```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief File block read and write.
 *
 * This library provides functions for reading and writing a text or binary
 * file using a predefined block size.
 */

#ifndef FBLOCK
#define FBLOCK


#include <stdio.h>


/** Mask for getting text/binary mode */
#define FBLOCK_MODE_MASK   0b01

/** Open file in text mode */
#define FBLOCK_MODE_TEXT   0b00

/** Open file in binary mode */
#define FBLOCK_MODE_BINARY 0b01

/** Mask for getting r/w mode */
#define FBLOCK_RW_MASK     0b10

/** Open file in read mode */
#define FBLOCK_READ        0b00

/** Open file in write mode */
#define FBLOCK_WRITE       0b10


/**
 * Structure which defines a file.
 */
struct fblock{
  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  */
  };
};


/**
 * Opens a file.
 *
 * @param filename    name of the file
 * @param block_size  size of the blocks
 * @param mode        mode (read, write, text, binary)
 * @return            fblock structure
 *
 * @see FBLOCK_MODE_TEXT
 * @see FBLOCK_MODE_BINARY
 * @see FBLOCK_WRITE
 * @see FBLOCK_READ
 */
```

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

/**
 * Reads next block_size bytes from file.
 *
 * @param m_fblock    fblock instance
 * @param buffer      block_size bytes buffer
 * @return            0 in case of success, otherwise number of bytes it could not read
 */
int fblock_read(struct fblock *m_fblock, char* buffer);

/**
 * Writes next block_size bytes to file.
 *
 * @param m_fblock    fblock instance
 * @param buffer      block_size bytes buffer
 * @param block_size  if set to a non-0 value, override block_size defined in fblock.
 * @return            0 in case of success, otherwise number of bytes it could not write
 */
int fblock_write(struct fblock *m_fblock, char* buffer, int block_size);

/**
 * Closes a file.
 *
 * @param m_fblock    fblock instance to be closed
 * @return            0 in case of success, EOF in case of failure
 *
 * @see fclose
 */
int fblock_close(struct fblock *m_fblock);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of fblock.h.
 *
 * @see fblock.h
 */


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


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


/**
 * Returns file length
 *
 * @param f  file pointer
 * @return   file length in bytes
 */
int get_length(FILE *f){
  int size;
  fseek(f, 0, SEEK_END); // seek to end of file
  size = ftell(f); // get current file pointer
  fseek(f, 0, SEEK_SET); // seek back to beginning of file
  return size;
}


struct fblock fblock_open(char* filename, int block_size, char mode){
  struct fblock m_fblock;
  m_fblock.block_size = block_size;
  m_fblock.mode = mode;

  char mode_str[4] = "";

  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);
}
```

```
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Contructor for TFTP messages.
 *
 * 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)
 */

#ifndef TFTP_MSGS
#define TFTP_MSGS


/** Read request message type */
#define TFTP_TYPE_RRQ   1

/** Write request message type */
#define TFTP_TYPE_WRQ   2

/** Data message type */
#define TFTP_TYPE_DATA  3

/** Acknowledgment message type */
#define TFTP_TYPE_ACK   4

/** Error message type */
#define TFTP_TYPE_ERROR 5

/** String for netascii */
#define TFTP_STR_NETASCII "netascii"

/** String for octet */
#define TFTP_STR_OCTET "octet"

/** Maximum filename length (do not defined in RFC) */
#define TFTP_MAX_FILENAME_LEN 255

/**
 * Maximum mode field string length
 *
 * Since there are only two options: 'netascii' and 'octet', len('netascii') is
 * the TFTP_MAX_MODE_LEN.
 */
#define TFTP_MAX_MODE_LEN 8

/** Maximum error message length (do not defined in RFC) */
#define TFTP_MAX_ERROR_LEN 255

/** Data block size as defined in RFC */
#define TFTP_DATA_BLOCK 512

/** Data message max size is equal to TFTP_DATA_BLOCK + 4 (header) */
#define TFTP_MAX_DATA_MSG_SIZE 516


/**
 * Retuns msg type given a message buffer.
```

```c
 *
 * @param buffer the buffer
 * @return message type
 *
 * @see TFTP_TYPE_RRQ
 * @see TFTP_TYPE_WRQ
 * @see TFTP_TYPE_DATA
 * @see TFTP_TYPE_ACK
 * @see TFTP_TYPE_ERROR
 */
int tftp_msg_type(char *buffer);


/**
 * Builds a read request message.
 *
 * ```
 *  2 bytes    string    1 byte    string    1 byte
 *  -----------------------------------------------
 * |   01  |  Filename  |   0  |     Mode     |   0  |
 *  -----------------------------------------------
 * ```
 *
 * @param filename  name of the file
 * @param mode      requested transfer mode ("netascii" or "octet")
 * @param buffer    data buffer where to build the message
 */
void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);

/**
 * Unpacks a read request message.
 *
 * @param buffer      data buffer where the message to read is [in]
 * @param buffer_len  length of the buffer [in]
 * @param filename    name of the file [out]
 * @param mode        requested transfer mode ("netascii" or "octet") [out]
 * @return
 * - 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 TFTP_TYPE_RRQ
 * @see TFTP_MAX_FILENAME_LEN
 * @see TFTP_MAX_MODE_LEN
 * @see TFTP_STR_NETASCII
 * @see TFTP_STR_OCTET
 */
int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode
);

/**
 * Returns size in bytes of a read request message.
 *
 * @param filename  name of the file
 * @param mode      requested transfer mode ("netascii" or "octet")
 * @return          size in bytes
 */
int tftp_msg_get_size_rrq(char* filename, char* mode);

/**
```

```
 * Builds a write request message.
 *
 * Message format:
 * ```
 *  2 bytes    string    1 byte    string    1 byte
 *  ----------------------------------------------
 * |   02  |  Filename  |   0  |    Mode     |   0  |
 *  ----------------------------------------------
 * ```
 *
 * @param filename  name of the file
 * @param mode      requested transfer mode ("netascii" or "octet")
 * @param buffer    data buffer where to build the message
 */
void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);

/**
 * Unpacks a write request message.
 *
 * @param buffer      data buffer where the message to read is [in]
 * @param buffer_len  length of the buffer [in]
 * @param filename    name of the file [out]
 * @param mode        requested transfer mode ("netascii" or "octet") [out]
 * @return
 * - 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 TFTP_TYPE_WRQ
 * @see TFTP_MAX_FILENAME_LEN
 * @see TFTP_MAX_MODE_LEN
 * @see TFTP_STR_NETASCII
 * @see TFTP_STR_OCTET
 */
int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode
);

/**
 * Returns size in bytes of a write request message.
 *
 * @param filename  name of the file
 * @param mode      requested transfer mode ("netascii" or "octet")
 * @return          size in bytes
 */
int tftp_msg_get_size_wrq(char* filename, char* mode);

/**
 * Builds a data message.
 *
 * Message format:
 * ```
 *  2 bytes    2 bytes       n bytes
 *  ---------------------------------
 * | 03    |    Block #  |    Data    |
 *  ---------------------------------
 * ```
 *
 * @param block_n   block sequence number
 * @param data      pointer to the buffer containing the data to be transfered
 * @param data_size data buffer size
```

```
 * @param buffer    data buffer where to build the message
 */
void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);

/**
 * Unpacks a data message.
 *
 * @param buffer      data buffer where the message to read is [in]
 * @param buffer_len  length of the buffer [in]
 * @param block_n     pointer where block_n will be written [out]
 * @param data        pointer where to copy data [out]
 * @return
 * - 0 in case of success.
 * - 1 in case of wrong operation code.
 * - 2 in case of missing fields (packet size is too small).
 *
 * @see TFTP_TYPE_DATA
 */
int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
int* data_size);

/**
 * Returns size in bytes of a data message.
 *
 * It just sums 4 to data_size.
 *
 * @param data_size data buffer size
 * @return          size in bytes
 */
int tftp_msg_get_size_data(int data_size);

/**
 * Builds an acknowledgment message.
 *
 * Message format:
 * ```
 *  2 bytes    2 bytes
 *  -------------------
 * | 04    |   Block #  |
 *  --------------------
 * ```
 *
 * @param block_n   block sequence number
 * @param buffer    data buffer where to build the message
 */
void tftp_msg_build_ack(int block_n, char* buffer);

/**
 * Unpacks an acknowledgment message.
 *
 * @param buffer     data buffer where the message to read is [in]
 * @param buffer_len length of the buffer [in]
 * @param block_n    pointer where block_n will be written [out]
 * @param data       pointer inside buffer where the data is [out]
 * @return
 * - 0 in case of success.
 * - 1 in case of wrong operation code.
 * - 2 in case of wrong packet size.
 *
 * @see TFTP_TYPE_ACK
 */
int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);

```

```c
/**
 * Returns size in bytes of an acknowledgment message.
 *
 * It just returns 4.
 *
 * @param data_size data buffer size
 * @return          size in bytes
 */
int tftp_msg_get_size_ack();

/**
 * 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.
 *
 * @param error_code error code (from 0 to 7)
 * @param error_msg  error message
 * @param buffer    data buffer where to build the message
 */
void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);

/**
 * Unpacks an error message.
 *
 * @param buffer     data buffer where the message to read is [in]
 * @param buffer_len length of the buffer [in]
 * @param error_code pointer where error_code will be written [out]
 * @param error_msg  pointer to error message inside the message [out]
 * @return
 * - 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 TFTP_TYPE_ERROR
 * @see TFTP_MAX_ERROR_LEN
 */
int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char* e
rror_msg);

/**
 * Returns size in bytes of an error message.
 *
 * @param error_msg  error message
```

```
 * @return           size in bytes
 */
int tftp_msg_get_size_error(char* error_msg);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of tftp_msgs.h .
 *
 * @see 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>


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


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+offse
t), 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);
}
```

```
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Utility funcions for managing inet addresses.
 *
 * 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 sockaddr_in
 * @see rand
 */

#ifndef INET_UTILS
#define INET_UTILS


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

/** Random port will be greater or equal to FROM_PORT */
#define FROM_PORT 49152

/** Random port will be lower or equal to TO_PORT */
#define TO_PORT   65535

/** Maximum number of trials before giving up opening a random port */
#define MAX_TRIES 256

/** Maximum number of characters of INET address to string (eg 123.156.189.123:45678) */
#define MAX_SOCKADDR_STR_LEN 22


/**
 * Binds socket to a random port.
 *
 * @param socket    socket ID
 * @param addr      inet addr structure
 * @return          0 in case of failure, port it could bind to otherwise
 *
 * @see FROM_PORT
 * @see TO_PORT
 * @see MAX_TRIES
 */
int bind_random_port(int socket, struct sockaddr_in *addr);

/**
 * Makes sockaddr_in structure given ip string and port of server.
 *
 * @param ip      ip address of server
 * @param port    port of the server
 * @return        sockaddr_in structure for the given server
 */
struct sockaddr_in make_sv_sockaddr_in(char* ip, int port);

/**
 * Makes sockaddr_in structure of this host.
 *
 * INADDR_ANY is used as IP address.
 *
 * @param port    port of the server
```

```c
 * @return        sockaddr_in structure this host on given port
 */
struct sockaddr_in make_my_sockaddr_in(int port);

/**
 * Compares INET addresses, returning 0 in case they're equal.
 *
 * @param sai1  first address
 * @param sai2  second address
 * @return      0 if thery're equal, 1 otherwise
 */
int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);

/**
 * Converts sockaddr_in structure to string to be printed.
 *
 * @param src   the input address
 * @param dst   the output string (must be at least MAX_SOCKADDR_STR_LEN long)
 */
void sockaddr_in_to_string(struct sockaddr_in src, char *dst);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of inet_utils.h.
 *
 * @see 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"


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


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);
}
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Utility functions for debugging.
 *
 * At the moment, this library implements only one function for dumping a
 * buffer using hexadecimal.
 */

#ifndef DEBUG_UTILS
#define DEBUG_UTILS


/**
 * Prints content of buffer to stdout, showing it as hex values.
 *
 * @param buffer    pointer to the buffer to be printed
 * @param len       the length (in bytes) of the buffer
 */
void dump_buffer_hex(char* buffer, int len);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of debug_utils.h.
 *
 * @see debug_utils.h
 */


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


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


void dump_buffer_hex(char* buffer, int len){
  char *str, tmp[4];
  int i;

  str = malloc(len*3+1);

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

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

```
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Common functions for TFTP client and server.
 *
 * 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.
 */

#ifndef TFTP
#define TFTP


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


/**
 * Send a RRQ message to a server.
 *
 * @param filename  the name of the requested file
 * @param mode      the desired mode of transfer (netascii or octet)
 * @param sd        socket id of the (UDP) socket to be used to send the message
 * @param addr      address of the server
 * @return          0 in case of success, 1 otherwise
 *
 * @see TFTP_STR_NETASCII
 * @see TFTP_STR_OCTET
 */
int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);

/**
 * Send a WRQ message to a server.
 *
 * Do not used in current implementation.
 *
 * @param filename  the name of the requested file
 * @param mode      the desired mode of transfer (netascii or octet)
 * @param sd        socket id of the (UDP) socket to be used to send the message
 * @param addr      address of the server
 * @return          0 in case of success, 1 otherwise
 *
 * @see TFTP_STR_NETASCII
 * @see TFTP_STR_OCTET
 */
int tftp_send_wrq(char* filename, char *mode, 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.
 *
 * @param error_code the code of the error (must be within 0 and 7)
 * @param error_msg  the message explaining the error
 * @param sd         socket id of the (UDP) socket to be used to send the messag
e
```

```c
 * @param addr       address of the client (server)
 * @return           0 in case of success, 1 otherwise
 */
int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in *addr);

/**
 * Send an ACK message.
 *
 * In current implementation it is only used for sending ACKs from client to
 * server.
 *
 * @param block_n    sequence number of the block to be acknowledged.
 * @param out_buffer buffer to be used for sending the ACK (useful for recycling the same buffer)
 * @param sd         socket id of the (UDP) socket to be used to send the message
 * @param addr       address of recipient of the ACK
 * @return           0 in case of success, 1 otherwise
 */
int tftp_send_ack(int block_n, char* out_buffer, 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!).
 *
 * @param m_fblock   block file where to write incoming data to
 * @param sd         socket id of the (UDP) socket to be used to send ACK messages
 * @param addr       address of the recipient of ACKs
 * @return
 * - 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.
 */
int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);

/**
 * Receive an ACK message.
 *
 * In current implementation it is only used for receiving ACKs from client.
 *
 * @param block_n [out] sequence number of the acknowledged block.
 * @param in_buffer     buffer to be used for receiving the ACK (useful for recycling the same buffer)
 * @param sd [in]       socket id of the (UDP) socket to be used to send the message
 * @param addr [in]     address of recipient of the ACK
 * @return
 * - 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 tftp_msg_unpack_ack
 */
int tftp_receive_ack(int *block_n, char* in_buffer, 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!).
 *
 * @param m_fblock   block file where to read incoming data from
 * @param sd         socket id of the (UDP) socket to be used to send DATA messa
ges
 * @param addr       address of the recipient of the file
 * @return
 * - 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.
 */
int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of tftp.h.
 *
 * @see 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"


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


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_WARN, "Received message from unexpected source: %s", addr_str);
        continue;
      } 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_WARN, "Received message from unexpected source: %s", addr_str);
        continue;
      } 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_WARN, "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, ret;

  // init sequence number
  block_n = 1;

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

    if (m_fblock->remaining > TFTP_DATA_BLOCK)
```

```c
      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");

    ret = tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr);

    if (ret == 2){  //unexpected source
      continue;
    } else if (ret != 0){
      LOG(LOG_ERR, "Error receiving ack: %d", ret);
      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;
}
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Conversion functions from netascii to Unix standard ASCII.
 *
 * 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 https://tools.ietf.org/html/rfc764
 */


#ifndef NETASCII
#define NETASCII


/**
 * Unix to netascii conversion.
 *
 * @param unix_filename     the filename of the input Unix file
 * @param netascii_filename the filename of the output netascii file
 * @return
 * - 0 in case of success
 * - 1 in case of an error opening unix_filename file
 * - 2 in case of an error opening netascii_filename file
 * - 3 in case of an error writing to netascii_filename file
 */
int unix2netascii(char *unix_filename, char* netascii_filename);

/**
 * Netascii to Unix conversion.
 *
 * @param netascii_filename the filename of the input netascii file
 * @param unix_filename     the filename of the output Unix file
 * @return
 * - 0 in case of success
 * - 1 in case of an error opening unix_filename file
 * - 2 in case of an error opening netascii_filename file
 * - 3 in case of an error writing to unix_filename file
 * - 3 in case of bad formatted netascii
 */
int netascii2unix(char* netascii_filename, char *unix_filename);


#endif
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of netascii.h.
 *
 * @see netascii.h
 */


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


/** LOG_LEVEL will be defined in another file */
extern const int LOG_LEVEL;


int unix2netascii(char *unix_filename, char* netascii_filename){
  FILE *unixf, *netasciif;
  char prev, tmp;
  int ret, result;

  unixf = fopen(unix_filename, "r");

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

  netasciif = fopen(netascii_filename, "w");

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

  prev = EOF;

  while ((tmp = (char) fgetc(unixf)) != EOF){
    if (tmp == '\n' && prev != '\r'){ // LF -> CRLF
      ret = putc('\r', netasciif);
      if (ret == EOF)
        break;

      ret = putc('\n', netasciif);
      if (ret == EOF)
        break;

    } else if (tmp == '\r'){  // CR -> CRNUL
      char next = (char) fgetc(unixf);
      if (next != '\0')
        ungetc(next, unixf);

      ret = putc('\r', netasciif);
      if (ret == EOF)
        break;

      ret = putc('\0', netasciif);
      if (ret == EOF)
        break;
    } else{
      ret = putc(tmp, netasciif);
      if (ret == EOF)
        break;
    }

    prev = tmp;
  }

  // Error writing to netasciif
  if (ret == EOF){
    LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
    result = 3;
  } else{
    LOG(LOG_INFO, "Unix file %s converted to netascii file %s", unix_filename, netascii_filename);
    result = 0;
  }

  fclose(unixf);
  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 0x%x after CR", next);
        result = 4;
        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_filen
ame, unix_filename);
      result = 0;
    }
  } // otherwise there was an error (4 or 5) and result was already set

  fclose(unixf);
  fclose(netasciif);

  return result;
}
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief 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>

/** Defining LOG_LEVEL for tftp_client executable */
const int LOG_LEVEL = LOG_WARN;


/** max stdin line length */
#define READ_BUFFER_SIZE 80

/** Maximum number of arguments for commands */
#define MAX_ARGS 3

/** String for txt */
#define MODE_TXT "txt"

/** String for bin*/
#define MODE_BIN "bin"


/**
 * Global transfer_mode variable for storing user chosen transfer mode string.
 *
 * @see MODE_TXT
 * @see MODE_BIN
 */
char* transfer_mode;


/**
 * Splits a string at each delim.
 *
 * Trailing LF will be removed. Consecutive delimiters will be considered as one
.
 *
 * @param line [in]      the string to split
 * @param delim [in]     the delimiter
 * @param max_argc [in] maximum number of parts to split the line into
 * @param argc [out]     counts of the parts the line is split into
 * @param argv [out]     array of parts the line is split into
 */
void split_string(char* line, char* delim, int max_argc, int *argc, char **argv)
```

```
{
  char *ptr;
  int len;/**
 * Prints command usage information.
 */
  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)++;
        }
}

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

/**
 * Handles !help command, printing information about available commands.
 */
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 <fil
ename> e lo salva localmente con il nome <nome_locale>\n");
  printf("!quit --> termina il client\n");
}

/**
 * Handles !mode command, changing transfer_mode to either bin or text.
 *
 * @see transfer_mode
 */
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);
  }
}

/**
 * Handles !get command, reading file from server.
 */
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
```

```c
      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;

  }

  /**
   * Handles !quit command.
   */
  void cmd_quit(){
    printf("Client terminato con successo\n");
    exit(0);
  }

  /** Main */
  int main(int argc, char** argv){
    char* sv_ip;
    short int sv_port;
    int ret, i;
    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(i = 0; i < cmd_argc; i++)
        free(cmd_argv[i]);
    }

    return 0;
  }
```

```c
/**
 * @file
 * @author Riccardo Mancini
 *
 * @brief Implementation of the TFTP server that can only handle read requests.
 *
 * The server is multiprocessed, with each process handling one request.
 */


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


/** Defining LOG_LEVEL for tftp_server executable */
const int LOG_LEVEL = LOG_INFO;


/** Maximum length for a RRQ message */
#define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4


/** Finds longest common prefix length of strings str1 and str2 */
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;
}

/**
 * Check whether file is inside dir.
 *
 * @param path  file absolute path (can include .. and .  and multiple /)
 * @param dir   directory real path (can't include .. and . and multiple /)
 * @return      1 if true, 0 otherwise
 *
 * @see realpath
 */
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;
}

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

/**
 * Sends file to a client.
 */
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_BI
NARY);
  } 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_MOD
E_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);

    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;
}

/** Main */
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 == -1){ // error
        LOG(LOG_FATAL, "Fork error");
        perror("Fork error:");
        return 1;
      } else 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;
}
```