# TFTP

Generated by Doxygen 1.8.14

# Contents

# 1 Simple TFTP implementation

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

The `project assignment` requires to:

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

The server can be started with the following syntax:

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

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

Example:

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

The client can be started with the following syntax:

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

The client should also support the following operations:

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

Example of client operation:

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

# 2 Data Structure Index

## 2.1 Data Structures

Here are the data structures with brief descriptions:

# 3 File Index

## 3.1 File List

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

# 4 Data Structure Documentation

## 4.1 fblock Struct Reference

Structure which defines a file.

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

### Data Fields

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

### 4.1.1 Detailed Description

Structure which defines a file.

Definition at line 40 of file fblock.h.

### 4.1.2 Field Documentation

#### 4.1.2.1 mode

```
char fblock::mode
```

Can be read xor write, text xor binary.

Definition at line 43 of file fblock.h.

# 5 File Documentation

## 5.1 debug_utils.c File Reference

Implementation of debug_utils.h.

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

### Functions

- void dump_buffer_hex (char *buffer, int len)

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

### Variables

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

### 5.1.1 Detailed Description

Implementation of debug_utils.h.

**Author**

Riccardo Mancini

**See also**

debug_utils.h

Definition in file debug_utils.c.

### 5.1.2 Function Documentation

#### 5.1.2.1 dump_buffer_hex()

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

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

**Parameters**

| | |
|---|---|
| *buffer* | pointer to the buffer to be printed |
| *len* | the length (in bytes) of the buffer |

Definition at line 22 of file debug_utils.c.

#### 5.1.3 Variable Documentation

##### 5.1.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.2 debug_utils.c

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

## 5.3 debug_utils.h File Reference

Utility functions for debugging.

**Functions**

- void dump_buffer_hex (char *buffer, int len)

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

#### 5.3.1 Detailed Description

Utility functions for debugging.

**Author**

Riccardo Mancini

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

Definition in file debug_utils.h.

#### 5.3.2 Function Documentation

##### 5.3.2.1 dump_buffer_hex()

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

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

**Parameters**

| | |
|---|---|
| *buffer* | pointer to the buffer to be printed |
| *len* | the length (in bytes) of the buffer |

Definition at line 22 of file debug_utils.c.

### 5.4 debug_utils.h

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

### 5.5 fblock.c File Reference

Implementation of fblock.h.

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

#### Functions

- int get_length (FILE *f)

  *Returns file length.*
- struct fblock fblock_open (char *filename, int block_size, char mode)

  *Opens a file.*
- int fblock_read (struct fblock *m_fblock, char *buffer)

  *Reads next block_size bytes from file.*
- int fblock_write (struct fblock *m_fblock, char *buffer, int block_size)

  *Writes next block_size bytes to file.*
- int fblock_close (struct fblock *m_fblock)

  *Closes a file.*

#### Variables

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

### 5.5.1 Detailed Description

Implementation of fblock.h.

#### Author

Riccardo Mancini

#### See also

fblock.h

Definition in file fblock.c.

### 5.5.2 Function Documentation

#### 5.5.2.1 fblock_close()

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

Closes a file.

#### Parameters

| | |
|---|---|
| *m_fblock* | fblock instance to be closed |

**Returns**

0 in case of success, EOF in case of failure

**See also**

fclose

Definition at line 100 of file fblock.c.

#### 5.5.2.2 fblock_open()

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

Opens a file.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *block_size* | size of the blocks |
| *mode* | mode (read, write, text, binary) |

**Returns**

fblock structure

**See also**

FBLOCK_MODE_TEXT
FBLOCK_MODE_BINARY
FBLOCK_WRITE
FBLOCK_READ

Definition at line 36 of file fblock.c.

#### 5.5.2.3 fblock_read()

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

Reads next block_size bytes from file.

**Parameters**

| | |
|---|---|
| *m_fblock* | fblock instance |
| *buffer* | block_size bytes buffer |

**Returns**

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

Definition at line 74 of file fblock.c.

#### 5.5.2.4 fblock_write()

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

Writes next block_size bytes to file.

**Parameters**

| | |
|---|---|
| *m_fblock* | fblock instance |
| *buffer* | block_size bytes buffer |
| *block_size* | if set to a non-0 value, override block_size defined in fblock. |

**Returns**

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

Definition at line 89 of file fblock.c.

#### 5.5.2.5 get_length()

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

Returns file length.

**Parameters**

| | |
|---|---|
| *f* | file pointer |

**Returns**

file length in bytes

Definition at line 27 of file fblock.c.

### 5.5.3 Variable Documentation

#### 5.5.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.6 fblock.c

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

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

## 5.7 fblock.h File Reference

File block read and write.

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

### Data Structures

- struct fblock

  *Structure which defines a file.*

### Macros

- #define FBLOCK_MODE_MASK 0b01

  *Mask for getting text/binary mode.*
- #define FBLOCK_MODE_TEXT 0b00

  *Open file in text mode.*
- #define FBLOCK_MODE_BINARY 0b01

  *Open file in binary mode.*
- #define FBLOCK_RW_MASK 0b10

  *Mask for getting r/w mode.*
- #define FBLOCK_READ 0b00

  *Open file in read mode.*
- #define FBLOCK_WRITE 0b10

  *Open file in write mode.*

### Functions

- struct fblock fblock_open (char ∗filename, int block_size, char mode)

  *Opens a file.*
- int fblock_read (struct fblock ∗m_fblock, char ∗buffer)

  *Reads next block_size bytes from file.*
- int fblock_write (struct fblock ∗m_fblock, char ∗buffer, int block_size)

  *Writes next block_size bytes to file.*
- int fblock_close (struct fblock ∗m_fblock)

  *Closes a file.*

### 5.7.1 Detailed Description

File block read and write.

**Author**

Riccardo Mancini

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

Definition in file fblock.h.

### 5.7.2 Function Documentation

#### 5.7.2.1 fblock_close()

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

Closes a file.

**Parameters**

| | |
|---|---|
| *m_fblock* | fblock instance to be closed |

**Returns**

0 in case of success, EOF in case of failure

**See also**

fclose

Definition at line 100 of file fblock.c.

#### 5.7.2.2 fblock_open()

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

Opens a file.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *block_size* | size of the blocks |
| *mode* | mode (read, write, text, binary) |

**Returns**

fblock structure

**See also**

FBLOCK_MODE_TEXT
FBLOCK_MODE_BINARY
FBLOCK_WRITE
FBLOCK_READ

Definition at line 36 of file fblock.c.

#### 5.7.2.3 fblock_read()

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

Reads next block_size bytes from file.

**Parameters**

| | |
|---|---|
| *m_fblock* | fblock instance |
| *buffer* | block_size bytes buffer |

**Returns**

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

Definition at line 74 of file fblock.c.

#### 5.7.2.4 fblock_write()

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

Writes next block_size bytes to file.

**Parameters**

| | |
|---|---|
| *m_fblock* | fblock instance |
| *buffer* | block_size bytes buffer |
| *block_size* | if set to a non-0 value, override block_size defined in fblock. |

**Returns**

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

Definition at line 89 of file fblock.c.

## 5.8 fblock.h

```
00001
00011 #ifndef FBLOCK
00012 #define FBLOCK
00013
00014
00015 #include <stdio.h>
00016
00017
00019 #define FBLOCK_MODE_MASK   0b01
00020
00022 #define FBLOCK_MODE_TEXT   0b00
00023
00025 #define FBLOCK_MODE_BINARY 0b01
00026
00028 #define FBLOCK_RW_MASK     0b10
00029
00031 #define FBLOCK_READ        0b00
00032
00034 #define FBLOCK_WRITE       0b10
00035
00036
00040 struct fblock{
00041   FILE *file;
00042   int block_size;
00043   char mode;
00044   union{
00045     unsigned int written;
00046     unsigned int remaining;
00047   };
00048 };
00049
00050
00064 struct fblock fblock_open(char* filename, int block_size, char
      mode);
00065
00074 int fblock_read(struct fblock *m_fblock, char* buffer);
00075
00086 int fblock_write(struct fblock *m_fblock, char* buffer, int
      block_size);
00087
00096 int fblock_close(struct fblock *m_fblock);
00097
00098
00099 #endif
```

## 5.9 inet_utils.c File Reference

Implementation of inet_utils.h.

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

#### Functions

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

  *Binds socket to a random port.*
- struct sockaddr_in make_sv_sockaddr_in (char *ip, int port)

  *Makes sockaddr_in structure given ip string and port of server.*
- struct sockaddr_in make_my_sockaddr_in (int port)

  *Makes sockaddr_in structure of this host.*
- int sockaddr_in_cmp (struct sockaddr_in sai1, struct sockaddr_in sai2)

  *Compares INET addresses, returning 0 in case they're equal.*
- void sockaddr_in_to_string (struct sockaddr_in src, char *dst)

  *Converts sockaddr_in structure to string to be printed.*

#### Variables

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

### 5.9.1 Detailed Description

Implementation of inet_utils.h.

#### Author

Riccardo Mancini

#### See also

inet_utils.h

Definition in file inet_utils.c.

### 5.9.2 Function Documentation

#### 5.9.2.1 bind_random_port()

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

Binds socket to a random port.

#### Parameters

| | |
|---|---|
| *socket* | socket ID |
| *addr* | inet addr structure |

**Returns**

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

**See also**

FROM_PORT
TO_PORT
MAX_TRIES

Definition at line 24 of file inet_utils.c.

#### 5.9.2.2 make_my_sockaddr_in()

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

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

**Parameters**

| | |
|---|---|
| *port* | port of the server |

**Returns**

sockaddr_in structure this host on given port

Definition at line 55 of file inet_utils.c.

#### 5.9.2.3 make_sv_sockaddr_in()

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

Makes sockaddr_in structure given ip string and port of server.

**Parameters**

| | |
|---|---|
| *ip* | ip address of server |
| *port* | port of the server |

**Returns**

sockaddr_in structure for the given server

Definition at line 45 of file inet_utils.c.

#### 5.9.2.4 sockaddr_in_cmp()

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

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

**Parameters**

| | |
|---|---|
| *sai1* | first address |
| *sai2* | second address |

**Returns**

0 if thery're equal, 1 otherwise

Definition at line 65 of file inet_utils.c.

#### 5.9.2.5 sockaddr_in_to_string()

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

Converts sockaddr_in structure to string to be printed.

**Parameters**

| | |
|---|---|
| *src* | the input address |
| *dst* | the output string (must be at least MAX_SOCKADDR_STR_LEN long) |

Definition at line 73 of file inet_utils.c.

### 5.9.3 Variable Documentation

#### 5.9.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.10 inet_utils.c

```
00001
00011 #include "include/inet_utils.h"
00012 #include <stdlib.h>
00013 #include <string.h>
00014 #include <sys/socket.h>
00015 #include <netinet/in.h>
00016 #include <arpa/inet.h>
00017 #include "include/logging.h"
00018
00019
00021 extern const int LOG_LEVEL;
00022
00023
00024 int bind_random_port(int socket, struct sockaddr_in *addr){
00025   int port, ret, i;
00026   for (i=0; i<MAX_TRIES; i++){
00027     if (i == 0) // first I generate a random one
00028       port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
00029     else  //if it's not free I scan the next one
00030       port = (port-FROM_PORT+1) % (TO_PORT - FROM_PORT + 1) +
      FROM_PORT;
00031
00032     LOG(LOG_DEBUG, "Trying port %d...", port);
00033
00034     addr->sin_port = htons(port);
00035     ret = bind(socket, (struct sockaddr*) addr, sizeof(*addr));
00036     if (ret != -1)
00037       return port;
00038     // consider only some errors?
00039   }
00040   LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
00041   return 0;
00042 }
00043
00044
00045 struct sockaddr_in make_sv_sockaddr_in(char* ip, int port){
00046   struct sockaddr_in addr;
00047   memset(&addr, 0, sizeof(addr));
00048   addr.sin_family = AF_INET;
00049   addr.sin_port = htons(port);
00050   inet_pton(AF_INET, ip, &addr.sin_addr);
00051   return addr;
00052 }
00053
00054
00055 struct sockaddr_in make_my_sockaddr_in(int port){
00056   struct sockaddr_in addr;
00057   memset(&addr, 0, sizeof(addr));
00058   addr.sin_family = AF_INET;
00059   addr.sin_port = htons(port);
00060   addr.sin_addr.s_addr = htonl(INADDR_ANY);
00061   return addr;
00062 }
00063
00064
00065 int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
00066   if (sai1.sin_port == sai2.sin_port &&
00067       sai1.sin_addr.s_addr == sai2.sin_addr.s_addr)
00068     return 0;
00069   else
00070     return 1;
00071 }
00072
00073 void sockaddr_in_to_string(struct sockaddr_in src, char *dst){
00074   char* port_str;
00075   const char *ret;
00076
00077   port_str = malloc(6);
00078   sprintf(port_str, "%d", ntohs(src.sin_port));
00079
00080   ret = inet_ntop(AF_INET, (void*) &src.sin_addr, dst, MAX_SOCKADDR_STR_LEN);
00081   if (ret != NULL){
00082     strcat(dst, ":");
00083     strcat(dst, port_str);
00084   } else{
00085     strcpy(dst, "ERROR");
00086   }
00087
00088   free(port_str);
00089 }
```

## 5.11 inet_utils.h File Reference

Utility funcions for managing inet addresses.

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

### Macros

- #define FROM_PORT 49152

  *Random port will be greater or equal to FROM_PORT.*
- #define TO_PORT 65535

  *Random port will be lower or equal to TO_PORT.*
- #define MAX_TRIES 256

  *Maximum number of trials before giving up opening a random port.*
- #define MAX_SOCKADDR_STR_LEN 22

  *Maximum number of characters of INET address to string (eg 123.156.189.123:45678).*

### Functions

- int bind_random_port (int socket, struct sockaddr_in ∗addr)

  *Binds socket to a random port.*
- struct sockaddr_in make_sv_sockaddr_in (char ∗ip, int port)

  *Makes sockaddr_in structure given ip string and port of server.*
- struct sockaddr_in make_my_sockaddr_in (int port)

  *Makes sockaddr_in structure of this host.*
- int sockaddr_in_cmp (struct sockaddr_in sai1, struct sockaddr_in sai2)

  *Compares INET addresses, returning 0 in case they're equal.*
- void sockaddr_in_to_string (struct sockaddr_in src, char ∗dst)

  *Converts sockaddr_in structure to string to be printed.*

### 5.11.1 Detailed Description

Utility funcions for managing inet addresses.

**Author**

Riccardo Mancini

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

**See also**

sockaddr_in
rand

Definition in file inet_utils.h.

### 5.11.2 Function Documentation

#### 5.11.2.1 bind_random_port()

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

Binds socket to a random port.

**Parameters**

| | |
|---|---|
| *socket* | socket ID |
| *addr* | inet addr structure |

**Returns**

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

**See also**

FROM_PORT
TO_PORT
MAX_TRIES

Definition at line 24 of file inet_utils.c.

#### 5.11.2.2 make_my_sockaddr_in()

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

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

**Parameters**

| | |
|---|---|
| *port* | port of the server |

**Returns**

sockaddr_in structure this host on given port

Definition at line 55 of file inet_utils.c.

#### 5.11.2.3 make_sv_sockaddr_in()

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

Makes sockaddr_in structure given ip string and port of server.

**Parameters**

| | |
|---|---|
| *ip* | ip address of server |
| *port* | port of the server |

**Returns**

sockaddr_in structure for the given server

Definition at line 45 of file inet_utils.c.

#### 5.11.2.4 sockaddr_in_cmp()

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

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

**Parameters**

| | |
|---|---|
| *sai1* | first address |
| *sai2* | second address |

**Returns**

0 if thery're equal, 1 otherwise

Definition at line 65 of file inet_utils.c.

#### 5.11.2.5 sockaddr_in_to_string()

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

Converts sockaddr_in structure to string to be printed.

**Parameters**

| | |
|---|---|
| *src* | the input address |
| *dst* | the output string (must be at least MAX_SOCKADDR_STR_LEN long) |

Definition at line 73 of file inet_utils.c.

## 5.12 inet_utils.h

```
00001
00015 #ifndef INET_UTILS
00016 #define INET_UTILS
00017
00018
00019 #include <sys/socket.h>
00020 #include <netinet/in.h>
00021
00023 #define FROM_PORT 49152
00024
00026 #define TO_PORT   65535
00027
00029 #define MAX_TRIES 256
00030
00035 #define MAX_SOCKADDR_STR_LEN 22
00036
00037
00049 int bind_random_port(int socket, struct sockaddr_in *addr);
00050
00058 struct sockaddr_in make_sv_sockaddr_in(char* ip, int port);
00059
00068 struct sockaddr_in make_my_sockaddr_in(int port);
00069
00077 int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);
00078
00085 void sockaddr_in_to_string(struct sockaddr_in src, char *dst);
00086
00087
00088 #endif
```

## 5.13 logging.h File Reference

Logging macro.

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

**Macros**

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

### 5.13.1 Detailed Description

Logging macro.

**Author**

Riccardo Mancini

This file contains a macro for logging in different levels.

There are 5 levels of logging:

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

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

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

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

in the file containing the main and

```
extern const int LOG_LEVEL;
```

in any other file using this macro.

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

Definition in file logging.h.

## 5.14 logging.h

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

## 5.15 netascii.c File Reference

Implementation of netascii.h.

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

### Functions

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

  *Unix to netascii conversion.*
- int netascii2unix (char *netascii_filename, char *unix_filename)

  *Netascii to Unix conversion.*

**Variables**

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

### 5.15.1 Detailed Description

Implementation of netascii.h.

**Author**

Riccardo Mancini

**See also**

netascii.h

Definition in file netascii.c.

### 5.15.2 Function Documentation

#### 5.15.2.1 netascii2unix()

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

Netascii to Unix conversion.

**Parameters**

| | |
|---|---|
| *netascii_filename* | the filename of the input netascii file |
| *unix_filename* | the filename of the output Unix file |

**Returns**

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

Definition at line 90 of file netascii.c.

#### 5.15.2.2 unix2netascii()

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

Unix to netascii conversion.

**Parameters**

| | |
|---|---|
| *unix_filename* | the filename of the input Unix file |
| *netascii_filename* | the filename of the output netascii file |

**Returns**

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

Definition at line 20 of file netascii.c.

### 5.15.3 Variable Documentation

#### 5.15.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.16 netascii.c

```
00001
00011 #include "include/netascii.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014
00015
00017 extern const int LOG_LEVEL;
00018
00019
00020 int unix2netascii(char *unix_filename, char* netascii_filename){
00021   FILE *unixf, *netasciif;
00022   char prev, tmp;
00023   int ret, result;
00024
00025   unixf = fopen(unix_filename, "r");
00026
00027   if (unixf == NULL){
00028     LOG(LOG_ERR, "Error opening file %s", unix_filename);
00029     return 1;
```

```
00030   }
00031
00032   netasciif = fopen(netascii_filename, "w");
00033
00034   if (unixf == NULL){
00035     LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00036     return 2;
00037   }
00038
00039   prev = EOF;
00040
00041   while ((tmp = (char) fgetc(unixf)) != EOF){
00042     if (tmp == '\n' && prev != '\r'){ // LF -> CRLF
00043       ret = putc('\r', netasciif);
00044       if (ret == EOF)
00045         break;
00046
00047       ret = putc('\n', netasciif);
00048       if (ret == EOF)
00049         break;
00050
00051     } else if (tmp == '\r'){  // CR -> CRNUL
00052       char next = (char) fgetc(unixf);
00053       if (next != '\0')
00054         ungetc(next, unixf);
00055
00056       ret = putc('\r', netasciif);
00057       if (ret == EOF)
00058         break;
00059
00060       ret = putc('\0', netasciif);
00061       if (ret == EOF)
00062         break;
00063     } else{
00064       ret = putc(tmp, netasciif);
00065       if (ret == EOF)
00066         break;
00067     }
00068
00069     prev = tmp;
00070   }
00071
00072   // Error writing to netasciif
00073   if (ret == EOF){
00074     LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
00075     result = 3;
00076   } else{
00077     LOG(LOG_INFO, "Unix file %s converted to netascii file %s",
00078         unix_filename,
00079         netascii_filename
00080     );
00081     result = 0;
00082   }
00083
00084   fclose(unixf);
00085   fclose(netasciif);
00086
00087   return result;
00088 }
00089
00090 int netascii2unix(char* netascii_filename, char *unix_filename){
00091   FILE *unixf, *netasciif;
00092   char tmp;
00093   int ret;
00094   int result = 0;
00095
00096   unixf = fopen(unix_filename, "w");
00097
00098   if (unixf == NULL){
00099     LOG(LOG_ERR, "Error opening file %s", unix_filename);
00100     return 1;
00101   }
00102
00103   netasciif = fopen(netascii_filename, "r");
00104
00105   if (unixf == NULL){
00106     LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00107     return 2;
00108   }
00109
00110   while ((tmp = (char) fgetc(netasciif)) != EOF){
00111     if (tmp == '\r'){  // CRLF -> LF ; CRNUL -> CR
00112       char next = (char) fgetc(netasciif);
00113       if (next == '\0'){  // CRNUL -> CR
00114        ret = putc('\r', unixf);
00115        if (ret == EOF)
00116         break;
```

```
00117       } else if (next == '\n'){  // CRLF -> LF
00118        ret = putc('\n', unixf);
00119        if (ret == EOF)
00120         break;
00121       } else if (next == EOF) { // bad format
00122         LOG(LOG_ERR, "Bad formatted netascii: unexpected EOF after CR");
00123         result = 4;
00124         break;
00125       } else{                   // bad format
00126         LOG(LOG_ERR, "Bad formatted netascii: unexpected 0x%x after CR", next);
00127         result = 4;
00128         break;
00129       }
00130     } else{
00131
00132       // nothing else needs to be done!
00133
00134       ret = putc(tmp, unixf);
00135       if (ret == EOF)
00136         break;
00137     }
00138   }
00139
00140   if (result == 0){
00141     // Error writing to unixf
00142     if (ret == EOF){
00143       LOG(LOG_ERR, "Error writing to file %s", unix_filename);
00144       result = 3;
00145     } else{
00146       LOG(LOG_INFO, "Netascii file %s converted to Unix file %s",
00147           netascii_filename,
00148           unix_filename
00149       );
00150       result = 0;
00151     }
00152   } // otherwise there was an error (4 or 5) and result was already set
00153
00154   fclose(unixf);
00155   fclose(netasciif);
00156
00157   return result;
00158 }
```

## 5.17 netascii.h File Reference

Conversion functions from netascii to Unix standard ASCII.

### Functions

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

  *Unix to netascii conversion.*
- int netascii2unix (char *netascii_filename, char *unix_filename)

  *Netascii to Unix conversion.*

### 5.17.1 Detailed Description

Conversion functions from netascii to Unix standard ASCII.

**Author**

Riccardo Mancini

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

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

**See also**

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

Definition in file netascii.h.

### 5.17.2 Function Documentation

#### 5.17.2.1 netascii2unix()

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

Netascii to Unix conversion.

**Parameters**

| | |
|---|---|
| *netascii_filename* | the filename of the input netascii file |
| *unix_filename* | the filename of the output Unix file |

**Returns**

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

Definition at line 90 of file netascii.c.

#### 5.17.2.2 unix2netascii()

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

Unix to netascii conversion.

**Parameters**

| | |
|---|---|
| *unix_filename* | the filename of the input Unix file |
| *netascii_filename* | the filename of the output netascii file |

**Returns**

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

Definition at line 20 of file netascii.c.

## 5.18 netascii.h

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

## 5.19 tftp.c File Reference

Implementation of tftp.h.

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

### Functions

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

  *Send a RRQ message to a server.*
- int tftp_send_wrq (char *filename, char *mode, int sd, struct sockaddr_in *addr)

  *Send a WRQ message to a server.*
- int tftp_send_error (int error_code, char *error_msg, int sd, struct sockaddr_in *addr)

  *Send an ERROR message to the client (server).*
- int tftp_send_ack (int block_n, char *out_buffer, int sd, struct sockaddr_in *addr)

  *Send an ACK message.*
- int tftp_receive_file (struct fblock *m_fblock, int sd, struct sockaddr_in *addr)

  *Handle the entire workflow required to receive a file.*
- int tftp_receive_ack (int *block_n, char *in_buffer, int sd, struct sockaddr_in *addr)

  *Receive an ACK message.*
- int tftp_send_file (struct fblock *m_fblock, int sd, struct sockaddr_in *addr)

  *Handle the entire workflow required to send a file.*

### Variables

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

#### 5.19.1 Detailed Description

Implementation of tftp.h.

**Author**

Riccardo Mancini

**See also**

tftp.h

Definition in file tftp.c.

#### 5.19.2 Function Documentation

##### 5.19.2.1 tftp_receive_ack()

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

Receive an ACK message.

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

**Parameters**

| | |
|---|---|
| *block_n* | [out] sequence number of the acknowledged block. |
| *in_buffer* | buffer to be used for receiving the ACK (useful for recycling the same buffer) |
| *sd* | [in] socket id of the (UDP) socket to be used to send the message |
| *addr* | [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 221 of file tftp.c.

#### 5.19.2.2 tftp_receive_file()

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

Handle the entire workflow required to receive a file.

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

**Parameters**

| | |
|---|---|
| *m_fblock* | block file where to write incoming data to |
| *sd* | socket id of the (UDP) socket to be used to send ACK messages |
| *addr* | 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 en error while writing to the file.
- 7 in case of an error message different from File Not Found (since it is the only erorr available in current implementation).
- 8 in case of the incoming message is neither DATA nor ERROR.

Definition at line 118 of file tftp.c.

#### 5.19.2.3 tftp_send_ack()

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

Send an ACK message.

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

**Parameters**

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

**Returns**

0 in case of success, 1 otherwise

Definition at line 97 of file tftp.c.

#### 5.19.2.4 tftp_send_error()

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

Send an ERROR message to the client (server).

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

**Parameters**

| | |
|---|---|
| *error_code* | the code of the error (must be within 0 and 7) |
| *error_msg* | the message explaining the error |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the client (server) |

**Returns**

0 in case of success, 1 otherwise

Definition at line 73 of file tftp.c.

#### 5.19.2.5 tftp_send_file()

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

Handle the entire workflow required to send a file.

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

**Parameters**

| | |
|---|---|
| *m_fblock* | block file where to read incoming data from |
| *sd* | socket id of the (UDP) socket to be used to send DATA messages |
| *addr* | 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 file too big

Definition at line 257 of file tftp.c.

#### 5.19.2.6 tftp_send_rrq()

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

Send a RRQ message to a server.

**Parameters**

| | |
|---|---|
| *filename* | the name of the requested file |
| *mode* | the desired mode of transfer (netascii or octet) |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the server |

**Returns**

0 in case of success, 1 otherwise

**See also**

TFTP_STR_NETASCII
TFTP_STR_OCTET

Definition at line 27 of file tftp.c.

#### 5.19.2.7 tftp_send_wrq()

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

Send a WRQ message to a server.

Do not used in current implementation.

**Parameters**

| | |
|---|---|
| *filename* | the name of the requested file |
| *mode* | the desired mode of transfer (netascii or octet) |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the server |

**Returns**

0 in case of success, 1 otherwise

**See also**

TFTP_STR_NETASCII
TFTP_STR_OCTET

Definition at line 50 of file tftp.c.

### 5.19.3 Variable Documentation

#### 5.19.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.20 tftp.c

```
00001
00011 #include "include/fblock.h"
00012 #include "include/tftp_msgs.h"
00013 #include "include/tftp.h"
00014 #include "include/debug_utils.h"
00015 #include "include/inet_utils.h"
00016 #include "include/logging.h"
00017 #include <arpa/inet.h>
00018 #include <sys/socket.h>
00019 #include <netinet/in.h>
00020 #include <stdlib.h>
00021
00022
00024 extern const int LOG_LEVEL;
00025
00026
00027 int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00028   int msglen, len;
00029   char *out_buffer;
00030
00031   msglen = tftp_msg_get_size_rrq(filename, mode);
00032   out_buffer = malloc(msglen);
00033
00034   tftp_msg_build_rrq(filename, mode, out_buffer);
00035   len = sendto(sd, out_buffer, msglen, 0,
00036                (struct sockaddr*) addr,
```

```
00037                 sizeof(*addr)
00038   );
00039   if (len != msglen){
00040     LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
00041     perror("Error");
00042     return 1;
00043   }
00044
00045   free(out_buffer);
00046   return 0;
00047 }
00048
00049
00050 int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
00051   int msglen, len;
00052   char *out_buffer;
00053
00054   msglen = tftp_msg_get_size_wrq(filename, mode);
00055   out_buffer = malloc(msglen);
00056
00057   tftp_msg_build_wrq(filename, mode, out_buffer);
00058   len = sendto(sd, out_buffer, msglen, 0,
00059                 (struct sockaddr*) addr,
00060                 sizeof(*addr)
00061   );
00062   if (len != msglen){
00063     LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
00064     perror("Error");
00065     return 1;
00066   }
00067
00068   free(out_buffer);
00069   return 0;
00070 }
00071
00072
00073 int tftp_send_error(int error_code, char* error_msg, int sd,
00074                     struct sockaddr_in *addr){
00075   int msglen, len;
00076   char *out_buffer;
00077
00078   msglen = tftp_msg_get_size_error(error_msg);
00079   out_buffer = malloc(msglen);
00080
00081   tftp_msg_build_error(error_code, error_msg, out_buffer);
00082   len = sendto(sd, out_buffer, msglen, 0,
00083                 (struct sockaddr*) addr,
00084                 sizeof(*addr)
00085   );
00086   if (len != msglen){
00087     LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
00088     perror("Error");
00089     return 1;
00090   }
00091
00092   free(out_buffer);
00093   return 0;
00094 }
00095
00096
00097 int tftp_send_ack(int block_n, char* out_buffer, int sd,
00098                   struct sockaddr_in *addr){
00099   int msglen, len;
00100
00101   msglen = tftp_msg_get_size_ack();
00102   tftp_msg_build_ack(block_n, out_buffer);
00103   len = sendto(sd, out_buffer, msglen, 0,
00104                 (struct sockaddr*) addr,
00105                 sizeof(*addr)
00106   );
00107
00108  if (len != msglen){
00109     LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);
00110     perror("Error");
00111     return 1;
00112   }
00113
00114   return 0;
00115 }
00116
00117
00118 int tftp_receive_file(struct fblock *m_fblock, int sd,
00119                       struct sockaddr_in *addr){
00120   char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[
      TFTP_DATA_BLOCK], out_buffer[4];
00121   int exp_block_n, rcv_block_n;
00122   int len, data_size, ret, type;
```

```
00123   unsigned int addrlen;
00124   struct sockaddr_in cl_addr, orig_cl_addr;
00125
00126   // init expected block number
00127   exp_block_n = 1;
00128
00129   addrlen = sizeof(cl_addr);
00130
00131   do{
00132     LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
00133
00134     len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(
      TFTP_DATA_BLOCK), 0,
00135                    (struct sockaddr*)&cl_addr,
00136                    &addrlen
00137     );
00138
00139     // first block -> I need to save servers TID (aka its "original" sockaddr)
00140     if (exp_block_n == 1){
00141       char addr_str[MAX_SOCKADDR_STR_LEN];
00142       sockaddr_in_to_string(cl_addr, addr_str);
00143
00144       if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
00145         LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
00146         continue;
00147       } else{
00148         LOG(LOG_INFO, "Receiving packets from %s", addr_str);
00149         orig_cl_addr = cl_addr;
00150       }
00151     } else{
00152       if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
00153         char addr_str[MAX_SOCKADDR_STR_LEN];
00154         sockaddr_in_to_string(cl_addr, addr_str);
00155         LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
00156         continue;
00157       } else{
00158         LOG(LOG_DEBUG, "Sender is the same!");
00159       }
00160     }
00161
00162     type = tftp_msg_type(in_buffer);
00163     if (type == TFTP_TYPE_ERROR){
00164       int error_code;
00165       char error_msg[TFTP_MAX_ERROR_LEN];
00166
00167       ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
00168       if (ret != 0){
00169         LOG(LOG_ERR, "Error unpacking error msg");
00170         return 5;
00171       }
00172
00173       if (error_code == 1){
00174         LOG(LOG_INFO, "File not found");
00175         return 1;
00176       } else{
00177         LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
00178         return 7;
00179       }
00180
00181     } else if (type != TFTP_TYPE_DATA){
00182       LOG(LOG_ERR, "Received packet of type %d, expecting DATA or ERROR.",type);
00183       return 8;
00184     }
00185
00186     ret = tftp_msg_unpack_data(in_buffer, len, &rcv_block_n, data, &data_size);
00187
00188     if (ret != 0){
00189       LOG(LOG_ERR, "Error unpacking data: %d", ret);
00190       return 4;
00191     }
00192
00193     if (rcv_block_n != exp_block_n){
00194       LOG(LOG_ERR,
00195           "Received unexpected block_n: rcv_block_n = %d != %d = exp_block_n",
00196           rcv_block_n,
00197           exp_block_n
00198       );
00199       return 3;
00200     }
00201
00202     exp_block_n++;
00203
00204     LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);
00205
00206     if (data_size != 0){
00207       if (fblock_write(m_fblock, data, data_size))
00208         return 6;
```

```
00209     }
00210
00211     LOG(LOG_DEBUG, "Sending ack");
00212
00213     if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
00214       return 2;
00215
00216   } while(data_size == TFTP_DATA_BLOCK);
00217   return 0;
00218 }
00219
00220
00221 int tftp_receive_ack(int *block_n, char* in_buffer, int sd,
00222                      struct sockaddr_in *addr){
00223   int msglen, len, ret;
00224   unsigned int addrlen;
00225   struct sockaddr_in cl_addr;
00226
00227   msglen = tftp_msg_get_size_ack();
00228   addrlen = sizeof(cl_addr);
00229
00230   len = recvfrom(sd, in_buffer, msglen, 0,
00231                  (struct sockaddr*)&cl_addr,
00232                  &addrlen
00233   );
00234
00235   if (sockaddr_in_cmp(*addr, cl_addr) != 0){
00236     char str_addr[MAX_SOCKADDR_STR_LEN];
00237     sockaddr_in_to_string(cl_addr, str_addr);
00238     LOG(LOG_WARN, "Message is coming from unexpected source: %s", str_addr);
00239     return 2;
00240   }
00241
00242   if (len != msglen){
00243     LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
00244     return 1;
00245   }
00246
00247   ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
00248   if (ret != 0){
00249     LOG(LOG_ERR, "Error unpacking ack: %d", ret);
00250     return 8+ret;
00251   }
00252
00253   return 0;
00254 }
00255
00256
00257 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
00258   char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[
      TFTP_MAX_DATA_MSG_SIZE];
00259   int block_n, rcv_block_n;
00260   int len, data_size, msglen, ret;
00261
00262   if (m_fblock->remaining > TFTP_MAX_FILE_SIZE){
00263     LOG(LOG_ERR, "File is too big: %d", m_fblock->remaining);
00264     tftp_send_error(0, "File is too big.", sd, addr);
00265     return 4;
00266   }
00267
00268   // init sequence number
00269   block_n = 1;
00270
00271   do{
00272     LOG(LOG_DEBUG, "Sending part %d", block_n);
00273
00274     if (m_fblock->remaining > TFTP_DATA_BLOCK)
00275       data_size = TFTP_DATA_BLOCK;
00276     else
00277       data_size = m_fblock->remaining;
00278
00279     if (data_size != 0)
00280       fblock_read(m_fblock, data);
00281
00282     LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);
00283
00284     msglen = tftp_msg_get_size_data(data_size);
00285     tftp_msg_build_data(block_n, data, data_size, out_buffer);
00286
00287     // dump_buffer_hex(out_buffer, msglen);
00288
00289     len = sendto(sd, out_buffer, msglen, 0,
00290                  (struct sockaddr*)addr,
00291                  sizeof(*addr)
00292     );
00293
00294     if (len != msglen){
```

```
00295       return 1;
00296     }
00297
00298     LOG(LOG_DEBUG, "Waiting for ack");
00299
00300     ret = tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr);
00301
00302     if (ret == 2){  //unexpected source
00303       continue;
00304     } else if (ret != 0){
00305       LOG(LOG_ERR, "Error receiving ack: %d", ret);
00306       return 2;
00307     }
00308
00309     if (rcv_block_n != block_n){
00310       LOG(LOG_ERR, "Received wrong block n: received %d != expected %d",
00311           rcv_block_n,
00312           block_n
00313       );
00314       return 3;
00315     }
00316
00317     block_n++;
00318
00319   } while(data_size == TFTP_DATA_BLOCK);
00320   return 0;
00321 }
```

## 5.21 tftp.h File Reference

Common functions for TFTP client and server.

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

### Macros

- #define TFTP_MAX_FILE_SIZE 33554431

  *Maximum file size to prevent block # overflow.*

### Functions

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

  *Send a RRQ message to a server.*
- int tftp_send_wrq (char *filename, char *mode, int sd, struct sockaddr_in *addr)

  *Send a WRQ message to a server.*
- int tftp_send_error (int error_code, char *error_msg, int sd, struct sockaddr_in *addr)

  *Send an ERROR message to the client (server).*
- int tftp_send_ack (int block_n, char *out_buffer, int sd, struct sockaddr_in *addr)

  *Send an ACK message.*
- int tftp_receive_file (struct fblock *m_fblock, int sd, struct sockaddr_in *addr)

  *Handle the entire workflow required to receive a file.*
- int tftp_receive_ack (int *block_n, char *in_buffer, int sd, struct sockaddr_in *addr)

  *Receive an ACK message.*
- int tftp_send_file (struct fblock *m_fblock, int sd, struct sockaddr_in *addr)

  *Handle the entire workflow required to send a file.*

### 5.21.1 Detailed Description

Common functions for TFTP client and server.

**Author**

Riccardo Mancini

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

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

Definition in file tftp.h.

### 5.21.2 Function Documentation

#### 5.21.2.1 tftp_receive_ack()

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

Receive an ACK message.

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

**Parameters**

| | |
|---|---|
| *block_n* | [out] sequence number of the acknowledged block. |
| *in_buffer* | buffer to be used for receiving the ACK (useful for recycling the same buffer) |
| *sd* | [in] socket id of the (UDP) socket to be used to send the message |
| *addr* | [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 221 of file tftp.c.

#### 5.21.2.2 tftp_receive_file()

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

Handle the entire workflow required to receive a file.

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

**Parameters**

| | |
|---|---|
| *m_fblock* | block file where to write incoming data to |
| *sd* | socket id of the (UDP) socket to be used to send ACK messages |
| *addr* | 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 en error while writing to the file.
- 7 in case of an error message different from File Not Found (since it is the only erorr available in current implementation).
- 8 in case of the incoming message is neither DATA nor ERROR.

Definition at line 118 of file tftp.c.

#### 5.21.2.3 tftp_send_ack()

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

Send an ACK message.

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

**Parameters**

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

**Returns**

0 in case of success, 1 otherwise

Definition at line 97 of file tftp.c.

#### 5.21.2.4 tftp_send_error()

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

Send an ERROR message to the client (server).

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

**Parameters**

| | |
|---|---|
| *error_code* | the code of the error (must be within 0 and 7) |
| *error_msg* | the message explaining the error |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the client (server) |

**Returns**

0 in case of success, 1 otherwise

Definition at line 73 of file tftp.c.

#### 5.21.2.5 tftp_send_file()

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

Handle the entire workflow required to send a file.

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

**Parameters**

| | |
|---|---|
| *m_fblock* | block file where to read incoming data from |
| *sd* | socket id of the (UDP) socket to be used to send DATA messages |
| *addr* | 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 file too big

Definition at line 257 of file tftp.c.

#### 5.21.2.6 tftp_send_rrq()

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

Send a RRQ message to a server.

**Parameters**

| | |
|---|---|
| *filename* | the name of the requested file |
| *mode* | the desired mode of transfer (netascii or octet) |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the server |

**Returns**

0 in case of success, 1 otherwise

**See also**

TFTP_STR_NETASCII
TFTP_STR_OCTET

Definition at line 27 of file tftp.c.

#### 5.21.2.7 tftp_send_wrq()

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

Send a WRQ message to a server.

Do not used in current implementation.

**Parameters**

| | |
|---|---|
| *filename* | the name of the requested file |
| *mode* | the desired mode of transfer (netascii or octet) |
| *sd* | socket id of the (UDP) socket to be used to send the message |
| *addr* | address of the server |

**Returns**

0 in case of success, 1 otherwise

**See also**

TFTP_STR_NETASCII
TFTP_STR_OCTET

Definition at line 50 of file tftp.c.

## 5.22 tftp.h

```
00001
00015 #ifndef TFTP
00016 #define TFTP
00017
00018
00019 #include <sys/socket.h>
00020 #include <netinet/in.h>
00021 #include "fblock.h"
00022
00024 #define TFTP_MAX_FILE_SIZE 33554431
00025
00026
00039 int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
00040
00055 int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
00056
00070 int tftp_send_error(int error_code, char* error_msg, int sd,
00071                     struct sockaddr_in *addr);
00072
00087 int tftp_send_ack(int block_n, char* out_buffer, int sd,
00088                   struct sockaddr_in *addr);
00089
00112 int tftp_receive_file(struct fblock *m_fblock, int sd,
00113                       struct sockaddr_in *addr);
00114
00134 int tftp_receive_ack(int *block_n, char* in_buffer, int sd,
00135                      struct sockaddr_in *addr);
00136
00154 int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
00155
00156
00157 #endif
```

## 5.23 tftp_client.c File Reference

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

```
#include "include/logging.h"
#include "include/tftp_msgs.h"
#include "include/tftp.h"
#include "include/fblock.h"
#include "include/inet_utils.h"
```

```
#include "include/debug_utils.h"
#include "include/netascii.h"
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

### Macros

- #define READ_BUFFER_SIZE 80

  *max stdin line length*
- #define MAX_ARGS 3

  *Maximum number of arguments for commands.*
- #define MODE_TXT "txt"

  *String for txt.*
- #define MODE_BIN "bin"

  *String for bin.*

### Functions

- void split_string (char ∗line, char ∗delim, int max_argc, int ∗argc, char ∗∗argv)

  *Splits a string at each delim.*
- void print_help ()

  *Prints command usage information.*
- void cmd_help ()

  *Handles !help command, printing information about available commands.*
- void cmd_mode (char ∗new_mode)

  *Handles !mode command, changing transfer_mode to either bin or text.*
- int cmd_get (char ∗remote_filename, char ∗local_filename, char ∗sv_ip, int sv_port)

  *Handles !get command, reading file from server.*
- void cmd_quit ()

  *Handles !quit command.*
- int main (int argc, char ∗∗argv)

  *Main.*

### Variables

- const int LOG_LEVEL = LOG_WARN

  *Defining LOG_LEVEL for tftp_client executable.*
- char ∗ transfer_mode

  *Global transfer_mode variable for storing user chosen transfer mode string.*

### 5.23.1 Detailed Description

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

**Author**

Riccardo Mancini

Definition in file tftp_client.c.

### 5.23.2 Function Documentation

#### 5.23.2.1 cmd_mode()

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

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

**See also**

transfer_mode

Definition at line 123 of file tftp_client.c.

#### 5.23.2.2 split_string()

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

Splits a string at each delim.

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

**Parameters**

| | |
|---|---|
| *line* | [in] the string to split |
| *delim* | [in] the delimiter |
| *max_argc* | [in] maximum number of parts to split the line into |
| *argc* | [out] counts of the parts the line is split into |
| *argv* | [out] array of parts the line is split into |

Prints command usage information.

Definition at line 63 of file tftp_client.c.

#### 5.23.3 Variable Documentation

##### 5.23.3.1 LOG_LEVEL

```
const int LOG_LEVEL = LOG_WARN
```

Defining LOG_LEVEL for tftp_client executable.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

##### 5.23.3.2 transfer_mode

```
char* transfer_mode
```

Global transfer_mode variable for storing user chosen transfer mode string.

**See also**

MODE_TXT
MODE_BIN

Definition at line 48 of file tftp_client.c.

### 5.24 tftp_client.c

```
00001
00009 #include "include/logging.h"
00010 #include "include/tftp_msgs.h"
00011 #include "include/tftp.h"
00012 #include "include/fblock.h"
00013 #include "include/inet_utils.h"
00014 #include "include/debug_utils.h"
00015 #include "include/netascii.h"
00016 #include <arpa/inet.h>
00017 #include <sys/types.h>
00018 #include <sys/socket.h>
00019 #include <netinet/in.h>
00020 #include <string.h>
00021 #include <stdio.h>
00022 #include <stdlib.h>
00023 #include <time.h>
00024
00026 const int LOG_LEVEL = LOG_WARN;
00027
00028
00030 #define READ_BUFFER_SIZE 80
00031
00033 #define MAX_ARGS 3
00034
00036 #define MODE_TXT "txt"
```

```
00037
00039 #define MODE_BIN "bin"
00040
00041
00048 char* transfer_mode;
00049
00050
00063 void split_string(char* line, char* delim, int max_argc, int *argc,
00064                   char **argv){
00065   char *ptr;
00066   int len;
00069   char *pos;
00070
00071   // remove trailing LF
00072   if ((pos=strchr(line, '\n')) != NULL)
00073     *pos = '\0';
00074
00075   // init argc
00076   *argc = 0;
00077
00078   // tokenize string
00079   ptr = strtok(line, delim);
00080
00081   while(ptr != NULL && *argc <= max_argc){
00082     len = strlen(ptr);
00083
00084     if (len == 0)
00085       continue;
00086
00087     LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);
00088
00089     argv[*argc] = malloc(strlen(ptr)+1);
00090     strcpy(argv[*argc], ptr);
00091
00092     ptr = strtok(NULL, delim);
00093     (*argc)++;
00094   }
00095 }
00096
00100 void print_help(){
00101   printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
00102   printf("Example: ./tftp_client 127.0.0.1 69");
00103 }
00104
00108 void cmd_help(){
00109   printf("Sono disponibili i seguenti comandi:\n");
00110   printf("!help --> mostra l'elenco dei comandi disponibili\n");
00111   printf("!mode {txt|bin} --> imposta il modo di trasferimento ");
00112   printf("dei file (testo o binario)\n");
00113   printf("!get filename nome_locale --> richiede al server il nome del file ");
00114   printf("<filename> e lo salva localmente con il nome <nome_locale>\n");
00115   printf("!quit --> termina il client\n");
00116 }
00117
00123 void cmd_mode(char* new_mode){
00124   if (strcmp(new_mode, MODE_TXT) == 0){
00125     transfer_mode = TFTP_STR_NETASCII;
00126     printf("Modo di trasferimento testo configurato\n");
00127   } else if (strcmp(new_mode, MODE_BIN) == 0){
00128     transfer_mode = TFTP_STR_OCTET;
00129     printf("Modo di trasferimento binario configurato\n");
00130   } else{
00131     printf("Modo di traferimento sconosciuto: %s. Modi disponibili: txt, bin\n",
00132            new_mode
00133     );
00134   }
00135 }
00136
00140 int cmd_get(char* remote_filename, char* local_filename, char* sv_ip,
00141             int sv_port){
00142   struct sockaddr_in my_addr, sv_addr;
00143   int sd;
00144   int ret, tid, result;
00145   struct fblock m_fblock;
00146   char *tmp_filename;
00147
00148   LOG(LOG_INFO, "Initializing...\n");
00149
00150   sd = socket(AF_INET, SOCK_DGRAM, 0);
00151   if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
00152     m_fblock = fblock_open(local_filename,
00153                            TFTP_DATA_BLOCK,
00154                            FBLOCK_WRITE|FBLOCK_MODE_BINARY
00155     );
00156   else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
00157     tmp_filename = malloc(strlen(local_filename)+5);
00158     strcpy(tmp_filename, local_filename);
```

```
00159     strcat(tmp_filename, ".tmp");
00160     m_fblock = fblock_open(tmp_filename,
00161                            TFTP_DATA_BLOCK,
00162                            FBLOCK_WRITE|FBLOCK_MODE_TEXT
00163     );
00164   }else
00165     return 2;
00166
00167   LOG(LOG_INFO, "Opening socket...");
00168
00169   sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
00170   my_addr = make_my_sockaddr_in(0);
00171   tid = bind_random_port(sd, &my_addr);
00172   if (tid == 0){
00173     LOG(LOG_ERR, "Error while binding to random port");
00174     perror("Could not bind to random port:");
00175     fblock_close(&m_fblock);
00176     return 1;
00177   } else
00178     LOG(LOG_INFO, "Bound to port %d", tid);
00179
00180   printf("Richiesta file %s (%s) al server in corso.\n",
00181          remote_filename,
00182          transfer_mode
00183   );
00184
00185   ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
00186   if (ret != 0){
00187     fblock_close(&m_fblock);
00188     return 8+ret;
00189   }
00190
00191   printf("Trasferimento file in corso.\n");
00192
00193   ret = tftp_receive_file(&m_fblock, sd, &sv_addr);
00194
00195
00196   if (ret == 1){    // File not found
00197     printf("File non trovato.\n");
00198     result = 0;
00199   } else if (ret != 0){
00200     LOG(LOG_ERR, "Error while receiving file!");
00201     result = 16+ret;
00202   } else{
00203     int n_blocks = (m_fblock.written+m_fblock.block_size-1)/m_fblock.
      block_size;
00204     printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);
00205     printf("Salvataggio %s completato.\n", local_filename);
00206
00207     result = 0;
00208   }
00209
00210   fblock_close(&m_fblock);
00211   if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
00212     netascii2unix(tmp_filename, local_filename);
00213     remove(tmp_filename);
00214     free(tmp_filename);
00215   }
00216
00217   return result;
00218
00219 }
00220
00224 void cmd_quit(){
00225   printf("Client terminato con successo\n");
00226   exit(0);
00227 }
00228
00230 int main(int argc, char** argv){
00231   char* sv_ip;
00232   short int sv_port;
00233   int ret, i;
00234   char read_buffer[READ_BUFFER_SIZE];
00235   int cmd_argc;
00236   char *cmd_argv[MAX_ARGS];
00237
00238   //init random seed
00239   srand(time(NULL));
00240
00241   // default mode = bin
00242   transfer_mode = TFTP_STR_OCTET;
00243
00244   if (argc != 3){
00245     print_help();
00246     return 1;
00247   }
00248
```

```
00249   // TODO: check args
00250   sv_ip = argv[1];
00251   sv_port = atoi(argv[2]);
00252
00253   while(1){
00254     printf("> ");
00255     fflush(stdout); // flush stdout buffer
00256     fgets(read_buffer, READ_BUFFER_SIZE, stdin);
00257     split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
00258
00259     if (cmd_argc == 0){
00260       printf("Comando non riconosciuto : ''\n");
00261       cmd_help();
00262     } else{
00263       if (strcmp(cmd_argv[0], "!mode") == 0){
00264         if (cmd_argc == 2)
00265           cmd_mode(cmd_argv[1]);
00266         else
00267           printf("Il comando richiede un solo argomento: bin o txt\n");
00268       } else if (strcmp(cmd_argv[0], "!get") == 0){
00269         if (cmd_argc == 3){
00270           ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
00271           LOG(LOG_DEBUG, "cmd_get returned value: %d", ret);
00272         } else{
00273           printf("Il comando richiede due argomenti:");
00274           printf(" <filename> e <nome_locale>\n");
00275         }
00276       } else if (strcmp(cmd_argv[0], "!quit") == 0){
00277         if (cmd_argc == 1){
00278           cmd_quit();
00279         } else{
00280           printf("Il comando non richiede argomenti\n");
00281         }
00282       } else if (strcmp(cmd_argv[0], "!help") == 0){
00283         if (cmd_argc == 1){
00284           cmd_help();
00285         } else{
00286           printf("Il comando non richiede argomenti\n");
00287         }
00288       } else {
00289         printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
00290         cmd_help();
00291       }
00292     }
00293
00294     // Free malloc'ed strings
00295     for(i = 0; i < cmd_argc; i++)
00296       free(cmd_argv[i]);
00297   }
00298
00299   return 0;
00300 }
```

## 5.25 tftp_msgs.c File Reference

Implementation of tftp_msgs.h .

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

### Functions

- int tftp_msg_type (char *buffer)

  *Retuns msg type given a message buffer.*
- void tftp_msg_build_rrq (char *filename, char *mode, char *buffer)

  *Builds a read request message.*

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

  *Unpacks a read request message.*
- int tftp_msg_get_size_rrq (char *filename, char *mode)

  *Returns size in bytes of a read request message.*
- void tftp_msg_build_wrq (char *filename, char *mode, char *buffer)

  *Builds a write request message.*
- int **tftp_msg_unpack_wrq** (char *buffer, int buffer_len, char *filename, char *mode)
- int tftp_msg_get_size_wrq (char *filename, char *mode)

  *Returns size in bytes of a write request message.*
- void tftp_msg_build_data (int block_n, char *data, int data_size, char *buffer)

  *Builds a data message.*
- int tftp_msg_unpack_data (char *buffer, int buffer_len, int *block_n, char *data, int *data_size)

  *Unpacks a data message.*
- int tftp_msg_get_size_data (int data_size)

  *Returns size in bytes of a data message.*
- void tftp_msg_build_ack (int block_n, char *buffer)

  *Builds an acknowledgment message.*
- int tftp_msg_unpack_ack (char *buffer, int buffer_len, int *block_n)

  *Unpacks an acknowledgment message.*
- int tftp_msg_get_size_ack ()

  *Returns size in bytes of an acknowledgment message.*
- void tftp_msg_build_error (int error_code, char *error_msg, char *buffer)

  *Builds an error message.*
- int tftp_msg_unpack_error (char *buffer, int buffer_len, int *error_code, char *error_msg)

  *Unpacks an error message.*
- int tftp_msg_get_size_error (char *error_msg)

  *Returns size in bytes of an error message.*

#### Variables

- const int LOG_LEVEL

  *LOG_LEVEL will be defined in another file.*

### 5.25.1 Detailed Description

Implementation of tftp_msgs.h .

#### Author

Riccardo Mancini

#### See also

tftp_msgs.h

Definition in file tftp_msgs.c.

### 5.25.2 Function Documentation

#### 5.25.2.1 tftp_msg_build_ack()

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

Builds an acknowledgment message.

Message format:

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

**Parameters**

| | |
|---|---|
| *block_n* | block sequence number |
| *buffer* | data buffer where to build the message |

Definition at line 178 of file tftp_msgs.c.

#### 5.25.2.2 tftp_msg_build_data()

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

Builds a data message.

Message format:

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

**Parameters**

| | |
|---|---|
| *block_n* | block sequence number |
| *data* | pointer to the buffer containing the data to be transfered |
| *data_size* | data buffer size |
| *buffer* | data buffer where to build the message |

Definition at line 145 of file tftp_msgs.c.

#### 5.25.2.3 tftp_msg_build_error()

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

Builds an error message.

Message format:

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

Error code meaning:

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

In current implementation only errors 1 and 4 are implemented.

**Parameters**

| | |
|---|---|
| *error_code* | error code (from 0 to 7) |
| *error_msg* | error message |
| *buffer* | data buffer where to build the message |

Definition at line 204 of file tftp_msgs.c.

#### 5.25.2.4 tftp_msg_build_rrq()

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

Builds a read request message.

```
 2 bytes    string    1 byte    string    1 byte
 -----------------------------------------------
|   01  |  Filename  |   0  |    Mode    |   0  |
 -----------------------------------------------
```

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |
| *buffer* | data buffer where to build the message |

Definition at line 29 of file tftp_msgs.c.

#### 5.25.2.5 tftp_msg_build_wrq()

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

Builds a write request message.

Message format:

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

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |
| *buffer* | data buffer where to build the message |

Definition at line 86 of file tftp_msgs.c.

#### 5.25.2.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

**Parameters**

| | |
|---|---|
| *data_size* | data buffer size |

**Returns**

size in bytes

Definition at line 199 of file tftp_msgs.c.

#### 5.25.2.7 tftp_msg_get_size_data()

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

Returns size in bytes of a data message.

It just sums 4 to data_size.

**Parameters**

| | |
|---|---|
| *data_size* | data buffer size |

**Returns**

size in bytes

Definition at line 173 of file tftp_msgs.c.

#### 5.25.2.8 tftp_msg_get_size_error()

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

Returns size in bytes of an error message.

**Parameters**

| | |
|---|---|
| *error_msg* | error message |

**Returns**

size in bytes

Definition at line 246 of file tftp_msgs.c.

#### 5.25.2.9 tftp_msg_get_size_rrq()

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

Returns size in bytes of a read request message.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |

**Returns**

size in bytes

Definition at line 81 of file tftp_msgs.c.

#### 5.25.2.10 tftp_msg_get_size_wrq()

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

Returns size in bytes of a write request message.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |

**Returns**

size in bytes

Definition at line 140 of file tftp_msgs.c.

#### 5.25.2.11 tftp_msg_type()

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

Retuns msg type given a message buffer.

**Parameters**

| | |
|---|---|
| *buffer* | the buffer |

**Returns**

message type

**See also**

TFTP_TYPE_RRQ
TFTP_TYPE_WRQ
TFTP_TYPE_DATA
TFTP_TYPE_ACK
TFTP_TYPE_ERROR

Definition at line 24 of file tftp_msgs.c.

#### 5.25.2.12 tftp_msg_unpack_ack()

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

Unpacks an acknowledgment message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *block_n* | pointer where block_n will be written [out] |
| *data* | 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 184 of file tftp_msgs.c.

#### 5.25.2.13 tftp_msg_unpack_data()

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

Unpacks a data message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *block_n* | pointer where block_n will be written [out] |
| *data* | 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 153 of file tftp_msgs.c.

#### 5.25.2.14 tftp_msg_unpack_error()

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

Unpacks an error message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *error_code* | pointer where error_code will be written [out] |
| *error_msg* | 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 unrecognize error code (must be within 0 and 7).

**See also**

TFTP_TYPE_ERROR
TFTP_MAX_ERROR_LEN

Definition at line 212 of file tftp_msgs.c.

#### 5.25.2.15 tftp_msg_unpack_rrq()

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

Unpacks a read request message.

Unpacks a write request message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *filename* | name of the file [out] |
| *mode* | 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**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *filename* | name of the file [out] |
| *mode* | 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 38 of file tftp_msgs.c.

#### 5.25.3 Variable Documentation

##### 5.25.3.1 LOG_LEVEL

```
const int LOG_LEVEL
```

LOG_LEVEL will be defined in another file.

LOG_LEVEL will be defined in another file.

Definition at line 26 of file tftp_client.c.

## 5.26 tftp_msgs.c

```
00001
00011 #include "include/tftp_msgs.h"
00012 #include "include/logging.h"
00013 #include <string.h>
00014 #include <strings.h>
00015 #include <stdio.h>
00016 #include <arpa/inet.h>
00017 #include <stdint.h>
00018
00019
00021 extern const int LOG_LEVEL;
00022
00023
00024 int tftp_msg_type(char *buffer){
00025   return (int) ntohs(*(uint16_t*)buffer);
00026 }
00027
00028
00029 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
00030   *((uint16_t*)buffer) = htons(TFTP_TYPE_RRQ);
00031   buffer += 2;
00032   strcpy(buffer, filename);
00033   buffer += strlen(filename)+1;
00034   strcpy(buffer, mode);
00035 }
```

```
00036
00037
00038 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
      mode){
00039   int offset = 0;
00040   if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
00041     LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
00042     return 1;
00043   }
00044
00045   offset += 2;
00046   if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00047     LOG(LOG_ERR, "Filename too long (%d > %d): %s",
00048         (int) strlen(buffer+offset),
00049         TFTP_MAX_FILENAME_LEN, buffer+offset
00050     );
00051     return 3;
00052   }
00053   strcpy(filename, buffer+offset);
00054
00055   offset += strlen(filename)+1;
00056   if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00057     LOG(LOG_ERR, "Mode string too long (%d > %d): %s",
00058         (int) strlen(buffer+offset),
00059         TFTP_MAX_MODE_LEN,
00060         buffer+offset
00061     );
00062     return 4;
00063   }
00064   strcpy(mode, buffer+offset);
00065
00066   offset += strlen(mode)+1;
00067   if (buffer_len != offset){
00068     LOG(LOG_ERR, "Packet contains unexpected fields");
00069     return 2;
00070   }
00071   if (strcasecmp(mode, TFTP_STR_NETASCII) == 0 ||
00072       strcasecmp(mode, TFTP_STR_OCTET) == 0)
00073     return 0;
00074   else{
00075     LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00076     return 5;
00077   }
00078 }
00079
00080
00081 int tftp_msg_get_size_rrq(char* filename, char* mode){
00082   return 4 + strlen(filename) + strlen(mode);
00083 }
00084
00085
00086 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
00087   *((uint16_t*)buffer) = htons(TFTP_TYPE_WRQ);
00088   buffer += 2;
00089   strcpy(buffer, filename);
00090   buffer += strlen(filename)+1;
00091   strcpy(buffer, mode);
00092 }
00093
00094
00095 int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename,
00096                         char* mode){
00097   int offset = 0;
00098   if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
00099     LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
00100     return 1;
00101   }
00102
00103   offset += 2;
00104   if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
00105     LOG(LOG_ERR, "Filename too long (%d > %d): %s",
00106         (int) strlen(buffer+offset),
00107         TFTP_MAX_FILENAME_LEN,
00108         buffer+offset
00109     );
00110     return 3;
00111   }
00112
00113   strcpy(filename, buffer+offset);
00114   offset += strlen(filename)+1;
00115   if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
00116     LOG(LOG_ERR, "Mode string too long (%d > %d): %s",
00117         (int) strlen(buffer+offset),
00118         TFTP_MAX_MODE_LEN,
00119         buffer+offset
00120     );
00121     return 4;
```

```
00122   }
00123
00124   strcpy(mode, buffer+offset);
00125   offset += strlen(mode)+1;
00126   if (buffer_len != offset){
00127     LOG(LOG_ERR, "Packet contains unexpected fields");
00128     return 2;
00129   }
00130
00131   if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode,
      TFTP_STR_OCTET) == 0)
00132     return 0;
00133   else{
00134     LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
00135     return 5;
00136   }
00137 }
00138
00139
00140 int tftp_msg_get_size_wrq(char* filename, char* mode){
00141   return 4 + strlen(filename) + strlen(mode);
00142 }
00143
00144
00145 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
00146   *((uint16_t*)buffer) = htons(TFTP_TYPE_DATA);
00147   *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
00148   buffer += 4;
00149   memcpy(buffer, data, data_size);
00150 }
00151
00152
00153 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
00154                          int* data_size){
00155   if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
00156     LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
00157     return 1;
00158   }
00159
00160   if (buffer_len < 4){
00161     LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
00162     return 2;
00163   }
00164
00165   *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
00166   *data_size = buffer_len - 4;
00167   if (*data_size > 0)
00168     memcpy(data, buffer+4, *data_size);
00169   return 0;
00170 }
00171
00172
00173 int tftp_msg_get_size_data(int data_size){
00174   return data_size + 4;
00175 }
00176
00177
00178 void tftp_msg_build_ack(int block_n, char* buffer){
00179   *((uint16_t*)buffer) = htons(TFTP_TYPE_ACK);
00180   *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
00181 }
00182
00183
00184 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
00185   if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
00186     LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
00187     return 1;
00188   }
00189
00190   if (buffer_len != 4){
00191     LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
00192     return 2;
00193   }
00194   *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
00195   return 0;
00196 }
00197
00198
00199 int tftp_msg_get_size_ack(){
00200   return 4;
00201 }
00202
00203
00204 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
00205   *((uint16_t*)buffer) = htons(TFTP_TYPE_ERROR);
00206   *((uint16_t*)(buffer+2)) = htons((uint16_t) error_code);
00207   buffer += 4;
```

```
00208   strcpy(buffer, error_msg);
00209 }
00210
00211
00212 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code,
00213                           char* error_msg){
00214   if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
00215       LOG(LOG_ERR, "Expected ERROR message (5), found %d",
00216           tftp_msg_type(buffer)
00217       );
00218       return 1;
00219     }
00220
00221     *error_code = (int) ntohs(*((uint16_t*)(buffer+2)));
00222     if (*error_code < 0 || *error_code > 7){
00223       LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
00224       return 4;
00225     }
00226
00227     buffer += 4;
00228     if(strlen(buffer) > TFTP_MAX_ERROR_LEN){
00229       LOG(LOG_ERR, "Error string too long (%d > %d): %s",
00230           (int) strlen(buffer),
00231           TFTP_MAX_ERROR_LEN,
00232           buffer
00233       );
00234       return 3;
00235     }
00236
00237     strcpy(error_msg, buffer);
00238     if (buffer_len != strlen(error_msg)+5){
00239       LOG(LOG_WARN, "Packet contains unexpected fields");
00240       return 2;
00241     }
00242     return 0;
00243 }
00244
00245
00246 int tftp_msg_get_size_error(char* error_msg){
00247   return 5 + strlen(error_msg);
00248 }
```

## 5.27 tftp_msgs.h File Reference

Contructor 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)

  *Retuns msg type given a message buffer.*
- void tftp_msg_build_rrq (char *filename, char *mode, char *buffer)

  *Builds a read request message.*
- int tftp_msg_unpack_rrq (char *buffer, int buffer_len, char *filename, char *mode)

  *Unpacks a read request message.*
- int tftp_msg_get_size_rrq (char *filename, char *mode)

  *Returns size in bytes of a read request message.*
- void tftp_msg_build_wrq (char *filename, char *mode, char *buffer)

  *Builds a write request message.*
- int tftp_msg_get_size_wrq (char *filename, char *mode)

  *Returns size in bytes of a write request message.*
- void tftp_msg_build_data (int block_n, char *data, int data_size, char *buffer)

  *Builds a data message.*
- int tftp_msg_unpack_data (char *buffer, int buffer_len, int *block_n, char *data, int *data_size)

  *Unpacks a data message.*
- int tftp_msg_get_size_data (int data_size)

  *Returns size in bytes of a data message.*
- void tftp_msg_build_ack (int block_n, char *buffer)

  *Builds an acknowledgment message.*
- int tftp_msg_unpack_ack (char *buffer, int buffer_len, int *block_n)

  *Unpacks an acknowledgment message.*
- int tftp_msg_get_size_ack ()

  *Returns size in bytes of an acknowledgment message.*
- void tftp_msg_build_error (int error_code, char *error_msg, char *buffer)

  *Builds an error message.*
- int tftp_msg_unpack_error (char *buffer, int buffer_len, int *error_code, char *error_msg)

  *Unpacks an error message.*
- int tftp_msg_get_size_error (char *error_msg)

  *Returns size in bytes of an error message.*

#### 5.27.1 Detailed Description

Contructor for TFTP messages.

**Author**

Riccardo Mancini

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

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

Definition in file tftp_msgs.h.

### 5.27.2 Macro Definition Documentation

#### 5.27.2.1 TFTP_MAX_MODE_LEN

```
#define TFTP_MAX_MODE_LEN 8
```

Maximum mode field string length.

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

Definition at line 50 of file tftp_msgs.h.

### 5.27.3 Function Documentation

#### 5.27.3.1 tftp_msg_build_ack()

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

Builds an acknowledgment message.

Message format:

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

**Parameters**

| | |
|---|---|
| *block_n* | block sequence number |
| *buffer* | data buffer where to build the message |

Definition at line 178 of file tftp_msgs.c.

#### 5.27.3.2 tftp_msg_build_data()

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

Builds a data message.

Message format:

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

**Parameters**

| | |
|---|---|
| *block_n* | block sequence number |
| *data* | pointer to the buffer containing the data to be transfered |
| *data_size* | data buffer size |
| *buffer* | data buffer where to build the message |

Definition at line 145 of file tftp_msgs.c.

#### 5.27.3.3 tftp_msg_build_error()

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

Builds an error message.

Message format:

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

Error code meaning:

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

In current implementation only errors 1 and 4 are implemented.

**Parameters**

| | |
|---|---|
| *error_code* | error code (from 0 to 7) |
| *error_msg* | error message |
| *buffer* | data buffer where to build the message |

Definition at line 204 of file tftp_msgs.c.

#### 5.27.3.4 tftp_msg_build_rrq()

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

Builds a read request message.

```
 2 bytes    string    1 byte    string    1 byte
 ------------------------------------------------
|   01   |  Filename  |   0   |    Mode    |   0   |
 ------------------------------------------------
```

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |
| *buffer* | data buffer where to build the message |

Definition at line 29 of file tftp_msgs.c.

#### 5.27.3.5 tftp_msg_build_wrq()

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

Builds a write request message.

Message format:

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

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |
| *buffer* | data buffer where to build the message |

Definition at line 86 of file tftp_msgs.c.

#### 5.27.3.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

**Parameters**

| | |
|---|---|
| *data_size* | data buffer size |

**Returns**

size in bytes

Definition at line 199 of file tftp_msgs.c.

#### 5.27.3.7 tftp_msg_get_size_data()

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

Returns size in bytes of a data message.

It just sums 4 to data_size.

**Parameters**

| | |
|---|---|
| *data_size* | data buffer size |

**Returns**

size in bytes

Definition at line 173 of file tftp_msgs.c.

#### 5.27.3.8 tftp_msg_get_size_error()

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

Returns size in bytes of an error message.

**Parameters**

| | |
|---|---|
| *error_msg* | error message |

**Returns**

size in bytes

Definition at line 246 of file tftp_msgs.c.

#### 5.27.3.9 tftp_msg_get_size_rrq()

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

Returns size in bytes of a read request message.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |

**Returns**

size in bytes

Definition at line 81 of file tftp_msgs.c.

#### 5.27.3.10 tftp_msg_get_size_wrq()

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

Returns size in bytes of a write request message.

**Parameters**

| | |
|---|---|
| *filename* | name of the file |
| *mode* | requested transfer mode ("netascii" or "octet") |

**Returns**

size in bytes

Definition at line 140 of file tftp_msgs.c.

#### 5.27.3.11 tftp_msg_type()

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

Retuns msg type given a message buffer.

**Parameters**

| | |
|---|---|
| *buffer* | the buffer |

**Returns**

message type

**See also**

TFTP_TYPE_RRQ
TFTP_TYPE_WRQ
TFTP_TYPE_DATA
TFTP_TYPE_ACK
TFTP_TYPE_ERROR

Definition at line 24 of file tftp_msgs.c.

#### 5.27.3.12 tftp_msg_unpack_ack()

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

Unpacks an acknowledgment message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *block_n* | pointer where block_n will be written [out] |
| *data* | 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 184 of file tftp_msgs.c.

#### 5.27.3.13 tftp_msg_unpack_data()

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

Unpacks a data message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *block_n* | pointer where block_n will be written [out] |
| *data* | 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 153 of file tftp_msgs.c.

#### 5.27.3.14 tftp_msg_unpack_error()

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

Unpacks an error message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *error_code* | pointer where error_code will be written [out] |
| *error_msg* | 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 unrecognize error code (must be within 0 and 7).

**See also**

TFTP_TYPE_ERROR
TFTP_MAX_ERROR_LEN

Definition at line 212 of file tftp_msgs.c.

#### 5.27.3.15 tftp_msg_unpack_rrq()

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

Unpacks a read request message.

Unpacks a write request message.

**Parameters**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *filename* | name of the file [out] |
| *mode* | 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**

| | |
|---|---|
| *buffer* | data buffer where the message to read is [in] |
| *buffer_len* | length of the buffer [in] |
| *filename* | name of the file [out] |
| *mode* | 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 38 of file tftp_msgs.c.

## 5.28 tftp_msgs.h

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

```
00022
00024 #define TFTP_TYPE_WRQ   2
00025
00027 #define TFTP_TYPE_DATA  3
00028
00030 #define TFTP_TYPE_ACK   4
00031
00033 #define TFTP_TYPE_ERROR 5
00034
00036 #define TFTP_STR_NETASCII "netascii"
00037
00039 #define TFTP_STR_OCTET "octet"
00040
00042 #define TFTP_MAX_FILENAME_LEN 255
00043
00050 #define TFTP_MAX_MODE_LEN 8
00051
00053 #define TFTP_MAX_ERROR_LEN 255
00054
00056 #define TFTP_DATA_BLOCK 512
00057
00059 #define TFTP_MAX_DATA_MSG_SIZE 516
00060
00061
00074 int tftp_msg_type(char *buffer);
00075
00076
00091 void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);
00092
00114 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename,
00115                         char* mode);
00116
00124 int tftp_msg_get_size_rrq(char* filename, char* mode);
00125
00141 void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);
00142
00164 int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename,
00165                         char* mode);
00166
00174 int tftp_msg_get_size_wrq(char* filename, char* mode);
00175
00192 void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);
00193
00208 int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
00209                          int* data_size);
00210
00219 int tftp_msg_get_size_data(int data_size);
00220
00235 void tftp_msg_build_ack(int block_n, char* buffer);
00236
00251 int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);
00252
00261 int tftp_msg_get_size_ack();
00262
00290 void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);
00291
00309 int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code,
00310                           char* error_msg);
00311
00318 int tftp_msg_get_size_error(char* error_msg);
00319
00320
00321 #endif
```

## 5.29 tftp_server.c File Reference

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

```
#include <stdlib.h>
#include "include/tftp_msgs.h"
#include "include/tftp.h"
#include "include/fblock.h"
#include "include/inet_utils.h"
#include "include/debug_utils.h"
#include "include/netascii.h"
#include <arpa/inet.h>
#include <sys/types.h>
```

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

#### Macros

- #define **_GNU_SOURCE**
- #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4

  *Maximum length for a RRQ message.*

#### Functions

- int strlcpl (const char *str1, const char *str2)

  *Finds longest common prefix length of strings str1 and str2.*
- int path_inside_dir (char *path, char *dir)

  *Check whether file is inside dir.*
- void print_help ()

  *Prints command usage information.*
- int send_file (char *filename, char *mode, struct sockaddr_in *cl_addr)

  *Sends file to a client.*
- int main (int argc, char **argv)

  *Main.*

#### Variables

- const int LOG_LEVEL = LOG_INFO

  *Defining LOG_LEVEL for tftp_server executable.*

### 5.29.1 Detailed Description

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

**Author**

Riccardo Mancini

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

Definition in file tftp_server.c.

### 5.29.2 Function Documentation

#### 5.29.2.1 path_inside_dir()

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

Check whether file is inside dir.

**Parameters**

| | |
|---|---|
| *path* | file absolute path (can include .. and . and multiple /) |
| *dir* | directory real path (can't include .. and . and multiple /) |

**Returns**

1 if true, 0 otherwise

**See also**

realpath

Definition at line 59 of file tftp_server.c.

#### 5.29.3 Variable Documentation

##### 5.29.3.1 LOG_LEVEL

```
const int LOG_LEVEL = LOG_INFO
```

Defining LOG_LEVEL for tftp_server executable.

LOG_LEVEL will be defined in another file.

Definition at line 36 of file tftp_server.c.

## 5.30 tftp_server.c

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

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

```
00146       result = 0;
00147     }
00148   }
00149
00150   fblock_close(&m_fblock);
00151
00152   if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
00153     LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
00154     remove(tmp_filename);
00155     free(tmp_filename);
00156   }
00157
00158   return result;
00159 }
00160
00162 int main(int argc, char** argv){
00163   short int my_port;
00164   char *dir_rel_path;
00165   char *ret_realpath;
00166   char dir_realpath[PATH_MAX];
00167   int ret, type, len;
00168   char in_buffer[MAX_MSG_LEN];
00169   unsigned int addrlen;
00170   int sd;
00171   struct sockaddr_in my_addr, cl_addr;
00172   int pid;
00173   char addr_str[MAX_SOCKADDR_STR_LEN];
00174
00175   if (argc != 3){
00176     print_help();
00177     return 1;
00178   }
00179
00180   my_port = atoi(argv[1]);
00181   dir_rel_path = argv[2];
00182
00183   ret_realpath = realpath(dir_rel_path, dir_realpath);
00184   if (ret_realpath == NULL){
00185     LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
00186     return 1;
00187   }
00188
00189   addrlen = sizeof(cl_addr);
00190
00191   sd = socket(AF_INET, SOCK_DGRAM, 0);
00192   my_addr = make_my_sockaddr_in(my_port);
00193   ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
00194   if (ret == -1){
00195     perror("Could not bind: ");
00196     LOG(LOG_FATAL, "Could not bind to port %d", my_port);
00197     return 1;
00198   }
00199
00200   LOG(LOG_INFO, "Server is running");
00201
00202   while (1){
00203     len = recvfrom(sd, in_buffer, MAX_MSG_LEN, 0,
00204                    (struct sockaddr*)&cl_addr,
00205                    &addrlen
00206     );
00207     type = tftp_msg_type(in_buffer);
00208     sockaddr_in_to_string(cl_addr, addr_str);
00209     LOG(LOG_INFO, "Received message with type %d from %s", type, addr_str);
00210     if (type == TFTP_TYPE_RRQ){
00211       pid = fork();
00212       if (pid == -1){ // error
00213         LOG(LOG_FATAL, "Fork error");
00214         perror("Fork error:");
00215         return 1;
00216       } else if (pid != 0 ){  // father
00217         LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
00218         continue; // father process continues loop
00219       } else{          // child
00220         char filename[TFTP_MAX_FILENAME_LEN], mode[
    TFTP_MAX_MODE_LEN];
00221         char file_path[PATH_MAX], file_realpath[PATH_MAX];
00222
00223         //init random seed
00224         srand(time(NULL));
00225
00226         ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);
00227
00228         if (ret != 0){
00229           LOG(LOG_WARN, "Error unpacking RRQ");
00230           tftp_send_error(0, "Malformed RRQ packet.", sd, &cl_addr);
00231           break; // child process exits loop
00232         }
```

```
00233
00234         strcpy(file_path, dir_realpath);
00235         strcat(file_path, "/");
00236         strcat(file_path, filename);
00237
00238         // check if file is inside directory (or inside any of its subdirs)
00239         if (!path_inside_dir(file_path, dir_realpath)){
00240           // it is not! I caught you, Trudy!
00241           LOG(LOG_WARN, "User tried to access file %s outside set directory %s",
00242               file_realpath,
00243               dir_realpath
00244           );
00245
00246           tftp_send_error(4, "Access violation.", sd, &cl_addr);
00247           break; // child process exits loop
00248         }
00249
00250         ret_realpath = realpath(file_path, file_realpath);
00251
00252         // file not found
00253         if (ret_realpath == NULL){
00254           LOG(LOG_WARN, "File not found: %s", file_path);
00255           tftp_send_error(1, "File Not Found.", sd, &cl_addr);
00256           break; // child process exits loop
00257         }
00258
00259         LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
00260
00261         ret = send_file(file_realpath, mode, &cl_addr);
00262         if (ret != 0)
00263           LOG(LOG_WARN, "Write terminated with an error: %d", ret);
00264         break;  // child process exits loop
00265       }
00266     } else{
00267       LOG(LOG_WARN, "Wrong op code: %d", type);
00268       tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
00269       // main process continues loop
00270     }
00271   }
00272
00273   LOG(LOG_INFO, "Exiting process %d", (int) getpid());
00274   return 0;
00275 }
```

# Index