1. Introduction

The Laser Beam Stabilization system can be controlled via a preconfigured interface: USB for easy use, R232 for rough environments and Ethernet e.g. for system integration are available. All interfaces make use of the same communication protocol. This document explains the protocol, the utilization via a serial terminal and the integration into your source code.

2. Protocol

The communication protocol was developed to achieve high speeds and the associated data rates. It is based on a combination of ASCII characters and hexadecimal (HEX) numbers that are transmitted and received. Depending on the request to the Laser Beam Stabilization, you receive a different answer, which can consist of ASCII characters and hexadecimal numbers. Therefore it is important to always check in the specification what kind of response you can expect.

As a simple example you can see the request for the active status of the system in figure 1. Here we send *GAS;* (Get Active Signal) over the interface. The command consists of three ASCII characters and is terminated by a semicolon. As a response we receive a confirmation that the command was understood, symbolized by a 0 (hexadecimal number) and a semicolon (ASCII). Afterwards we get two bytes with the payload followed by a second semicolon terminating the response.

To understand the response in the payload, you need to look up the specification in the protocol. In the case of *GAS;* we expect two hexadecimal numbers in unsigned char format, which means one byte each. The first byte of the payload represents the first stage, the second byte represents the second stage. In the case of the example, stage 1 is inactive (0) and the second stage is active (1).

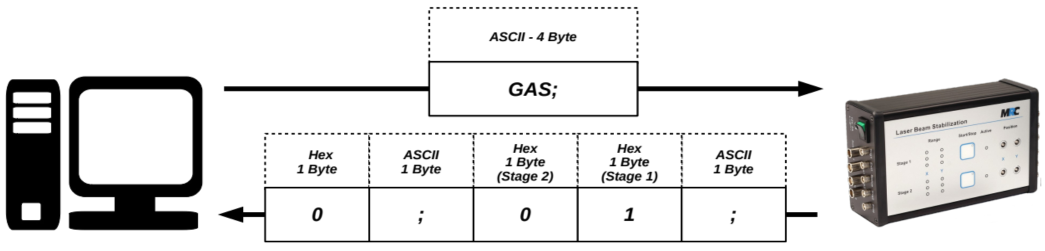


Figure 1: Get active status (GAS) command

Some commands require data to be sent to the Laser Beam Stabilization system. These commands are similar to those that consist of only a request. You can see an example of a request for data in the example in figure 2. It shows you can see the request to move the target position of the detector in x on the first control stage. The command is *SAI* followed by the axis, the control stage and the target position followed by a semicolon. The command starts with three ASCII characters followed by the payload and a semicolon as the termination. The answer consists of just the acknowledgement (0x00 followed by a semicolon), as there is no payload to be transferred.

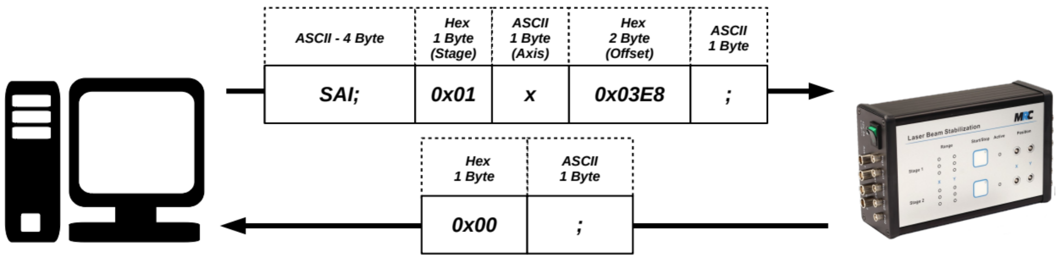


Figure 2: Set adjust in command

3. Errors

In contrast to the acknowledgement, the Laser Beam Stabilization system sends back a 1 (hexadecimal number) followed by a semicolon each time an error occurred. To invoke on the cause of the error, the command *GER;* can be used. It returns the name of the command that provoked the error followed by one signed char as a hexadecimal number that acts as an error code. These error codes are explained in section 3 of the protocol.

4. Data streaming

In order not to ask for each data package separately, there is the option to start a continuous data stream. During a stream, however, no further commands can be sent, as otherwise it would not be possible to distinguish between the data stream and the response to this request. The command *SLS;* starts a data stream, which can be automatically stopped after a specified number of packages were sent, but can also run indefinitely. In this case you will receive data records permanently until you terminate the stream with the command *CLS;*.

The data packages of a stream always consist of the same components and contain all important information about position, intensity, status and piezo voltages. Figure 3 shows a data package with example values.

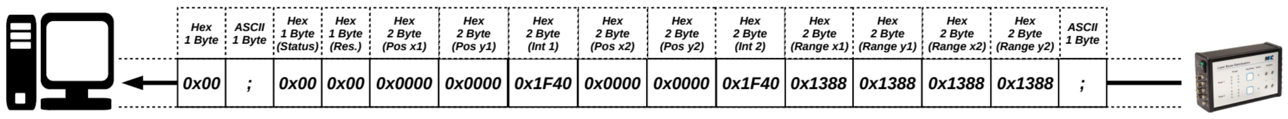


Figure 3: Data stream example

Since a *CLS;* command can arrive at the beam stabilization system at the same time as a data package is sent, there is a special bit in the status byte that marks the last package of the stream. Once the command has been processed, the first bit in the status byte is set to 1 followed by the remaining data of this stream package. After the stream ended, a 0 (hexadecimal number) and semicolon (ASCII) are sent as a confirmation.

5. Application via serial terminal

Using the protocol via a serial terminal is in principle possible, but requires correct settings and the possibility to transfer hexadecimal numbers to get the communication working. To open a connection you have to make sure that you set the correct parameters for the interface. You can find the factory settings in table 1.

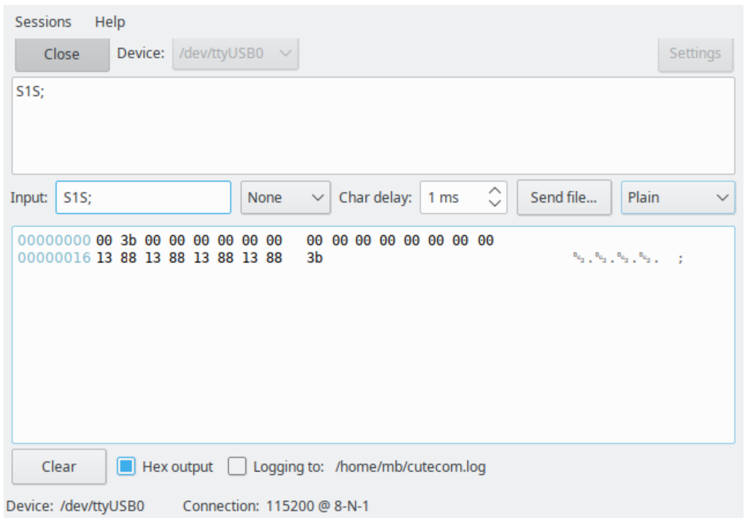
Parameter	Value
Baudrate [bits/s]	115200 (for USB and RS232)
Data bits	8
Parity	no
Stop bit	1
Handshaking	on
Flow control	hardware

Table 1: Parameters for the interface

Once you have opened a connection, you must make sure that the received data are shown as hexadecimal numbers, otherwise you will most likely be unable to understand the response to certain requests.

When entering a command in the input field you have to take care that the commands can partially consist of ASCII characters mixed with hexadecimal numbers. You must therefore switch between the two or convert the ASCII characters into hexadecimal numbers beforehand.

As a first example you can see the request for a single data package in figure 4. Here, the request can be sent with only ASCII characters, because there is no payload afterwards. The output is set to hexadecimal numbers, because the data package contains numerical values. As you can see in figure 4, the response is quite compact compared to sending the numerical values as ASCII characters, so that higher data transfer rates can be achieved.



Device: /dev/ttyUSB0

```
S1S;
```

Input: S1S; None Char delay: 1 ms Plain

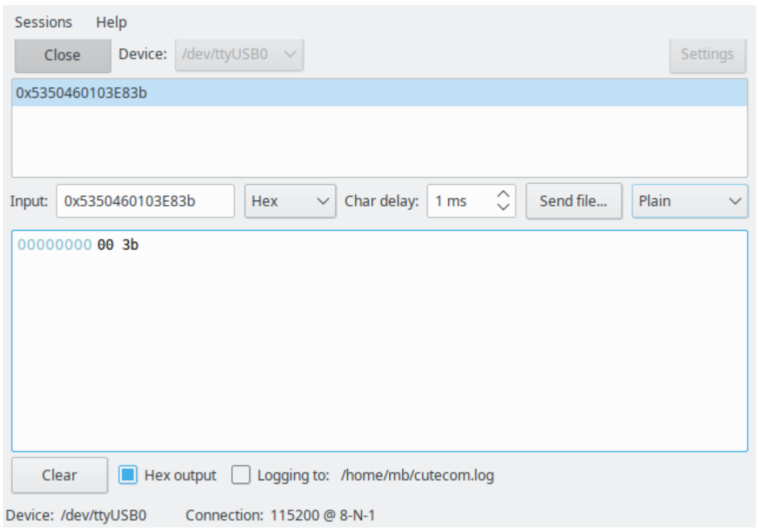
```
00000000 00 3b 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00000016 13 88 13 88 13 88 13 88   3b                          ␓.␓.␓.␓.  ;
```

Hex output [✓] Logging to: /home/mb/cutecom.log

Device: /dev/ttyUSB0 Connection: 115200 @ 8-N-1

Figure 4: Command via serial terminal (CuteCom)

In the second example we set the P factor of the first stage to 1 Volt. Since the command now consists of a mixture of numerical values and ASCII characters, we have translated the ASCII characters *SPF* into the hexadecimal value 0x535046 and then added the stage 0x01 and voltage 0x03E8 followed by the semicolon which translates to 0x3b. Combined, this results in the command 0x5350460103E83b, which is shown in figure 5.



Device: /dev/ttyUSB0

```
0x5350460103E83b
```

Input: 0x5350460103E83b   Hex   Char delay: 1 ms   Plain

```
00000000 00 3b
```

Hex output ✓   Logging to: /home/mb/cutecom.log

Device: /dev/ttyUSB0   Connection: 115200 @ 8-N-1

Figure 5: Set P factor (SPF) via a serial terminal

6. Implementation in source code

When implementing the protocol in your own source code, you can manage the interface by yourself. Doing so, the settings from table 1 must be applied, if it is a serial device. With an Ethernet system, you can connect with the Ethernet address[1] of the system.

After the connection is opened, you can transfer data to the Laser Beam Stabilization system as described in section 2. It is possible to use an array in which you enter the commands and the payload and then transmit them via the interface. After a short waiting period you should be able to read back the response from the Laser Beam Stabilization system via the interface. It is advised to wait until you have received the specified number of bytes as an answer to your request. In addition to examining the length of the received answer you should also check whether you received the acknowledgement (0x00 followed by a semicolon). By that you can be sure to have received a complete and valid answer before issuing the next request.

When utilizing an indefinite data stream, you must make sure that you have a termination criterion. Once this criterion occurs, you can stop the stream with the *CLS;* command. If you do not implement such a criterion, the Laser Beam Stabilization system will continue to stream data until it is manually stopped or the power is turned off. This can cause problems if you restart your program and want to send a request. In this case you will get unexpected data from the stream instead of the desired response or reaction to your request.

In the following section you will find a simple example in C for a USB system. In the example, the connection is opened and various commands and options are used to illustrate the implementation.

[1]: The Laser Beam Stabilization systems equipped with an Ethernet interface are delivered in the DHCP configuration. If the device is to be connected without a DHCP server, the WIZS2E ConfigTool can be used to set a static IP address.

MRC-0124-1-e

Contact

MRC Systems GmbH
Hans-Bunte-Str. 10
D-69123 Heidelberg, Germany
Phone: +49 6221/13803-00
Email: info@mrc-systems.de

7. C example for a USB device

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>
#include <algorithm>

//function to display an error message
void error(const char *msg)
{
   perror(msg);
   exit(0);
}

//function to apply the parameters of the interface (see table 1 in description of the serial_interface)
//as parameters we get the interface to open (fd), the baudrate we want (speed) and the bit parity of the protocol
int set_interface_attribs (int fd, int speed, int parity)
{
   //first we setup the memory where we save the parameters
   struct termios tty;
   memset (&tty, 0, sizeof tty);
   if (tcgetattr (fd, &tty) != 0)    //we check if the memory is available
   {
      error("ERROR from tcgetattr");
      return -1;
   }

   //we set the paramters to the memory first and apply them afterwards
   cfsetospeed (&tty, speed);
   cfsetispeed (&tty, speed);
   tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;     // 8-bit chars
   // disable IGNBRK for mismatched speed tests; otherwise receive break
   // as \000 chars
   tty.c_iflag &= ~IGNBRK;        // disable break processing
   tty.c_lflag = 0;               // no signaling chars, no echo,
   // no canonical processing
   tty.c_oflag = 0;               // no remapping, no delays
   tty.c_cc[VMIN]  = 0;           // read doesn't block
   tty.c_cc[VTIME] = 5;           // 0.5 seconds read timeout

   tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff ctrl

   tty.c_cflag |= (CLOCAL | CREAD);// ignore modem controls,
   // enable reading
   tty.c_cflag &= ~(PARENB | PARODD);      // shut off parity
   tty.c_cflag |= parity;
   tty.c_cflag &= ~CSTOPB;
   tty.c_cflag &= ~CRTSCTS;

   if (tcsetattr (fd, TCSANOW, &tty) != 0) //setting the values to the interface
   {
      error("ERROR from tcsetattr");
      return -1;
   }
   return 0;
}
```

```
//function to enable/disable the mode of the interface
void set_blocking (int fd, int should_block)
{
   //allocating the memory for the configuration
   struct termios tty;
   memset (&tty, 0, sizeof tty);

   if (tcgetattr (fd, &tty) != 0) //checking if memory allocation was successful
   {
      error("ERROR from tggetattr");
      return;
   }

  //setting the variable to the memory
   tty.c_cc[VMIN]  = should_block ? 1 : 0;
   tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout

  //apply
   if (tcsetattr (fd, TCSANOW, &tty) != 0)
      error("ERROR setting term attributes");
}


int main()
{
   ///--------------------------------------Init-----------------------------------------

   //here we connect to the serial port
   int serial_port = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_SYNC);

   //check for errors
   if(serial_port < 0)
      printf("Error %i from open: %s\n", errno, strerror(errno));

   set_interface_attribs (serial_port, B115200, 0);  // set speed to 115,200 bps, 8n1 (no parity)
   set_blocking (serial_port, 0);               // set no blocking


   ///-------------------------Communication start-----------------------------------------

   char bufferOut[256];
   char bufferIn[256];
   long check;

   //First we clear old data on the system

   ///***Writing***
   strcpy(bufferOut,"CLS;");   //Creating the message - "CLS" = Clear Live Stream
   check = write(serial_port, bufferOut, 4); //writing on the socket (4 = length of symbols)
   if (check < 0)  //checking the result
      error("ERROR writing to socket");

   usleep(10000); //we sleep here 10 milliseconds to allow the system to respond (latency via data transfer)

   ///***Reading***
   bzero(bufferIn,256); //cleaning the storage
   check = read(serial_port, bufferIn, sizeof(bufferIn)); //read from the serial port
   if (check < 0) //checking the result
      error("ERROR reading from socket");
```

```
 ///--------------------------------get board id----------------------------------------

//Now we can communicate with the system

///***Writing***
bzero(bufferOut,256); //cleaning the message
strcpy(bufferOut,"GID;");   //Creating the message "GID" = Get Identifier
check = write(serial_port, bufferOut, 4); //writing on the socket (4 = length of symbols)
if (check < 0)  //checking the result
   error("ERROR writing to socket");

usleep(10000);  //we sleep here 10 milliseconds to allow the system to respond (latency via data transfer)

///***Reading***
bzero(bufferIn,256); //cleaning the storage
check = read(serial_port, bufferIn, sizeof(bufferIn)); //read from the serial port
if (check < 0) //checking the result
   error("ERROR reading from socket");

//Searching for the ";" at the end
char* data = std::find(bufferIn, bufferIn + sizeof(bufferIn) / sizeof(bufferIn[0]),0x3b);
printf("%s\n", data);

 ///------------------------------------activating the stage-----------------------------------------

///***Parameters***
char stage = static_cast<char>(0x01); //Stage 1

///***Writing***
bzero(bufferOut,256); //cleaning the message

strcpy(bufferOut,"SEA");   //Creating the message "SEA" = Set external activation
strcat(bufferOut,&stage); //Choosing the stage
strcat(bufferOut, ";");

check = write(serial_port, bufferOut, 5); //writing on the socket (4 = length of symbols)
if (check < 0)  //checking the result
   error("ERROR writing to socket");

usleep(10000);  //we sleep here 10 milliseconds to allow the system to respond (latency via data transfer)

///***Reading***
bzero(bufferIn,256); //cleaning the storage
check = read(serial_port, bufferIn, sizeof(bufferIn)); //read from the serial port
if (check < 0) //checking the result
   error("ERROR reading from socket");

if(bufferIn[0] == 0 && bufferIn[1] == 0x3b)
   printf("Stage 1 is activated.\n");
else
   printf("Stage 1 couldn't be activated.\n");
```

```
///--------------------------------------a single stream block----------------------------------------

///***Writing***
bzero(bufferOut,256); //cleaning the message
strcpy(bufferOut,"S1S;");   //Creating the message "S1S" = SingleShot
check = write(serial_port, bufferOut, 4); //writing on the socket (4 = length of symbols)
if (check < 0)  //checking the result
   error("ERROR writing to socket");

usleep(10000);  //we sleep here 10 milliseconds to allow the system to respond (latency via data transfer)

///***Reading***
bzero(bufferIn,256); //cleaning the storage
check = read(serial_port, bufferIn, sizeof(bufferIn)); //read from the serial port
if (check < 0) //checking the result
   error("ERROR reading from socket");

//anaylsing the stream block
if(bufferIn[0] == 0 && bufferIn[1] == 0x3b) //Checking if acknowledgement was sent
{
   int status_flag = bufferIn[2];
   bool on1 = status_flag & 8;
   bool on2 = status_flag & 16;
   bool active1 = status_flag & 32;
   bool active2 = status_flag & 64;

   //each value is separated in two char values. Shifting the first value up and adding the second to get correct result
   int position_X1 = (bufferIn[4] << 8)  + (bufferIn[5] & 0xFF);
   int position_Y1 = (bufferIn[6] << 8)  + (bufferIn[7] & 0xFF);
   int power_I1 = (bufferIn[8] << 8)  + (bufferIn[9] & 0xFF);
   int position_X2 = (bufferIn[10] << 8) + (bufferIn[11] & 0xFF);
   int position_Y2 = (bufferIn[12] << 8) + (bufferIn[13] & 0xFF);
   int power_I2 = (bufferIn[14] << 8) + (bufferIn[15] & 0xFF);
   int range_X1 = (bufferIn[16] << 8) + (bufferIn[17] & 0xFF);
   int range_Y1 = (bufferIn[18] << 8) + (bufferIn[19] & 0xFF);
   int range_X2 = (bufferIn[20] << 8) + (bufferIn[21] & 0xFF);
   int range_Y2 = (bufferIn[22] << 8) + (bufferIn[23] & 0xFF);

   //displaying result
   printf("\n");

   if(on1 && active1)
      printf("Stage 1 is stabilizing: \n");
   printf("Position Stage 1 (x, y): %d,%d \n", position_X1, position_Y1);
   printf("Range Stage 1 (x, y): %d,%d \n\n", range_X1, range_Y1);

   if(on2 && active2)
      printf("Stage 2 is stabilizing: \n");
   printf("Position Stage 2 (x, y): %d,%d \n", position_X2, position_Y2);
   printf("Range Stage 2 (x, y): %d,%d \n\n", range_X2, range_Y2);

   printf("Detector Intensity (Stage1, Stage2): %d,%d \n\n", power_I1, power_I2);
}
else
   printf("Acknowledgement wasn’t sent\n\n");
```

```
    //-----------------------------set adjust values on stage 1 x to 0----------------------------------------------

    ///***Parameters***
    stage = static_cast<char>(0x01);      // Stage 1
    char axis = static_cast<char>('x');   // Axis X
    char adj_val_high = static_cast<char>(0x00);
    char adj_val_low = static_cast<char>(0x00);

    ///***Writing***
    bzero(bufferOut,256); //cleaning the message

    strcpy(bufferOut,"SAI");   //Creating the message "SAI" = Set adjust in
    strcat(bufferOut,&stage); //Choosing the stage
    strcat(bufferOut,&axis); //Choosing the axis
    strcat(bufferOut, &adj_val_high);
    strcat(bufferOut, &adj_val_low);

    char* p_bufferOut = bufferOut;
    memmove(p_bufferOut + 7, ";",1);    //we need to insert the end of the command, because "0" is not recognized by strcat()

    check = write(serial_port, bufferOut, 8); //writing on the socket (8 = length of symbols)

    if (check < 0)  //checking the result
       error("ERROR writing to socket");

    usleep(10000);  //we sleep here 10 milliseconds to allow the system to respond (latency via data transfer)

    ///***Reading***
    bzero(bufferIn,256); //cleaning the storage
    check = read(serial_port, bufferIn, sizeof(bufferIn)); //read from the serial port
    if (check < 0) //checking the result
       error("ERROR reading from socket");

    if(bufferIn[0] == 0 && bufferIn[1] == 0x3b)
       printf("Adjust value on stage 1 axis x set to 0.\n");
    else
       printf("Adjust value on stage 1 axis x could'nt be changed.\n");


    close(serial_port);
    return 0;
}
```