

TFTP

Generated by Doxygen 1.8.14

Contents

1	Data Structure Index	2
1.1	Data Structures	2
2	File Index	2
2.1	File List	2
3	Data Structure Documentation	3
3.1	fblock Struct Reference	3
3.1.1	Detailed Description	3
3.1.2	Field Documentation	3
4	File Documentation	4
4.1	fblock.c File Reference	4
4.1.1	Detailed Description	4
4.1.2	Function Documentation	4
4.2	fblock.c	6
4.3	fblock.h File Reference	7
4.3.1	Detailed Description	8
4.3.2	Function Documentation	8
4.4	fblock.h	10
4.5	tftp_client.c File Reference	10
4.5.1	Detailed Description	11
4.5.2	Function Documentation	12
4.6	tftp_client.c	12
4.7	tftp_msgs.c File Reference	15
4.7.1	Detailed Description	16
4.7.2	Function Documentation	16
4.8	tftp_msgs.c	23
4.9	tftp_msgs.h File Reference	25
4.9.1	Detailed Description	26
4.9.2	Function Documentation	27
4.10	tftp_msgs.h	33
4.11	tftp_server.c File Reference	34
4.11.1	Detailed Description	35
4.12	tftp_server.c	35

Index	39
-----------------------	----

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

fblock Structure which defines a file	3
--	---

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

<code>debug_utils.c</code>	??
<code>debug_utils.h</code>	??
fblock.c Implementation of fblock.h	4
fblock.h File block read and write	7
<code>inet_utils.c</code>	??
<code>inet_utils.h</code>	??
<code>logging.h</code>	??
<code>tftp.c</code>	??
<code>tftp.h</code>	??
tftp_client.c Implementation of the TFTP client making only read requests	10
tftp_msgs.c Implementation of tftp_msgs.h	15
tftp_msgs.h Constructor for TFTP messages	25
tftp_server.c Implementation of the TFTP server serving only read requests	34

3 Data Structure Documentation

3.1 fblock Struct Reference

Structure which defines a file.

```
#include <fblock.h>
```

Data Fields

- FILE * [file](#)
Pointer to the file.
- int [block_size](#)
Predefined block size for i/o operations.
- char [mode](#)
Can be read xor write, text xor binary.
- union {
 int [written](#)
 Bytes already written (for future use)
 int [remaining](#)
 Remaining bytes to read.
};

3.1.1 Detailed Description

Structure which defines a file.

Definition at line [31](#) of file [fblock.h](#).

3.1.2 Field Documentation

3.1.2.1 mode

```
char fblock::mode
```

Can be read xor write, text xor binary.

Definition at line [34](#) of file [fblock.h](#).

The documentation for this struct was generated from the following file:

- [fblock.h](#)

4 File Documentation

4.1 fblock.c File Reference

Implementation of [fblock.h](#) .

```
#include "include/fblock.h"
#include <stdio.h>
#include <string.h>
#include "include/logging.h"
```

Macros

- `#define LOG_LEVEL LOG_INFO`

Functions

- `int get_length (FILE *f)`
Returns file length.
- `struct fblock fblock_open (char *filename, int block_size, char mode)`
Opens a file.
- `int fblock_read (struct fblock *m_fblock, char *buffer)`
Reads next block_size bytes from file.
- `int fblock_write (struct fblock *m_fblock, char *buffer, int block_size)`
Writes next block_size bytes to file.
- `int fblock_close (struct fblock *m_fblock)`

4.1.1 Detailed Description

Implementation of [fblock.h](#) .

Author

Riccardo Mancini

Definition in file [fblock.c](#).

4.1.2 Function Documentation

4.1.2.1 [fblock_open\(\)](#)

```
struct fblock fblock_open (  
    char * filename,  
    int block_size,  
    char mode )
```

Opens a file.

Parameters

<i>filename</i>	name of the file
<i>block_size</i>	size of the blocks
<i>modern</i>	mode (read, write, text, binary)

Returns

fblock structure

See also

[FBLOCK_MODE_TEXT](#)
[FBLOCK_MODE_BINARY](#)
[FBLOCK_WRITE](#)
[FBLOCK_READ](#)

Definition at line 29 of file [fblock.c](#).

4.1.2.2 fblock_read()

```
int fblock_read (
    struct fblock * m_fblock,
    char * buffer )
```

Reads next *block_size* bytes from file.

Parameters

<i>m_fblock</i>	fblock instance
<i>buffer</i>	<i>block_size</i> bytes buffer

Returns

0 in case of success, otherwise number of bytes it could not read

Definition at line 67 of file [fblock.c](#).

4.1.2.3 fblock_write()

```
int fblock_write (
    struct fblock * m_fblock,
    char * buffer,
    int block_size )
```

Writes next *block_size* bytes to file.

Parameters

<i>m_fblock</i>	fblock instance
<i>buffer</i>	block_size bytes buffer
<i>block_size</i>	if set to a non-0 value, override block_size defined in fblock.

Returns

0 in case of success, otherwise number of bytes it could not write

Definition at line 82 of file [fblock.c](#).

4.1.2.4 get_length()

```
int get_length (
    FILE * f )
```

Returns file length.

Parameters

<i>f</i>	file pointer
----------	--------------

Returns

file length in bytes

Definition at line 20 of file [fblock.c](#).

4.2 fblock.c

```
00001
00007 #define LOG_LEVEL LOG_INFO
00008
00009 #include "include/fblock.h"
00010 #include <stdio.h>
00011 #include <string.h>
00012 #include "include/logging.h"
00013
00020 int get_length(FILE *f){
00021     int size;
00022     fseek(f, 0, SEEK_END); // seek to end of file
00023     size = ftell(f); // get current file pointer
00024     fseek(f, 0, SEEK_SET); // seek back to beginning of file
00025     return size;
00026 }
00027
00028
00029 struct fblock fblock_open(char* filename, int block_size, char
mode){
00030     struct fblock m_fblock;
00031     m_fblock.block_size = block_size;
00032     m_fblock.mode = mode;
00033
00034     char mode_str[4] = "";
00035
00036     LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
00037         filename,
```

```

00038     (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "
text",
00039     (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
00040     block_size
00041 );
00042
00043 if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
00044     strcat(mode_str, "w");
00045     m_fblock.written = 0;
00046 } else {
00047     strcat(mode_str, "r");
00048 }
00049
00050 if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
00051     strcat(mode_str, "b");
00052 // text otherwise
00053
00054 m_fblock.file = fopen(filename, mode_str);
00055 if (m_fblock.file == NULL){
00056     LOG(LOG_ERR, "Error while opening file %s", filename);
00057     return m_fblock;
00058 }
00059 if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
00060     m_fblock.remaining = get_length(m_fblock.file);
00061
00062 LOG(LOG_DEBUG, "Successfully opened file");
00063 return m_fblock;
00064 }
00065
00066
00067 int fblock_read(struct fblock *m_fblock, char* buffer){
00068     int bytes_read, bytes_to_read;
00069
00070     if (m_fblock->remaining > m_fblock->block_size)
00071         bytes_to_read = m_fblock->block_size;
00072     else
00073         bytes_to_read = m_fblock->remaining;
00074
00075     bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
00076     m_fblock->remaining -= bytes_read;
00077
00078     return bytes_to_read - bytes_read;
00079 }
00080
00081
00082 int fblock_write(struct fblock *m_fblock, char* buffer, int
block_size){
00083     int written_bytes;
00084
00085     if (!block_size)
00086         block_size = m_fblock->block_size;
00087
00088     written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->
file);
00089     m_fblock->written += written_bytes;
00090     return block_size - written_bytes;
00091 }
00092
00093 int fblock_close(struct fblock *m_fblock){
00094     return fclose(m_fblock->file);
00095 }

```

4.3 fblock.h File Reference

File block read and write.

```
#include <stdio.h>
```

Data Structures

- struct [fblock](#)

Structure which defines a file.

Macros

- `#define FBLOCK_MODE_MASK 0b01`
Mask for getting text/binary mode.
- `#define FBLOCK_MODE_TEXT 0b00`
Open file in text mode.
- `#define FBLOCK_MODE_BINARY 0b01`
Open file in binary mode.
- `#define FBLOCK_RW_MASK 0b10`
Mask for getting r/w mode.
- `#define FBLOCK_READ 0b00`
Open file in read mode.
- `#define FBLOCK_WRITE 0b10`
Open file in write mode.

Functions

- `struct fblock fblock_open` (char *filename, int block_size, char mode)
Opens a file.
- `int fblock_read` (struct fblock *m_fblock, char *buffer)
Reads next block_size bytes from file.
- `int fblock_write` (struct fblock *m_fblock, char *buffer, int block_size)
Writes next block_size bytes to file.
- `int fblock_close` (struct fblock *m_fblock)

4.3.1 Detailed Description

File block read and write.

Author

Riccardo Mancini This library provides functions for reading and writing a text or binary file using a predefined block size.

Definition in file [fblock.h](#).

4.3.2 Function Documentation

4.3.2.1 fblock_open()

```
struct fblock fblock_open (  
    char * filename,  
    int block_size,  
    char mode )
```

Opens a file.

Parameters

<i>filename</i>	name of the file
<i>block_size</i>	size of the blocks
<i>modern</i>	mode (read, write, text, binary)

Returns

fblock structure

See also

[FBLOCK_MODE_TEXT](#)
[FBLOCK_MODE_BINARY](#)
[FBLOCK_WRITE](#)
[FBLOCK_READ](#)

Definition at line 29 of file [fblock.c](#).

4.3.2.2 fblock_read()

```
int fblock_read (
    struct fblock * m_fblock,
    char * buffer )
```

Reads next *block_size* bytes from file.

Parameters

<i>m_fblock</i>	fblock instance
<i>buffer</i>	<i>block_size</i> bytes buffer

Returns

0 in case of success, otherwise number of bytes it could not read

Definition at line 67 of file [fblock.c](#).

4.3.2.3 fblock_write()

```
int fblock_write (
    struct fblock * m_fblock,
    char * buffer,
    int block_size )
```

Writes next *block_size* bytes to file.

Parameters

<i>m_fblock</i>	fblock instance
<i>buffer</i>	block_size bytes buffer
<i>block_size</i>	if set to a non-0 value, override block_size defined in fblock.

Returns

0 in case of success, otherwise number of bytes it could not write

Definition at line 82 of file [fblock.c](#).

4.4 fblock.h

```

00001
00009 #ifndef FBLOCK
00010 #define FBLOCK
00011
00012 #include <stdio.h>
00013
00015 #define FBLOCK_MODE_MASK    0b01
00016
00017 #define FBLOCK_MODE_TEXT    0b00
00018
00019 #define FBLOCK_MODE_BINARY  0b01
00020
00021 #define FBLOCK_RW_MASK      0b10
00022
00023 #define FBLOCK_READ          0b00
00024
00025 #define FBLOCK_WRITE         0b10
00026
00027
00031 struct fblock{
00032     FILE *file;
00033     int block_size;
00034     char mode;
00035     union{
00036         int written;
00037         int remaining;
00038     };
00039 };
00040
00053 struct fblock fblock_open(char* filename, int block_size, char
mode);
00054
00062 int fblock_read(struct fblock *m_fblock, char* buffer);
00063
00072 int fblock_write(struct fblock *m_fblock, char* buffer, int
block_size);
00073
00074 int fblock_close(struct fblock *m_fblock);
00075
00076 #endif

```

4.5 tftp_client.c File Reference

Implementation of the TFTP client making only read requests.

```

#include "include/logging.h"
#include "include/tftp_msgs.h"
#include "include/tftp.h"
#include "include/fblock.h"
#include "include/inet_utils.h"
#include "include/debug_utils.h"
#include <arpa/inet.h>

```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Macros

- `#define LOG_LEVEL LOG_INFO`
- `#define READ_BUFFER_SIZE 80`
max stdin line length
- `#define MAX_ARGS 3`
- `#define MODE_TXT "txt"`
- `#define MODE_BIN "bin"`

Functions

- void `split_string` (char *line, char *delim, int max_argc, int *argc, char **argv)
Splits a string at each delim.
- void `print_help` ()
Prints command usage information.
- void `cmd_help` ()
Handles !help command, printing information about available commands.
- void `cmd_mode` (char *new_mode)
Handles !mode command, changing mode to either bin or text.
- int `cmd_get` (char *remote_filename, char *local_filename, char *sv_ip, int sv_port)
Handles !get command, reading file from server.
- void `cmd_quit` ()
Handles !quit command.
- int `main` (int argc, char **argv)

Variables

- char * `transfer_mode`

4.5.1 Detailed Description

Implementation of the TFTP client making only read requests.

Author

Riccardo Mancini

Definition in file [tftp_client.c](#).

4.5.2 Function Documentation

4.5.2.1 split_string()

```
void split_string (
    char * line,
    char * delim,
    int max_argc,
    int * argc,
    char ** argv )
```

Splits a string at each delim.

Trailing LF will be removed. Consecutive delimiters will be considered as one.

Parameters

<i>line</i>	the string to split
<i>delim</i>	the delimiter
<i>max_argc</i>	maximum number of parts to split the line into
<i>argc</i>	parts count (out)
<i>argv</i>	array of parts (out)

Definition at line 45 of file [tftp_client.c](#).

4.6 tftp_client.c

```
00001
00007 #define LOG_LEVEL LOG_INFO
00008
00009 #include "include/logging.h"
00010 #include "include/tftp_msgs.h"
00011 #include "include/tftp.h"
00012 #include "include/fblock.h"
00013 #include "include/inet_utils.h"
00014 #include "include/debug_utils.h"
00015 #include <arpa/inet.h>
00016 #include <sys/types.h>
00017 #include <sys/socket.h>
00018 #include <netinet/in.h>
00019 #include <string.h>
00020 #include <stdio.h>
00021 #include <stdlib.h>
00022 #include <time.h>
00023
00025 #define READ_BUFFER_SIZE 80
00026
00027 #define MAX_ARGS 3
00028
00029 #define MODE_TXT "txt"
00030 #define MODE_BIN "bin"
00031
00032 char* transfer_mode;
00033
00045 void split_string(char* line, char* delim, int max_argc, int *argc, char **argv){
00046     char *ptr;
00047     int len;
00048     char *pos;
00049
00050     // remove trailing LF
00051     if ((pos=strchr(line, '\n')) != NULL)
00052         *pos = '\0';
00053
```

```

00054  *argc = 0;
00055
00056
00057  ptr = strtok(line, delim);
00058
00059  while(ptr != NULL && *argc <= max_argc){
00060      len = strlen(ptr);
00061      if (len == 0)
00062          continue;
00063
00064      LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);
00065
00066      argv[*argc] = malloc(strlen(ptr)+1);
00067      strcpy(argv[*argc], ptr);
00068
00069      ptr = strtok(NULL, delim);
00070      (*argc)++;
00071  }
00072 }
00073
00077 void print_help(){
00078     printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
00079     printf("Example: ./tftp_client 127.0.0.1 69\n");
00080 }
00081
00082
00086 void cmd_help(){
00087     printf("Sono disponibili i seguenti comandi:\n");
00088     printf("!help --> mostra l'elenco dei comandi disponibili\n");
00089     printf("!mode {txt|bin} --> imposta il modo di trasferimento dei file (testo o binario)\n");
00090     printf("!get filename nome_locale --> richiede al server il nome del file <filename> e lo salva localmente con il nome <nome_locale>\n");
00091     printf("!quit --> termina il client\n");
00092 }
00093
00094
00098 void cmd_mode(char* new_mode){
00099     if (strcmp(new_mode, MODE_TXT) == 0){
00100         transfer_mode = TFTP_STR_NETASCII;
00101         printf("Modo di trasferimento testo configurato\n");
00102     } else if (strcmp(new_mode, MODE_BIN) == 0){
00103         transfer_mode = TFTP_STR_OCTET;
00104         printf("Modo di trasferimento binario configurato\n");
00105     } else{
00106         printf("Modo di trasferimento sconosciuto: %s. Modi disponibili: txt, bin\n", new_mode);
00107     }
00108 }
00109
00110
00114 int cmd_get(char* remote_filename, char* local_filename, char* sv_ip, int sv_port){
00115     struct sockaddr_in my_addr, sv_addr;
00116     int sd;
00117     int ret, tid;
00118     struct fblock m_fblock;
00119
00120     LOG(LOG_INFO, "Initializing...\n");
00121
00122     sd = socket(AF_INET, SOCK_DGRAM, 0);
00123     if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
00124         m_fblock = fblock_open(local_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_MODE_BINARY);
00125     else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0)
00126         m_fblock = fblock_open(local_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_MODE_TEXT);
00127     else
00128         return 2;
00129
00130     LOG(LOG_INFO, "Opening socket...");
00131
00132     sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
00133     my_addr = make_my_sockaddr_in(0);
00134     tid = bind_random_port(sd, &my_addr);
00135     if (tid == 0){
00136         LOG(LOG_ERR, "Error while binding to random port");
00137         perror("Could not bind to random port:");
00138         fbblock_close(&m_fblock);
00139         return 1;
00140     } else
00141         LOG(LOG_INFO, "Bound to port %d", tid);
00142
00143     printf("Richiesta file %s (%s) al server in corso.\n", remote_filename, transfer_mode);
00144
00145     ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
00146     if (ret != 0){
00147         fbblock_close(&m_fblock);
00148         return 8+ret;
00149     }

```

```

00150
00151     printf("Trasferimento file in corso.\n");
00152
00153     ret = tftp_receive_file(&m_fblock, sd, &sv_addr);
00154
00155     if (ret == 1){      // File not found
00156         printf("File non trovato.\n");
00157         fblock_close(&m_fblock);
00158         return 0;
00159     } else if (ret != 0){
00160         LOG(LOG_ERR, "Error while receiving file!");
00161         fblock_close(&m_fblock);
00162         return 16+ret;
00163     } else{
00164         int n_blocks = m_fblock.written/m_fblock.block_size + 1;
00165         printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);
00166         printf("Salvataggio %s completato.\n", local_filename);
00167         fblock_close(&m_fblock);
00168         return 0;
00169     }
00170
00171 }
00172
00176 void cmd_quit(){
00177     printf("Client terminato con successo\n");
00178     exit(0);
00179 }
00180
00181 int main(int argc, char** argv){
00182     char* sv_ip;
00183     short int sv_port;
00184     int ret;
00185     char *read_buffer;
00186     int cmd_argc;
00187     char *cmd_argv[MAX_ARGS];
00188
00189     //init random seed
00190     srand(time(NULL));
00191
00192     // default mode = bin
00193     transfer_mode = TFTP_STR_OCTET;
00194
00195     read_buffer = malloc(READ_BUFFER_SIZE);
00196
00197     if (argc != 3){
00198         print_help();
00199         return 1;
00200     }
00201
00202     // TODO: check args
00203     sv_ip = argv[1];
00204     sv_port = atoi(argv[2]);
00205
00206     while(1){
00207         printf("> ");
00208         fflush(stdout); // flush stdout buffer
00209         fgets(read_buffer, READ_BUFFER_SIZE, stdin);
00210         split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
00211
00212         if (cmd_argc == 0){
00213             printf("Comando non riconosciuto : ''\n");
00214             cmd_help();
00215         } else{
00216             if (strcmp(cmd_argv[0], "!mode") == 0){
00217                 if (cmd_argc == 2)
00218                     cmd_mode(cmd_argv[1]);
00219                 else
00220                     printf("Il comando richiede un solo argomento: bin o txt\n");
00221             } else if (strcmp(cmd_argv[0], "!get") == 0){
00222                 if (cmd_argc == 3){
00223                     ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
00224                     LOG(LOG_INFO, "cmd_get returned value: %d", ret);
00225                 } else{
00226                     printf("Il comando richiede due argomenti: <filename> e <nome_locale>\n");
00227                 }
00228             } else if (strcmp(cmd_argv[0], "!quit") == 0){
00229                 if (cmd_argc == 1){
00230                     cmd_quit();
00231                 } else{
00232                     printf("Il comando non richiede argomenti\n");
00233                 }
00234             } else if (strcmp(cmd_argv[0], "!help") == 0){
00235                 if (cmd_argc == 1){
00236                     cmd_help();
00237                 } else{
00238                     printf("Il comando non richiede argomenti\n");
00239                 }
00240             }
00241         }
00242     }

```

```

00240     } else {
00241         printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
00242         cmd_help();
00243     }
00244 }
00245 }
00246
00247 return ret;
00248 }

```

4.7 tftp_msgs.c File Reference

Implementation of [tftp_msgs.h](#) .

```

#include "include/tftp_msgs.h"
#include <string.h>
#include <stdio.h>
#include "include/logging.h"

```

Macros

- `#define LOG_LEVEL LOG_INFO`

Functions

- `int tftp_msg_type (char *buffer)`
Returns msg type given message buffer.
- `void tftp_msg_build_rrq (char *filename, char *mode, char *buffer)`
Builds a read request message.
- `int tftp_msg_unpack_rrq (char *buffer, int buffer_len, char *filename, char *mode)`
Unpacks a read request message.
- `int tftp_msg_get_size_rrq (char *filename, char *mode)`
Returns size in bytes of a read request message.
- `void tftp_msg_build_wrq (char *filename, char *mode, char *buffer)`
Builds a write request message.
- `int tftp_msg_unpack_wrq (char *buffer, int buffer_len, char *filename, char *mode)`
- `int tftp_msg_get_size_wrq (char *filename, char *mode)`
Returns size in bytes of a write request message.
- `void tftp_msg_build_data (int block_n, char *data, int data_size, char *buffer)`
Builds a data message.
- `int tftp_msg_unpack_data (char *buffer, int buffer_len, int *block_n, char *data, int *data_size)`
Unpacks a data message.
- `int tftp_msg_get_size_data (int data_size)`
Returns size in bytes of a data message.
- `void tftp_msg_build_ack (int block_n, char *buffer)`
Builds an acknowledgment message.
- `int tftp_msg_unpack_ack (char *buffer, int buffer_len, int *block_n)`
Unpacks an acknowledgment message.
- `int tftp_msg_get_size_ack ()`
Returns size in bytes of an acknowledgment message.
- `void tftp_msg_build_error (int error_code, char *error_msg, char *buffer)`
Builds an error message.
- `int tftp_msg_unpack_error (char *buffer, int buffer_len, int *error_code, char *error_msg)`
Unpacks an error message.
- `int tftp_msg_get_size_error (char *error_msg)`
Returns size in bytes of an error message.

4.7.1 Detailed Description

Implementation of [tftp_msgs.h](#) .

Author

Riccardo Mancini

Definition in file [tftp_msgs.c](#).

4.7.2 Function Documentation

4.7.2.1 tftp_msg_build_ack()

```
void tftp_msg_build_ack (
    int block_n,
    char * buffer )
```

Builds an acknowledgment message.

Message format:

```

2 bytes      2 bytes
-----
| 04 |      Block # |
-----
```

Parameters

<i>block_n</i>	block sequence number
<i>buffer</i>	data buffer where to build the message

Definition at line [144](#) of file [tftp_msgs.c](#).

4.7.2.2 tftp_msg_build_data()

```
void tftp_msg_build_data (
    int block_n,
    char * data,
    int data_size,
    char * buffer )
```

Builds a data message.

Message format:

```

2 bytes      2 bytes      n bytes
-----
| 03 |      Block # |      Data |
-----
```

Parameters

<i>block_n</i>	block sequence number
<i>data</i>	pointer to the buffer containing the data to be transfered
<i>data_size</i>	data buffer size
<i>buffer</i>	data buffer where to build the message

Definition at line 113 of file [tftp_msgs.c](#).

4.7.2.3 tftp_msg_build_error()

```
void tftp_msg_build_error (
    int error_code,
    char * error_msg,
    char * buffer )
```

Builds an error message.

Message format:

```

  2 bytes  2 bytes      string    1 byte
-----
| 05      | ErrorCode | ErrMsg   | 0    |
-----
```

Parameters

<i>error_code</i>	error code (from 0 to 7)
<i>error_msg</i>	error message
<i>buffer</i>	data buffer where to build the message

Definition at line 169 of file [tftp_msgs.c](#).

4.7.2.4 tftp_msg_build_rrq()

```
void tftp_msg_build_rrq (
    char * filename,
    char * mode,
    char * buffer )
```

Builds a read request message.

2 bytes string 1 byte string 1 byte

```
| 01 | Filename | 0 | Mode | 0 |
```

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")
<i>buffer</i>	data buffer where to build the message

Definition at line 18 of file [tftp_msgs.c](#).

4.7.2.5 tftp_msg_build_wrq()

```
void tftp_msg_build_wrq (
    char * filename,
    char * mode,
    char * buffer )
```

Builds a write request message.

Message format:

```

2 bytes      string    1 byte      string    1 byte
-----
|  02  | Filename |    0  |   Mode   |    0  |
-----
```

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")
<i>buffer</i>	data buffer where to build the message

Definition at line 65 of file [tftp_msgs.c](#).

4.7.2.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

Parameters

<i>data_size</i>	data buffer size
------------------	------------------

Returns

size in bytes

Definition at line 165 of file [tftp_msgs.c](#).

4.7.2.7 tftp_msg_get_size_data()

```
int tftp_msg_get_size_data (  
    int data_size )
```

Returns size in bytes of a data message.

It just sums 4 to data_size.

Parameters

<i>data_size</i>	data buffer size
------------------	------------------

Returns

size in bytes

Definition at line 140 of file [tftp_msgs.c](#).

4.7.2.8 tftp_msg_get_size_error()

```
int tftp_msg_get_size_error (  
    char * error_msg )
```

Returns size in bytes of an error message.

Parameters

<i>error_msg</i>	error message
------------------	---------------

Returns

size in bytes

Definition at line 204 of file [tftp_msgs.c](#).

4.7.2.9 tftp_msg_get_size_rrq()

```
int tftp_msg_get_size_rrq (  
    char * filename,  
    char * mode )
```

Returns size in bytes of a read request message.

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")

Returns

size in bytes

Definition at line 61 of file [tftp_msgs.c](#).

4.7.2.10 tftp_msg_get_size_wrq()

```
int tftp_msg_get_size_wrq (  
    char * filename,  
    char * mode )
```

Returns size in bytes of a write request message.

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")

Returns

size in bytes

Definition at line 109 of file [tftp_msgs.c](#).

4.7.2.11 tftp_msg_type()

```
int tftp_msg_type (  
    char * buffer )
```

Returns msg type given message buffer.

Parameters

<i>buffer</i>	the buffer
---------------	------------

Returns

message type

See also

[TFTP_TYPE_RRQ](#)
[TFTP_TYPE_WRQ](#)
[TFTP_TYPE_DATA](#)
[TFTP_TYPE_ACK](#)
[TFTP_TYPE_ERROR](#)

Definition at line 14 of file [tftp_msgs.c](#).

4.7.2.12 tftp_msg_unpack_ack()

```
int tftp_msg_unpack_ack (
    char * buffer,
    int buffer_len,
    int * block_n )
```

Unpacks an acknowledgment message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>block_n</i>	pointer where block_n will be written [out]
<i>data</i>	pointer inside buffer where the data is [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 151 of file [tftp_msgs.c](#).

4.7.2.13 tftp_msg_unpack_data()

```
int tftp_msg_unpack_data (
    char * buffer,
    int buffer_len,
    int * block_n,
    char * data,
    int * data_size )
```

Unpacks a data message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>block_n</i>	pointer where block_n will be written [out]
<i>data</i>	pointer where to copy data [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 122 of file [tftp_msgs.c](#).

4.7.2.14 tftp_msg_unpack_error()

```
int tftp_msg_unpack_error (
    char * buffer,
    int buffer_len,
    int * error_code,
    char * error_msg )
```

Unpacks an error message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>error_code</i>	pointer where <i>error_code</i> will be written [out]
<i>error_msg</i>	pointer to error message inside the message [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 178 of file [tftp_msgs.c](#).

4.7.2.15 tftp_msg_unpack_rrq()

```
int tftp_msg_unpack_rrq (
    char * buffer,
    int buffer_len,
    char * filename,
    char * mode )
```

Unpacks a read request message.

Unpacks a write request message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>filename</i>	name of the file [out]
<i>mode</i>	requested transfer mode ("netascii" or "octet") [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 27 of file [tftp_msgs.c](#).

4.8 tftp_msgs.c

```

00001
00007 #define LOG_LEVEL LOG_INFO
00008
00009 #include "include/tftp_msgs.h"
00010 #include <string.h>
00011 #include <stdio.h>
00012 #include "include/logging.h"
00013
00014 int tftp_msg_type(char *buffer){
00015     return (((int)buffer[0]) << 8) + buffer[1];
00016 }
00017
00018 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
00019     buffer[0] = 0;
00020     buffer[1] = 1;
00021     buffer += 2;
00022     strcpy(buffer, filename);
00023     buffer += strlen(filename)+1;
00024     strcpy(buffer, mode);
00025 }
00026
00027 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode){
00028     int offset = 0;
00029     if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
00030         LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
00031         return 1;
00032     }
00033
00034     offset += 2;
00035     if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00036         LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset), TFTP_MAX_FILENAME_LEN,
buffer+offset);
00037         return 3;
00038     }
00039     strcpy(filename, buffer+offset);
00040
00041     offset += strlen(filename)+1;
00042     if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00043         LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset), TFTP_MAX_MODE_LEN,
buffer+offset);
00044         return 4;
00045     }
00046     strcpy(mode, buffer+offset);
00047
00048     offset += strlen(mode)+1;
00049     if (buffer_len != offset){
00050         LOG(LOG_ERR, "Packet contains unexpected fields");
00051         return 2;
00052     }
00053     if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode, TFTP_STR_OCTET) == 0)
00054         return 0;
00055     else{
00056         LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00057         return 5;
00058     }
00059 }
00060
00061 int tftp_msg_get_size_rrq(char* filename, char* mode){
00062     return 4 + strlen(filename) + strlen(mode);
00063 }
00064
00065 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
00066     buffer[0] = 0;
00067     buffer[1] = 2;
00068     buffer += 2;
00069     strcpy(buffer, filename);
00070     buffer += strlen(filename)+1;
00071     strcpy(buffer, mode);
00072 }
00073
00074 int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename, char* mode){
00075     int offset = 0;

```



```

00076     if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
00077         LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
00078         return 1;
00079     }
00080
00081     offset += 2;
00082     if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00083         LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset), TFTP_MAX_FILENAME_LEN,
00084             buffer+offset);
00085         return 3;
00086     }
00087     strcpy(filename, buffer+offset);
00088     offset += strlen(filename)+1;
00089     if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00090         LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset), TFTP_MAX_MODE_LEN,
00091             buffer+offset);
00092         return 4;
00093     }
00094     strcpy(mode, buffer+offset);
00095     offset += strlen(mode)+1;
00096     if (buffer_len != offset){
00097         LOG(LOG_ERR, "Packet contains unexpected fields");
00098         return 2;
00099     }
00100
00101     if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode, TFTP_STR_OCTET) == 0)
00102         return 0;
00103     else{
00104         LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00105         return 5;
00106     }
00107 }
00108
00109 int tftp_msg_get_size_wrq(char* filename, char* mode){
00110     return 4 + strlen(filename) + strlen(mode);
00111 }
00112
00113 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
00114     buffer[0] = 0;
00115     buffer[1] = 3;
00116     buffer[2] = block_n >> 8;
00117     buffer[3] = block_n;
00118     buffer += 4;
00119     memcpy(buffer, data, data_size);
00120 }
00121
00122 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data, int*
00123     data_size){
00124     if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
00125         LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
00126         return 1;
00127     }
00128     if (buffer_len < 4){
00129         LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
00130         return 2;
00131     }
00132
00133     *block_n = (((int)buffer[2]) << 8) + buffer[3];
00134     *data_size = buffer_len - 4;
00135     if (*data_size > 0)
00136         memcpy(data, buffer+4, *data_size);
00137     return 0;
00138 }
00139
00140 int tftp_msg_get_size_data(int data_size){
00141     return data_size + 4;
00142 }
00143
00144 void tftp_msg_build_ack(int block_n, char* buffer){
00145     buffer[0] = 0;
00146     buffer[1] = 4;
00147     buffer[2] = block_n >> 8;
00148     buffer[3] = block_n;
00149 }
00150
00151 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
00152     if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
00153         LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
00154         return 1;
00155     }
00156
00157     if (buffer_len != 4){
00158         LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
00159         return 2;

```

```

00160     }
00161     *block_n = (((int)buffer[2]) << 8) + buffer[3];
00162     return 0;
00163 }
00164
00165 int tftp_msg_get_size_ack(){
00166     return 4;
00167 }
00168
00169 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
00170     buffer[0] = 0;
00171     buffer[1] = 5;
00172     buffer[2] = error_code >> 8;
00173     buffer[3] = error_code;
00174     buffer += 4;
00175     strcpy(buffer, error_msg);
00176 }
00177
00178 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char*
error_msg){
00179     if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
00180         LOG(LOG_ERR, "Expected ERROR message (5), found %d", tftp_msg_type(buffer));
00181         return 1;
00182     }
00183
00184     *error_code = (((int)buffer[2]) << 8) + buffer[3];
00185     if (*error_code < 0 || *error_code > 7){
00186         LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
00187         return 4;
00188     }
00189
00190     buffer += 4;
00191     if(strlen(buffer) > TFTP_MAX_ERROR_LEN){
00192         LOG(LOG_ERR, "Error string too long (%d > %d): %s", (int) strlen(buffer), TFTP_MAX_ERROR_LEN, buffer)
;
00193         return 3;
00194     }
00195
00196     strcpy(error_msg, buffer);
00197     if (buffer_len != strlen(error_msg)+5){
00198         LOG(LOG_WARN, "Packet contains unexpected fields");
00199         return 2;
00200     }
00201     return 0;
00202 }
00203
00204 int tftp_msg_get_size_error(char* error_msg){
00205     return 5 + strlen(error_msg);
00206 }

```

4.9 tftp_msgs.h File Reference

Constructor for TFTP messages.

Macros

- `#define TFTP_TYPE_RRQ 1`
Read request.
- `#define TFTP_TYPE_WRQ 2`
Write request.
- `#define TFTP_TYPE_DATA 3`
Data.
- `#define TFTP_TYPE_ACK 4`
Acknowledgment.
- `#define TFTP_TYPE_ERROR 5`
Error.
- `#define TFTP_STR_NETASCII "netascii"`
- `#define TFTP_STR_OCTET "octet"`
- `#define TFTP_MAX_FILENAME_LEN 255`
- `#define TFTP_MAX_MODE_LEN 8`
- `#define TFTP_MAX_ERROR_LEN 255`
- `#define TFTP_DATA_BLOCK 512`

Functions

- int [tftp_msg_type](#) (char *buffer)
Returns msg type given message buffer.
- void [tftp_msg_build_rrq](#) (char *filename, char *mode, char *buffer)
Builds a read request message.
- int [tftp_msg_unpack_rrq](#) (char *buffer, int buffer_len, char *filename, char *mode)
Unpacks a read request message.
- int [tftp_msg_get_size_rrq](#) (char *filename, char *mode)
Returns size in bytes of a read request message.
- void [tftp_msg_build_wrq](#) (char *filename, char *mode, char *buffer)
Builds a write request message.
- int [tftp_msg_get_size_wrq](#) (char *filename, char *mode)
Returns size in bytes of a write request message.
- void [tftp_msg_build_data](#) (int block_n, char *data, int data_size, char *buffer)
Builds a data message.
- int [tftp_msg_unpack_data](#) (char *buffer, int buffer_len, int *block_n, char *data, int *data_size)
Unpacks a data message.
- int [tftp_msg_get_size_data](#) (int data_size)
Returns size in bytes of a data message.
- void [tftp_msg_build_ack](#) (int block_n, char *buffer)
Builds an acknowledgment message.
- int [tftp_msg_unpack_ack](#) (char *buffer, int buffer_len, int *block_n)
Unpacks an acknowledgment message.
- int [tftp_msg_get_size_ack](#) ()
Returns size in bytes of an acknowledgment message.
- void [tftp_msg_build_error](#) (int error_code, char *error_msg, char *buffer)
Builds an error message.
- int [tftp_msg_unpack_error](#) (char *buffer, int buffer_len, int *error_code, char *error_msg)
Unpacks an error message.
- int [tftp_msg_get_size_error](#) (char *error_msg)
Returns size in bytes of an error message.

4.9.1 Detailed Description

Constructor for TFTP messages.

Author

Riccardo Mancini This library provides functions for building TFTP messages. There are 5 types of messages:

- 1: Read request (RRQ)
- 2: Write request (WRQ)
- 3: Data (DATA)
- 4: Acknowledgment (ACK)
- 5: Error (ERROR)

Definition in file [tftp_msgs.h](#).

4.9.2 Function Documentation

4.9.2.1 tftp_msg_build_ack()

```
void tftp_msg_build_ack (
    int block_n,
    char * buffer )
```

Builds an acknowledgment message.

Message format:

```

2 bytes      2 bytes
-----
| 04      | Block # |
-----
```

Parameters

<i>block_n</i>	block sequence number
<i>buffer</i>	data buffer where to build the message

Definition at line 144 of file [tftp_msgs.c](#).

4.9.2.2 tftp_msg_build_data()

```
void tftp_msg_build_data (
    int block_n,
    char * data,
    int data_size,
    char * buffer )
```

Builds a data message.

Message format:

```

2 bytes      2 bytes      n bytes
-----
| 03      | Block # | Data      |
-----
```

Parameters

<i>block_n</i>	block sequence number
<i>data</i>	pointer to the buffer containing the data to be transfered
<i>data_size</i>	data buffer size
<i>buffer</i>	data buffer where to build the message

Definition at line 113 of file [tftp_msgs.c](#).

4.9.2.3 tftp_msg_build_error()

```
void tftp_msg_build_error (
    int error_code,
    char * error_msg,
    char * buffer )
```

Builds an error message.

Message format:

```

  2 bytes  2 bytes      string    1 byte
-----
| 05      | ErrorCode | ErrMsg   | 0    |
-----
```

Parameters

<i>error_code</i>	error code (from 0 to 7)
<i>error_msg</i>	error message
<i>buffer</i>	data buffer where to build the message

Definition at line 169 of file [tftp_msgs.c](#).

4.9.2.4 tftp_msg_build_rrq()

```
void tftp_msg_build_rrq (
    char * filename,
    char * mode,
    char * buffer )
```

Builds a read request message.

2 bytes string 1 byte string 1 byte

```
| 01 | Filename | 0 | Mode | 0 |
```

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")
<i>buffer</i>	data buffer where to build the message

Definition at line 18 of file [tftp_msgs.c](#).

4.9.2.5 tftp_msg_build_wrq()

```
void tftp_msg_build_wrq (
    char * filename,
    char * mode,
    char * buffer )
```

Builds a write request message.

Message format:

```

2 bytes      string    1 byte      string    1 byte
-----
|  02  | Filename |    0  |   Mode   |    0  |
-----
```

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")
<i>buffer</i>	data buffer where to build the message

Definition at line 65 of file [tftp_msgs.c](#).

4.9.2.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

Parameters

<i>data_size</i>	data buffer size
------------------	------------------

Returns

size in bytes

Definition at line 165 of file [tftp_msgs.c](#).

4.9.2.7 tftp_msg_get_size_data()

```
int tftp_msg_get_size_data (
    int data_size )
```

Returns size in bytes of a data message.

It just sums 4 to data_size.

Parameters

<i>data_size</i>	data buffer size
------------------	------------------

Returns

size in bytes

Definition at line 140 of file [tftp_msgs.c](#).

4.9.2.8 tftp_msg_get_size_error()

```
int tftp_msg_get_size_error (
    char * error_msg )
```

Returns size in bytes of an error message.

Parameters

<i>error_msg</i>	error message
------------------	---------------

Returns

size in bytes

Definition at line 204 of file [tftp_msgs.c](#).

4.9.2.9 tftp_msg_get_size_rrq()

```
int tftp_msg_get_size_rrq (
    char * filename,
    char * mode )
```

Returns size in bytes of a read request message.

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")

Returns

size in bytes

Definition at line 61 of file [tftp_msgs.c](#).

4.9.2.10 tftp_msg_get_size_wrq()

```
int tftp_msg_get_size_wrq (
    char * filename,
    char * mode )
```

Returns size in bytes of a write request message.

Parameters

<i>filename</i>	name of the file
<i>mode</i>	requested transfer mode ("netascii" or "octet")

Returns

size in bytes

Definition at line 109 of file [tftp_msgs.c](#).

4.9.2.11 tftp_msg_type()

```
int tftp_msg_type (
    char * buffer )
```

Returns msg type given message buffer.

Parameters

<i>buffer</i>	the buffer
---------------	------------

Returns

message type

See also

[TFTP_TYPE_RRQ](#)
[TFTP_TYPE_WRQ](#)
[TFTP_TYPE_DATA](#)
[TFTP_TYPE_ACK](#)
[TFTP_TYPE_ERROR](#)

Definition at line 14 of file [tftp_msgs.c](#).

4.9.2.12 tftp_msg_unpack_ack()

```
int tftp_msg_unpack_ack (
    char * buffer,
    int buffer_len,
    int * block_n )
```

Unpacks an acknowledgment message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>block_n</i>	pointer where block_n will be written [out]
<i>data</i>	pointer inside buffer where the data is [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 151 of file [tftp_msgs.c](#).

4.9.2.13 tftp_msg_unpack_data()

```
int tftp_msg_unpack_data (
    char * buffer,
    int buffer_len,
    int * block_n,
    char * data,
    int * data_size )
```

Unpacks a data message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>block_n</i>	pointer where block_n will be written [out]
<i>data</i>	pointer where to copy data [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 122 of file [tftp_msgs.c](#).

4.9.2.14 tftp_msg_unpack_error()

```
int tftp_msg_unpack_error (
    char * buffer,
    int buffer_len,
    int * error_code,
    char * error_msg )
```

Unpacks an error message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>error_code</i>	pointer where error_code will be written [out]
<i>error_msg</i>	pointer to error message inside the message [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 178 of file [tftp_msgs.c](#).

4.9.2.15 tftp_msg_unpack_rrq()

```
int tftp_msg_unpack_rrq (  
    char * buffer,  
    int buffer_len,  
    char * filename,  
    char * mode )
```

Unpacks a read request message.

Unpacks a write request message.

Parameters

<i>buffer</i>	data buffer where the message to read is [in]
<i>buffer_len</i>	length of the buffer [in]
<i>filename</i>	name of the file [out]
<i>mode</i>	requested transfer mode ("netascii" or "octet") [out]

Returns

0 if success, 1 if wrong opcode, 2 otherwise

Definition at line 27 of file [tftp_msgs.c](#).

4.10 tftp_msgs.h

```
00001  
00015 #ifndef TFTP_MSGS  
00016 #define TFTP_MSGS  
00017  
00019 #define TFTP_TYPE_RRQ 1  
00020  
00021 #define TFTP_TYPE_WRQ 2  
00022  
00023 #define TFTP_TYPE_DATA 3  
00024  
00025 #define TFTP_TYPE_ACK 4  
00026
```

```

00027 #define TFTP_TYPE_ERROR 5
00028
00029 #define TFTP_STR_NETASCII "netascii"
00030 #define TFTP_STR_OCTET "octet"
00031
00032 #define TFTP_MAX_FILENAME_LEN 255
00033 #define TFTP_MAX_MODE_LEN 8
00034 #define TFTP_MAX_ERROR_LEN 255
00035
00036 #define TFTP_DATA_BLOCK 512
00037
00038
00050 int tftp_msg_type(char *buffer);
00051
00052
00065 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);
00066
00076 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode);
00077
00085 int tftp_msg_get_size_rrq(char* filename, char* mode);
00086
00102 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);
00103
00113 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode);
00114
00122 int tftp_msg_get_size_wrq(char* filename, char* mode);
00123
00140 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);
00141
00151 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data, int*
data_size);
00152
00161 int tftp_msg_get_size_data(int data_size);
00162
00177 void tftp_msg_build_ack(int block_n, char* buffer);
00178
00188 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);
00189
00198 int tftp_msg_get_size_ack();
00199
00215 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);
00216
00226 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char*
error_msg);
00227
00234 int tftp_msg_get_size_error(char* error_msg);
00235
00236 #endif

```

4.11 tftp_server.c File Reference

Implementation of the TFTP server serving only read requests.

```

#include "include/tftp_msgs.h"
#include "include/tftp.h"
#include "include/fblock.h"
#include "include/inet_utils.h"
#include "include/debug_utils.h"
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "include/logging.h"
#include <unistd.h>
#include <time.h>

```

Macros

- `#define LOG_LEVEL LOG_INFO`

Functions

- void `print_help()`
- int `send_file` (char *filename, char *mode, struct sockaddr_in *cl_addr)
- int `main` (int argc, char **argv)

4.11.1 Detailed Description

Implementation of the TFTP server serving only read requests.

Author

Riccardo Mancini

Definition in file [tftp_server.c](#).

4.12 tftp_server.c

```

00001
00007 #define LOG_LEVEL LOG_INFO
00008
00009 #include "include/tftp_msgs.h"
00010 #include "include/tftp.h"
00011 #include "include/fblock.h"
00012 #include "include/inet_utils.h"
00013 #include "include/debug_utils.h"
00014 #include <arpa/inet.h>
00015 #include <sys/types.h>
00016 #include <sys/socket.h>
00017 #include <netinet/in.h>
00018 #include <string.h>
00019 #include <stdio.h>
00020 #include <stdlib.h>
00021 #include "include/logging.h"
00022 #include <sys/types.h>
00023 #include <unistd.h>
00024 #include <time.h>
00025
00026 void print_help(){
00027     printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
00028     printf("Example: ./tftp_server 69 .\n");
00029 }
00030
00031
00032 int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
00033     struct sockaddr_in my_addr;
00034     int sd;
00035     int ret, tid;
00036     struct fblock m_fblock;
00037
00038     sd = socket(AF_INET, SOCK_DGRAM, 0);
00039     my_addr = make_my_sockaddr_in(0);
00040     tid = bind_random_port(sd, &my_addr);
00041     if (tid == 0){
00042         LOG(LOG_ERR, "Could not bind to random port");
00043         perror("Could not bind to random port:");
00044         fblock_close(&m_fblock);
00045         return 2;
00046     } else
00047         LOG(LOG_INFO, "Bound to port %d", tid);
00048
00049     if (strcmp(mode, TFTP_STR_OCTET) == 0){
00050         m_fblock = fblock_open(filename, TFTP_DATA_BLOCK, FBLOCK_READ|
FBLOCK_MODE_BINARY);

```

```

00051     } else if (strcmp(mode, TFTP_STR_NETASCII) == 0){
00052         m_fblock = fblock_open(filename, TFTP_DATA_BLOCK, FBLOCK_READ|
FBLOCK_MODE_TEXT);
00053     } else{
00054         LOG(LOG_ERR, "Unknown mode: %s", mode);
00055         return 2;
00056     }
00057
00058     if (m_fblock.file == NULL){
00059         LOG(LOG_WARN, "Error opening file. Not found?");
00060         tftp_send_error(1, "File not found.", sd, cl_addr);
00061         return 1;
00062     }
00063
00064     LOG(LOG_INFO, "Sending file...");
00065
00066     ret = tftp_send_file(&m_fblock, sd, cl_addr);
00067     fblock_close(&m_fblock);
00068
00069     if (ret != 0){
00070         LOG(LOG_ERR, "Error sending file: %d", ret);
00071         return 16+ret;
00072     }
00073
00074     LOG(LOG_INFO, "File sent successfully");
00075     return 0;
00076 }
00077
00078 int main(int argc, char** argv){
00079     short int my_port;
00080     char *directory, *filename, *path, *mode;
00081     int ret, max_msglen, type, len;
00082     char* in_buffer;
00083     unsigned int addrlen;
00084     int sd;
00085     struct sockaddr_in my_addr, cl_addr;
00086     int pid;
00087
00088     if (argc != 3){
00089         print_help();
00090         return 1;
00091     }
00092
00093     my_port = atoi(argv[1]);
00094     directory = argv[2];
00095
00096     max_msglen = TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4;
00097     in_buffer = malloc(max_msglen);
00098     addrlen = sizeof(cl_addr);
00099
00100     sd = socket(AF_INET, SOCK_DGRAM, 0);
00101     my_addr = make_my_sockaddr_in(my_port);
00102     ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
00103     if (ret == -1){
00104         LOG(LOG_ERR, "Could not bind");
00105         return 1;
00106     }
00107
00108     LOG(LOG_INFO, "Server is running");
00109
00110     while (1){
00111         len = recvfrom(sd, in_buffer, max_msglen, 0, (struct sockaddr*)&cl_addr, &addrlen);
00112         type = tftp_msg_type(in_buffer);
00113         LOG(LOG_DEBUG, "Received message with type %d", type);
00114         if (type == TFTP_TYPE_RRQ){
00115             pid = fork();
00116             if (pid != 0){
00117                 LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
00118                 continue;
00119             }
00120
00121             //init random seed
00122             srand(time(NULL));
00123
00124             filename = malloc(TFTP_MAX_FILENAME_LEN);
00125             mode = malloc(TFTP_MAX_MODE_LEN);
00126             path = malloc(TFTP_MAX_FILENAME_LEN+strlen(directory));
00127             ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);
00128             path[0] = '\0';
00129             strcat(path, directory);
00130             strcat(path, "/");
00131             strcat(path, filename);
00132
00133             LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
00134
00135             ret = send_file(path, mode, &cl_addr);
00136             if (ret != 0)

```

```
00137         LOG(LOG_WARN, "Write terminated with an error: %d", ret);
00138         break;
00139     } else{
00140         LOG(LOG_WARN, "Wrong op code: %d", type);
00141         tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
00142     }
00143 }
00144
00145 LOG(LOG_INFO, "Exiting process %d", (int) getpid());
00146 return 0;
00147 }
```


Index

- fblock, 3
 - mode, 3
- fblock.c, 4, 6
 - fblock_open, 4
 - fblock_read, 5
 - fblock_write, 5
 - get_length, 6
- fblock.h, 7, 10
 - fblock_open, 8
 - fblock_read, 9
 - fblock_write, 9
- fblock_open
 - fblock.c, 4
 - fblock.h, 8
- fblock_read
 - fblock.c, 5
 - fblock.h, 9
- fblock_write
 - fblock.c, 5
 - fblock.h, 9
- get_length
 - fblock.c, 6
- mode
 - fblock, 3
- split_string
 - tftp_client.c, 12
- tftp_client.c, 10, 12
 - split_string, 12
- tftp_msg_build_ack
 - tftp_msgs.c, 16
 - tftp_msgs.h, 27
- tftp_msg_build_data
 - tftp_msgs.c, 16
 - tftp_msgs.h, 27
- tftp_msg_build_error
 - tftp_msgs.c, 17
 - tftp_msgs.h, 28
- tftp_msg_build_rrq
 - tftp_msgs.c, 17
 - tftp_msgs.h, 28
- tftp_msg_build_wrq
 - tftp_msgs.c, 18
 - tftp_msgs.h, 28
- tftp_msg_get_size_ack
 - tftp_msgs.c, 18
 - tftp_msgs.h, 29
- tftp_msg_get_size_data
 - tftp_msgs.c, 19
 - tftp_msgs.h, 29
- tftp_msg_get_size_error
 - tftp_msgs.c, 19
 - tftp_msgs.h, 30
- tftp_msg_get_size_rrq
 - tftp_msgs.c, 19
 - tftp_msgs.h, 30
- tftp_msg_get_size_wrq
 - tftp_msgs.c, 20
 - tftp_msgs.h, 30
- tftp_msg_type
 - tftp_msgs.c, 20
 - tftp_msgs.h, 31
- tftp_msg_unpack_ack
 - tftp_msgs.c, 21
 - tftp_msgs.h, 31
- tftp_msg_unpack_data
 - tftp_msgs.c, 21
 - tftp_msgs.h, 32
- tftp_msg_unpack_error
 - tftp_msgs.c, 22
 - tftp_msgs.h, 32
- tftp_msg_unpack_rrq
 - tftp_msgs.c, 22
 - tftp_msgs.h, 33
- tftp_msgs.c, 15, 23
 - tftp_msg_build_ack, 16
 - tftp_msg_build_data, 16
 - tftp_msg_build_error, 17
 - tftp_msg_build_rrq, 17
 - tftp_msg_build_wrq, 18
 - tftp_msg_get_size_ack, 18
 - tftp_msg_get_size_data, 19
 - tftp_msg_get_size_error, 19
 - tftp_msg_get_size_rrq, 19
 - tftp_msg_get_size_wrq, 20
 - tftp_msg_type, 20
 - tftp_msg_unpack_ack, 21
 - tftp_msg_unpack_data, 21
 - tftp_msg_unpack_error, 22
 - tftp_msg_unpack_rrq, 22
- tftp_msgs.h, 25, 33
 - tftp_msg_build_ack, 27
 - tftp_msg_build_data, 27
 - tftp_msg_build_error, 28
 - tftp_msg_build_rrq, 28
 - tftp_msg_build_wrq, 28
 - tftp_msg_get_size_ack, 29
 - tftp_msg_get_size_data, 29
 - tftp_msg_get_size_error, 30
 - tftp_msg_get_size_rrq, 30
 - tftp_msg_get_size_wrq, 30
 - tftp_msg_type, 31
 - tftp_msg_unpack_ack, 31
 - tftp_msg_unpack_data, 32
 - tftp_msg_unpack_error, 32
 - tftp_msg_unpack_rrq, 33
- tftp_server.c, 34, 35