Project Documentation

Stepper Motor Controller

Project number: 174

Revision: 0

Date: 15.05.2021

Stepper Motor Controller Rev. 0

Module Description

Table of Contents

Introduction 2
Hardware 2
The Pro Micro 2
The A4988 Stepper Driver 2
The Power Supply 3
The Display 3
The Rotary Encoder 3
The Piezo Buzzer 3
Serial RS-232-Interface 4
Programming 4
Pin Configuration 4
Stepper Driver 4
The Piezo Buzzer 5
The LCD-Display 5
The Timer 1 Interrupt 6
The Interrupt Service Routine 7
The Rotary Encoder 8
RS-232 10
State Machine 13
Connector Pinouts 14
J1 - Power Connector 14
J2 – RS-232 Serial Interface 14
J3 – Rotary Encoder 15
J4, J5 – Stepper Motor Outputs 15
J6 – Reserved I/O-Pins 15
J7 – I²C/Display Connector 15
J8 – TX/RX Reserve Header 16
Jumpers 16
Assembly 17
The PCB 17
Cables 17
3D Printed Case 19

Revision History.. 22
Rev. 0 22

Introduction

The Board is a multi-purpose stepper motor controller based on a Pro Micro (Arduino) and up to two (Allegro) A4988 stepper motor driver. The power supply of +12V (basic configuration) is provided via a barrel connector.

The user interface consists of an I²C display, a rotary encoded with integrated button switch and a piezo buzzer. The connectivity is either an RS232 serial interface or the USB serial interface of the Pro Micro.

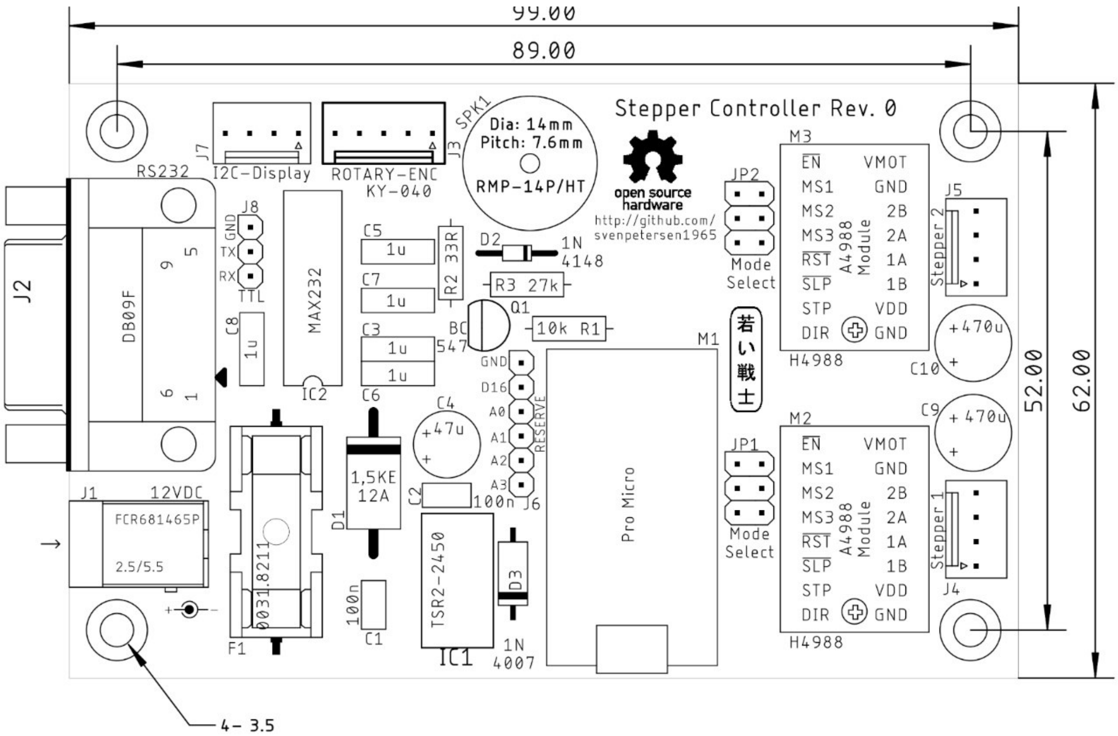


Figure 1: Dimensions of the Stepper Motor Controller

Hardware

The Pro Micro

The Pro Micro is an Arduino Leonardo (Software) compatible, Atmel ATmega32u4-based microprocessor module with 12 digital GPIO pins, 4 analog inputs (which can be used as digital I/Os as well), 32kB Flash, 2.5 kB RAM, 1kB EEPROM and a clock rate of 16MHz and a supply voltage of 5V (**3.3V** variants with 8MHz are available, but **not suitable** for this project).

The A4988 Stepper Driver

The stepper drivers are widely available as modules from Ebay, AliExpress and other online shopping platforms (search term “A4988”). They are based on the Allegro A4988 chip. This chip has five (hardware) selectable step modes: full steps, 1/2, 1/4, 1/8 and 1/16 micro steps. The digital supply voltage is either 5V (used) or 3.3V, the stepper motor supply voltage is up to 35V (12V are used in this projects) and is capable to drive up to 2A.

The micro step mode is selected via jumpers (JP1 for Stepper 1/Channel A and JP2 for Stepper 2/Channel B).

The stepper drivers are controlled by three digital signals:

1. /EN: Enable, active low
2. STEP pulse input
3. DIR direction

The Enable signal /EN needs to be low to activate the driver. A HIGH level will deactivate the drivers. No current flows through the step motor windings, they can be moved freely, stepping will not be executed. The /EN signal of both stepper controllers are interconnected.

A HIGH pulse on the STEP input will result in one (micro) step of the motor in the direction determined by the DIR pin (HIGH = clockwise, LOW = counterclockwise).

Both, the STEP and the DIR signals are separate for each of the two stepper drivers.

There is a small potentiometer on each stepper driver module, which is for adjusting the motor current. To provide approximately equal micro steps, this potentiometer has to be adjusted properly. An adjustment based on the acoustic impression while the stepper motor is running is sufficient for most purposes. It also determines the "hold current", which influences the supply current and the momentum, the stepper motor is capable to hold.

The Power Supply

The power supply is connected to a 5mm/2.1mm barrel connector (+12V at the inner contact). The supply voltage is fused and over voltage protected, it is connected to the stepper motor supply pin (VMOT) of the stepper driver. Hence, it is possible to modify the PCB in a way, that 24V stepper motors can be driven. For this purpose, the fuse and the TVS diode D1 need to be modified.

The 5V for supplying the Pro Micro, the digital side of the stepper driver, the display, the buzzer and the rotary encoder, is generated from the input supply voltage. For a +12V supply, a simple linear 7805 voltage regulator is sufficient. The PCB provides the space to install a DC/DC converter (Traco TSR2-2450) instead, that might be required for powering the PCB with a higher supply voltage.

The Display

The display is connected to the I²C bus via a four-pin connector, which also provides a +5V pin for power supply. This will operate with the wide spread 16 columns/two lines or 20 columns/four lines LCD displays (I²C).

It is of course possible to connect an OLED-display with 5V supply voltage with a suitable software running on the Pro Micro.

The Rotary Encoder

Rotary encoders with an integrated push button are usually pretty common and suitable for a menu-based mode selection and parameter entry. A widely available rotary encoder module (with on-board pull-up resistors) is the KX-040. The connector for the rotary encoder is adapted to this module. In case, a rotary encoder is not required for the supplication, the pins can be repurposed.

The Piezo Buzzer

The purpose of the piezo buzzer is providing acoustic feedback or a warning signal for the user. Since piezo elements can produce some nasty voltage spikes when they get bent or beaten, the buzzer is not directly connected to the processor pin, but there is a protection circuit with a transistor and a diode.

Serial RS-232-Interface

The Pro Micro provides a virtual serial interface via USB for communication and also a real serial interface with RX/TX pins. An RS-232 level driver is integrated on this board. The connector is the D-Sub 9, female, so the stepper controller is a DCE (Data Communication Equipment) like a modem. It connects to a computer (or other DTE = Data Terminal Equipment) via a 1:1 serial cable. Some feedbacks are provided (RTS-DTS and DTR-DRS-DCD).

Programming

The Stepper Motor Controller (actually the Pro Micro) can be programmed with the Arduino IDE. The following chapters describe the source code of stepper_framework.ino, which does not do anything useful, except testing the hardware, nevertheless, this can be a framework to customized applications.

Pin Configuration

The pin configuration looks like this:

Pin	Direction	Signal	Purpose
D0	out	TX	RS-232 Interface serial output
D1	in	RX	RS-232 Interface serial input
D2	-	SDA	I²C SDA signal
D3	-	SCL	I²C SCL signal
D4	in	ROT_SW	Rotary Encoder Switch
D5	in	ROT_DATA	Rotary Encoder Data (actually output A)
D6	out	STP_IN	Stepper driver 1 (Channel A) pulse input
D7	out	STP_DIR	Stepper driver 1 (Channel A) direction input
D8	input	ROT_CLK	Rotary Encoder Clock (actually output B)
D9	out	SND_OUT	Output to the piezo buzzer
D10	out	$\overline{\text{STP\_EN}}$	Stepper Enable (active low) for both stepper drivers
D14	out	STP2_IN	Stepper driver 2 (Channel B) pulse input
D15	out	STP2_DIR	Stepper driver 2 (Channel B) direction input
D16	-	D16	Reserved GPIO
A0	-	A0	Reserved (analog) GPIO
A1	-	A1	Reserved (analog) GPIO
A2	-	A2	Reserved (analog) GPIO
A3	-	A3	Reserved (analog) GPIO

Stepper Driver

The pins are defined like this:

```
#define step1 6                    // stepper 1 step
#define dir1 7                     // stepper 1 direction
#define step2 14                   // stepper 2 step
#define dir2 15                    // stepper 2 direction
#define nstpena 10                 // enable both steppers (active low)
```

Pin initialization:

```
 // setup stepper controller
digitalWrite( step1, LOW );                   // LOW -> Step Signal
digitalWrite( dir1,  st_direction  );         // default direction
digitalWrite( step2, LOW );                   // LOW -> Step Signal
```

```
    digitalWrite( dir2, st_direction );        // default direction

    digitalWrite( nstpena, HIGH ); // disable stepper by setting the enable
      pin HIGH

    pinMode( step1, OUTPUT );      // step1 is an output
    pinMode( dir1, OUTPUT );       // dir1 is an output
    pinMode( step2, OUTPUT );      // step1 is an output
    pinMode( dir2, OUTPUT );       // dir1 is an output
    pinMode( nstpena, OUTPUT );    // nstpena is an output, that is the
      enable output for both stepper controllers
```

Previously, the direction was set to the desired value:

```
  #define step_cw HIGH   // for clockwise, the direction pin is HIGH
  #define step_ccw LOW   // for counterclockwise, the direction pin is LOW
  […]
  digitalWrite( dir1,  step_cw  );
```

To switch on the stepper driver output and perform a step, the stepper drivers need to be enabled by setting the enable input LOW:

```
  digitalWrite( nstpena, LOW );   // enable stepper controller
```

Setting the direction:

```
  #define step_cw HIGH   // for clockwise, the direction pin is HIGH
  #define step_ccw LOW   // for counterclockwise, the direction pin is LOW
  […]
  digitalWrite( dir1,  step_cw  );
```

One step is performed like this:

```
  void makeStepA( void ) {                        // make a step on motor A
    digitalWrite( step1, HIGH );
    delayMicroseconds( puls_width );
    digitalWrite( step1, LOW );
  }
```

The Piezo Buzzer

Pin definition:

```
  #define buzzer 9                    // buzzer
```

Pin initialization:

```
  pinMode( buzzer, OUTPUT );                 // the buzzer is an output
```

Using the piezo buzzer:

```
     // beep "hello"
    tone( buzzer, 4000 );               // 1kz sound on buzzer
    delay( 1000 );                      // for 1 second
    noTone( buzzer );
```

The LCD-Display

The I²C LDC Display requires including two libraries:

```
  #include <Wire.h>                    // import Wire library
```

```
#include <LiquidCrystal_I2C.h>      // import LiquidCrystal_I2C library
```

The initialization includes the I²C address, the number of column and the number of lines. Here a 16x2 display is used.

```
LiquidCrystal_I2C lcd(0x27, 16, 2);  // define the lcd display (i²c address
                                     // 0x27, 16 columns, 2 lines
```

The Timer 1 Interrupt

The Timer 1 interrupt is actually the heartbeat of the software. This interrupt occurs periodically and the so called ISR (Interrupt Service Routine) is executed. The cycle duration/periodicity depends on the setup (parameters) of Timer 1.

Note: Many beginners and intermediate programmers are afraid of using interrupts. This fear is actually not required! The ISR can be imagined as a sub routine, that is executed on interrupt, which is not noticed by the main program and can happen at any time. It is a task, that is executed in the background. The communication between the main program and the ISR is accomplished with variables.

There are two main rules for the Interrupt Service Routine:

1. *Keep it simple (the execution times should not be too long)*
2. *The variables, which need to be seen by the main program and the ISR need to be declared as "volatile" to disable compiler optimizations, that might want to store such a variable in a register. The main program will look up the value by loading the register, while the ISR has changed the value in memory.*

Rule number 2 is violated quite often, which leads to an unreliable/unpredictable processing of the events, that cause the interrupt.

The declaration of a "volatile" variable looks like this:

```
volatile boolean step_flag = 0;
```

It works with all basic variable types. step_flag is set by the ISR (after a defines time has elapsed), the main loop is checking if step_flag is set, if so, a step is performed and step_flag is then reset by the main loop. That simple!

The setup of the timer interrupt looks like this:

```
/* ==================== TIMER 1 setup ======================= */
// Interrupt every 0.000496 sec (= 2016.13Hz)
// prescaler = 256
// Compare Match Register = 30

cli(); // disable interrupts
// reset timer1
TCCR1A = 0;                  // set TCCR1A register to 0
TCCR1B = 0;                  // set TCCR1B register to 0
TCNT1  = 0;                  // reset counter value

OCR1A = 30;                  // set compare match register of timer 1

TCCR1B |= (1 << CS12);       // 1:256 pre-scaling for timer1

TCCR1B |= (1 << WGM12);      // turn on CTC mode
```

```
TIMSK1 |= (1 << OCIE1A);      // enable timer compare interrupt

sei();                        // allow interrupts
```

The clock frequency is 16MHz for the Pro Micro (5V version). This frequency is divided by the prescaler (here 256). The result is a frequency of 62.5kHz, which is the input frequency of timer 1. The compare/match register is set to 30. So, timer 1 counts from 0 to 30 before the timer interrupt occurs. That are 31 counts.

$$T_{TIMER1} = \frac{prescaler \cdot (OCR1A + 1)}{16MHz} = \frac{256 \cdot (30 + 1)}{16MHz} = 0.496msec$$

CTC mode means the described cyclic counting mode.

The prescaler has a just few specific modes.

CS12	CS11	CS10	Prescaler
0	0	1	1:1 (no prescaler)
0	1	0	1:8
0	1	1	1:64
1	0	0	1:256
1	0	1	1:1024

A recommended reading about the timer interrupts is:
https://www.simsso.de/?type=arduino/timer-interrupts

This website includes a parameter calculator.

The Interrupt Service Routine

The ISR includes the timing for the steps and the processing of the rotary encoder signals.

```
ISR(TIMER1_COMPA_vect) {           // function which will be called
                                    // when an interrupt occurs at timer 1
  // ========= every one second ========
  if (++t1_ticks == t1_ticks_sec) {    // set tick_1s every 2016 IRQ
   cycles (every 1 second)
    tick_1s = 1;
    t1_ticks = 0;                      // reset the tick counter
  }

  // === count the time before the next stepper step ===
  if (++step_ticks == step_duration) {
    step_flag = 1;
    step_ticks = 0;
  }

  // ============== rotary encoder =====================
  rot_read = digitalRead( rot_clk );
  if ( rot_read != rot_clk_status) {         // is rot_clk different for
                                              // <debounce> cycles?
    if (--rot_clk_debounce == 0) {
      rot_clk_status = rot_read;             // if yes: change status
    }
  }
  else {
    rot_clk_debounce = debounce;             // if it is equal -> reset
                                              // the debounce counter
```

```
    }
    if (rot_clk_status != rot_clk_status_old) { // did a status change
                                               // occur?
       if (rot_clk_status == LOW) {             // is it a falling edge?
         if (digitalRead( rot_data ) == LOW) { // yes: set the roraty value
                                              //according to rot_data
           rot_value--;                       // LOW -> CCW
         }
         else {
           rot_value++;                         // HIGH -> CW
         }
       }
       rot_clk_status_old = rot_clk_status;     // update old status
    }
    rot_read = digitalRead( rot_sw );           // read rot switch
    if ( rot_read != rot_sw_status) {           // is rot_sw different for
                                               // <debounce> cycles?
      if (--rot_sw_debounce == 0) {
        rot_sw_status = rot_read;               // if yes: change status
      }
    }
    else {
      rot_sw_debounce = debounce;               // if it is equal -> reset
                                               // the debounce counter
    }
    if (rot_sw_status != rot_sw_status_old) {       // did a status change
                                                    // occur?
       if (rot_sw_status == LOW) {                  // is it a falling
                                                    // edge?
         rot_button = true;                         // set switch semaphore
       }
       rot_sw_status_old = rot_sw_status;           // update old status
    }
  }
```

The ISR consists of three parts:

1. A 1sec tick
2. A tick per desired (micro) step
3. The rotary encoder debouncing and processing (refer to the following chapter)

The 1sec tick is setting the "flag" `tick_1s` to 1. This flag can be monitored by the main loop. The main loop has to reset this flag after "detection". This is called a "semaphore", which means a structure, that is used for signaling between different processes (here it is the ISR and the main loop).

`step_flag` is the semaphore, which signalizes the main loop, that the time for the next set has elapsed. The main loop has to reset this flag after detection, as well.

The Rotary Encoder

The rotary encoder part of the ISR (refer to the previous chapter) is supervising the "clock" and the "switch" signals.

The rotary encoded used here has mechanic contacts. Those are bouncing, when they change state. That means they are quickly oscillating for a while, before they finally settle. To detect a valid status change of the rotary encoder, the supervised signal has to stay the same for the debouncing period of time, which is determined be the constant `debounce` (a certain number of ISR cycles).

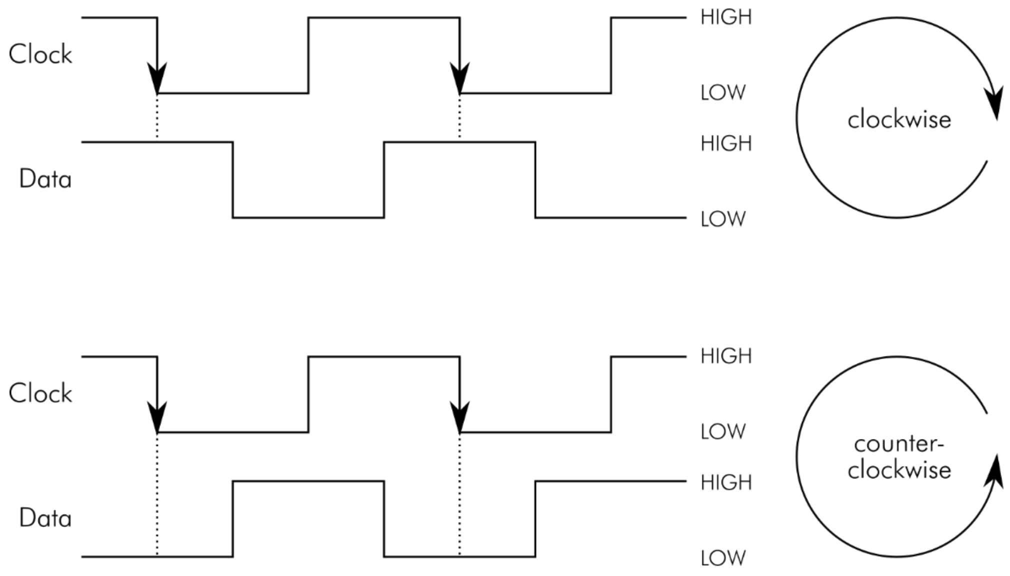


Figure 2: Operation of the rotary encoder

The relation between the clock and the data signal of the rotary encoder depends on the rotation direction. It is plus or minus a quarter period (that is +/- 90°, 360° is a complete period).

The direction can be detected by reading the data signal right after the falling edge of clock. This is accurate enough for a rotary encoder serving as an input device.

The interrupt service routine (ISR) is detecting a falling edge by comparing the recently read (valid) status of the clock signal (`rot_clk_status`) and the previously read status (`rot_clk_status_old`). In case both are different and the present status is LOW, a falling edge is detected. Now, the data signal is read. Depending on the state of the data signal, the variable, that holds the number of rotary encoder clicks (`rot_value`) is counted up (clockwise turn) or down (counterclockwise turn). `rot_value` is another semaphore, which is communicating the rotary encoder operation to the main loop.

The status of the rotary encoder's push button is also monitored by the ISR. Like the clock signal, it has to be debounced. Since the button is a switch to ground (GND), a pushed button shows as reading a LOW state. The (after debouncing) valid state of the button is stored in the semaphore `rot_button`. The variable is true, if a button push is detected.

Processing the rotary encoder in the main loop is fairly simple. Just the previously mentioned semaphores have to be monitored.

Detecting the button:

```
lcd.setCursor(14,0);
if (rot_button==true) {                          // rotary encoder:
                                                 // button pushed
    rot_button = false;                          // reset this flag
                                                 // after detection
    Serial1.print("Button->set (Channel ");
    if (stepper == step_cha) {
      Serial1.println("A)");
    }
    else {
      Serial1.println("B)");
    }
}
```

Or the direction of rotation:

```
if (rot_value != 0) {
        if (rot_value < 0) {
          num_steps = num_steps - 10;
          rot_value++;
        }
        else {
          num_steps = num_steps + 10;
          rot_value--;
        }
        lcd.setCursor(7,1);
        lcd.print( num_steps );
        lcd.print( "   " );
        Serial1.print( "Steps: " );
        Serial1.println( num_steps );
}
```

Here the number of steps (`num_steps`) is reduced by 10 on each "click" while `rot_value` is < 0 (direction = CCW) and increased by 10 while `rot_value` is > 0 (direction = CW).

RS-232

Other than the "regular" serial (via USB) interface, this RS-232 is addressed as ***Serial1*** in the Arduino IDE. The programming works like with the first serial interface. The recommended reading is https://www.arduino.cc/reference/en/language/functions/communication/serial/.

The initialization works with

```
Serial1.begin(9600);// the RS232 interface (RX, TX on pin 0 and 1) is
                    // Serial1.
```

This the 9600 bit/s (aka 9600 baud), 8N1 (8 data bits, No parity, 1 stop bit), a very common baud rate and data format.

Sending text via the serial interface is fairly simple:

```
Serial1.println( "Hello RS-232" );    // say hello
```

The RS-232 interface is also for receiving commands from a host. To collect this data and later process it (parsing the string = extracting the commands and parameters), some RAM space (a buffer) is allocated. For the parsing, the String functions are required

```
String recString = "";    // receive command buffer
```

[…]

```
recString.reserve(80);
```

A process, that is executed on every event on the Serial1 (RS-232) interface is

```
void serialEvent1() {
  while (Serial1.available()) {        // bytes available on Serial1?
    recByte = (char)Serial1.read();    // read it
    if (recByte == (char)10) {         // is it "new line character"?
      recComplete = true;              // yes: the line is complete
    }
    else {
```

```
      recString += recByte;          // no: append character to input
     string
      }
    }
  }
```

As long as there is data available on the interface, it is checked, if the data byte (actually a character) is a “Line Feed” (= (char)10). The line feed denotes the end of a command string. If so, a semaphore (`recComplete`) is set true. If it is a different character, this will be appended to the buffer `recString`.

The parsing of the received string in the main loop looks like this:

```
  if (recComplete) {       // a complete command line is received on RS-232
     lcd.setCursor(0,1);   // position the cursor on the second line
     lcd.print( recString );           // print the received string
     lcd.print( "                 " );   // clear the line after that string
     recString.toUpperCase();          // convert the line to upper case
                                       // (it is not case sensitive)
     syntaxError = false;              // reset the Syntax Error flag


     if (recString[0] == 'A') {       // is the first character an A?
       stepper = step_cha;            // yes: select Stepper#1 (channel A)
     }
     else  if (recString[0] == 'B') { // is the first character a B?
       stepper = step_chb;             // yes: select Stepper#2 (channel B)
     }
     else {
       syntaxError = true;             // else, the syntax is not correct.
     }
     if (!syntaxError) {               // syntax still correct?
       int pos = (int) recString.indexOf(","); // find the first comma ","
       if (pos < 1) {                  // if no comma is found
         syntaxError = true;           // set the Syntax Error Flag
       }
     else {                            // a comma is found:
       recString.remove(0,pos+1);      // remove the beginning, the comma
                                       // included
       num_steps = (int) recString.toInt();// convert the string to
                                            // integer -> num_steps
       if (num_steps<0) {              // is it a negative number?
          num_steps = -num_steps;      // yes: make it positive
          st_direction = step_ccw;     // the direction is counterclockwise
       }
       else {                           // no:
         st_direction = step_cw;        // the direction is clockwise
       }
       digitalWrite( dir1,  st_direction  );  // set the direction of both
       digitalWrite( dir2,  st_direction  );  // stepper motors
     }
  }
  if (syntaxError) {
     Serial1.println("? SYNTAX ERROR");    // report the syntax error
  }
  else {
    Serial1.print("Execute: Channel ");    // report the execution
    switch (stepper) {
      case step_cha: {
```

```
        Serial1.print("A, ");              // Channel A
        break;
      }
      case step_chb: {
        Serial1.print("B, ");              // Channel B
        break;
      }
    }
    Serial1.print( num_steps );            // and the number of steps
    if (st_direction == step_cw) {
      Serial1.println(" steps, CW");       // and the direction
    }
    else {
      Serial1.println(" steps, CCW");
    }
  }
  recComplete = false;         // reset the semaphore
  recString = "";              // reset the recString (receive buffer)
```

The command line will let the selected stepper motor turn CW or CCW by the sent number of steps.

A proper command line looks like this:

<channel>, <number of steps>

While <channel> is A or B or a or b (it is not case sensitive) and a positive <number of steps> will turn the clockwise and a negative <number of steps> will turn counter clockwise.

State Machine

The main loop is designed as a (finite-)state machine. This is a very common structure for micro controller software, which is quite useful to know.

Depending on the state, the actions and reactions are different. This (desired) behavior can be depicted in a state diagram (Figure 3).

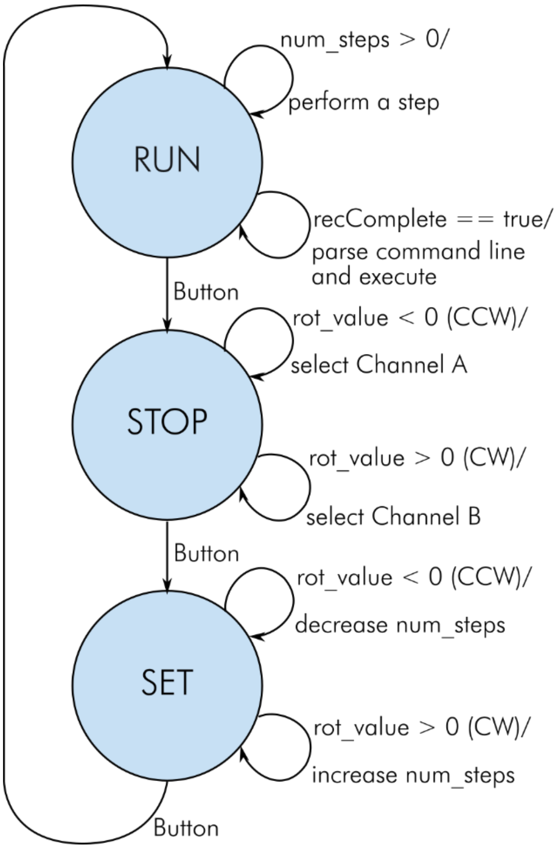


Figure 3: State diagram

The state machine has three states: RUN, STOP and SET. The states are represented by int values defined like this:

```
// state machine states
#define sm_stop  // assigning a numbers to the state machine states
#define sm_set 1
#define sm_run 2
```

The state RUN is entered first, it checks, if `num_steps` is greater 0, if so, it performs a step and decreases `num_steps` by one.

Also, it checks, if a command line (via RS-232) is complete (`recComplete` is true), if so, the command line is parsed and executed.

In case the push button of the rotary encoder is pressed, it enters the state STOP.

The state STOP watches the rotary encoder. If it is turned counterclockwise, Stepper channel A is selected, if it is clockwise, Stepper channel B is selected.

If the push button is pushed, the state SET becomes active.

While SET the number of steps to perform next and the direction is adjusted. If the rotary encoder is turned clockwise, the variable `num_steps` is increased by 10, while counterclockwise, `num_steps` is increased by 10. A negative number will select counterclockwise for both stepper motors. Pressing the push button will enter the state RUN.

The over-all structure of the state machine is a switch/case construction in the main loop.

```
switch (state_machine) {
  case sm_stop : {
```

[…]

```
    break;
  }

  case sm_set : {
```

[…]

```
    break;
  }
  case sm_run : {
```

[…]

```
    break;
  }
}
```

Connector Pinouts

J1 - Power Connector

J1 is a 2.5mm/5.5mm barrel jack.

Pin	Signal
Inner	+12VDC
Outer	GND

J2 – RS-232 Serial Interface

D-Sub, 9 pins, female

Pin	Signal	Note
1	DCD	Feedback DCD – DTR - DSR
2	RxD	Output
3	TxD	Input
4	DTR	Feedback DCD – DTR – DSR
5	GND	Ground
6	DSR	Feedback DCD – DTR – DSR
7	RTS	Feedback RTS - CTS
8	CTS	Feedback RTS – CTS
9	RI	Not connected

J3 – Rotary Encoder

KF2510 series, 5pin (or Molex P/N 22272051) – vertical header with friction lock. Pinout for KY-040 rotary encoder module.

Pin	Signal	Note
1	GND	
2	+5V	
3	Switch	Button
4	Data	Actually, Channel A
5	Clock	Actually, Channel B

The counterpart of J3 is a crimp housing (also KF2510 series, which is wide spread on Ebay, AliExpress and other online shops) and fitting crimp terminals. Alternatively, Molex P/N 22013047 and crimp terminal P/N 0850-0114 can be used.

J4, J5 – Stepper Motor Outputs

KF2510 series, 4pin (or Molex P/N 22272041) – vertical header with friction lock or horizontal (90°) Molex P/N 22057048. J4 is Channel A/Stepper 1, J5 is Channel B/Stepper 2.

Pin	Signal	Note
1	1B	stepper motor blue cable
2	1A	stepper motor red cable
3	2A	stepper motor green cable
4	2B	stepper motor black cable

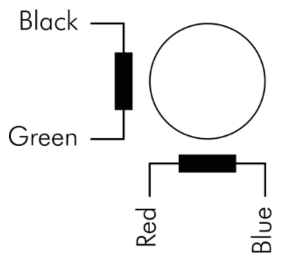


Figure 4: Windings of the stepper motor/cable colors

J6 – Reserved I/O-Pins

The free GPIO-Pins of the Pro Micro are connected to a pin header (6 pins, 2.54mm pitch) which does not require to be placed. These pins might serve for connecting sensors/switches/actors

Pin	Signal	Note
1	GND	
2	D16	GPIO D16, PCINT2
3	A0	Analog input/GPIO
4	A1	Analog input/GPIO
5	A2	Analog input/GPIO
6	A3	Analog input/GPIO

J7 – I²C/Display Connector

KF2510 series, 4pin (or Molex P/N 22272041) – vertical header with friction lock. Pinout for LCD Displays (16x2) with I²C controller.

Pin	Signal	Note
1	SCL	I^2C clock
2	SDA	I^2C data
3	+5V	
4	GND	

The counterpart of J7 is a crimp housing (also KF2510 series, which is wide spread on Ebay, AliExpress and other online shops) and fitting crimp terminals. Alternatively, Molex P/N 22012041 and crimp terminal P/N 0850-0114 can be used.

J8 – TX/RX Reserve Header

The TX and RX signals are routed to a pin header (3 pins, 2.54mm), which is not placed, in case it is desired to repurpose them or to watch the data exchange on the serial port/RS-232.

Pin	Signal	Note
1	GND	
2	TX	serial output, GPIO D1
3	RX	serial input, GPIO D2

Jumpers

There are two jumpers, JP1 and JP2. Both serve the setting of the step modes (full step, ½ step, ¼ step, 1/8 step and 1/16 step). JP1 determines the mode of stepper #1/channel A, JP2 does so for stepper#2/channel B).

MS1 / JPx 1-2	MS2 / JPx 3-4	MS3 / JPx 5-6	Mode/micro step resolution
L/open	L/open	L/open	Full Step
H/set	L/open	L/open	Half Step
L/open	H/set	L/open	Quarter Step
H/set	H/set	L/open	Eighth Step
H/set	H/set	H/set	Sixteenth Step

The footprints of the jumpers allow fix solder bridges instead of a pin header and jumper.

Assembly

The PCB

The PCB can be fully populated or some not requires parts can be omitted.

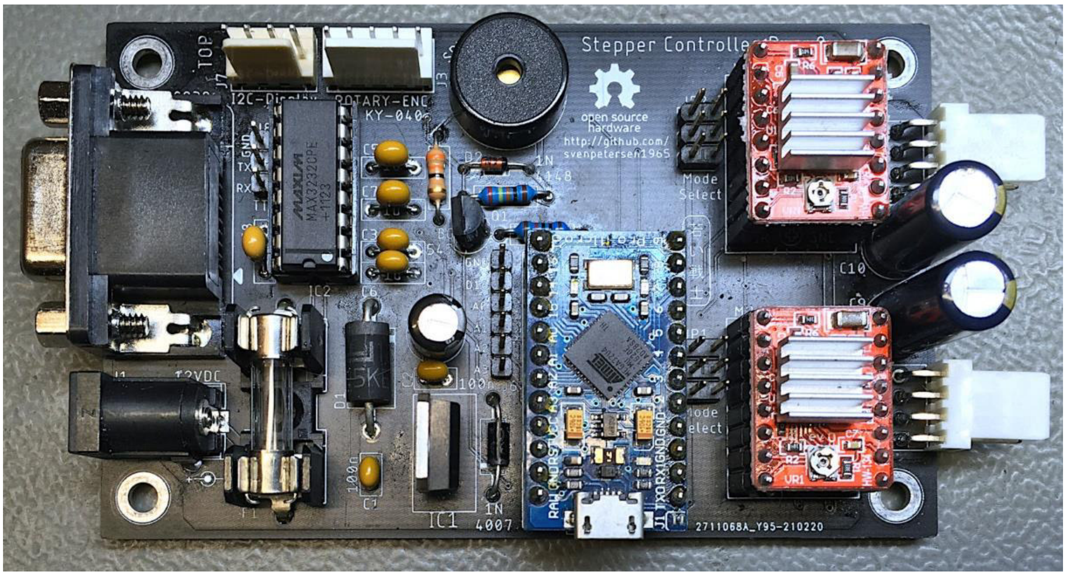


Figure 5: Fully populated PCB

Requirement	Components
Always populate	J1, F1, C1, C2, C4, D3, M1, M2, C9, J4
2nd stepper controller	M3, C10, J5
Buzzer	R1, R1, R2, R3, D2, SPK1
Display	J7
Rotary Encode	J3
RS-232	J2, IC2, C3, C5, C6, C7, C8
Micro steps required and jumpered	Channel A: JP1, Channel B: JP2. A fix setting of the jumpers can be accomplished with solder bridges
Do not place (until required)	J6, J8

Cables

In case the 90° pin headers are used for the stepper motor connection (J4, J5), there are only two cables required:

1. the display cable
2. the rotary encoder cable

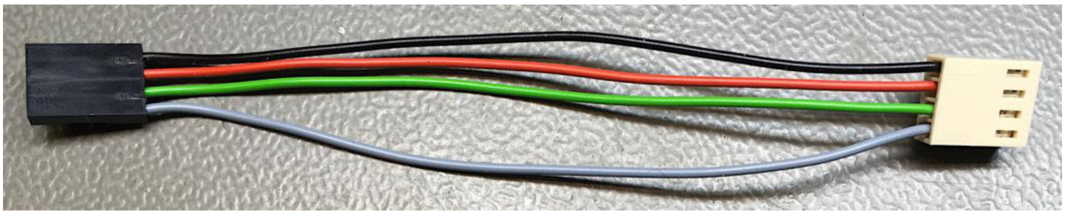


Figure 6: The display (I²C) cable

This display cable consists of the following components:

Pos.	Qty	Part
1	1	DuPont crimp housing, 4 pins
2	4	DuPont crimp terminals
3	14cm	Cable, 0.25mm²/AWG24, black
4	14cm	Cable, 0.25mm²/AWG24, red
5	14cm	Cable, 0.25mm²/AWG24, green
6	14cm	Cable, 0.25mm²/AWG24, grey
7	4	KF2510 crimp terminals
8	1	KF2510 crimp housing, 4 pins

The colors of the cables are a suggestion only.

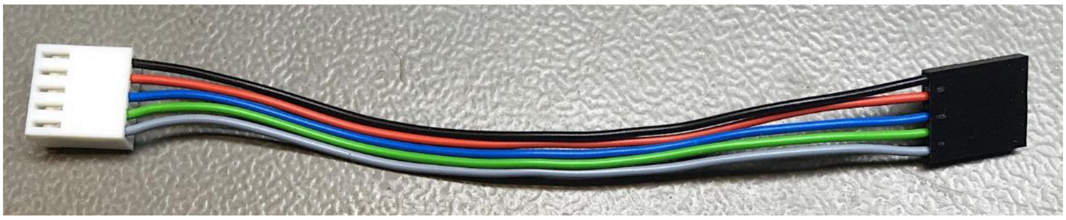


Figure 7: Rotary encoder cable

This rotary encoder cable consists of the following components:

Pos.	Qty	Part
1	1	DuPont crimp housing, 5 pins
2	5	DuPont crimp terminals
3	14cm	Cable, 0.25mm²/AWG24, black
4	14cm	Cable, 0.25mm²/AWG24, red
5	14cm	Cable, 0.25mm²/AWG24, blue
6	14cm	Cable, 0.25mm²/AWG24, green
7	14cm	Cable, 0.25mm²/AWG24, grey
8	5	KF2510 crimp terminals
9	1	KF2510 crimp housing, 5 pins

Optionally, a DIN connector can be used for the stepper motor connections.

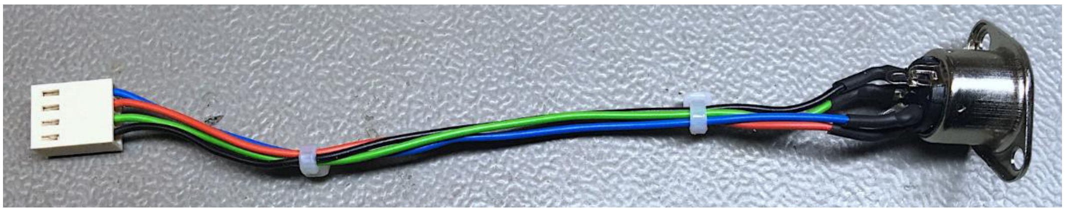


Figure 8: Stepper cable (internal, optional)

Pos.	Qty	Part
1	1	DIN Jack, 5 pins (e.g. Reichelt MAB 5S)
2	14cm	Cable, 0.25mm²/AWG24, black
3	14cm	Cable, 0.25mm²/AWG24, green
4	14cm	Cable, 0.25mm²/AWG24, red
5	14cm	Cable, 0.25mm²/AWG24, blue
6	5	KF2510 crimp terminals
7	1	KF2510 crimp housing, 5 pins
8	6cm	Shrinkable sleeve (2.4/1.2)

3D Printed Case

It is recommended to use the 3D printed case for this project. It consists of a top and a bottom shell and is constructed in Fusion 360. The STL files for 3D printing are provided as well as the Fusion 360 project.

The display is attached with 4 screws (C2.9x6.5 DIN 7981 recommended). The rotary encoder is attached with the nut included in it. The cables should be connected prior to mounting those components.

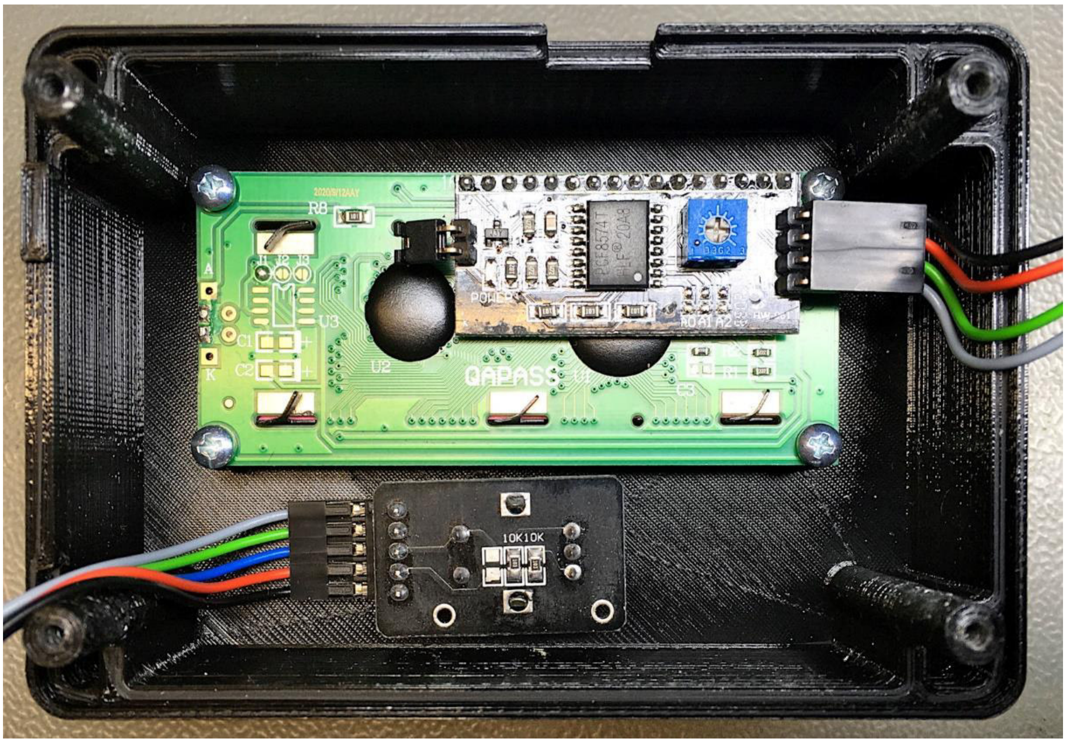


Figure 9: Top shell with mounted LCD display and rotary encoder KY-040

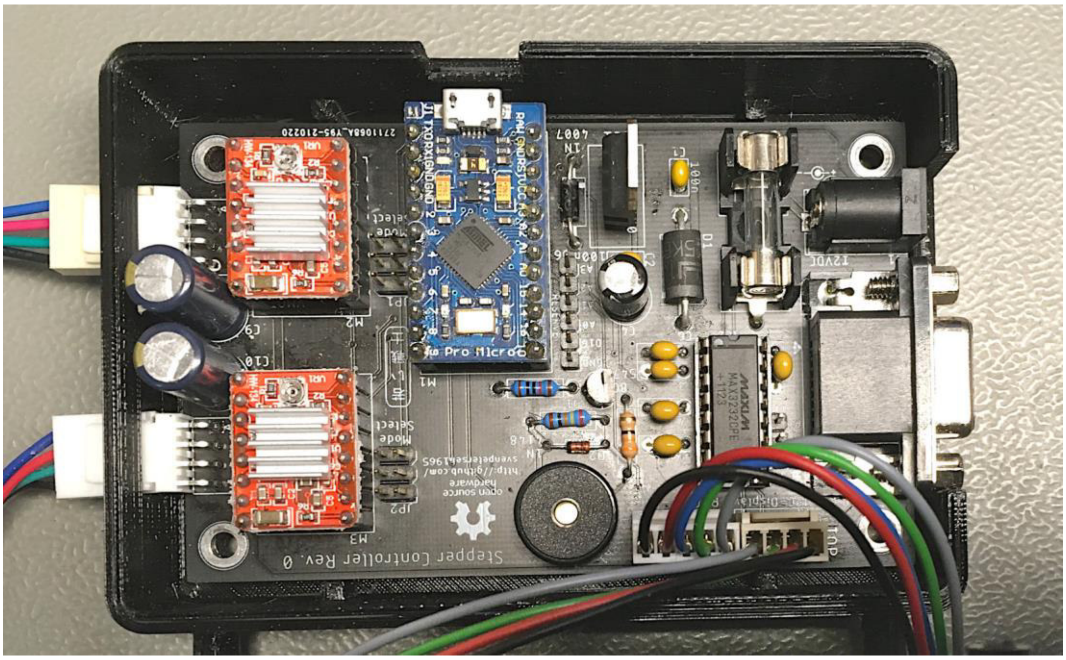


Figure 10: PCB in bottom shell

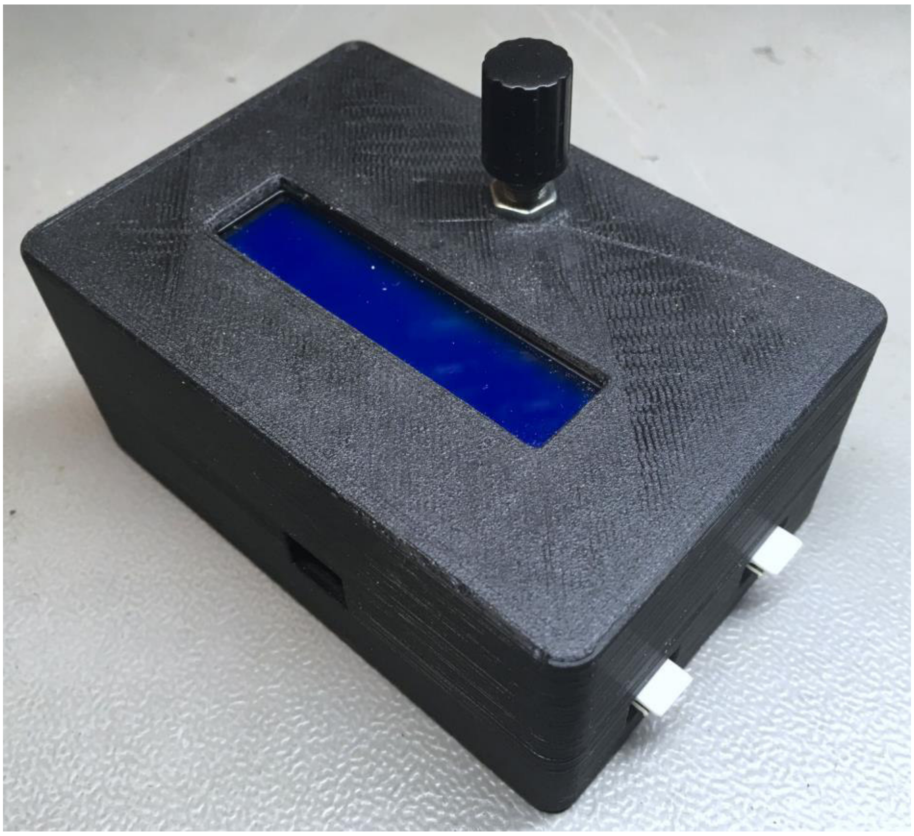


Figure 11: Case from top, left

Figure 12: case from bottom, right

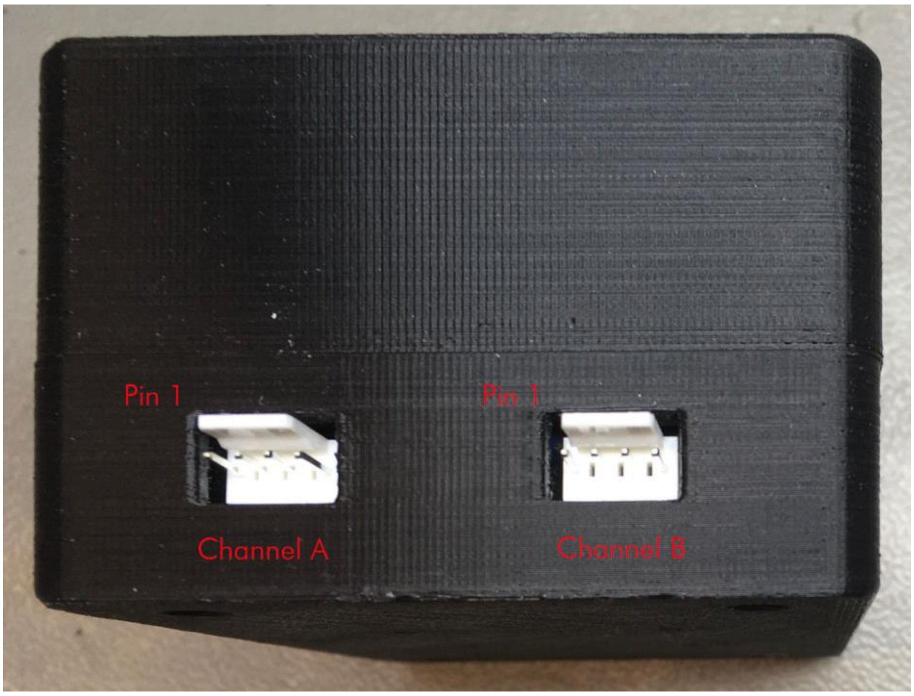


Figure 13: left side

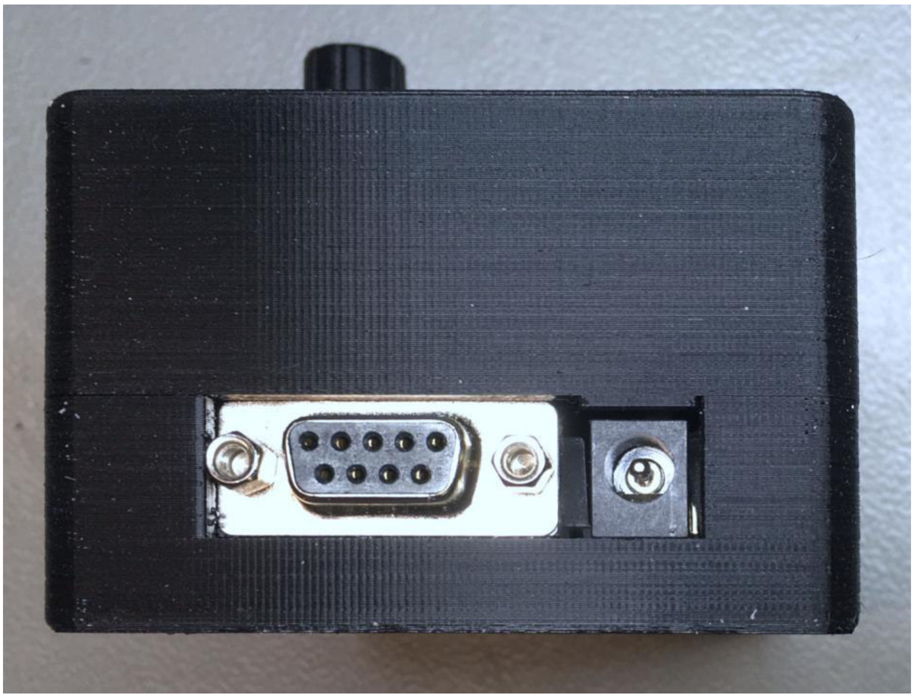


Figure 14: right side

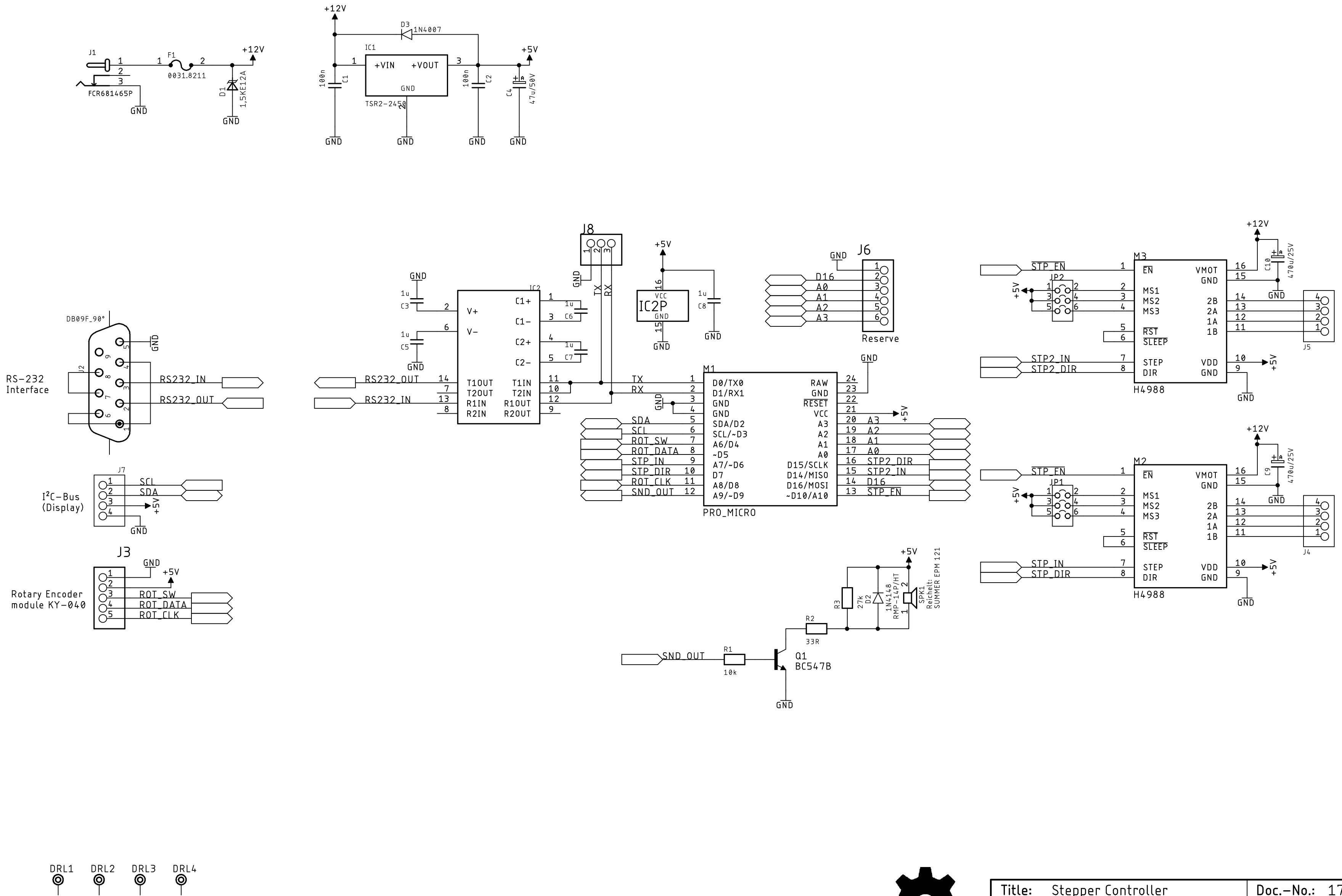
There are different versions of the top and bottom shell provided. The top shell can have a mounting space for a DIN connector (that could serve as a stepper motor connector), the bottom shell can have the cut outs for the 90° stepper motor connector as shown in Figure 10.

For closing the case, with 4 screws (C2.9x13 DIN 7981 recommended) are required.

Revision History

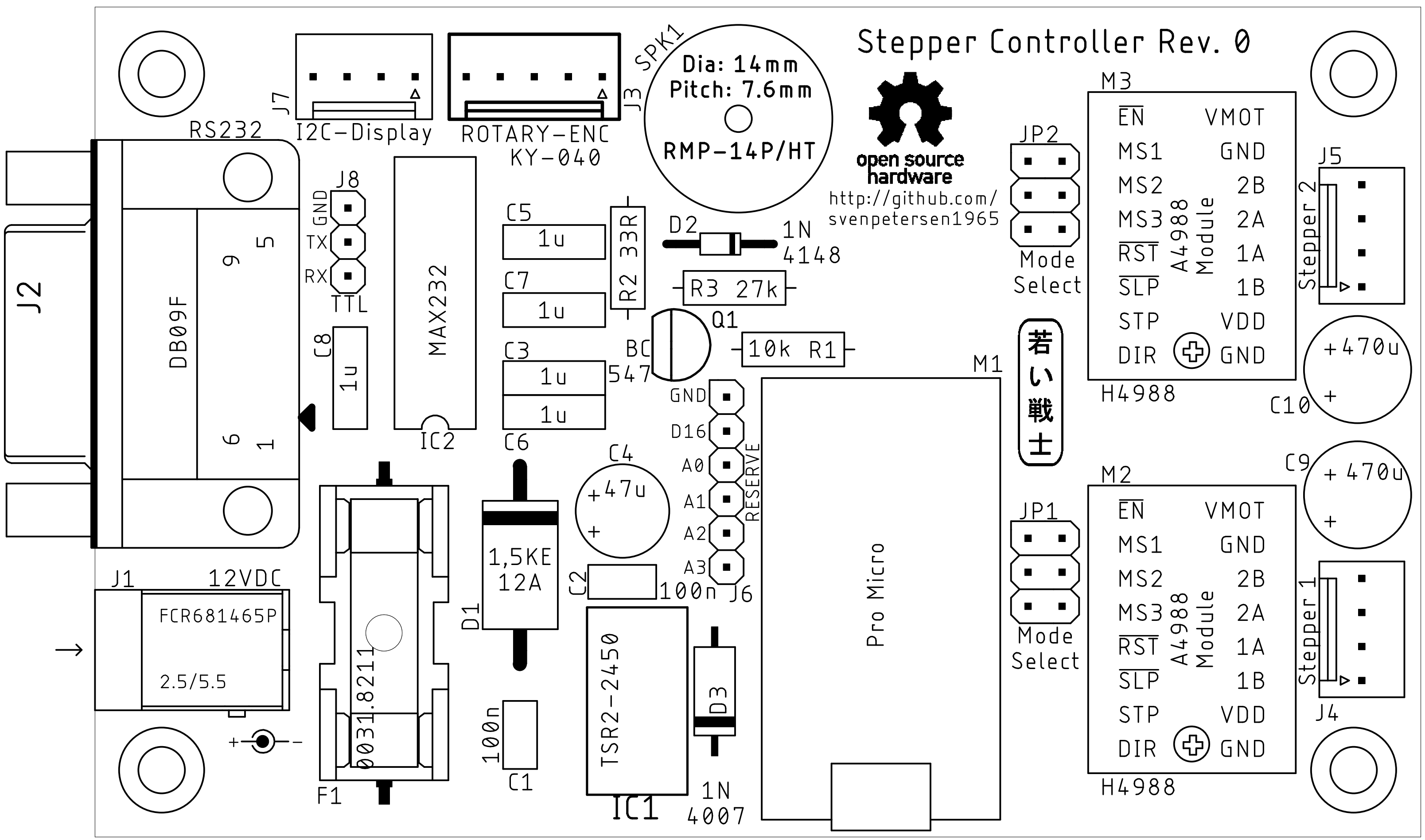
Rev. 0

- Fully functional prototype

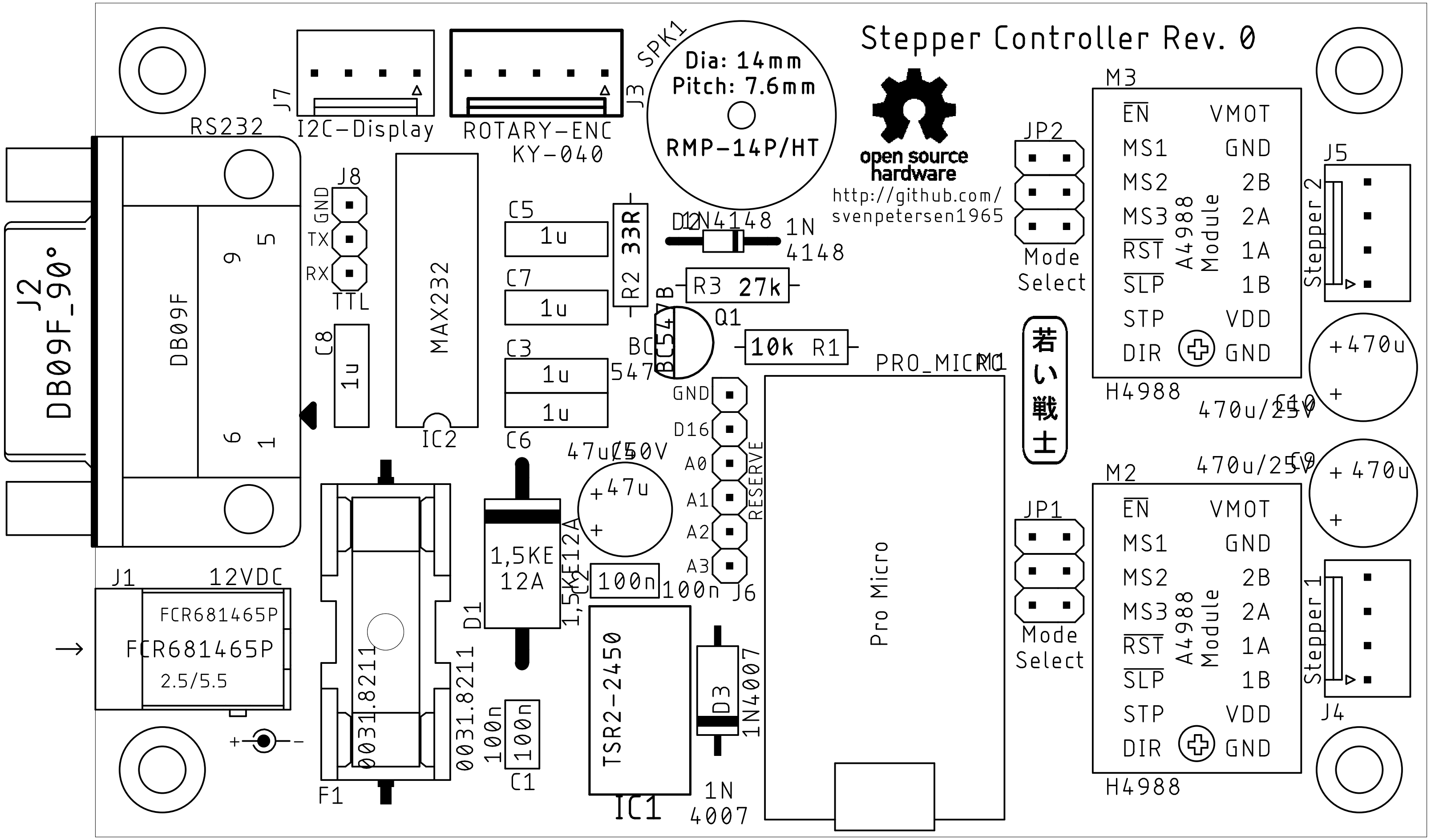


Title:	Stepper Controller 2 channels	Doc.-No.:	174-1-01-00
		Draft.:	Sven Petersen
Date:	18.02.2021 11:58	Rev.: 0	Page 1/1
File:	Stepper_Controller		
	http://github.com/svenpetersen1965		A3

Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:09		Rev.: 0
placement component side		

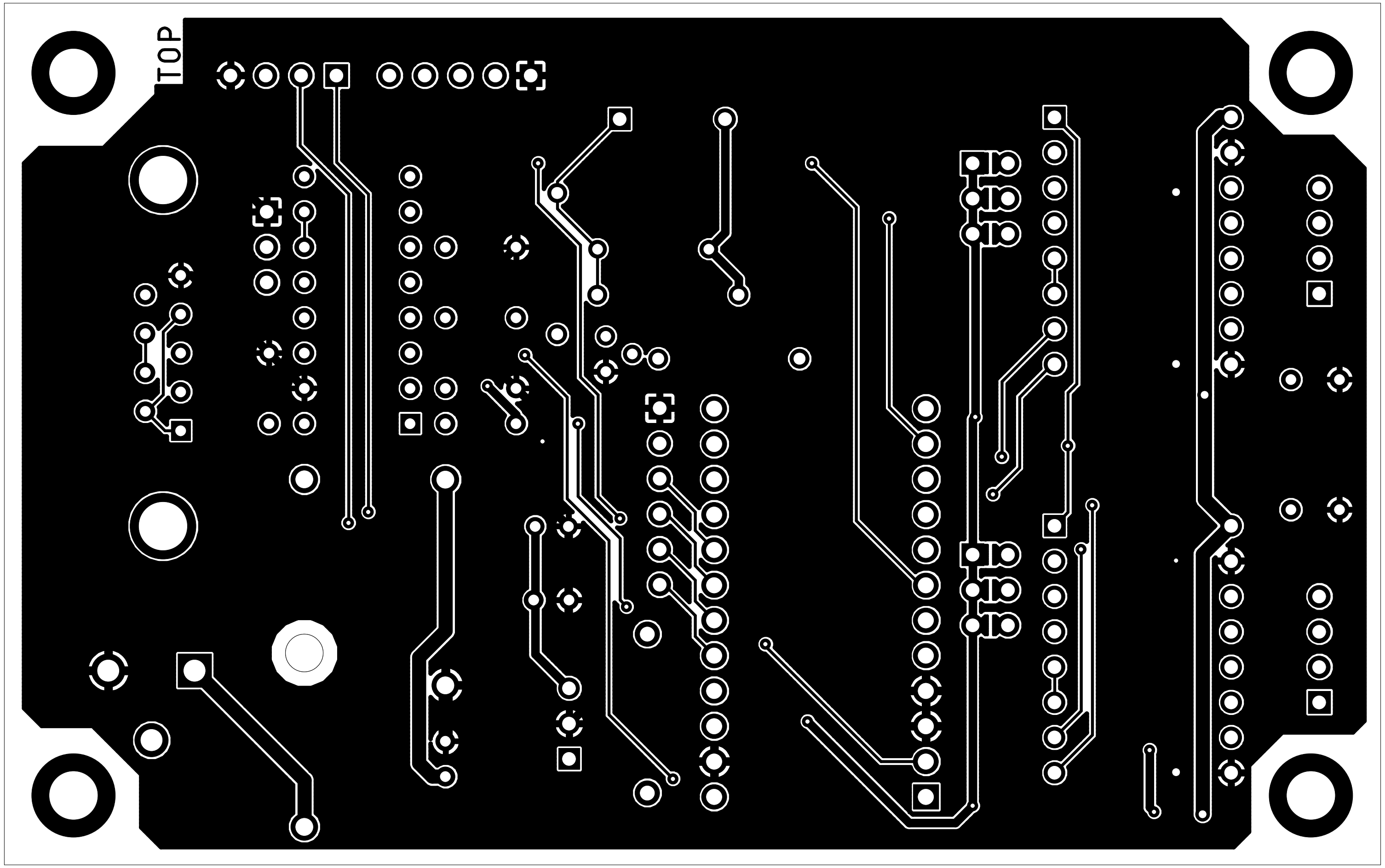


Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:20		Rev.: 0
placement component side		

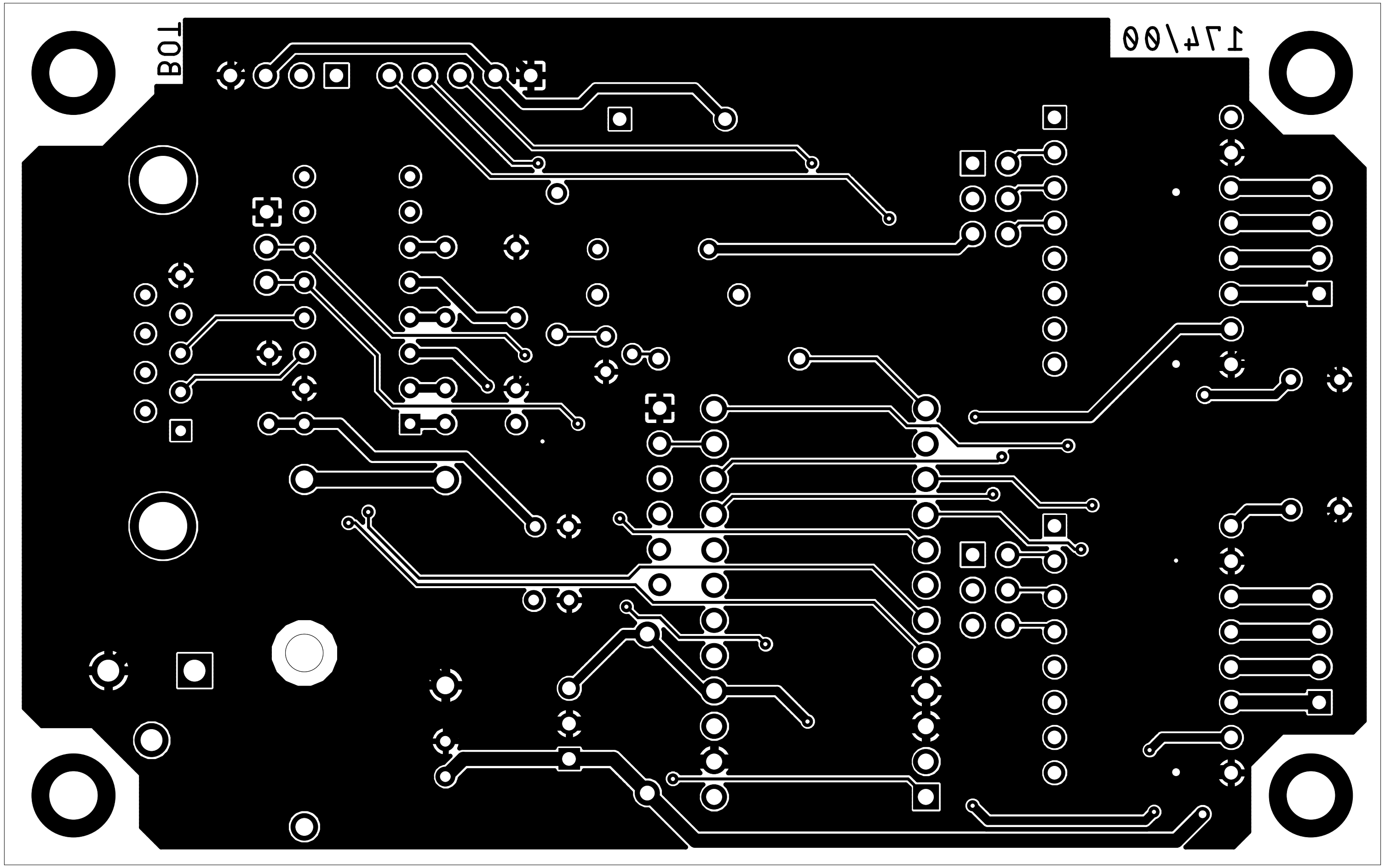


Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:20		Rev.: 0
placement solder side		

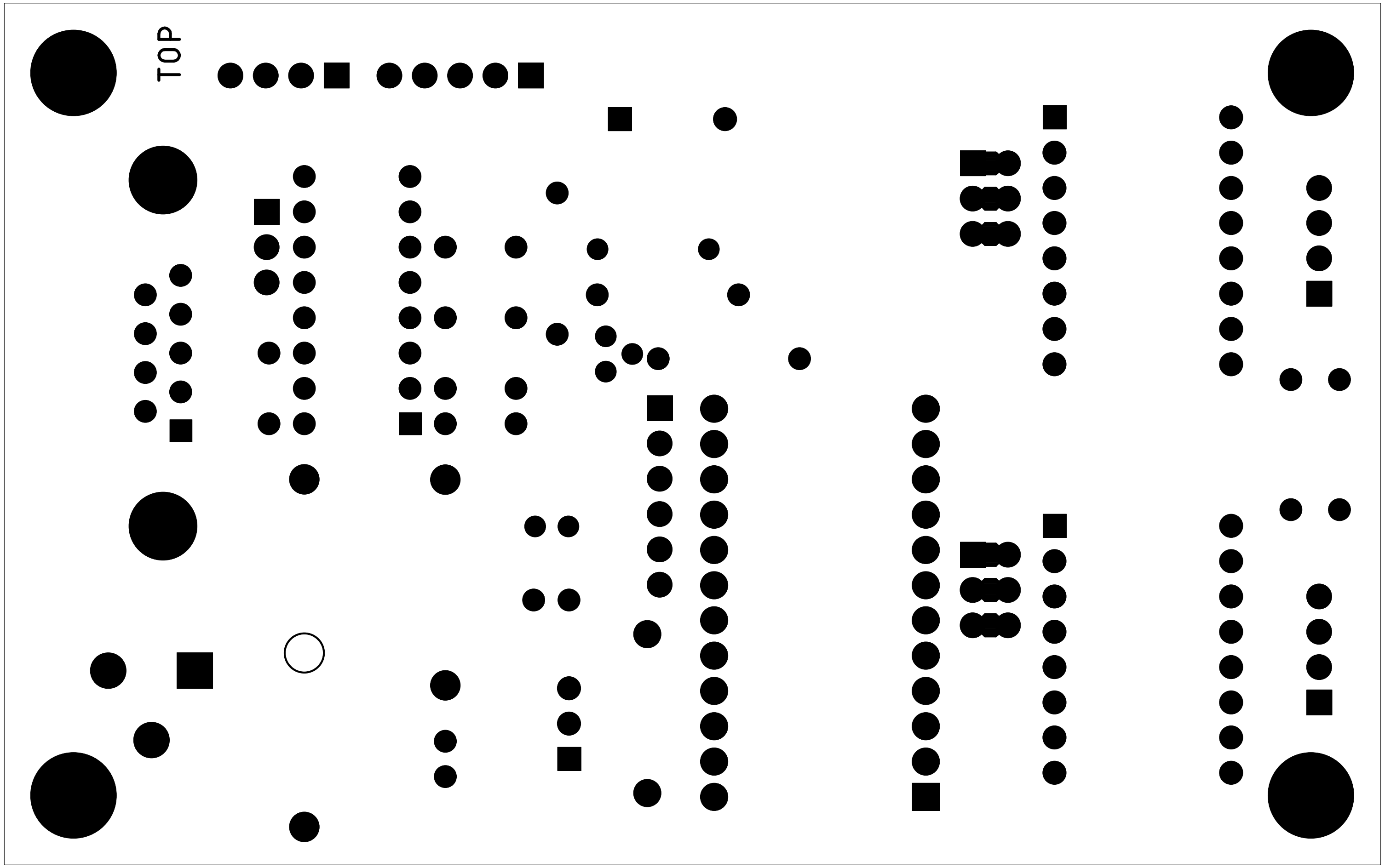
Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:20		Rev.: 0
top		



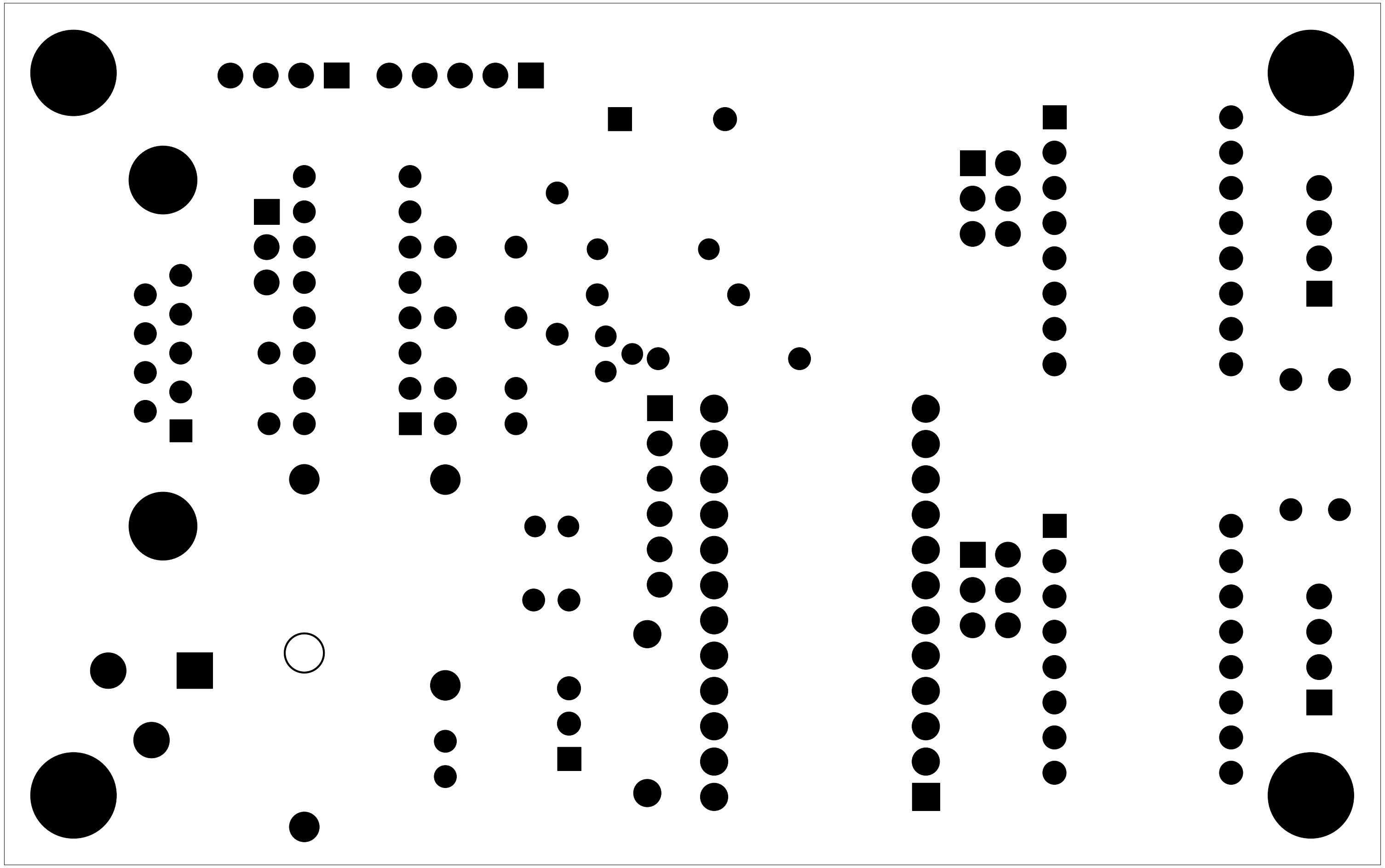
Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:20		Rev.: 0
bottom		



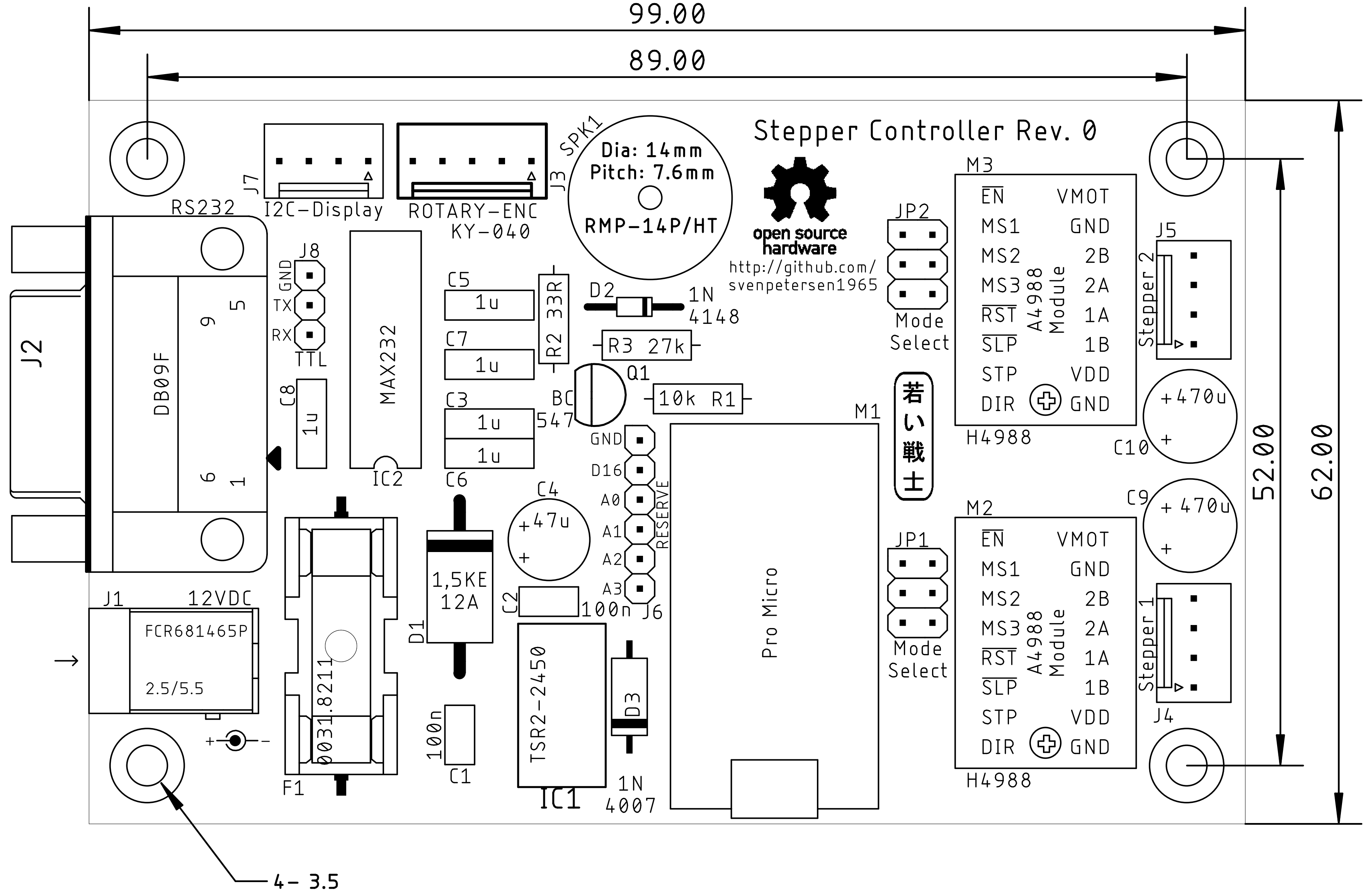
Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
nicht gespeichert!		Rev.: 0
stopmask component side		



Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
nicht gespeichert!		Rev.: 0
stopmask solder side		



Sven Petersen 2021	Doc.-No.: 174-2-01-00	
	Cu: 35µm	Cu-Layers: 2
Stepper_Controller		
18.02.2021 12:11		Rev.: 0
placement component side	measures	



Stepper Motor Controller Rev. 0

Functional Description

J1 is the power supply connector. F1 is the 5x20mm fuse, which might require to be adjusted to the application/stepper motors. D1 is a TVS diode for transient (0voltage spike) suppression. It is calculated for a 12V supply voltage. In case a different supply voltage is desired, a different TVS diode has to be selected.

IC1 is shown as a Traco TRS2-2450 DC/DC converter. Since this is pin compatible to the standard linear regulators, a 7805 can be populated instead of the DC/DC converter, which might be required for higher supply voltages (e.G. 24V for 24V stepper motors).

D3 is a protection when switching off the Stepper motor controller. It helps that the output voltage of IC1 is not much higher than the input voltage (due to charged capacitors).

M1 is a Pro Micro, a popular Atmel Atmega32U4 based micro processor module, which is compatible to the Arduino Leonardo. To send the program/script to the Pro Micro, the USB-B micro connector has to be connected to the PC, the Arduino IDE software is running, the Arduino Leonardo is selected and the proper COM-Port.

IC2 is the level translator for RS-232 (the logic levels of RS-232 are about +/-10V with the MAX232. C3, C5, C6 and C7 are part of the charge pump for generating the required voltages for the IC2. J2 is the RS-232 jack. CTS and RTS are connected as well as DTR, DSR and DCD.

J8 can be used to repurpose the RX and TX pins or for spying the serial traffic at TTL level.

J6 is a pin header for the unused I/O pins of M1. It usually does not require to be populated.

The circuit around Q1 serves as a buzzer amplifier/protection circuit. Actually, an IO-Pin of M1 could drive the buzzer, but the buzzer can act as a high voltage spike generator on a mechanical impact. Thus, the circuit. R1 is the base resistor of Q1, R2 is for current limiting, D2 is kind of a free-wheeling diode, that helps to prevent reverse voltages.

M2 and M3 are the Allegro A4988 based stepper drivers. M2 is for Channel A/Stepper 1 and M3 for Channel B/Stepper 2. C9 and C10 are the buffer capacitors for VMOT, the +12V supply voltage for the stepper motors. JP1 and JP2 configure the micro step mode. All jumpers open mean “full step”. An open jumper is interpreted as a LOW signale by the A4988.

There is one common Enable Signal ($\overline{\texttt{STP\_EN}}$) for both stepper motors. In case it is LOW, the stepper motors are active and the axis cannot be moved manually. It performs steps in this mode. In case $\overline{\texttt{STP\_EN}}$ is HIGH, no current flows through the stepper windings and the axis can be moved manually (at least, if the stepper does not have a gear).

J7 is a connector for the I²C-Bus. Usually, a display is connected. Since the I²C is a bus, additional I²C devices can be connected.

J3 is a connector for the rotary encoder. A KY-040 module, which is pretty wide spread and unexpensive can be connected here.

Stepper Motor Controller Rev. 0

Testing

Test Setup

Hardware

A fully populated Stepper Motor Controller Rev. 0 with a 16x2 I²C LCD-display was tested with three different stepper motors:

1. 17HS13-0404S (NEMA17, 12V, 0.4A/phase, 1.8° steps)
2. 17HS8-1004S (NEMA17, 12V, 1A/phase, 1.8° steps)
3. 17HS13-0404S-PG5 ((NEMA17, 12V, 0.4A/phase, 1.8° steps, 1:5.182 gear)

The voltage supply is a Goobay SE120P3000EU (12V/3A output on a 2.5mm/5.5mm barrel connector).

A USB-RS-232-Adapter was used for testing then RS-232 interface.

The software "stepper_framework.ino" was uploaded to the Pro Micro via the USB interface.

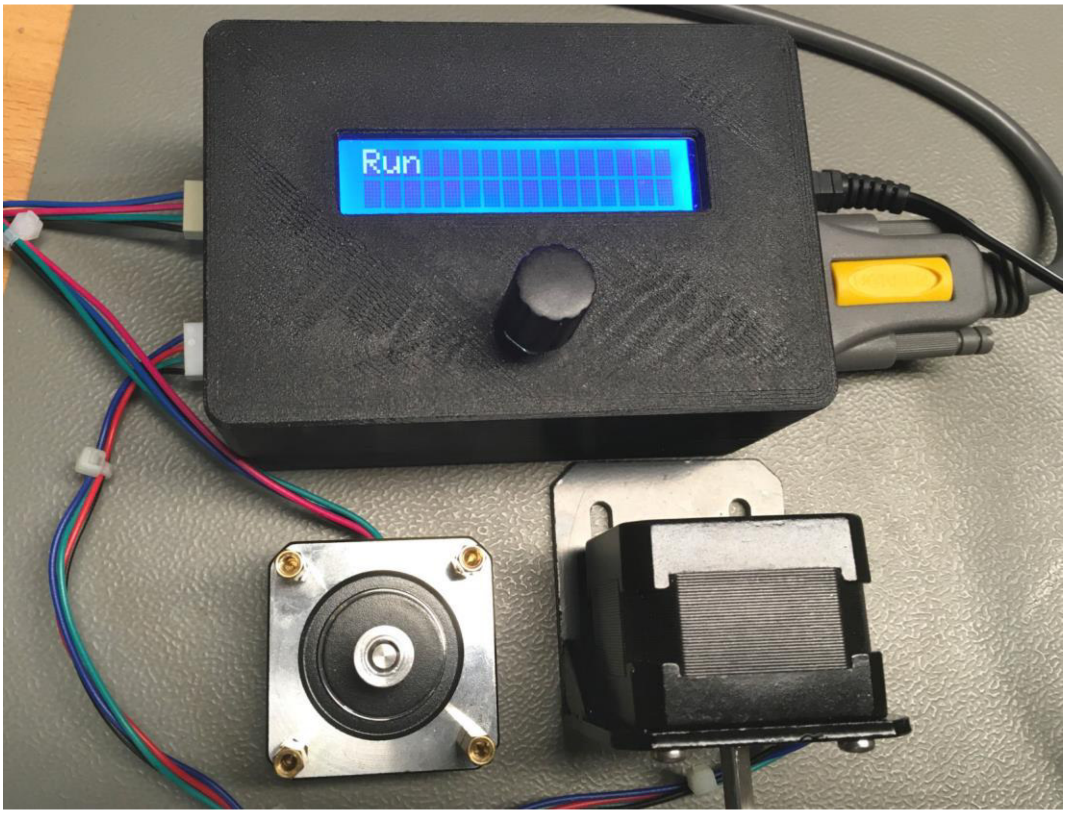


Figure 1: Test Setup

YAT – Terminal Program

To communicate with the Stepper Motor Controller via RS-232, a terminal program is required. The serial monitor of the Arduino IDE only works on the COM-Port, that is selected for sending the sketch. A recommended one is YAT ("Yet, Another Terminal").

https://sourceforge.net/projects/y-a-terminal/files/latest/download

Test Execution

Software upload

The software "stepper_framework.ino" was uploaded with the Arduino IDE successfully.

Power

After connection the +12V-PSU the Stepper Controller powered up properly. The +5V from the linear voltage regulator were measured as +5.03V at the stepper driver A4988. The +12V (12.3V without load) from the PSU were measured as +12.14V at the stepper driver, too.

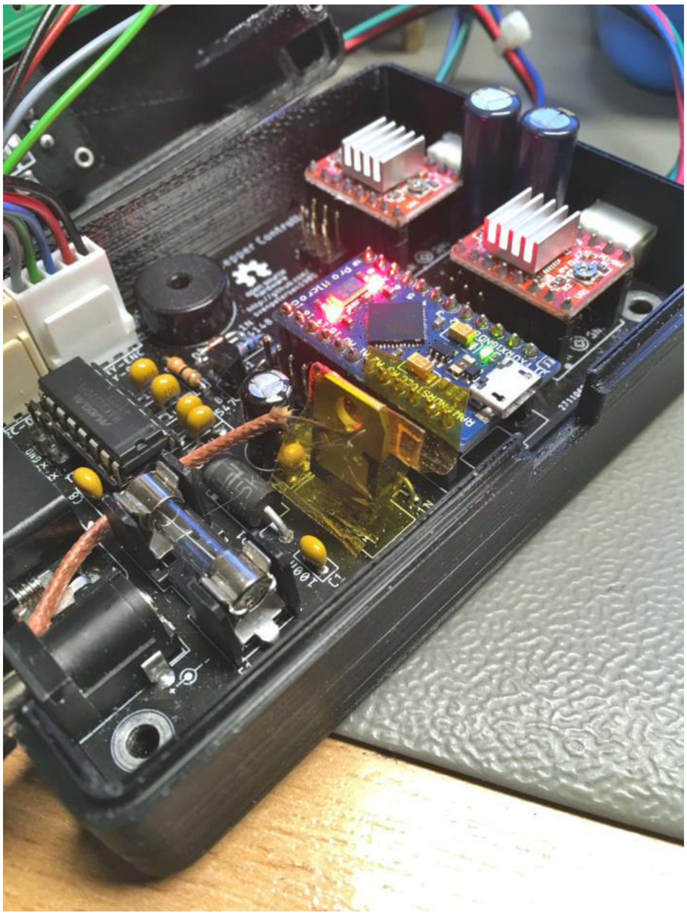


Figure 2: Thermal measurement on linear regulator IC1

The supply current was measured while in the "STOP" state (no current through the stepper motor windings). It is about 93.5mA. When in "RUN" state, a supply current of 850mA was observed. This does not change much while a stepper motor is running and mostly depends on the stepper motors used and the current adjustment of the stepper driver.

When powered from +12V, the linear regulator settles at about 57°C, which is non critical.

Time	Temperature
Start	21.6°C
Start + 75 minutes	56.9°C

The thermal resistance junction-case (R_{thJC}) is 5K/W according to the data sheet. The dissipated power is

$$P_D = (V_{in} - V_{out}) \cdot I = (12V - 5V) \cdot 93.5mA = 0.655W$$

Hence, the estimated junction temperature is

$$\vartheta_{Junction} = \vartheta_{Case} + R_{thJC} \cdot P_D = 56.9°C + 5\frac{°C}{W} \cdot 0.655W \cong 60.2°C$$

This is far below the absolute maximum rating of 150°C.

The measurements were executed with a EEVBlog 121GW multimeter. The thermo couple was fixed to the backside of the 78S05/IC1 (refer to Figure 2). The case was closed for the thermal measurement.

Piezo Buzzer

The piezo buzzer beeps after switching on the power. It can be heard well even while the case is closed.

LCD Display

The LCD display initialized properly and the text appeared as desired. The contrast had to be adjusted slightly (potentiometer on the LDC-display).

Rotary Encoder

The push button and both directions were detected properly with the installed software.

Stepper Driver

Both stepper motors executed the full steps set with the rotary encoder. Both directions work. Further, the half, quarter, 1/8 and 1/16 micro-steps were configured with the jumpers. The micro-steps were not carried out evenly, so an adjustment of the stepper current (potentiometer on the A4988 modules) was required. After setting the current limit properly, the steps were carried out evenly. The setting was adjusted "by ear".

All three stepper motors were tested.

RS-232 Interface

The RS-232 serial interface was tested with YAT (set to 9600baud, 8N1).

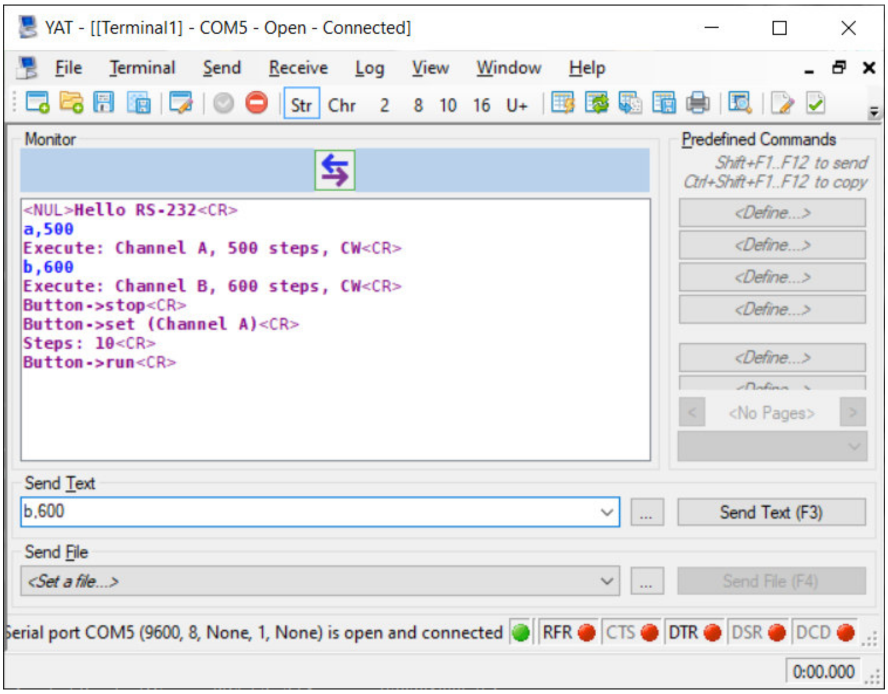


Figure 3: Terminal Program YAT communicating with the Stepper Motor Controller

Both, the send and receive direction work properly. The messages from the stepper controller were displayed correctly in YAT and the sent command were recognized by the stepper controller and executed.

Conclusion

The Stepper Motor Controller is working properly. The DC/DC-converter has not been tested.

Stepper Motor Controller Rev. 0
Bill of Material Rev. 0.0

Pos.	Qty	Value	Footprint	Ref.-No.	Comment
1	1	174-2-01-00	2 Layer	PCB Rev. 0	2 layer, Cu 35µ, HASL, 99.0mm x 62.0mm, 1.6mm FR4
2	2	KF2510 (vertical, 4pin)	6410-4P	J4, J5	or Molex 22-27-2041 (vertical, Reichelt MOLEX 22272041 or tme.eu:MX-6410-04A) or Molex P/N 22057048 (horizontal/90°, Reichelt: MOLEX 22057048 or tme.eu: MX-7395-04B)
3	1	KF2510 (vertical, 4pin)	6410-4P	J7	or Molex 22-27-2041 (always vertical, Reichelt MOLEX 22272041 or tme.eu:MX-6410-04A)
4	3	KF2510 crimp housing, 4 pins		(J4), (J5), (J6)	or Molex 22013047 (Reichelt: MOLEX 22013047, tme.eu: MX-22-01-3047)
5	12	KF2510 crimp terminals		(J4), (J5), (J6)	or Molex 08500114 (Reichelt: MOLEX 8500114) or Molex 08-70-0049 (tme.eu: MX-08-70-0049)
6	1	KF2510 (vertical, 5pin)	6410-5P	J3	or Molex 22-27-2051
7	1	KF2510 crimp housing, 4 pins		(J3)	or Molex 22013057
8	5	KF2510 crimp terminals		(J3)	or Molex 08500114 (Reichelt: MOLEX 8500114) or Molex 08-70-0049 (tme.eu: MX-08-70-0049)
9	1	MAX232CPE	DIL-16	IC2	Maxim or Renesas HIN232CPZ (Reichelt: MAX 232 DIP, tme.eu: HIN232CPZ)
10	1	pin header, 3 pin, 2.54mm pitch	1X03	J8	DNP
11	1	pin header, 6 pin, 2.54mm pitch	1X06	J6	DNP
12	1	0031.8211	318211	F1	Schurter fuse holder (Reichelt: PL OGN-25, tme.eu: 0031.8211)
13	1	5x20mm, 3.15AT		(F1)	This fuse depends on the power supply and the application
14	1	1,5KE12A	DO-201	D1	TVS-Diode, unidirektional, (Reichelt: 1,5KE 12A, tme.eu: 1.5KE12A-LF)
15	2	100n	C-2,5	C1, C2	ceramic capacitor, 2.5mm pitch
16	1	10k	R-10	R1	metal film resistor, 0.5W (or more), 5% or better

Stepper Motor Controller Rev. 0
Bill of Material Rev. 0.0

Pos.	Qty	Value	Footprint	Ref.-No.	Comment
17	1	1N4007	DO-41	D3	diode
18	1	1N4148	DO-35	D2	diode
19	5	1u	C-5	C3, C5, C6, C7, C8	ceramic capacitor, 5mm pitch (e.g. Reichelt Z5U-5 1,0μ)
20	1	27k	R-10	R3	metal film resistor, 0.5W (or more), 5% or better
21	1	33R	R-10	R2	metal film resistor, 0.5W (or more), 5% or better
22	2	470u/25V	C08/3,5	C9, C10	e-cap, ∅8mm, pitch 3.5mm, 105°C, e.g. Reichelt RAD FC 470/25
23	1	47u/50V	C07/2,5	C4	e-cap, ∅7mm, pitch 2.5mm, 105°C, e.g. Reichelt RAD FC 47/50
24	1	BC547B	TO92	Q1	universal NPN Transistor
25	2	COMBI-3X2P	COMBI-3X2	JP1, JP2	see document 174-6-01-**. Reichelt MPE 087-2-006
26	1	DB09F_90°	DS09F-H	J2	D-Sub, 9pin, female, 90°, (Reichelt
27	1	FCR681465P	FCR681465P	J1	Cliff barrel connector (2.5mm/5.5mm), Reichelt: CLIFF FCR681465P, tme.eu: FCR681465P
28	2	A4988	H4988	M2, M3	A4988 stepper driver module (AliExpress or Ebay etc.)
29	4	1x8 socket strip, 2.54mm		(M2), (M3)	e.g. MPE 115-1-008 (Reichelt: MPE 115-1-008, tme.eu: ZL307-1X8)
30	1	PRO_MICRO	PRO_MICRO	M1	**16MHz, 5V**, Micro Controller module (Arduino compatible) from AliExpress, Ebay etc. (ATTN: there are 8MHz, 3.3V versions, too, which are not suitable
31	2	1x12 socket strip, 2.54mm		(M1)	e.g. MPE 115-1-012 (Reichelt: MPE 115-1-012, tme.eu: ZL262-12SG)
32	1	RMP-14P/HT	RMP-14P/HT	SPK1	piezzo buzzer, ∅14mm, pitch 7.6mm, EKULIT (Reichelt SUMMER EPM 121): alternative tme.eu LD-BZPN-1002 (on cable)
33	1	7805 or TSR2-2450	TSR2	IC1	standard linear regulator or Traco DC/DC converter for higher supply voltages
34	1	LCD Display I²C, 16x2		(J7)	"standard" product from Ebay or AliExpress.
35	1	dupont crimp housing, 4 pins			For the I²C Display cable

Stepper Motor Controller Rev. 0
Bill of Material Rev. 0.0

Pos.	Qty	Value	Footprint	Ref.-No.	Comment
36	4	Dupont crimp terminals			For the I²C Display cable
37	14cm	cable, AWG24, black			For the I²C Display cable, suggested color
38	14cm	cable, AWG24, red			For the I²C Display cable, suggested color
39	14cm	cable, AWG24, green			For the I²C Display cable, suggested color
40	14cm	cable, AWG24, grey			For the I²C Display cable, suggested color
41	1	KY-040			Rotary encoder, "standard" product from Ebay or AliExpress.
42	1	dupont crimp housing, 4 pins			for the rotary encoder
43	4	Dupont crimp terminals			for the rotary encoder
44	14cm	cable, AWG24, black			for the rotary encoder, suggested color
45	14cm	cable, AWG24, red			for the rotary encoder, suggested color
46	14cm	cable, AWG24, green			for the rotary encoder, suggested color
47	14cm	cable, AWG24, grey			for the rotary encoder, suggested color
48	14cm	cable, AWG24, blue			for the rotary encoder, suggested color