

Servo-based mirror motor

Student Paper

Author(s):

Lempereur, Stephan

Publication date:

2023

Permanent link:

<https://doi.org/10.3929/ethz-b-000660479>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

Semester project
Servo-based mirror motor

Stephan Lempereur

December 2023

Contents

1	Introduction	3
2	The setup	3
3	Feetech STS3032 servomotor	4
3.1	Servomotor registers	4
3.2	Serial communication protocol	6
3.3	Instructions	7
4	Teensy 4.1 and Ethernet communication	8
4.1	Teensy pins	8
4.2	Ethernet communication	8
4.2.1	The Ethernet cable	9
4.2.2	The internet protocols	10
4.3	SCPI communication protocol	12
4.4	Controller program	13
5	System debugging	14
5.1	Ethernet protocol	14
5.1.1	The Direct Host Configuration Protocol	14
5.1.2	The TCP Three Way Handshake	14
5.1.3	The TCP data exchange	15
5.2	Serial communication	15
5.3	Teensy damage	15
6	Future possibilities	16
6.1	User interface	16
6.2	Multi-motor operation	16
6.3	Synchronous operation	17
7	Conclusion	17
8	Appendix	19

1 Introduction

Optical setups such as those used in cold atom experiments are vast and dense with precision optics, the alignment of which is crucial. Most optical components are controlled via precise, micro-metric screws which can be difficult to access without disturbing other parts of the setup. Some components, isolated to avoid exposure to the outside environment, are simply inaccessible. Adjustment by hand would require opening the setup and may not be compatible with the simultaneous measurement necessary to dynamically verify alignment. Accidental touching can also lead to misalignment, fingerprints, dust particles...

This work, performed over the spring semester of 2023, is part of an ongoing project in the Quantum Optics group at ETH Zurich aiming to automate control of mirrors through the use of servomotors. These motors perform rotations around an axis with precise control of angular position, velocity, and acceleration. By connecting them to the alignment screws of optical components, incredibly small adjustments are possible. Combined with electronic control at a distance, they can be placed in otherwise inaccessible parts of a setup all while exhibiting a much smaller spacial footprint than a human arm. A proper implementation of such a system must aim to combine this precision and reliability with the intuitive control and responsiveness of manual alignment. Such motorized kinetic mounts are already commercially available [1] but can come at a high cost, north of a thousand dollars. In this project, much cheaper servomotors (Feetech STS3032) are used in combination with an Arduino-like board (Teensy 4.1) to fulfill similar requirements for a much lower price.

In previous works on this project [2], the Feetech STS3032 motors were characterized and exhibited precise and repeatable movement making them good candidates. This work is instead centered on building a sturdy framework for the control of both the motors and Teensy board through an Ethernet connection to a computer.

This report aims to explain recent contributions, but also to act as partial documentation for the system as it exists now for reference during further work on this project. Here is compiled technical information about the servomotors, Teensy chip, implemented communication protocols, as well as practical information about system debugging and future ideas to extend the project. The following part will detail the components of the automated mirror setup, while parts 3 and 4 will describe the servomotor operation and communication methods respectively. Finally, various methods of debugging used during the project will be presented in section 5, as well as possible future work to further it (section 6).

2 The setup

Kinematic stages for optical devices can apply multiple transformations to their position and orientation. Mirror mounts usually have two screws controlling the two angles of rotation. Each screw requires one servomotor to operate. Though a single servomotor controls a single screw, simultaneous operation of servomotors allows for higher-level control of the mirror, such as changing the beam path of a reflected ray along a given direction, or automating fiber coupling.

The servomotor chosen is the Feetech STS3032, coming at around 25\$ apiece. Instructions are sent to servomotors using serial communication through a three-wire connection[3]: a ground wire, a direct current wire (6V) to power the motor and a signal wire. Bits are

transmitted through the signal wire as high voltage (1) or low/no voltage (0).

Serial communication with the motors is performed by a microcontroller board, either the URT-1 debugger board sold by the motor’s manufacturers, or in our case an Arduino-like chip loaded with our own software. The URT-1 board comes with a driver and GUI software and enables visual control and debugging of a servomotor [4]. It is powered by a USB connection to a computer and a 6V DC voltage applied to the underlying PCB to amplify the serial communication, power the motor and the micro controller. The provided software allows reading and writing to all internal registers of the motor, as well as control of its position, speed or angle. Kinematic parameters of the motor are plotted over time, making it a great tool for understanding the dynamics of the motor and verifying its state.

For a more programmatic usage of the motors, an Arduino-like microcontroller was chosen: the Teensy 4.1. It is also powered through USB or an external DC voltage generator, and requires an additional PCB board to power the motor and amplify the signal. The Teensy is programmable through the Arduino IDE interface in C++ and all standard arduino libraries are either compatible with the chip, or equivalent and identical ones have been ported to it [5]. Once the c++ program has been compiled and uploaded, the computer USB connection is no longer necessary for operation. Instead, once disconnected from USB, control of the Teensy and therefore the motor is performed through an Ethernet connection, either directly between the computer and the Teensy, or with an intermediate router. A final example of the configuration is given in figure (1).

3 Feetech STS3032 servomotor

The servomotor chosen for this project is the STS3032. These motors can be controlled through a serial connection described in section 3.2 by using a finite set of instructions detailed in 3.3. These instructions are used to read out and modify their internal registers, presented in section 3.1.

3.1 Servomotor registers

The Feetech’s registers contain all configuration parameters of the motor, as well as its current positional and kinematic information. Control over the register entries is performed over serial communication using a set of instructions detailed further in the report. The only documentation found for these registers corresponds to a slightly different version of the motor, and presents some differences [6]. The actual registers of the STS3032 servomotor are given in the appendix, and compiled using the URT-1 debugging software.

Two types of memory exist within the servomotor:

- The Random Access Memory (RAM) contains the live information regarding the functioning of the servomotor. It contains, for instance, the current position (byte 0x38-0x39), speed (0x3A-0x3B) or temperature (0x3F). As any RAM, it is lost upon powering off the device.
- the EEPROM, a non-volatile memory, for all other parameters which should not be erased upon power off. It contains, for instance, ID (0x05), baud rate (0x06) or minimum and maximum angle limits (0x09-0x0C).

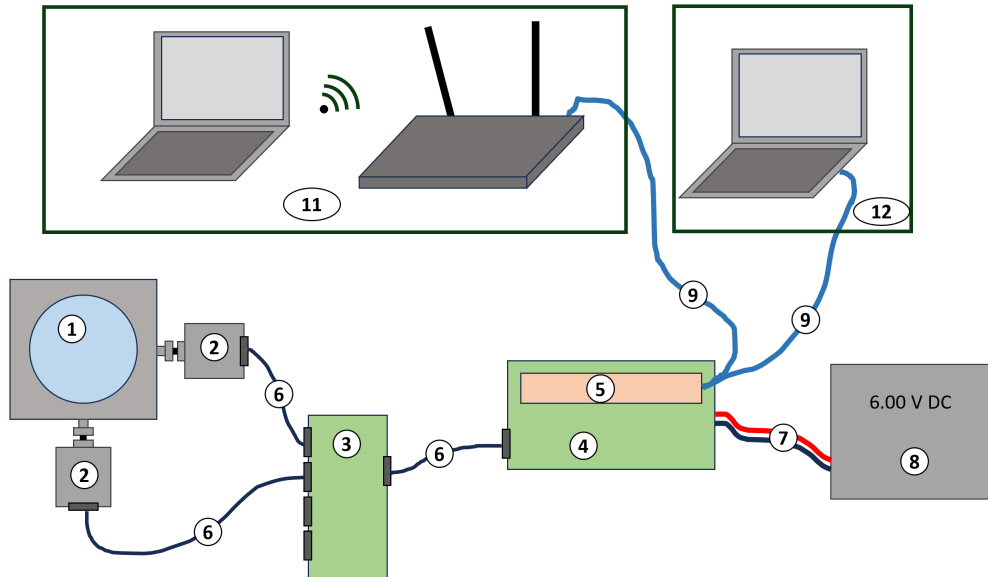


Figure 1: Typical configuration for controlling the mirror. The mirror is set in a kinematic mount (1) with two alignment screws to which servomotors (2) are connected. The motors are controlled and powered through a serial cable (6) connected to the microcontroller PCB (4). To supply every motor, the serial signal is split through another PCB (3) which distributes and amplifies the signals to each motor. Generation of the signal is performed by the Teensy 4.1 (5) and amplified by the microcontroller PCB. Power is supplied by a direct current generator (8). Control of the Teensy board is performed through Ethernet (9) either directly connected to a computer (12) or to a router to which a computer may connect (11).

EEPROM and RAM can be written to and read from using the same instructions.

Here are some useful register entries:

EEPROM will only remain modified after power off if the Lock function bit (0x30) is set to 0 and lock protection is deactivated. The user manual does however warn that repetitive writing with lock protection off can be damaging to the device and should be kept to a minimum.

Moving the servomotor is achieved by setting the Target position bytes (0x2A-0x2B) which takes values between -32 766 and 32 766. The 360 degree angle is only subdivided into 4096 parts and larger position settings will cause the servomotor to rotate more than one turn.

Depending on the desired usecase, the speed of rotation of the servomotor can be set either using the Running speed bytes (0x2E-0x2F), which set the angular velocity of the motor or the Running time bytes (0x2C-0x2D) which set the time it will take to reach the target position. The speed bytes will take priority over the time bytes.

The STS3032 motor can be configured in four operation modes:

- Position Control mode (0): Target position must be set between 0 (0 degrees) and 4095 (360 degrees). This mode is the most precise, but cannot perform the multi-turn rotations required to move a mirror's alignment screw.
- Closed Loop Motor Control mode (1): Sets a constant rotation speed regardless of the load applied to the motor.
- Open Loop Motor Control mode (2): Sets a constant rotation speed at zero load. Speed decreases with load.
- Step mode (3): Target position is always relative to current position. Setting the target position register to 10 will rotate the motor ten steps in one direction, then reset the register to 0. Setting target position to 10 again will repeat the same rotation in the same direction. Changing the sign of the target position will switch the direction of rotation.

For the purpose of mirror control, only the Step mode is of interest. It is obtained by setting all angle limits to 0 (0x09-0x0C) and the Operation Mode byte (0x21) to 3. This can be done with the debugging URT-1 chip, or by adequate instruction calls through serial.

Finally, the servo ID should be changed using the ID byte (0x05) while not forgetting to toggle the Lock function byte.

A few practical notes:

- Some register entries are read-only, instead of being read/write. These are the current software version, and the current position, speed, load, voltage and temperature.
- Some register entries span over two bytes instead of one. They are referred to as High (H) and Low (L) and link to the parameter value as such:

$$x = x_H \cdot 256 + x_L$$

where x is the parameter value, and x_H and x_L are the H and L bytes respectively. In memory, the high bit is stored before the low bit (for current position: H (0x38) L (0x39)).

3.2 Serial communication protocol

Serial communication with the servo motor is done by sending INSTRUCTION packets, and receiving REPLY packets. These instructions all read or write to the motor's registers, apart from the PING instruction, which simply requests a reply from the motor.

INSTRUCTION packets have the following structure:

Byte name	Header	Servo ID	Length	Instruction	N Parameters	Check Sum
Byte count	2	1	1	1	N	1

Here,

- Header: Bytes characteristic of the start of the packet: 0xff 0xff
- Servo ID: Servos have associated IDs between 0 and 253. Only servos with that given ID will respond to the instruction. ID number 254 corresponds to a broadcast ID: every servo will receive and execute the instruction. There will be no reply.
- Length: Bytes of information sent (equal to $N + 2$). The total packet size is Length + 4.
- Instruction: Instruction number (see following section).
- N Parameters: N successive bytes, each representing a given parameter. The number of parameters depends on the instruction.
- Check Sum: Sum of all bytes except itself. Will be used to verify packet integrity. More details in documentation.

REPLY packets have the following structure:

Byte name	Header	Servo ID	Length	ERROR	N Parameters	Check Sum
Byte count	2	1	1	1	N	1

Here,

- ERROR: Normal operation has this byte set to 0. Errors will be reflected here.
- N Parameters: These parameters are the result of a READ instruction query.

3.3 Instructions

There are 7 possible instructions for the servomotor:

- PING: Verify if a servo is connected through serial at a given ID.
Parameters (0): None
- READ: Read and return N consecutive registers.
Parameters (2): First register to read, Number of registers to read (N)
- WRITE: Write to N consecutive registers.
Parameters ($N + 1$): First register to write to, value 1, value 2...
- REGWRITE: Buffer a WRITE instruction which will only be executed upon receiving the ACTION instruction.
Parameters ($N + 1$): First register to write to, value 1, value 2...
- ACTION: Execute all REGWRITE instructions. The broadcast ID 0xFE can be used to execute all REGWRITE instruction in all servos simultaneously.
Parameters (0): None

- SYNCWRITE: WRITE to N registers of S servos. These instructions will be executed immediately and simultaneously. However, all write instructions will write to the same register for all servos. The broadcast ID should be used.

Parameters ($S(N + 1) + 2$): first register to write to, number of registers written (N), [servo ID 1, value 1, value 2, ..., value L], [Servo ID 2, value 1, ..., value L], ... [Servo ID S , value 1, ..., value L]

Though REGWRITE, ACTION and SYNCWRITE instruction are more complex than the other three and serve little purpose when a single servomotor is connected, they become essential once simultaneous and synchronized control of multiple servos is required.

Having explained how the STS3032 servomotor can be controlled through serial packets, we will explore in the next part how these packets are generated and sent by the Teensy microcontroller.

4 Teensy 4.1 and Ethernet communication

Teensys are microcontrollers equipped with an arduino-like operating system. It is programmable through the Arduino IDE and most Arduino libraries can be used. Component specific libraries, like the Ethernet library, have been redeveloped for the Teensy chip [5]. On top of the provided documentation [7], a very active and responsive forum is available[8] for all extra information.

4.1 Teensy pins

The Teensy has a multitude of pins (figures 2, 4) for power, input, output or parameter reading and writing. The totality of them can be found on the Teensy's manufacturer's website (Prjc) [9]. A few notable pins are of interest:

- At the center of the Teensy (figure 4), six pins control the Ethernet connectivity.
- There are three ground pins.
- When power is not supplied to the Teensy by USB, it must be supplied through the Vin pin.
- Serial communication is performed using pins 7 (receive) and 8 (transmit), though many other serial connections are available through other pins.

4.2 Ethernet communication

Though the Teensy chip can communicate directly with a computer and Arduino IDE using serial through USB, it is limited and does not allow the versatility that we desire. Instead, the Teensy chip can be adapted for Ethernet communication by soldering the proper pins and using the Native Ethernet library.

The most fundamental role of the Teensy chip is to simply reroute messages from the computer to the servo motor and back. To do so, a server runs on the Arduino board, allowing clients (computers) to connect.

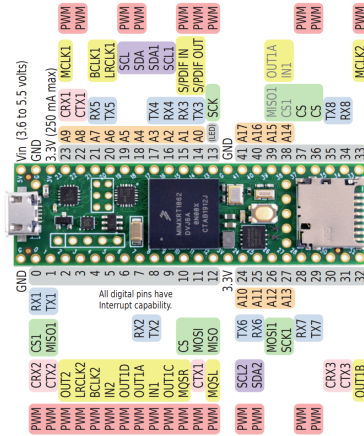


Figure 2: Teensy 4.1 pins, as represented by the manufacturer’s website 2.

4.2.1 The Ethernet cable

The Ethernet cable is today the default communication medium for most internet applications. Long Ethernet cables are readily available and can contend with serial ones in both speed and reliability. Combined with the Ethernet protocols, information loss is rare. Ethernet cables are made up of 8 wires, as seen in figure 3:

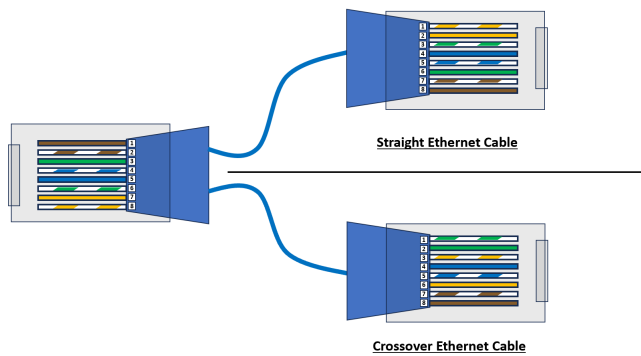


Figure 3: Wiring of Straight and Crossover Ethernet cables. Eight wires are present, paired two by two. Straight Ethernet cables are the most common, connecting a client to a router, whereas crossover cables allow client to client connections. They are rarer nowadays due to the advent of automated Medium Dependent Interface crossover (auto-MDIX) technology, detailed at the end of section 4.2.2.

In typical uses of the Ethernet cable, four of the wires (4, 5, 7, 8) serve no purpose and can be used as ground pins. The other wires are the TX+ and TX-, twisted and forming a balanced pair for the signal transmission, and RX+ and RX-, similar for signal reception. An Ethernet setup kit can be bought on the Teensy store, but this can also be performed

by properly wiring the Teensy pins to an Ethernet port on a PCB as per figure 4.

The availability and cost of Ethernet cables is also advantageous when using them as simple serial communication connections. Indeed, with a typical serial cable, the servo motor communication is performed through ground, voltage and signal wires which can be rerouted through the ground, RX and TX wires of an Ethernet cable, respectively. At an earlier stage in the project, a PCB board was created to connect the ground, voltage and output serial signal pins of the Teensy to an Ethernet port. At the end of the Ethernet cable, a second PCB board retrieves the ground, voltage and signal and rebroadcasts it through the usual serial cable to the servomotor. The servomotor is powered by the voltage wire in the serial cable, and the serial signal is required to have a certain amplitude for the motor to register it (0.45V for a LOW bit and 2V-5V for a HIGH bit.). The external power source provides 6V DC to the Teensy's PCB board which is then used through the Ethernet cable to power the motor and amplify the serial signal. In the case of multiple motors, a signal splitter and amplifier would be necessary.

Installation of an internet connection onto the Teensy chip is natively supported and requires the connection of five (optionally six, if the Ethernet socket LED is desired) pins (RX+, RX-, TX+, TX- and GRD) to an Ethernet socket. In fig 4, numbers on the right figure match those of fig 3. Though such Ethernet socket boards for soldering are available to purchase from the Teensy maker's website, it is a straightforward process to make and solder the PCB board by oneself.

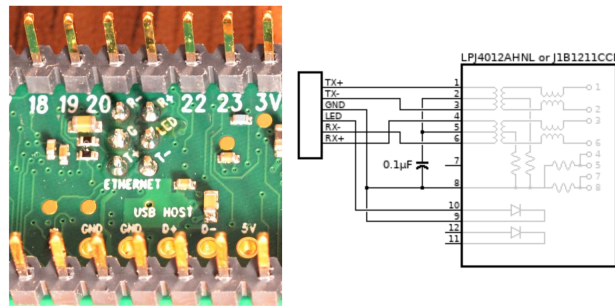


Figure 4: Teensy manufacturer instructions for Ethernet wiring. The wire numbers given on the rightmost subfigure correspond to those shown in figure 3.

4.2.2 The internet protocols

The teensy chip supports Ethernet communication just like any Arduino chip through one of two libraries: the NativeEthernet library and the QNEthernet library. The NativeEthernet library acts as a carbon copy of the default Arduino Ethernet library, while QNEthernet extends it to support additional functionalities proper to the Teensy chip. QNEthernet is overall a more complete library. Of interest for our project, the autodetection of hardware Medium Access Control (MAC) addresses and the verification of the auto-MDIX (Medium Dependent Interface crossover, detailed at the end of this section) functionalities of the chip are only offered by QNEthernet and could be of use. However, due to more readily available

documentation for the NativeEthernet library, it was used for this project.

To establish communication with our computer, the Teensy runs a Dynamic Host Configuration Protocol (DHCP) server on the local network to which the computer (the client) can connect. The Ethernet library assigns itself an IP address, which may need to be configured on the computer's side [10]. Creating a DHCP server requires a MAC address, which is a physical property of the Teensy's Ethernet hardware. QNEthernet automatically retrieves it, while NativeEthernet does not and it is necessary to find it in local registers. Configuration of a Domain Name System (DNS) or subnet is optional, and automatically performed by the library.

Once a connection has been established with a client, default communication occurs through the Transmission Control Protocol (TCP). TCP is the most widespread communication protocol on the internet and provides reliable error corrected exchange of bytes. NativeEthernet and QNEthernet also allow for User Datagram Protocol (UDP) communication which exchanges reliability and error correction for speed and time-sensitivity. However, the intended applications of mirror alignment automation are not time sensitive. Even should faster communication be desired, UDP could also lead to unrecoverable packet losses, which would directly translate into imprecision in the servomotor's movement when operating in a step-wise manner. A final advantage of TCP is that of congestion control. If both the number of servomotors simultaneously operated and the frequency at which instructions are sent significantly increase, a TCP connection can actually be faster than UDP by properly managing packet flow so as to not saturate the connection.

Establishing an Ethernet connection directly between the computer and the Teensy can be difficult. Although the communication will work perfectly with desktop computers and some laptops' USB to Ethernet adaptors, it simply never works with others.

A multitude of issues may contribute to this effect, but it is likely a problem with the auto-MDIX technology present in both the computer and the Teensy. Ethernet cables are designed to connect router to client, such that the output of one links to the input of the other. As is, this would not allow the connection of two clients or routers. Client to client communication can be remediated by using a crossover Ethernet cable (figure 3), where output/input wiring has been inverted, but it is generally unnecessary as most modern devices are equipped with so-called auto-MDIX technology which can auto detect whether the device is connected to a router or a client, and invert its input and output accordingly. For devices, like many modern laptops, which do not natively support Ethernet and instead rely on external USB to Ethernet dongles, this technology is not necessarily present and will vary from adaptor to adaptor.

According to the manufacturers of the Teensy chip, it natively supports Ethernet and is equipped with auto-MDIX. Very recent additions to the QNEthernet library now help to verify whether it is fully functional.

Instead of directly connecting the computer to the Teensy board, more universal and reliable control can be achieved by placing an intermediate router to which the Teensy is directly connected by Ethernet. Communication between the router and the computer can be done both through WIFI or Ethernet. This facilitates connectivity and solves all problems mentioned above.

4.3 SCPI communication protocol

Ethernet communication with the Teensy has been established using the SCPI communication protocol, in which instructions are stored as a tree. The tree is made of commands and subcommands. A command is one node of the tree, and can be called with various parameters. The call syntax (here for a command located at the third level of the tree) is as follows:

”COMMAND1:COMMAND2:COMMAND PARAM1 PARAM2 PARAM3 ...”,

where the command is sent as a single string line (not bytes). Many commands come with a contracted form like ”COMmand” where ”COM” will be interpreted as ”COMmand”. They are usually written with the short form in capital letters, and the optional characters of the long form in non-capitalized letters.

:RAWservo		
	PING?	<i>ID</i>
	READ?	<i>ID, start_reg, num_reg</i>
	WRITE	<i>ID, start_reg, param1, param2, ...</i>
	REGWrite	<i><regwrite parameters></i>
	ACTion	<i>ID</i>
	SYNCWrite	<i><syncwrite parameters></i>
:SERVO		
	POSition?	<i>ID</i>
	MOVing?	<i>ID</i>
	SPEED?	<i>ID</i>
	POSITION	<i>ID, target_position</i>
	ID	<i>ID, new_ID</i>
:TEENsy		
	ECHO?	<i>Message</i>
	LIST?	
	SCAN	

Figure 5: Current SCPI syntax. Command categories are bolded, while parameters are italicized and comma-separated. Parameters for REGWrite and SYNCWrite are not given here for clarity, and correspond to the servomotor’s instruction parameters.

The tree, as currently implemented is given in figure 5. It features two levels, organized around three categories:

- **SERVORAW:** unfiltered interface to the servomotor instructions. Commands are not checked and are simply forwarded as is to the motor. Using these, clients can control the motor as desired, even if not implemented in the Teensy’s program.

- **SERVO:** These instructions are slightly higher level instructions, and either transform, verify the data sent to the motor, or translate it into multiple elementary instruction calls.
- **TEENSY:** These instructions are not meant for the motor but the Teensy chip itself.

Here are a few call examples:

- Modify servomotor 0's ID to 101:

```
"SERVO:ID 0 101\n"
```

though it would also be possible to do so as:

```
"RAWservo:WRITE 0 0x05 101\n"
```

- Move servo 101 by a full rotation:

```
"SERVO:POSition 101 4096\n"
```

- Get servo temperature:

```
"RAWservo:READ? 101 0x3F 1\n"
```

- Scan and get a list of reachable servomotors:

```
"TEENsy:SCAN\n"  
"TEENsy:LIST?\n"
```

4.4 Controller program

As previously mentioned, the Teensy microcontroller will run, when active, a routine to receive Ethernet communications (in SCPI format), process them and transfer the byte instructions through serial to the motor.

More concretely, a communication handler object receives and parses messages which arrive from the computer over Ethernet or serial, processes them into proper servo-motor instructions, then sends them out through its serial port. SCPI instructions, single line strings, uses the SCPIparser library to route the parameters to the right callback function. Communicator objects handle the server-client connection and compile received messages until they can be processed. The MotorController object keeps track of the connected motors and converts its method calls into servo-motor-readable output serial communication.

Details of how this code functions is left out of this report, as it has been thoroughly commented and is self-explicit.

5 System debugging

A major part of this project was centered around identifying problems and finding their sources and possible solutions. Issues arose both at the hardware level and at the communication level. This part will detail some ways of debugging the setup.

5.1 Ethernet protocol

The Ethernet communication operates through a few key protocols for connection and data exchange in which many packets are exchanged between client and server. Understanding these packets and observing them is key to solving some of the communication issues.

For this, and there are many other options, Wireshark can be used to capture outgoing and incoming packets from your computer. By selecting IP filters, only those addressed to and from the Teensy and its chosen IP can be captured. However, some important protocols occur as a "broadcast" and no IP capture should be set when trying to debug them.

It may also be useful to capture all packets on the Ethernet local network in case the Teensy and computer cannot find each other. Doing this requires the capture software to be in "promiscuous mode" where packets not addressed to the computer will also be seen.

There are three important steps where issues can be seen and understood: the DHCP IP assignment step, the TCP three way handshake, and the TCP data exchange step. In the following sections, the expected packets for each protocol, as well as their roles are detailed.

5.1.1 The Direct Host Configuration Protocol

If a connection is failing to be established between client and server, it is possible that the DHCP did not conclude. DHCP automatically assigns an IP address to a requesting client for a later connection. It consists of four packets:

- DISCOVER: the client (our computer) broadcasts a message to the network (Ethernet LAN) requesting an IP.
- OFFER: the server (the Teensy) replies to the client with an IP address offer.
- REQUEST: the client broadcasts that it accepts this offer
- ACKNOWLEDGE: the server responds to the client and confirms the IP address assignment.

5.1.2 The TCP Three Way Handshake

Once an IP address has been assigned, a TCP connection must be established to allow for communication. It is known as the three way handshake:

- SYN: The client requests synchronization for data exchange.
- SYN-ACK: The server acknowledges this requests and accepts it.
- ACK: The client acknowledges the acknowledgement.

Similarly, when a connection is closed, three packets are exchanged as FIN, FIN-ACK, and ACK. This three way handshake is also responsible for initiating the "sequence number" that will be useful for the data exchange.

5.1.3 The TCP data exchange

TCP data exchanges are resilient to packet losses, data corruption, duplicate packets, or out of order packets by keeping track of a sequence number related to the total bytes sent. The sender emits the packet with the current sequence number appended with the number of bytes sent with the packet. The receiver acknowledges receipt by sending an ACK packet containing the current sequence number plus the number of bits received. Reception of an acknowledgment packet and detection of discrepancies in the sequence number is what allows for a reliable communication.

An exchange of data should be contained in two packets when properly functioning: one containing the data and one acknowledging its reception. Errors in the data exchange can be spotted by noticing duplicate packets (resent either due to detecting discrepancies in the sequence numbers or lack of acknowledgment.) The type of error can be deduced from the packet's sequence numbers.

5.2 Serial communication

Servomotor communication can be verified by using the URT-1 debugging chip provided by its manufacturers. It allows direct control and observation of the internal registers of the servomotor, as well as dynamic tests of its movement.

It is also useful to observe the voltages using a voltmeter along various parts of the serial communication, from the Teensy chip, to the underlying PCB board, Ethernet cable and up to the motor. Setting the Teensy's serial pins to HIGH or LOW - corresponding to the 1 and 0 of communication, will help ensure that the proper signal is getting to the final destination. Namely, the STS3032 servomotor requires a signal voltage between 0V and 0.45V for a LOW bit and 2V-5V for a HIGH bit. Over long Ethernet cables, and when dividing the signal among multiple servomotors, it may be required to add initial amplification on the PCB chip. How much amplification is necessary is deduced by how much voltage reaches the motors.

Pin identification can also be performed through this means, when installing Ethernet onto the Teensy chip or when verifying which Teensy pins output the serial communications.

5.3 Teensy damage

The voltages required to power the Teensy and servomotor (4.8-6V) are significantly higher than what can be applied to most pins of the Teensy board. When the Teensy is powered without a USB connection, power is supplied through the 5V pin. However, only the 5V and Vusb pins support such voltages, while all other pins are only 3.3V tolerant. Voltages upwards of 4 Volts to some of these pins can instantly kill the Teensy board.

Over the project, one Teensy died, likely due to a short within the chip. When a Teensy stops responding, a multitude of issues can be to blame. A good indicator to assess the damage to the chip is by verifying where a short occurred. Measuring the voltage difference between GROUND and the 3.3V pins should always read 3.3V. If a short has occurred, this number will be either 0V for a metal short or 0.5-0.9V for a semiconductor short. In the case of a metallic short, it is possible to be recovered by finding and removing the short. A

semiconductor short is likely inside the Teensy's chip and indicates a dead, and therefore irreparable chip.

6 Future possibilities

6.1 User interface

The program created for the Teensy during this project exposes an interface for any device to control it using a set of SCPI commands, either through a direct Ethernet connection or through an intermediate router. A human interface can therefore be easily developed by simply connecting to the Teensy's IP address and sending SCPI strings. Developing such an interface is a necessary continuation of this project. A first step would be expanding the program client (computer) side by keeping track of connected motors, assigning them a name and registering of their position over time. A simple graphical interface in which a certain number of steps to move can be input, for example.

The use of an intermediate router instead of a direct Ethernet connection has multiple advantages for usability. It does not limit which devices can control the motors, nor the number of devices. It also extends the possible distance between the computer and the Teensy. Furthermore, if a router is used, one may run a graphical interface as an HTML page on the Teensy, so that any device could connect and control the mirrors without having a locally installed program to do so. This would allow control using a smartphone, for instance.

It may also be interesting for users to have a more "hands-on" control of the mirrors (and not just input a set number of steps to move), which could be achieved by connecting a joystick or controller to the computer, and directly translating joystick movement to motor operation. Such extensions can once again be easily implemented by simply detecting joystick input and sending the appropriate SCPI instruction.

6.2 Multi-motor operation

It will, of course, be necessary to connect multiple motors to a single Teensy chip, as most optical components have multiple degrees of freedom.

One notable limitation is that of ID allocation. An instruction packet sent over the serial connection contains the instruction itself along with the ID of the motor we wish to control. All devices attached to the serial connection receive this message but only those with the required ID execute it. Two servomotors with the same ID on the same network will respond in the same way to all serial messages, and are therefore indistinguishable from the point of view of the Teensy.

However, all servomotors start by default on ID number 0. If two unconfigured servomotors are connected at the same time to the serial network, it will not be possible to assign them separate IDs. This must be circumvented by configuring the servomotors one by one, either by using the URT-1 chip, or by connecting them one by one to the Teensy chip and assigning them a non-zero ID at each new connection.

The control of multiple motors can be enhanced by the creation of higher level routines to decompose more complex movements of the mirror in the movement of each individual

motor. For instance, the reflection of an incident beam off of the mirror occurs at a certain solid angle Ω . Changing Ω to any other target angle will require movement of multiple degrees of freedom of the mirror and thus multiple servomotors. A higher level program could take as input by which angle (or distance on a certain plane) the reflected beam should move, and order the motors accordingly. To extend previous ideas, a joystick could thus control directly the angle of the mirror using multiple motors. Since it has two degrees of freedom, any angular motion of the mirror could be mapped to it.

This becomes increasingly interesting as the complexity increases, like with fiber injection, for example.

6.3 Synchronous operation

Most current tests have been performed asynchronously. An instruction was sent to a motor, and the motor's reply was received before the next instruction was issued. As the number of motors increases and communication with motors must become more frequent, latency limits may begin to appear. Combining the communication time and motor response time, we can expect the maximum frequency of asynchronous control of the motors to be below 1kHz.

For all uses relative to manual alignment, this is more than enough and latency would be completely acceptable. It is also likely that the STS3032 servomotors would not be appropriate to use in measurements, where both time and spatial precision are required. However, should an application be limited by the operation speed of the current setup, the use of REGWRITE, SYNCWRITE and ACTION could become interesting.

7 Conclusion

Automated control of hard to access optical components through the use of affordable servomotors and microcontrollers can be performed with relative ease using the Feetech STS3032 servomotor and the Teensy 4.1 chip. Their price comes at a fraction of that of a commercial motorized mount, all while providing perfectly adequate performance [2]. While the system is still not experimental usable as it is, a solid base of code was developed for the Teensy to allow for a reliable Ethernet communication and direct control of the servomotor movement through SCPI commands. Some issues also remain unresolved. Notably, the exact cause of the difficult connectivity between some laptops and the Teensy is still a mystery, and an elegant solution to the issue of servomotor ID assignment remains to be found. The use of Ethernet cables in lieu of serial ones was also explored but, though possible and effective, no properly tested and functioning electronic platform for this exists as of the end of this project. Beyond the aforementioned issues, continuation of the project could come in the form of multi-motor control, as well as a human interface for more user-friendly control of the motors.

References

- [1] *Thorlabs kinematic mounts*. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_ID=793. Accessed: 2023-12-05.
- [2] Lisa Peter. *Semester project Servo-based mirror motor*. 2022.
- [3] *Feetech STS3032 datasheet*. <https://www.feetechrc.com/Data/feetechrc/upload/file/20211019/6377025719620052024350972.pdf>. Accessed: 2023-12-11.
- [4] *Feetech URT-1 tutorial*. Accessed: 2023-12-12. URL: <https://feetechrc.com/Data/feetechrc/upload/file/20201127/start%20%20tutorial201015.pdf>.
- [5] *Teensyduo*. Accessed: 2023-12-01. URL: <https://www.pjrc.com/teensy/teensyduino.html>.
- [6] *Feetech SCS15 manual*. Accessed: 2023-12-01. URL: <https://grobotronics.com/images/companies/1/datasheets/SCS15&SCS115%20Manual.pdf?1516269264467>.
- [7] *Teensy documentation*. Accessed: 2023-12-12. URL: https://www.pjrc.com/teensy/td_libs.html.
- [8] *Teensy forum*. Accessed: 2023-12-12. URL: <https://forum.pjrc.com/index.php>.
- [9] *Teensy 4.1 manufacturer website*. Accessed: 2023-12-12. URL: <https://www.pjrc.com/store/teensy41.html>.
- [10] *Configuration protocol for the ethernet shield to connect to another client*. Accessed: 2023-12-11. URL: <https://alselectro.wordpress.com/2017/05/01/ethernet-shield-connecting-with-laptop-instead-of-direct-router/>.

8 Appendix

EEPROM memory	
Byte Hex (decimal)	Register name
0x00 (0)	FIRMWARE MAJOR
0x01 (1)	FIRMWARE MINOR
0x03 (3)	SERVO MAJOR
0x04 (4)	SERVO MINOR
0x05 (5)	ID
0x06 (6)	BAUDRATE
0x07 (7)	RESPONSE DELAY
0x08 (8)	RESPONSE STATUS LEVEL
0x09 (9)	MINIMUM ANGLE (H)
0x0A (10)	MINIMUM ANGLE (L)
0x0B (11)	MAXIMUM ANGLE (H)
0x0C (12)	MAXIMUM ANGLE (L)
0x0D (13)	MAXIMUM TEMPERATURE
0x0E (14)	MAXIMUM VOLTAGE
0x0F (15)	MINIMUM VOLTAGE
0x10 (16)	MAXIMUM TORQUE (H)
0x11 (17)	MAXIMUM TORQUE (L)
0x13 (19)	UNLOAD CONDITION
0x14 (20)	LED ALARM CONDITION
0x15 (21)	POS PROPORTIONAL GAIN
0x16 (22)	POS DERIVATIVE GAIN
0x17 (23)	POS INTEGRAL GAIN
0x18 (24)	MINIMUM STARTUP FORCE (H)
0x19 (25)	MINIMUM STARTUP FORCE (L)
0x1A (26)	CLOCKWISE INSENSITIVE RANGE
0x1B (27)	COUNTERCLOCKWISE INSENSITIVE REGION
0x1C (28)	CURRENT PROTECTION THRESHOLD (H)
0x1D (29)	CURRENT PROTECTION THRESHOLD (L)
0x1E (30)	ANGULAR RESOLUTION
0x1F (31)	POSITION CORRECTION (H)
0x20 (32)	POSITION CORRECTION (L)
0x21 (33)	OPERATION MODE
0x22 (34)	TORQUE PROJECTION THRESHOLD
0x23 (35)	TORQUE PROTECTION TIME
0x24 (36)	OVERLOAD TORQUE
0x25 (37)	SPEED PROPORTIONAL GAIN
0x26 (38)	OVERCURRENT TIME
0x27 (39)	SPEED INTEGRAL GAIN

Table of the STS3032's EEPROM registers.

RAM memory	
Byte Hex (decimal)	Register name
0x28 (40)	TORQUE SWITCH
0x29 (41)	TARGET ACCELERATION
0x2A (42)	TARGET POSITION (H)
0x2B (43)	TARGET POSITION (L)
0x2C (44)	RUNNING TIME (H)
0x2D (45)	RUNNING TIME (L)
0x2E (46)	RUNNING SPEED (H)
0x2F (47)	RUNNING SPEED (L)
0x30 (48)	TORQUE LIMIT
0x37 (55)	WRITE LOCK
0x38 (56)	CURRENT POSITION (H)
0x39 (57)	CURRENT POSITION (L)
0x3A (58)	CURRENT SPEED (H)
0x3B (59)	CURRENT SPEED (L)
0x3C (60)	CURRENT LOAD (H)
0x3D (61)	CURRENT LOAD (L)
0x3E (62)	CURRENT VOLTAGE
0x3F (63)	CURRENT TEMPERATURE
0x40 (64)	ASYNCHRONOUS WRITE STATUS
0x41 (65)	STATUS
0x42 (66)	MOVING STATUS
0x43 (67)	CURRENT CURRENT

Table of the STS3032's RAM registers.

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SERVO-BASED MIRROR MOTOR

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

LEMPEREUR

First name(s):

STEPHAN

With my signature I confirm that

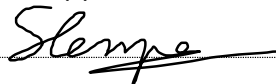
- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 30/01/2024

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.