# **Technical Analysis of TelemetryTx.ino: A Simulated Spacecraft Telemetry System**

## **1. Introduction**

This report provides a detailed technical analysis of the TelemetryTx.ino Arduino code, designed to function as a simulated spacecraft telemetry transmitter. The system integrates multiple sensors to gather environmental, inertial, magnetic, positioning, and power status data. This data is then formatted into packets adhering to the Consultative Committee for Space Data Systems (CCSDS) standards, specifically mimicking aspects of low-rate telemetry protocols used in space missions. Communication is established via a Long Range (LoRa) radio link, employing the SX1262 transceiver module controlled by the RadioLib library. To enhance data reliability over the potentially noisy LoRa channel, the system incorporates Reed-Solomon Forward Error Correction (RS-FEC) and a Cyclic Redundancy Check (CRC). An optional monochrome display provides local visualization of sensor readings and system status. The primary purpose of this system, as stated in its documentation, is educational, demonstrating fundamental concepts of spacecraft telemetry, data packetization, and robust communication techniques.

## **2. System Overview**

The core functionality of the TelemetryTx.ino code is to emulate the telemetry transmission subsystem of a spacecraft. It cyclically performs the following operations:

1. Acquires data from a suite of onboard sensors.
2. Processes and formats this data, along with system status information (like mission time and packet count), into a structured CCSDS Application Packet (AP).
3. Calculates a CRC-16 checksum for data integrity verification within the packet.
4. Applies Reed-Solomon Forward Error Correction encoding to the packet data to add redundancy.
5. Transmits the encoded packet over a LoRa radio link using an SX1262 transceiver.
6. Optionally displays sensor data and system status on a local monochrome display.

The system leverages established standards like CCSDS for packet structure and robust error correction techniques (RS-FEC, CRC) commonly employed in space communications to ensure reliable data transfer, albeit simulated over a terrestrial LoRa link rather than deep space networks.1 The selection of sensors provides a representative, though simplified, dataset comparable to what a spacecraft might collect (environmental conditions, orientation, position, power status).

## **3. Hardware Integration**

The system integrates several hardware components managed by an Arduino-compatible microcontroller platform. The specific pin mappings and board-level configurations are abstracted within the LoRaBoards.h header file, indicating dependence on a specific hardware target.

* **Microcontroller:** An Arduino-compatible microcontroller serves as the central processing unit, orchestrating sensor readings, data processing, communication protocols, and display updates. The exact model is determined by the definitions within LoRaBoards.h.
* **Sensors:**
  + **BME280:** An environmental sensor providing temperature, barometric pressure, and relative humidity.3 Altitude is derived from the pressure reading using a standard sea-level pressure reference (SEALEVELPRESSURE\_HPA). It communicates via the I2C interface (Wire.h) and is controlled by the Adafruit\_BME280 library.6
  + **QMC6310:** A 3-axis magnetometer measuring magnetic field strength along the X, Y, and Z axes.7 It features a wide dynamic range (±30 Gauss) and high resolution (down to 2 mGauss).7 The code derives a heading angle from the X and Y readings, applying a hardcoded magnetic declination correction. It communicates via I2C (Wire.h) using the custom SensorQMC6310.hpp library.9
  + **QMI8658:** A 6-axis Inertial Measurement Unit (IMU) combining a 3-axis accelerometer and a 3-axis gyroscope.11 It measures linear acceleration (up to ±16g) and angular velocity (up to ±2048 dps), and includes an embedded temperature sensor.12 Communication occurs via the SPI interface (SPI.h), managed by the SensorQMI8658.hpp library. The code configures specific ranges and output data rates for both the accelerometer and gyroscope.
  + **GPS Module:** A standard GPS module providing location (latitude, longitude, altitude), time (UTC date and time), and status information (satellites in view, Horizontal Dilution of Precision - HDOP). It communicates via a dedicated serial UART interface (SerialGPS) and data is parsed using the TinyGPS++.h library.14
  + **Power Management Unit (PMU):** An integrated PMU, accessed via a board-specific object (PMU, likely defined in LoRaBoards.h), monitors critical power parameters: battery voltage, estimated battery percentage, charging status, VBUS voltage (input voltage if present), and system voltage.
* **LoRa Transceiver:** An SX1262 LoRa radio module handles the wireless communication. It is controlled using the RadioLib library 16 and configured for operation in the 433 MHz ISM band. Connection to the microcontroller is via SPI, indicated by the Module constructor arguments (RADIO\_CS\_PIN, RADIO\_DIO1\_PIN, RADIO\_RST\_PIN, RADIO\_BUSY\_PIN).
* **Display Module:** An optional monochrome display, compatible with the U8g2lib library 19, provides local visualization of system data. Connection is likely via I2C, given the shared Wire.h dependency with I2C sensors.
* **Communication Interfaces:** The system utilizes multiple communication protocols:
  + **I2C (Wire.h):** Used for the BME280, QMC6310, and potentially the U8g2 display.
  + **SPI (SPI.h):** Used for the QMI8658 IMU and the SX1262 LoRa module.
  + **Serial (UART):** Serial is used for debugging output to a connected computer, while SerialGPS is dedicated to receiving NMEA data from the GPS module.

The following table summarizes the key hardware components:

| **Component** | **Type** | **Data Provided** | **Interface** | **Library/Driver** |
| --- | --- | --- | --- | --- |
| BME280 | Environmental Sensor | Temp, Pressure, Humidity, Altitude (derived) | I2C | Adafruit\_BME280.h |
| QMC6310 | Magnetometer | Magnetic Field (X, Y, Z), Heading (derived) | I2C | SensorQMC6310.hpp |
| QMI8658 | IMU (6-Axis) | Acceleration (X, Y, Z), Gyro Rate (X, Y, Z), Temp | SPI | SensorQMI8658.hpp |
| GPS | Positioning Module | Lat, Lon, Alt, Date, Time, Sats, HDOP | Serial | TinyGPS++.h |
| PMU | Power Monitor | Batt V, Batt %, Charging, VBUS V, System V | Internal | Board-specific (PMU->) |
| SX1262 | LoRa Transceiver | RF Communication | SPI | RadioLib.h |
| Display | Monochrome Display | Visual Output | I2C/SPI | U8g2lib.h |

The integration of diverse peripherals using multiple communication buses (I2C, SPI, two Serial ports) underscores a common challenge in embedded system design. Effective operation relies on careful pin allocation and resource management, which appears to be handled by the board-specific definitions in LoRaBoards.h. This abstraction layer is crucial for adapting the code to different hardware platforms but also means the system's portability is tied to the availability of such a board support package.

Furthermore, the choice of sensors (BME280, QMC6310, QMI8658) reflects components commonly found on hobbyist or educational development boards rather than space-qualified hardware.4 While their specifications are adequate for simulating telemetry data acquisition 5, they differ significantly from the robustness, radiation tolerance, and performance characteristics required for actual spaceflight instruments. This reinforces the system's role as an educational simulation.

## **4. Software Architecture and Control Flow**

The software architecture is organized around the standard Arduino setup() and loop() functions, utilizing several helper functions and external libraries to manage hardware and data processing tasks.

Initialization (setup() function):

The setup() function executes once at power-on or reset and is responsible for initializing all hardware components and software parameters. Key initialization steps include:

1. Initializing serial communication for debugging (Serial) and GPS data input (SerialGPS).
2. Executing board-specific hardware setup via setupBoards().
3. Initializing the U8g2 display (u8g2->begin()) and setting a default font.
4. Configuring and initializing the SX1262 LoRa radio using RadioLib. This involves setting the carrier frequency, bandwidth, spreading factor, coding rate, sync word, output power, current limit, and preamble length.17 Crucially, it disables the radio's hardware CRC (radio.setCRC(false)) and registers the setFlag function as the interrupt service routine (ISR) callback for the transmit completion event (radio.setPacketSentAction(setFlag)). An optional TCXO (Temperature Compensated Crystal Oscillator) is enabled if defined.
5. Initializing all sensors (bme.begin(), qmc.begin(), qmi.begin()). This includes critical checks for successful initialization; failure of any sensor triggers the displayFatalError function, halting the system.
6. Configuring sensor operational parameters, such as setting the measurement range and output data rate (ODR) for the QMI8658 accelerometer and gyroscope, and the operating mode, range, data rate, and oversampling for the QMC6310 magnetometer.
7. Performing an initial sensor data acquisition (updateSensorReadings()) and constructing the first telemetry packet (buildTelemetryPacket()).
8. Initiating the first LoRa transmission of the encoded packet using the non-blocking radio.startTransmit() function.

Main Execution Loop (loop() function):

The loop() function runs continuously after setup() completes. Its primary responsibilities are:

1. **Asynchronous Transmission Management:** It checks the transmittedFlag, a volatile boolean variable set by the setFlag ISR upon completion of a LoRa transmission. If the flag is true, it indicates the radio is ready for the next packet. The code then resets the flag, increments the packetCounter, updates all sensor readings (updateSensorReadings()), builds the next CCSDS telemetry packet (buildTelemetryPacket()), and initiates the next transmission using radio.startTransmit(). This creates a cycle where a new transmission starts only after the previous one has finished.
2. **Display Updates:** It calls updateDisplay() to manage the cycling between different information screens and drawDisplay() to render the content of the currently selected screen onto the U8g2 display.
3. **GPS Data Ingestion:** It continuously checks the SerialGPS port for incoming data using SerialGPS.available(). Any available bytes are immediately fed into the TinyGPS++ parser via gps.encode(SerialGPS.read()).
4. **Yielding:** A minimal delay(1) is included to prevent the loop from consuming 100% CPU and allow other low-priority tasks or interrupts to be processed.

**Key Function Analysis:**

* **buildTelemetryPacket():** This function is central to data formatting. It constructs the CCSDSPacket struct by:
  + Populating the 6-byte CCSDS primary header fields (version, type, secondary header flag, APID, sequence flags, sequence count, packet length) according to the defined constants and the current packetCounter. The construction adheres to the bitfield layout specified by the CCSDS standard.22
  + Filling the packet data field with the latest sensor readings obtained from global variables (updated by updateSensorReadings), the mission time (millis()), the predefined SPACECRAFT\_ID, and a dynamically generated message string containing the packet count. GPS data validity is checked before inclusion.
  + Calculating a CRC-16 checksum over the packet structure (up to the CRC field itself) using calculateCRC().
  + Performing Reed-Solomon encoding using the rs.Encode() method from the RS-FEC.h library. It copies the packet data into an encoding buffer and generates the parity bytes, storing the complete encoded message (data + parity) into the encodedData buffer for transmission. (Note: Potential issues with buffer sizes and the exact rs.Encode call are discussed in Section 7).
* **updateSensorReadings():** This function orchestrates the acquisition of data from most sensors. It checks if the sensorUpdateInterval (100ms) has elapsed since the last update. If so, it calls individual helper functions (updateBME280Data, updateQMC6310Data, updateQMI8658Data, updatePMUData) to read from each sensor subsystem. It also includes a loop to process any pending characters from the GPS serial port, ensuring GPS data is handled promptly.
* **updateData() Functions:** These helper functions interface directly with the sensor libraries (e.g., Adafruit\_BME280, SensorQMC6310, SensorQMI8658, PMU object) to retrieve raw data. They perform necessary unit conversions (e.g., BME280 pressure from Pascals to hectopascals) and derive values like altitude (from BME280 pressure) and heading (from QMC6310 X/Y readings, applying magnetic declination). Data readiness flags (isDataReady()) are checked for sensors that provide them (QMC6310, QMI8658) to ensure fresh data is acquired.
* **calculateCRC():** Implements the CRC-16-CCITT-FALSE algorithm. This specific variant uses the polynomial 0x1021 and an initial value of 0xFFFF, with no reflection of input bytes or the final CRC, and no final XOR operation.23 It iterates bitwise through the provided data buffer to compute the checksum, used for verifying data integrity at the receiver.
* **setFlag():** This is the Interrupt Service Routine (ISR) attached to the LoRa radio's DIO1 pin, configured via radio.setPacketSentAction(). When the SX1262 completes a transmission, it raises an interrupt on DIO1, triggering this function. The sole action of this ISR is to set the global volatile variable transmittedFlag to true, signaling the main loop that the transmission is complete.
* **displayFatalError():** A critical error handling routine called if sensor initialization fails in setup(). It prints a detailed error message to the Serial monitor, displays a prominent "SYSTEM FAILURE" message on the U8g2 display (wrapping text if necessary), and then enters an infinite loop. If LED\_BUILTIN is defined, it blinks the LED as a visual indicator of the fatal error state, effectively halting normal operation.

Library Dependencies and Roles:

The system relies heavily on external libraries to manage hardware interactions and complex algorithms:

* RadioLib.h: Provides high-level control over the SX1262 LoRa transceiver, handling SPI communication, LoRa modulation parameters, packet transmission, and interrupt management.16
* LoRaBoards.h: A custom, board-specific header defining pin assignments and potentially initializing hardware objects like PMU. Essential for hardware abstraction.
* Wire.h & SPI.h: Standard Arduino libraries for I2C and SPI communication, respectively.
* Adafruit\_Sensor.h & Adafruit\_BME280.h: Libraries for the BME280 environmental sensor.6
* SensorQMC6310.hpp & SensorQMI8658.hpp: Custom or vendor-supplied libraries for the QMC6310 magnetometer and QMI8658 IMU.
* U8g2lib.h: A versatile library for controlling monochrome displays, handling various protocols and display controllers.19
* math.h: Standard C library providing mathematical functions (atan2, pow, etc.).
* RS-FEC.h: Implements Reed-Solomon Forward Error Correction. Provides the RS::ReedSolomon class template and Encode method, likely based on implementations such as.26
* TinyGPS++.h: Parses NMEA sentences received from the GPS module.14
* stdio.h: Standard C library used for functions like sprintf for string formatting.

The software architecture demonstrates a mix of programming paradigms. The core transmission logic is event-driven, triggered by the hardware interrupt signaling packet completion via the transmittedFlag. Sensor updates (excluding GPS) are handled through periodic polling based on millis() and the sensorUpdateInterval. GPS data, arriving asynchronously via serial, is processed through continuous polling within the main loop and the sensor update function. This hybrid approach balances responsiveness to the completion of the time-consuming transmission task with regular sampling of sensor data and immediate handling of the serial GPS stream.

A significant aspect of the architecture is its reliance on external libraries. Complex operations like LoRa modulation/demodulation, NMEA parsing, Reed-Solomon encoding, and low-level sensor communication are encapsulated within these libraries. This simplifies the main application code (TelemetryTx.ino), making it primarily an integrator of these functionalities. However, it also means the system's overall behavior, performance, and correctness are heavily dependent on these external abstractions. Debugging issues related to timing, data corruption, or unexpected behavior might necessitate investigation within the library code itself.

The use of #pragma pack(push, 1) before the CCSDSPacket structure definition is a critical detail for ensuring protocol compliance. C/C++ compilers often insert padding bytes into structures to align members on word boundaries for performance. However, network protocols and data formats like CCSDS require precise byte layouts without padding. This pragma forces byte-alignment, ensuring the in-memory representation of the CCSDSPacket struct exactly matches the intended structure for transmission and interpretation by a compliant receiver.22

## **5. Data Acquisition Subsystem**

The data acquisition subsystem is responsible for reading data from all integrated sensors and the PMU.

Sensor Reading Logic:

Data acquisition for the BME280, QMC6310, QMI8658, and PMU is orchestrated by the updateSensorReadings() function. This function employs a non-blocking timer mechanism based on millis() and the sensorUpdateInterval constant, set to 100 milliseconds (10 Hz sampling rate). When the interval elapses, dedicated helper functions (updateBME280Data, updateQMC6310Data, updateQMI8658Data, updatePMUData) are called. These functions utilize the respective sensor libraries to perform the actual reads (e.g., bme.readTemperature(), qmc.readData(), qmi.getAccelerometer(), PMU->getBattVoltage()). Where available, sensor data-ready flags (qmc.isDataReady(), qmi.getDataReady()) are checked before reading to ensure data validity and freshness. Necessary post-processing, such as unit conversions (Pa to hPa for pressure) and derived value calculations (altitude from pressure, heading from magnetometer readings including declination correction), are performed within these helper functions.

GPS Data Parsing:

GPS data acquisition operates differently due to its serial NMEA stream nature. The TinyGPS++.h library is used for parsing.14 Bytes are read from the SerialGPS interface whenever available (SerialGPS.available() > 0) within both the main loop() and the updateSensorReadings() function. Each byte read is immediately passed to the gps.encode() method. This method internally processes the byte stream, identifies NMEA sentences (like GGA, RMC), and updates the library's internal objects representing location, date, time, altitude, speed, course, satellite count, and HDOP. The validity of these internal objects is checked using methods like gps.location.isValid() before the data is copied into the telemetry packet during the buildTelemetryPacket() process.15

Update Timing:

The primary sensor suite (BME, QMC, QMI, PMU) is sampled synchronously at a rate determined by sensorUpdateInterval (10 Hz). GPS data, however, is processed asynchronously as bytes arrive, with the effective update rate of the GPS information typically limited by the GPS module's output rate (commonly 1 Hz).

The 10 Hz sampling rate for the main sensors is relatively high compared to the potential transmission rate of the system, especially given the LoRa configuration (SF12, 125kHz BW) and the size of the encoded packet (255 bytes). The Time-on-Air for such a packet can easily exceed one second.27 Since sensor updates occur every 100ms, but a new packet is only built and transmitted *after* the previous lengthy transmission completes, multiple sensor readings (potentially 10 or more) will occur between packet transmissions. Only the single set of sensor values captured immediately before buildTelemetryPacket is called will be included in the outgoing telemetry. This implies that transient events or dynamics occurring faster than the packet transmission rate might be missed or aliased by the telemetry system, despite the higher sensor sampling rate.

Conversely, the frequent polling and processing of GPS data via gps.encode() ensures that the internal state of the TinyGPS++ object reflects the latest received NMEA information promptly. However, this GPS information is only incorporated into the tmPacket structure during the less frequent buildTelemetryPacket calls. The use of validity checks (isValid()) is crucial here; if the GPS loses its fix between packet transmissions, these checks prevent stale or invalid location/time data from being included in the telemetry, ensuring that default zero values are sent instead. This design efficiently handles the continuous GPS serial stream while synchronizing the *reporting* of GPS data with the slower telemetry transmission cycle.

## **6. Data Processing and Formatting**

Once sensor data is acquired, it undergoes significant processing and formatting before transmission, primarily involving structuring according to CCSDS standards, calculating a CRC, and applying Reed-Solomon encoding.

CCSDS Packet Structure:

The telemetry data is encapsulated within a CCSDSPacket structure, defined in C. The use of #pragma pack(push, 1) ensures that the structure members are packed contiguously in memory without padding bytes, matching the byte-oriented nature of the CCSDS standard. The packet consists of a mandatory Primary Header and a Packet Data Field (user data).

* **Primary Header (6 bytes):** This header follows the CCSDS Space Packet Protocol standard (CCSDS 133.0-B-1).22 It is composed of three 16-bit words:
  + version\_type\_apid: Contains the Packet Version Number (3 bits, set to 0b000), Packet Type (1 bit, set to 0b0 for Telemetry), Secondary Header Flag (1 bit, set to 0b1), and Application Process Identifier (APID) (11 bits, set to the example value 0x7FF).
  + sequence\_flags\_count: Contains the Sequence Flags (2 bits, set to 0b11 indicating an unsegmented packet) and the Packet Sequence Count (14 bits, derived from packetCounter modulo 16384).
  + packet\_length: A 16-bit field representing the length of the Packet Data Field (in bytes) minus one. It is calculated as sizeof(CCSDSPacket) - 7, which correctly reflects the standard's definition.22

The following table details the primary header fields as implemented in the code:

| **Field** | **Bits** | **Offset** | **Code Value(s)** | **Standard Meaning** | **Implementation Notes** |
| --- | --- | --- | --- | --- | --- |
| Packet Version Number | 3 | 0:0-2 | 0b000 | CCSDS Version 1 | Correctly implemented (CCSDS\_VERSION). |
| Packet Type | 1 | 0:3 | 0b0 | Telemetry Packet | Correctly implemented (CCSDS\_TYPE). |
| Secondary Header Flag | 1 | 0:4 | 0b1 | Secondary Header Present | Set to present, but no secondary header in struct. |
| APID | 11 | 0:5-15 | 0x7FF (2047) | Application Process ID | Example value used (CCSDS\_APID). |
| Sequence Flags | 2 | 2:0-1 | 0b11 | Unsegmented Data Packet | Hardcoded (CCSDS\_SEQ\_FLAGS), assumes no fragmentation. |
| Packet Sequence Count | 14 | 2:2-15 | packetCounter & 0x3FFF | Sequential Packet Counter | Correctly implemented, wraps at 16384. |
| Packet Data Length | 16 | 4:0-15 | sizeof(struct) - 7 | Packet Data Field Length - 1 | Correctly calculated based on struct size. |

* **Packet Data Field:** This field follows the primary header and contains the actual telemetry payload. In this implementation, it includes the spacecraft\_id string, mission\_time (from millis()), floating-point values for all sensor readings (BME280, QMC6310, QMI8658), GPS data (double for lat/lon, float for altitude, various integers for date/time/status), PMU data, a short message string, and finally, the 16-bit CRC checksum.

The hardcoding of the Secondary Header Flag to '1' (present) despite the lack of an actual secondary header in the CCSDSPacket struct represents a deviation from strict CCSDS compliance, likely a simplification for this specific implementation. Similarly, hardcoding the Sequence Flags to '11' (unsegmented) assumes packets will never need to be fragmented, limiting flexibility if different transmission constraints were imposed (e.g., smaller LoRa maximum payload size).

CRC Implementation:

Data integrity within the packet is initially checked using a 16-bit CRC. The calculateCRC() function implements the CRC-16-CCITT-FALSE algorithm, characterized by the polynomial 0x1021, initial value 0xFFFF, and specific processing steps (no data reflection, no final XOR).23 The CRC is computed over the entire CCSDSPacket structure, excluding the final two bytes reserved for the CRC itself (sizeof(CCSDSPacket) - 2). This checksum allows the receiver, after potentially correcting errors using RS-FEC, to verify if the reconstructed packet matches the original data sent by the transmitter.

Reed-Solomon Forward Error Correction (RS-FEC):

To enhance reliability over the LoRa link, the system employs RS(255, 223) Forward Error Correction. This is implemented using the RS-FEC.h library, instantiated as RS::ReedSolomon<rsMsgLen, ECC\_LENGTH> rs, where rsMsgLen is 223 and ECC\_LENGTH is 32.

* **Encoding Process:** Within buildTelemetryPacket, the code attempts to encode the telemetry packet. The line memcpy(encodedData, &tmPacket, rsMsgLen); copies the first rsMsgLen (223) bytes from the tmPacket structure into the encodedData buffer. Subsequently, rs.Encode(reinterpret\_cast<char\*>(&tmPacket), reinterpret\_cast<char\*>(encodedData)); is called. Based on the likely function signature void Encode(const char \*message, char \*encoded) 26, this call appears problematic. It seems to pass &tmPacket as the source message *again*, rather than using the data already copied into encodedData, and attempts to write the output back to encodedData. Furthermore, analysis reveals the CCSDSPacket structure size is approximately 142 bytes, significantly less than the expected rsMsgLen of 223 bytes. The memcpy operation thus only copies 142 bytes, leaving the subsequent portion of the intended 223-byte data block in encodedData uninitialized. The rs.Encode call then operates on this potentially incomplete or corrupted data block. This suggests a significant discrepancy or bug in the implementation's handling of the RS encoding relative to the packet structure size and the library's expected usage. Assuming a corrected implementation where exactly 223 bytes of data (potentially padded) are correctly passed to the encoder, the rs.Encode function would calculate 32 parity bytes based on the 223 data bytes. The resulting 255 bytes (223 data + 32 parity) would be stored in the encodedData buffer for transmission.
* **Error Correction Capability:** An RS(255, 223) code operating on 8-bit symbols (bytes) can correct up to t = (n-k)/2 = (255-223)/2 = 16 symbol errors (i.e., 16 incorrect bytes) anywhere within the 255-byte encoded block.1 If the locations of errors are known beforehand (termed "erasures"), the code can correct up to n-k = 32 erasures.1 This capability significantly enhances the probability of successfully recovering the original telemetry data even if the LoRa signal experiences corruption due to noise, interference, or fading, which is particularly valuable for long-range, low-power links.2 RS codes are known for their effectiveness against burst errors.2

The parameters and theoretical capabilities of the chosen RS code are summarized below:

| **Parameter** | **Value** | **Description** | **Source(s)** |
| --- | --- | --- | --- |
| Code Type | Reed-Solomon | Block-based error correcting code | 2 |
| Specification | RS(255, 223) | n=255 total symbols, k=223 data symbols | 2 |
| Symbol Size | 8 bits (byte) | Operates on bytes | 1 |
| Data Length (k) | 223 bytes | Assumed length of the original data block (rsMsgLen) | Code2 |
| Parity Length (n-k) | 32 bytes | Length of the added error correction code (ECC\_LENGTH) | Code2 |
| Codeword Length (n) | 255 bytes | Total length of the encoded block (encodedData buffer size) | Code2 |
| Error Correction (t) | 16 bytes | Can correct up to 16 symbol errors anywhere in the codeword | 1 |
| Erasure Correction | 32 bytes | Can correct up to 32 known erasures (error locations known) | 1 |
| Code Rate (k/n) | ~0.87 | Ratio of data bytes to total bytes (efficiency) | 1 |
| Redundancy | ~13% | Percentage of overhead added for error correction | 1 |

The identified discrepancy between the CCSDSPacket size (142 bytes) and the RS message length parameter (rsMsgLen = 223 bytes), coupled with the potentially incorrect rs.Encode call, imposes a critical constraint or indicates a flaw. If the RS library strictly requires 223 input bytes, the current implementation would fail or produce incorrect results. The system would need modification either by padding the CCSDSPacket to 223 bytes or by adjusting the RS parameters and encoding call to match the actual packet size.

The placement of the CRC calculation *before* RS encoding is a standard practice. The CRC verifies the integrity of the original 142-byte data structure, while the RS code protects the entire 255-byte transmission block (including the CRC and data) against channel errors. At the receiver, RS decoding is performed first. If successful, the receiver can then extract the original 142-byte packet (including the CRC field) and perform the CRC check for final validation. If RS decoding fails (detects uncorrectable errors), the packet is typically discarded, rendering the CRC check moot.

## **7. LoRa Communication Link**

The system utilizes the SX1262 LoRa transceiver, managed by the RadioLib library, for wireless data transmission. The configuration is optimized for long range and robustness.

Radio Configuration (SX1262 via RadioLib):

The following parameters are explicitly set in the setup() function 17:

* **Frequency (CARRIER\_FREQ):** 433.0 MHz. This falls within an ISM (Industrial, Scientific, and Medical) band commonly used for LoRa in regions like Europe (Region 1).
* **Transmit Power (TX\_POWER):** 20 dBm. This is a relatively high power level, aiming to maximize the transmission distance.
* **Bandwidth (BANDWIDTH):** 125.0 kHz. A common choice for LoRa, offering a balance between data rate and noise immunity. Lower bandwidths can improve sensitivity but require more stable oscillators (potentially TCXO) and reduce data rate.30
* **Spreading Factor (SPREADING\_FACTOR):** 12. This is the highest possible spreading factor for LoRa. It maximizes receiver sensitivity (allowing reception of signals significantly below the noise floor, down to -20 dB SNR) and communication range, but at the cost of the lowest data rate and the longest Time-on-Air (ToA).27 Each increment in SF provides approximately 2.5 dB improvement in link budget.30
* **Coding Rate (CODING\_RATE):** 6. This corresponds to a 4/6 coding rate in LoRa terminology, representing the amount of forward error correction applied *within* the LoRa physical layer itself (distinct from the application-level RS-FEC). Higher coding rates add more redundancy, improving robustness against errors but further reducing the effective data rate.30
* **Sync Word (SYNC\_WORD):** 0x35. This value acts as a network identifier. Using a custom value distinct from the standard LoRaWAN sync word (0x34) or the RadioLib default private sync word (0x12) ensures that the transmitter and intended receiver operate as a private network, ignoring packets from other LoRa systems.17
* **Preamble Length:** Set to 16 symbols using radio.setPreambleLength(16). A longer preamble helps the receiver synchronize with the incoming signal, especially in low SNR conditions associated with high spreading factors.
* **Hardware CRC:** Disabled via radio.setCRC(false). This is appropriate because data integrity is handled at the application layer by the CRC-16 calculation and the robust RS-FEC. Including the LoRa hardware CRC would add unnecessary overhead.
* **Current Limit:** Set to 140 mA using radio.setCurrentLimit(140). This is a safety feature to protect the radio module or power supply from excessive current draw during transmission.

The table below summarizes the LoRa parameters and their impact:

| **Parameter** | **Value Used** | **Rationale / Impact** |
| --- | --- | --- |
| Frequency | 433.0 MHz | Regional ISM band choice. |
| Bandwidth (BW) | 125.0 kHz | Standard LoRa BW; balances rate vs. noise immunity. Lower BW increases sensitivity but needs better clocks.30 |
| Spreading Factor (SF) | 12 | Maximizes range & sensitivity (~2.5dB gain/step 30). Lowest data rate, highest ToA, highest Tx power consumption. Best below-noise performance (-20dB SNR 27). |
| Coding Rate (CR) | 6 (4/6) | Increases robustness (LoRa PHY FEC). Reduces effective data rate further.30 |
| Output Power | 20 dBm | High power setting to maximize transmission distance. |
| Sync Word | 0x35 | Private network identifier, prevents interference with/from standard LoRaWAN (0x34) or other networks.17 |
| Preamble Length | 16 symbols | Longer preamble improves receiver lock-on reliability, especially with high SF / low SNR. |
| Hardware CRC | Disabled | Avoids redundancy; integrity handled by application CRC and RS-FEC. |

Transmission Process:

Packet transmission is initiated using the non-blocking function radio.startTransmit(encodedData, sizeof(encodedData)). This function commands the SX1262 module to start transmitting the contents of the encodedData buffer (which holds the 255-byte RS-encoded packet). The microcontroller is free to perform other tasks while the transmission is in progress. Upon completion of the transmission, the SX1262 signals this event by triggering an interrupt on its DIO1 pin. This interrupt invokes the setFlag ISR, which sets the transmittedFlag variable. The main loop() continuously polls this flag. When it finds transmittedFlag is true, it knows the previous transmission has finished and proceeds to prepare and initiate the next transmission. This asynchronous, interrupt-driven mechanism ensures efficient handling of the potentially long LoRa transmission times.

The specific combination of LoRa parameters chosen (SF12, 125kHz BW, CR 4/6) along with the large 255-byte payload results in a very significant Time-on-Air (ToA) for each packet, likely lasting several seconds. While maximizing theoretical range and robustness 27, this long ToA has several implications. It drastically reduces the potential data throughput, increases the energy consumed per packet transmission, and raises the probability of collisions if other devices are transmitting on the same frequency in the vicinity. This configuration is therefore well-suited for simulating a point-to-point, long-range link where data rate is not critical, but less practical for networked applications or scenarios requiring frequent updates or operating in spectrally congested environments.

Disabling the LoRa hardware CRC is a logical optimization. The combination of the application-level CRC-16 and the powerful RS(255, 223) code provides superior error detection and correction capabilities compared to the basic CRC built into the LoRa PHY. Relying on the application-level mechanisms simplifies the data flow and avoids redundant overhead.

The use of a custom sync word (0x35) is crucial for operating in shared ISM bands. It effectively isolates this communication link, preventing the transmitter from being received by unintended LoRa devices (e.g., LoRaWAN gateways using 0x34) and preventing the receiver (if implemented) from wasting resources trying to decode packets from other networks.17

## **8. User Interface Implementation**

The system includes an optional user interface implemented via a monochrome display controlled by the U8g2lib library.19

Display Initialization and Drawing:

The display is initialized in the setup() function using u8g2->begin(). The specific display controller and communication interface (I2C or SPI) are likely determined by the constructor provided within the LoRaBoards.h file. The drawDisplay() function, called repeatedly in the main loop, handles rendering content. It first clears the display's internal buffer (u8g2->clearBuffer()), sets the desired font (u8g2->setFont()), draws text and graphics corresponding to the currently selected screen (currentScreen), and finally transfers the buffer contents to the physical display (u8g2->sendBuffer()).

Information Screens:

A currentScreen variable controls which set of information is displayed. The drawDisplay() function uses a switch statement to select the content based on the value of currentScreen (ranging from 0 to 10). The available screens show:

* Screen 0: QMI8658 Accelerometer (X, Y, Z) and Temperature.
* Screen 1: QMI8658 Gyroscope (X, Y, Z).
* Screen 2: BME280 Environmental Data (Temp, Humidity, Pressure, Altitude).
* Screen 3: QMC6310 Magnetometer (X, Y, Z) and calculated Heading.
* Screen 4: QMI8658 Accelerometer (Detailed, 4 decimal places).
* Screen 5: QMI8658 Gyroscope (Detailed, 4 decimal places) and Temperature.
* Screen 6: GPS Date and Time (via drawGPSDateTimeScreen).
* Screen 7: GPS Location Data (Lat, Lon, Alt, Sats, HDOP) (via drawGPSLocationScreen).
* Screen 8: LoRa Transmission Parameters (Freq, Power, SF, BW, Last Packet Msg).
* Screen 9: PMU Status 1 (Battery Voltage, Percentage, Charging Status).
* Screen 10: PMU Status 2 (VBUS Voltage, System Voltage).

Formatted output is achieved using u8g2->printf(). Dedicated drawing functions (drawGPSDateTimeScreen, drawGPSLocationScreen) handle the layout for the GPS information screens, including checks for data validity (gps.date.isValid(), gps.location.isValid()) and displaying "Waiting for Fix" messages if necessary.

Screen Cycling:

The updateDisplay() function manages the automatic cycling through the different screens. It checks if defaultScreenDelay (2000 ms) has passed since the last switch using millis() and lastScreenSwitch. If the delay has elapsed, it increments currentScreen and updates lastScreenSwitch. The cycling logic uses the modulo operator: currentScreen = (currentScreen + 1) % 10.

This display serves as a valuable local diagnostic tool, providing real-time feedback on sensor readings, GPS status, LoRa configuration, and power status during development, testing, and operation. It allows for immediate verification that subsystems are functioning as expected. However, it is not essential for the core telemetry transmission function, which operates independently.

A potential issue exists in the screen cycling logic within updateDisplay(). The code defines 11 screens (indexed 0 through 10) in the drawDisplay switch statement. However, the cycling logic uses currentScreen = (currentScreen + 1) % 10. The result of the modulo 10 operation will always be between 0 and 9, inclusive. Consequently, currentScreen will never reach the value 10, and the content defined in case 10: (PMU Status 2) will never be displayed. This appears to be an off-by-one error; the modulo should likely be % 11 to cycle through all 11 defined screens. Additionally, a constant gpsScreenDelay is defined but remains unused in the updateDisplay function, suggesting an incomplete feature related to variable screen dwell times.

## **9. Error Handling and System Reliability**

The system incorporates several mechanisms to handle errors and enhance the reliability of data transmission, focusing on initialization and link robustness.

Initialization Checks:

The setup() function performs critical checks immediately after attempting to initialize each sensor (bme.begin(), qmc.begin(), qmi.begin()). If any of these initialization routines fail (return false), the displayFatalError() function is called. This function provides clear feedback by printing a message to the Serial monitor and displaying a "SYSTEM FAILURE" message on the U8g2 display. Crucially, it then halts system execution by entering an infinite loop (potentially blinking the built-in LED). This prevents the system from proceeding with potentially faulty or missing sensor data, ensuring a baseline level of hardware readiness before operation commences.

Runtime Data Validity Checks:

During normal operation, checks are performed to ensure the validity of certain data before it's included in the telemetry packet:

* **GPS Data:** Before populating GPS fields in buildTelemetryPacket(), the code explicitly checks gps.location.isValid(), gps.date.isValid(), and gps.time.isValid().15 If the GPS data is deemed invalid (e.g., loss of satellite fix), default values (typically 0) are inserted into the packet instead of potentially erroneous or stale data.
* **Sensor Readiness:** The updateQMC6310Data and updateQMI8658Data functions check isDataReady() flags provided by their respective sensor libraries before attempting to read new data. This helps ensure that readings correspond to complete measurement cycles.

Communication Reliability Enhancement:

Multiple layers contribute to the reliability of the LoRa communication link:

* **Reed-Solomon FEC:** The application-level RS(255, 223) encoding is the primary mechanism for combating channel errors. By adding 32 parity bytes to every 223 bytes of data, it allows the receiver to correct up to 16 byte errors within the transmitted 255-byte block.1 This significantly increases the likelihood of successful data recovery even in the presence of substantial noise or interference, which is characteristic of long-range radio links.2
* **LoRa Parameters:** The choice of LoRa parameters (SF12, CR 4/6) inherently maximizes link robustness. SF12 provides the highest possible receiver sensitivity, enabling communication well below the noise floor (down to -20 dB SNR) 27, while the 4/6 coding rate adds further error correction capability at the physical layer.30
* **Application Layer CRC:** The CRC-16 checksum calculated by calculateCRC() provides an end-to-end integrity check. After the receiver performs RS decoding, it can recalculate the CRC on the recovered data payload and compare it to the transmitted CRC value. A match provides high confidence that the recovered data is identical to the data originally sent, prior to RS encoding.

While the error handling for initialization failures is robust (halting the system) and the data link reliability is significantly enhanced through RS-FEC and careful LoRa parameter selection, the system exhibits limited runtime error handling. There are no apparent mechanisms to detect or recover from potential sensor failures *after* successful initialization (e.g., a sensor stops responding or returns invalid data). Similarly, the LoRa transmission process lacks explicit error handling; the code assumes radio.startTransmit() succeeds and that the transmittedFlag ISR will eventually be triggered. There are no timeouts or recovery strategies implemented if a transmission fails or hangs. The system's reliability strategy prioritizes surviving initial setup and combating channel noise, but assumes continuous correct operation of hardware components and successful completion of transmissions thereafter.

The combined use of CRC-16 and RS(255, 223) represents a layered approach to data integrity, common in critical communication systems. RS-FEC actively corrects errors introduced by the communication channel, while the CRC serves as a final verification step to ensure the integrity of the *corrected* payload. This provides a high degree of confidence in the received data, assuming the number of errors does not exceed the correction capability of the RS code.

## **10. Overall Analysis and Recommendations**

Synthesis of Findings:

The TelemetryTx.ino code successfully implements a simulated spacecraft telemetry transmitter. It effectively integrates a comprehensive suite of sensors, formats the acquired data into CCSDS-compliant packets, and transmits them over a LoRa radio link. The system demonstrates a strong emphasis on communication reliability through the implementation of RS(255, 223) FEC, application-layer CRC, and the selection of robust LoRa parameters (SF12, CR 4/6). The inclusion of a diagnostic display further enhances its utility as an educational tool.

**Design Choices Discussion:**

* **Strengths:**
  + Modular software design (separate functions for sensor updates, packet building, display).
  + Adherence to standards (CCSDS packet structure).
  + Robust communication link design (RS-FEC, CRC, high SF/CR LoRa parameters).
  + Comprehensive sensor data acquisition (environmental, inertial, magnetic, GPS, power).
  + Useful local diagnostics via the optional display.
  + Clear initialization checks for critical sensors.
* **Weaknesses:**
  + Potential critical bug in RS-FEC implementation due to mismatch between packet size (142 bytes) and expected RS message length (223 bytes), and potentially incorrect rs.Encode usage.
  + Rigid packet structure constrained by the (potentially flawed) RS implementation.
  + High sensor sampling rate (10 Hz) relative to very low transmission rate (due to SF12 and large packet), leading to potential aliasing or loss of transient data in telemetry.
  + Simplified CCSDS implementation (hardcoded flags, no actual secondary header).
  + Limited runtime error handling (no checks for sensor failure during operation, no transmission timeouts).
  + Potential bug in display screen cycling logic (screen 10 likely unreachable).
  + Hardcoded magnetic declination limits portability.

**Potential Limitations:**

* **Scalability:** The extremely long Time-on-Air associated with SF12 and the 255-byte encoded packet makes this configuration unsuitable for networks with multiple nodes due to high collision probability and channel occupancy. It is primarily suited for point-to-point links.
* **Power Consumption:** The combination of high transmit power (20 dBm) and very long ToA results in high energy consumption per transmitted packet, making it less ideal for long-term battery-powered operation without significant optimization.
* **Throughput:** The effective data rate is extremely low due to the choice of SF12, CR 4/6, and the overhead from RS-FEC.
* **Real-time Capability:** The long latency between sensor readings being captured and their eventual transmission makes the system unsuitable for applications requiring near real-time data.

**Suggestions for Improvement/Further Development:**

1. **Correct RS-FEC Implementation:** Critically review and correct the RS encoding process. Either pad the CCSDSPacket structure to 223 bytes or configure the RS encoder (if possible) to handle the actual 142-byte packet size (potentially requiring different RS parameters like shortening the code). Ensure the rs.Encode function call is used correctly according to the library's API.26
2. **Enhance CCSDS Compliance:** Implement dynamic setting of Sequence Flags if data segmentation might be needed under different LoRa configurations (e.g., lower SF resulting in smaller LoRa MTU). If a secondary header is flagged, actually include one (e.g., for high-resolution timestamps).
3. **Add Runtime Error Handling:** Implement timeouts for LoRa transmissions and checks for sensor read failures during operation, potentially setting status flags within the telemetry packet.
4. **Optimize Transmission Strategy:** Consider adaptive data rates or adjusting LoRa parameters based on measured link quality (e.g., SNR) if two-way communication is feasible, although this adds complexity. For power-constrained applications, reduce transmit power, transmit less frequently, or use lower spreading factors when possible.
5. **Fix Display Cycling:** Correct the modulo operation in updateDisplay to % 11 to allow cycling through all defined screens. Implement the gpsScreenDelay logic if variable dwell times are desired.
6. **Improve Portability:** Make the magnetic declination configurable or dynamically calculated based on GPS location if possible.
7. **Data Filtering/Averaging:** Given the discrepancy between sensor sampling rate and transmission rate, consider implementing filtering or averaging of sensor data between transmissions rather than just sending the latest sample, potentially providing a more representative picture of the conditions over the transmission interval.

## **11. Conclusion**

The TelemetryTx.ino system provides a functional demonstration of key concepts involved in spacecraft telemetry transmission. It successfully integrates multiple sensors, formats data according to the CCSDS standard, and employs robust techniques (LoRa with high SF/CR, RS-FEC, CRC) to simulate reliable long-range communication. Its strengths lie in its comprehensive sensor suite, adherence to relevant standards for packetization, and strong focus on data link reliability, making it a valuable educational tool.

However, the analysis reveals significant limitations and potential flaws, most critically in the implementation of the Reed-Solomon encoding relative to the defined packet structure. Furthermore, the chosen LoRa parameters lead to extremely low throughput and high Time-on-Air, limiting its applicability beyond point-to-point, low-rate simulation. The error handling, while robust at initialization, lacks comprehensive runtime checks. Addressing the identified RS-FEC implementation issue and considering the suggested improvements would enhance its correctness, flexibility, and robustness, moving it closer to a production-ready embedded system while retaining its educational value.

#### Works cited

1. Exploring communications technology: Reed-Solomon codes and error correction continued | OpenLearn - Open University, accessed April 11, 2025, <https://www.open.edu/openlearn/digital-computing/exploring-communications-technology/content-section-2.3>
2. reed-solomon codes, accessed April 11, 2025, <https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html>
3. www.seeedstudio.com, accessed April 11, 2025, <https://www.seeedstudio.com/blog/2019/11/01/getting-started-with-bme280-humidity-pressure-and-temperature-sensor/#:~:text=The%20BME280%20is%20an%20integrated,stability%20and%20high%20EMC%20robustness.>
4. Getting started with BME280 - Humidity, Pressure ... - Seeed Studio, accessed April 11, 2025, <https://www.seeedstudio.com/blog/2019/11/01/getting-started-with-bme280-humidity-pressure-and-temperature-sensor/>
5. Humidity Sensor BME280 - Bosch Sensortec, accessed April 11, 2025, <https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/>
6. BME280 Datasheet - Bosch Sensortec, accessed April 11, 2025, <https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf>
7. www.qstcorp.com, accessed April 11, 2025, <https://www.qstcorp.com/upload/pdf/202202/%EF%BC%88%E5%B7%B2%E4%BC%A0%EF%BC%8913-52-17%20QMC6310%20Datasheet%20Rev.C(1).pdf>
8. CoreElectronics/CE-PiicoDev-Magnetometer-QMC6310 - GitHub, accessed April 11, 2025, <https://github.com/CoreElectronics/CE-PiicoDev-Magnetometer-QMC6310>
9. PiicoDev Magnetometer QMC6310 - Guide for Raspberry Pi - Tutorial Australia, accessed April 11, 2025, <https://core-electronics.com.au/guides/piicodev-magnetometer-qmc6310-guide-for-raspberry-pi/>
10. PiicoDev Magnetometer QMC6310 - Guide for MicroBit - Tutorial Australia - Core Electronics, accessed April 11, 2025, <https://core-electronics.com.au/guides/piicodev-magnetometer-qmc6310-guide-for-micro-bit/>
11. Arduino library provides a convenient interface for working with the QMI8658C inertial measurement unit (IMU) on Arduino platforms - GitHub, accessed April 11, 2025, <https://github.com/ALICHOUCHENE/Qmi8658c>
12. CIC QMI8658 Breakout - Six-Axis (Gyro + Accelerometer) IMU ..., accessed April 11, 2025, <https://xpulabs.github.io/products/amnos/sensor/cic_sen0002_qmi8658.html>
13. CIC QMI8658 Breakout - Six-Axis IMU Sensor Breakout - AnalogLamb, accessed April 11, 2025, <https://www.analoglamb.com/shop/sensors/imu/cic-qmi8658-breakout/>
14. TinyGPSPlus - Arduino Documentation, accessed April 11, 2025, <https://docs.arduino.cc/libraries/tinygpsplus/>
15. TinyGPS++ - Arduiniana, accessed April 11, 2025, <https://arduiniana.org/libraries/tinygpsplus/>
16. LoRa , who is she? Where can I find her - XIAO - Seeed Studio Forum, accessed April 11, 2025, <https://forum.seeedstudio.com/t/lora-who-is-she-where-can-i-find-her/291710>
17. RadioLib: SX1262 Class Reference - GitHub Pages, accessed April 11, 2025, <https://jgromes.github.io/RadioLib/class_s_x1262.html>
18. RadioLib/examples/SX126x/SX126x\_Settings/SX126x\_Settings.ino at master · jgromes/RadioLib - GitHub, accessed April 11, 2025, <https://github.com/jgromes/RadioLib/blob/master/examples/SX126x/SX126x_Settings/SX126x_Settings.ino>
19. u8g2reference · olikraus/u8g2 Wiki - GitHub, accessed April 11, 2025, <https://github.com/olikraus/u8g2/wiki/u8g2reference>
20. U8g2 - Arduino Library List, accessed April 11, 2025, <https://www.arduinolibraries.info/libraries/u8g2>
21. U8g2 - Arduino Reference, accessed April 11, 2025, <https://reference.arduino.cc/reference/en/libraries/u8g2/>
22. CCSDS — CCSDSPy documentation, accessed April 11, 2025, <https://docs.ccsdspy.org/en/latest/user-guide/ccsds.html>
23. CRC-CCITT -- 16-bit, accessed April 11, 2025, <https://srecord.sourceforge.net/crc16-ccitt.html>
24. CRC-16/CCITT-FALSE - GitHub Gist, accessed April 11, 2025, <https://gist.github.com/tijnkooijmans/10981093>
25. CRC-16/CCITT function - File Exchange - MATLAB Central - MathWorks, accessed April 11, 2025, <https://www.mathworks.com/matlabcentral/fileexchange/127803-crc-16-ccitt-function/?s_tid=LandingPageTabfx>
26. mersinvald/Reed-Solomon: Reed Solomon BCH encoder ... - GitHub, accessed April 11, 2025, <https://github.com/mersinvald/Reed-Solomon>
27. What is LoRa ? - Tutorials - Arduino Forum, accessed April 11, 2025, <https://forum.arduino.cc/t/what-is-lora/595381>
28. CCSDS Telecommand and Telemetry Format Packet Standard, accessed April 11, 2025, <https://jastoolbox.sandia.gov/topic/communication-specification/jas-packets/ccsds-telecommand-and-telemetry-format-packet-standard/>
29. Linear Feedback Shift Registers for the Uninitiated, Part XVI: Reed-Solomon Error Correction - Jason Sachs, accessed April 11, 2025, <https://www.embeddedrelated.com/showarticle/1182.php>
30. FAQ | Semtech, accessed April 11, 2025, <https://www.semtech.com/design-support/faq/faq-lora>
31. SX12XX-LoRa/What is LoRa.md at master - GitHub, accessed April 11, 2025, <https://github.com/StuartsProjects/SX12XX-LoRa/blob/master/What%20is%20LoRa.md>
32. At 4.8 Kbit/s does LORA and FSK have the same sensitivity - Reddit, accessed April 11, 2025, <https://www.reddit.com/r/Lora/comments/1g5zy2b/at_48kbits_does_lora_and_fsk_have_the_same/>