Bringing back to life a Colinbus Profiler CNC-Router Project Report

Claude BAUMANN Computarium

last edited: Monday 29th March, 2021 (09:25)

Abstract

This lab report will document the development of the project to bring back to life a vintage Colinbus Profiler CNC-router, which has been acquired by donation to the Computarium and placed at the disposal to the author for cabinetmaking. The major problem to overcome consists in making the device compatible with G-code, the overall standard in CNC operation.[1]

I Colinbus Profiler

Last edited: 18/02/2021 (just corrected a few mistakes.)

Figure 1: Colinbus Profiler

[1] See a nice basic introduction at https://www.precifast.de/cnc-programmierung-mit-g-code/.

This milling machine was promoted in 2007 by the Elektor magazine (cf. Fig. 1).[2] It was sold as a kit. Professor Jean Mootz acquired this device for the purpose of milling circuit boards. He therefore used a homebrew board holder (cf. Fig. 2) instead of the aluprofil base. Robust construction, good accuracy, 30 x 40 cm working area. The machine is made from steel and aluminium. This combination provides sufficient weight and stability to withstand the forces of the running machine. The three axes of the Cartesian design uses a roller system travelling along precision steel rods. This allows good mechanical precision with minimum play. Each of the axes is driven by a spindle with a special zero-backlash nut. All mobile parts need cleansing and lubricating, and probably realignment. The z-axis allows the use of a high variety of milling and routing systems. This makes the machine interesting for limited wood working such as profiling, milling and engraving. However an attachment board is missing on the base.

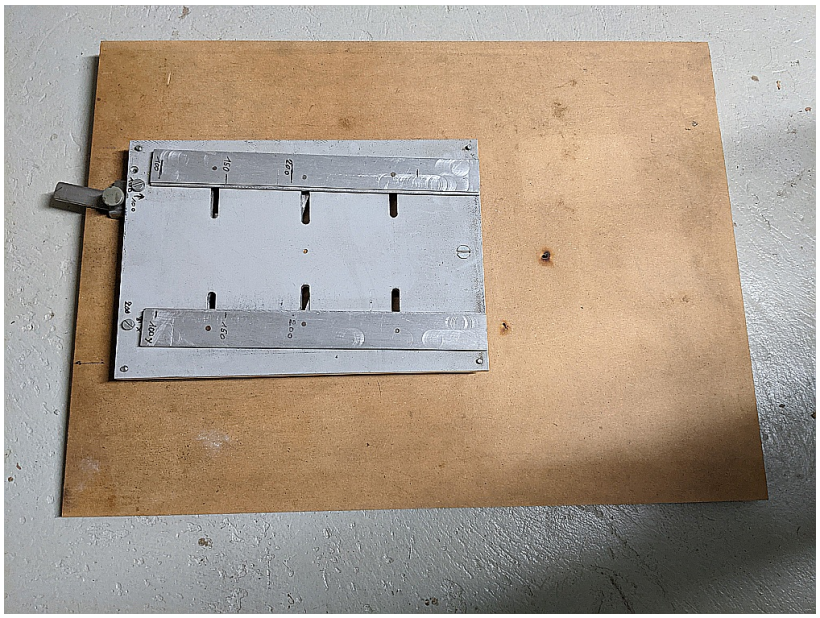


Figure 2: Homebrew circuit board holder for the milling process. The board is clamped by activating the lever. This assures a perfect parallel alignment to the main support.

The main controller board is fixed into an aluminium case beneath the working base (cf. Fig3). A separate torus transformer provides 30V and 8V alternate supply voltages, which are rectified. The board has three control systems for the three axes. A second board containing the microcontroller is mounted on top of the main board (cf. Fig. 4). It is strongly held by 3 connectors. The device uses a Renesas H8/3003 microcontroller and some ROM-memory. Although a row of ULN2003 / ULN2803 driver chips are being used for interfacing with the computer, this board has reduced features only. So, UART communication is the exclusive communication channel.
The machine uses three excellent Nanotec ST4118M1206-KFRA stepper motors (cf. Fig. 5).[3] According to the datasheet, these motors may be driven in bipolar mode with 1.2A/phase at 3.7V. Because of the *emf*, much higher voltages are allowed, but the current may never exceed 1.2A per coil. Holding torque is 0.28Nm, and detent torque is 9.8E-3Nm. Step angle is 1.8°. A switch is added to each motor. This allows easy zero-positioning.

[2] H. Baggen, *Profiler*, Elektor electronics, Nr. 1, (2007), pp. 14-18.
[3] https://en.nanotec.com/fileadmin/files/Datenblaetter/Schrittmotoren/ST4118/M/ST4118M1206-A.pdf.

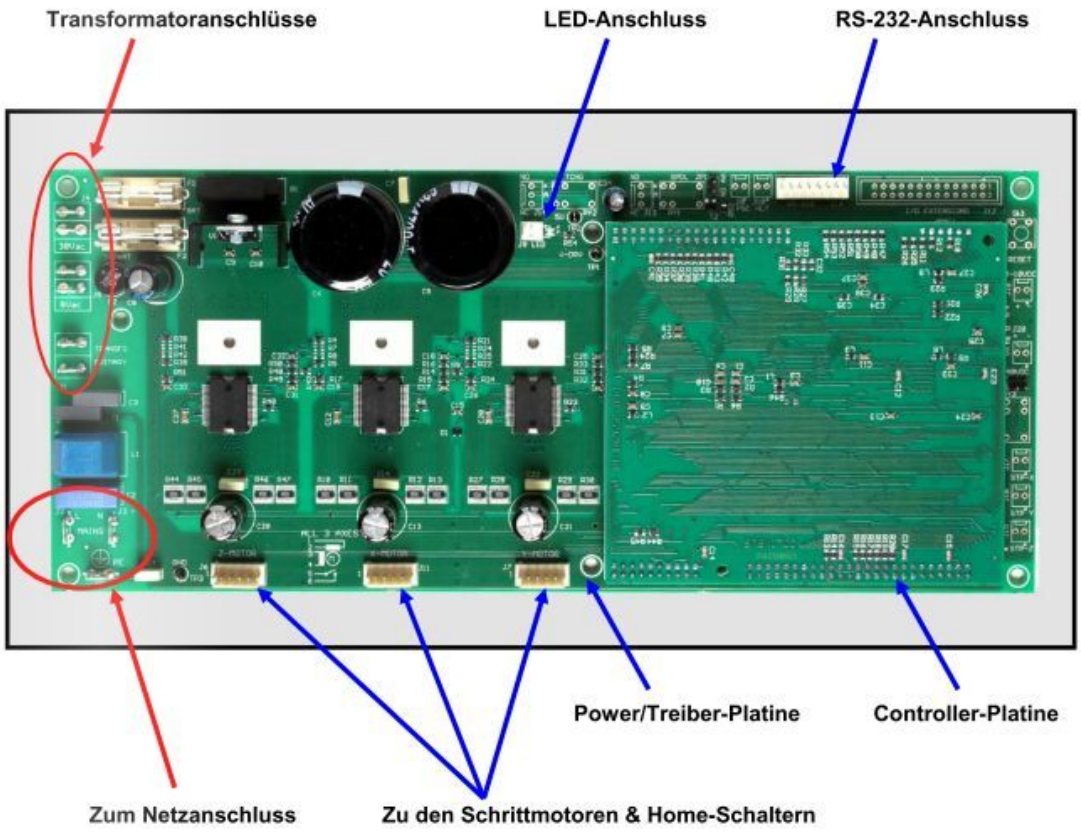


Figure 3: Main controller board.

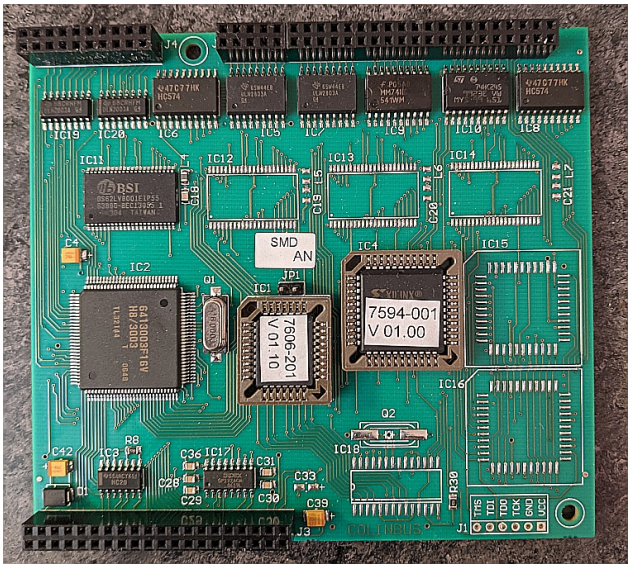


Figure 4: Microcontroller unit.

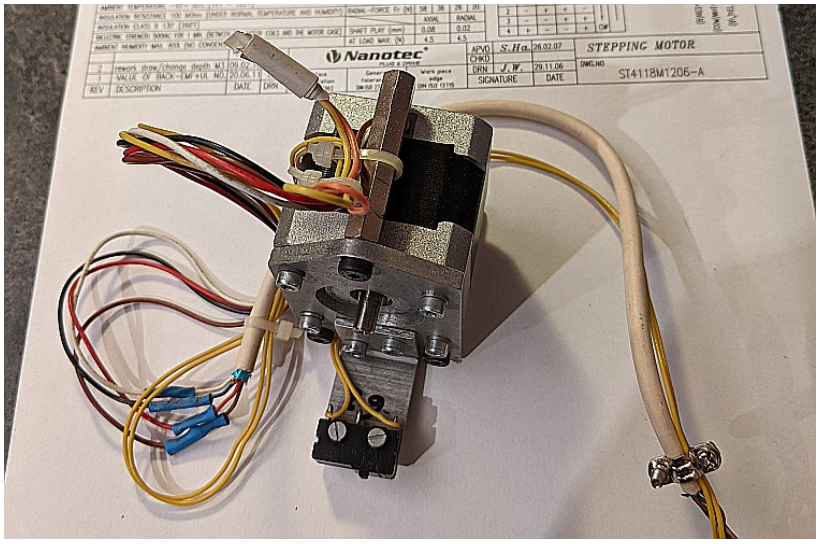


Figure 5: Stepper motor.

Figure 6: Actual state of the machine.

The whole machine has been partly disassembled in order to access the rods, rollers and spindle screws for maintenance (cf. Fig. 6). With the machine came a container with accessoires, cutters, drills, etc. (cf. Fig. 7).

Figure 7: Accessoires.

II Day 0

The author thanks his colleague Francis Massen for the opportunity of playing around with this nice device, and for the efforts in storing the heavy stuff in the car, ... and -last but not least- for the hours of scrutinizing the huge arsenal of material inherited from late Mr. Jean Mootz in order to find a copy of the ColiDrive software.

III Day 1

09/02/2021
The main problem with this machine appears to be the fact that it uses a special command system that was specific for the Colinbus products. However, this company has ceased to exist, and neither protocoles, firmware, nor any other profound descriptions are available anymore. The device was driven with the original ColiDrive software, ... again, a unique software package that has no documentation so ever about any internals. Even the baud rate for the UART is not specified. Comments on the Internet criticized this already in an early state of selling. It seems that a couple of alternative microntroller boards do exist, but none is available anymore. DIY-people out there have designed control boards that totally replace both the main unit and the original microcontroller board. Although the microcontroller board looks very straightforward, the amount of work needed to hack, disassemble and analyze the original firmware is not worthwhile, regarded the fact that there is one nasty limitation of the program, which is the impossibility to drive all three stepper motors at the same time. So, it can only drive two synchronously. In order to overcome all this, the author thinks of replacing the microcontroller unit with a DIY device while keeping the main board and the power supply untouched.
The conclusion is to reverse engineer the main unit, in order to understand:

1. how the stepper motors are controlled, and
2. which control pins are wired to the microntroller board.

A L6208 DMOS driver for bipolar stepper motor

The datasheet defines this IC as:

> The L6208 device is a DMOS fully integrated stepper motor driver with non-dissipative overcurrent protection, realized in BCD technology, which combines isolated DMOS power transistors with CMOS and bipolar circuits on the same chip. The device includes all the circuitry needed to drive a two phase bipolar stepper motor including: a dual DMOS full bridge, the constant off time PWM current controller that performs the chopping regulation and the phase sequence generator, that generates the stepping sequence. Available in PowerSO36 and SO24 (20 + 2 + 2) packages, the L6208 device features a non-dissipative overcurrent protection on the high-side power MOSFETs and thermal shutdown.

The L6208 is used for:

- decoding logic for stepper motor full and half step drive;
- cross conduction protection;
- thermal shutdown;
- undervoltage lockout.
- The IC has integrated fast freewheeling diodes;
- operating supply voltage ranges from 8 to 52 V;
- it can drive 5.6 A output peak current (2.8 A RMS);
- provides operating frequency up to 100 KHz;
- has non-dissipative overcurrent protection;
- dual independent constant t_{OFF} PWM current controllers;
- fast/slow decay mode selection;
- fast decay quasi-synchronous rectification

B Reverse engineering the main board

The hack of the main board and the stepper motor (cf. Fig. 8) reveals that the L6208 is used to control the motor in bipolar/1-winding mode. The IC is configured as recommended in its datasheet. Especially large copper pads on the main board are used for heat dissipation.

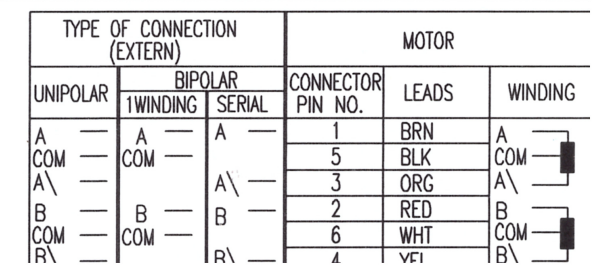


Figure 8: Motor wires.

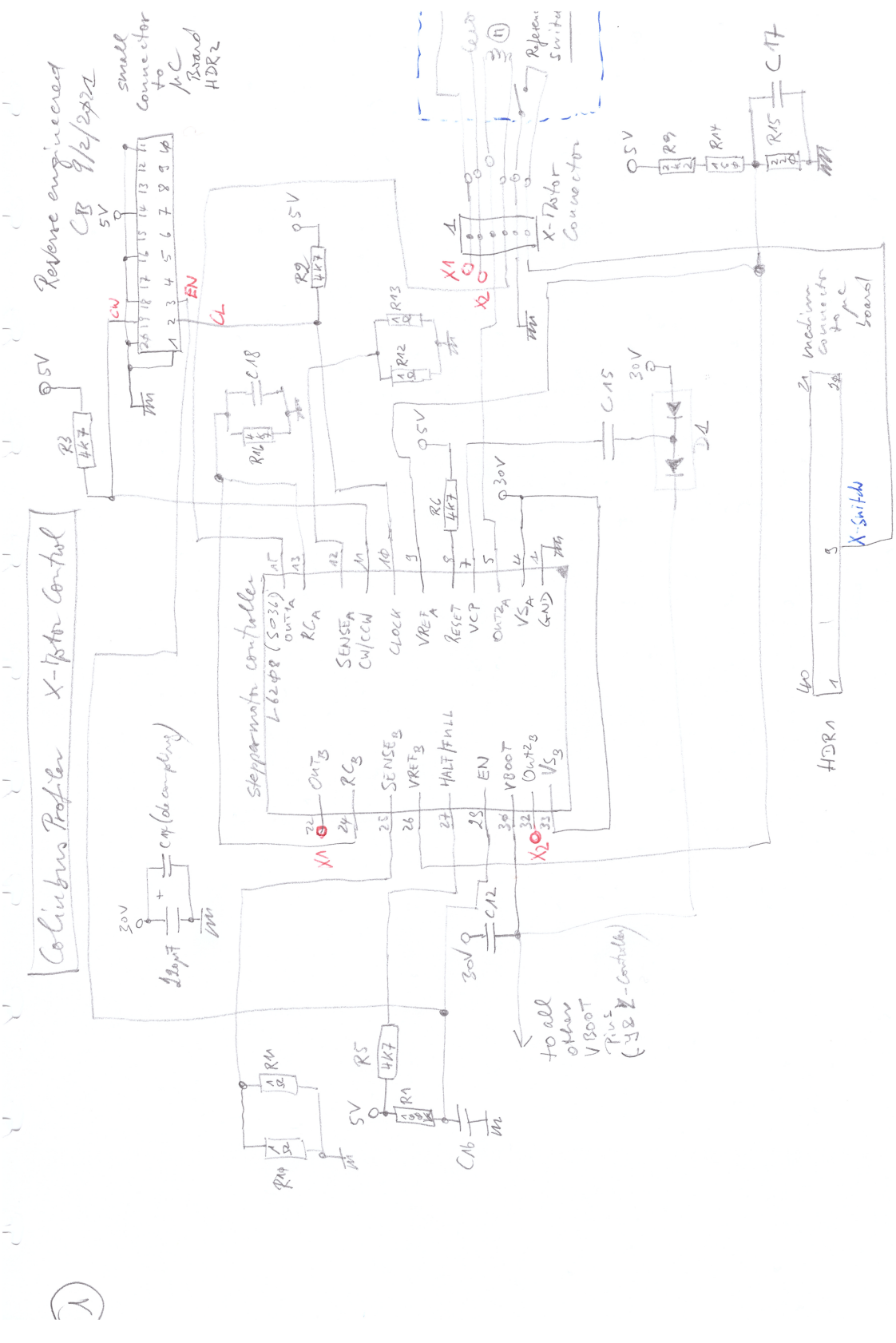


Figure 9: Circuit schematics of the x-axis control system.

Figure 10: μC board connections.

IV Day 2

10/02/2021
A brainstorming led to the following short term action plan (order will be chosen on the run):

- analyze the L6280 configuration
- attention: RX/TX labeling must be exchanged on the μC-board (PC-RX becomes Remote-TX...)
- learn stepper motor acceleration and deceleration features
- design a simple test device that drives a single motor
- get the motor ratings of the Profiler (with and without load at various torques; max speed; speed control)
- find the conversions equation that maps motor steps to distance travelled of the spindles on each axis.
- choose a valuable μC on the base of a list of criteria
- How do we detect that motors don't move, although they should (stalled situation, for instance)?

The μC should have the following features:

- UART (TX and RX)
- 1 digital input for Emergency stop (abort)
- 3 x 3 DI for x, y, z-motor control lines
- 3 DI for x, y, z-switches
- 1 DO for external milling on/off control
- optional: 1 PWM output for spindle speed control
- 1 DI for pausing (eventually later camera or sound sensor for tool break detection or missing motion)
- 1 DI/O (clock) + 1 birectional DI/O (data) for I_2C connection for future use
- reserve digital and analog pins for future use
- enough program and data space

This leads us to the Microchip PIC18F452, which has all of these features. For this IC, we can rely on the PICLab software package, written in 2006 by former LCD student Laurent Kneip (who b.t.w. now is Associate Professor at Shanghai University -School of Information Science and Technology). This almost unknown software for the rest of the world is based on the graphical ROBOLAB environment, which was developed at Tufts University for the LEGO Mindstorms project. PICLab allows easy programming of a list of Microchip PICs, generating an easy to read and to control MPASM-Assembler code. For some unkown reason the 18F452 (and also the 16F628) are no longer programmable with the special programming device that went with the PICLab software. The 18F452 will therefore be burned with the Microchip original MPLAB ICD2, an easy to use device.
Need to know: When running the 18F452, MCLR must be pulled up to VDD. Otherwise the floating pin will make the behaviour of the μC unpredictable.
It was remarkably easy to program a dimming LED (cf. Fig.11). Note that the PIC is programmed in situ.

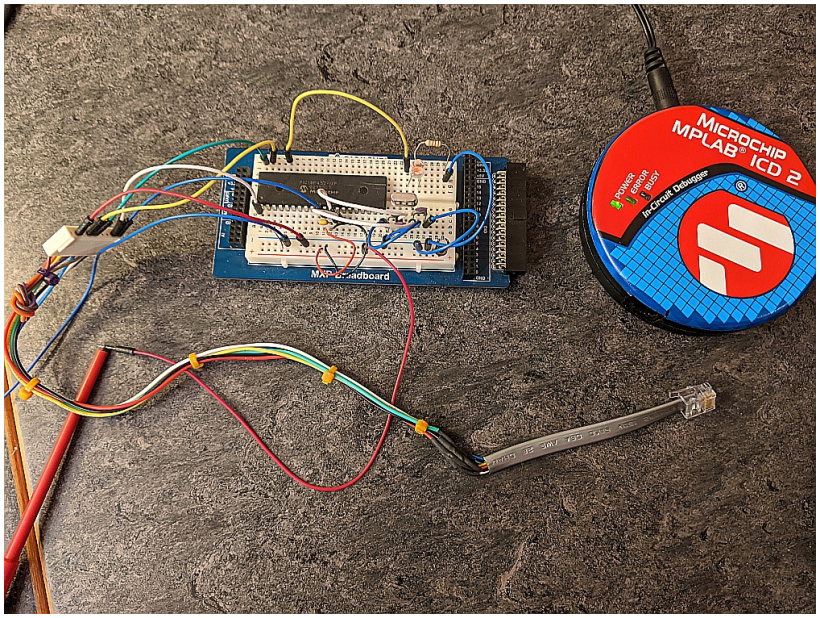


Figure 11: First test with the 18F452.

V Day 3

11/02/2021

A Errata

1. On Fig. 9 the *CONTROL*-pin is missing.
2. R16 is 47k instead of 4k7.
3. V_S is DC 21V (=30 x $\sqrt{2}/2$)

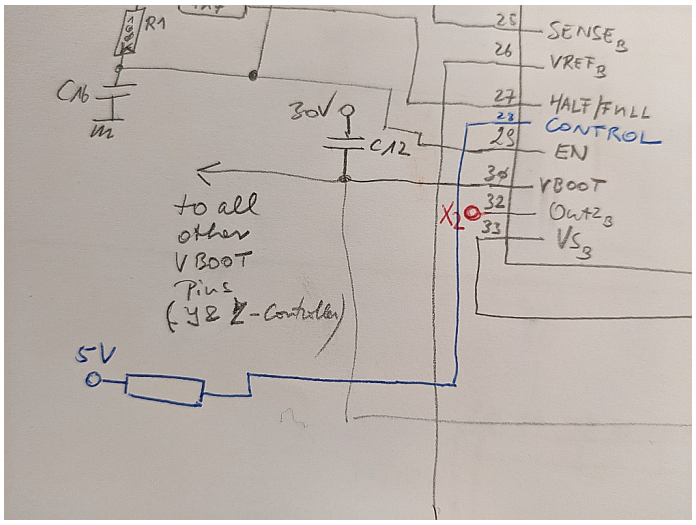


Figure 12: The *CONTROL*-pin is also pulled-up to 5V via resistor (R4=4k7).

B L6208 Configuration

From the completed diagram, we learn the following details of the L6208 functionality on the main board:

- The smd-capacitors don't show any value, so exact timing cannot be calculated in the L6208 equations (cf. datasheet). However, from the neat design of the board, we may conclude that the main board controls the stepper motors in an optimal way referred to the datasheet and the application notes.[4]
- *RESET* is pulled-up to 5V, with no connection to the μC board, which means that this pin isn't used.
- *CONTROL* also is hard pulled-up to 5V; no connection to the μC board. This is the decay mode selector. As configured, slow decay mode is set. According to the application notes, this allows lower power dissipation, lower ripple and avoids voltages below GND at output pins during recirculation. The L6208 has a built-in PWM that switches off and on the H-bridge in order to limit the current to the allowed amount, given by the $SENSE_{A\&B}$ pins.
- the chosen $SENSE_{A\&B}$ resistors (in parallel 0.5Ω, fro instance) correspond to the desired I_{max} current 1A.
- according to the equation $V_{REF} = I_{Motor} \times R_{Sense} = 1 \times 0.5 = 0.5V$, which can be verified by calculating the voltage at C17 (R9, R14, R15 network): $V_{REF} = 5 \times 220/(2200 + 150) \approx 0.5V$.
- *HALF/FULL* also is hard pulled-up to 5V; no connection to the μC board. This configures the IC in half step mode. So, we get a motor step resolution of 0.9° instead of 1.8°.
- the *CLOCK* pin reacts on rising edge. One pulse should correspond to one step of the motor.
- LOW logic on *EN* shuts down the H-bridge. (This pin is internally used by the overcurrent and thermal protection system.)
- *CW/CCW* HIGH logic level sets clockwise direction, whereas LOW logic level sets counterclockwise direction.

C First motor control test

The idea is to slowly flash a LED by timer interrupt and connect the PIC output pin to the *CLOCK* pin. This works without any problem. Behind the slow step from the PIC, you hear some high frequency switching that evidently comes from the L6208. Manually grounding the *EN* pin on the main board disables the motor. Grounding the *CW/CCW* pin changes the motor direction. This works for all three axis.

```
;===============================================================================
;                  Assembler code for isr.vi created with PICLab
;===============================================================================

        LIST    p=18F452

        #include "P18F452.INC" ; Include header file

        __CONFIG        _CONFIG1H, _OSCS_OFF_1H & _HS_OSC_1H
        __CONFIG        _CONFIG2L, _BOR_OFF_2L & _PWRT_ON_2L
        __CONFIG        _CONFIG2H, _WDT_OFF_2H
        __CONFIG        _CONFIG3H, _CCP2MX_ON_3H
        __CONFIG        _CONFIG4L, _STVR_OFF_4L & _LVP_OFF_4L & _DEBUG_OFF_4L
        __CONFIG        _CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
        __CONFIG        _CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H
        __CONFIG        _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
        __CONFIG        _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H
        __CONFIG        _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L
        __CONFIG        _CONFIG7H, _EBTRB_OFF_7H

;***********************************Variable definitions*****************************************

TEMPPORT8 EQU 0X0
```

[4] L6208 Application Note - *AN1451*, STMicroelectronics, (2003), p. 30.

```
TEMPX8 EQU 0X1
TEMPY8 EQU 0X2
RESULT8 EQU 0X3
TEMPX16 EQU 0X4
TEMPX16_H EQU 0X5
TEMPY16 EQU 0X6
TEMPY16_H EQU 0X7
RESULT16 EQU 0X8
RESULT16_H EQU 0X9
IDX16 EQU 0XA
IDX16_H EQU 0XB
TEMPYY EQU 0XC
OP_SIGN8 EQU 0XD
ADHBYTE EQU 0XE
ADLBYTE EQU 0XF
AD_RESULT EQU 0X10
AD_RESULT_H EQU 0X11
TEMPP1 EQU 0X12
TEMPP2 EQU 0X13
TEMPP3 EQU 0X14
ISR_TEMPPORT8 EQU 0X15
ISR_TEMPX8 EQU 0X16
ISR_TEMPY8 EQU 0X17
ISR_RESULT8 EQU 0X18
ISR_TEMPX16 EQU 0X19
ISR_TEMPX16_H EQU 0X1A
ISR_TEMPY16 EQU 0X1B
ISR_TEMPY16_H EQU 0X1C
ISR_RESULT16 EQU 0X1D
ISR_RESULT16_H EQU 0X1E
ISR_IDX16 EQU 0X1F
ISR_IDX16_H EQU 0X20
ISR_TEMPYY EQU 0X21
ISR_OP_SIGN8 EQU 0X22
ISR_STATUS EQU 0X23
ISR_W EQU 0X24
ISR_BSR EQU 0X25

;**************************Makro definitions and definitions of used operations**************************

BANK0   MACRO
        BCF BSR,0
        BCF BSR,1
        BCF BSR,2
        BCF BSR,3
        ENDM
BANK1   MACRO
        BSF BSR,0
        BCF BSR,1
        BCF BSR,2
        BCF BSR,3
        ENDM
BANK2   MACRO
        BCF BSR,0
        BSF BSR,1
        BCF BSR,2
        BCF BSR,3
        ENDM
BANK3   MACRO
        BSF BSR,0
        BSF BSR,1
        BCF BSR,2
        BCF BSR,3
        ENDM
BANK4   MACRO
        BCF BSR,0
        BCF BSR,1
        BSF BSR,2
        BCF BSR,3
        ENDM
BANK5   MACRO
        BSF BSR,0
        BCF BSR,1
        BSF BSR,2
        BCF BSR,3
        ENDM
BANK15  MACRO
        BSF BSR,0
        BSF BSR,1
        BSF BSR,2
        BSF BSR,3
```

```
        ENDM

                GOTO   START
                ORG 0X8
                GOTO LABEL_ISR

;*************************************BEGIN OF MAIN PROGRAM*************************************

START

   ;INITIALIZE PORT A AND E
   CLRF PORTA,0
   CLRF LATA,0
   CLRF PORTE,0
   CLRF LATE,0
   MOVLW 0X06
   MOVWF ADCON1,0
   MOVLW 0X0
   MOVWF TRISA,0
   MOVWF TRISE,0

   ;INITIALIZE PORT B
   CLRF    PORTB,0
   CLRF LATB,0
   MOVWF   TRISB,0

   ;INITIALIZE PORT C
   CLRF PORTC,0
   CLRF LATC,0
   MOVWF TRISC,0

   ;INITIALIZE PORT D
   CLRF PORTD,0
   CLRF LATD,0
   MOVWF TRISD,0

   ;CONFIGURE SINGLE PIN
   BCF TRISB,2,0

   ;CONFIGURE SINGLE PIN
   BCF TRISB,3,0

   ;CONFIGURE TMR1
   MOVLW    B'110001'
   MOVWF    T1CON,0

   ;START MONITORING INTERRUPTS
   BCF PIR1,TMR1IF,0
   BSF PIE1,TMR1IE,0
   BSF   INTCON,GIE,0
   BSF   INTCON,PEIE,0

LABEL_0

   GOTO LABEL_0

LABEL_1004

   GOTO LABEL_1004

;*********************************INTERRUPT SERVICE ROUTINE*********************************

LABEL_ISR
   MOVWF ISR_W,0
   SWAPF STATUS,W,0
   MOVWF ISR_STATUS,0
   MOVF BSR,W,0
   MOVWF ISR_BSR,0
   BANK0
   MOVF TEMPPORT8,W
   MOVWF ISR_TEMPPORT8
   BTFSS PIE1,TMR1IE,0
   GOTO LABEL_1005
   BTFSS PIR1,TMR1IF,0
   GOTO LABEL_1005
   CALL LABEL_EVENT0
   BCF PIR1,TMR1IF,0
LABEL_1005
   BANK0
   MOVF    ISR_TEMPPORT8,W
   MOVWF   TEMPPORT8
```

```
   MOVF  ISR_BSR,W
   MOVWF BSR,0
   SWAPF     ISR_STATUS,W,0
   MOVWF   STATUS,0
   SWAPF     ISR_W,F,0
   SWAPF     ISR_W,W,0
   RETFIE

;*************************************EVENT-ROUTINES*************************************

LABEL_EVENT0

   ;BEGIN OF IF-STRUCTURE (DEPENDING ON BIT/PIN)
   BTFSS PORTB,2,0
   GOTO LABEL_1002

                     ;SET SINGLE OUPUT PIN
                     BCF PORTB,2,0

   GOTO LABEL_1003
LABEL_1002

                     ;SET SINGLE OUPUT PIN
                     BSF PORTB,2,0

LABEL_1003
;END OF IF-STRUCTURE

   RETURN

   ;

   END
```

D PIC 18F452 test

In order to learn more from the 18F452, a test program was designed to flash a LED by timer interrupt as in the example before, and to dim a second LED by normal program routine. This also perfectly works.

VI Day 4

12/02/2021

A PICLab Timer0 issues

That's the good thing about PICLab!!! You get MPASM Assembler code as an output, where you easily can fix bugs. So, when switching to the 18F452, the MICROCHIP datasheet talk about compatibility of the periphery devices. That's true insofar that configuration registers and bits use the same names from type to type. However, addresses do not correpsond. In the case of Timer0, this device is 8bit on older types, and on the 18F452, you can make it 16bit. However, the default and boot configuration is 16bit. So, you must SET a configuration bit, in order to stay compatible with 8bit, which unfortunately PICLab is only able to handle for this Timer. A second issue was that the author Laurent Kneip forgot to switch the timer on at configuration. These two issues were rapidly fixed by manually adding b'11000000' to the T0CON register. Normally this kind of mistakes cannot be fixed in compilers by end users.

B Steppermotor speed ratings - zero load

In order to test the speed ratings of the ST4118M1206 stepper motor controlled by the L6208, a most simple setup was made, as can be seen on Fig. 18. An ordinary square LEGO piece exactly fits on the motor shaft. On top, a 24-gear was clamped using the 4 holes, and an LEGO TECHNIC axis was fixed to the gear together with the LEGO legacy rotation sensor.

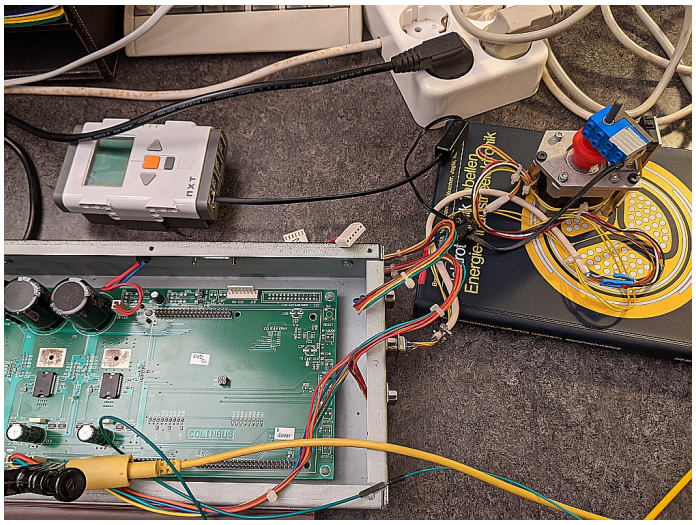


Figure 13: Rotation measurement using the LEGO NXT.

The LEGO NXT was programmed to count rotation pulses for the duration of 30 seconds. Note that the legacy rotation sensor has a resolution of 16 ticks per revolution, which isn't famous at all, but sufficient for the purpose. The rotation sensor is known to loose about 1/3000 ticks due to the sensor reading update rate of the LEGO older devices. This evidently appears at higher rotation speeds.[5] So, we therefore must expect some variations of the results, which could interfere with step losses of the steppermotor. Note that the sensor has bad measurements at very low speeds.
The 18F452 was programmed to drive one of the *CLOCK* lines of the main Profiler board with various pulse rates, in order to demonstrate the behaviour of the unloaded steppermotor.

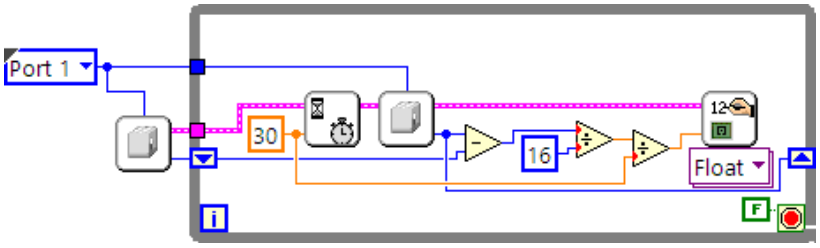


Figure 14: Simple NXT program in LabVIEW for LEGO Mindstorms.

Pulse frequency [Hz]	**NXT-measured speed R[rps]**	**From measured, computed ticks per sec.=N[1/s]**	**$\mid \Delta \mid$**
1.9084	0.0048*	1.92	0.0116
7.63	0.019	7.6	0.03
244	0.61	244	0
976	2.442	976.8	0.8
1960.8	4.883	1953.2	7.6
2994**	7.442	2976.8	17.2

Table 1: Experiment data.

[5] cf. www.philohome.com/sensors/legorot.htm

* NXT measurements were not precise enough here due to the slow motion issue with the legacy rotation sensor. So, this experiment was repeated manually using a timer.
** The experiment failed beyond 4kHz, because the steppermotor didn't move anymore. So, 3kHz should be considered as the maximum noload step frequency.
Note: $N = R \times 400$ (0.9° resolution of the steppermotor)-cf. Table1.

C Feed spindle motion

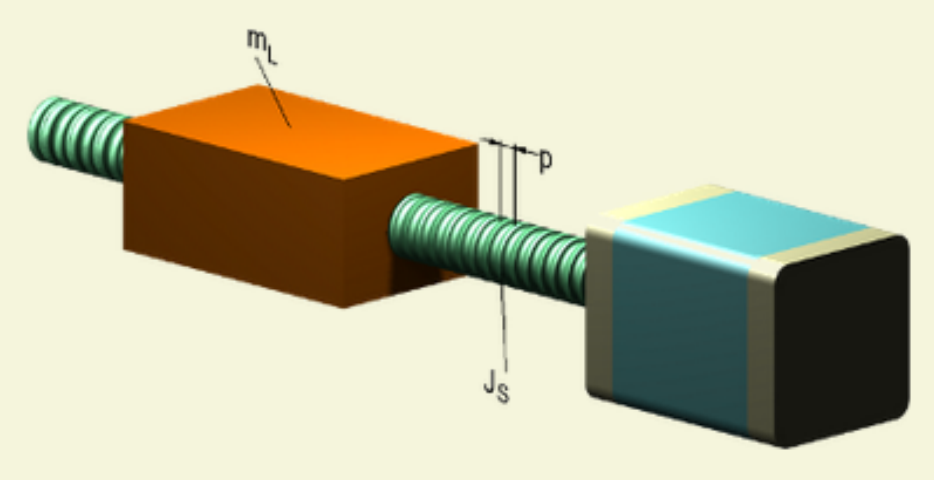


Figure 15: Feed spindle with nut and motor..

The Profiler spindle was measured: 10 revolutions exactly correspond to 3cm distance of the plastic nut. So, with these measurements, we can conclude that the unloaded Profiler could produce a maximum feed rate of approx. 1350[mm/min], which corresponds to 7.5×60[rpm] at noload.
The feed spindle is a single start screw with identical lead and pitch = 3mm. The feed spindle system has various physical variables to consider (cf. Fig. 15):[6]

m_L = load mass

p = lead (pitch) =3mm

J_s = moment of inertia of the spindle.

For the base feed spindle, we can estimate J_s, if we consider the mean radius $r = 5$mm, length $L_s = 0.49$m, mass $m_s = 0.36$kg:

$$J_s = \frac{m_s \times r^2}{2} = \frac{0.36 \times 25 \times 10^{-6}}{2} = 4.5 \times 10^{-6}\text{kgm}^2 \quad (1)$$

We also must add the rotor inertia, which in our case $= 5.7 \times 10^{-6}\text{kgm}^2$, according to the datasheet.
The Nylon nut has no backlash. In fact, it strongly embraces the spindle. The torque needed to overcome the static friction is 0.0266Nm. (Measurement conditions: spindle screw cleansed and lubricated with finest sewing machine oil, temperature = 20°C; $F = 228g \times g$[N]; radius of application=$1.19 \times 10^{-2}[m]$.)

[6] https://www.schweizer-fn.de/antrieb/kupplung/kupplung.php#spindelantrieb

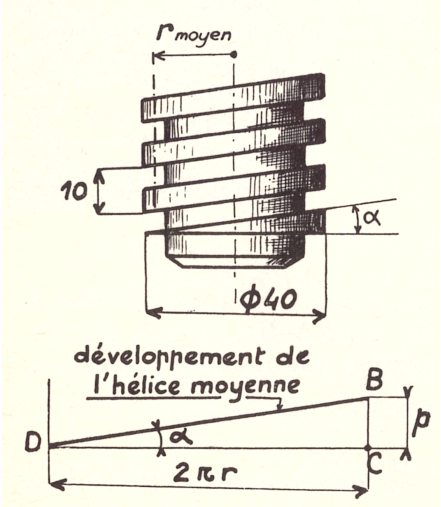


Figure 16: Screw example.

Fig. 16 shows a sketch of a screw.[7] The transmission of the rotational torque T and its conversion into the translational force Q obeys the following equation:[8]

$$T = Q \cdot r \cdot \tan(\alpha + \varphi) \tag{2}$$

$$\tan\alpha = \frac{p}{2\pi r} \tag{3}$$

where tanφ= friction coefficient.

We can roughly estimate this parameter through a simple experiment, by adding a sensitive dynamometer to the nut, and verify, which translational force is generated by the torque (that we determined experimentally to $T_0 = 0.0266$Nm). For the puropose we used the VERNIER Handheld Dynamometer that we are able to interface with the LEGO NXT.
From the data known so far, we have:

T_0=0.0266Nm

$r = 5 \times 10^{-3}$m

tan$\alpha = \frac{3}{2\pi \cdot 5} = 0.09549$, so

$\alpha = 5.4°$

Q=5.8N (measured)

$\Longrightarrow$ $\mu_{estimated} = \tan\varphi = 0.7$ and $\varphi_{estimated} = 35°$

Note that in the calculations we used the identity:

$$\tan(\alpha + \varphi) = \frac{\tan\alpha + \tan\varphi}{1 - \tan\alpha \cdot \tan\varphi} \tag{4}$$

This is quite a high value. If we consider technical tables, where coefficients are estimated to around 0.12 for lubricated nylon on steel, we must conclude that our measured value should in any case be readjusted. We therefore propose to choose the average $\mu_{adjusted}$=(worst case + best case)/2= 0.4. Note however that the high measured value matches with observation that the conversion of rotational into translational motion

[7]Picture taken from: R. Basquin, *Mcanique, Deuxime Partie*, Librairie Delagrave, Paris, (1971), p. 470.
[8]ibid, p.474.

using a screw thread is irreversible due to the strong holding friction. The condition for irreversibility is that $\alpha < \varphi$.

$\mu_{adjusted}$=0.4

$\varphi_{adjusted} = 21°$

The important conclusion of all this is that the trade-off for zero backlash is high static friction, which has to be overcome by the motor torque. The advantage of the holding force is that during the milling, routing or cutting process of the machine, shear forces are sufficiently compensated. An inconvenience is that the feed speed may not reach higher values, and that during acceleration and deceleration phases, the clock rate must be ramped up or decayed, in order to manage the compensation of static friction augmented with inertia effects. This is what we are going to examine next.

VII Day 5

15/02/2021

A Evaluating the load features of the motor

The obvious advantage of stepper motors is its open-loop control. No feedback is necessary of motor position. Each *CLOCK* pulse at the controller IC rotates the motor by a single step, or in the case of the actual configuration, a half step by 0.9°. However this is only true insofar as the motor is driven within certain limits. Two parameters play an essential role here, the load and the clock rate. The motor cannot handle too heavy loads, although the holding torque is impressively high (=0.28[Nm], according to the datasheet), the detent torque is very weak (= 9.8×10^{-3}[Nm]). The good news is that the L6208 controller uses slow decay mode, so the motor coils are powered a longer times, which positively affects the torque. Note that the use of the half-step method to control the motor reduces the torque. This is the trade-off for gaining precision.
The following setup was used to test the motor features (cf. Fig. 17).

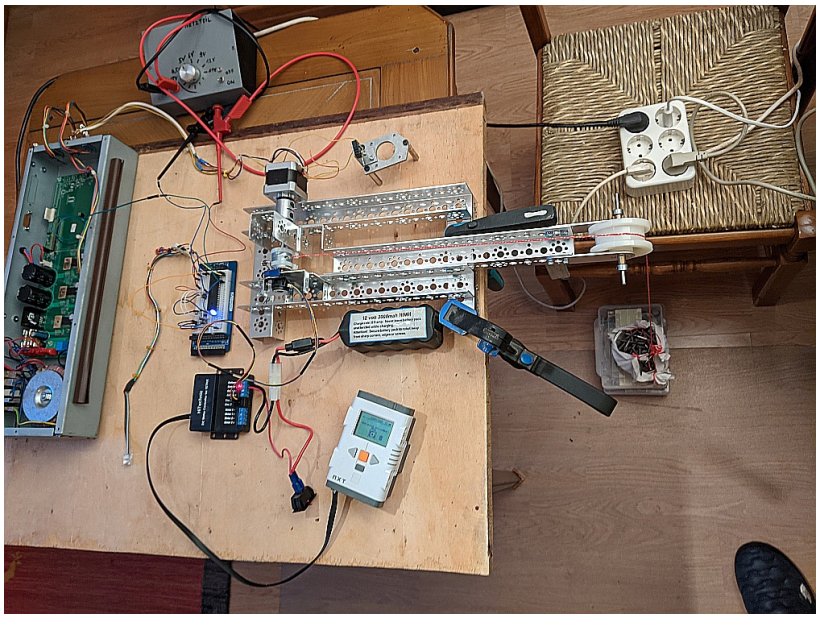


Figure 17: Test setup for measuring motor torque.

The device is built with the TETRIX robotics system, sold by PICSO.[9]. This system has a HITECHNIC interface to the LEGO NXT. It also uses USDIGITAL-E4P miniature encoders.[10] This encoder has high resolution of $4 \times 360 = 1440$ steps per revolution.
The idea is to have the motor mount a variable charge $P = g \cdot m_L$ over a poulley, while the motor is driven at different clock rates f_c.

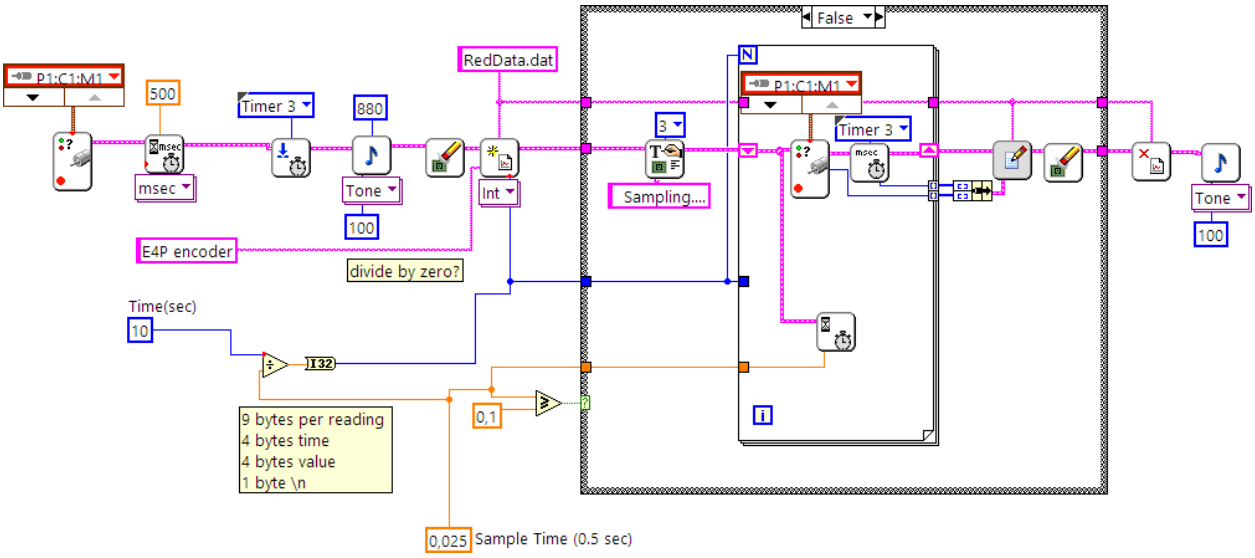


Figure 18: NXT program (LABVIEW for LEGO MINDSTORMS software.

m_L [kg]	T=motion time [s]	N=encoder [ticks]	motor steps $= \frac{N \cdot 400}{1440}$	step rate [1/s]	produced torque [Nm] $= m_L \cdot g \cdot r$
0.1	9.791	7776	2160	220	0.017
0.2	10.323	8170	2269.44	219.84	0.023
0.4	10.799	8531	2369.7	219.4	0.047
0.6	10.491	8259	2294.17	218.67	0.07
0.8**	6.44	4573	1270.28	197***	0.09****

Table 2: First experiment: Clock-rate 222Hz* (which would correspond to a feed speed of 100mm/min).

* The 4MHz crystal used in the PIC 18F452 circuit only produces 3.94MHz. This explains the difference with the step rate in the stable experiments.

** The weight fell down after a short traction time.

*** Obviously steps are lost with this heavy weight.

**** Since the torque produced by a stepper motor is not constant, but has low torque detent phases between the holding phases, as soon as the freefall of the weight is initiated, the motor torque is not strong enough to stop the process.

[9]https://www.pitsco.com/Shop/TETRIX-Robotics/
[10]https://cdn.usdigital.com/assets/datasheets/E4P_datasheet.pdf?k=637488676231252819

m_L [kg]	T=motion time [s]	N=encoder [ticks]	motor steps $= \frac{N \cdot 400}{1440}$	step rate [1/s]	produced torque [Nm] $= m_L \cdot g \cdot r$
0.6	6.132	6.132	2711.39	442	0.07
0.8^{+}	-	-	-	-	-

Table 3: Second experiment: Clock-rate 444Hz (200mm/min).

$^{+}$ The motor couldn't pull that weight at all.

m_L [kg]	T=motion time [s]	N=encoder [ticks]	motor steps $= \frac{N \cdot 400}{1440}$	step rate [1/s]	produced torque [Nm] $= m_L \cdot g \cdot r$
$0.6^{\#}$	3.332	7537	2093.61	628.33	0.07

Table 4: Third experiment: Clock-rate 666Hz (300mm/min).

$^{\#}$ Although the process was stable, a few steps have been lost.

Conclusion:

These experiments demonstrate that the stepper motors should only be reasonably used up to 300-400mm/min. Higher feed speeds could possibly be used, because the screw thread irreversibility prevents that shear forces are led to the motor. However, higher torque rates that would be necessary to deal with those forces at higher speeds while feeding forward will inevitably produce step losses. The operator must be careful here. However, the machine will not be seriously controlled without a ramping up method during the acceleration and decaying during the deceeration phases. We therefore must study the effects of inertia (translation) and moment of inertia (rotation) during the acceleration and deceleration process, in order to determine the duration of such phases.

VIII Day 6

17/02/2021

Evaluating acceleration time

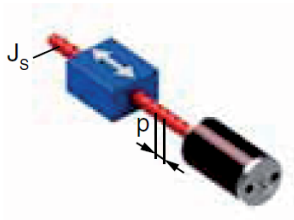
Fig. 19 depicts the way to calculate additional torque that is needed during acceleration time in order to keep acceleration constant.

Cutting force estimation: By experience with wood routing, we can estimate that cutting forces at high routing speeds ($>$ 20000rpm) with sharp cutters and slow feed speed ($<$ 400mm/s) will never exceed **50N**. Thus, we can calculate the required motor torque by using Eq. 2:

$$T = 50 \times 5 \times 10^{-3} \times \tan(5.4 + 21) = 0.12Nm \tag{5}$$

Maximum motor torque: We can estimate the maximum useable motor torque by considering that the holding torque and detent torque are delivered each during half of the time of a clock period. Thus, the motor is able to deliver maximal 0.28/2=**0.14Nm**, sufficient to handle stronger cutting forces, however only at slow step rates. It is up to the operator to choose the correct feed and cutting speed, in order to minimize the cutting forces and keep the required torque within reasonable bounds, so that (1) no steps are being lost, (2) the cutting result isn't spoiled, (3) tools don't break, and (4) everything is safe for the operator.
We could see so far that step losses do appear at torques $> 0.07 - 0.09$Nm, depending on the clock rate. From these considerations, we now arbitrarily fix the maximally allowed additional torque during acceleration phases at **0.03Nm**.

Spindelantrieb



Drehzahl	$n_{in} = \frac{60}{p} \cdot v_L$
Drehmoment	$M_{in} = \frac{p}{2\pi} \cdot \frac{F_L}{\eta}$
zusätzliches Drehmoment für konstante Beschleunigung (Drehzahländerung Δn_{in} während der Dauer Δt_a)	$M_{in,\alpha} = \left(J_{in} + J_S + \frac{m_L + m_S}{\eta} \cdot \frac{p^2}{4\pi^2} \right) \cdot \frac{\pi}{30} \cdot \frac{\Delta n_{in}}{\Delta t_a}$
Spiel, Positionsfehler	$\Delta\varphi_{in} = \Delta s_L \cdot \frac{2\pi}{p}$

Symbol	Name	SI
F_L	Lastkraft	N
J_{in}	Massenträgheitsmoment Eingang (Motor, Encoder, Bremse)	kgm²
J_S	Massenträgheitsmoment Spindel	kgm²
J_X	Massenträgheitsmoment Umlenkrolle X	kgm²
J_1	Massenträgheitsmoment antriebsseitig	kgm²
J_2	Massenträgheitsmoment Umlenkrolle 2	kgm²
M_{in}	Drehmoment Eingang	Nm
$M_{in,\alpha}$	Drehmoment für Beschleunigung	Nm
d_X	Durchmesser Umlenkrolle X	m
d_1	Durchmesser Antriebsrolle	m
d_2	Durchmesser Umlenkrolle 2	m
m_B	Masse Band	kg

Symbol	Name	SI
m_L	Masse Last	kg
m_S	Masse Spindel	kg
p	Spindelsteigung	m
v_L	Lastgeschwindigkeit	m/s
Δs_L	Mechanisches Spiel Abgang	m
Δt_a	Beschleunigungszeit	s
$\Delta\varphi_{in}$	Mechanisches Spiel Eingang	rad
η	Wirkungsgrad	

Symbol	Name	maxon
n_{in}	Drehzahl Eingang	min⁻¹
Δn_{in}	Drehzahländerung Eingang	min⁻¹

Figure 19: Formulae needed for the calculation of feed drive acceleration control. (Note that $\eta = \tan\alpha / \tan(\alpha + \varphi)$, and that the list here has a different use of the symbol α.
)

Reference: https://www.maxongroup.de/medias/sys_master/8819062800414.pdf?attachment=true.[11]

From Fig. 20 we may conclude that acceleration time must be at least 14ms, in order to keep torque below the limit of 0.03Nm in the case of a change of rotation speed by 300rpm. This seems to be an extremely short time. However, if we admit an abrupt speed change from 0 to 5rps (=300rpm), we'd risk a loss of $5 \times 0.014 \times 400 = 28$ steps!!! Thus, ramping up cannot be omitted.

[11] J. Braun, *Formelsammlung*, Maxon Academy, 2nd edition, (2012), p.22.

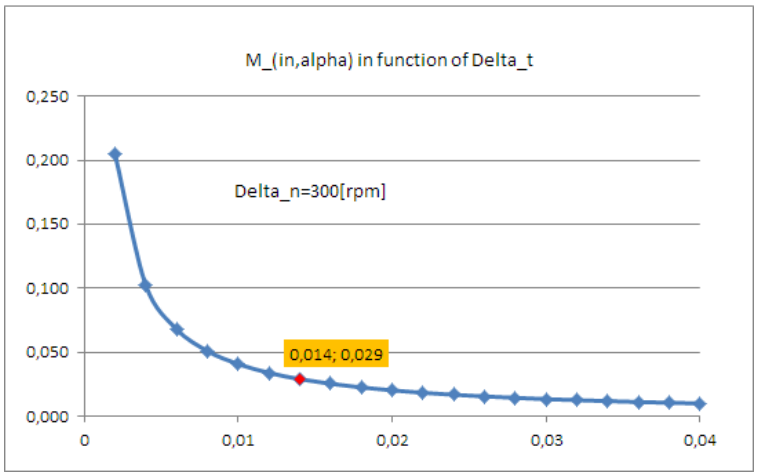


Figure 20: Torque at start needed to overcome inertia at maximal acceleration (reverse direction at 450mm/min); Δt =[s].

IX Day 7

18/02/2021

A Comparison of the experimental data with the Profiler technical specifications.

The technical specifications of the PROFILER published by the ELEKTOR MAGAZINE[12] display an optimistic positioning speed of $60mm/s = 3600mm/min$). This corresponds to a motor rotation speed of 60/3=20rps, or $20 \times 400 = 8000$ motor steps per second, which is far beyond the 4kHz, where we determined the unloaded motor limits.
ELEKTOR recommends to fix the maximum feed speed for the copper milling process (cricuit boards) to 10000micrometer/s. This corresponds to 600mm/min (=200rpm=3.33rps=1333.3 steps/sec). Although this feed speed might be too high for wood routing, it seems ok for circuit board making.[13]

B Stepper motor resonance

From plunging deeper into the world of stepper motors, we learned that typically between 100 and 200 steps per second, depending on many factors (motor characteristics, load, etc.), a resonance phenomen appears that prevents the motor from turning. One efficient method to reduce this effect is the use of half-step mode, which the L6208 provides for the PROFILER. We will have to investigate, whether the PROFILER shows up such effects, and this on each axis indivually.

C Variable representation

We must consider maximum values for step counters variables, in order to determine variable dimensions. The longest distance the PROFILER can handle is 400mm on the y-axis. This value corresponds to 400/3 =motor revolutions= $400 \times 400/3 = 53333$ single steps. This value is within the limits of U16 variables. One single step corresponds to a travelled distance = $3/400 = 0.0075mm$. ELEKTOR teaches us that the PROFILER firmware can only handle 0.025mm. This loss of precision by software is probably due to the originally used data size. As we will have to deal with trig. functions, we will probably have to switch to I32 representation, at least for certain data, in order to increase calculation precision. Although the 18F452

[12] cf.footnote 2
[13] *'Profiler' Tips & Tricks*, Elektor Magazine, Nr. 9, (2007), pp. 72-74.

is an 8-bit microcontroller, the data and program memories are genereous enough for implementing I32 primitives. Also, the μC may be clocked at 20MHz, which provides sufficient horsepower for doing the calculations in real-time. This definitely leads us to changing the programming environment, and start using the MPLAB C18 TOOLSUITE.

X Day 8

21/02/2021

Acceleration

State of art: Linear speed control of stepper motors with acceleration and decelration phases and its implementation on microcontrollers have been studied by ATMEL[14] on the base of the work by D. Austin.[15] Good reference also is the web site by book author D. W. Jones: http://www.cs.uiowa.edu/ jones/step/.

Using a LABVIEW simulation, we tried to visualize the difference in position and speed for the PROFILER's x,y axes. The exercise is to move the head from starting position $M_0(x_0, y_0) = (0, 0)$ to end position $M_e(x_e, y_e) = (100, 50)$; unit is [mm].
We admit the desired head speed vector with $\hat{v} = \|\overrightarrow{\mathrm{v_{desired}}}\|$=10[mm/s]. (Note that in wood routing, it is important for the quality of cutting, and the minimization of wear of the milling cutter to maintain a constant feed speed as much as possible, and this in any direction of motion.)
Drawing a line signifies driving both the x- and y-motors at speed ratios that are proportional to the tangent of the line angle.
Thus, for all v:

$$\begin{aligned} &\begin{cases} v_x = v \times \cos\psi \\ v_y = v \times \sin\psi \end{cases} \\ \text{with } \tan\psi &= \frac{y_e - y_0}{x_e - x_0} \end{aligned} \tag{6}$$

If $a = 4[mm/s^2]$ is the argument of the acceleration vector $\|\overrightarrow{\mathrm{a}}\|$, it follows:

$$\begin{cases} a_x = a \cdot \cos\psi \\ a_y = a \cdot \sin\psi \end{cases} \tag{7}$$

We now can calculate the acceleration time. Let $\hat{v}_x$ and $\hat{v}_y$ be the desired speeds that should have been reached after the acceleration times t_{ax} and t_{ay}.

$$\begin{cases} \hat{v}_x = a_x \cdot t_{ax} \\ \hat{v}_y = a_y \cdot t_{ay} \end{cases} \tag{8}$$

$$\implies t_{ax} = \frac{\hat{v}_x}{a_x} \text{ and } t_{ay} = \frac{\hat{v}_y}{a_y} \tag{9}$$

Because of Eq. 6 and Eq. 7, we get:

$$t_a \equiv t_{ax} = t_{ay} \tag{10}$$

This is true only insofar that $\cos\psi \neq 0$ and $\sin\psi \neq 0$. Because for a given angle this cannot happen at the same time, we only must take care for one of both conditions. Thus, if the cosine value is zero, this would mean that $v_x = 0$ and $a_x = 0$. In that case $v_y \neq 0$ and $a_y \neq 0$, unless $M_1 = M_0$, which wouldn't make any sense.

[14] Anonymous, *AVR446: Linear speed control of stepper motor*, ATMEL, (2006), available on http://ww1.microchip.com/downloads/en/appnotes/doc8017.pdf.
[15] D. Austin, *Generate stepper-motor speed profiles in real time*, in EE Times-India, (Jan. 2005), pp.1-5.

Note that deceleration time also is t_a.
Uniform motion time then is:

$$\begin{cases} x_e = x_0 + 2 \times \frac{1}{2} \cdot a_x \cdot t_a^2 + \hat{v}_x \cdot t_{\hat{v}x} \\ y_e = y_0 + 2 \times \frac{1}{2} \cdot a_y \cdot t_a^2 + \hat{v}_y \cdot t_{\hat{v}y} \end{cases}$$
$$\begin{cases} t_{\hat{v}x} = (x_e - x_0 - a_x \cdot t_a^2)/\hat{v}_x \\ t_{\hat{v}y} = (y_e - y_0 - a_y \cdot t_a^2)/\hat{v}_y \end{cases} \tag{11}$$

Because Eq. 6 and Eq. 7 and:

$$d \equiv \|\overrightarrow{\mathrm{M_0M_1}}\| = \sqrt{(x_e - x_0)^2 + (y_e - y_0)^2}$$
$$\begin{cases} x_e - x_0 &= d \cdot \cos\psi \\ y_e - y_0 &= d \cdot \sin\psi \end{cases} \tag{12}$$

we get

$$t_{\hat{v}} \equiv t_{\hat{v}x} = t_{\hat{v}y} \tag{13}$$

The LabVIEW simulation gives the result shown in Fig. 21 - 23. Due to the proportionality proven so far, the resulting line is perfectly straight.

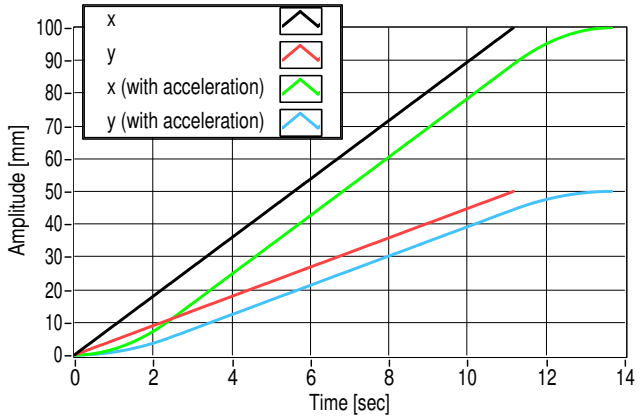


Figure 21: Position (data: $v = 10[mm/s]$ $a = 4[mm/s^2]$)

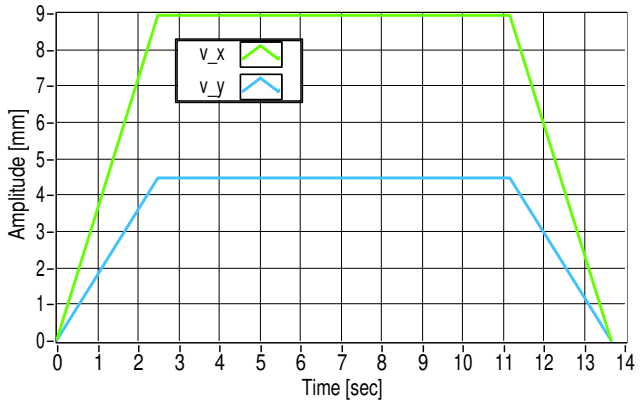


Figure 22: Speed patterns with uniform acceleration and deceleration phases.

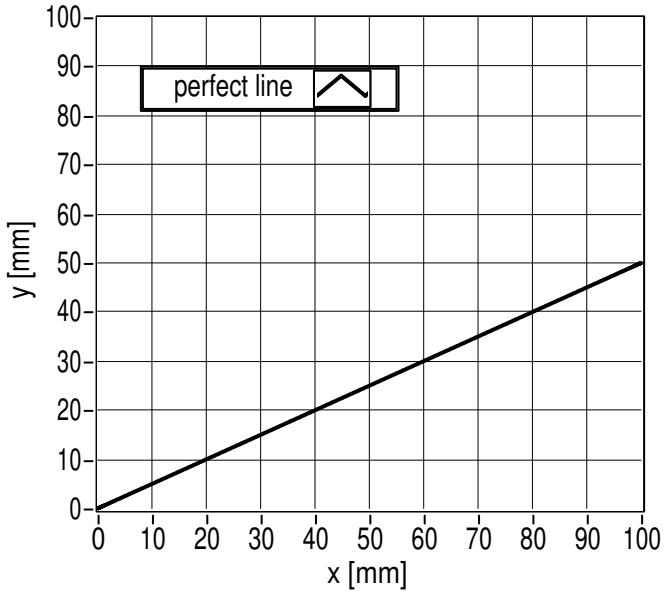


Figure 23: speed patterns with uniform acceleration and deceleration phases.

(Note on LabVIEW 2013: Using the invoke property *Export simplified picture to* ***.eps*** *format* on a graph doesn't work correctly in versions 2012 and 2013. You must open the **.eps** file with a text editor and replace everywhere the term *polyLine32* with *polyLine*. This is a known issue and workaround.)

(Note on the equations: for reasons of transcription of the math into low level C-Code for the PIC 18F452, we stick to algebraic notation instead of handy vector notation. However sometimes higher abstraction math may dissimulate much simpler equations behind otherwise elegant notations and short-cuts.)

XI Day 9

21/02/2021

Acceleration continued

We know from the mainboard configuration (cf. Section V.B) that the steppermotor has resolution $\alpha = 0.9°$. Thus one shaft revolution corresponds to 400 single steps. In other words, because one motor revolution advances the Profiler head on the specific axis by 3mm, the translational motion is made by steps of $\Delta s = 0.0075mm$ on each axis.
Now the question is, how the motor must be pulsed, in order to deliver uniform acceleration and deceleration during the start and end phases of the motion cycle.
If $M_a(x_a, y_a)$ is the point that the head has reached after the acceleration time, we may express the displacement as a function of time during the acceleration phase:

$$\begin{cases} x_a = x_0 + \frac{1}{2} \cdot a_x \cdot t_{ax}^2 \\ y_a = y_0 + \frac{1}{2} \cdot a_y \cdot t_{ay}^2 \end{cases} \tag{14}$$

However, we also may consider the accelaration time a function of the displacement:

$$\begin{cases} t_{ax} = \sqrt{2 \cdot (x_a - x_0)/a_x} \\ t_{ay} = \sqrt{2 \cdot (y_a - y_0)/a_y} \end{cases} \tag{15}$$

From Eq. 10 we learned that these durations are equal. This is true only, if referred to the **whole** acceleration process –except for the case, where the distances to cover are equal AND the accelerations on each axis are equal too.
Because of the digital process which is moving the head, this motion is broken up into tiny steps that are defined by the motor (or equivalently the feed) speeds and the motor resolution. Because in our case Δs is identical on both the x- and the y-motor, and accelerations are in general inequal, each step requires different durations to be fulfilled. For instance, the first pulse of motors control trains, advances the head on the x-axis and y-axis -as said- by Δs. However, the durations for these motions must obviously be different:

$$\begin{cases} t_{x1} = \sqrt{2 \cdot \Delta s / a_x} = \sqrt{2 \cdot \Delta s / (\cos\psi \cdot a)} = k_x \\ t_{y1} = \sqrt{2 \cdot \Delta s / a_y} = \sqrt{2 \cdot \Delta s / (\sin\psi \cdot a)} = k_y \end{cases} \tag{16}$$

where k_x and k_y are clearly constants.
The next step, with displacements $2\Delta s$ on each axis is defined as:

$$\begin{cases} t_{x2} = \sqrt{2 \cdot (2\Delta s) / a_x} = k_x \cdot \sqrt{2} \\ t_{y2} = \sqrt{2 \cdot (2\Delta s) / a_y} = k_y \cdot \sqrt{2} \end{cases} \tag{17}$$

And similarly for $3\Delta s$, and so on:

$$\begin{cases} t_{x3} = \sqrt{2 \cdot (3\Delta s) / a_x} = k_x \cdot \sqrt{3} \\ t_{y3} = \sqrt{2 \cdot (3\Delta s) / a_y} = k_y \cdot \sqrt{3} \end{cases} \tag{18}$$

Generally,

$$\begin{cases} t_{xi} = \sqrt{2 \cdot (i\Delta s) / a_x} = k_x \cdot \sqrt{i} \\ t_{yj} = \sqrt{2 \cdot (j\Delta s) / a_y} = k_y \cdot \sqrt{j} \end{cases} \tag{19}$$

The reader should have noticed the difference in indices. The reason therefore is that both axis need a different amount of steps to reach $M_a(x_a, y_a)$!!! If that point has been reached, we have:

$$\begin{cases} t_a = t_{xn} = k_x \cdot \sqrt{n} \\ t_a = t_{ym} = k_y \cdot \sqrt{m} \end{cases} \\ \Longrightarrow \begin{cases} n = (t_a / k_x)^2 \\ m = (t_a / k_y)^2 \end{cases} \tag{20}$$

Of course, the number of steps required can be calculated with the help of Eq. 14:

$$\begin{cases} n = (x_a - x_0)/\Delta s = 1/2 \cdot a_x \cdot {t_a}^2 / \Delta_s \\ m = (y_a - y_0)/\Delta s = 1/2 \cdot a_y \cdot {t_a}^2 / \Delta_s \end{cases} \tag{21}$$

Using Eq. 17 and making Δs expressions of k_x and k_y respectively, which then are replaced in Eq. 21, we easily prove the correctness of Eq. 20, and thus of the whole reasoning.

Conclusion: We have found a very simple way to determine the timing and the amount of single steps that are required for programming acceleration and deceleration phases of stepper motors controlling an x/y device.

ToDo:

- It is evident that n and m are integer numbers. So, they must be rounded up by a method that reduces the overall error.

- As the ATMEL paper recommends (cf. footnote 14), timing should be delivered easiest by 16-bit timer interrupt, where the counter is calculated from $t_x i$ and $t_y j$ respectively. ATMEL has a slightly more complicated development for the computation of the timing, needing Taylor approximations, which introduces further errors. The required numbers of steps aren't calculated at all. For instance, the authors check instead, whether the constant speed value has been reached, an conclude from there that the complex timing now has finished. However, they also recommend to keep the timer ISR routines very short.

  Starting from our own equations, it seems that we need to calculate the square-root of numbers, which normally is considered rather time consuming. However, we may choose between two excellent and very fast method: either, we store the numbers in a Look-up table (LUT), depending on the data size required, or we will implement TOEPLER's extremely fast algorithm for the calculation of the square-root.[16] The latter is almost indicated, because we can execute the algorithm with scaled integer numbers without loosing any precision. So, the usual time-consuming wrap around routines that are normaly necessary here, can be avoided. The LUT-method would certainly be the fastest, provided we have enough program memory. If this method is used, the square-root would be ready after just a couple of program steps... super-fast and easy to implement on a PIC microcontroller!!!

Note on the Atmel paper: The paper only covers one single motor control. So, it cannot be transposed to synchronized x/y-motors, because the algorithm does not respect any proportionality. It is astonishing that the authors of the paper (inspired by D. Austin) didn't stick to Eq. 19!! Instead of considering the running time, they focused on the time differences between each step (scaled to a number denoted c_n, which leads to the equation using two squareroots instead of one only:

$$c_n = c_0 \cdot \left(\sqrt{n+1} - \sqrt{n}\right) \tag{22}$$

Applying a Taylor method, they get an approximate iteration:

$$c_n = c_{n-1} - \frac{2 \cdot c_{n-1}}{4n+1} \tag{23}$$

If we considered this way of thinking in our equations, we'd get the same thing (we only have a look at the x-axis):

$$\begin{aligned} t_{x(i-1)} &= k_x \cdot \sqrt{i-1} \\ \Delta t_{xi} = t_{x(i)} - t_{x(i-1)} &= k_x \cdot \left(\sqrt{i} - \sqrt{i-1}\right) \end{aligned} \tag{24}$$

However, we don't need that computation anyway, because the time difference may be computed by simple subtraction. And, as said, either using TOEPLER's algorithm, which is faster than a normal integer division, or the super-fast LUT method, we can compute the running time without any difficulty.

Another advantage of our method is that we do not need to deal with the issue that appears, because deceleration could start before acceleration has terminated. In our method this will never happen.
A last note: ATMEL's paper considers different deceleration and acceleration rates. This is not necessary with our synchronized method.

XII Day 10

24/02/2021

A LabVIEW simulation

The graphical LABVIEW environment certainly is the most powerful tool there is for rapidly conceiving numerical simulation programs. The creation of the spaghetti code shown in Fig. 24 just took an hour's

[16] http://www.rechnerlexikon.de/artikel/T%F6pler-Verfahren

work. The code isn't optimized; repeated parts could be fusioned into soubroutines; no documentation etc. However, the stuff correctly works and is sufficient for the purpose.

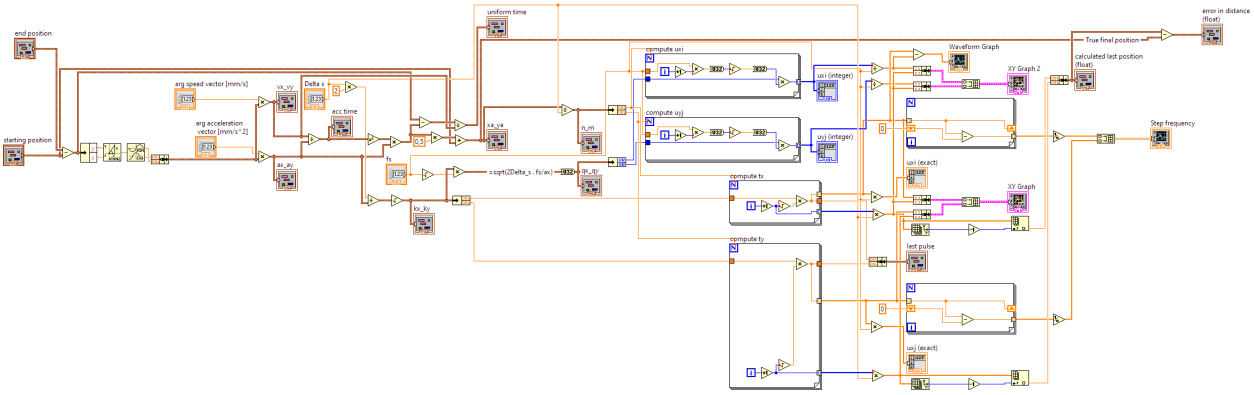


Figure 24: Jungle code in LabVIEW that simulates the acceleration process described so far.

- Single step distance: $\Delta s = 0.0075mm$
- Uniform speed: $v = 10mm/s (= 600mm/min)$
- Acceleration: $a = 4mm/s^2$
- Starting point: $M_0(x_0, y_0) = (0, 0)$
- End point: $M_e(x_e, y_e) = (100, 50)$
- Timer frequency: $f_s = 1MHz$*

*Timer frequency wasn't explained so far. In fact, we suppose the PIC 18F452 will be taught to run the timer at that frequency, which is to say that every microsecond the timer counter is incremented by 1. Motor steps occur at defined counter values u_x and u_y, where:

$$\begin{cases} u_x = t_{xi} \cdot f_s \\ u_y = t_{yj} \cdot f_s \end{cases} \tag{25}$$

Fig. 25 displays the x- and y-positions in function of time, whereas Fig. 26 shows the step frequencies on both axes. These frequencies are not to be confused with the timer frequency. These modulated frequencies are nothing else but the inverse of the changing time slots between two steps:

$$\begin{cases} f_x i = 1/\Delta_x t_i \\ f_y j = 1/\Delta_y t_j \end{cases} \tag{26}$$

where

$$\begin{cases} \Delta_x t_i = t_{xi} - t_{x(i-1)} \\ \Delta_y t_j = t_{yj} - t_{y(j-1)} \end{cases} \tag{27}$$

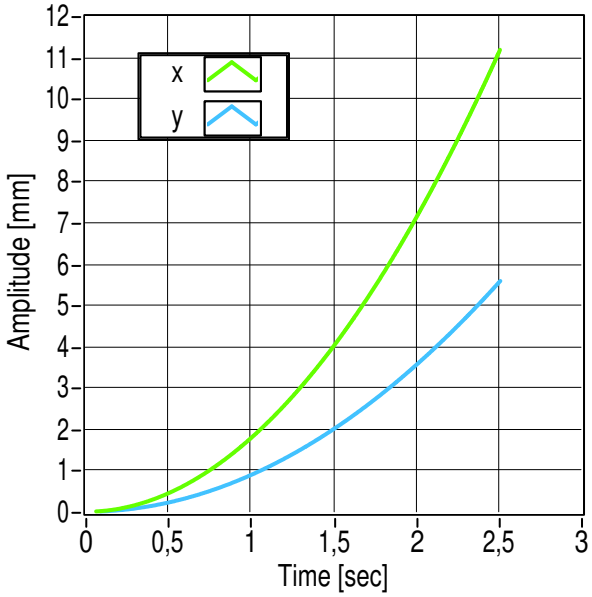


Figure 25: Pulse-generated motion with variable step timing.

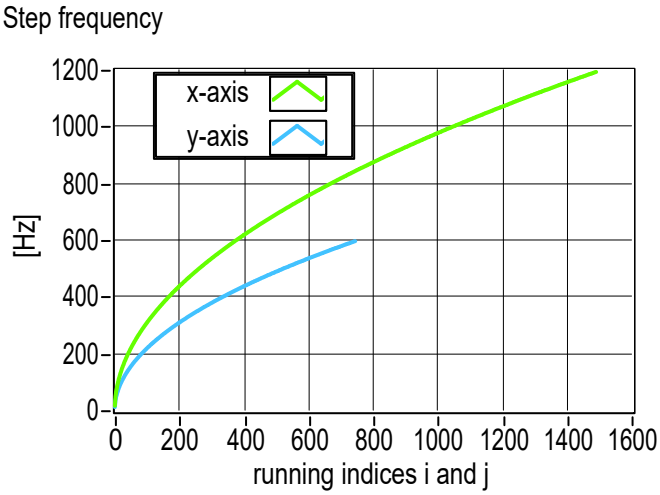


Figure 26: Step-frequencies on both axes in function of the running indices i and j. The x-motor needs more steps to reach its position than the y-motor. In order to maintain synchrony, the delays between steps are longer for the y-motor.

Running speed (x) $\hat{v}_x[mm/s]$	Running speed (y) $\hat{v}_y[mm/s]$	Acceleration (x) $a_x[mm/s^2]$	Acceleration (y) $a_y[mm/s^2]$
8.944	4.472	3.578	1.789
Number of steps (x) n	Number of steps (y) m	Acceleration time $t_a[s]$	Pos. after acc. $M_a(x_a, y_a)$
1491	745	2.5	(11.180, 5.590)
Constant (x) k_x	Constant (y) k_y	Constant2 (x) q_x	Constant2 (y) q_y
0.0647505	0.091571	65	92

Table 5: Results of the numerical simulation.**

**The second couple of constants q_x and q_y will be explained in the next section. They are computed on the base of integer calculation. The study of integer math in this context is necessary to find a way to keep execution time between steps short.

B Toepler's algorithm

Note: If acceleration is weak –here it is $4[mm/s^2]$–, the number of steps required to do the acceleration is impressively high. Depending on the overall program size, it will not be possible to use the LUT method for the computation of the squareroot.
The good news is that Toepler's algorithm (cf. Fig. 27) is probably the best way to do the squareroot on a computer.[17] The original algorithm, applied in the decimal system, could be regarded today as a curiosity. Its most important implementation was into the Friden SRW mechanical calculator (1952).[18] Note the missing credits to Toepler.[19]
Now, from the available documents about this astonishing method of calculating the squareroot, the extreme gain of calculation speed may not be obvious. In fact, the strength and the ingenuity of the algorithm lies in the application of the simple rule that the sum of a series of n consecutive odd numbers is n^2.[20]

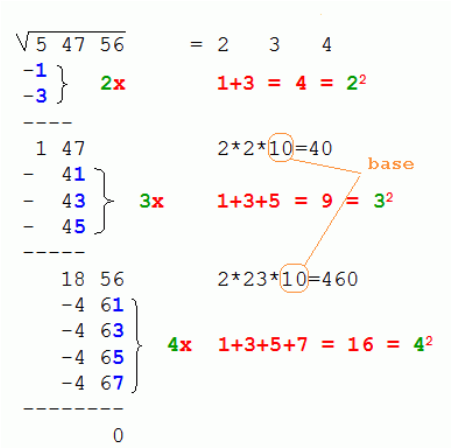


Figure 27: Self-explaning illustration of Topler's algorithm. The red marked lines deal with the term $2ab$ of the identity $(a+b)^2 = a^2 + 2ab + b^2$. (Source: cf. footnote 20.)

[17] cf. https://computarium.lcd.lu/literature/HISTORIC_CALCULATING/Square_Root/Toepler/Gaertner_toepler1.pdf
[18] https://www.hpmuseum.org/root.htm
[19] cf. http://dingler.culture.hu-berlin.de/article/pj179/ar179063
[20] https://computarium.lcd.lu/literature/HISTORIC_CALCULATING/Square_Root/Toepler/square_root_revisited.htm
(NB: some links on that page seem to be broken.)

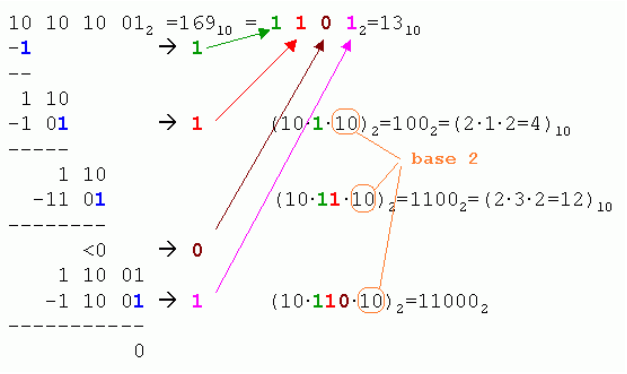


Figure 28: Transcription of the algorithm from base 10 into base 2. (Source: cf. footnote 20.)

However, if we switch to the binary system (cf. Fig. 28) things become crystal clear that this method requires 16 iterations only to compute the square root of an U32 number. This doesn't change, if one switches to single precision floating point representation. In this case, only the wrap around routines (normalization, and type casting) cost additional computation time. As can be seen, comparably to the traditional babylonian method, the digits of the input number are grouped by couples. In base 10, the maximal number of consecutive odd numbers to subtract per step is 9. Thus, the highest odd number to deduce in the first step is the 9th odd number (=17). By contrast, in base 2, there is only a single possible odd number to deduce per step! This makes the implmentation of the algorithm very easy, as can be seen here (NB: we recommend to run this program in VIRTUAL C in single step mode, and see how the the stack numbers evolve –attention: little Endian!):[21]

```
#include <stdio.h>
//Computes the squareroot of an integer number (I32)

int main(void){
        int x=2500;                     //x=argument value (example: 2500)
        int res=0;                      //start with zero
        int one=0x40000000;             //this sets the second highest bit to 1
        int tmp;                        //temporary variable
        while (one!=0) {                //REPEAT
                tmp=res | one;          //bit operation
                res=res>>1;             //shift res 1 digit to the right
                if(x>=tmp) {            //compare and BEGINIF
                        x=x-tmp;           //subtract
                        res=res |one;      //bit operation
                }                          //  ENDIF

                one=one>>2;             //shift "one" 2 digits to the right
        }                               //finish
        printf("%d\n",res);
        return res;
}
```

We now can reconsider Eq. 17ff., in order to stick to integer numbers. As we need the running time expressed

[21] C-program taken from: H. S. Warren, Jr., *Hacker's delight*, Addison Wesley, US, (2013), pp. 285-287. (NB: This book has no reference to Toepler. However, it asserts that the algorithm was described in J. von Neumann paper *First Draft of a Report on the EDVIAC*, in *Papers of John von Neumann on Computing and Computing Theory*, Vol. 12 in the Charles Babbage Institute reprint Series for the Histry of Computing, MIT Press, (1987).http://abelgo.cn/cs101/papers/Neumann.pdf).

in timer ticks rather than in seconds, let:

$$\begin{aligned} u_{xi} &= t_{xi} \cdot f_s = k_x \cdot \sqrt{i} \cdot f_s \\ &= k_x \cdot \sqrt{f_s} \cdot \sqrt{i \cdot f_s} \\ &= \sqrt{2 \cdot \Delta s / a_x \cdot f_s} \cdot \sqrt{i \cdot f_s} = q_x \cdot \sqrt{i \cdot f_s} \end{aligned} \tag{28}$$

Mathematically this operation doesn't make much sense. However, for the numerical calculation the gain is important, because q_x may be truncated or rounded to a valuable integer number, and precision is gained due to the application of the square-root to larger numbers. f_s must be chosen in order to make sure $i \cdot f_s$ always stay within the bounds of the U32 representation and no overflow is happening. The computation can be optimized, if f_s is made a power of 2, which reduces the product $i \cdot f_s$ to a simple left shifting of i. Also, q_x may be scaled by a factor before rounding, and division by the scale is made at the end of the operation. This will provide some more accuracy. Running the LabVIEW simulation with different speed and acceleration values showed that the error in position during acceleration time always stays below one single step. Depending on the input values errors may be in fact less than $10^{-4}mm$. Errors are not cumulative, if the motor control procedure starts from the numbers of steps to be executed on both axes, because motion by n and m steps means anyway that the Profiler has moved by the required amounts of elementary distances Δs.
Conclusion: The control method for motor acceleration using integer numbers during the process proves to be robust and optimal.

C General concept of the μC firmware

Now that we have explored the tools and conditions, it is time to conceive the general design of the control firmware destined to the PIC 18F452.

C.1 Firmware functions

The following list of functions is non-exhaustive compared to the Day 2 brainstorming.

1. Bi-directional serial communication with the computer $\Longrightarrow$ Get control commands from the computer (start, pause, abort, reset, G-Code, parameter setting). Use of serial RX- and TX-interrupts.
2. Analyse of the commands received (message integrity, G-Code command correctness)
3. Send replies to the computer (acknowledge command received, actually executed command, position data, speed data, internal state –accelelerating, running, decelerating, ready, waiting, aborted, reset, error–)
4. Extract the motion control data from the commands.
5. Calculation of the data required for the next movement (acceleration, run, deceleration phases)
6. Step pulse generation (acceleration, run, deceleration)
7. Motor-switches handler (step counter reset)
8. Error handling

C.2 General concept of the PIC 18F452 firmware

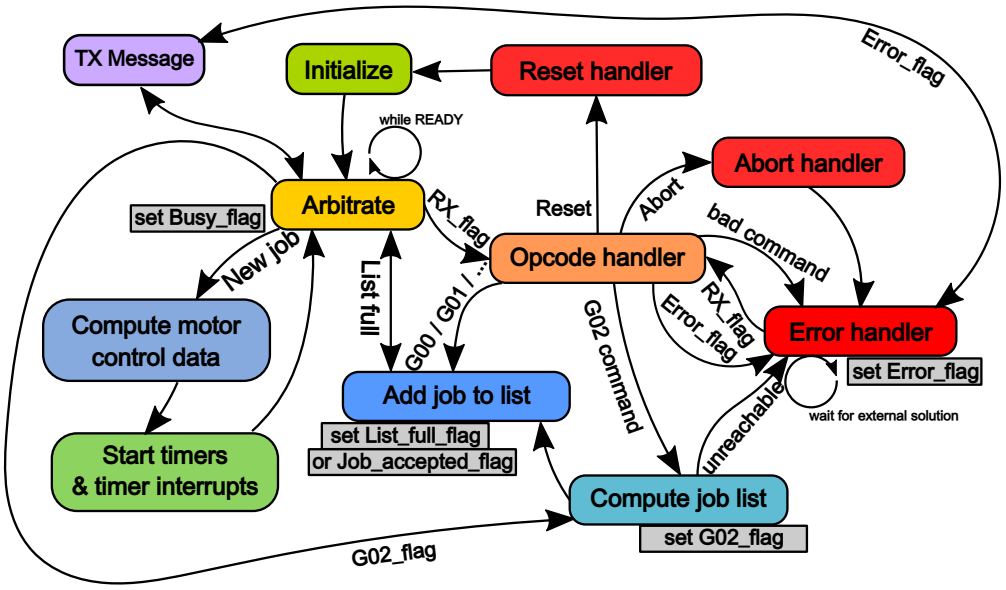


Figure 29: The main program.

Fig. 29 shows the complex state machine that is run in the main procedure. The firmware requires additional program parts:

1. TIMER1: increment the tick counter $timer_1$ at $f_s = 1Mhz$ (zero at start; timer is started from the state machine)
2. TIMER2: interrupt at $f_c = 10 - 50kHz \implies$ Purpose: check if $timer_1 > u_{xi,yj}$ and clock the motor accordingly, compute the next $u_{xi,yj}$ values. (Release the clock pulse after a few TIMER2 ticks – number still to be defined. TIMER2 interrupt is started from the state machine and stopped by the ISR, once the job is done (=number of desired steps have ben executed; sets the *Job_Ready_Flag*).*
3. UART RX ISR (store the command bytewise, and check for integrity.)
4. UART TX ISR (just empty the TX buffer.)

*by contrast to the ATMEL method presented above, the firmware has to deal with multiple motor events. The simple 1MHz interrupt proposed by ATMEL doesn't work for more than one motor, because it can happen that $u_{xi,yj}$ values are very close to each other, so that step computation could not be finished before the next interrupt should occur. This would lead to undesired step losses. We therefore propose the use of a second timer that regularly checks, whether $timer_1$ has counted more pulses than the next time indices $u_{xi,yj}$. We will have to do some experiments here, because alternatively we could simply reduce TIMER1 frequency and stick to a single timer. Anyway, $u_{xi,yj}$ values are inversely proportional to the squareroot of the acceleration and evolve in the intervall [5000, 100000], while the acceleration decays from 400 to $4mm/s^2$. So, timer interrupts may occur at much slower frequencies without any significant increase of errors.

XIII Day 11

27/02/2021

MPLAB C18 test

Switching to C18 is a bit adventurous, because you must learn how to tell the compiler, where the **.c .h** and **.lib** files are located (cf. Fig. 30). A second thing to experience is setting the configuration bits via the **#Pragma** command (cf. Fig. 31). The rest of it in the MPLAB environment is identical to building and programming **.asm** files. NB: The programming mode should be set to *Release* rather then *Debug*.

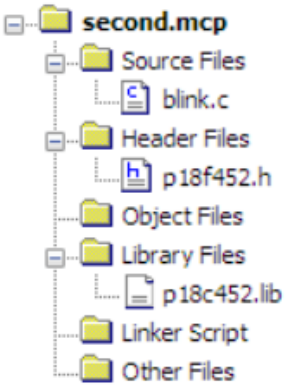


Figure 30: Directory info.

```
#include <p18F452.h>

#pragma config OSC = HS
#pragma config PWRT = ON
#pragma config WDT = OFF
#pragma config BOR = OFF
#pragma config DEBUG = OFF
#pragma config LVP = OFF

#define _XTAL_FREQ 4000000  //Hz

void delay (void)
{
  int i;

  for (i = 0; i < 5000; i++)
    ;
}

void main()
{
    PORTB = 0;         // Make PORTB zero
    TRISB = 0;        // Make PORTB all output

    while(1)
    {
        PORTB = 2;        // Make RB2 one
        delay();          // delay a bit

        PORTB = 0;       // Make RB2 zero
        delay();         // delay a bit
    }
}
```

Figure 31: First C program via MPLAB (test of configuration bits).

Inline Assembly code may be included through the **_asm** and **_endasm** commands. However, accessing

macros used in MPASM code is a real mess.

XIV Day 12

28/02/2021

Timer and serial interrupts

Here is another test code! The goal is to have the PIC18F452 accept single characters received via UART, and change the flashing rate according to the byte value (see also Fig. 32). Everything should be done with interrupts. The μC is connected to the PC via RS232. Therefore a MAX232 IC is used as an interface (cf. Fig. 33).

```
#include <p18F452.h>

#pragma config OSC = HS
#pragma config PWRT = ON
#pragma config WDT = OFF
#pragma config BOR = OFF
#pragma config DEBUG = OFF
#pragma config LVP = OFF

#define _XTAL_FREQ 20000000  //Hz

//void main (void);
void InterruptHandlerHigh (void);

//This program runs timer1 interrupts and UART RX interrupts. The value received
//changes the flashing rate.
//-----------------------------------------------------------------------------
// Main routine
char p;
int count;
char tmp;
void main()
{
   //Init UART 19200bd, 8
   TRISCbits.RC7=1; //set RX to input
   TRISCbits.RC6=0; //set TX to output
   SPBRG=0x40;      //19.2kbd 0.16%error
   TXSTAbits.BRGH=1; //needed to produce 19.2kbd with small error
   TXSTAbits.SYNC=0; //asynchronous mode
   RCSTAbits.SPEN=1; //configure ports as serial
   RCSTAbits.RX9=0;  //no error bit on RX
   TXSTAbits.TX9=0;  //idem on TX
   RCSTAbits.CREN=1; //continuous receive enabled
   TXSTAbits.TXEN=1; //enable TX
   PIR1bits.RCIF=0;  //clear RX interrupt flag
   PIE1bits.RCIE=1;  //enable RX interrupt

   //Use flashing PORTB by timer interrupt, RB2 to show, we're alive
   TRISB=0;
   PORTB=0;
   count=0;
   p=0;
   tmp=2;
   T1CON=0x31; //configure timer1
   PIR1bits.TMR1IF=0; //clear timer1 interrupt flag
   PIE1bits.TMR1IE=1; //enable timer1 interrupt

   INTCONbits.PEIE=1; //enable periphery interrupts
   INTCONbits.GIE=1;  //enable loabl interrupts

   while (1){} //do nothing
}
//-----------------------------------------------------------------------------
// High priority interrupt vector

#pragma code InterruptVectorHigh = 0x08


void
InterruptVectorHigh (void)
{
  _asm
```

```
    goto InterruptHandlerHigh //jump to interrupt routine
  _endasm
}

//-----------------------------------------------------------------------------
// High priority interrupt routine

#pragma code
#pragma interrupt InterruptHandlerHigh

void
InterruptHandlerHigh ()
{

  //first dispatch interrupts
  if((PIR1bits.TMR1IF=1)&&(PIE1bits.TMR1IE=1)) {   //handle timer interrupt i.e. blink
        count+=1;
        PIR1bits.TMR1IF=0;
         if (count>tmp) {
           p=~p;
           if(p!=0) {
            PORTB=2;
           }  else  {
            PORTB=0;
           }
          count=0;
         }
        }
   if((PIR1bits.RCIF=1)&&(PIE1bits.RCIE=1)) {//handle RX interrupt
      tmp=RCREG;
          PIR1bits.RCIF=0;
    }
}
```

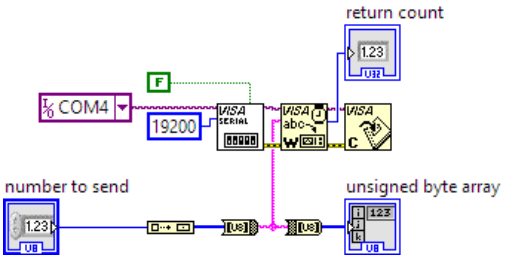


Figure 32: A most simple LabVIEW program sending a single character.

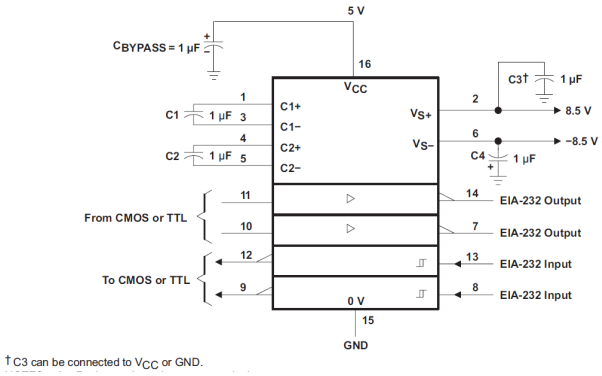


Figure 33: Typical application of the MAX232.

XV Day 13

28/02/2021

Send complement back

The following program snippet sends back the complement of the received value via TX. Note that the correctly returned value only appears in the second run. The reason therefore is that although the TXIF flag is briefly cleared, and then reset by the PIC, but the TXREG isn't necessarily empty. This is known as the *one cycle clearing delay problem.*[22]
(NB.: the flashing doesn't work correctly for values > 127. The reason therfore is that the compiler considers *tmp* as a signed character.)

```
...
if((PIR1bits.RCIF=1)&&(PIE1bits.RCIE=1)) {//handle RX interrupt
      tmp=RCREG;
          PIR1bits.RCIF=0;
      while(PIR1bits.TXIF=0){};
      tmp2=~tmp;
      TXREG=tmp2; //send back complement
      while(PIR1bits.TXIF=0){};
     }
```

In earlier projects we already had to deal with this issue. The solution is to put a pause between the sending of two consecutive characters, and at the and add one or two dummy characters to empty the PIC internal buffer.
Anyway, we think that RX should be handled by interrupt, because the μC does not know, when the controlling PC is sending. The polling method would waste too much computing time. By contrast, TX can better be handled by the main task, and interrupts aren't necessary here, because first, the returned TX messages are very short in comparison to the RX messages and second, the PIC is in full control of the moment of transmitting.
Here therefore a code snippet that is sending the contents of a buffer:

```
// Main routine
char p;
int count,i;
unsigned char tmp,j,out;
char buffer[5]={'c','o','d','e','\0'};
void main()
{
   //Init UART 19200bd, 8
   TRISCbits.RC7=1; //set RX to input ...
...
 while (1){ //send out continuously the contents of buffer
       for(j=0;j<5;j++) {
         out=buffer[j];//get the character from buffer
         TXREG=out;//send the character
       for(i=0;i<32000;i++) {}
      }
...
```

Conclusion: We now have got enough experience with the PIC 18F452 to program the firmware as planned.

XVI Day 14

7/3/2021
MPLAB C18 has its particularities. So, we have to play a bit to get the correct code for some details that we will need in the program: array handling, constant enumeration and ring buffer, for instance.

[22] cf. *Asynchronous Communication with the PICmicro USART, Application note AN774*, Microchip, p. 4.

Struct, array handling and enumeration in C

We'll need structured datatype, arrays for passing the jobs to the job handler. (Jobs are head movements, for instance.) Enumerations will be needed for making the code readable. This will especially be needed for states definitions.

```
#include <p18F452.h>
#include <stdlib.h>
#include <stddef.h>

#pragma config OSC = HS
#pragma config PWRT = ON
#pragma config WDT = OFF
#pragma config BOR = OFF
#pragma config DEBUG = OFF
#pragma config LVP = OFF

#define _XTAL_FREQ 4000000  //Hz

enum op_t {
        NO_OP,
        ADDITION,
        MULTIPLY
     } ;

struct t_job {
        unsigned long x;
        unsigned long y;
        unsigned int z;
};

struct t_job jobs[7];

void delay (void)
{
  int i;
  for (i = 0; i < 5000; i++)
    ;
}

void main()
{
    int i;
    enum op_t op = MULTIPLY;
    PORTB = 0;         // Make PORTB zero
    TRISB = 0;        // Make PORTB all output

    op = ADDITION;
    op = NO_OP;

    for (i=0;i<7;i++) {
        jobs[i].x=i;
        jobs[i].y=i*2;
        jobs[i].z=i*3;
    }
    while(1)
    {
        PORTB = 2;        // Make RB2 one
        delay();          // delay a bit

        PORTB = 0;       // Make RB2 zero
        delay();         // delay a bit
    }
}
```

Ring buffer in C

The job array will be accessed as a ring buffer, because it is asynchroneously filled and emptied. Filling will be done, when new programmed instructions will arrive from the UART; emptying will be executed from within the jobs handler. This test code only uses unsigned char type in the array.[23]

```
#include <p18F452.h>
#include <stdlib.h>
#include <stddef.h>
```

[23]This code was inspired –but maximally changed– by https://gist.github.com/jinckse/ab2a48d2816cb7b61509b05ca8f8f902.

```
#pragma config OSC = HS
#pragma config PWRT = ON
#pragma config WDT = OFF
#pragma config BOR = OFF
#pragma config DEBUG = OFF
#pragma config LVP = OFF

#define _XTAL_FREQ 4000000  //Hz

/*****************************************************************************
 * Function Name: Circ_Buf_Insert()
 * Returns: -
 * Parameters: unsigned char c
 * Description: Insert a BYTE into the circular buffer
*****************************************************************************/
#define MAX_BUF_SIZE 4
unsigned char buffer[MAX_BUF_SIZE];
int buffSize=0;
int head=0;
int tail=0;
enum buffState_t {
        EMPTY,
        FILLED,
        FULL
     } ;
enum buffState_t bufferState = EMPTY;

void Circ_Buf_Insert(unsigned char c) {

        if (bufferState!=FULL) {
                buffer[head] = c;
                // Advance pointer
                head++;
                head %= MAX_BUF_SIZE;

                // Update buffer size
                buffSize++;
                if (buffSize>=MAX_BUF_SIZE) {
                        bufferState=FULL;
                } else {
                        bufferState=FILLED;
                }
        }
}

/*****************************************************************************
 * Function Name: Circ_Buf_Retrieve()
 * Returns: char
 * Description: Retrieve a BYTE from the circular buffer.
*****************************************************************************/
unsigned char Circ_Buf_Retrieve() {
        unsigned char b;

        if (bufferState!=EMPTY) {
                b= buffer[tail];
                // Advance pointer
                tail++;
                tail %= MAX_BUF_SIZE;

                // Update buffer size
                buffSize--;
                if (buffSize<=0) {
                        bufferState=EMPTY;
                } else {
                        bufferState=FILLED;
                }
        }
        return b;
}

/*********************************************************/


void main()
{
    unsigned char content;
    Circ_Buf_Insert(7);
    Circ_Buf_Insert(8);
    content=Circ_Buf_Retreive();
    Circ_Buf_Insert(9);
```

```
    while(1){
    }
}
```

Note on this code: The *insert* function should be called only, if the bufferstate isn't **FULL**; the *retrieve* function should be called only, if the state isn't **EMPTY**.

States, flags and and messages

We will distinguish the following system states:

1. **Arbitrate:** READY, OPCODE_HANDLING, INSERT_JOB, RETRIEVE_JOB, DIRECT_MODE, ERROR_MODE
   - READY: Do nothing but checking the flags:
     - ◇ RX_flag: (we got a message via UART) $\Longrightarrow$ change to OPCODE_HANDLING
     - ◇ VALID_state: (we got a new message from the PC)
       - ∗ DIRECT: we have a direct command $\Longrightarrow$ change to DIRECT_MODE
       - ∗ PROGRAM: there is a new job to add to the list $\Longrightarrow$ change to INSERT_JOB
     - ◇ BUSY_flag: (at start this one is FALSE; it is cleared in the timer interrupt routine, once the job is done)
       - ∗ FALSE: we may execute a new job $\Longrightarrow$ change to RETRIEVE_JOB
       - ∗ TRUE: stay READY for coming missions
     - ◇ DONE_flag: (this flag is set in the timer interrupt routine, if the job is done) $\Longrightarrow$ TX DONE message and clear the flag. (We use this method instead of sending from the Timer ISR, in order to prevent any collision on the TX channel. Idea: make sure that TX is exclusively done in the main routine.)
   - OPCODE_HANDLING:
     - – if the VALID_state isn't NONE, the last message hasn't been handled so far $\Longrightarrow$ change to READY (we will return here soon; since the RX_flag isn't cleared, we will not admit any new message. Anyway, the computer shouldn't send, because the message hasn't been acknowledged, which is done, once the job has been executed.)
     - – VALID_state is NONE: check the validity of the new message:
       - ◇ valid: clear the RX_flag, set the VALID_state to DIRECT or PROGRAM, depending on the message $\Longrightarrow$ change to READY
       - ◇ non-valid: set the ERROR_state to BAD_COMMAND $\Longrightarrow$ change to ERROR_MODE
   - INSERT_JOB: check the JOB_LIST_state:
     - ◇ EMPTY or FILLED: add new job to the list and clear the VALID_state send the ACK message via TX $\Longrightarrow$ change to READY
     - ◇ FULL $\Longrightarrow$ change to READY (since the VALID_state isn't changed, the program will return here soon)
   - RETRIEVE_JOB: check the JOB_LIST_state:
     - ◇ FULL or FILLED: get the next job from the list, compute motor contraol data and start timer interrupt $\Longrightarrow$ set the BUSY_flag and change to READY
     - ◇ EMPTY $\Longrightarrow$ change to READY (nothing new to do)
   - DIRECT_MODE: ABORT, PAUSE **TODO**; so far, no idea yet (so, don't get into that state)
   - ERROR_MODE: **TODO**; so far, just stop everything
2. **ERROR_state:** NONE, BAD_COMMAND
3. **VALID_state:** NONE, DIRECT, PROGRAM
4. **JOB_LIST_state:** EMPTY, FILLED, FULL

5. **RX_flag:** TRUE, FALSE
6. **BUSY_flag:** TRUE, FALSE
7. **DONE_flag:** TRUE, FALSE
8. **TX_message:** ACK, DONE

In order to make sure that the **Arbitrate** procedure reads the flags pretty fairly, we do introduce a flag index, which is incremented everytime the procedure is called. This prevents locking the state machine in checking for a flag that is never set or cleared. Without such a Round-Robin method, READY state could freeze with the first RX_flag calling OPCODE_HANDLING. If VALID_state isn't NONE, there is no chance that RX_flag will ever be cleared. Retruring to READY will read RX_flag again, and the cycle will continue forever. However, if during next call of READY, the another flag –VALID_state, for instance– is checked, we pass either to DIRECT_MODE or INSERT_JOB, where VALID_state can be cleared.

XVII Day 15

8/3/2021

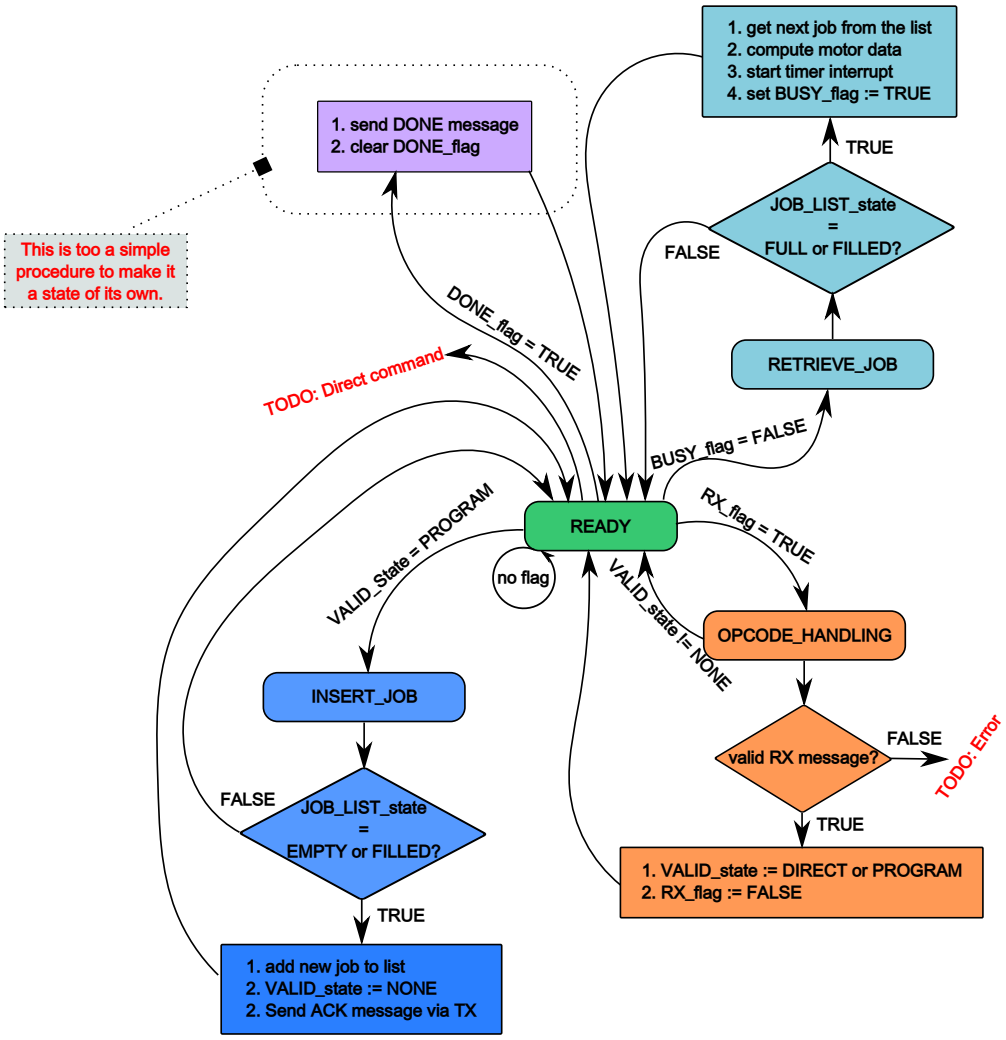


Figure 34: The main program as a state-machine.

Important notes:

- The RX_flag is set in the RX interrupt routine, if it has received a correct message. This is not to be confused with validity checking, which is the method to verify, if the content of the received message makes sense for the PROFILER control. The RX integrity test is just a control of the UART channel and answers to the question: *Did we receive a message that corresponds to the protocole?*
- Sending ACK back to the computer via TX means, we got a valid PROFILER command. This liberates the RX buffer, and a new command can be sent by the computer.
- Sending DONE back to the computer means that program step P_i has been correctly executed by the milling machine. Now the computer knows where we are in the program, and also the actual coordinates of the head, which must not be sent back to the computer, since anyway there is no additional information collected by the machine, regarded the fact that there are no motor position sensors.
- DONE_flag must be set in the timer interrupt routine, everytime a job has been made.
- The whole concept of this program pursues the goals to stay as briefly as possible in any of the states and in the interrupt service routines, in order to have a high rate of flag checking, excellent timing in RX, and –most important – to garantee a minimum of jitter in the timing control of the motors.

XVIII Day 16

9/3/2021
Idea: Eventually we could use the CTS/RTS handshaking for controlling the UART channel. This might be a special help for longer phases of UART silence.

Now, let's try to control a single motor with our acceleration/deceleration method.

Motor control with integer numbers

The PROFILER max dimensions are:

x: 300mm $\sim$ 100 spindle revolutions $\sim$ 40000 motor steps

y: 400mm $\sim$ 133.33... spindle revolutions $\sim$ 53333 motor steps (rounded, since there are no fractional steps)

y: 100mm $\sim$ 33.33... spindle revolutions $\sim$ 13333 motor steps (rounded)

$\Longrightarrow$ If we stick to absolute coordinates, we might limit the numbers of steps variables to type U16.

Idea: Have the computer accept normal G-Code and convert it to an easier to handle reduced instruction set. For instance, translations of the origin (G53-G59), could be done in the computer only, and converted to absolute coordinates always starting at the machine origin. Relative coordinates (G91) could also be converted to absolute coordinates in the computer. Also inch/mm (G20/21) could only be relevant to the computer only, where the coordinates would be converted to motor steps. This would it make possible to stick to U16 integer values (no fractions, no sign).

Compute the data for motion from $x_0(0)$ to $x_e(13333[steps])$.

$$\begin{aligned} \text{Let: } f_s &= 100[kHz] \\ v &= 1333[\text{steps/s}] \\ a &= 533[[steps/s^2] \\ \Delta s &= 1[\text{steps}] \\ n &= \frac{\cos\psi \cdot v^2}{2 \cdot a \cdot \Delta s} = \frac{1 \cdot 1333^2}{2 \cdot 533 \cdot 1} = \frac{32 \cdot 1333}{533} \cdot 1333/64 = 1667[\text{steps}]^* \end{aligned} \tag{29}$$

* Splitting and rearranging the calclations allows integer operations with sufficient precision. The product with 32 limits the maximum speed to 2047[steps/s]=921mm/s, in order stay within the U16 limit $2^{16} - 1 = 65535$.
Using Eq. 7, 8, 21 and 28:

$$q_x = \sqrt{\frac{2 \cdot \Delta s \cdot f_s}{\cos\psi \cdot a}} = 2 \cdot \sqrt{2 \cdot \frac{100000}{4}/1/533} = 2\sqrt{50000/533} \approx 2\sqrt{93} \approx 18^{**} \tag{30}$$

**The correct value should be 19. In this operation we risk to loose precision.

$\Longrightarrow$ It might be useful here to use long integer U32 for both operations, since increasing the acceleration will reduce n and q_x even more.

For t_a to be around some realistic 50ms, with $v = 10mm/s = 1333[\text{steps}/s]$, $a = v/t_a = 10/50 \times 10^{-3} = 200mm/s^2 = 26667[\text{steps}/s^2]$.

Using U32 with such a high acceleration, the calculations become much more precise:

$$\begin{aligned} n(16b) &= \frac{32 \cdot 1333}{26667} \cdot 1333/64 \approx 20[\text{steps}] \text{ !!!} \\ n(32b) &= \frac{65536 \cdot 1333}{26667} \cdot 1333/131072 = 33[\text{steps}] \\ q_x(16b) &= 2 \cdot \sqrt{2 \cdot \frac{100000}{4}/1/26667} = 2\sqrt{50000/26667} \approx 2 \\ q_x(32b) &= \sqrt{8192 \cdot 100000/1/26667}/64 = \sqrt{30719}/64 \approx 2 \text{ !!!}^{***} \end{aligned} \tag{31}$$

***There is no gain in precision here, albeit the first example yields the correct value 19.

Discussion

- The integer number idea flaws the initial goal of making the machine fully GRBL-compatible. However, a well documented LabVIEW control program could do most of the high-end computation job.
- As we can see so far, writing such a firmware for the PIC18F452 is an impressively time-consuming piece of work. The question now arises, if this is worth the effort, since meanwhile we have been aware that there exists a powerful solution with a cheap and yet comparable board, namely the Arduino MEGA 2560, which uses the Atmel ATmega2560 uC. The Arduino should run the opensource GRBL firmware.[24]. We could continue using the main control board, because the Arduino delivers everything we need: communication with the PC via USB, step pulses for three axes, limit switches for three axes, a single stepper Enable/Disable port, spindle direction, variable spindle PWM, probe, reset/abort, feed hold, cycle start/resume, coolant enable*. (The Arduino Uno seems a bit too weak for our purpose.)

[24] https://github.com/gnea/grbl-Mega

- An alternative would be to change from PIC 18F452 to Raspberry Pi. This would require additional peripherical electronics, because of the 3.3V limitation of the RasPi.
- A total alternative could be to forget about the good old control board and switch to an Arduino driven controller from the shelf like the J Tech Photonics GRBL board at the price of US$130.-.[25] This would require an additional power supply.

* This pin normally is used to control a cooling system. It could of course be employed for any other purpose.

Conclusion We will switch to the Arduino Mega alternative, and see, if we can successfully run the GRBL firmware.

Getting started with the ARDUINO MEGA 2560

Reference:https://www.arduino.cc/en/Guide/ArduinoMega2560

XIX Day 17

9/3/2021
Main GRBL Reference: https://github.com/gnea/grbl-Mega/wiki

GRBL features (everything we dream of):

- Grbl is for three axis machines. No rotation axes (yet) just X, Y, and Z.
- There are only two deviations from the written G-code standard listed below:
  - Multiple full circle arcs with G2 and G3 arcs with a P word is not supported.
  - Laser mode alters the operation of M3, M4, and spindle speed S word changes. See the Laser Mode page for details.
- Supported G-Codes in v1.1:
  - G0, G1: Linear Motions
  - G2, G3: Arc and Helical Motions
  - G4: Dwell
  - G10 L2, G10 L20: Set Work Coordinate Offsets
  - G17, G18, G19: Plane Selection
  - G20, G21: Units
  - G28, G30: Go to Pre-Defined Position
  - G28.1, G30.1: Set Pre-Defined Position
  - G38.2: Probing
  - G38.3, G38.4, G38.5: Probing
  - G40: Cutter Radius Compensation Modes
  - G43.1, G49: Dynamic Tool Length Offsets
  - G53: Move in Absolute Coordinates
  - G54, G55, G56, G57, G58, G59: Work Coordinate Systems
  - G61: Path Control Modes
  - G80: Motion Mode Cancel

[25] cf. https://jtechphotonics.com/wp-content/uploads/2019/03/Instruction_Manual_GRBL_Controller_V1_3.pdf.

  - G90, G91: Distance Modes
  - G91.1: Arc IJK Distance Modes
  - G92: Coordinate Offset
  - G92.1: Clear Coordinate System Offsets
  - G93, G94: Feedrate Modes
  - M0, M2, M30: Program Pause and End
  - M3, M4, M5: Spindle Control
  - M8, M9: Coolant Control

- Acceleration management – We read:

  > *In the early days, Arduino-based CNC controllers did not have acceleration planning and couldn't run at full speed without some kind of easing. Grbls constant acceleration-management with look ahead planner solved this issue and has been replicated everywhere in the micro controller CNC world, from Marlin to TinyG. Grbl intentionally uses a simpler constant acceleration model, which is more than adequate for home CNC use. Because of this, we were able to invest our time optimizing our planning algorithms and making sure motions are solid and reliable. When the installation of all the feature sets we think are critical are complete and no longer requires us to modify our planner to accommodate them, we intend to research and implement more-advanced motion control algorithms, which are usually reserved for machines only with very high feed rates (i.e. pick-and-place) or in production environments. Lastly, here's a link describing the basis of our high speed cornering algorithm so motions ease into the fastest feed rates and brake before sharp corners for fast, yet jerk free operation.*

  and also:

  > *Super Smooth Stepper Algorithm: Complete overhaul of the handling of the stepper driver to simplify and reduce task time per ISR tick. Much smoother operation with the new Adaptive Multi-Axis Step Smoothing (AMASS) algorithm which does what its name implies (see stepper.c source for details). Users should immediately see significant improvements in how their machines move and overall performance!*

List of other references:

- http://bengler.no/grbl : site of one of the open-source GRBL pioneers.
- https://www.duet.ac.bd/wp-content/uploads/2019/08/Thesis_122211P.pdf : excellent tutorial for Arduino-based GRBL routing (Inkscape tutorial included)
- https://www.silabs.com/documents/public/application-notes/an155.pdf : a rather complete explanation of stepper motor control (acceleration/deceleration, UART) (Flowchart and C-code included); no GRBL-compatibility.
- https://www.researchgate.net/publication/301943513_Control_Algorithm_of_Acceleration_Curve_for_Stepper_Motor: Zeng, Min & Hu, Cheng-Zu & Hu, Peng-Fei. (2016). Control Algorithm of Acceleration Curve for Stepper Motor. Journal of Control and Systems Engineering. 4. 32-39. 10.18005/JCSE0401004.

Important note on the educational process of this project:

Readers might be astonished of the evolution of the project, especially that eventually we chose to abandon the 18F452, although we did advance so far. Many would consider this as frustrating. However, those who followed older author projects, know that there is an educational approach greatly inspired by the constructivist method. *Constructivism is a theory in education that recognizes learners construct new understandings and knowledge, integrating with what they already know.*[26] This method generally is referred to Jean Piaget's work on cognitive development. Strangely, we have to say that most explanations of the method miss the

[26] https://en.wikipedia.org/wiki/Constructivism_(philosophy_of_education).

core, which is the fun you get from building, doing, and most importantly undoing, taking apart, hacking, re-building, altering other people's ideas, etc. Projects must involve you wholeheartedly, as William Kilpatrick wrote in *The Project Method*[27] We think that in no other context learning could be more efficient than in the project, where the subject is wholeheartedly concerned. The French philosopher Paul Valery certainly was one of the groundbreaker of the constructivist ideas. He wrote:

> *Celui qui n'a jamais saisi, ft-ce en rve ! le dessein d'une entreprise qu'il est le matre d'abandonner, l'aventure d'une construction finie quand les autres voient qu'elle commence, et qui n'a pas connu l'enthousiasme brlant une minute de lui-mme, le poison de la conception, le scrupule, la froideur des objections intrieures et cette lutte des penses alternatives o la plus forte et la plus universelle devrait triompher mme de l'habitude, mme de la nouveaut, celui qui n'a pas regard dans la blancheur de son papier une image trouble par le possible, et par le regret de tous les signes quio ne seront pas choisis, ni vu dans l'air limpide une btisse qui n'y est pas, celui que n'ont pas hant le vertige de l'loignement d'un but, l'inquitude des moyens, la prvision des lenteurs et des dsespoirs, le calcul des phases progressives, le raisonnement projet sur l'avenir, y dsignant mme ce qu'il ne faudra pas raisonner alors, celui-l ne connat pas davantage, quel que soit d'ailleurs son savoir, la richesse et la ressource et l'tendue spirituelle qu'illumine le fait conscient de construire.*[28]

So, even if at the end, we switch to a ready-from-the-shelf solution, during this project, we learned within 3 weeks:

- internals of the Colinbus Profiler, included: steppermotor caracteristics, ratings, limits, geometry, etc.
- hardware stepper motor control (H-bridge, L6208, current limitation, etc.)
- Profiler main board functions and wirings
- torque, acceleration, deceleration processes of open-loop stepper motors
- understanding of max. feed speed and acceleration ratings
- acceleration time, uniform acceleration, synchronous multi-axes acceleration
- integer calculation, for instance Toepler's algorithm (... and the marvelous *Hacker's delight* book).
- design of a complex state machine using flags
- running MPLAB with C18 and ICD2 in-circuit programming of the 18F452
- interrupt handling in C18
- implementation of a circular buffer
- UART handling
- GRBL compatibility
- G-Code
- to come: Arduino Mega 2560

$\Longrightarrow$ We know how to handle the Colinbus Profiler and how not to use it. We will be able without any difficulty to handle future troubleshooting.

[27] W. H. Kilpatrick, *The Project Method*, NY, (1929); see also: http://www.educationengland.org.uk/documents/kilpatrick1918/index.html.
[28] P. Valry, *Introduction la mthode de Lonard de Vinci*, Gallimard, Paris, (1957), pp. 46-47.

XX Day 18

9/3/2021
The Arduino board arrived today, and it took just an hour to get familiar with the stuff, install the GRBL-MEGA 1.1f.20170802 firmware, and run it succesfully. Using the open-source Candle software,[29] we could control the x-motor without any trouble in both directions, with acceleration and G00/01 commands.

- The Arduino is powered via USB.
- Pinout is:

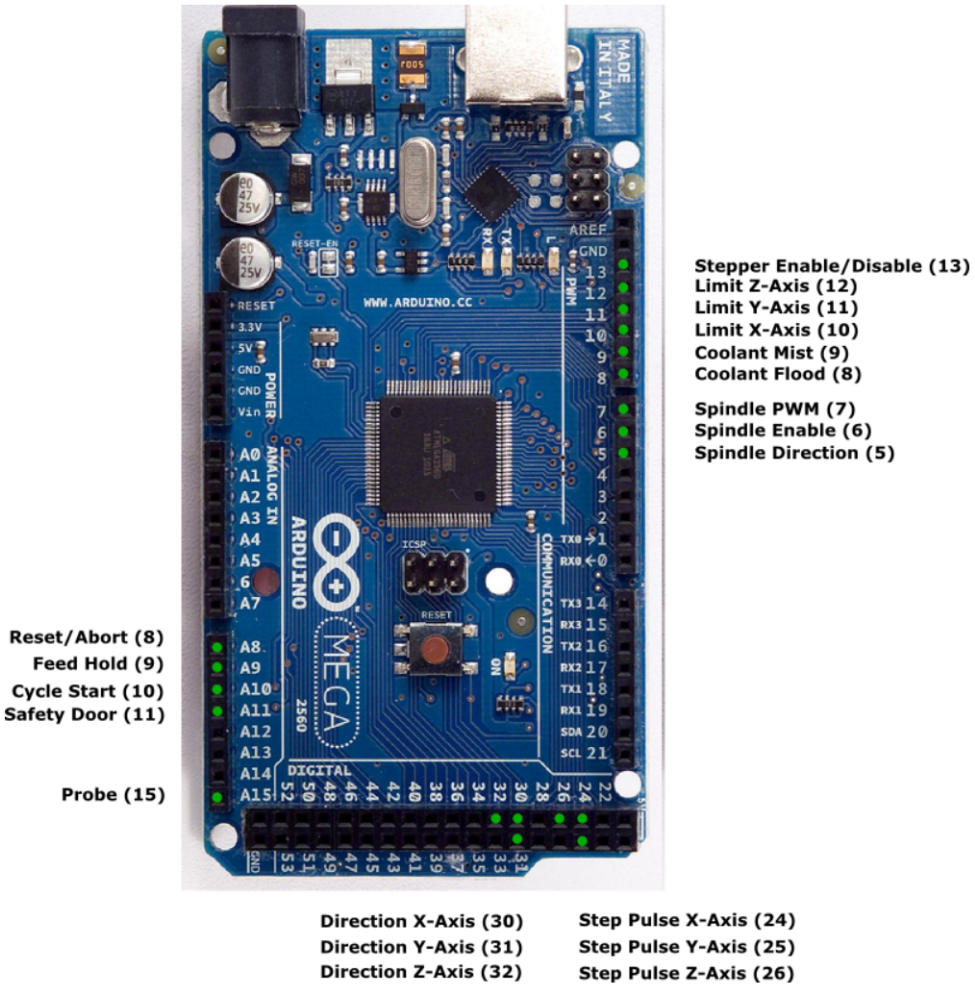


Figure 35: How to connect the Arduino MEGA 2560 to the Colinbus main control board.

- The correct COM-port must be set.
- The baud rate must be set to 115200.
- The maximum feedrate (G00) must be set to a valid speed (the default 2500 mm/min doesn't work). 500mm/min should be fine.
- Note that in order to work, the first call of G01 needs the feed-speed to be set. So, G01 100 F250, is fine.

[29] https://github.com/Denvi/Candle.

XXI Day 19

15/3/2021

Catastrophic event

After having rebuilt the PROFILER, we connected the ARDUINO board to the main control board. Everything worked fine: motors were running correctly in all axes. However, as soon, as the motors stopped, there was a humming noise on each motor that only stopped, if we manually connected the *ENABLE* pins to *GND*. Now, inadvertently, the pin came briefly into contact with the grounded casing... with a hearable spark... Ground $\neq$ *GND*!!! We should have paid more attention. Now the **THREE** L6208 motor driver ICs have died... very fine smd-soldering, chips strongly fixed to the board... IRREPARABLE.

So, after some googling, we choose to order three LEADSHINE DM542T Full Digital Stepper Drives.[30]

These drivers solve two persisent problems: the resonance issue and the "enable/disable" issue. Now, let's wait for them to come... very frustrating day.

XXII Day 20

16/3/2021
The ARDUINO cannot draw all the current needed by the DM542T input ports, since the internal opto-couplers on each of the *CLOCK (PULSE)*, *DIRECTION* and *ENABLE* inputs consume 18mA/port. This exceeds the 100mA total limit of the ARDUINO. Thus, a driver interface is required. Because of an old stock of BC107A transistors, we'll use a most simple design, based on emitter followers... space doesn't play any role in this project. However, the transistors invert the input signals. Fortunately, the GRBL firmware allows inverted configuration of most input ports.

After some testing, we see that the GRBL firmware wants to disable the motors, as soon as the routing process has finished. The ARDUINO uses a common *Enable* pin for all of the motors. So, we must join the individual *Enable* ports of the DM542s on the interface board and control them through a single transistor.

The DM542T datasheet recommends twisted pairs of wiring, and specifically advises against daisy chaining. So, each port must have two wires.

We will use a 24V/5A and a second 5V power supply.

This leads to the interface circuit in Fig. 36.

[30] http://www.leadshine.com/UploadFile/Down/DMShm_P.pdf

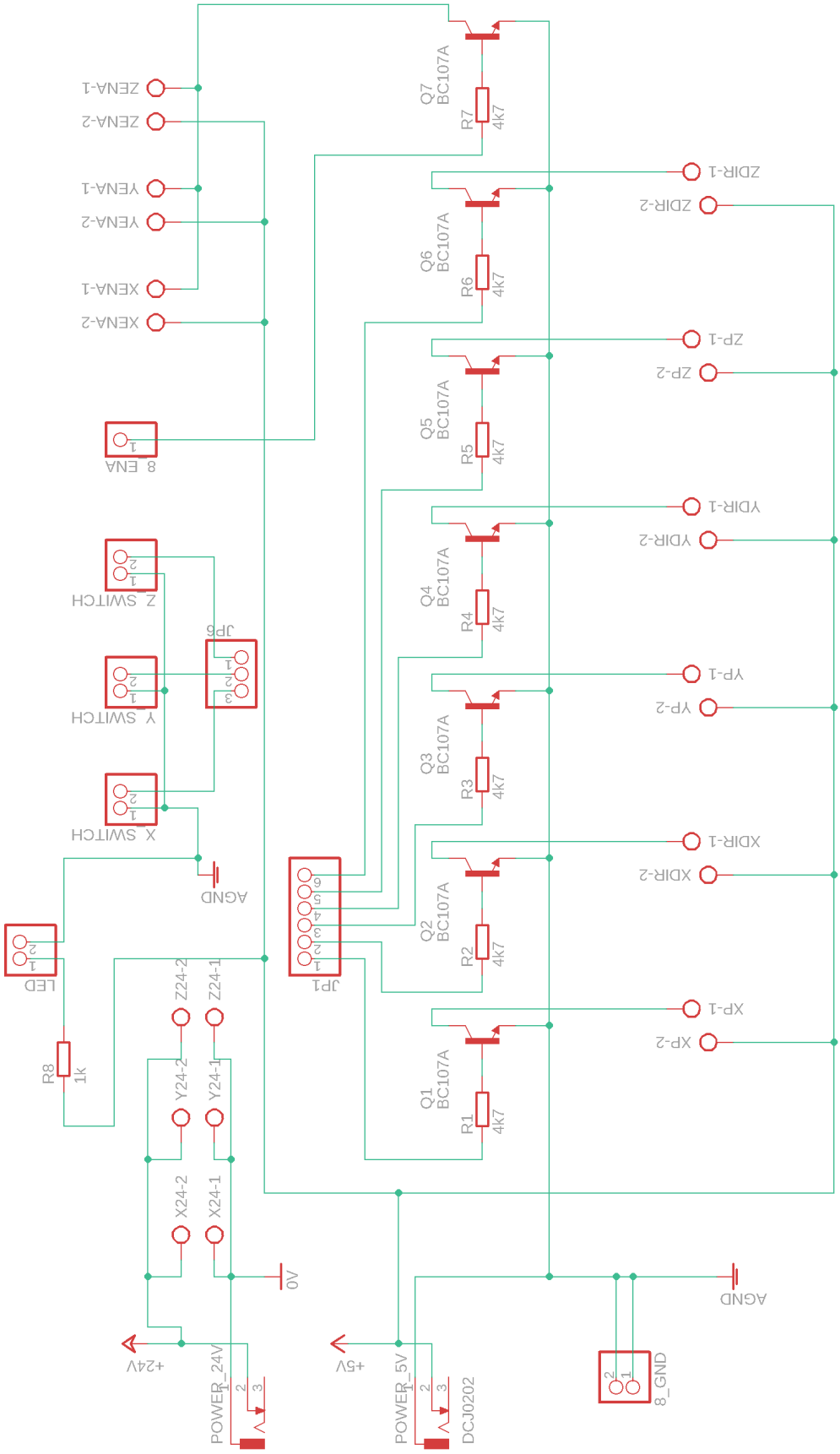


Figure 36: An interface board.

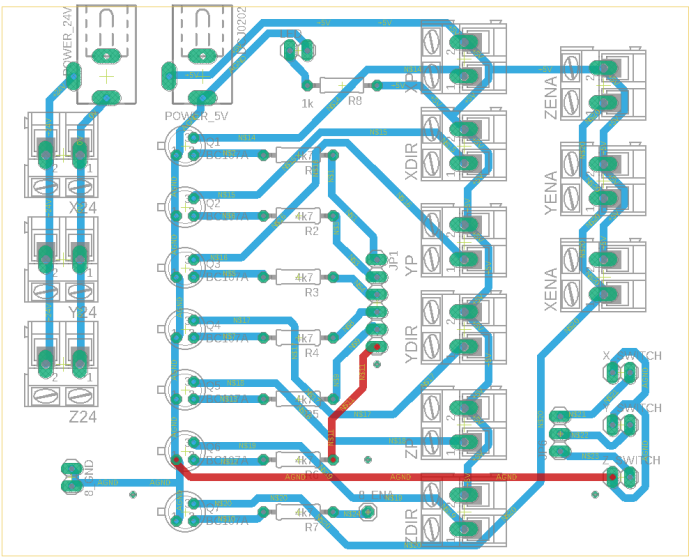


Figure 37: The PC board layout.

XXIII Day 21

17/3/2021

Because we are using two identical barrel connectors for the 24V and 5V power supplies, we deliberately wanted to configured the 5V connector to have +V on the outer pin and -V on the inner pin. However, we erroneously did the same with the 24V supply on the schematics in Fig. 36, and also forgot the protection diode in the 5V circuit. Here the correct picture (Fig. 38) with additional switch on the 24V side.

The reason for this measure is that if ever the user confounds both supplies, the diode blocks the overcurrent to the transistors. We will have a 0.6V voltage drop at the diode. However, this should not disturb the open-collector circuitry.

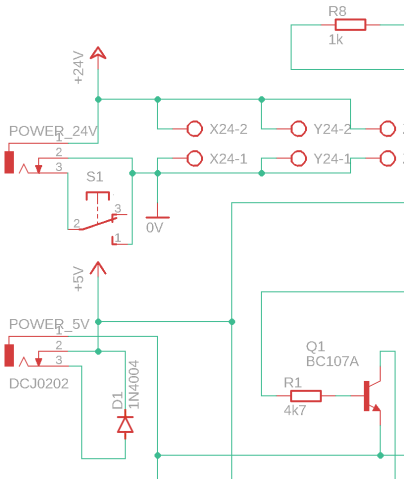


Figure 38: Exchange and protection of power-pins.

XXIV Day 22

20/3/2021

First test with the DM542T

Everything works fine on all thre axes: the enable dispatching, pulse and direction control. Motors run very smoothly and silently. Acceleration and deceleration work perfectly. Excellent overall performances!

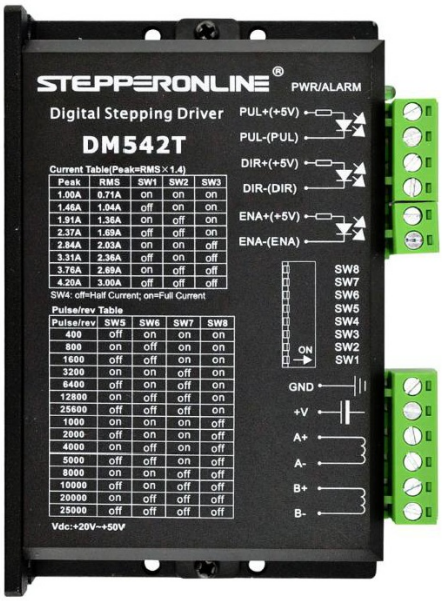


Figure 39: The motor driver module.

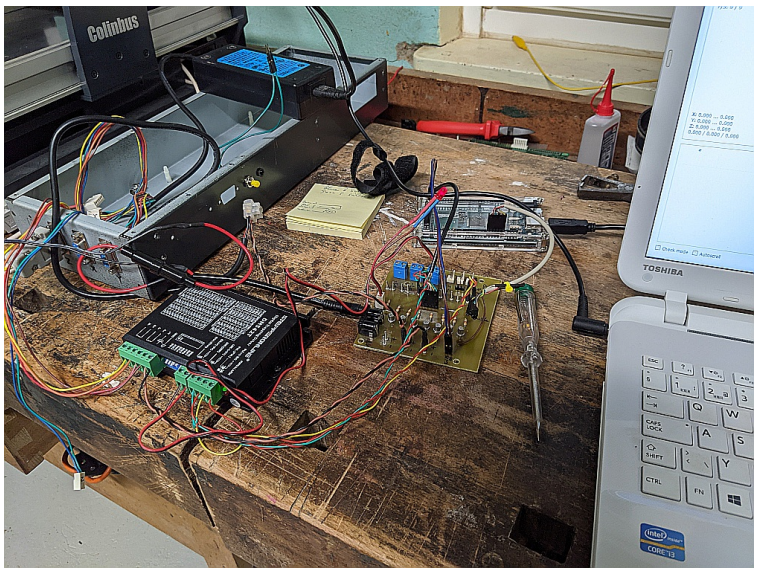


Figure 40: First test with the new DM542T.

Some important information copied from the GRBL site:

https://github.com/gnea/grbl/wiki.

Supported G-Codes in v1.1

- G0, G1: Linear Motions
- G2, G3: Arc and Helical Motions
- G4: Dwell
- G10 L2, G10 L20: Set Work Coordinate Offsets
- G17, G18, G19: Plane Selection
- G20, G21: Units
- G28, G30: Go to Pre-Defined Position
- G28.1, G30.1: Set Pre-Defined Position
- G38.2: Probing
- G38.3, G38.4, G38.5: Probing
- G40: Cutter Radius Compensation Modes OFF (Only)
- G43.1, G49: Dynamic Tool Length Offsets
- G53: Move in Absolute Coordinates
- G54, G55, G56, G57, G58, G59: Work Coordinate Systems
- G61: Path Control Modes
- G80: Motion Mode Cancel
- G90, G91: Distance Modes
- G91.1: Arc IJK Distance Modes
- G92: Coordinate Offset
- G92.1: Clear Coordinate System Offsets
- G93, G94: Feedrate Modes
- M0, M2, M30: Program Pause and End
- M3, M4, M5: Spindle Control
- M7* , M8, M9: Coolant Control
- M56* : Parking Motion Override Control

(*) denotes commands not enabled in config.h by default.

Safety Door Support

Safety door switches are now supported. Grbl will force a feed hold, shutdown the spindle and coolant, and wait until the door switch has closed and the user has issued a resume. Upon resuming, the spindle and coolant will re-energize after a configurable delay and continue.

Commands

- $$and $x=val - View and write Grbl settings
- $# - View gcode parameters
- $G - View gcode parser state
- $I - View build info
- $N - View startup blocks

  are the startup blocks that Grbl runs every time you power on Grbl or reset Grbl. In other words, a startup block is a line of G-code that you can have Grbl auto-magically run to set your G-code modal defaults, or anything else you need Grbl to do everytime you start up your machine. Grbl can store two blocks of G-code as a system default.

  So, when connected to Grbl, type $N and then enter. Grbl should respond with something short like:

  $N0= $N1= ok

  Not much to go on, but this just means that there is no G-code block stored in line $N0 for Grbl to run upon startup. $N1 is the next line to be run. $Nx=line - Save startup block

  To set a startup block, type $N0= followed by a valid G-code block and an enter. Grbl will run the block to check if it's valid and then reply with an ok or an error: to tell you if it's successful or something went wrong. If there is an error, Grbl will not save it.

  For example, say that you want to use your first startup block $N0 to set your G-code parser modes like G54 work coordinate, G20 inches mode, G17 XY-plane. You would type $N0=G20 G54 G17 with an enter and you should see an ok response. You can then check if it got stored by typing $N and you should now see a response like $N0=G20G54G17.

  Once you have a startup block stored in Grbl's EEPROM, everytime you startup or reset you will see your startup block printed back to you, starting with an open-chevron >, and a :ok response from Grbl to indicate if it ran okay. So for the previous example, you'll see:

  Grbl 1.1d ['$' for help] >G20G54G17:ok

  If you have multiple G-code startup blocks, they will print back to you in order upon every startup. And if you'd like to clear one of the startup blocks, (e.g., block 0) type $N0= without anything following the equal sign.

  NOTE: There are two variations on when startup blocks will not run. First, it will not run if Grbl initializes up in an ALARM state or exits an ALARM state via an $X unlock for safety reasons. Always address and cancel the ALARM and then finish by a reset, where the startup blocks will run at initialization. Second, if you have homing enabled, the startup blocks will execute immediately after a successful homing cycle, not at startup.

- $C - Check gcode mode
- $X - Kill alarm lock
- $H - Run homing cycle
- $J=line - Run jogging motion
- $RST=$, $RST=#, and $RST=*- Restore Grbl settings and data to defaults

These commands are not listed in the main Grbl $ help message, but are available to allow users to restore parts of or all of Grbl's EEPROM data. Note: Grbl will automatically reset after executing one of these commands to ensure the system is initialized correctly.

$RST=$: Erases and restores the $$ Grbl settings back to defaults, which is defined by the default settings file used when compiling Grbl. Often OEMs will build their Grbl firmwares with their machine-specific recommended settings. This provides users and OEMs a quick way to get back to square-one, if something went awry or if a user wants to start over. $RST=# : Erases and zeros all G54-G59 work coordinate offsets and G28/30 positions stored in EEPROM. These are generally the values seen in the $# parameters printout. This provides an easy way to clear these without having to do it manually for each set with a G20 L2/20 or G28.1/30.1 command. $RST=* : This clears and restores all of the EEPROM data used by Grbl. This includes $$ settings, $# parameters, $N startup lines, and $I build info string. Note that this doesn't wipe the entire EEPROM, only the data areas Grbl uses. To do a complete wipe, please use the Arduino IDE's EEPROM clear example project.

- $SLP - Enable Sleep Mode
- 0x18 (ctrl-x) : Soft-Reset (??? not working in Candle from the console!!!)
- ? : Status Report Query
- ∼ : Cycle Start / Resume
- ! : Feed Hold
- 0x84 : Safety Door
- 0x85 : Jog Cancel

(For details, see web-site!)

Grbl settings (with default values)

$0=10 Step pulse, microseconds

$1=25 Step idle delay, milliseconds

$2=0 Step port invert, mask

$3=0 Direction port invert, mask

$4=0 Step enable invert, boolean

$5=0 Limit pins invert, boolean

$6=0 Probe pin invert, boolean

$10=1 Status report, mask

$11=0.010 Junction deviation, mm

$12=0.002 Arc tolerance, mm

$13=0 Report inches, boolean

$20=0 Soft limits, boolean

$21=0 Hard limits, boolean

$22=1 Homing cycle, boolean

$23=0 Homing dir invert, mask

$24=25.000 Homing feed, mm/min

$25=500.000 Homing seek, mm/min

$26=250 Homing debounce, milliseconds

$27=1.000 Homing pull-off, mm

$30=1000. Max spindle speed, RPM

$31=0. Min spindle speed, RPM

$32=0 Laser mode, boolean

$100=250.000 X steps/mm

$101=250.000 Y steps/mm

$102=250.000 Z steps/mm

$110=500.000 X Max rate, mm/min

$111=500.000 Y Max rate, mm/min

$112=500.000 Z Max rate, mm/min

$120=10.000 X Acceleration, mm/sec^2

$121=10.000 Y Acceleration, mm/sec^2

$122=10.000 Z Acceleration, mm/sec^2

$130=200.000 X Max travel, mm

$131=200.000 Y Max travel, mm

$132=200.000 Z Max travel, mm

$2 Step port invert, mask

This setting inverts the step pulse signal. By default, a step signal starts at normal-low and goes high upon a step pulse event. After a step pulse time set by $0, the pin resets to low, until the next step pulse event. When inverted, the step pulse behavior switches from normal-high, to low during the pulse, and back to high. Most users will not need to use this setting, but this can be useful for certain CNC-stepper drivers that have peculiar requirements. For example, an artificial delay between the direction pin and step pulse can be created by inverting the step pin.
This invert mask setting is a value which stores the axes to invert as bit flags. You really don't need to completely understand how it works. You simply need to enter the settings value for the axes you want to invert. For example, if you want to invert the X and Z axes, you'd send $2=5 to Grbl and the setting should now read $2=5 (step port invert mask:00000101).

Setting Value	Mask	Invert X	Invert Y	Invert Z
0	00000000	N	N	N
1	00000001	Y	N	N
2	00000010	N	Y	N
3	00000011	Y	Y	N
4	00000100	N	N	Y
5	00000101	Y	N	Y
6	00000110	N	Y	Y
7	00000111	Y	Y	Y

Error codes

ID	Error Code Description
1	G-code words consist of a letter and a value. Letter was not found.
2	Numeric value format is not valid or missing an expected value.
3	Grbl '$' system command was not recognized or supported.
4	Negative value received for an expected positive value.
5	Homing cycle is not enabled via settings.
6	Minimum step pulse time must be greater than 3usec
7	EEPROM read failed. Reset and restored to default values.
8	Grbl '$' command cannot be used unless Grbl is IDLE. Ensures smooth operation during a job.
9	G-code locked out during alarm or jog state
10	Soft limits cannot be enabled without homing also enabled.
11	Max characters per line exceeded. Line was not processed and executed.
12	(Compile Option) Grbl '$' setting value exceeds the maximum step rate supported.
13	Safety door detected as opened and door state initiated.
14	(Grbl-Mega Only) Build info or startup line exceeded EEPROM line length limit.
15	Jog target exceeds machine travel. Command ignored.
16	Jog command with no '=' or contains prohibited g-code.
17	Laser mode requires PWM output.
20	Unsupported or invalid g-code command found in block.
21	More than one g-code command from same modal group found in block.
22	Feed rate has not yet been set or is undefined.
23	G-code command in block requires an integer value.
24	Two G-code commands that both require the use of the XYZ axis words were detected in the block.
25	A G-code word was repeated in the block.
26	A G-code command implicitly or explicitly requires XYZ axis words in the block, but none were detected.
27	N line number value is not within the valid range of 1 - 9,999,999.
28	A G-code command was sent, but is missing some required P or L value words in the line.
29	Grbl supports six work coordinate systems G54-G59. G59.1, G59.2, and G59.3 are not supported.
30	The G53 G-code command requires either a G0 seek or G1 feed motion mode to be active. A different motion was active.
31	There are unused axis words in the block and G80 motion mode cancel is active.
32	A G2 or G3 arc was commanded but there are no XYZ axis words in the selected plane to trace the arc.
33	The motion command has an invalid target. G2, G3, and G38.2 generates this error, if the arc is impossible to generate or if the probe target is the current position.
34	A G2 or G3 arc, traced with the radius definition, had a mathematical error when computing the arc geometry. Try either breaking up the arc into semi-circles or quadrants, or redefine them with the arc offset definition.
35	A G2 or G3 arc, traced with the offset definition, is missing the IJK offset word in the selected plane to trace the arc.
36	There are unused, leftover G-code words that aren't used by any command in the block.
37	The G43.1 dynamic tool length offset command cannot apply an offset to an axis other than its configured axis. The Grbl default axis is the Z-axis.
38	Tool number greater than max supported value.

XXV Day 23

21/3/2021

Grbl configuration

We now started to configure the Arduino:

- $3=6 (directions Y and Z inverted) in order to have the machine fix its origin at the position defined by the limit switches, which GRBL identifies as $O(-400, 0, 0)$, or more precisely $O(-398, -2, -2)$, because of the homing pull-off by 2mm (set with $27=2). Why does GRBL choose -400 for the **x** origin? We set the maximum value for X to 400 ($130=400). However, as the homing direction is not inverted for X, -400 is the home position on that axis. We will have to respect the **right hand rule**. Note that this means that Y values are negative on the working area.

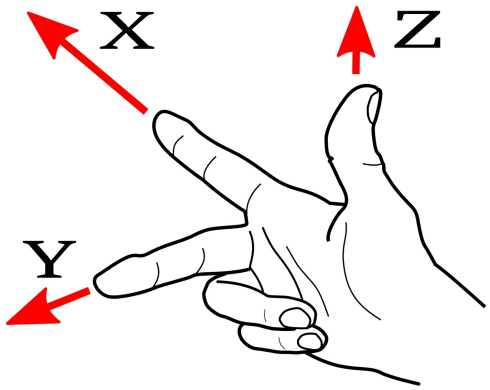


Figure 41: Left hand rule (Courtesy: https://github.com/gnea/grbl/wiki/Grbl-v1.1-Configuration)

- $4=1 (step enable inverted!)
- However, we don't need step pulse inversion, since the DM542T can handle everything correctly. (We'll have to play here, to find out, if ever setting $5=7 makes any difference. So far we didn't notice any issue with the default configuration.)
- $100,101,102=133.333 steps/mm
- $120,121,122=50mm/s^2 (acceleration)
- $130=400mm (max X)
- $131=300mm (max Y)
- $132=100mm (max Z)
- $21=1 (hard limits enabled): **This caused serious troubles.:** In fact, the machine erratically switches into ALARM mode. Initially the switches were hard-configured in *closed* state. First, we changed $5=1. This didn't solve the problem. Then we checked all the connections. However, as soon as the motor are powered, the switches are triggered without any physical contact. So, the reason must be instability of the Arduino digital input pins, because TTL levels might change very fast on variations of the power level, or on induced currents. Adding small capacitors (330pF) directly to the switches helped, but was not reliable enough. After some googling, we found this solution by an engineer named Tom Trabant:

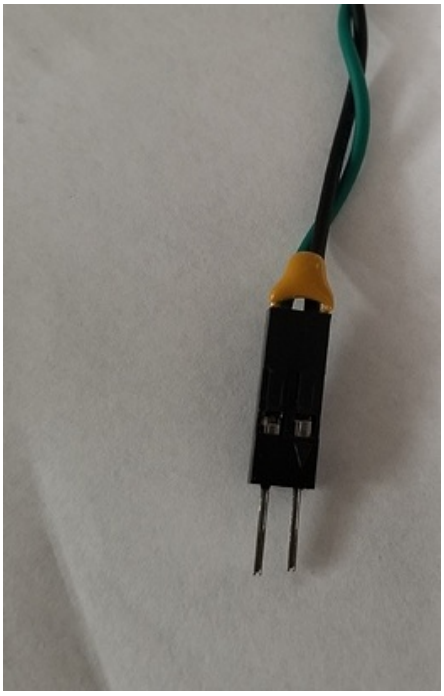


Figure 42: Adding a 10nF capacitor to the switch connector (just slit into the connector, no soldering) solves the nasty switch triggering issue.

XXVI Day 24

21/3/2021

Grbl configuration (II)

- $21=1 (Homing enabled)
- $23=1 (Homing direction invert mask for X, otherwise the x motor turns in the wrong direction while homing.)
- $27=2 (Homing pull-off is 2mm)
- $20=1 (Soft limits are set $\implies$ GRBL will not accept any command that would move the machine off-limits.
- $110,111,112=1000 (max rate on all axes is 1000 mm/min) This is more than we observed that the motors can handle. However, measurements were made with the original Colinbus control board. It seems that the DM542T can handle higher frequencies much better.

Grbl alarm codes

ID	Alarm Code Description
1	Hard limit triggered. Machine position is likely lost due to sudden and immediate halt. Re-homing is highly recommended.
2	G-code motion target exceeds machine travel. Machine position safely retained. Alarm may be unlocked.
3	Reset while in motion. Grbl cannot guarantee position. Lost steps are likely. Re-homing is highly recommended.
4	Probe fail. The probe is not in the expected initial state before starting probe cycle, where G38.2 and G38.3 is not triggered and G38.4 and G38.5 is triggered.
5	Probe fail. Probe did not contact the workpiece within the programmed travel for G38.2 and G38.4.
6	Homing fail. Reset during active homing cycle.
7	Homing fail. Safety door was opened during active homing cycle.
8	Homing fail. Cycle failed to clear limit switch when pulling off. Try increasing pull-off setting or check wiring.
9	Homing fail. Could not find limit switch within search distance. Defined as 1.5 * max_travel on search and 5 * pulloff on locate phases.
10	Homing fail. On dual axis machines, could not find the second limit switch for self-squaring.

Switch triggering issue

This issue was insufficiently solved with 10nF capacitors. We changed to 10μF, and everything works fine, except for the start-up phase, where always an *Alarm 1* is triggered. Obviously the ARDUINO configures the ports used for the switches from output to input at start. There seems no settling time in the configuration, so charging the new capacitors throws an interrupt on these ports, triggering the alarm message. Unfortunately the operator must manually reset GRBL in the CANDLE software and unlock the machine. Then only it is ready for use.

Work coordinates versus machine coordinates

The CANDLE GUI[31] has excellent features well adapted to GRBL, among which the possibility to reset the work coordinates. This makes G-programming very easy. Here the first result:

[31] https://github.com/Denvi/Candle.

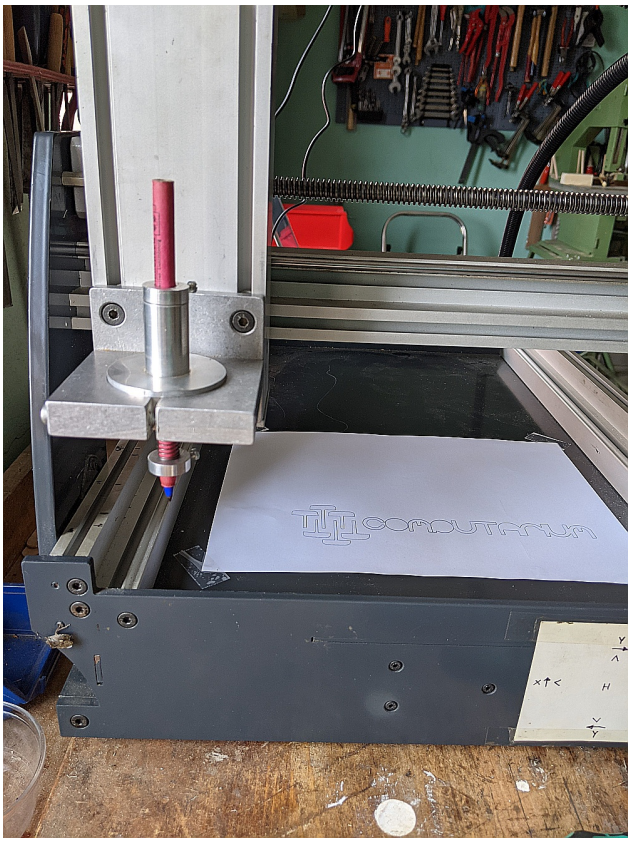


Figure 43: First test with the machine.

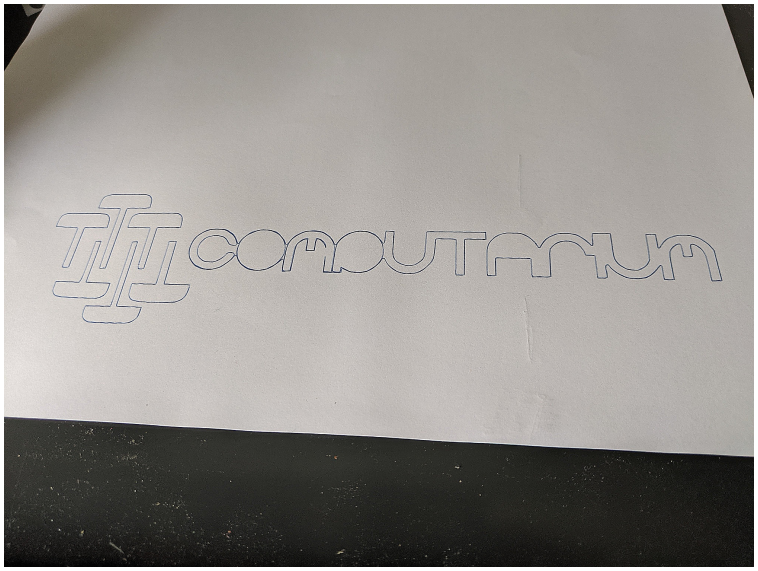


Figure 44: Excellent result.

Main board

Power-up procedure

1. Plug the 5V power supply to 220V.
2. Plug the connector to the main board. (The red LED should now glow.)
3. Plug the 24V power supply to 220V.
4. Plug the connector to the main board.
5. Switch on 24V power. (The COLINBUS motors should shaffle for a short time.)
6. Connect the USB plug from the PC.
7. Run CANDLE GUI (or any other compatible GUI) and let ARDUINO-GRBL take control over the machine.
8. Reset and unlock the machine manually in the CANDLE software.
9. Start the homing procedure.
10. Rezero all axes (Zero-buttons in the CANDLE software).

Figure 45: Main board.

Final notes:

- The *Abort* switch will be replaced with a push button, because the ARDUINO only reacts on falling edge.
- The *Safety door* function isn't very clear. In our tests, GRBL doesn't return to the G-Code program, as it should. We need some experiencing here.
- Using the Proxxon drilling machine that Prof. Mootz used with the COLINBUS, *Alarm 1* was sometimes triggered at the moment of switching on the spindle. We found out that this reaction depends on the position of the machine on the z-axis. We suppose that electromagnetic currents are induced in the z-switch wiring, dispite the capacitor. Changing the position solves the problem.
- We broke a couple of drills, before learning the correct start-up procedure of the machine. Mostly, we forgot re-zeroing the axis in CANDLE.
- There was some backlash in the ball-bearing of the z-axis guide-way. This needed some calibration.

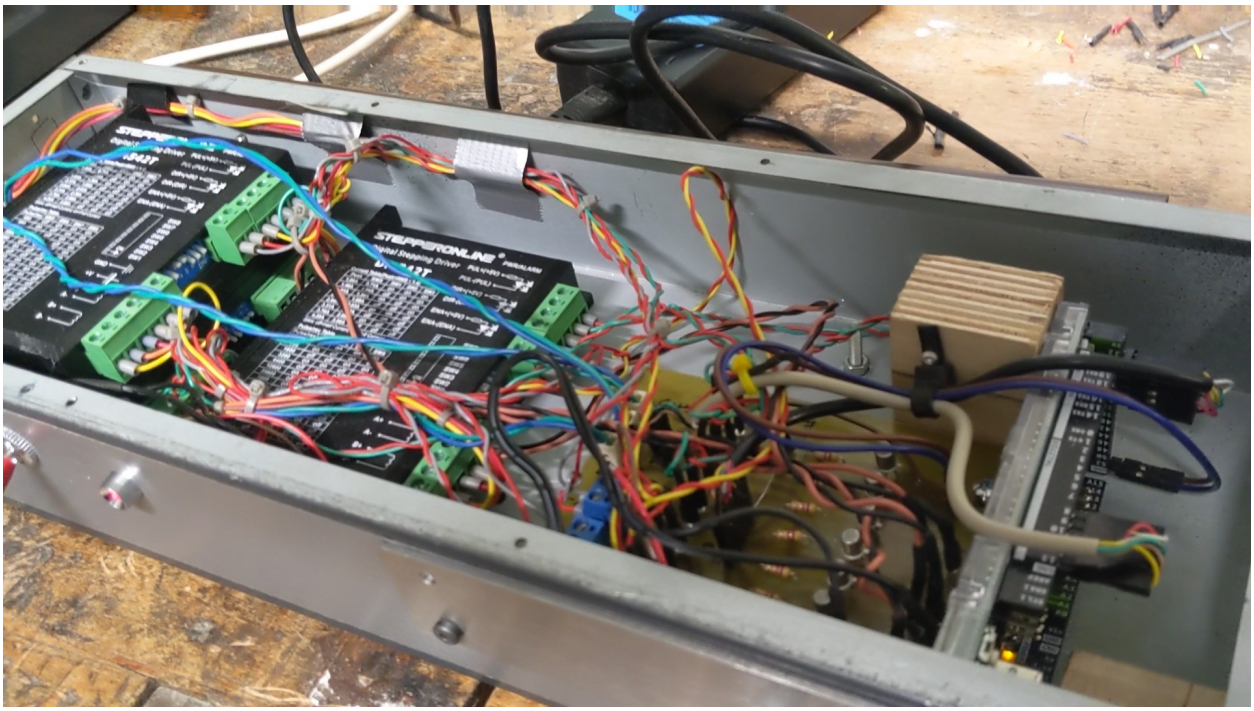


Figure 46: Internals.

This finishes the project. Thanks for all the support to colleague Prof. Francis Massen.