Transceiver IP Link Protocol rev.1

Abstract

This document specifies an application-level Transceiver IP Link Protocol (TILP) to be used to monitor and control devices compatible with CAT & RS-485 protocols remotely.
Table of contents

Goals ........................................................................... 3
Transceiver audio interface ........................................... 3
CAT interface ........................................................... 3
FSK output ............................................................... 3
PTT and CW output functions ...................................... 3
RS-485 ................................................................. 3
WinKey emulation ..................................................... 3
Terminology ................................................................ 4
TILP protocol details ................................................... 4
General structure of the data packet ......................... 4
Packet types overview ............................................... 5
Authorization packet .................................................. 6
Checksum calculation algorithm ............................... 7
Session keep alive ..................................................... 8
PTT packet ............................................................. 9
Audio packet .......................................................... 11
TTY packet ............................................................ 15
CONNERR packet structure ..................................... 19
Access levels packet ............................................... 20
Firmware version packet .......................................... 22
Packet aggregation ................................................... 23
Packet segmentation ................................................ 23
Alignment ............................................................... 23
PTT/CW stream ...................................................... 23
Appendix A An example of the short session .............. 24
Appendix B The short session with the aggregated packets ... 25
Appendix C The session with the audio streams example ... 26
Appendix D The session with the audio and CW/PTT streams ... 27
Appendix E The session keepalive example ................. 28
Appendix F The serial ports data transfer example ....... 29
Goals

Most popular ham radio applications for working with transceivers require multiple serial ports. When it comes to control the transceiver remotely it is necessary to use special devices for bind serial ports via the Internet or via a local area network. Or to use TILP-compatible devices to communicate with remote transceiver(s) via IP-based network.

Transceiver audio interface

TILP protocol provides audio stream encapsulation and delivery from the transceiver to control software and back.

CAT interface

CAT (Computer Aided Tuning) provides control of receiving and transmitting frequency, VFO, diversity reception, audio levels, memory and other operations by the computer software. Normally, modern transceivers have serial (with various signal levels) link providing CAT interface.

FSK output

FSK (Frequency Shift Keying) is a popular method of transmitting digital messages over radio primarily used in radioteletype (RTTY) mode. Most transceivers provide FSK modulator feature to make the RTTY signal stable and clear.

PTT and CW output functions

Transceivers provide PTT (Push To Talk) and CW (Continuous Wave) keyer inputs to allow setting the transmitter on or off and operating CW using external device (PTT pedal, CW bug or paddle, terminal node controller, or personal computer).

RS-485

Some types of equipment, such as antenna switches and relays can be controlled via a RS-485 port. Multiple devices may be connected in parallel and controlled by the computer software.

WinKey emulation

To operate CW the ham operator may use software which either directly manipulates the DTR line, or uses the WinKey protocol.

All the features mentioned above are supported by the TILP protocol.
Terminology

TILP device - any device that supports TILP protocol.

TILP protocol details

General structure of the data packet

RigExpert Wireless Transceiver Interfaces uses a proprietary protocol for exchanging data between the device and the computer.

This protocol has the following key features:
- data flow from PC software to TILP device and back is organized into packets
- packets are Checksum Protected
- The Payload is a variable length field

Packet structure:

<table>
<thead>
<tr>
<th>type</th>
<th>params</th>
<th>len</th>
<th>crc</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parameters (4 bytes)</td>
<td></td>
<td>Checksum (1 byte)</td>
<td>Packet payload (variable size)</td>
</tr>
<tr>
<td>Packet type (1 byte)</td>
<td>Payload (data) length (2 bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packet contents:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>Packet type. Field value depends on the packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>Parameters (depend on the packet type)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>Payload length (in bytes)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td>Checksum value. The checksum is calculated over the packet excluding the CRC field</td>
</tr>
<tr>
<td>8</td>
<td>up to 0xFFFF</td>
<td>data</td>
<td>Packet payload</td>
</tr>
</tbody>
</table>

NOTE: packet header consists of “type”, “params” and “len” fields (first 7 bytes)
NOTE: The checksum is calculated over the packet excluding the CRC field. I.e. checksum algorithm takes into account only header and data fields ("type", "params", "len" and "data").

Packet types overview

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Hex value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHORIZ_TYPE</td>
<td>0x00</td>
<td>Authorization packet</td>
</tr>
<tr>
<td>PTT_TYPE</td>
<td>0x01</td>
<td>PTT state packet</td>
</tr>
<tr>
<td>AUDIO_TYPE</td>
<td>0x02</td>
<td>Audio parameters packet (sends by both PC and TILP device)</td>
</tr>
<tr>
<td>TTY1_TYPE</td>
<td>0x03</td>
<td>Parameters and data for CAT serial port</td>
</tr>
<tr>
<td>TTY2_TYPE</td>
<td>0x04</td>
<td>Parameters and data for RS-485 port</td>
</tr>
<tr>
<td>TTY3_TYPE</td>
<td>0x05</td>
<td>Parameters and data for FSK port</td>
</tr>
<tr>
<td>CONERR_TYPE</td>
<td>0x08</td>
<td>Connection error type (sends by TILP device)</td>
</tr>
<tr>
<td>PERMISS_TYPE</td>
<td>0x09</td>
<td>Access right parameters (sends by TILP device)</td>
</tr>
<tr>
<td>FWVER_TYPE</td>
<td>0x0A</td>
<td>Firmware revision</td>
</tr>
</tbody>
</table>

```c
/**
 * @brief Packet types
 */
typedef enum TILProtocolPacketTypes {
  AUTHORIZ_TYPE = 0x00, // Authorization packet
  PTT_TYPE,            // PTT state packet
  AUDIO_TYPE,          // Audio parameters packet
  TTY1_TYPE,           // Parameters and data for CAT serial port
  TTY2_TYPE,           // Parameters and data for RS-485 port
  TTY3_TYPE,           // Parameters and data for FSK port
  RESERVED_6,
  RESERVED_7,
  CONERR_TYPE,        // Connection error type
  PERMISS_TYPE,        // Access rights parameters
  FWVER_TYPE,          // Firmware revision
};
```
Authorization packet

Payload contents:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x00</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>0x00</td>
<td>For Authorization packet this field value must be zeroed</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x00</td>
<td>Passphrase length (in bytes). Passphrase length must not exceed 32 bytes</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet (except CRC field)</td>
</tr>
<tr>
<td>8</td>
<td>up to 32</td>
<td>data</td>
<td></td>
<td>Passphrase</td>
</tr>
</tbody>
</table>

If the password doesn’t match the TILP device’s one, the TCP connection will be terminated by the TILP device.

/**
 * @brief send_authorization
 * @param sd - socket descriptor
 * @param pass - pointer to password string
 * @return
 */
int send_authorization(int sd, const char *pass)
{
    const uint32_t slen = strlen(pass);
    char buff[sizeof(packet_t) + slen];
    int retval;
    struct packet_t *pack;

    pack = (struct packet_t*)buff;
    pack->type = AUTHORIZ_TYPE;
    pack->params = 0;
    pack->len = slen;
    pack->crc = 0;

    memcpy(&buff[sizeof(packet_t)], pass, slen);
    pack->crc = crc8(buff, sizeof(packet_t) + pack->len);

    retval = write(sd, buff, sizeof(buff));

    return retval;
}
Checksum calculation algorithm

The checksum is calculated over the packet excluding the CRC field. I.e. checksum algorithm takes into account only header and data fields (“type”, “params”, “len” and “data”).

```c
struct packet_t {
    uint8_t     type;
    uint32_t    params;
    uint16_t    len;
    uint8_t     crc;
};
char buff[30];
struct packet_t *pack;
pack = (struct packet_t*)buff;
pack->len = sizeof(buff);
pack->crc = 0;
// Init other fields of struct packet_t
pack->crc = crc8(&pack, sizeof(packet_t) + pack->len);
```

For the CRC8 calculations use this function:

```c
/*
Name : CRC-8
Poly : 0x31 x^8 + x^5 + x^4 + 1
Init : 0xFF
Revert: false
XorOut: 0x00
Check : 0xF7 ("123456789")
*/
unsigned char crc8(unsigned char *pcBlock, unsigned int len) {
    unsigned char crc = 0xFF;
    unsigned int i;

    while (len--)
    {
        crc ^= *pcBlock++;

        for (i = 0; i < 8; i++)
            crc = crc & 0x80 ? (crc << 1) ^ 0x31 : crc << 1;
    }

    return crc;
}
```
Session keep alive

Within 8 (eight) seconds, the application and the interface should exchange with at least one packet. Usually this is the PTT status packet (the interface transmits this packet asynchronously).

If within 8 (eight) seconds the interface does not take a single packet, the TCP/IP connection will be terminated and the interface will go into a safe state (switched off PTT, CW and FSK).

If within 8 (eight) seconds the user application has not received a single packet from the interface, it must break the connection.
**PTT packet**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x01</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>0</td>
<td>For PTT packet this field value must be zeroed</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x01</td>
<td>For the PTT packet the payload always consist of one byte</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet (header and data)</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>data</td>
<td>0 or 1</td>
<td>0 = PTT OFF, 1 = PTT ON</td>
</tr>
</tbody>
</table>

PTT packet is sent both ways – to and from the interface. When the packet is sent from the application to the interface, the interface sets PTT as specified in the data field. Packets which send from the interface to the application have current status of PTT. Application can update PTT status from those packets if it needs.

Packet sending:

```c
/**
 * @param sd - Socket descriptor
 * @param state - ptt state
 * @return -> > 0 if Ok, <=0 if error
 */
int send_ptt(int sd, bool state)
{
    int retval;
    char buff[sizeof(packet_t) + 1];
    struct packet_t *pack;

    pack = (struct packet_t*)buff;
    pack->type = PTT_TYPE;
    pack->params = 0;
    pack->len = 1;
    pack->crc = 0;

    if (state) {
        buff[sizeof(packet_t)] = 1;
    } else {
        buff[sizeof(packet_t)] = 0;
    }
    /*Calculate CRC8*/
    pack->crc = crc8(&pack, sizeof(packet_t) + pack->len);
    /*Send data via TCP socket*/
```
retval = write(sd, buff, sizeof(buff));
    return retval;
}

Packet receiving:
int get_ptt(int sd)
{
    int retval;
    char buff[sizeof(packet_t) + 1];
    struct packet_t *pack;
    uint8_t crc;

    retval = read(sd, buff, sizeof(buff));

    if (retval <= 0 || retval != sizeof(buff)) {
        return -1;
    }

    pack = (struct packet_t*)buff;

    if (pack->type != PTT_TYPE || pack->len != 1) {
        return -1;
    }

    crc = pack->crc;
    pack->crc = 0;

    pack->crc = crc8(&pack, sizeof(packet_t) + pack->len);

    if (crc != pack->crc) {
        return -1;
    }

    return (int)buff[sizeof(packet_t)];
}
Audio packet

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x02</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td></td>
<td>Sampling rate &amp; codec used</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x00</td>
<td>Initial packet contains no payload</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet (header and data)</td>
</tr>
</tbody>
</table>

[Params] field structure for Audio packet

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>samplerate</td>
<td></td>
<td>The sampling rate (Hz)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>codec</td>
<td></td>
<td>Codec types:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = PCM</td>
<td>1 = µ-law</td>
</tr>
</tbody>
</table>

The application must send this packet to properly initialize the audio codec after a connection has been established.

typedef union {
    uint32_t data;
    struct {
        uint16_t samplerate;
        uint16_t codec;
    };
} audio_head_t;

/**
 * @brief send_audio_init
 * @param sd - socket descriptor
 * @param srate - Samplerate (8000, 12000, 16000)
 * @param codec - 0-PCM, 1-µLAW, 2-aLAW
 * @return
 */
bool send_audio_init(int sd, uint16_t srate, uint16_t codec) {
    int retval;
    char buff[sizeof(packet_t)];
    struct packet_t *pack;
    audio_head_t ahead;
transceiver_ip_link_protocol

```
ahead.samplerate = srate;
ahead.codec = codec;

pack = (struct packet_t*)buff;
pack->type = AUDIO_TYPE;
pack->params = ahead.data;
pack->len = 0;
pack->crc = 0;

/*Calculate CRC8*/
pack->crc = crc8(&pack, sizeof(packet_t));
/*Send data via TCP socket*/
retval = write(sd, buff, sizeof(buff));

if (retval < sizeof(buff)) {
    return false;
}

return true;
```

The interface in turn sends the packet that contains the input and output levels of the codec’s amplifiers:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x02</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>Sampling rate &amp; codec used</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td>CRC of the packet (header and data)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>outlvl</td>
<td>0..118</td>
<td>Out level. Interface’s audio output level. Range from 0 to 118 (0 = 0dB, 118 = 78.3dB)</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>inLlvl</td>
<td>0..110</td>
<td>In level. Left channel amplifier level. Range from 0 to 110. (0 = 0dB, 110 = 55dB).</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>inRlvl</td>
<td>0..110</td>
<td>In level. Right channel amplifier level. Range from 0 to 110. (0 = 0dB, 110 = 55dB).</td>
</tr>
</tbody>
</table>
/**
* @brief receive_audio_params
* @param [in] sd - socket descriptor
* @param [out] outlvl - current output level in the TILP device
* @param inLlvl [out] - current input level in the TILP device
* (left channel)
* @param inRlvl [out] - current input level in the TILP device
* (right channel)
* @return true of false
*/
bool receive_audio_params(int sd, uint8_t* outlvl, uint8_t* inLlvl, uint8_t* inRlvl)
{
    int retval;
    char buff[sizeof(packet_t)+3];
    struct packet_t *pack;
    audio_head_t ahead;

    retval = read(sd, buff, sizeof(buff));

    if (retval <= 0 || retval != sizeof(buff)) {
        return false;
    }

    pack = (struct packet_t*)buff;

    if (pack->type != AUDIO_TYPE || pack->len != 3) {
        return false;
    }

    crc = pack->crc;
    pack->crc = 0;

    pack->crc = crc8(&pack, sizeof(packet_t) + pack->len);

    if (crc != pack->crc) {
        return false;
    }

    *outlvl = buff[sizeof(packet_t)];
    *inLlvl = buff[sizeof(packet_t)+1];
    *inRlvl = buff[sizeof(packet_t)+2];

    return true;
}

To set levels the application must send this packet. If the levels are not changed again for 5 seconds, current values will be stored in the non-volatile RAM.
Setting audio levels example:
/**
 * @brief send_audio_params
 * @param sd - socket descriptor
 * @param srate - Samplerate (8000, 12000, 16000)
 * @param codec - 0-PCM, 1-uLAW, 2-aLAW
 * @param outlvl - output level for the TILP device
 * @param inLlvl - input level for the TILP device (left channel)
 * @param inRlvl - input level for the TILP device (right channel)
 * @return true of false
 */
bool send_audio_params(int sd, uint16_t srate, uint16_t codec, uint8_t outlvl, uint8_t inLlvl, uint8_t inRlvl)
{
    int retval;
    char buff[sizeof(packet_t)+3];
    struct packet_t *pack;
    audio_head_t ahead;

    ahead.samplerate = srate;
    ahead.codec = codec;

    pack = (struct packet_t*)buff;
    pack->type = AUDIO_TYPE;
    pack->params = ahead.data;
    pack->len = 3;
    pack->crc = 0;

    buff[sizeof(packet_t)] = outlvl;
    buff[sizeof(packet_t)+1] = inLlvl;
    buff[sizeof(packet_t)+2] = inRlvl;

    /*Calculate CRC8*/
    pack->crc = crc8(&pack, sizeof(packet_t)+pack->len);
    /*Send data via TCP socket*/
    retval = write(sd, buff, sizeof(buff));

    if (retval < sizeof(buff)) {
        return false;
    }
    return true;
}
## TTY packet

### Payload contents

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x03</td>
<td>Packet type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>..</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td></td>
<td>Serial port settings bitfield</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td></td>
<td>Payload length (in bytes)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet</td>
</tr>
<tr>
<td>8</td>
<td>Variable</td>
<td>data</td>
<td></td>
<td>Payload</td>
</tr>
</tbody>
</table>

Type - packet type. Values:
3 for serial CAT
4 for serial RS485
5 for serial FSK

### Port settings bitfield:

<table>
<thead>
<tr>
<th>isopen</th>
<th>databits</th>
<th>parity</th>
<th>stopbits</th>
<th>baudrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>25</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>29</td>
<td>26</td>
<td>23</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>28</td>
<td>27</td>
<td>24</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>23</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
<td>24</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>22</td>
<td>23</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>21</td>
<td>23</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
<td>23</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>19</td>
<td>23</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>18</td>
<td>23</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
<td>23</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

### [Params] field structure for TTY packet

<table>
<thead>
<tr>
<th>Length (bits)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>isopen</td>
<td>0 = closed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = open</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If any application opens this serial port, this bit must be set to 1. If port is closed - set to 0</td>
</tr>
<tr>
<td>4</td>
<td>databits</td>
<td>data width in bits</td>
</tr>
<tr>
<td>3</td>
<td>parity</td>
<td>0 = No parity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Odd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = Even</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = Mark</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = Space</td>
</tr>
<tr>
<td>2</td>
<td>stopbits</td>
<td>0 = 1 stop-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = 1.5 stop-bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = 2 stop-bits</td>
</tr>
<tr>
<td>22</td>
<td>baudrate</td>
<td>baud rate</td>
</tr>
</tbody>
</table>
The serial port buffer size in the TILP device is 256 bytes. The interface replies with the TTY packet setting the field “Param” to the buffer's free space. Also this packet may contain data, received by the interface from the transceiver. This packet may be sent by the interface asynchronously.

```c
typedef union {
    uint32_t data;
    struct {
        uint32_t isopen :1;
        uint32_t databits :4;
        uint32_t parity :3;
        uint32_t stopbits :2;
        uint32_t baudrate :22;
    }
} tty_head_t;
```

```c
static const uint32_t bufsize = 256;
static uint32_t buff_fill = 0;
```

```c
/**
 * @brief send_serial
 * @param sd - socket descriptor
 * @param spd - serial port descriptor
 * @param pd - pointer to data
 * @param len - data len
 * @return - how many bytes was sent
 */
int send_serial(int sd, int spd, const void* pd, uint32_t len) {
    char *buff;
    uint32_t wlen;
    struct packet_t *pack;
    tty_head_t serhdr;

    if (bufsize == buff_fill) {
        return 0;
    }

    wlen = len <= (bufsize - buff_fill) ? len : (bufsize - buff_fill);

    buff = malloc(sizeof(struct packet_t) + wlen);

    if (buff == NULL) {
        return -1;
    }

    pack = (struct packet_t *) buff;
```
// Fill serial port parameters
serhdr.isopen = serialport_state(spd);
serhdr.databits = serialport.databits(spd);
serhdr.parity = serialport_parity(spd);
serhdr.stopbits = serialport_stop(spd);
serhdr.baudrate = serialport_baud(spd);

pack->params = serhdr.data;
pack->type = TTY1_TYPE;
pack->len = wlen;
pack->crc = 0;

memcpy(&buff[sizeof(struct packet_t)], pd, wlen);
pack->crc = crc8(bufsize, sizeof(packet_t)+pack->len);
retval = write(sd, buff, sizeof(packet_t)+pack->len);

if (retval != sizeof(packet_t)+pack->len) {
  wlen = 0;
}

free(buff);

buff_fill += wlen;

return wlen;

/**
 * @brief receive_serial
 * @param sd - socket descriptor
 * @param pd - pointer to buffer
 * @param len - buffer size
 * @return - how many bytes was sent
 */
int receive_serial(int sd, void* pd, uint32_t len) {
  char buff[sizeof(packet_t)+bufsize];
  uint32_t rlen;
  struct packet_t *pack;
  uint8_t crc;
  uint32_t wlen;

  retval = read(sd, buff, sizeof(buff));

  if (retval < sizeof(packet_t)) {
    return 0;
  }

  pack = (struct packet_t*)buff;
if (pack->type != TTY1_TYPE || pack->len > bufsize) {
    return 0;
}

crc = pack->crc;
pack->crc = 0;

pack->crc = crc8(buff, sizeof(packet_t)+pack->len);

if (pack->crc != crc) {
    return 0;
}

if (len < pack->len) {
    pack->len = len;
}

memcpy(pd, &buff[sizeof(packet_t)], pack->len);

buff_fill = pack->params;

return pack->len;
**CONNERR packet structure**

When connection error occurs, the interface sends the CONNERR packet with the error code.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x08</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td></td>
<td>Error code</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x00</td>
<td>contains no payload</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet</td>
</tr>
</tbody>
</table>

The interface may send next error codes:

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>No errors (NoError)</td>
</tr>
<tr>
<td>0x01</td>
<td>Multiple connections (MultipleConnection). The error occurs when more than</td>
</tr>
<tr>
<td></td>
<td>one application attempt to connect to the interface</td>
</tr>
<tr>
<td>0x02</td>
<td>Wrong password (WrongPassword). The error occurs when the password in the</td>
</tr>
<tr>
<td></td>
<td>authentication packet does not match stored in TILP device password</td>
</tr>
<tr>
<td>0x03</td>
<td>Session timeout (Timeout). The error occurs when the application did not</td>
</tr>
<tr>
<td></td>
<td>communicate with interface for more than 8 seconds</td>
</tr>
<tr>
<td>0x04</td>
<td>Unknown packet type (UnknownPacket). Data in the packet has unknown format</td>
</tr>
</tbody>
</table>
Access levels packet
Access levels packets are transmitted by the interface only. The interface supports multiple access profiles. Data exchange according to the access level is performed by the application.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x09</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>0x00</td>
<td>For Access Level packet this field value must be zeroed</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x09</td>
<td>Payload length (in bytes)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>flags</td>
<td></td>
<td>Permissions bit field</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>worktime</td>
<td></td>
<td>Active session time limit in minutes. After the expiration of the time limit, the application must close the connection</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>pausetime</td>
<td></td>
<td>Connecting pause time in minutes. After closing the connection the application must wait a specified time before the new connection. When all fields are set to 1 (full access), the time limit fields must be ignored</td>
</tr>
</tbody>
</table>
Flags bitfield described as follows:

<table>
<thead>
<tr>
<th>Length (bits)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>enable</td>
<td>Field is set to 1 when the profile is active. In case this field set to 0, the application must terminate the connection</td>
</tr>
<tr>
<td>1</td>
<td>cat</td>
<td>Field is set to 1 when the application is allowed CAT data transfer (CAT serial port). If this field set to 0, any data received through the CAT interface must be ignored</td>
</tr>
<tr>
<td>1</td>
<td>ptt</td>
<td>Field is set to 1 when the application is allowed to turn on the transmitter by PTT. If this field set to 0, the application must ignore PTT commands from user</td>
</tr>
<tr>
<td>1</td>
<td>audio</td>
<td>Field is set to 1 when the application is allowed change audio levels. If this field set to 0, the application must ignore the audio levels change commands from user</td>
</tr>
</tbody>
</table>

```c
union flags_t {
    quint8 data;
    struct {
        quint8 enable:1;
        quint8 cat:1;
        quint8 ptt:1;
        quint8 audio:1;
        quint8 unused:4;
    };
};
```

Worktime field is a uint32_t variable, contains the active session time limit in minutes. After the expiration of the time limit, the application must close the connection.

Pausetime is a uint32_t variable, contains the connecting pause time in minutes. After closing the connection the application must wait a specified time before the new connection. When all fields are set to 1 (full access), the time limit fields must be ignored.
**Firmware version packet**
This packet transmits only by the interface.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>type</td>
<td>0x0A</td>
<td>Packet type</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>params</td>
<td>0x00</td>
<td>For Firmware packet this field value must be zeroed</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>len</td>
<td>0x0C</td>
<td>Payload length (in bytes)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>crc</td>
<td></td>
<td>CRC of the packet (header and data)</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>ver</td>
<td></td>
<td>Field contains the firmware version</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>subversion</td>
<td></td>
<td>Field contains the firmware subversion</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>pointversion</td>
<td></td>
<td>Field contains the firmware modification</td>
</tr>
</tbody>
</table>
Packet aggregation

Packet segmentation

For sending multiple packets at once, the application needs to place those packets one-by-one in the TCP segment. Total size of packets must not exceed 1446 bytes (MTU is 1500, minus header size 54 bytes). The TILP device can also combine packets in the one TCP segment.

<table>
<thead>
<tr>
<th>Authorization</th>
<th>Audio</th>
<th>PTT</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TCP segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alignment

All structures must be aligned on 1-byte boundaries.

Audio stream

Audio data is sent by the UDP protocol. The audio stream is RTP packets by RFC 3550. The audio stream from the interface to the application is always stereo. The audio stream from the application to the interface is always mono. The audio stream is full duplex. The interface sends audio data on that port and address, from which audio stream was received. When the TCP connection is closed, the audio stream must be terminated.

PTT/CW stream

For sending the CW manipulation state next packet type is used:

<table>
<thead>
<tr>
<th>Num</th>
<th>Ts</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>4 bytes</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

Num – packet number
Ts – timestamp in milliseconds
Data – PTT and CW state

This stream is sending onto the separate UDP port.
The PTT and CW states are stored in the following format.

```
 7 0
 0 7
... 0
```

DTR and RTS pins of the serial port are polled by timer with the 1ms interval. When the packet is filled with 16 states, the CW state packet is sending by the UDP to the address and port, as specified in settings. After sending the packet, the previous packet is resending (for redundancy backup). The current packet will be resend after sending the next packet. The polling and sending packets start after the connection is set, and terminated after the connection is closed.
Appendix A An example of the short session

```
tcp.socket.connect()
tcp.socket.connected == OK

Authentication packet
  type = 0

Audio settings packet
  type = 2

PTT state
  type = 1

PTT state
  type = 1

Audio settings packet
  type = 2

Access levels packet
  type = 9

WTI firmware version
  type = 10

tcp.socket.close()
tcp.socket.disconnected == OK
```
Appendix B The short session with the aggregated packets

tcp.socket.connect()

tcp.socket.connected == OK

type = 0

#1 Authentication packet

#2 Audio settings packet

#3 PTT state

type = 1

#4 Firmware version

#2 Access levels packet

#3 PTT state

#1 Audio settings packet

tcp.socket.close()

tcp.socket.disconnected == OK
Appendix C The session with the audio streams example

tcp.socket.connect()

tcp.socket.connected == OK

type = 0

type = 1

type = 2

TCP

PTT state

Audio settings packet

Authentication packet

type = 9

type = 10

TCP

PTT state

Audio settings packet

Access levels packet

Access levels packet

Access level packet

Firmware version

UDP

RTP audio stream start
(transmitted during session)

The interface have detected UDP source port. Audio stream will be send onto this port

UDP

RTP audio stream start
(transmitted during session)

tcp.socket.close()

tcp.socket.disconnected == OK

RTP streams stop
Appendix D The session with the audio and CW/PTT streams

tcp.socket.connect()

tcp.socket.connected == OK

type = 0
type = 2

type = 1

 tcp.socket.close()

tcp.socket.disconnected == OK

The session with the audio and CW/PTT streams

TCP

UDP 1

UDP 2

RTP audio stream start
(transmitted during session)

The interface have detected outbound audio stream. The inbound audio stream will be send to the same port.

PTT/CW audio stream start
(transmitted during session)

Terminating RTP streams
Appendix E The session keepalive example

```python
tcp.socket.connect()

tcp.socket.connected == OK

type = 1  type = 2  type = 0

#1 PTT state  #2 Audio settings packet  #3 Authentication packet

#1 PTT state  #2 Access levels packet  #3 Firmware version

The PTT state packets is sent for the session keepalive at least 1 every 8 seconds.

type = 1

#1 PTT state

type = 1

#1 PTT state

tcp.socket.close()

tcp.socket.disconnected == OK
```
**Appendix F** The serial ports data transfer example

```python
tcp.socket.connect()
tcp.socket.connected == OK
```

```
type = 1
PPG state

#1 Audio settings packet

#2 Authentication packet

#3 Access levels packet

#4 Firmware version
```

The PTT state packets is sent for the session keepalive at least 1 every 8 seconds.

```
type = 3...5
TTY packet
(the data received from PC serial port)
```

```
type = 3...5
TTY without data
(notification about free buffer space)
```

```
type = 3...5
TTY data packet
(data, received from the interface’s UART)
```

tcp.socket.close()
tcp.socket.disconnected == OK