

TFTP

Generated by Doxygen 1.8.14

Contents

1	Simple TFTP implementation	2
2	Data Structure Index	2
2.1	Data Structures	2
3	File Index	3
3.1	File List	3
4	Data Structure Documentation	4
4.1	fblock Struct Reference	4
4.1.1	Detailed Description	4
4.1.2	Field Documentation	4
5	File Documentation	5
5.1	debug_utils.c File Reference	5
5.1.1	Detailed Description	5
5.1.2	Function Documentation	5
5.1.3	Variable Documentation	6
5.2	debug_utils.c	6
5.3	debug_utils.h File Reference	6
5.3.1	Detailed Description	7
5.3.2	Function Documentation	7
5.4	debug_utils.h	7
5.5	fblock.c File Reference	7
5.5.1	Detailed Description	8
5.5.2	Function Documentation	8
5.5.3	Variable Documentation	10
5.6	fblock.c	11
5.7	fblock.h File Reference	12
5.7.1	Detailed Description	13
5.7.2	Function Documentation	13

5.8	fblock.h	15
5.9	inet_utils.c File Reference	15
5.9.1	Detailed Description	16
5.9.2	Function Documentation	16
5.9.3	Variable Documentation	18
5.10	inet_utils.c	19
5.11	inet_utils.h File Reference	19
5.11.1	Detailed Description	20
5.11.2	Function Documentation	20
5.12	inet_utils.h	22
5.13	logging.h File Reference	23
5.13.1	Detailed Description	23
5.14	logging.h	24
5.15	netascii.c File Reference	25
5.15.1	Detailed Description	25
5.15.2	Function Documentation	25
5.15.3	Variable Documentation	26
5.16	netascii.c	27
5.17	netascii.h File Reference	28
5.17.1	Detailed Description	29
5.17.2	Function Documentation	29
5.18	netascii.h	30
5.19	tftp.c File Reference	30
5.19.1	Detailed Description	31
5.19.2	Function Documentation	31
5.19.3	Variable Documentation	35
5.20	tftp.c	36
5.21	tftp.h File Reference	39
5.21.1	Detailed Description	39
5.21.2	Function Documentation	40

5.22	tftp.h	44
5.23	tftp_client.c File Reference	44
5.23.1	Detailed Description	45
5.23.2	Function Documentation	45
5.23.3	Variable Documentation	46
5.24	tftp_client.c	47
5.25	tftp_msgs.c File Reference	50
5.25.1	Detailed Description	51
5.25.2	Function Documentation	51
5.25.3	Variable Documentation	59
5.26	tftp_msgs.c	60
5.27	tftp_msgs.h File Reference	62
5.27.1	Detailed Description	64
5.27.2	Macro Definition Documentation	64
5.27.3	Function Documentation	64
5.28	tftp_msgs.h	72
5.29	tftp_server.c File Reference	73
5.29.1	Detailed Description	74
5.29.2	Function Documentation	74
5.29.3	Variable Documentation	75
5.30	tftp_server.c	75

1 Simple TFTP implementation

This repository contains a simple TFTP implementation ([RFC1350](#)), made as a project for the Course in Networking @ University of Pisa.

The [project assignment](#) requires to:

1. handle only read requests from client to server (download)
2. assume that the connection is reliable (no packets can be lost or altered, no retransmission)
3. handle only File Not Found and Illegal TFTP operation errors

The server can be started with the following syntax:

```
$ ./tftp_server <listening_port> <files_directory>
```

The server is implemented as multi-process, with each new process handling a new "connection".

Example:

```
$ path/to/tftp_server 9999 test/
```

The client can be started with the following syntax:

```
$ ./tftp_client <server_IP_address> <server_port>
```

The client should also support the following operations:

- `!help`: prints an help message.
- `!mode {txt|bin}`: change preferred transfer mode to netascii or octet.
- `!get <filename> <local_filename>`: download `<filename>` from server and save it to `<local_filename>`.
- `!quit`: exit client

Example of client operation:

```
$ path/to/tftp_client 127.0.0.1 9999
> !mode txt
...
> !get test.txt my_test.txt
...
> !quit
```

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

fblock	
Structure which defines a file	4

3 File Index

3.1 File List

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

debug_utils.c	
Implementation of debug_utils.h	5
debug_utils.h	
Utility functions for debugging	6
fblock.c	
Implementation of fblock.h	7
fblock.h	
File block read and write	12
inet_utils.c	
Implementation of inet_utils.h	15
inet_utils.h	
Utility functions for managing inet addresses	19
logging.h	
Logging macro	23
netascii.c	
Implementation of netascii.h	25
netascii.h	
Conversion functions from netascii to Unix standard ASCII	28
tftp.c	
Implementation of tftp.h	30
tftp.h	
Common functions for TFTP client and server	39
tftp_client.c	
Implementation of the TFTP client that can only make read requests	44
tftp_msgs.c	
Implementation of tftp_msgs.h	50
tftp_msgs.h	
Constructor for TFTP messages	62
tftp_server.c	
Implementation of the TFTP server that can only handle read requests	73

4 Data Structure Documentation

4.1 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.
};

4.1.1 Detailed Description

Structure which defines a file.

Definition at line [40](#) of file [fblock.h](#).

4.1.2 Field Documentation

4.1.2.1 mode

```
char fblock::mode
```

Can be read xor write, text xor binary.

Definition at line [43](#) of file [fblock.h](#).

5 File Documentation

5.1 debug_utils.c File Reference

Implementation of [debug_utils.h](#).

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

Functions

- void [dump_buffer_hex](#) (char *buffer, int len)
Prints content of buffer to stdout, showing it as hex values.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.1.1 Detailed Description

Implementation of [debug_utils.h](#).

Author

Riccardo Mancini

See also

[debug_utils.h](#)

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

5.1.2 Function Documentation

5.1.2.1 dump_buffer_hex()

```
void dump_buffer_hex (
    char * buffer,
    int len )
```

Prints content of buffer to stdout, showing it as hex values.

Parameters

<i>buffer</i>	pointer to the buffer to be printed
<i>len</i>	the length (in bytes) of the buffer

Definition at line 22 of file [debug_utils.c](#).

5.1.3 Variable Documentation**5.1.3.1 LOG_LEVEL**

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.2 debug_utils.c

```

00001
00011 #include "include/debug_utils.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014 #include <stdlib.h>
00015 #include <string.h>
00016
00017
00019 extern const int LOG_LEVEL;
00020
00021
00022 void dump_buffer_hex(char* buffer, int len){
00023     char *str, tmp[4];
00024     int i;
00025
00026     str = malloc(len*3+1);
00027
00028     str[0] = '\0';
00029     for (i=0; i<len; i++){
00030         sprintf(tmp, "%02x ", (unsigned char) buffer[i]);
00031         strcat(str, tmp);
00032     }
00033
00034     LOG(LOG_DEBUG, "%s", str);
00035     free(str);
00036 }
```

5.3 debug_utils.h File Reference

Utility functions for debugging.

Functions

- void [dump_buffer_hex](#) (char *buffer, int len)
Prints content of buffer to stdout, showing it as hex values.

5.3.1 Detailed Description

Utility functions for debugging.

Author

Riccardo Mancini

At the moment, this library implements only one function for dumping a buffer using hexadecimal.

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

5.3.2 Function Documentation

5.3.2.1 dump_buffer_hex()

```
void dump_buffer_hex (
    char * buffer,
    int len )
```

Prints content of buffer to stdout, showing it as hex values.

Parameters

<i>buffer</i>	pointer to the buffer to be printed
<i>len</i>	the length (in bytes) of the buffer

Definition at line 22 of file [debug_utils.c](#).

5.4 debug_utils.h

```
00001
00011 #ifndef DEBUG_UTILS
00012 #define DEBUG_UTILS
00013
00014
00021 void dump_buffer_hex(char* buffer, int len);
00022
00023
00024 #endif
```

5.5 fblock.c File Reference

Implementation of [fblock.h](#).

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

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)
Closes a file.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.5.1 Detailed Description

Implementation of [fblock.h](#).

Author

Riccardo Mancini

See also

[fblock.h](#)

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

5.5.2 Function Documentation

5.5.2.1 [fblock_close\(\)](#)

```
int fblock_close (  
    struct fblock * m_fblock )
```

Closes a file.

Parameters

<i>m_fblock</i>	fblock instance to be closed
-----------------	------------------------------

Returns

0 in case of success, EOF in case of failure

See also

fclose

Definition at line 100 of file [fblock.c](#).

5.5.2.2 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>mode</i>	mode (read, write, text, binary)

Returns

fblock structure

See also

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

Definition at line 36 of file [fblock.c](#).

5.5.2.3 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>	block_size bytes buffer

Returns

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

Definition at line 74 of file [fblock.c](#).

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

Returns

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

Definition at line 89 of file [fblock.c](#).

5.5.2.5 get_length()

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

Returns file length.

Parameters

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

Returns

file length in bytes

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

5.5.3 Variable Documentation

5.5.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.6 fblock.c

```
00001
00011 #include "include/fblock.h"
00012 #include <stdio.h>
00013 #include <string.h>
00014 #include "include/logging.h"
00015
00016
00018 extern const int LOG_LEVEL;
00019
00020
00027 int get_length(FILE *f){
00028     int size;
00029     fseek(f, 0, SEEK_END); // seek to end of file
00030     size = ftell(f); // get current file pointer
00031     fseek(f, 0, SEEK_SET); // seek back to beginning of file
00032     return size;
00033 }
00034
00035
00036 struct fblock fblock_open(char* filename, int block_size, char
mode){
00037     struct fblock m_fblock;
00038     m_fblock.block_size = block_size;
00039     m_fblock.mode = mode;
00040
00041     char mode_str[4] = "";
00042
00043     LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
00044         filename,
00045         (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "
text",
00046         (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
00047         block_size
00048     );
00049
00050     if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
00051         strcat(mode_str, "w");
00052         m_fblock.written = 0;
00053     } else {
00054         strcat(mode_str, "r");
00055     }
00056
00057     if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
00058         strcat(mode_str, "b");
00059     // text otherwise
00060
00061     m_fblock.file = fopen(filename, mode_str);
00062     if (m_fblock.file == NULL){
00063         LOG(LOG_ERR, "Error while opening file %s", filename);
00064         return m_fblock;
00065     }
00066     if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
00067         m_fblock.remaining = get_length(m_fblock.file);
00068
00069     LOG(LOG_DEBUG, "Successfully opened file");
00070     return m_fblock;
00071 }
00072
00073
00074 int fblock_read(struct fblock *m_fblock, char* buffer){
00075     int bytes_read, bytes_to_read;
00076
00077     if (m_fblock->remaining > m_fblock->block_size)
00078         bytes_to_read = m_fblock->block_size;
00079     else
00080         bytes_to_read = m_fblock->remaining;
```

```

00081
00082     bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
00083     m_fblock->remaining -= bytes_read;
00084
00085     return bytes_to_read - bytes_read;
00086 }
00087
00088
00089 int fblock_write(struct fblock *m_fblock, char* buffer, int
    block_size){
00090     int written_bytes;
00091
00092     if (!block_size)
00093         block_size = m_fblock->block_size;
00094
00095     written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->
    file);
00096     m_fblock->written += written_bytes;
00097     return block_size - written_bytes;
00098 }
00099
00100 int fblock_close(struct fblock *m_fblock){
00101     return fclose(m_fblock->file);
00102 }

```

5.7 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)
Closes a file.

5.7.1 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](#).

5.7.2 Function Documentation

5.7.2.1 fblock_close()

```
int fblock_close (
    struct fblock * m_fblock )
```

Closes a file.

Parameters

<i>m_fblock</i>	fblock instance to be closed
-----------------	------------------------------

Returns

0 in case of success, EOF in case of failure

See also

[fclose](#)

Definition at line [100](#) of file [fblock.c](#).

5.7.2.2 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>mode</i>	mode (read, write, text, binary)

Returns

fblock structure

See also

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

Definition at line 36 of file [fblock.c](#).

5.7.2.3 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 74 of file [fblock.c](#).

5.7.2.4 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 89 of file [fblock.c](#).

5.8 fblock.h

```

00001
00011 #ifndef FBLOCK
00012 #define FBLOCK
00013
00014
00015 #include <stdio.h>
00016
00017
00019 #define FBLOCK_MODE_MASK    0b01
00020
00022 #define FBLOCK_MODE_TEXT    0b00
00023
00025 #define FBLOCK_MODE_BINARY  0b01
00026
00028 #define FBLOCK_RW_MASK      0b10
00029
00031 #define FBLOCK_READ          0b00
00032
00034 #define FBLOCK_WRITE         0b10
00035
00036
00040 struct fblock{
00041     FILE *file;
00042     int block_size;
00043     char mode;
00044     union{
00045         int written;
00046         int remaining;
00047     };
00048 };
00049
00050
00064 struct fblock fblock_open(char* filename, int block_size, char
mode);
00065
00073 int fblock_read(struct fblock *m_fblock, char* buffer);
00074
00083 int fblock_write(struct fblock *m_fblock, char* buffer, int
block_size);
00084
00093 int fblock_close(struct fblock *m_fblock);
00094
00095
00096 #endif

```

5.9 inet_utils.c File Reference

Implementation of [inet_utils.h](#).

```

#include "include/inet_utils.h"
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "include/logging.h"

```

Functions

- int [bind_random_port](#) (int socket, struct sockaddr_in *addr)
Binds socket to a random port.
- struct sockaddr_in [make_sv_sockaddr_in](#) (char *ip, int port)
Makes sockaddr_in structure given ip string and port of server.
- struct sockaddr_in [make_my_sockaddr_in](#) (int port)
Makes sockaddr_in structure of this host.
- int [sockaddr_in_cmp](#) (struct sockaddr_in sai1, struct sockaddr_in sai2)
Compares INET addresses, returning 0 in case they're equal.
- void [sockaddr_in_to_string](#) (struct sockaddr_in src, char *dst)
Converts sockaddr_in structure to string to be printed.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.9.1 Detailed Description

Implementation of [inet_utils.h](#).

Author

Riccardo Mancini

See also

[inet_utils.h](#)

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

5.9.2 Function Documentation

5.9.2.1 bind_random_port()

```
int bind_random_port (  
    int socket,  
    struct sockaddr_in * addr )
```

Binds socket to a random port.

Parameters

<i>socket</i>	socket ID
<i>addr</i>	inet addr structure

Returns

0 in case of failure, port it could bind to otherwise

See also

[FROM_PORT](#)
[TO_PORT](#)
[MAX_TRIES](#)

Definition at line 24 of file [inet_utils.c](#).

5.9.2.2 make_my_sockaddr_in()

```
struct sockaddr_in make_my_sockaddr_in (  
    int port )
```

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

Parameters

<i>port</i>	port of the server
-------------	--------------------

Returns

sockaddr_in structure this host on given port

Definition at line 55 of file [inet_utils.c](#).

5.9.2.3 make_sv_sockaddr_in()

```
struct sockaddr_in make_sv_sockaddr_in (  
    char * ip,  
    int port )
```

Makes sockaddr_in structure given ip string and port of server.

Parameters

<i>ip</i>	ip address of server
<i>port</i>	port of the server

Returns

sockaddr_in structure for the given server

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

5.9.2.4 sockaddr_in_cmp()

```
int sockaddr_in_cmp (
    struct sockaddr_in sai1,
    struct sockaddr_in sai2 )
```

Compares INET addresses, returning 0 in case they're equal.

Parameters

<i>sai1</i>	first address
<i>sai2</i>	second address

Returns

0 if they're equal, 1 otherwise

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

5.9.2.5 sockaddr_in_to_string()

```
void sockaddr_in_to_string (
    struct sockaddr_in src,
    char * dst )
```

Converts sockaddr_in structure to string to be printed.

Parameters

<i>src</i>	the input address
<i>dst</i>	the output string (must be at least MAX_SOCKADDR_STR_LEN long)

Definition at line 72 of file [inet_utils.c](#).

5.9.3 Variable Documentation

5.9.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.10 inet_utils.c

```

00001
00011 #include "include/inet_utils.h"
00012 #include <stdlib.h>
00013 #include <string.h>
00014 #include <sys/socket.h>
00015 #include <netinet/in.h>
00016 #include <arpa/inet.h>
00017 #include "include/logging.h"
00018
00019
00021 extern const int LOG_LEVEL;
00022
00023
00024 int bind_random_port(int socket, struct sockaddr_in *addr){
00025     int port, ret, i;
00026     for (i=0; i<MAX_TRIES; i++){
00027         if (i == 0) // first I generate a random one
00028             port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
00029         else //if it's not free I scan the next one
00030             port = (port-FROM_PORT+1) % (TO_PORT - FROM_PORT + 1) +
FROM_PORT;
00031
00032         LOG(LOG_DEBUG, "Trying port %d...", port);
00033
00034         addr->sin_port = htons(port);
00035         ret = bind(socket, (struct sockaddr*) addr, sizeof(*addr));
00036         if (ret != -1)
00037             return port;
00038         // consider only some errors?
00039     }
00040     LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
00041     return 0;
00042 }
00043
00044
00045 struct sockaddr_in make_sv_sockaddr_in(char* ip, int port){
00046     struct sockaddr_in addr;
00047     memset(&addr, 0, sizeof(addr));
00048     addr.sin_family = AF_INET;
00049     addr.sin_port = htons(port);
00050     inet_pton(AF_INET, ip, &addr.sin_addr);
00051     return addr;
00052 }
00053
00054
00055 struct sockaddr_in make_my_sockaddr_in(int port){
00056     struct sockaddr_in addr;
00057     memset(&addr, 0, sizeof(addr));
00058     addr.sin_family = AF_INET;
00059     addr.sin_port = htons(port);
00060     addr.sin_addr.s_addr = htonl(INADDR_ANY);
00061     return addr;
00062 }
00063
00064
00065 int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
00066     if (sai1.sin_port == sai2.sin_port && sai1.sin_addr.s_addr == sai2.sin_addr.s_addr)
00067         return 0;
00068     else
00069         return 1;
00070 }
00071
00072 void sockaddr_in_to_string(struct sockaddr_in src, char *dst){
00073     char* port_str;
00074
00075     port_str = malloc(6);
00076     sprintf(port_str, "%d", ntohs(src.sin_port));
00077
00078     if (inet_ntop(AF_INET, (void*) &src.sin_addr, dst, MAX_SOCKADDR_STR_LEN) != NULL){
00079         strcat(dst, ":");
00080         strcat(dst, port_str);
00081     } else{
00082         strcpy(dst, "ERROR");
00083     }
00084
00085     free(port_str);
00086 }

```

5.11 inet_utils.h File Reference

Utility functions for managing inet addresses.

```
#include <sys/socket.h>
#include <netinet/in.h>
```

Macros

- `#define FROM_PORT 49152`
Random port will be greater or equal to FROM_PORT.
- `#define TO_PORT 65535`
Random port will be lower or equal to TO_PORT.
- `#define MAX_TRIES 256`
Maximum number of trials before giving up opening a random port.
- `#define MAX_SOCKADDR_STR_LEN 22`
Maximum number of characters of INET address to string (eg 123.156.189.123:45678)

Functions

- `int bind_random_port (int socket, struct sockaddr_in *addr)`
Binds socket to a random port.
- `struct sockaddr_in make_sv_sockaddr_in (char *ip, int port)`
Makes sockaddr_in structure given ip string and port of server.
- `struct sockaddr_in make_my_sockaddr_in (int port)`
Makes sockaddr_in structure of this host.
- `int sockaddr_in_cmp (struct sockaddr_in sai1, struct sockaddr_in sai2)`
Compares INET addresses, returning 0 in case they're equal.
- `void sockaddr_in_to_string (struct sockaddr_in src, char *dst)`
Converts sockaddr_in structure to string to be printed.

5.11.1 Detailed Description

Utility functions for managing inet addresses.

Author

Riccardo Mancini

This library provides functions for creating `sockaddr_in` structures from IP address string and integer port number and for binding to a random port (chosen using `rand()` builtin C function).

See also

`sockaddr_in`
`rand`

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

5.11.2 Function Documentation

5.11.2.1 `bind_random_port()`

```
int bind_random_port (
    int socket,
    struct sockaddr_in * addr )
```

Binds socket to a random port.

Parameters

<i>socket</i>	socket ID
<i>addr</i>	inet addr structure

Returns

0 in case of failure, port it could bind to otherwise

See also

[FROM_PORT](#)
[TO_PORT](#)
[MAX_TRIES](#)

Definition at line 24 of file [inet_utils.c](#).

5.11.2.2 make_my_sockaddr_in()

```
struct sockaddr_in make_my_sockaddr_in (  
    int port )
```

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

Parameters

<i>port</i>	port of the server
-------------	--------------------

Returns

sockaddr_in structure this host on given port

Definition at line 55 of file [inet_utils.c](#).

5.11.2.3 make_sv_sockaddr_in()

```
struct sockaddr_in make_sv_sockaddr_in (  
    char * ip,  
    int port )
```

Makes sockaddr_in structure given ip string and port of server.

Parameters

<i>ip</i>	ip address of server
<i>port</i>	port of the server

Returns

sockaddr_in structure for the given server

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

5.11.2.4 sockaddr_in_cmp()

```
int sockaddr_in_cmp (
    struct sockaddr_in sai1,
    struct sockaddr_in sai2 )
```

Compares INET addresses, returning 0 in case they're equal.

Parameters

<i>sai1</i>	first address
<i>sai2</i>	second address

Returns

0 if they're equal, 1 otherwise

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

5.11.2.5 sockaddr_in_to_string()

```
void sockaddr_in_to_string (
    struct sockaddr_in src,
    char * dst )
```

Converts sockaddr_in structure to string to be printed.

Parameters

<i>src</i>	the input address
<i>dst</i>	the output string (must be at least MAX_SOCKADDR_STR_LEN long)

Definition at line 72 of file [inet_utils.c](#).

5.12 inet_utils.h

```
00001
00015 #ifndef INET_UTILS
00016 #define INET_UTILS
00017
00018
00019 #include <sys/socket.h>
00020 #include <netinet/in.h>
```

```
00021
00023 #define FROM_PORT 49152
00024
00026 #define TO_PORT 65535
00027
00029 #define MAX_TRIES 256
00030
00032 #define MAX_SOCKADDR_STR_LEN 22
00033
00034
00046 int bind_random_port(int socket, struct sockaddr_in *addr);
00047
00055 struct sockaddr_in make_sv_sockaddr_in(char* ip, int port);
00056
00065 struct sockaddr_in make_my_sockaddr_in(int port);
00066
00074 int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);
00075
00082 void sockaddr_in_to_string(struct sockaddr_in src, char *dst);
00083
00084
00085 #endif
```

5.13 logging.h File Reference

Logging macro.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

Macros

- #define **LOG_FATAL** (1)
- #define **LOG_ERR** (2)
- #define **LOG_WARN** (3)
- #define **LOG_INFO** (4)
- #define **LOG_DEBUG** (5)
- #define **LOG**(level, ...)

5.13.1 Detailed Description

Logging macro.

Author

Riccardo Mancini

This file contains a macro for logging in different levels.

There are 5 levels of logging:

- fatal (LOG_FATAL)
- error (LOG_ERROR)
- warning (LOG_WARN)
- information (LOG_INFO)

- debug (LOG_DEBUG)

The first three will be outputted to stderr, the latter two to stdout.

You can define a LOG_LEVEL for hiding some of the logging messages in a per-executable basis. In order to do so, you need to put

```
const int LOG_LEVEL = LOG_INFO;
```

in the file containing the main and

```
extern const int LOG_LEVEL;
```

in any other file using this macro.

Adapted from <https://stackoverflow.com/a/328660>

Definition in file `logging.h`.

5.14 logging.h

```
00001
00033 #ifndef LOGGING
00034 #define LOGGING
00035
00036
00037 #include <stdio.h>
00038 #include <sys/types.h>
00039 #include <unistd.h>
00040
00041
00042 #define LOG_FATAL      (1)
00043 #define LOG_ERR        (2)
00044 #define LOG_WARN       (3)
00045 #define LOG_INFO       (4)
00046 #define LOG_DEBUG      (5)
00047
00048
00049 #define LOG(level, ...) do { \
00050     if (level <= LOG_LEVEL) { \
00051         FILE *dbgstream; \
00052         char where[25]; \
00053         switch(level){ \
00054             case LOG_FATAL: \
00055                 dbgstream = stderr; \
00056                 fprintf(dbgstream, "[FATAL]"); \
00057                 break; \
00058             case LOG_ERR: \
00059                 dbgstream = stderr; \
00060                 fprintf(dbgstream, "[ERROR]"); \
00061                 break; \
00062             case LOG_WARN: \
00063                 dbgstream = stderr; \
00064                 fprintf(dbgstream, "[WARN ]"); \
00065                 break; \
00066             case LOG_INFO: \
00067                 dbgstream = stdout; \
00068                 fprintf(dbgstream, "[INFO ]"); \
00069                 break; \
00070             case LOG_DEBUG: \
00071                 dbgstream = stdout; \
00072                 fprintf(dbgstream, "[DEBUG]"); \
00073                 break; \
00074             } \
00075             fprintf(dbgstream, "[%5d]", (int) getpid()); \
00076             snprintf(where, 25, "%s:%d", __FILE__, __LINE__); \
00077             fprintf(dbgstream, " %-25s ", where); \
00078             fprintf(dbgstream, __VA_ARGS__); \
00079             fprintf(dbgstream, "\n"); \
00080             fflush(dbgstream); \
00081         } \
00082     } while (0)
00083
00084
00085 #endif
```

5.15 netascii.c File Reference

Implementation of [netascii.h](#).

```
#include "include/netascii.h"
#include "include/logging.h"
#include <stdio.h>
```

Functions

- int [unix2netascii](#) (char *unix_filename, char *netascii_filename)
Unix to netascii conversion.
- int [netascii2unix](#) (char *netascii_filename, char *unix_filename)
Netascii to Unix conversion.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.15.1 Detailed Description

Implementation of [netascii.h](#).

Author

Riccardo Mancini

See also

[netascii.h](#)

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

5.15.2 Function Documentation

5.15.2.1 netascii2unix()

```
int netascii2unix (
    char * netascii_filename,
    char * unix_filename )
```

Netascii to Unix conversion.

Parameters

<i>netascii_filename</i>	the filename of the input netascii file
<i>unix_filename</i>	the filename of the output Unix file

Returns

- 0 in case of success
- 1 in case of an error opening *unix_filename* file
- 2 in case of an error opening *netascii_filename* file
- 3 in case of an error writing to *unix_filename* file
- 3 in case of bad formatted netascii

Definition at line 87 of file [netascii.c](#).

5.15.2.2 unix2netascii()

```
int unix2netascii (
    char * unix_filename,
    char * netascii_filename )
```

Unix to netascii conversion.

Parameters

<i>unix_filename</i>	the filename of the input Unix file
<i>netascii_filename</i>	the filename of the output netascii file

Returns

- 0 in case of success
- 1 in case of an error opening *unix_filename* file
- 2 in case of an error opening *netascii_filename* file
- 3 in case of an error writing to *netascii_filename* file

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

5.15.3 Variable Documentation**5.15.3.1 LOG_LEVEL**

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.16 netascii.c

```

00001
00011 #include "include/netascii.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014
00015
00017 extern const int LOG_LEVEL;
00018
00019
00020 int unix2netascii(char *unix_filename, char* netascii_filename){
00021     FILE *unixf, *netasciif;
00022     char prev, tmp;
00023     int ret, result;
00024
00025     unixf = fopen(unix_filename, "r");
00026
00027     if (unixf == NULL){
00028         LOG(LOG_ERR, "Error opening file %s", unix_filename);
00029         return 1;
00030     }
00031
00032     netasciif = fopen(netascii_filename, "w");
00033
00034     if (unixf == NULL){
00035         LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00036         return 2;
00037     }
00038
00039     prev = EOF;
00040
00041     while ((tmp = (char) fgetc(unixf)) != EOF){
00042         if (tmp == '\n' && prev != '\r'){ // LF -> CRLF
00043             ret = putc('\r', netasciif);
00044             if (ret == EOF)
00045                 break;
00046
00047             ret = putc('\n', netasciif);
00048             if (ret == EOF)
00049                 break;
00050
00051         } else if (tmp == '\r'){ // CR -> CRNUL
00052             char next = (char) fgetc(unixf);
00053             if (next != '\0')
00054                 ungetc(next, unixf);
00055
00056             ret = putc('\r', netasciif);
00057             if (ret == EOF)
00058                 break;
00059
00060             ret = putc('\0', netasciif);
00061             if (ret == EOF)
00062                 break;
00063         } else{
00064             ret = putc(tmp, netasciif);
00065             if (ret == EOF)
00066                 break;
00067         }
00068
00069         prev = tmp;
00070     }
00071
00072     // Error writing to netasciif
00073     if (ret == EOF){
00074         LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
00075         result = 3;
00076     } else{
00077         LOG(LOG_INFO, "Unix file %s converted to netascii file %s", unix_filename, netascii_filename);
00078         result = 0;
00079     }
00080
00081     fclose(unixf);
00082     fclose(netasciif);
00083
00084     return result;
00085 }
00086
00087 int netascii2unix(char* netascii_filename, char *unix_filename){
00088     FILE *unixf, *netasciif;
00089     char tmp;
00090     int ret;
00091     int result = 0;
00092
00093     unixf = fopen(unix_filename, "w");
00094

```

```

00095     if (unixf == NULL){
00096         LOG(LOG_ERR, "Error opening file %s", unix_filename);
00097         return 1;
00098     }
00099
00100     netasciif = fopen(netascii_filename, "r");
00101
00102     if (unixf == NULL){
00103         LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00104         return 2;
00105     }
00106
00107     while ((tmp = (char) fgetc(netasciif)) != EOF){
00108         if (tmp == '\r'){ // CRLF -> LF ; CRNUL -> CR
00109             char next = (char) fgetc(netasciif);
00110             if (next == '\0'){ // CRNUL -> CR
00111                 ret = putc('\r', unixf);
00112                 if (ret == EOF)
00113                     break;
00114             } else if (next == '\n'){ // CRLF -> LF
00115                 ret = putc('\n', unixf);
00116                 if (ret == EOF)
00117                     break;
00118             } else if (next == EOF) { // bad format
00119                 LOG(LOG_ERR, "Bad formatted netascii: unexpected EOF after CR");
00120                 result = 4;
00121                 break;
00122             } else{ // bad format
00123                 LOG(LOG_ERR, "Bad formatted netascii: unexpected 0x%x after CR", next);
00124                 result = 4;
00125                 break;
00126             }
00127         } else{
00128             // nothing else needs to be done!
00129
00130             ret = putc(tmp, unixf);
00131             if (ret == EOF)
00132                 break;
00133         }
00134     }
00135 }
00136
00137 if (result == 0){
00138     // Error writing to unixf
00139     if (ret == EOF){
00140         LOG(LOG_ERR, "Error writing to file %s", unix_filename);
00141         result = 3;
00142     } else{
00143         LOG(LOG_INFO, "Netascii file %s converted to Unix file %s", netascii_filename, unix_filename);
00144         result = 0;
00145     }
00146 } // otherwise there was an error (4 or 5) and result was already set
00147
00148 fclose(unixf);
00149 fclose(netasciif);
00150
00151 return result;
00152 }

```

5.17 netascii.h File Reference

Conversion functions from netascii to Unix standard ASCII.

Functions

- int [unix2netascii](#) (char *unix_filename, char *netascii_filename)
Unix to netascii conversion.
- int [netascii2unix](#) (char *netascii_filename, char *unix_filename)
Netascii to Unix conversion.

5.17.1 Detailed Description

Conversion functions from netascii to Unix standard ASCII.

Author

Riccardo Mancini

This library provides two functions to convert a file from netascii to Unix standard ASCII and viceversa. In particular, there are only two differences:

- LF in Unix becomes CRLF in netascii
- CR in Unix becomes CRNUL in netascii

See also

<https://tools.ietf.org/html/rfc764>

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

5.17.2 Function Documentation

5.17.2.1 netascii2unix()

```
int netascii2unix (
    char * netascii_filename,
    char * unix_filename )
```

Netascii to Unix conversion.

Parameters

<i>netascii_filename</i>	the filename of the input netascii file
<i>unix_filename</i>	the filename of the output Unix file

Returns

- 0 in case of success
- 1 in case of an error opening *unix_filename* file
- 2 in case of an error opening *netascii_filename* file
- 3 in case of an error writing to *unix_filename* file
- 3 in case of bad formatted netascii

Definition at line 87 of file [netascii.c](#).

5.17.2.2 unix2netascii()

```
int unix2netascii (
    char * unix_filename,
    char * netascii_filename )
```

Unix to netascii conversion.

Parameters

<i>unix_filename</i>	the filename of the input Unix file
<i>netascii_filename</i>	the filename of the output netascii file

Returns

- 0 in case of success
- 1 in case of an error opening *unix_filename* file
- 2 in case of an error opening *netascii_filename* file
- 3 in case of an error writing to *netascii_filename* file

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

5.18 netascii.h

```
00001
00017 #ifndef NETASCII
00018 #define NETASCII
00019
00020
00032 int unix2netascii(char *unix_filename, char* netascii_filename);
00033
00046 int netascii2unix(char* netascii_filename, char *unix_filename);
00047
00048
00049 #endif
```

5.19 tftp.c File Reference

Implementation of [tftp.h](#).

```
#include "include/fblock.h"
#include "include/tftp_msgs.h"
#include "include/debug_utils.h"
#include "include/inet_utils.h"
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include "include/logging.h"
```

Functions

- int [tftp_send_rrq](#) (char *filename, char *mode, int sd, struct sockaddr_in *addr)
Send a RRQ message to a server.
- int [tftp_send_wrq](#) (char *filename, char *mode, int sd, struct sockaddr_in *addr)
Send a WRQ message to a server.
- int [tftp_send_error](#) (int error_code, char *error_msg, int sd, struct sockaddr_in *addr)
Send an ERROR message to the client (server).
- int [tftp_send_ack](#) (int block_n, char *out_buffer, int sd, struct sockaddr_in *addr)
Send an ACK message.
- int [tftp_receive_file](#) (struct [fblock](#) *m_fblock, int sd, struct sockaddr_in *addr)
Handle the entire workflow required to receive a file.
- int [tftp_receive_ack](#) (int *block_n, char *in_buffer, int sd, struct sockaddr_in *addr)
Receive an ACK message.
- int [tftp_send_file](#) (struct [fblock](#) *m_fblock, int sd, struct sockaddr_in *addr)
Handle the entire workflow required to send a file.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.19.1 Detailed Description

Implementation of [tftp.h](#).

Author

Riccardo Mancini

See also

[tftp.h](#)

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

5.19.2 Function Documentation

5.19.2.1 tftp_receive_ack()

```
int tftp_receive_ack (  
    int * block_n,  
    char * in_buffer,  
    int sd,  
    struct sockaddr_in * addr )
```

Receive an ACK message.

In current implementation it is only used for receiving ACKs from client.

Parameters

<i>block↔ _n</i>	[out] sequence number of the acknowledged block.
<i>in_buffer</i>	buffer to be used for receiving the ACK (useful for recycling the same buffer)
<i>sd</i>	[in] socket id of the (UDP) socket to be used to send the message
<i>addr</i>	[in] address of recipient of the ACK

Returns

- 0 in case of success
- 1 in case of failure while receiving the message
- 2 in case of address and/or port mismatch in sender sockaddr
- error unpacking ACK message otherwise (8 + result of `tftp_msg_unpack_ack`)

See also

[tftp_msg_unpack_ack](#)

Definition at line 196 of file `tftp.c`.

5.19.2.2 tftp_receive_file()

```
int tftp_receive_file (
    struct fblock * m_fblock,
    int sd,
    struct sockaddr_in * addr )
```

Handle the entire workflow required to receive a file.

In current implementation it is only used in client but it could be also used on the server side, potentially (some tweaks may be needed, though!).

Parameters

<i>m_fblock</i>	block file where to write incoming data to
<i>sd</i>	socket id of the (UDP) socket to be used to send ACK messages
<i>addr</i>	address of the recipient of ACKs

Returns

- 0 in case of success.
- 1 in case of file not found.
- 2 in case of error while sending ACK.
- 3 in case of unexpected sequence number.
- 4 in case of an error while unpacking data.
- 5 in case of an error while unpacking an incoming error message.

- 6 in case of an error while writing to the file.
- 7 in case of an error message different from File Not Found (since it is the only error available in current implementation).
- 8 in case of the incoming message is neither DATA nor ERROR.

Definition at line 103 of file [tftp.c](#).

5.19.2.3 tftp_send_ack()

```
int tftp_send_ack (
    int block_n,
    char * out_buffer,
    int sd,
    struct sockaddr_in * addr )
```

Send an ACK message.

In current implementation it is only used for sending ACKs from client to server.

Parameters

<i>block_n</i>	sequence number of the block to be acknowledged.
<i>out_buffer</i>	buffer to be used for sending the ACK (useful for recycling the same buffer)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of recipient of the ACK

Returns

0 in case of success, 1 otherwise

Definition at line 86 of file [tftp.c](#).

5.19.2.4 tftp_send_error()

```
int tftp_send_error (
    int error_code,
    char * error_msg,
    int sd,
    struct sockaddr_in * addr )
```

Send an ERROR message to the client (server).

In current implementation it is only used for sending File Not Found and Illegal TFTP Operation errors to clients.

Parameters

<i>error_code</i>	the code of the error (must be within 0 and 7)
<i>error_msg</i>	the message explaining the error
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the client (server)

Returns

0 in case of success, 1 otherwise

Definition at line 66 of file [tftp.c](#).

5.19.2.5 tftp_send_file()

```
int tftp_send_file (
    struct fblock * m_fblock,
    int sd,
    struct sockaddr_in * addr )
```

Handle the entire workflow required to send a file.

In current implementation it is only used in server but it could be also used on the client side, potentially (some tweaks may be needed, though!).

Parameters

<i>m_fblock</i>	block file where to read incoming data from
<i>sd</i>	socket id of the (UDP) socket to be used to send DATA messages
<i>addr</i>	address of the recipient of the file

Returns

- 0 in case of success.
- 1 in case of error sending a packet.
- 2 in case of error while receiving the ack.
- 3 in case of unexpected sequence number in ack.
- 4 in case of an error while unpacking data.

Definition at line 228 of file [tftp.c](#).

5.19.2.6 tftp_send_rrq()

```
int tftp_send_rrq (
    char * filename,
    char * mode,
    int sd,
    struct sockaddr_in * addr )
```

Send a RRQ message to a server.

Parameters

<i>filename</i>	the name of the requested file
<i>mode</i>	the desired mode of transfer (netascii or octet)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the server

Returns

0 in case of success, 1 otherwise

See also

[TFTP_STR_NETASCII](#)

[TFTP_STR_OCTET](#)

Definition at line 26 of file [tftp.c](#).

5.19.2.7 tftp_send_wrq()

```
int tftp_send_wrq (
    char * filename,
    char * mode,
    int sd,
    struct sockaddr_in * addr )
```

Send a WRQ message to a server.

Do not used in current implementation.

Parameters

<i>filename</i>	the name of the requested file
<i>mode</i>	the desired mode of transfer (netascii or octet)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the server

Returns

0 in case of success, 1 otherwise

See also

[TFTP_STR_NETASCII](#)

[TFTP_STR_OCTET](#)

Definition at line 46 of file [tftp.c](#).

5.19.3 Variable Documentation**5.19.3.1 LOG_LEVEL**

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.20 tftp.c

```

00001
00011 #include "include/fblock.h"
00012 #include "include/tftp_msgs.h"
00013 #include "include/debug_utils.h"
00014 #include "include/inet_utils.h"
00015 #include <arpa/inet.h>
00016 #include <sys/socket.h>
00017 #include <netinet/in.h>
00018 #include <stdlib.h>
00019 #include "include/logging.h"
00020
00021
00023 extern const int LOG_LEVEL;
00024
00025
00026 int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00027     int msglen, len;
00028     char *out_buffer;
00029
00030     msglen = tftp_msg_get_size_rrq(filename, mode);
00031     out_buffer = malloc(msglen);
00032
00033     tftp_msg_build_rrq(filename, mode, out_buffer);
00034     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00035     if (len != msglen){
00036         LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
00037         perror("Error");
00038         return 1;
00039     }
00040
00041     free(out_buffer);
00042     return 0;
00043 }
00044
00045
00046 int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00047     int msglen, len;
00048     char *out_buffer;
00049
00050     msglen = tftp_msg_get_size_wrq(filename, mode);
00051     out_buffer = malloc(msglen);
00052
00053     tftp_msg_build_wrq(filename, mode, out_buffer);
00054     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00055     if (len != msglen){
00056         LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
00057         perror("Error");
00058         return 1;
00059     }
00060
00061     free(out_buffer);
00062     return 0;
00063 }
00064
00065
00066 int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in *addr){
00067     int msglen, len;
00068     char *out_buffer;
00069
00070     msglen = tftp_msg_get_size_error(error_msg);
00071     out_buffer = malloc(msglen);
00072
00073     tftp_msg_build_error(error_code, error_msg, out_buffer);
00074     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00075     if (len != msglen){
00076         LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
00077         perror("Error");
00078         return 1;
00079     }
00080
00081     free(out_buffer);
00082     return 0;
00083 }
00084
00085
00086 int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *addr){
00087     int msglen, len;
00088
00089     msglen = tftp_msg_get_size_ack();
00090     tftp_msg_build_ack(block_n, out_buffer);
00091     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00092
00093     if (len != msglen){
00094         LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);

```

```

00095     perror("Error");
00096     return 1;
00097 }
00098
00099     return 0;
00100 }
00101
00102
00103 int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
00104     char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[
TFTP_DATA_BLOCK], out_buffer[4];
00105     int exp_block_n, rcv_block_n;
00106     int len, data_size, ret, type;
00107     unsigned int addrlen;
00108     struct sockaddr_in cl_addr, orig_cl_addr;
00109
00110     // init expected block number
00111     exp_block_n = 1;
00112
00113     addrlen = sizeof(cl_addr);
00114
00115     do{
00116         LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
00117         // TODO: check client == server ?
00118         len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(
TFTP_DATA_BLOCK), 0, (struct sockaddr*)&cl_addr, &addrlen);
00119         if (exp_block_n == 1){ // first block -> I need to save servers TID (aka its "original" sockaddr)
00120             char addr_str[MAX_SOCKADDR_STR_LEN];
00121             sockaddr_in_to_string(cl_addr, addr_str);
00122
00123             if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
00124                 LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
00125                 continue;
00126             } else{
00127                 LOG(LOG_INFO, "Receiving packets from %s", addr_str);
00128                 orig_cl_addr = cl_addr;
00129             }
00130         } else{
00131             if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
00132                 char addr_str[MAX_SOCKADDR_STR_LEN];
00133                 sockaddr_in_to_string(cl_addr, addr_str);
00134                 LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
00135                 continue;
00136             } else{
00137                 LOG(LOG_DEBUG, "Sender is the same!");
00138             }
00139         }
00140
00141         type = tftp_msg_type(in_buffer);
00142         if (type == TFTP_TYPE_ERROR){
00143             int error_code;
00144             char error_msg[TFTP_MAX_ERROR_LEN];
00145
00146             ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
00147             if (ret != 0){
00148                 LOG(LOG_ERR, "Error unpacking error msg");
00149                 return 5;
00150             }
00151
00152             if (error_code == 1){
00153                 LOG(LOG_INFO, "File not found");
00154                 return 1;
00155             } else{
00156                 LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
00157                 return 7;
00158             }
00159         } else if (type != TFTP_TYPE_DATA){
00160             LOG(LOG_ERR, "Received packet of type %d, expecting DATA or ERROR.", type);
00161             return 8;
00162         }
00163
00164         ret = tftp_msg_unpack_data(in_buffer, len, &rcv_block_n, data, &data_size);
00165
00166         if (ret != 0){
00167             LOG(LOG_ERR, "Error unpacking data: %d", ret);
00168             return 4;
00169         }
00170
00171         if (rcv_block_n != exp_block_n){
00172             LOG(LOG_ERR, "Received unexpected block_n: rcv_block_n = %d != %d = exp_block_n", rcv_block_n,
exp_block_n);
00173             return 3;
00174         }
00175
00176         exp_block_n++;
00177
00178

```



```

00179     LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);
00180
00181     if (data_size != 0){
00182         if (fblock_write(m_fblock, data, data_size))
00183             return 6;
00184     }
00185
00186     LOG(LOG_DEBUG, "Sending ack");
00187
00188     if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
00189         return 2;
00190
00191 } while(data_size == TFTP_DATA_BLOCK);
00192 return 0;
00193 }
00194
00195
00196 int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *addr){
00197     int msglen, len, ret;
00198     unsigned int addrlen;
00199     struct sockaddr_in cl_addr;
00200
00201     msglen = tftp_msg_get_size_ack();
00202     addrlen = sizeof(cl_addr);
00203
00204     len = recvfrom(sd, in_buffer, msglen, 0, (struct sockaddr*)&cl_addr, &addrlen);
00205
00206     if (sockaddr_in_cmp(*addr, cl_addr) != 0){
00207         char str_addr[MAX_SOCKADDR_STR_LEN];
00208         sockaddr_in_to_string(cl_addr, str_addr);
00209         LOG(LOG_WARN, "Message is coming from unexpected source: %s", str_addr);
00210         return 2;
00211     }
00212
00213     if (len != msglen){
00214         LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
00215         return 1;
00216     }
00217
00218     ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
00219     if (ret != 0){
00220         LOG(LOG_ERR, "Error unpacking ack: %d", ret);
00221         return 8+ret;
00222     }
00223
00224     return 0;
00225 }
00226
00227
00228 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
00229     char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[
00230         TFTP_MAX_DATA_MSG_SIZE];
00231     int block_n, rcv_block_n;
00232     int len, data_size, msglen, ret;
00233
00234     // init sequence number
00235     block_n = 1;
00236
00237     do{
00238         LOG(LOG_DEBUG, "Sending part %d", block_n);
00239
00240         if (m_fblock->remaining > TFTP_DATA_BLOCK)
00241             data_size = TFTP_DATA_BLOCK;
00242         else
00243             data_size = m_fblock->remaining;
00244
00245         if (data_size != 0)
00246             fblock_read(m_fblock, data);
00247
00248         LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);
00249
00250         msglen = tftp_msg_get_size_data(data_size);
00251         tftp_msg_build_data(block_n, data, data_size, out_buffer);
00252
00253         // dump_buffer_hex(out_buffer, msglen);
00254
00255         len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*)addr, sizeof(*addr));
00256         if (len != msglen){
00257             return 1;
00258         }
00259
00260         LOG(LOG_DEBUG, "Waiting for ack");
00261
00262         ret = tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr);
00263
00264         if (ret == 2){ //unexpected source

```

```

00265         continue;
00266     } else if (ret != 0){
00267         LOG(LOG_ERR, "Error receiving ack: %d", ret);
00268         return 2;
00269     }
00270
00271     if (rcv_block_n != block_n){
00272         LOG(LOG_ERR, "Received wrong block n: received %d != expected %d", rcv_block_n, block_n);
00273         return 3;
00274     }
00275
00276     block_n++;
00277
00278     } while(data_size == TFTP_DATA_BLOCK);
00279     return 0;
00280 }

```

5.21 tftp.h File Reference

Common functions for TFTP client and server.

```

#include <sys/socket.h>
#include <netinet/in.h>
#include "fblock.h"

```

Functions

- int [tftp_send_rrq](#) (char *filename, char *mode, int sd, struct sockaddr_in *addr)
Send a RRQ message to a server.
- int [tftp_send_wrq](#) (char *filename, char *mode, int sd, struct sockaddr_in *addr)
Send a WRQ message to a server.
- int [tftp_send_error](#) (int error_code, char *error_msg, int sd, struct sockaddr_in *addr)
Send an ERROR message to the client (server).
- int [tftp_send_ack](#) (int block_n, char *out_buffer, int sd, struct sockaddr_in *addr)
Send an ACK message.
- int [tftp_receive_file](#) (struct [fblock](#) *m_fblock, int sd, struct sockaddr_in *addr)
Handle the entire workflow required to receive a file.
- int [tftp_receive_ack](#) (int *block_n, char *in_buffer, int sd, struct sockaddr_in *addr)
Receive an ACK message.
- int [tftp_send_file](#) (struct [fblock](#) *m_fblock, int sd, struct sockaddr_in *addr)
Handle the entire workflow required to send a file.

5.21.1 Detailed Description

Common functions for TFTP client and server.

Author

Riccardo Mancini

This library provides functions for sending requests, errors and exchanging files using the TFTP protocol.

Even though the project assignment does not require the client to send files to the server, I still decided to include those functions in a common library in case in the future I decide to complete the TFTP implementation.

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

5.21.2 Function Documentation

5.21.2.1 tftp_receive_ack()

```
int tftp_receive_ack (
    int * block_n,
    char * in_buffer,
    int sd,
    struct sockaddr_in * addr )
```

Receive an ACK message.

In current implementation it is only used for receiving ACKs from client.

Parameters

<i>block_n</i>	[out] sequence number of the acknowledged block.
<i>in_buffer</i>	buffer to be used for receiving the ACK (useful for recycling the same buffer)
<i>sd</i>	[in] socket id of the (UDP) socket to be used to send the message
<i>addr</i>	[in] address of recipient of the ACK

Returns

- 0 in case of success
- 1 in case of failure while receiving the message
- 2 in case of address and/or port mismatch in sender sockaddr
- error unpacking ACK message otherwise (8 + result of tftp_msg_unpack_ack)

See also

[tftp_msg_unpack_ack](#)

Definition at line 196 of file [tftp.c](#).

5.21.2.2 tftp_receive_file()

```
int tftp_receive_file (
    struct fblock * m_fblock,
    int sd,
    struct sockaddr_in * addr )
```

Handle the entire workflow required to receive a file.

In current implementation it is only used in client but it could be also used on the server side, potentially (some tweaks may be needed, though!).

Parameters

<i>m_fblock</i>	block file where to write incoming data to
<i>sd</i>	socket id of the (UDP) socket to be used to send ACK messages
<i>addr</i>	address of the recipient of ACKs

Returns

- 0 in case of success.
- 1 in case of file not found.
- 2 in case of error while sending ACK.
- 3 in case of unexpected sequence number.
- 4 in case of an error while unpacking data.
- 5 in case of an error while unpacking an incoming error message.
- 6 in case of an error while writing to the file.
- 7 in case of an error message different from File Not Found (since it is the only error available in current implementation).
- 8 in case of the incoming message is neither DATA nor ERROR.

Definition at line 103 of file [tftp.c](#).

5.21.2.3 tftp_send_ack()

```
int tftp_send_ack (
    int block_n,
    char * out_buffer,
    int sd,
    struct sockaddr_in * addr )
```

Send an ACK message.

In current implementation it is only used for sending ACKs from client to server.

Parameters

<i>block_n</i>	sequence number of the block to be acknowledged.
<i>out_buffer</i>	buffer to be used for sending the ACK (useful for recycling the same buffer)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of recipient of the ACK

Returns

0 in case of success, 1 otherwise

Definition at line 86 of file [tftp.c](#).

5.21.2.4 tftp_send_error()

```
int tftp_send_error (
    int error_code,
    char * error_msg,
    int sd,
    struct sockaddr_in * addr )
```

Send an ERROR message to the client (server).

In current implementation it is only used for sending File Not Found and Illegal TFTP Operation errors to clients.

Parameters

<i>error_code</i>	the code of the error (must be within 0 and 7)
<i>error_msg</i>	the message explaining the error
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the client (server)

Returns

0 in case of success, 1 otherwise

Definition at line 66 of file [tftp.c](#).

5.21.2.5 tftp_send_file()

```
int tftp_send_file (
    struct fblock * m_fblock,
    int sd,
    struct sockaddr_in * addr )
```

Handle the entire workflow required to send a file.

In current implementation it is only used in server but it could be also used on the client side, potentially (some tweaks may be needed, though!).

Parameters

<i>m_fblock</i>	block file where to read incoming data from
<i>sd</i>	socket id of the (UDP) socket to be used to send DATA messages
<i>addr</i>	address of the recipient of the file

Returns

- 0 in case of success.
- 1 in case of error sending a packet.
- 2 in case of error while receiving the ack.
- 3 in case of unexpected sequence number in ack.

- 4 in case of an error while unpacking data.

Definition at line 228 of file [tftp.c](#).

5.21.2.6 tftp_send_rrq()

```
int tftp_send_rrq (
    char * filename,
    char * mode,
    int sd,
    struct sockaddr_in * addr )
```

Send a RRQ message to a server.

Parameters

<i>filename</i>	the name of the requested file
<i>mode</i>	the desired mode of transfer (netascii or octet)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the server

Returns

0 in case of success, 1 otherwise

See also

[TFTP_STR_NETASCII](#)
[TFTP_STR_OCTET](#)

Definition at line 26 of file [tftp.c](#).

5.21.2.7 tftp_send_wrq()

```
int tftp_send_wrq (
    char * filename,
    char * mode,
    int sd,
    struct sockaddr_in * addr )
```

Send a WRQ message to a server.

Do not used in current implementation.

Parameters

<i>filename</i>	the name of the requested file
<i>mode</i>	the desired mode of transfer (netascii or octet)
<i>sd</i>	socket id of the (UDP) socket to be used to send the message
<i>addr</i>	address of the server

Returns

0 in case of success, 1 otherwise

See also

[TFTP_STR_NETASCII](#)

[TFTP_STR_OCTET](#)

Definition at line 46 of file [tftp.c](#).

5.22 tftp.h

```

00001
00015 #ifndef TFTP
00016 #define TFTP
00017
00018
00019 #include <sys/socket.h>
00020 #include <netinet/in.h>
00021 #include "fblock.h"
00022
00023
00036 int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
00037
00052 int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
00053
00066 int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in *addr);
00067
00080 int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *addr);
00081
00102 int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
00103
00121 int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *addr);
00122
00139 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
00140
00141
00142 #endif

```

5.23 tftp_client.c File Reference

Implementation of the TFTP client that can only make 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 "include/netascii.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 READ_BUFFER_SIZE 80`
max stdin line length
- `#define MAX_ARGS 3`
Maximum number of arguments for commands.
- `#define MODE_TXT "txt"`
String for txt.
- `#define MODE_BIN "bin"`
String for 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 transfer_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)`
Main.

Variables

- `const int LOG_LEVEL = LOG_WARN`
Defining LOG_LEVEL for tftp_client executable.
- `char * transfer_mode`
Global transfer_mode variable for storing user chosen transfer mode string.

5.23.1 Detailed Description

Implementation of the TFTP client that can only make read requests.

Author

Riccardo Mancini

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

5.23.2 Function Documentation

5.23.2.1 cmd_mode()

```
void cmd_mode (
    char * new_mode )
```

Handles !mode command, changing transfer_mode to either bin or text.

See also

[transfer_mode](#)

Definition at line 119 of file [tftp_client.c](#).

5.23.2.2 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>	[in] the string to split
<i>delim</i>	[in] the delimiter
<i>max_argc</i>	[in] maximum number of parts to split the line into
<i>argc</i>	[out] counts of the parts the line is split into
<i>argv</i>	[out] array of parts the line is split into

Prints command usage information.

Definition at line 62 of file [tftp_client.c](#).

5.23.3 Variable Documentation

5.23.3.1 LOG_LEVEL

```
const int LOG_LEVEL = LOG_WARN
```

Defining LOG_LEVEL for tftp_client executable.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.23.3.2 transfer_mode

```
char* transfer_mode
```

Global transfer_mode variable for storing user chosen transfer mode string.

See also

[MODE_TXT](#)
[MODE_BIN](#)

Definition at line 48 of file [tftp_client.c](#).

5.24 tftp_client.c

```
00001
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 "include/netascii.h"
00016 #include <arpa/inet.h>
00017 #include <sys/types.h>
00018 #include <sys/socket.h>
00019 #include <netinet/in.h>
00020 #include <string.h>
00021 #include <stdio.h>
00022 #include <stdlib.h>
00023 #include <time.h>
00024
00026 const int LOG_LEVEL = LOG_WARN;
00027
00028
00030 #define READ_BUFFER_SIZE 80
00031
00033 #define MAX_ARGS 3
00034
00036 #define MODE_TXT "txt"
00037
00039 #define MODE_BIN "bin"
00040
00041
00048 char* transfer_mode;
00049
00050
00062 void split_string(char* line, char* delim, int max_argc, int *argc, char **argv){
00063     char *ptr;
00064     int len;
00065     char *pos;
00066
00068     // remove trailing LF
00069     if ((pos=strchr(line, '\n')) != NULL)
00070         *pos = '\0';
00071
00072     // init argc
00073     *argc = 0;
00074
00076     // tokenize string
00077     ptr = strtok(line, delim);
00078
00079     while(ptr != NULL && *argc <= max_argc){
00080         len = strlen(ptr);
00081
00082         if (len == 0)
00083             continue;
00084
00085         LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);
00086
00087         argv[*argc] = malloc(strlen(ptr)+1);
00088         strcpy(argv[*argc], ptr);
00089
00090         ptr = strtok(NULL, delim);
00091         (*argc)++;
00092     }
```

```

00093 }
00094
00098 void print_help(){
00099     printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
00100     printf("Example: ./tftp_client 127.0.0.1 69\n");
00101 }
00102
00106 void cmd_help(){
00107     printf("Sono disponibili i seguenti comandi:\n");
00108     printf("!help --> mostra l'elenco dei comandi disponibili\n");
00109     printf("!mode {txt|bin} --> imposta il modo di trasferimento dei file (testo o binario)\n");
00110     printf("!get filename nome_locale --> richiede al server il nome del file <filename> e lo salva
localmente con il nome <nome_locale>\n");
00111     printf("!quit --> termina il client\n");
00112 }
00113
00119 void cmd_mode(char* new_mode){
00120     if (strcmp(new_mode, MODE_TXT) == 0){
00121         transfer_mode = TFTP_STR_NETASCII;
00122         printf("Modo di trasferimento testo configurato\n");
00123     } else if (strcmp(new_mode, MODE_BIN) == 0){
00124         transfer_mode = TFTP_STR_OCTET;
00125         printf("Modo di trasferimento binario configurato\n");
00126     } else{
00127         printf("Modo di trasferimento sconosciuto: %s. Modi disponibili: txt, bin\n", new_mode);
00128     }
00129 }
00130
00134 int cmd_get(char* remote_filename, char* local_filename, char* sv_ip, int sv_port){
00135     struct sockaddr_in my_addr, sv_addr;
00136     int sd;
00137     int ret, tid, result;
00138     struct fbblock m_fblock;
00139     char *tmp_filename;
00140
00141     LOG(LOG_INFO, "Initializing...\n");
00142
00143     sd = socket(AF_INET, SOCK_DGRAM, 0);
00144     if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
00145         m_fblock = fbblock_open(local_filename, TFTP_DATA_BLOCK,
FBLOCK_WRITE|FBLOCK_MODE_BINARY);
00146     else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
00147         tmp_filename = malloc(strlen(local_filename)+5);
00148         strcpy(tmp_filename, local_filename);
00149         strcat(tmp_filename, ".tmp");
00150         m_fblock = fbblock_open(tmp_filename, TFTP_DATA_BLOCK,
FBLOCK_WRITE|FBLOCK_MODE_TEXT);
00151     } else
00152         return 2;
00153
00154     LOG(LOG_INFO, "Opening socket...");
00155
00156     sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
00157     my_addr = make_my_sockaddr_in(0);
00158     tid = bind_random_port(sd, &my_addr);
00159     if (tid == 0){
00160         LOG(LOG_ERR, "Error while binding to random port");
00161         perror("Could not bind to random port:");
00162         fbblock_close(&m_fblock);
00163         return 1;
00164     } else
00165         LOG(LOG_INFO, "Bound to port %d", tid);
00166
00167     printf("Richiesta file %s (%s) al server in corso.\n", remote_filename,
transfer_mode);
00168
00169     ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
00170     if (ret != 0){
00171         fbblock_close(&m_fblock);
00172         return 8+ret;
00173     }
00174
00175     printf("Trasferimento file in corso.\n");
00176
00177     ret = tftp_receive_file(&m_fblock, sd, &sv_addr);
00178
00179     if (ret == 1){ // File not found
00180         printf("File non trovato.\n");
00181         result = 0;
00182     } else if (ret != 0){
00183         LOG(LOG_ERR, "Error while receiving file!");
00184         result = 16+ret;
00185     } else{
00186         int n_blocks = (m_fblock.written + m_fblock.block_size - 1)/m_fblock.
block_size;
00187         printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);

```

```

00189     printf("Salvataggio %s completato.\n", local_filename);
00190
00191     result = 0;
00192 }
00193
00194 fblock_close(&m_fblock);
00195 if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
00196     netascii2unix(tmp_filename, local_filename);
00197     remove(tmp_filename);
00198     free(tmp_filename);
00199 }
00200
00201 return result;
00202 }
00203
00204 void cmd_quit(){
00205     printf("Client terminato con successo\n");
00206     exit(0);
00207 }
00208
00209 int main(int argc, char** argv){
00210     char* sv_ip;
00211     short int sv_port;
00212     int ret, i;
00213     char read_buffer[READ_BUFFER_SIZE];
00214     int cmd_argc;
00215     char *cmd_argv[MAX_ARGS];
00216
00217     //init random seed
00218     srand(time(NULL));
00219
00220     // default mode = bin
00221     transfer_mode = TFTP_STR_OCTET;
00222
00223     if (argc != 3){
00224         print_help();
00225         return 1;
00226     }
00227
00228     // TODO: check args
00229     sv_ip = argv[1];
00230     sv_port = atoi(argv[2]);
00231
00232     while(1){
00233         printf("> ");
00234         fflush(stdout); // flush stdout buffer
00235         fgets(read_buffer, READ_BUFFER_SIZE, stdin);
00236         split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
00237
00238         if (cmd_argc == 0){
00239             printf("Comando non riconosciuto : ''\n");
00240             cmd_help();
00241         } else{
00242             if (strcmp(cmd_argv[0], "!mode") == 0){
00243                 if (cmd_argc == 2)
00244                     cmd_mode(cmd_argv[1]);
00245                 else
00246                     printf("Il comando richiede un solo argomento: bin o txt\n");
00247             } else if (strcmp(cmd_argv[0], "!get") == 0){
00248                 if (cmd_argc == 3){
00249                     ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
00250                     LOG_DEBUG, "cmd_get returned value: %d", ret);
00251                 } else{
00252                     printf("Il comando richiede due argomenti: <filename> e <nome_locale>\n");
00253                 }
00254             } else if (strcmp(cmd_argv[0], "!quit") == 0){
00255                 if (cmd_argc == 1){
00256                     cmd_quit();
00257                 } else{
00258                     printf("Il comando non richiede argomenti\n");
00259                 }
00260             } else if (strcmp(cmd_argv[0], "!help") == 0){
00261                 if (cmd_argc == 1){
00262                     cmd_help();
00263                 } else{
00264                     printf("Il comando non richiede argomenti\n");
00265                 }
00266             } else {
00267                 printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
00268                 cmd_help();
00269             }
00270         }
00271     }
00272
00273     // Free malloc'ed strings
00274     for(i = 0; i < cmd_argc; i++)
00275         free(cmd_argv[i]);
00276 }

```

```

00280     }
00281
00282     return 0;
00283 }

```

5.25 tftp_msgs.c File Reference

Implementation of [tftp_msgs.h](#).

```

#include "include/tftp_msgs.h"
#include "include/logging.h"
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <stdint.h>

```

Functions

- int [tftp_msg_type](#) (char *buffer)
Returns msg type given a 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.

Variables

- const int [LOG_LEVEL](#)
LOG_LEVEL will be defined in another file.

5.25.1 Detailed Description

Implementation of [tftp_msgs.h](#) .

Author

Riccardo Mancini

See also

[tftp_msgs.h](#)

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

5.25.2 Function Documentation

5.25.2.1 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 163 of file [tftp_msgs.c](#).

5.25.2.2 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 130 of file [tftp_msgs.c](#).

5.25.2.3 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 |
-----
```

Error code meaning:

- 0: Not defined, see error message (if any).
- 1: File not found.
- 2: Access violation.
- 3: Disk full or allocation exceeded.
- 4: Illegal TFTP operation.
- 5: Unknown transfer ID.
- 6: File already exists.
- 7: No such user.

In current implementation only errors 1 and 4 are implemented.

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 190 of file [tftp_msgs.c](#).

5.25.2.4 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 29 of file [tftp_msgs.c](#).

5.25.2.5 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 79 of file [tftp_msgs.c](#).

5.25.2.6 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 185 of file [tftp_msgs.c](#).

5.25.2.7 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 158 of file [tftp_msgs.c](#).

5.25.2.8 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 226 of file [tftp_msgs.c](#).

5.25.2.9 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 74 of file [tftp_msgs.c](#).

5.25.2.10 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 125 of file [tftp_msgs.c](#).

5.25.2.11 tftp_msg_type()

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

Returns msg type given a 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 24 of file [tftp_msgs.c](#).

5.25.2.12 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of wrong packet size.

See also

[TFTP_TYPE_ACK](#)

Definition at line 170 of file [tftp_msgs.c](#).

5.25.2.13 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of missing fields (packet size is too small).

See also

[TFTP_TYPE_DATA](#)

Definition at line 139 of file [tftp_msgs.c](#).

5.25.2.14 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields.
- 3 in case of error string exceeding TFTP_MAX_ERROR_LEN.
- 4 in case of unrecognized error code (must be within 0 and 7).

See also

[TFTP_TYPE_ERROR](#)
[TFTP_MAX_ERROR_LEN](#)

Definition at line 199 of file [tftp_msgs.c](#).

5.25.2.15 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields inside message.
- 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.

- 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
- 5 in case of unrecognized transfer mode.

See also

[TFTP_TYPE_RRQ](#)
[TFTP_MAX_FILENAME_LEN](#)
[TFTP_MAX_MODE_LEN](#)
[TFTP_STR_NETASCII](#)
[TFTP_STR_OCTET](#)

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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields inside message.
- 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
- 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
- 5 in case of unrecognized transfer mode.

See also

[TFTP_TYPE_WRQ](#)
[TFTP_MAX_FILENAME_LEN](#)
[TFTP_MAX_MODE_LEN](#)
[TFTP_STR_NETASCII](#)
[TFTP_STR_OCTET](#)

Definition at line 39 of file [tftp_msgs.c](#).

5.25.3 Variable Documentation

5.25.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file [tftp_client.c](#).

5.26 tftp_msgs.c

```

00001
00011 #include "include/tftp_msgs.h"
00012 #include "include/logging.h"
00013 #include <string.h>
00014 #include <strings.h>
00015 #include <stdio.h>
00016 #include <arpa/inet.h>
00017 #include <stdint.h>
00018
00019
00021 extern const int LOG_LEVEL;
00022
00023
00024 int tftp_msg_type(char *buffer){
00025     return (((int)buffer[0]) << 8) + buffer[1];
00026 }
00027
00028
00029 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
00030     buffer[0] = 0;
00031     buffer[1] = 1;
00032     buffer += 2;
00033     strcpy(buffer, filename);
00034     buffer += strlen(filename)+1;
00035     strcpy(buffer, mode);
00036 }
00037
00038
00039 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode){
00040     int offset = 0;
00041     if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
00042         LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
00043         return 1;
00044     }
00045
00046     offset += 2;
00047     if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00048         LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
00049         return 3;
00050     }
00051     strcpy(filename, buffer+offset);
00052
00053     offset += strlen(filename)+1;
00054     if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00055         LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
00056         return 4;
00057     }
00058     strcpy(mode, buffer+offset);
00059
00060     offset += strlen(mode)+1;
00061     if (buffer_len != offset){
00062         LOG(LOG_ERR, "Packet contains unexpected fields");
00063         return 2;
00064     }
00065     if (strcasecmp(mode, TFTP_STR_NETASCII) == 0 || strcasecmp(
mode, TFTP_STR_OCTET) == 0)
00066         return 0;
00067     else{
00068         LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00069         return 5;
00070     }
00071 }
00072
00073
00074 int tftp_msg_get_size_rrq(char* filename, char* mode){
00075     return 4 + strlen(filename) + strlen(mode);
00076 }
00077
00078
00079 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
00080     buffer[0] = 0;
00081     buffer[1] = 2;
00082     buffer += 2;
00083     strcpy(buffer, filename);
00084     buffer += strlen(filename)+1;
00085     strcpy(buffer, mode);
00086 }
00087
00088
00089 int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename, char* mode){
00090     int offset = 0;

```

```

00091  if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
00092      LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
00093      return 1;
00094  }
00095
00096  offset += 2;
00097  if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00098      LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
00099      return 3;
00100  }
00101
00102  strcpy(filename, buffer+offset);
00103  offset += strlen(filename)+1;
00104  if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00105      LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
00106      return 4;
00107  }
00108
00109  strcpy(mode, buffer+offset);
00110  offset += strlen(mode)+1;
00111  if (buffer_len != offset){
00112      LOG(LOG_ERR, "Packet contains unexpected fields");
00113      return 2;
00114  }
00115
00116  if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode,
TFTP_STR_OCTET) == 0)
00117      return 0;
00118  else{
00119      LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00120      return 5;
00121  }
00122 }
00123
00124
00125 int tftp_msg_get_size_wrq(char* filename, char* mode){
00126     return 4 + strlen(filename) + strlen(mode);
00127 }
00128
00129
00130 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
00131     buffer[0] = 0;
00132     buffer[1] = 3;
00133     *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
00134     buffer += 4;
00135     memcpy(buffer, data, data_size);
00136 }
00137
00138
00139 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data, int*
data_size){
00140     if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
00141         LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
00142         return 1;
00143     }
00144
00145     if (buffer_len < 4){
00146         LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
00147         return 2;
00148     }
00149
00150     *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
00151     *data_size = buffer_len - 4;
00152     if (*data_size > 0)
00153         memcpy(data, buffer+4, *data_size);
00154     return 0;
00155 }
00156
00157
00158 int tftp_msg_get_size_data(int data_size){
00159     return data_size + 4;
00160 }
00161
00162
00163 void tftp_msg_build_ack(int block_n, char* buffer){
00164     buffer[0] = 0;
00165     buffer[1] = 4;
00166     *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
00167 }
00168
00169
00170 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
00171     if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
00172         LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
00173         return 1;

```



```

00174     }
00175
00176     if (buffer_len != 4){
00177         LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
00178         return 2;
00179     }
00180     *block_n = (int) ntohs(*(uint16_t*)(buffer+2));
00181     return 0;
00182 }
00183
00184
00185 int tftp_msg_get_size_ack(){
00186     return 4;
00187 }
00188
00189
00190 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
00191     buffer[0] = 0;
00192     buffer[1] = 5;
00193     *(uint16_t*)(buffer+2) = htons((uint16_t) error_code);
00194     buffer += 4;
00195     strcpy(buffer, error_msg);
00196 }
00197
00198
00199 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char*
error_msg){
00200     if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
00201         LOG(LOG_ERR, "Expected ERROR message (5), found %d", tftp_msg_type(buffer));
00202         return 1;
00203     }
00204
00205     *error_code = (int) ntohs(*(uint16_t*)(buffer+2));
00206     if (*error_code < 0 || *error_code > 7){
00207         LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
00208         return 4;
00209     }
00210
00211     buffer += 4;
00212     if (strlen(buffer) > TFTP_MAX_ERROR_LEN){
00213         LOG(LOG_ERR, "Error string too long (%d > %d): %s", (int) strlen(buffer),
TFTP_MAX_ERROR_LEN, buffer);
00214         return 3;
00215     }
00216
00217     strcpy(error_msg, buffer);
00218     if (buffer_len != strlen(error_msg)+5){
00219         LOG(LOG_WARN, "Packet contains unexpected fields");
00220         return 2;
00221     }
00222     return 0;
00223 }
00224
00225
00226 int tftp_msg_get_size_error(char* error_msg){
00227     return 5 + strlen(error_msg);
00228 }

```

5.27 tftp_msgs.h File Reference

Constructor for TFTP messages.

Macros

- `#define TFTP_TYPE_RRQ 1`
Read request message type.
- `#define TFTP_TYPE_WRQ 2`
Write request message type.
- `#define TFTP_TYPE_DATA 3`
Data message type.
- `#define TFTP_TYPE_ACK 4`
Acknowledgment message type.
- `#define TFTP_TYPE_ERROR 5`

- Error message type.*

 - `#define TFTP_STR_NETASCII` "netascii"

String for netascii.
- `#define TFTP_STR_OCTET` "octet"

String for octet.
- `#define TFTP_MAX_FILENAME_LEN` 255

Maximum filename length (do not defined in RFC)
- `#define TFTP_MAX_MODE_LEN` 8

Maximum mode field string length.
- `#define TFTP_MAX_ERROR_LEN` 255

Maximum error message length (do not defined in RFC)
- `#define TFTP_DATA_BLOCK` 512

Data block size as defined in RFC.
- `#define TFTP_MAX_DATA_MSG_SIZE` 516

Data message max size is equal to TFTP_DATA_BLOCK + 4 (header)

Functions

- `int tftp_msg_type` (char *buffer)
Returns msg type given a 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.

5.27.1 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](#).

5.27.2 Macro Definition Documentation

5.27.2.1 TFTP_MAX_MODE_LEN

```
#define TFTP_MAX_MODE_LEN 8
```

Maximum mode field string length.

Since there are only two options: 'netascii' and 'octet', len('netascii') is the TFTP_MAX_MODE_LEN.

Definition at line 50 of file [tftp_msgs.h](#).

5.27.3 Function Documentation

5.27.3.1 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 163 of file [tftp_msgs.c](#).

5.27.3.2 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 130 of file [tftp_msgs.c](#).

5.27.3.3 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   |
-----
```

Error code meaning:

- 0: Not defined, see error message (if any).
- 1: File not found.
- 2: Access violation.
- 3: Disk full or allocation exceeded.
- 4: Illegal TFTP operation.
- 5: Unknown transfer ID.
- 6: File already exists.
- 7: No such user.

In current implementation only errors 1 and 4 are implemented.

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 190 of file [tftp_msgs.c](#).

5.27.3.4 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 29 of file [tftp_msgs.c](#).

5.27.3.5 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 79 of file [tftp_msgs.c](#).

5.27.3.6 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 185 of file [tftp_msgs.c](#).

5.27.3.7 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 158 of file [tftp_msgs.c](#).

5.27.3.8 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 226 of file [tftp_msgs.c](#).

5.27.3.9 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 74 of file [tftp_msgs.c](#).

5.27.3.10 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 125 of file [tftp_msgs.c](#).

5.27.3.11 tftp_msg_type()

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

Retuns msg type given a 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 24 of file [tftp_msgs.c](#).

5.27.3.12 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of wrong packet size.

See also

[TFTP_TYPE_ACK](#)

Definition at line 170 of file [tftp_msgs.c](#).

5.27.3.13 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of missing fields (packet size is too small).

See also

[TFTP_TYPE_DATA](#)

Definition at line 139 of file [tftp_msgs.c](#).

5.27.3.14 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields.
- 3 in case of error string exceeding TFTP_MAX_ERROR_LEN.
- 4 in case of unrecognized error code (must be within 0 and 7).

See also

[TFTP_TYPE_ERROR](#)
[TFTP_MAX_ERROR_LEN](#)

Definition at line 199 of file [tftp_msgs.c](#).

5.27.3.15 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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields inside message.
- 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
- 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
- 5 in case of unrecognized transfer mode.

See also

[TFTP_TYPE_RRQ](#)
[TFTP_MAX_FILENAME_LEN](#)
[TFTP_MAX_MODE_LEN](#)
[TFTP_STR_NETASCII](#)
[TFTP_STR_OCTET](#)

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 in case of success.
- 1 in case of wrong operation code.
- 2 in case of unexpected fields inside message.
- 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
- 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
- 5 in case of unrecognized transfer mode.

See also

[TFTP_TYPE_WRQ](#)
[TFTP_MAX_FILENAME_LEN](#)
[TFTP_MAX_MODE_LEN](#)
[TFTP_STR_NETASCII](#)
[TFTP_STR_OCTET](#)

Definition at line 39 of file [tftp_msgs.c](#).

5.28 tftp_msgs.h

```

00001
00016 #ifndef TFTP_MSGS
00017 #define TFTP_MSGS
00018
00019
00021 #define TFTP_TYPE_RRQ    1

```

```

00022
00024 #define TFTP_TYPE_WRQ    2
00025
00027 #define TFTP_TYPE_DATA    3
00028
00030 #define TFTP_TYPE_ACK     4
00031
00033 #define TFTP_TYPE_ERROR   5
00034
00036 #define TFTP_STR_NETASCII "netascii"
00037
00039 #define TFTP_STR_OCTET    "octet"
00040
00042 #define TFTP_MAX_FILENAME_LEN 255
00043
00050 #define TFTP_MAX_MODE_LEN 8
00051
00053 #define TFTP_MAX_ERROR_LEN 255
00054
00056 #define TFTP_DATA_BLOCK 512
00057
00059 #define TFTP_MAX_DATA_MSG_SIZE 516
00060
00061
00074 int tftp_msg_type(char *buffer);
00075
00076
00091 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);
00092
00114 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode);
00115
00123 int tftp_msg_get_size_rrq(char* filename, char* mode);
00124
00140 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);
00141
00163 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode);
00164
00172 int tftp_msg_get_size_wrq(char* filename, char* mode);
00173
00190 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);
00191
00206 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data, int*
data_size);
00207
00216 int tftp_msg_get_size_data(int data_size);
00217
00232 void tftp_msg_build_ack(int block_n, char* buffer);
00233
00248 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);
00249
00258 int tftp_msg_get_size_ack();
00259
00287 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);
00288
00306 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char*
error_msg);
00307
00314 int tftp_msg_get_size_error(char* error_msg);
00315
00316
00317 #endif

```

5.29 tftp_server.c File Reference

Implementation of the TFTP server that can only handle read requests.

```

#include <stdlib.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 "include/netascii.h"
#include <arpa/inet.h>
#include <sys/types.h>

```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include "include/logging.h"
#include <unistd.h>
#include <time.h>
#include <linux/limits.h>
#include <libgen.h>
```

Macros

- `#define _GNU_SOURCE`
- `#define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4`
Maximum length for a RRQ message.

Functions

- `int strlcpy (const char *str1, const char *str2)`
Finds longest common prefix length of strings str1 and str2.
- `int path_inside_dir (char *path, char *dir)`
Check whether file is inside dir.
- `void print_help ()`
Prints command usage information.
- `int send_file (char *filename, char *mode, struct sockaddr_in *cl_addr)`
Sends file to a client.
- `int main (int argc, char **argv)`
Main.

Variables

- `const int LOG_LEVEL = LOG_INFO`
Defining LOG_LEVEL for tftp_server executable.

5.29.1 Detailed Description

Implementation of the TFTP server that can only handle read requests.

Author

Riccardo Mancini

The server is multiprocessed, with each process handling one request.

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

5.29.2 Function Documentation

5.29.2.1 path_inside_dir()

```
int path_inside_dir (
    char * path,
    char * dir )
```

Check whether file is inside dir.

Parameters

<i>path</i>	file absolute path (can include .. and . and multiple /)
<i>dir</i>	directory real path (can't include .. and . and multiple /)

Returns

1 if true, 0 otherwise

See also

realpath

Definition at line 59 of file [tftp_server.c](#).

5.29.3 Variable Documentation

5.29.3.1 LOG_LEVEL

```
const int LOG_LEVEL = LOG_INFO
```

Defining LOG_LEVEL for tftp_server executable.

LOG_LEVEL will be defined in another file.

Definition at line 36 of file [tftp_server.c](#).

5.30 tftp_server.c

```

00001
00011 #define _GNU_SOURCE
00012 #include <stdlib.h>
00013
00014 #include "include/tftp_msgs.h"
00015 #include "include/tftp.h"
00016 #include "include/fblock.h"
00017 #include "include/inet_utils.h"
00018 #include "include/debug_utils.h"
00019 #include "include/netascii.h"
00020 #include <arpa/inet.h>
00021 #include <sys/types.h>
00022 #include <sys/socket.h>
00023 #include <netinet/in.h>
00024 #include <string.h>
00025 #include <strings.h>
00026 #include <stdio.h>
00027 #include "include/logging.h"
00028 #include <sys/types.h>
00029 #include <unistd.h>
00030 #include <time.h>
00031 #include <linux/limits.h>
00032 #include <libgen.h>
00033
00034
00036 const int LOG_LEVEL = LOG_INFO;
00037
00038
00040 #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4
00041
00042

```

```

00044 int strlcpl(const char* str1, const char* str2){
00045     int n;
00046     for (n = 0; str1[n] != '\0' && str2[n] != '\0' && str1[n] == str2[n]; n++);
00047     return n;
00048 }
00049
00059 int path_inside_dir(char* path, char* dir){
00060     char *parent, *orig_parent, *ret_realpath;
00061     char parent_realpath[PATH_MAX];
00062     int result;
00063
00064     orig_parent = parent = malloc(strlen(path) + 1);
00065     strcpy(parent, path);
00066
00067     do{
00068         parent = dirname(parent);
00069         ret_realpath = realpath(parent, parent_realpath);
00070     } while (ret_realpath == NULL);
00071
00072     if (strlcpl(parent_realpath, dir) < strlen(dir))
00073         result = 0;
00074     else
00075         result = 1;
00076
00077     free(orig_parent);
00078     return result;
00079 }
00080
00084 void print_help(){
00085     printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
00086     printf("Example: ./tftp_server 69 .\n");
00087 }
00088
00092 int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
00093     struct sockaddr_in my_addr;
00094     int sd;
00095     int ret, tid, result;
00096     struct fblock m_fblock;
00097     char *tmp_filename;
00098
00099     sd = socket(AF_INET, SOCK_DGRAM, 0);
00100     my_addr = make_my_sockaddr_in(0);
00101     tid = bind_random_port(sd, &my_addr);
00102     if (tid == 0){
00103         LOG(LOG_ERR, "Could not bind to random port");
00104         perror("Could not bind to random port:");
00105         fblock_close(&m_fblock);
00106         return 4;
00107     } else
00108         LOG(LOG_INFO, "Bound to port %d", tid);
00109
00110     if (strcasecmp(mode, TFTP_STR_OCTET) == 0){
00111         m_fblock = fblock_open(filename, TFTP_DATA_BLOCK,
00112         FBLOCK_READ|FBLOCK_MODE_BINARY);
00113     } else if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
00114         tmp_filename = malloc(strlen(filename)+5);
00115         strcpy(tmp_filename, filename);
00116         strcat(tmp_filename, ".tmp");
00117         ret = unix2netascii(filename, tmp_filename);
00118         if (ret != 0){
00119             LOG(LOG_ERR, "Error converting text file to netascii: %d", ret);
00120             return 3;
00121         }
00122         m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK,
00123         FBLOCK_READ|FBLOCK_MODE_TEXT);
00124     } else{
00125         LOG(LOG_ERR, "Unknown mode: %s", mode);
00126         return 2;
00127     }
00128
00129     if (m_fblock.file == NULL){
00130         LOG(LOG_WARN, "Error opening file. Not found?");
00131         tftp_send_error(1, "File not found.", sd, cl_addr);
00132         result = 1;
00133     } else{
00134         LOG(LOG_INFO, "Sending file...");
00135         ret = tftp_send_file(&m_fblock, sd, cl_addr);
00136
00137         if (ret != 0){
00138             LOG(LOG_ERR, "Error sending file: %d", ret);
00139             result = 16+ret;
00140         } else{
00141             LOG(LOG_INFO, "File sent successfully");
00142             result = 0;
00143         }
00144     }
00145 }

```

```

00144     fblock_close(&m_fblock);
00145
00146     if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
00147         LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
00148         remove(tmp_filename);
00149         free(tmp_filename);
00150     }
00151
00152     return result;
00153 }
00154
00156 int main(int argc, char** argv){
00157     short int my_port;
00158     char *dir_rel_path;
00159     char *ret_realpath;
00160     char dir_realpath[PATH_MAX];
00161     int ret, type, len;
00162     char in_buffer[MAX_MSG_LEN];
00163     unsigned int addrlen;
00164     int sd;
00165     struct sockaddr_in my_addr, cl_addr;
00166     int pid;
00167     char addr_str[MAX_SOCKADDR_STR_LEN];
00168
00169     if (argc != 3){
00170         print_help();
00171         return 1;
00172     }
00173
00174     my_port = atoi(argv[1]);
00175     dir_rel_path = argv[2];
00176
00177     ret_realpath = realpath(dir_rel_path, dir_realpath);
00178     if (ret_realpath == NULL){
00179         LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
00180         return 1;
00181     }
00182
00183     addrlen = sizeof(cl_addr);
00184
00185     sd = socket(AF_INET, SOCK_DGRAM, 0);
00186     my_addr = make_my_sockaddr_in(my_port);
00187     ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
00188     if (ret == -1){
00189         perror("Could not bind: ");
00190         LOG(LOG_FATAL, "Could not bind to port %d", my_port);
00191         return 1;
00192     }
00193
00194     LOG(LOG_INFO, "Server is running");
00195
00196     while (1){
00197         len = recvfrom(sd, in_buffer, MAX_MSG_LEN, 0, (struct sockaddr*)&cl_addr, &addrlen);
00198         type = tftp_msg_type(in_buffer);
00199         sockaddr_in_to_string(cl_addr, addr_str);
00200         LOG(LOG_INFO, "Received message with type %d from %s", type, addr_str);
00201         if (type == TFTP_TYPE_RRQ){
00202             pid = fork();
00203             if (pid == -1){ // error
00204                 LOG(LOG_FATAL, "Fork error");
00205                 perror("Fork error:");
00206                 return 1;
00207             } else if (pid != 0){ // father
00208                 LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
00209                 continue; // father process continues loop
00210             } else{ // child
00211                 char filename[TFTP_MAX_FILENAME_LEN], mode[
TFTP_MAX_MODE_LEN];
00212                 char file_path[PATH_MAX], file_realpath[PATH_MAX];
00213
00214                 //init random seed
00215                 srand(time(NULL));
00216
00217                 ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);
00218
00219                 if (ret != 0){
00220                     LOG(LOG_WARN, "Error unpacking RRQ");
00221                     tftp_send_error(0, "Malformed RRQ packet.", sd, &cl_addr);
00222                     break; // child process exits loop
00223                 }
00224
00225                 strcpy(file_path, dir_realpath);
00226                 strcat(file_path, "/");
00227                 strcat(file_path, filename);
00228
00229                 // check if file is inside directory (or inside any of its subdirectories)
00230                 if (!path_inside_dir(file_path, dir_realpath)){

```



```
00231         // it is not! I caught you, Trudy!
00232         LOG(LOG_WARN, "User tried to access file %s outside set directory %s",
00233             file_realpath,
00234             dir_realpath
00235         );
00236
00237         tftp_send_error(4, "Access violation.", sd, &cl_addr);
00238         break; // child process exits loop
00239     }
00240
00241     ret_realpath = realpath(file_path, file_realpath);
00242
00243     // file not found
00244     if (ret_realpath == NULL){
00245         LOG(LOG_WARN, "File not found: %s", file_path);
00246         tftp_send_error(1, "File Not Found.", sd, &cl_addr);
00247         break; // child process exits loop
00248     }
00249
00250     LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
00251
00252     ret = send_file(file_realpath, mode, &cl_addr);
00253     if (ret != 0)
00254         LOG(LOG_WARN, "Write terminated with an error: %d", ret);
00255     break; // child process exits loop
00256 }
00257 } else{
00258     LOG(LOG_WARN, "Wrong op code: %d", type);
00259     tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
00260     // main process continues loop
00261 }
00262 }
00263
00264 LOG(LOG_INFO, "Exiting process %d", (int) getpid());
00265 return 0;
00266 }
```

Index

- bind_random_port
 - inet_utils.c, [16](#)
 - inet_utils.h, [20](#)
- cmd_mode
 - tftp_client.c, [45](#)
- debug_utils.c, [5](#), [6](#)
 - dump_buffer_hex, [5](#)
 - LOG_LEVEL, [6](#)
- debug_utils.h, [6](#), [7](#)
 - dump_buffer_hex, [7](#)
- dump_buffer_hex
 - debug_utils.c, [5](#)
 - debug_utils.h, [7](#)
- fblock, [4](#)
 - mode, [4](#)
- fblock.c, [7](#), [11](#)
 - fblock_close, [8](#)
 - fblock_open, [9](#)
 - fblock_read, [9](#)
 - fblock_write, [10](#)
 - get_length, [10](#)
 - LOG_LEVEL, [10](#)
- fblock.h, [12](#), [15](#)
 - fblock_close, [13](#)
 - fblock_open, [13](#)
 - fblock_read, [14](#)
 - fblock_write, [14](#)
- fblock_close
 - fblock.c, [8](#)
 - fblock.h, [13](#)
- fblock_open
 - fblock.c, [9](#)
 - fblock.h, [13](#)
- fblock_read
 - fblock.c, [9](#)
 - fblock.h, [14](#)
- fblock_write
 - fblock.c, [10](#)
 - fblock.h, [14](#)
- get_length
 - fblock.c, [10](#)
- inet_utils.c, [15](#), [19](#)
 - bind_random_port, [16](#)
 - LOG_LEVEL, [18](#)
 - make_my_sockaddr_in, [17](#)
 - make_sv_sockaddr_in, [17](#)
 - sockaddr_in_cmp, [18](#)
 - sockaddr_in_to_string, [18](#)
- inet_utils.h, [19](#), [22](#)
 - bind_random_port, [20](#)
 - make_my_sockaddr_in, [21](#)
 - make_sv_sockaddr_in, [21](#)
 - sockaddr_in_cmp, [22](#)
 - sockaddr_in_to_string, [22](#)
- LOG_LEVEL
 - debug_utils.c, [6](#)
 - fblock.c, [10](#)
 - inet_utils.c, [18](#)
 - netascii.c, [26](#)
 - tftp.c, [35](#)
 - tftp_client.c, [46](#)
 - tftp_msgs.c, [59](#)
 - tftp_server.c, [75](#)
- logging.h, [23](#), [24](#)
- make_my_sockaddr_in
 - inet_utils.c, [17](#)
 - inet_utils.h, [21](#)
- make_sv_sockaddr_in
 - inet_utils.c, [17](#)
 - inet_utils.h, [21](#)
- mode
 - fblock, [4](#)
- netascii.c, [25](#), [27](#)
 - LOG_LEVEL, [26](#)
 - netascii2unix, [25](#)
 - unix2netascii, [26](#)
- netascii.h, [28](#), [30](#)
 - netascii2unix, [29](#)
 - unix2netascii, [29](#)
- netascii2unix
 - netascii.c, [25](#)
 - netascii.h, [29](#)
- path_inside_dir
 - tftp_server.c, [74](#)
- sockaddr_in_cmp
 - inet_utils.c, [18](#)
 - inet_utils.h, [22](#)
- sockaddr_in_to_string
 - inet_utils.c, [18](#)
 - inet_utils.h, [22](#)
- split_string
 - tftp_client.c, [46](#)
- TFTP_MAX_MODE_LEN
 - tftp_msgs.h, [64](#)
- tftp.c, [30](#), [36](#)
 - LOG_LEVEL, [35](#)
 - tftp_receive_ack, [31](#)
 - tftp_receive_file, [32](#)
 - tftp_send_ack, [33](#)
 - tftp_send_error, [33](#)
 - tftp_send_file, [34](#)
 - tftp_send_rrq, [34](#)

- tftp_send_wrq, 35
- tftp.h, 39, 44
 - tftp_receive_ack, 40
 - tftp_receive_file, 40
 - tftp_send_ack, 41
 - tftp_send_error, 41
 - tftp_send_file, 42
 - tftp_send_rrq, 43
 - tftp_send_wrq, 43
- tftp_client.c, 44, 47
 - cmd_mode, 45
 - LOG_LEVEL, 46
 - split_string, 46
 - transfer_mode, 46
- tftp_msg_build_ack
 - tftp_msgs.c, 51
 - tftp_msgs.h, 64
- tftp_msg_build_data
 - tftp_msgs.c, 51
 - tftp_msgs.h, 65
- tftp_msg_build_error
 - tftp_msgs.c, 52
 - tftp_msgs.h, 65
- tftp_msg_build_rrq
 - tftp_msgs.c, 53
 - tftp_msgs.h, 66
- tftp_msg_build_wrq
 - tftp_msgs.c, 53
 - tftp_msgs.h, 66
- tftp_msg_get_size_ack
 - tftp_msgs.c, 54
 - tftp_msgs.h, 67
- tftp_msg_get_size_data
 - tftp_msgs.c, 54
 - tftp_msgs.h, 67
- tftp_msg_get_size_error
 - tftp_msgs.c, 54
 - tftp_msgs.h, 68
- tftp_msg_get_size_rrq
 - tftp_msgs.c, 55
 - tftp_msgs.h, 68
- tftp_msg_get_size_wrq
 - tftp_msgs.c, 55
 - tftp_msgs.h, 68
- tftp_msg_type
 - tftp_msgs.c, 56
 - tftp_msgs.h, 69
- tftp_msg_unpack_ack
 - tftp_msgs.c, 56
 - tftp_msgs.h, 69
- tftp_msg_unpack_data
 - tftp_msgs.c, 57
 - tftp_msgs.h, 70
- tftp_msg_unpack_error
 - tftp_msgs.c, 57
 - tftp_msgs.h, 70
- tftp_msg_unpack_rrq
 - tftp_msgs.c, 58
- tftp_msgs.h, 71
- tftp_msgs.c, 50, 60
 - LOG_LEVEL, 59
 - tftp_msg_build_ack, 51
 - tftp_msg_build_data, 51
 - tftp_msg_build_error, 52
 - tftp_msg_build_rrq, 53
 - tftp_msg_build_wrq, 53
 - tftp_msg_get_size_ack, 54
 - tftp_msg_get_size_data, 54
 - tftp_msg_get_size_error, 54
 - tftp_msg_get_size_rrq, 55
 - tftp_msg_get_size_wrq, 55
 - tftp_msg_type, 56
 - tftp_msg_unpack_ack, 56
 - tftp_msg_unpack_data, 57
 - tftp_msg_unpack_error, 57
 - tftp_msg_unpack_rrq, 58
- tftp_msgs.h, 62, 72
 - TFTP_MAX_MODE_LEN, 64
 - tftp_msg_build_ack, 64
 - tftp_msg_build_data, 65
 - tftp_msg_build_error, 65
 - tftp_msg_build_rrq, 66
 - tftp_msg_build_wrq, 66
 - tftp_msg_get_size_ack, 67
 - tftp_msg_get_size_data, 67
 - tftp_msg_get_size_error, 68
 - tftp_msg_get_size_rrq, 68
 - tftp_msg_get_size_wrq, 68
 - tftp_msg_type, 69
 - tftp_msg_unpack_ack, 69
 - tftp_msg_unpack_data, 70
 - tftp_msg_unpack_error, 70
 - tftp_msg_unpack_rrq, 71
- tftp_receive_ack
 - tftp.c, 31
 - tftp.h, 40
- tftp_receive_file
 - tftp.c, 32
 - tftp.h, 40
- tftp_send_ack
 - tftp.c, 33
 - tftp.h, 41
- tftp_send_error
 - tftp.c, 33
 - tftp.h, 41
- tftp_send_file
 - tftp.c, 34
 - tftp.h, 42
- tftp_send_rrq
 - tftp.c, 34
 - tftp.h, 43
- tftp_send_wrq
 - tftp.c, 35
 - tftp.h, 43
- tftp_server.c, 73, 75
 - LOG_LEVEL, 75

 path_inside_dir, [74](#)
transfer_mode
 tftp_client.c, [46](#)

unix2netascii
 netascii.c, [26](#)
 netascii.h, [29](#)