Table of Contents

Concept of Operations

Command and Data Interface

Header (8 Bytes)	Payload (0 to 255 Bytes)	Payload Check Sum A (1 Byte)	Payload Check Sum B (1 Byte)

Figure 1--Packet structure for the Command and Data Interface (CDI).

Sync Characters (2 Bytes)	Command Type (2 Byte)	Payload Size (2 Bytes)	Header Check Sum A (1 Byte)	Header Check Sum B (1 Byte)

Figure 2--Description of the packet header used in the CDI.

Users interface to the Helium radios through the Command and Data Interface (CDI). Through the CDI, users configure the radio, query radio-specific telemetry points, send data to be transmitted, and request data that was received.

Users access the CDI through the UART port at 0-3.3V levels

The packet format for the digital interfaces is pictured above in Figure 1. It consists of an 8 byte, fixed length header, a variable payload segment from 0 to 255 bytes, and 2 check sum bytes. The header is described in Figure 2. The sync characters of the header are a two byte sequence[1]:

Sync Character 0: `0x48` or `'H'`

Sync Character 1: `0x65` or `'e'`

The next two bytes in a message header are the command type. Commands can be divided into two types representing the direction of the communications. Data entering the radio is noted as I-messages. I-messages are command types that begin with 0x10. Data leaving the radio is noted as O-messages. O-messages are messages that begin with 0x20. The full command list is given later in this documentation in section *Transceiver serial communications interface description*.

The "Payload Size" field of the header is a two byte, unsigned short integer containing the total number of bytes in the packet payload. The most significant byte (MSB) is given first. The maximum payload size is 255.

Two checksum bytes are appended to the header for error detection. The 8-bit Fletcher algorithm (see RFC 1145 which describes TCP) is used to calculate the checksums. The algorithm works as follows:

A buffer, `Buffer[N]`, contains data over which the checksum is to be calculated. The two checksum values (`CK_A` and `CK_B`) are 8-bit unsigned integers only. Note, if you implement it with larger sized integers, be sure to mask both `CK_A` and `CK_B` with `0xFF` after the calculations complete to ensure they are 8-bit. Psuedo-code for checksum calculation is given below.

```
CK_A = 0, CK_B = 0
For(I=0;I<N;I++)
{
    CK_A = CK_A + Buffer[I]
    CK_B = CK_B + CK_A
}
```

This loop calculates `CK_A` and `CK_B` which are then appended to the header. Following the header is the packet payload, which has a length as specified in the header. A payload checksum is then used to verify the accuracy of the payload. The checksum is calculated across all pertinent bytes of the message excluding the two sync characters of each message 'He'.

Radio Configuration

The default radio configuration is set at factory load during radio acceptance testing. This default configuration is stored in flash and is applied during power up and soft reset. The user can change the radio configuration after radio processor power up by providing a valid configuration command. The configuration message can change multiple settings at once. Changes take effect immediately, however the user should allow a settling time of at least 250 ms for settings to be applied. Default settings are described in section *Transceiver serial communications interface description*.

Data Protocol Description

Helium radios support a subset of the AX.25 packet radio protocol as defined by http://www.tapr.org/pub_ax25.html. Only the handling

[1] Bit stuffing is not needed because checksums and packet lengths are used.

of UI frames is implemented, not full connected mode. Users are able to configure the source and destination call signs, the packet length, and the TX tail and head parameters (see the command list in Table 1). The radio performs all packetization functions such as bit stuffing and check sum calculations.

Transmit Overview

Radio transmissions are performed based on the radio mode setting. All data received from the hardware interface is immediately transmitted unless the radio is busy with a current transmission. Data to be transmitted will be temporarily retained in the transmission buffer based upon the RF baud rate and the buffer length. When the buffer is full the radio will issue a NACK to the transmit command. The NACK response is not a high priority task and may not be transmitte at high radio work loads. Users should monitor for NACK messages to be able to complete proper flow control but also should time out in case there is a lost NACK. In case UART synchronization is lost users should also note that NACK or ACK responses may also be lost due to the radio UART interface believing it is within a data packet.

During transmission, the radio operates the as described previously in the Command and Data Interface section of this document.

These parameters default to standard AX.25 settings and are defined later in section *Transceiver serial communications interface description*.

Receive Overview

During reception, the radio maintains the interface as described previously in the Command and Data Interface section of this document. When reading data in a polled method the delivery of data consists of the He header, followed by the command type and the payload size. The raw data received is placed into the payload section of the message.

These parameters default to standard AX.25 settings and are defined later in section *Transceiver serial communications interface description*.

Example communications session, sending a No-Op command:

First, the user implements a No-Op request. This is performed by loading an array with the proper values and sending them to the radio over a serial connection, below in pseudo code.

```
buffer[0] = SYNC_1; //This is a #define value of ‘H’
buffer[1] = SYNC_2; //This is a #define value of ‘e’

buffer[2] = I_MESSAGE_TYPE; //This is a #define value of 0x10
buffer[3] = NO_OP_COMMAND; //This is a #define value of 0x01

buffer[4] = 0x00; //There is no payload size information in a No-Op request
buffer[5] = 0x00;

calculate_header_checksum(&buffer[2]);  //The first two synch bytes are not included in the checksum

serial.Write( &buffer[0], 8 ); //send the information out your serial port
```

The radio then responds to the request with either an acknowledge or not-acknowledge. An acknowledge is a response with the value 0xA0A in the payload bytes. For example:

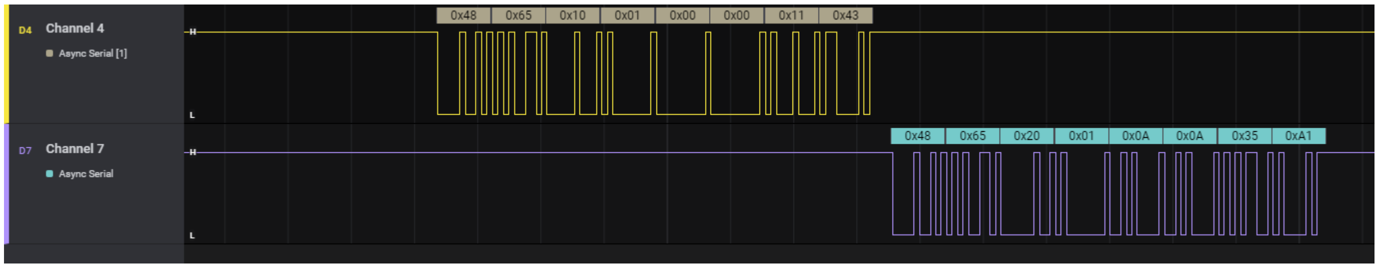
```
Byte[0] = 'H';
Byte [1] = 'e';
Byte [2] = 0x20;
Byte [3] = NO_OP_COMMAND;
Byte [4] = 0x0A; //top 4 bits are uart status flags, lower 4 bits are 0xA ACK pattern
Byte [5] = 0x0A;
Byte [6] = Checksum A;
Byte [7] = Checksum B;
```

A not-acknowledge is a response with 0xFFF in the payload bytes. For example:

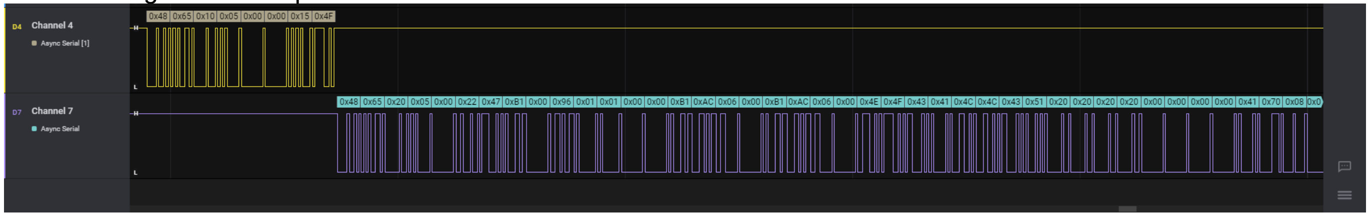
```
Byte[0] = 'H';
Byte [1] = 'e';
```

Byte [2] = 0x20;
Byte [3] = NO_OP_COMMAND;
Byte [4] = 0xFF; //top 4 bits are uart status flags, lower 4 bits are 0xF NACK pattern
Byte [5] = 0xFF;
Byte [6] = Checksum A;
Byte [7] = Checksum B;

UART Lines Example NO OP Transaction:


D4 Channel 4
Async Serial [1]
H
L
0x48 0x65 0x10 0x01 0x00 0x00 0x11 0x43
D7 Channel 7
Async Serial
H
L
0x48 0x65 0x20 0x01 0x0A 0x0A 0x35 0xA1

This is how most communications are performed with the radio. When messages include a payload, the response or command must contain a payload length value and the payload with the message. To retrieve the current radio configuration the user would send the message:
Host to the Radio:
Byte[0] = 'H';
Byte [1] = 'e';
Byte [2] = 0x10;
Byte [3] = GET_TRANSCEIVER_CONFIG;
Byte [4] = 0x00;
Byte [5] = 0x00;
Byte [6] = Checksum A;
Byte [7] = Checksum B;

Radio Configuration Response:

D4 Channel 4
Async Serial [1]
H
L
0x48 0x65 0x10 0x05 0x00 0x00 0x15 0x4F
D7 Channel 7
Async Serial
H
L
0x48 0x65 0x20 0x05 0x00 0x22 0x47 0xB1 0x00 0x96 0x01 0x01 0x00 0x00 0xB1 0xAC 0x06 0x00 0xB1 0xAC 0x06 0x00 0x4E 0x4F 0x43 0x41 0x4C 0x4C 0x43 0x51 0x20 0x20 0x20 0x20 0x00 0x00 0x00 0x00 0x41 0x70 0x08 0x0
The response from the radio would be the entire confugration structure. Refer to section “The Configuration Structure” for most up to date command list.

Example Command List:

```
//Configuration Commands Release 4.01
#define NO_OP_COMMAND           0x01
#define RESET_SYSTEM            0x02
#define TRANSMIT_DATA           0x03
#define RECEIVE_DATA            0x04
#define GET_TRANSCEIVER_CONFIG  0x05
#define SET_TRANSCEIVER_CONFIG  0x06
#define TELEMETRY_QUERY         0x07
#define WRITE_FLASH             0x08
#define RF_CONFIG               0x09
#define BEACON_DATA             0x10
```

```
#define BEACON_CONFIG           0x11
#define READ_FIRMWARE_REVISION  0x12
#define WRITE_OVER_AIR_KEY      0x13
#define FIRMWARE_UPDATE         0x14 //Reserved to special Firmware
#define FIRMWARE_PACKET         0x15 //Reserved to special Firmware
#define WRITE_KEY_A_128         0x16 //Reserved to special Firmware
#define WRITE_KEY_B_128          0x17 //Reserved to special Firmware
#define WRITE_KEY_A_256         0x18 //Reserved to special Firmware
#define WRITE_KEY_B_256          0x19 //Reserved to special Firmware
#define FAST_PA_SET             0x20
#define INVALIDATE_FLASH        0x21 //Reserved to special Firmware
#define TOGGLE_IO_DIRECT        0x22
#define TRANSMIT_DATA_NO_HEADER 0x31
#define TRANSMIT_BEACON_DATA    0x32
```

Beacon Use

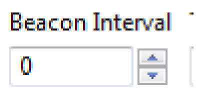
This section provides an overview of the beacon. The beacon is a message that is transmitted intermittently based on user setting. The beacon data consists of up to 256 bytes and is set using the write beacon data message. The UART message to load the beacon data matches the standard Li interface format. The beacon data can be updated at any time during normal powered operation.

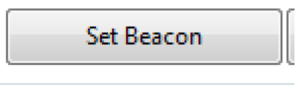
The beacon is enabled by setting the beacon interval in the beacon configuration message to a value greater than zero. Each digit of interval corresponds to (n+1)*2.5 seconds of delay, graphic. For example a beacon configuration set with a interval of five will result in a beacon being transmitted every 15 seconds.

To set the beacon interval in the configuration program first insert a number into the beacon interval box and then press the “Set Beacon” button. The beacon will operate until the beacon interval is set to 0 again. The beacon contents are defined by the user using a Beacon Data command. This command loads up to 256 bytes of data into a fixed buffer for transmission. The contents of the beacon can be modified at any time and begin empty, with 0x00 at power up. If the user does not update the contents the radio will send one byte of 0x00 for the beacon including the standard AX.25 header information configured as default.

Beacon amplifier power level is the configured power level.

Beacon interleveing is automatically performed during a communication session unless the internal transmission buffers are full at the designated time of beacon transmission. If messages are currently in the buffer the beacon transmission will be added to the buffer and will be transmitted upon its turn.


Beacon Interval
0



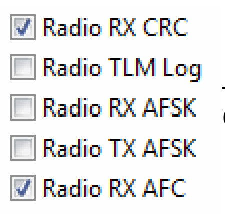
Receiver Options

This section provides an overview of the receiver options. The receive options are AFSK or CRC enable.

Audio FSK reception is only available as a hardware ordered option.

Packets are received over the air and sent to the user as fast as possible. Only one option is available for filtering the packets by CRC. The CRC check can be enabled or disabled. If the packet has a invalid CRC the packet is rejected and not sent to the user. If the CRC is not enabled then the packet regardless of its checksum is sent to the user.

The receive CRC check is toggled by radio box on the windows configuration program.


[✓] Radio RX CRC
[ ] Radio TLM Log
[ ] Radio RX AFSK
[ ] Radio TX AFSK
[✓] Radio RX AFC

Telemetry

This section provides an overview of the telemetry capabilities. The telemetry system is accessable from both UART interface and over the air by ping command. Each telemetry packet has a radio CPU time count associated with its collection time allowing for alignment with other time stamps. Each time stamp corresponds to one 2.5 second 'tick'.

Over the air telemetry access is enabled by the UART interface and users transmitting the correct access packet to download a telemetry packet, please refer to the OA Key for OA access to the ping.

The telemetry packet holds the following structure of information:

```
typedef struct telem_type
{
    uint_2 op_counter;
    sint_2 msp430_temp;
    uint_1 time_count[3];
    uint_1 rssi;
    uint_4 bytes_received;
    uint_4 bytes_transmitted;
    uint_1 rssi_lastpacket;
    uint_1 rtc_alarm_flag;
} TELEMETRY_STRUCTURE_type;
```

Digital IO and Over the Air Digital IO

This section provides an overview of the digital input and output pins available on the radio. The digital IO lines can be used for any functionality desired. \Digital IO allows users to use activity signals generated by the radio to operate system redundancies in the spacecraft. The digital IO can be triggered by a 2.5 Hz timer event, received packet, or transmitted packet events. For example, the user can configure a 30 second beacon which upon transmission will toggle the digital IO line to verify radio operation.

Over the air digital IO allows users to access the low level pin functions to perform system restore activities.

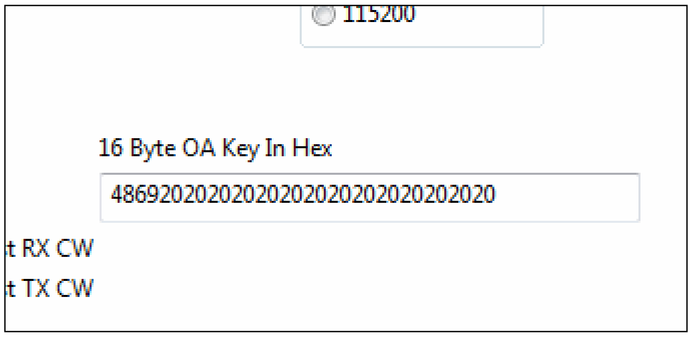
Digital IO pins can be configured in the Windows program using the three radio boxes. Each radio box configures the possible behavior for each pin. Users should select the desired behavior then press the write configuration button. Note, to ensure that current default values in the radio are not over written, press the "read configuration" button first.

Over the UART digital IO allows users to access the low level pins by direct UART command.

OA Key

Users may use a 16 byte key to access the digital IO commands from a ground station. To set the 16 byte key the user should type in the HEX value in the OA Key window in the configuration screen, then press the "Write OA Key" button. Next the user should enable over the air access to the radio configuration. This enables these functions to be interpreted by the radio.

Step 1: Enter a OA Key: Example “Hi   “ = 0x48692020202020202020202020202020

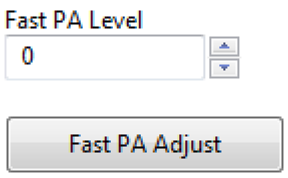

115200

16 Byte OA Key In Hex

48692020202020202020202020202020

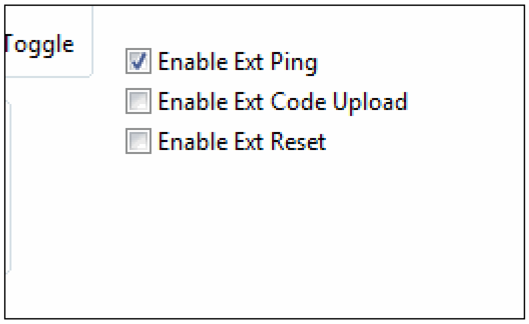
t RX CW

t TX CW


Fast PA Level

0

Step 2: Enable OA Functions: Note: “Enable Ping” is currently mis-labeled.

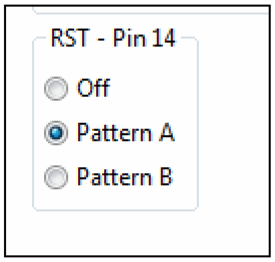


[✓] Enable Ext Ping

[ ] Enable Ext Code Upload

[ ] Enable Ext Reset

Step 3: Enable desired reset pin behavior:
Pattern A: Pin Logic Low to Logic High, Latches High
Pattern B: Pin Logic Low to Logic High to Logic Low. Pulse approx. 72 milliseconds


RST - Pin 14

Off [✓ Pattern A] Pattern B

Step 4: Sending a digital command from a ground station:

There are currently 4 commands available for users in OA functions. They are:

```
#define TELEMETRY_DUMP_COMMAND 0x30 //Reserved
#define PING_RETURN_COMMAND    0x31 //Responds with telemetry structure
#define CODE_UPLOAD_COMMAND    0x32 //Reserved
#define RADIO_RESET_COMMAND    0x33 //Performs soft reset of radio
```

```
#define PIN_TOGGLE_COMMAND     0x34 //Directly toggles OA Pin
#define BSL_SEQ_INITIATE       0x42 //Initiates BSL Mode Reserved
```

The command is located in a packet immediately proceeding the OA Key.

To send a pin toggle to reset a spacecraft, using the OA Key described previously, the user should thus send" "Hi   4" or 0x4869202020202020202020202020202034

Fast Commands

This section provides an overview of the fast commands capability. A set of reduced size commands are used for fast adjustment of the radio during pass operations.

> Power Amplifier Setting:
> To allow users to quickly adjust the power output level to changes in system voltage, dynamic link adjustment, or efficiency optimization the power amplifier command can be adjusting using a 1 byte payload packet. This is done with the 0x1020 command which has a one byte payload that over writes the current power amplifier setting in the full radio configuration. Within the configuration program this can be found on the "Telemetry and Misc" page, graphic.

This functionality is in progress as of release V3.11.

Data Rates

This section provides an overview of the multi data rate capabilities. The data rate settings of the radio are specific to the hardware type purchased. Increasing data rate uses different filter, modulation, and power settings than traditional AX.25 HAM communications. Each option is described below.

> Audio FSK: Audio FSK operates at 1200 baud. When configuring for Audio FSK the user needs to ensure that the data rate setting matches the selection of modulation. If the settings do not match the radio will not receive over the air information and will not notify the user has entered an undesirable operation mode. Audio FSK receive functions are only available in the Li-2 model radio.
>
> 9.6 kbps GFSK: Standard HAM AX.25 data rate.
>
> 19.2 kbps GFSK: Higher speed GFSK modulation.
>
> 38.4 kbps GFSK: Higher speed GFSK modulation.

Low Level RF Configuration (In Progress)

This section provides an overview of the low level RF configuration. The low level RF configuration allows users to adjust for on the fly efficiency, dopplar, and performance.

Refer to the fast RF set structure.

Transceiver Serial Communications Interface Description

This section provides an overview of commands sent from the host to the radio over the CDI and the response from the radio.

Table 1 -- Summary (not complete, refer to code snippets in "configuration structure" section of this document) of Input (I) Commands and Output (O) Radio Output.

Op Code	Command/Response Name	Arguments	Total Bytes	Summary
0x1001	No-Op	-	0	No-op command. Increments command processing counter.
0x2001	No-Op Ack	-	0	No-op Acknowledge.
0x1002	Reset	-	0	Reset radio processors and systems.
0x2002	Reset Ack	-	0	Reset Acknowledge.
0x1003	Transmit	Bytes	N	Send n number of bytes to radio board.
0x2003	Transmit Ack	-	0	Transmit Acknowledge.
0x2004	Received Data	Bytes	N	Received n number of bytes AX.25 packet
0x1005	Get Transceiver Configuration	-	0	Read radio configuration.
0x2005	Transceiver Configuration	Configuration Structure [2]	N	Radio configuration structure.
0x1006	Set Transceiver Configuration	Configuration Structure	N	Set radio configuration.
0x2006	Set Transceiver Configuration Ack	-	0	Set radio configuration Acknowledge.
0x1007	Telemetry	-	0	Query a telemetry frame.
0x2007	Telemetry	Telemetry Structure	N	Telemetry frame.[3]
0x1008	Write Flash	16 Byte MD5	16	Write Flash with MD5 Checksum
0x2008	Write Flash Ack	-	0	Write Flash Acknowledge
0x1009	RF Configure	RF Structure	N	Low Level RF Configuration
0x2009	RF Configure Ack	-	0	RF Configuration Acknowledge
0x1010	Beacon Data	Bytes	N	Set Beacon Contents
0x2010	Beacon Data Ack	-	0	Ack Set Beacon Contents
0x1011	Beacon Configure	Beacon Structure	N	Set beacon configuration
0x2011	Beacon Conf. Ack	-	0	Ack Beacon Configuration
0x1012	Read Firmware Rev	-	0	Read radio firmware revision.
0x2012	Firmware Rev	Bytes	4	Firmware number, float 4 byte
0x1013	DIO Key Write	Bytes	16	DIO Key Write
0x2013	DIO Key Write Ack	-	0	Ack DIO Key Write
0x1014	Firmware Update	16 Byte MD5	16	Firmware Update Command
0x2014	Firmware Update Ack	-	0	Firmware Update Ack
0x1015	Firmware Packet	Bytes	TBD	Firmware Packet Write
0x2015	Firmware Packet Ack	-	0	Firmware Packet Ack
0x1020	Fast Set PA	Byte	1	Power Amplifier Level Set High Speed
0x2020	Fast Set PA Ack	-	0	Power Amplifier Set Ack

[2] Refer to transceiver configuration message description.
[3] Telemetry points and their code are described in the next section.

No-Op Message: 0x01
The user sends:

Hex	0x48	0x65	0x10	0x01	0x00	0x00	Ck_A	Ck_B
ASCII/Dec	‘H’	‘e’	16	1	0	0	Calculated	Calculated

On Success the radio replys with and Acknowledge:

Hex	0x48	0x65	0x20	0x01	0x0A	0x0A	Ck_A	Ck_B
ASCII/Dec	‘H’	‘e’	32	1	11	11	Calculated	Calculated

On Failure the radio replys with Not-Acknowledge:

Hex	0x48	0x65	0x20	0x01	0xFF	0xFF	Ck_A	Ck_B
ASCII/Dec	‘H’	‘e’	32	1	N/A	N/A	Calculated	Calculated

The Configuration Structure

Compiler Directives:

```
//Configuration Commands
#define NO_OP_COMMAND           0x01
#define RESET_SYSTEM            0x02
#define TRANSMIT_DATA           0x03
#define RECEIVE_DATA            0x04
#define GET_TRANSCEIVER_CONFIG  0x05
#define SET_TRANSCEIVER_CONFIG  0x06
#define TELEMETRY_QUERY         0x07
#define WRITE_FLASH             0x08
#define RF_CONFIG               0x09
#define BEACON_DATA             0x10
#define BEACON_CONFIG           0x11
#define READ_FIRMWARE_REVISION  0x12
#define WRITE_OVER_AIR_KEY      0x13
#define FIRMWARE_UPDATE         0x14 //<--Disabled in default firmware
#define FIRMWARE_PACKET         0x15 //<--Disabled in default firmware
#define WRITE_KEY_A_128         0x16 //<--Disabled in default firmware
#define WRITE_KEY_B_128                0x17 //<--Disabled in default firmware
#define WRITE_KEY_A_256         0x18 //<--Disabled in default firmware
#define WRITE_KEY_B_256                0x19 //<--Disabled in default firmware
#define FAST_PA_SET             0x20
#define INVALIDATE_FLASH        0x21 //<--Disabled in default firmware
#define TOGGLE_IO_DIRECT        0x22
#define TRANSMIT_DATA_NO_HEADER 0x31
#define TRANSMIT_BEACON_DATA    0x32

#define GET_RTC                 0x41
#define SET_RTC                 0x42
#define ALARM_RTC               0x43

#define BAUD_RATE_9600    0
#define BAUD_RATE_19200   1
#define BAUD_RATE_38400   2
#define BAUD_RATE_57600   3
#define BAUD_RATE_115200  4

#define RF_BAUD_RATE_1200    0
```

```
#define RF_BAUD_RATE_9600    1
#define RF_BAUD_RATE_19200   2
#define RF_BAUD_RATE_38400   3
#define RF_BAUD_RATE_57600   4
#define RF_BAUD_RATE_115200  5

#define RF_MODULATION_GFSK   0
#define RF_MODULATION_AFSK   1
#define RF_MODULATION_BPSK   2

#define TELEMETRY_DUMP_COMMAND 0x30 //<--Disabled in default firmware
#define PING_RETURN_COMMAND    0x31
#define CODE_UPLOAD_COMMAND    0x32 //<--Disabled in default firmware
#define RADIO_RESET_COMMAND    0x33
#define PIN_TOGGLE_COMMAND     0x34
#define BSL_SEQ_INITIATE       0x42 //<--Disabled in default firmware

typedef struct
{
  uint_1 interface_baud_rate;   //Radio Interface Baud Rate (9600=0x00)
  uint_1 tx_power_amp_level;    //Tx Power Amp level (min = 0x00 max = 0xFF)
  uint_1 rx_rf_baud_rate;       //Radio RX RF Baud Rate (9600=0x00)
  uint_1 tx_rf_baud_rate;       //Radio TX RF Baud Rate (9600=0x00)
  uint_1 rx_modulation;         //(0x00 = GFSK);
  uint_1 tx_modulation;         //(0x00 = GFSK);
  uint_4 rx_freq;               //Channel Rx Frequency (ex: 45000000)
  uint_4 tx_freq;               //Channel Tx Frequency (ex: 45000000)
  unsigned char source[6];      //AX25 Mode Source Call Sign (default NOCALL)
  unsigned char destination[6]; //AX25 Mode Destination Call Sign (default CQ)
  uint_2 tx_preamble;           //AX25 Mode Tx Preamble Byte Length (0x00 = 20 flags)
  uint_2 tx_postamble;          //AX25 Mode Tx Postamble Byte Length (0x00 = 20 flags)
  uint_2 function_config;       //Radio Configuration Discrete Behaviors
  uint_2 function_config2;       //Radio Configuration Discrete Behaviors 2
} RADIO_CONFIGURATION_TYPE;
#define RADIO_CONFIG_SIZE   34  //sizeof(RADIO_CONFIGURATION_TYPE)

typedef struct
{
  uint_1 front_end_level; //0 to 63 Value
  uint_1 tx_power_amp_level; //0 to 255 value, non-linear
  uint_4 tx_frequency_offset; //Up to 20 kHz
  uint_4 rx_frequency_offset; //Up to 20 kHz
  uint_1 tx_frequency_deviation; //Set for your baud rate options: 0 (2.7 kHz),1 (5.4 kHz),2
(10.8 kHz),3 (21.6 kHz),4 (43.2 kHz) CAUTION
  uint_1 rx_frequency_deviation; //N/A for release 3.10
  uint_1 pre_transmit_delay;//Delay in tens of milliseconds. Default 1 second = 100;
  uint_1 post_transmit_delay;//Delay in tens of milliseconds. Default 0  (NOT IMPLEMENTED
USE POSTAMBLE)
} RADIO_RF_CONFIGURATION_TYPE;
#define RF_CONFIG_SIZE   14

typedef struct
{
  uint_1 beacon_interval;  //value of 0 is off, 2.5 sec delay per LSB
} RADIO_BEACON_CONFIGURATION_TYPE;
```

```
#define BEACON_CONFIG_SIZE  1



typedef struct telem_type
{
    uint_2 op_counter;
    sint_2 msp430_temp;
    uint_1 time_count[3];
    uint_1 rssi;
    uint_4 bytes_received;
    uint_4 bytes_transmitted;
    uint_1 rssi_lastpacket;
    uint_1 rtc_alarm_flag;
} TELEMETRY_STRUCTURE_type;
```

//Binary Coded Decimal Internal RTC MSP430 please refer to Real-Time Clock (RTC)

```
typedef struct RTC_type
{
    unsigned short int year;                          // Year = 0x2021
    unsigned char mon;                             // Month = 0x10 = October
    unsigned char day;                             // Day = 0x07 = 7th
    unsigned char dow;                             // Day of week = 0x05 = Friday
    unsigned char hour;                            // Hour = 0x11
    unsigned char min;                             // Minute = 0x59
    unsigned char sec;                             // Seconds = 0x30
    unsigned char alarm_dow;                        // RTC Day of week alarm = 0x2
    unsigned char alarm_day;                          // RTC Day Alarm = 0x20
    unsigned char alarm_hour;                         // RTC Hour Alarm
    unsigned char alarm_min;                          // RTC Minute Alarm
} RTC_STRUCTURE_type;
```

Packed Configruation Structure Words 1 and 2

```
  uint_2 function_config;       //Radio Configuration Discrete Behaviors
/*
Li2=GPIO_B, Pin #10
0000 0000 0000 xx00     Off Logic Low
0000 0000 0000 xx01     2.0 second Toggle
0000 0000 0000 xx10     TX Packet Toggle
0000 0000 0000 xx11     Rx Packet Toggle
Li2=Config Pin 1, Pin #14
0000 0000 0000 00xx     Off Logic Low
0000 0000 0000 01xx     Tx/Rx Switch
0000 0000 0000 10xx     2.0 hz WDT
0000 0000 0000 11xx     Rx Packet Toggle
Li2=External Event, Pin #12
0000 0000 xx00 xxxx     Off Logic Low
0000 0000 xx01 xxxx     Enable
0000 0000 xx0X xxxx     Pattern A
0000 0000 xx1X xxxx     Pattern B

CRC Functions
0000 0000 x1xx xxxx     RX CRC Enable 1/Disable 0
0000 0000 1xxx xxxx     TBD // TX CRC Enable 1/Disable 0
```

```
Telemetry Functions
0000 xxx1 xxxx xxxx     Telemetry Packet Logging Enable 1/Disable 0
0000 x01x xxxx xxxx     Logging Rate 0 1/10 Hz, 1 1 Hz, 2 2 Hz,3 4 Hz
0000 1xxx xxxx xxxx     Telemetry Dump Enable 1/Disable 0

Beacon Functions
0xx1 xxxx xxxx xxxx     Ping Return Enable 1/Disable 0
0x1x xxxx xxxx xxxx     Code Upload Enable 1/Disable 0
01xx xxxx xxxx xxxx     System Reset Enable 1/Disable 0

Factory Defaults Restored
1xxx xxxx xxxx xxxx     Factory settings restore complete flag


  uint_2 function_config2;       //Radio Configuration Discrete Behaviors 2
//Config 2
Functions
0000 0000 0000 xxx1 AFC Enable 1/Disable 0
0000 0000 0000 xx0x RX CW
0000 0000 0000 x0xx TX CW
0000 0000 0000 0xxx Radio Alarm Transmit Enable 1/Disable 0

0000 0000 xxx1 xxxx Encryption A Enabled
0000 0000 xx1x xxxx Encryption B Enabled
0000 0000 x1xx xxxx Radio Configration Duration Enabled
0000 0000 1xxx xxxx No AX25 Header but has 16 bit CRC

0000 xxx0 xxxx xxxx BSL TBD
0000 xx0x xxxx xxxx BSL TBD
0000 x0xx xxxx xxxx BSL TBD
0000 0xxx xxxx xxxx BSL TBD

xxx0 xxxx xxxx xxxx BSL TBD
xx0x xxxx xxxx xxxx BSL TBD
x0xx xxxx xxxx xxxx BSL TBD
0xxx xxxx xxxx xxxx BSL TBD


Packed UART Status Bits
UART ACK Status Bits
xxx0 A A A Transmit Buffer Full Flag
xx0x A A A GPIO A State High = 1 Low = 0
x0xx A A A GPIO A State High = 1 Low = 0
0xxx A A A External Event  State High = 1 Low = 0

UART NACK Status Bits
xxx0 F F F BSL Transmit Buffer Full Flag
xx0x F F F GPIO A State High = 1 Low = 0
x0xx F F F GPIO B State High = 1 Low = 0
0xxx F F F External Event State High = 1 Low = 0

*/
```

Example UART Read Loop

```
    while(!terminateThreadFlag) //While the thread is still active
    {
        Yield(); //Yield to the operating system
        Sleep(1); //Every 10 ms we go check for more data
        if( init_check == 1 ) //Ensure that the UART is still initialized on this host and failure has not occured
        {
            serial.Read(&data[1],1,&read,NULL,INFINITE); //Read in one byte into the buffer
            if( read > 0 ) //if there was a read
            {
                if( (data[0] == 'H') && (data[1] == 'e') ) //Check for sync characters in the first two
positions
                {
                    serial.Read(&data[2],6,&read,NULL,INFINITE); //grab a header chunk
                    if(read != 6) //If we read a header
                    {
                        Yield(); //Yield to the operating system
                        Sleep(1); //Every 1 ms we go check for more data
                        serial.Read(&data[2+read],6-read,&read,NULL,INFINITE);//Do an
extra read just in case the buffer had not recevied everything
                    }
                    if(1 == test_header_checksum(&data[2]) ) //test the header checksum for
validity
                    {
                        identify_message(&data[0]); //perform our message identification
                    }else{
                        memset(&data[0],0,sizeof(data)); //clear our buffer because the
header was bad
                        //printf("BAD HEADER CRC\n"); //DEBUG
                    }
                }else{
                    data[0] = data[1]; //move the read byte down to sync
                }
            }
        }
    }
```

Example Message Identification

```
void identify_message( uint_1 *data )
{
    DWORD read=0;
    uint_2 read_location = 0;
    uint_2 size = 0;
    uint_2 size_to_read = 0;
    uint_2 iterations = 0;
    uint_2 t=0;


    if( ((data[4]&0x0F) == 0x0A) && data[5] == 0x0A )
    {
```

```
                if( (data[4] & 0x80)==0x80 ) buffer_full_stop_tx = 1; else buffer_full_stop_tx = 0;
                if( (data[4] & 0x40)==0x40 ) gpio_status_a = 1; else gpio_status_a = 0;
                if( (data[4] & 0x20)==0x20 ) gpio_status_b = 1; else gpio_status_b = 0;
                if( (data[4] & 0x10)==0x10 ) gpio_status_c = 1; else gpio_status_c = 0;

                switch( data[3] )
                {
                case NO_OP_COMMAND:
                        printf("Valid No Op Ack Received\n");
                break;
                case RESET_SYSTEM:
                        printf("Valid Reset System Ack Received\n");
                break;
                case TRANSMIT_DATA:
                        //printf("Valid Transmit Data Ack Received\n");
                        SetEvent(ACKEvent);
                break;
        case GET_TRANSCEIVER_CONFIG:
                        printf("Valid Transceiver Get Config Ack Received\n");

        break;
        case SET_TRANSCEIVER_CONFIG:
                        printf("Valid Transceiver Set Config Ack Received\n");

        break;
        case RF_CONFIG:
                        printf("Valid RF Configuration Ack Received\n");

        break;
        case BEACON_DATA:
                        printf("Valid Beacon Data Ack Received\n");

        break;
        case BEACON_CONFIG:
                        printf("Valid Beacon Config Ack Received\n");

        break;
        case FAST_PA_SET:
                        printf("Valid Fast PA Ack Received\n");

        break;
        case TELEMETRY_QUERY:
                        printf("Valid Telemetry Query Ack Received\n");

        break;
        case WRITE_OVER_AIR_KEY:
                        printf("Valid OA Write Ack Received\n");

        break;
        case WRITE_FLASH:
                        printf("Valid Flash Write Ack Received\n");

        break;
                case WRITE_KEY_A_128:
                        printf("Valid Key A 128 Ack Received\n");
```

```
break;
              case WRITE_KEY_B_128:
                        printf("Valid Key B 128 Ack Received\n");

break;
              case WRITE_KEY_A_256:
                        printf("Valid Key A 256 Ack Received\n");

break;
case WRITE_KEY_B_256:
                        printf("Valid Key B 256 Ack Received\n");

break;
case TRANSMIT_BEACON_DATA:
                        printf("Valid Transmit Beacon Ack Received\n");

break;
case INVALIDATE_FLASH:
                        printf("Flash Inv Ack Received\n");

break;
              default:
break;
              }

   }else if( ((data[4]&0x0F) == 0x0F) && data[5] == 0xFF ){
              if( (data[4] & 0x80)==0x80 ) buffer_full_stop_tx = 1; else buffer_full_stop_tx = 0;
              if( (data[4] & 0x40)==0x40 ) gpio_status_a = 1; else gpio_status_a = 0;
              if( (data[4] & 0x20)==0x20 ) gpio_status_b = 1; else gpio_status_b = 0;
              if( (data[4] & 0x10)==0x10 ) gpio_status_c = 1; else gpio_status_c = 0;
              switch( data[3] )
              {
              case NO_OP_COMMAND:
                        printf("No Op Nack Received\n");

              break;
              case RESET_SYSTEM:
                        printf("Reset System Nack Received\n");

              break;
              case TRANSMIT_DATA:
                        printf("Transmit Data Nack Received\n");
                        SetEvent(NACKEvent);
              break;
              case TRANSMIT_DATA_NO_HEADER:
                        printf("Transmit Data No Header Nack Received\n");

              break;
              case GET_TRANSCEIVER_CONFIG:
                        printf("Transceiver Get Config Nack Received\n");

break;
case SET_TRANSCEIVER_CONFIG:
                        printf("Transceiver Set Config Nack Received\n");

break;
```

```
                case RECEIVE_DATA:
                        printf("Receive Data Nack Received\n");

        break;
        case TELEMETRY_QUERY:
                        printf("Telemetry Query Nack Received\n");
        break;

        case WRITE_FLASH:
                        printf("Flash Write Nack Received\n");
        break;
        case INVALIDATE_FLASH:
                        printf("Flash Inv Nack Received\n");
        break;
        default:
                        printf("Unk Nack Received\n");
                break;
                }

                SetEvent(NACKEvent);
          }else{
                size = ((data[4] << 8) | (data[5]));
                size_to_read = size+2;
                Yield();//read in the remaining data
                Sleep(10);
                serial.Read(&data[8],size_to_read,&read,NULL,INFINITE);
                read_location = 8;
                while(read != (size_to_read))
                {
                        size_to_read = size_to_read - (unsigned short) read;
                        read_location = read_location + (unsigned short) read;
                        Yield();
                        Sleep(5);
                        serial.Read(&data[read_location],size_to_read,&read,NULL,INFINITE);
                        iterations++;
                        if(iterations > 20)
                                break;
                }
                if(test_payload_checksum(&data[2],size+6))
                {
                        switch( data[3] )
                        {
                                case TELEMETRY_QUERY:
                                        memcpy( (void *)&received_telemetry, &data[8],
sizeof(Radio_Telemetry_Type) );
                                        printf("Valid Transceiver Telemetry Received\n");

                                break;
                                case GET_TRANSCEIVER_CONFIG:
                                        memcpy( (void *)&received_configuration, &data[8],
sizeof(Radio_Configuration_Type) );
                                        printf("Valid Transceiver Config Received\n");

                                break;
                                case RECEIVE_DATA:
                                        if( size < 4085 ){
```

```
                                                        memset(
&data_received_buffer[0],0x00,sizeof(data_received_buffer));
                                                        memcpy(&data_received_buffer[0], &data[0], size + 10);
                                                        for(int k=0;k<size+10;k++) //GUI ONLY We have to take out all the
nulls or else the string will not print
                                                                if( data_received_buffer[k] == 0x00 )
data_received_buffer[k] = ' ';
                                                }else{
                                                        break;
                                                }
                                                printf("Valid Received Data\n");
                                        break;
                                        case READ_FIRMWARE_REVISION:
                                                memcpy( (void *)&firmware, &data[8],  4 );
                                                printf("Valid Transceiver Firmware Rev Received\n");
                                        break;
                                        default:
                                        break;
                                }
                        }else{
                                printf("Bad Packet Checksum Received");
                        }
                }
}
```

Example Transmit Cycle

```
//Message Sync characters
#define SYNC_1                                                  'H'
#define SYNC_2                                                  'e'

//Message types
#define I_MESSAGE_TYPE                                  0x10
#define O_MESSAGE_TYPE                                  0x20

retransmit:
                if( buffer_full_stop_tx == 0 )
                        send_data_imessage(buffer_out,size);
                else
                {
wait_status:
                        Sleep(100);
                        send_header_only_imessage( NO_OP_COMMAND );
                        if(buffer_full_stop_tx == 0 )
                        {
                                Sleep(1);
                                goto retransmit;
                        }
                        else
                                goto wait_status;
                }

void send_header_only_imessage( uint_1 command )
{
        buffer[0] = SYNC_1;
        buffer[1] = SYNC_2;
```

```
	buffer[2] = I_MESSAGE_TYPE;
	buffer[3] = command;

	buffer[4] = 0x00;
	buffer[5] = 0x00;

	calculate_header_checksum(&buffer[2]);

	serial.Write( &buffer[0], 8 );
}

void send_data_imessage( uint_1 type, uint_1 *data, uint_2 size )
{
	buffer[0] = SYNC_1;
	buffer[1] = SYNC_2;

	buffer[2] = I_MESSAGE_TYPE;
	buffer[3] = type;

	buffer[4] = size>>8;
	buffer[5] = size & 0xFF;

	calculate_header_checksum(&buffer[2]);

	memcpy(&buffer[8],data,size);

	calculate_payload_checksum(&buffer[2], size+6); //remove the sync chars

	serial.Write( &buffer[0], 8+size+2 );
}

void calculate_header_checksum( uint_1 *data )
{
	uint_2 i = 0;
   uint_1 ck_a = 0;
   uint_1 ck_b = 0;

	for( i = 0; i < 4; i++ )
	{
		ck_a += data[i];
		ck_b += ck_a;
	}

	data[4] = ck_a;
	data[5] = ck_b;

	return;
}

void calculate_payload_checksum( uint_1 *data, uint_2 size )
{
	uint_2 i = 0;
   uint_1 ck_a = 0;
   uint_1 ck_b = 0;
```

```
        for( i = 0; i < size; i++ )
        {
                ck_a += data[i];
                ck_b += ck_a;
        }

        data[size] = ck_a;
        data[size+1] = ck_b;

        return;
}
```

The MD5 Checksum

Please refer to the standard MD5 algorithm:
http://en.wikipedia.org/wiki/MD5

Trademarks
In progress.

Disclaimer
All information in this document is subject to change at anytime. Look for continued updates at:
http://www.astrodev.com/

Notes
Prototype Windows source C++ code for communicating with the radio is available by email request.