MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION
FEDERAL STATE AUTONOMOUS
EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
"MOSCOW POLYTECHNIC UNIVERSITY"
/MOSCOW POLYTECHNICAL UNIVERSITY/

Faculty of Urban Studies and Urban Economy

Department: "Electrical equipment and industrial electronics"

Direction 13.03.02 "Electric Power Engineering and Electrical Engineering"

EXPLANATORY NOTE

FOR THE FINAL QUALIFICATION WORK ON THE TOPIC:

"Development of an extreme manipulator control system using a stepper motor and TMC2130 driver"

Student of group 201-421 _____ / Kankwanda D. D /
signature Full name

Head of the final qualifying work,
Senior Lecturer,
Department of "EiPE" _____ / Varlamov D. O. /
academic staff, academic level, position signature Full name

Norm control _____ / Kuksa V.V. /
signature Full name

The following are allowed to defend the final qualifying work:

Head of the Department
of "EiPE",
PhD, Associate Professor _____ / Shishkov A. N. /
academic staff, academic level, position signature Full name

«___»_____2024

FEDERAL STATE AUTONOMOUS
EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
"MOSCOW POLYTECHNIC UNIVERSITY"
/MOSCOW POLYTECHNICAL UNIVERSITY/

Faculty of Urban Studies and Urban Economy
Department: "Electrical equipment and industrial electronics"
Direction 13.03.02 "Electric Power Engineering and Electrical Engineering"

I approve

Head of the Department of "EiPE", PhD, Associate Professor _____ / Shishkov A. N. /
academic staff, academic level, position — signature — Full name

«___»_____2024

EXERCISE

FOR THE STUDENT'S FINAL QUALIFICATION WORK

Kankvanda Daglox Daglox groups 201-421
(last name, first name, patronymic, group number)

1. Subject of the final qualifying work: Development of an extreme manipulator control system using stepper motor and driver TMC2130

Approved by Order No. " " dated "__" _____2024.

2. Initial data for the final qualifying work Literary sources, technical description, experiments in the laboratory,

3. Contents of the explanatory note (list of relevant issues)
List of symbols; Introduction; Chapter 1. Theoretical information; Chapter 2 Practical part. Conclusion; List of used literature; Appendices.

4. List of graphic materials (with precise indication of required drawings)
Drawing of the device; Specification; Electrical diagrams; Economic calculation; Experimental graphs; Calculated tables of characteristics, etc.

Date of assignment issue "___"_____2024

Head of the final qualifying work,
Senior Lecturer, Department of "EiPE" _____ / Varlamov D. O. /
academic staff, academic level, position — signature — Full name

Norm control _____ / Kuksa V.V. /
signature — Full name

The student accepted the assignment for execution _____ / Kankwanda D.D. /
signature — Full name

CONTENT

Sub.						
Exchang						
Inv.						
Sub.					*MPU.13.03.02.201-421.10.00.P3*	
	Lit	*Change*	*Document no.*	*Sub. Date*		
	Developed	*Usernkwanda D.*			*Development of extreme manipulator control systems using a stepper motor and TMC2130 driver*	*Lit* / *Sheet* / *Sheets*: *0* / *74*
	Prov.	*Varlamov D. O.*				
	T. counter.					*Moscow Polytechnic University*
Inv.	*N. counter.*	*Kuksa V.V.*				*Group 201-421*
	Approved by	*Shishkov A.N.*				

ABSTRACT

Final qualifying work __71__ p., ____ drawings, _5__ tables, _10_ sources, __3_ appendices, _0_ pages of graphic material.

Keywords: _____

_____

_____

The object of the study **is:**_____

_____

_____

Purpose of the work: definition of common configuration values _____ TMC2130 drivers for starting and controlling a stepper motor. _____

Research methods: _____

_____

Provide the conclusions of the work: _____

_____

_____

_____

_____

_____

_____

INTRODUCTION

The modern world is in full development in all corners planets. Following the development of industrial technology in several developed countries we have entered the era of the fourth generation of development. This era is based on a new technology called artificial intelligence. In in the industry over several years the development of automation has been focused on smart machines that can operate without continuous human participation. Such machines include robotic arms or mechanized manipulators supported by artificial intelligence. These mechanized manipulators, in many ways equal in complexity, made it possible to solve many problems of mechanical movement, which made it possible to perform tasks that great industrial machines could execute. (Add the date of creation of the first robot).

Industrial manipulators are complex in design, but also in management, but they all work one by one and have complex structures in depending on the model. In general, they consist of several servos or steppers engines that can be controlled according to a specific scheme. Selection engines depends on the purpose of the manipulators. Engines can provide different degrees of freedom, and servo or stepper motors have different clamping moment corresponding to the load they can bear support, size, materials and management system. Many large companies specialize in creating robotic manipulators of various types. One of the important points for these industrial generation, which is mentioned above, is the integration an intelligent system capable of learning specific movements depending on the tasks at hand

As a final year BSc Electrical Engineering student, I I chose a project that allows me to understand the basic principles

construction and operation of an industrial manipulator experimentally.

In this work, the integration will be presented experimentally driver capable of controlling a stepper motor with high precision. To create an industrial manipulator based on a specific task we will determine the important parameters that need to be taken into account, we will choose driver and we will use its capabilities, and also develop a system control of one or more stepper motors.

1. THEORETICAL PART

In this section we will look at the preliminary concepts of stepping engine, its characteristics, important parameters, general mathematical model, existing models in the general market, and the choice of engine for our experiment. Then the concepts of the stepper driver will be developed engine, driver selection and description of some of its characteristics.

After defining the two previous concepts, we will move on to the concept communication between the engine and the driver, which will focus on SPI interface, and the final part will be devoted to the description of the complete diagrams of our device, including a microcontroller (selection and its characteristics) and driver.

1.1 STEPPER MOTOR

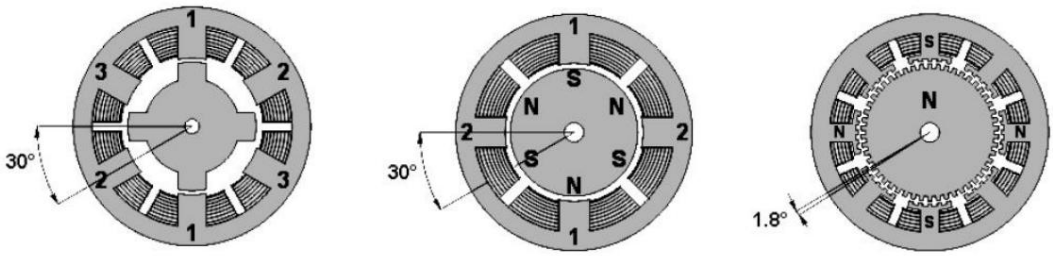
1.1.1 General information about stepper motors

A stepper motor is a type of electric motor that converts digital electrical signals into mechanical movement. difference from other devices capable of performing the same or similar functions, the stepper motor control system has a number of significant advantages: firstly, it does not require feedback, usually necessary for controlling position or speed of rotation; secondly, it does not accumulate errors in position; thirdly, the stepper motor compatible with modern digital devices.

There are different types of stepper motors, including variable speed motors. magnetic resistance, permanent magnet motors and hybrid engines.

The variable reluctance stepper motor has several poles on the stator and rotor of a toothed shape made of soft magnetic material material. There is no rotor magnetization (see Figure 1, a).

A permanent magnet stepper motor consists of a stator, which has windings, and a rotor containing permanent magnets (see Figure 1, b). The moving poles of the rotor have a rectilinear shape and are located parallel to the engine axis. A hybrid engine combines both previous engines.



A b V

Figure 1 - Example of a stepper motor with variable speed magnetic resistance (a), permanent magnet motor (b) and hybrid engine (in)

Depending on the number of excitation windings in each phase, the stepper motor can be bipolar, unipolar or four-winding. Bipolar motor has two windings and, accordingly, four terminals, which for changing the direction of the magnetic field must be added by the driver (see Figure 3, a). [3]

A unipolar motor has two windings, one in each phase, but a tap is made in the middle of the winding (see Figure 3, c). Several unipolar Motors have 4, 5, 6, 8 or more power leads (see Figure 2).[3]

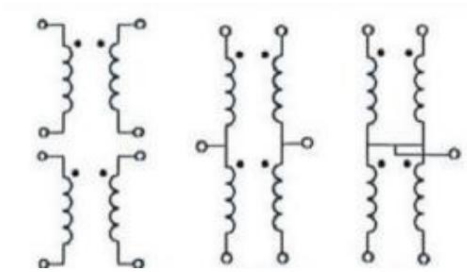


Figure 2 - Example of several unipolar configurations

engine

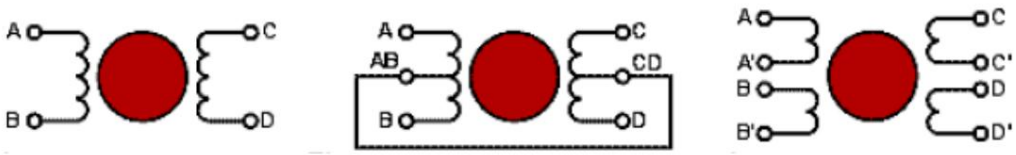


Figure 3 - Types of motors according to the winding diagram in each phase. (a), Bipolar motor, (b) unipolar motor and four-winding motor (c)

engine

In the diagram, the stepper motor is designated in this way according to GOST 2.722-68 ((see. Figure 4).

Figure 4 - Stepper motor designation

1.1.2 Mathematical modeling of a stepper motor

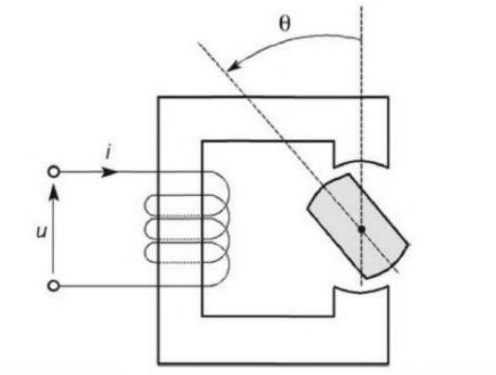


Figure 5 - Simplified diagram of the stator and rotor of the stepper motor

From Kirchhoff's law we write the equations of electrical equilibrium

voltage and EMF of the stator windings of the stepper motor:

$$= \quad _1 + \quad \frac{1}{} + \quad _{12} \ddot{y} \quad \frac{2}{} - \quad _1 \qquad (1)$$

$$= \quad _2 + \quad \frac{2}{} + \quad _{12} \ddot{y} \quad \frac{1}{} - \quad _2 \qquad (2)$$

where is the phase voltage;

— active phase resistance;

1, 2—currents of phases 1 and 2, respectively;

12— mutual inductance of phase windings;

1 and 2— induced EMF in coils of phases 1 and 2, respectively. [5]

The EMF in the coil of phase 1 and 2 is determined by the following expressions:

$$_1 = \ddot{y} \ddot{y} \quad \sin(\ ) \qquad (3)$$

$$_2 = \ddot{y} \ddot{y} \quad \sin(\ddot{y}\ ) \qquad (4)$$

where = is the angular velocity of the rotor;

— number of rotor pole pairs;

— maximum flux linkage;

— rotor position;

— phase shift angle. [5] According to our phase shift angle = $\frac{}{2}$.

$$= \quad _1 + \quad \frac{1}{} \ \ddot{y}\,\ddot{y} \ \frac{}{} \quad \sin(\ ) \tag{5}$$

$$= \quad _2 + \quad \frac{2}{} + \ \frac{}{} \quad \ddot{y} \quad \cos(\ ) \tag{6}$$

The electromagnetic moment created by the stator phase windings can be be defined by the following expression:

$$em1 \ ^{=\ddot{y}} \quad ' \ _1 \sin \text{ and } em2 \quad ^{=\ddot{y}} \quad ' \ 2 \cos \tag{7}$$

$$_{Em} = \quad em1 + \quad em2 \ ^{=\ddot{y}} \quad ' \ _1 \sin + \ddot{y} \quad 2 \cos \tag{8}$$

$$= \ \frac{nom}{\ddot{y}\ nom\ddot{y}2} \tag{9}$$

Equation of moments acting on the rotor:

$$\frac{}{} \ \ddot{y} \ \frac{2}{2} + \ddot{y} + = \quad _{Em} - \quad n \tag{10}$$

where = + $_n$ — the total moment of inertia reduced to the rotor shaft;

$\frac{2}{2}$ — angular acceleration of rotor rotation;

— coefficient of viscous friction;

— moment of sliding friction in bearings;

$_n$ — moment of load resistance [2]

The system of equations of the mathematical model of a stepper motor in in demonic mode can be written as follows:

$$= \quad _1 + \quad \frac{1}{} \ \ddot{y}\ \ddot{y}\ \ddot{y} \qquad \sin(\ )$$

$$= \quad _2 + \quad \frac{2}{} + \ddot{y}\ \ddot{y} \qquad \cos(\ )$$

$$\frac{}{} \ \ddot{y} \ \frac{2}{2} + \ddot{y} + = \ddot{y} \qquad ' \qquad \sin + \ddot{y}\ 1 \qquad 2 \cos \ddot{y} \qquad _n \qquad (11)$$

$$\{ \qquad = \frac{}{}$$

Let us consider a simplified mathematical model for a two-phase stepper motor (see Figure 6).

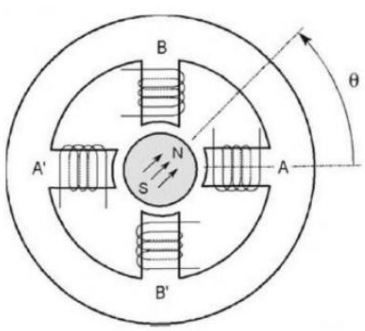


Figure 6 - Simplified diagram of a two-phase stepper motor.[2]

The combined action of the stator magnetic field and the stator current induces electromagnetic torque on the rotor of a stepper motor. By feeding sequence of voltages on each phase, the rotor will take one step.

As a result, the step size will be equal to [1]

$$= \quad \frac{2\ \ddot{y}}{} \ , \qquad (12)$$

In expression (11) a mathematical model was created using Matlab, and it looks like this (see Figure 7).

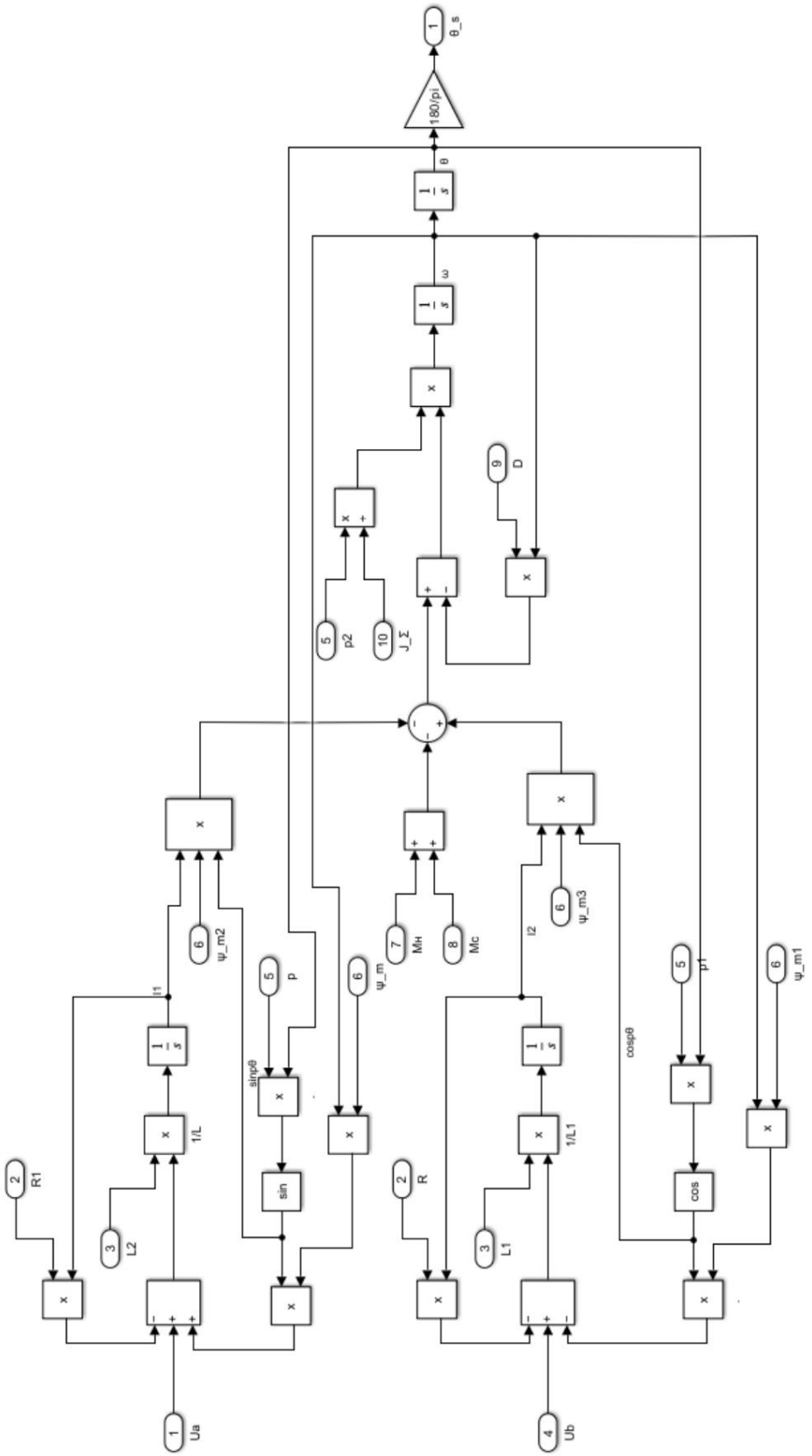


Figure 7 — Structure of the mathematical model of a two-phase hybrid stepper motor.

1.1.3 Stepper Motor Control Methods

The fundamental task of stepper motor control is to ensure rotor rotation at a given angle. The minimum possible angular step is called movement of the stepper motor rotor. When one rotor control signal is supplied, moves, takes one step, that is, turns at an angle that is embedded in its design [6] and 2 ÿ in full step mode. There are several methods of controlling a stepper motor:

- Full step mode
- Step fractionation (half-step and micro-step control mode)
- Vector control

Full step mode is a control mode in which current is supplied immediately to two windings of the stepper motor. In this mode, the stepper motor will provide a greater value electromagnetic moment (see Figure 8).

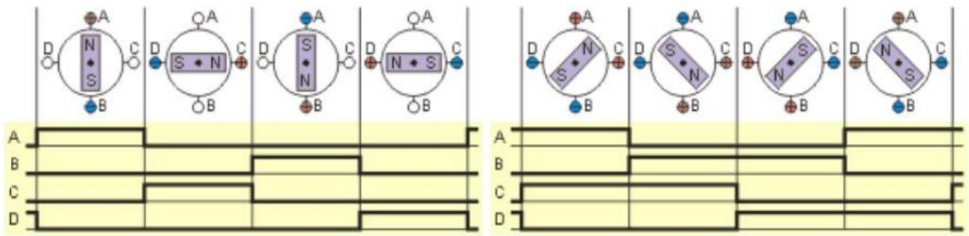


Figure 8 - Full-step mode, (a) one phase is on and (b) two phases are included.[7]

Split mode is a control mode in which the stepper motor is controlled by reducing the minimum angle of rotation of the rotor engine.

In half-step mode, voltage is supplied to two phases of the motor. In this case, the rotor rotates at half the angle it should. turn at full step engine control mode, that is, on half a step.

In microstepping mode, voltage is also supplied to two windings. immediately, only now the windings are supplied with current of different values and change according to the law of sin and cos (see Figure 9). The meaning of microstepping mode lies in the supply of a signal to the stepper motor windings, which in its form resembles sinusoid [4]. At the present moment, there are many different microstepping modes, with step sizes from ½ to 1/256

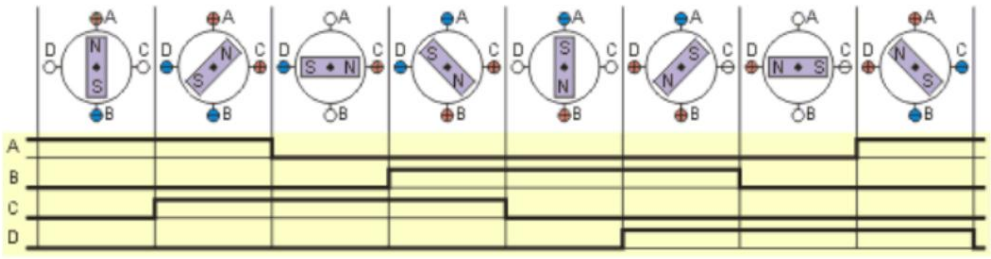


Figure 9 - Half-step mode [7]

1.1.4 Parameters and main characteristics of the stepper motor

The main parameters of the stepper motor are:

• Step (step value) is the equivalent value of the rotation angle rotor of a reactive stepper motor.

$$= = \frac{2}{} \quad (13)$$

Where m is the spatial number of shifted control windings;

Zp = 2p – number of rotor teeth;

p – number of pole pairs.

• Maximum torque M Dependent on static moment at different values of current in the control windings from the angle

rotor rotation relative to the initial position

stable equilibrium determines the static characteristic.

$$M(\alpha) = M_{max} \sin \alpha \qquad (14)$$

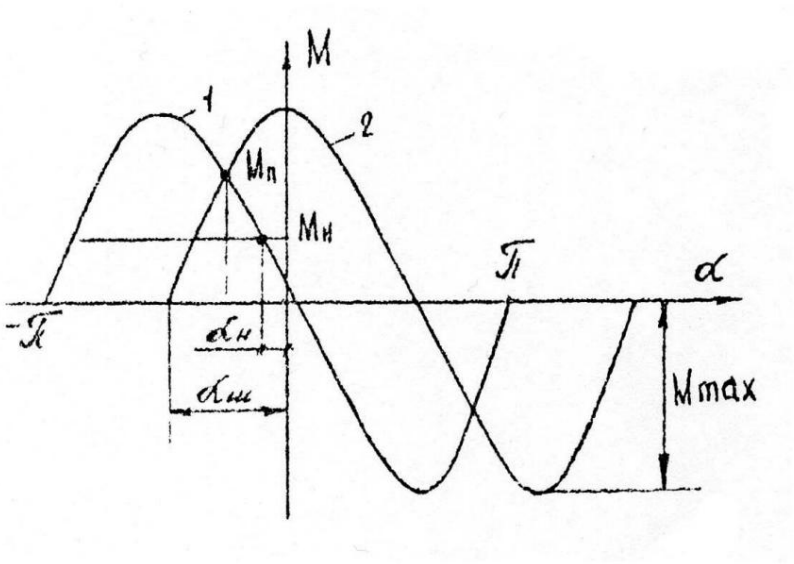


Figure 10 - Static characteristic of the stepper motor or Dependence electromagnetic torque acting on the stator of a bipolar stepper motor at an angle rotor rotation.

When two static characteristics are in position (see Figure 10), shifted relative to each other by the value , it is possible to determine maximum starting torque. The load torque at which the maximum starting torque is possible. starting and further operation of the engine without missing pulses is equal to the ordinate intersection points of the synchronizing torque curves for two adjacent stable states 1 and 2 can be seen from Fig. 10.

- Starting torque is the greatest load moment at which starting and further operation of the engine without misfiring is possible impulses.
- Engine stability is the ability of the stepper motor to operate without loss steps
- Phase number

- Current in the winding in each phase
- Rotor inertia
- Supply voltage
- Winding resistance
- Winding inductance
- Maximum pulse frequency

The mechanical characteristic of the stepper motor (see Figure 11) is called dependence of the torque developed by the engine on the frequency of the control impulses. This dependence can be defined as follows:

$$_{n} = \frac{}{1+ 2ÿ(1 \quad 2+ \quad \frac{2}{2})} \qquad (15)$$

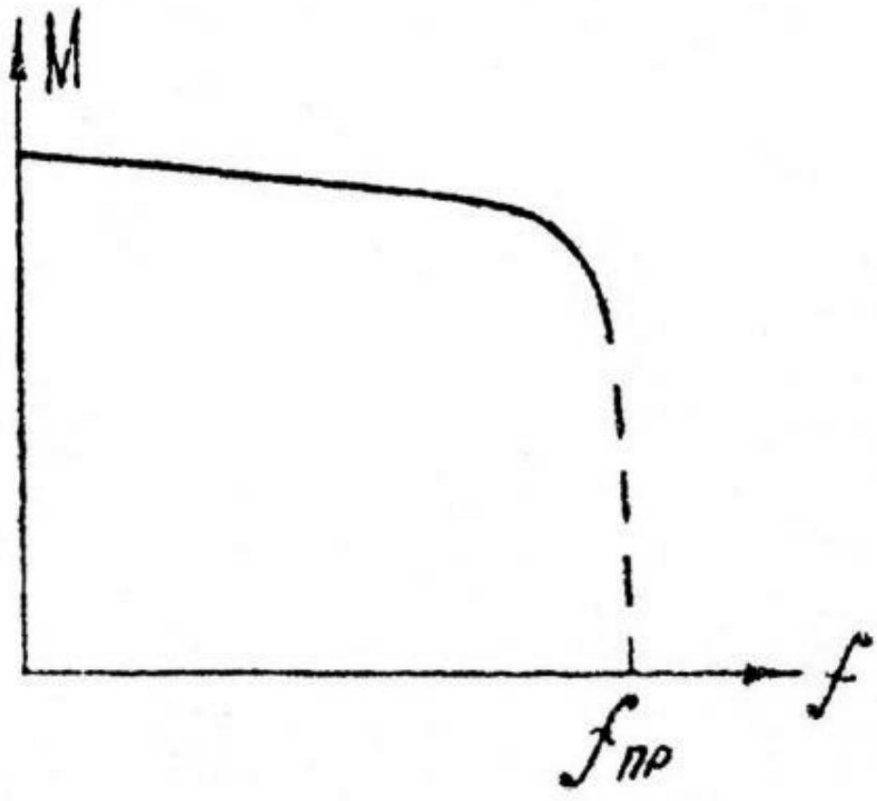


Figure 11 — Mechanical characteristics of the stepper motor

1.1.5 Description of the Nema 17 stepper motor

Below (see Figure 12) is a complete description of Nema 17.

Standard shaft motor shown.

Phases	2
Steps/Revolution	200
Step Accuracy	±5%
Shaft Load	20,000 Hours at 1000 RPM
Axial	25 N (5.6 lbs.) Push 65 N (15 lbs.) Pull
Radial	29 N (6.5 lbs.) At Flat Center
IP Rating	40
Approvals	RoHS
Operating Temp	-20° C to +40° C
Insulation Class	B, 130° C
Insulation Resistance	100 MegOhms

Figure 12 - Nema 17 Single Stack Description.

Nema 17 - bipolar hybrid stepper motor has the following parameters:

- Minimum rotation angle: 1.8°
- Step: 200
- Number of phases: 2
- Maximum current in the winding in each phase: 2 A
- Maximum load moment: 0.480 N.m
- Winding inductance: 2.1 mH
- Winding resistance: 1.04 Ohm
- Rotor inertia: 0.038 ÿ 10ÿ410 kg.m²
- Maximum Supply Voltage: 48V

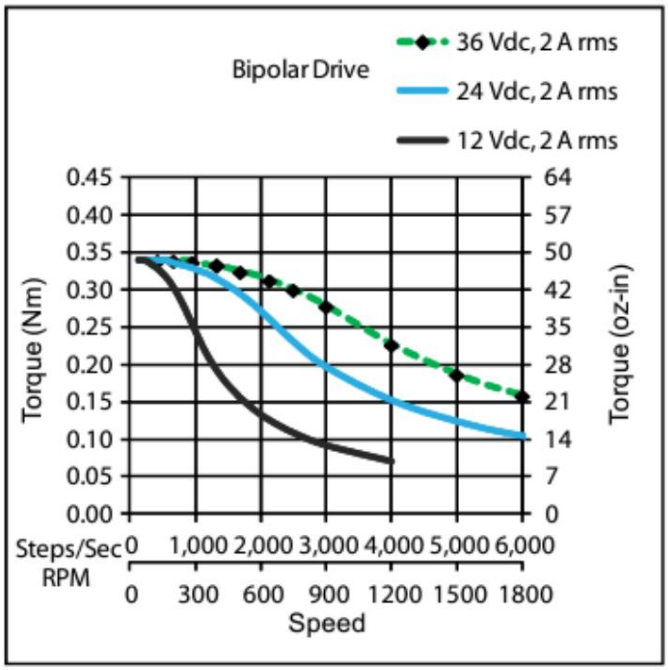


Figure 13 — Dependence of the stepper motor torque on the rotation speed

different supply voltages.

1.2 . STEPPER MOTOR DRIVER

1.2.1. TMC2130 Driver Description

Stepper Motor Driver - An electrical device that is based on digital control signals control high-voltage power windings of the stepper motor and allows the stepper motor to take steps (rotation). All perfect drivers have standard output signals STEP/DIR control, Controls the stepper motor in direction and ENABLE, which Activates the control method.

- STEP — signal BUS by number of steps
- DIR — signal of the direction of the SD

There are many different types of stepper motor drivers. In our work we we will use TMC2130.

TMC2130 is a high-performance stepper motor driver from the company TRINAMICs have the following characteristics:

- 2 phase control (maximum current in phase winding is 2.0 A)
- Supply voltage limit 4.75…46 V
- SPI interface
- Microsteps 256 (9 different movement resolutions: full step, ½ step, ¼ step, 1/8 steps, 1/16 steps, 1/32 steps, 1/64 steps, 1/128 steps and 1/256 steps.
- StealthChop mode
- SpreadCycle mode
- StallGuard2 mode
- DcStep mode
- CoolStep mode
- Protection against short circuit to ground and load.

There are three ways to connect the TMC2130 driver, which otherwise called operating modes: STEP/DIR mode (driver mode), standalone mode and SPI mode (driver mode).
In our work, the stepper motor will be connected to the driver in the mode STEP/DIR (drive mode).

In STEP/DIR mode (driver mode), the driver controls the motor with using the STEP and DIR signals it receives from the microcontroller, as shown (see Figure 14).

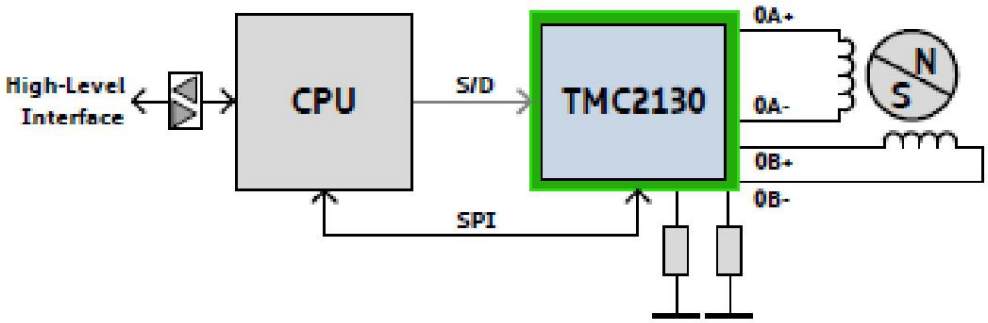


Figure 14 — TMC2130 connection diagram in STEP/DIR driver mode mode [8].

In addition to the characteristics specified in the documentation, the driver has several features described in the structural diagram (see Figure 15).

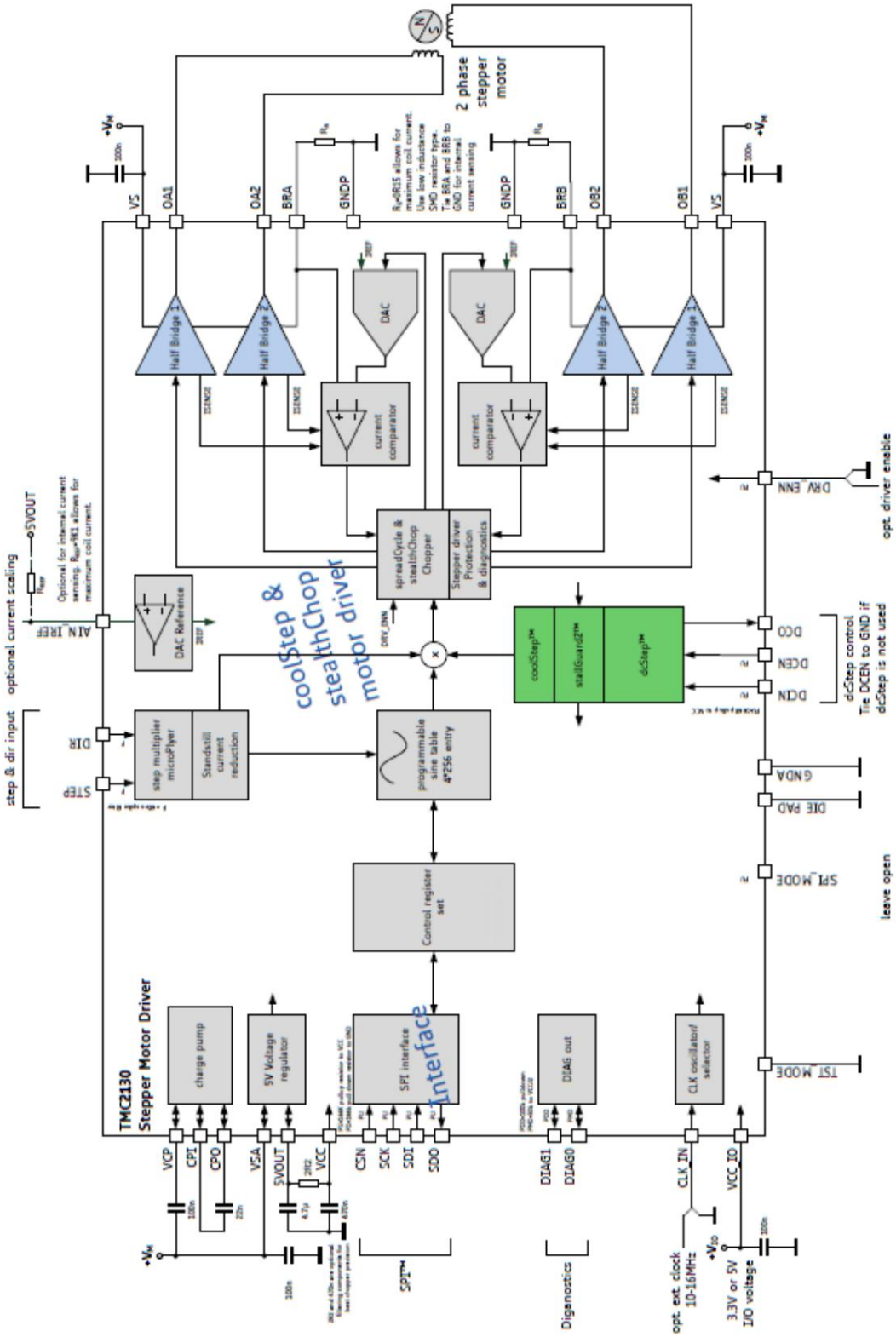


Figure 15 — Structural diagram of the TMC2130 driver [8]

The selected prepared printed circuit board driver has the following

Conclusions:

Table 1 Description of TMC2130 driver pins

Conclusion	Description	Conclusion	Description
GND	Earth	EN	Enable Permissions drivers
VM	Power supply of stepper motor	SDO	MISO : Slave Output – host entrance
VIO	Driver power supply	SDI	MOSI : Master Output - slave input
M1A	Coil Input A	SCK	Signal output synchronization
M1B	Winding input B	CS	MC activation input as slave device
M2A	Coil Output A	DCIN	dcStep control input
M2B	Winding B Output	DCEN	Enable dcStep mode
STEP	BUS signal by number of steps	DIAG0	Diagnostic input 0
DIR	Sinhalese directions of the school	DIAG1	Diagnostic input 1

1.2.2. SPI interface

Serial Peripheral Interface (SPI)
Interface) implemented in all microcontrollers, is used for two
main goals:

1. It can be used to exchange data between
microcontroller and various peripheral devices such as
digital potentiometers, DAC/ADC, shift registers, various
sensors, etc. This interface can also be used to
data exchange between multiple AVR microcontrollers.

2. Programming can be done via SPI interface
microcontroller (the so-called serial mode)
programming) [9]

The SPI interface uses four pins of the microcontroller or
master device. Example of a simple connection between two devices via SPI
is given (see Figure 16).

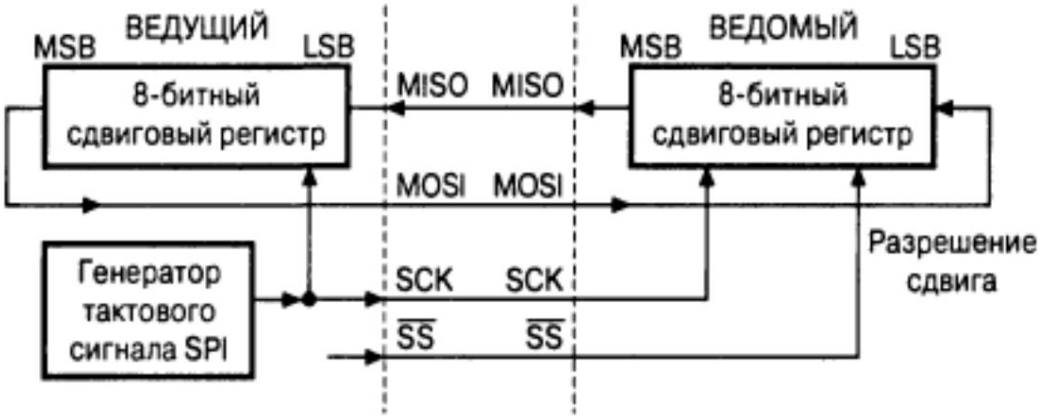


Figure 16 - SPI connection diagram for two devices

Data transfer via SPI interface is shown in (see Figure 17-18)
is carried out in this way: the SCK output of the master device starts
generate a clock signal and the data starts to be output bit by bit
MOSI output and accordingly go to the MISO output of the slave device.

The order of data bit transmission is determined by the state of the DORD bit.
SPCR register. If the bit is set to 1, the least significant bit is transmitted first.
byte if reset to 0 - the most significant. After issuing the last bit of the byte
the pulse generator is stopped by setting the flag to 1. The SPIE bit from
SPCR register will allow a break after the previous one is received
bytes if set to 1. After this, the master device can start
transmit the next byte or allow the slave to transmit
data to the master. The remaining bits of the SPCR register are described in (Table 2).

Table 2 SPCR register

Bit	Name	Description
7	SPIE	Enable interrupt from SPI
6	SPE	Enable/Disable SPI
5	DORD	Data transfer procedure
4	MSTR	Selecting the operating mode (Master/Slave)
3	CPOL	Clock signal polarity
2	CPHA	Clock phase
1.0	SPR1 ;SPR0	Transfer speed

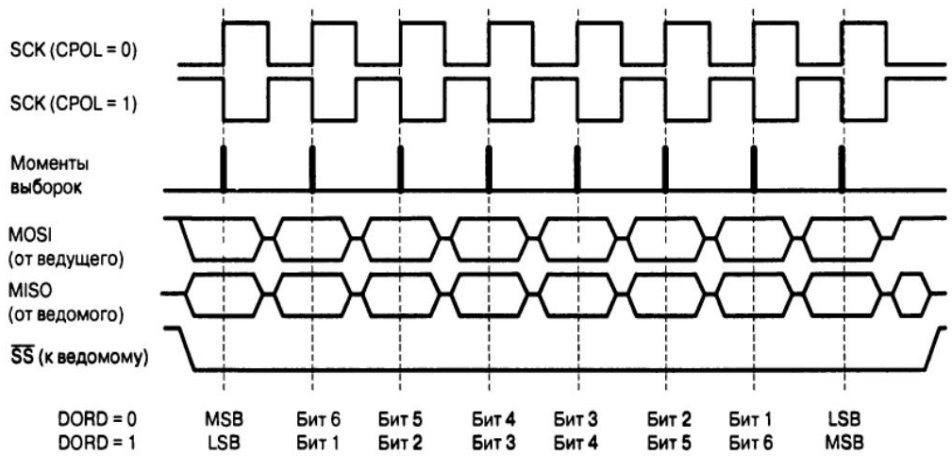


Figure 17 - Data transfer with CPHA=0

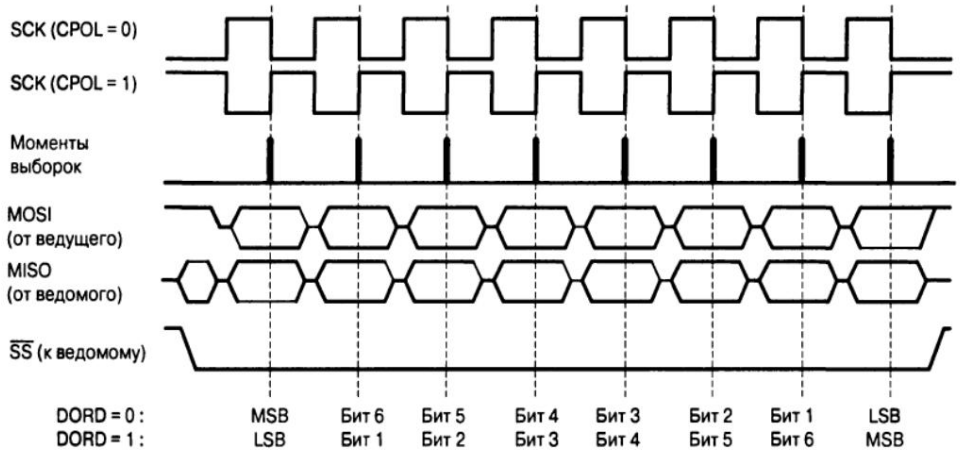


Figure 18 - Data transfer with CPHA=1

1.2.2. SPI interface in TMC2130 and its own registers

The TMC2130 driver contains SPI diagrams for sending 40 bits of data using the SPI interface. The driver transmits five consecutive bytes to the microcontroller and back. SPI structure diagram Drivers is represented as follows:

The first byte determines the transfer mode (write or read) from the master device (microcontroller) and the address of the register in which the data should be be read or written. To be written, the first bit of the first byte must be set to 1 and to read to 0. The remaining 32 bits are used for data transfer.

The TMC2130 driver has the ability to transfer data to microcontroller and back at the same time. SPI structure diagram is given (see Figure 19)

SPI DATAGRAM STRUCTURE				
MSB (transmitted first)		40 bit		LSB (transmitted last)
39 ...				... 0
→ 8 bit address ← 8 bit SPI status	← → 32 bit data			
39 ... 32	31 ... 0			
→ to TMC2130 RW + 7 bit address ← from TMC2130 8 bit SPI status	8 bit data	8 bit data	8 bit data	8 bit data
39 / 38 ... 32	31 ... 24	23 ... 16	15 ... 8	7 ... 0
w / 38...32	31...28 / 27...24	23...20 / 19...16	15...12 / 11...8	7...4 / 3...0
39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Figure 19 — SPI structure diagram of the TMC2130 Driver

The TMC2130 driver has 24 registers. Most of these registers are used to configure the driver.

Table 3 TMC2130 driver registers

	Method of the regime transmissions	Registry Name Address	register	Maximum number bits data
1	Write/read GCONF		0x00	17
2	Reading with reset	GSTAT	0x01	3
3	Reading	IOIN	0x04	16
4	Recording	IHOLD_IRUN	0x10	14
5	Recording	TPOWER DOWN 0x11		8
6	Reading	TSTEP	0x12	20
7	Recording	TPWMTHRS	0x13	20
8	Recording	TCOOLTHRS	0x14	20
9	Entry	THIGH	0x15	20
10	XDIRECT Write/Read		0x2D	32
11	Entry	VDCMIN	0x33	23

12 Entry	MSLUT [0…7]	0x60… 0x67	8x32
13 Entry	MSLUTSEL	0x68	32
14 Entry	MSLUTSTART	0x69	16
15 Reading	MSCNT	0x6A	10
16 Reading	MSCURACT	0x6B	18
17 Write/read CHOPCONF		0x6C	32
18 Entry	COOLCONF	0x6D	25
19 Entry	DCCTRL	0x6E	24
20 Reading	DRV_STATUS	0x6F	32
21 Entry	PWMCONF	0x70	22
22 Reading	PWM_SCALE	0x71	8
23 Entry	ENCM_CTRL	0x72	2
24 Reading	LOST_STEP	0x73	20

All data registers return in the 0x00 state when the driver powered off. The GCONF register is read and written the main driver state. (Bit 2 enables the bus mode, bit 4 changes direction of rotation of the stepper motor, bit 16 enables the method determining the magnitude of currents in the windings and the direction of currents through the register XDIRECT.

One important register is IOIN, with which we can check the status of SPI communication between drivers and microcontrollers. Reading the fourth byte (31st to 24th bit) should always show the byte, equal to 0x11. The CHOPCONF register is used to define microsteps, full step frequency.

1.3 . GENERAL SYSTEM

1.3.1. Description of Atmega32Ua4 and Pro Micro Arduino

Microcontroller Microchip (Atmel) ATmega32U4 (see Figure 20) —
chip on the AVR architecture with 8-bit resolution and a clock frequency of 16 MHz.

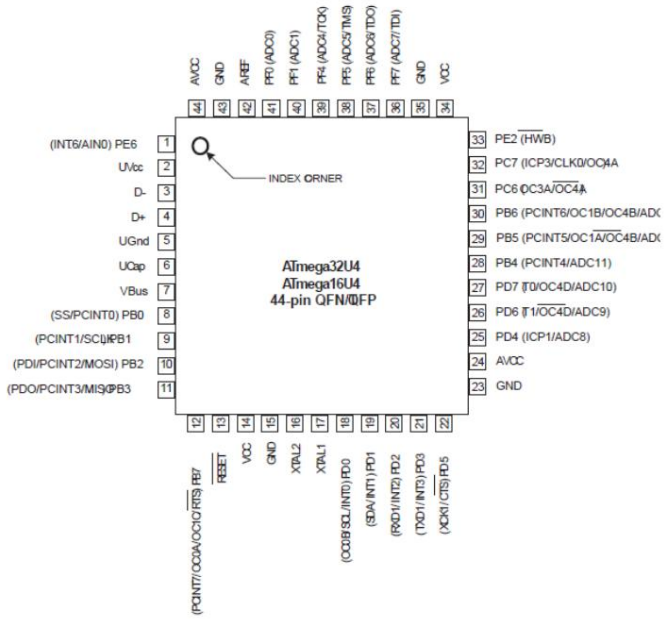


Figure 20 — Schematic diagram of the atmega32u4 microcontroller

ATmega32U4 has the following features:

- 8-bit megaAVR architecture.
- Clock frequency up to 16 MHz, external generator support
  impulses
- 135 instructions, most of which are executed in 1 clock cycle
- 2.5 KB SRAM, 32 KB Flash and 1 KB EEPROM memory.
- Supply voltage from 2.7 to 5.5 V
- 10-bit ADC on 8 channels
- Supports USB 2.0 (Full Speed) interface.
- Hardware interfaces (USART, I2C(TWI) and SPI)

• JTAG interface

• 32x8 registers

• Timers/counters: (2 x 8 bit and 1 x 16 bit)

Arduino Pro Micro (see Figure 21)) is built on a microcontroller ATmega32U4 and behaves like a standard Arduino Leonardo with additional functions.

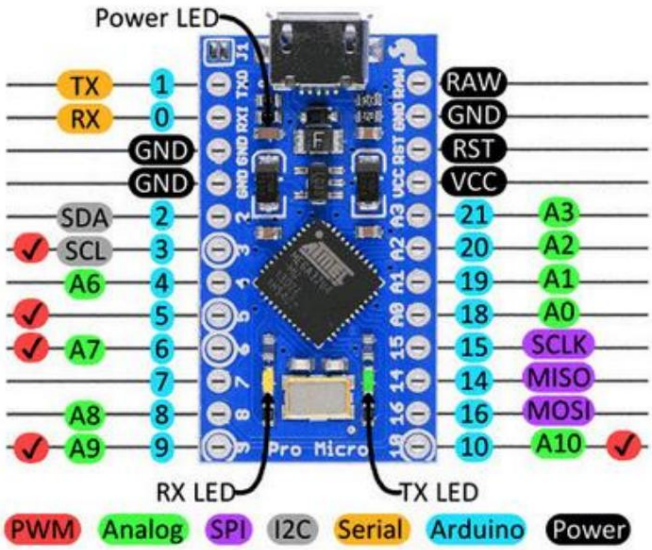


Figure 21 - Arduino Pro Micro

1.3.2. General diagram of the driver and microcontroller

From the existing schematic (see Figure 21) of the Arduino Pro Micro and TMC2130 drivers, resulting in a general circuit that allows see common connections. The driver connects to the controller via the SPI interface via pins 9, 10 and 11, and the slave device select pin is defined on D10. The trigger pulse is generated by one output pin of the Arduino, connected to the STEP pin. The DIR direction and EN enable pins controlled by two other pins 8 and 9. This circuit is shown in figure (see figure 22).

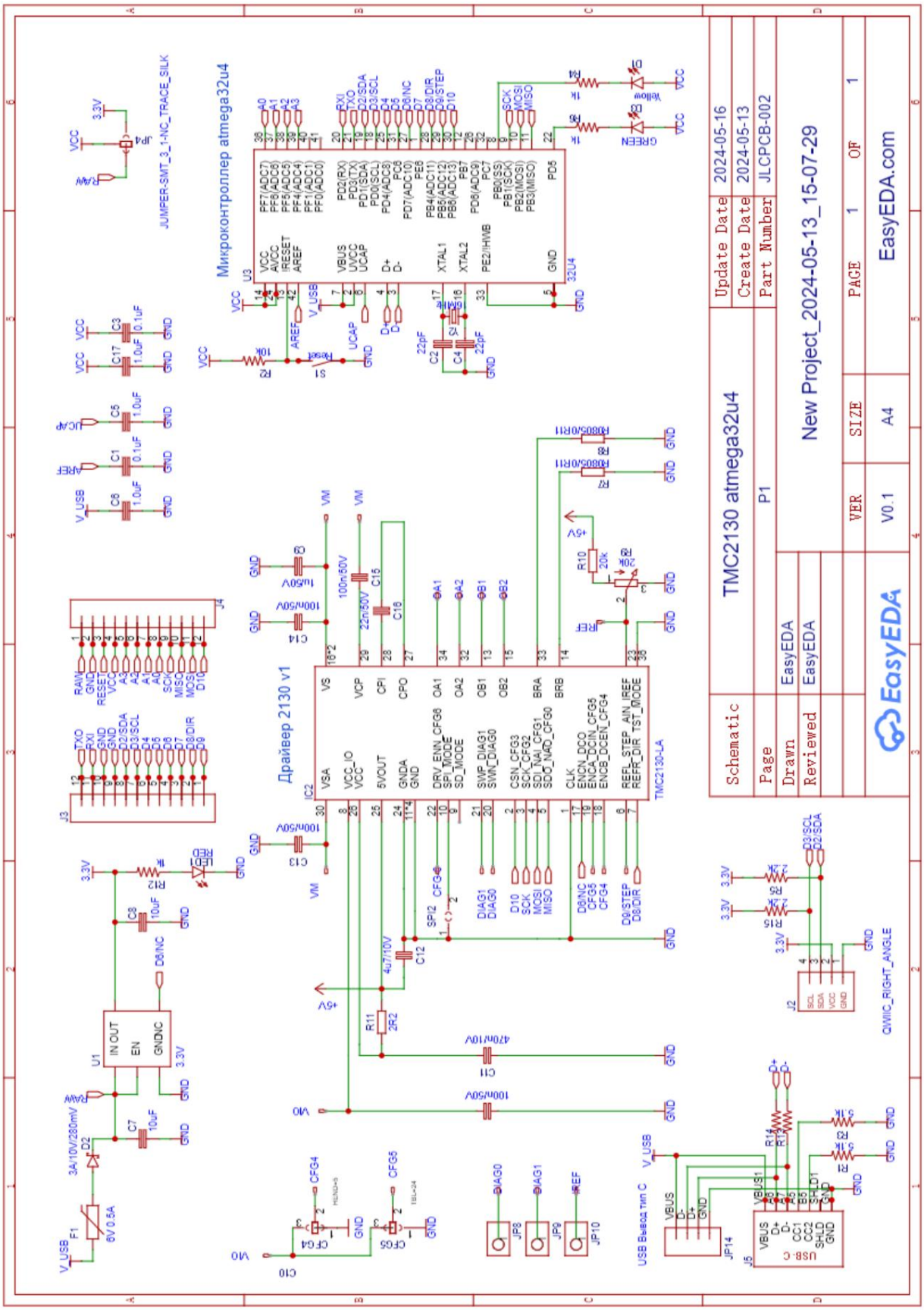


Figure 22 - Complete connection diagram of the driver and the microcontroller

2. PRACTICAL PART

The practical part includes the selection of components, characteristics to consider when designing a system

In general, the nominal values of components such as power supply,

data values for register setup, library reconfiguration

SPI in C++, adaptations of the TMC2130 register configuration library and

Adding motor control features to Arduino.

2.1. SELECTION OF COMPONENTS

Our experimental project contains the minimum necessary components, and here they are:

- Arduino pro micro (due to its size, price and easy access to SPI interface pins, convenient for placement in a small frame, the perfect solution for our project)
- TMC2130 driver version 1, as described above.
- NEMA 17 stepper motor as described in (see Figure 12).
- AC-DC power supply unit with 12V maximum current 2.5A: Our driver works at a maximum voltage of 5V. However, the NEMA 17 stepper motor operates on different supply voltages (12, 24 and 36 V), as described in (see Figure 15). For our purpose, 12V is more than enough engine control. The driver supports a maximum of 2.0 A current. Therefore, the choice of a 12 V power supply with a rated power of 2.5 A is the optimal solution.
- DPDT pushbutton switch (allows switching on the common circuit)
- 12V fan for driver cooling
- Printed circuit board area 90 x 70 mm

- Connectors 4 outputs B4B-PH-SM4-TB(LF)(SN)
- 2-output connector

Table 4 – Quantity and cost of components

Component	Type	Quantity (pcs.) Price, in rubles	
Arduino	Micro pro	1	612 rub
Driver TMC2130 v.3		1	712 rub
NEMA 17 Stepper Motor		1	700 rub
AC-DC Power Supply Unit	Converter 12V 10A	1	834 rub
Button switching	DPDT	1	106 rub
12V DC Fan		1	
PCB	PCB 90×70 mm green	1	370 rub
Board stand	H-L1700-2300-503-1N1W	4	144.88 RUB
Connectors 4 outputs Plug	corner on board S4B-XH-A 2.5 mm 4	1	15 rub
Connector 2 outputs	JST XH-2 2 pin mom	1	13 rub

The complete system of our experimental project is presented at (see figure 23).

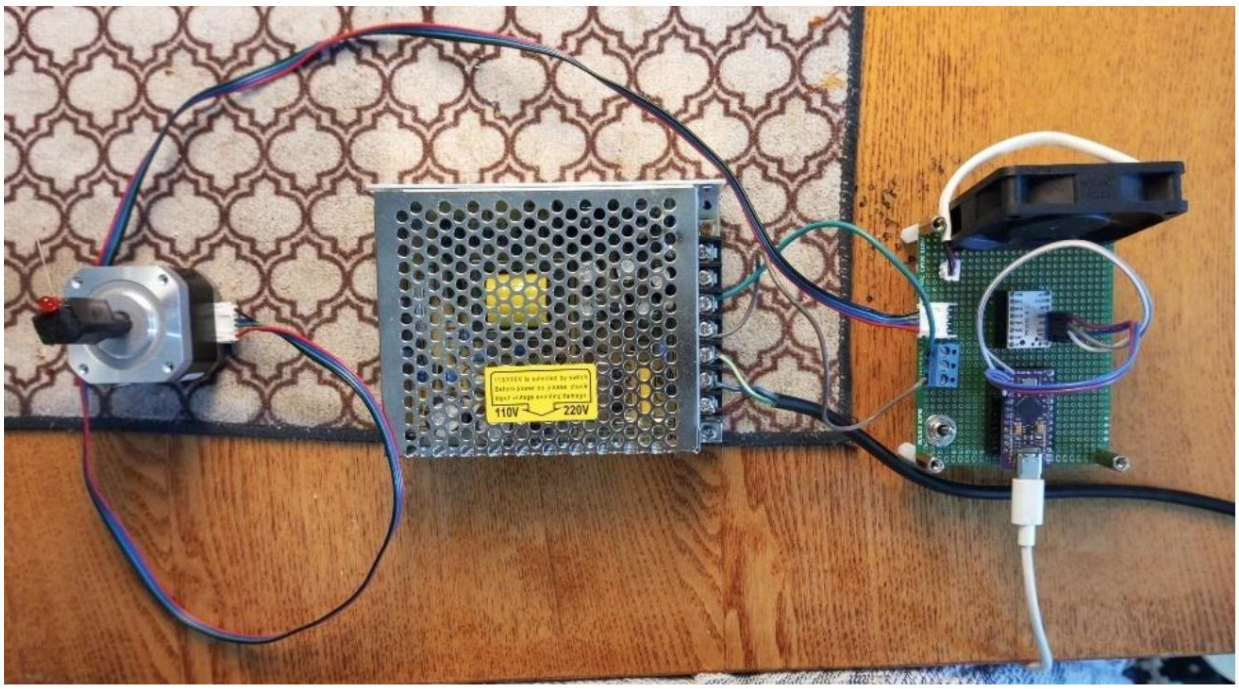


Figure 23 - The complete system of our experimental project

2.2. DRIVER REGISTER CONFIGURATION

Driver configuration begins with defining description values SineWave drivers via MSCT and MSLUTSEL registers. SineWave, shown (see Figure 24) on the driver, is a sinusoidal curve describing the step. The registers control only quarter of this curve. The MSLUTSEL register defines the quarter of the curve, and MSCNT adjusts the curve according to the desired parameters.

The MSLUTSEL register consists of 32-bit data that are represented as follows: X3-X2-X1-W[W3-W2-W1-W0]. In W 2-bit parameters are used to describe the way the quarter is constructed curve (add or decrease). 0b00 - decrease the curve by -1 step, 0b01 - increase by +1 step, 0b10 - increase by +2 steps, 0b11 - increase by +3 step. A quarter of the curve is constructed as described in (Fig. 26).

To simplify the experiment, we will define a default value MSLUTSEL in register = 0xFFFFFFFF. This means: X3 = 0xFF = 255, X2 = 0xFF = 255, X1 = 0xFF = 255, W = 0xFF. In this case, the points X1, X2, X3 are identical. Thus, the curve correction is carried out by 8 registers MSCNT[0...8], which are the table of microsteps. By default These registers are set to 0xFFFFFFFF. 2 bits of each register represent the default curve adjustment. Quarter Curve limited on the y-axis by two values defined by the START_SIN bytes (bits: 7...0) and START_SIN_90 (bits: 23...16) of the MSLUTSTAR register.

Let's set the value to 0x0F0000, and we get START_SIN = 0 and START_SIN90 = 16.

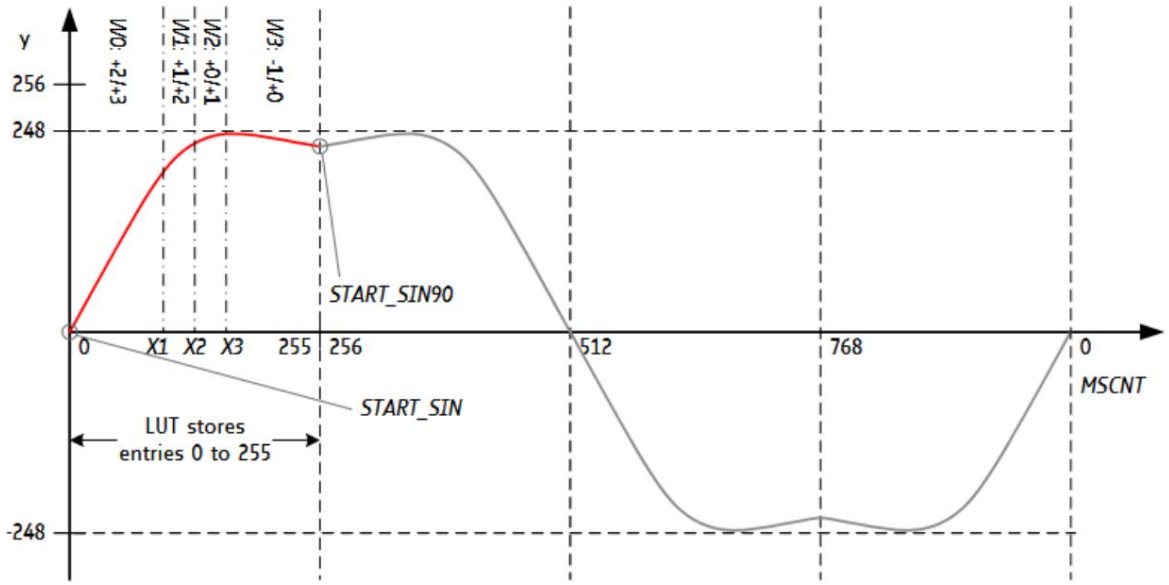


Figure 24 —Sine-Wave curve [8]

We need to define the default frequency value for the full step. This is the definition value from the PWMCONF register. First, we must define the divisor, which is determined by bits 16 and 17 PWM_FREQ=%11. Based on the fact that Arduino uses a pulse generator (oscillator) 16 MHz, we get the frequency.

$$= 16 \text{ ÿ } \frac{2}{410} = 78.0 \text{ kHz.} \quad (16)$$

One of the most important values for our project is definition of the value of the steps generated by the driver. It corresponds to the value assigned to the TSTEP register. Its value is:

$$= \quad / \quad 256 \quad (17)$$

The value of fSTEP256 is determined using the CHOPCONF register, configurates 4 bits of MRES as follows: 128, 64, 32, 16, 8, 4, 2, Full step for 0b0000…0b1000. Our driver supports up to 128 microsteps, which are distributed over 256 interpolation values that must be enabled via bit 28 intpol.

256 = 256 ÿ 2^ (18)

By default, we set MRES: 0b0001.

The current generated by the driver at the output plays an important role in determining the value of the support moment created by the stepper engine. The driver produces an average current , proportional current values in each phase. Its value is determined by the following way:

$$= \frac{+1}{32} \cdot \frac{}{+20\ \ddot{y}} \cdot \frac{1}{\ddot{y}2} \quad (19)$$

Where CS is the current ratio constant for each of the phase currents, which must be defined using register parameters IHOLD_IRUN, which are IHOLD (4…0 bits) and IRUN (8…12 bits) with the following value: 0=1/32…31=32/32. — this is the value of the full scaling voltage, determined by the vsense bit of the CHOPCONF register. For vsense = 1, V_Fs=180 mV, and for vsense = 0, V_Fs=325 mV. Thus, knowing the maximum value of CS=31 and the value of vsense, we can determine meaning from the table (Selecting Sense Resistors [8]).

For vsense = 0, = 0.1 ÿ we get ÿ 1.92

The current generated in the motor is:

$$= \frac{/}{248} \cdot \frac{+1}{32} \cdot \frac{}{+20\ \ddot{y}} = \frac{/}{248}\ \ddot{y} \qquad \ddot{y}\ \ddot{y}2 \quad (20)$$

/ represents the current micro-step motor current for phases A and B, defined by the MSCURACT register (CUR_A: bit 24…16 and CUR_B: bit 8…0), which can be divided into 255 bits. Thus, in order to change the value of the internal current of the motor to adjust the torque, It is enough to change the two previous parameters. By default we will set CUR_A: 0x1F and CUR_B: 0x1F0000, which corresponds to the value 3 and is 12.5% of the average current.

The TMC2130 driver has an operating mode called Coolstep, which allows you to automatically adjust the current depending on the set restrictions. This adjustment is used when the engine is carrying a load, exceeding the torque created by the current. This results in decrease in current. And if this current exceeds the stipulated limits, CoolStep mode regulates the current. The regulation is carried out in accordance with with how so in (see figure 25)

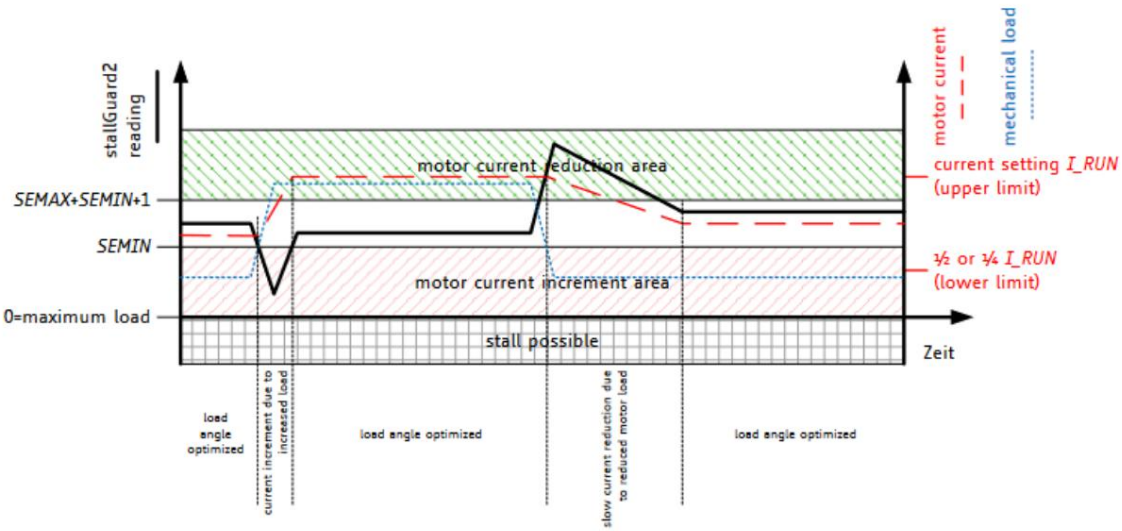


Figure 25 — Adaptation curve of the motor phase current to the load in the mode CoolStep [8]

The setting is done using the COOLCONF register. How shown in (see Figure 25), the current values in phases A and B of the motor should not go beyond the interval [SEMIN, SEMIN+SEMAX+1] * 32. SEMIN and SEMAX correspond to bits 4…0 and bits 11…8 respectively. SEUP and SEDN define the recovery value when the interval is exceeded. That is yes, if the current is below SEMIN, the current automatically increases to the value SEMIN+SEUP, and similarly, if the value SEMIN+SEMAX+1 is exceeded, the current decreases to the value SEMIN+SEMAX+1-SEDN. SEIMIN corresponds to the current value in intelligent control.

For our project, we will define its values as follows:

- SEMIN: 0b1111
- SEMAX: 0b1111
- SEUP: 0b0110
- SEDN: 0b0110
- SEIMIN: 0b1000

This corresponds to the value of the COOLCONF register: 0xFF6F6F

The TMC2130 driver documentation provides an example of the initial configuration of some registers for engine start. Besides just settings made, we can use the rest of the presented settings to complement our configuration.

Enable PWM mode using the en_pwm_mode bit in the register GCONF, direction of rotation using shaft bit and direct_mode bit, which allows changing the current through the XDIRECT register rather than MSCURACT.

The last register that needs to be configured is PWMCONF. This The register allows you to adjust mainly the amplitude and frequency of signals PWM. The default value is 0x00050480. This corresponds to the amplitude value PWM_AMP = 128 and the amplitude gradient PWM_GRAD = 4. The next two bits represent the frequency division factor, generated by the controller as defined above. The remaining parameters can be left with default values. This changes the meaning register at 0x00030480.

Thus, the amplitude value that was generated is is determined by the following formula:

= _ + _ ÿ 256 / (21)

Our driver offers three important parameters (SGT, sflit and SG_RESULT), which allow measuring the resulting moment

engine under load using stallGuard2 mode. SGT is generated 7
bits of the COOLCONF register, representing a value from -64 to +63. This
The value indicates the motor's sensitivity to resistive load.
The higher this value, the lower the sensitivity of the engine. This value
must be adjusted at each change in phase current or at
changing microsteps. Sflit in the same register allows to increase the accuracy
sensitivity measurements. And the SG_RESULT value in the register
DRV_STATUS is formed by 10 bits with a range from 0 to 1023. It
represents in real time the effect of load on the total moment
motor. A high value indicates low torque as shown in
the following illustration (see Figure 26).

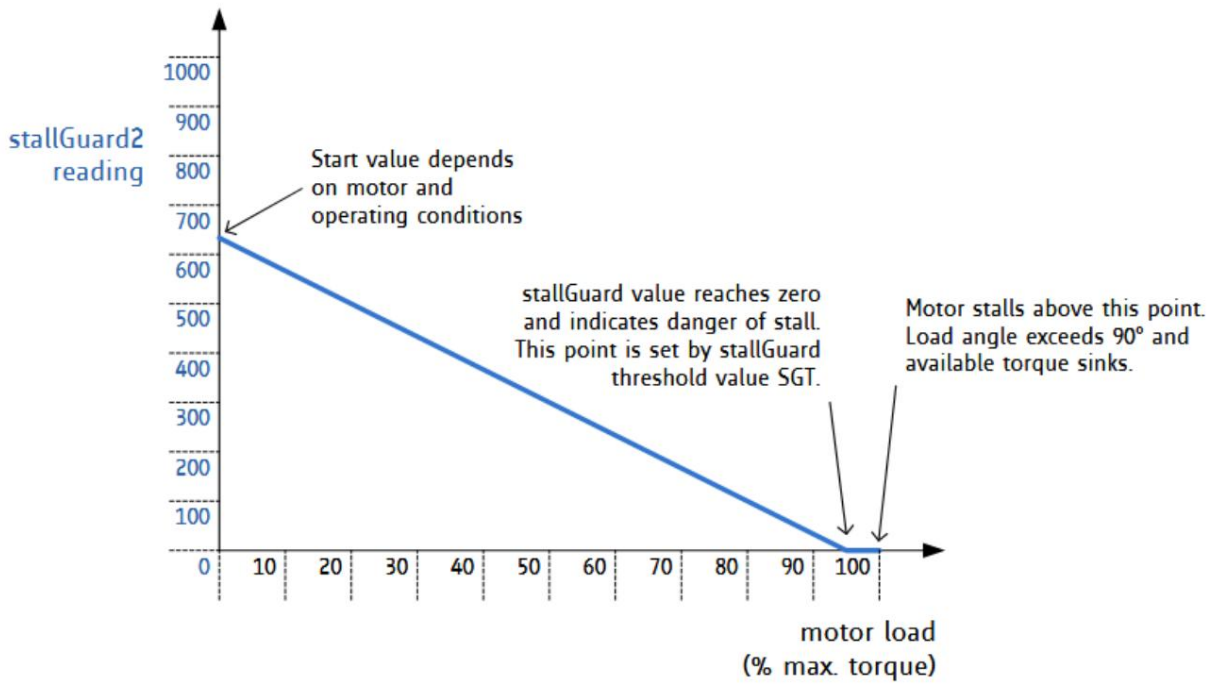


Figure 26 — Operating principle of the StallGuard2 mode for measuring torque rotor

2.3. LIBRARIES AND CODE ADAPTATION

2.3.1. Adaptation of SPI library

During the implementation of this project, one of the problems that I faced was had to face, was the use of the existing SPI library to provide communication between the microcontroller and the driver. The first the next step is to check that the Arduino pin configuration is correct corresponds to the one used in the SPI library. Configuration conclusions Arduino located V file pins_arduino.h (C:\Users###\AppData\Local\Arduino15\packages\arduino\hardware\avr\1.8.6\variants###\pins_arduino.h).

```
#define PIN_SPI_SS (17)
#define PIN_SPI_MOSI (16)
#define PIN_SPI_MISO (14)
#define PIN_SPI_SCK (15)

static const uint8_t SS = PIN_SPI_SS;
static const uint8_t MOSI = PIN_SPI_MOSI;
static const uint8_t MISO = PIN_SPI_MISO;
static const uint8_t SCK = PIN_SPI_SCK;
```

Considering that the values of SS, MOSI, MISO and SCK cannot be changed, it is necessary to make sure that their values exactly match the diagram the Arduino we use. Our TMC2130 driver transmits data in both

directions, that is, it sends data through MISO and receives data via MOSI. The standard Arduino SPI library is in the SPI.h file C:\Users\###\AppData\Local\Arduino15\packages\arduino\hardware\avr\1.8.6\libraries\SPI\src\SPI. provides one-way communication. I added the transfer data via MISO as an input to the begin() function. This done as follows:

```
pinMode(SCK, OUTPUT);
pinMode(MOSI, OUTPUT);
pinMode(MISO, INPUT);
```

This library only sends 8 bits of data on each transmission. To adapt it to our driver, a transfer function is needed, capable of transmitting 40 bits. One of the existing TMC2130 libraries offers an SPI module that matches the driver scheme. For each SPI transmission sends 5 times 8 bits. The transmission function is in the module SW_SPI.h (Appendix A).

2.3.2. TMC2130Stepper Library

One of the most difficult tasks was to establish communication between the driver and a controller to configure it. There are several libraries TMC2130, I chose TMC2130Stepper. But before using it, some functions need to be rewritten and others supplemented.

The TMC2130.h file contains driver configuration functions for each register, as defined above, the connection establishment function between the driver and the controller, functions for reading and writing to registers,

which allow you to perform specific functions for each register.
Based on the read register of the general configuration driver since
GCONF, described earlier, creates the shaft() function to change
direction of rotation. To simplify writing and reading in registers
common functions are used, and the SPI data transfer function takes
as a parameter the address of the register. If it is necessary to write data to
register, 4 additional bytes are sent. Each time the
data is returned in response. This data is read-only. In
according to the description in the datasheet, to write or read data to the driver
The transfer must be performed twice.

The TMC2130Stepper.cpp file contains the send2130() function,
accepting two parameters: the addressByte register addresse-command and its 32-
bit configuration. According to the datasheet, for writing to the driver
it is necessary to set the value 1 for the 8th bit of the register. That is, adresse-
command = address-register | 0x80. To read from the register adresse-command =
the address is a register and you have to send 32 bits twice to get
the values of the bits stored in the register

MACROS.h allows generalizing reading and writing to registers with
using the name of each register and the notation previously
defined by the author of this module.

```
#define TMC_WRITE_REG(R) send2130(TMC2130_WRITE|REG_##R,
&R##_sr);

#define TMC_READ_REG(R) send2130(TMC2130_READ|REG_##R,
&R##_sr); return R##_sr

#define TMC_MOD_REG(REG, SETTING) REG##_sr &= ~SETTING##_bm;
REG##_sr |= ((uint32_t)B<<SETTING##_bp)&SETTING##_bm;
TMC_WRITE_REG(REG);
```

The two lines of code above show two macros for reading and driver entries. Values TMC2130_WRITE = 0x80, TMC2130_READ = 0x00 and the union of REG_##R and R##_sr form predefined values in TMC2130Stepper_REGDEFS.h (Appendix B). The latter contains all configurations for each register as defined above.

TMC2130Stepper.cpp provides the complete functions that allow control our driver. The send2130() function or method is already partially described above, is written as follows:

First I initialize the SPI connection using the begin() function, we are installing parameters SPI with help functions beginTransaction(SPISettings(16000000/8, MSBFIRST, SPI_MODE3)). Here SPISettings is an SPI class that allows you to configure the transmission frequency. data, MSBFIRST sets the value of the DORD bit in the SPCR SPI- register. interface. SPI_MODE3 represents the transfer mode, which affects CPOL bit. Then we select the slave device we want to communicate with via SPI, using digitalWrite(_pinCS, LOW). After the following steps are completed, initialization, configuration and slave selection, we can start transfer data using transfer() method. Read 8 bits from SPDR register is carried out as follows:

```
static uint8_t transfer(uint8_t data) {
    SPDR = data;
    while (!(SPSR & _BV(SPIF))) ;
    return SPDR;
}
```

To record, first send adresse-command, then split 32-bit configuration into 4 bytes and transmit them sequentially. For reading

first we clear the register by sending 32 bits, then we read from the desired address transfer(addressByte & 0xFF) and read the remaining 4 bytes, sending a stupid address transfer(0x0).

After data transfer we disconnect the slave device using digitalWrite(_pinCS, HIGH). We then complete the communication by setting SREG = 0 in endTransaction() function.

The function is defined, it is easy to interact with all registers. Let's start by defining the method for setting up microsteps. Microsteps are determined using the MRES register according to formula (16). Our driver can operate in nine microstepping modes. For each requested we assign values to our mres() function and send it configuration via SPI. And also to read the current value microsteps.

Next, you can define a function that modifies and reads the current value in the two motor windings. This can be done either through the average value of the effective current, or through direct currents CUR_A and CUR_B. Based on formula (17), it is necessary to create a function for definitions of IHOLD and IRUN for CS, vsense values as described above, and Rsense values (see Figure 20) are represented by the rms_current() function.

The SineWave correction function can also be redefined with using the MSCURACT and MSLUT registers.

Finally, the rest of the functions do not require processing for our project. Their configuration is carried out directly by changing the register bits without use additional features, and they can also save their default values.

2.4. GENERAL ENGINE CONTROL PROGRAM VIA ARDUINO

To perfectly control our NEMA17, we must create easy way to change rotation speed, number of steps, frequency pulse generator driver, torque, driver operating mode, current values in two coils and changes in direction.

One of the most important tasks is to determine the constants, characterizing the accuracy of the engine rotation angle.

2.4.1. Atmega32u4 Timer1

By default, the arduino.h module is configured to run at maximum the value of the clock generator that can be set on any microcontroller. In our case, this value is 16 MHz, connected to ATmega32U4.

To reimplement the time counting function, let's let's reassign a new value to the internal frequency using Timer 1 (Figure 7) In the TCCR1B register, change the WGM12:1 bit to enable CTC mode, and CS10:1 for division ratio = /1 , and also bit OCIE1A of TIMSK1 register to enable counter interrupt. AVR offers two built-in functions that allow you to enable and disable counter. These are cli() and sei().

```
void initTimer() {
    cli();
    TCCR1A = 0;
```

```
    TCCR1B = 0;
    TCNT1 = 0;
    OCR1A = 61;
    TCCR1B |= (1 << WGM12);
    TCCR1B |= (1 << CS10);
    TIMSK1 &= ~(1 << OCIE1A);
    sei();
}
```

So in our code we create two new functions millis_() and delay_(), which work the same way as the built-in millis() and delay() functions, but with a different frequency.

2.4.2. Calibration constant

For easy control of our driver we use the mode Step/Direction Driver Mode. This means that PWM pulses are generated our controller and are transmitted to the PWM output. Thanks to to redefine our frequency we open and close the PWM output in series with a frequency proportional to the frequency generated by Timer 1. The value of this frequency is determined as follows:

Our Nema17 is set to 200 steps/min for 1.8°, which corresponds to full step. Experimentally with the interval of opening and closing PWM output 10000 microseconds. At full motor step and MRES 255 the engine made a complete revolution in the time measured in the following table:

Table 5 Time required for a complete revolution

1	2	3	4	5	6	7	8	9	10
2.3	4.65 6.94 9.26 11.6 13.86 16.21 18.5 20.78 23.14								
11	12	13	14	15	16	17	18	19	20
25.47 27.75 30.11 32.26 34.66 37.01 39.32 41.63 43.91 46.26									
21	22	23	24 25		26				
48.62 50.93 53.2 55.55 57.83 60.11									

The average time between two consecutive measurements is 1.32 seconds:

$$1°\sim \frac{1.32}{360} = 0.0036 \text{ s} \quad (23)$$

$$\text{for } 1.8°\sim 0.0066 \text{ ÿ} \quad (24)$$

Due to systematic errors in measuring time, I introduced correction constant , which will be determined. For establishing correspondence between 200 steps of the motor, full step drivers and the time obtained with the above mentioned interval:

$$1 \text{ step ÿ } 200 \text{ ÿ } 0.0066 \text{ ÿ} = 1.32 \text{ ÿ } 360°$$

$$1 \text{ step ÿ } 200 \text{ ÿ } \frac{0.0066}{360} \text{ ÿ} = 0.0036 \text{ ÿ } 1° \quad (25)$$

For 255 microsteps,

$$( ) = 0.0066 \text{ ÿ } 200 \text{ ÿ } \frac{}{360} \text{ ÿ} \quad (26)$$

In expression (23), we defined the value for 1 degree as follows: so we introduce the angle of change. Then, since our time is based on counter of our microcontroller, we introduce a change based on

the interval of opening and closing of the BUS output defined above, and

We calculate time in microseconds.

$$( ) = \text{ÿ } 6600 \text{ ÿ } 200 \text{ ÿ } \frac{\text{ÿ}}{360} \frac{\text{ÿ}}{0} \quad (27)$$

Where, — the number of microsteps determined by the function

microstep() ;

— the rotation angle in degrees based on a certain

microstep and frequency of generated pulses;

- time interval between opening and closing

PWM output, determined based on our pulse module,

configured on Timer1 ;

$_0$ - Time interval between opening and closing

PWM output by default, in our case equal to 10000;

- Calibration constant for angle determination accuracy.

The calibration constant is determined experimentally, and after

after several tests it is equal:

$$= 1.5163 \quad (28)$$

The time value for performing a certain angle is determined

as follows:

```
time_move = microstep_ *(6600.0*200.0) *(angle/360) * (intevale_frequence/
(10000.0)) *1.5163;
```

The rest of the motor controller code for angle detection

is located in (Appendix B).

2.4.3. Evaluation of the change in torque according to different parameters

Our project is an experimental miniature version of the stepper motor control system for the manipulator or robotic arm. Access to important information such as current in windings, rotation speed and other parameters, is carried out through driver registers. However, one of the important characteristics is holding torque of the engine at a specific moment in time in accordance with the parameters configured in the driver.

Knowing the engine holding moment at any given moment will allow you to adjust the engine depending on the load it supports loads. When describing the registers of our driver, three parameters are important, including SG_RESULT. To adjust this 10-bit value in real time, we will conduct several tests to demonstrate the characteristics engine by changing the following parameters:

- Step switching frequency
- Current in both excitation windings of the motor
- SGT

a) Evaluation by changing the step switching frequency:

The frequency affects the speed of switching between two steps. Our research shows that the higher the frequency, the more the engine produces sufficiently audible noise. The following graphs show the minimum the value of the torque represented by SGT_RESULT when the engine blocked at full step, SGT = 16 and phase current is 600 A

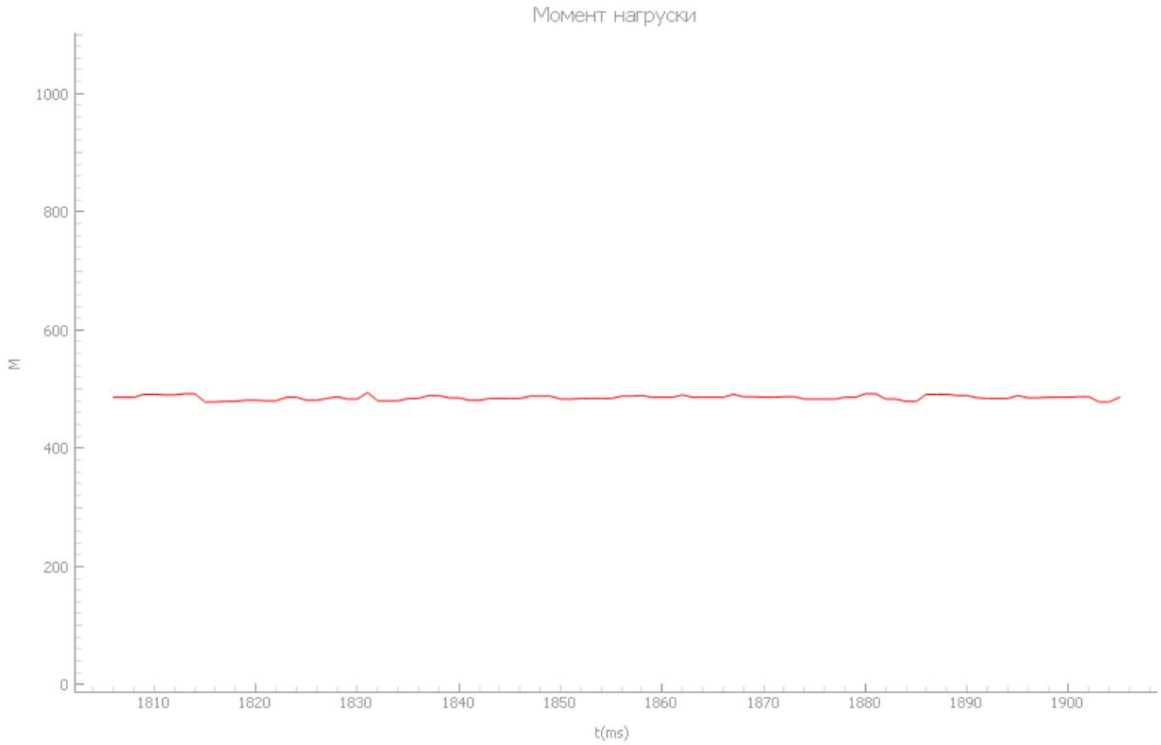


Figure 27 - Load torque at 5000 Hz

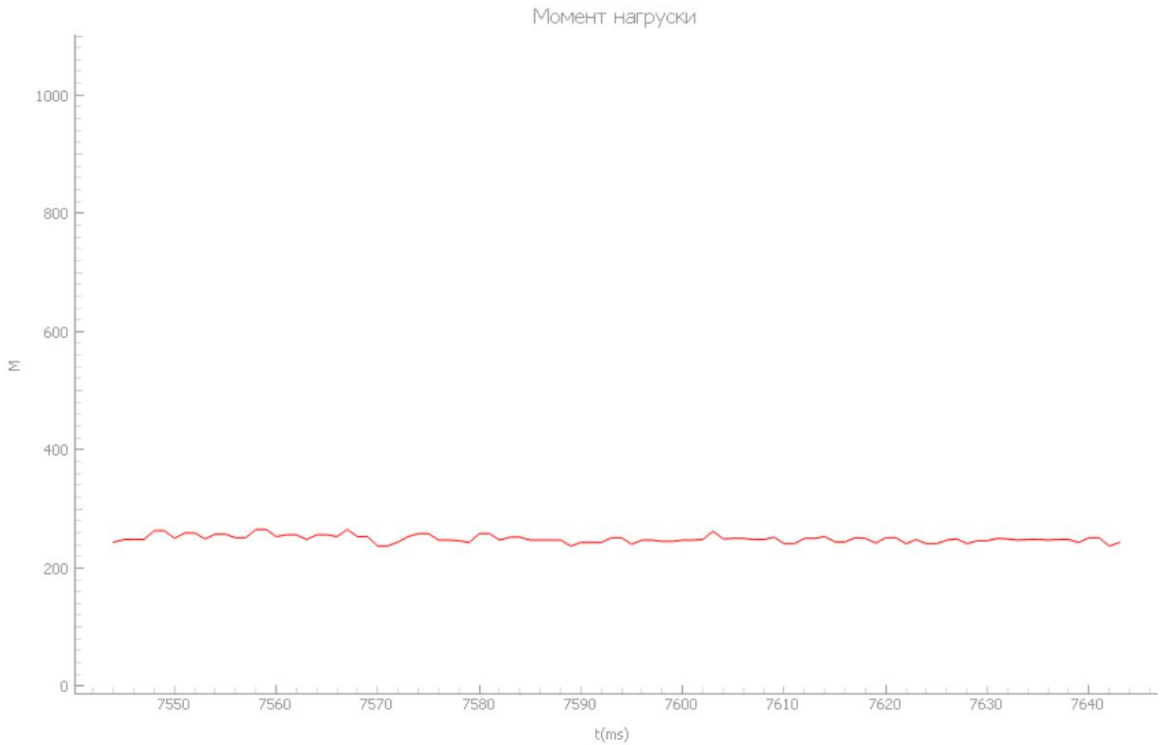


Figure 28 - Load torque at 2000 Hz

According to the two graphs, the torque will increase as the frequency decreases. switching.

c) Evaluation by changing the current of both phases:

The current generated by the driver at the output plays an important role in determining the value of the support moment created by the stepper engine. The driver produces an average current I_RMS, proportional current values in each phase. The following graphs show the minimum value of the moment represented by SGT_RESULT when motor locked at full step, SGT=16 and switching frequency steps 10000 Hz

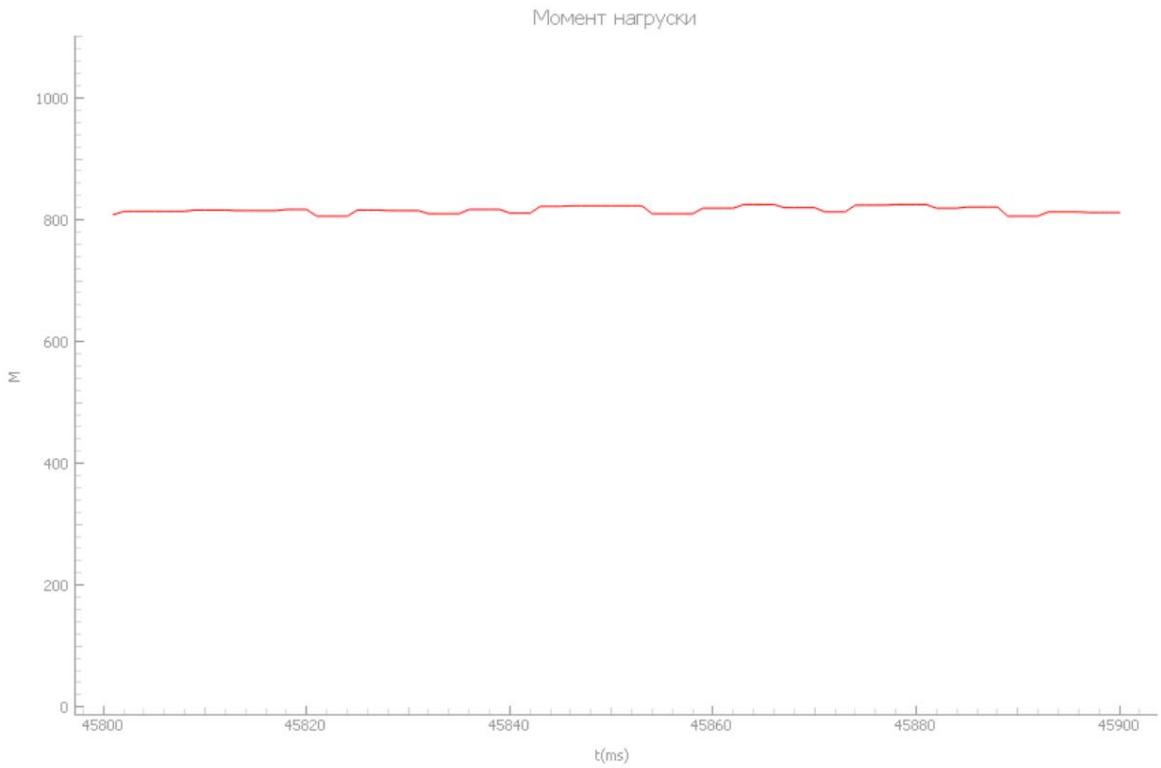


Figure 29 - Load torque at 200 mA

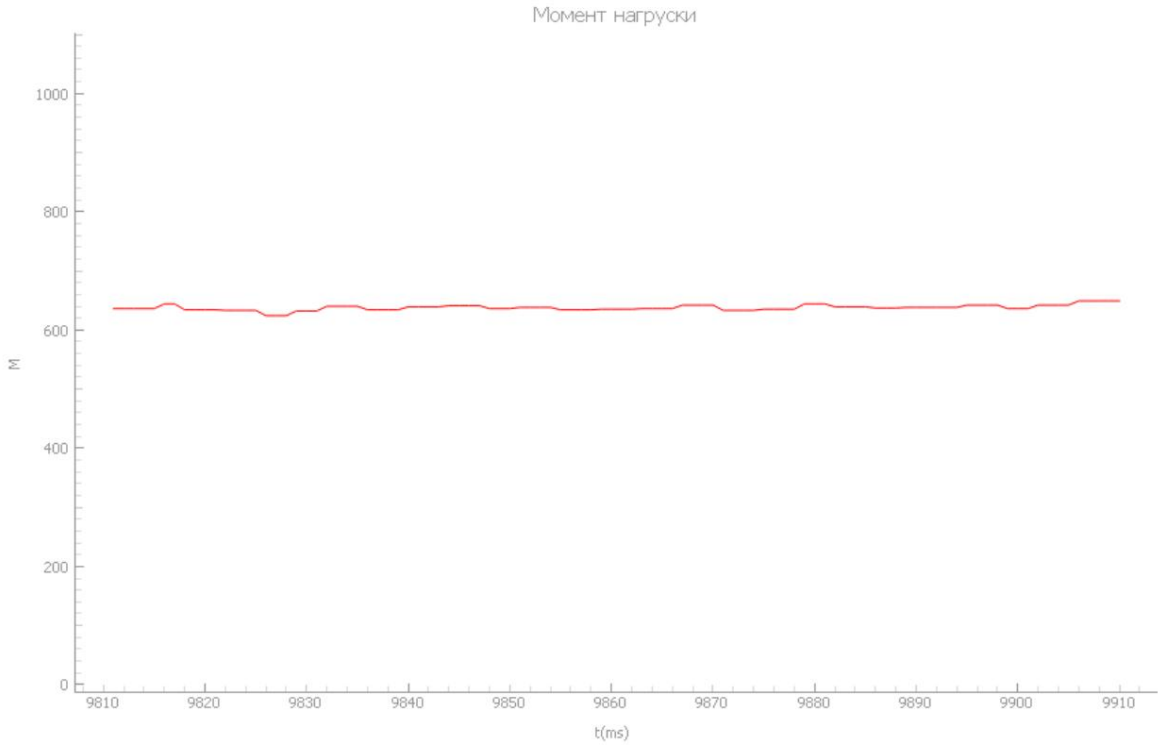


Figure 30 - Load torque at 600 mA

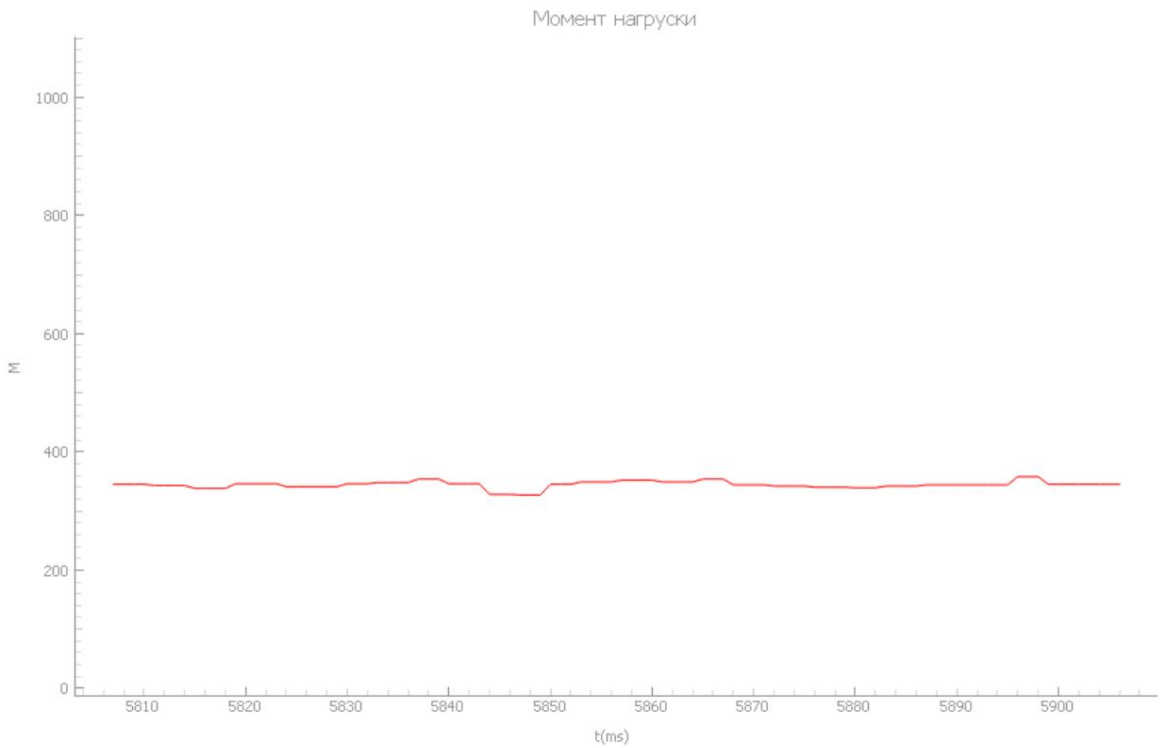


Figure 31 - Load torque at 900 mA

The graph below shows that the more the current increases, the more the moment increases.

b) Evaluation by changing the SGT parameter

SGT plays an important role in ensuring stability and accuracy SG_RESULT values. The SGT value must change each time changing the above parameters. The following graphs show the minimum value of the moment represented by SGT_RESULT when the engine is blocked at full step, the current in the excitation windings is equal to 600 mA and step switching frequency of 10000 Hz

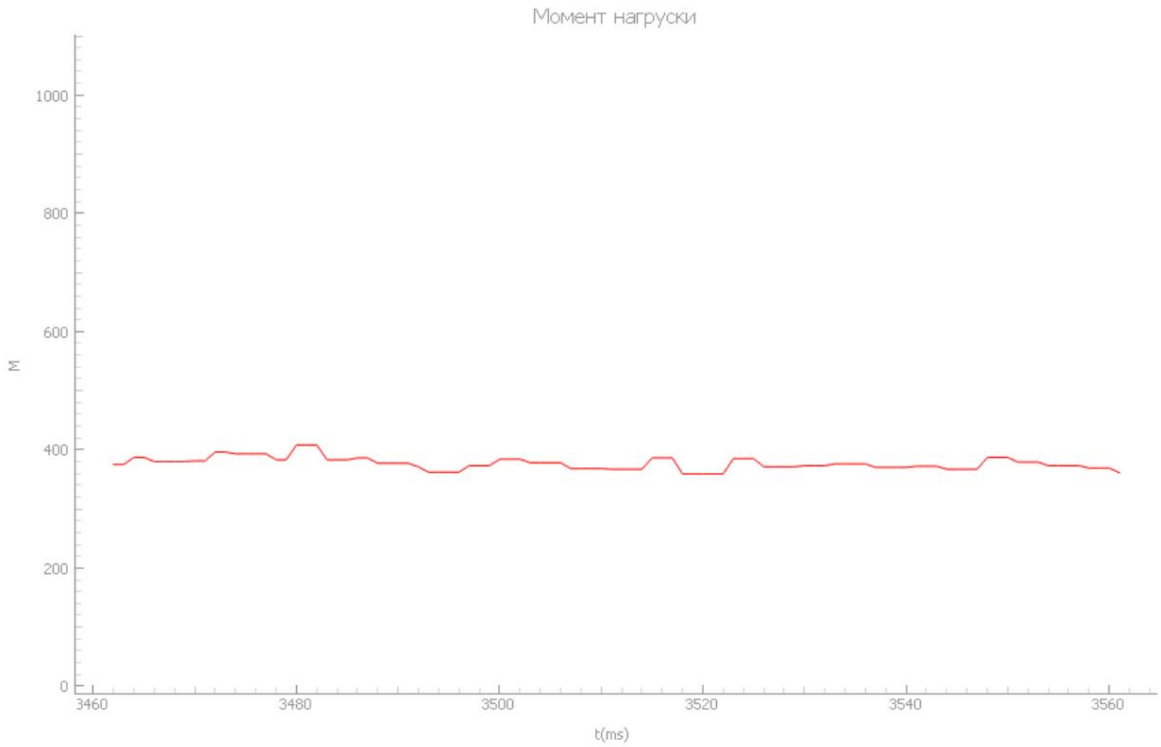


Figure 32 — Load moment at SGT=11

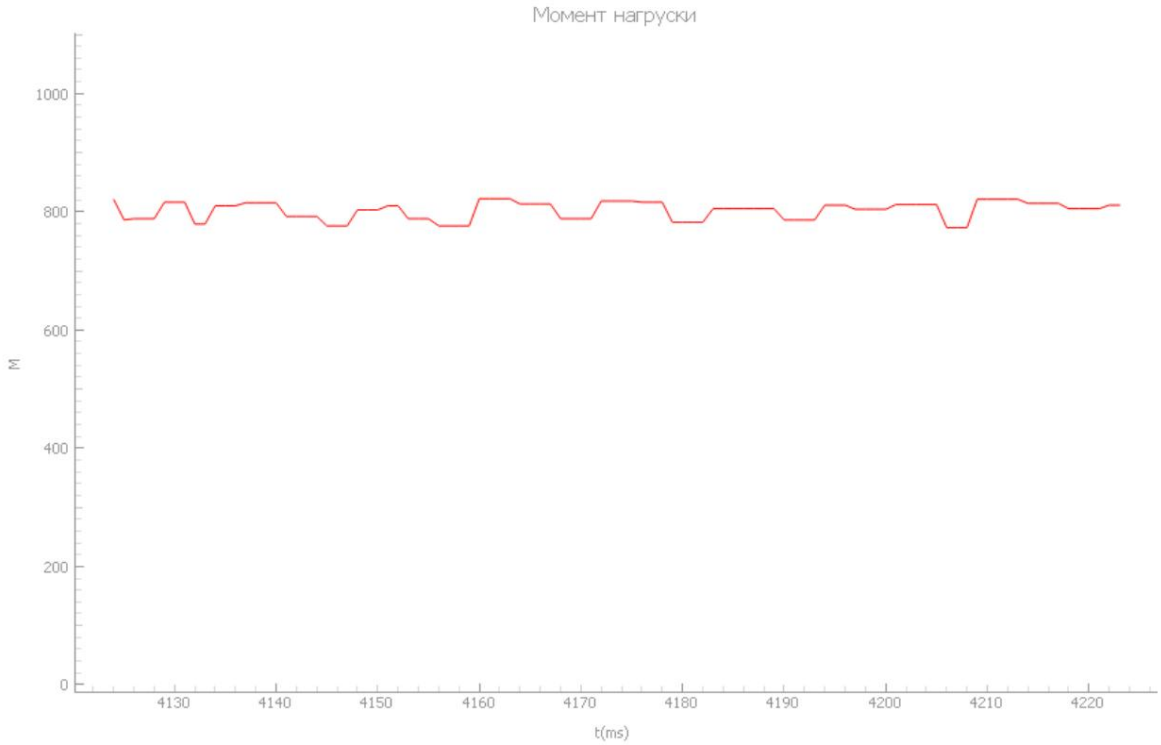


Figure 33 — Load moment at SGT=15

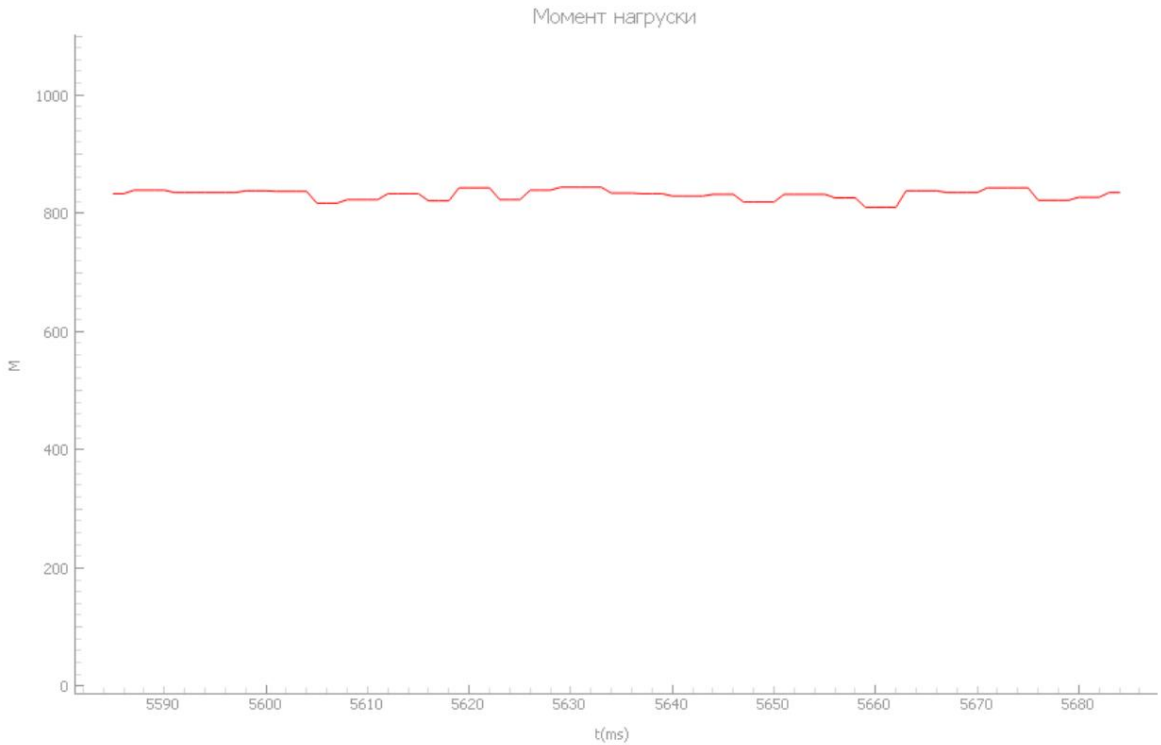


Figure 34 — Load moment at SGT=16

These 3 graphs show that the more SGT increases, the the more the moment increases.

CONCLUSION

LIST OF USED SOURCES

1. T. Kenyo, Stepper motors and their microprocessor systems management, Moscow: - Energoatomizdat, 1987 - 189 p.
2. Alain et Ursula Bouteveille, Moteurs Electriques pour la Robotique, Paris: — Dunod, 2016 — 300 p.
3. A.V. Emelyanov, A.N. Shilin, Stepper Motors, Volgograd: — Volga State Technical University, 2005 - 48 p.
4. J. Marcel, Electromécanique, Lausanne: - Revue et augmentée, 2004 - 391 pp.
5. M.A. Ankuda, S.E. Zharsky, N.M. Oliferovich, M.L. Heifetz, Mathematical model of stepper motor for control system 3D printer drive, Minsk: UDC 621.313
6. R. T. Emelyanov, A.S. Klimov, I.B. Olenev, E.S. Turysheva, A.I. Avlasevich, Study of the dynamics of the manipulator drive with a stepper drive, Krasnoyarsk: - Syrian Federal University
7. Stepper motor controller. [electronic resource] Tolerance mode: http://rfanat.ru/stanki_chpu/stat2.html
8. TMC2130 datasheet. [electronic resource] Tolerance mode: https://www.analog.com/media/en/technical-documentation/data-sheets/TMC2130_datasheet_rev1.15.pdf
9. D.O. Varlamov, S.M. Zuev, Yu.M. Shmatkov, A.A. Lavrikov, A.A. Timoshenko, Basics of Working with Sequential Peripherals interface (SPI), Moscow: - Moscow Polytechnic University, 2019. - 41 p.
10. ATmega16U4/ATmega32U4 datasheet. [electronic resource] Mode permission: https://ww1.microchip.com/downloads/en/devicedoc/atmel-7766-8-bit-avr-atmega16u4-32u4_datasheet.pdf

APPLICATIONS

APPENDIX A

SPI_SW.h

```
#include <Arduino.h>


class SW_SPIClass
    { public:
        void setPins(uint16_t sw_mosi_pin, uint16_t sw_miso_pin, uint16_t sw_sck_pin); void
init(); void
        begin() {}; byte
        transfer(uint8_t ulVal,
        uint8_t ulBitOrder=MSBFIRST); uint16_t transfer16(uint16_t data); void
        endTransaction() {}; private: uint16_t mosi_pin,
        miso_pin, sck_pin; uint8_t
    mosi_bm,
        miso_bm, sck_bm;
            volatile
            uint8_t
        *mosi_register,


        *miso_register, *sck_register;


};

extern SW_SPIClass TMC_SW_SPI;
```

APPENDIX B

(TMC2130Stepper_REGDEFS.h)

```
#ifndef TMC2130Stepper_REGDEFS_h
#define TMC2130Stepper_REGDEFS_h

constexpr uint8_t TMC2130_READ          = 0x00;
constexpr uint8_t TMC2130_WRITE         = 0x80;

// Register memory positions
constexpr uint8_t REG_GCONF             = 0x00;
constexpr uint8_t REG_GSTAT             = 0x01;
constexpr uint8_t REG_IOIN              = 0x04;
constexpr uint8_t REG_IHOLD_IRUN        = 0x10;
constexpr uint8_t REG_TPOWERDOWN        = 0x11;
constexpr uint8_t REG_TSTEP             = 0x12;
constexpr uint8_t REG_TPWMTHRS          = 0x13;
constexpr uint8_t REG_TCOOLTHRS         = 0x14;
constexpr uint8_t REG_THIGH             = 0x15;
constexpr uint8_t REG_XDIRECT           = 0x2D;
constexpr uint8_t REG_VDCMIN            = 0x33;
constexpr uint8_t REG_MSLUT0            = 0x60;
constexpr uint8_t REG_MSLUT1            = 0x61;
constexpr uint8_t REG_MSLUT2            = 0x62;
constexpr uint8_t REG_MSLUT3            = 0x63;
constexpr uint8_t REG_MSLUT4            = 0x64;
constexpr uint8_t REG_MSLUT5            = 0x65;
constexpr uint8_t REG_MSLUT6            = 0x66;
constexpr uint8_t REG_MSLUT7            = 0x67;
constexpr uint8_t REG_MSLUTSEL          = 0x68;
constexpr uint8_t REG_MSLUTSTART        = 0x69;
constexpr uint8_t REG_MSCNT             = 0x6A;
constexpr uint8_t REG_MSCURACT          = 0x6B;
constexpr uint8_t REG_CHOPCONF          = 0x6C;
constexpr uint8_t REG_COOLCONF          = 0x6D;
constexpr uint8_t REG_DCCTRL            = 0x6E;
constexpr uint8_t REG_DRV_STATUS        = 0x6F;
constexpr uint8_t REG_PWMCONF           = 0x70;
constexpr uint8_t REG_PWM_SCALE         = 0x71;
constexpr uint8_t REG_ENCM_CTRL         = 0x72;
constexpr uint8_t REG_LOST_STEPS        = 0x73;

// SPI_STATUS
constexpr uint8_t RESET_FLAG_bp = 0;
constexpr uint8_t DRIVER_ERROR_bp = 1;
constexpr uint8_t SG2_bp = 2;
constexpr uint8_t STANDSTILL_bp = 3;
constexpr uint32_t RESET_FLAG_bm = 0x1UL;
constexpr uint32_t DRIVER_ERROR_bm = 0x2UL;
constexpr uint32_t SG2_bm = 0x4UL;
```

```
constexpr uint32_t STANDSTILL_bm                = 0x8UL;

// GCONF
constexpr uint8_t I_SCALE_ANALOG_bp = 0;
constexpr uint8_t INTERNAL_RSENSE_bp = 1;
constexpr uint8_t EN_PWM_MODE_bp = 2;
constexpr uint8_t ENC_COMMUTATION_bp = 3;
constexpr uint8_t SHAFT_bp = 4;
constexpr uint8_t DIAG0_ERROR_bp = 5;
constexpr uint8_t DIAG0_OTPW_bp = 6;
constexpr uint8_t DIAG0_STALL_bp = 7;
constexpr uint8_t DIAG1_STALL_bp = 8;
constexpr uint8_t DIAG1_INDEX_bp = 9;
constexpr uint8_t DIAG1_ONSTATE_bp = 10;
constexpr uint8_t DIAG1_STEPS_SKIPPED_bp = 11;
constexpr uint8_t DIAG0_INT_PUSHPULL_bp = 12;
constexpr uint8_t DIAG1_PUSHPULL_bp = 13;
constexpr uint8_t SMALL_HYSTERESIS_bp = 14;
constexpr uint8_t STOP_ENABLE_bp = 15;
constexpr uint8_t DIRECT_MODE_bp = 16;
constexpr uint32_t GCONF_bm = 0x3FFFFUL;
constexpr uint32_t I_SCALE_ANALOG_bm = 0x1UL;
constexpr uint32_t INTERNAL_RSENSE_bm = 0x2UL;
constexpr uint32_t EN_PWM_MODE_bm = 0x4UL;
constexpr uint32_t ENC_COMMUTATION_bm = 0x8UL;
constexpr uint32_t SHAFT_bm = 0x10UL;
constexpr uint32_t DIAG0_ERROR_bm = 0x20UL;
constexpr uint32_t DIAG0_OTPW_bm = 0x40UL;
constexpr uint32_t DIAG0_STALL_bm = 0x80UL;
constexpr uint32_t DIAG1_STALL_bm = 0x100UL;
constexpr uint32_t DIAG1_INDEX_bm = 0x200UL;
constexpr uint32_t DIAG1_ONSTATE_bm = 0x400UL;
constexpr uint32_t DIAG1_STEPS_SKIPPED_bm = 0x800UL;
constexpr uint32_t DIAG0_INT_PUSHPULL_bm constexpr      = 0x1000UL;
uint32_t DIAG1_PUSHPULL_bm constexpr uint32_t           = 0x2000UL;
SMALL_HYSTERESIS_bm constexpr uint32_t                  = 0x4000UL;
STOP_ENABLE_bm constexpr uint32_t                       = 0x8000UL;
DIRECT_MODE_bm                                          = 0x10000UL;
// GSTAT
constexpr uint8_t RESET_bp                      = 0;
constexpr uint8_t DRV_ERR_bp                    = 1;
constexpr uint8_t UV_CP_bp                      = 2;
constexpr uint32_t GSTAT_bm                     = 0x7UL;
constexpr uint32_t RESET_bm                     = 0b1UL;
constexpr uint32_t DRV_ERR_bm                   = 0b10UL;
constexpr uint32_t UV_CP_bm                     = 0b100UL;
// IOIN
constexpr uint8_t STEP_bp                       = 0;
constexpr uint8_t DIR_bp constexpr              = 1;
uint8_t DCEN_CFG4_bp constexpr uint8_t          = 2;
DCIN_CFG5_bp                                    = 3;
```

```
constexpr uint8_t DRV_ENN_CFG6_bp = 4;
constexpr uint8_t DCO_bp = 5;
constexpr uint8_t VERSION_bp = 24;
constexpr uint32_t IOIN_bm = 0xFF00003FUL;
constexpr uint32_t STEP_bm = 0x1UL;
constexpr uint32_t DIR_bm = 0x2UL;
constexpr uint32_t DCEN_CFG4_bm = 0x4UL;
constexpr uint32_t DCIN_CFG5_bm = 0x8UL;
constexpr uint32_t DRV_ENN_CFG6_bm = 0x10UL;
constexpr uint32_t DCO_bm = 0x20UL;
constexpr uint32_t VERSION_bm = 0xFF000000UL;
// IHOLD_IRUN
constexpr uint8_t IHOLD_bp                  = 0;
constexpr uint8_t IRUN_bp                   = 8;
constexpr uint8_t IHOLDDELAY_bp             = 16;
constexpr uint32_t IHOLD_IRUN_bm            = 0xF1F1FUL;
constexpr uint32_t IHOLD_bm                 = 0x1FUL;
constexpr uint32_t IRUN_bm                  = 0x1F00UL;
constexpr uint32_t IHOLDDELAY_bm            = 0xF0000UL;
//TPOWERDOWN
constexpr uint8_t TPOWERDOWN_bp             = 0;
constexpr uint32_t TPOWERDOWN_bm            = 0xFFUL;
//TSTEPS
constexpr uint8_t TSTEP_bp                  = 0;
constexpr uint32_t TSTEP_bm                 = 0xFFFFFUL;
//TPWMTHRS
constexpr uint8_t TPWMTHRS_bp               = 0;
constexpr uint32_t TPWMTHRS_bm              = 0xFFFFFUL;
// TCOOLTHRS
constexpr uint8_t TCOOLTHRS_bp              = 0;
constexpr uint32_t TCOOLTHRS_bm             = 0xFFFFFUL;
// THIGH
constexpr uint8_t THIGH_bp                  = 0;
constexpr uint32_t THIGH_bm                 = 0xFFFFFUL;
// XDIRECT
constexpr uint8_t XDIRECT_bp                = 0;
constexpr uint32_t XDIRECT_bm               = 0xFFFFFFFFUL;
constexpr uint8_t COIL_A_bp constexpr       = 0;
uint8_t COIL_B_bp constexpr uint32_t        = 16;
COIL_A_bm constexpr uint32_t                = 0x1FFUL;
COIL_B_bm                                   = 0x1FF0000UL;
// VDCMIN
constexpr uint8_t VDCMIN_bp                 = 0;
constexpr uint32_t VDCMIN_bm                = 0x7FFFFFUL;
//MSLUT0
constexpr uint8_t MSLUT0_bp                 = 0;
constexpr uint32_t MSLUT0_bm                = 0xFFFFFFFFUL;
//MSLUT1
constexpr uint8_t MSLUT1_bp                 = 0;
constexpr uint32_t MSLUT1_bm                = 0xFFFFFFFFUL;
//MSLUT2
```

```
constexpr uint8_t MSLUT2_bp = 0;
constexpr uint32_t MSLUT2_bm = 0xFFFFFFFFUL;
//MSLUT3
constexpr uint8_t MSLUT3_bp = 0;
constexpr uint32_t MSLUT3_bm = 0xFFFFFFFFUL;
//MSLUT4
constexpr uint8_t MSLUT4_bp = 0;
constexpr uint32_t MSLUT4_bm = 0xFFFFFFFFUL;
//MSLUT5
constexpr uint8_t MSLUT5_bp = 0;
constexpr uint32_t MSLUT5_bm = 0xFFFFFFFFUL;
//MSLUT6
constexpr uint8_t MSLUT6_bp = 0;
constexpr uint32_t MSLUT6_bm = 0xFFFFFFFFUL;
//MSLUT7
constexpr uint8_t MSLUT7_bp = 0;
constexpr uint32_t MSLUT7_bm = 0xFFFFFFFFUL;
//MSLUTSEL
constexpr uint8_t MSLUTSEL_bp = 0;
constexpr uint32_t MSLUTSEL_bm = 0xFFFFFFFFUL;
//MSLUTSTART
constexpr uint8_t START_SIN_bp = 0;
constexpr uint8_t START_SIN90_bp = 16;
constexpr uint32_t START_SIN_bm = 0xFFUL;
constexpr uint32_t START_SIN90_bm = 0xFF0000UL;
//MSCNT
constexpr uint8_t MSCNT_bp = 0;
constexpr uint32_t MSCNT_bm = 0x3FFUL;
//MSCURACT
constexpr uint8_t CUR_A_bp = 0;
constexpr uint8_t CUR_B_bp = 16;
constexpr uint32_t CUR_A_bm = 0x1FFUL;
constexpr uint32_t CUR_B_bm = 0x1FF0000UL;
// CHOPCONF
constexpr uint8_t TOFF_bp = 0;
constexpr uint8_t HSTRT_bp = 4;
constexpr uint8_t FD_bp = 4;
constexpr uint8_t HEND_bp = 7;
constexpr uint8_t DISFDCC_bp = 12;
constexpr uint8_t RNDTF_bp = 13;
constexpr uint8_t CHM_bp = 14;
constexpr uint8_t TBL_bp = 15;
constexpr uint8_t VSENSE_bp = 17;
constexpr uint8_t VHIGHFS_bp = 18;
constexpr uint8_t VHIGHCHM_bp = 19;
constexpr uint8_t SYNC_bp = 20;
constexpr uint8_t MRES_bp = 24;
constexpr uint8_t INTPOL_bp = 28;
constexpr uint8_t DEDGE_bp = 29;
constexpr uint8_t DISS2G_bp = 30;
constexpr uint32_t CHOPCONF_bm = 0xFFFFFFFFUL;
```

```
constexpr uint32_t TOFF_bm          = 0xFUL;
constexpr uint32_t HSTRT_bm         = 0x70UL;
constexpr uint32_t FD_bm            = 0x830UL;
constexpr uint32_t HEND_bm          = 0x780UL;
constexpr uint32_t DISFDCC_bm       = 0x1000UL;
constexpr uint32_t RNDTF_bm         = 0x2000UL;
constexpr uint32_t CHM_bm           = 0x4000UL;
constexpr uint32_t TBL_bm           = 0x18000UL;
constexpr uint32_t VSENSE_bm        = 0x20000UL;
constexpr uint32_t VHIGHFS_bm       = 0x40000UL;
constexpr uint32_t VHIGHCHM_bm      = 0x80000UL;
constexpr uint32_t SYNC_bm          = 0xF00000UL;
constexpr uint32_t MRES_bm          = 0xF000000UL;
constexpr uint32_t INTPOL_bm        = 0x10000000UL;
constexpr uint32_t DEDGE_bm         = 0x20000000UL;
constexpr uint32_t DISS2G_bm        = 0x40000000UL;
// COOLCONF
constexpr uint8_t SEMIN_bp          = 0;
constexpr uint8_t SEUP_bp           = 5;
constexpr uint8_t SEMAX_bp          = 8;
constexpr uint8_t SEDN_bp           = 13;
constexpr uint8_t SEIMIN_bp         = 15;
constexpr uint8_t SGT_bp            = 16;
constexpr uint8_t SFILT_bp          = 24;
constexpr uint32_t COOLCONF_bm      = 0x3FFFFFFUL;
constexpr uint32_t SEMIN_bm         = 0xFUL;
constexpr uint32_t SEUP_bm          = 0x60UL;
constexpr uint32_t SEMAX_bm         = 0xF00UL;
constexpr uint32_t SEDN_bm          = 0x6000UL;
constexpr uint32_t SEIMIN_bm        = 0x8000UL;
constexpr uint32_t SGT_bm           = 0x7F0000UL;
constexpr uint32_t SFILT_bm         = 0x1000000UL;
// DCCTRL
constexpr uint8_t DC_TIME_bp        = 0;
constexpr uint8_t DC_SG_bp          = 16;
constexpr uint32_t DC_TIME_bm       = 0x3FFUL;
constexpr uint32_t DC_SG_bm         = 0xFF0000UL;
// DRV_STATUS
constexpr uint8_t SG_RESULT_bp      = 0;
constexpr uint8_t FSACTIVE_bp       = 15;
constexpr uint8_t CS_ACTUAL_bp      = 16;
constexpr uint8_t STALLGUARD_bp     = 24;
constexpr uint8_t OT_bp             = 25;
constexpr uint8_t OTPW_bp           = 26;
constexpr uint8_t S2GA_bp           = 27;
constexpr uint8_t S2GB_bp           = 28;
constexpr uint8_t OLA_bp            = 29;
constexpr uint8_t OLB_bp            = 30;
constexpr uint8_t STST_bp           = 31;
constexpr uint32_t DRV_STATUS_bm    = 0xFFFFFFFFUL;
constexpr uint32_t SG_RESULT_bm     = 0x3FFUL;
```

```
constexpr uint32_t FSACTIVE_bm                = 0x8000UL;
constexpr uint32_t CS_ACTUAL_bm               = 0x1F0000UL;
constexpr uint32_t STALLGUARD_bm              = 0x1000000UL;
constexpr uint32_t OT_bm                      = 0x2000000UL;
constexpr uint32_t OTPW_bm                    = 0x4000000UL;
constexpr uint32_t S2GA_bm                    = 0x8000000UL;
constexpr uint32_t S2GB_bm                    = 0x10000000UL;
constexpr uint32_t OLA_bm                     = 0x20000000UL;
constexpr uint32_t OLB_bm                     = 0x40000000UL;
constexpr uint32_t STST_bm                    = 0x80000000UL;
// PWMCONF
constexpr uint8_t PWM_AMPL_bp = 0;
constexpr uint8_t PWM_GRAD_bp = 8;
constexpr uint8_t PWM_FREQ_bp = 16;
constexpr uint8_t PWM_AUTOSCALE_bp = 18;
constexpr uint8_t PWM_SYMMETRIC_bp = 19;
constexpr uint8_t FREEWHEEL_bp = 20;
constexpr uint32_t PWMCONF_bm = 0x7FFFFFUL;
constexpr uint32_t PWM_AMPL_bm = 0xFFUL;
constexpr uint32_t PWM_GRAD_bm = 0xFF00UL;
constexpr uint32_t PWM_FREQ_bm = 0x30000UL;
constexpr uint32_t PWM_AUTOSCALE_bm = 0x40000UL;
constexpr uint32_t PWM_SYMMETRIC_bm = 0x80000UL;
constexpr uint32_t FREEWHEEL_bm = 0x300000UL;
// PWM_SCALE
constexpr uint8_t PWM_SCALE_bp                = 0;
constexpr uint32_t PWM_SCALE_bm               = 0xFFUL;
// ENCM_CTRL
constexpr uint8_t INV_bp constexpr            = 0;
uint8_t MAXSPEED_bp constexpr                 = 1;
uint32_t INV_bm constexpr uint32_t            = 0x1UL;
MAXSPEED_bm                                   = 0x2UL;
// LOST_STEPS
constexpr uint8_t LOST_STEPS_bp               = 0;
constexpr uint32_t LOST_STEPS_bm              = 0xFFFFFUL;

#endif
```

APPENDIX B

simple_control.cpp

```
#define EN_PIN 6
#define DIR_PIN 8
#define STEP_PIN 9
#define CS_PIN 10

#define STEP_PORT PORTF
#define STEP_BIT 0

const long BAUD = 115200;

int speed = 10; bool
running = false; float Rsense
= 0.11; float hold_x = 0.5;
boolean toggle1 = 0;
uint8_t stall_value = 9;
volatile uint32_t step_counter =
0; const uint32_t steps_per_mm = 80 * 16; // @
256 microsteps const uint16_t fsteps_per_rotation= 255; const uint32_t MHz =
16000000>>8; const uint16_t acceleration = 2000;
uint16_t intevale_frequence = 5000;



#include <TMC2130Stepper.h>
TMC2130Stepper myStepper = TMC2130Stepper(CS_PIN);

ISR(TIMER1_COMPA_vect){
    STEP_PORT |= 1 << STEP_BIT;
    STEP_PORT &= ~(1 << STEP_BIT);
    step_counter++;
}

void initTimer() { cli();


    TCCR1A = 0; // Sets the TCCR1A register to 0
    TCCR1B = 0; // Also for TCCR1B
    TCNT1 = 0; // Initialize counter value to 0 // set compare match register
    for 2kHz increments
    OCR1A = 61;// = (16*10^6) / (8*2000) - 1 // Set CTC
    mode
    TCCR1B |= (1 << WGM12);
    TCCR1B |= (1 << CS10); //
    Timer1 interrupt enable flag for events
    TIMSK1 &= ~(1 << OCIE1A); sei();

}

unsigned long ms;
```

```
unsigned long millis_(){ cli(); ms =

    step_counter; sei(); return
    ms;

}

void delay_(unsigned long ms){
    unsigned long start = millis_(); while(millis_() -
    start < ms){

    }
}

void serialTuple(String cmd, int arg) { Serial.print("Received
    command: "); Serial.print(cmd); Serial.print("(");
    Serial.print(arg);
    Serial.println()");


}

void setup()
    { initTimer();
    Serial.begin(BAUD);

    pinMode(EN_PIN, OUTPUT);
    pinMode(DIR_PIN, OUTPUT);
    pinMode(STEP_PIN, OUTPUT);
    pinMode(CS_PIN, OUTPUT);
    digitalWrite(EN_PIN, HIGH);
    digitalWrite(DIR_PIN, LOW);
    digitalWrite(STEP_PIN, LOW);
    digitalWrite(CS_PIN, HIGH);
    SPI.begin();
    pinMode(MISO, INPUT_PULLUP);

    myStepper.push();
    myStepper.tbl(1); //blank_time(24);
    myStepper.power_down_delay(255);
    myStepper.toff(4);

    // Effective hysteresis = 0
    myStepper.hstrt(0);
    myStepper.hend(2);

    myStepper.en_pwm_mode(true);
    myStepper.pwm_freq(1);
    myStepper.pwm_autoscale(true);
    myStepper.pwm_ampl(255);
    myStepper.pwm_grad(4);
    myStepper.rms_current(800); // 800 mA
    myStepper.microsteps(0);
```

```
    myStepper.diag1_stall(1);
    myStepper.diag1_active_high(1);
    digitalWrite(EN_PIN, LOW);

} bool run_ = false; bool
only_run = false; unsigned
long times_ = 0; float
angular_speed = 0;

void loop()
    { if(run_ && only_run)
        { digitalWrite(STEP_PIN, HIGH);
        digitalWrite(STEP_PIN, LOW);
        delay_(intevale_frequence-times_) ;

    } if (Serial.available() > 0) {

        String cmd = Serial.readStringUntil(' '); String strArg =
        Serial.readStringUntil('\n');


        int arg = strArg.toInt();

        if (cmd == "run")
            { serialTuple("run", arg); running
            = arg; arg?
            TIMSK1 |= 1 << OCIE1A : TIMSK1 &= ~(1 << OCIE1A); run_ = arg?
            true: false; angular_speed =
            angluar_speed();



        } else if (cmd == "fsps") {
            serialTuple("fsps", arg); cli();
            OCR1A
            = calculateFSPSTimer(arg); Serial.print("Set
            OCR1A to "); Serial.println(OCR1A);
            sei();


        } else if (cmd == "rps") {
            serialTuple("rps", arg); cli();
            OCR1A
            = calculateRPSTimer(arg); Serial.print("Set
            OCR1A to "); Serial.println(OCR1A);
            sei();


        } else if (cmd == "setCurrent") {
            serialTuple("setCurrent", arg);
            myStepper.setCurrent(arg, Rsense, hold_x);

        } else if (cmd == "torque") {
```

```
    torque_measurement();

} else if (cmd == "microstep") {
    serialTuple("microstep", arg);
    myStepper.microsteps(arg);
    angular_speed = angluar_speed();

} else if (cmd == "only_run") {
    serialTuple("only run", arg); only_run =
    arg;

} else if (cmd == "goto") {
    serialTuple("goto", arg);
    moveAngle(arg);

} else if (cmd == "frequency") {
    serialTuple("frequency", arg);
    intevale_frequence = arg;
    angular_speed = angluar_speed();

} else if (cmd == "Rsense")
    { Serial.print("Setting R sense value to: "); Serial.println(arg);
    Rsense = arg;


} else if (cmd == "GCONF") {
    Serial.print("GCONF: 0b");
    Serial.println(myStepper.GCONF(), BIN);

} else if (cmd == "I_scale_analog") {
    serialTuple("I_scale_analog", arg);
    myStepper.I_scale_analog(arg);

} else if (cmd == "internal_Rsense") {
    serialTuple("internal_Rsense", arg);
    myStepper.internal_Rsense(arg);

} else if (cmd == "en_pwm_mode") {
    serialTuple("en_pwm_mode", arg);
    myStepper.en_pwm_mode(arg);

} else if (cmd == "enc_commutation") {
    serialTuple("enc_commutation", arg);
    myStepper.enc_commutation(arg);

} else if (cmd == "shaft") {
    serialTuple("shaft", arg);
    myStepper.shaft(arg);

} else if (cmd == "diag0_errors") {
    serialTuple("diag0_errors", arg);
    myStepper.diag0_errors(arg);
}
```

```
else if (cmd == "small_hysteresis") {
    serialTuple("small_hysteresis", arg);
    myStepper.small_hysteresis(arg);

} else if (cmd == "stop_enable") {
    serialTuple("stop_enable", arg);
    myStepper.stop_enable(arg);

} else if (cmd == "direct_mode") {
    serialTuple("direct_mode", arg);
    myStepper.direct_mode(arg);
}
// IHOLD_IRUN
else if (cmd == "ihold") {
    serialTuple("ihold", arg);
    myStepper.ihold(arg);

} else if (cmd == "irun") {
    serialTuple("irun", arg);
    myStepper.irun(arg);

} else if (cmd == "iholddelay") {
    serialTuple("iholddelay", arg);
    myStepper.iholddelay(arg);
}

else if (cmd == "microstep_time")
    { Serial.print("microstep_time: ");
    Serial.println(myStepper.microstep_time());
}

else if (cmd == "TPWMTHRS") {
    serialTuple("TPWMTHRS", arg);
    myStepper.TPWMTHRS(arg);

} else if (cmd == "TCOOLTHRS") {
    serialTuple("TCOOLTHRS", arg);
    myStepper.TCOOLTHRS(arg);

} else if (cmd == "THIGH") {
    serialTuple("THIGH", arg);
    myStepper.THIGH(arg);
}
// XDIRECT
else if (cmd == "coil_A") {
    serialTuple("coil_A", arg);
    myStepper.coil_A(arg);

} else if (cmd == "coil_B") {
    serialTuple("coil_B", arg);
    myStepper.coil_B(arg);
}
```

```
else if (cmd == "VDCMIN") {
    serialTuple("VDCMIN", arg);
    myStepper.VDCMIN(arg);
}

else if (cmd == "CHOPCONF") {
    Serial.print("CHOPCONF: 0b");
    Serial.println(myStepper.CHOPCONF(), BIN);

} else if (cmd == "toff") {
    serialTuple("toff", arg);
    myStepper.toff(arg);

} else if (cmd == "hstrt") {
    serialTuple("hstrt", arg);
    myStepper.hstrt(arg);

} else if (cmd == "end") {
    serialTuple("hand", arg);
    myStepper.hend(arg);

} else if (cmd == "vsense") {
    serialTuple("vsense", arg);
    myStepper.vsense(arg);

} else if (cmd == "sync") {
    serialTuple("sync", arg);
    myStepper.sync(arg);

} else if (cmd == "mres") {
    serialTuple("mres", arg);
    myStepper.mres(arg);

} else if (cmd == "intpol") {
    serialTuple("intpol", arg);
    myStepper.intpol(arg);

} else if (cmd == "dedge") {
    serialTuple("dedge", arg);
    myStepper.dedge(arg);
}
// COOLCONF
else if (cmd == "semin") {
    serialTuple("semin", arg);
    myStepper.semin(arg);

} else if (cmd == "seup") {
    serialTuple("seup", arg);
    myStepper.seup(arg);

} else if (cmd == "semax") {
    serialTuple("semax", arg);
    myStepper.semax(arg);
```

```
} else if (cmd == "sedn") {
    serialTuple("sedn", arg);
    myStepper.sedn(arg);

} else if (cmd == "seimin") {
    serialTuple("seimin", arg);
    myStepper.seimin(arg);

} else if (cmd == "sgt") {
    serialTuple("sgt", arg);
    myStepper.sgt(arg);

} else if (cmd == "sfilt") {
    serialTuple("sfilt", arg);
    myStepper.sfilt(arg);
}
// PWMCONF
else if (cmd == "pwm_ampl") {
    serialTuple("pwm_ampl", arg);
    myStepper.pwm_ampl(arg);

} else if (cmd == "pwm_grad") {
    serialTuple("pwm_grad", arg);
    myStepper.pwm_grad(arg);

} else if (cmd == "pwm_freq") {
    serialTuple("pwm_freq", arg);
    myStepper.pwm_freq(arg);

} else if (cmd == "pwm_autoscale") {
    serialTuple("pwm_autoscale", arg);
    myStepper.pwm_autoscale(arg);

} else if (cmd == "pwm_symmetric") {
    serialTuple("pwm_symmetric", arg);
    myStepper.pwm_symmetric(arg);

} else if (cmd == "freewheel") {
    serialTuple("freewheel", arg);
    myStepper.freewheel(arg);
}
// ENCM_CTRL
else if (cmd == "inv") {
    serialTuple("inv", arg);
    myStepper.inv(arg);

} else if (cmd == "maxspeed") {
    serialTuple("maxspeed", arg);
    myStepper.maxspeed(arg);
}

else if (cmd == "DRVSTATUS") {
```

```
            Serial.print("DRVSTATUS: 0b");
            Serial.println(myStepper.DRV_STATUS(), BIN);

        } else if (cmd == "PWM_SCALE") {
            Serial.print("PWM_SCALE: 0b");
            Serial.println(myStepper.PWM_SCALE(), DEC);

        } else if (cmd == "LOST_STEPS") {
            Serial.print("LOST_STEPS: 0b");
            Serial.println(myStepper.LOST_STEPS(), DEC);

        } else
            { Serial.println("Invalid command!");
        }
    }
}


unsigned long time = 0; unsigned
long time_total = 0; unsigned long
current_time_ms = 0; unsigned long current_time_
= 0; unsigned long microstep_ = 0; double
time_move = 0; /*


*
    coal */

void moveAngle(float angle){ if(run_ && !
    only_run){ current_time_ms = 0;
        time_total = 0; time =
        intevale_frequence+1;
        if(angle<0){ myStepper.shaft(1); angle =
        angle*(-1); }
            else{ myStepper.shaft(0);}
            microstep_ =
        myStepper.microsteps(); time_move =
        (microstep_==0?(2^8)-1:((2^8)-


1)*microstep_)*(21.500*200/255)*(angle*intevale_frequence/5000)*18.446353516 061; Serial.print("estimated
time
      (s): "); Serial.println(time_move); Serial.println(time_move/
        (255000)); while(time_total<time_move)
        {

            current_time_ms = millis_();

            digitalWrite(STEP_PIN, HIGH);
            digitalWrite(STEP_PIN, LOW);

            current_time_ = millis_() - current_time_ms;
            delay_(intevale_frequence);

            time = millis_() - current_time_ms - current_time_;
```

```
            time_total += time;
        }
    }
}

float torque = 0; float
Ra = 1.04 ;// Ohm float Rb =
1.04 ; // Ohm float U = 13.2 ;// V
float I_a, I_b; float
torque_measurement()
{ I_a = (myStepper.cur_a()/1000.00);
    I_b = (myStepper.cur_b()/1000.00); torque =
    (U*I_a + U*I_b - (Ra*(I_a*I_a) + Rb*(I_b*I_b)))/
    angular_speed; Serial.print("torque : "); Serial.println(torque); return torque;



} float curren_t1, angle1, curren_t2, angle2; float
angluar_speed(){
    curren_t1 = millis_(); angle1 =
    angle_at(curren_t1/1000.00); curren_t2 = millis_();
    angle2 =
    angle_at(curren_t2/1000.00); return (angle2-angle1)/
    (curren_t2-curren_t1);

} float angle;
float angle_at(float time_move){
    microstep_ = myStepper.microsteps(); angle =
    (time_move*(255000))/((microstep_==0?(2^8)-1:((2^8)-
1)*microstep_)*(21.500*200/255)*(intevale_frequence/5000)*18.446353516061);
    return angle;
}
```