

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief File block read and write.
6:  *
7:  * This library provides functions for reading and writing a text or binary
8:  * file using a predefined block size.
9:  */
10:
11: #ifndef FBLOCK
12: #define FBLOCK
13:
14:
15: #include <stdio.h>
16:
17:
18: /** Mask for getting text/binary mode */
19: #define FBLOCK_MODE_MASK 0b01
20:
21: /** Open file in text mode */
22: #define FBLOCK_MODE_TEXT 0b00
23:
24: /** Open file in binary mode */
25: #define FBLOCK_MODE_BINARY 0b01
26:
27: /** Mask for getting r/w mode */
28: #define FBLOCK_RW_MASK 0b10
29:
30: /** Open file in read mode */
31: #define FBLOCK_READ 0b00
32:
33: /** Open file in write mode */
34: #define FBLOCK_WRITE 0b10
35:
36:
37: /**
38:  * Structure which defines a file.
39:  */
40: struct fblock{
41:     FILE *file; /**< Pointer to the file */
42:     int block_size; /**< Predefined block size for i/o operations */
43:     char mode; /**< Can be read xor write, text xor binary. */
44:     union{
45:         int written; /**< Bytes already written (for future use) */
46:         int remaining; /**< Remaining bytes to read */
47:     };
48: };
49:
50:
51: /**
52:  * Opens a file.
53:  *
54:  * @param filename    name of the file
55:  * @param block_size  size of the blocks
56:  * @param mode        mode (read, write, text, binary)
57:  * @return            fblock structure
58:  *
59:  * @see FBLOCK_MODE_TEXT
60:  * @see FBLOCK_MODE_BINARY
61:  * @see FBLOCK_WRITE
62:  * @see FBLOCK_READ
63:  */
```

```
64: struct fblock fblock_open(char* filename, int block_size, char mode);
65:
66: /**
67:  * Reads next block_size bytes from file.
68:  *
69:  * @param m_fblock    fblock instance
70:  * @param buffer      block_size bytes buffer
71:  * @return            0 in case of success, otherwise number of bytes it could n
ot read
72:  */
73: int fblock_read(struct fblock *m_fblock, char* buffer);
74:
75: /**
76:  * Writes next block_size bytes to file.
77:  *
78:  * @param m_fblock    fblock instance
79:  * @param buffer      block_size bytes buffer
80:  * @param block_size  if set to a non-0 value, override block_size defined in fb
lock.
81:  * @return            0 in case of success, otherwise number of bytes it could n
ot write
82:  */
83: int fblock_write(struct fblock *m_fblock, char* buffer, int block_size);
84:
85: /**
86:  * Closes a file.
87:  *
88:  * @param m_fblock    fblock instance to be closed
89:  * @return            0 in case of success, EOF in case of failure
90:  *
91:  * @see fclose
92:  */
93: int fblock_close(struct fblock *m_fblock);
94:
95:
96: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of fblock.h.
6:  *
7:  * @see fblock.h
8:  */
9:
10:
11: #include "include/fblock.h"
12: #include <stdio.h>
13: #include <string.h>
14: #include "include/logging.h"
15:
16:
17: /** LOG_LEVEL will be defined in another file */
18: extern const int LOG_LEVEL;
19:
20:
21: /**
22:  * Returns file length
23:  *
24:  * @param f file pointer
25:  * @return file length in bytes
26:  */
27: int get_length(FILE *f){
28:     int size;
29:     fseek(f, 0, SEEK_END); // seek to end of file
30:     size = ftell(f); // get current file pointer
31:     fseek(f, 0, SEEK_SET); // seek back to beginning of file
32:     return size;
33: }
34:
35:
36: struct fblock fblock_open(char* filename, int block_size, char mode){
37:     struct fblock m_fblock;
38:     m_fblock.block_size = block_size;
39:     m_fblock.mode = mode;
40:
41:     char mode_str[4] = "";
42:
43:     LOG(LOG_DEBUG, "Opening file %s (%s %s), block_size = %d",
44:         filename,
45:         (mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY ? "binary" : "text",
46:         (mode & FBLOCK_RW_MASK) == FBLOCK_WRITE ? "write" : "read",
47:         block_size
48:     );
49:
50:     if ((mode & FBLOCK_RW_MASK) == FBLOCK_WRITE){
51:         strcat(mode_str, "w");
52:         m_fblock.written = 0;
53:     } else {
54:         strcat(mode_str, "r");
55:     }
56:
57:     if ((mode & FBLOCK_MODE_MASK) == FBLOCK_MODE_BINARY)
58:         strcat(mode_str, "b");
59:     // text otherwise
60:
61:     m_fblock.file = fopen(filename, mode_str);
62:     if (m_fblock.file == NULL){
63:         LOG(LOG_ERR, "Error while opening file %s", filename);
```

```
64:     return m_fblock;
65: }
66: if ((mode & FBLOCK_RW_MASK) == FBLOCK_READ)
67:     m_fblock.remaining = get_length(m_fblock.file);
68:
69: LOG(LOG_DEBUG, "Successfully opened file");
70: return m_fblock;
71: }
72:
73:
74: int fblock_read(struct fblock *m_fblock, char* buffer){
75:     int bytes_read, bytes_to_read;
76:
77:     if (m_fblock->remaining > m_fblock->block_size)
78:         bytes_to_read = m_fblock->block_size;
79:     else
80:         bytes_to_read = m_fblock->remaining;
81:
82:     bytes_read = fread(buffer, sizeof(char), bytes_to_read, m_fblock->file);
83:     m_fblock->remaining -= bytes_read;
84:
85:     return bytes_to_read - bytes_read;
86: }
87:
88:
89: int fblock_write(struct fblock *m_fblock, char* buffer, int block_size){
90:     int written_bytes;
91:
92:     if (!block_size)
93:         block_size = m_fblock->block_size;
94:
95:     written_bytes = fwrite(buffer, sizeof(char), block_size, m_fblock->file);
96:     m_fblock->written += written_bytes;
97:     return block_size - written_bytes;
98: }
99:
100: int fblock_close(struct fblock *m_fblock){
101:     return fclose(m_fblock->file);
102: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Contructor for TFTP messages.
6:  *
7:  * This library provides functions for building TFTP messages.
8:  * There are 5 types of messages:
9:  * - 1: Read request (RRQ)
10:  * - 2: Write request (WRQ)
11:  * - 3: Data (DATA)
12:  * - 4: Acknowledgment (ACK)
13:  * - 5: Error (ERROR)
14:  */
15:
16: #ifndef TFTP_MSGS
17: #define TFTP_MSGS
18:
19:
20: /** Read request message type */
21: #define TFTP_TYPE_RRQ 1
22:
23: /** Write request message type */
24: #define TFTP_TYPE_WRQ 2
25:
26: /** Data message type */
27: #define TFTP_TYPE_DATA 3
28:
29: /** Acknowledgment message type */
30: #define TFTP_TYPE_ACK 4
31:
32: /** Error message type */
33: #define TFTP_TYPE_ERROR 5
34:
35: /** String for netascii */
36: #define TFTP_STR_NETASCII "netascii"
37:
38: /** String for octet */
39: #define TFTP_STR_OCTET "octet"
40:
41: /** Maximum filename length (do not defined in RFC) */
42: #define TFTP_MAX_FILENAME_LEN 255
43:
44: /**
45:  * Maximum mode field string length
46:  *
47:  * Since there are only two options: 'netascii' and 'octet', len('netascii') is
48:  * the TFTP_MAX_MODE_LEN.
49:  */
50: #define TFTP_MAX_MODE_LEN 8
51:
52: /** Maximum error message length (do not defined in RFC) */
53: #define TFTP_MAX_ERROR_LEN 255
54:
55: /** Data block size as defined in RFC */
56: #define TFTP_DATA_BLOCK 512
57:
58: /** Data message max size is equal to TFTP_DATA_BLOCK + 4 (header) */
59: #define TFTP_MAX_DATA_MSG_SIZE 516
60:
61:
62: /**
63:  * Returns msg type given a message buffer.
```

```

64:  *
65:  * @param buffer the buffer
66:  * @return message type
67:  *
68:  * @see TFTP_TYPE_RRQ
69:  * @see TFTP_TYPE_WRQ
70:  * @see TFTP_TYPE_DATA
71:  * @see TFTP_TYPE_ACK
72:  * @see TFTP_TYPE_ERROR
73:  */
74: int tftp_msg_type(char *buffer);
75:
76:
77: /**
78:  * Builds a read request message.
79:  *
80:  * '''
81:  * 2 bytes      string      1 byte      string      1 byte
82:  * -----
83:  * | 01 | Filename | 0 | Mode | 0 |
84:  * -----
85:  * '''
86:  *
87:  * @param filename name of the file
88:  * @param mode      requested transfer mode ("netascii" or "octet")
89:  * @param buffer    data buffer where to build the message
90:  */
91: void tftp_msg_build_rrq(char* filename, char* mode, char* buffer);
92:
93: /**
94:  * Unpacks a read request message.
95:  *
96:  * @param buffer    data buffer where the message to read is [in]
97:  * @param buffer_len length of the buffer [in]
98:  * @param filename  name of the file [out]
99:  * @param mode      requested transfer mode ("netascii" or "octet") [out]
100:  * @return
101:  * - 0 in case of success.
102:  * - 1 in case of wrong operation code.
103:  * - 2 in case of unexpected fields inside message.
104:  * - 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
105:  * - 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
106:  * - 5 in case of unrecognized transfer mode.
107:  *
108:  * @see TFTP_TYPE_RRQ
109:  * @see TFTP_MAX_FILENAME_LEN
110:  * @see TFTP_MAX_MODE_LEN
111:  * @see TFTP_STR_NETASCII
112:  * @see TFTP_STR_OCTET
113:  */
114: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode
);
115:
116: /**
117:  * Returns size in bytes of a read request message.
118:  *
119:  * @param filename name of the file
120:  * @param mode      requested transfer mode ("netascii" or "octet")
121:  * @return          size in bytes
122:  */
123: int tftp_msg_get_size_rrq(char* filename, char* mode);
124:
125: /**

```

```

126:  * Builds a write request message.
127:  *
128:  * Message format:
129:  * ```
130:  *  2 bytes      string      1 byte      string      1 byte
131:  *  -----
132:  *  |    02    |  Filename  |    0    |    Mode    |    0    |
133:  *  -----
134:  *  ```
135:  *
136:  * @param filename  name of the file
137:  * @param mode      requested transfer mode ("netascii" or "octet")
138:  * @param buffer    data buffer where to build the message
139:  */
140: void tftp_msg_build_wrq(char* filename, char* mode, char* buffer);
141:
142: /**
143:  * Unpacks a write request message.
144:  *
145:  * @param buffer      data buffer where the message to read is [in]
146:  * @param buffer_len  length of the buffer [in]
147:  * @param filename    name of the file [out]
148:  * @param mode        requested transfer mode ("netascii" or "octet") [out]
149:  * @return
150:  * - 0 in case of success.
151:  * - 1 in case of wrong operation code.
152:  * - 2 in case of unexpected fields inside message.
153:  * - 3 in case of filename exceeding TFTP_MAX_FILENAME_LEN.
154:  * - 4 in case of mode string exceeding TFTP_MAX_MODE_LEN.
155:  * - 5 in case of unrecognized transfer mode.
156:  *
157:  * @see TFTP_TYPE_WRQ
158:  * @see TFTP_MAX_FILENAME_LEN
159:  * @see TFTP_MAX_MODE_LEN
160:  * @see TFTP_STR_NETASCII
161:  * @see TFTP_STR_OCTET
162:  */
163: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode
);
164:
165: /**
166:  * Returns size in bytes of a write request message.
167:  *
168:  * @param filename  name of the file
169:  * @param mode      requested transfer mode ("netascii" or "octet")
170:  * @return          size in bytes
171:  */
172: int tftp_msg_get_size_wrq(char* filename, char* mode);
173:
174: /**
175:  * Builds a data message.
176:  *
177:  * Message format:
178:  * ```
179:  *  2 bytes      2 bytes      n bytes
180:  *  -----
181:  *  |  03  |  Block #  |  Data  |
182:  *  -----
183:  *  ```
184:  *
185:  * @param block_n    block sequence number
186:  * @param data        pointer to the buffer containing the data to be transfered
187:  * @param data_size   data buffer size

```

```
188:  * @param buffer    data buffer where to build the message
189:  */
190: void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer);
191:
192: /**
193:  * Unpacks a data message.
194:  *
195:  * @param buffer    data buffer where the message to read is [in]
196:  * @param buffer_len length of the buffer [in]
197:  * @param block_n   pointer where block_n will be written [out]
198:  * @param data      pointer where to copy data [out]
199:  * @return
200:  * - 0 in case of success.
201:  * - 1 in case of wrong operation code.
202:  * - 2 in case of missing fields (packet size is too small).
203:  *
204:  * @see TFTP_TYPE_DATA
205:  */
206: int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
int* data_size);
207:
208: /**
209:  * Returns size in bytes of a data message.
210:  *
211:  * It just sums 4 to data_size.
212:  *
213:  * @param data_size data buffer size
214:  * @return          size in bytes
215:  */
216: int tftp_msg_get_size_data(int data_size);
217:
218: /**
219:  * Builds an acknowledgment message.
220:  *
221:  * Message format:
222:  * ```
223:  * 2 bytes      2 bytes
224:  * -----
225:  * | 04      | Block # |
226:  * -----
227:  * ```
228:  *
229:  * @param block_n   block sequence number
230:  * @param buffer    data buffer where to build the message
231:  */
232: void tftp_msg_build_ack(int block_n, char* buffer);
233:
234: /**
235:  * Unpacks an acknowledgment message.
236:  *
237:  * @param buffer    data buffer where the message to read is [in]
238:  * @param buffer_len length of the buffer [in]
239:  * @param block_n   pointer where block_n will be written [out]
240:  * @param data      pointer inside buffer where the data is [out]
241:  * @return
242:  * - 0 in case of success.
243:  * - 1 in case of wrong operation code.
244:  * - 2 in case of wrong packet size.
245:  *
246:  * @see TFTP_TYPE_ACK
247:  */
248: int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n);
249:
```



```

250: /**
251:  * Returns size in bytes of an acknowledgment message.
252:  *
253:  * It just returns 4.
254:  *
255:  * @param data_size data buffer size
256:  * @return          size in bytes
257:  */
258: int tftp_msg_get_size_ack();
259:
260: /**
261:  * Builds an error message.
262:  *
263:  * Message format:
264:  * '''
265:  *   2 bytes  2 bytes          string    1 byte
266:  *   -----
267:  *   | 05      | ErrorCode | ErrMsg   | 0 |
268:  *   -----
269:  *   '''
270:  *
271:  * Error code meaning:
272:  * - 0: Not defined, see error message (if any).
273:  * - 1: File not found.
274:  * - 2: Access violation.
275:  * - 3: Disk full or allocation exceeded.
276:  * - 4: Illegal TFTP operation.
277:  * - 5: Unknown transfer ID.
278:  * - 6: File already exists.
279:  * - 7: No such user.
280:  *
281:  * In current implementation only errors 1 and 4 are implemented.
282:  *
283:  * @param error_code error code (from 0 to 7)
284:  * @param error_msg  error message
285:  * @param buffer     data buffer where to build the message
286:  */
287: void tftp_msg_build_error(int error_code, char* error_msg, char* buffer);
288:
289: /**
290:  * Unpacks an error message.
291:  *
292:  * @param buffer     data buffer where the message to read is [in]
293:  * @param buffer_len length of the buffer [in]
294:  * @param error_code pointer where error_code will be written [out]
295:  * @param error_msg  pointer to error message inside the message [out]
296:  * @return
297:  * - 0 in case of success.
298:  * - 1 in case of wrong operation code.
299:  * - 2 in case of unexpected fields.
300:  * - 3 in case of error string exceeding TFTP_MAX_ERROR_LEN.
301:  * - 4 in case of unrecognized error code (must be within 0 and 7).
302:  *
303:  * @see TFTP_TYPE_ERROR
304:  * @see TFTP_MAX_ERROR_LEN
305:  */
306: int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char* e
rror_msg);
307:
308: /**
309:  * Returns size in bytes of an error message.
310:  *
311:  * @param error_msg  error message

```

```
312:  * @return          size in bytes
313:  */
314: int tftp_msg_get_size_error(char* error_msg);
315:
316:
317: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of tftp_msgs.h .
6:  *
7:  * @see tftp_msgs.h
8:  */
9:
10:
11: #include "include/tftp_msgs.h"
12: #include "include/logging.h"
13: #include <string.h>
14: #include <strings.h>
15: #include <stdio.h>
16: #include <arpa/inet.h>
17: #include <stdint.h>
18:
19:
20: /** LOG_LEVEL will be defined in another file */
21: extern const int LOG_LEVEL;
22:
23:
24: int tftp_msg_type(char *buffer){
25:     return (((int)buffer[0]) << 8) + buffer[1];
26: }
27:
28:
29: void tftp_msg_build_rrq(char* filename, char* mode, char* buffer){
30:     buffer[0] = 0;
31:     buffer[1] = 1;
32:     buffer += 2;
33:     strcpy(buffer, filename);
34:     buffer += strlen(filename)+1;
35:     strcpy(buffer, mode);
36: }
37:
38:
39: int tftp_msg_unpack_rrq(char* buffer, int buffer_len, char* filename, char* mode
){
40:     int offset = 0;
41:     if (tftp_msg_type(buffer) != TFTP_TYPE_RRQ){
42:         LOG(LOG_ERR, "Expected RRQ message (1), found %d", tftp_msg_type(buffer));
43:         return 1;
44:     }
45:
46:     offset += 2;
47:     if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
48:         LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
49:         return 3;
50:     }
51:     strcpy(filename, buffer+offset);
52:
53:     offset += strlen(filename)+1;
54:     if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
55:         LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
56:         return 4;
57:     }
58:     strcpy(mode, buffer+offset);
59:
60:     offset += strlen(mode)+1;
```

```
61:  if (buffer_len != offset){
62:      LOG(LOG_ERR, "Packet contains unexpected fields");
63:      return 2;
64:  }
65:  if (strcasecmp(mode, TFTP_STR_NETASCII) == 0 || strcasecmp(mode, TFTP_STR_OCTE
T) == 0)
66:      return 0;
67:  else{
68:      LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
69:      return 5;
70:  }
71: }
72:
73:
74: int tftp_msg_get_size_rrq(char* filename, char* mode){
75:     return 4 + strlen(filename) + strlen(mode);
76: }
77:
78:
79: void tftp_msg_build_wrq(char* filename, char* mode, char* buffer){
80:     buffer[0] = 0;
81:     buffer[1] = 2;
82:     buffer += 2;
83:     strcpy(buffer, filename);
84:     buffer += strlen(filename)+1;
85:     strcpy(buffer, mode);
86: }
87:
88:
89: int tftp_msg_unpack_wrq(char* buffer, int buffer_len, char* filename, char* mode
){
90:     int offset = 0;
91:     if (tftp_msg_type(buffer) != TFTP_TYPE_WRQ){
92:         LOG(LOG_ERR, "Expected WRQ message (2), found %d", tftp_msg_type(buffer));
93:         return 1;
94:     }
95:
96:     offset += 2;
97:     if (strlen(buffer+offset) > TFTP_MAX_FILENAME_LEN){
98:         LOG(LOG_ERR, "Filename too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_FILENAME_LEN, buffer+offset);
99:         return 3;
100:    }
101:
102:    strcpy(filename, buffer+offset);
103:    offset += strlen(filename)+1;
104:    if (strlen(buffer+offset) > TFTP_MAX_MODE_LEN){
105:        LOG(LOG_ERR, "Mode string too long (%d > %d): %s", (int) strlen(buffer+offset),
TFTP_MAX_MODE_LEN, buffer+offset);
106:        return 4;
107:    }
108:
109:    strcpy(mode, buffer+offset);
110:    offset += strlen(mode)+1;
111:    if (buffer_len != offset){
112:        LOG(LOG_ERR, "Packet contains unexpected fields");
113:        return 2;
114:    }
115:
116:    if (strcmp(mode, TFTP_STR_NETASCII) == 0 || strcmp(mode, TFTP_STR_OCTET) == 0)
117:        return 0;
118:    else{
119:        LOG(LOG_ERR, "Unrecognized transfer mode: %s", mode);
```

```
120:     return 5;
121: }
122: }
123:
124:
125: int tftp_msg_get_size_wrq(char* filename, char* mode){
126:     return 4 + strlen(filename) + strlen(mode);
127: }
128:
129:
130: void tftp_msg_build_data(int block_n, char* data, int data_size, char* buffer){
131:     buffer[0] = 0;
132:     buffer[1] = 3;
133:     *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
134:     buffer += 4;
135:     memcpy(buffer, data, data_size);
136: }
137:
138:
139: int tftp_msg_unpack_data(char* buffer, int buffer_len, int* block_n, char* data,
int* data_size){
140:     if (tftp_msg_type(buffer) != TFTP_TYPE_DATA){
141:         LOG(LOG_ERR, "Expected DATA message (3), found %d", tftp_msg_type(buffer));
142:         return 1;
143:     }
144:
145:     if (buffer_len < 4){
146:         LOG(LOG_ERR, "Packet size too small for DATA: %d > 4", buffer_len);
147:         return 2;
148:     }
149:
150:     *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
151:     *data_size = buffer_len - 4;
152:     if (*data_size > 0)
153:         memcpy(data, buffer+4, *data_size);
154:     return 0;
155: }
156:
157:
158: int tftp_msg_get_size_data(int data_size){
159:     return data_size + 4;
160: }
161:
162:
163: void tftp_msg_build_ack(int block_n, char* buffer){
164:     buffer[0] = 0;
165:     buffer[1] = 4;
166:     *((uint16_t*)(buffer+2)) = htons((uint16_t) block_n);
167: }
168:
169:
170: int tftp_msg_unpack_ack(char* buffer, int buffer_len, int* block_n){
171:     if (tftp_msg_type(buffer) != TFTP_TYPE_ACK){
172:         LOG(LOG_ERR, "Expected ACK message (4), found %d", tftp_msg_type(buffer));
173:         return 1;
174:     }
175:
176:     if (buffer_len != 4){
177:         LOG(LOG_ERR, "Wrong packet size for ACK: %d != 4", buffer_len);
178:         return 2;
179:     }
180:     *block_n = (int) ntohs(*((uint16_t*)(buffer+2)));
181:     return 0;
```

```
182: }
183:
184:
185: int tftp_msg_get_size_ack(){
186:     return 4;
187: }
188:
189:
190: void tftp_msg_build_error(int error_code, char* error_msg, char* buffer){
191:     buffer[0] = 0;
192:     buffer[1] = 5;
193:     *((uint16_t*)(buffer+2)) = htons((uint16_t) error_code);
194:     buffer += 4;
195:     strcpy(buffer, error_msg);
196: }
197:
198:
199: int tftp_msg_unpack_error(char* buffer, int buffer_len, int* error_code, char* e
rror_msg){
200:     if (tftp_msg_type(buffer) != TFTP_TYPE_ERROR){
201:         LOG(LOG_ERR, "Expected ERROR message (5), found %d", tftp_msg_type(buffer)
);
202:         return 1;
203:     }
204:
205:     *error_code = (int) ntohs(*((uint16_t*)(buffer+2)));
206:     if (*error_code < 0 || *error_code > 7){
207:         LOG(LOG_ERR, "Unrecognized error code: %d", *error_code);
208:         return 4;
209:     }
210:
211:     buffer += 4;
212:     if(strlen(buffer) > TFTP_MAX_ERROR_LEN){
213:         LOG(LOG_ERR, "Error string too long (%d > %d): %s", (int) strlen(buffer),
TFTP_MAX_ERROR_LEN, buffer);
214:         return 3;
215:     }
216:
217:     strcpy(error_msg, buffer);
218:     if (buffer_len != strlen(error_msg)+5){
219:         LOG(LOG_WARN, "Packet contains unexpected fields");
220:         return 2;
221:     }
222:     return 0;
223: }
224:
225:
226: int tftp_msg_get_size_error(char* error_msg){
227:     return 5 + strlen(error_msg);
228: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility functions for managing inet addresses.
6:  *
7:  * This library provides functions for creating sockaddr_in structures from
8:  * IP address string and integer port number and for binding to a random
9:  * port (chosen using rand() builtin C function).
10:  *
11:  * @see sockaddr_in
12:  * @see rand
13:  */
14:
15: #ifndef INET_UTILS
16: #define INET_UTILS
17:
18:
19: #include <sys/socket.h>
20: #include <netinet/in.h>
21:
22: /** Random port will be greater or equal to FROM_PORT */
23: #define FROM_PORT 49152
24:
25: /** Random port will be lower or equal to TO_PORT */
26: #define TO_PORT 65535
27:
28: /** Maximum number of trials before giving up opening a random port */
29: #define MAX_TRIES 256
30:
31: /** Maximum number of characters of INET address to string (eg 123.156.189.123:4
5678) */
32: #define MAX_SOCKADDR_STR_LEN 22
33:
34:
35: /**
36:  * Binds socket to a random port.
37:  *
38:  * @param socket    socket ID
39:  * @param addr      inet addr structure
40:  * @return          0 in case of failure, port it could bind to otherwise
41:  *
42:  * @see FROM_PORT
43:  * @see TO_PORT
44:  * @see MAX_TRIES
45:  */
46: int bind_random_port(int socket, struct sockaddr_in *addr);
47:
48: /**
49:  * Makes sockaddr_in structure given ip string and port of server.
50:  *
51:  * @param ip        ip address of server
52:  * @param port      port of the server
53:  * @return          sockaddr_in structure for the given server
54:  */
55: struct sockaddr_in make_sv_sockaddr_in(char* ip, int port);
56:
57: /**
58:  * Makes sockaddr_in structure of this host.
59:  *
60:  * INADDR_ANY is used as IP address.
61:  *
62:  * @param port      port of the server
```

```
63:  * @return      sockaddr_in structure this host on given port
64:  */
65:  struct sockaddr_in make_my_sockaddr_in(int port);
66:
67:  /**
68:   * Compares INET addresses, returning 0 in case they're equal.
69:   *
70:   * @param sai1   first address
71:   * @param sai2   second address
72:   * @return      0 if they're equal, 1 otherwise
73:   */
74:  int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2);
75:
76:  /**
77:   * Converts sockaddr_in structure to string to be printed.
78:   *
79:   * @param src    the input address
80:   * @param dst    the output string (must be at least MAX_SOCKADDR_STR_LEN long)
81:   */
82:  void sockaddr_in_to_string(struct sockaddr_in src, char *dst);
83:
84:
85: #endif
```



```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of inet_utils.h.
6:  *
7:  * @see inet_utils.h
8:  */
9:
10:
11: #include "include/inet_utils.h"
12: #include <stdlib.h>
13: #include <string.h>
14: #include <sys/socket.h>
15: #include <netinet/in.h>
16: #include <arpa/inet.h>
17: #include "include/logging.h"
18:
19:
20: /** LOG_LEVEL will be defined in another file */
21: extern const int LOG_LEVEL;
22:
23:
24: int bind_random_port(int socket, struct sockaddr_in *addr){
25:     int port, ret, i;
26:     for (i=0; i<MAX_TRIES; i++){
27:         if (i == 0) // first I generate a random one
28:             port = rand() % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
29:         else //if it's not free I scan the next one
30:             port = (port-FROM_PORT+1) % (TO_PORT - FROM_PORT + 1) + FROM_PORT;
31:
32:         LOG(LOG_DEBUG, "Trying port %d...", port);
33:
34:         addr->sin_port = htons(port);
35:         ret = bind(socket, (struct sockaddr*) addr, sizeof(*addr));
36:         if (ret != -1)
37:             return port;
38:         // consider only some errors?
39:     }
40:     LOG(LOG_ERR, "Could not bind to random port after %d attempts", MAX_TRIES);
41:     return 0;
42: }
43:
44:
45: struct sockaddr_in make_sv_sockaddr_in(char* ip, int port){
46:     struct sockaddr_in addr;
47:     memset(&addr, 0, sizeof(addr));
48:     addr.sin_family = AF_INET;
49:     addr.sin_port = htons(port);
50:     inet_pton(AF_INET, ip, &addr.sin_addr);
51:     return addr;
52: }
53:
54:
55: struct sockaddr_in make_my_sockaddr_in(int port){
56:     struct sockaddr_in addr;
57:     memset(&addr, 0, sizeof(addr));
58:     addr.sin_family = AF_INET;
59:     addr.sin_port = htons(port);
60:     addr.sin_addr.s_addr = htonl(INADDR_ANY);
61:     return addr;
62: }
63:
```

```
64:
65: int sockaddr_in_cmp(struct sockaddr_in sai1, struct sockaddr_in sai2){
66:     if (sai1.sin_port == sai2.sin_port && sai1.sin_addr.s_addr == sai2.sin_addr.s_
addr)
67:         return 0;
68:     else
69:         return 1;
70: }
71:
72: void sockaddr_in_to_string(struct sockaddr_in src, char *dst){
73:     char* port_str;
74:
75:     port_str = malloc(6);
76:     sprintf(port_str, "%d", ntohs(src.sin_port));
77:
78:     if (inet_ntop(AF_INET, (void*) &src.sin_addr, dst, MAX_SOCKADDR_STR_LEN) != NU
LL){
79:         strcat(dst, ":");
80:         strcat(dst, port_str);
81:     } else{
82:         strcpy(dst, "ERROR");
83:     }
84:
85:     free(port_str);
86: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Utility functions for debugging.
6:  *
7:  * At the moment, this library implements only one function for dumping a
8:  * buffer using hexadecimal.
9:  */
10:
11: #ifndef DEBUG_UTILS
12: #define DEBUG_UTILS
13:
14:
15: /**
16:  * Prints content of buffer to stdout, showing it as hex values.
17:  *
18:  * @param buffer    pointer to the buffer to be printed
19:  * @param len       the length (in bytes) of the buffer
20:  */
21: void dump_buffer_hex(char* buffer, int len);
22:
23:
24: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of debug_utils.h.
6:  *
7:  * @see debug_utils.h
8:  */
9:
10:
11: #include "include/debug_utils.h"
12: #include "include/logging.h"
13: #include <stdio.h>
14: #include <stdlib.h>
15: #include <string.h>
16:
17:
18: /** LOG_LEVEL will be defined in another file */
19: extern const int LOG_LEVEL;
20:
21:
22: void dump_buffer_hex(char* buffer, int len){
23:     char *str, tmp[4];
24:     int i;
25:
26:     str = malloc(len*3+1);
27:
28:     str[0] = '\0';
29:     for (i=0; i<len; i++){
30:         sprintf(tmp, "%02x ", (unsigned char) buffer[i]);
31:         strcat(str, tmp);
32:     }
33:
34:     LOG(LOG_DEBUG, "%s", str);
35:     free(str);
36: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Common functions for TFTP client and server.
6:  *
7:  * This library provides functions for sending requests, errors and exchanging
8:  * files using the TFTP protocol.
9:  *
10: * Even though the project assignment does not require the client to send files
11: * to the server, I still decided to include those functions in a common library
12: * in case in the future I decide to complete the TFTP implementation.
13: */
14:
15: #ifndef TFTP
16: #define TFTP
17:
18:
19: #include <sys/socket.h>
20: #include <netinet/in.h>
21: #include "fblock.h"
22:
23:
24: /**
25:  * Send a RRQ message to a server.
26:  *
27:  * @param filename the name of the requested file
28:  * @param mode the desired mode of transfer (netascii or octet)
29:  * @param sd socket id of the (UDP) socket to be used to send the message
30:  * @param addr address of the server
31:  * @return 0 in case of success, 1 otherwise
32:  *
33:  * @see TFTP_STR_NETASCII
34:  * @see TFTP_STR_OCTET
35:  */
36: int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
37:
38: /**
39:  * Send a WRQ message to a server.
40:  *
41:  * Do not used in current implementation.
42:  *
43:  * @param filename the name of the requested file
44:  * @param mode the desired mode of transfer (netascii or octet)
45:  * @param sd socket id of the (UDP) socket to be used to send the message
46:  * @param addr address of the server
47:  * @return 0 in case of success, 1 otherwise
48:  *
49:  * @see TFTP_STR_NETASCII
50:  * @see TFTP_STR_OCTET
51:  */
52: int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr);
53:
54: /**
55:  * Send an ERROR message to the client (server).
56:  *
57:  * In current implementation it is only used for sending File Not Found and
58:  * Illegal TFTP Operation errors to clients.
59:  *
60:  * @param error_code the code of the error (must be within 0 and 7)
61:  * @param error_msg the message explaining the error
62:  * @param sd socket id of the (UDP) socket to be used to send the messag
```

```

63:  * @param addr      address of the client (server)
64:  * @return          0 in case of success, 1 otherwise
65:  */
66: int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in
*addr);
67:
68: /**
69:  * Send an ACK message.
70:  *
71:  * In current implementation it is only used for sending ACKs from client to
72:  * server.
73:  *
74:  * @param block_n    sequence number of the block to be acknowledged.
75:  * @param out_buffer buffer to be used for sending the ACK (useful for recycling
the same buffer)
76:  * @param sd         socket id of the (UDP) socket to be used to send the messag
e
77:  * @param addr      address of recipient of the ACK
78:  * @return          0 in case of success, 1 otherwise
79:  */
80: int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *add
r);
81:
82: /**
83:  * Handle the entire workflow required to receive a file.
84:  *
85:  * In current implementation it is only used in client but it could be also
86:  * used on the server side, potentially (some tweaks may be needed, though!).
87:  *
88:  * @param m_fblock   block file where to write incoming data to
89:  * @param sd         socket id of the (UDP) socket to be used to send ACK messag
es
90:  * @param addr      address of the recipient of ACKs
91:  * @return
92:  * - 0 in case of success.
93:  * - 1 in case of file not found.
94:  * - 2 in case of error while sending ACK.
95:  * - 3 in case of unexpected sequence number.
96:  * - 4 in case of an error while unpacking data.
97:  * - 5 in case of an error while unpacking an incoming error message.
98:  * - 6 in case of an error while writing to the file.
99:  * - 7 in case of an error message different from File Not Found (since it is th
e only error available in current implementation).
100:  * - 8 in case of the incoming message is neither DATA nor ERROR.
101:  */
102: int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr)
;
103:
104: /**
105:  * Receive an ACK message.
106:  *
107:  * In current implementation it is only used for receiving ACKs from client.
108:  *
109:  * @param block_n [out] sequence number of the acknowledged block.
110:  * @param in_buffer buffer to be used for receiving the ACK (useful for recy
cling the same buffer)
111:  * @param sd [in]    socket id of the (UDP) socket to be used to send the mes
sage
112:  * @param addr [in]  address of recipient of the ACK
113:  * @return
114:  * - 0 in case of success
115:  * - 1 in case of failure while receiving the message
116:  * - 2 in case of address and/or port mismatch in sender sockaddr

```

```
117:  * - error unpacking ACK message otherwise (8 + result of tftp_msg_unpack_ack)
118:  *
119:  * @see tftp_msg_unpack_ack
120:  */
121: int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *
addr);
122:
123: /**
124:  * Handle the entire workflow required to send a file.
125:  *
126:  * In current implementation it is only used in server but it could be also
127:  * used on the client side, potentially (some tweaks may be needed, though!).
128:  *
129:  * @param m_fblock    block file where to read incoming data from
130:  * @param sd           socket id of the (UDP) socket to be used to send DATA messa
ges
131:  * @param addr         address of the recipient of the file
132:  * @return
133:  * - 0 in case of success.
134:  * - 1 in case of error sending a packet.
135:  * - 2 in case of error while receiving the ack.
136:  * - 3 in case of unexpected sequence number in ack.
137:  * - 4 in case of an error while unpacking data.
138:  */
139: int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr);
140:
141:
142: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of tftp.h.
6:  *
7:  * @see tftp.h
8:  */
9:
10:
11: #include "include/fblock.h"
12: #include "include/tftp_msgs.h"
13: #include "include/debug_utils.h"
14: #include "include/inet_utils.h"
15: #include <arpa/inet.h>
16: #include <sys/socket.h>
17: #include <netinet/in.h>
18: #include <stdlib.h>
19: #include "include/logging.h"
20:
21:
22: /** LOG_LEVEL will be defined in another file */
23: extern const int LOG_LEVEL;
24:
25:
26: int tftp_send_rrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
27:     int msglen, len;
28:     char *out_buffer;
29:
30:     msglen = tftp_msg_get_size_rrq(filename, mode);
31:     out_buffer = malloc(msglen);
32:
33:     tftp_msg_build_rrq(filename, mode, out_buffer);
34:     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr)
);
35:     if (len != msglen){
36:         LOG(LOG_ERR, "Error sending RRQ: len (%d) != msglen (%d)", len, msglen);
37:         perror("Error");
38:         return 1;
39:     }
40:
41:     free(out_buffer);
42:     return 0;
43: }
44:
45:
46: int tftp_send_wrq(char* filename, char *mode, int sd, struct sockaddr_in *addr){
47:     int msglen, len;
48:     char *out_buffer;
49:
50:     msglen = tftp_msg_get_size_wrq(filename, mode);
51:     out_buffer = malloc(msglen);
52:
53:     tftp_msg_build_wrq(filename, mode, out_buffer);
54:     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr)
);
55:     if (len != msglen){
56:         LOG(LOG_ERR, "Error sending WRQ: len (%d) != msglen (%d)", len, msglen);
57:         perror("Error");
58:         return 1;
59:     }
60:
61:     free(out_buffer);
```



```
62:     return 0;
63: }
64:
65:
66: int tftp_send_error(int error_code, char* error_msg, int sd, struct sockaddr_in
*addr){
67:     int msglen, len;
68:     char *out_buffer;
69:
70:     msglen = tftp_msg_get_size_error(error_msg);
71:     out_buffer = malloc(msglen);
72:
73:     tftp_msg_build_error(error_code, error_msg, out_buffer);
74:     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr)
);
75:     if (len != msglen){
76:         LOG(LOG_ERR, "Error sending ERROR: len (%d) != msglen (%d)", len, msglen);
77:         perror("Error");
78:         return 1;
79:     }
80:
81:     free(out_buffer);
82:     return 0;
83: }
84:
85:
86: int tftp_send_ack(int block_n, char* out_buffer, int sd, struct sockaddr_in *add
r){
87:     int msglen, len;
88:
89:     msglen = tftp_msg_get_size_ack();
90:     tftp_msg_build_ack(block_n, out_buffer);
91:     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*) addr, sizeof(*addr)
);
92:
93:     if (len != msglen){
94:         LOG(LOG_ERR, "Error sending ACK: len (%d) != msglen (%d)", len, msglen);
95:         perror("Error");
96:         return 1;
97:     }
98:
99:     return 0;
100: }
101:
102:
103: int tftp_receive_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr)
{
104:     char in_buffer[TFTP_MAX_DATA_MSG_SIZE], data[TFTP_DATA_BLOCK], out_buffer[4];
105:     int exp_block_n, rcv_block_n;
106:     int len, data_size, ret, type;
107:     unsigned int addrlen;
108:     struct sockaddr_in cl_addr, orig_cl_addr;
109:
110:     // init expected block number
111:     exp_block_n = 1;
112:
113:     addrlen = sizeof(cl_addr);
114:
115:     do{
116:         LOG(LOG_DEBUG, "Waiting for part %d", exp_block_n);
117:         // TODO: check client == server ?
118:         len = recvfrom(sd, in_buffer, tftp_msg_get_size_data(TFTP_DATA_BLOCK), 0, (s
truct sockaddr*)&cl_addr, &addrlen);
```

```
119:     if (exp_block_n == 1){ // first block -> I need to save servers TID (aka its
"original" sockaddr)
120:         char addr_str[MAX_SOCKADDR_STR_LEN];
121:         sockaddr_in_to_string(cl_addr, addr_str);
122:
123:         if (addr->sin_addr.s_addr != cl_addr.sin_addr.s_addr){
124:             LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
125:             continue;
126:         } else{
127:             LOG(LOG_INFO, "Receiving packets from %s", addr_str);
128:             orig_cl_addr = cl_addr;
129:         }
130:     } else{
131:         if (sockaddr_in_cmp(orig_cl_addr, cl_addr) != 0){
132:             char addr_str[MAX_SOCKADDR_STR_LEN];
133:             sockaddr_in_to_string(cl_addr, addr_str);
134:             LOG(LOG_WARN, "Received message from unexpected source: %s", addr_str);
135:             continue;
136:         } else{
137:             LOG(LOG_DEBUG, "Sender is the same!");
138:         }
139:     }
140:
141:     type = tftp_msg_type(in_buffer);
142:     if (type == TFTP_TYPE_ERROR){
143:         int error_code;
144:         char error_msg[TFTP_MAX_ERROR_LEN];
145:
146:         ret = tftp_msg_unpack_error(in_buffer, len, &error_code, error_msg);
147:         if (ret != 0){
148:             LOG(LOG_ERR, "Error unpacking error msg");
149:             return 5;
150:         }
151:
152:         if (error_code == 1){
153:             LOG(LOG_INFO, "File not found");
154:             return 1;
155:         } else{
156:             LOG(LOG_ERR, "Received error %d: %s", error_code, error_msg);
157:             return 7;
158:         }
159:
160:     } else if (type != TFTP_TYPE_DATA){
161:         LOG(LOG_ERR, "Received packet of type %d, expecting DATA or ERROR.", type)
;
162:         return 8;
163:     }
164:
165:     ret = tftp_msg_unpack_data(in_buffer, len, &rcv_block_n, data, &data_size);
166:
167:     if (ret != 0){
168:         LOG(LOG_ERR, "Error unpacking data: %d", ret);
169:         return 4;
170:     }
171:
172:     if (rcv_block_n != exp_block_n){
173:         LOG(LOG_ERR, "Received unexpected block_n: rcv_block_n = %d != %d = exp_block_n", rcv_block_n, exp_block_n);
174:         return 3;
175:     }
176:
177:     exp_block_n++;
178:
```

```
179:     LOG(LOG_DEBUG, "Part %d has size %d", rcv_block_n, data_size);
180:
181:     if (data_size != 0){
182:         if (fblock_write(m_fblock, data, data_size))
183:             return 6;
184:     }
185:
186:     LOG(LOG_DEBUG, "Sending ack");
187:
188:     if (tftp_send_ack(rcv_block_n, out_buffer, sd, &cl_addr))
189:         return 2;
190:
191: } while(data_size == TFTP_DATA_BLOCK);
192: return 0;
193: }
194:
195:
196: int tftp_receive_ack(int *block_n, char* in_buffer, int sd, struct sockaddr_in *
addr){
197:     int msglen, len, ret;
198:     unsigned int addrlen;
199:     struct sockaddr_in cl_addr;
200:
201:     msglen = tftp_msg_get_size_ack();
202:     addrlen = sizeof(cl_addr);
203:
204:     len = recvfrom(sd, in_buffer, msglen, 0, (struct sockaddr*)&cl_addr, &addrlen)
;
205:
206:     if (sockaddr_in_cmp(*addr, cl_addr) != 0){
207:         char str_addr[MAX_SOCKADDR_STR_LEN];
208:         sockaddr_in_to_string(cl_addr, str_addr);
209:         LOG(LOG_WARN, "Message is coming from unexpected source: %s", str_addr);
210:         return 2;
211:     }
212:
213:     if (len != msglen){
214:         LOG(LOG_ERR, "Error receiving ACK: len (%d) != msglen (%d)", len, msglen);
215:         return 1;
216:     }
217:
218:     ret = tftp_msg_unpack_ack(in_buffer, len, block_n);
219:     if (ret != 0){
220:         LOG(LOG_ERR, "Error unpacking ack: %d", ret);
221:         return 8+ret;
222:     }
223:
224:     return 0;
225: }
226:
227:
228: int tftp_send_file(struct fblock *m_fblock, int sd, struct sockaddr_in *addr){
229:     char in_buffer[4], data[TFTP_DATA_BLOCK], out_buffer[TFTP_MAX_DATA_MSG_SIZE];
230:     int block_n, rcv_block_n;
231:     int len, data_size, msglen, ret;
232:
233:     // init sequence number
234:     block_n = 1;
235:
236:     do{
237:         LOG(LOG_DEBUG, "Sending part %d", block_n);
238:
239:         if (m_fblock->remaining > TFTP_DATA_BLOCK)
```

```
240:     data_size = TFTP_DATA_BLOCK;
241:     else
242:         data_size = m_fblock->remaining;
243:
244:     if (data_size != 0)
245:         fblock_read(m_fblock, data);
246:
247:     LOG(LOG_DEBUG, "Part %d has size %d", block_n, data_size);
248:
249:     msglen = tftp_msg_get_size_data(data_size);
250:     tftp_msg_build_data(block_n, data, data_size, out_buffer);
251:
252:     // dump_buffer_hex(out_buffer, msglen);
253:
254:     len = sendto(sd, out_buffer, msglen, 0, (struct sockaddr*)addr, sizeof(*addr
));
255:
256:     if (len != msglen){
257:         return 1;
258:     }
259:
260:     LOG(LOG_DEBUG, "Waiting for ack");
261:
262:     ret = tftp_receive_ack(&rcv_block_n, in_buffer, sd, addr);
263:
264:     if (ret == 2){ //unexpected source
265:         continue;
266:     } else if (ret != 0){
267:         LOG(LOG_ERR, "Error receiving ack: %d", ret);
268:         return 2;
269:     }
270:
271:     if (rcv_block_n != block_n){
272:         LOG(LOG_ERR, "Received wrong block n: received %d != expected %d", rcv_blo
ck_n, block_n);
273:         return 3;
274:     }
275:
276:     block_n++;
277:
278: } while(data_size == TFTP_DATA_BLOCK);
279: return 0;
280: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Conversion functions from netascii to Unix standard ASCII.
6:  *
7:  * This library provides two functions to convert a file from netascii to Unix
8:  * standard ASCII and viceversa.
9:  * In particular, there are only two differences:
10:  * - 'LF' in Unix becomes 'CRLF' in netascii
11:  * - 'CR' in Unix becomes 'CRNUL' in netascii
12:  *
13:  * @see https://tools.ietf.org/html/rfc764
14:  */
15:
16:
17: #ifndef NETASCII
18: #define NETASCII
19:
20:
21: /**
22:  * Unix to netascii conversion.
23:  *
24:  * @param unix_filename the filename of the input Unix file
25:  * @param netascii_filename the filename of the output netascii file
26:  * @return
27:  * - 0 in case of success
28:  * - 1 in case of an error opening unix_filename file
29:  * - 2 in case of an error opening netascii_filename file
30:  * - 3 in case of an error writing to netascii_filename file
31:  */
32: int unix2netascii(char *unix_filename, char* netascii_filename);
33:
34: /**
35:  * Netascii to Unix conversion.
36:  *
37:  * @param netascii_filename the filename of the input netascii file
38:  * @param unix_filename the filename of the output Unix file
39:  * @return
40:  * - 0 in case of success
41:  * - 1 in case of an error opening unix_filename file
42:  * - 2 in case of an error opening netascii_filename file
43:  * - 3 in case of an error writing to unix_filename file
44:  * - 3 in case of bad formatted netascii
45:  */
46: int netascii2unix(char* netascii_filename, char *unix_filename);
47:
48:
49: #endif
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of netascii.h.
6:  *
7:  * @see netascii.h
8:  */
9:
10:
11: #include "include/netascii.h"
12: #include "include/logging.h"
13: #include <stdio.h>
14:
15:
16: /** LOG_LEVEL will be defined in another file */
17: extern const int LOG_LEVEL;
18:
19:
20: int unix2netascii(char *unix_filename, char* netascii_filename){
21:     FILE *unixf, *netasciif;
22:     char prev, tmp;
23:     int ret, result;
24:
25:     unixf = fopen(unix_filename, "r");
26:
27:     if (unixf == NULL){
28:         LOG(LOG_ERR, "Error opening file %s", unix_filename);
29:         return 1;
30:     }
31:
32:     netasciif = fopen(netascii_filename, "w");
33:
34:     if (unixf == NULL){
35:         LOG(LOG_ERR, "Error opening file %s", netascii_filename);
36:         return 2;
37:     }
38:
39:     prev = EOF;
40:
41:     while ((tmp = (char) fgetc(unixf)) != EOF){
42:         if (tmp == '\n' && prev != '\r'){ // LF -> CRLF
43:             ret = putc('\r', netasciif);
44:             if (ret == EOF)
45:                 break;
46:
47:             ret = putc('\n', netasciif);
48:             if (ret == EOF)
49:                 break;
50:
51:         } else if (tmp == '\r'){ // CR -> CRNUL
52:             char next = (char) fgetc(unixf);
53:             if (next != '\0')
54:                 ungetc(next, unixf);
55:
56:             ret = putc('\r', netasciif);
57:             if (ret == EOF)
58:                 break;
59:
60:             ret = putc('\0', netasciif);
61:             if (ret == EOF)
62:                 break;
63:         } else{
```

```
64:         ret = putc(tmp, netasciif);
65:         if (ret == EOF)
66:             break;
67:     }
68:
69:     prev = tmp;
70: }
71:
72: // Error writing to netasciif
73: if (ret == EOF){
74:     LOG(LOG_ERR, "Error writing to file %s", netascii_filename);
75:     result = 3;
76: } else{
77:     LOG(LOG_INFO, "Unix file %s converted to netascii file %s", unix_filename, n
etascii_filename);
78:     result = 0;
79: }
80:
81: fclose(unixf);
82: fclose(netasciif);
83:
84: return result;
85: }
86:
87: int netascii2unix(char* netascii_filename, char *unix_filename){
88:     FILE *unixf, *netasciif;
89:     char tmp;
90:     int ret;
91:     int result = 0;
92:
93:     unixf = fopen(unix_filename, "w");
94:
95:     if (unixf == NULL){
96:         LOG(LOG_ERR, "Error opening file %s", unix_filename);
97:         return 1;
98:     }
99:
100:    netasciif = fopen(netascii_filename, "r");
101:
102:    if (unixf == NULL){
103:        LOG(LOG_ERR, "Error opening file %s", netascii_filename);
104:        return 2;
105:    }
106:
107:    while ((tmp = (char) fgetc(netasciif)) != EOF){
108:        if (tmp == '\r'){ // CRLF -> LF ; CRNUL -> CR
109:            char next = (char) fgetc(netasciif);
110:            if (next == '\0'){ // CRNUL -> CR
111:                ret = putc('\r', unixf);
112:                if (ret == EOF)
113:                    break;
114:            } else if (next == '\n'){ // CRLF -> LF
115:                ret = putc('\n', unixf);
116:                if (ret == EOF)
117:                    break;
118:            } else if (next == EOF) { // bad format
119:                LOG(LOG_ERR, "Bad formatted netascii: unexpected EOF after CR");
120:                result = 4;
121:                break;
122:            } else{ // bad format
123:                LOG(LOG_ERR, "Bad formatted netascii: unexpected 0x%x after CR", next);
124:                result = 4;
125:                break;
```

```
126:     }
127:   } else{
128:
129:     // nothing else needs to be done!
130:
131:     ret = putc(tmp, unixf);
132:     if (ret == EOF)
133:       break;
134:   }
135: }
136:
137: if (result == 0){
138:   // Error writing to unixf
139:   if (ret == EOF){
140:     LOG(LOG_ERR, "Error writing to file %s", unix_filename);
141:     result = 3;
142:   } else{
143:     LOG(LOG_INFO, "Netascii file %s converted to Unix file %s", netascii_filename, unix_filename);
144:     result = 0;
145:   }
146: } // otherwise there was an error (4 or 5) and result was already set
147:
148: fclose(unixf);
149: fclose(netasciif);
150:
151: return result;
152: }
```



```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of the TFTP client that can only make read requests.
6:  */
7:
8:
9: #include "include/logging.h"
10: #include "include/tftp_msgs.h"
11: #include "include/tftp.h"
12: #include "include/fblock.h"
13: #include "include/inet_utils.h"
14: #include "include/debug_utils.h"
15: #include "include/netascii.h"
16: #include <arpa/inet.h>
17: #include <sys/types.h>
18: #include <sys/socket.h>
19: #include <netinet/in.h>
20: #include <string.h>
21: #include <stdio.h>
22: #include <stdlib.h>
23: #include <time.h>
24:
25: /** Defining LOG_LEVEL for tftp_client executable */
26: const int LOG_LEVEL = LOG_WARN;
27:
28:
29: /** max stdin line length */
30: #define READ_BUFFER_SIZE 80
31:
32: /** Maximum number of arguments for commands */
33: #define MAX_ARGS 3
34:
35: /** String for txt */
36: #define MODE_TXT "txt"
37:
38: /** String for bin*/
39: #define MODE_BIN "bin"
40:
41:
42: /**
43:  * Global transfer_mode variable for storing user chosen transfer mode string.
44:  *
45:  * @see MODE_TXT
46:  * @see MODE_BIN
47:  */
48: char* transfer_mode;
49:
50:
51: /**
52:  * Splits a string at each delim.
53:  *
54:  * Trailing LF will be removed. Consecutive delimiters will be considered as one
55:  *
56:  * @param line [in] the string to split
57:  * @param delim [in] the delimiter
58:  * @param max_argc [in] maximum number of parts to split the line into
59:  * @param argc [out] counts of the parts the line is split into
60:  * @param argv [out] array of parts the line is split into
61:  */
62: void split_string(char* line, char* delim, int max_argc, int *argc, char **argv)
```

```
{
63:  char *ptr;
64:  int len;/**
65:  * Prints command usage information.
66:  */
67:  char *pos;
68:
69:  // remove trailing LF
70:  if ((pos=strchr(line, '\n')) != NULL)
71:      *pos = '\0';
72:
73:  // init argc
74:  *argc = 0;
75:
76:  // tokenize string
77:  ptr = strtok(line, delim);
78:
79:  while(ptr != NULL && *argc <= max_argc){
80:      len = strlen(ptr);
81:
82:      if (len == 0)
83:          continue;
84:
85:      LOG(LOG_DEBUG, "arg[%d] = '%s'", *argc, ptr);
86:
87:      argv[*argc] = malloc(strlen(ptr)+1);
88:      strcpy(argv[*argc], ptr);
89:
90:      ptr = strtok(NULL, delim);
91:      (*argc)++;
92:  }
93: }
94:
95: /**
96:  * Prints command usage information.
97:  */
98: void print_help(){
99:     printf("Usage: ./tftp_client SERVER_IP SERVER_PORT\n");
100:    printf("Example: ./tftp_client 127.0.0.1 69");
101: }
102:
103: /**
104:  * Handles !help command, printing information about available commands.
105:  */
106: void cmd_help(){
107:     printf("Sono disponibili i seguenti comandi:\n");
108:     printf("!help --> mostra l'elenco dei comandi disponibili\n");
109:     printf("!mode {txt|bin} --> imposta il modo di trasferimento dei file (testo o
binario)\n");
110:     printf("!get filename nome_locale --> richiede al server il nome del file <fil
ename> e lo salva localmente con il nome <nome_locale>\n");
111:     printf("!quit --> termina il client\n");
112: }
113:
114: /**
115:  * Handles !mode command, changing transfer_mode to either bin or text.
116:  *
117:  * @see transfer_mode
118:  */
119: void cmd_mode(char* new_mode){
120:     if (strcmp(new_mode, MODE_TXT) == 0){
121:         transfer_mode = TFTP_STR_NETASCII;
122:         printf("Modo di trasferimento testo configurato\n");
```

```
123:     } else if (strcmp(new_mode, MODE_BIN) == 0){
124:         transfer_mode = TFTP_STR_OCTET;
125:         printf("Modo di trasferimento binario configurato\n");
126:     } else{
127:         printf("Modo di traferimento sconosciuto: %s. Modi disponibili: txt, bin\n",
new_mode);
128:     }
129: }
130:
131: /**
132:  * Handles !get command, reading file from server.
133:  */
134: int cmd_get(char* remote_filename, char* local_filename, char* sv_ip, int sv_por
t){
135:     struct sockaddr_in my_addr, sv_addr;
136:     int sd;
137:     int ret, tid, result;
138:     struct fblock m_fblock;
139:     char *tmp_filename;
140:
141:     LOG(LOG_INFO, "Initializing...\n");
142:
143:     sd = socket(AF_INET, SOCK_DGRAM, 0);
144:     if (strcmp(transfer_mode, TFTP_STR_OCTET) == 0)
145:         m_fblock = fblock_open(local_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_
MODE_BINARY);
146:     else if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
147:         tmp_filename = malloc(strlen(local_filename)+5);
148:         strcpy(tmp_filename, local_filename);
149:         strcat(tmp_filename, ".tmp");
150:         m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK, FBLOCK_WRITE|FBLOCK_MO
DE_TEXT);
151:     }else
152:         return 2;
153:
154:     LOG(LOG_INFO, "Opening socket...");
155:
156:     sv_addr = make_sv_sockaddr_in(sv_ip, sv_port);
157:     my_addr = make_my_sockaddr_in(0);
158:     tid = bind_random_port(sd, &my_addr);
159:     if (tid == 0){
160:         LOG(LOG_ERR, "Error while binding to random port");
161:         perror("Could not bind to random port:");
162:         fblock_close(&m_fblock);
163:         return 1;
164:     } else
165:         LOG(LOG_INFO, "Bound to port %d", tid);
166:
167:     printf("Richiesta file %s (%s) al server in corso.\n", remote_filename, transf
er_mode);
168:
169:     ret = tftp_send_rrq(remote_filename, transfer_mode, sd, &sv_addr);
170:     if (ret != 0){
171:         fblock_close(&m_fblock);
172:         return 8+ret;
173:     }
174:
175:     printf("Trasferimento file in corso.\n");
176:
177:     ret = tftp_receive_file(&m_fblock, sd, &sv_addr);
178:
179:
180:     if (ret == 1){        // File not found
```

```
181:     printf("File non trovato.\n");
182:     result = 0;
183: } else if (ret != 0){
184:     LOG(LOG_ERR, "Error while receiving file!");
185:     result = 16+ret;
186: } else{
187:     int n_blocks = (m_fblock.written + m_fblock.block_size - 1)/m_fblock.block_s
ize;
188:     printf("Trasferimento completato (%d/%d blocchi)\n", n_blocks, n_blocks);
189:     printf("Salvataggio %s completato.\n", local_filename);
190:
191:     result = 0;
192: }
193:
194: fblock_close(&m_fblock);
195: if (strcmp(transfer_mode, TFTP_STR_NETASCII) == 0){
196:     netascii2unix(tmp_filename, local_filename);
197:     remove(tmp_filename);
198:     free(tmp_filename);
199: }
200:
201: return result;
202:
203: }
204:
205: /**
206:  * Handles !quit command.
207:  */
208: void cmd_quit(){
209:     printf("Client terminato con successo\n");
210:     exit(0);
211: }
212:
213: /** Main */
214: int main(int argc, char** argv){
215:     char* sv_ip;
216:     short int sv_port;
217:     int ret, i;
218:     char read_buffer[READ_BUFFER_SIZE];
219:     int cmd_argc;
220:     char *cmd_argv[MAX_ARGS];
221:
222:     //init random seed
223:     srand(time(NULL));
224:
225:     // default mode = bin
226:     transfer_mode = TFTP_STR_OCTET;
227:
228:     if (argc != 3){
229:         print_help();
230:         return 1;
231:     }
232:
233:     // TODO: check args
234:     sv_ip = argv[1];
235:     sv_port = atoi(argv[2]);
236:
237:     while(1){
238:         printf("> ");
239:         fflush(stdout); // flush stdout buffer
240:         fgets(read_buffer, READ_BUFFER_SIZE, stdin);
241:         split_string(read_buffer, " ", MAX_ARGS, &cmd_argc, cmd_argv);
242:
```

```
243:     if (cmd_argc == 0){
244:         printf("Comando non riconosciuto : ''\n");
245:         cmd_help();
246:     } else{
247:         if (strcmp(cmd_argv[0], "!mode") == 0){
248:             if (cmd_argc == 2)
249:                 cmd_mode(cmd_argv[1]);
250:             else
251:                 printf("Il comando richiede un solo argomento: bin o txt\n");
252:         } else if (strcmp(cmd_argv[0], "!get") == 0){
253:             if (cmd_argc == 3){
254:                 ret = cmd_get(cmd_argv[1], cmd_argv[2], sv_ip, sv_port);
255:                 LOG(LOG_DEBUG, "cmd_get returned value: %d", ret);
256:             } else{
257:                 printf("Il comando richiede due argomenti: <filename> e <nome_locale>
\n");
258:             }
259:         } else if (strcmp(cmd_argv[0], "!quit") == 0){
260:             if (cmd_argc == 1){
261:                 cmd_quit();
262:             } else{
263:                 printf("Il comando non richiede argomenti\n");
264:             }
265:         } else if (strcmp(cmd_argv[0], "!help") == 0){
266:             if (cmd_argc == 1){
267:                 cmd_help();
268:             } else{
269:                 printf("Il comando non richiede argomenti\n");
270:             }
271:         } else {
272:             printf("Comando non riconosciuto : '%s'\n", cmd_argv[0]);
273:             cmd_help();
274:         }
275:     }
276:
277:     // Free malloc'ed strings
278:     for(i = 0; i < cmd_argc; i++)
279:         free(cmd_argv[i]);
280: }
281:
282: return 0;
283: }
```

```
1: /**
2:  * @file
3:  * @author Riccardo Mancini
4:  *
5:  * @brief Implementation of the TFTP server that can only handle read requests.
6:  *
7:  * The server is multiprocessed, with each process handling one request.
8:  */
9:
10:
11: #define _GNU_SOURCE
12: #include <stdlib.h>
13:
14: #include "include/tftp_msgs.h"
15: #include "include/tftp.h"
16: #include "include/fblock.h"
17: #include "include/inet_utils.h"
18: #include "include/debug_utils.h"
19: #include "include/netascii.h"
20: #include <arpa/inet.h>
21: #include <sys/types.h>
22: #include <sys/socket.h>
23: #include <netinet/in.h>
24: #include <string.h>
25: #include <strings.h>
26: #include <stdio.h>
27: #include "include/logging.h"
28: #include <sys/types.h>
29: #include <unistd.h>
30: #include <time.h>
31: #include <linux/limits.h>
32: #include <libgen.h>
33:
34:
35: /** Defining LOG_LEVEL for tftp_server executable */
36: const int LOG_LEVEL = LOG_INFO;
37:
38:
39: /** Maximum length for a RRQ message */
40: #define MAX_MSG_LEN TFTP_MAX_MODE_LEN+TFTP_MAX_FILENAME_LEN+4
41:
42:
43: /** Finds longest common prefix length of strings str1 and str2 */
44: int strlcp1(const char* str1, const char* str2){
45:     int n;
46:     for (n = 0; str1[n] != '\0' && str2[n] != '\0' && str1[n] == str2[n]; n++);
47:     return n;
48: }
49:
50: /**
51:  * Check whether file is inside dir.
52:  *
53:  * @param path file absolute path (can include .. and . and multiple /)
54:  * @param dir directory real path (can't include .. and . and multiple /)
55:  * @return 1 if true, 0 otherwise
56:  *
57:  * @see realpath
58:  */
59: int path_inside_dir(char* path, char* dir){
60:     char *parent, *orig_parent, *ret_realpath;
61:     char parent_realpath[PATH_MAX];
62:     int result;
63: }
```

```
64: orig_parent = parent = malloc(strlen(path) + 1);
65: strcpy(parent, path);
66:
67: do{
68:     parent = dirname(parent);
69:     ret_realpath = realpath(parent, parent_realpath);
70: } while (ret_realpath == NULL);
71:
72: if (strlcpy(parent_realpath, dir) < strlen(dir))
73:     result = 0;
74: else
75:     result = 1;
76:
77: free(orig_parent);
78: return result;
79: }
80:
81: /**
82:  * Prints command usage information.
83:  */
84: void print_help(){
85:     printf("Usage: ./tftp_server LISTEN_PORT FILES_DIR\n");
86:     printf("Example: ./tftp_server 69 .\n");
87: }
88:
89: /**
90:  * Sends file to a client.
91:  */
92: int send_file(char* filename, char* mode, struct sockaddr_in *cl_addr){
93:     struct sockaddr_in my_addr;
94:     int sd;
95:     int ret, tid, result;
96:     struct fblock m_fblock;
97:     char *tmp_filename;
98:
99:     sd = socket(AF_INET, SOCK_DGRAM, 0);
100:    my_addr = make_my_sockaddr_in(0);
101:    tid = bind_random_port(sd, &my_addr);
102:    if (tid == 0){
103:        LOG(LOG_ERR, "Could not bind to random port");
104:        perror("Could not bind to random port:");
105:        fblock_close(&m_fblock);
106:        return 4;
107:    } else
108:        LOG(LOG_INFO, "Bound to port %d", tid);
109:
110:    if (strcasecmp(mode, TFTP_STR_OCTET) == 0){
111:        m_fblock = fblock_open(filename, TFTP_DATA_BLOCK, FBLOCK_READ|FBLOCK_MODE_BINARY);
112:    } else if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
113:        tmp_filename = malloc(strlen(filename)+5);
114:        strcpy(tmp_filename, filename);
115:        strcat(tmp_filename, ".tmp");
116:        ret = unix2netascii(filename, tmp_filename);
117:        if (ret != 0){
118:            LOG(LOG_ERR, "Error converting text file to netascii: %d", ret);
119:            return 3;
120:        }
121:        m_fblock = fblock_open(tmp_filename, TFTP_DATA_BLOCK, FBLOCK_READ|FBLOCK_MODE_E_TEXT);
122:    } else{
123:        LOG(LOG_ERR, "Unknown mode: %s", mode);
124:        return 2;
```

```
125:     }
126:
127:     if (m_fblock.file == NULL){
128:         LOG(LOG_WARN, "Error opening file. Not found?");
129:         tftp_send_error(1, "File not found.", sd, cl_addr);
130:         result = 1;
131:     } else{
132:         LOG(LOG_INFO, "Sending file...");
133:         ret = tftp_send_file(&m_fblock, sd, cl_addr);
134:
135:         if (ret != 0){
136:             LOG(LOG_ERR, "Error sending file: %d", ret);
137:             result = 16+ret;
138:         } else{
139:             LOG(LOG_INFO, "File sent successfully");
140:             result = 0;
141:         }
142:     }
143:
144:     fblock_close(&m_fblock);
145:
146:     if (strcasecmp(mode, TFTP_STR_NETASCII) == 0){
147:         LOG(LOG_DEBUG, "Removing temp file %s", tmp_filename);
148:         remove(tmp_filename);
149:         free(tmp_filename);
150:     }
151:
152:     return result;
153: }
154:
155: /** Main */
156: int main(int argc, char** argv){
157:     short int my_port;
158:     char *dir_rel_path;
159:     char *ret_realpath;
160:     char dir_realpath[PATH_MAX];
161:     int ret, type, len;
162:     char in_buffer[MAX_MSG_LEN];
163:     unsigned int addrlen;
164:     int sd;
165:     struct sockaddr_in my_addr, cl_addr;
166:     int pid;
167:     char addr_str[MAX_SOCKADDR_STR_LEN];
168:
169:     if (argc != 3){
170:         print_help();
171:         return 1;
172:     }
173:
174:     my_port = atoi(argv[1]);
175:     dir_rel_path = argv[2];
176:
177:     ret_realpath = realpath(dir_rel_path, dir_realpath);
178:     if (ret_realpath == NULL){
179:         LOG(LOG_FATAL, "Directory not found: %s", dir_rel_path);
180:         return 1;
181:     }
182:
183:     addrlen = sizeof(cl_addr);
184:
185:     sd = socket(AF_INET, SOCK_DGRAM, 0);
186:     my_addr = make_my_sockaddr_in(my_port);
187:     ret = bind(sd, (struct sockaddr*) &my_addr, sizeof(my_addr));
```



```
248:         }
249:
250:         LOG(LOG_INFO, "User wants to read file %s in mode %s", filename, mode);
251:
252:         ret = send_file(file_realpath, mode, &cl_addr);
253:         if (ret != 0)
254:             LOG(LOG_WARN, "Write terminated with an error: %d", ret);
255:         break; // child process exits loop
256:     }
257: } else{
258:     LOG(LOG_WARN, "Wrong op code: %d", type);
259:     tftp_send_error(4, "Illegal TFTP operation.", sd, &cl_addr);
260:     // main process continues loop
261: }
262: }
263:
264: LOG(LOG_INFO, "Exiting process %d", (int) getpid());
265: return 0;
266: }
```