Implementation

GPS Stats Message (Hypothetical)

```
Message Type: GPS Stats (HdrType = 2)

Field Definitions:
- SatsUSA: 0-128, step 4 (33 values, 5.044 bits)
- SatsChina: 0-128, step 4 (33 values, 5.044 bits)
- SatsRussia: 0-128, step 4 (33 values, 5.044 bits)
- SatsEU: 0-128, step 4 (33 values, 5.044 bits)
- SatsIndia: 0-128, step 4 (33 values, 5.044 bits)
- hdop: 0-10, step 2 (6 values, 2.585 bits)

Encoding Analysis:
- Available bits: 29.180
- Used bits: 27.807 (95.29%)
```

```
- Remaining bits: 1.373 (4.71%)
```

Packing Example

For GPS Stats message, packing order would be:

```
1. hdop (message field - last defined)
2. SatsIndia
3. SatsEU
4. SatsRussia
5. SatsChina
6. SatsUSA (message field - first defined)
7. HdrSlot (header field)
8. HdrType
9. HdrRESERVED
10. HdrTelemetryType (header field - first)
```

Integration Guidelines

Receiver Implementation

1. **Message Detection**: Check HdrTelemetryType = 0
2. **Version Compatibility**: Ignore messages with HdrRESERVED ≠ 0
3. **Type Handling**: Use HdrType to determine message structure
4. **Slot Management**: Use HdrSlot for sender identification

Sender Implementation

1. **Header Setup**: Always set HdrTelemetryType = 0, HdrRESERVED = 0
2. **Type Selection**: Choose appropriate HdrType for message content
3. **Slot Assignment**: Select HdrSlot based on transmission schedule
4. **Value Processing**: Apply clamping and rounding before encoding

Fingerprinting Logic

For websites that use a variety of “fingerprinting” algos, it is useful to be similar for deciding how to fingerprint when Basic and Extended Telemetry exists.

For telemetry detection in any 10-minute window:

1. Check slot 0 for both Regular and Extended messages
2. If only Regular found: use as reference frequency
3. If only Extended found: use as reference frequency
4. If both found: prefer Regular message as reference

Error Handling

Invalid Messages

Receivers should ignore messages with:

- HdrTelemetryType ≠ 0
- HdrRESERVED ≠ 0

- Unknown HdrType values (beyond implemented range)
- Invalid field values outside defined ranges

Backward Compatibility

Extended Telemetry maintains compatibility by:

- Using existing WSPR message format
- Preserving Regular Type 1 and Basic Telemetry timing
- Not interfering with existing transmission patterns

Future Extensions

The protocol supports future growth through:

- Extensible message types (add fields to existing types)
- HdrRESERVED field for protocol enhancements
- User-defined and vendor-defined message types
- Flexible transmission scheduling

WsprEncoded C++ Library

Overview

WsprEncoded is a header-only C++ library that implements telemetry encoding and decoding functionality for the WSPR (Weak Signal Propagation Reporter) protocol. This library enables developers to build trackers and other telemetry systems that can transmit sensor data through WSPR's minimal bandwidth constraints.

Features

Core Capabilities

- **Basic Telemetry**: Encode/decode common measurements (altitude, voltage, temperature, etc.)
- **Extended Telemetry**: Define and encode arbitrary telemetry fields
- **Channel Mapping**: Look up WSPR channel details by band and frequency

Design Principles

- **Memory Efficient**: No dynamic memory allocations (`malloc`, `new`, or STL containers)
- **Widely Compatible**: C++11 language standard for older compiler support
- **Header-Only**: Simple integration without separate compilation
- **Cross-Platform**: Works on Arduino, embedded systems, and desktop platforms

Installation

Arduino Library Manager

```
# Install via Arduino IDE Library Manager
# Search for "WsprEncoded" and click Install
```

CMake Integration

Option 1: FetchContent (Recommended)

```
include(FetchContent)
FetchContent_Declare(
  WsprEncoded
  GIT_REPOSITORY https://github.com/your-repo/WsprEncoded.git
  GIT_TAG        v1.0.0
)
FetchContent_MakeAvailable(WsprEncoded)

target_link_libraries(YourExecutable WsprEncoded)
```

Option 2: Git Submodules

```
# Add as submodule
git submodule add https://github.com/your-repo/WsprEncoded.git
third-party/WsprEncoded

# In CMakeLists.txt
add_subdirectory(third-party/WsprEncoded)
target_link_libraries(YourExecutable WsprEncoded)
```

Option 3: External Directory

```
# If WsprEncoded is in a sibling directory
add_subdirectory(../WsprEncoded WsprEncoded)
target_link_libraries(YourExecutable WsprEncoded)
```

API Reference

Core Headers

```
#include <WsprEncoded/TelemetryBasic.h>      // Basic telemetry types
#include <WsprEncoded/TelemetryExtended.h>   // User-defined telemetry
#include <WsprEncoded/ChannelMap.h>          // Channel mapping utilities
```

Basic Telemetry API

Common Measurement Types

```
namespace WsprEncoded {
    // Predefined measurement types
    struct AltitudeMeasurement {
        static constexpr uint16_t min_value = 0;
        static constexpr uint16_t max_value = 21000;  // meters
        static constexpr uint16_t resolution = 20;     // 20m steps
    };

    struct VoltageMeasurement {
```

```
        static constexpr uint16_t min_value = 0;
        static constexpr uint16_t max_value = 5000;    // millivolts
        static constexpr uint16_t resolution = 10;     // 10mV steps
    };

    struct TemperatureMeasurement {
        static constexpr int16_t min_value = -40;
        static constexpr int16_t max_value = 80;       // celsius
        static constexpr uint16_t resolution = 1;      // 1°C steps
    };
}
```

Encoding Functions

```
class BasicTelemetryEncoder {
public:
    // Encode single measurement
    template<typename MeasurementType>
    static bool encode(uint16_t value, uint8_t& encoded_index);

    // Encode multiple measurements into WSPR message
    static bool encodeToWSPR(
        const uint16_t* values,
        const uint8_t* measurement_types,
        uint8_t count,
        WSPRMessage& message
    );

    // Decode from WSPR message
    static bool decodeFromWSPR(
        const WSPRMessage& message,
        uint16_t* values,
        uint8_t* measurement_types,
        uint8_t& count
    );
};
```

Usage Example

```
#include <WsprEncoded/TelemetryBasic.h>

void encodeBasicTelemetry() {
    using namespace WsprEncoded;

    // Prepare measurements
    uint16_t altitude = 1200;  // meters
    uint16_t voltage = 3300;   // millivolts
    int16_t temperature = 25;  // celsius

    // Create encoder
    BasicTelemetryEncoder encoder;
    WSPRMessage message;

    // Encode measurements
    uint16_t values[] = {altitude, voltage, temperature};
    uint8_t types[] = {
        MeasurementType::ALTITUDE,
        MeasurementType::VOLTAGE,
        MeasurementType::TEMPERATURE
```

```
    };

    if (encoder.encodeToWSPR(values, types, 3, message)) {
        // Success - message ready for transmission
        printf("Encoded WSPR: %s %s %d\n",
               message.callsign, message.grid, message.power);
    }
}
```

Extended Telemetry API

Custom Field Definition

```
class ExtendedTelemetryEncoder {
public:
    // Define custom measurement field
    struct FieldDefinition {
        uint16_t min_value;
        uint16_t max_value;
        uint16_t resolution;
        uint8_t bit_count;
    };

    // Add field to encoder
    bool addField(const FieldDefinition& field);

    // Encode custom telemetry
    bool encode(const uint16_t* values, uint8_t count, WSPRMessage& message);

    // Decode custom telemetry
    bool decode(const WSPRMessage& message, uint16_t* values, uint8_t& count);
};
```

Custom Telemetry Example

```
#include <WsprEncoded/TelemetryExtended.h>

void encodeCustomTelemetry() {
    using namespace WsprEncoded;

    ExtendedTelemetryEncoder encoder;

    // Define custom pressure field (800-1200 hPa, 1 hPa resolution)
    ExtendedTelemetryEncoder::FieldDefinition pressure_field = {
        .min_value = 800,
        .max_value = 1200,
        .resolution = 1,
        .bit_count = 9  // 2^9 = 512 values (enough for 400 hPa range)
    };

    // Define custom humidity field (0-100%, 1% resolution)
    ExtendedTelemetryEncoder::FieldDefinition humidity_field = {
        .min_value = 0,
        .max_value = 100,
        .resolution = 1,
        .bit_count = 7  // 2^7 = 128 values (enough for 100% range)
    };

    // Add fields to encoder
```

```
    encoder.addField(pressure_field);
    encoder.addField(humidity_field);

    // Encode measurements
    uint16_t values[] = {1013, 65};  // 1013 hPa, 65% humidity
    WSPRMessage message;

    if (encoder.encode(values, 2, message)) {
        // Success - custom telemetry encoded
        printf("Custom telemetry encoded successfully\n");
    }
}
```

Channel Map API

Channel Information

```
class ChannelMap {
public:
    struct ChannelInfo {
        uint8_t channel_id;
        uint8_t start_minute;     // Minute within 2-hour window
        uint32_t frequency_hz;    // Exact frequency in Hz
        uint8_t band;            // Amateur radio band
    };

    // Get channel info by band and channel
    static bool getChannelInfo(uint8_t band, uint8_t channel, ChannelInfo& info);

    // Get all channels for a band
    static uint8_t getChannelsForBand(uint8_t band, ChannelInfo* channels, uint8_t
max_count);

    // Find optimal channel for current time
    static bool findOptimalChannel(uint8_t band, uint8_t current_minute,
ChannelInfo& info);
};
```

Channel Map Example

```
#include <WsprEncoded/ChannelMap.h>

void useChannelMap() {
    using namespace WsprEncoded;

    // Get 20m band, channel 13 information
    ChannelMap::ChannelInfo info;
    if (ChannelMap::getChannelInfo(20, 13, info)) {
        printf("Channel 13: Start minute %d, Frequency %lu Hz\n",
               info.start_minute, info.frequency_hz);
    }

    // Find optimal channel for current time (assuming minute 15)
    ChannelMap::ChannelInfo optimal;
    if (ChannelMap::findOptimalChannel(20, 15, optimal)) {
        printf("Optimal channel: %d at %lu Hz\n",
               optimal.channel_id, optimal.frequency_hz);
    }
}
```

Complete Integration Example

Arduino Tracker Implementation

```
#include <WsprEncoded/TelemetryBasic.h>
#include <WsprEncoded/ChannelMap.h>

class WSPRTracker {
private:
    WsprEncoded::BasicTelemetryEncoder encoder;
    WsprEncoded::ChannelMap channelMap;

public:
    bool transmitTelemetry(uint16_t altitude, uint16_t voltage, int16_t temp) {
        // Encode telemetry
        WsprEncoded::WSPRMessage message;
        uint16_t values[] = {altitude, voltage, temp};
        uint8_t types[] = {
            WsprEncoded::MeasurementType::ALTITUDE,
            WsprEncoded::MeasurementType::VOLTAGE,
            WsprEncoded::MeasurementType::TEMPERATURE
        };

        if (!encoder.encodeToWSPR(values, types, 3, message)) {
            return false;
        }


        // Find optimal transmission channel.
        // Usually a tracker will be configured to use a single channel.
        WsprEncoded::ChannelMap::ChannelInfo channel;
        uint8_t current_minute = getCurrentMinute();

        if (!channelMap.findOptimalChannel(20, current_minute, channel)) {
            return false;
        }

        // Transmit WSPR message
        return transmitWSPR(message, channel.frequency_hz);
    }

private:
    bool transmitWSPR(const WsprEncoded::WSPRMessage& msg, uint32_t freq) {
        // Platform-specific WSPR transmission implementation
        // ...
        return true;
    }

    uint8_t getCurrentMinute() {
        // Get current minute within 2-hour WSPR window
        // ...
        return 0;
    }
};
```

Desktop Application Integration

```
#include <WsprEncoded/TelemetryExtended.h>
#include <iostream>
```

```
int main() {
    using namespace WsprEncoded;

    // Create extended telemetry encoder for weather station
    ExtendedTelemetryEncoder encoder;

    // Define weather measurements
    ExtendedTelemetryEncoder::FieldDefinition pressure = {800, 1200, 1, 9};
    ExtendedTelemetryEncoder::FieldDefinition humidity = {0, 100, 1, 7};
    ExtendedTelemetryEncoder::FieldDefinition wind_speed = {0, 200, 1, 8};

    encoder.addField(pressure);
    encoder.addField(humidity);
    encoder.addField(wind_speed);

    // Simulate weather readings
    uint16_t readings[] = {1013, 65, 12};  // 1013 hPa, 65%, 12 km/h
    WSPRMessage message;

    if (encoder.encode(readings, 3, message)) {
        std::cout << "Weather telemetry encoded: "
                  << message.callsign << " " << message.grid << " " <<
message.power
                  << std::endl;
    }

    return 0;
}
```

Error Handling

Common Error Codes

```
namespace WsprEncoded {
    enum class ErrorCode {
        SUCCESS = 0,
        INVALID_RANGE,          // Value outside measurement range
        INSUFFICIENT_BITS,      // Not enough bits for encoding
        DECODE_FAILED,          // Decoding operation failed
        INVALID_CHANNEL,        // Channel not found
        MEMORY_FULL             // No more fields can be added
    };
}
```

Best Practices

- Always check return values from encoding/decoding functions
- Validate measurement values are within defined ranges
- Handle bit capacity limitations gracefully
- Use appropriate measurement resolutions to optimize data usage

Performance Notes

Memory Usage

- Header-only library adds minimal overhead
- No dynamic allocations - all memory usage is compile-time determined
- Typical memory footprint: < 1KB for basic telemetry, < 2KB for extended

Processing Speed

- Encoding/decoding operations are O(n) where n is number of measurements
- Optimized for embedded systems with limited CPU resources
- Typical encoding time: < 1ms on Arduino-class processors

Platform Support

Tested Platforms

- **Arduino**: Uno, Nano, ESP32, ESP8266
- **Desktop**: Windows, Linux, macOS
- **Embedded**: ARM Cortex-M, AVR, PIC32

Compiler Requirements

- **Minimum**: C++11 support
- **Recommended**: C++14 or later for better template support
- **Tested**: GCC 4.9+, Clang 3.5+, MSVC 2015+