Firmware Development for an Open-Source Small Satellite Amateur Radio Transceiver

Ricardo J. Saborio Borbon, E. Glenn Lightsey,
Sterling Peet, and Aaron J. McDaniel
Georgia Institute of Technology
Atlanta, GA US

Abstract

The communications subsystem is a vital component of every space mission. However, the necessary hardware and infrastructure often consume a significant portion of the allotted budget for a project. This poses a problem for university teams developing small satellites with limited funds. Open-source projects like Earth imagery company Planet's OpenLST integrated hardware transceiver have attempted to solve this issue. While the OpenLST project addresses the hardware cost issue, it does not provide an affordable solution for the infrastructure problem. For this paper, a series of firmware modifications were completed for the OpenLST transceiver to allow for compatibility with amateur packet radio protocols. By implementing well-known protocols like AX.25, it is possible to leverage the existing infrastructure of amateur radio to reduce costs. The paper outlines the key differences between the existing protocol and AX.25, how these were addressed, and the overall structure of the final firmware modifications.

1. Introduction

Small satellites have become an attractive solution for teams looking for an affordable testbed for mission concepts and technology demonstrations. However, space-grade components come at an unavoidable premium that comprises a significant portion of a mission's cost. This is particularly true in the case of the communications subsystem which, in addition to the hardware costs, requires a significant additional investment to develop the necessary ground station infrastructure.

Earth-imaging company Planet has attempted to address the small satellite communications issue by releasing an open-source version of their flight-proven UHF radio (Klofas, 2018). The product, dubbed the OpenLST, is an integrated hardware transceiver made purely of inexpensive, off-the-shelf components. Nevertheless, the OpenLST uses Planet's proprietary communications protocol and requires a dedicated ground station using an OpenLST receiver. Therefore, this product only partially solves the small satellite communications issue.

The second half of the solution is provided by amateur packet radio. If a small satellite radio were to use an amateur packet radio protocol like AX.25, it could leverage the existing amateur radio stations worldwide

Corresponding Author: Ricardo J. Saborio Borbon – ricardo.j.saborio@gmail.com
Publication History: Submitted – 07/24/21; Revision Accepted – 12/15/21; Published – 02/08/22

and bypass the requirement for dedicated ground station infrastructure. Thus, a full solution to the small satellite communications problem would combine off-the-shelf flight-proven hardware, like OpenLST, with an amateur packet radio protocol like AX.25. This article describes a full solution to the small satellite communications problem that was derived at Georgia Tech's Space Systems Design Laboratory by modifying the firmware on Planet’s OpenLST to allow for AX.25 compatibility.

2. Hardware, Firmware, and Protocols Overview

This section outlines the main features of the OpenLST hardware, as well as the key characteristics of the OpenLST and AX.25 protocols in terms of their protocol specifications and frame formatting. The logic behind the flow of information in each protocol is also addressed.

2.1. OpenLST Integrated Hardware Transceiver

The OpenLST is an integrated hardware radio based off Texas Instruments’ CC1110 Low-Power RF Transceiver chip. This integrated circuit (IC) includes an Intel 8051 MCU core that allows for onboard data processing and provides a range of peripherals including ADCs, GPIO pins, timers, and UART ports. In addition, the chip features an onboard packet engine that handles the detection and parsing of incoming messages, as long as the message format adheres to the CC1110 standards. The hardware interfaces consist of two UART ports, three debug GPIO pins, and one SMA port.

The OpenLST release also features a Python ground station toolbox to interface with the radio. This includes custom commands to downlink telemetry from the OpenLST board, relay information between radios, and configure the transceiver itself. More importantly, the toolbox includes the functionality necessary for OTA (over-the-air) reprogramming of the firmware.

2.1.1. Protocol Specifications

The protocol specifications of the OpenLST are divided into the specifications for the physical connection with the host computer and those for the radio frequency (RF) link with other radios. The interface with the host computer consists of two UART connections configured with eight data bits, no parity, one high stop bit, one low start bit, and flow control enabled. Data is sent in little-endian order at a baud rate of 115200 baud.

In the case of the RF interface, the default configuration of the OpenLST uses 2-FSK modulation with a carrier frequency of 437 MHz and a deviation of 3.71 kHz to transmit and receive data. The bits are NRZ(L) encoded and are sent MSb first (except for the protocol header bytes) at a rate of 7416 baud. In addition, data whitening and FEC encoding is used to evenly distribute the power of the transmitted bitstream.

2.1.2. Frame Formatting

The frame formatting of the OpenLST is divided into two separate frame structures: one for the host computer interface (referred to as the OpenLST protocol) and one for the RF interface (referred to as the CC1110 protocol). The OpenLST protocol frame format consists of a variable length packet with a fixed-length eight-byte header. The frame structure decomposition is shown in Figure 1. The frame is transmitted LSB first.

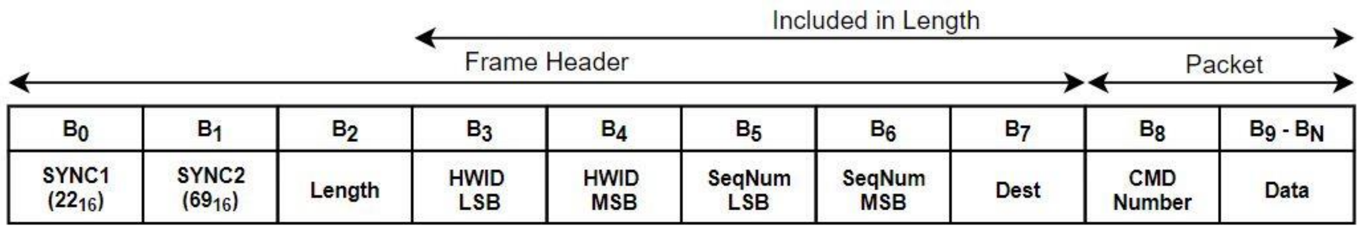


Figure 1. OpenLST protocol frame structure.

Frame synchronization is achieved with the two-byte sequence 6922_{16} given by B0 and B1. A non-inclusive length byte, given by B2, stores the number of bytes that come after B0-B2. A 16-bit HWID is split into B3 and B4. Each OpenLST board has a unique HWID that is used to address the packet to a specific device. A 16-bit sequence number, split into B5 and B6, serves as an identification number used to map command replies to a particular packet. The destination field, given by B7, is used to distinguish between packets addressed to the CC1110 and those addressed to the host computer. The remaining data field varies in length and contains the packet data to be relayed between the radios. The first byte in this segment, or B8, will always correspond to the command number.

In the case of the CC1110 protocol, a frame with a variable length packet, a ten-byte header, and a four-byte footer is implemented. The frame structure is intended to be compatible with the CC1110 packet engine to allow for automation of the data reception process. The frame header features a four-byte preamble, and the footer features a two-byte CRC for data integrity checks. The frame is transmitted LSB first, except for the SYNC byte sequence which is transmitted MSB first. Header bits are transmitted LSb first, but the bits in B9-BN are transmitted MSb first. The decomposition of the CC1110 frame structure is shown in Figure 2.

The preamble bytes required for clock synchronization consist of the AA_{16} sequence repeated four times over B0-B3. Byte synchronization is achieved by using a four-byte SYNC word consisting of the two-byte sequence $D391_{16}$ repeated twice using B4-B7. A non-inclusive length byte is stored in B8, which includes the number of bytes that follow B0-B8. A series of miscellaneous flags used for message forwarding out of the UART ports are stored in B9.

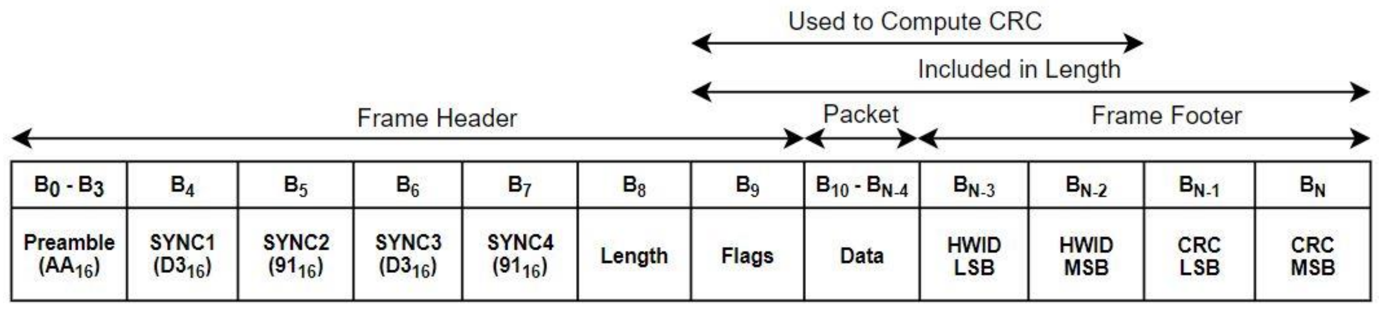
The packet itself is stored in B10-B(N-4), where N denotes the total number of bytes in the frame. The 16-bit HWID of the OpenLST is stored in BN-3 and BN-2, whereas the 16-bit CRC-16 is split between BN-1 and BN.

2.1.3. Data Flow

The flow of information in the OpenLST can be divided into two segments: one focusing on the high-level data flow between radios and host computers, and one focusing on the packing/unpacking of packet data during transmission/reception. The former can be described in terms of a network consisting of four nodes as shown in Figure 3.

The network shown consists of two nodes acting as the local device and two nodes acting as the remote device. Nodes 1 and 4 correspond to the host computers, whereas Nodes 2 and 3 correspond to OpenLST boards with distinct HWIDs. Connections to Nodes 1 and 4 consist of UART connections, while connections between Nodes 2 and 3 consist of an RF link. The flow of information in the network is controlled via the HWID and destination fields of the OpenLST protocol. The HWID will determine whether the message is sent to the local or remote OpenLST board and the destination field will indicate if the message should be processed by the CC1110 MCU or the host computer, allowing for direct communication with each node in the network.

The segment of the data flow involving the packing and unpacking of information can be described using the block diagram included in Figure 4. The logic



B_0 - B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10} - B_{N-4}	B_{N-3}	B_{N-2}	B_{N-1}	B_N
Preamble (AA_{16})	SYNC1 ($D3_{16}$)	SYNC2 (91_{16})	SYNC3 ($D3_{16}$)	SYNC4 (91_{16})	Length	Flags	Data	HWID LSB	HWID MSB	CRC LSB	CRC MSB

Figure 2. CC1110 protocol frame structure.

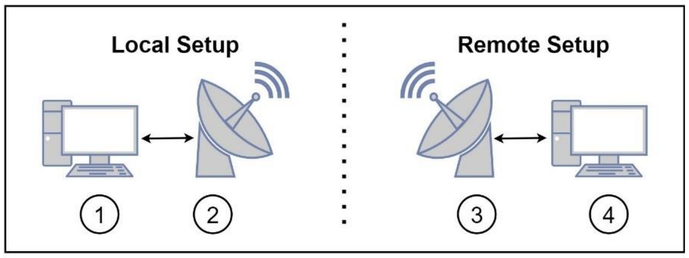


Figure 3. Visualization of the network between two OpenLST radios.

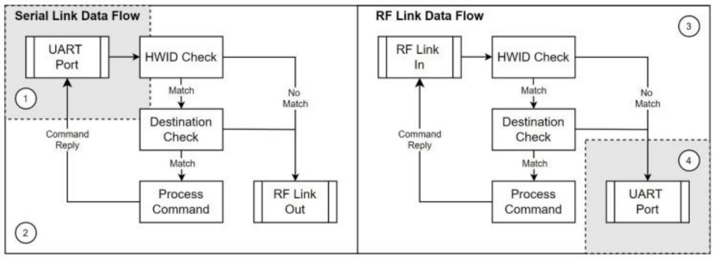


Figure 4. Data flow visualization for an OpenLST radio.

behind this flow of information is governed by the following conditions:

1. The OpenLST will only respond to messages addressed to its own HWID. If there is a HWID mismatch, outgoing messages will be forwarded out of the RF link and incoming messages out of the serial link.
2. The CC1110 MCU will only process messages with a destination field addressed to the OpenLST board. If there is a destination mismatch, incoming messages will be forwarded out of the UART port and outgoing messages out of the RF link. A destination field sequence of 01_{16} is used for the CC1110 MCU and a sequence of 11_{16} is used for the host computer.
3. Messages addressed to the CC1110 MCU will always generate a reply. The replies to a command will be sent out of the same medium it was received on (i.e., serial or RF link).

The previously mentioned network nodes have been included in Figure 4 to illustrate their role in the data flow. Note that for the above logic to take place, it is necessary to convert messages to and from the CC1110 and OpenLST protocols. The data rearrangement during the packing and unpacking process is depicted in Figure 5.

2.2. AX.25 Protocol

The AX.25 protocol is an HDLC-derived protocol (ISO, 2002) developed by the amateur radio community to provide a standard for amateur packet radio communications (Beech et al., 1998). This article will focus on the protocol specifications and frame structure of AX.25 and will assume that the network portion of the protocol has been implemented in the host machine. This will allow for the OpenLST to act as a TNC whose only purpose is to relay AX.25 frames.

2.2.1. Protocol Specifications

This article presents the protocol specifications of the AX.25 protocol within the context of a small satellite mission licensed to transmit in the UHF amateur radio band. The physical interface with the host computer consists of a UART connection, where data is transmitted at 57600 baud using eight data bits, one stop bit, no parity, and no flow control. These parameters are based off the KISS TNC and the satellite's mission requirements.

The OTA interface consists of a 2-FSK modulated signal with a carrier frequency of 437.175MHz and a deviation of 3.2kHZ. The bitstream uses NRZ(I) encoding and is sent LSb first, except for the 16-bit FCS (discussed later), at a rate of 9600 baud. In addition, G3RUH scrambling is used to evenly distribute the power of the signal during transmission (Miller, 1995). Lastly, the bitstream has to be bit stuffed to avoid specific bit sequences from appearing in the data. These requirements are governed by the HDLC protocol specification, and they have the largest impact on the overall modifications to the OpenLST. As a result, the specifics of NRZ(I) encoding, bit stuffing, and G3RUH scrambling are presented.

Traditionally, binary streams are encoded using the NRZ(L) convention, where "1" bits are defined with a logic high and "0" bits with a logic low. However, AX.25 is often paired with the NRZ(I) convention that defines bits in terms of logic level transitions. The exact definition depends on the implementation but, in HDLC, a 0 bit is defined as a transition from one logic level to another, whereas a 1 bit is defined as a constant logic level or no transition.

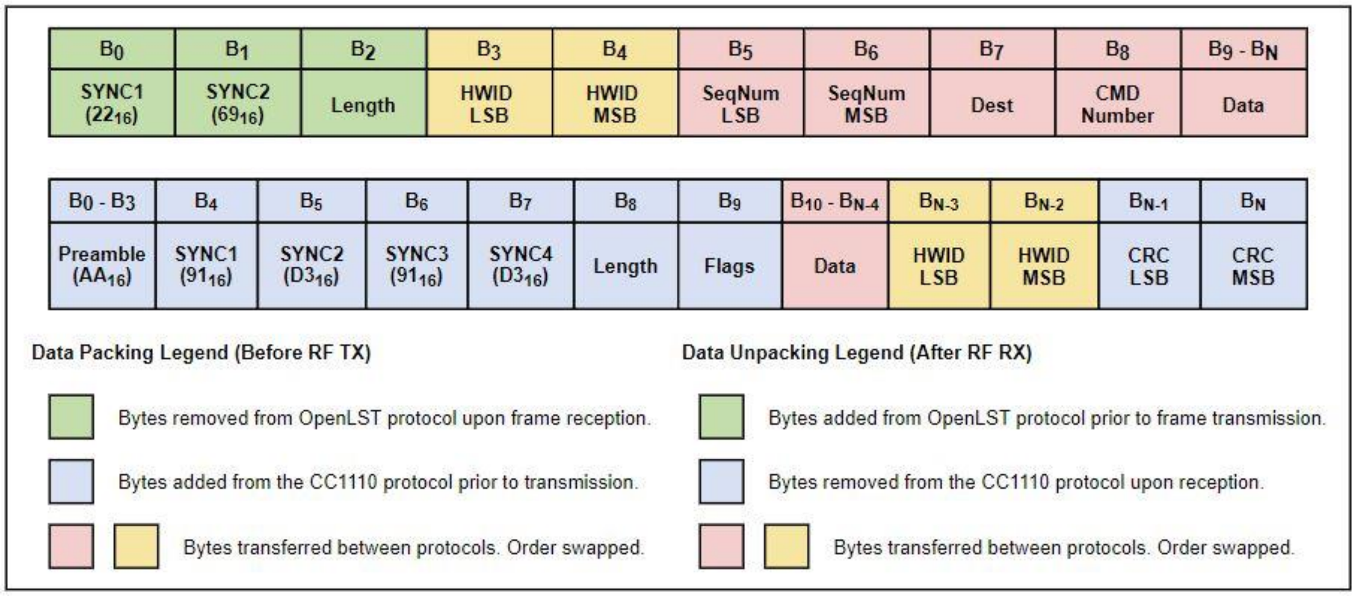

B0
B1
B2
B3
B4
B5
B6
B7
B8
B9 - BN
SYNC1 (22₁₆)
SYNC2 (69₁₆)
Length
HWID LSB
HWID MSB
SeqNum LSB
SeqNum MSB
Dest
CMD Number
Data
B0 - B3
B4
B5
B6
B7
B8
B9
B10 - BN-4
BN-3
BN-2
BN-1
BN
Preamble (AA₁₆)
SYNC1 (91₁₆)
SYNC2 (D3₁₆)
SYNC3 (91₁₆)
SYNC4 (D3₁₆)
Length
Flags
Data
HWID LSB
HWID MSB
CRC LSB
CRC MSB
Data Packing Legend (Before RF TX)
Bytes removed from OpenLST protocol upon frame reception.
Bytes added from the CC1110 protocol prior to transmission.
Bytes transferred between protocols. Order swapped.
Data Unpacking Legend (After RF RX)
Bytes added from OpenLST protocol prior to frame transmission.
Bytes removed from the CC1110 protocol upon reception.
Bytes transferred between protocols. Order swapped.

Figure 5. Breakdown of data re-arrangement during packing and unpacking of OpenLST protocol messages.

Frame synchronization in AX.25 is achieved by using a repeated byte sequence as both header and footer flags for each frame. This flag byte, known as the HDLC flag, is given by the sequence $7E_{16}$ or 01111110_2 (note the long run of six consecutive 1 bits). Since this sequence denotes the start and end of each AX.25 frame, it is crucial that the sequence does not appear anywhere else in the frame. This is the main purpose of bit stuffing in HDLC-based protocols. For every five consecutive 1 bits in the outgoing bit stream, a 0 bit will be added (or stuffed) to the data. For incoming streams, for every five consecutive 1 bits, a 0 bit will be ignored. The stuffed bit also forces transitions to occur due to the NRZ(I) convention, which helps prevent clock skew by eliminating long runs of 1 bits.

Data randomization is also necessary to guarantee an even distribution of power during transmission and to guarantee consistent logic level transitions required for clock synchronization. The G3RUH scrambling scheme, adapted from Steve Goode's (K9NG) 17-bit LFSR, is often used with the AX.25 protocol for this purpose (Miller, 1995). The method consists of a 17-bit shift register with XOR gates (or taps) on b0, b11, and b16 of the shift register. Each outgoing bit is XOR-ed with the taps, added to the first element of the shift register, and then sent out the RF link. Bits in the shift register are left-shifted by one with each new bit and the oldest bit in the register is removed. The exact same procedure can be repeated to unscramble the bit-stream.

2.2.2. Frame Formatting

The frame formatting is divided into separate conventions for the host computer connection and for the RF link. The host computer interface consists of a KISS TNC that adheres to a frame structure known as the KISS protocol (Chepponis et al., 1998). This consists of a variable length packet with a two-byte header and a single-byte footer. The KISS protocol is intended to encapsulate other frame structures and allows for compatibility regardless of the underlying frame structure. The logic behind the KISS TNC revolves around four special characters or bytes: FEND ($C0_{16}$), FESC (DB_{16}), TFEND (DC_{16}), and TFESC (DD_{16}). The TNC itself must follow the conditions below during message transmission and reception:

1. A single FEND on either side will delimit a KISS frame. Two FENDs in a row should not be considered an empty frame and reception of a FEND marks the end of the frame.
2. A single byte following the starting FEND indicates the port (upper nibble) and command (lower nibble).

3. During packing, a FEND will be replaced with a FESC followed by a TFEND, and a FESC will be replaced with a FESC followed by a TFESC.
4. During unpacking, a FESC followed by a TFEND will be replaced with a FEND, and a FESC followed by a TFESC will be replaced with a FESC.

Just as with bit stuffing, the above logic is intended to prevent the frame delimiter flags from appearing in the data being transmitted. Note that the KISS protocol does not necessarily define a frame structure, but it rather defines a set of rules for transporting other pre-existing frame structures. A simple example of the packing and unpacking process performed by a KISS TNC is included in Figure 6.

The frame formatting for the OTA interface consists of the AX.25 protocol UI frame. The UI frame is delimited by at least one HDLC flag on either end, which are intended for byte synchronization. The actual number of flags included varies between implementations. The frame contains a variable length header and packet data, as well as a 16-bit CRC (also called FCS in the context of AX.25) footer. The frame is transmitted LSB first, and bytes are transmitted LSb first, with the exception of the FCS which is transmitted MSB first with its respective bytes transmitted MSb first (Finnegan, 2014). A breakdown of the AX.25 UI frame structure is shown in Figure 7.

The header of an AX.25 frame consists of a variable length address field and a control field. The first seven bytes of the former (B0 to B6 in the diagram) consist of the amateur radio callsign of the packet destination and the following seven bytes correspond to the packet source callsign (B7 to B13). There is an option to append additional callsigns

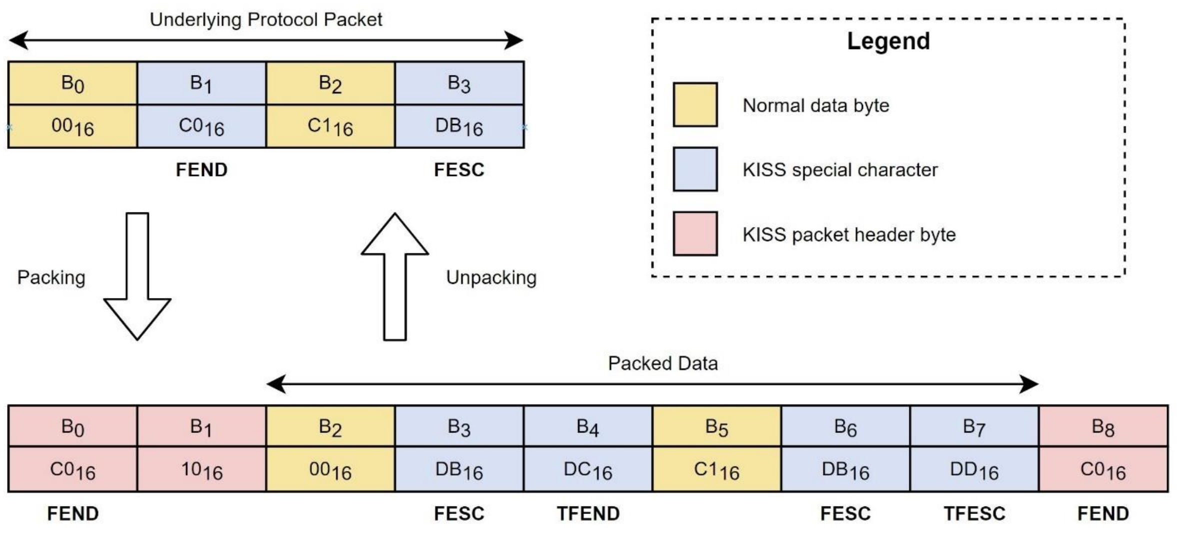


Figure 6. Sample KISS protocol packing and unpacking sequence.

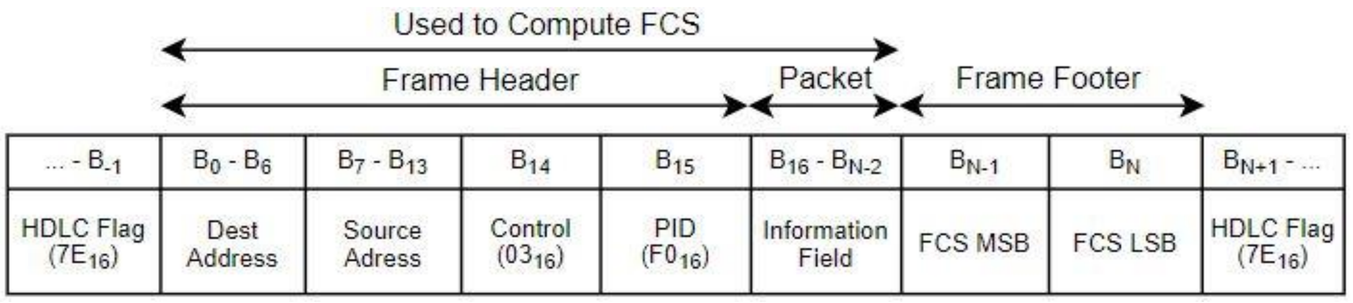


Figure 7. AX.25 UI frame structure.

associated with the repeating stations that the packet went through during transmission. However, for the context of this article, this option will be ignored. The callsigns consist of six uppercase alpha numeric ASCII characters, that are left shifted by one bit, and a single byte used for message forwarding inside an AX.25 network. An SSID nibble is stored in b1-b4 of this last byte to distinguish between stations using the same callsign. The control and PID bytes are fixed to the sequence $F003_{16}$ in UI frames and the packet data of the frame itself is stored in the variable length information field. The last two bytes in the frame, BN-1 and BN, contain the 16-bit FCS computed using the CRC16-CCITT convention.

2.2.3. Data Flow

The data flow in an AX.25 transceiver is significantly simpler than that of the OpenLST since there is no way of directly addressing messages to the transceivers. The KISS TNC is responsible for moving an AX.25 frame from the host computer to the transmitter. The transmitter then unpacks the message, performs necessary encoding/scrambling, appends the HDLC flags, and relays the information out of the RF link. This process is shown in Figure 8. Note that the opposite of the process described occurs during message reception.

2.3. OpenLST Firmware Logic and Structure

The firmware of the OpenLST is structured around a series of configurable interrupts used to monitor incoming and outgoing data. Two dedicated interrupts are responsible for monitoring each UART port individually and relay OpenLST protocol frames between the host computer and MCU. An additional RF link ISR monitors a carrier sense flag, that indicates when an incoming RF signal surpasses a set power threshold, and an EOM flag, which is set by the packet engine. These are used to signal the start and end of RF transmissions to the main program. A main loop periodically checks the flags associated with new incoming or outgoing messages and executes the associated subroutines when a new message is available.

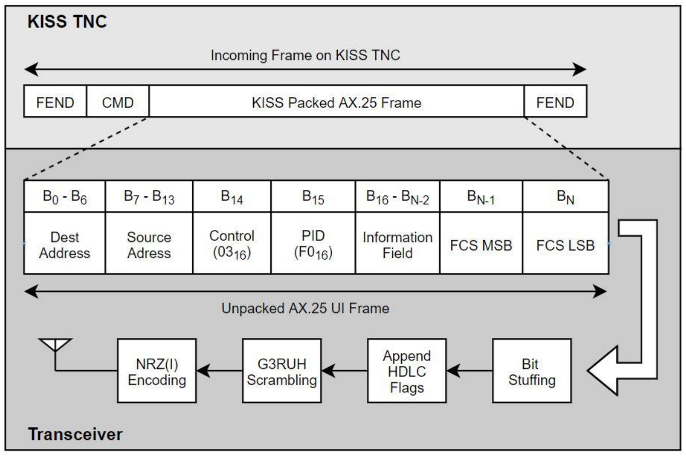


Figure 8. Data flow in an AX.25 transceiver.

These subroutines are responsible for the packing and unpacking of messages as described in Figure 5.

The OpenLST also leverages on the packet handling engine and DMA controller of the CC1110. The engine permanently runs in the background looking for the preamble and SYNC bytes of the CC1110 protocol in the demodulator output. Upon finding a message, a signal is sent to the DMA controller to move the data bytes to the receive buffer. This occurs as a routine independent from the main program. The DMA controller is also responsible for moving bytes from the transmit buffer to the packet engine, which appends the preamble and SYNC bytes prior to sending the message to the modulator. As can be seen, the modulator input and output are never accessible to the main program. The block diagram in Figure 9 depicts a visual representation of the OpenLST firmware structure and logic.

3. Necessary Firmware Modifications

The modifications in this study strived to maintain as much of the original OpenLST functionality as possible to conserve the solutions that the product already introduced. More specifically, the modifications maintain compatibility with the existing Python toolbox and OpenLST protocol commands, both of which have space heritage.

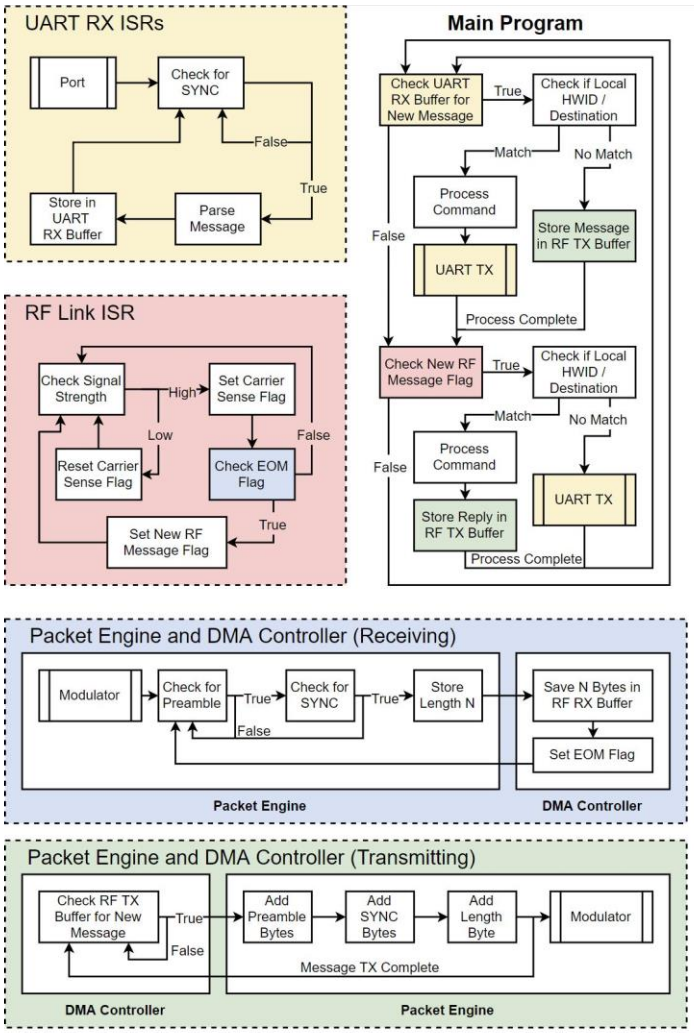


Figure 9. OpenLST firmware logic. ISRs have been surrounded by dashed boxes. Processes associated with UART ports are shown in yellow, with the RF ISR in red, with RF transmission in green, and with RF reception in blue.

3.1. Protocol Specifications

The most significant changes to the OpenLST firmware occurred in the protocol specifications. While the UART interfaces only required changes to the baud rate and flow control settings, the OTA interface required major restructuring. The transceiver configurations had to be changed to remove the FEC encoding and data whitening that came with the CC1110 protocol, as well as to change the carrier frequency, deviation, and data rate settings. The scrambling methods had to be replaced with G3RUH scrambling and bit stuffing. However, the CC1110 does not include configuration settings for such techniques. Moreover, the NRZ(L) encoding had to be replaced with NRZ(I), but the CC1110 is incompatible with said convention. In addition, the bit order of the incoming and outgoing data had to be reversed to replace the MSb first convention introduced by the CC1110 modulator with the LSb first convention used by AX.25.

3.2. Frame Formatting

The host computer interface must be compatible with both the KISS and OpenLST protocols. The KISS protocol compatibility is required to allow for the OpenLST to be interfaced with a regular AX.25 modem, whereas the OpenLST protocol compatibility is needed to use Planet's ground station and reprogramming software. To maintain the command functionality of the OpenLST protocol, the interface was modified to accept KISS-packed OpenLST protocol commands. The host computer interface was also redefined to output responses to each protocol using the appropriate format.

On the other hand, the OTA interface was only required to be compatible with the AX.25 protocol due to licensing requirements. Therefore, the CC1110 protocol was removed from the OpenLST for normal operations. However, some compatibility with the CC1110 protocol was maintained to allow for OTA reprogramming of the board. In addition, a method for packing OpenLST protocol commands in an AX.25 UI frame was implemented to maintain the OpenLST command functionality.

3.3. Firmware Logic

A key component of the firmware logic that needed to be maintained was the network functionality associated with the OpenLST protocol commands. A method was developed to address AX.25 messages to the OpenLST MCU and not just the host computer. Likewise, the modified firmware logic incorporated a method of addressing specific UART ports, as well as a method for distinguishing between AX.25 messages and AX.25-packed OpenLST protocol commands.

However, the largest modification required in the firmware logic involved the restructuring of the transmission and reception process. Since the packet engine is incompatible with the AX.25 encoding and scrambling requirements, the CC1110 is uncapable of recognizing NRZ(I) encoded messages and thus preamble and SYNC word detection would never occur. Therefore, the incoming transmissions had to be extracted directly from the modulator output. This required the bypassing of both the DMA controller and the packet engine. Likewise, the transmission process required additional steps to encode and scramble the outgoing messages prior to handing them to the DMA controller and packet engine.

4. Implementation of Firmware Modifications

4.1. UART Interface Transmission/Reception Modifications

The host computer interface logic was modified to allow for compatibility with both KISS-packed messages and the OpenLST protocol. This was achieved by replacing the UART RX ISR and UART TX functions (shown in Figure 9) with the KISS TNC RX ISR and KISS TNC TX functions, respectively. Both processes act as a dual KISS modem and OpenLST parser capable of discerning between protocols. Each UART port has a dedicated KISS TNC for transmission and reception of messages. A general overview of their respective logic is shown in Figure 10.

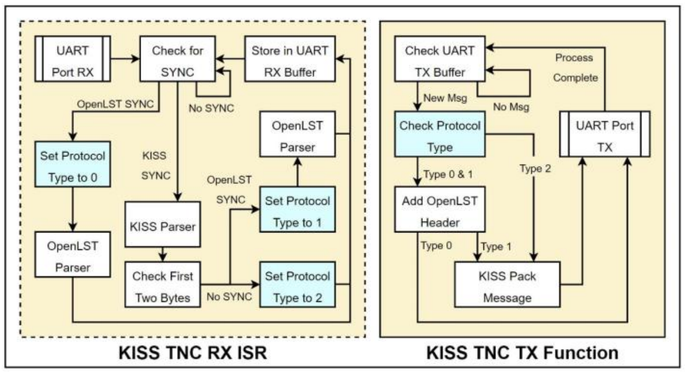


Figure 10. Firmware logic for KISS TNCs.

The parsing and packing process of both routines hinge on a global protocol type identifier (marked in cyan) used to identify the underlying protocol and packing of a message. Protocol Type 0 refers to an OpenLST protocol frame, Type 1 to a KISS-packed OpenLST protocol frame, and Type 2 to a KISS-packed AX.25 frame. This variable is shared by all processes and is useful for determining if a message should be treated as an OpenLST protocol command.

The KISS TNC RX ISR works by continuously inspecting the incoming byte stream from the UART port and looking for the known SYNC words of the KISS and OpenLST protocols. These bytes are unique and thus can be used to determine between Type 0 and Types 1 or 2 protocols. In the case of a KISS-packed message, there is an additional check to determine if an OpenLST protocol command is stored within the packet. After completion of the parsing process, the KISS TNC RX ISR will store the message bytes in the UART RX buffer for the main program to access. For Type 2 messages, these will correspond to a complete AX.25 UI frame. In the case of Type 0 or 1 messages, the stored bytes will be B3-BN (see Figure 1) of the OpenLST command.

The KISS TNC TX function performs the reverse of the process outlined above. The routine looks at the UART TX buffer for outgoing messages from the main program and performs the necessary packing based on the value of the protocol type identifier. Upon completion, the packed message is sent out of the UART port.

4.2. Protocol Modifications

The protocol formatting issue was addressed by introducing the protocol type identifier. However, this variable only exists locally and there is no default method in place to relay this information through the OTA interface. Therefore, the default structure of the AX.25 UI frame was modified to introduce a set of flags specifying the type of message that is contained in the frame. These flags are also used to route the message to the appropriate UART port and to determine if the UART output should be KISS packed. The information is stored in the SSID nibble of the destination callsign, where individual bits in the SSID are used as flags for specifying the message format. This is possible given the assumption that the destination callsign will be unique.

The flags in the SSID nibble are driven by the protocol type identifier of the source device. The first bit of the nibble, or b1, is set high if the frame contains an AX.25 message (i.e., protocol Type 2). The destination UART port is given by b2, which is set high for UART1 and low for UART0. Lastly, b3 is set high whenever the output to the host computer should be KISS packed (i.e., protocol Type 1 or 2). These flags are automatically set by the OpenLST based on the protocol type identifier and the UART port source of the message. That is, messages originating from UART1 will be addressed to UART1 of the destination device and so on. However, an override bit (given by b4) can be set high to create a custom flag sequence.

The CC1110 protocol was removed altogether from the RF interface during normal operations. Instead, a message packing scheme was defined in which OpenLST protocol messages would be stored in the information field of the AX.25 UI frames. This allows for the OpenLST protocol command structure to remain unmodified while being relayed between devices and thus removes the need to modify the command processing logic from the main program. A breakdown of the outgoing/incoming RF frame structure is shown in Figure 11.

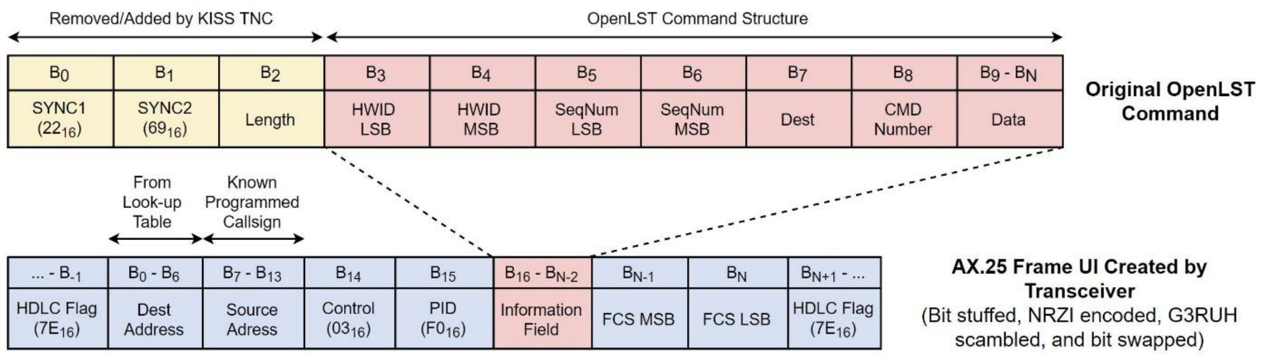
Bytes B3 to BN of the OpenLST protocol (highlighted in red) are packed into the information field of the AX.25 UI frame. For an outgoing message, the transceiver is responsible for appending the header and footer bytes of the AX.25 frame to the data (shown in blue) before transmitting. The destination address is obtained from a lookup table where OpenLST HWIDs are mapped to their respective callsigns. The source address is obtained from the preprogrammed local callsign. For incoming messages, the transceiver is responsible for removing the header and footer bytes before relaying the information to the main program.

4.3. RF Interface Modifications

4.3.1. Transmission Logic Modifications

The modifications to the transmission logic of the RF interface were minimized by taking advantage of the existing implementation of the packet engine and DMA controller. Even though these two processes are unable to perform the proper encoding for the outgoing data, they are still capable of reliably transmitting bytes from a buffer regardless of their format. Therefore, both routines can transmit an AX.25 UI frame as long as it has been properly encoded before being stored in said buffer. This removed the need for the development of a dedicated routine for encoding and sending a bitstream to the CC1110 modulator. Therefore, an outgoing message on the RF link is formatted as shown in Figure 12.

The yellow segments in the outgoing message represent the header bytes added by the packet engine. Note that the default SYNC bytes were changed to match the HDLC flag. The AX.25 UI frame itself is stored in the green segment of the message and includes all necessary encoding as well as the header and footer HDLC flags. Even though the final outgoing message does not exactly match an AX.25 UI frame, it is still effectively a valid AX.25 message. Therefore, yellow portion of the message as noise. This achieves the desired AX.25 transmitting behavior with minimal changes to the firmware structure.



Removed/Added by KISS TNC
OpenLST Command Structure
B_0 SYNC1 (22_{16})
B_1 SYNC2 (69_{16})
B_2 Length
B_3 HWID LSB
B_4 HWID MSB
B_5 SeqNum LSB
B_6 SeqNum MSB
B_7 Dest
B_8 CMD Number
B_9 - B_N Data
Original OpenLST Command
From Look-up Table
Known Programmed Callsign
... - B_{-1} HDLC Flag ($7E_{16}$)
B_0 - B_6 Dest Address
B_7 - B_{13} Source Adress
B_{14} Control (03_{16})
B_{15} PID ($F0_{16}$)
B_{16} - B_{N-2} Information Field
B_{N-1} FCS MSB
B_N FCS LSB
B_{N+1} - ... HDLC Flag ($7E_{16}$)
AX.25 Frame UI Created by Transceiver
(Bit stuffed, NRZI encoded, G3RUH scambled, and bit swapped)

Figure 11. Frame structure of an OpenLST protocol message packed in an AX.25 UI frame.

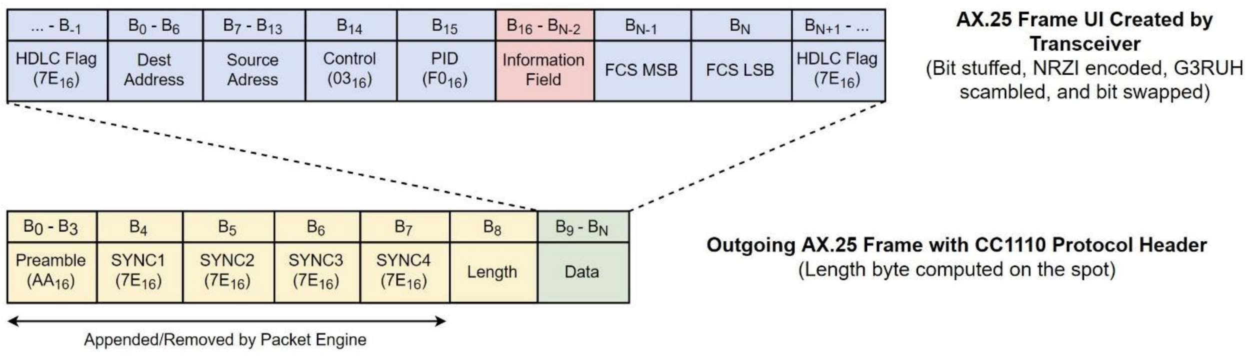


Figure 12. Frame structure for RF transmissions.

The above modification required the development of a transmitting function responsible for encoding and formatting outgoing messages into the AX.25 convention. Upon completion, the function stores the formatted outgoing frame in the RF transmit buffer, which is accessible to the packet engine and DMA controller. The overall process is outlined in the RF TX section of Figure 14.

4.3.2. Initial Reception Logic Modifications

The initial reception logic modifications revolved around the compatibility issues of the packet engine with NRZ(I) encoding and G3RUH scrambling. A new process had to be implemented to directly monitor the bitstream output of the CC1110 modulator. However, the modulator raw output is not accessible from within the MCU, it can only be accessed externally. Therefore, one of the debug pins had to be configured to output the raw modulator bitstream. The pin could then be read internally by the CC1110 MCU to gain access to the bits.

The new RF reception routine was configured as a timed interrupt triggered at 9.6 kHz (9600 baud) to match the expected data rate from incoming RF transmissions. At each interrupt trigger, the modulator bitstream would be sampled if the carrier sense flag was asserted. The sampled bits were NRZ(I) decoded and G3RUH unscrambled before being stored in a single byte acting as an eight-bit shift register. After each interrupt trigger, the routine would discard the oldest bit in the shift register and then append the newest sampled bit.

A separate section within the same ISR would check for an HDLC flag in the shift register after the first eight bits of the transmission were read. If an HDLC flag was not found, the shift register would be left shifted by one bit and the check would be performed again on the next interrupt trigger. This process was repeated until the first HDLC flag was detected and thus bit synchronization was achieved with the bitstream. Once synchronized, the bit counter would be reset, and the shift register would be sampled every eight interrupt triggers. The sample would then be saved as a byte in the RF receive buffer for the main program to access. The process would repeat itself until the first HDLC footer flag was detected, which indicates the end of the frame. Figure 13 provides an overview of how the bitstream and frame parser logic was implemented in the RF receive ISR.

The above modification was used to successfully receive AX.25 UI frames with the OpenLST hardware.

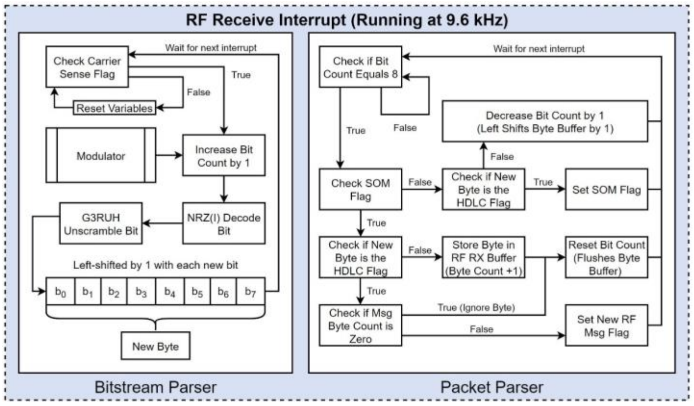

RF Receive Interrupt (Running at 9.6 kHz)
Check Carrier Sense Flag
Wait for next interrupt
True
False
Reset Variables
Modulator
Increase Bit Count by 1
NRZ(I) Decode Bit
G3RUH Unscramble Bit
Left-shifted by 1 with each new bit
b0 b1 b2 b3 b4 b5 b6 b7
New Byte
Bitstream Parser
Check if Bit Count Equals 8
Wait for next interrupt
True
False
Decrease Bit Count by 1 (Left Shifts Byte Buffer by 1)
False
Check SOM Flag
False
Check if New Byte is the HDLC Flag
True
Set SOM Flag
True
Check if New Byte is the HDLC Flag
False
Store Byte in RF RX Buffer (Byte Count +1)
Reset Bit Count (Flushes Byte Buffer)
True
Check if Msg Byte Count is Zero
True (Ignore Byte)
False
Set New RF Msg Flag
Packet Parser

Figure 13. Logic for the first attempt at modifying the reception process.

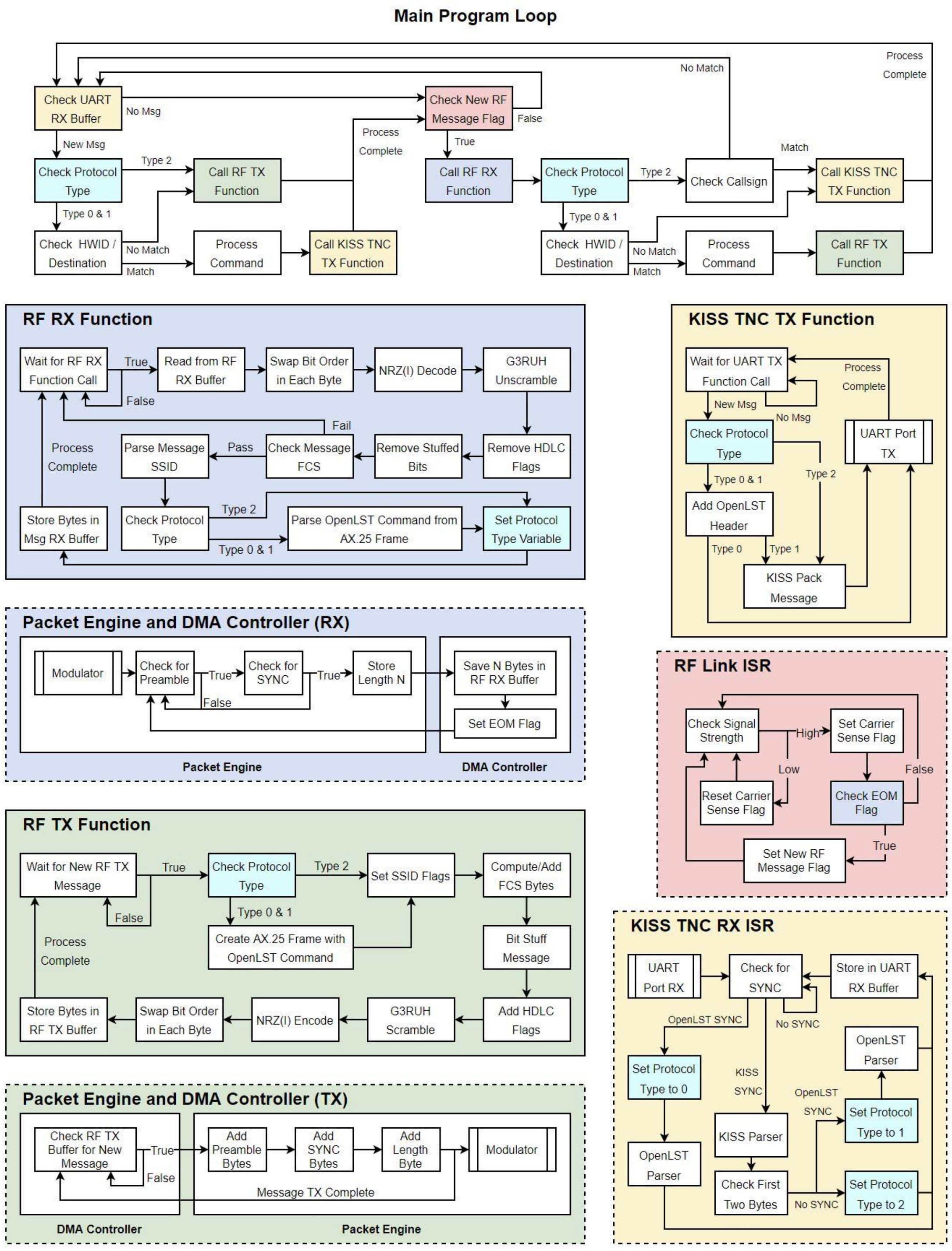


Figure 14. General overview of the final modified OpenLST firmware.

However, frames were often improperly decoded during reception as a result of clock skew. The sampling of the bitstream would drift and cause incoming bits to be skipped or sample twice, which would then lead to the entire frame being improperly decoded. This issue was expected, since the timing of the interrupt was not directly tied to the clock signal of the incoming message, but its effect on the parsing process was more significant than expected. The clock skew observed would also be worsened by the expected variations in baud rate introduced by doppler shift during satellite transmissions, further increasing the packet loss.

A possible solution to mitigate the effects of clock skew would involve tying the trigger of the interrupt to the incoming clock signal instead of a timer. This is technically possible since the incoming clock signal from the CC1110 modulator can be set as a debug pin output. However, in practice, this is not feasible due to hardware constraints on the OpenLST board. Alternative solutions involving clock recovery schemes were also explored. Nevertheless, these require the incoming signal to be pulse-shaped and sampled from an ADC and is incompatible with the bitstream output available to the MCU.

4.3.3. Final Reception Logic Modifications

The previously outlined hardware constraints limited the completeness of the solution that could be achieved with only software modifications. Since using the modulator output was not a viable option, it was decided to compromise on the receiving requirements and implement a pseudo-AX.25 solution that would use the existing packet engine and DMA controller routines. The modification hinges on the frame structure of the transmission process outlined in Section 4.3.1. The header portion of the message is automatically created by the packet engine upon transmission and thus adheres to an encoding format that is compatible with the CC1110. This implies that the header of this message can be detected by a receiving CC1110 packet engine, while the remainder of the frame can be moved to the RF receive buffer by the DMA controller. This exact process was implemented in the final modification to the reception logic. An additional receive function was implemented to parse and decode the bytes stored in the receive buffer. The overall process is shown in the RF RX section of Figure 14.

The final modifications for the receiving logic adhere to the AX.25 format needed for OTA transmissions. The additional header bytes can be treated as noise and thus the licensing requirements are met by the modifications. Nonetheless, the solution compromises on the full compatibility with an AX.25 modem. Messages transmitted by the OpenLST will be compatible with amateur radio ground stations. However, a message transmitted by these ground stations will not be recognized by a modified OpenLST unless the extra header bytes are included.

4.4. Modified OpenLST Firmware Logic

The overall modified OpenLST firmware structure and associated logic is shown in Figure 14. Note how the main program loop closely resembles the original main program shown in Figure 9. The main difference lies in the additional check for the protocol type identifier. This guarantees that command processing only occurs on OpenLST protocol commands, thus maintaining the original network structure of the firmware. The block diagram also depicts the functions available to the main program (included in solid-edged boxes) and the ISRs running in the background (included in dashed-edged boxes). The different routines have been color-coded to allow for better interpretation of their overall role in the firmware. Processes associated with the KISS TNCs are shown in yellow, with the RF transmission in green, with the RF reception in blue, and with the RF ISR in red. The global protocol type identifier has been colored in cyan for reference. The modified firmware was implemented and successfully tested using off-the-shelf SDRs (Saborio, 2021).

5. Conclusion

Developing an affordable and reliable communications solution for small satellites would solve one of the largest challenges faced by teams working on small missions. An open-source solution provides the flexibility necessary to allow for firmware and hardware modifications to best meet the mission requirements, while a solution compatible with amateur packet radio protocols allows teams to leverage the existing amateur radio infrastructure when developing their communication systems. The OpenLST integrated hardware transceiver developed by Planet provides a framework to develop this solution. As this study has shown, it is possible to achieve compatibility with amateur packet radio protocols solely through software modifications. The modifications implemented allowed for full compatibility with AX.25 while transmitting and partial compatibility while receiving. However, it was noted that full compatibility in either direction was possible with the proper hardware modifications. This research has demonstrated that a modified version of the OpenLST transceiver can provide a viable solution to the small satellite communications problem.

References

Beech, W. A., Nielsen, D. E., and Taylor, J. (1998): AX.25 Link Access Protocol for Amateur Packet Radio. Tucson Amateur Packet Radio Corporation. Available at: https://www.tapr.org/pdf/AX25.2.2.pdf (accessed Jun. 1, 2020).

Chepponis, M. and Karn, P. (1997): The KISS TNC: A Simple Host-to-TNC Communications Protocol. Available at: http://www.ax25.net/kiss.aspx (accessed Jun. 1, 2020).

Finnegan, K. W. (2014): Examining Ambiguities in the Automatic Packet Reporting System, Dept. of Electrical Eng., California Polytechnic State Univ., San Luis Obispo, CA. Available at: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=2449&context=theses (accessed Jun. 1, 2020).

Information technology – Telecommunications and Information Exchange Between Systems – High-level Data Link Control (HDLC) Procedures (2002): ISO/IEC 13239:2002. Available at: https://www.iso.org/standard/37010.html (accessed May 15, 2020).

Klofas, B. (2018): Planet Releases OpenLST, An Open Radio Solution. Available at: https://www.planet.com/pulse/planet-openlst-radio-solution-for-cubesats/ (accessed Apr. 1, 2021).

Miller, J. (1995): 9600 Baud Packet Radio Modem Design. Available at: https://www.amsat.org/amsat/articles/g3ruh/109.html (accessed Jun. 1, 2020).

Saborio, R. J. (2021): Development of an open-Source Amateur Radio Transceiver for Small Satellites, Dept. Aerospace Eng., Georgia Institute of Technology, Atlanta, GA. Available at: http://www.ssdl.gatech.edu/sites/default/files/ssdl-files/papers/mastersProjects/SaborioR-8900.pdf (accessed Apr. 1, 2021).

APPENDIX 1: Nomenclature

2-FSK	=	Two Frequency-Shift Keying
ADC	=	Analog-to-Digital Converter
ASCII	=	American Standard Code for Information Interchange
b_i	=	**i-th bit of a byte. LSb is given by zeroth bit.**
B_i	=	**i-th byte of a frame. LSB is given by zeroth byte.**
CRC	=	Cyclic Redundancy Check
DMA	=	Direct Memory Address
EOM	=	End of Message
FCS	=	Frame Checking Sequence
FEC	=	Forward Error Correction
GPIO	=	General Purpose Input Output
HDLC	=	High-level Data Link Control
HWID	=	Hardware Identification Number
IC	=	Integrated Circuit
ISR	=	Interrupt Service Routine
KISS	=	"Keep It Simple, Stupid"
LFSR	=	Linear-Feedback Shift Register
LSB	=	Least Significant Byte
LSb	=	Least Significant Bit
LSn	=	Least Significant Nibble
MCU	=	Microcontroller Unit
MSB	=	Most Significant Byte
MSb	=	Most Significant Bit
MSn	=	Most Significant Nibble
Msg	=	Message
NRZ(I)	=	Non-Return-to-Zero Inverted
NRZ(L)	=	Non-Return-to-Zero Level
OTA	=	Over-the-Air
RF	=	Radio Frequency
SDR	=	Software-Defined Radio
SMA	=	Sub-miniature version A
SOM	=	Start of Message
SSID	=	Secondary Station Identifier
TNC	=	Terminal Node Controller
UART	=	Universal Asynchronous Receive Transmit
UHF	=	Ultra High Frequency
X_2	=	base 2 number. X denotes the number. MSb is shown first.
X_{16}	=	base 16 number. X denotes the number. MSn is shown first.