TSIP Decoder

Generated by Doxygen 1.8.13

Contents

Chapter 1

TSIP Decoder

1.1 Introduction

This program is designed to read and decode TSIP data packets.

1.2 Installation

Both .pro files to compile with QT and a simple Makefile are included. To compile the program simply run make from the main program directory.

1.3 Decoder

The packet is decoded first. The decoding process removes the escape characters and maps the flags, marking the packet starting and ending points.

```
while (i < *raw_len) {
  // check for DLE
  if (raw[i] == DLE) {
    // check if there are still bytes left in the buffer
    if (i + 1 < *raw_len) {
      // check the byte after the DLE
      i++;
      switch (raw[i]) {
      case ETX:
        // end of packet
        flag[*flag_count] = *processed_len;
        flag_type[*flag_count] = end_flag;
        (*flag_count)++;
        break;
      case DLE:
        // escape flag, don't do anything
        processed[(*processed_len)++] = raw[i];
        break;
      default:
        // data start
        flag[*flag_count] = *processed_len;
        flag_type[*flag_count] = start_flag;
        (*flag_count)++;
        processed[(*processed_len)++] = raw[i];
        break;
      }
      i++;
    } else {
      // end of raw data, just store it as-is and figure it out
```

```
      // later
      *hanging_dle = true;
      processed[(*processed_len)++] = raw[i];
      i++;
    }
  } else {
    processed[(*processed_len)++] = raw[i];
    i++;
  }
}
```

Once the data block is processed, the previous data buffer is checked. This is done to ensure that if a packet is spread across several blocks it is still identified and parsed.

```
    // check if the last block ended on a DLE flag
    hanging_dle_test(processed, processed_len, flag, flag_type, flag_count);
    // make sure there are some flags found
    if ((*flag_count) > 0) {
      // check if there's data from before
      if (g_inter_buffer_len > 0) {

        if (flag_type[0] == end_flag) {
          // patch the data together
          memcpy(g_inter_buffer + g_inter_buffer_len, processed,
                 flag[0] * sizeof(uint8_t));
          g_inter_buffer_len += flag[0];
          // validate and parse
          if (validate_packet(g_inter_buffer, 0, g_inter_buffer_len) == 0) {
            ParseTsipData(g_inter_buffer, g_inter_buffer_len - 4);
            packet_counter++;
          }
          // reset buffer
          g_inter_buffer_len = 0;
          g_inter_buffer_dle = false;
        }
      }
```

If a complete packet is identified, it is validated and parsed. The validation includes checking that the packet size does not exceed minimum or maximum limits, that the first byte ID is legal and a CRC checksum comparison.

```
// packet size
if (len < 6 || len > MAX_DATA_SIZE + 6) {
  return size_mismatch;
}
// ID
if (buffer[start] == DLE) {
  return illegal_id;
}

// chksum
uint32_t chksum_start = end - 4;
uint32_t chksum_int = arr_to_int(buffer + chksum_start, 4);
uint32_t chksum_chk = crc32(buffer, start, chksum_start);
if (chksum_int != chksum_chk) {
  return chksum_mismatch;
}
```

If the packet is not correct, a flag indicating the failure mode is returned. After validating the packets that might have started in the previous block, the current block is scanned for packets, which are also validated and parsed.

```
      // check the uninterrupted packets
      for (i = 0; i < (*flag_count) - 1; i++) {
        if (flag_type[i] == start_flag && flag_type[i + 1] ==
    end_flag) {
          // uninterrupted packet, validate and parse
          if (validate_packet(processed, flag[i], flag[i + 1]) == 0) {
            ParseTsipData(processed + flag[i], flag[i + 1] - flag[i] - 4);
            packet_counter++;
          }
        }
      }
```

1.4 Tests

The tests included are the output and the speed tests. Two test data files are included in "data" folder: the tsip_↩ sample with 3 valid data blocks and the tsip_sample_ext with 30000 blocks.

The output test is run on the shorter block, while the timing test is run on the longer one. The shorter output test can be run by appending a binary filename as a program parameter for running extra tests.

```
if (argc > 1) {
  // load a provided file
  test_data_load(argv[1]);
} else {
  // or a default one if none provided
  test_data_load("data/tsip_sample");
}
```

The tests are run by executing the TaskUplink200Hz() command. This function starts the threads and keeps running until the entire data set is decoded.

```
g_data_read_seq = 0;
for (uint8_t i = 0; i < N_THREADS; i++) {
  ints[i] = i;
  pthread_create(&thread[i], NULL, extract_data, &ints[i]);
}
```

The example program output is shown below:

```
Running verbose test
Loaded file data/tsip_sample
Size: 135
ID1: 165 ID2: 9 CHKSUM: 0xe85817fd
56 96 8a f1
cb 6e 13 50
8f 3c 84 44
c1 03 7c fc
3e 00 db 04
1c 05 00 76
1d 10 00 ff
ff 00 00 00
00 00 00 00
00 b1 46 d6
43 20 4e

ID1: 165 ID2: 10 CHKSUM: 0xc4c61b3e
db 04 1c 05
f9 f6 6d 44
00 00 c5 0a
76 1d 10 00
20 4e

ID1: 57 ID2: 2 CHKSUM: 0x476753ec
64 61 3e 0f
c4 c6 8a 40


Decoded 3 packets

Running a timed test
Loaded file data/tsip_sample_ext
Size: 1350000
Decoded 29535 packets
Time taken 0 seconds 9 milliseconds
```

Note that the tsip_sample_ext file actually contains 30000 packet samples, so the program may still needs some improvements in that regard. The main problem comes from the fact that a single packet may span several blocks of data read from the COM port, and separate blocks must be merged without losing data. Every possible condition must be taken into account. The decoder functionality and methods are described in the following section.

The verbose test shows that the packets are interpreted correctly. The timed test allows to compare the performance for different setup parameters, defined in advance.

```
#define N_THREADS 2

#define BLOCK_SIZE 2048

#define MAX_COM_SIZE 512

#define MAX_DATA_SIZE 256
```

The number of threads is defined by the variable N_THREADS. In theory spreading the load across several threads should increase the execution efficiency and reduce the execution times. In practice, a lot of this depends on the hardware architecture. On this machine (Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz running Debian GNU/↩ Linux) two threads performed slightly faster than a single thread, and the performance started to decrease once the number of threads exceeded 4. Among the disadvantages of a threaded approach is that it adds to complexity of the code, and any performance benefits of running separate threads could be negated by the demands of thread management itself.

The BLOCK_SIZE variable specifies the data block size to be processed by a single thread. It was found that larger block size results in a better performance. Which is natural, since it takes less effort to analyze one continuous data block rather than a series of discontinuous ones.

The MAX_COM_SIZE parameter is the maximum allowable data size that is allowed to read from the COM port in one go. While the MAX_DATA_SIZE parameter is used to define the maximum packet size, and it is used for data validation and sanity checks.

Chapter 2

uavnav_tsip

A decoder for the TSIP protocol. See the docs folder for full documentation or run doxygen to generate docs.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

File Documentation

4.1 DOXYGEN_FRONTPAGE.md File Reference

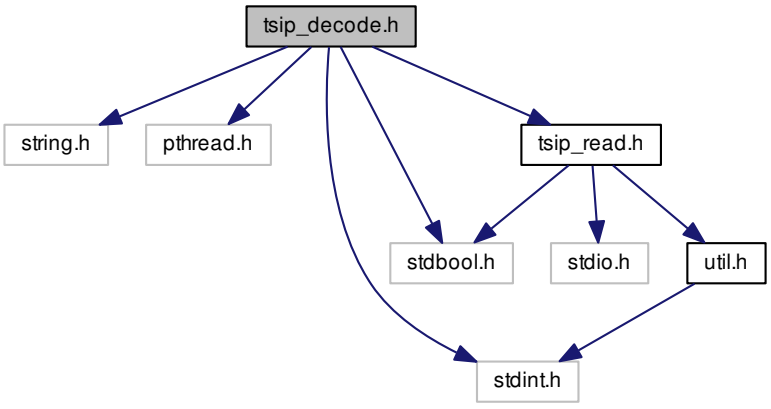
4.2 README.md File Reference

4.3 tsip_decode.h File Reference

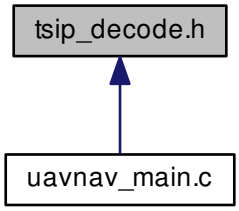
Header containing the decoding functions. The main decoder code is contained here.

```
#include "string.h"
#include <pthread.h>
#include <stdbool.h>
#include <stdint.h>
#include "tsip_read.h"
```

Include dependency graph for tsip_decode.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define N_THREADS 2

  [Setup parameters]
- #define BLOCK_SIZE 2048

  Data block size to be processed by a single thread.
- #define MAX_COM_SIZE 512

  Maximum allowable raw data size.
- #define MAX_DATA_SIZE 256

  Maximum allowable decoded data size.
- #define DEBUG false

  [Setup parameters]

Enumerations

- enum validate_error { none, size_mismatch, illegal_id, chksum_mismatch }

  Validation error types.
- enum flag_type_enum { start_flag, end_flag }

  Special character flag types.

Functions

- uint8_t validate_packet (const uint8_t *buffer, const uint32_t start, const uint32_t end)

  Validate the packet.
- void data_read (uint8_t *raw, uint32_t *raw_len, uint8_t id)

  Read the recieved raw data, multithread-protected.
- void packet_decode (uint8_t *processed, uint32_t *processed_len, uint8_t *raw, uint32_t *raw_len, uint32_t *flag, uint8_t *flag_type, uint32_t *flag_count, bool *hanging_dle)

  Process raw data.
- void hanging_dle_test (uint8_t *processed, uint32_t processed_len, uint32_t *flag, uint8_t *flag_type, uint32_t *flag_count)

  Test for special condition that the previous block ended on a DLE.
- void trailing_data_store (uint8_t *processed, uint32_t processed_len, uint32_t *flag, uint8_t *flag_type, uint32_t *flag_count, bool hanging_dle)

Store the trailing data.

- void data_parse (uint8_t *processed, uint32_t processed_len, uint32_t *flag, uint8_t *flag_type, uint32_t *flag_count, uint8_t id, bool hanging_dle)

  Parse the decoded data, multithread-protected.
- void * extract_data (void *tid)

  Data processing thread function.
- void TaskUpLink200Hz ()

  Decoder task periodic function.

4.3.1 Detailed Description

Header containing the decoding functions. The main decoder code is contained here.

4.3.2 Macro Definition Documentation

4.3.2.1 BLOCK_SIZE

```
#define BLOCK_SIZE 2048
```

Data block size to be processed by a single thread.

4.3.2.2 DEBUG

```
#define DEBUG false
```

[Setup parameters]

Debug flag.

4.3.2.3 MAX_COM_SIZE

```
#define MAX_COM_SIZE 512
```

Maximum allowable raw data size.

4.3.2.4 MAX_DATA_SIZE

```
#define MAX_DATA_SIZE 256
```

Maximum allowable decoded data size.

4.3.2.5 N_THREADS

```
#define N_THREADS 2
```

[Setup parameters]

Number of concurrent threads.

4.3.3 Enumeration Type Documentation

4.3.3.1 flag_type_enum

```
enum flag_type_enum
```

Special character flag types.

Enumerator

start_flag	
end_flag	

4.3.3.2 validate_error

```
enum validate_error
```

Validation error types.

Enumerator

none	
size_mismatch	
illegal_id	
chksum_mismatch	

4.3.4 Function Documentation

4.3.4.1 data_parse()

```
void data_parse (
            uint8_t * processed,
```

```
            uint32_t processed_len,
            uint32_t * flag,
            uint8_t * flag_type,
            uint32_t * flag_count,
            uint8_t id,
            bool hanging_dle )
```

Parse the decoded data, multithread-protected.

Format the data, run the checks on the preceding data block. Run the validation checks. Parse the data if it's correct. Store the trailing part that has not been parsed. Only one thread can parse the data at a time to make sure that it's parsed in the same order as it comes in.

Parameters

in	*processed*	Pointer to processed data.
in	*processed_len*	Size of processed data.
in	*flag*	Pointer to flags, contains flag locations.
in	*flag_type*	Types of marked flags.
in	*flag_count*	Number of flags.
in	*id*	Thread id.
in	*hanging_dle*	Hanging DLE character indicator.

Returns

[Checking previous buffer]

[Checking previous buffer]

[Checking uninterrupted packets]

[Checking uninterrupted packets]

4.3.4.2 data_read()

```
void data_read (
            uint8_t * raw,
            uint32_t * raw_len,
            uint8_t id )
```

Read the recieved raw data, multithread-protected.

Read the data. Only one thread is allowed to read data at a time.

Parameters

raw	Pointer to raw data destination.
raw_len	Pointer to the size of the data read.
id	Thread id.

Returns

Error type, 0 if no errors are found.

[Reading COM data]

[Reading COM data]

4.3.4.3 extract_data()

```
void* extract_data (
            void * tid )
```

Data processing thread function.

Thread function tasked with reading, decoding and parsing the data. Only one thread can read or parse the data at a time, while the decoding is done concurrently.

Parameters

in	*tid*	Thread id.

Returns

4.3.4.4 hanging_dle_test()

```
void hanging_dle_test (
            uint8_t * processed,
            uint32_t processed_len,
            uint32_t * flag,
            uint8_t * flag_type,
            uint32_t * flag_count )
```

Test for special condition that the previous block ended on a DLE.

Check if the previous raw data block ended on a DLE. Correct the following data block structure and data to match the trailing DLE character.

Parameters

in	*processed*	Pointer to processed data
in	*processed_len*	Size of processed data
	flag	Pointer to flags, contains flag locations
	flag_type	Types of marked flags
	flag_count	Number of flags

Returns

[Checking hanging DLE flags]

[Checking hanging DLE flags]

4.3.4.5 packet_decode()

```
void packet_decode (
            uint8_t * processed,
            uint32_t * processed_len,
            uint8_t * raw,
            uint32_t * raw_len,
            uint32_t * flag,
            uint8_t * flag_type,
            uint32_t * flag_count,
            bool * hanging_dle )
```

Process raw data.

Process the raw data, removing the escape characters and marking packet start and end points

Parameters

processed	Pointer to destination.
processed_len	Size of processed data.
raw	Pointer to source raw data.
raw_len	Size of source data.
flag	Pointer to flags, contains flag locations.
flag_type	Types of marked flags.
flag_count	Number of flags.
hanging_dle	A flag to indicate a hanging DLE character.

Returns

[Removing escape characters]

[Removing escape characters]

4.3.4.6 TaskUpLink200Hz()

```
void TaskUpLink200Hz ( )
```

Decoder task periodic function.

This function spawns several decoder threads

Returns

[Starting threads]

[Starting threads]

4.3.4.7 trailing_data_store()

```
void trailing_data_store (
            uint8_t * processed,
            uint32_t processed_len,
            uint32_t * flag,
            uint8_t * flag_type,
            uint32_t * flag_count,
            bool hanging_dle )
```

Store the trailing data.

If no DLE/ETX combo is found at the end of the packet, pass on it's trailing part to the next evaluation.

Parameters

in	*processed*	Pointer to processed data.
in	*processed_len*	Size of processed data.
in	*flag*	Pointer to flags, contains flag locations.
in	*flag_type*	Types of marked flags.
in	*flag_count*	Number of flags.
in	*hanging_dle*	Hanging DLE character indicator.

Returns

[Storing trailing data]

[Storing trailing data]

4.3.4.8 validate_packet()

```
uint8_t validate_packet (
            const uint8_t * buffer,
            const uint32_t start,
            const uint32_t end )
```

Validate the packet.

Check the packet integrity, correct size and parameters. Run a CRC check.

Parameters

buffer	Pointer to packet source.
start	Packet data start.
end	Packet data end.

Returns

Error type, 0 if no errors are found.

[Validating packet]

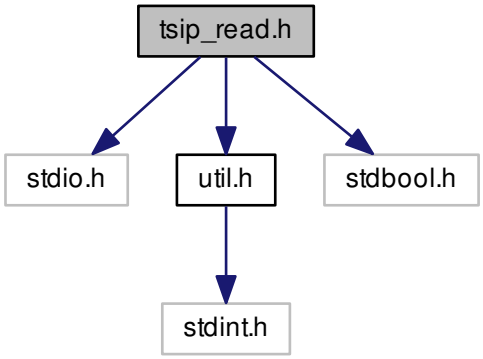
[Validating packet]

4.4 tsip_read.h File Reference

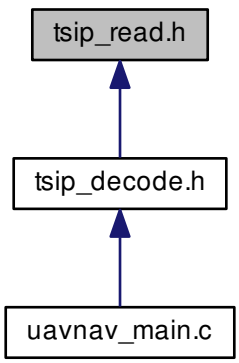
Header containing the testing functions. These functions are designed to simulate the UAV parser and the COM interface.

```
#include "stdio.h"
#include "util.h"
#include "stdbool.h"
```

Include dependency graph for tsip_read.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define DLE 0x10
  DLE flag.
- #define ETX 0x03
  ETX flag.

Functions

- uint32_t uavnComRead (uint8_t *const buffer, const uint32_t count)
  Read received raw data from COM port.
- void ParseTsipData (const uint8_t *const buffer, const uint32_t numberOfBytes)
  Parse a TSIP data.

Variables

- unsigned char * g_test_data
  Test data array.
- uint32_t g_test_data_len
  Test data length.
- uint32_t g_test_data_start
  Start of the test data.
- bool g_verbose_output
  Verbose output parameter.

4.4.1 Detailed Description

Header containing the testing functions. These functions are designed to simulate the UAV parser and the COM interface.

4.4.2 Macro Definition Documentation

4.4.2.1 DLE

```
#define DLE 0x10
```

DLE flag.

4.4.2.2 ETX

```
#define ETX 0x03
```

ETX flag.

4.4.3 Function Documentation

4.4.3.1 ParseTsipData()

```
void ParseTsipData (
            const uint8_t *const buffer,
            const uint32_t numberOfBytes )
```

Parse a TSIP data.

Parameters

in	*buffer*	Const pointer where data is located.
in	*numberOfBytes*	It is filled with the number bytes in the TSIP packet.

4.4.3.2 uavnComRead()

```
uint32_t uavnComRead (
            uint8_t *const buffer,
            const uint32_t count )
```

Read received raw data from COM port.

This is a non-blocking read function. This means that only available received data will be served. User may decide to call this function within a loop until the desired amount of data is received.

Parameters

buffer	Pointer to destination buffer where to copy received data.
count	Maximum number of bytes to be read.

Returns

Number of read bytes. This value will always be <= count.

4.4.4 Variable Documentation

4.4.4.1 g_test_data

```
unsigned char* g_test_data
```

Test data array.

4.4.4.2 g_test_data_len

```
uint32_t g_test_data_len
```

Test data length.

4.4.4.3 g_test_data_start

```
uint32_t g_test_data_start
```

Start of the test data.

4.4.4.4 g_verbose_output

```
bool g_verbose_output
```

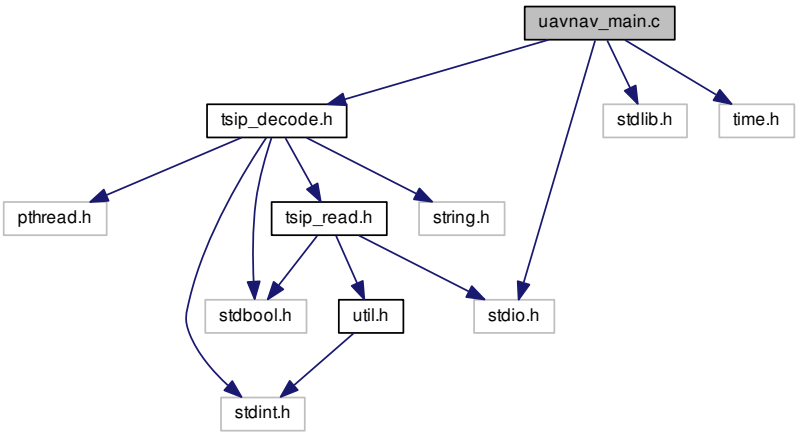
Verbose output parameter.

4.5 uavnav_main.c File Reference

Main tester application. Loads the test data files, runs tests, outputs the.

```
#include "tsip_decode.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Include dependency graph for uavnav_main.c:



Functions

- void test_data_load (const char *fname)
  Load a test data file.
- int main (int argc, char *argv[])
  Main function.

Variables

- unsigned char * g_test_data = NULL
  Test data array.
- uint32_t g_test_data_len
  Test data length.
- uint32_t g_test_data_start
  Start of the test data.
- bool g_verbose_output
  Verbose output parameter.

4.5.1 Detailed Description

Main tester application. Loads the test data files, runs tests, outputs the.

4.5.2 Function Documentation

4.5.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function.

Loads a tester data file, runs tests.

Parameters

in	*argc*	Number of arguments.
in	*argv*	Arguments.

Returns

[Loading a test file]

[Loading a test file] [Running a verbose test]

[Running a verbose test]

[Running a timed test]

[Running a timed test]

4.5.2.2 test_data_load()

```
void test_data_load (
            const char * fname )
```

Load a test data file.

Loads a tester data file into workspace.

Parameters

in	*fname*	File name.

Returns

4.5.3 Variable Documentation

4.5.3.1 g_test_data

```
unsigned char* g_test_data = NULL
```

Test data array.

4.5.3.2 g_test_data_len

```
uint32_t g_test_data_len
```

Test data length.

4.5.3.3 g_test_data_start

```
uint32_t g_test_data_start
```

Start of the test data.

4.5.3.4 g_verbose_output

```
bool g_verbose_output
```

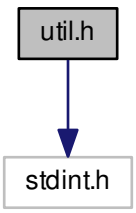
Verbose output parameter.

4.6 util.h File Reference

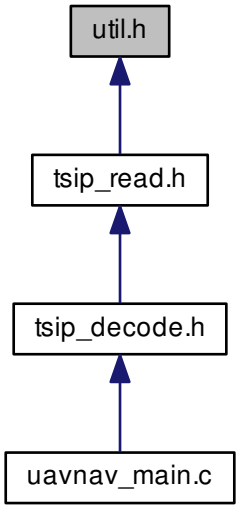
Header containing the utility functions. Various utility functions, including the CRC checker.

```
#include <stdint.h>
```

Include dependency graph for util.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define max(a, b)
- #define min(a, b)

Functions

- uint32_t crc32 (const uint8_t *buf, const uint32_t start, const uint32_t end)

  CRC calculation for TSIP packets.
- uint32_t arr_to_int (const uint8_t *in, const uint32_t n_bytes)

  Convert an array of bytes into an integer for CRC check.

Variables

- const uint32_t crc32_table [ ]

  Table of constants used for CRC calculation.

4.6.1 Detailed Description

Header containing the utility functions. Various utility functions, including the CRC checker.

4.6.2 Macro Definition Documentation

4.6.2.1 max

```
#define max(
                a,
                b )
```

Value:

```
({                                                                  \
    __typeof__(a) _a = (a);                                          \
    __typeof__(b) _b = (b);                                          \
    _a > _b ? _a : _b;                                               \
  })
```

4.6.2.2 min

```
#define min(
                a,
                b )
```

Value:

```
({                                                                  \
    __typeof__(a) _a = (a);                                          \
    __typeof__(b) _b = (b);                                          \
    _a < _b ? _a : _b;                                               \
  })
```

4.6.3 Function Documentation

4.6.3.1 arr_to_int()

```
uint32_t arr_to_int (
              const uint8_t * in,
              const uint32_t n_bytes )
```

Convert an array of bytes into an integer for CRC check.

Parameters

in	Pointer to array.
n_bytes	Number of bytes to merge

Returns

Calcutated integer .

4.6.3.2 crc32()

```
uint32_t crc32 (
            const uint8_t * buf,
            const uint32_t start,
            const uint32_t end )
```

CRC calculation for TSIP packets.

Parameters

buf	Pointer to buffer containing data to process.
start	Offset value of first data byte (within buf) to be processed.
end	Offset value of last data byte (within buf) to be processed.

Returns

Calculated CRC value.

4.6.4 Variable Documentation

4.6.4.1 crc32_table

```
const uint32_t crc32_table[]
```

Table of constants used for CRC calculation.

Index