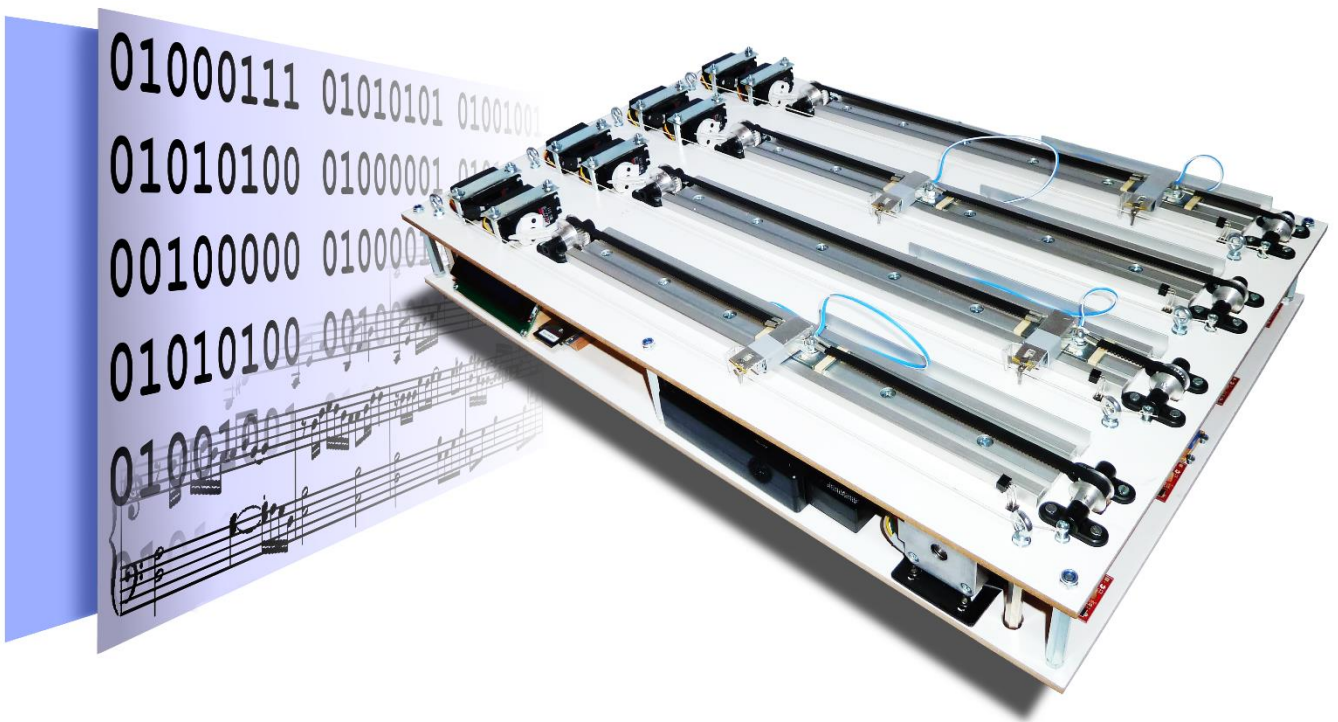


The GuitarBot

of Bits and Tones



Matura Paper, Kantonsschule Sargans

Author:	Frank Schaufelberger, 4bNPW
Supervisor:	Thomas Büsser
Submitted:	January 6, 2014

Management Summary

“GuitarBot” is an ambitious project to build a robot which is able to play music on four guitar strings. The main part consists of the practical work, the development of the robot. The present paper is a detailed documentation of how the GuitarBot turned from an idea into a working product. It describes the many hurdles encountered by the mechanical aspects of the GuitarBot, the various pitfalls of the electronic components and eventually the programming of the microcontroller.

The project was highly instructive and provided experience and insight into the world of engineering and robotics, sectors in which I can see myself in the future.

Table of Contents

1	Introduction	1
1.1	Aim of the Project	1
1.2	Motivation and Inspiration	1
1.3	Structure of the Paper.....	1
1.4	Terminology	2
2	Hardware	3
2.1	Morphology	3
2.2	Choice of Components	3
2.2.1	Fretting the String.....	3
2.2.2	Linear Motion	4
2.2.3	Picking the String	4
2.2.4	Layout	4
2.3	Prototyping.....	5
2.4	Final Production	6
3	Electronics	7
3.1	The Microcontroller	7
3.2	Electronic Components	7
3.2.1	Stepper Driver.....	8
3.2.2	Servo Controller	8
3.2.3	Relay Board	8
3.2.4	SD Card Reader	9
3.2.5	Sensor Dock	9
3.2.6	Shield	9
3.2.7	Other Parts.....	10
3.3	Power Supply.....	10
3.3.1	Overvoltage and Flyback Diode	10
4	Software	12
4.1	Programming the Arduino.....	12
4.1.1	Libraries	12
4.2	Controlling the Components	12
4.2.1	Stepper	13
4.2.2	Servos.....	13
4.2.3	Relays.....	14
4.2.4	Sensor	14
4.3	Program Ideas	14
4.4	Final Program	15
4.4.1	Calibration	16
4.4.2	Read a Line.....	16
4.4.3	Move Position	17
4.4.4	Play a Tone.....	18
4.4.5	Stop a Tone	18
5	Conclusion	19
5.1	What's next?	19
	References	IV
	List of Figures	V
	List of Abbreviations	VII

1 Introduction

1.1 Aim of the Project

The aim of the project “GuitarBot” is to build a machine, which is capable of playing music. An idea should be created and realized from scratch. The development process of a product should be experienced. The project is an opportunity to gain insight into the world of engineering, especially robotics.

1.2 Motivation and Inspiration

I have always been fascinated by the development of technology over time. Furthermore, music plays a crucial role in my everyday life. There’s also the fact that I’m extremely interested in engineering and product development. Given these conditions, the GuitarBot is the perfect project, combining my probable future profession area with a lifetime passion. The outcome should not only be a working product but also unique experience for life.

Of course, the GuitarBot is not the first project in the world to combine music and technology. “Compressorhead” is a group of humanoid robots which play real instruments. “MechBass” is a machine similar to the GuitarBot. Nevertheless, the ambition was to build a music robot from scratch, based on own ideas, not to copy an existing one.

1.3 Structure of the Paper

The whole document is structured like the development of the GuitarBot itself. After discussing some ideas, the final form of the robot emerges more and more. A prototype has to be built, suitable components have to be chosen. With the mechanics fully assembled comes the part where the electronics has to be installed. The last chapter consists of software based aspects.

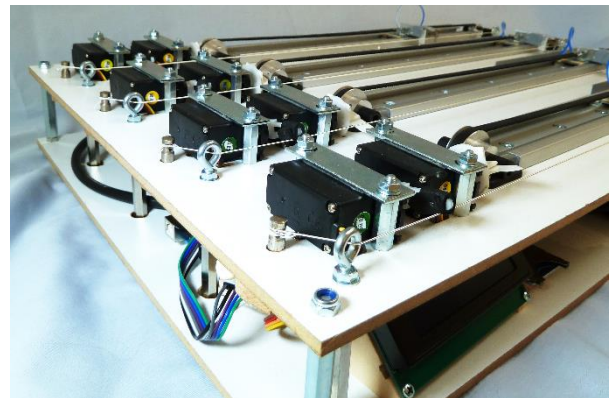


Figure 1: The GuitarBot with the servos in the foreground

The use of technical terms couldn’t be avoided.

Therefore, the next chapter explains a few important terms, also words I invented by myself to describe certain components. A more extensive list of expressions can be found in the appendix.

1.4 Terminology

Fretter The fretter is the part of the GuitarBot which moves along the linear slide. When toggled, a solenoid with a metal rod through its shaft pulls the string towards it. The housing of the fretter also acts as the fret itself (Figure 15).

Muter The muter of the GuitarBot is the servo with rubber foam at the end of its arm. When the muter lowers, the rubber foam mutes the string, and the tone is stopped (Figure 2).

Picker The picker of the GuitarBot is the servo with a pick (guitar plectrum) attached. Up and down movements of the servo cause the pick to play the string (Figure 2).

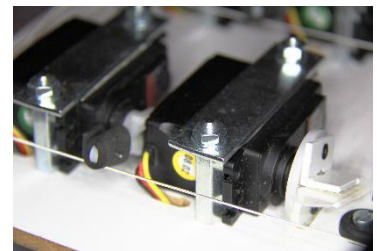


Figure 2: Two servos acting as a picker and a muter

Servo motor A servo motor (Figure 2), or just servo, is an actuator which allows precise positioning. However, a servo has a limited rotary range.

Solenoid In engineering, a solenoid is a device which converts current into linear force. There's a metal rod inside a tightly wound coil. Whenever current flows through the coil, a magnetic field is created and the metal rod is pulled inside the coil (Figure 5).

Stepper motor A stepper motor (Figure 6), or just stepper, is a motor which divides the rotation into small, consistent steps. By telling the stepper how many steps to take, a precise rotation can be achieved.

Timing belt A Timing belt (Figure 3) is a toothed belt which can be used to transfer a rotation from one axis onto another. In the GuitarBot, a timing belt is used to transform the rotation into a linear movement.



Figure 3: Timing belt with the joints made of cable connectors

2 Hardware

The following chapters deal with the development of the GuitarBot's hardware. Like humans, robots also have "intelligence" on one side, and a body on the other side. The important point is that they work together properly. Therefore, the constructor has to think ahead and adjust the form of the machine to its intended purpose. Or in other words: Form follows function.

2.1 Morphology

The function of the GuitarBot is to play guitar. If we take a look at the human body playing a guitar, we can clearly see two parts of the action. There's one hand plucking the strings and therefore deciding on the timing of a tone. The second hand moves along the fretboard and presses the strings down to change the pitch of a tone. This is also called "fretting" the strings.

There are two approaches for a guitar playing robot. One is to construct a machine which plays an actual guitar, like humans do. The other approach is to abstract the guitar and rebuild it as simple as possible. Since the real guitar was constructed to be played by a human, it's just logical to redesign the guitar so that it can be easily played by a machine. Based on the tasks mentioned above, the guitar can be reduced to the strings (Figure 4, red) and the frets (Figure 4, green).

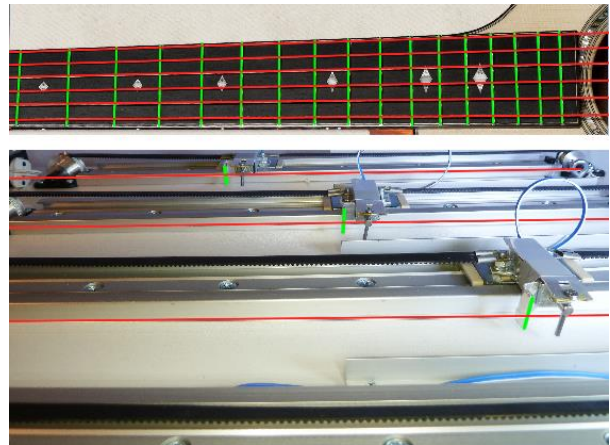


Figure 4: A real guitar compared to the GuitarBot with the strings and frets highlighted (picture above, source: taylor-guitars.com)

2.2 Choice of Components

An important step is the choice of appropriate components. They should have a good price-performance ratio.

2.2.1 Fretting the String

When playing a real guitar, the string is fretted by pressing it down between two frets. Now the string can only vibrate from the fret to the other end, which results in a higher tone. A fixed fret, which shuttles along the string is easy to build, but it produces disturbing noises and damages the string while moving back and forth. The solution is a solenoid which pulls the string towards itself when triggered. The solenoid is mounted in a self-made housing which also acts as the fret.



Figure 5: A solenoid is used to fret the string

2.2.2 Linear Motion

Somehow, the linear movement of the fretting hand has to be imitated by the GuitarBot. There are linear motors, but they are very expensive and unsuitable. So the rotation of a usual motor has to be transferred into a linear motion. The best solution appears to be a timing belt. Because ordinary DC motors can't be driven precisely (which is essential to position the fretter), the solution is either a servo motor or a stepper motor. The disadvantage of the servo: It can't rotate indefinitely. On the other side, stepper motors are



Figure 7: A segment of a linear slide

much more complex to control. Nevertheless, the stepper motor is chosen because the rotary limitation of the servo is a huge drawback.

To stay on a straight track, the fretter moves back and forth on a linear slide driven by the mentioned timing belt.



Figure 6: A stepper motor with a pulley

2.2.3 Picking the String

In order to achieve an efficient way of picking, the string has to be played in downstrokes (the pick makes a downward movement) and upstrokes (the pick makes an upward movement). The problem with solenoids is, that they only provide force in one direction, say in the downstroke direction. That means that the force for the upstroke has to be caused by another part, e.g. a spring, which would result in a greater effort in design. A satisfying solution seems to be a servo motor. Its limited range, in our case about 180°, isn't a problem here since the movement of the pick isn't a full rotation. Another servo with foam rubber at the end of its arm is attached to mute the tone.

There would also be the opportunity of attaching several picks to a stepper motor, as it is done at the MechBass. However, that would be a more expensive solution besides the mentioned complexity in controlling a stepper.

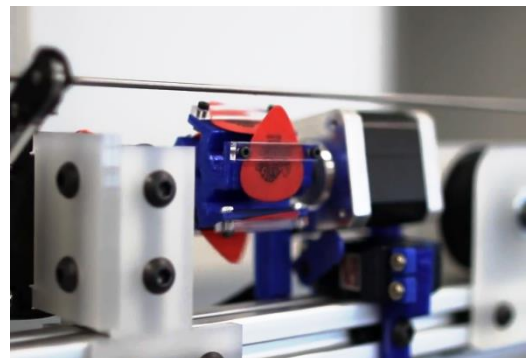


Figure 8: The picking mechanism at the MechBass (source: youtube.com)

2.2.4 Layout

Once the components are chosen, there are many different ways to arrange them. For most of the parts there's the question whether the string is located above or beside the whole mechanics. Above would mean that the individual entities get quite narrow but high. However, the option with the string next to the mechanics is easier to build since the construction is lower. One of the remaining issues,

which is worth mentioning, is the position of the stepper motor. Because it's easier to have the pulleys installed horizontally (the axis is horizontal), the stepper motor is also mounted horizontally. Still, there are two options for its location. Either the motor is on the same layer as the rest of the mechanics, or it's located on a second, lower level. A second level comes in handy when we get to the electronics and its installation. Here's where thinking ahead pays off!

Another aspect of thinking ahead is that the dimensions of the GuitarBot are chosen so that the whole machine can be put in a standard-box (width: 36cm, length: 56cm).



Figure 9: The GuitarBot in a box

2.3 Prototyping

A prototype is usually built to test whether the concepts made on paper work in reality. The way to the final product is an iteration of improvements and new prototypes. The very first GuitarBot prototype was built even before the majority of the ideas above have been developed. It only consisted of

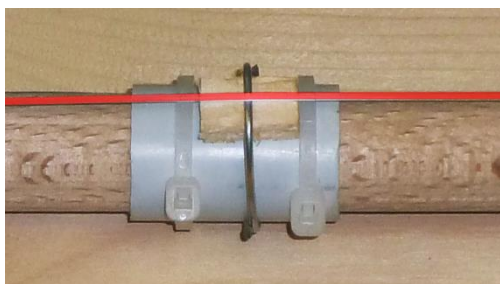


Figure 10: The first prototype's linear slide with the string highlighted in red

a primitive linear slide (a part of a PVC tube around a wooden rod), a fixed fret, and a string tensioned over the construction. Of course the prototype wasn't even able to play a tone because there weren't any motors or other electronics attached. It was just useful to have an image of the whole concept and it was easier to get to the later ideas.

The second, more serious GuitarBot prototype had almost all the components that the final GuitarBot has as well. It consisted of only one string and the according mechanics. It revealed that a limit switch is necessary to calibrate the stepper. The infrared sensor prevailed against a mechanical switch, because it is non-contact and works well at short range. An additional pulley with a spring (to tension the timing belt) turned out to

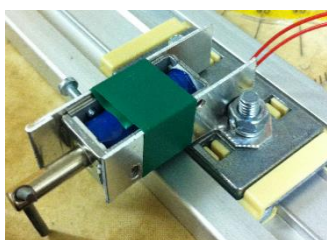


Figure 12: The first fretter

be unnecessary, if the timing belt has the perfect length. Of course the self-designed fretter also needed a revision and got, among other things, a guide for the small rod which frets the string. There were also other, less significant changes to the final GuitarBot which are not mentioned here.

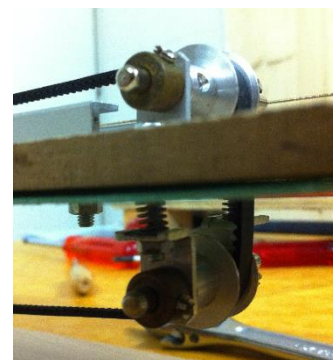


Figure 11: The additional pulley of the second prototype

2.4 Final Production

In order to assemble the final robot as trouble-free as possible, some preparation is to be done. All the components are digitally drawn using Microsoft Visio. Now, the objects can be moved around on the sketch to get to an optimal layout. An important part of the layout are the boreholes. Once finished, the layout is printed at scale 1:1. With this blueprint, the boreholes can be mapped onto the board and are drilled in no time.

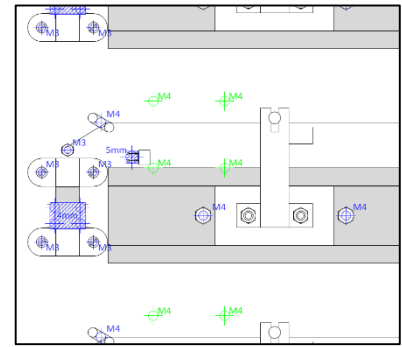


Figure 13: A sector of the layout (the whole layout is in the appendix)

The Handcraft Challenge

It seems that the rest is just putting together the parts. On one side it's true. On the other side, it's not that easy since the GuitarBot doesn't base on an assembly kit such as LEGO Technics or MakerBeam. Some parts are even self-made, such as the fretter housing (Figure 14, Figure 15). It's an elaborate object which isn't made using a CNC milling machine, but with tinsnips and files. A lot of other tools are used, e.g. a plunge saw, an awl, a rasp, or a soldering iron.



Figure 14: An aluminum sheet cut and bent...

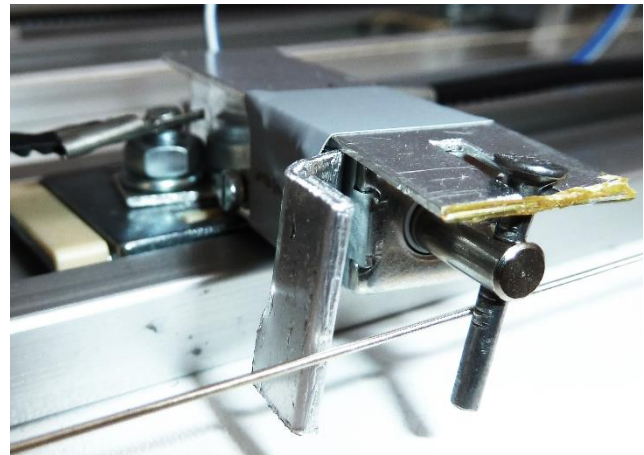


Figure 15: ...turns into the housing of the fretter

3 Electronics

The Electronics is the part of the robot which controls the body. It's the thing between hardware and software which makes the two of them work together. Of course all the motors and sensors mentioned above are also electronic parts, but in this chapter we will talk about the parts that are controlling the others.

3.1 The Microcontroller

As already said, a robot needs a brain, a part on which the software is running and which controls the other electronic components. There's a microcontroller in almost every device nowadays. For the GuitarBot, the microcontroller has to be effective but also easy to handle.

After extensive research and a conversation with Prof. Reto Bonderer (Embedded Software Engineering at the HSR), the microcontroller board "Arduino" turns out to be suitable for the GuitarBot. An Arduino consists of a microcontroller with an open-source hardware board built around it. Most of the microcon-

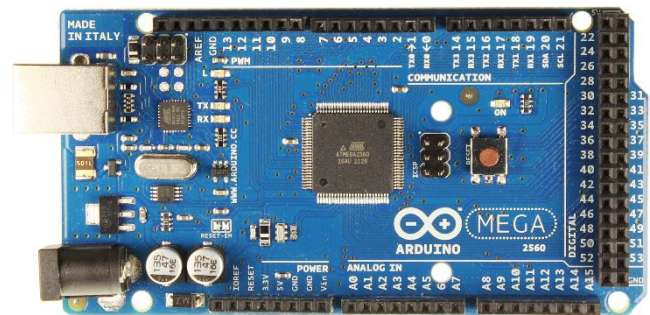


Figure 16: Arduino Mega (source: arduino.cc)

troller's pins are exposed on the Arduino, which means connecting it to other components is very easy. The pins are divided into digital and analog pins. Digital pins, whether input or output, can only distinguish between on and off. Analog pins, on the other hand, have a resolution of 10 bits, which means they can have 1024 different values. What that means exactly and how we can use it should become clear when we come to programming the Arduino (chapter 4.1). Probably the best thing about Arduino is, that it's completely open source. Meaning that all the software and plans for the modules are published under a free license. As a result, there is a huge amount of additional boards available, some of which will be mentioned in the following chapters.

3.2 Electronic Components

Besides the Arduino, there are a lot of other electronic boards inside the GuitarBot. The Arduino can be seen as the highest controlling entity. It controls the other boards, which control specific components themselves.

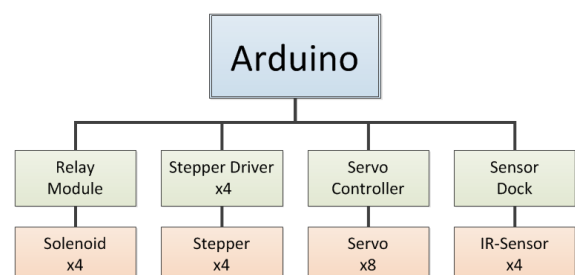


Figure 17: Electronic hierarchy

3.2.1 Stepper Driver

Stepper drivers are designed to undertake the complex process of controlling a stepper motor, while the driver itself can be easily controlled. There are masses of different drivers, all with their pros and

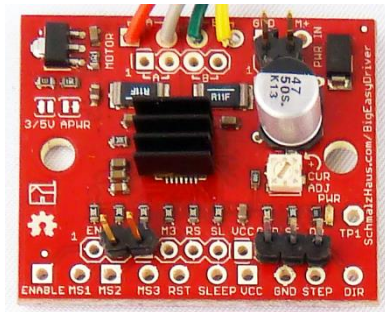


Figure 18: Big EasyDriver by Sparkfun

cons. The “Big EasyDriver” by Sparkfun is a suitable driver for the GuitarBot’s stepper motors. It is controlled over only two wires, “Step” and “Direction”. As expected by the name, the driver causes a step every time it receives a signal on “Step”. “Direction” is used to set the direction in which the stepper motor is supposed to turn. Therefore, the four stepper motors can be controlled over only eight digital pins on the Arduino.

3.2.2 Servo Controller

When all the eight servos were controlled individually, they would occupy eight pins on the Arduino. With the “16-Channel PWM/Servo Driver” by Adafruit, we only need two pins to control 16 servos. The driver uses the I²C bus, which transfers data serially over two special pins on the Arduino. The servos can now be controlled by just sending commands to the servo driver. Since every device which uses the I²C bus has its own address, we could control multiple boards with the same two pins. Therefore, we could chain up 62 of these servo drivers to control up to 992 servos at a time with only two pins (Earl, 2013).

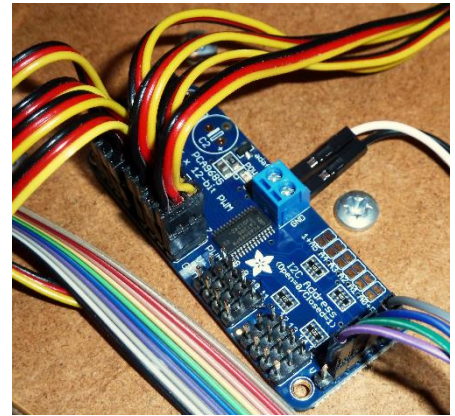


Figure 19: The servo controller already hooked up to the GuitarBot

3.2.3 Relay Board

The solenoids require 12 volts, whereas the Arduino can only provide 5 volts. Hence we need a separate power supply for the solenoids. The Arduino still has to be able to control the solenoids (i.e. turn them on and off). A relay is an electrical switch, toggled by a low-power signal. Because it’s isolated, it can be used to turn a high-power circuit on and off. The “4-Channel Relay Module” by Sainsmart works well with Arduino and is a suitable choice.

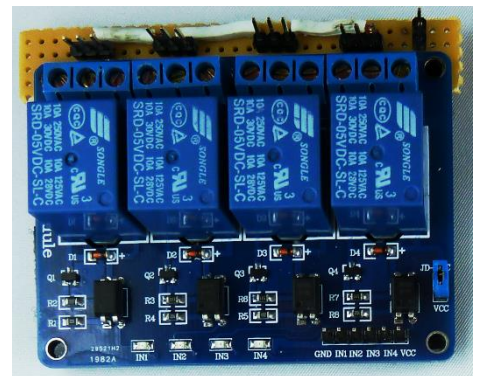


Figure 20: Relay module with additional pins

3.2.4 SD Card Reader

In most cases, an Arduino uses an SD Card to log data, which it has collected by reading from sensors. Why the GuitarBot needs an SD Card, will be explained in chapter 4.3. There are several different card readers available to work with the Arduino. The one used in the GuitarBot communicates with the Arduino over the SPI bus, a serial bus similar to the I²C bus.

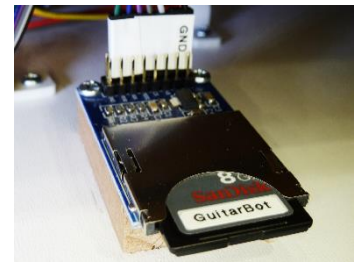


Figure 21: SD card reader

3.2.5 Sensor Dock

The self-made sensor dock isn't a "real electronic" board, it is just a board where some pins are merged. The infrared sensor contains two separate parts. An infrared LED, and a phototransistor. Therefore, it has four pins. One for the 5V supply, one to read the sensor's value, and two ground pins (Figure 22, above). Since four of these sensors are used, there are 16 wires. To reduce the number of wires going to the Arduino, all the ground pins and the 5V pins are merged (Figure 22, below). Now there are 16 wires going onto the sensor dock, and just 6 coming out of it. Because the sensor can't just be hooked up to the Arduino and the power supply (ameyer, 2011), the respective circuit has to be quadrupled and soldered onto the board.

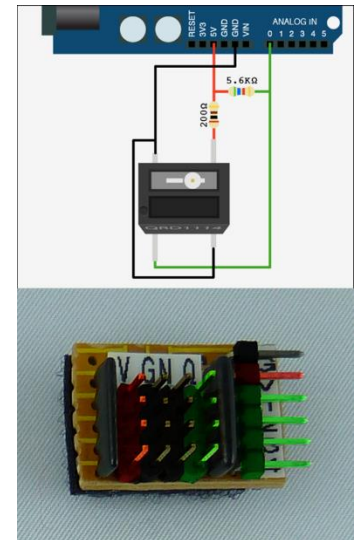


Figure 22: Same pins highlighted in the same color (picture above, source: bildr.org)

3.2.6 Shield

Talking about Arduino, a shield is an electronic board with the pins arranged like the pins of the Arduino. Thus, a shield can just be plugged on top of the Arduino.

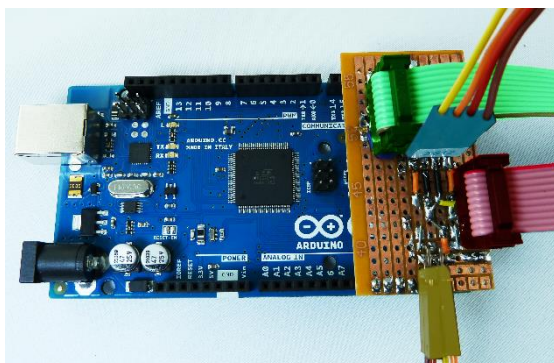


Figure 23: The shield plugged into the Arduino with the components connected and highlighted

parts attached, one can imagine that there are a lot of wires to be connected to the Arduino. With the self-made shield, the Arduino can be easily disconnected from the GuitarBot. There are several connectors on the shield where the wires can be plugged into. From there, they are connected to their according pins. The shield is also the place where the power from the supply (highlighted in yellow) is distributed to the specific parts (highlighted in blue).

3.2.7 Other Parts

The stepper driver doesn't only have the two mentioned pins "Step" and "Direction". In fact, there are seven wires coming from each stepper driver, i.e. 28 altogether (Figure 24, highlighted in red). To minimize the number of wires going to the Arduino (Figure 23, highlighted in red), a motor dock is constructed where the wires are plugged in. Power supply pins are merged, and pins which can be used to set the stepping type are connected to a switch on the board. Thereby, the number of wires going to the Arduino (Figure 24, highlighted in green) can be reduced from 28 to 10.

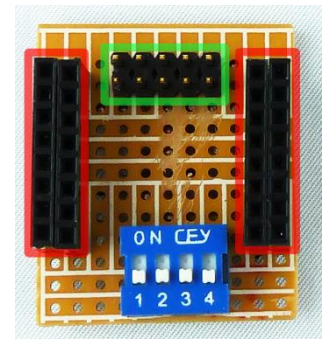


Figure 24: Inputs highlighted in red, outputs in green

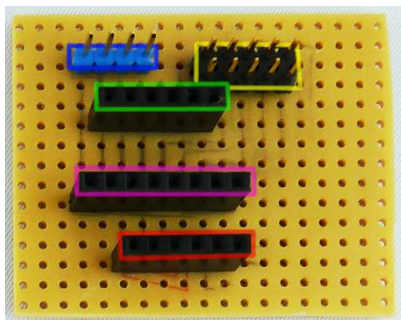


Figure 25: Inputs for the sensor dock (red), relay module (pink), and servo controller (green) and output to the Arduino (yellow)

Another dock is located near the relay board, the servo driver, and the sensor dock. There, the wires of these boards meet. Again, pins for the power supply are merged (Figure 25, highlighted in blue) and the wires going to the Arduino (Figure 23, highlighted in green) are reduced from 20 to 14.

On some photographs, one can also see a liquid-crystal display and a joystick. These parts were used for test programs but are not required for the final program.

3.3 Power Supply

To supply the GuitarBot with power, a power adapter which provides 5 and 12 volts is used. The 5V and 12V wire, as well as the ground wires, go onto the shield where the power is distributed.

3.3.1 Overvoltage and Flyback Diode

Switching off the solenoids cause an interference which severely disturbs the other components and crashes the Arduino. This interference can be reduced using a "flyback diode". This measure protects the Arduino. Unfortunately, there is still an issue with the solenoids. Whenever a solenoid is turned off, the stepper motors move one step, because they are connected to the same 12V supply. Maybe, the interference could be removed by another electronic component (e.g. an RC-filter, a resistor-capacitor circuit). But the easiest solution is to use an individual 12V power supply for the solenoids. Herewith, the occurred problems with the solenoids are solved.

The reason why the flyback diode works is rather tricky and not important throughout the rest of the paper. Nevertheless, here's a short explanation:

Turning off the power supply of an inductor isn't as harmless as it might seem. When an inductor (like a solenoid) is provided with voltage, a magnetic field is created. According to physics, moving electrons create a magnetic field. Alternatively, a magnetic field changing its strength or polarity causes electrons to move. Let us apply that to the solenoid. If the solenoid is supplied with voltage, a magnetic field is created. If we now, all of a sudden, turn off the voltage, the magnetic field collapses. The solenoid now acts as a generator, giving the electrons some extra push. A voltage peak occurs. Although the power supply for the solenoid is just 12 volts, this voltage peak can be hundreds of volts. The quicker the shut-off, the greater the peak. This

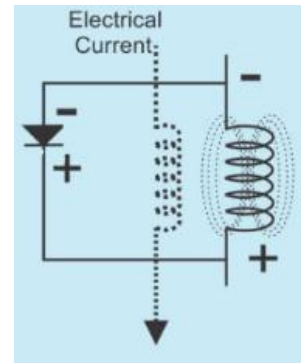


Figure 26: Electron flux when the solenoid is provided with current (source: douglaskrantz.com)

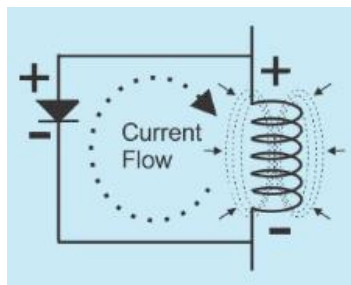


Figure 27: Electron flux shortly after turning off the power supply (source: douglaskrantz.com)

peak causes the Arduino to crash and reboot.

A so called flyback diode seems to be the solution to the problem. A regular diode is attached to the circuit, so that it's not conducting when the solenoid is turned on (Figure 26). As soon as the power supply is turned off, the flyback diode shunts the voltage back into the solenoid (Figure 27). As a result, the magnetic field collapses much slower and the generated voltage will be lower (Krantz, n.d.).

4 Software

The only thing missing now, is the software. In this chapter, we will get to know the program which is running on the GuitarBot. The goal is to figure out what the program does, without knowing the specific coding language.

4.1 Programming the Arduino

The Arduino provides its own IDE (integrated development environment). The code, in which the Arduino is programmed is simplified C++. But it's not as complicated as it sounds. There are a lot of libraries that make coding easier. A program on the Arduino needs two essential parts. There is the "setup()" -part, which is executed once the program starts, and there's the "loop()" -part, which is looped all the time until the program is interrupted (Figure 28). These two parts are important for the under-

```
void setup() {  
  
    pinMode(33, OUTPUT);    // pin 33 -> output  
}  
  
void loop() {  
  
    digitalWrite(33, LOW);  // turn on relay  
    delay(500);             // wait .5 sec  
    digitalWrite(33, HIGH); // turn off relay  
    delay(500);             // wait .5 sec  
}
```

Figure 28: A simple Arduino program which turns a relay connected to pin 33 on and off

standing of the program itself. We already know that the Arduino has two kinds of pins. Digital and analog pins. Every pin can be used as an input or an output. This means that we can either "read" from a pin or "write" to a pin. If we read from a digital pin using "digitalRead()", the returned value is either "HIGH" or "LOW". Accordingly, when we write to a digital pin using "digitalWrite()", it provides 5V if we write a "HIGH" value, or 0V if we write a "LOW" value. The analog pins have a resolution of 10-bit, which means if we read from an analog pin using "analogRead()", the input between 0V and 5V is mapped to a value from 0 to 1023. If we write to an analog pin using "analogWrite()", it can simulate voltages between 0V and 5V, using pulse-width modulation (PWM, which we won't discuss here any further).

4.1.1 Libraries

As already mentioned, there are a lot of libraries available for everything one could imagine. If we take the example of the servo driver, which uses the I²C-Bus, it would require a lot of complicated code to send a single command to the driver. With the corresponding library "Adafruit_PWM_ServoDriver", moving a servo to a certain position can be achieved in one line of code.

4.2 Controlling the Components

Already during the process of prototyping, first programs were written to test the components and to get used to how to control them.

4.2.1 Stepper

To control the stepper motors, the library “AccelStepper” by Mike McCauley is used. The library is compatible with most drivers and has some great features. AccelStepper can not only move multiple stepper motors at a constant speed, but it can also implement accelerations. Furthermore, it has a position

```
void setup()
{
  stepper1.setMaxSpeed(maxSpeed); // maximum speed set to 1000 steps per second
  stepper1.moveTo(targetPos);      // move to position 300 (300 steps from 0)
  stepper1.setSpeed(stepSpeed);     // set speed to "stepSpeed"
}

void loop()
{
  // if at target position, wait half a second and go to -(target position)
  if (stepper1.distanceToGo() == 0){
    stepper1.moveTo(-stepper1.currentPosition());
    stepper1.setSpeed(stepSpeed);
    delay(500);
  }
  stepper1.runSpeedToPosition(); // make steps at defined speed
}
```

Figure 29: Example of driving a stepper using AccelStepper

tracker to know at which “position” the stepper is. An essential feature of the AccelStepper library is, that it’s non-blocking. When we tell the stepper motor to move to a certain position, the program doesn’t wait until the position is reached. To move the stepper, we just have to update the target position of the stepper object. Every time we call “stepper.runSpeedToPosition()”, the motor moves one step, if a step is due. In order to obtain a fluent rotation, “stepper.runSpeedToPosition()” must be called as frequently as possible (Figure 29).

4.2.2 Servos

The servo driver uses its own library called “Adafruit_PWMServoDriver”. We create a driver object, on which we can execute the commands to control the servos. The driver has 16 ports which can be controlled over the same object. A servo motor expects a PWM signal. Depending on the pulse width, the servo then moves to the corresponding position. In other words, we can’t tell the servo to “move to position 90°”, but we have to send a “HIGH” signal for 1.6ms (pulse width) instead.

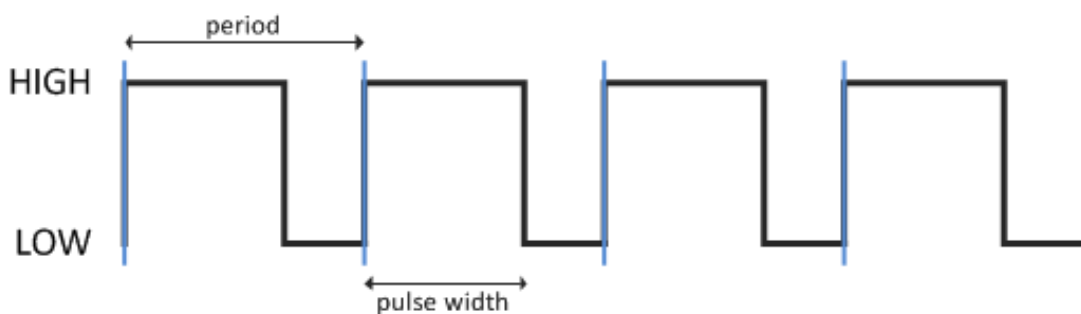


Figure 30: Concept of pulse-width modulation

Analog servos run at about 50 Hz updates, which means the PWM period measures 20 milliseconds (Salt, n.d.). With the mentioned library, a servo’s position is set with “pwm.setPWM(int port, int on, int off)”, where “port” is the port on the driver to be updated. “On” (a value between 0 and 4095), is the time in the period when the signal is turned on. “Off” (a value between 0 and 4095), is the time in the period when the signal is turned off again. The time during which the signal is “HIGH” is the pulse width (Figure 30).

Our servos range from a pulse width of approximately 0.6ms (on – 0, off – 150), which is all the way to one side, to 2.6ms (on – 0, off – 630), which is all the way to the other side. This results in a rotation range of about 180°.

4.2.3 Relays

Controlling the relay board is really simple. Each of the four relays has a separate pin which is connected to a digital pin on the Arduino. Now if for example “digitalWrite(33, LOW)” is executed, the relay connected to pin 33 is turned on. Correspondingly, if “digitalWrite(33, HIGH)” is executed, the relay on pin 33 is turned off (Figure 28).

4.2.4 Sensor

With the function “analogRead(int port)”, we can easily get the sensor’s value (Figure 31). A value between 0 and 1023 is returned. The closer the object is to the sensor, the lower the value. Because the sensor is only suitable at short range, there is a remarkable drop in the returned value even when the object gets 1mm closer to the sensor. That’s very convenient, because the stepper can be calibrated highly accurate. Under normal circumstances the sensor returns a value below 300, when the fretter is at the end of the linear slide. Therefore, 300 is an appropriate limit value. Usually, an infrared sensor isn’t influenced by visible light. Nevertheless, extreme brightness can cause minor changes in the returned values.

```
void loop() {  
  
    val = analogRead(A3);  
  
}
```

Figure 31: Reading the value from the sensor on port "A3" and storing it to "val"

4.3 Program Ideas

The whole concept of the program is to play a preset (i.e. hard coded) song with the GuitarBot. First thoughts revealed that the program is just some functions put together, namely “move”, “play”, and “stop”. Each of the four strings should be controlled individually.

Unfortunately, the Arduino is not multi-threading, i.e. it can’t run multiple functions simultaneously. There’s no problem with that until it comes to moving the stepper motors. A lot of stepper libraries have blocking functions, which is unsuitable when two steppers should run at the same time. Luckily, the AccelStepper library is non-blocking as mentioned above.

The unsatisfactory thing about the concept of a preset song is, that a program can only play that exact song. If another song wants to be played, a new program has to be written. So the song should not be hard coded. The solution is to store the commands of the song in a CSV-file and write a program which executes these commands one after another.

One command consists of three or four parts:

Time; String; Command; Steps

For example:

```
2000; 2; m; 370    // move the fretter on the 3rd string to position 370

2500; 2; p;        // after 500 milliseconds, play a tone on the 3rd string

2980; 2; s;        // stop the tone on the 3rd string after 480 milliseconds
```

The first part is the time (in milliseconds) at which the command has to be executed. Followed by the string (number from 0 to 3) on which the command has to be executed. Then comes the command itself (“m”, “p”, or “s”), i.e. “move”, “play” or “stop”. If the command is “m”, there’s another part containing the target position.

Another option is to store these values in four different arrays. However, arrays can get quite inconvenient, since they have to be edited in the program itself. Also, arrays with a lot of values get very unclear. In comparison, values in a CSV file can be easily created and edited using Excel, and every command has its own line (as indicated above). The CSV file is saved on an SD card which is later read by the Arduino. The downside of this option is that reading out a file from an SD card with the Arduino is not as simple as reading values from an array. But after all, the advantages outweigh the disadvantages.

4.4 Final Program

First, some preferences are set. Limit values for the sensors, positions for the servos, and the speed of the stepper motors, just to mention a few. In the setup of the program, all the servos are moved to their start position, the stepper motors are calibrated, and the file “guitarbot.csv” on the SD card is opened. Before the loop begins, the first line of the CSV file is read and the values of the line are stored into individual variables. The loop itself consists of a very simple procedure. If the timer reaches the

value of “myTime”, the according command is executed and a new line is read. Whether or not a command was executed, “runSpeedToPosition” is called 1000 times for each stepper motor, in order to achieve a fluent movement.

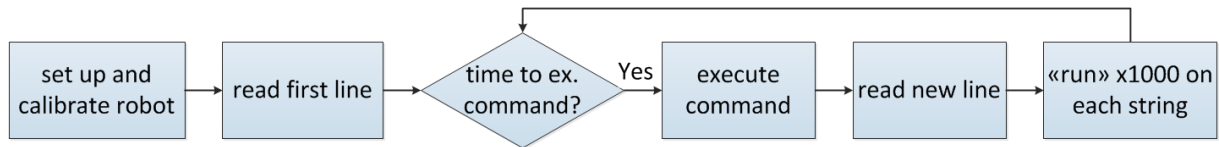


Figure 32: Flowchart of the program running on the GuitarBot

4.4.1 Calibration

When the GuitarBot is turned on, the fretters can be at any position on the linear slide. Because the Arduino itself doesn’t know where they are, the stepper motors have to be calibrated. This means that every fretter moves toward the beginning of its linear slide, until the IR-sensor reports that the end is reached (Figure 33). The position tracker is then set to zero. From now on, the Arduino always knows

```

/* ##### CALIBRATE ##### */
// move the fretter close to the sensor and set this as 0 position
void calibrate(int _string) {
    stepper[_string].setSpeed(vStepper); // set speed for the stepper
    irValue[_string] = analogRead(IR[_string]); // put the sensor value into the array

    while(irValue[_string] > irLimit[_string]){ // run till the beginning of the slide is reached
        irValue[_string] = analogRead(IR[_string]); // read from the sensor all the time
        stepper[_string].runSpeed(); // run towards the beginning of the slide
    }
    stepper[_string].stop(); // stop if reached
    stepper[_string].setCurrentPosition(0); // set current position as 0
}

```

Figure 33: Calibration of the stepper

where the fretter is situated, thanks to the position tracker of the Accel-Stepper library (except if the timing belt slips, i.e. the stepper makes a step but the fretter doesn’t move, which fortunately doesn’t happen a lot).

4.4.2 Read a Line

In order to understand the reading of a line, some things have to be clarified. First, the SD card library only reads one character at a time when reading from a file. Furthermore, the CSV file is plain ASCII text. The structure of the CSV file is also important (see chapter 4.3). For a clearer understanding, there’s a flowchart at the end of this chapter (Figure 34).

To get the first part of the line (the time), every character until the first semicolon is stored to “myTime”, using the following procedure:

$$myTime = myTime \times 10 + (c - 48)$$

“c” is the currently read character. “0” is the 48th character in the ASCII table. Therefore, “0” converted to an integer would be “48” (hence “c - 48”). Since the numbers are treated as single characters, the previous value of “myTime” has to be multiplied by 10 before the new number is added. As an example: The number is “23”. The first number stored to “myTime” is “2”. The second number to be stored is “3”. If we multiply “myTime” by 10 before the addition, we get to the desired value of “23”.

After reading the semicolon, the second part of the line (a number) is stored to the variable “myString”. This number indicates on which string the command has to be executed. “0” is the first string, “3” is the fourth string. Here, too, we have to subtract 48 from the character in order to get the right value.

The next character should be a semicolon again, followed by the command. There are three options: “m” for move, “p” for play, or “s” for stop. Since the data-type of the variable “myCommand” is “character”, our command can be stored as it is (no subtracting, like with the numbers).

If the command is “m”, another number (the new target position) is expected after the semicolon. The number is stored to “mySteps” with the same procedure like the time has been stored to “myTime”. The very last characters on each line are “\r” – carriage return, and “\n” – line feed (both not visible when viewing the file in an editor), which indicate the end of the line, and therefore the end of the function “readLine()”.

We now have the individual values from the line in the variables “myTime”, “myString”, “myCommand”, and “mySteps”.

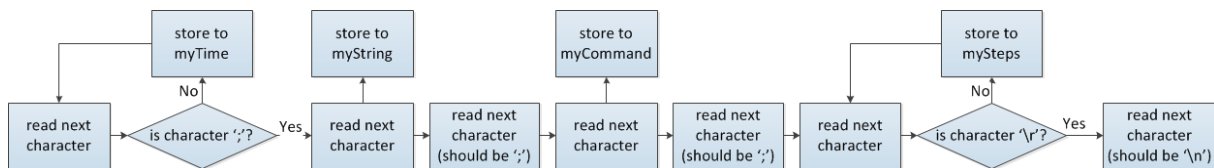


Figure 34: Simplified flowchart to the function “readLine()”

4.4.3 Move Position

The function “movePosition(int _string, int _position)” requests two arguments. The first one is the string on which the function is applied to, and the second one is the position to where the fretter has to be moved. If the fretter is not

```

/****** MOVE *****/
// change the target position of the stepper to the position in the argument
void movePosition(int _string, int _pos)
{
    if(!stringReady[_string]) { // force stop, if not ready to move
        stopTone(_string);
    }
    if(abs(_pos) > abs(maxSteps)) { // if _pos isn't within range, cancel the method
        return;
    }
    stepper[_string].moveTo(_pos); // set target position
    stepper[_string].setSpeed(vStepper); // set speed (has to be set after moveTo());
}

```

Figure 35: The function executed when “myCommand” is “m”

ready to move (a tone is played on that string), a forced “stopTone(int _string)” is called. When the fretter is ready to move and “_pos” is within a certain limit (maxSteps), the target position of the according stepper motor is updated to the new position. The actual movements of the stepper motors are caused by the call of “stepper[i].runSpeedToPosition()”, for i from 0 to 3, at the end of each loop (Figure 32).

4.4.4 Play a Tone

One of the most important functions is “playTone(int _string)”. The number delivered in the argument tells the function, which string has to be plucked. We get the number of the servo-ports on the controller by the following two functions:

$$_PICKER = _string \times 2$$

$$_MUTER = _string \times 2 + 1$$

This means that for string number 1 (the 2nd string), the picker is on port 2 and the muter is on port 3. After the fretter is triggered, the muter is lifted. If the picker is below the string, an upstroke is implemented, if the picker is above the string, the function causes a downstroke (Figure 36).

```

/****** PLAY *****/
// play the tone on the string in the argument
void playTone(int _string)
{
    int _PICKER = _string * 2;    // get the picker port number from the string number
    int _MUTER = _string * 2 + 1; // get the muter port number from the string number

    digitalWrite(RELAY[_string], LOW);    // trigger the fretter
    stringReady[_string] = 0;             // string is not ready to move position
    pwm.setPWM(_MUTER, 0, servoUp[_MUTER]); // lift the muter

    if(servoPos[_PICKER])                // if picker is in position "down"
    {
        pwm.setPWM(_PICKER, 0, servoUp[_PICKER]); // upstroke
        servoPos[_PICKER] = 0;                  // update servo position
    }
    else                                  // if picker is in position "up"
    {
        pwm.setPWM(_PICKER, 0, servoDown[_PICKER]); // downstroke
        servoPos[_PICKER] = 1;                    // update servo position
    }
}

```

Figure 36: Code snippet showing the function "playTone(int _string)"

4.4.5 Stop a Tone

Stopping a tone is the easiest of all functions. When the function is called with the string number in the argument, the muter lowers and the fretter is released. The fretter is now ready to change position again (Figure 37). “stopTone(int _string)” uses the same function to get to the muter-port from the string number as “playTone(int _string)”.

```

/****** STOP *****/
// stop the tone on the string in the argument
void stopTone(int _string)
{
    int _MUTER = _string * 2 + 1; // get the muter port number from the string number

    pwm.setPWM(_MUTER, 0, servoDown[_MUTER]); // lower the muter
    digitalWrite(RELAY[_string], HIGH);      // release the fretter
    stringReady[_string] = 1;                 // string ready to move position
}

```

Figure 37: "stopTone(_string)", used to mute a played tone

5 Conclusion

The GuitarBot was a great project because it gave an insight into the procedure of developing a product. It's impressive how labor-intensive and time-consuming it is to realize an idea from scratch. The production of an object runs through a lot of phases and although the way to the final product seems obvious in the end, there's a lot of try and error involved in the origina-

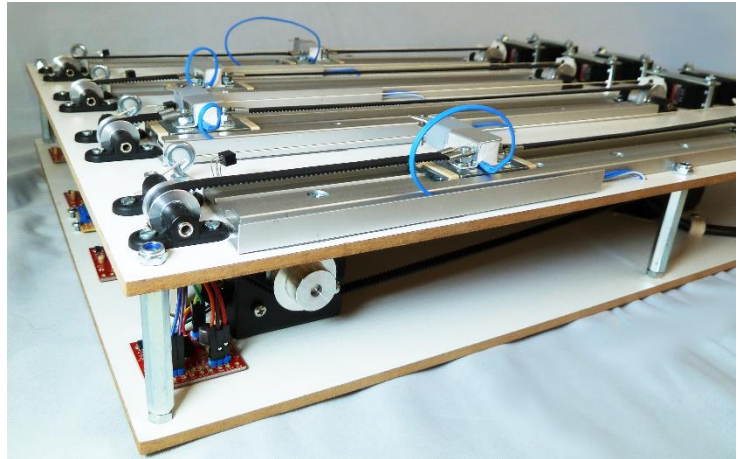


Figure 38: The GuitarBot from another perspective

tion process. Furthermore, a lot of issues are encountered during the production which weren't taken into consideration in the planning (e.g. the interference of the solenoids, see chapter 3.3.1). Of course the creation of a machine in general is only possible if certain resources are available. For example various components to choose from (see chapter 2.2), or specific tools to assemble the mechanics (see chapter 2.4), just to mention a few.

Although the GuitarBot was a lot of work, it was highly instructive and a lot of experience could be gained.

5.1 What's next?

The GuitarBot in its present state is working as intended. But of course, it can be improved further. There are several things that can be added to extend the GuitarBot even more. For example a program which converts a MIDI file into a GuitarBot-compatible CSV file, or the amplification of the tone by a special pickup system. But that would certainly go beyond the scope of this thesis.

References

ameyer. (2011, March 8). *bldr » Are we getting close? Proximity Sensors + Arduino*. Retrieved from bldr.org: www.bldr.org/2011/03/various-proximity-sensors-arduino/

Arduino LLC. (2012). *Arduino Projects Book*. Turin.

Arduino LLC. (n.d.). *Arduino - Reference*. Retrieved from arduino.cc: www.arduino.cc/en/Reference/HomePage

Bonderer, R. (2013, June 14). Diskussion über Robotikplattformen. (F. Schaufelberger, Interviewer)

Earl, B. (2013, June 20). *Overview | Adafruit 16-Channel Servo Driver with Arduino | Adafruit Learning System*. Retrieved from learn.adafruit.com: learn.adafruit.com/16-channel-pwm-servo-driver/overview

Krantz, D. (n.d.). *Flyback Diode*. Retrieved from douglaskrantz.com: www.douglaskrantz.com/Flyback_Diode.html

McVay, J. (2012, November 21). *MechBass - Hysteria - YouTube*. Retrieved from youtube.com: www.youtube.com/watch?v=5UYMnzXQEtw

Roberts, D. (2011). *Making Things Move*. Sebastopol: O'Reilly.

Salt, J. (n.d.). *Understanding RC Servos – Digital, Analog, Coreless, Brushless*. Retrieved from rchelicopterfun.com: www.rchelicopterfun.com/rc-servos.html

Taylor Guitars. (n.d.). *510ce | Taylor Guitars*. Retrieved from taylorguitars.com: www.taylorguitars.com/guitars/acoustic/510ce

List of Figures

Figure 1: The GuitarBot with the servos in the foreground	1
Figure 2: Two servos acting as a picker and a muter	2
Figure 3: Timing belt with the joints made of cable connectors.....	2
Figure 4: A real guitar compared to the GuitarBot with the strings and frets highlighted (picture above, source: taylorguitars.com)	3
Figure 5: A solenoid is used to fret the string	3
Figure 6: A stepper motor with a pulley	4
Figure 7: A segment of a linear slide	4
Figure 8: The picking mechanism at the MechBass (source: youtube.com).....	4
Figure 9: The GuitarBot in a box	5
Figure 10: The first prototype's linear slide with the string highlighted in red	5
Figure 11: The additional pulley of the second prototype.....	5
Figure 12: The first fretter	5
Figure 13: A sector of the layout (the whole layout is in the appendix).....	6
Figure 14: An aluminum sheet cut and bent.....	6
Figure 15: ...turns into the housing of the fretter.....	6
Figure 16: Arduino Mega (source: arduino.cc).....	7
Figure 17: Electronic hierarchy.....	7
Figure 18: Big EasyDriver by Sparkfun.....	8
Figure 19: The servo controller already hooked up to the GuitarBot.....	8
Figure 20: Relay module with additional pins.....	8
Figure 21: SD card reader	9
Figure 22: Same pins highlighted in the same color (picture above, source: bildr.org)	9
Figure 23: The shield plugged into the Arduino with the components connected and highlighted	9
Figure 24: Inputs highlighted in red, outputs in green.....	10
Figure 25: Inputs for the sensor dock (red), relay module (pink), and servo controller (green) and output to the Arduino (yellow)	10
Figure 26: Electron flux when the solenoid is provided with current (source: douglaskrantz.com)	11
Figure 27: Electron flux shortly after turning off the power supply (source: douglaskrantz.com).....	11
Figure 28: A simple Arduino program which turns a relay connected to pin 33 on and off.....	12
Figure 29: Example of driving a stepper using AccelStepper	13
Figure 30: Concept of pulse-width modulation	13
Figure 31: Reading the value from the sensor on port "A3" and storing it to "val"	14
Figure 32: Flowchart of the program running on the GuitarBot.....	16

Figure 33: Calibration of the stepper	16
Figure 34: Simplified flowchart to the function "readLine()"	17
Figure 35: The function executed when "myCommand" is "m"	17
Figure 36: Code snippet showing the function "playTone(int _string)"	18
Figure 37: "stopTone(_string)", used to mute a played tone	18
Figure 38: The GuitarBot from another perspective.....	19

Unless specified otherwise, the figures were self-created.

List of Abbreviations

ASCII	American Standard Code for Information Interchange
CNC	Computer Numerical Control
CSV	Comma-Separated Values
DC	Direct Current
HSR	Hochschule für Technik Rapperswil
I ² C	Integrated Circuit
IDE	Integrated Development Environment
LED	Light Emitting Diode
MIDI	Musical Instrument Digital Interface
PWM	Pulse-Width Modulation
SD	Secure Digital
SPI	Serial Peripheral Interface

Appendix

Terminology	A1
Program Code	A4
Layout	A9
Electrical Scheme	A10
List of Materials	A11

Terminology

Arduino	Arduino (Figure 16) is an open source microcontroller board. The pins are exposed so that it can be easily connected to other things. The program running on it is written in simplified C++.
Array	An array is a list of values. A variable only holds one value, whereas multiple values can be stored in an array.
Blocking	In terms of programming, a command can be blocking. This means that the general flow of a program is interrupted, until the command is fully executed. For example the rotation of a stepper motor can be a blocking command.
C++	C++ is one of the most popular coding languages. It is often used for embedded systems because it's rather low-level, which can be understood as "close to the hardware".
CSV	"Comma-Separated Values" is a file format, which allows to store tabular data as plain text. Like in a table, there are rows and columns. Each line represents one row. The cells are separated by commas. A sample line of a CSV file can be seen in chapter 4.3. In the GuitarBot's CSV files, the values are separated by semicolons. Since the separator character doesn't have to be a comma, CSV is also called "Character-Separated Values".
Diode	A diode is an electronic component which lets current only flow in one direction.
Fretboard / Fret	The Fretboard is the board attached to a guitar's neck. The frets inserted in this board are used to change the pitch of a tone. The string can be pushed down onto the fretboard between two frets. Now the string can only vibrate from one fret to the other end of the string.
Fretter	The fretter is the part of the GuitarBot which moves along the linear slide. When toggled, a solenoid with a metal rod through its shaft pulls the string towards it. The housing of the fretter also acts as the fret itself (Figure 15).

Humanoid	A robot which is humanoid has a body similar to a human body. For example a head, two arms and two legs.
int	int, short for integer, is a data-type. Every time a new variable is created, its data-type has to be declared. For example in “int x”, the variable “x” can only be filled with an integer. In “digitalRead(int port)”, it means that the function requests an integer as an argument (which will be immediately put into the variable “port”).
Linear slide	A linear slide is a bearing which provides free motion in only one dimension (Figure 7).
Microsoft Visio	Microsoft Visio is an application for drawing diagrams and vector graphics. It’s for example used to draw flowcharts, but it’s also convenient to draw mechanical constructions, e.g. the layout to the GuitarBot (Figure 13).
Muter	The muter of the GuitarBot is the servo with rubber foam at the end of its arm. When the muter lowers, the rubber foam mutes the string, and the tone is stopped (Figure 2).
Pick	A plectrum (also called pick) is a tool, usually made of plastic, to strum the strings of a guitar.
Picker	The picker of the GuitarBot is the servo with a pick attached (Figure 2). Up and down movements of the servo cause the pick to play the string. The pick moving down across the string is called a downstroke, whereas the pick moving up across the string is called an upstroke.
Pulley	A pulley (Figure 6) is a wheel used with a timing belt. It can be used to drive the timing belt (when connected to a motor), to support the movement of the timing belt, or to transmit force from the timing belt (when connected to another component).
Relay	A relay is an electrically operated switch. It can be toggled using a low power circuit on one side of the relay. On the other side of the relay, where it acts as a switch, can be a high power circuit.

Servo motor	A servo motor (Figure 2), or just servo, is an actuator which allows precise positioning. However, a servo has a limited rotary range.
Shield	Talking about Arduino, a shield is a board with the pins arranged like the Arduino's pins. Hence, it can be easily plugged in by stacking it on top of the Arduino.
Solenoid	In engineering, a solenoid is a device which converts current into a linear force. There's a metal rod inside a tightly wound coil. Whenever current flows through the coil, a magnetic field is created and the metal rod is pulled inside the coil (Figure 5).
Stepper motor	A stepper motor (Figure 6), or just stepper, is a motor which divides the rotation into small steps. By telling the stepper how many steps to take, a precise rotation can be achieved.
Timing belt	A Timing belt (Figure 3) is a toothed belt which can be used to transfer a rotation from one axis onto another. In the GuitarBot, a timing belt is used to transform the rotation into a linear movement.

Program Code

```
// GuitarBot
// created 2013-10-02 by Frank Schaufelberger
//
// Read commands from SD card to move fretter and play notes, i.e. play a song on GuitarBot
// command file: plain text ASCII ";" delimited with CRLF, i.e. \r and \n at end of each line
// command format: time;string;command;[position]
// time: milliseconds
// string: 0-3
// command: m (move), p (play) or s (stop)
// position: only for move: the stepper position (positive in file, negative in program)

// remark: "string" in this context is always the guitar string, not a character string
/***** LIBRARIES *****/

#include <SD.h> // Arduino's SD library. uses SPI
#include <AccelStepper.h> // AccelStepper library
#include <Wire.h> // Library used by the PWMServoDriver Library
#include <Adafruit_PWMServoDriver.h> // PWMServoDriver Library

/***** PORTS *****/

int RELAY[] = {29, 27, 31, 33}; // pins for the fretter relays
int IR[] = {A8, A9, A11, A10}; // pins for the sensors
int STEP[] = {36, 38, 42, 44}; // step pins (stepper)
int DIR[] = {37, 39, 43, 45}; // direction pins (stepper)

// Default pins for SPI bus (SD Card)
// UNO MEGA
// MOSI 11 50
// MISO 12 51
// CLK 13 52
// Chip Select:
#define PIN_SD_CS 53

/***** OBJECTS *****/

AccelStepper stepper0(AccelStepper::DRIVER, STEP[0], DIR[0]); // create the Stepper on string 0
AccelStepper stepper1(AccelStepper::DRIVER, STEP[1], DIR[1]); // create the Stepper on string 1
AccelStepper stepper2(AccelStepper::DRIVER, STEP[2], DIR[2]); // create the Stepper on string 2
AccelStepper stepper3(AccelStepper::DRIVER, STEP[3], DIR[3]); // create the Stepper on string 3
AccelStepper stepper[] = {stepper0, stepper1, stepper2, stepper3}; // place the objects in an array

File myFile; // instance of the file on the SD card

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(); // construct a pwm object (servo controller)

/***** PRESETS *****/

int irLimit[] = {300, 300, 300, 300}; // IR values at position 0
int servoUp[] = {480, 500, 520, 490, 500, 511, 520, 490}; // servo position up (picker and muter)
int servoDown[] = {390, 480, 427, 470, 419, 490, 455, 470}; // servo position down (picker and muter)
int maxSteps = -870; // length of the slide in steps
// (all step positions are negative)
int vStepper = 500; // speed in steps per second
// go faster and the timing belt's slipping)

/***** STATUS *****/

int servoPos[] = {0, 1, 0, 1, 0, 1, 0, 1}; // 0 -> servo is in "up" position. 1 -> servo is in "down" position
int stringReady[] = {1, 1, 1, 1}; // 1 -> ready to move position (no tone playing, fretter released)
int fileStatus = 0; // fileStatus. 0 -> everything okay.
// 1 -> end of file. >1 -> error on position "fileStatus-1"
```

```

/****** GLOBAL VARIABLES *****/

unsigned long myTime;      // Time (ms) for the command of the current line
                          // this lasts for 50 days, an integer would only cover 32 seconds
int myString;             // string for the command of the current line
char myCommand;           // command of the current line ('m', 'p', 's')
int mySteps;              // position for the stepper if command = 'm'
int myLine = 0;           // number of current line (for debugging)
int irValue[4];           // array for sensor values
unsigned long startTime;  // the internal arduino clock at the time when the first command is read
int SDstatus=0;           // 1 if init is successful

// variables in subroutines with "_" are private, i.e. with local scope

/****** SETUP *****/

void setup()              // set up and calibration
{
    pinMode(PIN_SD_CS, OUTPUT);          // pin for SD Card Select
    if (SD.begin(PIN_SD_CS)) {           // initialize the SD Card
        myFile = SD.open("gibt.csv", FILE_READ); // opens the said file in read-only mode
        if (myFile) {
            SDstatus=1;                   // success
        }
    }
    if (SDstatus == 0) {                   // can't read SD -> exit here
        return;
    }

    pwm.begin();                          // start communicating with the servo controller
    pwm.setPWMFreq(60);                   // Analog servos run at ~60 Hz updates

    for(int i=0; i < 4; i++) // for all strings...
    {
        stepper[i].setMaxSpeed(vStepper); // maximum speed set to 1000 steps per second
        pinMode(IR[i], INPUT);             // pin settings
        pinMode(RELAY[i], OUTPUT);         //
        digitalWrite(RELAY[i], HIGH);      // turn off the relays
        pwm.setPWM(2*i, 0, servoUp[2*i]);  // picker to position UP
        pwm.setPWM(2*i+1, 0, servoDown[2*i+1]); // muter to position DOWN
        calibrate(i);                      // calibrate the stepper, i.e. go to start position
    }
    startTime = millis(); // "start my timer"
}

/****** MAIN LOOP *****/
void loop()
{
    if((millis()-startTime) >= myTime) // if the time has come... (or already passed)
    {
        if(myCommand == 'm')           // if the command says "move"
        {
            movePosition(myString, mySteps); // set the new target for the stepper
        }
        else if(myCommand == 'p')       // if the command says "play"
        {
            playTone(myString);         // play the tone
        }
        else if(myCommand == 's')       // if the command says "stop"
        {
            stopTone(myString);         // stop the tone
        }
        readLine();                     // read the next line
    }
}

```

```

    for(int i = 0; i<1000; i++)          // give about 20ms to bring the steppers some steps forward
    {
        for(int s = 0; s < 4; s++)      // for every string
        {
            stepper[s].runSpeedToPosition(); // move the stepper some steps closer to the target position
        }
    }
}

/****** CALIBRATE *****/
// move the fretter close to the sensor and set this as 0 position
void calibrate(int _string) {

    stepper[_string].setSpeed(vStepper); // set speed for the stepper
    irValue[_string] = analogRead(IR[_string]); // put the sensor value into the array

    while(irValue[_string] > irLimit[_string]){ // run till the beginning of the slide is reached
        irValue[_string] = analogRead(IR[_string]); // read from the sensor all the time
        stepper[_string].runSpeed(); // run towards the beginning of the slide
    }
    stepper[_string].stop(); // stop if reached
    stepper[_string].setCurrentPosition(0); // set current position as 0
}

/****** MOVE *****/
// change the target position of the stepper to the position in the argument
void movePosition(int _string, int _pos)
{
    if(!stringReady[_string]) { // force stop, if not ready to move
        stopTone(_string);
    }
    if(abs(_pos) > abs(maxSteps)) { // if _pos isn't within range, cancel the method (absolute value)
        return;
    }
    stepper[_string].moveTo(_pos); // set target position
    stepper[_string].setSpeed(vStepper); // set speed (has to be set after moveTo());
}

/****** PLAY *****/
// play the tone on the string in the argument
void playTone(int _string)
{
    int _PICKER = _string * 2; // get the picker port number from the string number
    int _MUTER = _string * 2 + 1; // get the muter port number from the string number

    digitalWrite(RELAY[_string], LOW); // trigger the fretter
    stringReady[_string] = 0; // string is not ready to move position
    pwm.setPWM(_MUTER, 0, servoUp[_MUTER]); // lift the muter

    if(servoPos[_PICKER]) // if picker is in position "down"
    {
        pwm.setPWM(_PICKER, 0, servoUp[_PICKER]); // upstroke
        servoPos[_PICKER] = 0; // update servo position
    }
    else // if picker is in position "up"
    {
        pwm.setPWM(_PICKER, 0, servoDown[_PICKER]); // downstroke
        servoPos[_PICKER] = 1; // update servo position
    }
}

/****** STOP *****/
// stop the tone on the string in the argument
void stopTone(int _string)
{
    int _MUTER = _string * 2 + 1; // get the muter port number from the string number

    pwm.setPWM(_MUTER, 0, servoDown[_MUTER]); // lower the muter
    digitalWrite(RELAY[_string], HIGH); // release the fretter
    stringReady[_string] = 1; // string ready to move position
}

```

```

/***** READ LINE *****/
// returns 0 if everything is okay, returns 1 if end of file is reached,
// returns position + 1 if an error occurred (for debugging)
int readLine()
{
    int pos = 1;                // position of the cursor. is returned if an error occurred
    char c;                    // currently read character
    myTime = 0;                // time of the command is reset
    mySteps = 0;               // value of steps is reset
    myLine++;                  // current line number (for debugging)

    if(!myFile.available())    // end of file or file not readable
    { return 1; }

    c = myFile.read();          // read first char
    pos++;                     // update position

    // ASCII characters to integers:
    // the numbers (time, string and position) have to be converted from ASCII characters to integers
    // the ASCII character "0" has the value 48 as an integer. therefore "c-48"
    // "10*myTime", because the numbers are read one by one.

    /*----- Time -----*/

    while(c != ';') {          // repeat until the first semicolon appears (but at least once)
        if(c < '0' || c > '9') // die if the read character isn't a number
        { return pos; }
        myTime = 10 * myTime + int(c) - 48; // see comment above, ASCII characters to integers
        c = myFile.read();           // read next char (semicolon, in the last loop)
        pos++;                       // update position
    }

    /*----- String -----*/

    c = myFile.read();          // read next char (should be 0-3)
    pos++;                     // update position
    if(c < '0' || c > '3')      // die if char isn't 0-3
    { return pos; }
    myString = int(c) - 48;     // see comment above, ASCII characters to integers

    /*----- Semicolon -----*/

    c = myFile.read();          // read next char (should be semicolon)
    pos++;                     // update position
    if( c != ';')              // die if char isn't a semicolon
    { return pos; }

    /*----- Command -----*/

    c = myFile.read();          // read next char (should be 'm', 'p', or 's')
    pos++;                     // update position
    if(c == 'm' || c == 'p' || c == 's') // store the char in myCommand if it's valid
    { myCommand = c; }
    else                       // die if it's not valid
    { return pos; }

    /*----- Semicolon -----*/

    c = myFile.read();          // read next char (should be semicolon)
    pos++;                     // update position
    if( c != ';')              // die if char isn't a semicolon
    { return pos; }

```

```

/*----- Steps -----*/

c = myFile.read();           // read next char (number or '\r')
pos++;                       // update position
if(myCommand == 'm') {       // if command is "move", get the position (mySteps)
    if(c < '0' || c > '9')    // die if the character isn't a number
        { return pos; }
    while(c != '\r') {       // get the rest of the number until the end of the line
        mySteps = 10 * mySteps + int(c) - 48; // see comment above, ASCII characters to integers
        c = myFile.read();   // read next char ('\r', in the last loop)
        pos++;               // update position
    }
    if(maxSteps < 0) {        // convert the steps in negative steps,
        mySteps *= (-1);     // if the length of the slide is in negative steps
    }
}
else if (c != '\r')          // if no move command: must be a \r
    { return pos; }

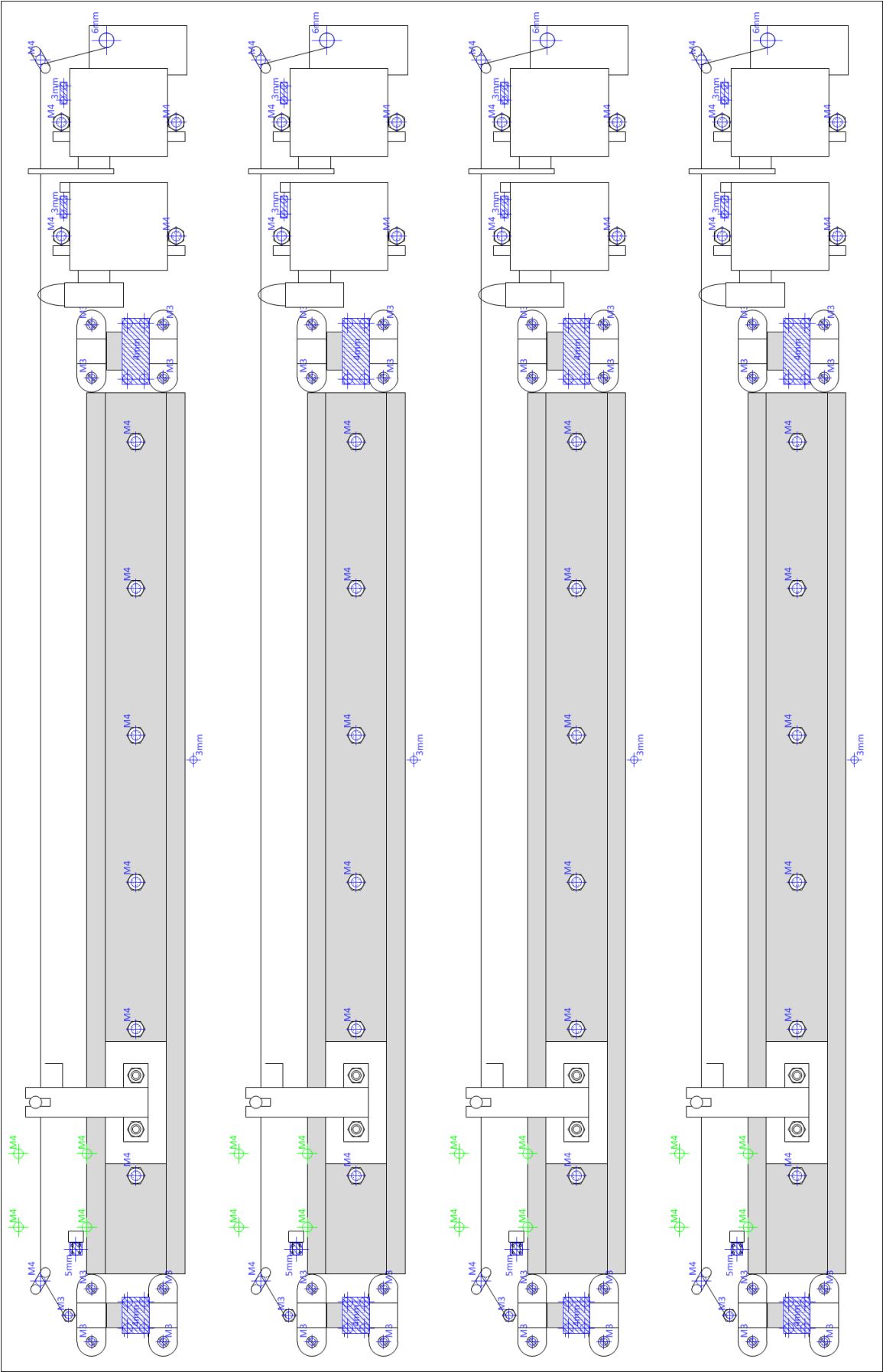
/*----- Line Feed -----*/

c = myFile.read();           // read next char (should be '\n', line feed)
pos++;                       // update position
if( c != '\n')               // die if char isn't a line feed
    { return pos; }

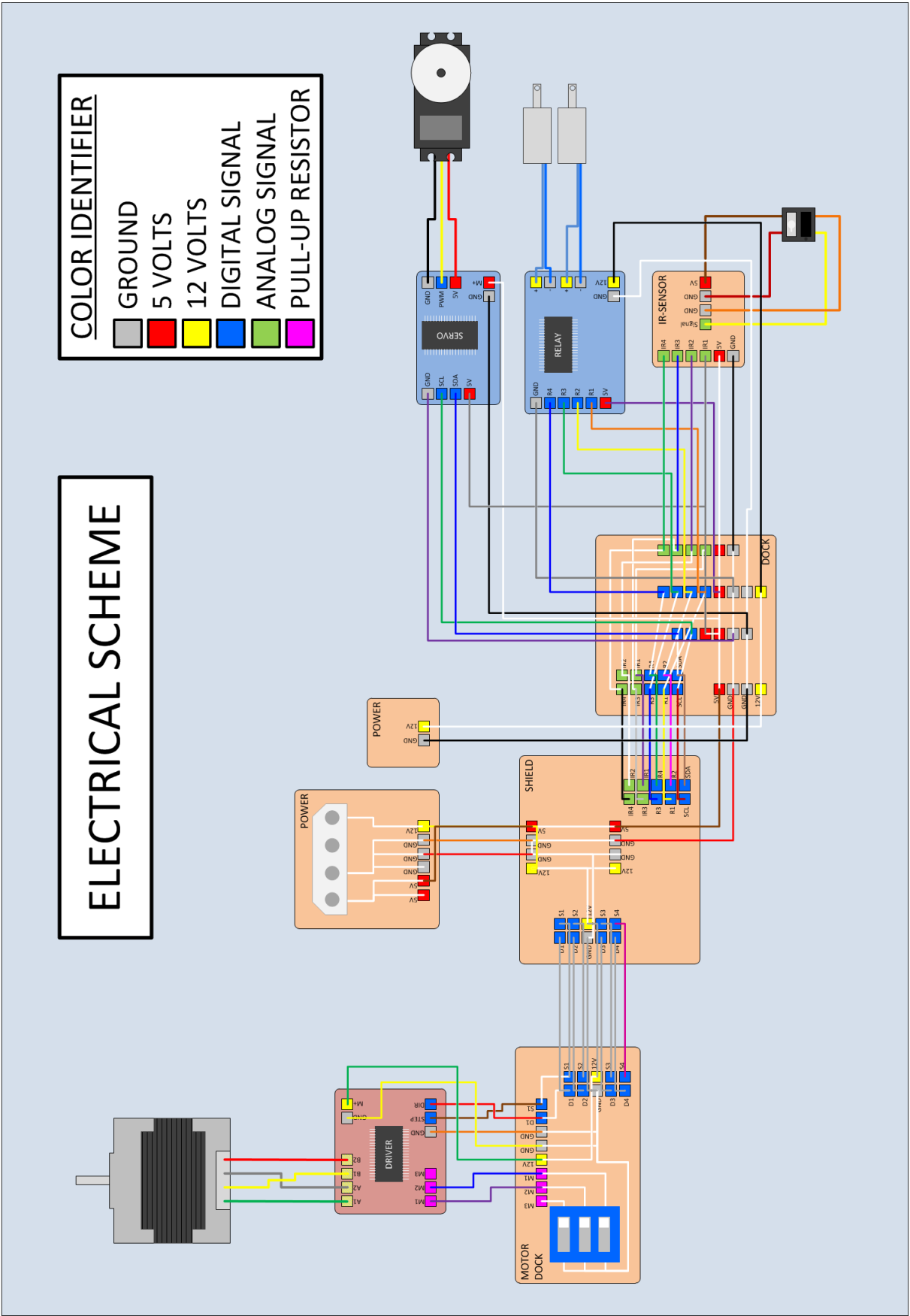
return 0;                    // everything okay
}

```

Layout



Electrical Scheme



List of Materials

	<i>Bezugsquelle</i>	<i>Anzahl</i>
Allgemein		
Holzplatten	Coop	2
Diverse Kabel / Stecker	Play-Zone.ch	
Breadboard Kabel Set	Play-Zone.ch	1
Distanzbolzen 50mm	Conrad.ch	6
Diverse Schrauben	Coop	
Kiste Utz Rako	Jumbo	1
Mainboard		
Arduino Mega	sparkfun.com	1
SD Kartenleser	Play-Zone.ch	1
Relais Modul	Play-Zone.ch	1
Power Supply	sparkfun.com	1
Power Molex Connector	sparkfun.com	1
Servos		
Servo/PWM Driver	Play-Zone.ch	1
Servo Hitec HS-311	Conrad.ch	8
Plektron	Musik Hug	4
Distanzbolzen 20mm	Conrad.ch	16
Metallbügel	Bauhaus	8
Antrieb		
DIP-Schalter 4P	Play-Zone.ch	1
Big EasyDriver	sparkfun.com	4
Heatsink	sparkfun.com	4
Halterung für 5mm Achse	hobbymodellbau.ch	16
Montagewinkel NEMA-17	physicalcomputing.at	4
Pulley GT2, 36 Teeth	robodigg.com	8
Pulley GT2, 20 Teeth	robodigg.com	4
GT2 Teeth Belt	robodigg.com	4
NEMA-17 Stepper Motor	robodigg.com	4

Fretter

Linearführung	igus.ch	4
IR Sensor	physicalcomputing.at	4
Widerstandsnetzwerk	Conrad.ch	2
Zylinderspule	Conrad.ch	5 (1 ging kaputt)
Alu Blech	Coop	1

Saiten

Gitarren-Saiten	Musik Hug	4
Ringschrauben	Coop	8
Distanzbolzen 60mm	Pusterla	8
Saiten-Spanner	(im Keller gefunden)	4

Declaration of Authenticity

I hereby declare that the work submitted is my own and that all passages and ideas that are not mine have been fully and properly acknowledged.

Place and date: _____

Signature: _____