



Technical Documentation

**48 Output Programmable
Wired Pyrotechnic Ignition System
based on the ATTINY 861**

by Markus K. 2023

Contents

1	Introduction	3
1.1	Legal Disclaimer	3
1.2	Concept	4
2	Hardware	5
2.1	The Ignition Voltage Generator	5
2.1.1	Circuit	5
2.1.2	How does the circuit work?	6
2.1.3	Testing	7
2.1.3.1	Results	7
2.1.3.2	Conclusion	8
2.2	Controller	9
2.2.1	Circuit	9
2.2.2	Components of the Controller	10
2.2.2.1	Peripheral Managing and Controlling Logic	10
2.2.2.2	Attiny 861	10
2.2.2.3	USB serial communication	10
2.2.2.4	Arm safety circuit	11
2.2.2.5	Additional circuits	11
2.2.3	Sockets and Plugs	11
2.2.4	Circuit board	12
2.2.5	Housing of the Controller	13
2.3	Trigger	15
2.3.1	Circuit	15
2.3.2	How does the trigger work?	16
2.3.3	Housing of the Trigger	16
2.4	Ignition Modules	17
2.4.1	Circuit	17
2.4.2	How does the Ignition Module work?	18
2.4.2.1	Ignition Circuit	18
2.4.2.2	Controlling the Ignition Circuits	19
2.4.2.3	Additional circuits	19
2.4.3	Housing of the Ignition Modules	20
3	Firmware	21
3.1	Pinout	21
3.2	Setup	22
3.3	Ignition of a Port	23
3.3.1	Ignition Module addressing	23
3.3.2	Serial Data Transmission	24
3.4	State Machine	25
3.4.1	State Variables	26
3.4.1.1	PG (Program Mode)	27
3.4.1.2	TF (Trigger Fire)	27
3.4.1.3	DN (Done)	27
3.4.1.4	WT (Wait)	27
3.4.2	States	28
3.4.2.1	<i>PG_MODE</i>	28
3.4.2.2	<i>INIT</i>	28
3.4.2.3	<i>NO_TRIGGER</i>	28
3.4.2.4	<i>ARMED</i>	28
3.4.2.5	<i>WAIT</i>	28
3.4.2.6	<i>FIRE</i>	28
3.5	Program Mode	28

3.5.1	Entering the Program Mode	28
3.5.2	Instruction Set	28
3.5.3	Instruction Decoding	28
3.6	Programming the Attiny861	28
4	Appendix	29
4.1	Pricing	29
4.2	Conclusion	29
4.3	Firmware Code	30
4.4	Recognitions	37

1 Introduction

The ZK-48 ignition (German "Zündkasten 48 Ausgänge") system was developed because of the lack of affordable programmable firing systems. Therefore the goal was set for a low cost, yet reliable and robust device, that works in any condition. This document provides all information surrounding this project and anyone with basic knowledge in electronics should be able to rebuild it for them self. Although this should not be viewed as an instruction on how to build this device, the author hopes it will be useful to a pyrotechnic and or electronics enthusiast. The finished device is shown in Figure 1 with the trigger on the right and one ignition module on the left.

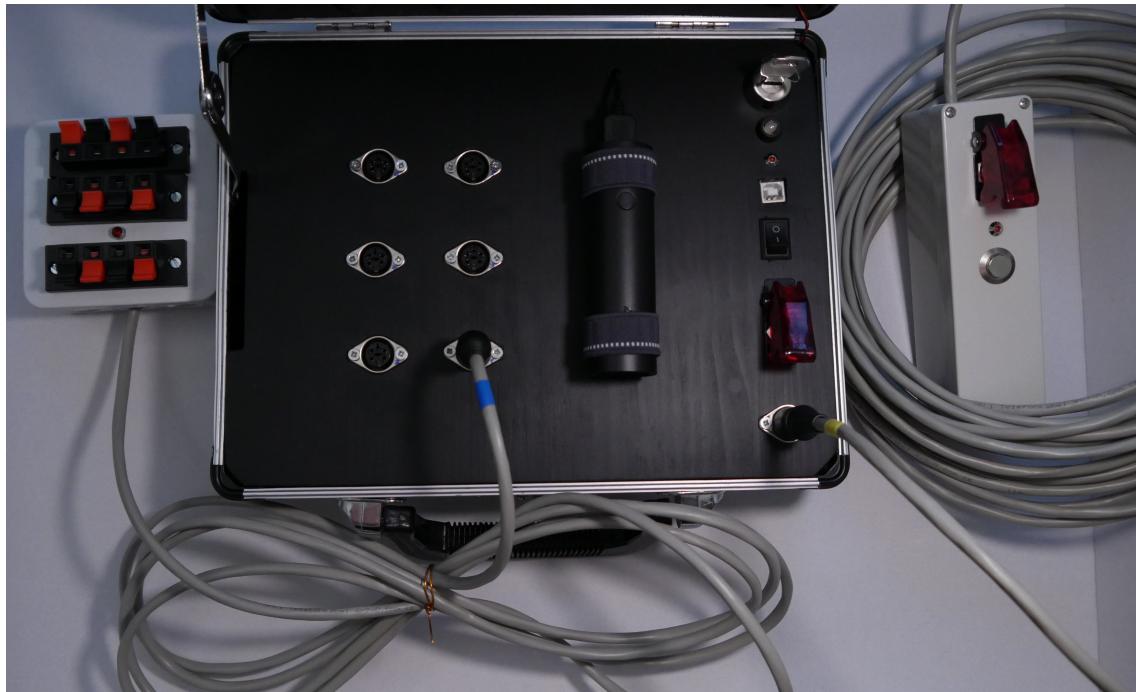


Figure 1: The resulting complete system with trigger and one ignition module.

1.1 Legal Disclaimer

Please check your local laws before you decide on building a system such as this, by your self! Some countries may have laws prohibiting the construction of home-made ignition systems. So get informed! The author of this document does not endorse illegal activities nor can he be held responsible for any person's illegal activities. According to Austrian law, to which the author is subject, the construction and use of self-made ignition systems in the private and professional sphere, is not regulated! This is not legal advice, so do your own research!

In this document electronic ignitable matches are refereed to as *Bridge wire detonators* or short *detonators* just for readability purposes. But this does not mean actual detonators (also in English called *blasting caps*) are used, as they contain high explosives which are not legal without license in Austria! The author refers to the in German called "A-Typ Brückenzünder"¹ which are legal to purchase by any person over 18 years of age. Again, this is not legal advise and do proper research on your local laws!

¹<https://de.wikipedia.org/wiki/Br%C3%BCckenz%C3%BCnder>

1.2 Concept

The device is divided into four units: controller, ignition voltage generator, trigger and ignition modules (See Figure 2), whereby the controller makes up the center piece. On board the controller are the main on/off key-switch, the arm switch, a USB port for programming, status LEDs and sockets for the trigger and ignition modules, as well as the micro controller (μ C) which controls everything. The trigger is connected by cable to the controller and got mounted a arm switch and the trigger button for setting of the firework. There are six ignition modules and each module is able to ignite eight bridge wire detonators (A-Type Only). The modules are also connected by cable to the controller. Therefore the complete system is capable of setting of 48 detonators.

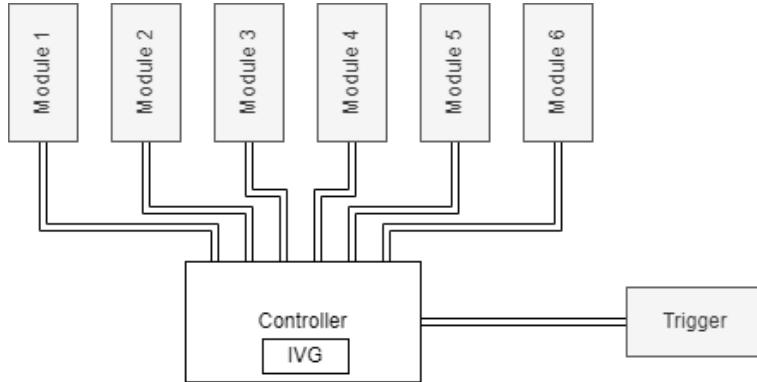


Figure 2: Basic layout of the three components controller, trigger and modules.

The system operates by the fire-and-forget principle, by which the user arms both the controller and the trigger and after pressing the trigger button the controller will automatically ignite the firework in the programmed sequence. No user input is needed after setting off. When the trigger is pressed the system goes into a 10s delay phase in order for the user to get to safety. After the delay phase is finished, the ignition phase is entered, where the programmed sequence will play through until the predetermined endpoint is reached. The trigger can be replaced with a RF-trigger although this is not recommended due safety and legal concerns in some countries. The delay period also serves as a fail safe, because in the case of an accidental triggering of the system, the user is able to abort the start by disarming the system. During the ignition phase it is also possible to halt the program by disarming the system, although re-triggering will repeat the delay phase.

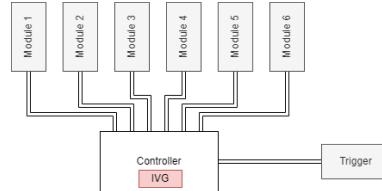
The ignition sequence is programmed over USB (See ??). Programming the sequence can be accomplished by directly connecting to the serial port and typing the commands by hand or by using the Software provided (See ??).

As the device is most likely to be placed in an open-air field, there is no way for it to be connected to mains power, therefore it has to be operated by batteries. However, the usage of rechargeable batteries such as LiPo-batteries was not desirable for this project, due cost issues and the additional requirement of protection circuits. A better alternative was found by using common 5V portable powerbanks for smart phones. Those already provide steady 5V with build-in protective mechanisms. Furthermore, nowadays many people use powerbanks and if the user forgets to charge or forgets the powerbank altogether, there is a high chance some will be able to provide one as replacement. A powerbank can be charged by a simple micro USB cable which is also very common and removes the need for a specialized charger.

2 Hardware

As explained in section 1.2, the device is split into four parts. This section will explore each separately by looking into the design choices that were made.

2.1 The Ignition Voltage Generator



The ignition voltage generator shown in Figure 3 works by the basic principle of a step-up/boost converter, by storing energy in form of a magnetic field inside an inductor and releasing the energy as a current into a capacitor. Repeat this process at high frequency and a high voltage is produced at the output compared. For a better explanation see the document about this topic by *Texas Instruments*².

2.1.1 Circuit

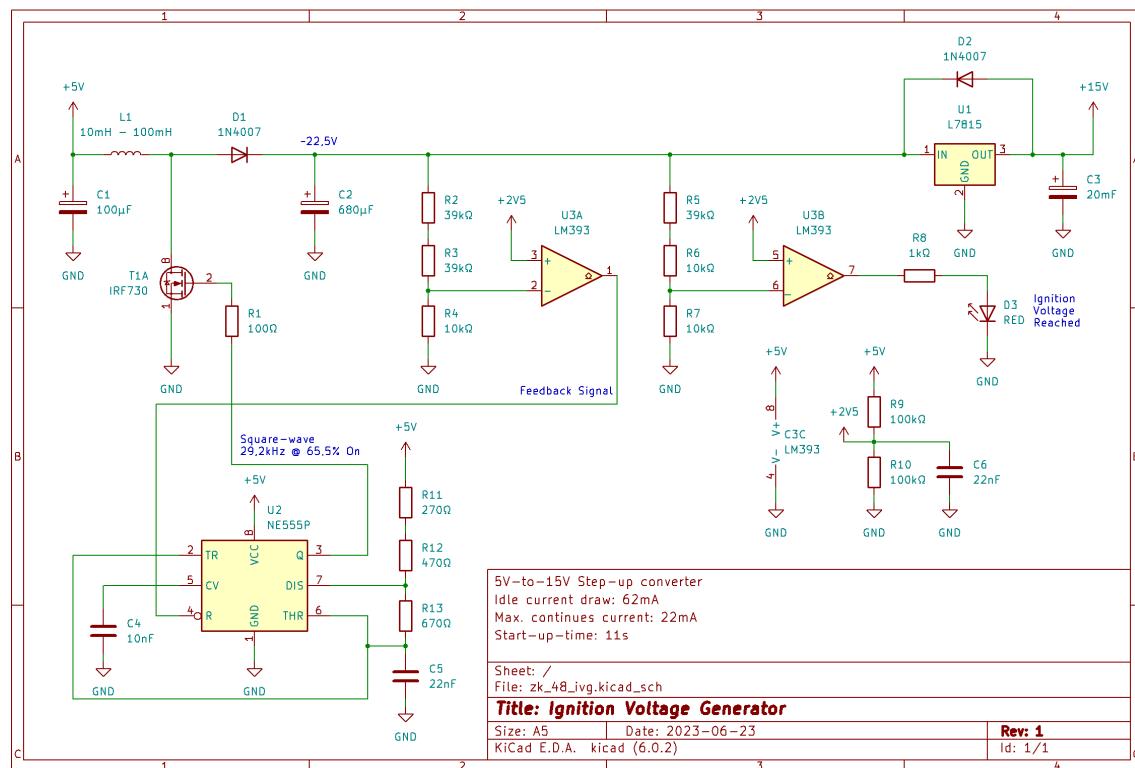


Figure 3: Circuit of the step-up converter that generates 15V DC for the ignition voltage

²<https://www.ti.com/lit/an/snva731/snva731.pdf>

2.1.2 How does the circuit work?

The circuit shown in Figure 3 is based on the popular NE555³, the dual comparator LM393⁴ and the fixed 15V linear voltage regulator L7815⁵. The NE555 is used as a square wave oscillator to generate a 29,2kHz 5V_{pp} signal with a 65,5% on-time. This signal drives the IRF730⁶ N-channel MOSFET, which switches current through the inductor L_1 and therefore charging the capacitor C_2 . This voltage is named the intermediate voltage.

Through a voltage divider the voltage at C_2 is stepped down by a factor of $\frac{1}{9}$ and compared to a 2,5V reference voltage by $U3A$. This results in a 5V output signal at the output of $U3A$, if the capacitors C_2 voltage is lower than 22,5V. This signal is called the feedback signal used to turn off the NE555, if the intermediate voltage has reached 22,5V. Using schmitt-trigger would result in a oscillating feedback signal, which is undesirable in this configuration. If the intermediate voltage reaches 22,5V, the feedback signal will no longer be 5V or 0V, instead it will drop to a constant 2,5V. This is expected, but will result in unpredictable behaviour by the NE555, as it expects a binary reset signal. This seems like a problem, but in reality will result in the NE555 output voltage to drop below the on-voltage of the IRF730, thus turning it off. It is also observable that the frequency and on-time of the NE555 output is rising, but this has no effect, because the voltage already dropped significantly to turn off the MOSFET.

The intermediate voltage is then converted by the L7815 to stable 15V, which charges the large 20mF ignition capacitor C_3 . This voltage is called the ignition voltage. Equation (1) calculates the total energy stored in the system with all six modules connected (Thus the total capacitance being 26mF, but read more in ??) which equates to around 3J and therefore does not pose any danger to life⁷. Touching the fully charged capacitors with wet hands did not result in any shock or pain.

$$W_{el} = \frac{U^2 \cdot C_{tot}}{2} = \frac{(15V)^2 \cdot 26mF}{2} = 2,925J \quad (1)$$

The second comparator $U3B$ in the LM393 package was used to turn on a red LED D_3 whose purpose is to indicate whether the intermediate voltage is bigger than 14,75V. This shows the user if the system is ready for operation and if ignition voltage is present.

Note: Please note that the ignition voltage generator(as well as everything else in this document) was designed by the author. The ignition voltage generator is by no means ideal nor optimized. This was the first attempt at creating a step-up convert from scratch with components that were on hand. It does the job well (See section 2.1.3), but any bought step-up convert will do just fine and most likely better.

³<https://www.ti.com/lit/ds/symlink/ne555.pdf>

⁴<https://www.ti.com/lit/ds/symlink/lm393.pdf>

⁵<https://www.st.com/resource/en/datasheet/178.pdf>

⁶<https://www.vishay.com/docs/91047/91047.pdf>

⁷https://www.ehss.vt.edu/programs/ELR_capacitors.php

2.1.3 Testing

For testing the ignition voltage generator displayed in Figure 3, a resistor with 4.7Ω was placed into port 1 on module 1. This resistance is equal to two bridge wire detonators connected in series which makes sure, the system is tested in a realistic setting as often more than one detonator is used. Figure 4 shows the complete test set-up, with the pulse generator symbolically representing the controller. The system was programmed to fire(a fire pulse is $10ms$) port 1 on module 1 eight times with a $10ms$ delay between each pulse.

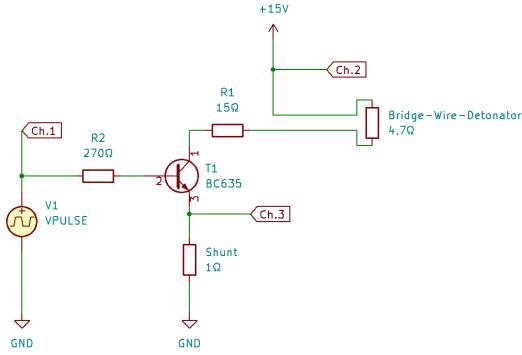


Figure 4: Circuit for testing stability of ignition voltage and current though a simulated bridge wire detonator.

2.1.3.1 Results

The first test consisted of 8 pulses to simulate fast consecutive firing of pyrotechnic single-shots⁸ or other pyrotechnic effects. Normally firing the same port multiple times does not make sense, but for testing purposes this is equivalent to firing eight separate ports. The resulting waveform was captured with the Quad DSO (digital storage oscilloscope) *Rigol DS1054z*⁹.

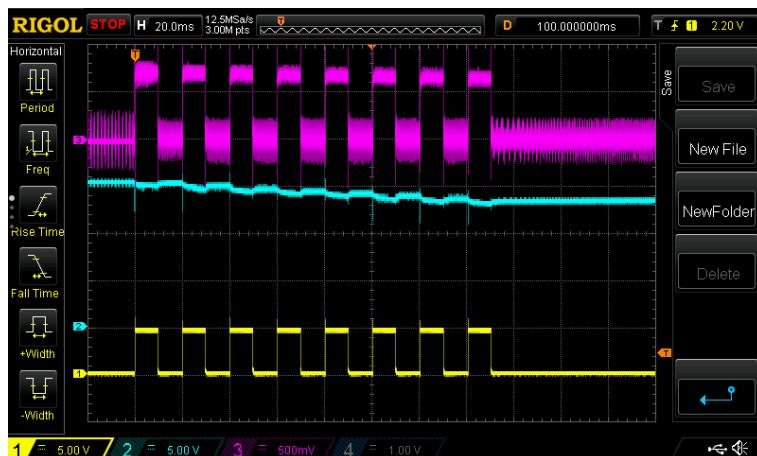


Figure 5: Recorded waveforms by the oscilloscope of eight pulses in close up.

The main objective of this test is to confirm that rapid firing does not affect the system ability to ignite further detonators. This can evidently be confirmed by looking at the curve of Ch.3 and Ch.2 in Figure 5. The ignition voltage is represented by Ch.2 which drops by only $2V$, thus being neglectable. By looking at Ch.3, which shows the current through the bridge wire detonator, its clearly recognizable that current is above $700mA$, even at the last pulse. Past testing showed, bridge wire detonators already ignite at around $300mA - 400mA$. This confirms the efficacy and the systems ability to continuously ignite pyrotechnic effects at a high rate.

⁸<https://www.youtube.com/watch?v=UgVG9NcA5CM>

⁹https://www.battronix.com/pdf/Rigol/UserGuide/DS1000Z_UserGuide_EN.pdf

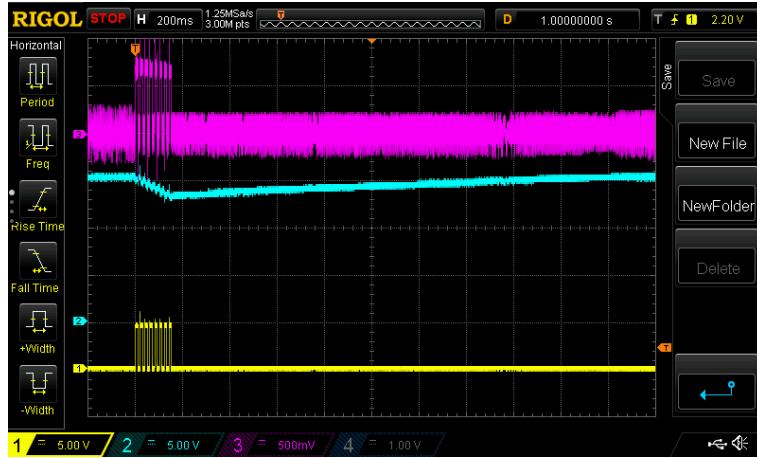


Figure 6: Recorded waveforms by the oscilloscope of eight pulses, until ignition voltage returns to 15V.

In Figure 6 the time was measured for the ignition voltage to return back to 15V after firing eight times. This time amounts to about 2s which is acceptable, but could be improved.

The third test focused on finding the absolute limit of the system. To achieve this objective the ZK-48 was programmed to use all its 48 sequence slots and fire port 1 on module 1 48 times.



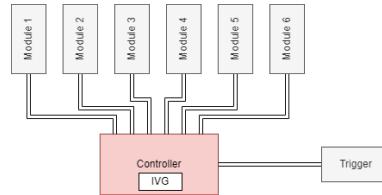
Figure 7: Recorded waveforms by the oscilloscope of 48 firing pulses.

The number of maximal firing pulses was determined by counting the number of pulses until the current through the detonator fell below 500mA. This current value was chosen because it is the average between the lowest recorded current at which a detonator ignites(300mA) and the current at the first ignition pulse(700mA). By this definition, the maximal number of consecutive firing pulses with 10ms delay is about 24.

2.1.3.2 Conclusion

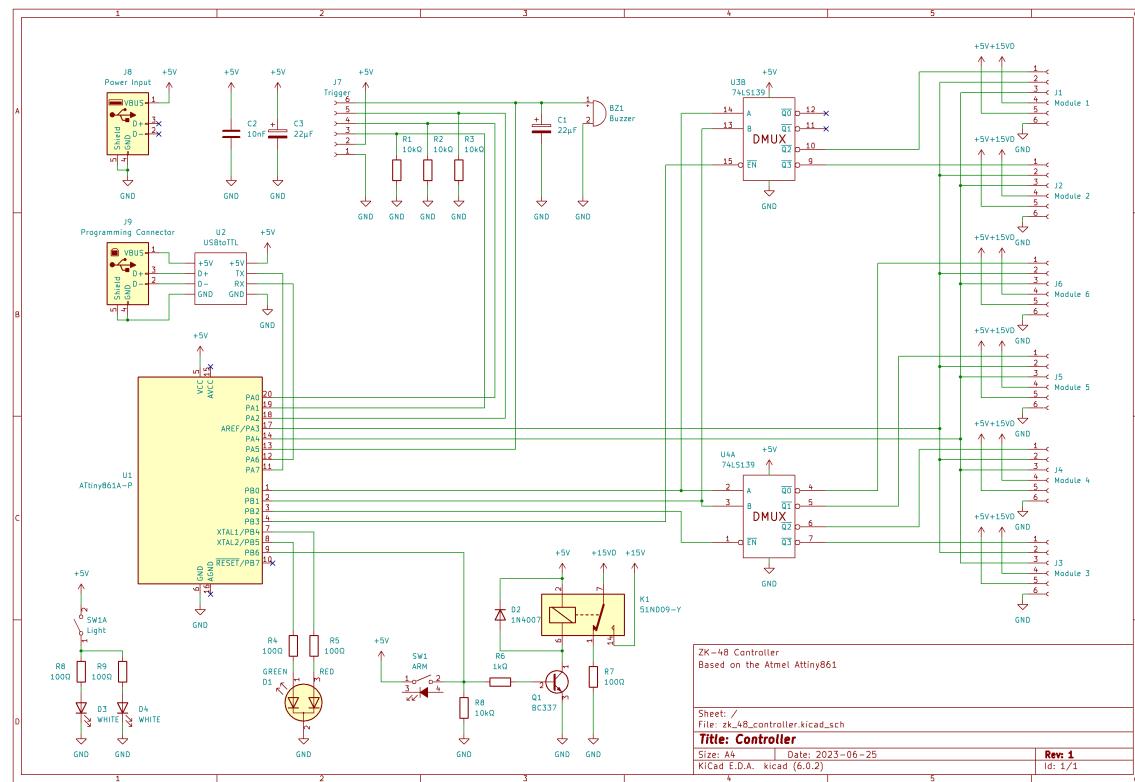
The ignition voltage generator is able to provide a steady voltage, even under load(See Figure 4) and is able to ignite 24 bridge wire detonators with 10ms delay. This is just theoretical and needs to be confirmed with real detonators, however this is already a good indicator for proper operation. Although the generator performs well, it has many flaws and should be subject of rigorous optimization. For example, the ignition voltage should ideally not drop at all and remain constant even if continues 700mA are drawn. By using a bought step-up converter, with proper control logic, such as current regulation, instead of a simple feedback loop, most problems would be rectified. The projects goal is not buying electronic modules and soldering them together, instead this is an exploration of circuit design and should be viewed as a learning experience.

2.2 Controller



The controller is responsible for interpreting all user inputs such as the arm switch, the trigger signal and more, which are processed by the µC to correctly control all modules and the status LEDs. The ignition voltage generator is also soldered onto the same physical circuit board as the controller, but more about that in Section 2.2.4.

2.2.1 Circuit



2.2.2 Components of the Controller

The controller consists of the peripheral managing and controlling logic, the USB serial communication and the arm safety circuit.

2.2.2.1 Peripheral Managing and Controlling Logic

This part of the circuit is located on the right side of the circuit depicted in Figure 8. Its purpose is the selection of the correct ignition module to fire. To understand how this works, it is important to first explain how the µC fires a port on a module: In Figure 8 notice that all ignition modules sockets DATA and CLK lines are connected together(For the pin-out see Section 2.2.3). First, the µC serially transmits to all modules shift registers the port that is about to be fired. In the second step, the µC selects one out of the six modules, which then fires the port that was set in the first step.

The selection process of the ignition module is accomplished by two demultiplexer *U4A* and *U3B* which are both contained in the dual DMUX 74LS139¹⁰. Instructions are given to the DMUX by the µC. Circuit design only allows to select one module at a time, thus the system being unable to fire to ports on different modules at the same time. Currently firmware does not support simultaneous firing of two ports anyway. This is a disadvantage, but is manageable by programming the system to fire two ports with no delay, which results in a 10ms between each firing. Our human eyes cannot pick up on such short delays and pyrotechnics are also not timed perfectly, therefore this does not impose a problem.

The socket of the external trigger also falls into peripheral circuitry, but it does not contain any logic. Resistors *R1 – R3* pull down the sockets FIRE, ARMED and CONCD lines(For the pin-out see Section 2.2.3). Those are inputs of the µC and are able to not have a defined potential if the user unplugs the trigger when the system is powered, therefore needing to be pulled down. The local signal buzzer *BZ1* is wired to the sockets BUZZER signal and its voltage is stabilized by the capacitor *C1*.

2.2.2.2 Attiny 861

The heart of the controller is the 8-Bit AVR micro controller Attiny 861¹¹. Its most notable features are the 512 bytes in-system programmable EEPROM that stores the firing sequence, 16 I/O pins, the 8 kilobyte of program flash memory, as well as many more features. Further information will be provided in Section 3.

2.2.2.3 USB serial communication

For programming the firing sequence to µC has to communicate with a computer. As the Attiny 861 does not have UART support, the TTL serial communication was used instead. This requires to translate the TTL to the UART protocol, which is often done by a FTDI chip¹². At the time where the hardware for this device was being built no FTDI chip or substitute chip could be obtained with reasonable pricing. Therefore the decision was made to save time and money by buying a UART-to-TTL module of Amazon. As this is a bought module it will be replaced in the near future, when a FTDI chip is obtainable.



Figure 9: The bought UART-to-TTL converter from *Amazon*

Source of Figure 9: <https://amzn.eu/d/3QKPgB9>

¹⁰<https://www.ti.com/lit/ds/symlink/sn54ls139a-sp.pdf>

¹¹https://www.mouser.com/datasheet/2/268/Atmel_2588_8_bit_AVR_Microcontrollers_tinyAVR_ATti-1315472.pdf

¹²<https://ftdichip.com/>

2.2.2.4 Arm safety circuit

This part of the controller circuit is shown in Figure 8 on the bottom and is very important. For safety reasons, the ignition voltage gets galvanically disconnected from the ignition voltage line of the modules, if the system is unarmed. Without this circuit, it is possible for a bridge wire detonator to ignite prematurely during power up or if a module is disconnected and reconnected.

The cause of this dangerous behaviour is explained by the shift registers used in the ignition modules. Most shift registers will, when connected to power, take on random values inside their internal registers. If the output is disabled through the *Output Enable* pin on the 74HC595 (For further information see Section 2.4.2 "Controlling the Ignition Circuits"), this could not happen. However those enable pins are driven by the DMUX on the controller (See Figure 8) and those DMUX are controlled by the pC. But pC takes longer to boot than the shift registers. Thus for a very short period ($< 20s$) turning on random outputs of the shift registers and making premature ignition possible. Although this cannot be prevented by firmware, it is possible to remove the ignition voltage and pulling down the modules ignition voltage line with a 100Ω resistor (See resistor $R7$ in Figure 8). The relay $K1$ is only connecting the ignition voltage to the modules power lines, if the system is armed and therefore making it technically impossible to accidentally set off pyrotechnics unintentionally.

2.2.2.5 Additional circuits

As the system most likely will be operated in the dark, the decision was made to add a small LED light inside the casing of the controller to illuminate the interface. The circuit can be found in the bottom left corner in Figure 8, although the LEDs are not soldered onto the circuit-board. They are placed inside of the lid (See Section 2.2.5), thus shining downwards on the interface if the case is opened.

2.2.3 Sockets and Plugs

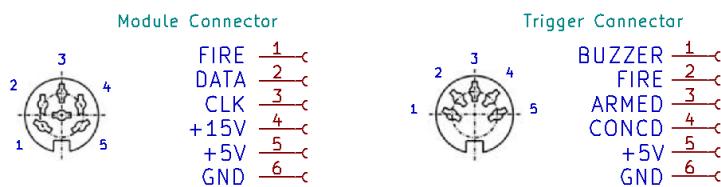


Figure 10: The pin-out of the trigger and modules socket/plug.

Source of socket symbol in Figure 10: https://cdn-reichelt.de/documents/datenblatt/C160/MAB%205S-ROHS_XX.pdf

To connect the peripheral devices to the controller DIN¹³ plugs and sockets were used. The modules socket and plug where chosen differently than the triggers, because plugging in the trigger into a module socket would damage the triggers electronics. For the pin-out of each socket see Figure 10 and below in Figure 11 the modules socket and plug is visible.



Figure 11: DIN socket and plug of the modules.

Source of Figure 11: https://cdn-reichelt.de/bilder/web/xxl_ws/C160/MAS_50.png
 Source of Figure 11: https://cdn-reichelt.de/bilder/web/xxl_ws/C160/MAB_5.png

¹³https://en.wikipedia.org/wiki/DIN_connector

2.2.4 Circuit board

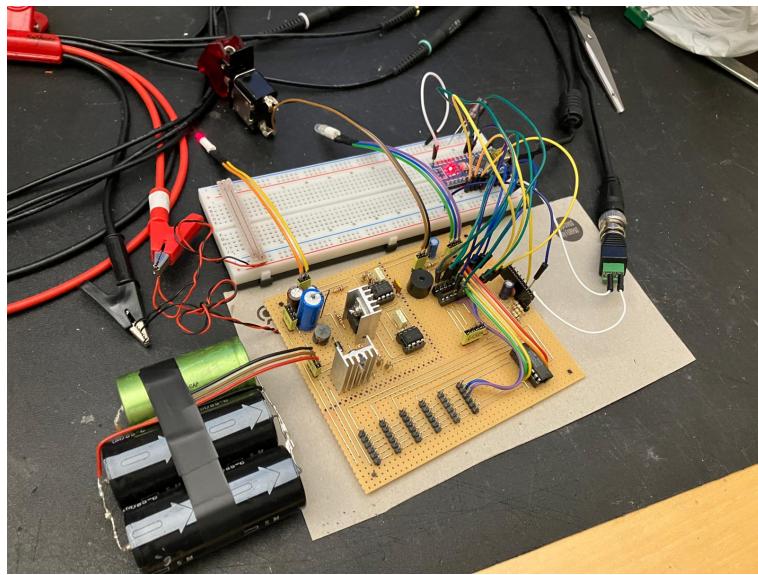


Figure 12: The controller circuit during development.

A perfboard with one sided solder copper pads with the dimensions of 100x115mm was used as the base for the controller circuit. Both the controller circuit (See Figure 8) and the ignition voltage generator (See Figure 3) were soldered onto the same board. Ideally, a PCB should be used to improve EMC, but this would make it hard to change things in the future.

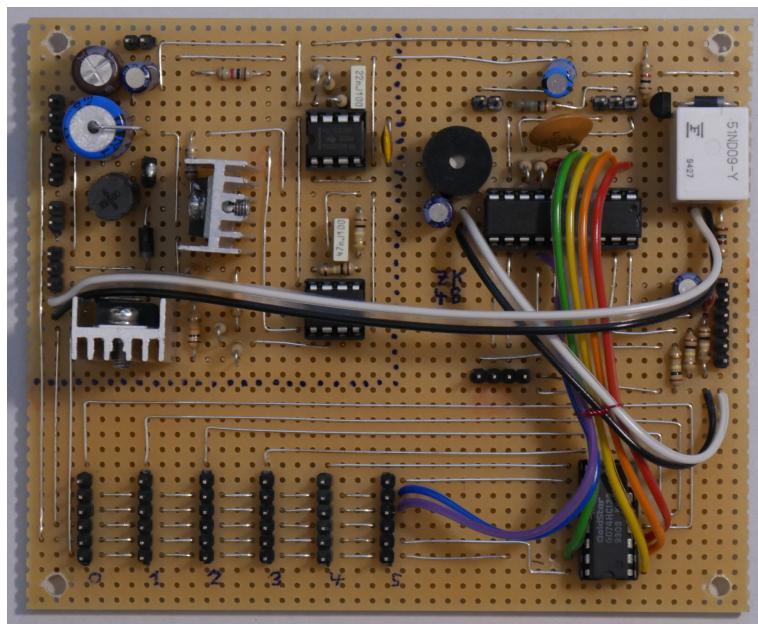


Figure 13: The controller circuit (See Figure 8) and the ignition voltage generator (See Figure 3) soldered onto a perfboard.

The ignition voltage generator is visible in the left top corner, with heat sinks mounted onto the MOSFET and voltage regulator. The right top corner contains the µC, all the pin-headers for connecting the switches, LEDs and the arm safety circuit. On the bottom, the peripheral managing logic is visible together with the six pin-headers for the modules.

2.2.5 Housing of the Controller

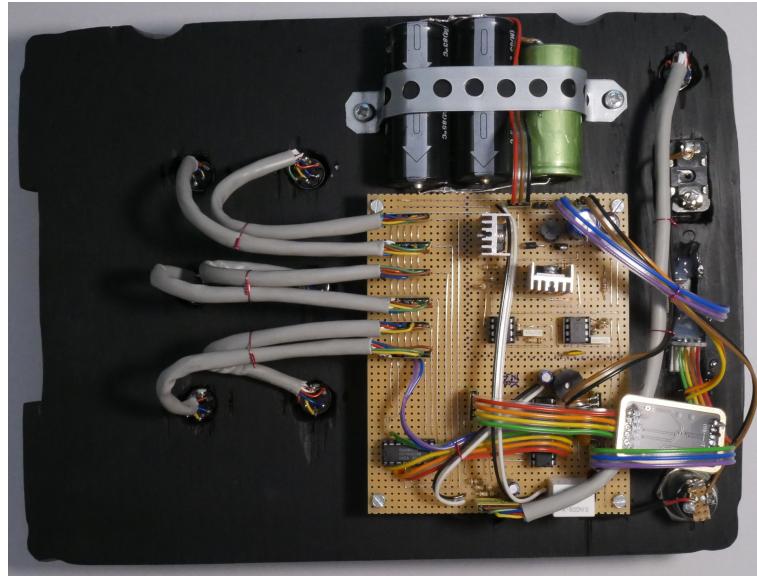


Figure 14: The circuit board depicted in Figure 13 mounted and wired onto the backside of the devices interface.

The image in Figure 14 shows the controller with all peripheral items connected. On the top are the two big capacitors that store the ignition and intermediate voltage, secured to the board by a metal band.

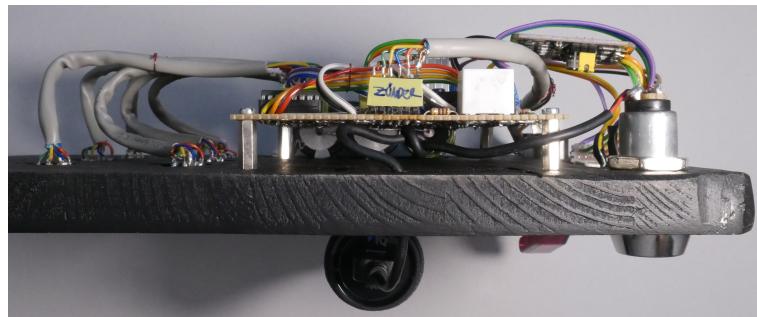


Figure 15: The circuit board depicted in Figure 13 mounted and wired on the backside of the devices interface viewed from the side.

As the baseplate for the interface, a 2cm thick wooden plate was used. Holes were drilled to allow switches, status LED and the sockets to be mounted. Then the plate was painted black and everything got screwed on. In Figure 15 the controller circuit board is shown mounted by 1,5cm four standoffs. For wiring, ribbon cables were used to connect the switches and other input or output items. The sockets of the trigger and modules were wired to the controller circuit by a 5-line telephone cable¹⁴ with shielding. This cable was also the choice for externally connecting the trigger and ignition modules to the controller.

¹⁴<https://amzn.eu/d/65ZALnq>



Figure 16: The gun case used for housing the controller.

Source of Figure 16: <https://amzn.eu/d/eWSlyFs>

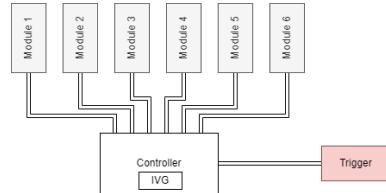
A gun case was bought of Amazon (See Figure 16), and the cushioning in the lower part of the case was removed, to make room for the controller. The complete controller/interface was put into a case and screwed in from the sides. Figure 17 depicts the finished controller turned on. The earlier mentioned lighting(See Section 2.2.2 "Additional circuits") was installed in to top part of the case and the wires were hidden behind the cushioning. In Figure 17 the wires can be seen coming out from the top part of the case and going to the lower part(See Figure 17 right top corner).



Figure 17: The complete controller and interface inside its case.

1. ON/OFF key-switch
2. Status LED
3. Ignition voltage reached indicator (See Section 2.1)
4. USB female socket for programming
5. Light ON/OFF switch
6. Controller arm switch
7. Trigger socket
8. Power bank
9. Module sockets

2.3 Trigger



The external trigger is responsible for starting the pyrotechnic show. Its purpose is to tell the controller when to start, but also tell the user what the state the controller is currently in. This is very important, because the user must be far away from the controller when the pyrotechnics are ignited, but also needs to be informed if the system is working properly or not.

2.3.1 Circuit

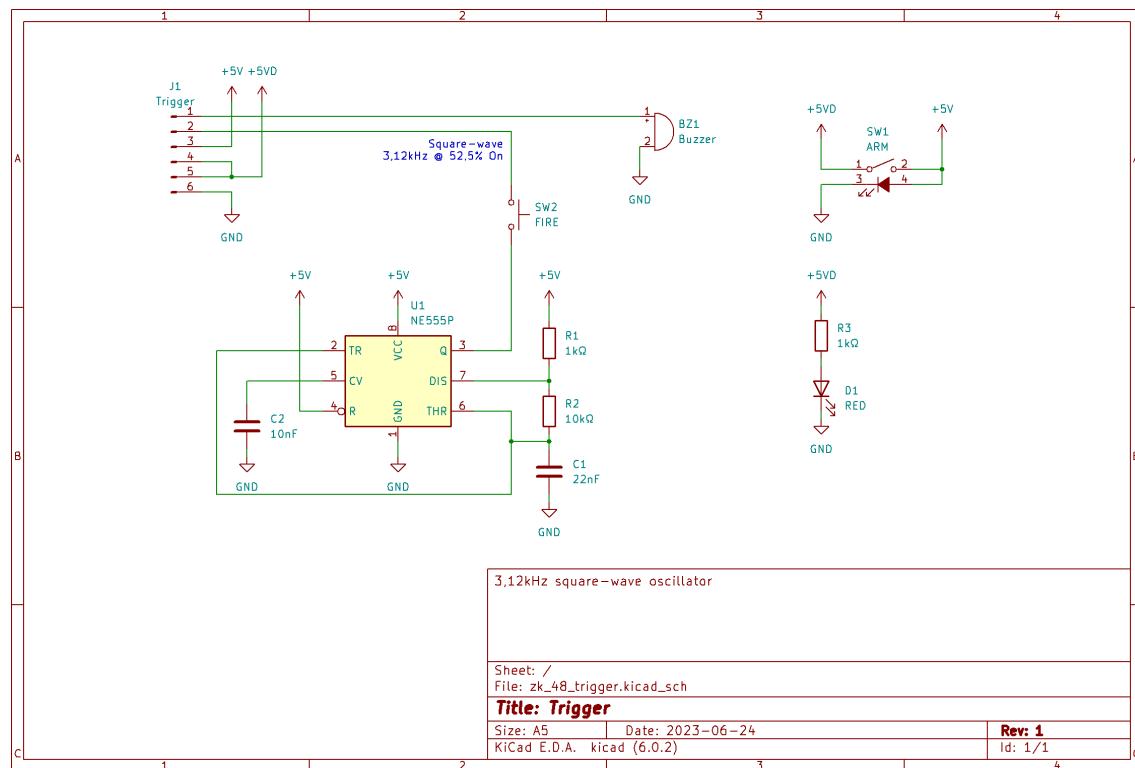


Figure 18: The circuit of the external trigger.

2.3.2 How does the trigger work?

The trigger receives one signal and sends three signals back. The first input signal is going to buzzer and is connected to the buzzer on board the controller (See Figure 8). This buzzer gives the user feedback on the state of the system by changing the frequency of the sound produced. But more about this topic in Section 3. One of the three outputs is the CONCD signal, which stands for "Connected" and tells the µC if the trigger is plugged in. This output is straight connected to the +5V power line. The second output is the ARMED signal which indicates to the µC if the trigger is armed. This signal is produced by toggling the arm switch SW1 which also turns on the NE555 oscillator. If the trigger is armed, the NE555 will generate a 3,12kHz square-wave signal with a 52,2% on-time. This signal is not passed through, until the user holds down the fire SW2 button. Then, the square-wave signal will be put through on the FIRE line of the trigger, therefore starting the firing sequence. A small LED D1 is also present on the trigger to indicate if the trigger has power.

2.3.3 Housing of the Trigger

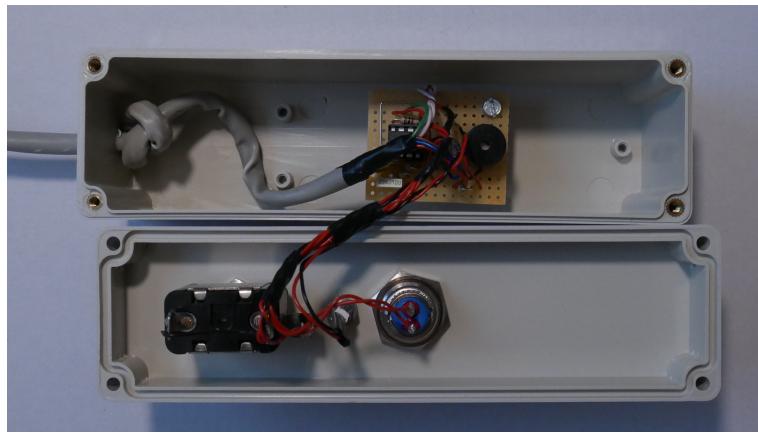


Figure 19: The external trigger circuit board mounted inside its case.

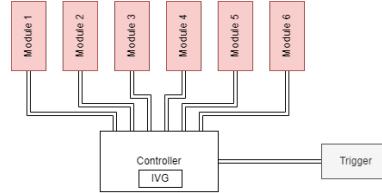
The circuit diagram of the trigger shown in Figure 8, got soldered on a small perfboard and together with the arm switch, power LED and fire button were mounted into a ABS housing with the dimensions of 150x45x55mm as shown in Figure 19. The trigger uses a telephone cable, as described in Section 2.2.5, with a length of approximately 15m to connect to the controller. In Figure 20 the finished controller is visible with the cable and plug.



Figure 20: The external trigger with its cable.

1. Arm switch
2. Power LED
3. Fire button

2.4 Ignition Modules



The ignition modules are responsible for the ignition of the pyrotechnic bridge wire detonators and therefore carry a large responsibility. There is no margin of error, as the whole system depends on the ignition modules proper function. Also the cost of each ignition module must be considered, as six such modules need to be build.

2.4.1 Circuit

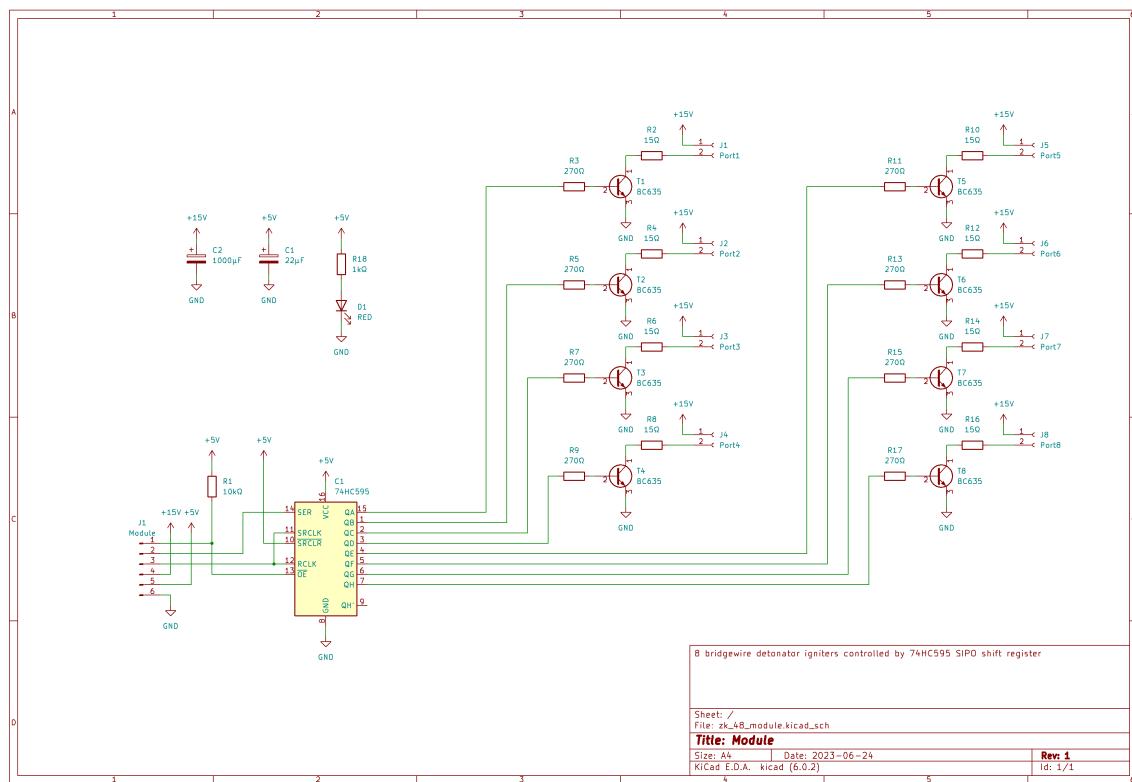


Figure 21: The circuit of the ignition module.

2.4.2 How does the Ignition Module work?

2.4.2.1 Ignition Circuit

When setting off a bridge wire detonator, a current larger than $500mA$ will flow, for guaranteed ignition. The switching transistor must therefore be able to handle such currents repeatedly without being damaged. Furthermore, considering that detonators can be faulty, for example have an internal short-circuit, those larger currents should not destroy the transistor as well. Additional to the actual electrical requirements, the cost of each module should stay on the lower end, as six modules are going to be build with eight ignition circuits.

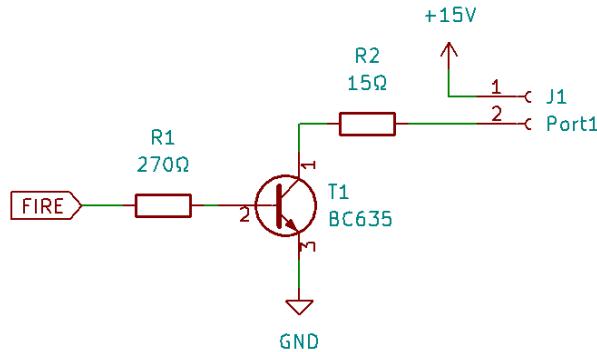


Figure 22: The circuit designed for setting off one bridge wire detonator inside the ignition module.

All aforementioned considerations were dealt by limiting the current through the transistor. Limiting the current was accomplished by using a 15Ω resistor after the bridge wire detonator (See Figure 22 R_2) which limits the current, at $15V$ ignition voltage, to maximal $1A$. Knowing the maximal current and considering the other requirements, the NPN transistor BC635¹⁵ in a TO-92 package, was chosen as the switching transistor. Its continuous collector current is equal to $1A$ and it costs 9 cents per piece, thus making this transistor ideal. To drive the BC635 a 270Ω resistor was used to limit the base current to circa $16mA$ (See Equation (2)), which results in the saturation of the transistor. Figure 22 displays the complete ignition circuit.

$$I_B = \frac{U_{IC,OUT} - U_{BE}}{R} = \frac{5V - 0,7V}{270\Omega} = 15,93mA \quad (2)$$

Designing this circuit also required selecting the ignition voltage. The reason why the ignition voltage is $15V$ and not $5V$, which would make things much simpler, lays inside Ohm's law: If the ignition voltage is $5V$ instead of $15V$ and the current is still gets limited to $1A$ in by a 5Ω resistor, a reduction of the current through the detonator will occur. The calculations below (Equation (3) and Equation (4)) calculate the current for both cases, by simulating a bridge wire detonate with the same resistor value of $4,7\Omega$ as used in Section 2.1.3.

$$I = \frac{U_{IG}}{R_2 + R_{BWD}} = \frac{15V}{15\Omega + 4,7\Omega} = 761,14mA \quad (3)$$

$$I = \frac{U_{IG}}{R_2 + R_{BWD}} = \frac{5V}{5\Omega + 4,7\Omega} = 515,54mA \quad (4)$$

Comparing result Equation (3) to result of Equation (4), a strong drop in current is noticeable. This is a problem that worsens with additional detonators connected. In a typical pyrotechnic show it is not unlikely for three or more detonators to be connected in series (Parallel wiring is not recommended in general). A stronger transistor could be used but this could raise cost which is unwanted as 48 transistors are required.

¹⁵https://www.tme.eu/Document/aa873a3a455ba8f13e3474bd76bcc1d9/BC635_40.pdf

2.4.2.2 Controlling the Ignition Circuits

As mentioned in previous sections, the modules are based on a 8-Bit SIPO (Serial-in Parallel-out) shift register 74HC595¹⁶.

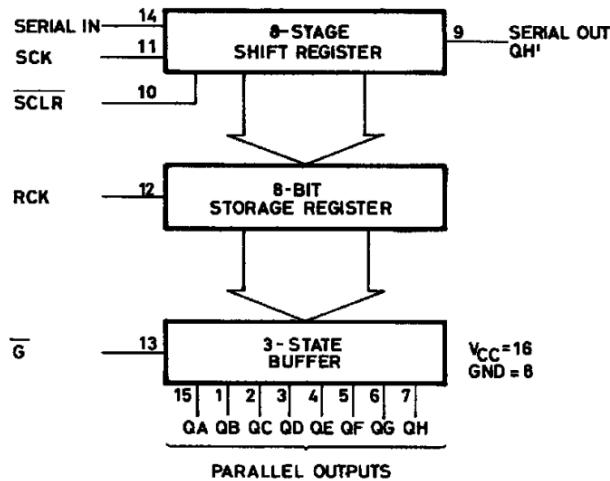


Figure 23: Functional diagram of the 74HC595

Source of Figure 23: <https://cdn-reichelt.de/documents/datenblatt/1240/74HC595%23STM.pdf>

The 74HC595 is made up of three components: the shift register, a register and a tri-state buffer (See Figure 23). To fire a port, the µC sends a byte mask serially to the shift-register (In Figure 23 see pin 14 SERIALIN and 11 SCK) where only one bit is set, indicating which port is being fired. Because the register clock RCK and shift register clock SCK are wired together (See Figure 21) the register will take on the values from the shift register. As explained in Section 2.2.2 "Peripheral Managing and Controlling Logic" the µC afterwards selects the correct module being fired by pulling low of the enable pin on the tri-state buffer (pin 13 in Figure 23), thus outputting the stored bit mask of the register onto the parallel outputs. Figure 21 shows that each output is directly connected to one ignition circuit, whereby the setting of an output fires a detonator.

2.4.2.3 Additional circuits

To reduce the currents through the cable to the ignition modules when firing, a large 1000F capacitor C_2 was added to the 15V ignition voltage line inside the ignition module (See Figure 21). As charge would be stored closer to detonator theoretically this should in theory reduce currents, however this is speculative and the capacitor might not be needed at all.

¹⁶<https://www.ti.com/lit/ds/symlink/sn74hc595.pdf>

2.4.3 Housing of the Ignition Modules

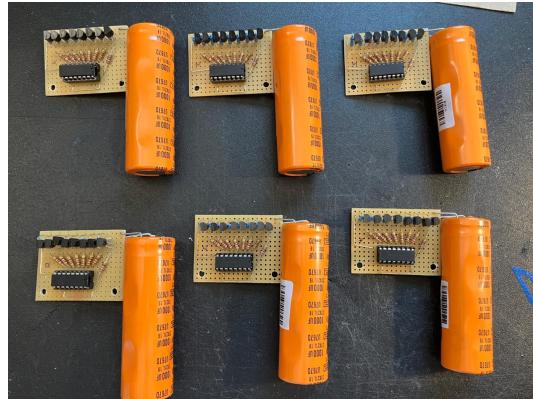


Figure 24: All six ignition modules soldered onto perfboards.

The circuit got soldered onto a perfboard and for the housing a common electrical junction box, from the hardware store, was used as it is cheap, light weight, easy to cut and can be screwed against something. For connecting to the controller, four pieces of 4m and four pieces of 8m telephone cables were cut to size and soldered to the each of the eight modules circuit boards as seen in Figure 25. Also visible in Figure 25 is the orange stabilizing 1000 μ F capacitor C2 seen in Figure 21.



Figure 25: The ignition module from the inside.

To connect the bridge wire detonators to the circuit, speaker terminals were used. In Figure 26 the bottom two terminal rows represent the eight ports and the top row are all 15V. This design is less ideal, because for two ports there is only one 15V terminal hole, but due limited space on top the housing this was necessary.



Figure 26: The complete ignition module with eight ports on the bottom and one power rail on the top.

3 Firmware

The Attiny861 is running a Firmware, based on a basic Moore state machine¹⁷ with eight states. Each state performs an essential function, with distinct sound and status LED outputs. More about the state machine in ???. The code was written in *C++*(Arduino *C++*) with the Arduino IDE. Read more about the programming of the Attiny861 in Section 3.6. Please note that the firing sequence, that controls the pyrotechnic show, is called program.

3.1 Pinout

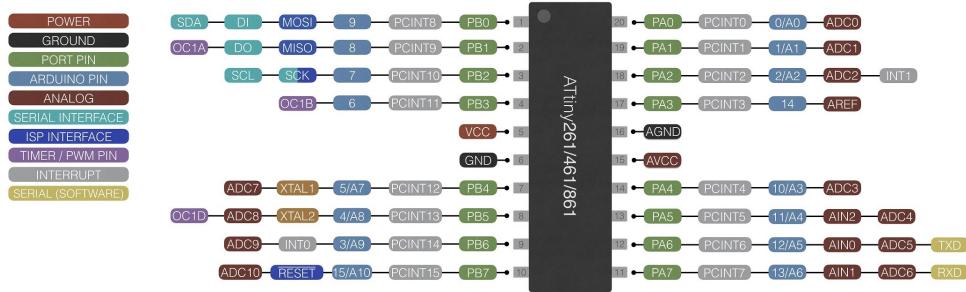


Figure 27: Pinout of the Attiny861 µC.

Source of Figure 27: <https://github.com/SpenceKonde/ATTinyCore>

All pins, shown in Figure 27, of the µC were used except pin 18 as this is the reset pin. The pin descriptors, used inside the firmware, are listed in Table 1.

Pin	Name	Type	Description
3	MODUL_DATA	OUT	Dataline for serial data transmission
4	MODUL_CLK	OUT	Clock signal for serial data transmission
2	TRIGGER_FIRE	INT	Receives the trigger signal from the trigger
0	TRIGGER_ARMED	IN	High if trigger is armed
1	TRIGGER_CONNECTED	IN	High if trigger is connected
8	TRIGADD_A0	OUT	Address line for selecting the ignition module
9	TRIGADD_A1	OUT	Address line for selecting the ignition module
10	TRIGADD_1E	OUT	Enable line for selecting the ignition module
11	TRIGADD_2E	OUT	Enable line for selecting the ignition module
12	STATUS_RED	OUT	Turns on the red status LED on the controller
13	STATUS_GREEN	OUT	Turns on the green status LED on the controller
12	STATUS_PIEP	OUT	Turns on the buzzer on controller and trigger
14	THIS_ARMED	IN	High if controller is armed
7	RX	IN	Receive signal for serial TTL communication
6	TX	OUT	Transmit signal for serial TTL communication

Table 1: Signal descriptions of all pins used of the Attiny861

¹⁷https://en.wikipedia.org/wiki/Moore_machine

3.2 Setup

The pinout listed in Table 1 gets configured inside the *setup* function(See Listing 1) which executes once on boot. Inside the same function call, the interrupt will be configured, to invoke a function called *fire* if the pin voltage level changes from low to high on pin 2. Furthermore, the baud rate for the TTL serial communication is set to *57600*. More about the serial communication in Section 3.5.

```

1 | Serial.begin(BAUDRATE);
2 |
3 | attachInterrupt(digitalPinToInterrupt(TRIGGER_FIRE), fire, RISING);
4 |
5 | pinMode(MODUL_DATA, OUTPUT);
6 | pinMode(MODUL_CLK, OUTPUT);
7 | pinMode(TRIGADD_A0, OUTPUT);
8 | pinMode(TRIGADD_A1, OUTPUT);
9 | pinMode(TRIGADD_1E, OUTPUT);
10 | pinMode(TRIGADD_2E, OUTPUT);
11 | pinMode(STATUS_PIEP, OUTPUT);
12 | pinMode(STATUS_RED, OUTPUT);
13 | pinMode(STATUS_GREEN, OUTPUT);
14 | pinMode(TRIGGER_FIRE, INPUT);
15 | pinMode(TRIGGER_ARMED, INPUT);
16 | pinMode(TRIGGER_CONNECTED, INPUT);
17 | pinMode(THIS_ARMED, INPUT);

```

Listing 1: Code configuring the IO pins of the Attiny861.

All the register of the ignition modules are cleared during *setup*, by calling the *transmitData*(See Section 3.3.2) function with an out of range index(See Listing 2). This sets all ignition modules internal shift registers to 0. This is done due safety reasons, to prevent accidental premature firing, as already stated in Section 2.2.2 "Arm safety circuit". For the same reason, the *setModule* function(See Section 3.3.1) is used to disable all ignition modules. The program endpoint is also read from the internal EEPROM(More about that in Section 3.5).

```

1 | transmitData(CLEAR_REG);
2 | setModule(ALL_OFF);
3 | pg_stop = readProgramEndpoint();

```

Listing 2: Code clearing and disabling ignition modules and reading the program endpoint.

The code snipped shown in Listing 3 is also contained in the setup function. Without the code in Listing 3 the firmware would execute at a eighth of normal speed. This means, a delay of 10ms takes 80ms. This increase of the delay was measured, by programming the Attiny861 to output a 10ms pulse which got measured by a oscilloscope. The oscilloscope measured 80ms instead of 10ms. Research led to a forum post where the code in Listing 3 was found and subsequently tested, which proved to be successful in restoring the correct timing. Why this increase in delay occurs in the first place is unknown.

```

1 | cli(); // Disable interrupts
2 | CLKPR = (1<<CLKPCE); // Prescaler enable
3 | CLKPR = 0x00; // Clock division factor
4 | sei(); // Enable interrupts

```

Listing 3: Code for removing the clock division factor.

Source of the Code in Listing 3: <https://community.platformio.org/t/attiny-8mhz-wrong-timing-in-delay-fast-led-and-neopixel/24992/3>

3.3 Ignition of a Port

```

1 | void ignite( byte module , byte port ) //
2 |
3 | {
4 |     transmitData( port );
5 |     setModule( module+1 ); // +1 because 0 (ALL_OFF) is turning off all modules
6 |
7 |     delay( IGNITION_TIME );
8 |
9 |     setModule( ALL_OFF );
10|    transmitData( CLEAR_REG ); // clear registers for safety reasons
10| }
```

Listing 4: Ignition function for setting of a bridge wire detonator given a port and ignition module

The function *ignite*, shown in Listing 4, sets off a given a port index and a ignition modules index. First, on line 3 in Listing 4 the port gets set on all ignition modules. Right after, in line 4, the correct ignition modules gets selected, thereby firing the port. After a delay called ignition time, all modules get unselected and the ignition module are cleared.

3.3.1 Ignition Module addressing

```

1 | #define nA1 0
2 | #define nA0 1
3 | #define E1 2
4 | #define E2 3
5 |
6 | // !! do not change !!
7 | #define ALL_OFF 0
8 | #define CLEAR_REG 10
9 | //nA1 nA0 ,1E 2E
10| const byte module_resolve[7][4]=
11| {
12|     {0,0,1,1}, // ALL_OFF
13|     {0,0,0,1},
14|     {1,0,0,1},
15|     {0,1,0,1},
16|     {1,1,0,1},
17|     {1,1,1,0},
18|     {0,1,1,0}
19| };
```

Listing 5: Bit array for controlling the DMUX

For addressing the modules from code, a 2D byte array is used for controlling the two DMUX (See Section 2.2.2). This allows the ignition modules to be addressed by just a single integer value. In Listing 5 the constant *module_resolve* contains a unique bit pattern, allowing for all ignition modules to be enabled, behaving like a lookup-table. Line 2 till 5 in Listing 5 define aliases for mapping the column in *module_resolve* (as shown in line 10) on the correct outputs *TRIGADD_A0*, *TRIGADD_A1*, *TRIGADD_E1* and *TRIGADD_E2*.

```

1 | void setModule( byte module ) //
2 |
3 | {
4 |     digitalWrite( TRIGADD_A0, module_resolve[ module ][ nA0 ] );
5 |     digitalWrite( TRIGADD_A1, module_resolve[ module ][ nA1 ] );
6 |     digitalWrite( TRIGADD_1E, module_resolve[ module ][ E1 ] );
7 |     digitalWrite( TRIGADD_2E, module_resolve[ module ][ E2 ] );
7 | }
```

Listing 6: Code selecting a module by its index

The function *setModule* in Listing 6 receives a byte, which represents the index of the ignition module, by which the correct row in the *module_resolve* array gets indexed, to set the address and enable lines on the DMUX to enable the correct module.

3.3.2 Serial Data Transmission

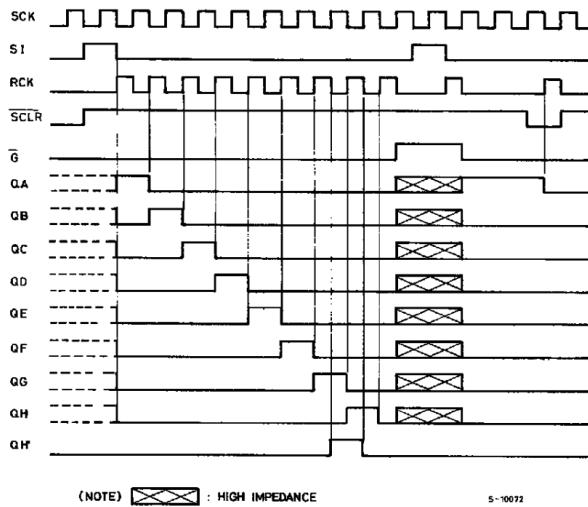


Figure 28: Timing diagram of the 74HC595.

Source of Figure 28: <https://cdn-reichelt.de/documents/datenblatt/1240/74HC595%23STM.pdf>

As explained in Section 2.4.2 "Controlling the Ignition Circuits" the ignition modules get the information, about what port to fire, over a serial data transmission. The timing diagram in Figure 28 shows how a bit gets shifted though the shift register and outputted on the outputs. This is also the same working principle of the port setting process. However when firing a port, the enable pin(Pin G in Figure 28) of the tri-sate buffer is turned off during the transmission and only gets enabled for 10ms afterwards, to fire the set port (Please read Section 2.4.2).

```

1   }
2   delayMicroseconds(TRANSMISSION_SPEED);
3   digitalWrite(MODUL_CLK,HIGH);
4   delayMicroseconds(TRANSMISSION_SPEED);
5   digitalWrite(MODUL_CLK,LOW);
6   digitalWrite(MODUL_DATA,LOW);
7   }
8   }
9
10 // selects what module is meant to ignite the set Port
11 void setModule(byte module) //
12 {
13   digitalWrite(TRIGADD_A0,module_resolve[module][nA0]);
14   digitalWrite(TRIGADD_A1,module_resolve[module][nA1]);
15   digitalWrite(TRIGADD_1E,module_resolve[module][E1]);
16   digitalWrite(TRIGADD_2E,module_resolve[module][E2]);
17 }
```

Listing 7: Function for transmitting the port to the ignition modules

Function *transmitData* displayed Listing 7 sets the port on all ignition modules. Line 12 till 15 are creating a nine impulse clock signal with a $5\mu s$ ontime. The reason for nine impulse instead of eight can be explained by viewing Figure 28. In this timing diagram the shift register clock *SCK* and the register clock *RCK* are 180 Degrees phase shifted, but in ?? *RCK* and *SCK* are connected together. This means both the shift register clock and register clock receive the clock impulse at the same time. Thus an additional impulse is needed to fully push through the bit. Lines 6 till 10 in Listing 7 are setting the data line to high, when the set port is equal to the port index.

3.4 State Machine

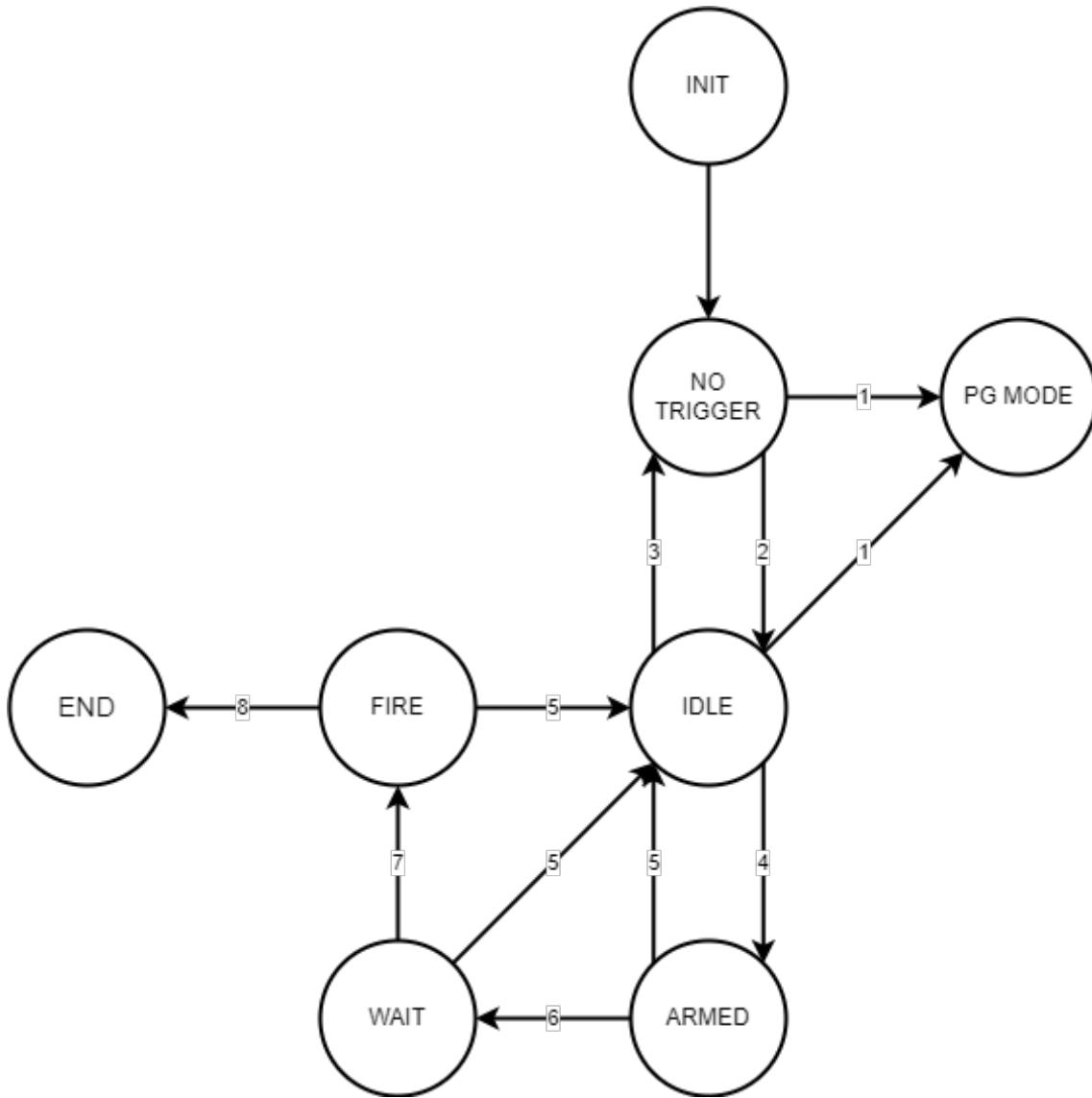


Figure 29: State diagram of the realised state machine.

The firmware starts at INIT and immediately enters NO_TRIGGER. If no trigger is connected to the controller the system stays in NO_TRIGGER until the program mode gets called over USB or a trigger is connected. In state IDLE the systems waits for both arm switches to be armed. If both are armed the system transitions into ARMED. When in state ARMED the system waits until it receives a fire signal from the trigger, therefore transitioning to state WAIT. In WAIT the system stays for 10s then transitions to state FIRE thus playing through the firing sequence as programmed in EEPROM until it reaches its end therefore entering state END. If during state FIRE, WAIT or ARMED the controller or the trigger gets unarmed the system returns to state IDLE. This state machine is depicted by Figure 29 together with the transition conditions in Table 3 and the LED and buzzer output for each state in Table 2.

State	Green	Red	Buzzer	Delay
PG_MODE	1	1	0	-
INIT (3x)	1	1	1	100ms
	0	0	0	100ms
NO_TRIGGER	1	0	0	250ms
	0	1	0	250ms
IDLE	1	0	0	100ms
	0	0	0	500ms
ARMED	0	1	0	100ms
	0	0	0	500ms
WAIT	0	1	1	50ms
	0	0	0	50ms
FIRE	0	1	1	{IGNITION_TIME}ms
	1	0	0	{PROGRAM[PG_INDEX].DELAY}ms
END	0	1	0	-

Table 2: Truth table for state depended LED and buzzer output with timing information.

Nr	PG	TF	TA	TC	CA	WT	DN
1	1	X	X	X	X	X	X
2	X	X	X	0	X	X	X
3	X	X	0	1	0	X	X
4	X	X	1	1	1	X	X
5	X	X	0	X	X	X	X
5	X	X	X	X	0	X	X
6	X	1	1	1	1	1	X
7	X	X	1	1	1	0	X
8	X	X	X	X	X	X	1

Table 3: Truth table for transition condition based on state variables. ("X" = Don't care)

3.4.1 State Variables

State Variable	SV	Condition
Program Mode	PG	checkPGM()
Trigger Fire	TF	fire_pulse_counter >= FIRE_PULSE_SETOFF
Done	DN	pg_index == pg_stop
Wait	WT	wait_counter <= FIRE_DELAY
Trigger Armed	TA	digitalRead(TRIGGER_ARMED)
Trigger Connected	TC	digitalRead(TRIGGER_CONNECTED)
Controller Armed	CA	digitalRead(THIS_ARMED)

Table 4: List of all state variables and their conditions

The state variables listed in Table 4 govern the behaviour of the state machine as listed in Table 3. Some state variables are directly read from the IO pins of the Attiny861 and other are the result of internal counter values. In the following paragraphs below the state variables will be explained, with the exception of TA, TC and CA as they are generated externally(Please see Figure 8 in Section 2.2).

3.4.1.1 PG (Program Mode)

The program mode state variable is set by a function called `checkPGM()`, which checks if a keyword is received over USB. In Listing 8 the code for checking the keyword is depicted.

```

1 // checks if program mode must be entered
2 char checkPGM()
3 {
4     if (Serial.available() > 0)
5     {
6         String str_buffer = Serial.readString();
7
8         if(str_buffer.compareTo("#pgm:") == 0)
9         {
10            return true;
11        }
12    }
13 }
```

Listing 8: Checking if the keyword for entering the program mode is entered.

The function listed in Listing 8 reads from the serial(Line 3) stream coming from a external PC and then checks if the string is equal to the keyword(Line 7). The '#' and ':' are only included due consistency of all keyword. More about the syntax in Section 3.5.2.

3.4.1.2 TF (Trigger Fire)

To safely detect when the system should fire i.e go into state WAIT, the trigger has to send over 1000 pulses under 600ms on pin 2 of the Attiny861. Inside the setup function, in Section 3.2, the pin 2 (alias TRIGGER_FIRE) gets handled by an interrupt, which on signal change, invokes a function called `fire()`. This function is visible in Listing 10.

```

1 void fire()
2 {
3     fire_pulse_counter++;
4 }
```

Listing 9: Interrupt function for handling the fire signal.

This `fire()` function counts up a variable called `fire_pulse_counter`, whereby the value is representing the number of pulses counted. If this counter reaches a value higher than 1000, the TF state variable takes on 1(See Table 4). Directly after the comparison `fire_pulse_counter` must be set to 0, which is shown in ?? in line ?.

3.4.1.3 DN (Done)

The state variable DN is also a results of a comparison of an up counter and a constant. In Section 3.2 Listing 2 line 3, the program endpoint is read and saved into a variable called `pg_stop`. This constant stands for the number of steps in the pyrotechnic show. To go through each step of the pyrotechnic show, a up counter is used, incrementing after each step is complete. This will be explained closely in Section 3.4.2 "FIRE". That counter is called `pg_index` and increments every time state FIRE is called. To generate the state variable DN `pg_index` and `pg_stop` are compared if equal.

3.4.1.4 WT (Wait)

To generate the state variable WT a counter is again used. This counter stored in variable `wait_counter` counts up every time state WAIT is called.

```

1 #define FIRE_DELAY 100 // 1 = 100ms 100 = 10s
2 byte wait_counter;
```

Listing 10: Definition of `wait_counter` and the .

3.4.2 States

3.4.2.1 *PG_MODE*

3.4.2.2 *INIT*

3.4.2.3 *NO_TRIGGER*

3.4.2.4 *ARMED*

3.4.2.5 *WAIT*

3.4.2.6 *FIRE*

3.5 Program Mode

3.5.1 Entering the Program Mode

3.5.2 Instruction Set

3.5.3 Instruction Decoding

3.6 Programming the Attiny861

4 Appendix

4.1 Pricing

Name	Amount	Price (EUR)	Total (EUR)
Transistors	48	0.09	4.32
Resistors	105	0.012	1.26
Resistors 15Ohm	48	0.05	2.42
Buzzer	2	1.00	2.00
Perfboard	2	2.725	5.45
Speaker Terminal	18	0.404	7.272
MAB 5 (Plug)	7	0.72	5.04
MAS 50 (Socket)	7	0.675	4.725
74HC595	6	0.313	1.878
74LS139	1	0.81	0.81
Attiny 861	1	2.42	2.42
UART-To-TTL Converter	1	4.02	4.02
Solder	1	12.02	12.02
Power Bank	1	20.16	20.16
Gun case	1	22.99	22.99
Ignition Module case	6	3.99	23.94
Telephone Cable	1	24.79	24.79
Trigger ABS housing	1	4.99	4.99
Cost for Buttons , Switches and LEDs (ca.)	1	10.00	10.00
Screws and Standoffs (ca.)	1	10.00	10.00
Capcaitors,Diodes,Relays, IC's and more (ca.)	1	15.00	15.00
			185.50

Table 5: All parts of the ZK-48 listed with prices. higher.

In Table 5, the cost of the complete system is split into all parts that make it up. Some values are estimates, as some parts were not bought specifically for this, instead were already on hand, which could result in a lower price or higher depending on the source. Not considering the many months of development a price of 185.50 Euro is comparatively low as the price range for similar systems(without programming capability) range from 400 Euro ¹⁸ to 700 Euro ¹⁹.

4.2 Conclusion

¹⁸<https://www.firework-shop.de/Funkzuendanlage-48-Kanal-DBR01-X4-48::104404.html>

¹⁹<https://feuerwerkershop.de/produkt/funkzuendanlagen/sonstige-zuendanlagen/48-kanal-zuendanlage-db24-set/>

4.3 Firmware Code

```
1 // Copyright: Markus K. 2023
2 // Github: https://github.com/inimodo
3
4 #include <EEPROM.h>
5
6 #define MODUL_DATA 3
7 #define MODUL_CLK 4
8 #define TRIGGER_FIRE 2
9 #define TRIGGER_ARMED 0
10 #define TRIGGER_CONNECTED 1
11 #define TRIGADD_A0 8
12 #define TRIGADD_A1 9
13 #define TRIGADD_1E 10
14 #define TRIGADD_2E 11
15 #define STATUS_RED 12
16 #define STATUS_PIEP 5
17 #define STATUS_GREEN 13
18 #define THIS_ARMED 14
19 #define RX 7
20 #define TX 6
21
22 // pin mapping for selection of the module
23 //D11/A0 -> nA1
24 //D10/A1 -> nA0
25 //D9 / A2 -> 1E
26 //D8 / A3 -> 2E
27
28 // !! do not change !!
29 #define nA1 0
30 #define nA0 1
31 #define E1 2
32 #define E2 3
33
34 // !! do not change !!
35 #define ALL_OFF 0
36 #define CLEAR_REG 10
37 //nA1 nA0,1E 2E
38 const byte module_resolve[7][4]=
39 {
40     {0,0,1,1}, // ALL_OFF
41     {0,0,0,1},
42     {1,0,0,1},
43     {0,1,0,1},
44     {1,1,0,1},
45     {1,1,1,0},
46     {0,1,1,0}
47 };
48
49 typedef struct _section {
50     byte module;
51     byte port;
52     unsigned short msdelay;
53 } section;
54
55 // the program length (number of ports) higher values do not make sense
56 #define PG_LENGTH 48
57 byte pg_stop = 0;
58 byte pg_index = 0;
59
```

```
60 // chars for program mode operation
61 #define PGM_ARG_CHAR ':'
62 #define PGM_START_CHAR '#'
63 #define PGM_ANS_CHAR '!'
64 #define PGM_MAXARGS 6
65
66 const char* cmd_set = "set";
67 const char* cmd_list = "list";
68 const char* cmd_stop = "stop";
69 const char* cmd_pgm = "pgm";
70
71 // threshold for the number of impulses before starting the wait sequence
72 #define FIRE_PULSE_SETOFF 1000
73 short fire_pulse_counter = 0;
74 // delay before ignition of the program
75 #define FIRE_DELAY 100 // 1 = 100ms 100 = 10s
76 byte wait_counter;
77
78
79 // time each ignition pulse takes - depends on e-igniter
80 #define IGNITION_TIME 10 // ms
81 // clock speed of transmitData() !! do not change !!
82 #define TRANSMISSION_SPEED 5 //us
83
84 #define BAUDRATE 57600
85
86 #define STATE_END 8
87 #define STATE_WAIT 7
88 #define STATE_FIRE 6
89 #define STATE_ARMED 5
90 #define STATE_IDLE 4
91 #define STATE_NO_TRIGGER 3
92 #define STATE_INIT 2
93 #define STATE_PG_MODE 1
94
95 byte state = STATE_INIT;
96
97 // ##### // Setup // #####
98
99 void setup()
100 {
101     // this is not my code!
102     // source: https://community.platformio.org/t/
103     //         attiny-8mhz-wrong-timing-in-delay-fastled-and-neopixel/24992/3
104     // start
105     cli(); // Disable interrupts
106     CLKPR = (1<<CLKPCE); // Prescaler enable
107     CLKPR = 0x00; // Clock division factor
108     sei(); // Enable interrupts
109     // end
110
111     Serial.begin(BAUDRATE);
112
113     attachInterrupt(digitalPinToInterrupt(TRIGGER_FIRE), fire, RISING);
114
115     pinMode(MODUL_DATA, OUTPUT);
116     pinMode(MODUL_CLK, OUTPUT);
117     pinMode(TRIGADD_A0, OUTPUT);
118     pinMode(TRIGADD_A1, OUTPUT);
119     pinMode(TRIGADD_1E, OUTPUT);
```

```
121 pinMode(TRIGADD_2E,OUTPUT);
122 pinMode(STATUS_PIEP,OUTPUT);
123 pinMode(STATUS_RED,OUTPUT);
124 pinMode(STATUS_GREEN,OUTPUT);
125 pinMode(TRIGGER_FIRE,INPUT);
126 pinMode(TRIGGER_ARMED,INPUT);
127 pinMode(TRIGGER_CONNECTED,INPUT);
128 pinMode(THIS_ARMED,INPUT);
129
130 transmitData(CLEAR_REG);
131 setModule(ALL_OFF);
132 pg_stop = readProgramEndpoint();
133 }
135
136 // ##### EEPROM Functions #####
137
138 void writeProgramEndpoint( byte endpoint )
139 {
140   EEPROM.write(0,endpoint);
141 }
142
143 byte readProgramEndpoint()
144 {
145   return EEPROM.read(0);
146 }
147
148 void writeProgramPoint( byte index, section pg_point )
149 {
150   short real_address = index*sizeof(section) + sizeof(byte);
151   EEPROM.put(real_address,pg_point);
152
153 }
154
155 section readProgramPoint( byte index )
156 {
157   section pg_point;
158   short real_address = index*sizeof(section) + sizeof(byte);
159   EEPROM.get(real_address,pg_point);
160   return pg_point;
161 }
162
163 // ##### Program Mode #####
164
165 // checks if program mode must be entered
166 char checkPGM()
167 {
168   if (Serial.available() > 0)
169   {
170     String str_buffer = Serial.readString();
171
172     if(str_buffer.compareTo("#pgm:") == 0)
173     {
174       return true;
175     }
176   }
177   return false;
178 }
179
180 // programs the internal EEPROM via serial commands
181 byte programmMode()
```

```
182 {
183     if (Serial.available() > 0)
184     {
185         String str_buffer = Serial.readString();
186         delay(1);
187
188         char split_index[PGM_MAXARGS];
189         split_index[0] = str_buffer.indexOf(PGM_START_CHAR);
190         for(int index=1; index<PGM_MAXARGS; index++)
191         {
192             split_index[index] = str_buffer.indexOf(
193                 PGM_ARG_CHAR,
194                 split_index[index-1]+1);
195         }
196
197         String cmd = str_buffer.substring(split_index[0]+1,split_index[1]);
198
199         if(cmd == cmd_list)
200         {
201             Serial.println("!list:");
202             Serial.print(PGM_ANS_CHAR);
203             Serial.print("stop:");
204             Serial.print(pg_stop);
205             Serial.println(PGM_ARG_CHAR);
206
207             for(int index=0;index<PG_LENGTH;index++)
208             {
209                 section pgpoint = readProgramPoint(index);
210                 Serial.print(PGM_ANS_CHAR);
211                 Serial.print("set:");
212                 Serial.print(index);
213                 Serial.print(PGM_ARG_CHAR);
214                 Serial.print(pgpoint.module);
215                 Serial.print(PGM_ARG_CHAR);
216                 Serial.print(pgpoint.port);
217                 Serial.print(PGM_ARG_CHAR);
218                 Serial.print(pgpoint.msdelay);
219                 Serial.println(PGM_ARG_CHAR);
220             }
221             return;
222         }else if(cmd==cmd_stop)
223         {
224             pg_stop = str_buffer.substring(split_index[1]+1,split_index[2]).toInt();
225
226             writeProgramEndpoint(pg_stop);
227             Serial.println("!stop:");
228             return;
229         }else if(cmd==cmd_set)
230         {
231             byte index = str_buffer.substring(
232                 split_index[1]+1,
233                 split_index[2]).toInt();
234
235             section new_pgpoint;
236
237             new_pgpoint.msdelay = (unsigned short)str_buffer.substring(
238                 split_index[4]+1,
239                 split_index[5]).toInt();
240
241             new_pgpoint.port = (byte)str_buffer.substring(
242                 split_index[3]+1,
```

```
243     split_index[4]).toInt();
244
245     new_pgpoint.module = (byte)str_buffer.substring(
246         split_index[2]+1,
247         split_index[3]).toInt();
248
249     writeProgramPoint(index,new_pgpoint);
250
251     Serial.println("!set:");
252     return;
253 }
254 Serial.println("!error:");
255 }
256 }
257
258 // ##### // IO Functions // #####
259
260 // sets the port on ALL modules
261 void transmitData(byte port) //
262 {
263
264     for(byte port_index = 0; port_index < 9; port_index++)
265     {
266         if(port_index == port)
267         {
268             digitalWrite(MODUL_DATA,HIGH);
269         }else{
270             digitalWrite(MODUL_DATA,LOW);
271         }
272         delayMicroseconds(TRANSMISSION_SPEED);
273         digitalWrite(MODUL_CLK,HIGH);
274         delayMicroseconds(TRANSMISSION_SPEED);
275         digitalWrite(MODUL_CLK,LOW);
276         digitalWrite(MODUL_DATA,LOW);
277     }
278 }
279
280 // selects what module is meant to ignite the set Port
281 void setModule(byte module) //
282 {
283     digitalWrite(TRIGADD_A0,module_resolve[module][nA0]);
284     digitalWrite(TRIGADD_A1,module_resolve[module][nA1]);
285     digitalWrite(TRIGADD_1E,module_resolve[module][E1]);
286     digitalWrite(TRIGADD_2E,module_resolve[module][E2]);
287 }
288
289 // ignites a port on a given module
290 void ignite(byte module, byte port) //
291 {
292     transmitData(port);
293     setModule(module+1); // +1 because 0 (ALL_OFF) is turning off all modules
294
295     delay(IGNITION_TIME);
296
297     setModule(ALL_OFF);
298     transmitData(CLEAR_REG); // clear registers for safety reasons
299 }
300
301 // sets status led
302 void setLEDState(byte G,byte R,byte P)
303 {
```

```
304     digitalWrite(STATUS_RED,R);
305     digitalWrite(STATUS_GREEN,G);
306     digitalWrite(STATUS_PIEP,P);
307 }
308
309 // ##### // State Machine // #####
310
311 // does what the name implies
312 byte FSM( byte state )
313 {
314     byte PG = checkPGM();
315     byte TF = fire_pulse_counter >= FIRE_PULSE_SETOFF;
316     byte DN = pg_index == pg_stop;
317     byte WT = wait_counter <= FIRE_DELAY;
318     byte TA = digitalRead(TRIGGER_ARMED);
319     byte TC = digitalRead(TRIGGER_CONNECTED);
320     byte CA = digitalRead(THIS_ARMED);
321
322     byte new_state = state;
323
324     switch( state )
325     {
326         case STATE_PG_MODE:
327             Serial.println("!pgm:");
328             setLEDState(HIGH,HIGH,LOW);
329             while(1)
330             {
331                 programmMode();
332             }
333             break;
334
335         case STATE_INIT:
336             if(PG == 0) new_state = STATE_NO_TRIGGER;
337             else if(CA == 0 && TC == 1 && TA == 0) new_state = STATE_IDLE;
338
339             for( int i = 0; i < 3; i++ ){
340                 setLEDState(HIGH,HIGH,HIGH);
341                 delay(100);
342                 setLEDState(LOW,LOW,LOW);
343                 delay(100);
344             }
345
346             break;
347
348         case STATE_NO_TRIGGER:
349             if(PG == 1) new_state = STATE_PG_MODE;
350             else if(CA == 0 && TC == 1 && TA == 0) new_state = STATE_IDLE;
351
352             setLEDState(HIGH,LOW,LOW);
353             delay(250);
354             setLEDState(LOW,HIGH,LOW);
355             delay(250);
356             break;
357
358         case STATE_IDLE:
359             if(PG == 1) new_state = STATE_PG_MODE;
360             else if(TC == 0) new_state = STATE_NO_TRIGGER;
361             else if(CA == 1 && TA == 1) new_state = STATE_ARMED;
362
363             wait_counter = 0;
364 }
```

```
365     setLEDState(HIGH,LOW,LOW);
366     delay(100);
367     setLEDState(LOW,LOW,LOW);
368     delay(500);
369     break;
370
371 case STATE_ARMED:
372     if(TA == 0) new_state = STATE_IDLE;
373     else if(CA == 0) new_state = STATE_IDLE;
374     if(TF == 1) new_state = STATE_WAIT;
375     fire_pulse_counter = 0;
376
377     setLEDState(LOW,HIGH,HIGH);
378     delay(100);
379     setLEDState(LOW,LOW,LOW);
380     delay(500);
381
382     break;
383
384 case STATE_WAIT:
385     if(TA == 0) new_state = STATE_IDLE;
386     else if(TA == 0) new_state = STATE_IDLE;
387     else if(WT == 0) new_state = STATE_FIRE;
388
389     wait_counter++;
390
391     setLEDState(LOW,HIGH,HIGH);
392     delay(50);
393     setLEDState(LOW,LOW,LOW);
394     delay(50);
395
396     break;
397
398 case STATE_FIRE:
399     if(TA == 0) new_state = STATE_IDLE;
400     else if(TA == 0) new_state = STATE_IDLE;
401     else if(DN == 1) new_state = STATE_END;
402
403     section pgpoint = readProgramPoint(pg_index);
404
405     setLEDState(HIGH,LOW,LOW);
406     delay(pgpoint.msdelay);
407     setLEDState(LOW,HIGH,HIGH);
408     ignite(pgpoint.module,pgpoint.port);
409
410     pg_index++;
411
412     break;
413
414 case STATE_END:
415     setLEDState(LOW,HIGH,LOW);
416     break;
417
418 }
419
420 // ##### // Main Loop // #### //
421
422 void loop()
423 {
424     state = FSM(state);
425 }
```

```
426
427 void fire()
428 {
429     fire_pulse_counter++;
430 }
```

4.4 Recognitions

All circuits were drawn with the help of *KiCad* 6.0²⁰

All flowcharts and diagrams were drawn with the help of *Drawio*²¹

²⁰<https://www.kicad.org/>

²¹<https://app.diagrams.net/>