

TFTP

Generated by Doxygen 1.8.14

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	1
2.1	File List	1
3	Data Structure Documentation	2
3.1	fblock Struct Reference	2
3.1.1	Detailed Description	3
3.1.2	Field Documentation	3
4	File Documentation	3
4.1	debug_utils.c File Reference	3
4.1.1	Detailed Description	4
4.1.2	Function Documentation	4
4.2	debug_utils.c	4
4.3	debug_utils.h File Reference	5
4.3.1	Detailed Description	5
4.3.2	Function Documentation	5
4.4	debug_utils.h	6
4.5	fblock.c File Reference	6
4.5.1	Detailed Description	6
4.5.2	Macro Definition Documentation	7
4.5.3	Function Documentation	7
4.6	fblock.c	9
4.7	fblock.h File Reference	10
4.7.1	Detailed Description	11
4.7.2	Function Documentation	11
4.8	fblock.h	13
4.9	inet_utils.c File Reference	14

4.9.1 Detailed Description	15
4.9.2 Macro Definition Documentation	15
4.9.3 Function Documentation	15
4.10 inet_utils.c	17
4.11 inet_utils.h File Reference	18
4.11.1 Detailed Description	19
4.11.2 Function Documentation	19
4.12 inet_utils.h	21
4.13 logging.h File Reference	22
4.13.1 Detailed Description	22
4.14 logging.h	23
4.15 netascii.c File Reference	23
4.15.1 Detailed Description	24
4.15.2 Function Documentation	24
4.16 netascii.c	25
4.17 netascii.h File Reference	26
4.17.1 Detailed Description	27
4.17.2 Function Documentation	27
4.18 netascii.h	28
4.19 tftp.c File Reference	28
4.19.1 Detailed Description	29
4.19.2 Macro Definition Documentation	29
4.19.3 Function Documentation	29
4.20 tftp.c	33
4.21 tftp.h File Reference	37
4.21.1 Detailed Description	37
4.21.2 Function Documentation	37
4.22 tftp.h	41
4.23 tftp_client.c File Reference	42
4.23.1 Detailed Description	43

4.23.2	Macro Definition Documentation	43
4.23.3	Function Documentation	43
4.23.4	Variable Documentation	44
4.24	tftp_client.c	45
4.25	tftp_msgs.c File Reference	47
4.25.1	Detailed Description	48
4.25.2	Macro Definition Documentation	49
4.25.3	Function Documentation	49
4.26	tftp_msgs.c	57
4.27	tftp_msgs.h File Reference	60
4.27.1	Detailed Description	61
4.27.2	Macro Definition Documentation	62
4.27.3	Function Documentation	62
4.28	tftp_msgs.h	70
4.29	tftp_server.c File Reference	71
4.29.1	Detailed Description	72
4.29.2	Macro Definition Documentation	72
4.29.3	Function Documentation	73
4.30	tftp_server.c	73
Index		77

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

fblock

Structure which defines a file

2

2 File Index

2.1 File List

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

debug_utils.c	
Implementation of debug_utils.h	3
debug_utils.h	
Utility functions for debugging	5
fblock.c	
Implementation of fblock.h	6
fblock.h	
File block read and write	10
inet_utils.c	
Implementation of inet_utils.h	14
inet_utils.h	
Utility functions for managing inet addresses	18
logging.h	
Logging macro	22
netascii.c	
Implementation of netascii.h	23
netascii.h	
Conversion functions from netascii to Unix standard ASCII	26
tftp.c	
Implementation of tftp.h	28
tftp.h	
Common functions for TFTP client and server	37
tftp_client.c	
Implementation of the TFTP client that can only make read requests	42
tftp_msgs.c	
Implementation of tftp_msgs.h	47
tftp_msgs.h	
Constructor for TFTP messages	60
tftp_server.c	
Implementation of the TFTP server that can only handle read requests	71

3 Data Structure Documentation

3.1 fblock Struct Reference

Structure which defines a file.

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

Data Fields

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

3.1.1 Detailed Description

Structure which defines a file.

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

3.1.2 Field Documentation

3.1.2.1 mode

```
char fblock::mode
```

Can be read xor write, text xor binary.

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

4 File Documentation

4.1 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.

4.1.1 Detailed Description

Implementation of [debug_utils.h](#).

Author

Riccardo Mancini

See also

[debug_utils.h](#)

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

4.1.2 Function Documentation

4.1.2.1 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 18 of file [debug_utils.c](#).

4.2 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  
00018 void dump_buffer_hex(char* buffer, int len){  
00019     char *str, tmp[4];  
00020
```

```
00021  str = malloc(len*3+1);
00022
00023  str[0] = '\\0';
00024  for (int i=0; i<len; i++){
00025      sprintf(tmp, "%02x ", (unsigned char) buffer[i]);
00026      strcat(str, tmp);
00027  }
00028
00029  LOG(LOG_DEBUG, "%s", str);
00030  free(str);
00031 }
```

4.3 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.

4.3.1 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](#).

4.3.2 Function Documentation

4.3.2.1 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 18 of file [debug_utils.c](#).

4.4 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
```

4.5 fblock.c File Reference

Implementation of [fblock.h](#).

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

Macros

- `#define LOG_LEVEL LOG_INFO`
Defines log level to this file.

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.

4.5.1 Detailed Description

Implementation of [fblock.h](#).

Author

Riccardo Mancini

See also

[fblock.h](#)

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

4.5.2 Macro Definition Documentation

4.5.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 11 of file [fblock.c](#).

4.5.3 Function Documentation

4.5.3.1 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 99 of file [fblock.c](#).

4.5.3.2 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 35 of file [fblock.c](#).

4.5.3.3 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 73 of file [fblock.c](#).

4.5.3.4 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 88 of file [fblock.c](#).

4.5.3.5 get_length()

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

Returns file length.

Parameters

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

Returns

file length in bytes

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

4.6 fblock.c

```
00001
00011 #define LOG_LEVEL LOG_INFO
00012
00013
00014 #include "include/fblock.h"
00015 #include <stdio.h>
00016 #include <string.h>
00017 #include "include/logging.h"
00018
00019
00026 int get_length(FILE *f){
00027     int size;
00028     fseek(f, 0, SEEK_END); // seek to end of file
00029     size = ftell(f); // get current file pointer
00030     fseek(f, 0, SEEK_SET); // seek back to beginning of file
00031     return size;
00032 }
00033
00034
00035 struct fblock fblock_open(char* filename, int block_size, char
mode){
00036     struct fblock m_fblock;
00037     m_fblock.block_size = block_size;
00038     m_fblock.mode = mode;
00039
00040     char mode_str[4] = "";
00041
```

```

00042 LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
00043     filename,
00044     (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "
text",
00045     (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
00046     block_size
00047 );
00048
00049 if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
00050     strcat(mode_str, "w");
00051     m_fblock.written = 0;
00052 } else {
00053     strcat(mode_str, "r");
00054 }
00055
00056 if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
00057     strcat(mode_str, "b");
00058 // text otherwise
00059
00060 m_fblock.file = fopen(filename, mode_str);
00061 if (m_fblock.file == NULL){
00062     LOG(LOG_ERR, "Error while opening file %s", filename);
00063     return m_fblock;
00064 }
00065 if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
00066     m_fblock.remaining = get_length(m_fblock.file);
00067
00068 LOG(LOG_DEBUG, "Successfully opened file");
00069 return m_fblock;
00070 }
00071
00072
00073 int fblock_read(struct fblock *m_fblock, char* buffer){
00074     int bytes_read, bytes_to_read;
00075
00076     if (m_fblock->remaining > m_fblock->block_size)
00077         bytes_to_read = m_fblock->block_size;
00078     else
00079         bytes_to_read = m_fblock->remaining;
00080
00081     bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
00082     m_fblock->remaining -= bytes_read;
00083
00084     return bytes_to_read - bytes_read;
00085 }
00086
00087
00088 int fblock_write(struct fblock *m_fblock, char* buffer, int
block_size){
00089     int written_bytes;
00090
00091     if (!block_size)
00092         block_size = m_fblock->block_size;
00093
00094     written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->
file);
00095     m_fblock->written += written_bytes;
00096     return block_size - written_bytes;
00097 }
00098
00099 int fblock_close(struct fblock *m_fblock){
00100     return fclose(m_fblock->file);
00101 }

```

4.7 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.

4.7.1 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](#).

4.7.2 Function Documentation

4.7.2.1 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 99 of file [fblock.c](#).

4.7.2.2 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 35 of file [fblock.c](#).

4.7.2.3 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>	block_size bytes buffer

Returns

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

Definition at line 73 of file [fblock.c](#).

4.7.2.4 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 88 of file [fblock.c](#).

4.8 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

```

4.9 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"

```

Macros

- `#define LOG_LEVEL LOG_INFO`
Defines log level to this file.

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.

4.9.1 Detailed Description

Implementation of [inet_utils.h](#).

Author

Riccardo Mancini

See also

[inet_utils.h](#)

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

4.9.2 Macro Definition Documentation

4.9.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file [inet_utils.c](#).

4.9.3 Function Documentation

4.9.3.1 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](#).

4.9.3.2 `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](#).

4.9.3.3 `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](#).

4.9.3.4 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](#).

4.9.3.5 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](#).

4.10 inet_utils.c

```
00001
00012 #define LOG_LEVEL LOG_INFO
00013
00014
00015 #include "include/inet_utils.h"
00016 #include <stdlib.h>
00017 #include <string.h>
00018 #include <sys/socket.h>
00019 #include <netinet/in.h>
00020 #include <arpa/inet.h>
00021 #include "include/logging.h"
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 }

```

4.11 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.

4.11.1 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](#).

4.11.2 Function Documentation

4.11.2.1 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](#).

4.11.2.2 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](#).

4.11.2.3 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](#).

4.11.2.4 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](#).

4.11.2.5 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](#).

4.12 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
```

4.13 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** LOG_DEBUG
- #define **LOG**(level, ...)

4.13.1 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 per-file LOG_LEVEL for hiding some of the logging messages.

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

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

4.14 logging.h

```

00001
00022 #ifndef LOGGING
00023 #define LOGGING
00024
00025
00026 #include <stdio.h>
00027 #include <sys/types.h>
00028 #include <unistd.h>
00029
00030
00031 #define LOG_FATAL      (1)
00032 #define LOG_ERR        (2)
00033 #define LOG_WARN       (3)
00034 #define LOG_INFO       (4)
00035 #define LOG_DEBUG      (5)
00036
00037
00038 #ifndef LOG_LEVEL
00039 #define LOG_LEVEL LOG_DEBUG
00040 #endif
00041
00042
00043 #define LOG(level, ...) do { \
00044     if (level <= LOG_LEVEL) { \
00045         FILE *dbgstream; \
00046         char where[25]; \
00047         switch(level){ \
00048             case LOG_FATAL: \
00049                 dbgstream = stderr; \
00050                 fprintf(dbgstream, "[FATAL]"); \
00051                 break; \
00052             case LOG_ERR: \
00053                 dbgstream = stderr; \
00054                 fprintf(dbgstream, "[ERROR]"); \
00055                 break; \
00056             case LOG_WARN: \
00057                 dbgstream = stderr; \
00058                 fprintf(dbgstream, "[WARN ]"); \
00059                 break; \
00060             case LOG_INFO: \
00061                 dbgstream = stdout; \
00062                 fprintf(dbgstream, "[INFO ]"); \
00063                 break; \
00064             case LOG_DEBUG: \
00065                 dbgstream = stdout; \
00066                 fprintf(dbgstream, "[DEBUG]"); \
00067                 break; \
00068             } \
00069             fprintf(dbgstream, "[%5d]", (int) getpid()); \
00070             snprintf(where, 25, "%s:%d", __FILE__, __LINE__); \
00071             fprintf(dbgstream, " %-25s ", where); \
00072             fprintf(dbgstream, __VA_ARGS__); \
00073             fprintf(dbgstream, "\n"); \
00074             fflush(dbgstream); \
00075         } \
00076     } while (0)
00077
00078
00079 #endif

```

4.15 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.

4.15.1 Detailed Description

Implementation of [netascii.h](#).

Author

Riccardo Mancini

See also

[netascii.h](#)

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

4.15.2 Function Documentation

4.15.2.1 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

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

4.15.2.2 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

Definition at line 16 of file [netascii.c](#).

4.16 netascii.c

```

00001
00011 #include "include/netascii.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014
00015
00016 int unix2netascii(char *unix_filename, char* netascii_filename){
00017     FILE *unixf, *netasciif;
00018     char prev, tmp;
00019     int ret, result;
00020
00021     unixf = fopen(unix_filename, "r");
00022
00023     if (unixf == NULL){
00024         LOG(LOG_ERR, "Error opening file %s", unix_filename);
00025         return 1;
00026     }
00027
00028     netasciif = fopen(netascii_filename, "w");
00029
00030     if (unixf == NULL){
00031         LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00032         return 2;
00033     }
00034
00035     prev = EOF;
00036
00037     while ((tmp = (char) fgetc(unixf)) != EOF){
00038         if (tmp == '\n' && (prev == EOF || prev != '\r')){ // LF -> CRLF
00039             ret = putc('\r', netasciif);
00040             if (ret == EOF)
00041                 break;
00042
00043             ret = putc('\n', netasciif);
00044             if (ret == EOF)
00045                 break;
00046
00047         } else if (tmp == '\r'){ // CR -> CRNUL
00048             char next = (char) fgetc(unixf);
00049             if (next != '\0')
00050                 ungetc(next, unixf);
00051
00052             ret = putc('\r', netasciif);
00053             if (ret == EOF)
00054                 break;
00055
00056             ret = putc('\0', netasciif);
00057             if (ret == EOF)
00058                 break;
00059
00060         } else if (tmp == '\0'){
00061             ret = putc('\0', netasciif);
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 %x after CR", next);
00124                 result = 5;
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     if (result == 0){
00137         // Error writing to unixf
00138         if (ret == EOF){
00139             LOG(LOG_ERR, "Error writing to file %s", unix_filename);
00140             result = 3;
00141         } else{
00142             LOG(LOG_INFO, "Netascii file %s converted to Unix file %s", netascii_filename, unix_filename);
00143             result = 0;
00144         }
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 }

```

4.17 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.

4.17.1 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`.

4.17.2 Function Documentation

4.17.2.1 netascii2unix()

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

Netascii to Unix conversion.

Parameters

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

Returns

- 0 in case of success

Definition at line 87 of file `netascii.c`.

4.17.2.2 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

Definition at line 16 of file [netascii.c](#).

4.18 netascii.h

```
00001
00017 #ifndef NETASCII
00018 #define NETASCII
00019
00020
00029 int unix2netascii(char *unix_filename, char* netascii_filename);
00030
00039 int netascii2unix(char* netascii_filename, char *unix_filename);
00040
00041
00042 #endif
```

4.19 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"
```

Macros

- `#define LOG_LEVEL LOG_INFO`
Defines log level to this file.

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.

4.19.1 Detailed Description

Implementation of [tftp.h](#).

Author

Riccardo Mancini

See also

[tftp.h](#)

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

4.19.2 Macro Definition Documentation

4.19.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file [tftp.c](#).

4.19.3 Function Documentation

4.19.3.1 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 195 of file `tftp.c`.

4.19.3.2 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.
- 9 in case of message from unexpected source

Definition at line 102 of file [tftp.c](#).

4.19.3.3 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 85 of file [tftp.c](#).

4.19.3.4 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 65 of file [tftp.c](#).

4.19.3.5 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 227 of file [tftp.c](#).

4.19.3.6 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 25 of file [tftp.c](#).

4.19.3.7 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 45 of file [tftp.c](#).

4.20 tftp.c

```
00001
00012 #define LOG_LEVEL LOG_INFO
00013
00014 #include "include/fblock.h"
00015 #include "include/tftp_msgs.h"
00016 #include "include/debug_utils.h"
00017 #include "include/inet_utils.h"
00018 #include <arpa/inet.h>
00019 #include <sys/socket.h>
```

```

00020 #include <netinet/in.h>
00021 #include <stdlib.h>
00022 #include "include/logging.h"
00023
00024
00025 int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00026     int msglen, len;
00027     char *out_buffer;
00028
00029     msglen = tftp_msg_get_size_rrq(filename, mode);
00030     out_buffer = malloc(msglen);
00031
00032     tftp_msg_build_rrq(filename, mode, out_buffer);
00033     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00034     if (len != msglen){
00035         LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
00036         perror("Error");
00037         return 1;
00038     }
00039
00040     free(out_buffer);
00041     return 0;
00042 }
00043
00044
00045 int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00046     int msglen, len;
00047     char *out_buffer;
00048
00049     msglen = tftp_msg_get_size_wrq(filename, mode);
00050     out_buffer = malloc(msglen);
00051
00052     tftp_msg_build_wrq(filename, mode, out_buffer);
00053     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00054     if (len != msglen){
00055         LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
00056         perror("Error");
00057         return 1;
00058     }
00059
00060     free(out_buffer);
00061     return 0;
00062 }
00063
00064
00065 int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in *addr){
00066     int msglen, len;
00067     char *out_buffer;
00068
00069     msglen = tftp_msg_get_size_error(error_msg);
00070     out_buffer = malloc(msglen);
00071
00072     tftp_msg_build_error(error_code, error_msg, out_buffer);
00073     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00074     if (len != msglen){
00075         LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
00076         perror("Error");
00077         return 1;
00078     }
00079
00080     free(out_buffer);
00081     return 0;
00082 }
00083
00084
00085 int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *addr){
00086     int msglen, len;
00087
00088     msglen = tftp_msg_get_size_ack();
00089     tftp_msg_build_ack(block_n, out_buffer);
00090     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
00091
00092     if (len != msglen){
00093         LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);
00094         perror("Error");
00095         return 1;
00096     }
00097
00098     return 0;
00099 }
00100
00101
00102 int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
00103     char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[
00104         TFTP_DATA_BLOCK], out_buffer[4];
00105     int exp_block_n, rcv_block_n;
00106     int len, data_size, ret, type;

```

```

00106     unsigned int addrlen;
00107     struct sockaddr_in cl_addr, orig_cl_addr;
00108
00109     // init expected block number
00110     exp_block_n = 1;
00111
00112     addrlen = sizeof(cl_addr);
00113
00114     do{
00115         LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
00116         // TODO: check client == server ?
00117         len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(
TFTP_DATA_BLOCK), 0, (struct sockaddr*)&cl_addr, &addrlen);
00118         if (exp_block_n == 1){ // first block -> I need to save servers TID (aka its "original" sockaddr)
00119             char addr_str[MAX_SOCKADDR_STR_LEN];
00120             sockaddr_in_to_string(cl_addr, addr_str);
00121
00122             if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
00123                 LOG(LOG_ERR, "Received message from unexpected source: %s", addr_str);
00124                 return 9;
00125             } else{
00126                 LOG(LOG_INFO, "Receiving packets from %s", addr_str);
00127                 orig_cl_addr = cl_addr;
00128             }
00129         } else{
00130             if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
00131                 char addr_str[MAX_SOCKADDR_STR_LEN];
00132                 sockaddr_in_to_string(cl_addr, addr_str);
00133                 LOG(LOG_ERR, "Received message from unexpected source: %s", addr_str);
00134                 return 9;
00135             } else{
00136                 LOG(LOG_DEBUG, "Sender is the same!");
00137             }
00138         }
00139
00140         type = tftp_msg_type(in_buffer);
00141         if (type == TFTP_TYPE_ERROR){
00142             int error_code;
00143             char error_msg[TFTP_MAX_ERROR_LEN];
00144
00145             ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
00146             if (ret != 0){
00147                 LOG(LOG_ERR, "Error unpacking error msg");
00148                 return 5;
00149             }
00150
00151             if (error_code == 1){
00152                 LOG(LOG_INFO, "File not found");
00153                 return 1;
00154             } else{
00155                 LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
00156                 return 7;
00157             }
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         LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);
00179
00180         if (data_size != 0){
00181             if (fblock_write(m_fblock, data, data_size))
00182                 return 6;
00183         }
00184
00185         LOG(LOG_DEBUG, "Sending ack");
00186
00187         if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
00188             return 2;
00189
00190     } while(data_size == TFTP_DATA_BLOCK);

```

```

00191     return 0;
00192 }
00193
00194
00195 int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *addr){
00196     int msglen, len, ret;
00197     unsigned int addrlen;
00198     struct sockaddr_in cl_addr;
00199
00200     msglen = tftp_msg_get_size_ack();
00201     addrlen = sizeof(cl_addr);
00202
00203     len = recvfrom(sd, in_buffer, msglen, 0, (struct sockaddr*)&cl_addr, &addrlen);
00204
00205     if (sockaddr_in_cmp(*addr, cl_addr) != 0){
00206         char str_addr[MAX_SOCKADDR_STR_LEN];
00207         sockaddr_in_to_string(cl_addr, str_addr);
00208         LOG(LOG_ERR, "Message is coming from unexpected source: %s", str_addr);
00209         return 2;
00210     }
00211
00212     if (len != msglen){
00213         LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
00214         return 1;
00215     }
00216
00217     ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
00218     if (ret != 0){
00219         LOG(LOG_ERR, "Error unpacking ack: %d", ret);
00220         return 8+ret;
00221     }
00222
00223     return 0;
00224 }
00225
00226
00227 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
00228     char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[
00229         TFTP_MAX_DATA_MSG_SIZE];
00230     int block_n, rcv_block_n;
00231     int len, data_size, msglen;
00232
00233     // init sequence number
00234     block_n = 1;
00235
00236     do{
00237         LOG(LOG_DEBUG, "Sending part %d", block_n);
00238
00239         if (m_fblock->remaining > TFTP_DATA_BLOCK)
00240             data_size = TFTP_DATA_BLOCK;
00241         else
00242             data_size = m_fblock->remaining;
00243
00244         if (data_size != 0)
00245             fblock_read(m_fblock, data);
00246
00247         LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);
00248
00249         msglen = tftp_msg_get_size_data(data_size);
00250         tftp_msg_build_data(block_n, data, data_size, out_buffer);
00251
00252         // dump_buffer_hex(out_buffer, msglen);
00253
00254         len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*)addr, sizeof(*addr));
00255
00256         if (len != msglen){
00257             return 1;
00258         }
00259
00260         LOG(LOG_DEBUG, "Waiting for ack");
00261
00262         if (tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr)){
00263             LOG(LOG_ERR, "Error receiving ack");
00264             return 2;
00265         }
00266
00267         if (rcv_block_n != block_n){
00268             LOG(LOG_ERR, "Received wrong block n: received %d != expected %d", rcv_block_n, block_n);
00269             return 3;
00270         }
00271
00272         block_n++;
00273     } while(data_size == TFTP_DATA_BLOCK);
00274     return 0;
00275 }

```

4.21 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.

4.21.1 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](#).

4.21.2 Function Documentation

4.21.2.1 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 195 of file `tftp.c`.

4.21.2.2 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.
- 9 in case of message from unexpected source

Definition at line 102 of file [tftp.c](#).

4.21.2.3 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 85 of file [tftp.c](#).

4.21.2.4 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 65 of file [tftp.c](#).

4.21.2.5 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 227 of file [tftp.c](#).

4.21.2.6 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 25 of file [tftp.c](#).

4.21.2.7 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 45 of file [tftp.c](#).

4.22 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
00103 int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
00104
00122 int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *addr);
00123
00140 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
00141
00142
00143 #endif

```

4.23 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 LOG_LEVEL LOG_INFO`
Defines log level to this file.
- `#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

- char * [transfer_mode](#)
Global transfer_mode variable for storing user chosen transfer mode string.

4.23.1 Detailed Description

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

Author

Riccardo Mancini

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

4.23.2 Macro Definition Documentation

4.23.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 9 of file [tftp_client.c](#).

4.23.3 Function Documentation

4.23.3.1 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](#).

4.23.3.2 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](#).

4.23.4 Variable Documentation

4.23.4.1 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](#).

4.24 tftp_client.c

```

00001
00009 #define LOG_LEVEL LOG_INFO
00010
00011
00012 #include "include/logging.h"
00013 #include "include/tftp_msgs.h"
00014 #include "include/tftp.h"
00015 #include "include/fblock.h"
00016 #include "include/inet_utils.h"
00017 #include "include/debug_utils.h"
00018 #include "include/netascii.h"
00019 #include <arpa/inet.h>
00020 #include <sys/types.h>
00021 #include <sys/socket.h>
00022 #include <netinet/in.h>
00023 #include <string.h>
00024 #include <stdio.h>
00025 #include <stdlib.h>
00026 #include <time.h>
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;
00067     char *pos;
00068
00069     // remove trailing LF
00070     if ((pos=strchr(line, '\n')) != NULL)
00071         *pos = '\0';
00072
00073     // init argc
00074     *argc = 0;
00075
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 traferimento 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
00180     if (ret == 1){ // File not found
00181         printf("File non trovato.\n");
00182         result = 0;
00183     } else if (ret != 0){
00184         LOG(LOG_ERR, "Error while receiving file!");
00185         result = 16+ret;
00186     } else{
00187         int n_blocks = (m_fblock.written + m_fblock.block_size - 1)/m_fblock.
block_size;
00188         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     fbblock_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
00208 void cmd_quit(){
00209     printf("Client terminato con successo\n");
00210     exit(0);
00211 }
00212
00214 int main(int argc, char** argv){

```

```

00215 char* sv_ip;
00216 short int sv_port;
00217 int ret;
00218 char read_buffer[READ_BUFFER_SIZE];
00219 int cmd_argc;
00220 char *cmd_argv[MAX_ARGS];
00221
00222 //init random seed
00223 srand(time(NULL));
00224
00225 // default mode = bin
00226 transfer_mode = TFTP_STR_OCTET;
00227
00228 if (argc != 3){
00229     print_help();
00230     return 1;
00231 }
00232
00233 // TODO: check args
00234 sv_ip = argv[1];
00235 sv_port = atoi(argv[2]);
00236
00237 while(1){
00238     printf("> ");
00239     fflush(stdout); // flush stdout buffer
00240     fgets(read_buffer, READ_BUFFER_SIZE, stdin);
00241     split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
00242
00243     if (cmd_argc == 0){
00244         printf("Comando non riconosciuto : ''\n");
00245         cmd_help();
00246     } else{
00247         if (strcmp(cmd_argv[0], "!mode") == 0){
00248             if (cmd_argc == 2)
00249                 cmd_mode(cmd_argv[1]);
00250             else
00251                 printf("Il comando richiede un solo argomento: bin o txt\n");
00252         } else if (strcmp(cmd_argv[0], "!get") == 0){
00253             if (cmd_argc == 3){
00254                 ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
00255                 LOG(LOG_DEBUG, "cmd_get returned value: %d", ret);
00256             } else{
00257                 printf("Il comando richiede due argomenti: <filename> e <nome_locale>\n");
00258             }
00259         } else if (strcmp(cmd_argv[0], "!quit") == 0){
00260             if (cmd_argc == 1){
00261                 cmd_quit();
00262             } else{
00263                 printf("Il comando non richiede argomenti\n");
00264             }
00265         } else if (strcmp(cmd_argv[0], "!help") == 0){
00266             if (cmd_argc == 1){
00267                 cmd_help();
00268             } else{
00269                 printf("Il comando non richiede argomenti\n");
00270             }
00271         } else {
00272             printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
00273             cmd_help();
00274         }
00275     }
00276
00277     // Free malloc'ed strings
00278     for (int i = 0; i < cmd_argc; i++)
00279         free(cmd_argv[i]);
00280 }
00281
00282 return 0;
00283 }

```

4.25 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>
```

Macros

- `#define LOG_LEVEL LOG_INFO`
Defines log level to this file.

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.

4.25.1 Detailed Description

Implementation of [tftp_msgs.h](#).

Author

Riccardo Mancini

See also

[tftp_msgs.h](#)

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

4.25.2 Macro Definition Documentation

4.25.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file [tftp_msgs.c](#).

4.25.3 Function Documentation

4.25.3.1 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](#).

4.25.3.2 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](#).

4.25.3.3 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](#).

4.25.3.4 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](#).

4.25.3.5 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](#).

4.25.3.6 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](#).

4.25.3.7 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](#).

4.25.3.8 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](#).

4.25.3.9 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](#).

4.25.3.10 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](#).

4.25.3.11 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](#).

4.25.3.12 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](#).

4.25.3.13 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](#).

4.25.3.14 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](#).

4.25.3.15 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](#).

4.26 tftp_msgs.c

```

00001
00012 #define LOG_LEVEL LOG_INFO
00013
00014
00015 #include "include/tftp_msgs.h"
00016 #include "include/logging.h"
00017 #include <string.h>
00018 #include <strings.h>
00019 #include <stdio.h>
00020 #include <arpa/inet.h>
00021 #include <stdint.h>
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 }

```

4.27 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.

4.27.1 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](#).

4.27.2 Macro Definition Documentation**4.27.2.1 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](#).

4.27.3 Function Documentation**4.27.3.1 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](#).

4.27.3.2 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](#).

4.27.3.3 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](#).

4.27.3.4 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](#).

4.27.3.5 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](#).

4.27.3.6 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](#).

4.27.3.7 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](#).

4.27.3.8 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](#).

4.27.3.9 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](#).

4.27.3.10 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](#).

4.27.3.11 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](#).

4.27.3.12 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](#).

4.27.3.13 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](#).

4.27.3.14 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](#).

4.27.3.15 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](#).

4.28 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

```

4.29 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 LOG_LEVEL LOG_INFO`
Defines log level to this file.
- `#define _GNU_SOURCE`
- `#define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4`
Maximum length for a RRQ message.

Functions

- `int strlcp1 (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.

4.29.1 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](#).

4.29.2 Macro Definition Documentation

4.29.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file [tftp_server.c](#).

4.29.3 Function Documentation

4.29.3.1 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 58 of file [tftp_server.c](#).

4.30 tftp_server.c

```
00001  
00012 #define LOG_LEVEL LOG_INFO  
00013  
00014 #define _GNU_SOURCE  
00015 #include <stdlib.h>  
00016  
00017 #include "include/tftp_msgs.h"  
00018 #include "include/tftp.h"  
00019 #include "include/fblock.h"  
00020 #include "include/inet_utils.h"  
00021 #include "include/debug_utils.h"  
00022 #include "include/netascii.h"  
00023 #include <arpa/inet.h>  
00024 #include <sys/types.h>  
00025 #include <sys/socket.h>  
00026 #include <netinet/in.h>  
00027 #include <string.h>  
00028 #include <strings.h>  
00029 #include <stdio.h>  
00030 #include "include/logging.h"
```

```

00031 #include <sys/types.h>
00032 #include <unistd.h>
00033 #include <time.h>
00034 #include <linux/limits.h>
00035 #include <libgen.h>
00036
00037
00039 #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4
00040
00041
00043 int strlcpy(const char* str1, const char* str2){
00044     int n;
00045     for (n = 0; str1[n] != '\0' && str2[n] != '\0' && str1[n] == str2[n]; n++);
00046     return n;
00047 }
00048
00058 int path_inside_dir(char* path, char* dir){
00059     char *parent, *orig_parent, *ret_realpath;
00060     char parent_realpath[PATH_MAX];
00061     int result;
00062
00063     orig_parent = parent = malloc(strlen(path) + 1);
00064     strcpy(parent, path);
00065
00066     do{
00067         parent = dirname(parent);
00068         ret_realpath = realpath(parent, parent_realpath);
00069     } while (ret_realpath == NULL);
00070
00071     if (strlcpy(parent_realpath, dir) < strlen(dir))
00072         result = 0;
00073     else
00074         result = 1;
00075
00076     free(orig_parent);
00077     return result;
00078 }
00079
00083 void print_help(){
00084     printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
00085     printf("Example: ./tftp_server 69 .\n");
00086 }
00087
00091 int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
00092     struct sockaddr_in my_addr;
00093     int sd;
00094     int ret, tid, result;
00095     struct flock m_fblock;
00096     char *tmp_filename;
00097
00098     sd = socket(AF_INET, SOCK_DGRAM, 0);
00099     my_addr = make_my_sockaddr_in(0);
00100     tid = bind_random_port(sd, &my_addr);
00101     if (tid == 0){
00102         LOG(LOG_ERR, "Could not bind to random port");
00103         perror("Could not bind to random port:");
00104         fblock_close(&m_fblock);
00105         return 4;
00106     } else
00107         LOG(LOG_INFO, "Bound to port %d", tid);
00108
00109     if (strcasecmp(mode, TFTP_STR_OCTET) == 0){
00110         m_fblock = fblock_open(filename, TFTP_DATA_BLOCK,
FBLOCK_READ|FBLOCK_MODE_BINARY);
00111     } else if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
00112         tmp_filename = malloc(strlen(filename)+5);
00113         strcpy(tmp_filename, filename);
00114         strcat(tmp_filename, ".tmp");
00115         ret = unix2netascii(filename, tmp_filename);
00116         if (ret != 0){
00117             LOG(LOG_ERR, "Error converting text file to netascii: %d", ret);
00118             return 3;
00119         }
00120         m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK,
FBLOCK_READ|FBLOCK_MODE_TEXT);
00121     } else{
00122         LOG(LOG_ERR, "Unknown mode: %s", mode);
00123         return 2;
00124     }
00125
00126     if (m_fblock.file == NULL){
00127         LOG(LOG_WARN, "Error opening file. Not found?");
00128         tftp_send_error(1, "File not found.", sd, cl_addr);
00129         result = 1;
00130     } else{
00131         LOG(LOG_INFO, "Sending file...");
00132         ret = tftp_send_file(&m_fblock, sd, cl_addr);

```

```

00133
00134     if (ret != 0){
00135         LOG(LOG_ERR, "Error sending file: %d", ret);
00136         result = 16+ret;
00137     } else{
00138         LOG(LOG_INFO, "File sent successfully");
00139         result = 0;
00140     }
00141 }
00142
00143 fblock_close(&m_fblock);
00144
00145 if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
00146     LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
00147     remove(tmp_filename);
00148     free(tmp_filename);
00149 }
00150
00151 return result;
00152 }
00153
00155 int main(int argc, char** argv){
00156     short int my_port;
00157     char *dir_rel_path;
00158     char *ret_realpath;
00159     char dir_realpath[PATH_MAX];
00160     int ret, type, len;
00161     char in_buffer[MAX_MSG_LEN];
00162     unsigned int addrlen;
00163     int sd;
00164     struct sockaddr_in my_addr, cl_addr;
00165     int pid;
00166     char addr_str[MAX_SOCKADDR_STR_LEN];
00167
00168     if (argc != 3){
00169         print_help();
00170         return 1;
00171     }
00172
00173     my_port = atoi(argv[1]);
00174     dir_rel_path = argv[2];
00175
00176     ret_realpath = realpath(dir_rel_path, dir_realpath);
00177     if (ret_realpath == NULL){
00178         LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
00179         return 1;
00180     }
00181
00182     addrlen = sizeof(cl_addr);
00183
00184     sd = socket(AF_INET, SOCK_DGRAM, 0);
00185     my_addr = make_my_sockaddr_in(my_port);
00186     ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
00187     if (ret == -1){
00188         perror("Could not bind: ");
00189         LOG(LOG_FATAL, "Could not bind to port %d", my_port);
00190         return 1;
00191     }
00192
00193     LOG(LOG_INFO, "Server is running");
00194
00195     while (1){
00196         len = recvfrom(sd, in_buffer, MAX_MSG_LEN, 0, (struct sockaddr*)&cl_addr, &addrlen);
00197         type = tftp_msg_type(in_buffer);
00198         sockaddr_in_to_string(cl_addr, addr_str);
00199         LOG(LOG_INFO, "Received message with type %d from %s", type, addr_str);
00200         if (type == TFTP_TYPE_RRQ){
00201             pid = fork();
00202             if (pid != 0){ // father
00203                 LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
00204                 continue; // father process continues loop
00205             } else{ // child
00206                 char filename[TFTP_MAX_FILENAME_LEN], mode[
TFTP_MAX_MODE_LEN];
00207                 char file_path[PATH_MAX], file_realpath[PATH_MAX];
00208
00209                 //init random seed
00210                 srand(time(NULL));
00211
00212                 ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);
00213
00214                 if (ret != 0){
00215                     LOG(LOG_WARN, "Error unpacking RRQ");
00216                     tftp_send_error(0, "Malformed RRQ packet.", sd, &cl_addr);
00217                     break; // child process exits loop
00218                 }
00219

```

```
00220     strcpy(file_path, dir_realpath);
00221     strcat(file_path, "/");
00222     strcat(file_path, filename);
00223
00224     // check if file is inside directory (or inside any of its subdirectories)
00225     if (!path_inside_dir(file_path, dir_realpath)){
00226         // it is not! I caught you, Trudy!
00227         LOG(LOG_WARN, "User tried to access file %s outside set directory %s",
00228             file_realpath,
00229             dir_realpath
00230         );
00231
00232         tftp_send_error(4, "Access violation.", sd, &cl_addr);
00233         break; // child process exits loop
00234     }
00235
00236     ret_realpath = realpath(file_path, file_realpath);
00237
00238     // file not found
00239     if (ret_realpath == NULL){
00240         LOG(LOG_WARN, "File not found: %s", file_path);
00241         tftp_send_error(1, "File Not Found.", sd, &cl_addr);
00242         break; // child process exits loop
00243     }
00244
00245     LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
00246
00247     ret = send_file(file_realpath, mode, &cl_addr);
00248     if (ret != 0)
00249         LOG(LOG_WARN, "Write terminated with an error: %d", ret);
00250     break; // child process exits loop
00251 }
00252 } else{
00253     LOG(LOG_WARN, "Wrong op code: %d", type);
00254     tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
00255     // main process continues loop
00256 }
00257 }
00258
00259 LOG(LOG_INFO, "Exiting process %d", (int) getpid());
00260 return 0;
00261 }
```

Index

- bind_random_port
 - inet_utils.c, 15
 - inet_utils.h, 19
- cmd_mode
 - tftp_client.c, 43
- debug_utils.c, 3, 4
 - dump_buffer_hex, 4
- debug_utils.h, 5, 6
 - dump_buffer_hex, 5
- dump_buffer_hex
 - debug_utils.c, 4
 - debug_utils.h, 5
- fblock, 2
 - mode, 3
- fblock.c, 6, 9
 - fblock_close, 7
 - fblock_open, 7
 - fblock_read, 8
 - fblock_write, 8
 - get_length, 9
 - LOG_LEVEL, 7
- fblock.h, 10, 13
 - fblock_close, 11
 - fblock_open, 12
 - fblock_read, 12
 - fblock_write, 13
- fblock_close
 - fblock.c, 7
 - fblock.h, 11
- fblock_open
 - fblock.c, 7
 - fblock.h, 12
- fblock_read
 - fblock.c, 8
 - fblock.h, 12
- fblock_write
 - fblock.c, 8
 - fblock.h, 13
- get_length
 - fblock.c, 9
- inet_utils.c, 14, 17
 - bind_random_port, 15
 - LOG_LEVEL, 15
 - make_my_sockaddr_in, 16
 - make_sv_sockaddr_in, 16
 - sockaddr_in_cmp, 16
 - sockaddr_in_to_string, 17
- inet_utils.h, 18, 21
 - bind_random_port, 19
 - make_my_sockaddr_in, 20
 - make_sv_sockaddr_in, 20
 - sockaddr_in_cmp, 21
 - sockaddr_in_to_string, 21
- LOG_LEVEL
 - fblock.c, 7
 - inet_utils.c, 15
 - tftp.c, 29
 - tftp_client.c, 43
 - tftp_msgs.c, 49
 - tftp_server.c, 72
 - logging.h, 22, 23
- make_my_sockaddr_in
 - inet_utils.c, 16
 - inet_utils.h, 20
- make_sv_sockaddr_in
 - inet_utils.c, 16
 - inet_utils.h, 20
- mode
 - fblock, 3
- netascii.c, 23, 25
 - netascii2unix, 24
 - unix2netascii, 24
- netascii.h, 26, 28
 - netascii2unix, 27
 - unix2netascii, 27
- netascii2unix
 - netascii.c, 24
 - netascii.h, 27
- path_inside_dir
 - tftp_server.c, 73
- sockaddr_in_cmp
 - inet_utils.c, 16
 - inet_utils.h, 21
- sockaddr_in_to_string
 - inet_utils.c, 17
 - inet_utils.h, 21
- split_string
 - tftp_client.c, 44
- TFTP_MAX_MODE_LEN
 - tftp_msgs.h, 62
- tftp.c, 28, 33
 - LOG_LEVEL, 29
 - tftp_receive_ack, 29
 - tftp_receive_file, 30
 - tftp_send_ack, 31
 - tftp_send_error, 31
 - tftp_send_file, 32
 - tftp_send_rrq, 32
 - tftp_send_wrq, 33
- tftp.h, 37, 41
 - tftp_receive_ack, 37
 - tftp_receive_file, 38

- tftp_send_ack, 39
- tftp_send_error, 39
- tftp_send_file, 40
- tftp_send_rrq, 40
- tftp_send_wrq, 41
- tftp_client.c, 42, 45
 - cmd_mode, 43
 - LOG_LEVEL, 43
 - split_string, 44
 - transfer_mode, 44
- tftp_msg_build_ack
 - tftp_msgs.c, 49
 - tftp_msgs.h, 62
- tftp_msg_build_data
 - tftp_msgs.c, 49
 - tftp_msgs.h, 63
- tftp_msg_build_error
 - tftp_msgs.c, 50
 - tftp_msgs.h, 63
- tftp_msg_build_rrq
 - tftp_msgs.c, 51
 - tftp_msgs.h, 64
- tftp_msg_build_wrq
 - tftp_msgs.c, 51
 - tftp_msgs.h, 64
- tftp_msg_get_size_ack
 - tftp_msgs.c, 51
 - tftp_msgs.h, 65
- tftp_msg_get_size_data
 - tftp_msgs.c, 52
 - tftp_msgs.h, 65
- tftp_msg_get_size_error
 - tftp_msgs.c, 52
 - tftp_msgs.h, 66
- tftp_msg_get_size_rrq
 - tftp_msgs.c, 53
 - tftp_msgs.h, 66
- tftp_msg_get_size_wrq
 - tftp_msgs.c, 53
 - tftp_msgs.h, 66
- tftp_msg_type
 - tftp_msgs.c, 53
 - tftp_msgs.h, 67
- tftp_msg_unpack_ack
 - tftp_msgs.c, 54
 - tftp_msgs.h, 67
- tftp_msg_unpack_data
 - tftp_msgs.c, 55
 - tftp_msgs.h, 68
- tftp_msg_unpack_error
 - tftp_msgs.c, 55
 - tftp_msgs.h, 68
- tftp_msg_unpack_rrq
 - tftp_msgs.c, 56
 - tftp_msgs.h, 69
- tftp_msgs.c, 47, 57
 - LOG_LEVEL, 49
 - tftp_msg_build_ack, 49
 - tftp_msg_build_data, 49
 - tftp_msg_build_error, 50
 - tftp_msg_build_rrq, 51
 - tftp_msg_build_wrq, 51
 - tftp_msg_get_size_ack, 51
 - tftp_msg_get_size_data, 52
 - tftp_msg_get_size_error, 52
 - tftp_msg_get_size_rrq, 53
 - tftp_msg_get_size_wrq, 53
 - tftp_msg_type, 53
 - tftp_msg_unpack_ack, 54
 - tftp_msg_unpack_data, 55
 - tftp_msg_unpack_error, 55
 - tftp_msg_unpack_rrq, 56
- tftp_msgs.h, 60, 70
 - TFTP_MAX_MODE_LEN, 62
 - tftp_msg_build_ack, 62
 - tftp_msg_build_data, 63
 - tftp_msg_build_error, 63
 - tftp_msg_build_rrq, 64
 - tftp_msg_build_wrq, 64
 - tftp_msg_get_size_ack, 65
 - tftp_msg_get_size_data, 65
 - tftp_msg_get_size_error, 66
 - tftp_msg_get_size_rrq, 66
 - tftp_msg_get_size_wrq, 66
 - tftp_msg_type, 67
 - tftp_msg_unpack_ack, 67
 - tftp_msg_unpack_data, 68
 - tftp_msg_unpack_error, 68
 - tftp_msg_unpack_rrq, 69
- tftp_receive_ack
 - tftp.c, 29
 - tftp.h, 37
- tftp_receive_file
 - tftp.c, 30
 - tftp.h, 38
- tftp_send_ack
 - tftp.c, 31
 - tftp.h, 39
- tftp_send_error
 - tftp.c, 31
 - tftp.h, 39
- tftp_send_file
 - tftp.c, 32
 - tftp.h, 40
- tftp_send_rrq
 - tftp.c, 32
 - tftp.h, 40
- tftp_send_wrq
 - tftp.c, 33
 - tftp.h, 41
- tftp_server.c, 71, 73
 - LOG_LEVEL, 72
 - path_inside_dir, 73
- transfer_mode
 - tftp_client.c, 44
- unix2netascii

netascii.c, [24](#)
netascii.h, [27](#)