MIDI INTERPRETER SOFTWARE

HAMK
UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Information Technology

Forssa, 25 November 2009

Timo Vahtera

HAMK
UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS

Information Technology
HAMK University of Applied Sciences

Title	MIDI Interpreter Software
Author	Timo Vahtera
Supervised by	Juha Sarkula
Approved on	_____._____.20_____
Approved by	

HAMK
UNIVERSITY OF APPLIED SCIENCES

FORSSA
Degree Programme in Information Technology

Author	Timo Vahtera	**Year** 2009
Subject of Bachelor's thesis	MIDI interpreter software	

ABSTRACT

The MIDI interpreter was part of the HAMK Örch Orchestra project. The goal of the Örch Orchestra was to compete in the Artemis musical robot competition held in Athens 3.6.2008.

The MIDI interpreter is a standalone hardware and software solution that interprets MIDI messages for a piano playing robot. This thesis involves everything from designing and creating the MIDI interpreter software, including relevant information about the hardware it was programmed for and about the Örch Orchestra project as a whole.

The function of the MIDI interpreter software is to receive MIDI messages for a MIDI sequencer and to interpret the messages into commands for the piano playing mechanical fingers of the robot.

The MIDI interpreter software's final version was completed on 28.5.2008; it was successfully used in the Artemis competition 3.6.2008.

Keywords MIDI, Interpreter, Music, Instrument, Piano, Robotics

Pages 33 pp. + appendices 10 pp.

HAMK
HÄMEEN AMMATTIKORKEAKOULU

FORSSA
Tietotekniikan koulutusohjelma

Tekijä	Timo Vahtera	**Vuosi** 2009
Työn nimi	MIDI-tulkkiohjelma	

TIIVISTELMÄ

MIDI-tulkki oli osa Hämeen ammattikorkeakoulun Örch Orkesteri projektia. Örch Orkesterin tavoitteena oli kilpailla Artemis musiikkirobottikilpailussa joka pidettiin Ateenassa 3.6.2008.

MIDI-tulkki on itsenäinen laitteisto- ja ohjelmistosovitus joka tulkitsee MIDI-viestejä pianoa soittavalle mekaniikalle. Tämä opinnäytetyö sisältää kaiken MIDI-tulkin suunnittelusta toteutukseen, sekä kaiken tarvittavan itse alustasta ja HAMK Örch Orkesterista.

MIDI-tulkin tehtävä on ottaa vastaan MIDI-viestejä ja tulkata niitä komennoiksi robotin pianoa soittavalle sormimekaniikalle.

MIDI-tulkin ohjelmiston viimeinen versio oli valmis 28.5.2008; tulkkia käytettiin onnistuneesti Artemis kilpailussa 3.6.2008.

Avainsanat MIDI, Tulkki, Musiikki, Instrumentti, Piano, Robotiikka

Sivut 33 s. + liitteet 10 s.

CONTENTS

1 INTRODUCTION

The purpose of this thesis was to design and program a MIDI interpreter software as part of the HAMK Örch Orchestra Musical Robot project.

Problem

The problem of this thesis is controlling an instrument playing robots mechanical fingers with MIDI messages.

Solution

The solution is to create an embedded electronics controller board that receives MIDI messages, interprets those messages, and then controls the robot's mechanics according to the received messages.

1.1 Premise

The Örch Orchestra team's motive was to create a musical instrument playing robot to compete in the Artemis Orchestra contest. The contest is an event directed towards electronics enthusiasts who would like to test their skill and gain experience in the field of embedded systems development. (Artemis Orchestra Contest 2008)

The aim of the Artemis contest is to create an embedded system that plays an instrument. The Artemis contest jury decides the winner by observing the execution, style and outcome of each contestant robot or robots. Essentially, whoever builds the most impressive solution wins. (Artemis Orchestra Contest 2008)

The creation of the Örch Orchestra robot was a team effort. Each member of the ten person team had their own role within the project. My responsibility was to program the software that would interpret MIDI messages into mechanical commands for the robot; this is also the main subject of this thesis.

1.1.1 Contest Rules

The robots entering the contest had to follow the Artemis contest rules set by the Artemisia Association. The rulebook was a 10 page PDF document given to the contest participants. It had 51 separate rules, some of them covering the procedures on entering the contest, others on common competition rules, scoring and so on.

Here are some robot specific rules worth mentioning:

- The instrument that was played should not be modified in any way
- The instrument should be detachable from the robot
- The robot must play the instrument from the same interface a human would play it
- The robots weight limit is 20 kg when not including the instrument
- The robot should wirelessly fetch the music file from a special contest server
- Robots should play the instruments using LilyPond as the music file format
- The robot should follow conducting using a conductors baton and change the speed of playing accordingly

(Artemis Orchestra Contest Rules 2008)

The idea of the contest was to specifically make a machine play an unmodified instrument the same way a human would. This is why the *no modified instruments*, *detachable instruments* and *human interface* rules were the three most important rules in the contest. The detaching rule ensured that the robot was built in a way that the instrument is exchangeable and that the instrument is not part of the robot.

In the end these rules turned out to be more like guidelines as not even one contestant followed all of the rules to the letter. For example our robot had a total weight of about 75 kg, almost four times as much as the contest limit. Even with the extra weight we were allowed to enter the contest.

The other rule that was completely ignored by all the contestants was the LilyPond rule. At the contest site it turns out that none of the contestants used LilyPond files. Instead of using LilyPond as the robot's default music file media, all of the contestants used MIDI files. The reason for this was that converting MIDI files to LilyPond files was very troublesome and playing them with a robot doubly so. The LilyPond rule was incomprehensible to begin with because LilyPond is a program used to create music engraving (music engraving is the art of drawing music notation). This was the reason why nobody used it in the competition. MIDI files are really common as there are thousands upon thousands of MIDI music files on the internet, anything from classical music to contemporary TV-series themes. MIDI files are only files that contain note data; this is better explained in chapter *2.1 MIDI Primer*. MIDI files are specifically created to be used in musical instruments and that is why every contestant used MIDI files instead of LilyPond files.

A conducting solution was also built as stated in the rules. The conducting unit is called the Tempo Recognition Unit. More information on the tempo recognition unit can be found in chapter *1.2.1 Tempo recognition unit*.

1.2 Robot

The instrument chosen by our team was a 61-key electronic piano. The piano was chosen because it is a fairly common instrument and it is relatively easy to play by a machine. There was discussion about building a guitar playing robot and a drum playing robot in addition to the piano so we would have an actual orchestra to enter the contest with. Unfortunately the additional robots were never built as we did not have enough time or manpower.

After the instrument was chosen two of our team members began to build a 61 fingered mechanical apparatus. This was to be the mechanical part of the robot that would do the physical playing. The other members focused on building a tempo recognition unit, a MIDI sequencer and a MIDI interpreter.

A previously made flute robot was also part of our orchestra. The flute robot was created by a different team for an Artemis contest held the previous year. It was incorporated in our Örch Orchestra project as the second instrument playing robot. In the competition the flute robot played beside the piano robot.

There are four parts in the piano robot all in all: The tempo recognition unit; the MIDI sequencer; the MIDI interpreter with the mechanics controllers and the mechanics. These four parts as a whole make the robot. These four parts can be seen in Figure 1. Each of these components is explained in its respective chapter. The piano is obviously not counted as a part of the robot.

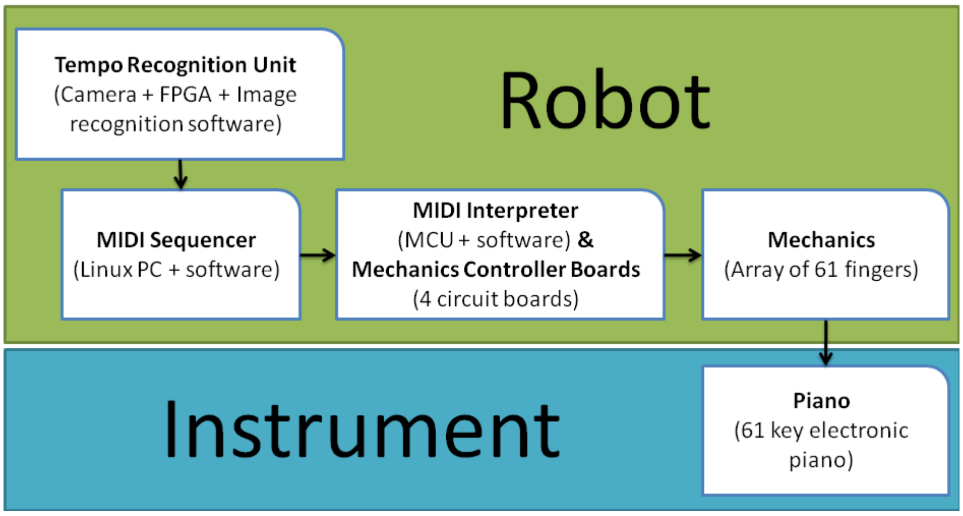


Figure 1 – *Diagram showing the four parts of the robot and the relationships of these parts. Arrows show flow of data, except the last arrow from mechanics to piano stands for mechanical operation.*

1.2.1 Tempo Recognition Unit

The Tempo Recognition Unit (also known as the Conducting Unit) is a camera connected to an FPGA board that has software programmed to recognize the movements of a specially made conductor's baton using a camera. The conductor's baton is a short stick with a red LED mounted on the other end.

When the LED is moved in front of the camera its position is being observed by the Tempo Recognition Unit software programmed for the FPGA. Between every image capture interval the tempo software calculates the speed of the baton's movement. Every time a new value is calculated the software sends the tempo information to the MIDI sequencer. Figure 2 shows the tempo recognition unit in use.

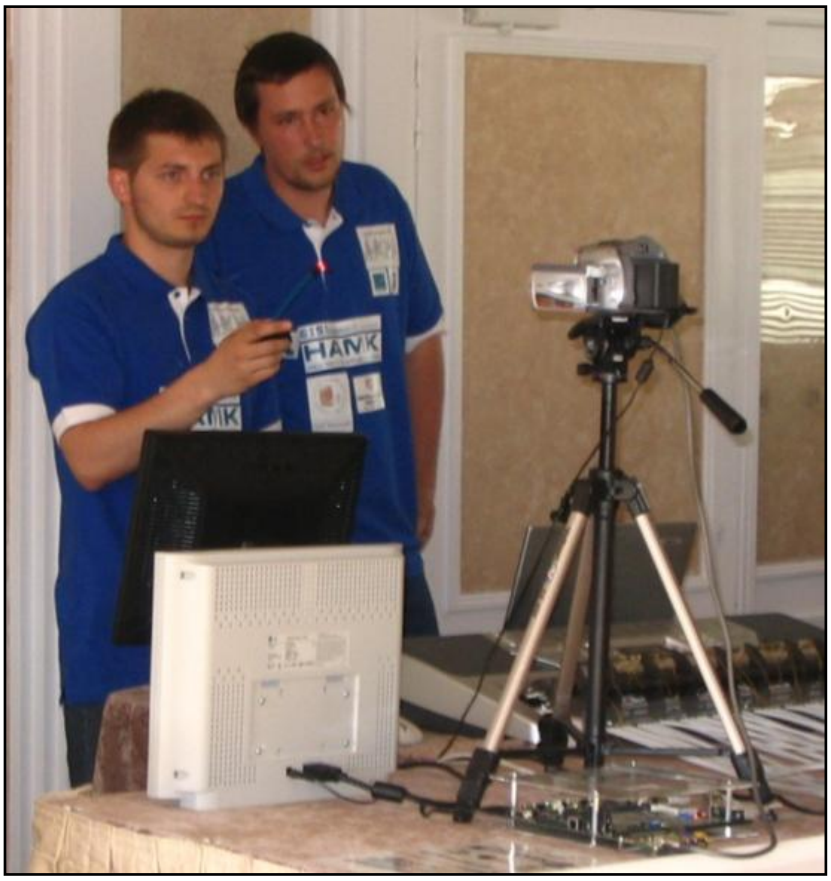


Figure 2 – *Tempo recognition unit in use at the Artemis competition. The tempo recognition units FPGA board is in the lower right corner of the image. Above the FPGA board is the camera. (Photo: Artemis Project 2008)*

1.2.2 MIDI Sequencer

A MIDI sequencer is in essence a MIDI music file player program. This means it opens a MIDI file and plays it through speakers or sends it to some other device that is capable of receiving and acting upon MIDI messages.

The sequencer created for this project is a program that runs on a Linux operating system. Its function is to read Standard MIDI Files and parse the relevant information for sending to the MIDI interpreter.

What is special about this sequencer is that it is also continuously receiving information from the tempo recognition unit. Using the tempo recognition unit's information the sequencer adjusts the speed of the playback.

An already existing MIDI sequencer could have been used if the contest would not have required special conducting of the robot. As there was no MIDI sequencer with such special capabilities, a sequencer capable of receiving real-time tempo information had to be programmed from scratch.

1.2.3 MIDI interpreter and Mechanics Controller Boards

The interpreter is a small microcontroller circuit board that is connected to 4 additional circuit boards that are called the Mechanics Controller Boards. The interpreter and the mechanics controllers work together to control the mechanical fingers.

When MIDI messages from the sequencer arrive to the interpreter, the interpreter uses the messages to control the Mechanics Controller Boards which in term control the mechanical fingers. The four MCBs can control each finger in the 61-finger array individually; corresponding to the interpreter's received messages.

The interpreter's hardware and software are better covered in their respective chapters *3.1.1 Hardware* and *3.1.2 Software*.

1.2.4 Mechanics

The mechanics were built from 61 linear solenoids connected to an array of mechanical fingers, one solenoid for each finger on the finger array. A solenoid can pull one finger down and hold it down for as long as a Mechanics Controller Board controls it to do so. Each of the four Mechanics Controller Boards is mounted to an acrylic glass frame with 16 solenoids as seen in Figure 3.

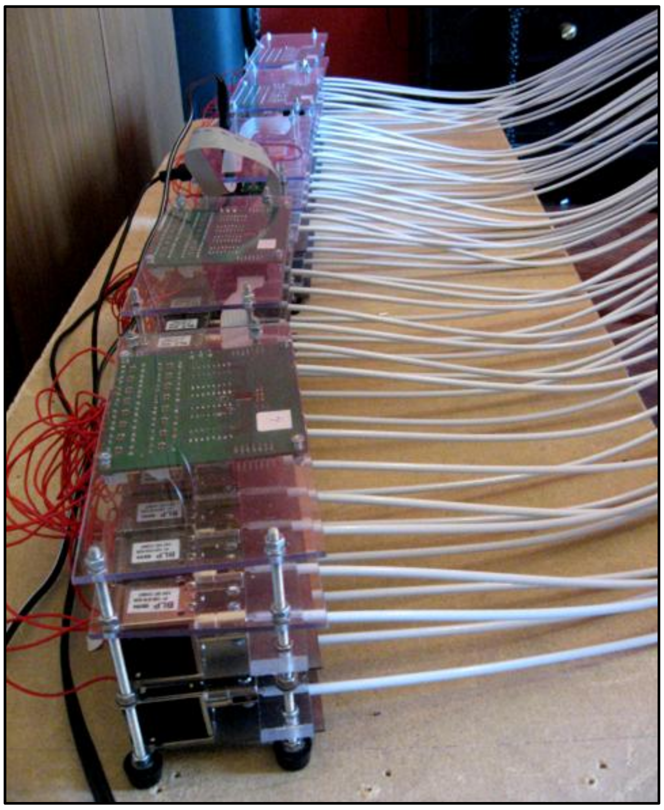


Figure 3 –*Four mechanics controller board frames with one MIDI interpreter frame in the middle. The white cabling running to the right is attached to the mechanical fingers. (Photo: Artemis Project 2008)*

A solenoid's operating principle is to retract whenever electric current is run through it. When a MIDI Note-On message is received by the interpreter it sends a command to the mechanics controller board to let electric current through for a specific solenoid controlling the right finger. This pulls the finger down and presses the key on the piano. Note-Off messages work the same way but only in reverse. Figure 4 shows a close-up photo of the fingers on top of piano keys.

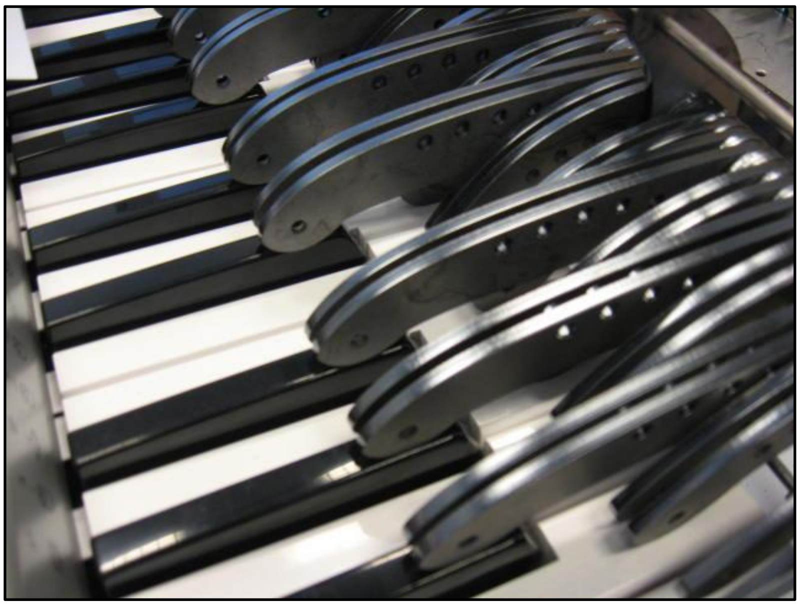


Figure 4 – *Photo of a portion of the keys. In this image the fingers are not yet connected to the cables and the pull-up rubber bands are missing as well. (Photo: Artemis Project 2008)*

Figures 5 and 6 both show the finger mechanics from two different perspectives. Unfortunately no better photos can be taken, as at the time of writing this thesis the finger mechanics have been dismantled. Figure 7 is an image of the entire finger array when it was still under development.

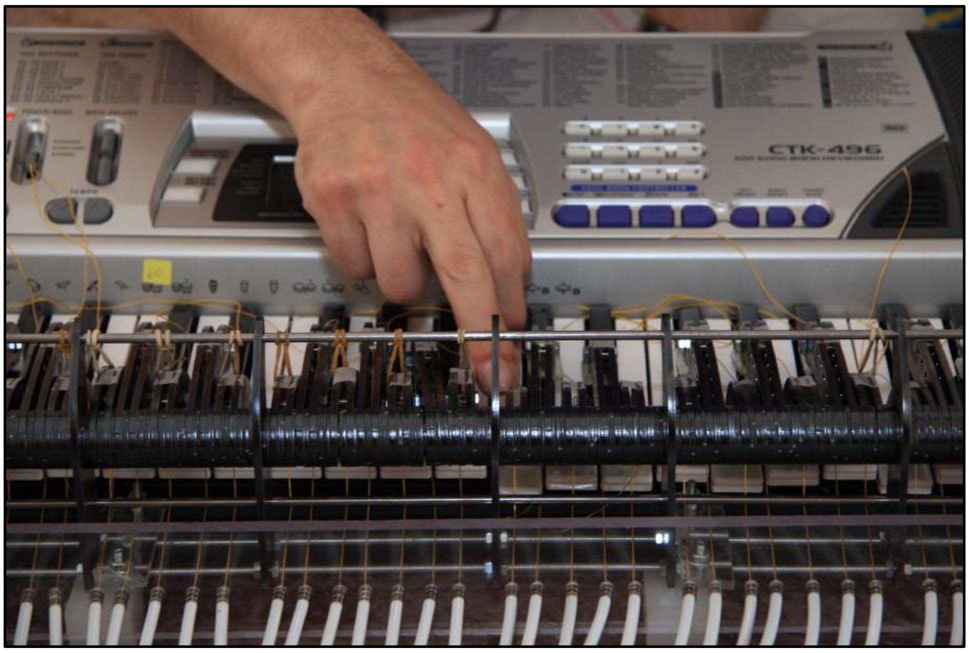


Figure 5 – *Finger mechanics placed on top of the piano as seen from behind. On the bottom the white cables are each connected to one of the fingers with a yellow string, the other end of the cables are connected to solenoids. (Photo: Artemis Project 2008)*

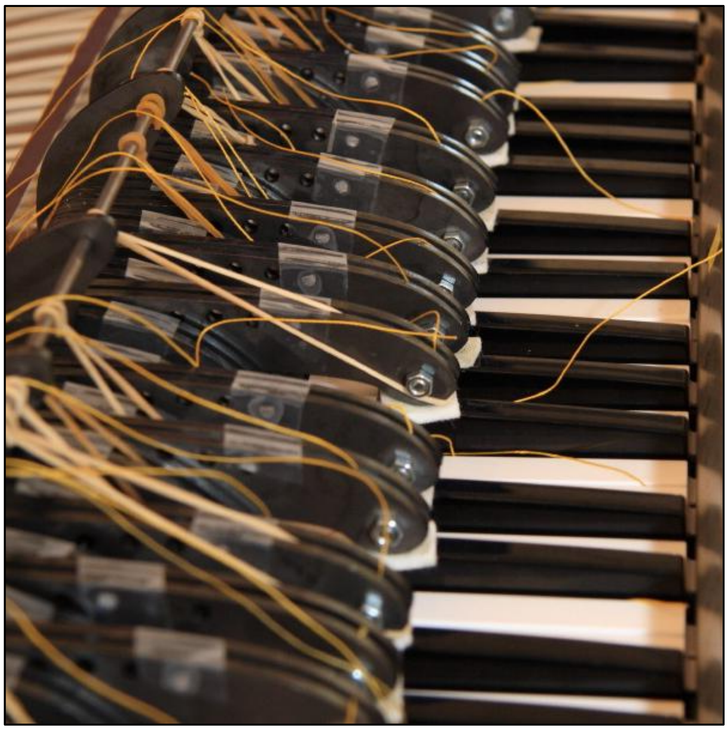


Figure 6 – *Finger mechanics placed on top of piano keys. This design was deemed non-functional because the fingers did not go down and back up correctly most of the time. This realization came unfortunately too late, this was the reason we came last in the contest. (Photo: Artemis Project 2008)*

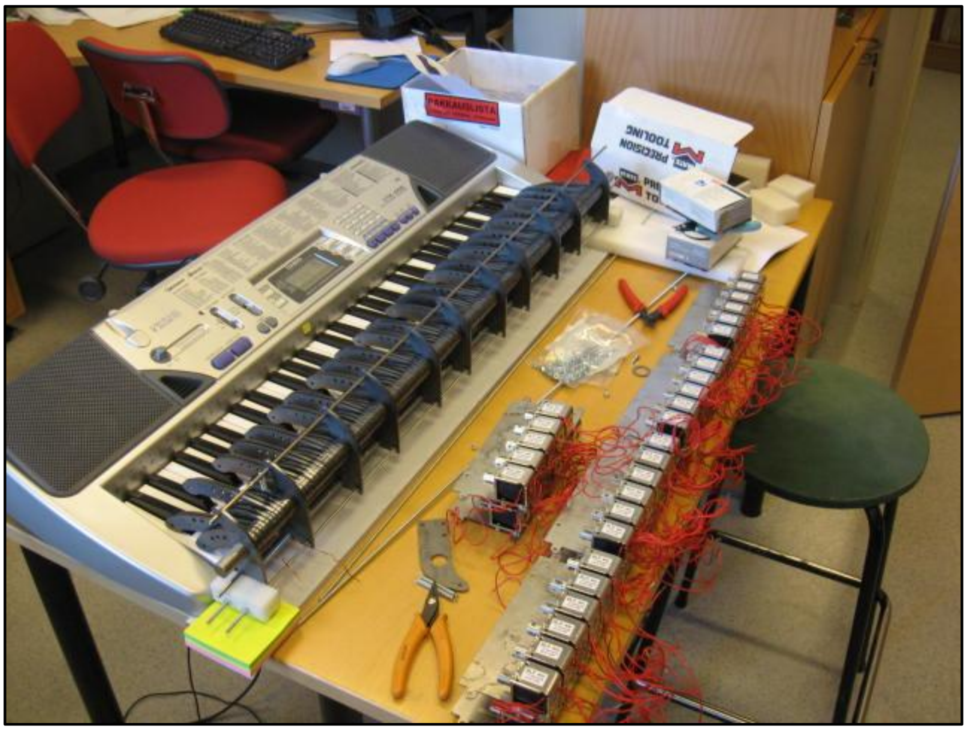


Figure 7 – *Incomplete finger mechanics. There are 61 solenoids on the right side of the table. The solenoids were later put into acrylic frames with their respective mechanics controller boards. (Photo: Artemis Project 2008)*

Piano

The piano used was a Casio CTK-496, a full-size 61-key electronic piano. Any piano could have been used as the mechanical fingers could play any kind of a piano as long as the piano keys are of standard size and the finger mechanics are placed correctly in front of the piano. There was no special reason for choosing this specific piano, other than the fact that it was pretty cheap. Figure 8 is a picture of the piano.

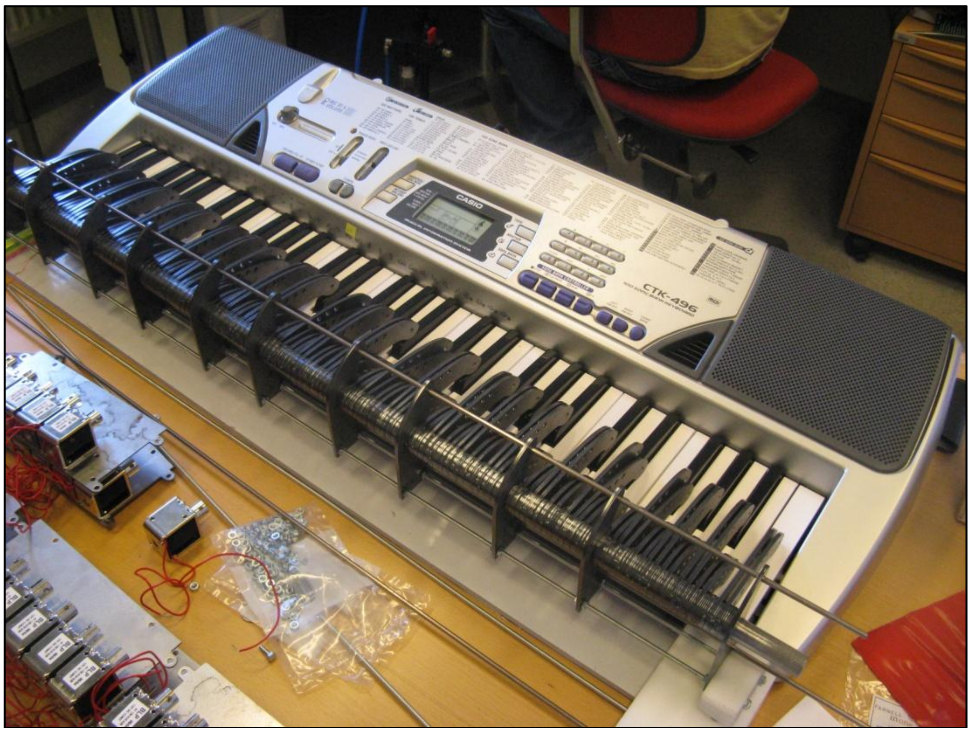


Figure 8 – *A Casio CTK-496, the piano used by our robot. (Photo: Artemis Project 2008)*

2 TECHNOLOGIES

The three main technologies used by the interpreter are MIDI, UART and I^{2}C. Each one of these technologies is better explained in their respective chapters. MIDI and UART walk hand in hand, as MIDI is the way musical performances are represented in digital format and UART is the technology used to transmit MIDI between devices. I^{2}C is used between the MIDI interpreter and the four Mechanics Controller Boards.

2.1 MIDI Primer

MIDI is short for Musical Instrument Digital Interface. It was created by the MIDI Manufacturer's Association for the purpose of enabling electronic musical instruments to communicate with each other. (MIDI Manufacturers Association 2008a)

In most cases MIDI devices fall into one of two categories. These two categories are:

- MIDI controlling devices
- MIDI controllable devices

This is somewhat self-explanatory as *MIDI controlling devices* are devices that send MIDI messages to *MIDI controllable devices*. MIDI controllable devices are devices which generate sound according to the received MIDI messages. (MIDI Manufacturers Association 2008a)

Most typical electronic pianos – also known as keyboard synthesizers – are both MIDI controllers and sound generators. The reason for this is that electronic pianos have built-in speakers to produce the necessary sounds for musical playback and the ability to control other instruments attached to them. Yet there are electronic pianos available that have no speakers, these are intended to be used as purely with computers or other MIDI controllable devices. (MIDI Manufacturers Association 2008a)

Most modern musical keyboards have MIDI capabilities. Even though MIDI was primarily designed for keyboard players, MIDI is also used in different kinds of instruments. There are MIDI wind controllers, MIDI guitars, MIDI drums, even MIDI accordions. These non-keyboard devices are referred to as 'alternate MIDI controllers.' (MIDI Manufacturers Association 2008a)

Specialized control devices are quite separate from alternate MIDI controllers or electronic pianos because they are not used to trigger notes on a MIDI controllable device; they control the mechanics of a MIDI music device using controllers similar to that of a mixing console. Such MIDI controllers can be used to control anything from music software to a lighting rig used in concerts. (MIDI Manufacturers Association 2008a)

All modern computers have the ability to create and play MIDI files, and also the ability to connect with other MIDI enabled devices with appropriate accessories. Anyone with a personal computer can compose, arrange, and record music, or use a computer to learn about music or how to play an instrument. Given that the computer has the necessary software installed. (MIDI Manufacturers Association 2008a)

There are countless software applications that involve MIDI. For example:

- MIDI sequencers and editors
- Auto accompaniment applications
- Notation programs
- Music teaching software
- Games
- DJ and remixing environments

(MIDI Manufacturers Association 2008a)

2.1.1 MIDI Technology

MIDI devices do not transmit recorded audio, nor are MIDI files recorded audio. MIDI files and MIDI transmissions are a series of *event messages* which hold information such as the musical notes to play. MIDI music files are somewhat analogous to music sheets, but in electronic form. Since the music is simply note data rather than recorded audio, the file size of MIDI files is tiny compared to recorded audio.

Usually MIDI devices are equipped with separate MIDI-In and a MIDI-Out connectors. MIDI devices are connected to each other with one-way MIDI cables through these connectors. MIDI devices use the standardized DIN 5-pin at 180° connectors as the MIDI connectors. Illustration of the port can be seen in Figure 9. The DIN connectors are circular, 13.2 mm in diameter, with a notched round metal shield that limits the orientation in which the plug and the socket can connect. There are 5 metallic pins inside the shield positioned in a semicircle, hence the “180°” in the name. (MIDI Technical Brainwashing Center 2009a)

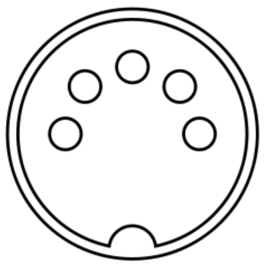


Figure 9 – *MIDI devices use DIN 5-pin at 180° connector for cables and connectors. The connectors on instruments are female, cables are male.*

The major MIDI messages fall into four types. These categories are *Channel Voice* messages, *Channel Mode* messages, *System Common* messages and *System Real Time* messages. Descriptions of each message type can be found in Table 1. Table 2 is a closer inspection of messages in the *Channel Voice* message type. There is no reason for closer inspection of the other types as the interpreter only uses *Channel Voice Messages*. (MIDI Manufacturers Association 2008b)

Most MIDI messages represent a common musical performance event or gesture such as *note-on*, *note-off*, *volume*, *pedal*, *modulation signals*, *pitch bend*, *program change*, *aftertouch*, *channel pressure* and so on. The *System Common* and *System Real Time* messages affect the playing device itself and have less to do with the actual musical performance. (MIDI Manufacturers Association 2008b)

Table 1 – *Shows the main MIDI Message types and their descriptions.*

Message Type	Description
Channel Voice	Contains musical performance information.
Channel Mode	Controls how synthesizers respond to MIDI messages.
System Common	Intended for all receivers in the system.
System Real Time	Used for synchronization between clock-based MIDI components.

Table 2 – *Channel Voice Messages and their descriptions. Messages used by the interpreter are highlighted in green.*

Message	Description
Note-Off	Note information. Sent when a note ends.
Note-On	Note information. Sent when a note begins.
Polyphonic Key Pressure	Pressure information. Works only on pressure sensitive synthesizers. (Aka. Aftertouch)
Control Change	Controls controller volume, pedal, etc.
Program Change	Specify instrument on a given channel.
Channel Pressure	Pressure information. Sends single greatest pressure value. (Aka. Aftertouch)
Pitch Wheel Change	Pitch information. Change in pitch wheel.

In this project however, we are not using MIDI exactly as it is supposed to be used. Instead of connecting the MIDI sequencer to an electronic piano's MIDI-In port, we do the playing in a roundabout way by connecting the MIDI sequencer to a device - the interpreter – which translates the MIDI messages to movement of the mechanical fingers which in term play the piano from the actual keys.

MIDI messages are transferred between devices using UART. This means that the MIDI sequencer is connected to the interpreter with a DIN 5-pin cable and the MIDI is transmitted using UART.

2.2 UART Primer

UART stands for Universal Asynchronous Receiver/Transmitter. It is used for serial communication between devices such as PCs, modems, terminals and MCUs. It also happens to be the technology MIDI uses to transmit MIDI messages between devices. (Extreme Electronics 2008)

The UART transmitter sends bytes of data bit by bit in sequence, serially over one data line, as opposed to transferring data in parallel over multiple lines. At the receiving end an UART receiver gathers the received bits and re-assembles them to complete bytes again. (Extreme Electronics 2008)

2.2.1 Protocol

The essence of asynchronous transmission is that the data can be sent without a clock line, using only a data line. The sending and receiving units are configured beforehand to have the right clock timing and bit parameters. This allows the sending of bits without having a separate clock line to signal the receiving device when to read the data line. *Word* is the unit of one transmission burst; this consists of a start bit, 5 to 8 data bits, an optional parity bit and a stop bit. Figure 10 is an illustration of a typical data word. (Extreme Electronics 2008)

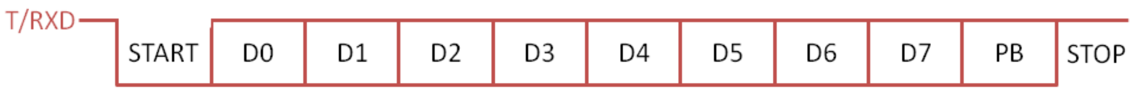


Figure 10 – *UART data line with 8 data bits and a parity bit. TXD stands for Transmitter on the sending device and RXD for the Receiver in the receiving device.*

Each word sent out by the transmitter starts with a start bit. The receiving device uses the start bit to synchronize its internal clock to be in the same phase as that of the transmitters. This is the key component in asynchronous data transfer. Every start bit sent synchronizes the internal clock of the receiver unit. This allows the receiver to read the rest of the data bits from the data line by looking the lines high/low state at exactly the right times. (Extreme Electronics 2008)

Following the start bit are the data bits. The data bits are sent starting with the Least Significant Bit. All of the bits are transmitted for exactly the same amount of time. Taking advantage of this, the receiver examines the state of the data line exactly at the midpoint of every bit sent in the line. Because the transmitter and receiver are preconfigured to have the same transmission settings the receiver knows how many data bits to expect and also if there is a parity bit before the stop bit. (Extreme Electronics 2008)

The parity bit is used to make the bit count always even or odd, depending on the transmission settings. This way the receiver will be able to do simple error checking on the received word. If the received word has an odd number of high bits, but the transmission parameters were set so that the words would always have an even number of high bits, the receiver can detect that at least one of the bits has changed its value. Of course this only works when the errors occur in odd numbers, because if two bits would swap their respective values the parity would not change. If one assumes that the receiver, transmitter and the data line are configured correctly and working as they are supposed to, the error rate drops to be so infinitesimally small that if more than one bit ever changes its value inside a data word it would be a very strong sign of something being wrong with one of the devices or the data line. (Extreme Electronics 2008)

The final bit in the word is the stop bit. It signals the end of the data word. It is only after the receiver has received the stop bit that it assembles the serially received data bits to a complete byte. The start, stop and parity bits are not saved; they are only used as means to an end and therefore are disregarded when the whole word has arrived. If for some reason the receiver does not receive a stop bit, it will regard the word as corrupt data and raise a Framing Error. The most usual cause for the Framing Error is that the transmitter's and receiver's clocks are not in sync, or are running at different speeds. (Extreme Electronics 2008)

The start bit of a new word can be sent immediately after the previous word's stop bit was sent. And in this manner the receiver and the transmitter communicate as long as there are words to send. When there is nothing to send, the data line can be idle. Whenever new data words are sent, the receiver self synchronizes itself with the transmitter when it receives the start bit. (Extreme Electronics 2008)

The speed of UART transmission is measured in Bauds. One Baud is equivalent to *one state change per second*. So a *baud rate* of 19200 would mean that the data line can change 19200 times in a second. This however does not translate to 19200 bits transferred per second because each byte is accompanied by at least two extra bits (three with parity) that do not count as data, these are the start and stop bits. (Extreme Electronics 2008)

The MIDI devices use UART in a configuration that has one start bit, 8 data bits and one stop bit with the baud rate of 31250 baud. Thus the MIDI sequencer sending the MIDI messages and the MIDI interpreter receiving the MIDI messages have both been configured with these parameters. (Extreme Electronics 2008)

2.3 I^{2}C Primer

I^{2}C, short for Inter-Integrated Circuit-bus, is a low speed serial bus connection. Because of licensing issues some device manufacturers have implemented the same technology under the name of Two Wire Interface (TWI) which is essentially the same thing. In the Atmel ATmega16 MCU we are using in the interpreter the bus is called TWI. In this thesis however this technology is addressed as I^{2}C because that is the name majority of people recognize. (Robot Electronics 2009)

I^{2}C is used by the interpreter to control the Mechanics Controller Boards.

Describing I^{2}C is easier when it is cut down into separate parts. These separate parts are physical design, hardware protocol and software protocol.

2.3.1 Physical design

The I^{2}C bus consists of two wires. These two wires are:

- SCL - Clock line
- SDA - Data line

All devices in the bus are connected to these two lines. Both of these lines are open drain drivers. This means that the bus lines are always high unless pulled down by one of the devices. Both of the lines are pulled back up by a 5V supply voltage with a pull-up resistor for each. In Figure 11 below is an illustration of a typical I^2C connection between a master device and slaves.

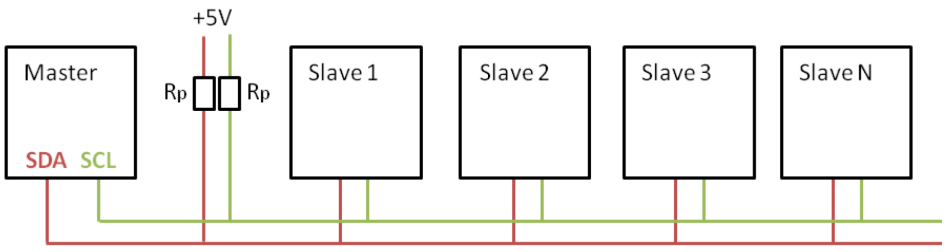


Figure 11 – *A typical I^2C connection. Red line stands for SDA (data line), green line stands for SCL (clock line). R_P is a pull-up resistor, one for each line.*

2.3.2 Hardware Protocol

Devices on the bus are either masters or slaves. The master is the device which controls the clock line. The master is also the only device that can initiate data transfer. In a typical I^2C bus setup there is a single master and multiple slaves. Having a bus with multiple masters is possible but this setup is unusual. (Robot Electronics 2009)

Every time a master device wishes to start or stop a data transfer it has to send a special sequence on the bus. These two sequences are known as the start sequence and the stop sequence, shown in Figure 11. What is special about these two sequences is that both of them change the data line (SDA) when the clock line (SCL) is up. This is how the start and stop sequence are identified by the slave devices. All other data transfer happens only when the clock line is low. (Robot Electronics 2009)

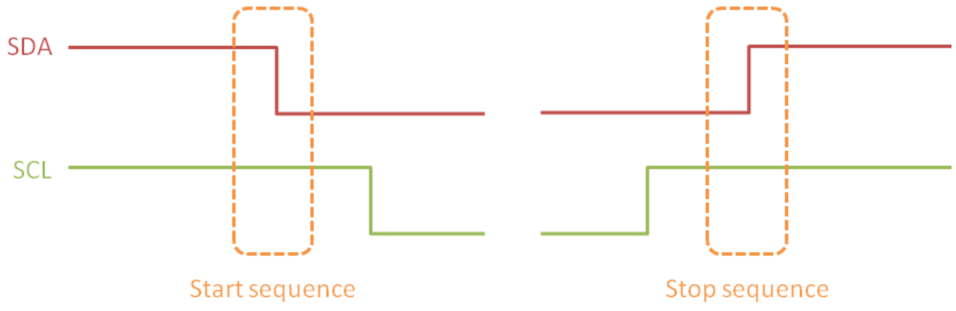


Figure 11 – *Start and stop sequences. These two sequences are the only ones which change the data line when the clock line is high.*

After the start sequence the data is transferred. In I^2C the data is sent in 8 bit sequences starting with the MSB (Most Significant Bit). After each byte the device receiving the data sends back an acknowledgement bit

(ACK). So for each sent data byte there are 9 clock cycles, 8 cycles for each bit in the byte and one for the ACK bit. Figure 12 shows the data and clock lines when transmitting a byte. A low ACK bit means that the receiving device is ready for more data. If for some reason the receiving device sends a high ACK bit it means that it cannot accept any more data (memory full or error of some sort). If the master receives a high ACK bit it will terminate the transfer and send the stop sequence. (Robot Electronics 2009)

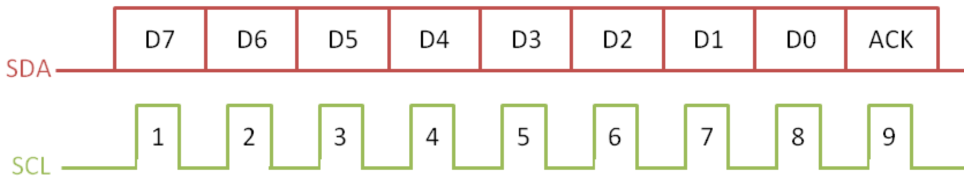


Figure 12 – *The clock and data lines during a data byte transfer.*

The master can send data to and receive data from each one of the slaves individually. This is achieved by giving each of the slaves a unique address. In most cases these addresses are 7 bit, however there are some devices that can handle 10 bit addresses. The 7 bits used for device addressing allows for a total of 128 slave devices on a single I^{2}C bus. As all data, this one is also sent out as a byte. The 8th bit after the address is the Read/Write bit. Both lines are shown in Figure 13. (Robot Electronics 2009)

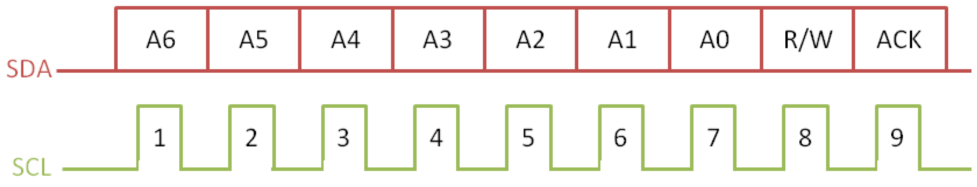


Figure 13 – *The clock and data lines during an address & R/W byte transfer.*

The original I^{2}C specification had a clock speed of 100 KHz. The specification was later updated by Philips to accommodate faster data transfer. These new speeds were called Fast mode (400 KHz) and High Speed mode (3.4 MHz). (Robot Electronics 2009)

2.3.3 Software Protocol

When the master wants to write to one of its slave devices first it has to send out the start sequence. After that, the address bits with the R/W bit. Then a byte is sent containing the internal register number where the master will write to. Then all the data bytes are sent, one or more in consecutive order. After all the necessary data bytes are sent the master sends a stop sequence to mark the ending of the transaction. The writing process from the master's perspective looks something like this:

1. Send a start sequence
2. Send the 7 address bits and the R/W bit (R/W bit is low)
    a. Receive ACK
3. Send internal register number that the master wants to write to
    a. Receive ACK
4. Send the data byte
    a. Receive ACK
5. [Send Optional data bytes]
    a. [Receive ACK]
6. Sends a stop sequence.

The reading is a bit more complicated, but not by much. When the master wants to read from one of its slave devices it has to send the starting sequence and the address bits with the R/W like before. After this the internal register number is sent, this is to inform the slave that the master wants to read that particular register. Then the start sequence is sent again, this is called a restart. This is where the process differs from writing. After the restart sequence the master sends the address bits again, but this time the R/W bit is high, signaling that it is ready to receive the slave's data. The slave then takes control of the data line, sending its internal registers byte to the master. After this the master sends the stop sequence to terminate the transaction. In short it happens like this:

1. Send a start sequence
2. Send the 7 address bits and the R/W bit (R/W bit is low)
    a. Receive ACK
3. Send internal register number that the master wants to read from
    a. Receive ACK
4. Send a start sequence again
5. Send the 7 address bits and the R/W bit (R/W bit is high)
    a. Receive ACK
6. Read data byte from slave
    a. Send ACK
7. Send the stop sequence

The MIDI interpreter uses only the writing aspect of I^2C. Each time the MIDI interpreter communicates with one of the Mechanics Controller Boards the interpreter opens a master slave communication and writes to both of the two registers the Mechanics Controller Board slave devices has. Each of the two registers in one MCB controls 8 fingers, 16 in total. This is better explained in the *3.2.2 Interpretation process* chapter. (Robot Electronics 2009)

3 MIDI INTERPRETER

3.1 Development

The interpreter software was developed to run on an Atmel ATmega16 microcontroller. The ATmega16 microcontroller unit was chosen because:

- It is sufficiently powerful for handling MIDI interpretation
- It has UART capabilities
    - Used to receive MIDI messages
- It has I^2C capabilities
    - Used to control the Mechanics Controller Boards
- It was readily available
- I knew how to program for it

(AVR Freaks 2009 & Atmel Product Page 2009)

The reason for ATmega16's high availability was that it is the microcontroller used in HAMK microcontroller programming courses. The idea was to use our school's own resources as the contest jurors favored self-built solutions over expensive commercial ones. If ATmega16 had not been chosen, any similarly capable microcontroller would have worked as an alternative.

3.1.1 Hardware

Figure 14 shows the connection between different parts of the interpreter unit and the mechanics.

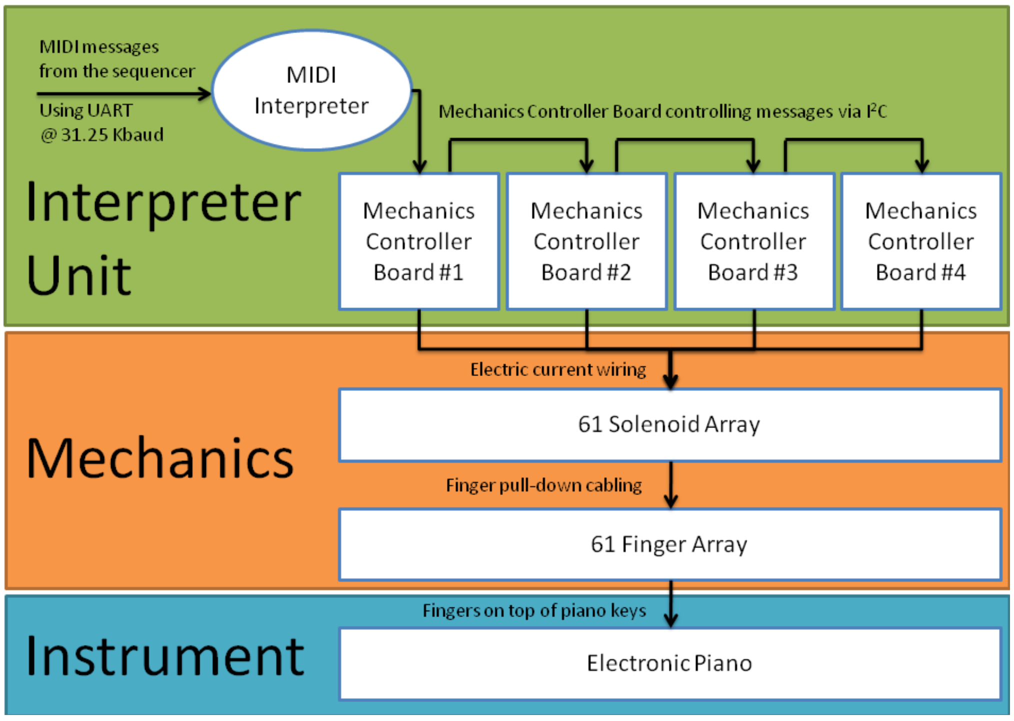

MIDI messages from the sequencer
Using UART @ 31.25 Kbaud
MIDI Interpreter
Mechanics Controller Board controlling messages via I²C
Interpreter Unit
Mechanics Controller Board #1
Mechanics Controller Board #2
Mechanics Controller Board #3
Mechanics Controller Board #4
Electric current wiring
Mechanics
61 Solenoid Array
Finger pull-down cabling
61 Finger Array
Fingers on top of piano keys
Instrument
Electronic Piano

Figure 14 – *Diagram showing the relations and connections between the interpreter unit, the mechanics and the instrument*

A specially made integrated circuit board was designed to hold the ATmega16 and the needed input and output connectors. On the circuit board there are two MIDI connection interfaces, one for MIDI input from the sequencer and another one for MIDI pass-through. The pass-through is used to pass the MIDI messages to the flute playing robot. There is also a connector that is used for sending I^2C messages to the Mechanics Controller Boards. A picture of the MIDI interpreter circuit board is shown in Figure 15. The schematics for the interpreter board are in Appendix 2.

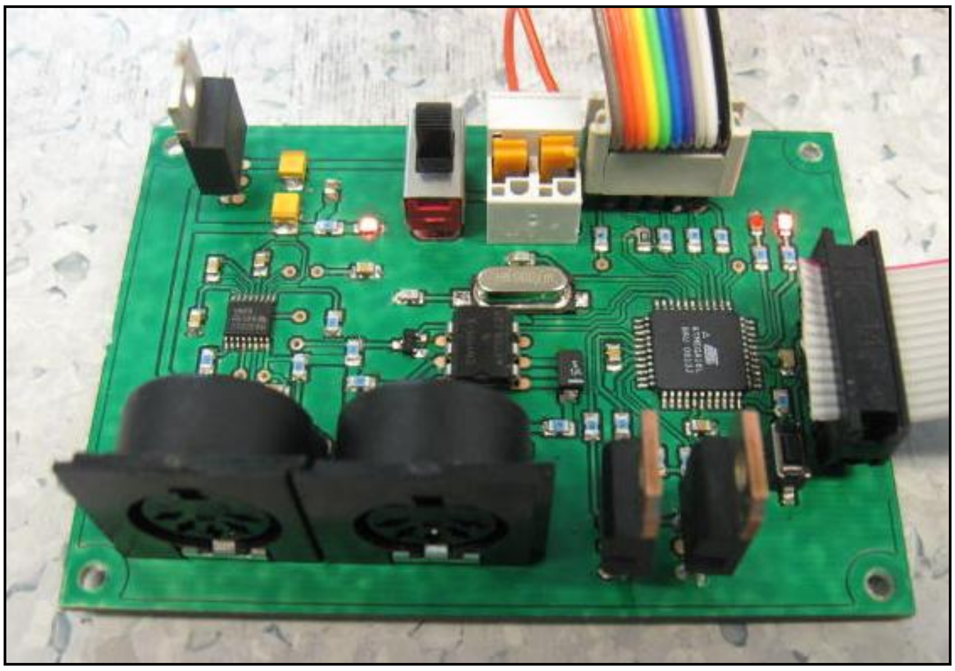


Figure 15 – *The MIDI interpreter board. Two MIDI connectors on the lower left side of the board, I2C bus cabling on the right, ATmega16 programming cable on top right. (Photo: Artemis Project 2008)*

The ATmega16 was set to operate at 8 MHz which was more than enough to handle the MIDI interpretation. The reason for choosing an 8 MHz clock was to get the right baud rate to receive MIDI, as the UART receiver is dependent on the clock rate of the device it is part of. Therefore a clock rate had to be used which was compatible with a baud rate of 31250 baud.

When powered on the interpreter continuously sends I^2C messages to all of the mechanics controller boards. These messages contain information on which solenoids should retract.

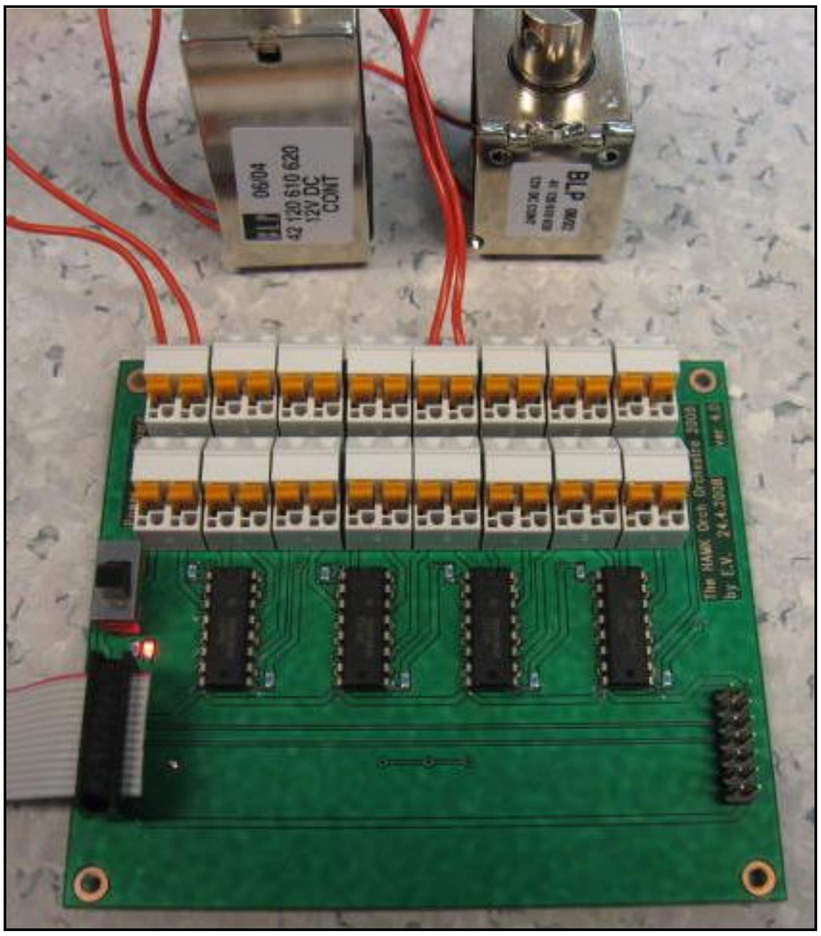


Figure 16 – *Mechanics controller board with two solenoids attached. On the left there is a cable connected to the interpreter, and to the right there is a connector for additional mechanics controller boards to be connected in sequence. (Photo: Artemis Project 2008)*

There are four Mechanics Controller Boards. Because of limitations in supplying power to the MCBs and practical circuit board sizes the 61 fingers are separated into four groups. Each one of the mechanics controller boards can control up to 16 fingers. Therefore the first three controllers control 16 fingers each, and the last one controls only 13 making a total of 61 fingers.

The MCBs are connected to each other in sequence where only the first one is connected to the interpreter. Figure 16 is an image of one of the Mechanics Controller Boards. All of the four boards are similar in appearance.

3.1.2 Software

The interpreter software was developed using:

- A standard desktop PC
- Ubuntu Linux, operating system
- Gedit, source code editor
- GCC, GNU Compiler Connection

- AVRDUDE, AVR downloader and uploader

There were two reasons to choose a Linux operating system. The main reason was performance as the computer used to develop the interpreter was quite old. It was best to use a lightweight operating system so that working would not be hindered by performance issues. The second reason was that the available software was easily downloadable and non-proprietary. (Canonical Ltd. 2009)

For source code editing the default Ubuntu text editor – gedit (written in lowercase on purpose) – was used. A proper programming IDE like Anjuta or KDevelop could have also been used with built-in source code compilers, debuggers and interpreters. Even without any advanced functionality, gedit was sufficient as a text editor because of its syntax highlighting. (Gedit 2009)

The source code for the interpreter was compiled using the GNU C Compiler which is part of the GNU Compiler Collection. (GNU Compiler Collection 2009)

AVRDUDE is software for programming AVR microcontrollers. AVRDUDE is purely a command line operated program without a graphical interface. It was used to upload the compiled interpreter software from the computer to the microcontroller unit on the MIDI interpreter board. (AVRDUDE 2009)

The compiled binary hex file was uploaded to the ATmega16 using AVRDUDE with the following parameters:

```
avrdude -p m16 -e -c stk200 -U flash:w:pohja.hex
```

Where:

- p, part number
    - m16 = ATmega16
- e, erase chip content
- c, programmer id
- U, memory operation
    - flash = type of memory
    - w = write
    - pohja.hex = filename

(AVRDUDE Option Descriptions 2009)

3.2 Interpretation

The interpretation happens by analyzing messages received by the interpreter and then acting upon these messages accordingly. After each received message byte the interpreter goes through several phases to determine the message's content. After receiving a whole message the interpreter sends a message to the correct Mechanics Controller Board, informing which finger should be down and which up. More on this in the *3.2.2 Interpretation process* chapter.

3.2.1 Messages

From the abundance of messages only few are relevant from the interpreter's point of view. When playing a piano the most relevant information is the piano key pressed, how long it is down, and when it is released. In terms of MIDI messaging these are known as *Note-On* and *Note-Off* messages. Any other messages are being ignored by the interpreter as the mechanics did not need anything else to play successfully. The specially made MIDI Sequencer even parsed all the unnecessary messages and only sent Note-On and Note-Off messages.

MIDI messages are usually of variable length depending on the type of the message. Fortunately for the purposes of the interpreter the Note-On and Note-Off messages are both always three bytes long.

The first byte in the MIDI message is the status byte and the two bytes following it are the note byte and velocity byte. The status byte contains the type of the message and the channel it is on. A status bytes upper nibble holds the *message type*, and the lower nibble holds the *channel* of the message. Status bytes always have the seventh bit set. A dissection of a MIDI Note-On message can be seen in Figure 17. (MIDI Technical Brainwashing Center 2009b)

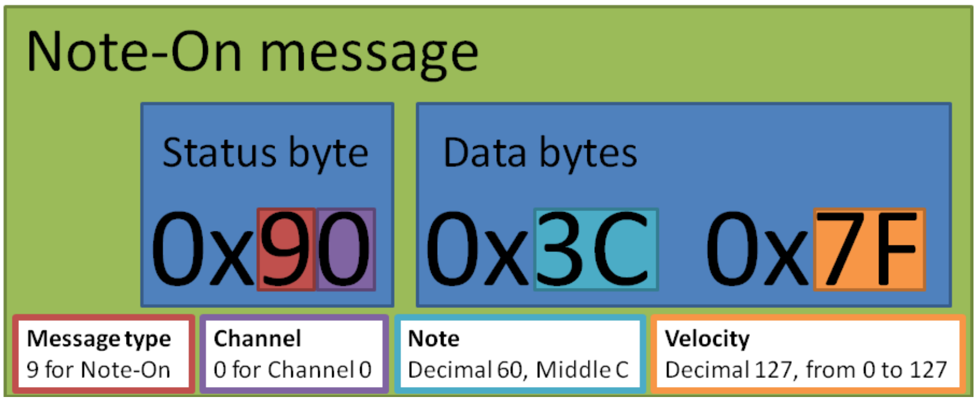


Figure 17 - *Dissection of a typical Note-On message. Note that the bytes are represented as hexadecimal values using the C programming notation where hexadecimal values are indicated with a preceding 0x.*

All MIDI messages include a channel number to separate different instruments. There are 16 channels in the MIDI protocol. In some MIDI files especially made for pianos the left and right hand portions are separated by putting them on a different channel. There is no rule how the channels should be used, but usually they are divided by instruments.

The purpose of the Note-On message is to signal when a note starts playing, and at what velocity. Velocity is the force at which the note is played, in this case, how strongly the piano key on the keyboard is pressed. In our case the velocity value is completely ignored as the mechanics that press the keys use constant force and cannot change the power at which the keys are pressed. The only exception regarding the velocity is that when the velocity of a Note-On message is 0x00 (zero), the Note-On message is to be interpreted as a Note-Off message. (MIDI Technical Brainwashing Center 2009c)

Note-Off works with the exact same principle as the Note-On message, instead of turning notes on it turns them off. The value of a Note-Off status byte is 0x80 (instead of 0x90 like the Note-On message). The velocity byte on the Note-Off message indicates how quickly the note should be released, as with the Note-On velocity, this byte is disregarded as the mechanics do not have such fine level of control. (MIDI Technical Brainwashing Center 2009d)

Running status is a special case of MIDI messaging where the status byte is omitted for maximizing efficiency. When several consecutive messages with the same status are sent, the status byte can be left out, sending only the two data bytes. This is called a *running status*. The status byte is left out for as long as messages with the same status are sent. A sequence of MIDI messages without running status can be seen in Table 3. (MIDI Technical Brainwashing Center 2009e)

In Table 4 for example, three Note-On messages are sent with only the first one having a status byte. The fourth message in the sequence is a Note-Off message, therefore a Note-Off status byte is sent to mark the end of the Note-On running status. All the messages after the first Note-Off status byte will be Note-Off messages until a new message with a different status byte is sent. (MIDI Technical Brainwashing Center 2009e)

A device that is receiving MIDI messages must always remember the last status byte received. If after two received data bytes the device does not receive a status byte, but a data byte instead, the receiving device will assume that the messages are using running status and continue processing the messages using the latest status byte received. This is how the interpreter does it as well.

Legend
🟥 Status byte
🟦 First data byte (note)
🟧 Second data byte (velocity)

Table 3 – *Sequence of MIDI messages without running status.*

0x90	0x3C	0x7F	0x90	0x40	0x7F	0x90	0x43	0x7F	...
First message			Second message			Third message			...

Table 4 – *Sequence of MIDI messages with running status.*

0x90	0x3C	0x7F	0x40	0x7F	0x43	0x7F	0x80	0x40	...
First message			Second		Third		Fourth		

(MIDI Technical Brainwashing Center 2009e)

3.2.2 Interpretation process

The interpretation process can be categorized into five phases, including the initialization even though it only happens once in the beginning. The walkthrough for each phase can be found after the phase flowchart in Figure 18. The phase flowchart is a simplification of the interpretation and it only shows the general outline of the interpretation process. A bit more detailed explanation of each phase is in its own subchapter after the chart.

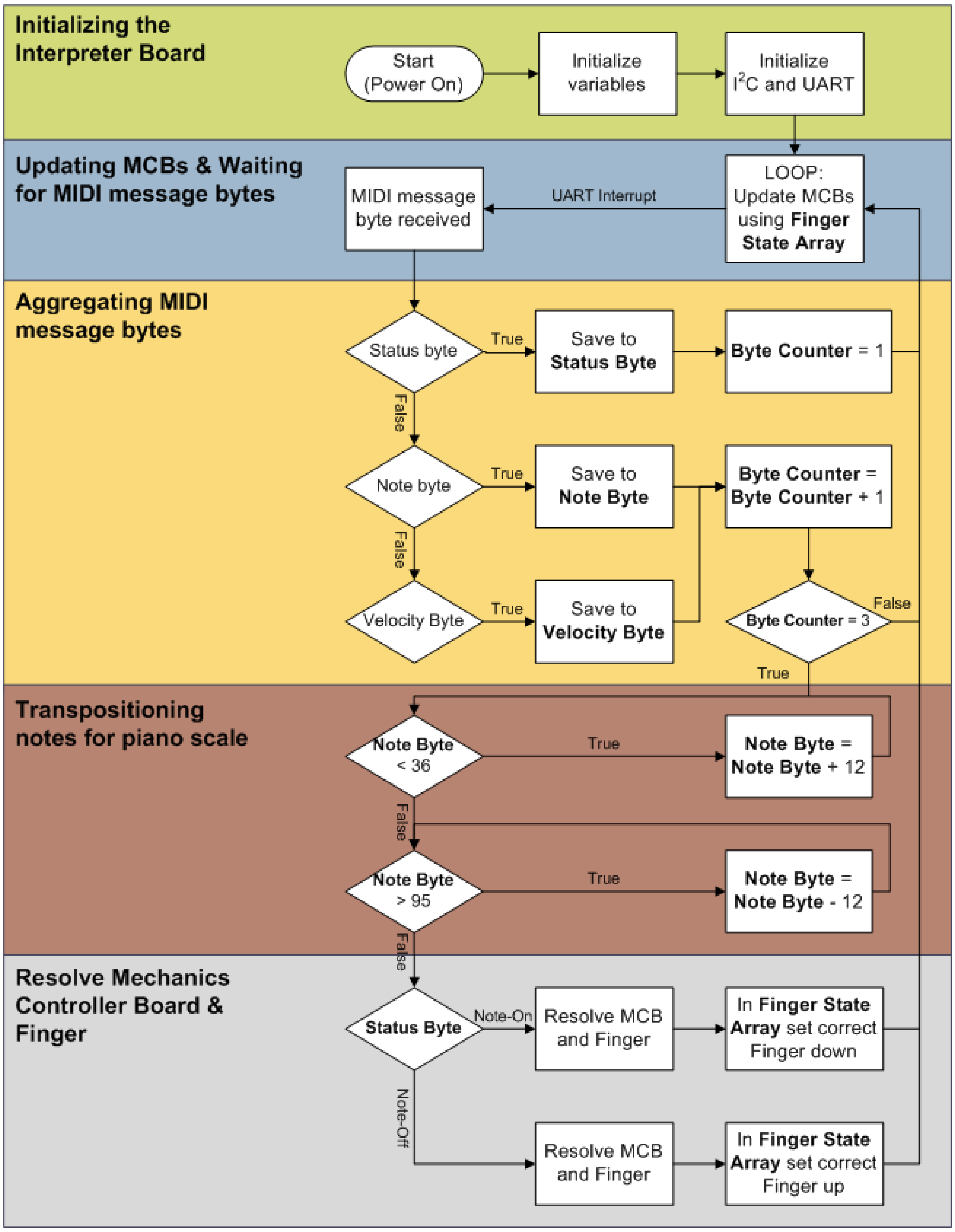

Initializing the Interpreter Board
Start (Power On)
Initialize variables
Initialize I²C and UART
Updating MCBs & Waiting for MIDI message bytes
MIDI message byte received
UART Interrupt
LOOP: Update MCBs using Finger State Array
Aggregating MIDI message bytes
Status byte
True
False
Save to Status Byte
Byte Counter = 1
Note byte
Save to Note Byte
Byte Counter = Byte Counter + 1
Velocity Byte
Save to Velocity Byte
Byte Counter = 3
Transpositioning notes for piano scale
Note Byte < 36
Note Byte = Note Byte + 12
Note Byte > 95
Note Byte = Note Byte - 12
Resolve Mechanics Controller Board & Finger
Status Byte
Note-On
Note-Off
Resolve MCB and Finger
In Finger State Array set correct Finger down
In Finger State Array set correct Finger up

Figure 18 – *Flowchart of the MIDI interpreter software's general operating principle. An in depth explanation of each phase is in the following subchapters. Texts that are in bold are variables.*

3.2.3 Initialization of the board

When powered on the board goes through the initialization phase. During the initialization all the variables are initialized as well as I^{2}C and UART.

The UART receiver is set for a baud rate of 31250 baud with a word of 8 data bits without a parity bit. As covered before, this is the right configuration for receiving MIDI message bytes from the MIDI sequencer.

The I^{2}C is set to work in a master transmitter mode with a clock rate of 83.33 kHz. Being the master of the I^{2}C gives the interpreter complete control over the mechanics controller boards connected to it as slaves as described in *2.3 I^{2}C Primer*.

The most noteworthy variables in the interpreter are:

- Status Byte
    - Holds the status byte of a received MIDI message
    - From Figure 17 this variable would hold 0x90
- Note Byte
    - Holds the note byte of a received MIDI message
    - From Figure 17 this variable would hold 0x3C
- Velocity Byte
    - Holds the velocity byte of a received MIDI message
    - From Figure 17 this variable would hold 0x7F
- Byte Counter
    - Goes from 0 to 3
    - 0 = no bytes received
    - 1 = status byte received
    - 2 = status byte and note byte received
    - 3 = status byte, note byte and velocity byte received
- MCB Finger State array
    - $4 \times 16\ bits$

The Status, Note and Velocity Byte variables are quite self-explanatory. Each holds the corresponding byte of an incoming message.

Byte Counter counts the number of bytes that have been received. This is used to determine the contents of the data bytes.

MCB Finger State array is an array of four 16-bit sequences, one 16-bit sequence for every MCB. Each bit represents a finger that the MCB controls. If a finger has its corresponding bit set to 0 it means that the finger is not pressed (finger is up). A bit set to 1 would mean that the finger is pressed (finger is down).

3.2.4 Updating MCBs & Waiting for MIDI message bytes

The main program loop of the interpreter is the MCB updating loop. It takes the MCB Finger State array and sends it over to the MCBs using I^{2}C. The first 16 bit sequence of the Finger State array is sent to the first card, the second sequence is sent to the second card and so on. This is done by sending each 16-bit sequence in two parts to the corresponding address of the MCB I^{2}C slave device. The I^{2}C slave device on each of the boards looks at the content of the Finger State array it has received and sets its corresponding outputs to match the array. The slave device has 16 outputs, and each of these outputs is connected to a single solenoid (finger). This way each of the MCBs is updated with the information on which finger should be down and which up.

On system start up the MCB Finger State array is initialized to be full of zeros. This means that all of the fingers should be up. The first time the program enters the loop it sends to all of the MCBs this empty Finger State array. All the fingers are up. This is expected as the sequencer has not yet started sending any MIDI message bytes. The loop keeps on updating the MCBs even though there is nothing happening. Nothing will happen before the MIDI Sequencer program starts sending message bytes.

When the first MIDI message byte arrives to the UART receiver, it creates an interrupt. This means that the interpreter stops going through the loop and goes to the UART interrupt function. The UART interrupt function is called every time a data word arrives to the UART buffer. In this case the byte that arrives is a MIDI message byte. The program then goes from the loop to the interrupt function to operate on the received byte.

3.2.5 Aggregating MIDI message bytes

The first thing checked in the UART interrupt function is the type of the message byte. This happens by examining two things: examining the most significant bit of the byte and the relation to previously received bytes. In the case of the first received byte the Byte Counter variable is 0 and the most significant bit is set.

The examination of the most significant bit tells us if the byte is a status byte or a data byte. All status bytes always have the MSB set, and all data bytes do not. With this the status bytes can be easily identified. When the first byte has been identified as a status byte it is saved into the Status Byte variable after which the Byte Counter is set to 1. After this the interrupt function runs to its end and the program is back in the MCB update loop. Nothing has changed from the MCBs point of view and nothing will happen on the MCB end until a complete MIDI message has been received by the interpreter.

Data bytes go into two categories, note number and velocity. Note number byte comes always after the status byte, and before the velocity byte. There is no special bit to identify data bytes, so the only way to identify a data byte is by observing the previous bytes. If the previous byte was a status byte then it is fair to assume that the byte is a note byte. If the previous byte was a note byte, then of course the byte currently under observation is a velocity byte.

Promptly after the status byte a note byte follows. This raises the interrupt function again, only this time the MSB is not set and the Byte Counter is 1. Using this, the interpreter determines that the received byte is a note byte. It is then saved into the Note Byte variable. The Byte Counter is incremented by one to a value of 2. Again the interrupt function ends and the interpreter goes back to the MCB update loop.

After the status byte comes the velocity byte. Once again the interrupt function is called. The MSB is not set and the Byte Counter is 2. This makes the arrived byte a velocity byte. The value is then saved to the Velocity Byte variable and the Byte Counter is incremented by one and is now 3. A full MIDI message has now been received.

After aggregating a complete set of bytes the interpreter then goes onto the next phase.

3.2.6 Transpositioning notes for piano scale

It was decided that if our robot would ever play a song that had notes it could not play they would be transpositioned into the playable range instead of just not playing the notes.

Transpositioning is essentially moving a note up or down one octave at a time. The reason for doing this is simple; if the note is too high or too low the piano we used cannot play it. Our piano has keys from 36 to 96 using the MIDI note numbering. In Table 5 there is a table that shows how the MIDI note numbers translate to actual notes in different octaves.

If a note were too high for our piano to play it would be transpositioned to a lower octave. This happens by subtracting 12 from the note number for as long as the note number gets below 97. If the note were from an octave too low the same transpositioning would happen, but only in reverse. The reason for subtracting or adding 12 each time is that each octave is 12 notes long.

After ensuring that the Note Byte is in playable range the interpreter will move on to the next phase.

Table 5 – *Note numbers to note and octave. The green areas in the middle represent the octaves playable by the piano we use. The four different green shadings separated by a thick black border represent each of the MCBs playing range. The red areas are out of range of the piano and are transpositioned to the nearest octave.*

Octave	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Legend
Mechanics Controller Board #1
Mechanics Controller Board #2
Mechanics Controller Board #3
Mechanics Controller Board #4

3.2.7 Resolve mechanics controller board & finger

In the final phase the interpreter looks at the Status Byte first. It does not matter if the Status Byte is a Note-On or a Note-Off; the interpreter has to resolve the correct MCB and finger either way. Resolving the MCB essentially means finding out which of the four MCB plays the note in the Note Byte variable. In Table 6 the green areas are split into four parts. Each of these parts represents an area playable by one of the MCBs, the lightest green for the first MCB, second lightest for the second MCB and so forth.

If the Note Byte holds the value 76 (decimal), the resolved MCB would be the third MCB. Resolving the finger means finding out which one of the fingers controlled by the MCB plays the note found in Note Byte. In the case of note 76 the finger would be the fifth finger that the third MCB controls. After resolving the MCB and the finger the interpreter then changes the corresponding bit in the Finger State Array. This bit would be the fifth bit in the third 16-bit sequence.

After changing the Finger State Array the interrupt function comes to an end. The software goes back to the MCB update loop. This time around the Finger State array has been changed, the new Finger State Array data is sent to each of the MCBs. The third MCB receives a 16-bit sequence where one of the bits is 1 instead of 0. This turns on the state of one of the outputs in the slave device. This output is connected to the fifth finger in the third MCB. The finger goes down and plays the key on the piano.

In this manner the interpreter goes on playing the piano. If a Note-Off message arrived, the process would be exactly the same except that the bit in Finger State Array would be changed to 0 instead of 1. The resulting change from 1 to 0 would change the corresponding output of the slave device to turn off. This loosens the solenoid and lets the rubber bands pull the finger back up, releasing the piano key.

3.2.8 Running Status and error handling

The running status is handled simply by never erasing the Status Byte. If a status byte is not received the interpreter uses the previous status as long as a new one arrives.

There is no error handling built into the interpreter. If for whatever reason the MIDI Sequencer starts sending bytes that are not part of a MIDI message the interpreter would most likely do nothing. The interpreter specifically “listens” for status messages that are either Note-On or Note-Off messages on channel zero. This means that if the MIDI Sequencer sends anything else than 0x90 or 0x80 as its first byte, the byte will be ignored. Were the note or velocity byte to be a random number, the interpreter would handle them just like any other note or velocity byte as there is no way of knowing what the note or velocity is supposed to be.

3.2.9 Source code

There is not much to say about the source code itself. The source code consists of approximately 550 lines of code when not counting empty lines. The source code file size is about 8 kilobytes. It is written using the C programming language. The compiled hex file is 5 kilobytes in size.

The well commented source code for the interpreter can be found in Appendix 1. This is not the exact source code used in the interpreter at the time of the competition. In functionality it is the same, but the names of several variables have been changed to better reflect their respective functions. This was the best way to ensure the legibility of the code.

4 CONCLUSION

4.1 Future

The interpreter was development only the piano robot mechanics in mind. Hence the MIDI interpretation was focused only on the Note-On and Note-Off messages. This simple approach works on an instrument like the piano because of its simple operation principle. With alterations to the source code it would theoretically be possible to add functionality which would enable the interpreter to operate with other instruments. However the Mechanics Controller Boards were specifically designed to be part of the piano finger mechanics and would be of little use with other instruments.

Each new instrument would require a different type of mechanics controllers. The modular design of the interpreter unit board and the mechanics controllers is well suited for such scenarios would anyone be interested to use it. If the interpreter were used to play different instruments parts of the source code would need to be rewritten.

After the 2008 Artemis contest development of the interpreter software has seized. A new team was assembled to enter the 2009 Artemis contest. The team of 2009 built new mechanics but used the interpreter discussed in this thesis. Unfortunately the interpreter had an unexpected power surge and burned into a fine crisp when attempted to be used at the 2009 contest. Consequently the 2009 team became last in the contest.

4.2 Interpreter

One could come to the conclusion that the interpreter in its entirety was a success, even though the mechanics failed to play their part. The problem of controlling the robot mechanics with MIDI messages was solved by creating a simple system to receive MIDI messages, interpret them, and control the mechanics according to them. A fairly simple idea, but developing it into a working program took several months of painstaking planning and programming. The final version of the interpreter software works finely.

In retrospect the source code of the interpreter software could have been more robust and efficient, but as the first serious microcontroller unit programming work I have ever done it ended up better than I dared to hope when I first started working on it.

Unfortunately the MIDI interpreter project itself did not contribute to the scientific field of computer engineering in any way, nor was it particularly ground breaking. It could be concluded that this project was a personal learning experience more than it was anything else.

SOURCES

Artemis Orchestra Contest 2008. Orchestra Competition '08 Contest & Gallery website. Read 28.8.2009.
https://www.artemisia-association.org/photo_gallery_2007-2008

Artemis Orchestra Contest Rules 2008.
Given to contest participants.

Atmel ATmega16 product page. Read 21.10.2009
http://www.atmel.com/dyn/products/Product_card.asp?part_id=2010

AVR Freaks 2009. Atmel ATmega16 device page. Read 30.8.2009.
http://www.avrfreaks.net/index.php?module=Freaks%20Devices&func=displayDev&objectid=56

AVRDUDE 2009. AVRDUDE website. Read 30.8.2009.
http://savannah.nongnu.org/projects/avrdude/

AVRDUDE 2009 Option Descriptions. AVRDUDE website. Read 31.10.2009.
http://www.nongnu.org/avrdude/user-manual/avrdude_4.html

Canonical Ltd. 2009. Ubuntu website. What is Ubuntu? Read 30.8.2009.
http://www.ubuntu.com/products/whatisubuntu

Extreme Electronics 2008. RS232 Communication – The Basics. Read 26.10.2009.
http://extremeelectronics.co.in/avr-tutorials/rs232-communication-the-basics/

Gedit 2009. Gedit website. Read 30.8.2009.
http://projects.gnome.org/gedit/index.html

GNU Compiler Collection 2009. GCC website. Read 30.8.2009.
http://gcc.gnu.org/

MIDI Manufacturers Association 2008a. MIDI Products. Read 30.8.2009.
http://www.midi.org/aboutmidi/products.php

MIDI Manufacturers Association 2008b. MIDI Messages. Read 21.10.2009.
http://www.midi.org/techspecs/midimessages.php

MIDI Technical Brainwashing Center 2009a. Hardware. Read 21.10.2009.
http://home.roadrunner.com/~jgglatt/tech/midispec.htm

MIDI Technical Brainwashing Center 2009b. Messages. Read 30.8.2009.
http://home.roadrunner.com/~jgglatt/tech/midispec/messages.htm

MIDI Technical Brainwashing Center 2009c. Note-On. Read 30.8.2009.
http://home.roadrunner.com/~jgglatt/tech/midispec/noteon.htm

MIDI Technical Brainwashing Center 2009d. Note-Off. Read 30.8.2009.
http://home.roadrunner.com/~jgglatt/tech/midispec/noteoff.htm

MIDI Technical Brainwashing Center 2009e. Running Status. Read 30.8.2009.
http://home.roadrunner.com/~jgglatt/tech/midispec/run.htm

Robot Electronics 2009. Using the I2C Bus. Read 26.10.2009.
http://www.robot-electronics.co.uk/htm/using_the_i2c_bus.htm

APPENDIX 1/1

MIDI INTERPRETER SOURCE CODE

```
/***********************************************************************

           Designer:              Timo Vahtera
           Date:                  28.5.2008
           Version:               1.2
           State:                 Finished / Working
           Description:           MIDI-Interpreter for ATmega16

;**********************************************************************/

#include <avr/io.h>
#include <avr/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <math.h>

// UART baud rate (MIDI input)
#define BAUDRATE 15 // (15 = 31250 Bauds @ 8MHz)

// Mechanics Controller Board I2C slave addresses
#define CARD1 0x42
#define CARD2 0x44
#define CARD3 0x46
#define CARD4 0x48

// MIDI status byte upper nibble
#define NOTEON 0x90
#define NOTEOFF 0x80

// Electronic piano scale (MIDI note numbering)
#define STARTNOTE 36
#define ENDNOTE 95

// ### UART Function prototypes ###

// Initialize UART, baudrate as parameter
void UART_Init (unsigned int);

// ### I2C Function prototypes ###

// I2C Initialization, MCB board address as parameter
void I2C_Init(unsigned char);
// I2C Data sending, MCB
void I2C_Send(unsigned char, unsigned short int);

// ### Interpreter Internal Function prototypes ###

// Which Mechanics Controller Board?
unsigned char Resolve_MCB(unsigned char);
// Which finger on the Mechanics Controller Board?
unsigned short int Resolve_MCB_Finger(unsigned char);

// Array of MCB Finger states
// MCB_Finger_States[0] = MCB1, MCB_Finger_States[1] = MCB2, etc.
volatile unsigned short int MCB_Finger_States[4] =
{0x0000,0x0000,0x0000,0x0000};

// ### MAIN ### //
```

APPENDIX 1/2

MIDI INTERPRETER SOURCE CODE

```
int main(void)
{
    // Initialize I2C ports
    I2C_Init(CARD1);
    I2C_Init(CARD2);
    I2C_Init(CARD3);
    I2C_Init(CARD4);

    // Must disable JTAG for PORTC usage (I2C/TWI uses PORTC)
    MCUCSR = 0x80;    // JTAG disabled (JTD = 1)
    MCUCSR = 0x80;    // twice to deactivate

    // MOSFET Init
    // Used by UART
    DDRD = 0xC0;
    PORTD = 0xC0;

    DDRB = 0xFF;
    PORTB = 0x00;

    // Initialize UART for MIDI input, baudrate as parameter (15 =
31250 @ 8MHz)
    UART_Init(BAUDRATE);

    while (1) {
        I2C_Send(CARD1, MCB_Finger_States[0]);
        I2C_Send(CARD2, MCB_Finger_States[1]);
        I2C_Send(CARD3, MCB_Finger_States[2]);
        I2C_Send(CARD4, MCB_Finger_States[3]);
    }

}

// ### Variables used in UART Interrupt ###

// UART input counter [0-3], because Note-ON & Note-OFF are 3 bytes
volatile unsigned int Byte_Counter = 0;

volatile unsigned char Status_Byte;
volatile unsigned char Note_Byte;
volatile unsigned char Velocity_Byte;

// ### UART Receive Interrupt ###

ISR(UART_RXC_vect)
{
    unsigned char Received_Byte = UDR;
    unsigned char MSB_Test;

    MSB_Test = Received_Byte & 0x80;

    // Find out if the received byte is Status, Note or Velocity
    if (MSB_Test == 0x80 ) {
        Status_Byte = Received_Byte;
        Byte_Counter = 1;
    } else if ( (Status_Byte == NOTEON || Status_Byte == NOTEOFF) &&
Byte_Counter == 1) {
        Note_Byte = Received_Byte;
```

APPENDIX 1/3

MIDI INTERPRETER SOURCE CODE

```
        Byte_Counter++;
    } else if ( (Status_Byte == NOTEON || Status_Byte == NOTEOFF) &&
Byte_Counter == 2) {
        Velocity_Byte = Received_Byte;
        Byte_Counter++;
    }

    // Byte_Count = 3 when a full MIDI message is received
    if (Byte_Counter == 3) {

        // Transposition for notes that go over or under our scale
        while (Note_Byte < STARTNOTE) {
            Note_Byte += 12;
        }
        while (Note_Byte > ENDNOTE) {
            Note_Byte -= 12;
        }

        // Message is a NOTE ON, Set the correct bit to 1 in
MCB_Finger_States
        if (Status_Byte == NOTEON && Velocity_Byte != 0x00) {

            switch (Resolve_MCB(Note_Byte)) {
            case CARD1:
                MCB_Finger_States[0] |= Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD2:
                MCB_Finger_States[1] |= Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD3:
                MCB_Finger_States[2] |= Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD4:
                MCB_Finger_States[3] |= Resolve_MCB_Finger(Note_Byte);
                break;
            }

        }

        // Message is a NOTE OFF, Set the correct bit to 0 in
MCB_Finger_States
        else if ((Status_Byte == NOTEON && Velocity_Byte == 0x00) ||
(Status_Byte == NOTEOFF)) {

            switch (Resolve_MCB(Note_Byte)) {
            case CARD1:
                MCB_Finger_States[0] &=
~Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD2:
                MCB_Finger_States[1] &=
~Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD3:
                MCB_Finger_States[2] &=
~Resolve_MCB_Finger(Note_Byte);
                break;
            case CARD4:
```

APPENDIX 1/4

MIDI INTERPRETER SOURCE CODE

```
                MCB_Finger_States[3] &=
~Resolve_MCB_Finger(Note_Byte);
                break;
            }

        }

    }

    // Running Status fix
    if ( Byte_Counter >= 3 || (Status_Byte != NOTEON && Status_Byte !=
NOTEOFF) ) {
        Byte_Counter = 1;
    }

}

unsigned char MCB_Address_Table[] = {CARD1,CARD2,CARD3,CARD4};

// Finds out which MCB controls the finger that plays the note
unsigned char Resolve_MCB(unsigned char pNoteByte)
{
    unsigned char card = (unsigned char)floor((pNoteByte-
STARTNOTE)/16);
    return MCB_Address_Table[card];
}

unsigned short int Finger_Table[] = {
    0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,0x0080,
    0x0100,0x0200,0x0400,0x0800,0x1000,0x2000,0x4000,0x8000,

    0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,0x0080,
    0x0100,0x0200,0x0400,0x0800,0x1000,0x2000,0x4000,0x8000,

    0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,0x0080,
    0x0100,0x0200,0x0400,0x0800,0x1000,0x2000,0x4000,0x8000,

    0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,0x0080,
    0x0100,0x0200,0x0400,0x0800,0x1000,0x2000,0x4000,0x8000
};

// Finds out which finger plays the note on the MCB
unsigned short int Resolve_MCB_Finger(unsigned char pNoteByte)
{
    return Finger_Table[pNoteByte-STARTNOTE];
}

// ### INITIALIZE UART, baudrate as parameter (15 = 31250 @ 8MHz) ###
//

void UART_Init (unsigned int ubrr)
{
    /* Set baud rate */
    UBRRH = (unsigned char)(ubrr>>8);
    UBRRL = (unsigned char)ubrr;

    /* Enable receiver and transmitter */
    UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
```

MIDI INTERPRETER SOURCE CODE

```
    /* Set frame format: 8data, 1stop bit */
    UCSRC = (1<<URSEL)|(0<<USBS)|(1<<UCSZ1)|(1<<UCSZ0);

    // Enable the Global Interrupt Enable flag so that interrupts can
be processed
    sei();
}

// ### I2C PORT INITIALIZATION, card address as parameter ### //

void I2C_Init(unsigned char card)
{
    // Set I2C Bitrate
    TWBR=10;
    TWSR=((1<<TWPS0)|(0<<TWPS1)); // Prescaler value = 4

    // SCL frequency = CPU Clock frequency / (16+2(TWBR)*4^TWPS)
    // SCL frequency = 8 Mhz / (16+2(10)*4^1) = 0.08333 MHz = 83.33
kHz

    // ### PORT 1 ###

    // Start Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
    while (!(TWCR & (1<<TWINT)));

    // Address
    TWDR = card;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // PORT Configuration Selection
    //                  * 0x06 = Port 1
    //                  * 0x07 = Port 2
    TWDR = 0x06;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // Make selected PORT => output
    TWDR = 0x00;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // ### PORT 2 ###

    // Start Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
    while (!(TWCR & (1<<TWINT)));

    // Address
    TWDR = card;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    TWDR = 0x07;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));
```

APPENDIX 1/6

MIDI INTERPRETER SOURCE CODE

```
    // Make selected PORT => output
    TWDR = 0x00;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // Stop Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
}

// ### I2C DATA SENDING, card number and data (=2 bytes) as parameter
### //

void I2C_Send(unsigned char card, unsigned short int data) //(unsigned
char port, unsigned char data)
{
    // ### PORT 1 ###

    // Start Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
    while (!(TWCR & (1<<TWINT)));

    // Address
    //                  * 0x42 = Card 1
    //                  * 0x44 = Card 2
    //                  * 0x46 = Card 3
    //                  * 0x48 = Card 4
    TWDR = card;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // PORT Selection
    //                  * 0x02 = Port 1
    //                  * 0x03 = Port 2
    TWDR = 0x02;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // Data to send
    TWDR = (unsigned char)(data & 0x00FF) ;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // ### PORT 2 ###

    // Start Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
    while (!(TWCR & (1<<TWINT)));

    // Address
    //                  * 0x42 = Card 1
    //                  * 0x44 = Card 2
    //                  * 0x46 = Card 3
    //                  * 0x48 = Card 4
    TWDR = card;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // PORT Selection
```

MIDI INTERPRETER SOURCE CODE

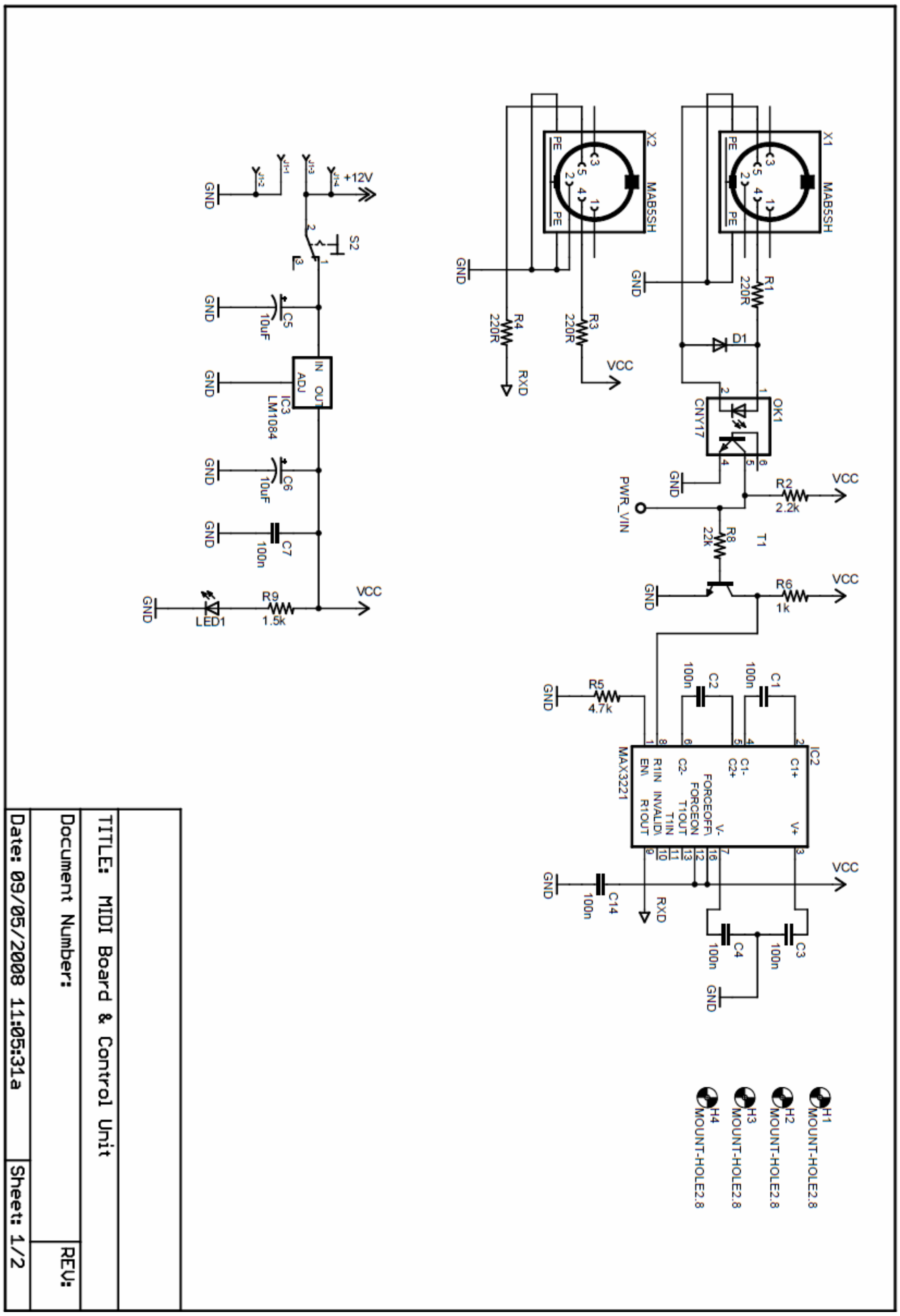
```
    //                  * 0x02 = Port 1
    //                  * 0x03 = Port 2
    TWDR = 0x03;
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // Data to send
    TWDR = (unsigned char)(data >> 8);
    TWCR=(1<<TWINT) | (1<<TWEN);
    while (!(TWCR & (1<<TWINT)));

    // Stop Condition
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
}
```

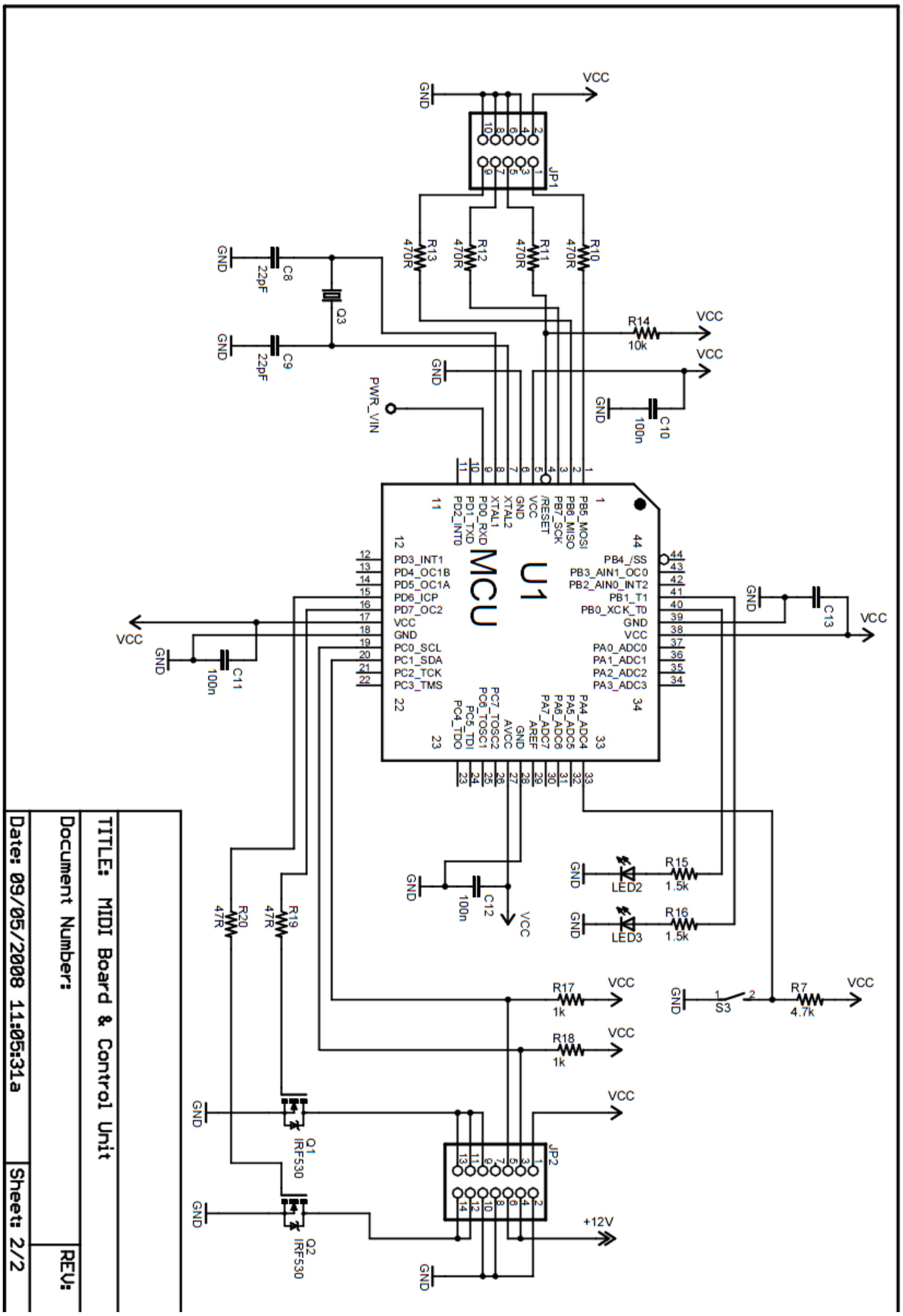
APPENDIX 2/1

MIDI INTERPRETER HARDWARE SCHEMATIC



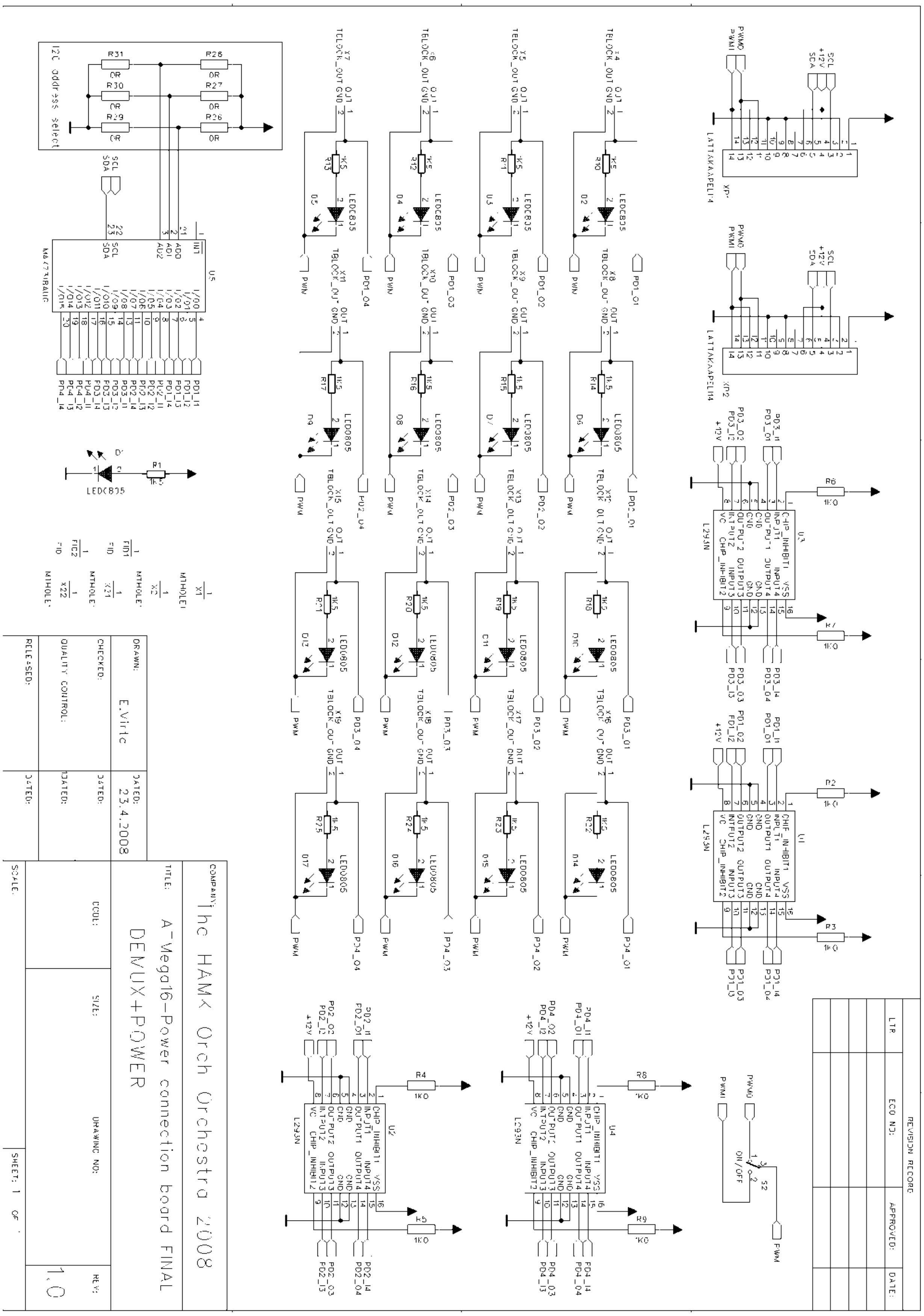
APPENDIX 2/2

MIDI INTERPRETER HARDWARE SCHEMATIC



APPENDIX 3/1

MECHANICS CONTROLLER BOARD HW SCHEMATIC


COMPANY: The HAMK Orch Orchestra 2008
TITLE: A-Mega16-Power connection board FINAL DEMUX+POWER
DRAWN: E.Virta
DATED: 23.4.2008
CHECKED:
QUALITY CONTROL:
RELEASED:
REV: 1.0
SHEET: 1 OF 1
REVISION RECORD
I2C address select
U5
U1
U2
U3
U4
L293N
LATTAKAAPELI14
X1
X2
S2
ON/OFF
PWM
R1
LED0805