TFTP

Generated by Doxygen 1.8.14

Contents

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

fblock
Structure which defines a file **2**

2 File Index

2.1 File List

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

3 Data Structure Documentation

3.1 fblock Struct Reference

Structure which defines a file.

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

Data Fields

- FILE ∗ file

  Pointer to the file.
- int block_size

  Predefined block size for i/o operations.
- char mode

  Can be read xor write, text xor binary.
- union {
    int written
      Bytes already written (for future use)
    int remaining
      Remaining bytes to read.
  };

3.1.1 Detailed Description

Structure which defines a file.

Definition at line 40 of file fblock.h.

3.1.2 Field Documentation

3.1.2.1 mode

```
char fblock::mode
```

Can be read xor write, text xor binary.

Definition at line 43 of file fblock.h.

4 File Documentation

4.1 debug_utils.c File Reference

Implementation of debug_utils.h.

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

Functions

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

4.1.1 Detailed Description

Implementation of debug_utils.h.

Author

Riccardo Mancini

See also

debug_utils.h

Definition in file debug_utils.c.

4.1.2 Function Documentation

4.1.2.1 dump_buffer_hex()

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

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

Parameters

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

Definition at line 18 of file debug_utils.c.

4.2 debug_utils.c

```
00001
00011 #include "include/debug_utils.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014 #include <stdlib.h>
00015 #include <string.h>
00016
00017
00018 void dump_buffer_hex(char* buffer, int len){
00019   char *str, tmp[4];
00020
```

```
  str = malloc(len*3+1);

  str[0] = '\0';
  for (int i=0; i<len; i++){
    sprintf(tmp, "%02x ", (unsigned char) buffer[i]);
    strcat(str, tmp);
  }

  LOG(LOG_DEBUG, "%s", str);
  free(str);
}
```

4.3 debug_utils.h File Reference

Utility functions for debugging.

Functions

- void dump_buffer_hex (char *buffer, int len)

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

4.3.1 Detailed Description

Utility functions for debugging.

Author

Riccardo Mancini

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

Definition in file debug_utils.h.

4.3.2 Function Documentation

4.3.2.1 dump_buffer_hex()

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

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

Parameters

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

Definition at line 18 of file debug_utils.c.

4.4 debug_utils.h

```

#ifndef DEBUG_UTILS
#define DEBUG_UTILS


void dump_buffer_hex(char* buffer, int len);


#endif
```

4.5 fblock.c File Reference

Implementation of fblock.h.

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

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.

Functions

- int get_length (FILE ∗f)

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

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

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

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

  Closes a file.

4.5.1 Detailed Description

Implementation of fblock.h.

Author

Riccardo Mancini

See also

fblock.h

Definition in file fblock.c.

4.5.2 Macro Definition Documentation

4.5.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 11 of file fblock.c.

4.5.3 Function Documentation

4.5.3.1 fblock_close()

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

Closes a file.

Parameters

m_fblock	fblock instance to be closed

Returns

0 in case of success, EOF in case of failure

See also

fclose

Definition at line 99 of file fblock.c.

4.5.3.2 fblock_open()

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

Opens a file.

Parameters

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 35 of file fblock.c.

4.5.3.3 fblock_read()

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

Reads next block_size bytes from file.

Parameters

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 73 of file fblock.c.

4.5.3.4 fblock_write()

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

Writes next block_size bytes to file.

Parameters

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 88 of file fblock.c.

4.5.3.5 get_length()

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

Returns file length.

Parameters

f	file pointer

Returns

file length in bytes

Definition at line 26 of file fblock.c.

4.6 fblock.c

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

```
  LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
      filename,
      (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "text",
      (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
      block_size
  );

  if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
    strcat(mode_str, "w");
    m_fblock.written = 0;
  } else {
    strcat(mode_str, "r");
  }

  if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
    strcat(mode_str, "b");
  // text otherwise

  m_fblock.file = fopen(filename, mode_str);
  if (m_fblock.file == NULL){
    LOG(LOG_ERR, "Error while opening file %s", filename);
    return m_fblock;
  }
  if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
    m_fblock.remaining = get_length(m_fblock.file);

  LOG(LOG_DEBUG, "Successfully opened file");
  return m_fblock;
}


int fblock_read(struct fblock *m_fblock, char* buffer){
  int bytes_read, bytes_to_read;

  if (m_fblock->remaining > m_fblock->block_size)
    bytes_to_read = m_fblock->block_size;
  else
    bytes_to_read = m_fblock->remaining;

  bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
  m_fblock->remaining -= bytes_read;

  return bytes_to_read - bytes_read;
}


int fblock_write(struct fblock *m_fblock, char* buffer, int block_size){
  int written_bytes;

  if (!block_size)
    block_size = m_fblock->block_size;

  written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->file);
  m_fblock->written += written_bytes;
  return block_size - written_bytes;
}

int fblock_close(struct fblock *m_fblock){
  return fclose(m_fblock->file);
}
```

4.7 fblock.h File Reference

File block read and write.

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

Data Structures

- struct fblock

    Structure which defines a file.

Macros

- #define FBLOCK_MODE_MASK 0b01

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

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

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

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

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

  Open file in write mode.

Functions

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

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

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

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

  Closes a file.

4.7.1 Detailed Description

File block read and write.

Author

Riccardo Mancini

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

Definition in file fblock.h.

4.7.2 Function Documentation

4.7.2.1 fblock_close()

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

Closes a file.

Parameters

m_fblock	fblock instance to be closed

Returns

0 in case of success, EOF in case of failure

See also

fclose

Definition at line 99 of file fblock.c.

4.7.2.2 fblock_open()

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

Opens a file.

Parameters

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 35 of file fblock.c.

4.7.2.3 fblock_read()

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

Reads next block_size bytes from file.

Parameters

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 73 of file fblock.c.

4.7.2.4 fblock_write()

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

Writes next block_size bytes to file.

Parameters

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 88 of file fblock.c.

4.8 fblock.h

```

#ifndef FBLOCK
#define FBLOCK


#include <stdio.h>


#define FBLOCK_MODE_MASK   0b01

#define FBLOCK_MODE_TEXT   0b00

#define FBLOCK_MODE_BINARY 0b01

#define FBLOCK_RW_MASK     0b10

#define FBLOCK_READ        0b00

#define FBLOCK_WRITE       0b10


struct fblock{
  FILE *file;
  int block_size;
  char mode;
  union{
    int written;
    int remaining;
  };
};


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

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

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

int fblock_close(struct fblock *m_fblock);


#endif
```

4.9 inet_utils.c File Reference

Implementation of inet_utils.h.

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

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.

Functions

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

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

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

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

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

  Converts sockaddr_in structure to string to be printed.

4.9.1 Detailed Description

Implementation of inet_utils.h.

Author

Riccardo Mancini

See also

inet_utils.h

Definition in file inet_utils.c.

4.9.2 Macro Definition Documentation

4.9.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file inet_utils.c.

4.9.3 Function Documentation

4.9.3.1 bind_random_port()

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

Binds socket to a random port.

Parameters

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.

4.9.3.2 make_my_sockaddr_in()

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

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

Parameters

port	port of the server

Returns

sockaddr_in structure this host on given port

Definition at line 55 of file inet_utils.c.

4.9.3.3 make_sv_sockaddr_in()

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

Makes sockaddr_in structure given ip string and port of server.

Parameters

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.

4.9.3.4 sockaddr_in_cmp()

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

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

Parameters

sai1	first address
sai2	second address

Returns

0 if therý're equal, 1 otherwise

Definition at line 65 of file inet_utils.c.

4.9.3.5 sockaddr_in_to_string()

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

Converts sockaddr_in structure to string to be printed.

Parameters

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

Definition at line 72 of file inet_utils.c.

4.10 inet_utils.c

```
#define LOG_LEVEL LOG_INFO


#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"


int bind_random_port(int socket, struct sockaddr_in *addr){
  int port, ret, i;
  for (i=0; i<MAX_TRIES; i++){
    if (i == 0) // first I generate a random one
      port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
    else  //if it's not free I scan the next one
      port = (port-FROM_PORT+1) % (TO_PORT - FROM_PORT + 1) +
FROM_PORT;

    LOG(LOG_DEBUG, "Trying port %d...", port);

    addr->sin_port = htons(port);
    ret = bind(socket, (struct sockaddr*) addr, sizeof(*addr));
    if (ret != -1)
      return port;
    // consider only some errors?
  }
  LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
  return 0;
}


struct sockaddr_in make_sv_sockaddr_in(char* ip, int port){
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  inet_pton(AF_INET, ip, &addr.sin_addr);
  return addr;
}


struct sockaddr_in make_my_sockaddr_in(int port){
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  addr.sin_addr.s_addr = htonl(INADDR_ANY);
  return addr;
}


int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
  if (sai1.sin_port == sai2.sin_port && sai1.sin_addr.s_addr == sai2.sin_addr.s_addr)
    return 0;
  else
    return 1;
}

void sockaddr_in_to_string(struct sockaddr_in src, char *dst){
  char* port_str;

  port_str = malloc(6);
  sprintf(port_str, "%d", ntohs(src.sin_port));

  if (inet_ntop(AF_INET, (void*) &src.sin_addr, dst, MAX_SOCKADDR_STR_LEN) != NULL){
    strcat(dst, ":");
    strcat(dst, port_str);
  } else{
    strcpy(dst, "ERROR");
  }

  free(port_str);
}
```

4.11 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.

4.11.1 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.

4.11.2 Function Documentation

4.11.2.1 bind_random_port()

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

Binds socket to a random port.

Parameters

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.

4.11.2.2 make_my_sockaddr_in()

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

Makes sockaddr_in structure of this host.

INADDR_ANY is used as IP address.

Parameters

port	port of the server

Returns

sockaddr_in structure this host on given port

Definition at line 55 of file inet_utils.c.

4.11.2.3 make_sv_sockaddr_in()

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

Makes sockaddr_in structure given ip string and port of server.

Parameters

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.

4.11.2.4 sockaddr_in_cmp()

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

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

Parameters

sai1	first address
sai2	second address

Returns

0 if thery're equal, 1 otherwise

Definition at line 65 of file inet_utils.c.

4.11.2.5 sockaddr_in_to_string()

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

Converts sockaddr_in structure to string to be printed.

Parameters

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

Definition at line 72 of file inet_utils.c.

4.12 inet_utils.h

```

#ifndef INET_UTILS
#define INET_UTILS


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

#define FROM_PORT 49152

#define TO_PORT   65535

#define MAX_TRIES 256
```

```

#define MAX_SOCKADDR_STR_LEN 22


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

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

struct sockaddr_in make_my_sockaddr_in(int port);

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

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


#endif
```

4.13 logging.h File Reference

Logging macro.

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

Macros

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

4.13.1 Detailed Description

Logging macro.

Author

Riccardo Mancini

This file contains a macro for logging in different levels.

There are 5 levels of logging:

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

The first three will be outputted to stderr, the latter two to stdout. You can define a per-file LOG_LEVEL for hiding some of the logging messages.

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

Definition in file logging.h.

4.14 logging.h

```

#ifndef LOGGING
#define LOGGING


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


#define LOG_FATAL    (1)
#define LOG_ERR      (2)
#define LOG_WARN     (3)
#define LOG_INFO     (4)
#define LOG_DEBUG    (5)


#ifndef LOG_LEVEL
  #define LOG_LEVEL LOG_DEBUG
#endif


#define LOG(level, ...) do {  \
                          if (level <= LOG_LEVEL) { \
                            FILE *dbgstream; \
                            char where[25]; \
                            switch(level){ \
                              case LOG_FATAL: \
                                dbgstream = stderr; \
                                fprintf(dbgstream, "[FATAL]"); \
                                break; \
                              case LOG_ERR: \
                                dbgstream = stderr; \
                                fprintf(dbgstream, "[ERROR]"); \
                                break; \
                              case LOG_WARN: \
                                dbgstream = stderr; \
                                fprintf(dbgstream, "[WARN ]"); \
                                break; \
                              case LOG_INFO: \
                                dbgstream = stdout; \
                                fprintf(dbgstream, "[INFO ]"); \
                                break; \
                              case LOG_DEBUG: \
                                dbgstream = stdout; \
                                fprintf(dbgstream, "[DEBUG]"); \
                                break; \
                            } \
                            fprintf(dbgstream, "[%-5d]", (int) getpid()); \
                            snprintf(where, 25, "%s:%d", __FILE__, __LINE__); \
                            fprintf(dbgstream, " %-25s ", where); \
                            fprintf(dbgstream, __VA_ARGS__); \
                            fprintf(dbgstream, "\n"); \
                            fflush(dbgstream); \
                          } \
                        } while (0)


#endif
```

4.15 netascii.c File Reference

Implementation of netascii.h.

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

Functions

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

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

  Netascii to Unix conversion.

4.15.1 Detailed Description

Implementation of netascii.h.

Author

Riccardo Mancini

See also

netascii.h

Definition in file netascii.c.

4.15.2 Function Documentation

4.15.2.1 netascii2unix()

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

Netascii to Unix conversion.

Parameters

netascii_filename	the filename of the input netascii file
unix_filename	the filename of the output Unix file

Returns

- 0 in case of success

Definition at line 87 of file netascii.c.

4.15.2.2 unix2netascii()

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

Unix to netascii conversion.

Parameters

unix_filename	the filename of the input Unix file
netascii_filename	the filename of the output netascii file

Returns

- 0 in case of success

Definition at line 16 of file netascii.c.

4.16 netascii.c

```
00001
00011 #include "include/netascii.h"
00012 #include "include/logging.h"
00013 #include <stdio.h>
00014
00015
00016 int unix2netascii(char *unix_filename, char* netascii_filename){
00017   FILE *unixf, *netasciif;
00018   char prev, tmp;
00019   int ret, result;
00020
00021   unixf = fopen(unix_filename, "r");
00022
00023   if (unixf == NULL){
00024     LOG(LOG_ERR, "Error opening file %s", unix_filename);
00025     return 1;
00026   }
00027
00028   netasciif = fopen(netascii_filename, "w");
00029
00030   if (unixf == NULL){
00031     LOG(LOG_ERR, "Error opening file %s", netascii_filename);
00032     return 2;
00033   }
00034
00035   prev = EOF;
00036
00037   while ((tmp = (char) fgetc(unixf)) != EOF){
00038     if (tmp == '\n' && (prev == EOF || prev != '\r')){ // LF -> CRLF
00039       ret = putc('\r', netasciif);
00040       if (ret == EOF)
00041         break;
00042
00043       ret = putc('\n', netasciif);
00044       if (ret == EOF)
00045         break;
00046
00047     } else if (tmp == '\r'){  // CR -> CRNUL
00048       char next = (char) fgetc(unixf);
00049       if (next != '\0')
00050         ungetc(next, unixf);
00051
00052       ret = putc('\r', netasciif);
00053       if (ret == EOF)
00054         break;
00055
00056       ret = putc('\0', netasciif);
00057       if (ret == EOF)
00058         break;
00059
00060     } else if (tmp == '\0'){
00061       ret = putc('\0', netasciif);
00062       break;
00063     } else{
00064       ret = putc(tmp, netasciif);
00065       if (ret == EOF)
00066         break;
00067     }
00068
00069     prev = tmp;
00070   }
00071
00072   // Error writing to netasciif
00073   if (ret == EOF){
00074     LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
00075     result = 3;
00076   } else{
00077     LOG(LOG_INFO, "Unix file %s converted to netascii file %s", unix_filename, netascii_filename);
00078     result = 0;
00079   }
00080
00081   fclose(unixf);
00082   fclose(netasciif);
```

```

  return result;
}

int netascii2unix(char* netascii_filename, char *unix_filename){
  FILE *unixf, *netasciif;
  char tmp;
  int ret;
  int result = 0;

  unixf = fopen(unix_filename, "w");

  if (unixf == NULL){
    LOG(LOG_ERR, "Error opening file %s", unix_filename);
    return 1;
  }

  netasciif = fopen(netascii_filename, "r");

  if (unixf == NULL){
    LOG(LOG_ERR, "Error opening file %s", netascii_filename);
    return 2;
  }

  while ((tmp = (char) fgetc(netasciif)) != EOF){
    if (tmp == '\r'){  // CRLF -> LF ; CRNUL -> CR
      char next = (char) fgetc(netasciif);
      if (next == '\0'){  // CRNUL -> CR
       ret = putc('\r', unixf);
       if (ret == EOF)
        break;
      } else if (next == '\n'){  // CRLF -> LF
       ret = putc('\n', unixf);
       if (ret == EOF)
        break;
      } else if (next == EOF) { // bad format
        LOG(LOG_ERR, "Bad formatted netascii: unexpected EOF after CR");
        result = 4;
        break;
      } else{                    // bad format
        LOG(LOG_ERR, "Bad formatted netascii: unexpected %x after CR", next);
        result = 5;
        break;
      }
    } else{

      // nothing else needs to be done!

      ret = putc(tmp, unixf);
      if (ret == EOF)
        break;
    }
  }

  if (result == 0){
    // Error writing to unixf
    if (ret == EOF){
      LOG(LOG_ERR, "Error writing to file %s", unix_filename);
      result = 3;
    } else{
      LOG(LOG_INFO, "Netascii file %s converted to Unix file %s", netascii_filename, unix_filename);
      result = 0;
    }
  } // otherwise there was an error (4 or 5) and result was already set

  fclose(unixf);
  fclose(netasciif);

  return result;
}
```

4.17 netascii.h File Reference

Conversion functions from netascii to Unix standard ASCII.

Functions

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

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

  Netascii to Unix conversion.

4.17.1 Detailed Description

Conversion functions from netascii to Unix standard ASCII.

Author

Riccardo Mancini

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

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

See also

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

Definition in file netascii.h.

4.17.2 Function Documentation

4.17.2.1 netascii2unix()

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

Netascii to Unix conversion.

Parameters

netascii_filename	the filename of the input netascii file
unix_filename	the filename of the output Unix file

Returns

- 0 in case of success

Definition at line 87 of file netascii.c.

4.17.2.2 unix2netascii()

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

Unix to netascii conversion.

Parameters

unix_filename	the filename of the input Unix file
netascii_filename	the filename of the output netascii file

Returns

- 0 in case of success

Definition at line 16 of file netascii.c.

4.18 netascii.h

```

#ifndef NETASCII
#define NETASCII


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

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


#endif
```

4.19 tftp.c File Reference

Implementation of tftp.h.

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

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.

Functions

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

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

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

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

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

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

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

  Handle the entire workflow required to send a file.

4.19.1 Detailed Description

Implementation of tftp.h.

Author

Riccardo Mancini

See also

tftp.h

Definition in file tftp.c.

4.19.2 Macro Definition Documentation

4.19.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file tftp.c.

4.19.3 Function Documentation

4.19.3.1 tftp_receive_ack()

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

Receive an ACK message.

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

Parameters

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 195 of file tftp.c.

4.19.3.2 tftp_receive_file()

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

Handle the entire workflow required to receive a file.

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

Parameters

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.
- 9 in case of message from unexpected source

Definition at line 102 of file tftp.c.

4.19.3.3 tftp_send_ack()

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

Send an ACK message.

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

Parameters

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 85 of file tftp.c.

4.19.3.4 tftp_send_error()

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

Send an ERROR message to the client (server).

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

Parameters

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 65 of file tftp.c.

4.19.3.5 tftp_send_file()

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

Handle the entire workflow required to send a file.

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

Parameters

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 an error while unpacking data.

Definition at line 227 of file tftp.c.

4.19.3.6 tftp_send_rrq()

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

Send a RRQ message to a server.

Parameters

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 25 of file tftp.c.

4.19.3.7 tftp_send_wrq()

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

Send a WRQ message to a server.

Do not used in current implementation.

Parameters

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 45 of file tftp.c.

4.20 tftp.c

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

```
#include <netinet/in.h>
#include <stdlib.h>
#include "include/logging.h"


int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
  int msglen, len;
  char *out_buffer;

  msglen = tftp_msg_get_size_rrq(filename, mode);
  out_buffer = malloc(msglen);

  tftp_msg_build_rrq(filename, mode, out_buffer);
  len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
  if (len != msglen){
    LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
    perror("Error");
    return 1;
  }

  free(out_buffer);
  return 0;
}


int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
  int msglen, len;
  char *out_buffer;

  msglen = tftp_msg_get_size_wrq(filename, mode);
  out_buffer = malloc(msglen);

  tftp_msg_build_wrq(filename, mode, out_buffer);
  len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
  if (len != msglen){
    LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
    perror("Error");
    return 1;
  }

  free(out_buffer);
  return 0;
}


int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in *addr){
  int msglen, len;
  char *out_buffer;

  msglen = tftp_msg_get_size_error(error_msg);
  out_buffer = malloc(msglen);

  tftp_msg_build_error(error_code, error_msg, out_buffer);
  len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));
  if (len != msglen){
    LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
    perror("Error");
    return 1;
  }

  free(out_buffer);
  return 0;
}


int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *addr){
  int msglen, len;

  msglen = tftp_msg_get_size_ack();
  tftp_msg_build_ack(block_n, out_buffer);
  len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr));

 if (len != msglen){
    LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);
    perror("Error");
    return 1;
  }

  return 0;
}


int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
  char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[
TFTP_DATA_BLOCK], out_buffer[4];
  int exp_block_n, rcv_block_n;
  int len, data_size, ret, type;
  unsigned int addrlen;
  struct sockaddr_in cl_addr, orig_cl_addr;

  // init expected block number
  exp_block_n = 1;

  addrlen = sizeof(cl_addr);

  do{
    LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
    // TODO: check client == server ?
    len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(
TFTP_DATA_BLOCK), 0, (struct sockaddr*)&cl_addr, &addrlen);
    if (exp_block_n == 1){ // first block -> I need to save servers TID (aka its "original" sockaddr)
      char addr_str[MAX_SOCKADDR_STR_LEN];
      sockaddr_in_to_string(cl_addr, addr_str);

      if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
        LOG(LOG_ERR, "Received message from unexpected source: %s", addr_str);
        return 9;
      } else{
        LOG(LOG_INFO, "Receiving packets from %s", addr_str);
        orig_cl_addr = cl_addr;
      }
    } else{
      if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
        char addr_str[MAX_SOCKADDR_STR_LEN];
        sockaddr_in_to_string(cl_addr, addr_str);
        LOG(LOG_ERR, "Received message from unexpected source: %s", addr_str);
        return 9;
      } else{
        LOG(LOG_DEBUG, "Sender is the same!");
      }
    }

    type = tftp_msg_type(in_buffer);
    if (type == TFTP_TYPE_ERROR){
      int error_code;
      char error_msg[TFTP_MAX_ERROR_LEN];

      ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
      if (ret != 0){
        LOG(LOG_ERR, "Error unpacking error msg");
        return 5;
      }

      if (error_code == 1){
        LOG(LOG_INFO, "File not found");
        return 1;
      } else{
        LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
        return 7;
      }

    } else if (type != TFTP_TYPE_DATA){
      LOG(LOG_ERR, "Received packet of type %d, expecting DATA or ERROR.", type);
      return 8;
    }

    ret = tftp_msg_unpack_data(in_buffer, len, &rcv_block_n, data, &data_size);

    if (ret != 0){
      LOG(LOG_ERR, "Error unpacking data: %d", ret);
      return 4;
    }

    if (rcv_block_n != exp_block_n){
      LOG(LOG_ERR, "Received unexpected block_n: rcv_block_n = %d != %d = exp_block_n", rcv_block_n,
exp_block_n);
      return 3;
    }

    exp_block_n++;

    LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);

    if (data_size != 0){
      if (fblock_write(m_fblock, data, data_size))
        return 6;
    }

    LOG(LOG_DEBUG, "Sending ack");

    if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
      return 2;

  } while(data_size == TFTP_DATA_BLOCK);
  return 0;
}


int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *addr){
  int msglen, len, ret;
  unsigned int addrlen;
  struct sockaddr_in cl_addr;

  msglen = tftp_msg_get_size_ack();
  addrlen = sizeof(cl_addr);

  len = recvfrom(sd, in_buffer, msglen, 0, (struct sockaddr*)&cl_addr, &addrlen);

  if (sockaddr_in_cmp(*addr, cl_addr) != 0){
    char str_addr[MAX_SOCKADDR_STR_LEN];
    sockaddr_in_to_string(cl_addr, str_addr);
    LOG(LOG_ERR, "Message is coming from unexpected source: %s", str_addr);
    return 2;
  }

  if (len != msglen){
    LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
    return 1;
  }

  ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
  if (ret != 0){
    LOG(LOG_ERR, "Error unpacking ack: %d", ret);
    return 8+ret;
  }

  return 0;
}


int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
  char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[
TFTP_MAX_DATA_MSG_SIZE];
  int block_n, rcv_block_n;
  int len, data_size, msglen;

  // init sequence number
  block_n = 1;

  do{
    LOG(LOG_DEBUG, "Sending part %d", block_n);

    if (m_fblock->remaining > TFTP_DATA_BLOCK)
      data_size = TFTP_DATA_BLOCK;
    else
      data_size = m_fblock->remaining;

    if (data_size != 0)
      fblock_read(m_fblock, data);

    LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);

    msglen = tftp_msg_get_size_data(data_size);
    tftp_msg_build_data(block_n, data, data_size, out_buffer);

    // dump_buffer_hex(out_buffer, msglen);

    len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*)addr, sizeof(*addr));

    if (len != msglen){
      return 1;
    }

    LOG(LOG_DEBUG, "Waiting for ack");

    if (tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr)){
      LOG(LOG_ERR, "Error receiving ack");
      return 2;
    }

    if (rcv_block_n != block_n){
      LOG(LOG_ERR, "Received wrong block n: received %d != expected %d", rcv_block_n, block_n);
      return 3;
    }

    block_n++;

  } while(data_size == TFTP_DATA_BLOCK);
  return 0;
}
```

4.21 tftp.h File Reference

Common functions for TFTP client and server.

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

Functions

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

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

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

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

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

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

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

  Handle the entire workflow required to send a file.

4.21.1 Detailed Description

Common functions for TFTP client and server.

Author

Riccardo Mancini

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

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

Definition in file tftp.h.

4.21.2 Function Documentation

4.21.2.1 tftp_receive_ack()

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

Receive an ACK message.

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

Parameters

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 195 of file tftp.c.

4.21.2.2 tftp_receive_file()

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

Handle the entire workflow required to receive a file.

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

Parameters

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.
- 9 in case of message from unexpected source

Definition at line 102 of file tftp.c.

4.21.2.3 tftp_send_ack()

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

Send an ACK message.

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

Parameters

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 85 of file tftp.c.

4.21.2.4 tftp_send_error()

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

Send an ERROR message to the client (server).

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

Parameters

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 65 of file tftp.c.

4.21.2.5 tftp_send_file()

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

Handle the entire workflow required to send a file.

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

Parameters

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 an error while unpacking data.

Definition at line 227 of file tftp.c.

4.21.2.6 tftp_send_rrq()

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

Send a RRQ message to a server.

Parameters

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 25 of file tftp.c.

4.21.2.7 tftp_send_wrq()

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

Send a WRQ message to a server.

Do not used in current implementation.

Parameters

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 45 of file tftp.c.

4.22 tftp.h

```

#ifndef TFTP
#define TFTP


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

```

```

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

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

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

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

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

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

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


#endif
```

4.23 tftp_client.c File Reference

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

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

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.
- #define READ_BUFFER_SIZE 80

  max stdin line length
- #define MAX_ARGS 3

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

  String for txt.
- #define MODE_BIN "bin"

  String for bin.

Functions

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

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

  Prints command usage information.
- void cmd_help ()

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

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

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

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

  Main.

Variables

- char ∗ transfer_mode

  Global transfer_mode variable for storing user chosen transfer mode string.

4.23.1 Detailed Description

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

Author

Riccardo Mancini

Definition in file tftp_client.c.

4.23.2 Macro Definition Documentation

4.23.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 9 of file tftp_client.c.

4.23.3 Function Documentation

4.23.3.1 cmd_mode()

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

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

See also

transfer_mode

Definition at line 119 of file tftp_client.c.

4.23.3.2 split_string()

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

Splits a string at each delim.

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

Parameters

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 62 of file tftp_client.c.

4.23.4 Variable Documentation

4.23.4.1 transfer_mode

```
char* transfer_mode
```

Global transfer_mode variable for storing user chosen transfer mode string.

See also

MODE_TXT
MODE_BIN

Definition at line 48 of file tftp_client.c.

4.24 tftp_client.c

```c

#define LOG_LEVEL LOG_INFO


#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>


#define READ_BUFFER_SIZE 80

#define MAX_ARGS 3

#define MODE_TXT "txt"

#define MODE_BIN "bin"


char* transfer_mode;


void split_string(char* line, char* delim, int max_argc, int *argc, char **argv){
  char *ptr;
  int len;
  char *pos;

  // remove trailing LF
  if ((pos=strchr(line, '\n')) != NULL)
    *pos = '\0';

  // init argc
  *argc = 0;

  // tokenize string
  ptr = strtok(line, delim);

  while(ptr != NULL && *argc <= max_argc){
    len = strlen(ptr);

    if (len == 0)
      continue;

    LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);

    argv[*argc] = malloc(strlen(ptr)+1);
    strcpy(argv[*argc], ptr);

    ptr = strtok(NULL, delim);
    (*argc)++;
  }
}

void print_help(){
  printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
  printf("Example: ./tftp_client 127.0.0.1 69");
}

void cmd_help(){
  printf("Sono disponibili i seguenti comandi:\n");
  printf("!help --> mostra l'elenco dei comandi disponibili\n");
  printf("!mode {txt|bin} --> imposta il modo di trasferimento dei file (testo o binario)\n");
  printf("!get filename nome_locale --> richiede al server il nome del file <filename> e lo salva localmente con il nome <nome_locale>\n");
  printf("!quit --> termina il client\n");
}

void cmd_mode(char* new_mode){
  if (strcmp(new_mode, MODE_TXT) == 0){
    transfer_mode = TFTP_STR_NETASCII;
    printf("Modo di trasferimento testo configurato\n");
  } else if (strcmp(new_mode, MODE_BIN) == 0){
    transfer_mode = TFTP_STR_OCTET;
    printf("Modo di trasferimento binario configurato\n");
  } else{
    printf("Modo di traferimento sconosciuto: %s. Modi disponibili: txt, bin\n", new_mode);
  }
}

int cmd_get(char* remote_filename, char* local_filename, char* sv_ip, int sv_port){
  struct sockaddr_in my_addr, sv_addr;
  int sd;
  int ret, tid, result;
  struct fblock m_fblock;
  char *tmp_filename;

  LOG(LOG_INFO, "Initializing...\n");

  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
    m_fblock = fblock_open(local_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_MODE_BINARY);
  else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
    tmp_filename = malloc(strlen(local_filename)+5);
    strcpy(tmp_filename, local_filename);
    strcat(tmp_filename, ".tmp");
    m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_MODE_TEXT);
  }else
    return 2;

  LOG(LOG_INFO, "Opening socket...");

  sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
  my_addr = make_my_sockaddr_in(0);
  tid = bind_random_port(sd, &my_addr);
  if (tid == 0){
    LOG(LOG_ERR, "Error while binding to random port");
    perror("Could not bind to random port:");
    fblock_close(&m_fblock);
    return 1;
  } else
    LOG(LOG_INFO, "Bound to port %d", tid);

  printf("Richiesta file %s (%s) al server in corso.\n", remote_filename, transfer_mode);

  ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
  if (ret != 0){
    fblock_close(&m_fblock);
    return 8+ret;
  }

  printf("Trasferimento file in corso.\n");

  ret = tftp_receive_file(&m_fblock, sd, &sv_addr);


  if (ret == 1){    // File not found
    printf("File non trovato.\n");
    result = 0;
  } else if (ret != 0){
    LOG(LOG_ERR, "Error while receiving file!");
    result = 16+ret;
  } else{
    int n_blocks = (m_fblock.written + m_fblock.block_size - 1)/m_fblock.block_size;
    printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);
    printf("Salvataggio %s completato.\n", local_filename);

    result = 0;
  }

  fblock_close(&m_fblock);
  if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
    netascii2unix(tmp_filename, local_filename);
    remove(tmp_filename);
    free(tmp_filename);
  }

  return result;

}

void cmd_quit(){
  printf("Client terminato con successo\n");
  exit(0);
}

int main(int argc, char** argv){
  char* sv_ip;
  short int sv_port;
  int ret;
  char read_buffer[READ_BUFFER_SIZE];
  int cmd_argc;
  char *cmd_argv[MAX_ARGS];

  //init random seed
  srand(time(NULL));

  // default mode = bin
  transfer_mode = TFTP_STR_OCTET;

  if (argc != 3){
    print_help();
    return 1;
  }

  // TODO: check args
  sv_ip = argv[1];
  sv_port = atoi(argv[2]);

  while(1){
    printf("> ");
    fflush(stdout); // flush stdout buffer
    fgets(read_buffer, READ_BUFFER_SIZE, stdin);
    split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);

    if (cmd_argc == 0){
      printf("Comando non riconosciuto : ''\n");
      cmd_help();
    } else{
      if (strcmp(cmd_argv[0], "!mode") == 0){
        if (cmd_argc == 2)
          cmd_mode(cmd_argv[1]);
        else
          printf("Il comando richiede un solo argomento: bin o txt\n");
      } else if (strcmp(cmd_argv[0], "!get") == 0){
        if (cmd_argc == 3){
          ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
          LOG(LOG_DEBUG, "cmd_get returned value: %d", ret);
        } else{
           printf("Il comando richiede due argomenti: <filename> e <nome_locale>\n");
        }
      } else if (strcmp(cmd_argv[0], "!quit") == 0){
        if (cmd_argc == 1){
          cmd_quit();
        } else{
           printf("Il comando non richiede argomenti\n");
        }
      } else if (strcmp(cmd_argv[0], "!help") == 0){
        if (cmd_argc == 1){
          cmd_help();
        } else{
           printf("Il comando non richiede argomenti\n");
        }
      } else {
        printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
        cmd_help();
      }
    }

    // Free malloc'ed strings
    for (int i = 0; i < cmd_argc; i++)
      free(cmd_argv[i]);
  }

  return 0;
}
```

4.25 tftp_msgs.c File Reference

Implementation of tftp_msgs.h .

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

```
#include <arpa/inet.h>
#include <stdint.h>
```

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.

Functions

- int tftp_msg_type (char ∗buffer)

  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.

4.25.1 Detailed Description

Implementation of tftp_msgs.h .

Author

Riccardo Mancini

See also

tftp_msgs.h

Definition in file tftp_msgs.c.

4.25.2 Macro Definition Documentation

4.25.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file tftp_msgs.c.

4.25.3 Function Documentation

4.25.3.1 tftp_msg_build_ack()

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

Builds an acknowledgment message.

Message format:

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

Parameters

block←↩ _n	block sequence number
buffer	data buffer where to build the message

Definition at line 163 of file tftp_msgs.c.

4.25.3.2 tftp_msg_build_data()

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

Builds a data message.

Message format:

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

Parameters

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 130 of file tftp_msgs.c.

4.25.3.3 tftp_msg_build_error()

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

Builds an error message.

Message format:

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

Error code meaning:

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

In current implementation only errors 1 and 4 are implemented.

Parameters

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

Definition at line 190 of file tftp_msgs.c.

4.25.3.4 tftp_msg_build_rrq()

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

Builds a read request message.

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

Parameters

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.

4.25.3.5 tftp_msg_build_wrq()

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

Builds a write request message.

Message format:

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

Parameters

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

Definition at line 79 of file tftp_msgs.c.

4.25.3.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

Parameters

data_size	data buffer size

Returns

size in bytes

Definition at line 185 of file tftp_msgs.c.

4.25.3.7 tftp_msg_get_size_data()

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

Returns size in bytes of a data message.

It just sums 4 to data_size.

Parameters

data_size	data buffer size

Returns

size in bytes

Definition at line 158 of file tftp_msgs.c.

4.25.3.8 tftp_msg_get_size_error()

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

Returns size in bytes of an error message.

Parameters

error_msg	error message

Returns

size in bytes

Definition at line 226 of file tftp_msgs.c.

4.25.3.9 tftp_msg_get_size_rrq()

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

Returns size in bytes of a read request message.

Parameters

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

Returns

size in bytes

Definition at line 74 of file tftp_msgs.c.

4.25.3.10 tftp_msg_get_size_wrq()

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

Returns size in bytes of a write request message.

Parameters

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

Returns

size in bytes

Definition at line 125 of file tftp_msgs.c.

4.25.3.11 tftp_msg_type()

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

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.

4.25.3.12 tftp_msg_unpack_ack()

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

Unpacks an acknowledgment message.

Parameters

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 170 of file tftp_msgs.c.

4.25.3.13 tftp_msg_unpack_data()

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

Unpacks a data message.

Parameters

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 139 of file tftp_msgs.c.

4.25.3.14 tftp_msg_unpack_error()

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

Unpacks an error message.

Parameters

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 199 of file tftp_msgs.c.

4.25.3.15 tftp_msg_unpack_rrq()

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

Unpacks a read request message.

Unpacks a write request message.

Parameters

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 39 of file tftp_msgs.c.

4.26 tftp_msgs.c

```

#define LOG_LEVEL LOG_INFO


#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>


int tftp_msg_type(char *buffer){
  return (((int)buffer[0]) << 8) + buffer[1];
}
```

```
void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
  buffer[0] = 0;
  buffer[1] = 1;
  buffer += 2;
  strcpy(buffer, filename);
  buffer += strlen(filename)+1;
  strcpy(buffer, mode);
}


int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char*
mode){
  int offset = 0;
  if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
    LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
    return 1;
  }

  offset += 2;
  if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
    LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
    return 3;
  }
  strcpy(filename, buffer+offset);

  offset += strlen(filename)+1;
  if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
    LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
    return 4;
  }
  strcpy(mode, buffer+offset);

  offset += strlen(mode)+1;
  if (buffer_len != offset){
    LOG(LOG_ERR, "Packet contains unexpected fields");
    return 2;
  }
  if (strcasecmp(mode, TFTP_STR_NETASCII) == 0 || strcasecmp(
mode, TFTP_STR_OCTET) == 0)
    return 0;
  else{
    LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
    return 5;
  }
}


int tftp_msg_get_size_rrq(char* filename, char* mode){
  return 4 + strlen(filename) + strlen(mode);
}


void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
  buffer[0] = 0;
  buffer[1] = 2;
  buffer += 2;
  strcpy(buffer, filename);
  buffer += strlen(filename)+1;
  strcpy(buffer, mode);
}


int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename, char* mode){
  int offset = 0;
  if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
    LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
    return 1;
  }

  offset += 2;
  if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
    LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
    return 3;
  }

  strcpy(filename, buffer+offset);
  offset += strlen(filename)+1;
  if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
    LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
    return 4;
  }

  strcpy(mode, buffer+offset);
  offset += strlen(mode)+1;
  if (buffer_len != offset){
    LOG(LOG_ERR, "Packet contains unexpected fields");
    return 2;
  }

  if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode, TFTP_STR_OCTET) == 0)
    return 0;
  else{
    LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
    return 5;
  }
}


int tftp_msg_get_size_wrq(char* filename, char* mode){
  return 4 + strlen(filename) + strlen(mode);
}


void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
  buffer[0] = 0;
  buffer[1] = 3;
  *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
  buffer += 4;
  memcpy(buffer, data, data_size);
}


int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data, int* data_size){
  if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
    LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
    return 1;
  }

  if (buffer_len < 4){
    LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
    return 2;
  }

  *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
  *data_size = buffer_len - 4;
  if (*data_size > 0)
    memcpy(data, buffer+4, *data_size);
  return 0;
}


int tftp_msg_get_size_data(int data_size){
  return data_size + 4;
}


void tftp_msg_build_ack(int block_n, char* buffer){
  buffer[0] = 0;
  buffer[1] = 4;
  *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
}


int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
  if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
    LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
    return 1;
  }

  if (buffer_len != 4){
    LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
    return 2;
  }
  *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
  return 0;
}


int tftp_msg_get_size_ack(){
  return 4;
}


void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
  buffer[0] = 0;
  buffer[1] = 5;
  *((uint16_t*)(buffer+2)) = htons((uint16_t) error_code);
  buffer += 4;
  strcpy(buffer, error_msg);
}


int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char*
      error_msg){
  if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
      LOG(LOG_ERR, "Expected ERROR message (5), found %d", tftp_msg_type(buffer));
      return 1;
    }

    *error_code = (int) ntohs(*((uint16_t*)(buffer+2)));
    if (*error_code < 0 || *error_code > 7){
      LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
      return 4;
    }

    buffer += 4;
    if(strlen(buffer) > TFTP_MAX_ERROR_LEN){
      LOG(LOG_ERR, "Error string too long (%d > %d): %s", (int) strlen(buffer),
      TFTP_MAX_ERROR_LEN, buffer);
      return 3;
    }

    strcpy(error_msg, buffer);
    if (buffer_len != strlen(error_msg)+5){
      LOG(LOG_WARN, "Packet contains unexpected fields");
      return 2;
    }
    return 0;
}


int tftp_msg_get_size_error(char* error_msg){
  return 5 + strlen(error_msg);
}
```

4.27 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.

4.27.1 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.

4.27.2 Macro Definition Documentation

4.27.2.1 TFTP_MAX_MODE_LEN

```
#define TFTP_MAX_MODE_LEN 8
```

Maximum mode field string length.

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

Definition at line 50 of file tftp_msgs.h.

4.27.3 Function Documentation

4.27.3.1 tftp_msg_build_ack()

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

Builds an acknowledgment message.

Message format:

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

Parameters

block←↩ _n	block sequence number
buffer	data buffer where to build the message

Definition at line 163 of file tftp_msgs.c.

4.27.3.2 tftp_msg_build_data()

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

Builds a data message.

Message format:

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

Parameters

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 130 of file tftp_msgs.c.

4.27.3.3 tftp_msg_build_error()

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

Builds an error message.

Message format:

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

Error code meaning:

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

In current implementation only errors 1 and 4 are implemented.

Parameters

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

Definition at line 190 of file tftp_msgs.c.

4.27.3.4 tftp_msg_build_rrq()

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

Builds a read request message.

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

Parameters

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.

4.27.3.5 tftp_msg_build_wrq()

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

Builds a write request message.

Message format:

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

Parameters

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

Definition at line 79 of file tftp_msgs.c.

4.27.3.6 tftp_msg_get_size_ack()

```
int tftp_msg_get_size_ack ( )
```

Returns size in bytes of an acknowledgment message.

It just returns 4.

Parameters

data_size	data buffer size

Returns

size in bytes

Definition at line 185 of file tftp_msgs.c.

4.27.3.7 tftp_msg_get_size_data()

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

Returns size in bytes of a data message.

It just sums 4 to data_size.

Parameters

data_size	data buffer size

Returns

size in bytes

Definition at line 158 of file tftp_msgs.c.

4.27.3.8 tftp_msg_get_size_error()

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

Returns size in bytes of an error message.

Parameters

error_msg	error message

Returns

size in bytes

Definition at line 226 of file tftp_msgs.c.

4.27.3.9 tftp_msg_get_size_rrq()

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

Returns size in bytes of a read request message.

Parameters

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

Returns

size in bytes

Definition at line 74 of file tftp_msgs.c.

4.27.3.10 tftp_msg_get_size_wrq()

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

Returns size in bytes of a write request message.

Parameters

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

Returns

size in bytes

Definition at line 125 of file tftp_msgs.c.

4.27.3.11 tftp_msg_type()

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

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.

4.27.3.12 tftp_msg_unpack_ack()

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

Unpacks an acknowledgment message.

Parameters

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 170 of file tftp_msgs.c.

4.27.3.13 tftp_msg_unpack_data()

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

Unpacks a data message.

Parameters

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 139 of file tftp_msgs.c.

4.27.3.14 tftp_msg_unpack_error()

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

Unpacks an error message.

Parameters

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 199 of file tftp_msgs.c.

4.27.3.15 tftp_msg_unpack_rrq()

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

Unpacks a read request message.

Unpacks a write request message.

Parameters

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 39 of file tftp_msgs.c.

4.28 tftp_msgs.h

```

#ifndef TFTP_MSGS
#define TFTP_MSGS


#define TFTP_TYPE_RRQ    1
```

```

#define TFTP_TYPE_WRQ   2

#define TFTP_TYPE_DATA  3

#define TFTP_TYPE_ACK   4

#define TFTP_TYPE_ERROR 5

#define TFTP_STR_NETASCII "netascii"

#define TFTP_STR_OCTET "octet"

#define TFTP_MAX_FILENAME_LEN 255

#define TFTP_MAX_MODE_LEN 8

#define TFTP_MAX_ERROR_LEN 255

#define TFTP_DATA_BLOCK 512

#define TFTP_MAX_DATA_MSG_SIZE 516


int tftp_msg_type(char *buffer);


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

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

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

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

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

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

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

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

int tftp_msg_get_size_data(int data_size);

void tftp_msg_build_ack(int block_n, char* buffer);

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

int tftp_msg_get_size_ack();

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

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

int tftp_msg_get_size_error(char* error_msg);


#endif
```

4.29 tftp_server.c File Reference

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

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

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

Macros

- #define LOG_LEVEL LOG_INFO

  Defines log level to this file.
- #define **_GNU_SOURCE**
- #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4

  Maximum length for a RRQ message.

Functions

- int 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.

4.29.1 Detailed Description

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

Author

Riccardo Mancini

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

Definition in file tftp_server.c.

4.29.2 Macro Definition Documentation

4.29.2.1 LOG_LEVEL

```
#define LOG_LEVEL LOG_INFO
```

Defines log level to this file.

Definition at line 12 of file tftp_server.c.

4.29.3 Function Documentation

4.29.3.1 path_inside_dir()

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

Check whether file is inside dir.

Parameters

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 58 of file tftp_server.c.

4.30 tftp_server.c

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

```
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <linux/limits.h>
#include <libgen.h>


#define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4


int strlcpl(const char* str1, const char* str2){
  int n;
  for (n = 0; str1[n] != '\0' && str2[n] != '\0' && str1[n] == str2[n]; n++);
  return n;
}

int path_inside_dir(char* path, char* dir){
  char *parent, *orig_parent, *ret_realpath;
  char parent_realpath[PATH_MAX];
  int result;

  orig_parent = parent = malloc(strlen(path) + 1);
  strcpy(parent, path);

  do{
    parent = dirname(parent);
    ret_realpath = realpath(parent, parent_realpath);
  } while (ret_realpath == NULL);

  if (strlcpl(parent_realpath, dir) < strlen(dir))
    result = 0;
  else
    result = 1;

  free(orig_parent);
  return result;
}

void print_help(){
  printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
  printf("Example: ./tftp_server 69 .\n");
}

int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
  struct sockaddr_in my_addr;
  int sd;
  int ret, tid, result;
  struct fblock m_fblock;
  char *tmp_filename;

  sd = socket(AF_INET, SOCK_DGRAM, 0);
  my_addr = make_my_sockaddr_in(0);
  tid = bind_random_port(sd, &my_addr);
  if (tid == 0){
    LOG(LOG_ERR, "Could not bind to random port");
    perror("Could not bind to random port:");
    fblock_close(&m_fblock);
    return 4;
  } else
    LOG(LOG_INFO, "Bound to port %d", tid);

  if (strcasecmp(mode, TFTP_STR_OCTET) == 0){
    m_fblock = fblock_open(filename, TFTP_DATA_BLOCK,
FBLOCK_READ|FBLOCK_MODE_BINARY);
  } else if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
    tmp_filename = malloc(strlen(filename)+5);
    strcpy(tmp_filename, filename);
    strcat(tmp_filename, ".tmp");
    ret = unix2netascii(filename, tmp_filename);
    if (ret != 0){
      LOG(LOG_ERR, "Error converting text file to netascii: %d", ret);
      return 3;
    }
    m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK,
FBLOCK_READ|FBLOCK_MODE_TEXT);
  } else{
    LOG(LOG_ERR, "Unknown mode: %s", mode);
    return 2;
  }

  if (m_fblock.file == NULL){
    LOG(LOG_WARN, "Error opening file. Not found?");
    tftp_send_error(1, "File not found.", sd, cl_addr);
    result = 1;
  } else{
    LOG(LOG_INFO, "Sending file...");
    ret = tftp_send_file(&m_fblock, sd, cl_addr);
```

```c

    if (ret != 0){
      LOG(LOG_ERR, "Error sending file: %d", ret);
      result = 16+ret;
    } else{
      LOG(LOG_INFO, "File sent successfully");
      result = 0;
    }
  }

  fblock_close(&m_fblock);

  if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
    LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
    remove(tmp_filename);
    free(tmp_filename);
  }

  return result;
}

int main(int argc, char** argv){
  short int my_port;
  char *dir_rel_path;
  char *ret_realpath;
  char dir_realpath[PATH_MAX];
  int ret, type, len;
  char in_buffer[MAX_MSG_LEN];
  unsigned int addrlen;
  int sd;
  struct sockaddr_in my_addr, cl_addr;
  int pid;
  char addr_str[MAX_SOCKADDR_STR_LEN];

  if (argc != 3){
    print_help();
    return 1;
  }

  my_port = atoi(argv[1]);
  dir_rel_path = argv[2];

  ret_realpath = realpath(dir_rel_path, dir_realpath);
  if (ret_realpath == NULL){
    LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
    return 1;
  }

  addrlen = sizeof(cl_addr);

  sd = socket(AF_INET, SOCK_DGRAM, 0);
  my_addr = make_my_sockaddr_in(my_port);
  ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
  if (ret == -1){
    perror("Could not bind: ");
    LOG(LOG_FATAL, "Could not bind to port %d", my_port);
    return 1;
  }

  LOG(LOG_INFO, "Server is running");

  while (1){
    len = recvfrom(sd, in_buffer, MAX_MSG_LEN, 0, (struct sockaddr*)&cl_addr, &addrlen);
    type = tftp_msg_type(in_buffer);
    sockaddr_in_to_string(cl_addr, addr_str);
    LOG(LOG_INFO, "Received message with type %d from %s", type, addr_str);
    if (type == TFTP_TYPE_RRQ){
      pid = fork();
      if (pid != 0){  // father
        LOG(LOG_INFO, "Received RRQ, spawned new process %d", (int) pid);
        continue; // father process continues loop
      } else{         // child
        char filename[TFTP_MAX_FILENAME_LEN], mode[
TFTP_MAX_MODE_LEN];
        char file_path[PATH_MAX], file_realpath[PATH_MAX];

        //init random seed
        srand(time(NULL));

        ret = tftp_msg_unpack_rrq(in_buffer, len, filename, mode);

        if (ret != 0){
          LOG(LOG_WARN, "Error unpacking RRQ");
          tftp_send_error(0, "Malformed RRQ packet.", sd, &cl_addr);
          break; // child process exits loop
        }

        strcpy(file_path, dir_realpath);
        strcat(file_path, "/");
        strcat(file_path, filename);

        // check if file is inside directory (or inside any of its subdirectories)
        if (!path_inside_dir(file_path, dir_realpath)){
          // it is not! I caught you, Trudy!
          LOG(LOG_WARN, "User tried to access file %s outside set directory %s",
              file_realpath,
              dir_realpath
          );

          tftp_send_error(4, "Access violation.", sd, &cl_addr);
          break; // child process exits loop
        }

        ret_realpath = realpath(file_path, file_realpath);

        // file not found
        if (ret_realpath == NULL){
          LOG(LOG_WARN, "File not found: %s", file_path);
          tftp_send_error(1, "File Not Found.", sd, &cl_addr);
          break; // child process exits loop
        }

        LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);

        ret = send_file(file_realpath, mode, &cl_addr);
        if (ret != 0)
          LOG(LOG_WARN, "Write terminated with an error: %d", ret);
        break;  // child process exits loop
      }
    } else{
      LOG(LOG_WARN, "Wrong op code: %d", type);
      tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
      // main process continues loop
    }
  }

  LOG(LOG_INFO, "Exiting process %d", (int) getpid());
  return 0;
}
```

Index