Onda Connection Library
Source Code

Morten Fischer Rasmussen

April 24, 2013

Contents

1 MATLAB routines

1.1 Connection

```
%
%
% onda_initialize([host_addr])
%
% Initializes the connection to the Onda system.
%
% host_addr: string containing the IP-address of the host. Defaults ↩
    to 10.59.44.100.
%
%
% By MFR,
% Version 1.0, 2012-04-15, Init version.
% Version 1.1, 2013-04-15, Added check of whether IP address is valid
%




function onda_initialize(host_addr)

IPv4_pattern = '^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d↩
    |25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d↩
    |25[0-5])$';

if nargin == 0
    onda_lib('init_connection', '10.59.44.100');
else
    if ~ischar(host_addr)
        error('The host address must be a string');
    else
        is_valid_addreess = ~isempty(regexp(host_addr, IPv4_pattern))
        if ~is_valid_addreess
            error('The host address appears not to be a valid IPv4 ↩
                address');
        end
    end
    onda_lib('init_connection', host_addr);
end

%↩
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%↩

%%% onda_initialise.m ends here
```

Listing 1.1: onda_initialize.m

```
%
%
% onda_terminate()
```

```
%
% $Id: onda_terminate.m 5 2012-04-16 13:31:11Z mf $
%
% By MFR, 2012-04-15
% Version 1.0 Init version.
%




function onda_terminate

onda_lib('terminate_connection');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% onda_initialise.m ends here
```

Listing 1.2: onda_terminate.m

1.2 Movement

```
%
% onda_move_absolute(dist [,axis])
%
% dist:  distance in meters to move in absolute coordinates.
%        Must be a vector with three elements representing
%        [dist_x, dist_y, dist_z]. When a scaler is given, ''axis''
%        must also be set.
%
% axis:  Optional to use. Can be a vector or scaler. Choses which
%        axis to move. 0=x, 1=y, 2=z, 3=rot1, 4=rot2.
%
% By MFR,
% Version 1.0, 2012-04-15, Init version.
% Version 1.1, 2013-04-17, Changed to accept single vector argument.
% Version 1.2, 2013-04-22, Re-added ''axis'', now as an optional ↩
    argument
%
%


function onda_move_absolute(dist, axis)

if nargin == 1
    % assume a vector for movement in (x,y,z) is given
    axis = [0 1 2]; %x,y,z
end
%make sure all dist-elements are numbers
if ~isnumeric(dist)
    error('dist must contain only numbers.')
end
%make sure all axis-elements are numbers
if ~isnumeric(axis)
    error('dist must contain only numbers.')
end
% make sure dist and axis have same length
if length(dist) ~= length(axis)
    error(['axis and dist must contain the same number of ' ...
           'elements.'])
end


% Send the commnand to the Onda system
for idx = 1:length(axis)
    if axis(idx) >= 0 && axis(idx) <= 2
        dist = dist*1000;  %m to cm convertion
    end

    onda_lib('move_absolute', axis(idx), dist(idx));
end
```

Listing 1.3: onda_move_absolute.m

```
%
% onda_move_relative(dist [,axis])
%
% dist:  distance in meters to move relative to current position.
%        It must be a vector with three elements representing:
%        [dist_x, dist_y, dist_z]. When a scaler is given, ''axis''
%        must also be set.
```

```
%
% axis:  Optional to use. Can be a vector or scaler. Choses which
%        axis to move. 0=x, 1=y, 2=z, 3=rot1, 4=rot2.
%
% By MFR,
% Version 1.0, 2012-04-15, Init version.
% Version 1.1, 2013-04-17, Changed to accept single vector argument.
% Version 1.2, 2013-04-22, Re-added ''axis'', now as an optional ↩
    argument
%
%


function onda_move_relative(dist, axis)

if nargin == 1
    % assume a vector for movement in (x,y,z) is given
    axis = [0 1 2]; %x,y,z
end
%make sure all dist-elements are numbers
if ~isnumeric(dist)
    error('dist must contain only numbers.')
end
%make sure all axis-elements are numbers
if ~isnumeric(axis)
    error('dist must contain only numbers.')
end
% make sure dist and axis have same length
if length(dist) ~= length(axis)
    error(['axis and dist must contain the same number of ' ...
           'elements.'])
end


% Send the commnand to the Onda system
for idx = 1:length(axis)
    if axis(idx) >= 0 && axis(idx) <= 2
        dist = dist*1000;  %m to cm convertion
    end

    onda_lib('move_relative', axis(idx), dist(idx));
end
```

Listing 1.4: onda_move_relative.m

```
%
%
% onda_set_position(position)
%
% position:  Vector with three elements setting the position of the
%            x-, y- and z-axis. In units of meters.
%
%
% By MFR, 2012-04-15
% Version 1.0 Init version.
% Version 1.1, 2013-04-17, Changed to set the position for x,y,z axes↩
    , always.
%


function onda_set_position(position)
```

```
%make sure all position-elements are numbers
if ~isnumeric(position)
    error('dist must contain only numbers.');
end
if length(position) ~= 3
    error('''position'' must be a vector with 3 elements.');
end

axis = [0 1 2]; %x,y,z
% Send the comnmand to the Onda system
for idx = 1:length(axis)
    if axis(idx) >= 0 && axis(idx) <= 2
        position = position*1000;  %m to mm convertion
    end

    onda_lib('set_position', axis(idx), position(idx));
end
```

Listing 1.5: onda_set_position.m

```
%
%
% position = onda_get_position
%
% By MFR,
% Version 1.0, 2012-04-15, Init version.
% Version 1.1, 2013-04-17, Changed to return position for x,y,z axes,↩
     always.
%


function position = onda_get_position


pos_x = onda_lib('get_position', 0);
pos_y = onda_lib('get_position', 1);
pos_z = onda_lib('get_position', 2);

position = [pos_x pos_y pos_z]';
position = position/1000;  %mm to m convertion
```

Listing 1.6: onda_get_position.m

1.3 Movement

```
%
%
% onda_set_high_limit (axis, limit)
%
% axis:       integer or strinf choosing which axis to use. (usually ↩
    0,1,2 or 'x','y','z').
% limit:      limit in meters on the chosen axis.
%
%
% By MFR,
% Version 1.0, 2013−04−15, Init version.
% Version 1.1, 2013−04−15, Now also accepts strings to identify axes.
%


function onda_set_high_limit(axis, limit)

if nargin ~= 2
    error('This function requires two inputs.');
end

limit = limit*1000;  %m to mm convertion

if strcmp(axis,'x') || axis == 0
    onda_lib('set_high_limit', 0, limit);
elseif strcmp(axis,'y') || axis == 1
    onda_lib('set_high_limit', 1, limit);
elseif strcmp(axis,'z') || axis == 2
    onda_lib('set_high_limit', 2, limit);
else
    warning('Did not identify the axis.')
end
```

Listing 1.7: onda_set_high_limit.m

```
%
%
% onda_get_high_limit(axis)
%
% axis:       integer choosing which axis to use. (0,1,2 or 'x','y','z↩
    ')
%
%
% By MFR,
% Version 1.0, 2013−04−15, Init version.
% Version 1.1, 2013−04−15, Now also accepts strings to identify axes.
%


function limit = onda_get_high_limit(axis)

if nargin ~= 1
    error('This function requires one input.');
end


if strcmp(axis,'x') || axis == 0
    limit = onda_lib('get_high_limit', 0);
elseif strcmp(axis,'y') || axis == 1
```

```
    limit = onda_lib('get_high_limit', 1);
elseif strcmp(axis,'z') || axis == 2
    limit = onda_lib('get_high_limit', 2);
else
    warning('Did not identify the axis.')
end

limit = limit/1000;  %mm to m convertion
```

Listing 1.8: onda_get_high_limit.m

```
%
%
% onda_set_low_limit(axis, limit)
%
% axis:      integer choosing which axis to use. (0,1,2 or 'x','y','z↩
    ')
% limit:     position limit in meters on the chosen axis.
%
%
% By MFR,
% Version 1.0, 2013-04-15, Init version.
% Version 1.1, 2013-04-15, Now also accepts strings to identify axes.
%


function onda_set_low_limit(axis, limit)

if nargin ~= 2
    error('This function requires two inputs.');
end

limit = limit*1000;  %m to mm convertion

if strcmp(axis,'x') || axis == 0
    onda_lib('set_low_limit', 0, limit);
elseif strcmp(axis,'y') || axis == 1
    onda_lib('set_low_limit', 1, limit);
elseif strcmp(axis,'z') || axis == 2
    onda_lib('set_low_limit', 2, limit);
else
    warning('Did not identify the axis.')
end
```

Listing 1.9: onda_set_low_limit.m

```
%
%
% onda_get_low_limit(axis)
%
% axis:      integer choosing which axis to use. (0,1,2 or 'x','y','z↩
    ')
%
%
% By MFR,
% Version 1.0, 2013-04-15, Init version.
% Version 1.1, 2013-04-15, Now also accepts strings to identify axes.
%


function onda_get_low_limit(axis, limit)
```

```
if nargin ~= 2
    error('This function requires two inputs.');
end

limit = limit*1000;  %m to mm convertion

if strcmp(axis,'x') || axis == 0
    onda_lib('get_low_limit', 0);
elseif strcmp(axis,'y') || axis == 1
    onda_lib('get_low_limit', 1);
elseif strcmp(axis,'z') || axis == 2
    onda_lib('get_low_limit', 2);
else
    warning('Did not identify the axis.')
end
```

Listing 1.10: onda_get_low_limit.m

2

C++ Library

2.1 onda_lib

```
#ifndef C_PRINT_H
#define C_PRINT_H

#ifndef MATLAB_MEX_FILE
#define err_printf    printf
#define info_printf   printf
#else
#include "mex.h"
#define info_printf   mexPrintf
#define err_printf    mexErrMsgTxt
#endif

#endif
```

Listing 2.1: err_printf.h

```

#ifndef     ONDA_POSITIONER_H_
# define    ONDA_POSITIONER_H_


 int PositionerMoveRel(int axis, float value);
 int PositionerMoveAbs(int axis, float value);
 int SetPosition(int axis, float value);
 int GetPosition(int axis, float* result);

 int SetPositionerLowLimit(int axis, float value);
 int GetPositionerLowLimit(int axis, float* result);
 int SetPositionerHighLimit(int axis, float  value);
 int GetPositionerHighLimit(int axis, float* result);
 int GetPositionerStepsPerSecond(int axis, int* result);
 int GetPositionerMinStepsPerSecond(int axis, int* result);



#endif /* !ONDA_POSITIONER_H_*/
```

Listing 2.2: onda_positioner.h

```
/**
 * onda_lib.cpp
 * This file makes the connection between MATLAB and the C++ library ↩
    which
 * connects to the Onda system.
 *
 */

```

```

#include <cstdio>
#include <cstring>
#include <mex.h>
#include "onda_positioner.h"
#include "tcp.h"
#include "err_printf.h"

#define HOST_PORT 49999

void
mexFunction(int nlhs,mxArray *plhs[], int nrhs,const mxArray *prhs[])
{

  int axis = 0, retval;
  float value = 0;
  float* value_p = NULL;

  /* string buffers */
  enum {host_addr_len  = 20};
  enum {func_str_len   = 30};
  char host_addr[host_addr_len];
  char func_str[func_str_len];


  /* output pointers */
  mxArray* val_p = NULL;

  if (nrhs < 1) err_printf("Minimum one input required.");
  if (nlhs > 1) err_printf("Maximum one output argument allowed.");

  /* get input vars */
  mxGetString(prhs[0], func_str, func_str_len);
  if (nrhs > 1)  axis  = (int)mxGetScalar(prhs[1]);
  if (nrhs > 2)  value = (float)mxGetScalar(prhs[2]);

  /* set output pointer */
  val_p = plhs[0];
  val_p = mxCreateDoubleMatrix(1, 1, mxREAL);




  /* Get Version of this library */
  // TODO: Get define the version during compiling.
  if (strcmp("version", func_str) == 0){
      info_printf("Onda connection library, version 1.3, 2013-04-22.\↩
          n");
  }


  /* move relative */
  else if (strcmp("move_relative", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");

    retval = PositionerMoveRel(axis, value);
    if (retval){
      err_printf("Error in Positioner move relative\n");
      return;
    }
  }


  /* move absolute */
  else if (strcmp("move_absolute", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");

    retval = PositionerMoveAbs(axis, value);
    if (retval) err_printf("Error in positioner move absolut.");
  }


  /* set position */
  else if (strcmp("set_position", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");

    retval = SetPosition(axis, value);
    if (retval) err_printf("Error in Set position.");
  }


  /* get position */
  else if (strcmp("get_position", func_str) == 0){
    if (nrhs != 2) err_printf("This function requires two input ↩
        arguments.");

    retval = GetPosition(axis, value_p);
    if (retval) err_printf("Error in Get position.");
    info_printf("Onda pos: %f\n", *value_p);
    /* set output */
    mxGetPr(val_p)[0] = *value_p;
  }


  /* Get low limit */
  else if (strcmp("get_low_limit", func_str) == 0){
    if (nrhs != 2) err_printf("This function requires two input ↩
        arguments.");

    retval = GetPositionerLowLimit(axis, value_p);
    if (retval) err_printf("Onda: Error in Get low limit.");
    /* set output */
    mxGetPr(val_p)[0] = *value_p;
  }


  /* set low limit */
  else if (strcmp("set_low_limit", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");

    retval = SetPositionerLowLimit(axis, value);
    if (retval) err_printf("Onda: Error in Set low limit.");
  }


  /* Get high limit */
  else if (strcmp("get_high_limit", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");
    retval = GetPositionerHighLimit(axis, value_p);
    if (retval) err_printf("Error in Get high limit.");
    info_printf("Onda: Pos low limit: %f\n", *value_p);
    /* set output */
    mxGetPr(val_p)[0] = *value_p;
  }


  /* Set high limit */
  else if (strcmp("set_high_limit", func_str) == 0){
    if (nrhs != 3) err_printf("This function requires three input ↩
        arguments.");

    retval = SetPositionerHighLimit(axis, value);
    if (retval) err_printf("Error in Set high limit.");
  }


  /* Init connection */
  else if (strcmp("init_connection", func_str) == 0){
    if (nrhs != 2) err_printf("This function requires two input ↩
        arguments.");

    mxGetString(prhs[1], host_addr, host_addr_len);
    retval = tcp_init (host_addr, HOST_PORT);
    if (retval)
      err_printf("Onda: Could not initialise connection.\n");
    else
      info_printf("Onda: connection initialised.\n");

    /* automatic close connection when MATLAB clears or exits */
    mexAtExit(tcp_auto_term);
  }


  /* terminate connection */
  else if (strcmp("terminate_connection", func_str) == 0){
    retval = tcp_term ();
    if (retval)
      err_printf("Onda: Could not terminate connection.\n");
    else
      info_printf("Onda connection terminated.\n");
  }





  /* CMD not found*/
  else {
    err_printf("Command not found.");
  }
}
```

Listing 2.3: onda_lib.cpp

2.2 TCP connection

```
/*
** tcp_lib.h
**
** Made by Morten Fischer Rasmussen
** Login   <mf@mf-black>
**
** Started on  Wed Jul 30 20:21:19 2008 Morten Fischer Rasmussen
** Last update Wed Jul 30 20:21:19 2008 Morten Fischer Rasmussen
*/

#ifndef     TCP_H_
# define    TCP_H_

#define ONDA_NOT_CONNECTED 0
#define ONDA_CONNECTED     1

/* struct contraining all connection information */
typedef struct connection
{
  char* name;
  int buf_size;
  char *rx_buf;
  char *tx_buf;
  char* addr;
  unsigned int port;
  int connected;
  int sock;
}  connection_t ;

/* typedef struct connection Connection; */


/* prototypes */
extern int tcp_init (const char* host_addr, const int host_port);
extern int tcp_term (void);
extern void tcp_auto_term (void);
extern int tcp_tx (const char* send_str);
extern int tcp_rx (char** data_ptr);
extern int tcp_query (const char* send_str, char** data_ptr);
//extern int tcp_clear_rx_buff (void);



/* error numbers */

//const char *ERR_RCV_TIMEOUT_STR ="Connection timed out. No data was↩
     received.\n";
enum {ERR_RCV_TIMEOUT  = -10};
enum {ERR_RCV_SYS_ERR = -11};
enum {ERR_RCV_NO_DATA = -12};
enum {ERR_RCV_TIMOUT2  = -13};


#endif      /* !TCP_H_ */
```

Listing 2.4: tcp.h

```
/*
** tcp.cpp
```

```
**
** This file implements the TCP connection to the Onda system.
** It was originally written in C for the SARUS project.
**
** $Id: tcp.cpp 6 2012-04-16 15:40:01Z mf $
**
** Made by (Morten Fischer Rasmussen)
** Login   <mf@mf-black>
**
** Started on  Wed Jul 30 20:22:23 2008 Morten Fischer Rasmussen
** Modified for Onda on Wed Feb 22 12:27:10 2012 Morten Fischer ↩
    Rasmussen
*/

#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>  /* provides close () */
#include <netinet/in.h>
/* #include <fcntl.h> */
#include <errno.h>
#include <sys/select.h>
#include <sys/time.h>

#include "tcp.h"
#include "err_printf.h"


#define ONDA_BUFF_SIZE 9096
#define ONDA_PORT 49999


/* input/output buffers */
static char rx_buf[ONDA_BUFF_SIZE];
static char tx_buf[ONDA_BUFF_SIZE];
static char onda_addr[30];


static connection_t onda_st = {(char*)"Onda",
              ONDA_BUFF_SIZE,
              rx_buf,
              tx_buf,
              onda_addr,
              ONDA_PORT,
              ONDA_NOT_CONNECTED,
              0};



/**
 *  Clears the local buffer and system buffer
 */
static int
tcp_clear_rx_buff (void)
{
    int retval;
    int nfds = onda_st.sock + 1;
    fd_set read_fd;
    struct timeval timeout;
    /* make sure pointer does not contain random value */
    // *data_ptr = NULL;

    if (onda_st.connected != ONDA_CONNECTED){
  err_printf ("Error on clearing buffer: Cannot receive on non-↩
      existing connection.\n");
  return -1;
    }

    /* clear the struct */
    FD_ZERO (&read_fd);
    /* add the socket to the struct/list */
    FD_SET (onda_st.sock, &read_fd);
    /* set timeout to 0.001 sec */
    timeout.tv_sec = 0;
    timeout.tv_usec = 1000;
    /* Wait until data is available or timeout has passed */
    retval = select (nfds, &read_fd, NULL, NULL, &timeout);

    if (retval > 0) /* data present in buffer */
  {
      /* clear buffer */
      /* data available -> receive it */
      recv (onda_st.sock, onda_st.rx_buf, onda_st.buf_size -1, 0);
  }

    /* Reset local buffer */
    rx_buf [0] = '\0';

    return 0;
}




/**
 * Creates a socket that uses TCP/IP and connects the destination adr↩
     +port
 * Input:   -struct containing all connection info
 *          -pointer to  a char buffer where error messages are ↩
     printed
 *
 * Returns:  0 on success
 *          -1 on failure
 */
int
tcp_init (const char *addr_host, const int port_host)
{

    if (onda_st.connected == ONDA_CONNECTED)
  {
      err_printf("Onda: Connection already initialised.\n");
      return -1;
  }
    sprintf(onda_st.addr, "%s", addr_host);
    onda_st.port = port_host;
    /* clear buffers */
    rx_buf[0] = '\0';
    tx_buf[0] = '\0';


    struct sockaddr_in dst_server;
    if (onda_st.connected != ONDA_NOT_CONNECTED){
  info_printf ("Onda interface err.: Can not initialize an already ↩
      excisting connection.\n",
          onda_st.name);
  return −1;
    }


    /* input validation */
    if (onda_st.port > 65535){
  info_printf ("Failed to create socket for %s: maximum value of ↩
      destination port is 65535\n", onda_st.name);
  return −1;
    }
    /* We assume that destination adresses are always valid (this is ↩
        not necessary true) */
    /* Create the IP/TCP socket */
    if ((onda_st.sock = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)) <↩
        0){
  info_printf ("Failed to create socket for %s.\nSystem returned: %s↩
      .\n", onda_st.name, strerror(errno));
  return −1;
    }

    /* Construct the server sockaddr_in structure */
    memset (&dst_server, 0, sizeof (dst_server));          /* Clear ↩
        struct */
    dst_server.sin_family = AF_INET;                        /* ↩
        Internet/IP */
    dst_server.sin_addr.s_addr = inet_addr (onda_st.addr);   /* IP ↩
        address */
    dst_server.sin_port = htons (onda_st.port);              /* ↩
        server port */

    /* Establish Connection */
    if (connect (onda_st.sock, (struct sockaddr *) &dst_server, ↩
        sizeof(dst_server))  < 0){
  info_printf ("Error: Failed to connect with %s on address: %s port:↩
      %i.\nSystem returned: %s.\n",
          onda_st.name, onda_st.addr, onda_st.port, strerror(errno));
  return −1;
    }
    else
  onda_st.connected = ONDA_CONNECTED;

    return 0;
}






int
tcp_term (void)
{
    int retval;
    if (onda_st.connected != ONDA_CONNECTED){
  info_printf ("Onda interface err.: Can not close non existing ↩
      connection: %s.\n",
            onda_st.name);
        return -1;
    }

    retval = close(onda_st.sock);
    if (retval == 0){
        onda_st.connected = ONDA_NOT_CONNECTED;
        onda_st.sock = 0;
        info_printf("Onda interface: closed connection.\n");
    }else{
        info_printf ("Onda interface err.: Could not terminate connection.\↩
            nSytem returned: %s.\n", strerror(errno));
        return retval;
    }

    return 0;
}







void
tcp_auto_term (void)
{
    if (onda_st.connected == ONDA_CONNECTED)
        tcp_term();

    return;
}









/**
 * Sends raw data using the socket handle
 * Input:   -struct containing all connection info
 *          -pointer to a char buffer where error messages are ↩
     printed
 *
 * Returns:  0 on success
 *          -1 on failure
 */
int
tcp_tx (const char* send_str)
{
    int echolen;
    int retval;
    /* set the maximum number of file descriptors */
    int nfds = onda_st.sock + 1;
    fd_set write_fd;
    struct timeval timeout;

    if (onda_st.connected != ONDA_CONNECTED){
        info_printf ("Error: can not send data. Connection is non ↩
            initialised.\n");
        return -1;
```

```
  }

  /* Clear the receive buffers */
  retval = tcp_clear_rx_buff();
  if (retval)
return -1;

  /* clear the struct */
  FD_ZERO (&write_fd);
  /* add the socket to the struct/list */
  FD_SET (onda_st.sock, &write_fd);

  /* set timeout to 5 sec */
  timeout.tv_sec = 5;
  timeout.tv_usec = 0;

  /* Wait until there is room in the socket send buffer or select()↩
      times out */
  retval = select (nfds, NULL, &write_fd, NULL, &timeout);

  switch (retval)
{
    /* timeout */
case 0:
    info_printf ("Unable to send data. Socket is not ready to send ↩
        data.\n");
    return -1;
    break;

    /* error */
case -1:
    info_printf ("Unable to send data.\nSystem returned: %s\n", ↩
        strerror(errno));
    return -1;
    break;

    /* AOK */
default: break;
}


  /* Send the string to the server */
  /* echolen = strlen (onda_st.tx_buf); */
  /* retval  = send (onda_st.sock, onda_st.tx_buf, echolen, 0); /\*↩
      "0" for IP-protocol *\/ */
  echolen = strlen (send_str);
  retval  = send (onda_st.sock, send_str, echolen, 0); /* "0" for ↩
      IP-protocol */

  /* test for success */
  if (retval == -1)
{
    info_printf ("Unable to send data\nSystem returned: %s\n", ↩
        strerror(errno));
    return -1;
}
  else if (retval != echolen)
{
    info_printf ("Unable to send all data. Should have sent: %i, ↩
        but only %i bytes was sent.\nSystem returned: %s\n",
     echolen, retval, strerror(errno));
    return -1;
  }
    return 0;
}






/**
 * Receives data using the socket handle
 * Input:   -struct containing all connection info
 *          -pointer to a char buffer where error messages are ↩
     printed
 *
 * Returns:  0 on success
 *          -1 on failure
 */
int
tcp_rx (char** data_ptr)
{
    int retval;
    int nfds = onda_st.sock + 1;
    fd_set read_fd;
    struct timeval timeout;
    /* make sure pointer does not contain random value */
    // *data_ptr = NULL;

    if (onda_st.connected != ONDA_CONNECTED){
  info_printf ("Onda interface err.: Can not receive on non-existing ↩
      connection.\n");
  return -1;
    }


    /* clear the struct */
    FD_ZERO (&read_fd);
    /* add the socket to the struct/list */
    FD_SET (onda_st.sock, &read_fd);
    /* set timeout to 5 sec */
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    /* Wait until data is available or timeout has passed */
    retval = select (nfds, &read_fd, NULL, NULL, &timeout);

    /* Wait a little extra time, to assure the entire string is ↩
        received. */
    timeout.tv_sec = 0;
    timeout.tv_usec = 50000; /* 0.05sec */
    select (nfds, NULL, NULL, NULL, &timeout);

    switch (retval)
  {
      /* time out */
  case 0:
      return ERR_RCV_TIMEOUT;
      break;

      /* error */
  case -1:
      info_printf ("Error: could not receive data.\nSystem returned: ↩
            %s.\n", strerror(errno));
        return ERR_RCV_SYS_ERR;
        break;

        /* AOK */
    default: break;
    }

      /* data available -> receive it */
      if ((retval = recv (onda_st.sock, onda_st.rx_buf, onda_st.↩
          buf_size -1, 0)) < 0){
    rx_buf [0] = '\0';
    /* error handling */
    info_printf("Error: could not receive data.\nSystem returned: %s.\n↩
        ", strerror(errno));
    /* Assure null terminated string */
    return ERR_RCV_NO_DATA;
      }
      else
    /* Assure null terminated string */
    rx_buf [retval] = '\0';


      /* set pointer to buffer */
      *data_ptr = rx_buf;

      return 0;
}






/**
 * Sends AND receives data using the socket handle
 * Input:   -c-string containing the TX string
 *          -pointer to a char buffer where error messages are ↩
     printed
 *
 * Returns:  0 on success
 *          -1 on failure
 */
int
tcp_query (const char* send_str, char** receive_ptr)
{
    int retval;
    retval = tcp_tx(send_str);
    if (retval)
  return retval;

    /* receive string */
    retval = tcp_rx(receive_ptr);
    if (retval)
  return retval;

    return 0;
}
```

Listing 2.5: tcp.cpp