LOW ENERGY ELECTRON BEAM CHOPPER SYSTEM DESIGN

by

Arda Ünal

B.S., Electrical and Electronics Engineering, Gaziantep University, 2020

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Physics
Boğaziçi University
2024

ACKNOWLEDGEMENTS

I would like to show appreciation to my supervisors, Prof. Veysi Erkcan Özcan and Dr. Gökhan Ünel, for guiding me over the last three years. Dr. Gökhan Ünel always encouraged me to do what was in my mind, which gave me self-confidence, supported me in every possible way and answered every question that I asked.

I would like to express my sincere gratitude to Prof. Arif Nacaroğlu and Prof. Ahmet Bingül, who inspired me and supported my development throughout my undergraduate life. They greatly influenced the development of my personal and professional life.

I would like to thank my mother, Yasemin Ünal, who always believed in me, supported me, and never gave up on me. I owe her everything that I have made since I was born. Lastly, I would like to thank my friend Hüseyin Kılıç and my girlfriend Tuğba Kara, who supported me and helped me in many different ways throughout my life.

ABSTRACT

LOW ENERGY ELECTRON BEAM CHOPPER SYSTEM DESIGN

A new electron accelerator, Rhodotron, has been designed and constructed at Boğaziçi University Kandilli Detector, Accelerator, and Instrumentation Laboratory (KAHVELab). Rhodotron accelerator accepts an electron beam with a bunch length of at most 1 ns. Therefore, a chopper system is needed to accomplish 1 ns or less electron bunch from 1 μs electron bunch. This master's thesis aims to design a chopper system to bunch electrons for the Rhodotron accelerator.

ÖZET

DÜŞÜK ENERJİLİ ELEKTRON DEMETİ KESİM SİSTEMİ

Boğaziçi Üniversitesi Kandilli Dedektör, Hızlandırıcı ve Enstrümentasyon Laboratuvarı (KAHVELab) tarafından yeni bir elektron hızlandırıcısı dizayn edilip üretilmiştir. Rhodotron hızlandırıcısı maksimum 1 ns uzunluğunda demeti kabul etmektedir. Bu yüzden, μs uzunluğundaki elektron demetinin 1 ns uzunluğa kısıltılması gerekmektedir. Bu yüksek lisans tezinin amacı Rhodotron hızlandırıcısına girecek elektron demetini kısaltma sistemini dizayn etmektir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

$\vec{\boldsymbol{a}}$	Acceleration
$\vec{\boldsymbol{B}}$	Magnetic Field
c	Speed of Light
C	Capacitance
d	Distance
$\vec{\boldsymbol{E}}$	Electric Field
E_{total}	Total Energy
$\vec{\boldsymbol{F}}$	Force
I	Current
KE	Kinetic Energy
L	Inductance
$\vec{\boldsymbol{p}}$	Momentum
Q	Charge
R	Resistance
S	Area
$\vec{\boldsymbol{v}}$	Velocity
α	Deflection Angle
β	Beta Factor
γ	Lorentz Factor
ϵ	Emittance
ϵ_0	Permittivity
ϵ_{ijk}	Levi-Civita Identity
σ	Surface Charge
Ω	Ohm
$\vec{\boldsymbol{\nabla}}$	Gradient

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
ack	Acknowledgement
A	Ampère
API	Application Interface
B	Byte
C	Coulomb
CSV	Comma Separated Value
COM	Communication Port
CPU	Central Processing Unit
DC	Direct Current
D	Diode
DTL	Drift Tube Linac
EDA	Electronic Design Automation
EMI	Electromagnetic Interference
eV	Electron Volt
F	Farad
GCC	GNU Compiler Collection
GDB	GNU Debugger
GUI	Graphical User Interface
GPU	Graphical Process Unit
KAHVELab	Kandilli Detector Accelerator and Instrumentation Laboratory
HAL	Hardware Abstraction Layer
HRTIM	High Resolution Timer
Hz	Hertz
IC	Integrated Circuit
I/O	Input Output

K-D	K-Dimensional
LEBT	Low Energy Beam Transfer
LED	Light Emitted Diode
LHC	Large Hadron Collider
Linac	Linear Accelerator
m	Meter
MEBT	Medium Energy Beam Transfer
MCU	Microcontroller
nack	Not Acknowledgement
ODE	Ordinary Differential Equation
PCB	Printed Circuit Board
PLL	Phase Locked Loop
PWM	Pulse Width Modulation
rad	Radian
RAM	Random Access Memory
RCD	Resistor Diode Capacitor
RF	Radio Frequency
RFQ	Radio Frequency Quadrupole
ROM	Read Only Memory
s	Second
TEM	Transverse Electromagnetic
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous and Asynchronous Receiver Transmitter
USB	Universal Serial Bus
V	Volt

1. INTRODUCTION AND THEORY

Accelerators have been among the most important tools in research since the beginning of the 20th century. They bring electrons, protons, or high atomic number atoms to relativistic speeds, up to almost the speed of light, using RF cavities and powerful magnets. A wide variety of research disciplines benefit from the fast particles. Particle physicists have them collide to observe subatomic particles (such as Higgs boson). Chemists, materials scientists, and biologists use accelerators to produce X-rays to closely examine a wide variety of materials – from aircraft turbines to microchip semiconductors to vital proteins. There are numerous types of accelerators, such as linear accelerators, synchrotrons, rhodotrons, and so on.

These different types of acceleration methods have been developed to accelerate charged particles in the most efficient manner. The acceleration process occurs when charged particles pass through a longitudinal electric field. Electric fields are generally periodic and mainly depend on the accelerator cavity geometry. Therefore, the charged particle group, called a bunch, should be in phase with the electric field for acceleration. Achieving a synchronized bunch can be achieved with many different methods. One of the synchronization processes is bunching a continuous or partially continuous beam via deflecting plates.

Deflecting plates use electric fields to change charged particle trajectories. They are generally used for low-energy charged particles because the limitations in the magnitude of the electric field could create electric arcs. Therefore, they are generally placed on low-energy beam transfer (LEBT) and medium-energy beam transfer (MEBT) lines. Depending on the purpose of the application, the electric field could be applied continuously or periodically. While pulsed electric fields bunch a continuous beam or partially continuous beams, continuous electric fields change the trajectory of the particles.

Generating these electric fields requires high voltages. Thus, different methods have been developed since the 20th century. One of the most famous is the Marx generator, proposed by Erwin Otto Marx in 1924. The first purpose of the Marx generator is to test high-voltage components. Since then, the Marx generator has been used in fundamental science research areas such as high-energy physics, chemistry, and biology.

1.1. Particle Accelerators

Particle accelerators have many application areas, such as high-energy physics, chemistry, biology, and industry. There are different types of specifications to classify accelerators. For example, there are proton accelerators, electron accelerators and isotope accelerators. On the other hand, particle accelerators can be linear, circular or just a cavity. Many of these specifications greatly depend upon the application.

Protons weigh approximately two thousand more than electrons. Therefore, electrons can be accelerated more quickly than protons without using too much power. For example, electrons are accelerated by a static electric field in an electron microscope, while the Large Hadron Collider (LHC) at CERN needs RF cavities to accelerate protons.

Since electrons accelerate much more quickly, they radiate when they accelerate in different directions. Therefore, circular accelerators are more suitable for heavier particles. For example, circular accelerators can be used as storage rings because protons lose less energy than electrons when they accelerate. On the other hand, circular accelerators that accelerate electrons could be used as X-ray sources. Meanwhile, linear accelerators can be also used to accelerate charged particles with low masses, such as electrons.

Particle acceleration has many stages. Before acceleration, particles are injected into the machine. After that, diagnostic tools are used to identify the properties of injected particles, such as spatial distribution, emittance, and total current. Before acceleration, the beam can be chopped into bunches, and particles can be focused using electromagnetic lenses. After acceleration, particles gain energy. Therefore, magnetic fields like dipole and quadrupole can change their direction. However, acceleration must be done by an electric field because the magnetic field cannot provide energy.

One of the cavity electron accelerators is called Rhodtron. The Rhodotron accelerator was proposed by J. Pottier from the French Atomic Energy Agency (CEA) in 1989 [1]. IBA built the first industrial Rhodotron accelerator in 1993 [2].

1.2. Rhodotron

A Rhodotron accelerator is a continuous wave electron accelerator. Rhodotron cavities have the geometry of a shorted coaxial line and resonate at the $\lambda/2$ mode. Since both ends of the cavity are short, a transverse electromagnetic (TEM) field is established. Rhodotron accelerator mainly consists of a coaxial RF cavity and bending magnets around the cavity.

The electric field is in the radial direction, and magnetic fields are zero in the beam direction. Therefore, particles move in a radial axis. Electrons are accelerated by the electric field and directed by using bending magnets. There are four time phases for the acceleration of electron bunch:

1. Initially, electrons accelerate through the center of the cavity where the inner cylinder stays.
2. Secondly, they stay inside the inner cylinder, which behaves as a Faraday cage. During this time, the direction of the electric field is reversed.
3. In the third step, electrons exit the inner cylinder and accelerate through the cavity walls.
4. Finally, electrons enter the magnetic field, and their trajectory returns to the cavity.

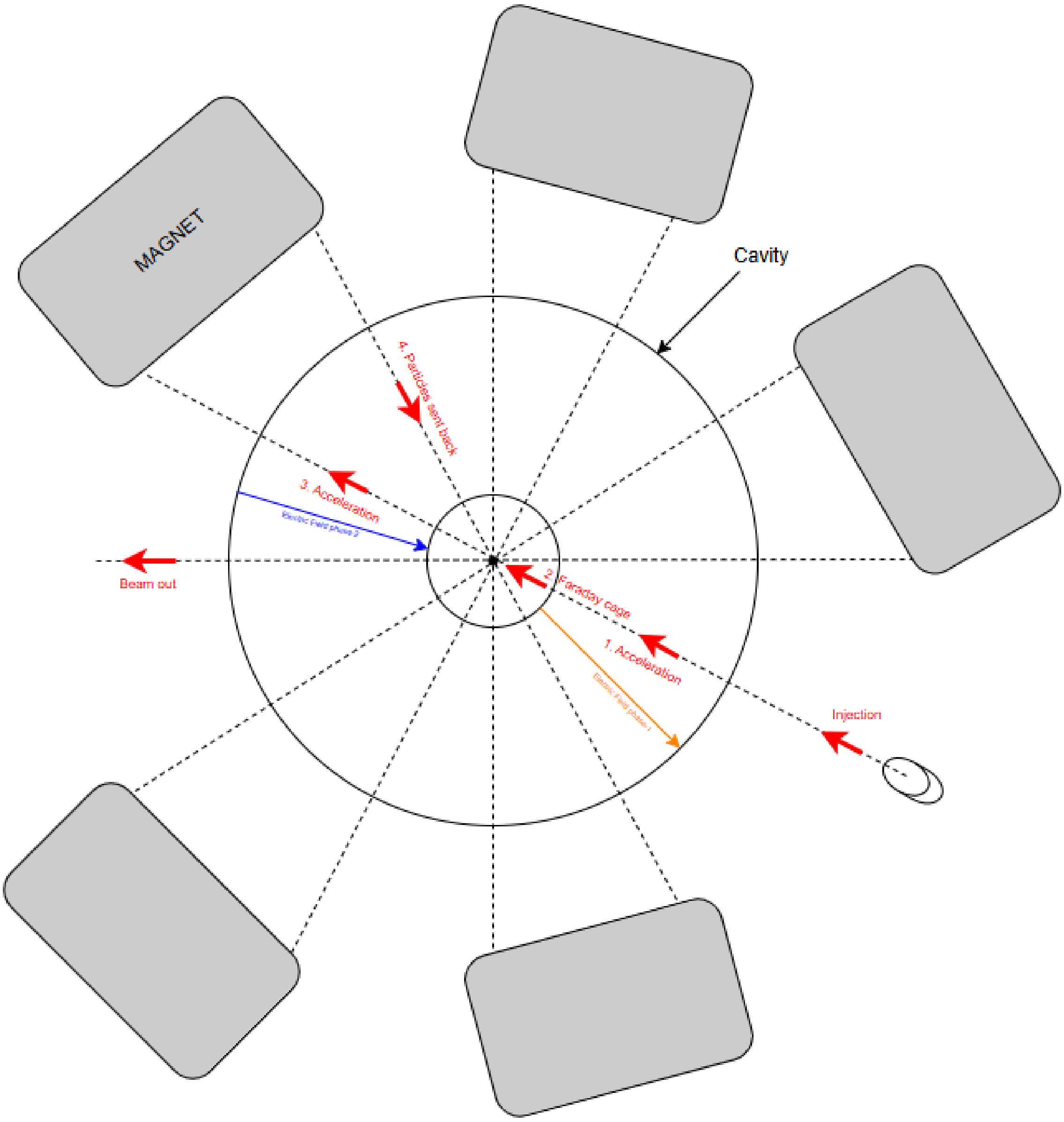

Figure 1.1. Rhodotron and acceleration cycles

1.2.1. Synchronization

In the Rhodotron accelerator, designed at KAHVELab, incoming electrons are accelerated to an energy between 2 MeV to 5 MeV. Rhodotron design simulations have shown that a 1 ns electron bunch can achieve the best synchronization of the electron bunch and the RF electric field [3]. Electrons outside 1 ns bunch will become unsynchronized and eventually separate from the synchronous bunch. These uncontrolled electrons could hit the cavity walls, causing damage to the cavity wall and unwanted radiation such as X-rays.

The electron gun, which produces electrons for the accelerator, can produce an electron bunch of a few μs length. Therefore, a new method should be found to decrease the length of this bunch to a 1 ns bunch. Based on the studies in the literature, a beam chopper design was deemed suitable to perform this function.

1.3. Beam Choppers

Accelerator systems use beam choppers to bunch a continuous or partially continuous beam. In general, the purpose of the chopper systems is to synchronize particles with the electric field of the acceleration section of the accelerator. The working principle of the beam choppers is to accelerate particles transversely by applying an electric field in the transverse direction of the beam. Charged particles follow a circular trajectory while passing through choppers. Therefore, there will be an angle between the initial and final paths. Although there are many different types of chopper systems, I will discuss three types of chopper systems.

1.3.1. Parallel Plate Choppers

Parallel plates are the most basic chopper configurations used in many accelerator systems as beam choppers, which can deflect a beam. When charged particles pass through a parallel plate with applied voltage on one side, they are exposed to an electric field. While negatively charged particles are accelerated to the reversed direction of the electric field, positively charged particles are accelerated to the direction of the electric field. Therefore, their trajectory changes with the paths they are supposed to follow.

Furthermore, low-energy particles are deflected more than high-energy particles because they spend more time in the electric field. Therefore, parallel plates can be used as separators as well. Moreover, parallel plates can be used as kickers. If the voltage applied on the parallel plate is pulsed, only some of the particles interact with the electric field. Therefore, using this analogy, a beam can be bunched, or unwanted bunches can be kicked. These principles are generally used to choose the bunches before radio-frequency quadrupole (RFQ) or drift tube linac (DTL).

1.3.1.1. Electric Field of a Parallel Plate. There are two plates parallel to each other, and voltage is applied to one of them, which creates an electric field directed to the other plate. Therefore, an electric field is formed between plates. The electric field between plates is constant, which can be shown using Maxwell's equations. Assuming an infinite plane carries a uniform surface charge, σ, then Gauss's law states that

$$\vec{\nabla} \cdot \vec{E} = \frac{\sigma}{\epsilon_0} \tag{1.1}$$

where $\vec{E}$ is electric field, $\vec{\nabla}$ is gradient operator and ϵ_0 is vacuum permittivity. In integral form, Gauss's law can be rewritten as:

$$\oint_S \vec{E} \cdot \vec{ds} = \frac{Q_{enc}}{\epsilon_0}. \tag{1.2}$$

As shown in Figure 1.2, a pillbox has six faces, but only two of them are perpendicular

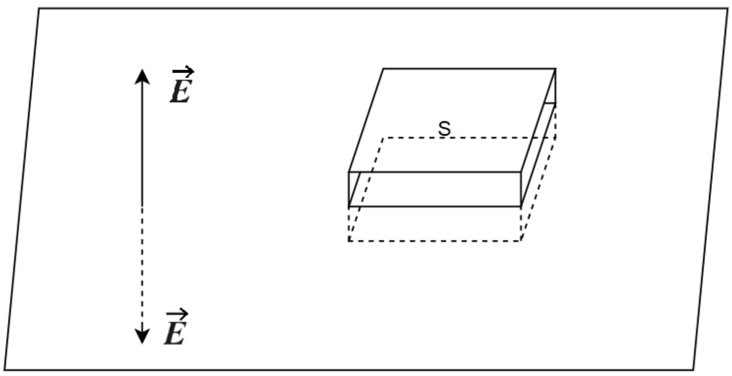

Figure 1.2. Infinite plane with Gaussian pillbox

to the electric field. Therefore, considering the upper and lower faces, the integral has two electric field components are:

$$2ES = \frac{Q_{enc}}{\epsilon_0}, \tag{1.3}$$

$$\vec{E} = \frac{\sigma}{2\epsilon_0}\hat{z}. \tag{1.4}$$

If two plates are parallel to each other, but one has a negative charge distribution, as seen in Figure 1.3, then their electric field can be summed because, now, electric field directions are the same. Thus, the electric field vector becomes:

$$\vec{\boldsymbol{E}} = \frac{\sigma}{\epsilon_0}\hat{\boldsymbol{z}}. \tag{1.5}$$

Moreover, the electric field between parallel plates can also be formulated using the fundamental relationship between potential and electric field as follows:

$$V = -\int_C \vec{\boldsymbol{E}} \cdot \vec{\boldsymbol{dl}}. \tag{1.6}$$

In our case, it becomes:

$$V = -\int_0^d \vec{\boldsymbol{E}} \cdot \vec{\boldsymbol{dz}}. \tag{1.7}$$

Figure 1.3 shows electric field lines are directed in the positive z-axis. Therefore,

$$\vec{\boldsymbol{E}} = \frac{V}{d}\hat{\boldsymbol{z}} \tag{1.8}$$

which also represents a constant electric field between plates.

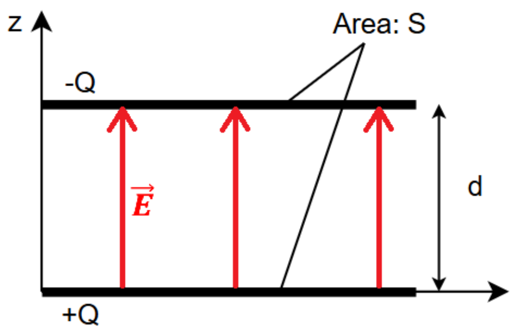

Figure 1.3. Parallel plates with opposite signed charge distribution

1.3.1.2. Capacitance of a Parallel Plate. Using the two equations of the electric field, the capacitance of the parallel plate can be found as follows:

$$\frac{V}{d} = \frac{Q}{S\epsilon_0} \tag{1.9}$$

$$Q = CV \tag{1.10}$$

where

$$C = \epsilon_0 \frac{S}{d} \tag{1.11}$$

is called capacitance, and S is the area of the plates. The capacitance value depends purely on the geometry of the object.

1.3.2. Slow Wave Choppers

Slow wave choppers consist of a dielectric plate and strip line with a meander line shape, as shown in Figure 1.4. A meandering strip line structure is used to synchronize the deflecting electric field with the bunch velocity. As a result, the deflecting electric field propagates at the speed of the beam. As a train of bunches passes through the chopper plate, only chosen bunches are affected by the chopper field. [4]

They are mainly used where high-frequency pulses are required. High frequency and high voltage pulses have rise and fall times in the order of a few ns with a high repetition rate. The strip lines generally have a 50 Ω characteristic impedance to achieve maximum efficiency and minimum distortion. Additionally, dielectric materials below strip lines are also special materials, such as Rogers' duroid RT/6002, to minimize pulse distortion and losses. [5]

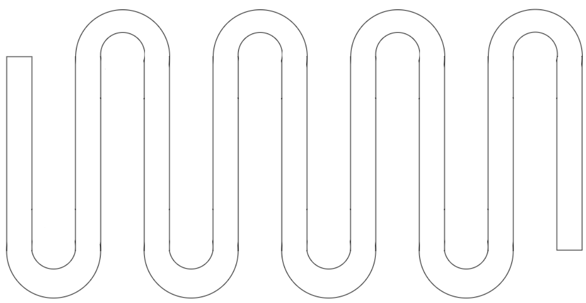

Figure 1.4. Meander strip line structure

1.3.3. Hybrid Systems as Choppers

Hybrid systems, also called ExB, consist not only of an electric field but also of an electromagnetic field. Magnetic fields can be provided either by permanent magnets or by electromagnetic coils. There are few examples of hybrid systems yet, but an example system has been designed at the Frankfurt Neutron Source FRANZ for the LEBT line [6]. A permanent dipole provides a magnetic field on the beamline. There is also a parallel plate that works with high-voltage pulses.

When a beam enters the hybrid chopper, its trajectory is bent by a magnetic field in the absence of an electric field, as shown in Figure 1.5. When the parallel plate is pulsed, the electric field force on the particles becomes equal to the magnetic field force. Therefore, only the allowed particles enter the accelerator, and the rest of the beam is ejected.

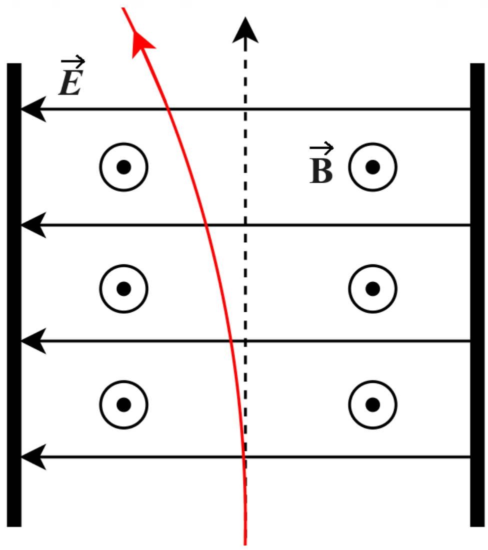

Figure 1.5. Hybrid chopper system. The red line represents the beam trajectory in the absence of the electric field.

1.4. Motion of Charged Particles

1.4.1. Lorentz Force

Lorentz Force is a combination of electric and magnetic field force acting on a charged point particle, and it is formulated by using Newton's second law as:

$$\vec{\boldsymbol{F}} = \frac{\partial \vec{\boldsymbol{p}}}{\partial t} = m\vec{\boldsymbol{a}} = q(\vec{\boldsymbol{E}} + \vec{\boldsymbol{v}} \times \vec{\boldsymbol{B}}) \tag{1.12}$$

where $\vec{\boldsymbol{p}}$ is classical momentum $m\vec{\boldsymbol{v}}$, $\vec{\boldsymbol{v}}$ is the velocity of the particle and $\vec{\boldsymbol{a}}$ is the acceleration.

1.4.2. Relativistic Acceleration in 3D

Including special relativity, new momentum definition becomes $\vec{\boldsymbol{p}} = \gamma m\vec{\boldsymbol{v}}$ where γ is called Lorentz Factor and defined as:

$$\gamma = \frac{1}{\sqrt{1-\frac{v^2}{c^2}}} = \frac{1}{\sqrt{1-\beta^2}} = \frac{1}{\sqrt{1-\vec{\boldsymbol{\beta}}\cdot\vec{\boldsymbol{\beta}}}} \tag{1.13}$$

and $\vec{\boldsymbol{\beta}}$ is defined as:

$$\vec{\boldsymbol{\beta}} = \frac{\vec{\boldsymbol{v}}}{c} \tag{1.14}$$

where c is the speed of light. Therefore, Newton's second law changes dramatically:

$$\vec{\boldsymbol{F}} = \frac{\partial \vec{\boldsymbol{p}}}{\partial t} = \frac{d}{dt}(\gamma m_0 \vec{\boldsymbol{v}}) = m_0\frac{d\vec{\boldsymbol{v}}}{dt}\gamma + m_0\frac{d\gamma}{dt}\vec{\boldsymbol{v}} \tag{1.15}$$

where m_0 is rest mass of the particle. Time derivatives of relativistic momentum and gamma factor can be rewritten as:

$$\frac{\partial \gamma}{\partial t} = \frac{\partial}{\partial t}\Big(1-\frac{v^2}{c^2}\Big)^{-1/2} = \frac{\gamma^3}{c}\vec{\boldsymbol{\beta}}\cdot\vec{\boldsymbol{a}}, \tag{1.16}$$

$$\frac{\partial \vec{\boldsymbol{p}}}{\partial t} = m_0\Big[\frac{\gamma^3}{c}(\vec{\boldsymbol{\beta}}\cdot\vec{\boldsymbol{a}})\vec{\boldsymbol{v}} + \gamma\frac{\partial \vec{\boldsymbol{v}}}{\partial t}\Big]. \tag{1.17}$$

Thus, relativistic correction of the Newton's second law becomes

$$\vec{\boldsymbol{F}} = \gamma m_0 \Big[\vec{\boldsymbol{a}} + \gamma^2 (\vec{\boldsymbol{\beta}} \cdot \vec{\boldsymbol{a}}) \vec{\boldsymbol{\beta}} \Big]. \tag{1.18}$$

Acceleration due to Lorentz Force in 3D can be found as:

$$\gamma m_0 \Big[\vec{\boldsymbol{a}} + \gamma^2 (\vec{\boldsymbol{\beta}} \cdot \vec{\boldsymbol{a}}) \vec{\boldsymbol{\beta}} \Big] = q(\vec{\boldsymbol{E}} + \vec{\boldsymbol{v}} \times \vec{\boldsymbol{B}}), \tag{1.19}$$

$$a_i + \gamma^2 \delta_{lm} \beta_l a_m \beta_i = \frac{q(E_i + \epsilon_{ijk} v_j B_k)}{\gamma m_0}, \tag{1.20}$$

$$a_i (1 + \gamma^2 \beta_i^2) + \gamma^2 (\vec{\boldsymbol{\beta}} \cdot \vec{\boldsymbol{a}} - \beta_i a_i) \beta_i = \frac{q(E_i + \epsilon_{ijk} v_j B_k)}{\gamma m_0}, \tag{1.21}$$

$$a_i = \frac{1}{1 + \gamma^2 \beta_i^2} \left[\frac{q(E_i + \epsilon_{ijk} v_j B_k)}{m_0 \gamma} - \gamma^2 (\vec{\boldsymbol{\beta}} \cdot \vec{\boldsymbol{a}} - \beta_i a_i) \beta_i \right]. \tag{1.22}$$

1.4.3. Parallel Plate's Deflection Angle

A charged particle follows a circular trajectory under a constant transverse electric field. Since the particle has a circular path, motion can be considered as circular motion. Therefore, the radius of this circle can be found via Newton's second law:

$$\frac{mv^2}{r} = qE \tag{1.23}$$

where $\frac{mv^2}{r}$ is called centrifugal force and E is electric field. If relativistic effects are taken into account, then the equation can be rewritten as:

$$\frac{pv}{r} = qE \tag{1.24}$$

where $p = \gamma m_0 v$ is relativistic momentum and m_0 is rest mass of the particle. Relativistic kinetic energy and momentum formulas can be combined as follows:

$$\frac{KE}{p} = \frac{(\gamma - 1) m_0 c^2}{\gamma m_0 v} \tag{1.25}$$

where kinetic energy is defined as $KE = E_{total} - m_0c^2$ and $E_{total} = \gamma m_0 c^2$ is total energy of the particle. Therefore, it can be rewritten as:

$$pv = \beta^2 \frac{\gamma KE}{\gamma - 1}, \tag{1.26}$$

$$pv = \beta^2 E_{total}. \tag{1.27}$$

Therefore, the radius of the trajectory of the charged particle can be rewritten as:

$$r = \frac{\beta^2 E_{total}}{qE}. \tag{1.28}$$

The deflection angle, α, of the particle in the transverse electric field can be defined as: [7]

$$\alpha = \frac{\ell}{r} = \frac{qE\ell}{\beta^2 E_{total}} \tag{1.29}$$

where ℓ is the path length in a constant transverse electric field, as shown in Figure 1.6.

The electric field has a constant value, $E = V/d$, between parallel plates, where V is the voltage on parallel plates and d is the distance between parallel plates. Thus, the angle of the deflection for a parallel plate can be written as:

$$\alpha = \frac{qV\ell}{d\beta^2 E_{total}}. \tag{1.30}$$

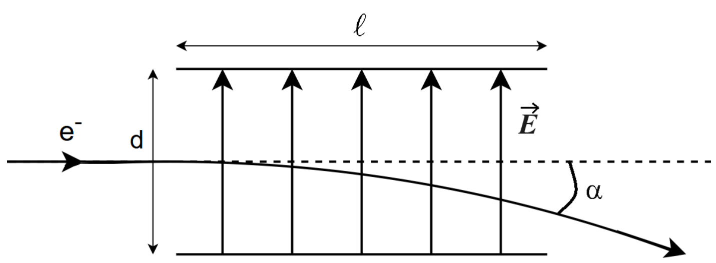

Figure 1.6. Deflection angle of the electron passing through a parallel plate

1.4.4. Emittance

Emittance is an ellipse in phase space that visualizes the beam envelope. It measures a particle beam's spatial and angular spread in beam dynamics. It is derived from Hill's equation, $x'' + K(s)x = 0$, which is the equation of the motion. Here, s is the actual, or designed, trajectory of the synchronized particles, as shown in Figure 1.7. The solution of the equation of motion is

$$x = \sqrt{\epsilon b} \cos \phi \tag{1.31}$$

and ϵ is called the emittance and is defined as:

$$\epsilon = \beta^2 x'^2 + 2x' x\alpha + \gamma x^2 \tag{1.32}$$

where α, β and γ are called the Courant and Snyder Twiss parameters. Here, x is the coordinate axis whereas x' can be defined as:

$$x' = \frac{dx}{ds} = \frac{dv_x}{dv_z} = \frac{dp_x}{dp_z} \approx \frac{dp_x}{dp}. \tag{1.33}$$

Therefore, its unit is mrad.

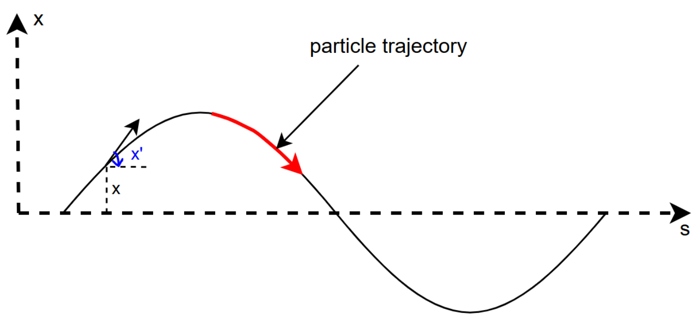

Figure 1.7. Particle trajectory in an accelerator

The emittance is a constant of motion because of Liouville's theorem, which states that the particle density in phase space is constant under the influence of conservative forces. Therefore, under electromagnetic forces, ϵ, is invariant. The area of the ellipse in phase space is $\pi\epsilon$, as seen in Figure 1.8. [8]

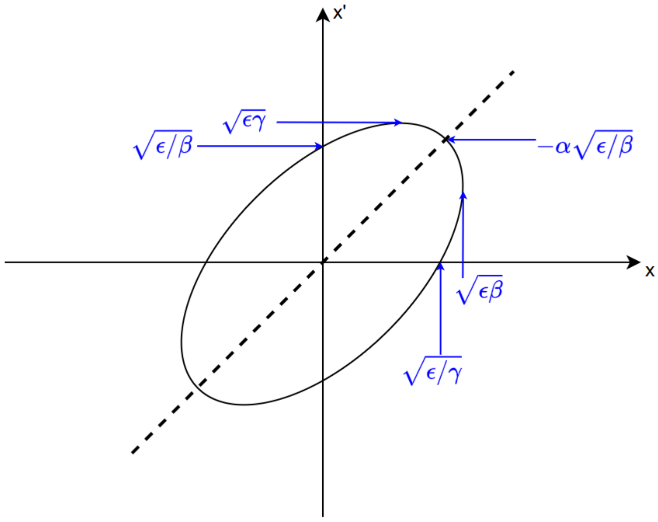

Figure 1.8. Phase space with emittance ellipse

1.5. Marx Generator

Marx generator, proposed and designed by Erwin Otto Marx in 1924, generates high-voltage pulses using a low-voltage DC source. Although it has limited application areas, it is frequently used in fundamental science research, such as high-energy physics, biology, chemistry, and electrical engineering. [9]

Marx generator consists of capacitors, spark gaps, and a DC voltage source, as shown in Figure 1.9. The circuit consists of N stages with an N capacitor. Initially, capacitors are parallel to DC voltage sources. Since they charge in parallel, all capacitors have the same voltage if all the components are identical. Whenever they reach an enough voltage, a breakdown occurs between the spark gaps. During breakdown, the capacitors are connected in series through the spark gaps. Therefore, each capacitor's voltage is added in series. The peak output voltage on the R_{load} is equal to $V_{out} = NV_{DC}$. After the pulse occurs, capacitors again start to charge, and all the stages arise repeatedly.

In every stage, resistors, capacitors, and spark gaps are identical. In some applications, it might be important to achieve high rise and fall time for the output pulse of the Marx generator. Identically placed components and reducing inductance, sourced by connectors and components, will help to achieve high rise and fall times at the circuit's output. Resistor and capacitor values should be chosen according to repetition rate and output power, respectively. The repetition rate depends on the $\tau = RC$ time constant value, which directly affects the charging time of the capacitors. Conversely, resistor values must be high enough to force current into spark gaps during breakdown and reduce leakage currents of capacitors. C capacitor values are important for output power since $Q = CV$ or $I = C\frac{dV}{dt}$. If the circuit has capacitors with high capacitance values, they can produce a large current.

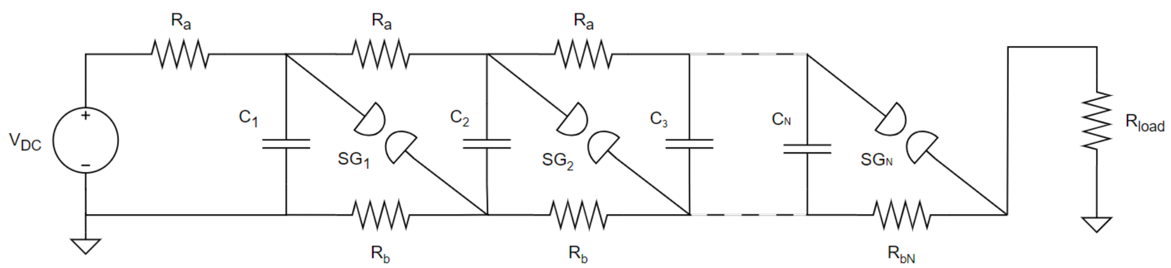

Figure 1.9. Marx Generator circuit schematic

1.5.1. Solid-State Marx Generator

The solid-state Marx Generator is an upgrade of the classical Marx Generator by replacing spark gaps with semiconductor switches, such as MOSFETs and IGBTs. Using semiconductor switches has many advantages. They allow the control of pulse time, pulse duration, and peak voltage level. Another advantage of the semiconductor switches is that capacitors do not need to discharge completely. This means that by using a capacitor with a huge capacitance value, one can achieve almost perfect square-shaped high-voltage pulses. Furthermore, using MOSFETs as a switch allows extremely high rise and fall times.

Moreover, spark gaps are heavy, occupy large areas, and become corrupted with time. Meanwhile, semiconductor switches occupy less space, are lightweight, and have no corruption with time if thermal protection is good enough. Additionally, resistors are replaced with semiconductor components, which reduces the leakage currents and decreases capacitors' charging and discharging time.

Despite the advantages of using semiconductor switches, they also have some disadvantages. It is not easy to design gate drivers for high-side switches, and it is not easy to achieve synchronization. A solid-state Marx Generator could achieve high-voltage square pulses with extremely small rise and fall times by careful design. There are two types of solid-state Marx generators.

1.5.1.1. Solid-State Marx Generator Type-1. Solid-state Marx Generator type-1 consists of diodes and semiconductor switches instead of resistors and spark gaps, respectively, as shown in Figure 1.2. Diodes have low active resistance, which hugely reduces the charging time of capacitors. Therefore, repetition frequency can be increased. Additionally, they block reverse leakage current between the capacitors. On the other hand, the semiconductor switches control output pulse width, pulse duration, and repetition rate. Moreover, the output voltage can be directly controlled by changing the input voltage, V_{DC}, which is another massive advantage of a solid-state Marx Generator compared to a classical one. [10]

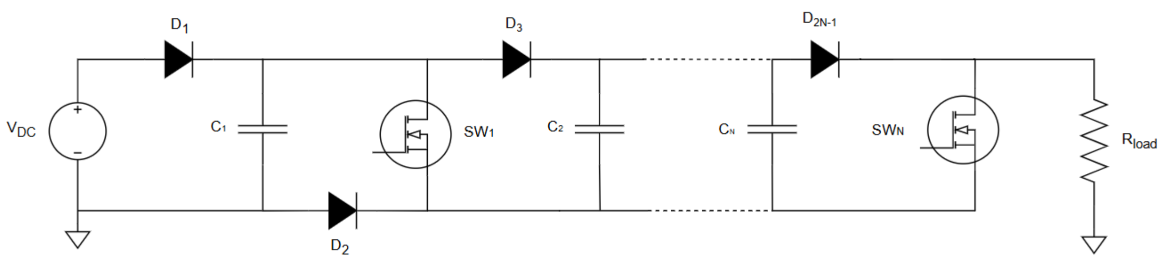

Figure 1.10. Solid-state Marx Generator Type-1

1.5.1.2. Solid-State Marx Generator Type-2. Solid-state Marx generator type-2 has an improved switching topology. It often uses a more synchronized method to control current flow, enhancing efficiency and reducing switching loss. Type-2 is usually favoured in applications demanding ultra-fast rise times, higher peak voltages, and more precise pulse shapes, such as advanced particle accelerators and medical imaging technologies. The only difference between type-1 and type-2 topologies is the replacement of the bottom diodes, 2N, by semiconductor switches, as shown in Figure 1.11.

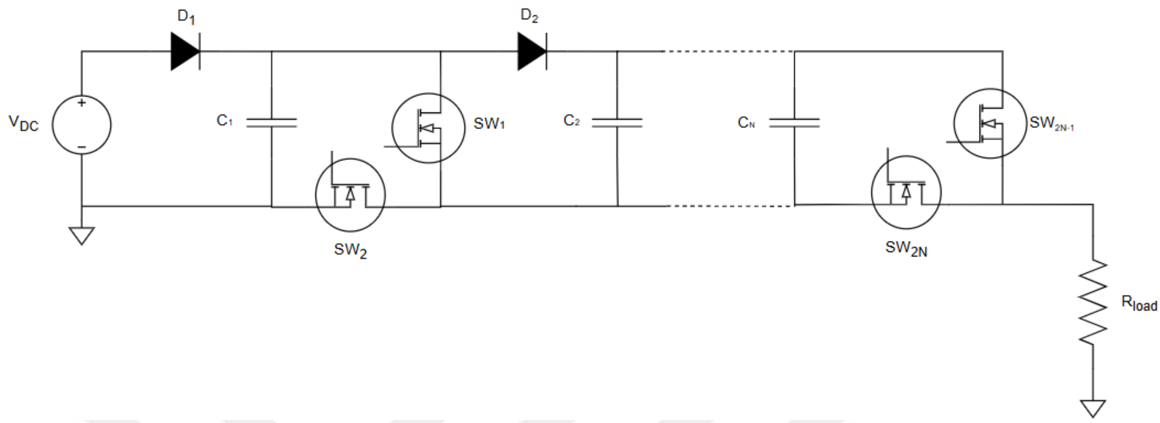

Figure 1.11. Solid-state Marx Generator Type-2

Solid-state Marx Generator type-2 have several advantages over type-1. The most crucial advantage of type-2 is that if the load is an energy storage circuit element, type-2 provides a fast connection to the ground as compared to type-1. This might be important to reduce the fall time of the pulses. Therefore, this topology is much more useful for chopper applications.

There are only two operating modes of Marx Generator type-2. In the charging mode, switches 1, 3, ..., 2N-1 are off, switches 2, 4,.., 2N are on, and capacitors are charging in parallel, as seen in Figure 1.12. In this stage, diodes and switches can be replaced with their R_{on} resistances, and semiconductor switches are open circuits by assuming there are no inductive parasitic effects, such as the layout of the PCB, connectors and component packages. [11]

In the discharging mode, switches 1,3,...,2N-1 are on, switches 2,4,..,2N are off, and capacitors are now connected in series, as seen in Figure 1.13. Therefore, the capacitors' voltages are summed up at the output, so the total output voltage is approximately equal to NV.

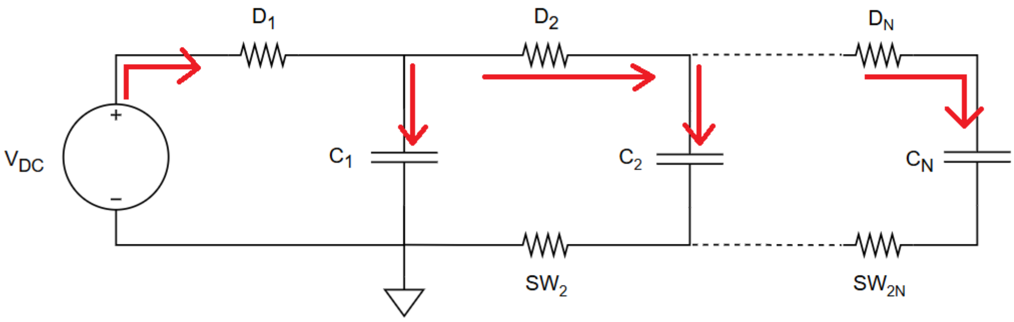

Figure 1.12. Type-2 charging

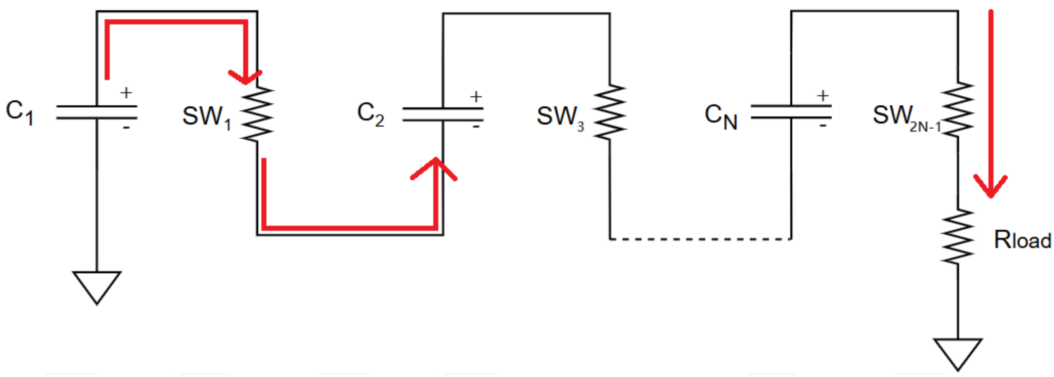

Figure 1.13. Type-2 Discharging

1.5.2. Silicon Carbide(SiC) MOSFET

SiC MOSFET is a type of power MOSFET with a wide bandgap that can be used as a semiconductor switch for the Marx generator. SiC MOSFETs also have very short rise and fall times. This makes them extremely useful for high-frequency applications. Meanwhile, the larger bandgap makes SiC MOSFETs less sensitive to temperature and able to withstand higher electric breakdown fields. The higher critical electric field allows for higher breakdown voltages without compromising resistance, R_{on}.

Many different types of SiC MOSFET can be found in the market. As an example, the physical model of an n-channel SiC MOSFET consists of N^+ substrate, N^- drift region, P^+ base region, N^+ region, gate oxide layer, source, drain, and gate, as shown in Figure 1.14. [12]

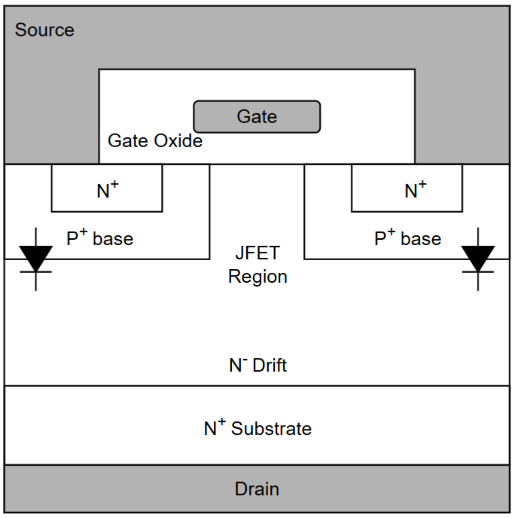

Figure 1.14. Structure diagram of an n-channel SiC MOSFET device

Although there are three output characteristics of a SiC MOSFET, there are only two operating modes of SiC MOSFET when it is used as a switch:

(i) ON State: When a positive voltage is applied to the gate terminal relative to the source, it creates an electric field that attracts electrons, forming a conductive channel between the source and drain terminals. This conductive channel allows current to flow.

(ii) Off State: Applying zero or negative voltage to the gate turns off the electric field and blocks current flow.

The electrical equivalent circuit of a SiC MOSFET consists of three parasitic capacitors and a body diode, as shown in Figure 1.15. C_{GS} and C_{GD} are called parasitic input capacitances. The effective input capacitance, $C_{iss} = C_{GS} + C_{GD}$, is a parasitic capacitance that directly influences the rise time of the pulse. C_{DS} represents the capacitance of the body diode, which affects reverse recovery time and charge. Therefore, C_{DS} and the body diode are directly associated with the ringing effect. Finally, the body diode blocks current flow when MOSFET is in the off state and also provides a path for inductive loads.

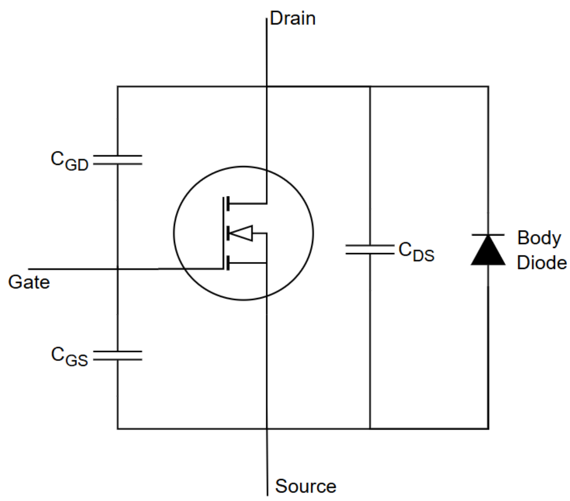

Figure 1.15. Electrical equivalent circuit of a SiC MOSFET

1.5.3. Gate Drivers

Every stage of the solid-state Marx generator type-2 essentially consists of a half-bridge topology, as seen in Figure 1.16. In this configuration, one MOSFET's source is connected to the ground, called the low side, while the other's source is connected to the load, called the high side. The gate voltage should be higher than the source voltage to use a MOSFET in on mode. Therefore, a special technique must be used to drive high-side MOSFETs.The literature contains many procedures for driving high-side MOSFETs, but only two of them are mentioned here. [13]

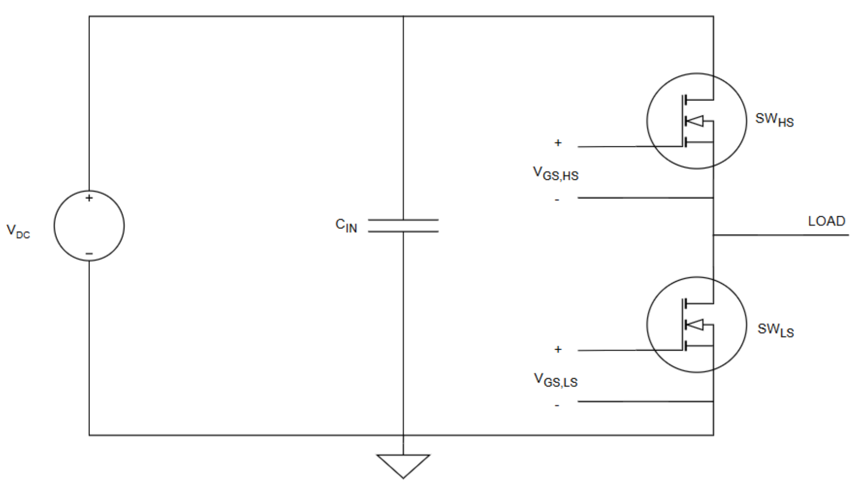

Figure 1.16. Half-bridge topology

<u>1.5.3.1. Double-Ended Isolation Transformer Gate Driver.</u> A double-ended isolation transformer consists of a ferromagnetic and three windings. The first winding is connected to a low-voltage pulse width modulation (PWM) source, and the other two windings are connected to MOSFETs. High-side MOSFET inductor windings are wrapped in the same direction as the primary winding, while the low-side is wrapped in the reverse direction. Therefore, when the primary side is high, the low-side MOSFET turns off, and the high-side MOSFET turns on.

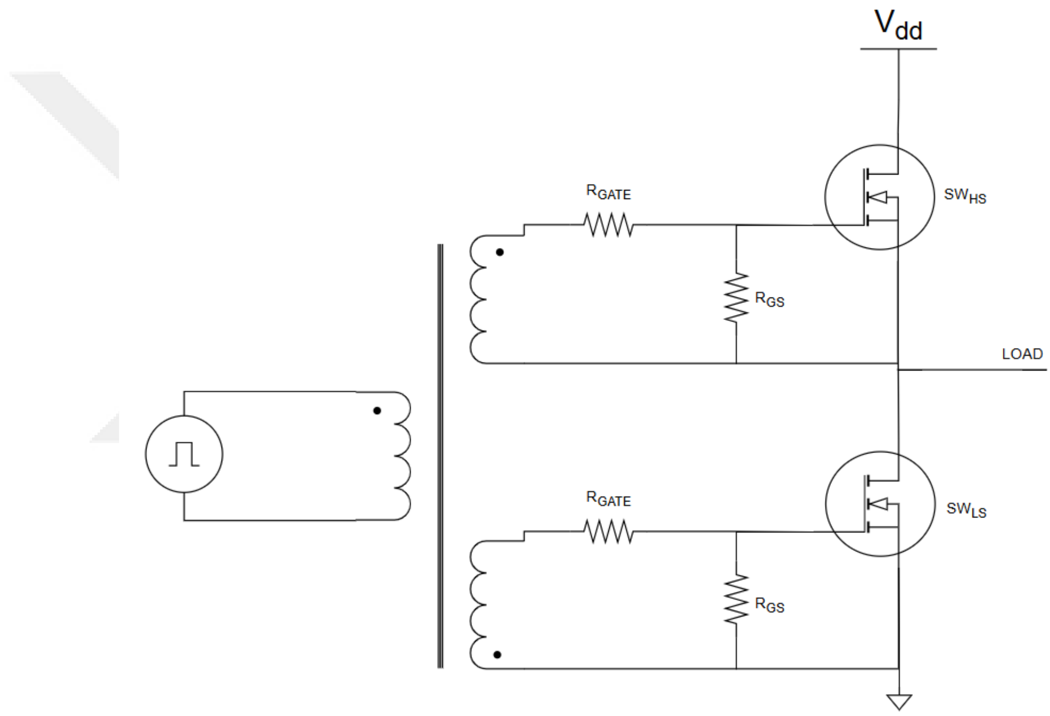

Figure 1.17. Basic double-ended isolated transformer gate driver topology

<u>1.5.3.2. Optocoupler Driver IC.</u> An optocoupler is a device which transfers electric signals between input and output using light, as shown in Figure 1.18. Therefore, it provides electrical isolation between input and output. Thus, optocouplers are useful devices to drive high-side MOSFETs. Optocoupler ICs generally consist of an LED and a photosensor. The LED converts the input signal into light, and the photosensor converts incoming light into an electric signal.

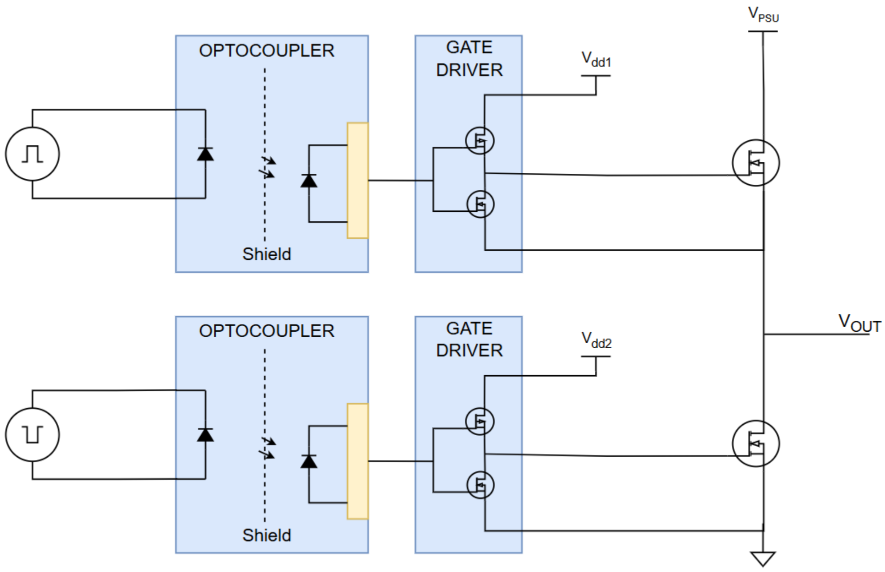

Figure 1.18. Simple structure of Optocoupler gate driver topology

1.6. Microcontrollers

A microcontroller (MCU) is an integrated circuit that contains a central processing unit (CPU), random access memory (RAM), read-only memory (ROM), timers, and input/output (I/O) peripherals, as seen in Figure 1.19. MCUs are generally used in embedded applications. Since 1969, they have been used in control systems. Depending on the architecture, processors could have 8-bit, 16-bit, 32-bit or 64-bit word length. They generally have an internal oscillator for timing and an external clock support. Maximum processing frequency also depends on the architecture.[14] There are some important concepts in microcontrollers:

(i) Timer: is an electronic counter that counts time precisely. It consists of a counter and a register. The counter counts incoming digital pulses and increments the register. Timers are used to generate PWM signals, schedule tasks and event synchronization.

(ii) Interrupt: An interrupt is a signal that temporarily stops the normal execution of a program to allow the microcontroller to address a high-priority task. Two types of interrupts exist in microcontrollers.

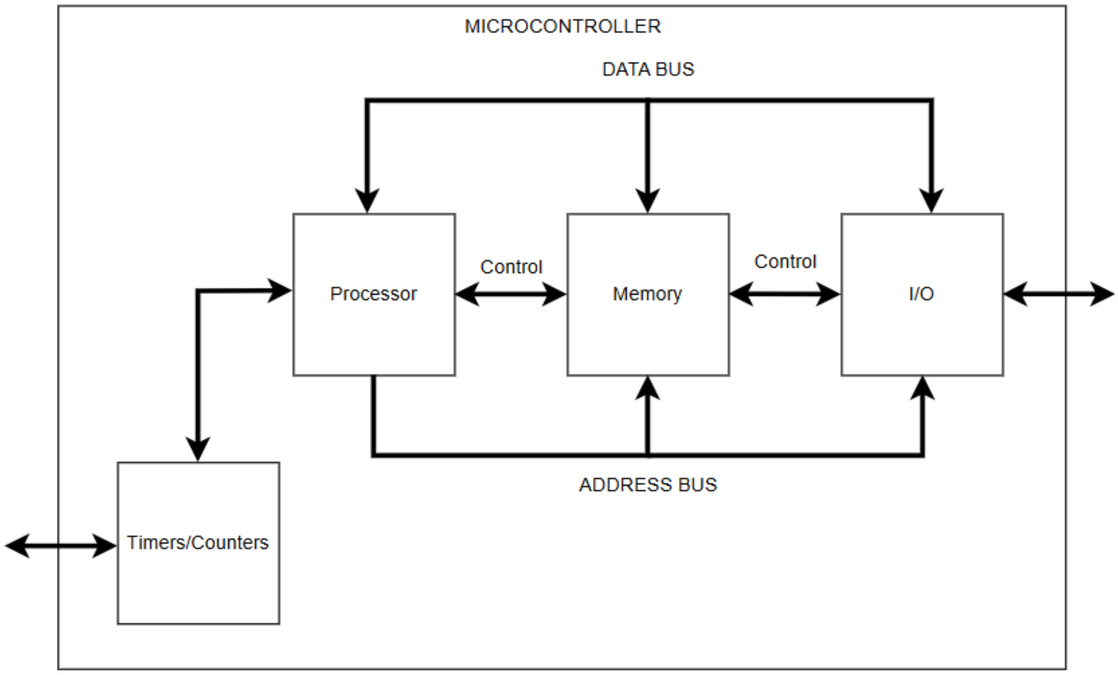

Figure 1.19. A simple block diagram of microcontrollers

(a) Hardware Interrupt: A hardware interrupt occurs when an external condition is sent to notify MCU.

(b) Software Interrupt: A software interrupt occurs when an internal condition is sent to notify MCU.

(iii) High Resolution Timer (HRTIM): HRTIM is a highly accurate timer that can generate complex waveforms such as PWM and phase-shifted. It comprises six 16-bit up counters and a controller timer with very fine timing resolution.[15]

2. TOOLS

A chopper system design consists of many stages. Various tools were used for particle dynamics simulations, electronic circuit design and simulations, and microcontroller programming. This chapter describes these tools and their necessity in the design.

2.1. CST Studio Suite

The CST Studio Suite is a commercial tool for analyzing, simulating, and designing highly complex systems and components. It includes solvers for different systems such as static electromagnetic solver, time domain solver, frequency solver and particle tracking solver. Moreover, CST supports multiphysics integration to perform hybrid simulations of electromagnetic, thermal, and mechanical effects to evaluate real-world performance. Therefore, its applications span various industries such as telecommunications, electronics, aerospace, automotive and biomedical.

2.2. KiCad EDA

The KiCad application is a free and open-source electronic design automation (EDA) tool capable of developing PCB layouts and creating circuit diagrams. KiCad EDA has many features, including a schematic editor, PCB editors with auto-router, a 3D board rendering tool, and a library and footprint editor. KiCad can create hierarchical schematics, generate multi-layer PCBs and include many features such as differential pair routing, DRC checker and net highlighting for efficient designs. It is frequently preferred by hobbyists, academics and even professionals. Today, professionals at CERN prefer KiCad, and CERN contributes to the development of the KiCad.

2.3. LTspice

LTspice is free SPICE simulator software that includes an analog circuit simulator, a schematic designer, and a waveform viewer. It also offers additional circuitry enhancements and models to improve the quality of the simulation for analog circuits. Its graphic schematics capture interface enables users to tap on the schematics and get simulation outputs.

2.4. STM32CubeIDE

STM32CubeIDE is a comprehensive embedded C/C++ development platform designed for STM32 microcontrollers and microprocessors. It offers a range of features, including peripheral configuration, code generation, compilation, and debugging. It is built on the Eclipse framework. It uses the GNU compiler collection (GCC) toolchain for development and GNU Debugger (GDB) for debugging, where GNU is an extensive collection of free software.

Furthermore, STM32CubeIDE features STM32CubeMX functionalities for configuring and creating STM32 projects. STM32CubeMX is used to configure the peripherals and middleware before programming. According to configurations, the hardware abstraction layer (HAL) library provides user API, which allows users to use pre-written functions and definitions.

2.5. Software Tools

2.5.1. Leapfrog Algorithm

The Leapfrog method is a type of finite difference method that approximates the solution of an ordinary differential equation (ODE) by discretizing both time and space. It is widely used in simulating physical systems. The idea comes from numerical integration and differentiation, where the slope of a chord between two points on a function, (x_0, f_0) and (x_1, f_1), is a much better approximation of the derivative at the midpoint than at either end. Therefore, the same idea can be applied to ODEs. The algorithm is as follows:

$$\begin{aligned} a_i &= F(x_i), \\ v_{i+1/2} &= v_{i-1/2} + a_i \Delta t, \\ x_{i+1} &= x_i + v_{i+1/2} \Delta t, \end{aligned} \tag{2.1}$$

where Δt should be constant for stability. More stabilization can be done by the method so-called kick-drift-kick method, which is generally used whenever variable time steps are used, as follows:

$$\begin{aligned} v_{i+1/2} &= v_i + a_i \frac{\Delta t}{2}, \\ x_{i+1} &= x_i + v_{i+1/2} \Delta t, \\ v_{i+1} &= v_{i+1/2} + a_{i+1} \frac{\Delta t}{2}. \end{aligned} \tag{2.2}$$

2.5.2. K-D Tree Nearest Neighbour Search Algorithm

A k-d tree (k-dimensional tree) is a binary search tree optimized for arranging points in k-dimensional space, making it efficient for range searches. It orders the partition size at each tree level, with each node representing more or fewer points. From this, creating a tree that divides the space into left and right subtrees usually involves choosing a scale mean.

Queries such as category search and nearest neighbour search use iteration and backtracking traces to find points efficiently. This typically achieves an average time of $O(log(n))$. Although it works well on low-dimensional data, performance deteriorates in high dimensions due to the curse of dimensionality. Applications include computer graphics, robotics, and machine learning.

2.5.3. Numba Module

Numba is a just-in-time compiler for Python, optimized for code using NumPy arrays, functions, and loops. It is generally used with decorating functions, which serves as a hint to Numba that it should compile the function. A decorator is a function that takes a function as a parameter and returns a function. When a Numba decorated function is called, it's compiled to machine code just before execution and runs at native machine code speed.

The Numba library has the functionality to handle floating point numbers much better than Python itself. For example, the "nopython" compilation mode works by compiling the decorated function to execute entirely without relying on the Python interpreter. Therefore, the Numba library is an excellent choice for the physics simulations.

Numba can also support CUDA GPU programming by compiling part of the restricted subset of Python code to CUDA kernels and device functions that follow the CUDA execution model. The kernels can now access NumPy arrays directly integrated with seamless transfer between CPU and GPU. Numba's CUDA support contains tools for defining and managing the thread hierarchy resembling the functionality available in NVIDIA's CUDA C language.

3. DESIGN

In this chapter, the concept of the chopper system is examined alongside the high-voltage pulse hardware and timing system. Overall, the system design consists of three stages:

1. Chopping system
2. High-voltage pulse hardware
3. Timer system

First, a chopper system is designed to extract 1 ns electron bunch from 1 μs electron bunch. Then, hardware is chosen and designed to fulfil the high-voltage pulse for parallel plates. Finally, a timer system is selected to control high-voltage pulse times.

3.1. Chopper System

The Rhodotron's chopper system has unique requirements. It must be able to bunch a 1 μs beam with 45 keV kinetic energy into a 1 ns beam. No such problem was revealed in the literature, so the system was designed from scratch.

The designed chopper system consists of two parallel plates, a timer, and two high-voltage pulser circuits. The timer is used to synchronize high-voltage pulses and change pulse widths. The beamline consists of an electron gun, a solenoid, and two parallel plates. The beam pipe diameter is 80 mm. The first parallel plate deflects the electron beam outwards by applying a high-voltage pulse, creating a 15-degree angle in the z-direction. The second parallel plate is placed parallel to the bent bunch, pulling the bunch up so that the electron bunch is almost in the z-direction again. The rest of the alignment corrections can be done using steering magnets and solenoids.

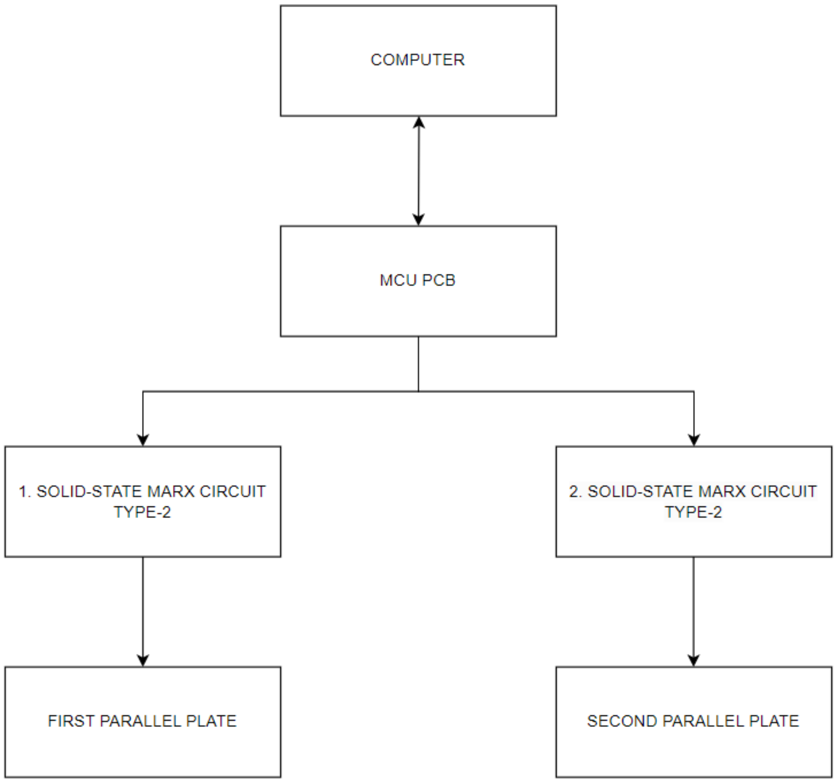

Figure 3.1. A simple block diagram of working principle of chopper system.

The length of the parallel plates was chosen as 118.15 mm, corresponding to 1 ns of the bunch length, as shown in Figure 3.2. The distance between two parallel plates is 40 mm. The area of the parallel plates is 0.0059075 m^2. Furthermore, it was decided to separate parallel plates by 380 mm to avoid field interactions between plates because they might affect their capacitance, as seen in Figure 3.3. However, it is a flexible parameter, so it could be shortened if needed.

The capacitance of the parallel plates can be calculated by using the equation 1.11:

$$C = 8.854 \times 10^{-12} \times \frac{0.0059075}{0.04} = 1.3 \ pF. \tag{3.1}$$

Incoming beam kinetic energy equals 45 keV, corresponding to $\beta = 0.3941$. Therefore, the deflecting particle trajectory with a 15-degree angle can be calculated by using the equation 1.30:

$$V = \frac{\alpha d \beta^2 \epsilon}{\ell} = 7638.9\ Volts \tag{3.2}$$

where $E_{total} = \gamma m_0 c^2 = 0.556\ MeV$ and $m_0 c^2 = 0.511\ MeV$.

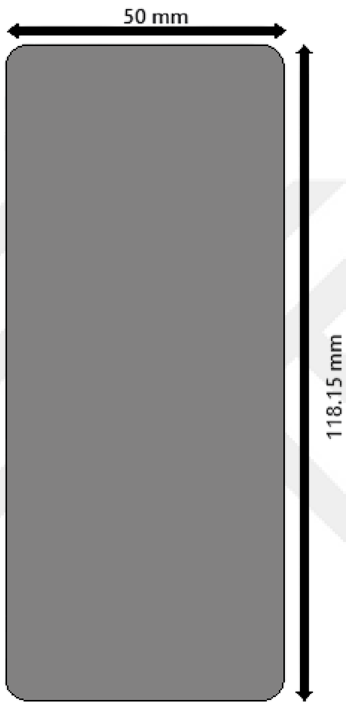

Figure 3.2. The dimensions of the chopper plate

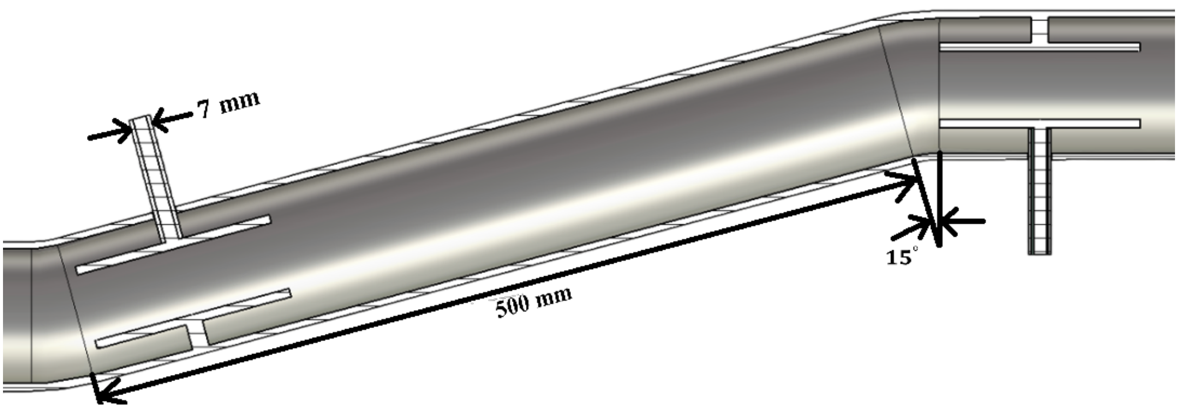

Figure 3.3. Designed beam pipe with dimensions

3.2. High-Voltage Pulse Hardware

The solid-state Marx generator type-2 is chosen to generate the necessary high-voltage pulses. Marx generator type-2 has many advantages, but the main reason behind this choice is to have flexibility in pulse magnitude, pulse frequency and pulse width.

Solid-state Marx generator type-2 consists of two SiC MOSFETs.In our system, C2M1000170D from Wolfspeed is used, which has a 19 ns rise time and 63 ns fall time. Its drain-to-source voltage is equal to 1700 V, which allows a wide range of variable voltage at the output. Despite its limited pulsed current capability, it is well-suited to our design. Two SiC Schottky diodes, C4D02120A from Wolfspeed, have also been selected to block the inverse current between capacitors.

To simplify the design, an optoelectronic gate driver IC is used to drive MOSFETs, SI8261BBD-C-IS, which has 10 kV surge withstand capability and isolation up to 5000 V_{rms}. It has a low-power diode emulator and silicon isolation technology. It has a maximum rise time of 15 ns and a maximum fall time of 20 ns. Since the chopper system frequency is planned to operate with a maximum of 10 Hz, any heat-related analyses or simulations are unnecessary.

3.2.1. Snubber Design

In hard-switched MOSFET circuits, some of the energy stored in parasitic inductance and capacitance causes unwanted oscillation, also called ringing, at the output waveform. The ringing leads to switching energy losses as well. Additionally, when SiC MOSFET turns off, it could be exposed to overvoltage spikes, which could cause the MOSFET to fail. The unwanted parasitic capacitance and inductances are sourced from component packages and printed circuit board (PCB) layout.

To overcome the ringing issue, the PCB layout could be optimized to reduce parasitic inductance, the gate driver circuit could be improved, and capacitors and component packages could be chosen as low inductive and capacitive. Although all the necessary cautions are taken, there will be an inevitable oscillation in the output. An effective way to reduce ringing is to design a snubber circuit to improve output waveform and reduce switching losses. Therefore, a resistor-capacitor-diode (RCD) snubber circuit is designed to reduce ringing in a Marx generator unit. [16]

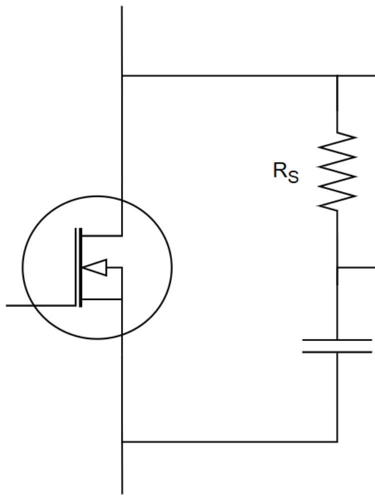

Figure 3.4. RCD snubber circuit implementation to a MOSFET

An RCD snubber consists of a resistor parallel to a diode, which is connected to a capacitor in series, as shown in Figure 3.4. When designing an RCD snubber, rise time should be taken into account because the chosen capacitor value directly affects current rise time according to $I = C\frac{\Delta V}{\Delta t}$, where is the maximum peak switch current, ΔV is the peak voltage the capacitor will charge to and Δt is the rise time of the voltage. On the other hand, RCD snubber can be used as a clamp circuit, which clamps maximum ringing voltage when the switch is turned on. The clamp snubber circuit aims to minimize oscillations but will also cause a slow-rising edge in the output waveform. In this design, the snubber circuit design is intended to clamp maximum voltage and achieve the maximum possible rise time, chosen as 50 ns. The peak reverse recovery current of the chosen SiC MOSFET is 3 A. Therefore, if V_{DC} is assumed to be 500 V, then R_s can be calculated as

$$R_s = \frac{V_{DC}}{I_{max}} = \frac{500}{3} = 166.667 \ \Omega \tag{3.3}$$

where V_{DC} is the maximum voltage on switch and I_{max} is peak reverse recovery current. The snubber capacitor value can be calculated by

$$\frac{\Delta V}{\Delta t} = \frac{0.632 \times V_{DC}}{C_s R_s} \tag{3.4}$$

where 0.632 is the time interval of time constant, $\tau = C_s R_s$. Thus,

$$C_s = \frac{0.632 \times 500}{10^{10} \times 166.67} = 190 \ pF. \tag{3.5}$$

Therefore, optimal values are chosen as $C_s = 200 \ pF$ and $R_s = 200 \ \Omega$ and rise time becomes 60 ns.

3.2.2. Gate Driver Circuit

The gate driver circuit of the MOSFET essentially consists of a driver IC, boost capacitor, R_{on} and R_{off} resistors, a pull-down resistor and a diode to specify the path of the current, as seen in Figure 3.5. The external gate resistors play a crucial role in achieving an optimized output waveform. While low resistance values cause overshooting, high resistance values cause high rise time. Therefore, optimization is necessary when choosing external gate resistors. The gate driver equivalent circuit consists of R_{driver}, $R_{G(ext)}$, L_s, $R_{G(int)}$ and C_{iss}, where R_{driver} is the driver IC resistance for high and low, $R_{G(ext)}$ is external resistor to be chosen, L_s is parasitic inductance, $R_{G(int)}$ is MOSFET internal gate resistance and C_{iss} is total input capacitance, $C_{iss} = C_{gs} + C_{gd}$, as seen in Figure 3.6. To eliminate ringing, the quality factor of this oscillator circuit must be greater than 0.5 (critically damped). Oscillation frequency can be found experimentally. Therefore, a rough calculation can be done as a starting point. The Miller point, the point after the Miller region where V_{gs} starts to increase for the second time, is found in the datasheet as $Q_{gate} = 17 \ nC$ at $V_{gs} = 10 \ V$. Therefore, the optimal resistor's resistance value calculation is [17]

$$I_{gate} = \frac{Q_{gate}}{t_{on}} = 0.2833 \ A \tag{3.6}$$

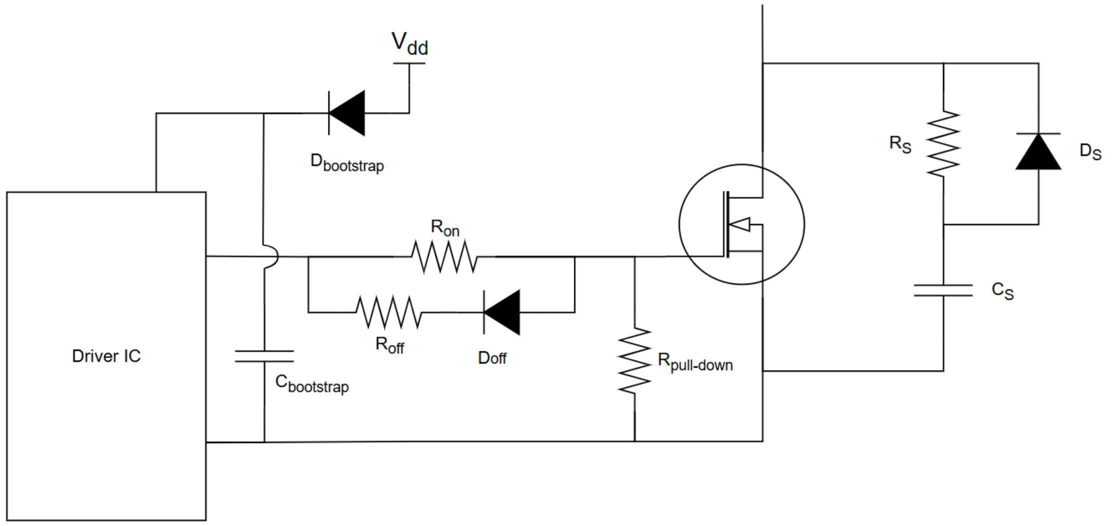

Figure 3.5. Power MOSFET driver circuit

where t_{on} is rise time of the signal. Thus, the total gate resistance is

$$R_{G(total)} = \frac{V_{driver} - V_{gs}}{I_{gate}} = \frac{20 - 10}{0.2833} = 35.3 \ \Omega \tag{3.7}$$

and $R_{G(ext)} = R_{G(total)} - R_{G(int)} - R_{driver} = 30.2 \ \Omega$. Starting from this value, an optimal gate resistance can be experimentally calculated for the best rise time and low ringing corresponding to the turn-on and the turn-off paths of the gate driver circuit.

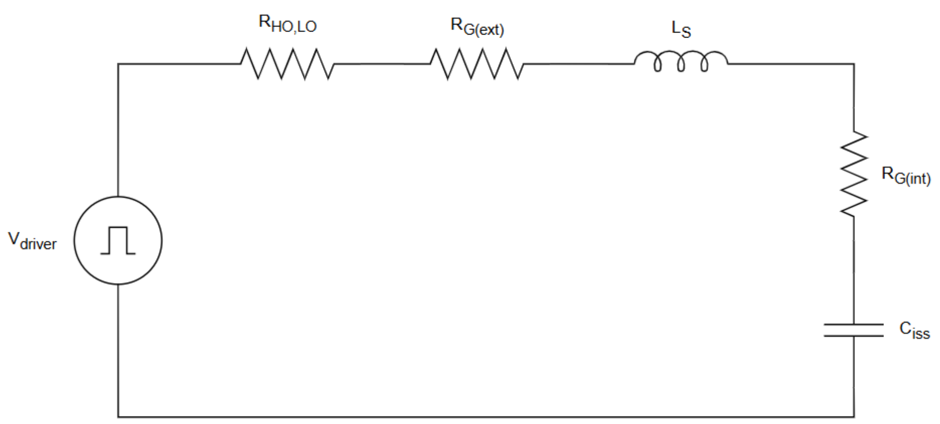

Figure 3.6. Gate driver equivalent circuit

Another important component is the bootstrap capacitor of the driver IC. The bootstrap capacitor has two important duties in the driver circuit. The first duty is to work as a bypass capacitor for the driver IC. The second one is that when the high-side

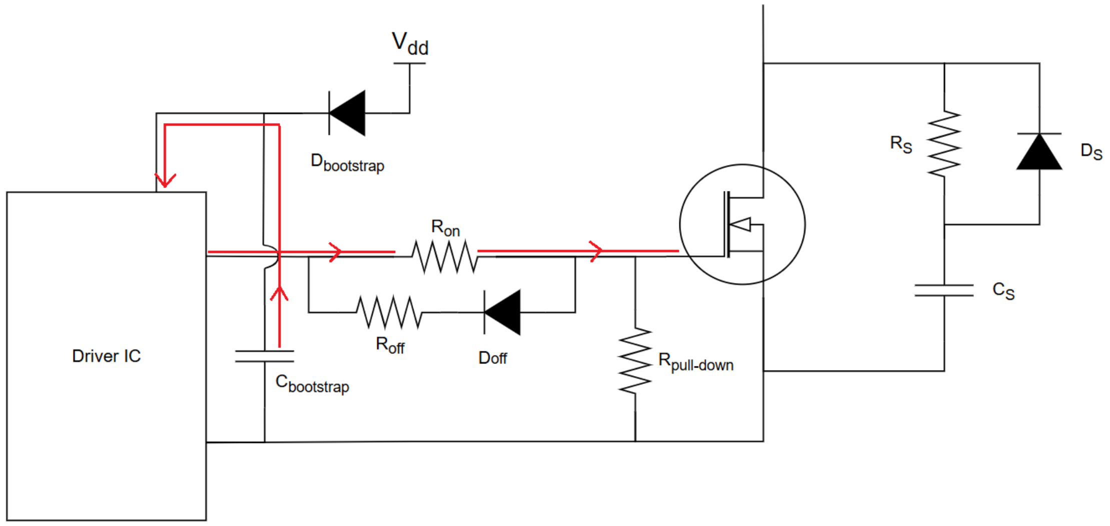

Figure 3.7. MOSFET ON current path

MOSFET is turned on, its source voltage will be changed due to the floating ground of Marx generator circuit units. Therefore, the diode $D_{bootstrap}$ will be turned off, and the only power source left will be $C_{bootstrap}$. Power is extracted from $C_{bootstrap}$ as shown in Figure 3.7 during the on state of MOSFET. Thus, $C_{bootstrap}$ should be chosen carefully. As a rule of thumb, the bootstrap capacitor's capacitance value should be at least ten times the MOSFET input gate capacitance value, C_{iss}. Therefore, $C_{bootstrap} = 10 \times C_{iss} = 2.15\ nF$. Alternatively, the minimum capacitance value of $C_{bootstrap}$ can be calculated according to pulse time for high-side MOSFET. The total gate charge can be calculated by:

$$Q_{gate(total)} = Q_{gate} + I_{leakage} t_{on} + Q_{ls} \tag{3.8}$$

where Q_{gate} is the gate charge of the MOSFET, $I_{leakage}$ is leakage current during on time, and Q_{ls} is the charge required by the internal level shifter, which is typically 3 nC.

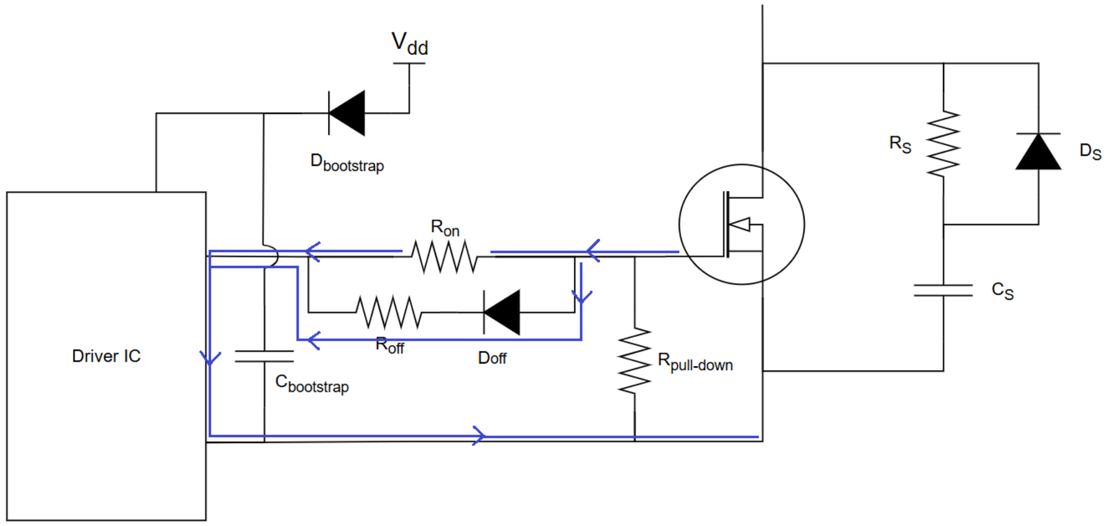

Figure 3.8. MOSFET OFF current path

Therefore,

$$C_{bootstrap} = \frac{Q_{gate(total)}}{\Delta V_{boot}} \tag{3.9}$$

where ΔV_{boot} is the maximum allowed voltage drop during on time. Using this equation, the minimum capacitance value of $C_{bootstrap}$ can be found as 3.215 pF.

Additionally, a Schottky diode with a low forward voltage drop and low junction capacitance should be used as $D_{bootstrap}$ to minimize losses associated with the diode's reverse recovery characteristics and ground noise bounce. Finally, although driver IC cannot produce negative voltage to turn off MOSFET, it was decided to use a R_{off} resistor and a diode, D_{off}, to set falling edge time, as seen in Figure 3.8. MOSFET off path, then, has two resistors parallel to each other.

3.2.3. PCB Design

Based on calculations and design parameters, components are chosen and two Marx generator unit PCBs are designed. The first design is optimized for handmade PCBs, and the second is produced by a third-party PCB producer. During designs, two essential goals are taken into account. The first one is to reduce parasitic inductance. Parasitic inductance sources are long copper routings and electromagnetic interference (EMI) between copper routings. Additionally, a ground plane should be used below routings to shorten the return path of the high-frequency signals. The ground plane also reduces parasitic inductance because almost all the magnetic fields are cumulated between the ground plane and the signal route. The second important design consideration is the isolation between low-voltage and high-voltage sides. Note that these PCBs are prepared to prove the concept. A more compact and optimized design should be done later.

<u>3.2.3.1. Homemade PCB Design.</u> Some optimizations were necessary during the design of the homemade PCB because of the limitations of the production process. A few additional pin header connectors are used to connect grounds because via could not be used. Additionally, routings are much longer and broader. The bottom ground planes are separated on the high and low sides. Moreover, it is mirrored when PCB is printed on paper and then pressed into a copper plate. To avoid mirroring problems, signal routes are drawn at the bottom, and the ground plane is drawn at the front.

<u>3.2.3.2. Produced PCB Design.</u> Third-party production has many advantages compared to homemade PCB production, such as using vias, thinner routings, etc. Since there is no need for high current, routings for high voltage lines also can be thinner. Furthermore, mounting holes are added. Another difference between prototypes is that the pulse capacitor is changed from a film capacitor to a ceramic capacitor.

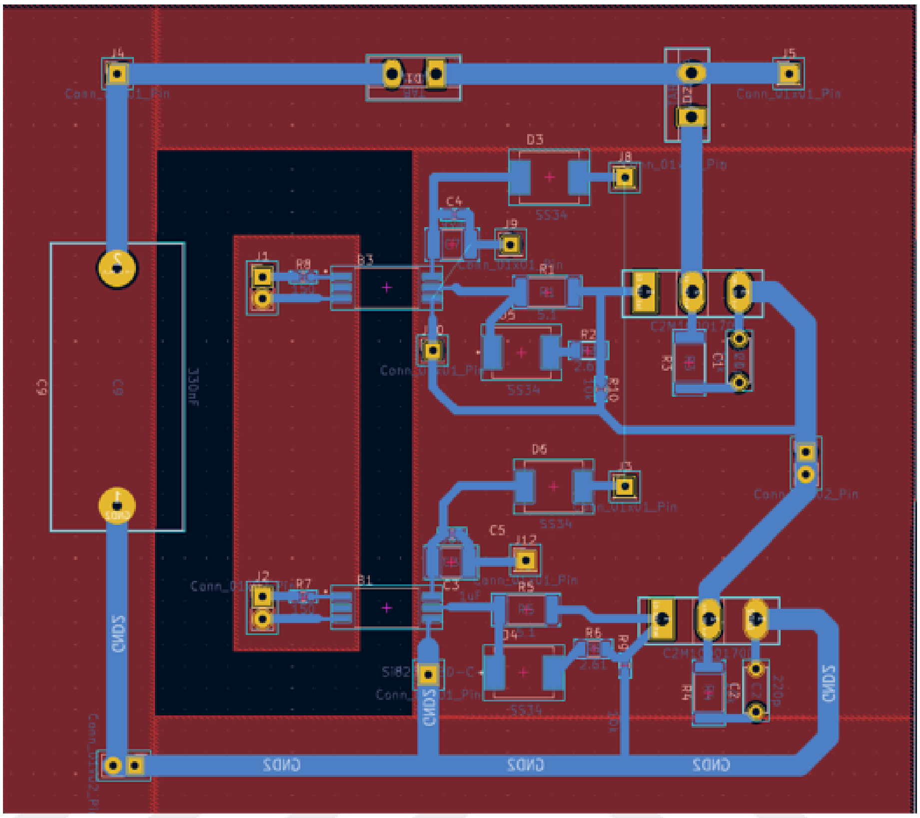

Figure 3.9. 2D PCB editor view of the homemade PCB

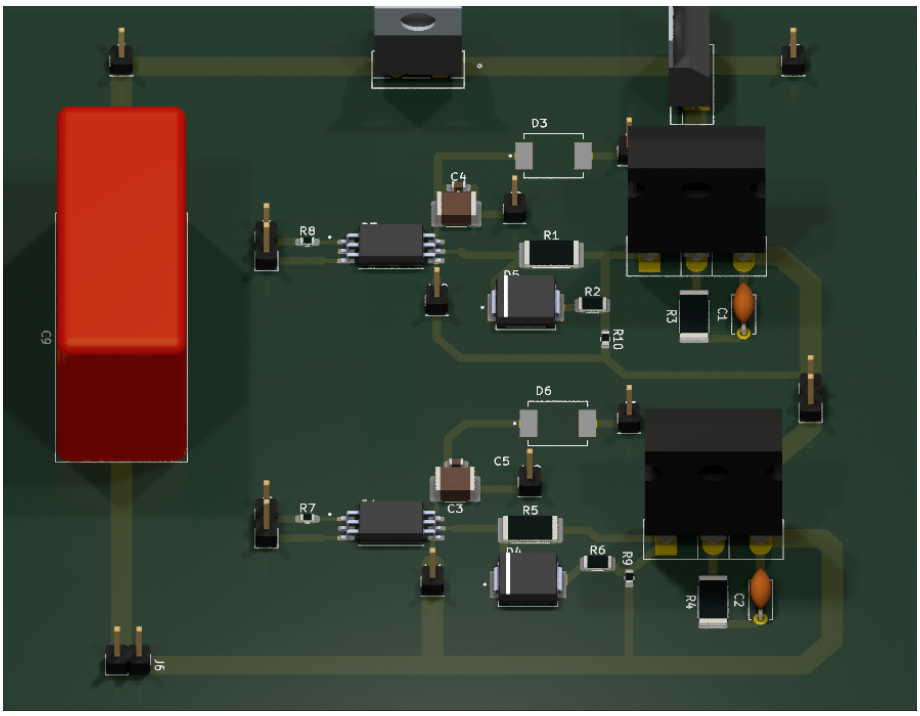

Figure 3.10. 3D PCB editor view of the homemade PCB

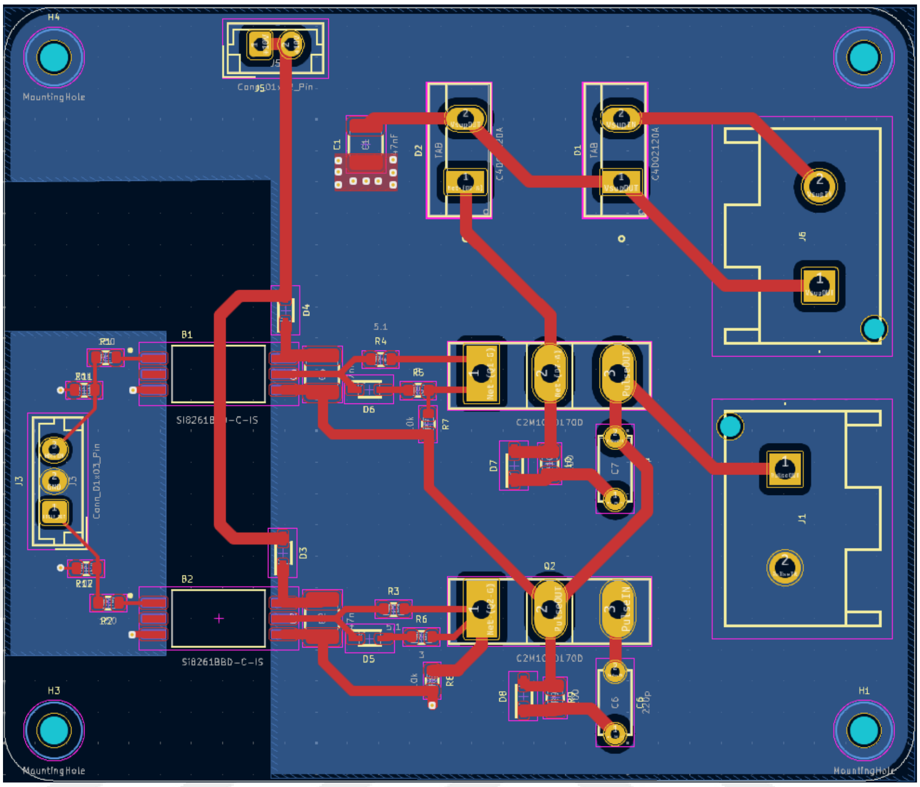

Figure 3.11. 2D PCB editor view of the produced PCB

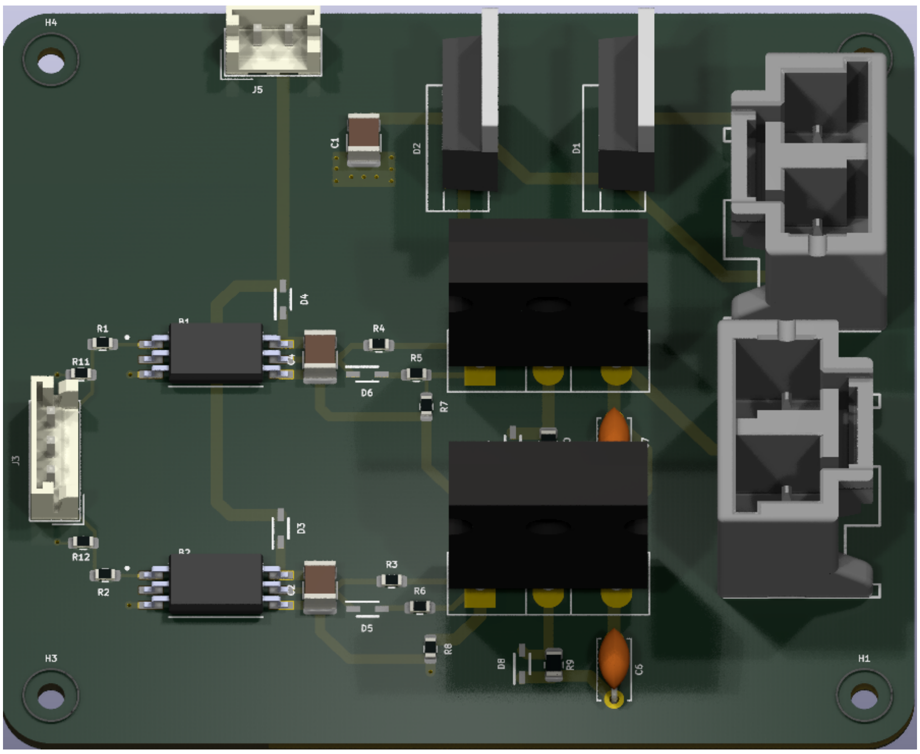

Figure 3.12. 3D PCB editor view of the produced PCB

3.3. Timer System

The timer system is another crucial design step of the chopper system. The timer system controls high-voltage pulse widths and the delay between two high-voltage pulses. Therefore, it should have picosecond resolution to achieve 1 *ns* electron bunch. Field programmable gate arrays (FPGA) are generally used to achieve such precision timers. However, some MCUs with picosecond timers are found after deep research. These MCUs are initially used for switched power supplies and consist of a particular hardware called HRTIM. Therefore, it was decided to use an MCU to control the Marx generator and decrease the complexity of the design.

An MCU with the development kit, STM32F3348DISCOVERY, from ST microelectronics is selected. A program is developed for the MCU using the C programming language. In addition, a graphical user interface (GUI) to control the MCU is developed using the Python programming language.

3.3.1. MCU Software

The MCU software is designed using STM32CubeIDE, which automatically creates driver libraries and user application interfaces (API) called hardware abstraction layer (HAL) library. The design starts with the MCU's pinout and clock configuration, which can be done using STM32CubeIDE software. Two peripheral pins, USART Rx and Tx, are used for USART communication. Two peripherals are used for HRTIM, which consists of two timers, timer A and timer B, as seen in Figure 3.13. The master timer, which has a clock frequency of 4.608 GHz, is enabled. Four different compare units, essentially counter interrupts, are set as shown in Figure 3.16. Compare unit values are controlled by two-byte registers. Every 1-bit change corresponds to a 217 *ps* change in timer variables. Therefore, changing these registers can control pulse set and reset times. On the other hand, clock configurations are adjusted to achieve maximum frequency for HRTIM. Therefore, the phase-locked loop (PLL) multiplier value is arranged to have maximum frequency according to the maximum achievable system frequency, which is 72 MHz, as shown in Figure 3.14.

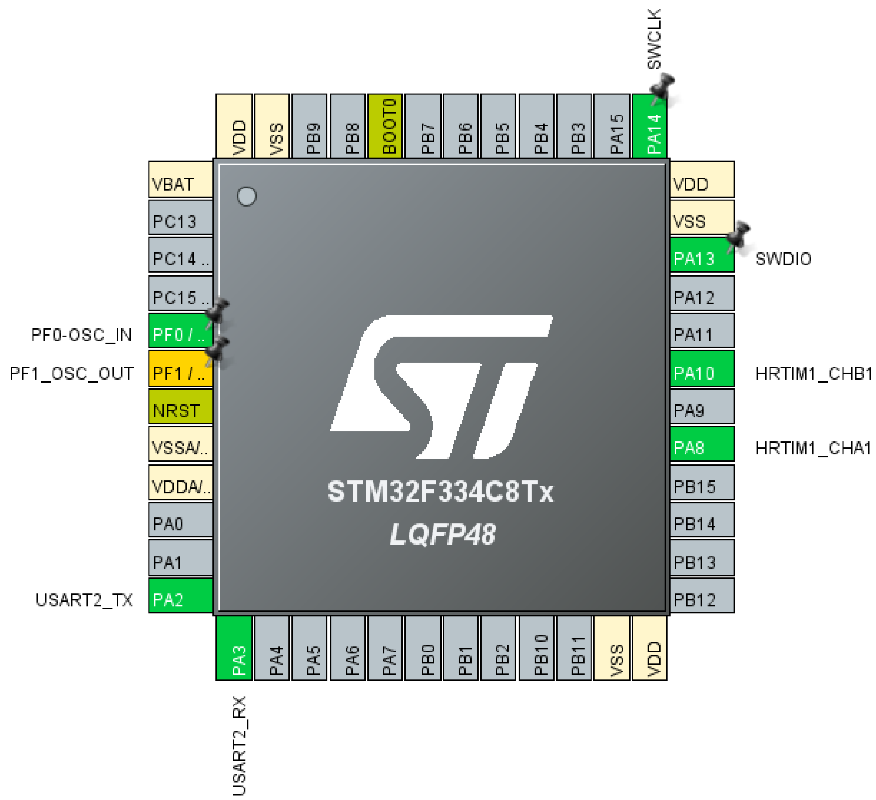

Figure 3.13. Microcontroller pinout configuration

Moreover, USART and HRTIM interrupts are activated, which is particularly necessary for HRTIM because since the master timer's frequency equals 4.608 GHz, the resulting PWM frequency could be a minimum of 70.348 kHz, as seen in Figure 3.15. Thus, an interrupt must be used to stop the PWM whenever one master clock cycle is completed.

When code is generated via STM32CubeIDE, all initialization functions and interrupts are defined automatically. In the HRTIM init function, timer compare values are replaced with variable names that are defined globally such that they can be changed by using communication. Therefore, a USART communication protocol is developed. Every message consists of four bytes. The first byte is divided into two parts: header and mode. The header is a constant value for every message and is used as a first filter to check if the received message is valid. The mode specifies the duty of the received message. There are six different modes:

1. 0x1 : Connect
2. 0x2 : Trigger pulse
3. 0x3 : Pulse 1 start
4. 0x4 : Pulse 1 stop
5. 0x5 : Pulse 2 start
6. 0x6 : Pulse 2 stop

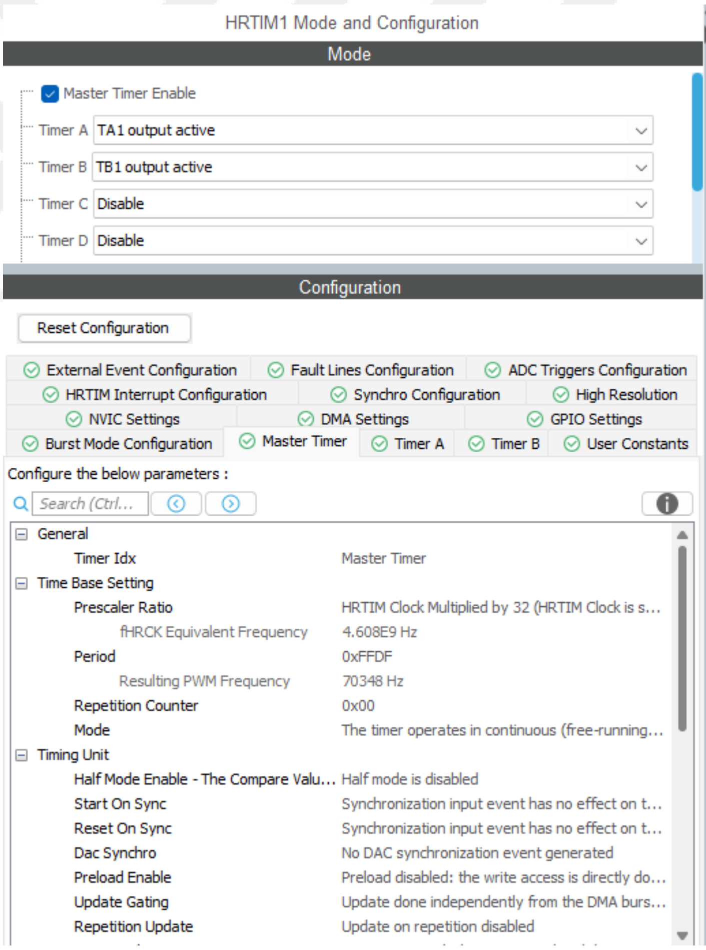

Figure 3.14. Master timer configuration

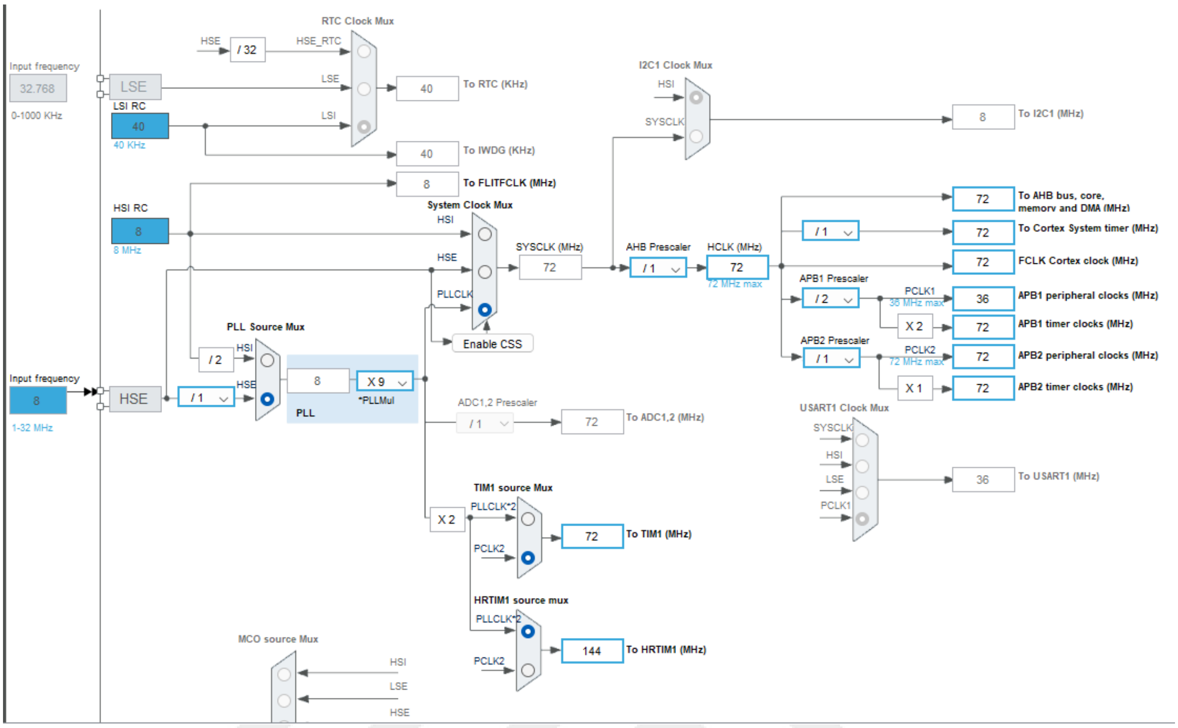

Figure 3.15. Microcontroller clock configuration

Interrupt Requests Sources Selection…	1
1st Interrupt Request Source	Timer repetition interrupt enable
Number of Master Timer Internal DMA…	0
⊟ Compare Unit 1	
Compare Unit 1 Configuration	Enable
Compare Value	0x0060
⊟ Compare Unit 2	
Compare Unit 2 Configuration	Enable
Compare Value	0x1260
⊟ Compare Unit 3	
Compare Unit 3 Configuration	Enable
Compare Value	0x3F19
⊟ Compare Unit 4	
Compare Unit 4 Configuration	Enable
Compare Value	0x5119
⊟ Burst DMA Controller	
Burst DMA Configuration	Disable

Figure 3.16. Compare unit settings of master timer

After the first byte, two bytes of the data part of the message package are used to modify the registers of the HRTIM in the MCU. By changing one bit of these two bytes of data, the time value of any mode can be changed by 217 ps. Finally, the message includes a check byte. The check byte is simply the sum of every bit in the message. Therefore, when a message is sent, the receiver can validate the incoming message by using the checksum method.

When a message is sent to MCU through a computer, USART interrupt triggers and receives a 4-byte message. The message is affirmed in the received data function when four bytes are received. If the incoming data is valid, the message is processed in the message handling function according to its mode. Then, the mode of the message is checked in the switch case. If there is a corresponding mode in the message's mode, data is processed, and an acknowledgement (ack) message is sent to the computer. Otherwise, not acknowledgement (nack) is sent.

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
union Packet
{
  uint32_t packetValue;
  struct
  {
    uint32_t header       :4;
    uint32_t mode         :4;
    uint32_t message      :16;
    uint32_t check        :8;
  }packetFields;
};

typedef union sendPacket {
  union Packet sendMessage;
    uint8_t buff[4];
}sendPacket_t;
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define HEADER    0xAU
#define ACK       0xFU
#define NACK      0x0U
/* USER CODE END PD */
```

Figure 3.17. MCU message structure for transmission and reception

```
/**
  * @brief This function handles HRTIM master timer global interrupt.
  */
void HRTIM1_Master_IRQHandler(void)
{
  /* USER CODE BEGIN HRTIM1_Master_IRQn 0 */

  /* USER CODE END HRTIM1_Master_IRQn 0 */
  HAL_HRTIM_IRQHandler(&hhrtim1,HRTIM_TIMERINDEX_MASTER);
  /* USER CODE BEGIN HRTIM1_Master_IRQn 1 */
  HAL_HRTIM_WaveformOutputStop(&hhrtim1, HRTIM_OUTPUT_TA1 | HRTIM_OUTPUT_TB1);
  HAL_HRTIM_WaveformCountStop(&hhrtim1, HRTIM_TIMERID_TIMER_A | HRTIM_TIMERID_TIMER_B | HRTIM_TIMERID_MASTER);
  /* USER CODE END HRTIM1_Master_IRQn 1 */
}
```

Figure 3.18. HRTIM master timer interrupt function

```c
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart == &huart2)
  {
    message_flag = 1;
    HAL_UART_Receive_IT(&huart2, received_msg, 4);
  }
}

uint32_t check_sum(uint32_t msg)
{
  uint32_t sum = 0;
  for(int8_t i = 0; i < 24; i++)
  {
    sum += (msg >> i) & 0x1;
  }
  return sum;
}

void received_data(void)
{
  for(int i = 0; i < 4; i++)
  {
    message |= received_msg[i] << i*8;
  }
  memset(received_msg, 0, sizeof(received_msg));

  union Packet packet;
  packet.packetValue = message;

  if (packet.packetFields.header == HEADER && check_sum(message) == packet.packetFields.check)
  {
    message_handling();
  }
  else
  {
    nack_send();
  }
}
```

Figure 3.19. Message receive handle functions including interrupt callback

```c
void ack_send(void)
{
  union Packet packet = {0};
  packet.packetValue = 0;
  packet.packetFields.header = HEADER;
  packet.packetFields.mode = ACK;
  packet.packetFields.message = 0x0000;
  packet.packetFields.check = check_sum(packet.packetValue);
  sendPacket_t sendPacket;
  sendPacket.sendMessage = packet;
  if (HAL_UART_Transmit(&huart2, &sendPacket.buff[0], 4, HAL_MAX_DELAY) != HAL_OK)
  {
    Error_Handler();
  }
}

void nack_send(void)
{
  union Packet packet = {0};
  packet.packetValue = 0;
  packet.packetFields.header = HEADER;
  packet.packetFields.mode = NACK;
  packet.packetFields.message = 0x0000;
  packet.packetFields.check = check_sum(packet.packetValue);
  sendPacket_t sendPacket;
  sendPacket.sendMessage = packet;

  if (HAL_UART_Transmit(&huart2, &sendPacket.buff[0], 4, HAL_MAX_DELAY) != HAL_OK)
  {
    Error_Handler();
  }
}
```

Figure 3.20. ACK and NACK message functions

```
void message_handling(void)
{
    union Packet packet;
    packet.packetValue = message;
    switch((uint8_t) packet.packetFields.mode)
    {
        case 1:
            ack_send();
            break;
        case 2:
            HAL_HRTIM_WaveformOutputStart(&hhrtim1, HRTIM_OUTPUT_TA1 | HRTIM_OUTPUT_TB1);
            HAL_HRTIM_WaveformCounterStart(&hhrtim1, HRTIM_TIMERID_TIMER_A |
                                                     HRTIM_TIMERID_TIMER_B | HRTIM_TIMERID_MASTER);
            ack_send();
              break;
        case 3:
            pulse1_start = packet.packetFields.message;
            HAL_HRTIM_SoftwareReset(&hhrtim1, HRTIM_TIMERID_MASTER);
            MX_HRTIM1_Init();
            __HAL_HRTIM_MASTER_ENABLE_IT(&hhrtim1, HRTIM_MASTER_IT_MREP);
            ack_send();
              break;
        case 4:
            pulse1_stop = packet.packetFields.message;
            HAL_HRTIM_SoftwareReset(&hhrtim1, HRTIM_TIMERID_MASTER);
            MX_HRTIM1_Init();
            __HAL_HRTIM_MASTER_ENABLE_IT(&hhrtim1, HRTIM_MASTER_IT_MREP);
            ack_send();
            break;
        case 5:
            pulse2_start = packet.packetFields.message;
            HAL_HRTIM_SoftwareReset(&hhrtim1, HRTIM_TIMERID_MASTER);
            MX_HRTIM1_Init();
            __HAL_HRTIM_MASTER_ENABLE_IT(&hhrtim1, HRTIM_MASTER_IT_MREP);
            ack_send();
            break;
        case 6:
            pulse2_stop = packet.packetFields.message;
            HAL_HRTIM_SoftwareReset(&hhrtim1, HRTIM_TIMERID_MASTER);
            MX_HRTIM1_Init();
            __HAL_HRTIM_MASTER_ENABLE_IT(&hhrtim1, HRTIM_MASTER_IT_MREP);
            ack_send();
            break;
        default:
            nack_send();
            break;
    }
    message = 0;
}
```

Figure 3.21. Message handling function

The message handling function processes the data according to the mode of the received message. For one pulse trigger, the HRTIM waveform and counter are started. After one period is completed for the main timer, another interrupt is activated for HRTIM, which stops the waveform and resets the counter for the main timer so that only one pulse in each HRTIM peripheral is triggered. If timer variables are changed with the received message, the HRTIM peripheral is reinitialized to rewrite registers accordingly.

3.3.2. GUI Software

A GUI is designed to control MCUs more efficiently and user-friendly, as shown in Figure 3.22. The GUI is based on the PyQt5 module and serial communication module. The same communication protocol is embedded in GUI software as in MCU. Four different classes are created for other purposes.

The main class is called window, inherited from QMainWindow in the PyQt5 module. The window class consists of application widgets, window layout, and all the other core functions used for communication. The second class is PlotCanvas, which is inherited from the Matloplib module. PlotCanvas is primarily used to plot pulse shapes when the user sets them. Additionally, it can show images. The third class is called TaskScheduler, which sends repeated pulses by assigning a thread. Finally, a combo box class inherited from PyQt5 is created.

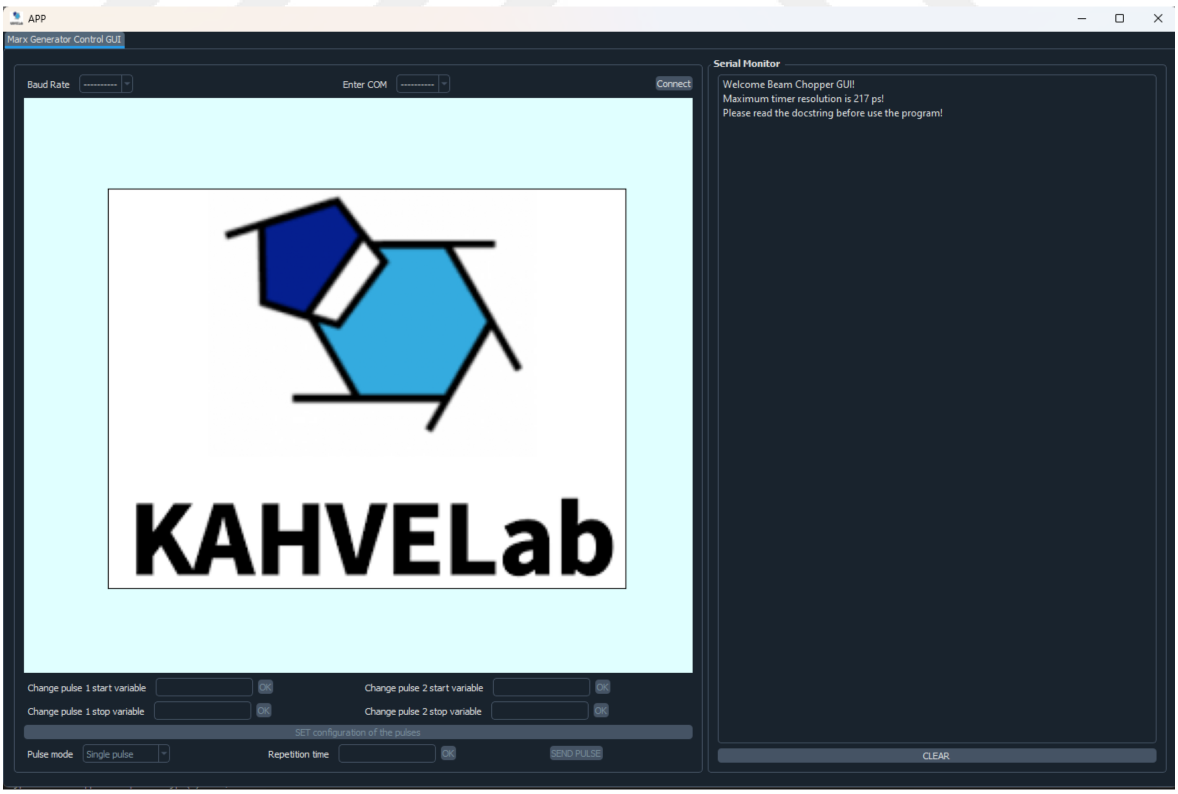

Figure 3.22. Graphical user interface window

The window layout is actually formed by a horizontal box layout, which is called the main layout. The main layout is divided into two pieces: the right and left layouts. The right layout consists of a serial monitor to monitor communication and a button to clear the serial monitor, while the left layout consists of six different layouts. The first layout is a horizontal box consisting of a baud rate combobox and its text label, a com port combo box and its text label, and a button to connect the MCU. The second layout is a vertical box, which only includes a plot canvas. Below it, there is another horizontal box which contains four text boxes, four text labels, and four buttons to change pulse parameters. Another horizontal box holds configured pulses when the user changes pulse parameters. At the end of the left layout, there is one more horizontal box to hold a combo box and its layout text; a text box, text label and a button; and finally, a button to send pulses.

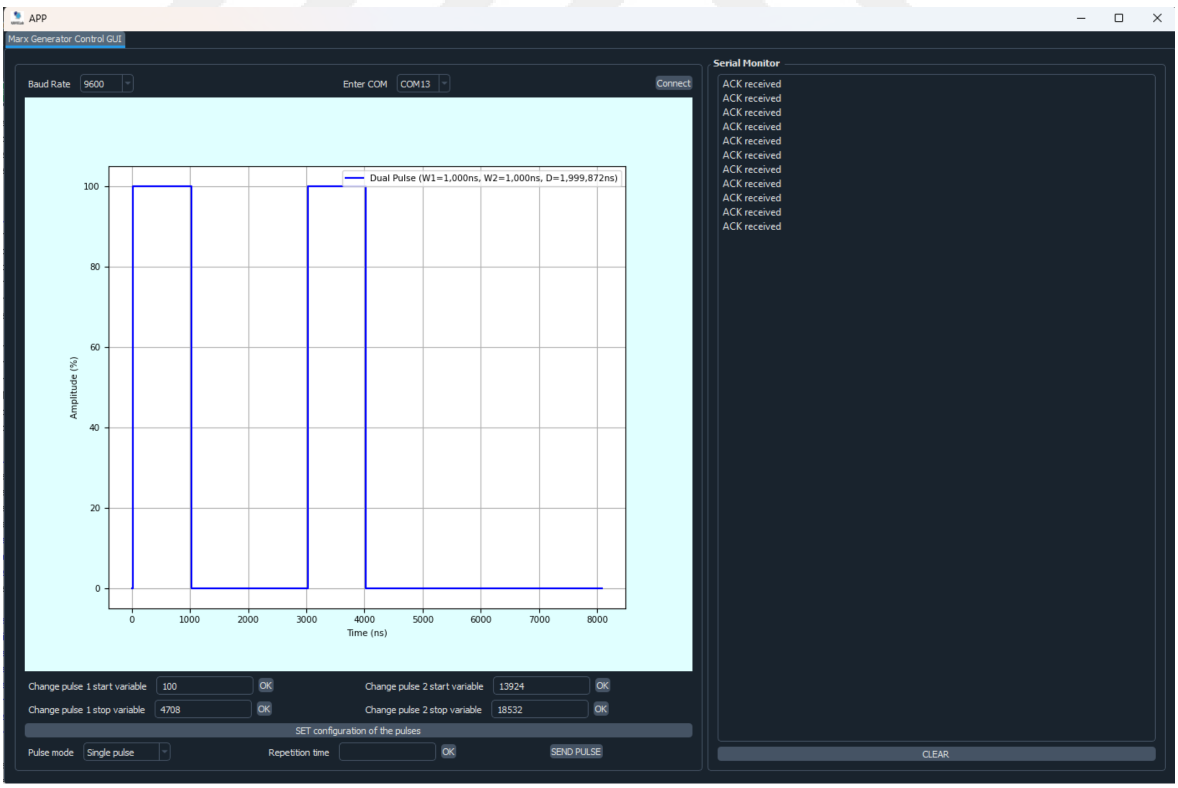

Figure 3.23. Graphical user interface window when configurations are set

The baud rate combo box contains every possible baud rate that can be used in serial communication systems. Before connecting, the user can set the baud rate according to the device. Moreover, the COM port must be specified. Therefore, a combo box shows all the devices connected to the computer. It also refreshes its content whenever it is closed and reopened. The plot function works whenever pulse settings are changed. The plot function plots two square waves, which the user specifies by pulse variables below the canvas. It is also synchronized with user input. Whenever a pulse configuration changes, the new plot is replaced by the old one. Furthermore, pulse type can be set using a pulse type combo box with two options: single pulse mode and repeated pulses. When the repeated pulses option is selected, the user should specify the repetition time of the pulse as well. The repetition pulse is restricted in the code, which allows a maximum of 10 Hz.

Before connection, all the buttons in the GUI are deactivated. When GUI is used to connect a device, buttons are activated, and it immediately sends a communication message with mode 0x1, and calls receive function for the response. GUI is programmed to receive only two messages, ack and nack. If it receives an ack or nack, it prints it to the serial monitor. The receive function has an argument to specify the maximum repetition number for repeated listening. This number is initialized by three. Therefore, three times, it listens and checks if the received message is valid or not. If the incoming message is corrupted, it prints to the serial monitor to inform the user. A send function is programmed to send messages. Every time a message is sent, the receiver function is also called to listen if the device understands the message. Moreover, whenever pulse configurations are changed, the user should click the button to send new configurations to the MCU.

```python
def message_packet(self, mode: bytes, message: bytes):

    if not (0x0 <= mode <= 0xF):
        info_box = QMessageBox.information(self, "WARNING!", "It is not a valid input!")
        return -1
    if not (0x0 <= message <= 0xFFFF):
        info_box = QMessageBox.information(self, "WARNING!", "It is not a valid input!")
        return -1
    packet_check = self.header | (mode << 4) | (message << 8)
    check = self.check_func(packet_check)
    message = packet_check | (check << 24)
    return message
```

Figure 3.24. Message packet creation function

```python
    def receive(self, max_retries: int = 3):
        retry_count = 0
        while retry_count < max_retries:

            received_msg = self.mcu.read(self.packet_size)
            if len(received_msg) != self.packet_size:
                retry_count += 1
                continue
            else:
                packet_value = int.from_bytes(received_msg, 'little')
                received_header = packet_value & 0xF
                received_mode = (packet_value >> 4) & 0xF

                if received_header != self.header:
                    retry_count += 1
                    continue
                if self.check_func(packet_value) != (packet_value >> 24) & 0xF:
                    retry_count += 1
                if received_mode == self.ack:
                    self.serial_monitor("ACK received")
                    retry_count = 3
                elif received_mode == self.nack:
                    self.serial_monitor("NACK received")
                    retry_count = 3
                else:
                    self.serial_monitor(f"Unknown message!")
                    retry_count += 1
                    continue

    def send(self, modee: bytes, msg: bytes):

        message = self.message_packet(mode = modee, message = msg)
        if message != -1:
            message_bytes = struct.pack('<I', message)
            self.mcu.reset_input_buffer()
            self.mcu.write(message_bytes)
            self.mcu.flush()
            time.sleep(0.01)
            self.receive()
```

Figure 3.25. Receive and sent functions

4. SIMULATION

After theoretical calculations, simulations of the different parts of the system are necessary to support the design and validate calculations. Initially, the beam chopper system is simulated using the CST Studio Suite. Following the CST Studio Suite simulations, a Python program is developed to simulate beam distribution and emittance after the beam passes through the chopper system. Moreover, validation of the electronic circuit design is also necessary before production. Therefore, the Marx generator circuit simulations are made using the LTspice simulation program.

4.1. Chopper System

Two different simulations of the chopper system are made using CST Studio and Python. The Python program is needed to observe beam parameters such as spatial distribution and emittance. Additionally, CST Studio cannot simulate the system dynamically. Therefore, a numerical simulation is required to visualize the system's working principle.

4.1.1. Simulation in the CST Studio Suite

The CST Studio Suite was used to create a detailed and trustworthy simulation of the concept. The beam line is designed using an electron gun, a solenoid, and two parallel plates. Simulation results showed that the beam angle becomes much higher when a theoretically calculated voltage is applied to the parallel plate. Hence, the beam hits a parallel plate because of the fringe effects.

In CST Studio Suite simulations, the first parallel plate deflects the electron beam downwards by applying 5600 V, creating a 15-degree angle in the z-direction. The second parallel plate is placed parallel to the bent bunch, pulling the bunch up by applying 5600 V so that the electron bunch is almost in the z-direction again. The rest of the alignment corrections can be done using steering magnets and solenoids.

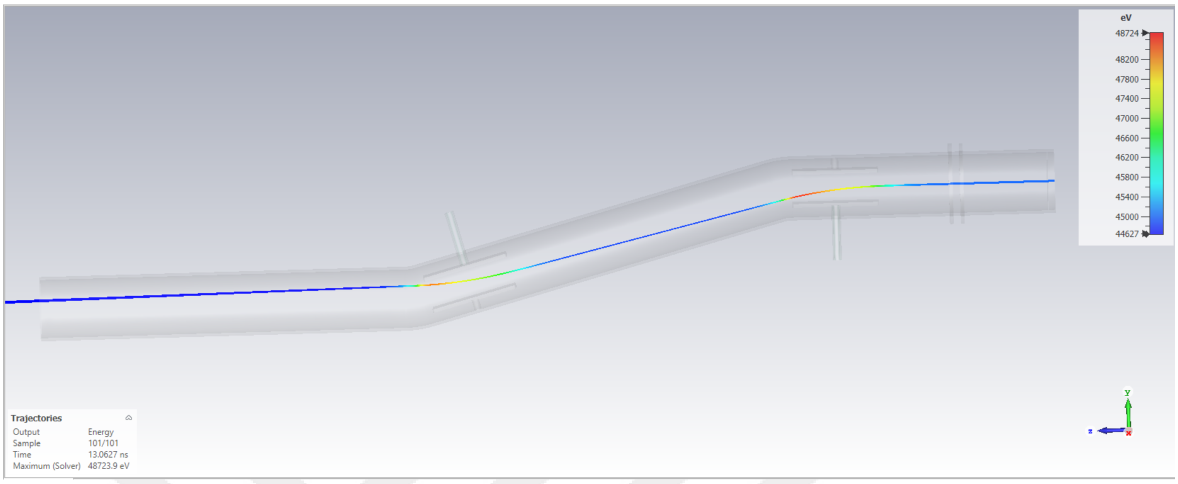

Figure 4.1. 45 keV electron bunch simulation in designed beam chopper

4.1.2. Numerical Simulation

4.1.2.1. Simulation Algorithm. The numerical simulation of the system is programmed using Python. Electric field coordinates and magnitudes are simulated and exported using the CST Studio Suite as a CSV file, as shown in Figure 4.3. The electrostatic solver is used for the same beam pipe design in the CST Studio Suite simulation. The resolution of the exported field in a 3D coordinate system is 0.5 mm. The exported electric file size is initially 6.3 GB, but there are also field values in which particles never pass through them. Therefore, before simulation, the CSV file is cleared by these fields using a Python script and beam pipe geometry, as shown in Figure 4.2. This process reduces the file size to 2.9 GB.

The time resolution of the simulation was initially chosen as 0.2 ns. For every 0.2 ns, the electric field of the beam pipe is exported. Simulation results showed that time resolution should be much lower. Therefore, the new time resolution is chosen as 0.01 ns, and the electric field is pulsed numerically instead of exporting every time stamp. The pulse algorithm works by defining a scale factor for every timestamp. Since it is a square pulse, a ramp is defined at the rising and falling edges. Moreover, since there should be a delay between pulses for parallel plates, the electric field matrix is divided into the middle of the parallel plates. For the first parallel plate, no delay was added. For the second parallel plate electric field, a delay was added such that the intersection of the two pulses corresponds to 1 ns bunch.

```python
def check_electricfield_boundaries(x, y, z):
    """
    Check if electric field is in beam pipe
    """
    # Region 1: -5000 < z < -168.15 in mm
    if -5 < z and z < -0.05:
        if abs(x) > 0.041 or abs(y) > 0.041:
            return False

    # Region 2: -50 < z < 381.85 in mm
    elif -0.05 < z and z < 0.5:
        if abs(x) > 0.041 or y > (-0.268*z+0.0266+0.01) or y < (-0.268*z-0.05397-0.01):
            return False

    # Region 3: 500 < z < 718 in mm
    elif 0.5 < z and z < 0.718:
        if abs(x) > 0.041 or y < -0.181 or y > -0.119:
            return False

    return True

df = pd.read_csv(file, delimiter = ",")
df.rename(columns={"#x [m]": "x [m]", "y [m]":"y [m]", "z [m]":"z [m]",
                   "Ex [V/m]":"Ex [V/m]", "Ey [V/m]":"Ey [V/m]", "Ez [V/m]":"Ez [V/m]"},
                   inplace=True)
df = df[df['Ey [V/m]'].notna()]
mask = df.apply(lambda row: check_electricfield_boundaries(
        row['x [m]'],
        row['y [m]'],
        row['z [m]']
    ), axis=1)
filtered_df = df[mask].copy()
del(df)
filtered_df.reset_index(drop=True, inplace=True)
filtered_df.to_csv(file, index = False, encoding='utf-8')
```

Figure 4.2. Python script to clear CSV file

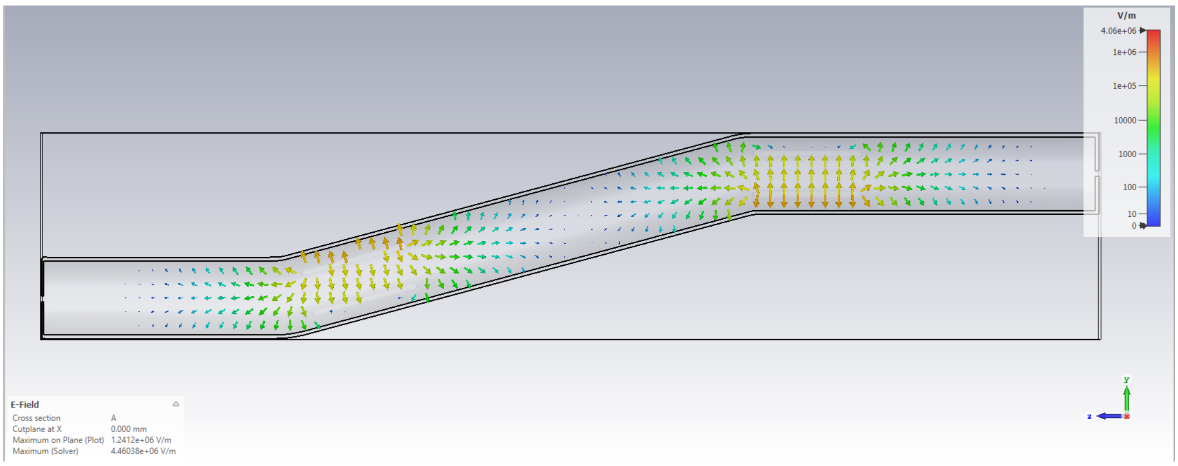

Figure 4.3. Electrostatic field simulation results in the CST Studio Suite

Python program algorithm starts by creating an electron beam with a random distribution in space and velocity. Initially, the beam's transverse velocity is chosen as zero in x and y. Next, the exported CSV file is read and modified as a NumPy matrix.

```python
def defineBeam(num_of_particles: int) -> np.ndarray:
    """
        Initialize the beam for given number of particles. It uses
        normal distribution to fill beam position and velocity values.
        Arguments:
        num_of_particles: Number of particles
    """
    c = 299792458                                                           # m/s the speed of light
    vz = 0.3941 * c                                                         # m/s where beta = 0.3941
    beam = np.zeros((num_of_particles, 9), dtype = np.float32)
    beam[:, 0] = np.random.normal(0, 0.0001, num_of_particles)               # x position
    beam[:, 1] = np.random.normal(0, 0.0001, num_of_particles)               # y position
    beam[:, 2] = np.random.normal(-1.5, 1, num_of_particles)                 # z position
    beam[:, 5] = np.random.normal(vz, vz*1e-6, num_of_particles)             # z velocity
    return beam

def read_electric_field_file() -> np.ndarray:
    """
        This function is used to read electric field CSV file that is
        exported from the CST Studio Suite. After it is read, it vectorizes
        the electric field by using Numpy array.
    """
    electric_field_file = pd.read_csv(path + "efield2.csv", delimiter=",")
    electric_field_file.dropna(subset=["x [m]", "y [m]", "z [m]", "Ex [V/m]", "Ey [V/m]", "Ez [V/m]"],inplace=True)
    electric_field = np.array([electric_field_file["x [m]"], electric_field_file["y [m]"], electric_field_file["z [m]"],
                               electric_field_file["Ex [V/m]"], electric_field_file["Ey [V/m]"],
                               electric_field_file["Ez [V/m]"]], dtype= np.float32).T
    return electric_field
```

Figure 4.4. Electron beam matrix and electric field matrix definitions

A function is developed to change electric field magnitudes according to time. However, the pulsing process is a bit tricky because there should be a time delay between the pulse in the first plate and the pulse in the second plate. Therefore, the pulse function separates electric fields from the middle of the beam pipe and multiplies for a specified value according to time and the specified delay. Moreover, parallelization and optimization are added by using the Numba module.

After new electric field magnitudes are calculated, a function finds electric field magnitudes in 3D for every particle's coordinates defined in the beam. The returned electric field magnitudes are directly in order with the corresponding particle in the beam matrix, which is actually necessary for calculating the beam dynamics using GPU. The k-d tree algorithm is used to optimize the search process from the SciPy module.

Ordered electric field magnitudes and beam matrices are sent into GPU memory by using CUDA functions from the Numba library. Furthermore, a GPU kernel is created with $blocksize = 256$, and the grid size is calculated by the shape of the beam matrix. Using @*cuda.jit* decorator, a GPU kernel is developed for the kick-drift-kick leapfrog algorithm. Formulas that are used in the GPU kernel are derived in section 1.4.2. After calculations in the kernel are completed, the beam matrix is copied back to the computer.

```python
@jit(nopython=True, parallel=True)
def pulser(e_field: np.ndarray, time: float) -> np.ndarray:
    """
    Pulser is used to convolve given electric field magnitudes
    according to timestamp.
    Arguments:
    1. e_field : Electric field matrix
    2. time : Timestamp of the simulation
    """
    rise_time = 2.5
    duration = 10
    fall_start = duration - rise_time
    new_e_field = np.zeros((len(e_field), 6))
    threshold = 131.184/1000
    delays = np.where(e_field[:, 2] < threshold, 0.0, 2.5 + 3.232)

    for i in range(len(e_field)):
        adjusted_time = time - delays[i]

        if adjusted_time < 0.0:
            new_e_field[i] = [e_field[i, 0], e_field[i, 1], e_field[i, 2], 0, 0, 0]

        elif adjusted_time < rise_time:
            scale = adjusted_time / rise_time
            new_e_field[i] = [e_field[i, 0], e_field[i, 1], e_field[i, 2],
                              scale * e_field[i, 3], scale * e_field[i, 4], scale * e_field[i, 5]]

        elif adjusted_time < fall_start:
            new_e_field[i] = e_field[i]

        elif adjusted_time < duration:
            scale = 1 - (adjusted_time - fall_start) / rise_time
            new_e_field[i] = [e_field[i, 0], e_field[i, 1], e_field[i, 2],
                              scale * e_field[i, 3], scale * e_field[i, 4], scale * e_field[i, 5]]

        else:
            new_e_field[i] = [e_field[i, 0], e_field[i, 1], e_field[i, 2], 0, 0, 0]
```

Figure 4.5. Electric field pulser function

In every iteration, particle velocities and positions are saved into a CSV file with a timestamp. At the end of the simulation, the CSV file size, which includes beam information, is 349 MB. The simulation time is approximately two and a half hours in Google Colab with Tesla T4 GPU.

```
def find_fields(particles: np.ndarray, e_field : np.ndarray,
                efield_tree: cKDTree) -> np.ndarray:
    """
    Function finds electric fields for particles using KD-tree algorithm.
    Arguments:
    1. particles: Beam matrix
    2. e_field: Electric field matrix
    3. efield_tree: cKDTree object
    """
    particle_positions = particles[:, [0, 1, 2]]
    _, efield_indices = efield_tree.query(particle_positions, k = 1, workers = -1)
    fields = np.zeros((particles.shape[0], 3))
    fields[:, 0] = e_field[efield_indices, 3]  # Ex
    fields[:, 1] = e_field[efield_indices, 4]  # Ey
    fields[:, 2] = e_field[efield_indices, 5]  # Ez

    return fields
```

Figure 4.6. Extracting corresponding electric fields for particles in the beam

```
@cuda.jit
def update_positions_gpu(particles: np.ndarray, fields: np.ndarray):
    """
    CUDA kernel for updating particle positions and velocities using
    kick-drift-kick leapfrog algorithm.
    Arguments:
    1.  particles: Particle's coordinates, velocities and accelerations
        particles[:,0:3] -> positions (x,y,z)
        particles[:,3:6] -> velocities (vx,vy,vz)
        particles[:,6:9] -> accelerations (ax,ay,az)
    2. fields: Electromagnetic field data (Ex,Ey,Ez)
    """
    idx = cuda.grid(1)
    qmRatio = -1.75882e+11     # C * kg^(-1)
    dT = 1e-11                 # 0.01 nanosecond time step
    halfDt = 0.5 * dT          # Half time step for leapfrog
    c = 299792458              # m/s

    if idx < particles.shape[0]:

        particles[idx, 3] += particles[idx, 6] * halfDt  # vx
        particles[idx, 4] += particles[idx, 7] * halfDt  # vy
        particles[idx, 5] += particles[idx, 8] * halfDt  # vz

        particles[idx, 0] += particles[idx, 3] * dT  # x
        particles[idx, 1] += particles[idx, 4] * dT  # y
        particles[idx, 2] += particles[idx, 5] * dT  # z

        beta = (particles[idx, 3]**2 + particles[idx, 4]**2 + particles[idx, 5]**2)**0.5 / c
        beta_x = particles[idx, 3] / c
        beta_y = particles[idx, 4] / c
        beta_z = particles[idx, 5] / c
        gamma = 1.0 / (1.0 - beta**2)**0.5

        Ex = fields[idx, 0]
        Ey = fields[idx, 1]
        Ez = fields[idx, 2]

        particles[idx, 6] = (qmRatio*Ex/gamma - gamma**2 * (beta_y*particles[idx, 7] +
                            beta_z*particles[idx, 8])*beta_x) / (1.0 + gamma**2 * beta_x**2)  # ax
        particles[idx, 7] = (qmRatio*Ey/gamma - gamma**2 * (beta_x*particles[idx, 6] +
                            beta_z*particles[idx, 8])*beta_y) / (1.0 + gamma**2 * beta_y**2)  # ay
        particles[idx, 8] = (qmRatio*Ez/gamma - gamma**2 * (beta_y*particles[idx, 7] +
                            beta_x*particles[idx, 6])*beta_z) / (1.0 + gamma**2 * beta_z**2)  # az

        particles[idx, 3] += particles[idx, 6] * halfDt  # vx
        particles[idx, 4] += particles[idx, 7] * halfDt  # vy
        particles[idx, 5] += particles[idx, 8] * halfDt  # vz
```

Figure 4.7. GPU kernel function

4.1.2.2. Visualization. The beam visualization program is programmed as a class. It consists of four figures: particle dynamics, x-y distribution, x-x' phase space, and phase space. Therefore, a canvas, which consists of four subfigures, is created. The big figure on the left shows how a continuous electron beam behaves in the chopper system. It also contains a time text and a text box to show how many particles are survived at that timestamp. The text box also includes the mean values of the living particles. On the right side of the canvas, three figures visualize synchronized bunch distribution in x-y, x-x' phase space and y-y' phase space from top to bottom, respectively.

Some of the particles in the beam are called synchronized particles. These particles are in a range in the z-axis, which was initially calculated. Then, a new column is added to the beam simulation result data, which allows the specification of particles. A function is used to mark synchronized particles in the initialization function of the class. That was necessary for the analysis of the beam parameters.

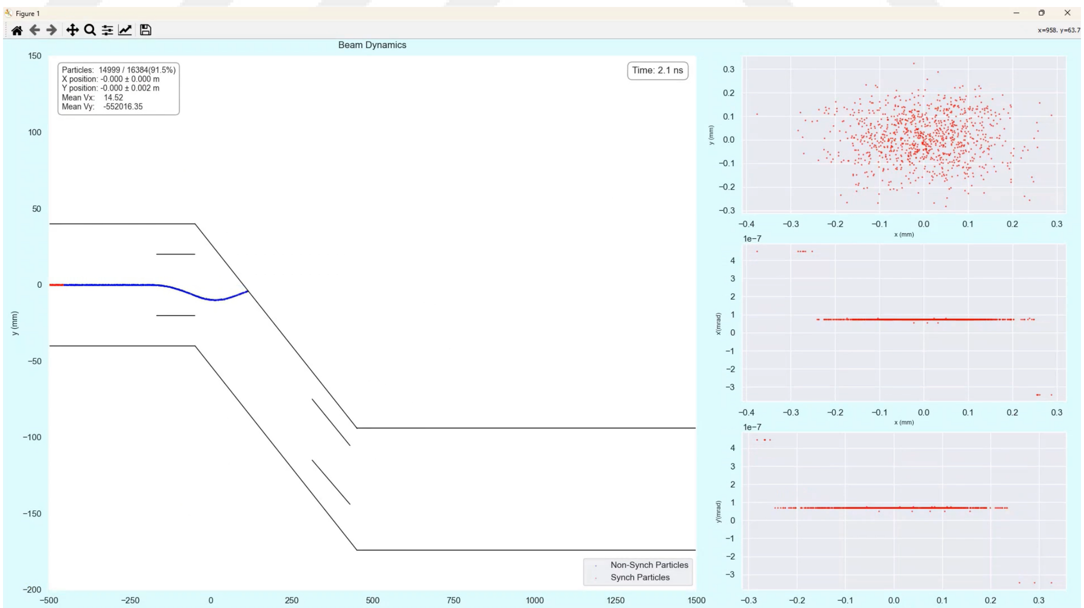

Figure 4.8. Beam parameters before chopper system. Initially, the beam has no transverse velocity.

The main iteration loop range is chosen according to unique timestamps from the simulation result file. Initially, an algorithm masks particles outside of the given geometry. Therefore, during visualization, if particles hit the walls of the accelerator pipe, they vanish for good. Next, unsynchronized and synchronized particles are separated. Synchronized particles are coloured red, and unsynchronized particles are coloured blue, labelled in the figure as well.

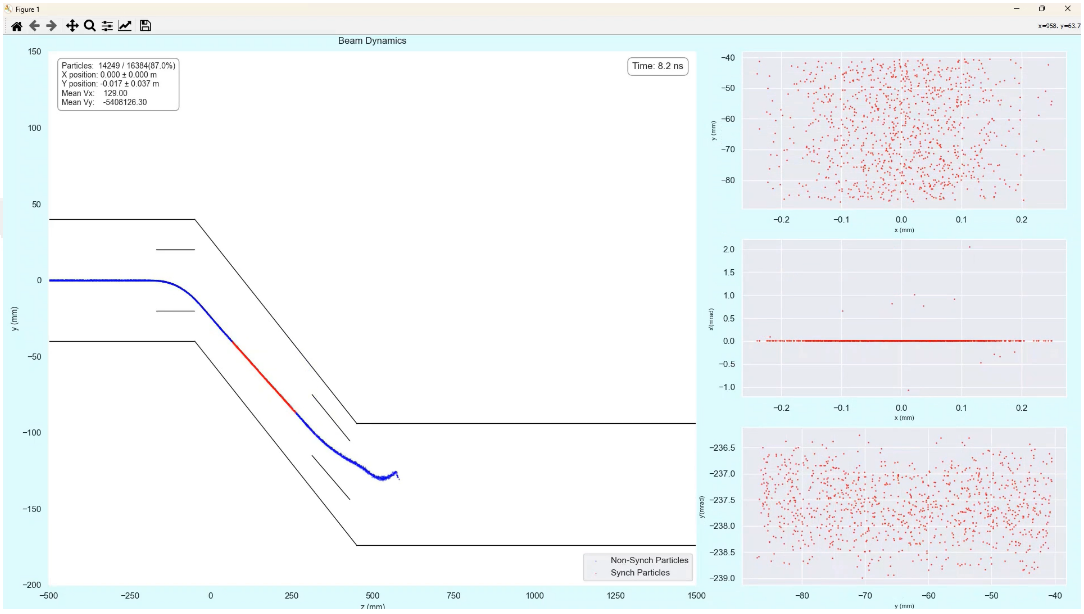

Figure 4.9. Beam parameters after first parallel plate.

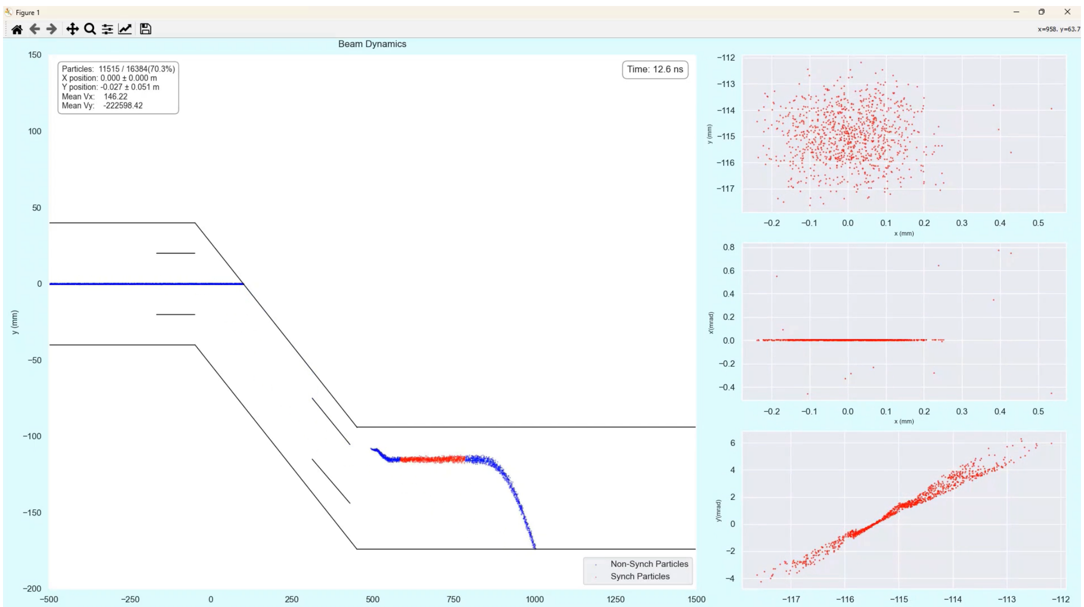

Figure 4.10. Beam parameters after the chopper system.

4.2. Solid-state Marx Generator Type-2

Using the LTspice, two different Marx generator circuits are simulated. First, a half-bridge circuit, including parasitic inductance, is simulated to validate theoretical calculations of the snubber and gate resistor. Second, combined Marx generator units are simulated to observe the total output wave.

4.2.1. Marx Generator Unit

LTspice simulation software was used to simulate circuits. A unit Marx Circuit, also known as a half-bridge circuit, is simulated to observe the behaviour of the output wave with parasitic inductances. The simulation results reveal that theoretical calculations are coherent with the simulation results.

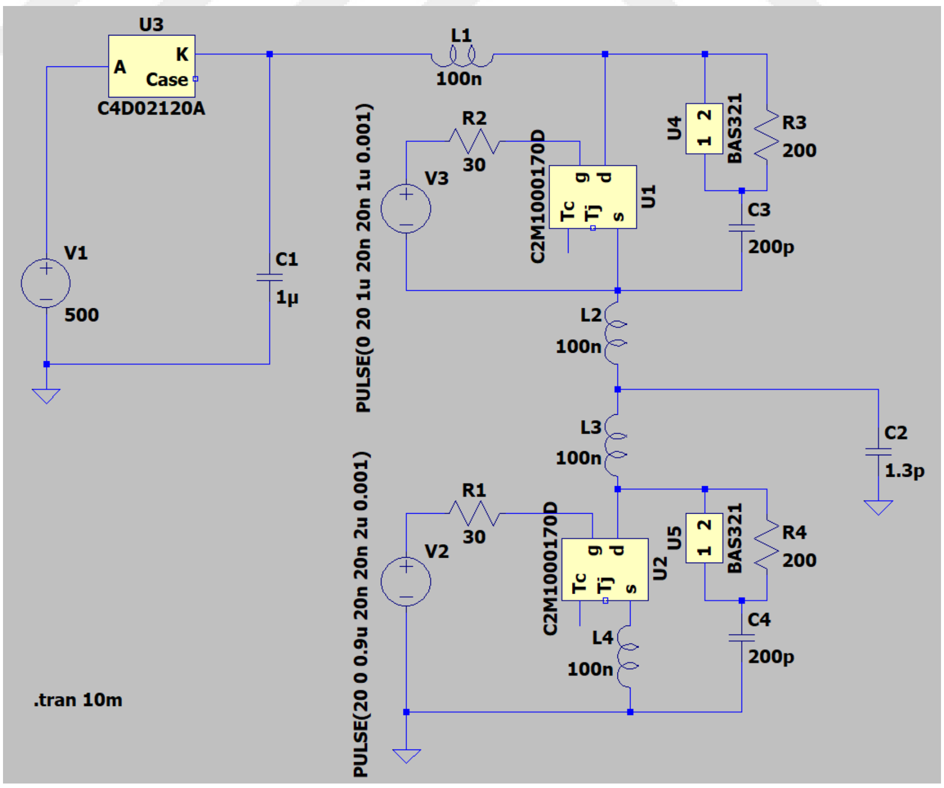

Figure 4.11. Unit Marx generator circuit with parasitic inductances for simulation

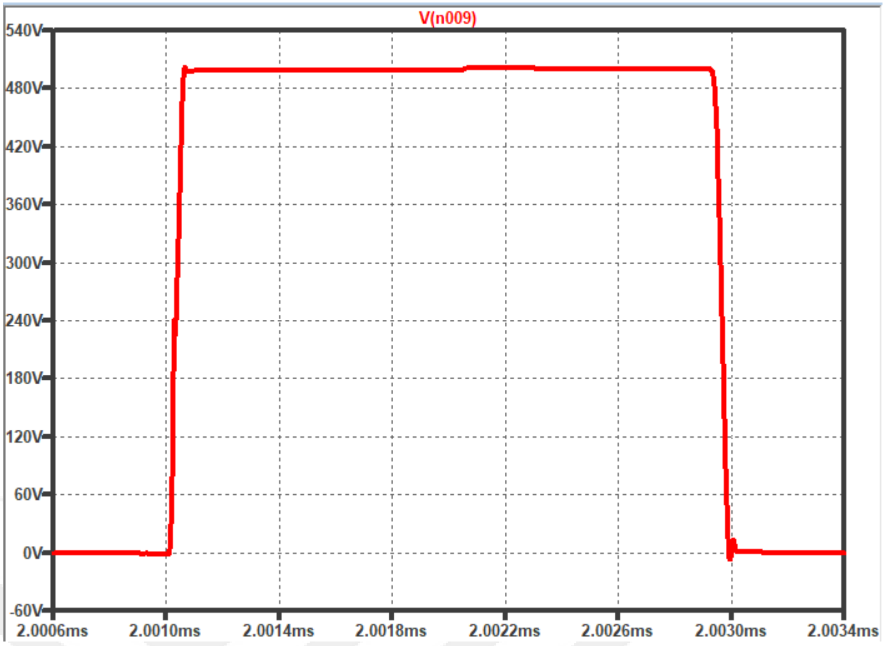

Figure 4.12. Unit Marx generator simulation result with parasitic inductances and snubber circuit

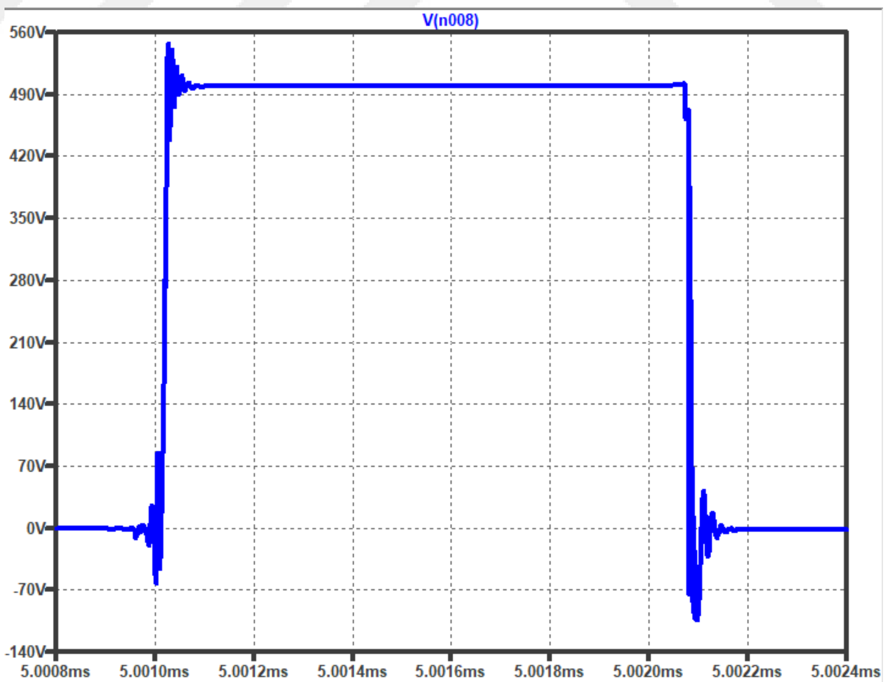

Figure 4.13. Unit Marx generator simulation result with parasitic inductances and without snubber circuit

4.2.2. Cascaded Marx Generator Units

In the Marx generator simulation, Marx generator units are cascaded. Diodes are omitted in the snubber circuit. Five Marx generator units are connected and fed by a voltage source of 1 kV. The output is connected to a 1.3 pF capacitor, as in the half-bridge circuit. The output waveform's magnitude is 5 kV, as expected.

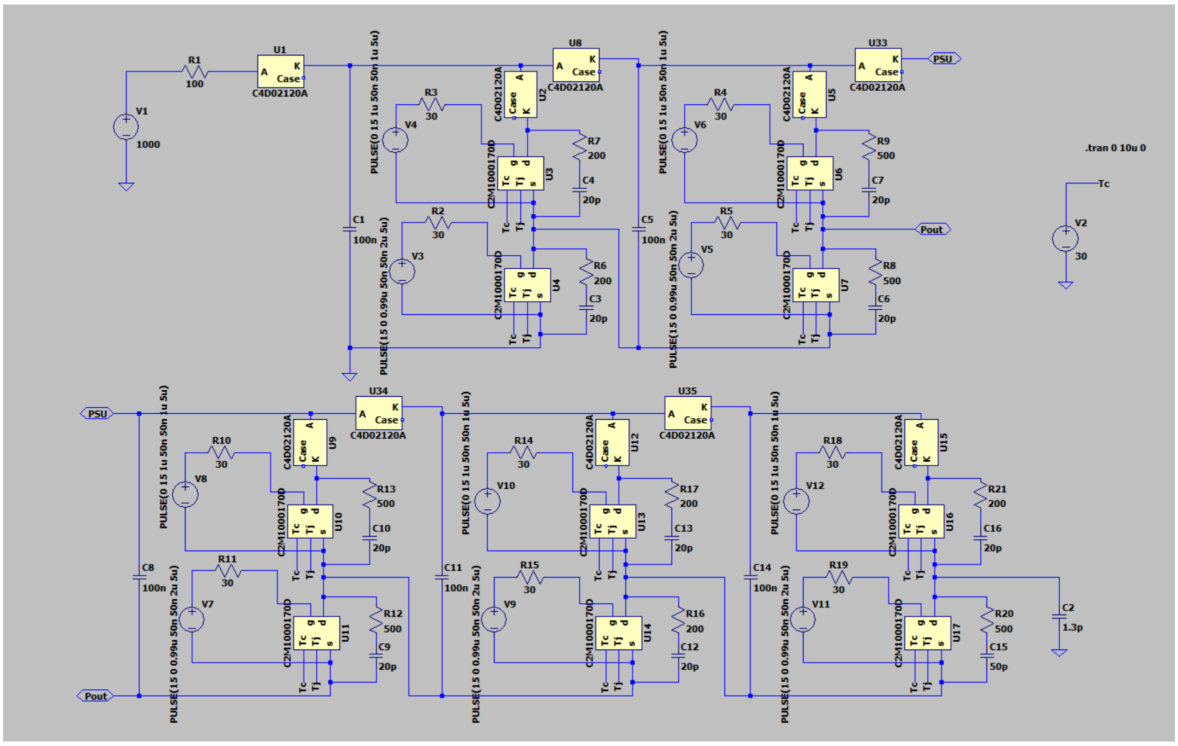

Figure 4.14. Simulation schematic of the Marx generator circuit with five stages

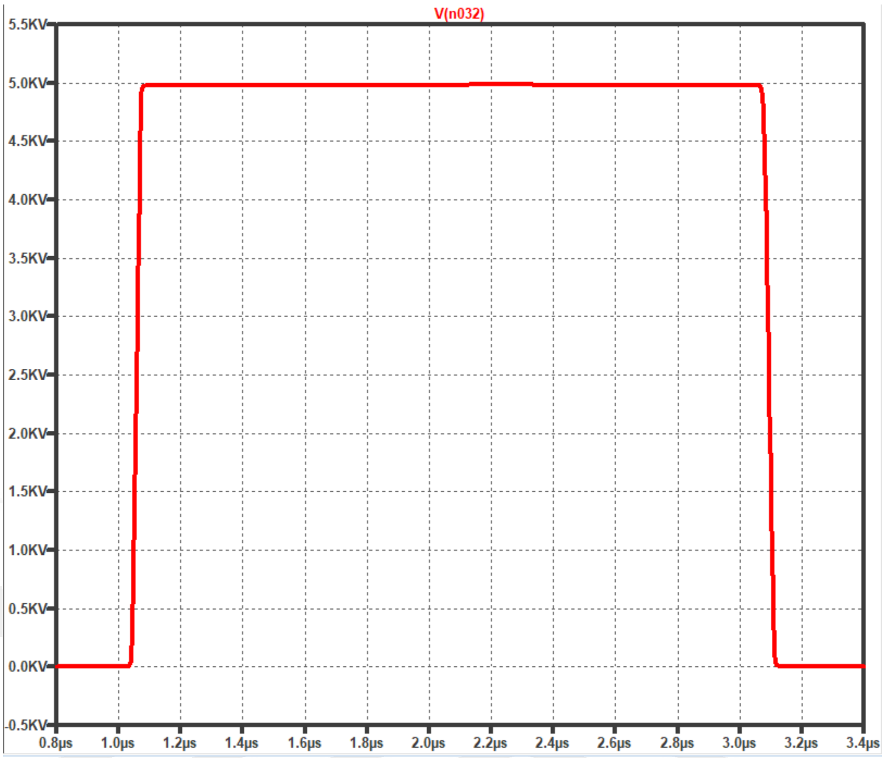

Figure 4.15. Output waveform of the Marx generator circuit with five stages

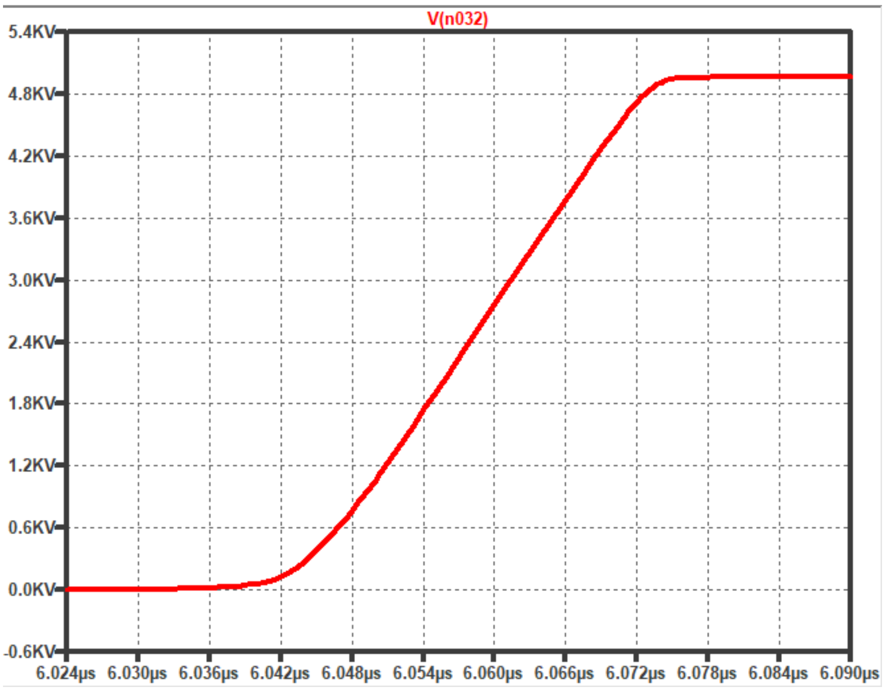

Figure 4.16. Rising edge of the output waveform

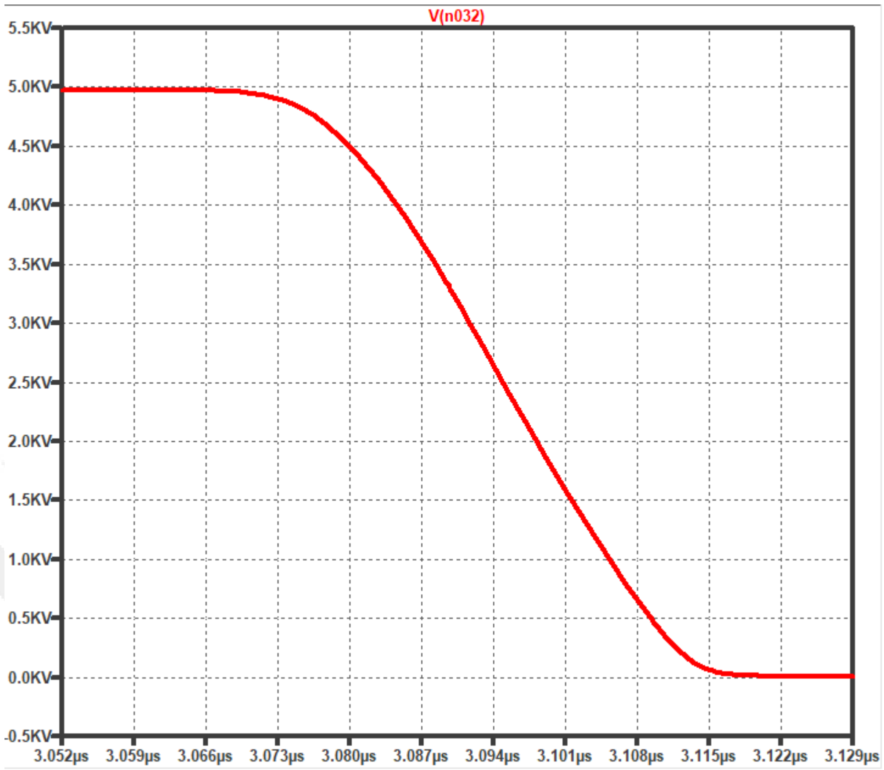

Figure 4.17. Falling edge of the output waveform

5. TESTS

Two different prototypes of the Marx generator units are produced. The first prototype is produced in the laboratory using a double-sided copper plate. Since the results of the first prototype were not promising for high voltages, it was decided to design a second prototype for production by a third-party PCB producer. Moreover, GUI and MCU programs are tested using a UART to USB converter and oscilloscope.

5.1. Marx Generator Prototype 1

Prototype 1 Marx generator unit PCBs are tested using different laboratory instruments such as an oscilloscope, digital multimeter, power supply and MCU kits. In addition to STM32 MCU, Arduino Uno is used to control driver ICs when Marx generator units are cascaded. This is because Arduino Uno is capable of providing more current, 50 mA, than STM32 MCU, 25 mA. Therefore, an interrupt circuit is built using a breadboard, which is used to control driver ICs with Arduino Uno. The interrupt circuit consists of an LED, a button and a resistor. Whenever the button is clicked, Arduino hardware interrupt rises to create 1 μs pulse. One power supply feeds the gate driver ICs and the other main capacitor.

The output of the circuit was connected to a voltage divider that consists of 1 $M\Omega$ and 10 $k\Omega$. These resistors were too inductive, and therefore, acquired output waveforms are affected adversely. This mistake was understood during the tests of prototype 2. In addition, the oscilloscope probe was problematic, which negatively affected the results. During tests, 1 PCB was burned because of a high voltage spark. Therefore, instead of producing one more PCB, it was decided to redesign PCBs and produce via third-party PCB producer.

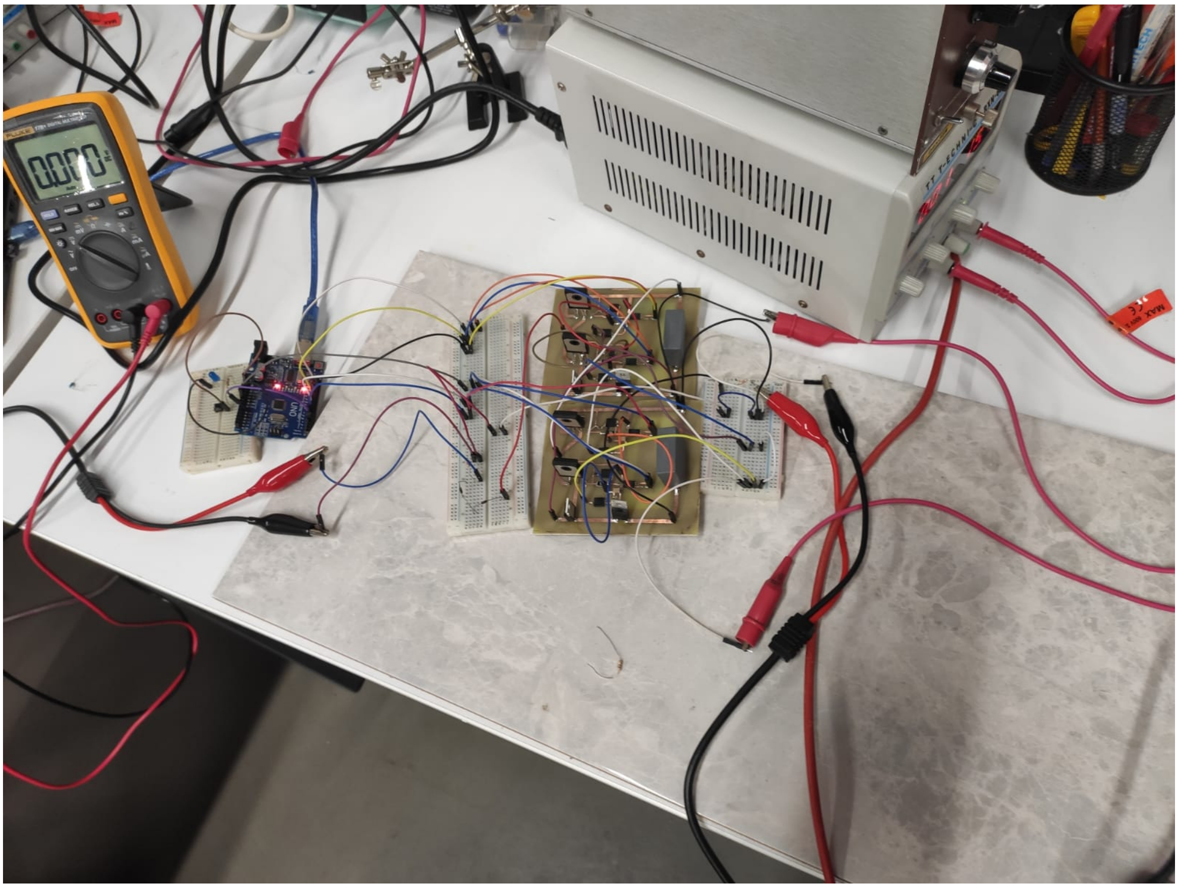

Figure 5.1. The test setup of the cascaded homemade PCBs

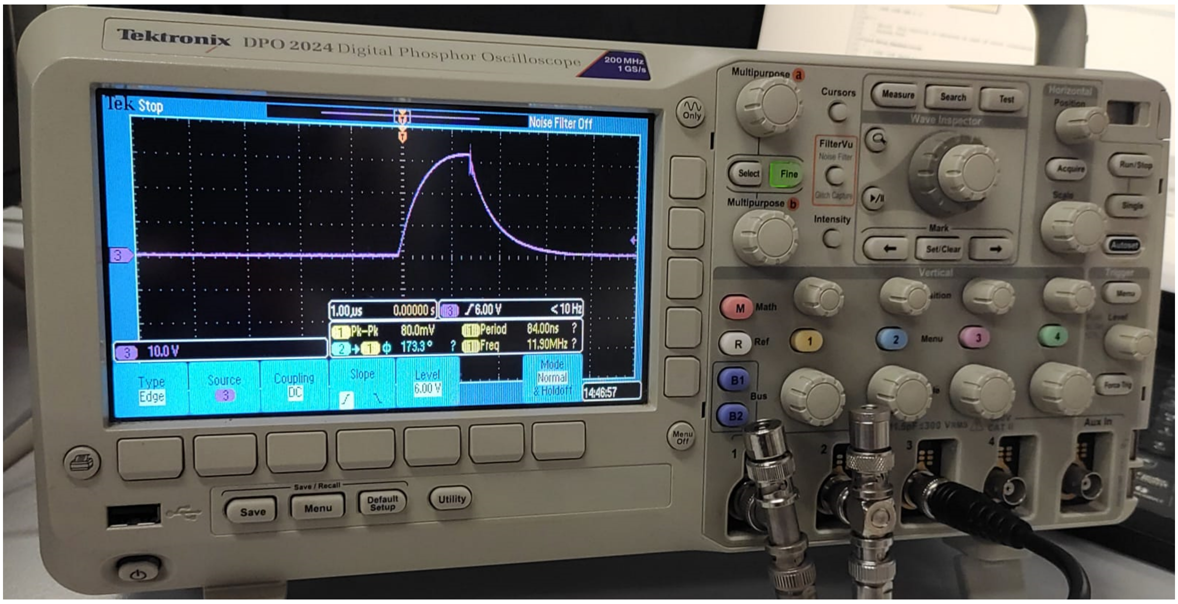

Figure 5.2. Homemade 1 PCB output result which is controlled by STM MCU

Figure 5.3. Homemade cascaded PCB output result which is controlled by Arduino Uno

5.2. Marx Generator Prototype 2

Although the first prototype results were not promising, the same design was decided upon. The second prototype results are tested by a new voltage probe. Voltage divider resistors are replaced with stone resistors, which have less parasitic inductance and capacitance. Prototype 2 PCBs were studied, and a lack of knowledge was revealed. First, the diodes between the power supply and gate driver ICs are the wrong choice because isolated power supplies were not used to supple driver ICs. Therefore, they are replaced with Schottky diodes. Moreover, a not gate is used in the control circuit because the Arduino clock is so slow for this application.

Output waveforms are significantly improved in prototype 2 by changing the snubber resistor and capacitor values. It was observed that when three PCBs are combined, the output waveform is disturbed and limited by 1.5 kV. This limit is guessed as the isolation limit of the power supply which feeds driver ICs. Another issue was inconsistency in the ringing problem, which changes via the voltage level of the high-voltage power supply. When the Marx generator is fed by 500 V, there are ringing problems. When it is fed by 700 V, the output waveform approaches a perfect square wave, which is not yet understood. Moreover, the square pulse's measured rise and fall time does not deviate by voltage level.

During tests, it was observed that when feed voltage exceeds 800 V, driver ICs are damaged. This can be avoided by using an isolated DC-DC converter, which supplies power to the driver ICs in the final design. As a result, gate driver IC power supplies should be isolated DC-DC converters. Otherwise, pulsed transformers should replace them. Note that output waveforms were measured via a voltage divider. Therefore, every 1 V corresponds to 100 V in the oscilloscope waveforms.

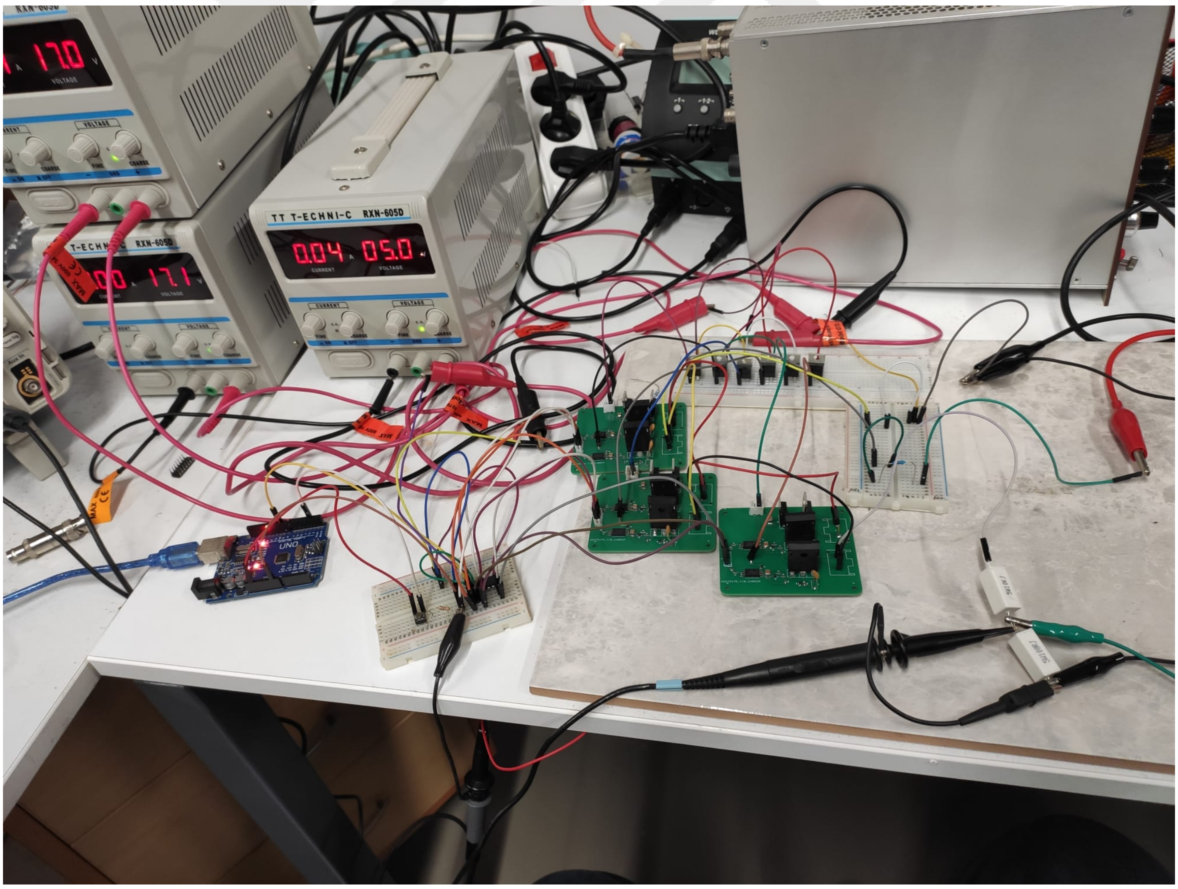

Figure 5.4. Prototype 2 test setup

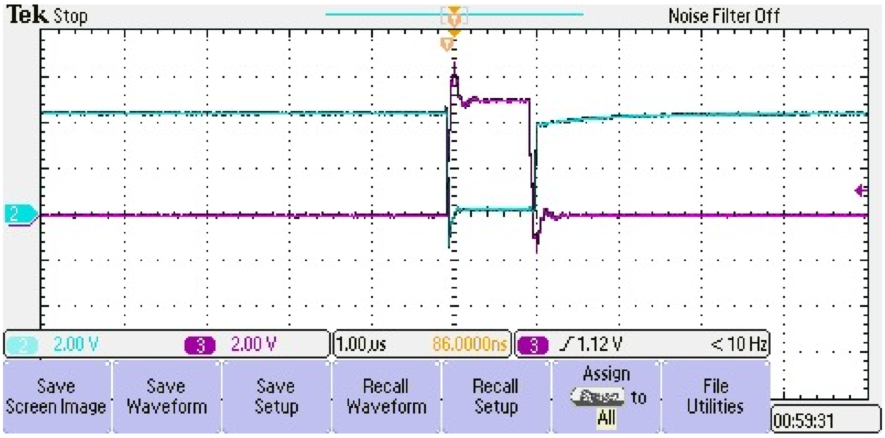

Figure 5.5. Control pulses of the Arduino Uno and not gate IC

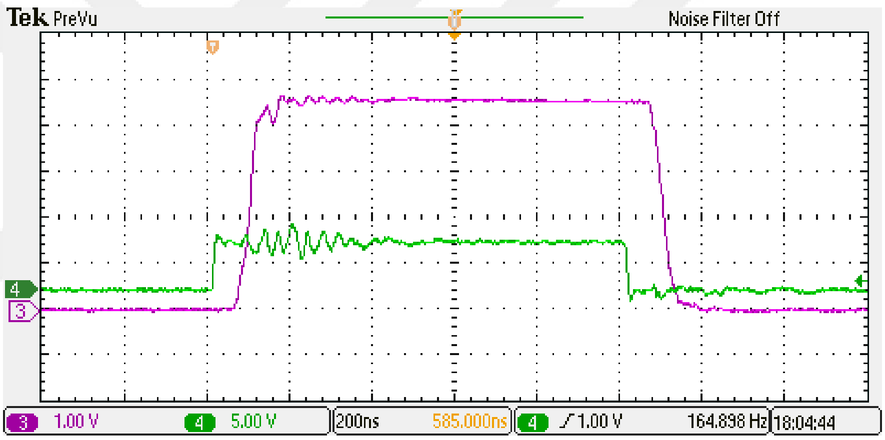

Figure 5.6. 500 V output waveform with 1 μs width by a PCB which is waveform 3. Waveform 4 is the Arduino trigger signal.

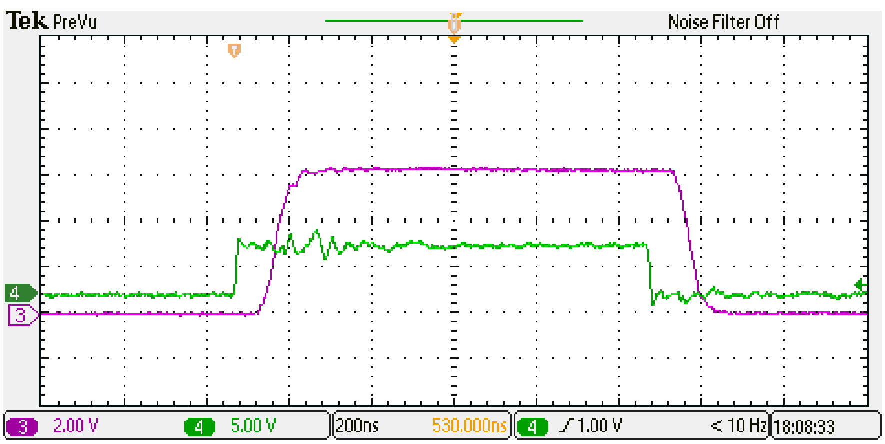

Figure 5.7. 700 V output waveform with 1 μs width by a PCB which is waveform 3. Waveform 4 is the Arduino trigger signal.

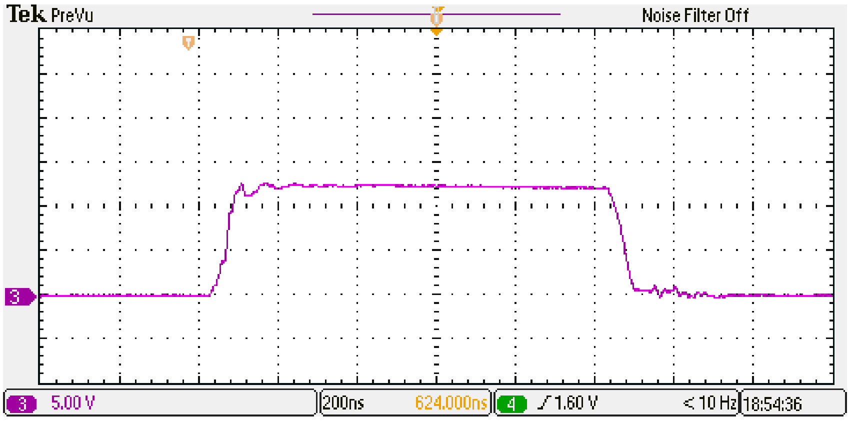

Figure 5.8. Output waveform of the cascaded two PCBs supplied by 700 V with 1 μs width.

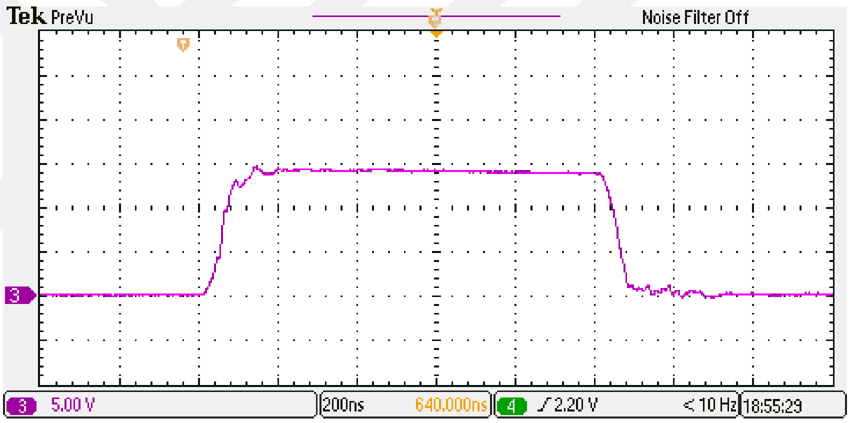

Figure 5.9. Output waveform of the cascaded two PCBs supplied by 800 V with 1 μs width

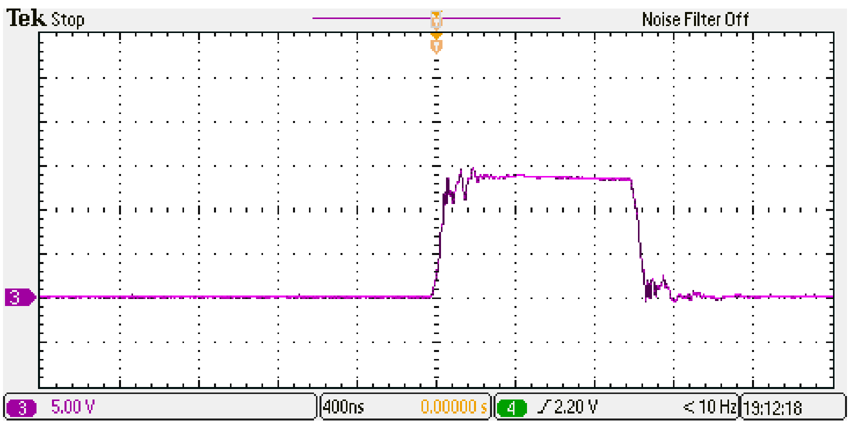

Figure 5.10. Output waveform of the cascaded three PCBs supplied by 700 V with 1 μs width

5.3. GUI and MCU Programs

Microcontroller and GUI programs are tested using a real-time communication and acquisition system. A USB to TTL converter is used to communicate with the MCU using the computer. A logic analyzer is used to observe HRTIM pulses. Moreover, a detailed pulse timing analysis should done by using a high-frequency oscilloscope.

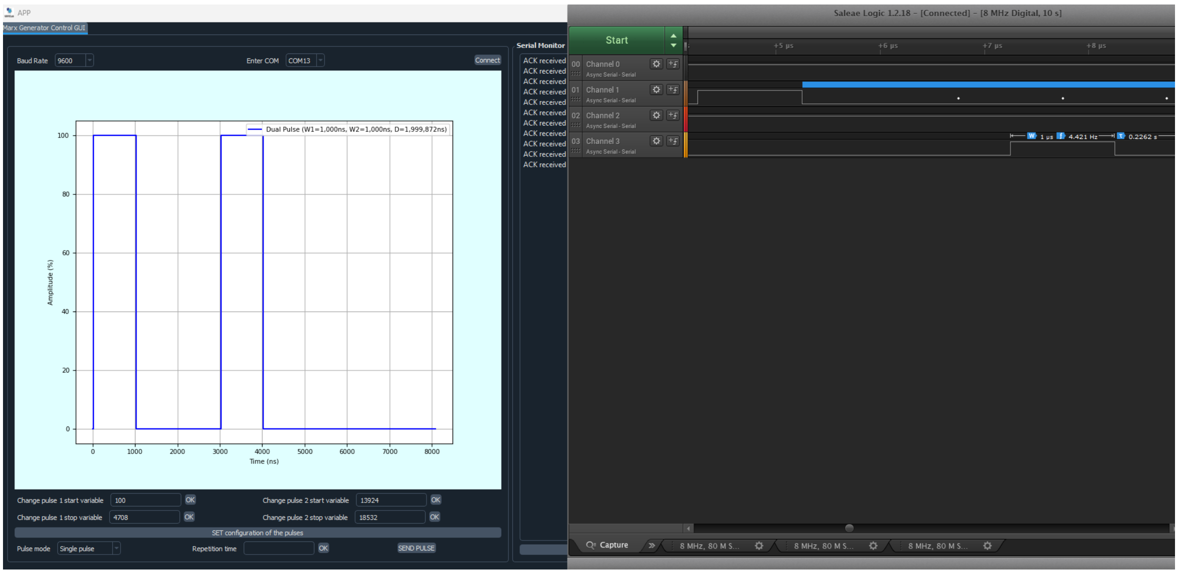

Figure 5.11. 1 μs and 1 μs pulses with 2 μs delay

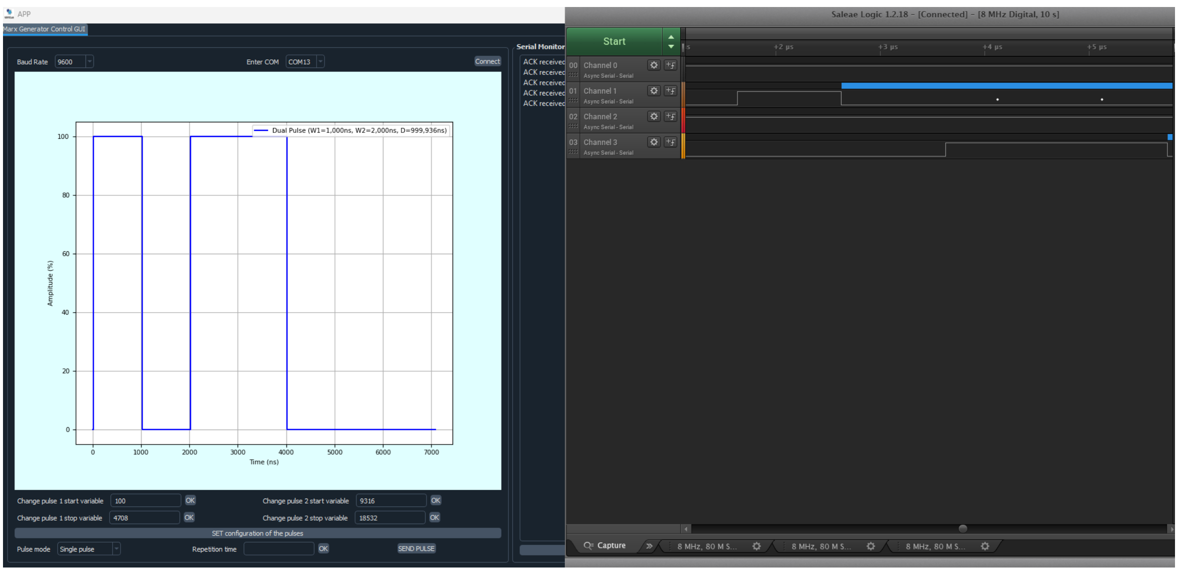

Figure 5.12. 1 μs and 2 μs pulses with 1 μs delay

6. CONCLUSION

Simulation results of the CST Studio Suite and numerical algorithms are coherent. The Marx generator prototypes showed that producing high-voltage square pulses is possible. The control mechanism of the system has a good enough time resolution for the 1 ns beam chopping process. Therefore, the chopper system design efforts exhibit that such a system can be realized.

6.1. Future Work

A new Marx generator PCB design is required, including isolated DC-DC converters and fibre optic receivers. A new layout with wider and shorter paths should be used in the high-frequency signal paths, which reduces the parasitic inductance of the tracks. Furthermore, a control circuit consisting of STM32F334C8xx, UART-USB converter and fibre optic transceivers should be designed and produced. Fibre optic control is necessary for more synchronized control.

The control algorithm can be further improved by using a new algorithm. There are four timers in the HRTIM peripheral. This allows complementary pulse signals to be generated for each Marx generator. One of the advantages of this algorithm would be avoiding digital non-gate buffers and reducing glitches caused by clock crossings. Another improvement can be made by writing register level C code for MCU instead of using HAL. Although HAL is an extremely useful tool, it has limited functionality for some purposes.

In addition, a non-uniform magnetic field or a static electric field should be added at the end of the system to stabilize beam expansion in the y-axis. Finally, chopper plates and beam pipes should be fabricated to observe the system's actual behaviour.

REFERENCES

1. Pottier, J., *A New Type of Electron Accelerator: The Rhodotron*, Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms, Vol. 40-41, pp. 943-945, 1989.

2. Jongen, Y., Abs, M., Delvigne, T., Herer, A., Capdevila, J.M., Genin, F., and Nguyen, A., *Rhodotron Accelerators for Industrial Electron-Beam Processing: A Progress Report*, Nuclear Instruments and Methods in Physics Research, Vol. 113, No. 1, pp. 523-528, 1996.

3. Er, M.F., *Design and Simulation Software for Rhodotron Type Electron Accelerators*, Master's Thesis, Graduate Program in Physics, Boğaziçi University, Istanbul, 2023.

4. Caspers, F., *Review of Fast Beam Chopping*, Proceedings of Linear Accelerator Conference, pp. 168-172, Lübeck, 2004.

5. Wiesner, C., Chau, L.P., Dinter, H., Droba, M., Joshi, N., Meusel, O., Mueller, I., and Ratzinger, U., *The Development of a Fast Beam Chopper for Next Generation High Power Proton Drivers*, Physical Review Special Topics - Accelerators and Beams, Vol. 15, No. 3, pp. 031302-031310, 2012.

6. Clarke-Gayther, M.A., *E×B Chopper System for High Intensity Proton Beams*, Nuclear Instruments and Methods in Physics Research Section A, Vol. 590, pp. 178-184 (2008).

7. Alekseev, V.I., Eliseev, A.N., Ivashchuk, O.O., Kishin, I.A., Kubankin, A.S., Oleinik, A.N., Sotnikova, V.S., Chepurnov, A.S., Grigoriev, Y.V., and Shchagin, A.V., *Pyroelectric Deflector of Relativistic Electron Beam*, Chinese Journal of Physics, Vol. 77, pp. 2298-2306, 2022.

8. Martini, M., *An Introduction to Transverse Beam Dynamics in Accelerators*, CERN/PS 96-11 (PA), 1996.

9. Shao, T., and Zhang, C., *Pulsed Discharge Plasmas Characteristics and Applications*, Springer Series in Plasma Science and Technology, Berlin, 2019.

10. Jiang, W., Diao, W., and Wang, X., *Marx Generator Using Power MOSFETs*, Proceedings of IEEE Pulsed Power Conference, pp. 408-410, Washington DC, 2009.

11. Redondo, L.M., Kandratsyeu, A., Barnes, M.J., Calatroni, S., and Wuensch, W., *Solid-State Marx Generator for the Compact Linear Collider Breakdown Studies*, IEEE International Power Modulator and High Voltage Conference, pp. 187-192, 2016.

12. Yao, J., *Working Principle and Characteristic Analysis of SiC MOSFET*, Journal of Physics: Conference Series, Vol. 2435, pp. 012022, 2023.

13. Balogh, L., *Fundamentals of MOSFET and IGBT Gate Driver Circuits*, Texas Instruments Application Report SLUA618A, 2017.

14. Güven, Y., Coşgun, E., Kocaoğlu, S., Gezici, H., and Yilmazlar, E., *Understanding the Concept of Microcontroller Based Systems to Choose the Best Hardware for Applications*, International Journal of Engineering and Science, Vol. 6, No. 9, pp. 38-44, 2017.

15. *HRTIM Cookbook*, STMicroelectronics Application Note AN4539, 2015.

16. Kassakian, J.G., Perreault, D.J., Verghese, G.C., and Schlecht, M.F., *Principles of Power Electronics*, Second Edition, Cambridge University Press, Cambridge, 2022.

17. Begue, M., *External Gate Resistor Design Guide for Gate Drivers*, Texas Instruments Technical Notes, 2021.

APPENDIX A: FIRST SIMULATION PROGRAM

The first simulation program and its algorithm are shared here.

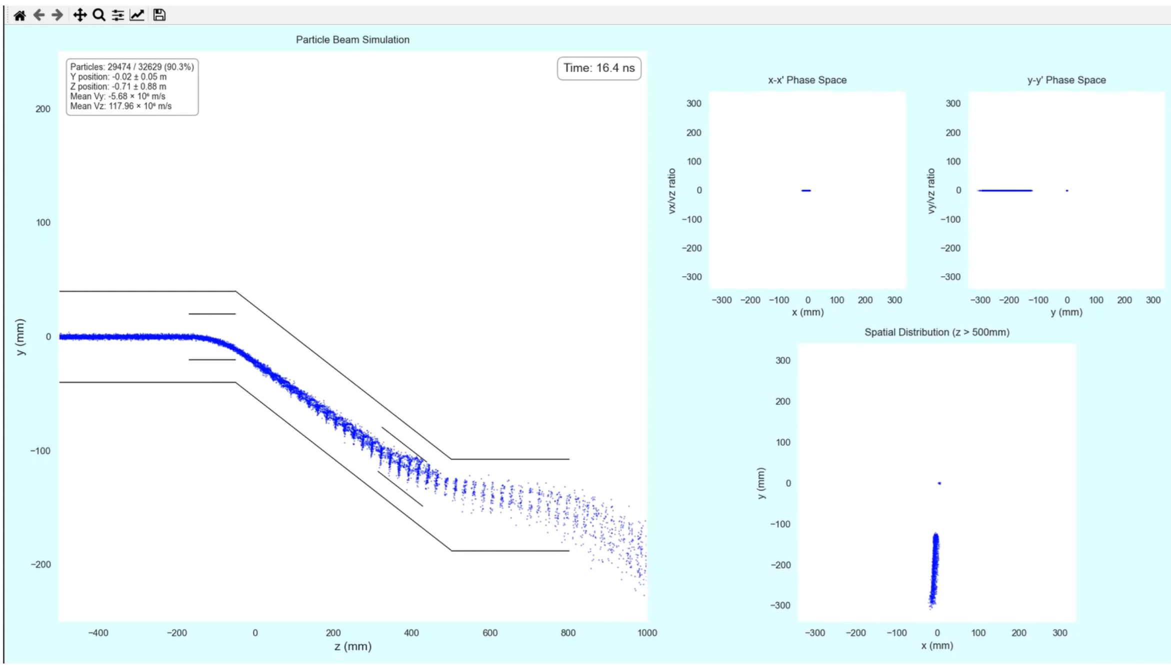

Figure 6.1. First simulation result

```python
@cuda.jit
def update_positions_gpu(particles, fields):
    """
    CUDA kernel for updating particle positions and velocities
    particles: Particle's coordinates, velocities and accelerations
    field_data: DataFrame containing field data
    """
    idx = cuda.grid(1)
    qmRatio = -1.75882e+11      # C * kg^(-1)
    dT = 2e-10                  # 0.2 nanosecond time step
    c = 299792458               # m/s
    if idx < particles.shape[0]:

        beta = (particles[idx, 1]**2 + particles[idx, 3]**2 + particles[idx, 5]**2)**0.5 / c
        beta_x = particles[idx, 1] / c
        beta_y = particles[idx, 3] / c
        beta_z = particles[idx, 5] / c

        gamma = 1.0 / (1.0 - beta**2)**0.5

        x_field = fields[idx, 0] + particles[idx, 3] * fields[idx, 5] - particles[idx, 5] * fields[idx, 4] # Ex + vy*Bz
        y_field = fields[idx, 1] + particles[idx, 5] * fields[idx, 3] - particles[idx, 1] * fields[idx, 5] # Ey + vz*Bx
        z_field = fields[idx, 2] + particles[idx, 1] * fields[idx, 4] - particles[idx, 3] * fields[idx, 3] # Ez + vx*By

        particles[idx, 6] = (qmRatio*x_field/gamma - gamma**2 * (beta_y*particles[idx, 7] +
                             beta_z*particles[idx, 8])*beta_x) / (1.0 + gamma**2 * beta_x**2)  # ax
        particles[idx, 7] = (qmRatio*y_field/gamma - gamma**2 * (beta_x*particles[idx, 6] +
                             beta_z*particles[idx, 8])*beta_y) / (1.0 + gamma**2 * beta_y**2)  # ay
        particles[idx, 8] = (qmRatio*z_field/gamma - gamma**2 * (beta_y*particles[idx, 7] +
                             beta_x*particles[idx, 6])*beta_z) / (1.0 + gamma**2 * beta_z**2)  # az

        particles[idx, 0] += particles[idx, 1] * dT + 0.5 * particles[idx, 6] * dT**2  # x
        particles[idx, 2] += particles[idx, 3] * dT + 0.5 * particles[idx, 7] * dT**2  # y
        particles[idx, 4] += particles[idx, 5] * dT + 0.5 * particles[idx, 8] * dT**2  # z

        particles[idx, 1] += particles[idx, 6] * dT  # vx
        particles[idx, 3] += particles[idx, 7] * dT  # vy
        particles[idx, 5] += particles[idx, 8] * dT  # vz
```

Figure 6.2. GPU kernel including electric and magnetic fields

```python
def find_fields(particles, e_field, b_field):
    """
    Function finds electric fields for particles using KD-tree algorithm.
    particles: Particle's coordinates, velocities and accelerations
    field_data: DataFrame containing field data
    """
    e_field_positions = np.column_stack([
        e_field["x [m]"].values,
        e_field["y [m]"].values,
        e_field["z [m]"].values
    ])

    b_field_positions = np.column_stack([
        b_field["x [m]"].values,
        b_field["y [m]"].values,
        b_field["z [m]"].values
    ])

    efield_tree = cKDTree(e_field_positions)
    bfield_tree = cKDTree(b_field_positions)
    particle_positions = particles[:, [0, 2, 4]]

    _, efield_indices = efield_tree.query(particle_positions, k=1)
    _, bfield_indices = bfield_tree.query(particle_positions, k=1)

    fields = np.zeros((particles.shape[0], 6))

    fields[:, 0] = e_field["x [V/m]"].values[efield_indices]  # Ex
    fields[:, 1] = e_field["y [V/m]"].values[efield_indices]  # Ey
    fields[:, 2] = e_field["z [V/m]"].values[efield_indices]  # Ez
    fields[:, 3] = b_field["x [V.s/m^2]"].values[bfield_indices]  # Bx
    fields[:, 4] = b_field["y [V.s/m^2]"].values[bfield_indices]  # By
    fields[:, 5] = b_field["z [V.s/m^2]"].values[bfield_indices]  # Bz

    return fields

def gpu_particle_iteration(beam_array, e_field, b_field):
    """
        Main function to send data to the GPU and launch the GPU kernels.
        beam_array: Particle's coordinates, velocities and accelerations
        field_file: Electric field list for particles
    """
    # Prepare data for GPU
    particles_gpu = cuda.to_device(beam_array)
    fields = find_fields(beam_array, e_field, b_field)
    fields_gpu = cuda.to_device(fields)

    # Configure CUDA grid
    threadsperblock = 512
    blockspergrid = (beam_array.shape[0] + (threadsperblock - 1)) // threadsperblock

    # Launch kernel
    update_positions_gpu[blockspergrid, threadsperblock](particles_gpu, fields_gpu)

    # Copy results back to CPU
    return particles_gpu.copy_to_host()
```

Figure 6.3. Electromagnetic search function and GPU kernel call function

```python
def save_data_batch(time, beam):
    """
        Function saves given data together with time stamp
        time: Time value of the particles
        beam: Particle's coordinates, velocities and accelerations
    """
    rows = []

    for particle in beam:
        rows.append({
            "time": time,
            "x": particle[0],
            "vx": particle[1],
            "y": particle[2],
            "vy": particle[3],
            "z": particle[4],
            "vz": particle[5],
        })

    with open(path + "\\results.csv", "a", newline='') as csv_file:
        writer = csv.DictWriter(csv_file, fieldnames=features)
        writer.writerows(rows)

def defineBeam(num_of_particles):
    """
        Initialize the beam for given number of particles
        num_of_particles: Number of particles
    """
    beam = np.zeros((num_of_particles, 9), dtype = np.float64)
    beam[:, 0] = np.random.normal(0, 0.001, num_of_particles)      # x position
    beam[:, 2] = np.random.normal(0, 0.001, num_of_particles)      # y position
    beam[:, 4] = np.random.normal(-2.5, 1, num_of_particles)       # z position
    beam[:, 1] = np.random.normal(vz*1e-4, vz*1e-8, num_of_particles)   # x velocity
    beam[:, 3] = np.random.normal(vz*1e-4, vz*1e-8, num_of_particles)   # y velocity
    beam[:, 5] = np.random.normal(vz, vz*1e-6, num_of_particles)   # z velocity
    return beam
```

Figure 6.4. Intermediate functions for beam creation and saving results

```python
def iteration(simulation_time):
    """
        The main iteration loop for time. Number of the particles specified
        and beam is created. Beam iteration occurs according to time stamps.
        simulation_time: Simulation time
    """
    num_particles = 1 << 12
    beam = defineBeam(num_particles)
    magnetic_field_file = pd.read_csv(path + f"\\b-field.csv", delimiter=";")
    time = 0.0

    # Initialize the CSV file for the particle coordinates and velocities
    with open(path + "\\results.csv", "w", newline='') as csv_file:
        csv_writer = csv.DictWriter(csv_file, fieldnames=features)
        csv_writer.writeheader()

    try:
        for step in range(simulation_time):
            # Read electric field file for given time stamp
            electric_field_file = pd.read_csv(path + f"\\e-field_{round(time, 1)}.csv", delimiter=",")
            # Iterate beam
            beam = gpu_particle_iteration(beam, electric_field_file, magnetic_field_file)
            # Save data
            save_data_batch(time, beam)
            time += 0.2
            time = round(time, 1)
            print("Time: {} and particles: {}".format(time, len(beam)))

    # Check GPU related errors
    except cuda.cudadrv.driver.CudaAPIError as e:
        print(f"CUDA Error: {e}")
        print("Falling back to CPU computation...")

    # Check other errors
    except Exception as e:
        print(f"Error during simulation: {e}")
        raise

def main():

    # Check for CUDA device
    if not cuda.is_available():
        print("No CUDA device found.")
        return

    print(f"CUDA device found: {cuda.get_current_device().name}")
    iteration(166)

if __name__ == "__main__":
    main()
```

Figure 6.5. Main loop of the simulation and the main function

APPENDIX B: MCU PROGRAM

The MCU program, including the initialization and configuration functions, can be found in the following.

```
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_HRTIM1_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */
  __HAL_HRTIM_MASTER_ENABLE_IT(&hhrtim1, HRTIM_MASTER_IT_MREP); // Enables master clock
  HAL_UART_Receive_IT(&huart2, incoming_byte, 4);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    if (message_flag == 1)
    {
      received_data();
      message_flag = 0;
    }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Figure 6.6. Main function and infinite while loop

```c
/* Private variables ---------------------------------------------------------*/
HRTIM_HandleTypeDef hhrtim1;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
uint16_t pulse1_start  = 0x0160;
uint16_t pulse1_stop   = 0x2560;
uint16_t pulse2_start  = 0x3F20;
uint16_t pulse2_stop   = 0x9260;
uint32_t message = 0;
uint8_t incoming_byte[10] = {0};
uint8_t message_flag = 0;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_HRTIM1_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */
void message_handling(void);
void ack_send(void);
void nack_send(void);
void processes(void);
uint32_t check_sum(uint32_t message);
void received_data();
/* USER CODE END PFP */
```

Figure 6.7. Some private variable definitions and function prototypes

```c
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_HRTIM1;
  PeriphClkInit.Hrtim1ClockSelection = RCC_HRTIM1CLK_PLLCLK;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
}
```

Figure 6.8. Clock configuration initialization function

```
/**
  * @brief HRTIM1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_HRTIM1_Init(void)
{

  /* USER CODE BEGIN HRTIM1_Init 0 */

  /* USER CODE END HRTIM1_Init 0 */

  HRTIM_TimeBaseCfgTypeDef pTimeBaseCfg = {0};
  HRTIM_TimerCfgTypeDef pTimerCfg = {0};
  HRTIM_CompareCfgTypeDef pCompareCfg = {0};
  HRTIM_OutputCfgTypeDef pOutputCfg = {0};

  /* USER CODE BEGIN HRTIM1_Init 1 */

  /* USER CODE END HRTIM1_Init 1 */
  hhrtim1.Instance = HRTIM1;
  hhrtim1.Init.HRTIMInterruptResquests = HRTIM_IT_NONE;
  hhrtim1.Init.SyncOptions = HRTIM_SYNCOPTION_NONE;
  if (HAL_HRTIM_Init(&hhrtim1) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_HRTIM_DLLCalibrationStart(&hhrtim1, HRTIM_CALIBRATIONRATE_14) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_HRTIM_PollForDLLCalibration(&hhrtim1, 10) != HAL_OK)
  {
    Error_Handler();
  }
  pTimeBaseCfg.Period = 0xFFDF;
  pTimeBaseCfg.RepetitionCounter = 0x00;
  pTimeBaseCfg.PrescalerRatio = HRTIM_PRESCALERRATIO_MUL32;
  pTimeBaseCfg.Mode = HRTIM_MODE_CONTINUOUS;
  if (HAL_HRTIM_TimeBaseConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, &pTimeBaseCfg) != HAL_OK)
  {
    Error_Handler();
  }
  pTimerCfg.InterruptRequests = HRTIM_MASTER_IT_MREP;
  pTimerCfg.DMARequests = HRTIM_MASTER_DMA_NONE;
  pTimerCfg.DMASrcAddress = 0x0000;
  pTimerCfg.DMADstAddress = 0x0000;
  pTimerCfg.DMASize = 0x1;
```

Figure 6.9. HRTIM initialization function part 1

```c
pTimerCfg.HalfModeEnable = HRTIM_HALFMODE_DISABLED;
pTimerCfg.StartOnSync = HRTIM_SYNCSTART_DISABLED;
pTimerCfg.ResetOnSync = HRTIM_SYNCRESET_DISABLED;
pTimerCfg.DACSynchro = HRTIM_DACSYNC_NONE;
pTimerCfg.PreloadEnable = HRTIM_PRELOAD_DISABLED;
pTimerCfg.UpdateGating = HRTIM_UPDATEGATING_INDEPENDENT;
pTimerCfg.BurstMode = HRTIM_TIMERBURSTMODE_MAINTAINCLOCK;
pTimerCfg.RepetitionUpdate = HRTIM_UPDATEONREPETITION_DISABLED;
if (HAL_HRTIM_WaveformTimerConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, &pTimerCfg) != HAL_OK)
{
  Error_Handler();
}
pCompareCfg.CompareValue = pulse1_start;
if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, HRTIM_COMPAREUNIT_1, &pCompareCfg) != HAL_OK)
{
  Error_Handler();
}
pCompareCfg.CompareValue = pulse1_stop;
if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, HRTIM_COMPAREUNIT_2, &pCompareCfg) != HAL_OK)
{
  Error_Handler();
}
pCompareCfg.CompareValue = pulse2_start;
if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, HRTIM_COMPAREUNIT_3, &pCompareCfg) != HAL_OK)
{
  Error_Handler();
}
pCompareCfg.CompareValue = pulse2_stop;
if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1, HRTIM_TIMERINDEX_MASTER, HRTIM_COMPAREUNIT_4, &pCompareCfg) != HAL_OK)
{
  Error_Handler();
}
if (HAL_HRTIM_TimeBaseConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_A, &pTimeBaseCfg) != HAL_OK)
{
  Error_Handler();
}
pTimerCfg.InterruptRequests = HRTIM_TIM_IT_NONE;
pTimerCfg.DMARequests = HRTIM_TIM_DMA_NONE;
pTimerCfg.PushPull = HRTIM_TIMPUSHPULLMODE_DISABLED;
pTimerCfg.FaultEnable = HRTIM_TIMFAULTENABLE_NONE;
```

Figure 6.10. HRTIM initialization function part 2

```
  pTimerCfg.FaultEnable = HRTIM_TIMFAULTENABLE_NONE;
  pTimerCfg.FaultLock = HRTIM_TIMFAULTLOCK_READWRITE;
  pTimerCfg.DeadTimeInsertion = HRTIM_TIMDEADTIMEINSERTION_DISABLED;
  pTimerCfg.DelayedProtectionMode = HRTIM_TIMER_A_B_C_DELAYEDPROTECTION_DISABLED;
  pTimerCfg.UpdateTrigger = HRTIM_TIMUPDATETRIGGER_NONE;
  pTimerCfg.ResetTrigger = HRTIM_TIMRESETTRIGGER_NONE;
  pTimerCfg.ResetUpdate = HRTIM_TIMUPDATEONRESET_DISABLED;
  if (HAL_HRTIM_WaveformTimerConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_A, &pTimerCfg) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_HRTIM_WaveformTimerConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_B, &pTimerCfg) != HAL_OK)
  {
    Error_Handler();
  }
  pOutputCfg.Polarity = HRTIM_OUTPUTPOLARITY_HIGH;
  pOutputCfg.SetSource = HRTIM_OUTPUTSET_MASTERCMP1;
  pOutputCfg.ResetSource = HRTIM_OUTPUTRESET_MASTERCMP2;
  pOutputCfg.IdleMode = HRTIM_OUTPUTIDLEMODE_NONE;
  pOutputCfg.IdleLevel = HRTIM_OUTPUTIDLELEVEL_INACTIVE;
  pOutputCfg.FaultLevel = HRTIM_OUTPUTFAULTLEVEL_NONE;
  pOutputCfg.ChopperModeEnable = HRTIM_OUTPUTCHOPPERMODE_DISABLED;
  pOutputCfg.BurstModeEntryDelayed = HRTIM_OUTPUTBURSTMODEENTRY_REGULAR;
  if (HAL_HRTIM_WaveformOutputConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_A, HRTIM_OUTPUT_TA1, &pOutputCfg) != HAL_OK)
  {
    Error_Handler();
  }
  pOutputCfg.SetSource = HRTIM_OUTPUTSET_MASTERCMP3;
  pOutputCfg.ResetSource = HRTIM_OUTPUTRESET_MASTERCMP4;
  if (HAL_HRTIM_WaveformOutputConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_B, HRTIM_OUTPUT_TB1, &pOutputCfg) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_HRTIM_TimeBaseConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_B, &pTimeBaseCfg) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN HRTIM1_Init 2 */

  /* USER CODE END HRTIM1_Init 2 */
  HAL_HRTIM_MspPostInit(&hhrtim1);

}
```

Figure 6.11. HRTIM initialization function part 3

```
/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 9600;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}
```

Figure 6.12. USART initialization function

APPENDIX C: GUI PROGRAM

Intermediate functions of the GUI program can be found below.

```
class TaskScheduler:
    def __init__(self, initial_interval):
        self.interval = initial_interval
        self._stop_event = threading.Event()
        self.job = None
        self._is_running = False
        self._thread = None

    def start(self):
        if self.job is None:
            raise ValueError("Job function not set.")
        if not self._is_running:
            self._stop_event.clear()
            self._thread = threading.Thread(target=self._run, daemon=True)
            self._thread.start()
            self._is_running = True

    def stop(self):
        if self._is_running:
            self._stop_event.set()
            if self._thread:
                self._thread.join()
            self._is_running = False

    def _run(self):
        while not self._stop_event.is_set():
            if self.job:
                self.job()
            self._stop_event.wait(self.interval)

    def set_interval(self, new_interval):
        self.interval = new_interval

class ComboBox(QComboBox):

    popupAboutToBeShown = pyqtSignal()
    def showPopup(self):
        self.popupAboutToBeShown.emit()
        super(ComboBox, self).showPopup()
```

Figure 6.13. Thread and combo box classes

```python
class PlotCanvas(FigureCanvas):

    def __init__(self, parent=None, width=15, height=15, dpi=100):
        self.fig = Figure(figsize = (width, height), dpi = dpi)
        self.axes = self.fig.add_subplot(111)
        FigureCanvas.__init__(self, self.fig)
        FigureCanvas.setSizePolicy(self,
            QSizePolicy.Expanding,
            QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def plot(self, x, y, color = 'blue', variable_name = 'Signal'):
        self.axes.clear()
        self.axes.plot(x, y, color = color, label = variable_name)
        self.axes.set_xlabel("Time (ns)", fontsize = 7.5)
        self.axes.set_ylabel("Amplitude (%)", fontsize = 7.5)
        self.axes.tick_params(axis = 'both', which = 'major', labelsize = 7.5)
        self.axes.grid(True)
        self.axes.legend(fontsize = 7.5)
        self.axes.set_aspect('auto')
        self.draw()

    def show_image(self, img):
        self.axes.clear()
        self.fig.patch.set_facecolor('lightcyan')
        self.axes.imshow(img)
        self.axes.set_xticks([])
        self.axes.set_yticks([])
        self.draw()

    def clear(self):
        self.axes.clear()
        self.draw()

class Window(QMainWindow):

    def __init__(self):

        super().__init__()
        self.setGeometry(50, 50, 1400, 900)
        self.setWindowTitle("APP")
        self.setWindowIcon(QIcon('kahvelab.png'))
        self.tabWidget()
        self.Widgets()
        self.layouts()
        self.init_variables()
        self.show()

    def tabWidget(self):

        self.tabs = QTabWidget()
        self.setCentralWidget(self.tabs)
        self.tab1 = QWidget()
        self.tabs.addTab(self.tab1, "Marx Generator Control GUI")
```

Figure 6.14. Plot class and window class initializations

```python
def Widgets(self):

    self.pltObj = PlotCanvas(self)

    self.baudrate_list_qlabel = QLabel("Baud Rate")
    self.baudrate_cb = QComboBox(self)
    self.baudrate_cb.addItems(["----------", "300", "600", "1200", "2400",
                               "4800", "9600", "14400", "19200",
                               "28800", "38400", "56000", "57600",
                               "115200", "128000", "256000"])
    self.baudrate_cb.currentTextChanged.connect(self.baudrate_clicked_func)

    self.p1_start_label = QLabel("Change pulse 1 start variable", self)
    self.p1_start_txtbox = QLineEdit(self)
    self.p1_start_button = QPushButton("OK", self)
    self.p1_start_button.clicked.connect(self.p1_start_func)
    self.p1_start_button.setEnabled(False)

    self.p1_stop_qlabel = QLabel("Change pulse 1 stop variable", self)
    self.p1_stop_txtbox = QLineEdit(self)
    self.p1_stop_button = QPushButton("OK", self)
    self.p1_stop_button.clicked.connect(self.p1_stop_func)
    self.p1_stop_button.setEnabled(False)

    self.p2_start_label = QLabel("Change pulse 2 start variable", self)
    self.p2_start_txtbox = QLineEdit(self)
    self.p2_start_button = QPushButton("OK", self)
    self.p2_start_button.clicked.connect(self.p2_start_func)
    self.p2_start_button.setEnabled(False)

    self.p2_stop_qlabel = QLabel("                 Change pulse 2 stop variable", self)
    self.p2_stop_txtbox = QLineEdit(self)
    self.p2_stop_button = QPushButton("OK", self)
    self.p2_stop_button.clicked.connect(self.p2_stop_func)
```

Figure 6.15. Widgets function part 1

```python
    self.p2_stop_button.setEnabled(False)

    self.rep_rate_qlabel = QLabel("Repetition time", self)
    self.rep_rate_txtbox = QLineEdit(self)
    self.rep_rate_button = QPushButton("OK", self)
    self.rep_rate_button.clicked.connect(self.rep_func)
    self.rep_rate_button.setEnabled(False)

    self.com_qlabel = QLabel("Enter COM", self)
    self.com_cb = ComboBox(self)
    self.com_cb.addItem("----------")
    self.com_cb.popupAboutToBeShown.connect(self.update_coms_func)

    self.pulse_qlabel = QLabel("Pulse mode", self)
    self.pulse_cb = ComboBox(self)
    self.pulse_cb.addItem("Single pulse")
    self.pulse_cb.addItem("Repeated pulses")
    self.pulse_cb.activated[str].connect(self.mode_var_func)

    self.connect_button = QPushButton("Connect", self)
    self.connect_button.clicked.connect(self.connect_MCU)

    self.trigger_button = QPushButton("SEND PULSE", self)
    self.trigger_button.clicked.connect(self.send_pulse)
    self.trigger_button.setEnabled(False)

    self.configure_button = QPushButton("SET configuration of the pulses", self)
    self.configure_button.clicked.connect(self.send_configurations)
    self.configure_button.setEnabled(False)

    self.serial_monitor_list =  QListWidget()
    self.serial_monitor_list.addItem("Welcome Beam Chopper GUI!")
    self.serial_monitor_list.addItem("Maximum timer resolution is 217 ps!")
    self.serial_monitor_list.addItem("Please read the docstring before use the program!")

    self.clear_serial_monitor = QPushButton("CLEAR", self)
    self.clear_serial_monitor.clicked.connect(self.clear_list)

    img = mpimg.imread('kahvelab.png')
    self.pltObj.show_image(img)
```

Figure 6.16. Widgets function part 2

```python
def init_variables(self):

    self.packet_size = 4
    self.baud_rate = 0
    self.com = ""
    self.rep_time = 1.0
    self.pulse_mode = 0  # 0 means that single pulse, 1 means that repeated pulses
    self.header = 0xA
    self.ack = 0xF
    self.nack = 0x0
    self.send_list = []
    self.scheduler = TaskScheduler(self.rep_time)

def message_packet(self, mode: bytes, message: bytes):

    if not (0x0 <= mode <= 0xF):
        info_box = QMessageBox.information(self, "WARNING!", "It is not a valid input!")
        return -1
    if not (0x0 <= message <= 0xFFFF):
        info_box = QMessageBox.information(self, "WARNING!", "It is not a valid input!")
        return -1
    packet_check = self.header | (mode << 4) | (message << 8)
    check = self.check_func(packet_check)
    message = packet_check | (check << 24)
    return message

def check_func(self, message: bytes) -> int:

    sum = 0
    for i in range(24):
        sum += (message >> i) & 0b1
    return sum

def mode_var_func(self, mode_text):

    if mode_text == "Repeated pulses":
        self.pulse_mode = 1
        if self.rep_time <= 0:
            self.rep_time = 1.0
        self.scheduler.job = self.trigger_pulse
        self.scheduler.set_interval(self.rep_time)
        self.scheduler.start()
    elif mode_text == "Single pulse":
        self.scheduler.stop()
        self.pulse_mode = 0

def send_pulse(self):
    if self.pulse_mode == 0:
        self.trigger_pulse()
    elif self.pulse_mode == 1:
        # Optional: You might want to remove this or handle differently
        # since starting is now managed in mode_var_func
        self.scheduler.start()
```

Figure 6.17. Intermediate functions part 1

```python
def serial_monitor(self, item):

    self.serial_monitor_list.addItem(item)

def clear_list(self):

    self.serial_monitor_list.clear()

def trigger_pulse(self):

    self.send(modee=0x2, msg=0x0000)

def validate_range(self, value):

    MIN_VALUE = 0x0060  # 96 in decimal
    MAX_VALUE = 0xFFFF  # 65535 in decimal
    if value < MIN_VALUE:
        return MIN_VALUE
    elif value > MAX_VALUE:
        return MAX_VALUE
    return value

def p1_start_func(self):

    try:
        p1_start = self.validate_range(int(self.p1_start_txtbox.text()))
        self.send_list.append([0x3, p1_start])
    except ValueError:
        QMessageBox.information(self, "WARNING!", "Please enter valid numbers between 96 and 65535!")
        return -1

def p1_stop_func(self):

    try:
        p1_stop = self.validate_range(int(self.p1_stop_txtbox.text()))
        self.send_list.append([0x4, p1_stop])
    except ValueError:
        QMessageBox.information(self, "WARNING!", "Please enter valid numbers between 96 and 65535!")
        return -1

def p2_start_func(self):

    try:
        p2_start = self.validate_range(int(self.p2_start_txtbox.text()))
        self.send_list.append([0x5, p2_start])
    except ValueError:
        QMessageBox.information(self, "WARNING!", "Please enter valid numbers between 96 and 65535!")
        return -1
```

Figure 6.18. Intermediate functions part 2

```python
    def p2_stop_func(self):

        try:
            p2_stop = self.validate_range(int(self.p2_stop_txtbox.text()))
            self.send_list.append([0x6, p2_stop])
        except ValueError:
            QMessageBox.information(self, "WARNING!", "Please enter valid numbers between 96 and 65535!")
            return -1

    def rep_func(self):

        try:
            self.rep_time = float(self.rep_rate_txtbox.text())
            if self.rep_time < 0.1:
                self.rep_time = 0.1
            elif self.rep_time < 0:
                self.rep_time = 0
            self.scheduler.set_interval(self.rep_time)
        except ValueError:
            info_box = QMessageBox.information(self, "WARNING!", "It is not a valid input!")
            return -1

    def send_configurations(self):

        if self.mcu:
            for data in self.send_list:
                self.send(modee = data[0], msg = data[1])
                time.sleep(0.01)
                self.receive()
            self.plot_pulse_signal()
            self.send_list = []
        else:
            info_box = QMessageBox.information(self, "WARNING!", "NO MCU CONNECTION!")

    def baudrate_clicked_func(self):

        self.baud_rate = int(self.baudrate_cb.currentText())
```

Figure 6.19. Intermediate functions part 3

```python
def connect_MCU(self):

    self.com = self.com_cb.currentText()
    self.clear_list()

    if self.com != "" and self.baud_rate != 0:

        self.mcu = serial.Serial(self.com, self.baud_rate, timeout = 0.3)
        time.sleep(0.5)
        self.send(modee = 0x1, msg = 0xFF)
        self.receive()
        # Buttons and comboboxs are enabled if MCU is connected!
        self.p1_stop_button.setEnabled(True)
        self.p2_stop_button.setEnabled(True)
        self.p2_start_button.setEnabled(True)
        self.p1_start_button.setEnabled(True)
        self.rep_rate_button.setEnabled(True)
        self.trigger_button.setEnabled(True)
        self.configure_button.setEnabled(True)
        self.pulse_cb.setEnabled(True)

def update_coms_func(self):

    self.com_cb.clear()
    com_list = list_ports.comports()
    for port in com_list:
        port = str(port)
        port_name = port.split("-")
        self.com_cb.addItem(port_name[0])

def plot_pulse_signal(self):

    p1_start = self.validate_range(int(self.p1_start_txtbox.text()))
    p1_stop  = self.validate_range(int(self.p1_stop_txtbox.text()))
    p2_start = self.validate_range(int(self.p2_start_txtbox.text()))
    p2_stop  = self.validate_range(int(self.p2_stop_txtbox.text()))
    p1_start_ps   = p1_start * 217
    p1_stop_ps    = p1_stop * 217
    p2_start_ps   = p2_start * 217
    p2_stop_ps    = p2_stop * 217

    total_time_ps = p1_start_ps + p1_stop_ps + p2_start_ps + p2_stop_ps
    t = [0, p1_start_ps, p1_start_ps, p1_stop_ps, p1_stop_ps, p2_start_ps,
         p2_start_ps, p2_stop_ps, p2_stop_ps, total_time_ps]
    t = [i/1000 for i in t]
    signal = [0, 0, 100, 100, 0, 0, 100, 100, 0, 0]
    p1_width = (p1_stop_ps - p1_start_ps)/1000
    p2_width = (p2_stop_ps - p2_start_ps)/1000
    delay_btw_pulses = (p2_start_ps - p1_stop_ps)

    self.pltObj.clear()
    self.pltObj.plot(
        x=t,
        y=signal,
        color='blue',
        variable_name=f'Dual Pulse (W1={p1_width:,.0f}ns, W2={p2_width:,.0f}ns, D={delay_btw_pulses:,.0f}ns)'
    )
```

Figure 6.20. Intermediate functions part 4

```
def layouts(self):

    self.main_layout   = QHBoxLayout()
    self.right_layout  = QFormLayout()
    self.left_layout   = QFormLayout()
    self.plot_layout   = QVBoxLayout()
    self.left_Hbox1    = QHBoxLayout()
    self.left_Hbox2    = QHBoxLayout()
    self.left_Hbox3    = QHBoxLayout()
    self.left_Hbox4    = QHBoxLayout()
    self.left_Hbox5    = QHBoxLayout()
    self.right_Vbox    = QVBoxLayout()

    # Left layout
    self.left_layout_group_box = QGroupBox("")
    self.left_Hbox1.addWidget(self.baudrate_list_qlabel)
    self.left_Hbox1.addWidget(self.baudrate_cb)
    self.left_Hbox1.addStretch()
    self.left_Hbox1.addWidget(self.com_qlabel)
    self.left_Hbox1.addWidget(self.com_cb)
    self.left_Hbox1.addStretch()
    self.left_Hbox1.addWidget(self.connect_button)
    self.left_layout.addRow(self.left_Hbox1)

    self.plot_layout.addWidget(self.pltObj)
    self.left_layout.addRow(self.plot_layout)

    self.left_Hbox2.addWidget(self.p1_start_label)
    self.left_Hbox2.addWidget(self.p1_start_txtbox)
    self.left_Hbox2.addWidget(self.p1_start_button)
    self.left_Hbox2.addStretch()
    self.left_Hbox2.addWidget(self.p2_start_label)
    self.left_Hbox2.addWidget(self.p2_start_txtbox)
    self.left_Hbox2.addWidget(self.p2_start_button)
    self.left_Hbox2.addStretch()
    self.left_layout.addRow(self.left_Hbox2)

    self.left_Hbox3.addWidget(self.p1_stop_qlabel)
    self.left_Hbox3.addWidget(self.p1_stop_txtbox)
    self.left_Hbox3.addWidget(self.p1_stop_button)
    self.left_Hbox3.addStretch()
    self.left_Hbox3.addWidget(self.p2_stop_qlabel)
    self.left_Hbox3.addWidget(self.p2_stop_txtbox)
    self.left_Hbox3.addWidget(self.p2_stop_button)
    self.left_Hbox3.addStretch()
    self.left_layout.addRow(self.left_Hbox3)

    self.left_Hbox3.addStretch()
    self.left_Hbox4.addWidget(self.configure_button)
    self.left_layout.addRow(self.left_Hbox4)
```

Figure 6.21. Layout part 1

```
        self.left_Hbox5.addWidget(self.pulse_qlabel)
        self.left_Hbox5.addWidget(self.pulse_cb)
        self.left_Hbox5.addStretch()
        self.left_Hbox5.addWidget(self.rep_rate_qlabel)
        self.left_Hbox5.addWidget(self.rep_rate_txtbox)
        self.left_Hbox5.addWidget(self.rep_rate_button)
        self.left_Hbox5.addStretch()
        self.left_Hbox5.addWidget(self.trigger_button)
        self.left_Hbox5.addStretch()
        self.left_layout.addRow(self.left_Hbox5)

        self.left_layout_group_box.setLayout(self.left_layout)

        # Right layout
        self.right_layout_group_box = QGroupBox("Serial Monitor")

        self.right_Vbox.addWidget(self.serial_monitor_list)
        self.right_Vbox.addWidget(self.clear_serial_monitor)
        self.right_layout.addRow(self.right_Vbox)

        self.right_layout_group_box.setLayout(self.right_layout)

        self.main_layout.addWidget(self.left_layout_group_box, 60)
        self.main_layout.addWidget(self.right_layout_group_box, 40)
        self.tab1.setLayout(self.main_layout)

def main():

    app = QApplication(sys.argv)
    app.setStyleSheet(qdarkstyle.load_stylesheet())
    window = Window()
    sys.exit(app.exec_())

if __name__ == "__main__":

    main()
```

Figure 6.22. Layout part 2 and main function

APPENDIX D: ARDUINO UNO PROGRAM

The Arduino Uno program is shared below.

```
const int buttonPin = 2;  // Button connected to 2 pin
volatile bool pulseGenerating = false;

void setup() {
  pinMode(buttonPin, INPUT); // activate internal pull-down resistor for button pin
  pinMode(9, OUTPUT);             // Set pin 9 as output (OC1A) (Pulse Pin)

  // Configure Timer1
  noInterrupts();           // Disable all interrupts
  TCCR1A = 0;               // Clear Timer1 control register A
  TCCR1B = 0;               // Clear Timer1 control register B
  TCNT1 = 0;                // Clear Timer1 counter register

  TCCR1A |= (1 << WGM11) | (1 << COM1A1);   // Set Timer1 to Fast PWM mode with ICR1 as top and non-inverting mode on OC1A
  TCCR1B |= (1 << WGM12) | (1 << WGM13);    // Set Timer1 to Fast PWM mode with ICR1 as top

  ICR1 = 159;                // Set the top value to 159 for a 10-microsecond period

  OCR1A = 15;                // Set the compare value to 15 for a 1-microsecond pulse width
                             // set the value 31 for 2-microseconds
                             // set the value 7 for 500-nanoseconds
                             // set the value  3 for 250-nanoseconds


  interrupts();             // Enable all interrupts
}

void loop() {
  if (digitalRead(buttonPin) == HIGH) {
    delay(200); //button debouncing
    if (!pulseGenerating) {
      generatePulse();
    }
  }
}

void generatePulse() {
  if (!pulseGenerating) {
    pulseGenerating = true;

    TCCR1A |= (1 << COM1A1);  // Enable the output compare match on OC1A (pin 9)
    TCCR1B |= (1 << CS10);    // Start the timer with no prescaler
    TIMSK1 |= (1 << OCIE1A);  // Enable Timer1 output compare A match interrupt
  }
}

ISR(TIMER1_COMPA_vect) {
  // This ISR will be called when the timer reaches target value
  TCCR1A &= ~(1 << COM1A1); // Disable the output compare match on OC1A (pin 9)
  TCCR1B &= ~(1 << CS10);   // Stop the timer
  TIMSK1 &= ~(1 << OCIE1A); // Disable Timer1 output compare A match interrupt

  pulseGenerating = false;  // Reset the flag after pulse generation
}
```

Figure 6.23. Arduino Uno test code

APPENDIX E: BEAM VISUALIZATION PROGRAM

The intermediate codes of the beam visualization program are shared here.

```
class ParticleDynamicsSimulator:

    def __init__(self, data_path):

        sns.set_theme()
        df = pd.read_csv(data_path, delimiter=",")
        self.data = np.zeros(len(df), dtype=[
            ('time', 'f8'),
            ('x', 'f8'),
            ('y', 'f8'),
            ('z', 'f8'),
            ('vx', 'f8'),
            ('vy', 'f8'),
            ('vz', 'f8'),
            ('synch', 'i4')
        ])
        self.data['time'] = df['time'].values
        self.data['x'] = df['x'].values
        self.data['y'] = df['y'].values
        self.data['z'] = df['z'].values
        self.data['vx'] = df['vx'].values
        self.data['vy'] = df['vy'].values
        self.data['vz'] = df['vz'].values
        del(df)

        self.time_stamps = np.unique(self.data['time'])
        self.len_initial_particles = len(self.data[self.data['time'] == self.time_stamps[0]])
        initial_particles = self.data[self.data['time'] == self.time_stamps[0]]
        self.data['synch'] = self.synch_particles(initial_particles, self.time_stamps)
        self.mask_list = np.ones(self.len_initial_particles, dtype = bool)
        self.eliminated_particles = 0
        self.setup_multi_plot()

    def synch_particles(self, beam, timestamps):
            synch = []
            is_synch = np.array([1 if (beam['z'][i] < -0.7 and beam['z'][i] > -0.9)
                                     else 0 for i in range(len(beam))])
            for _ in timestamps:
                synch.append(is_synch)
            synch_array = np.array(synch).reshape(len(self.data))
            return synch_array
```

Figure 6.24. Beam visualization init function and synchronous particles mark algorithm

```python
def setup_multi_plot(self):

    self.fig = plt.figure(figsize=(20, 15))
    self.fig.patch.set_facecolor('lightcyan')
    gs = self.fig.add_gridspec(3, 2, width_ratios=[2, 1], height_ratios=[1, 1, 1])

    # Beam dynamics plot
    self.ax1 = self.fig.add_subplot(gs[:, 0])
    self.ax1.set_xlim(-500, 1500)
    self.ax1.set_ylim(-200, 150)
    self.ax1.set_facecolor('white')
    self.ax1.grid(True, linestyle='--', alpha=0.3)
    self.ax1.set_xlabel('z (mm)', fontsize=10)
    self.ax1.set_ylabel('y (mm)', fontsize=10)
    self.ax1.set_title('Particle Beam Trajectory', fontsize=12, pad=10)

    # X-Y Distribution plot
    self.ax2 = self.fig.add_subplot(gs[0, 1])
    self.ax2.set_title('Synch Particles X-Y Distribution', fontsize=10)
    self.ax2.set_xlabel('x (mm)', fontsize=8)
    self.ax2.set_ylabel('y (mm)', fontsize=8)

    # X-X' Phase Space plot
    self.ax3 = self.fig.add_subplot(gs[1, 1])
    self.ax3.set_title('Synch Particles X-X\' Phase Space', fontsize=10)
    self.ax3.set_xlabel('x (mm)', fontsize=8)
    self.ax3.set_ylabel("x' (mrad)", fontsize=8)

    # Y-Y' Phase Space plot
    self.ax4 = self.fig.add_subplot(gs[2, 1])
    self.ax4.set_title('Synch Particles Y-Y\' Phase Space', fontsize=10)
    self.ax4.set_xlabel('y (mm)', fontsize=8)
    self.ax4.set_ylabel("y' (mrad)", fontsize=8)

    plt.tight_layout()
```

Figure 6.25. Multi plot function

```python
def draw_boundaries(self):
    """Draw boundary lines for different regions"""
    # Region 1: -5 < z < -0.16815
    z_start, z_end = -5000, -50
    z_limits = self.ax1.get_xlim()
    total_width = z_limits[1] - z_limits[0]
    xmin = (z_start - z_limits[0]) / total_width
    xmax = (z_end - z_limits[0]) / total_width

    self.ax1.axhline(y=40, xmin=xmin, xmax=xmax, color='black', linewidth=1)
    self.ax1.axhline(y=-40, xmin=xmin, xmax=xmax, color='black', linewidth=1)

    # Region 2: -0.16815 <= z <= -0.05
    z_start, z_end = -168.15, -50
    xmin = (z_start - z_limits[0]) / total_width
    xmax = (z_end - z_limits[0]) / total_width

    self.ax1.axhline(y=20, xmin=xmin, xmax=xmax, color='black', linewidth=1)
    self.ax1.axhline(y=-20, xmin=xmin, xmax=xmax, color='black', linewidth=1)

    # Region 3: -0.05 < z < 0.5
    z_start, z_end = -49, 470
    z_points = np.array([z_start, z_end])
    y_base = 40
    tan_angle = np.tan(np.deg2rad(180-15))
    y_points = y_base + (z_points - z_start) * tan_angle
    self.ax1.plot(z_points, y_points, 'k-', linewidth=1)

    z_start, z_end = -49, 418
    z_points = np.array([z_start, z_end])
    y_base = -40
    y_points = y_base + (z_points - z_start) * tan_angle
    self.ax1.plot(z_points, y_points, 'k-', linewidth=1)

    # Additional boundary lines in Region 3
    z_start, z_end = 313, 430
    z_points = np.array([z_start, z_end])
    y_start, y_end = -75, -105.5
    y_points = np.array([y_start, y_end])
    self.ax1.plot(z_points, y_points, 'k-', linewidth=1)

    z_start, z_end = 305, 419
    z_points = np.array([z_start, z_end])
    y_start, y_end = -118.15, -148.728
    y_points = np.array([y_start, y_end])
    self.ax1.plot(z_points, y_points, 'k-', linewidth=1)

    # Region 4: 0.454 < z < 1.500
    z_start, z_end = 470, 1500
    xmin = (z_start - z_limits[0]) / total_width
    xmax = (z_end - z_limits[0]) / total_width
    self.ax1.axhline(y=-134 + 35, xmin=xmin, xmax=xmax, color='black', linewidth=1)

    z_start, z_end = 420, 1500
    xmin = (z_start - z_limits[0]) / total_width
    xmax = (z_end - z_limits[0]) / total_width
    self.ax1.axhline(y=-134 - 31, xmin=xmin, xmax=xmax, color='black', linewidth=1)
```

Figure 6.26. Geometry definition function of the beam pipe

```
def check_particle_boundaries(self, x, y, z):

    # Region 1: z < -0.5 mm
    if z < -0.168:
        if abs(x) > 0.04 or abs(y) > 0.04:
            return False

    # Region 2: -0.168 <= z <= -0.05 mm
    elif -0.168 <= z <= -0.05:
        if abs(x) > 0.04 or abs(y) > 0.02:
            return False

    # Region 3: -0.05 < z < 0.454 mm
    elif -0.05 < z < 0.324:
        y_upper = 0.04 + (z + 0.049) * (-0.2672)
        y_lower = -0.04 + (z + 0.049) * (-0.2672)

        if abs(x) > 0.04:
            return False

        # Main y boundaries
        if y > y_upper or y < y_lower:
            return False

    # Parallel plate boundary check
    elif 0.324 <= z <= 0.438:
        y_upper = -0.075 + (z - 0.313) * (-0.2672)
        y_lower = -0.118 + (z - 0.313) * (-0.2672)
        if abs(x) > 0.04:
            return False

        if y > y_upper or y < y_lower:
            return False


    # Region 4: 0.438 <= z < 1 mm
    elif 0.438 < z <= 1.5:
        if abs(x) > 0.04:
            return False
        if y > (-0.134 + 0.039) or y < (-0.134 - 0.031):
            return False

    return True
```

Figure 6.27. Particle boundary check function

```python
def run_simulation(self):

    self.fig.show()
    for time in self.time_stamps:

        beam = self.get_beam_at_time(time)
        mask = []
        for i in range(len(beam)):
            is_valid = self.check_particle_boundaries(
                beam['x'][i],
                beam['y'][i],
                beam['z'][i]
            )
            mask.append(is_valid)

        mask = np.array(mask, dtype=bool)
        self.mask_list = self.mask_list & mask
        beam = beam[self.mask_list]
        for ax in [self.ax1, self.ax2, self.ax3, self.ax4]:
            ax.clear()

        self.ax1.set_xlim(-500, 1500)
        self.ax1.set_ylim(-200, 150)
        self.ax1.set_facecolor('white')
        self.ax1.grid(True, linestyle='--', alpha=0.3)
        self.ax1.set_xlabel('z (mm)', fontsize=10)
        self.ax1.set_ylabel('y (mm)', fontsize=10)
        self.ax1.set_title('Beam Dynamics', fontsize=12, pad=10)

        self.draw_boundaries()

        non_synch_particles = beam[beam['synch'] == 0]
        synch_particles = beam[beam['synch'] == 1]

        # Main trajectory plot
        self.ax1.scatter(non_synch_particles['z']*1000, non_synch_particles['y']*1000,
                        s=0.2, c='blue', alpha=0.6, label='Non-Synch Particles')
        self.ax1.scatter(synch_particles['z']*1000, synch_particles['y']*1000,
                        s=0.2, c='red', alpha=0.6, label='Synch Particles')
        self.ax1.legend(loc='lower right')

        # X-Y distribution for synch particles
        self.ax2.set_xlabel('x (mm)', fontsize=8)
        self.ax2.set_ylabel('y (mm)', fontsize=8)
        self.ax2.scatter(synch_particles['x']*1000, synch_particles['y']*1000,
                         s=1, c='red', alpha=0.6)
```

Figure 6.28. Plot loop part 1

```python
            # Calculate x' and y' for synch particles
            x_prime = synch_particles['vx'] / synch_particles['vz']
            y_prime = synch_particles['vy'] / synch_particles['vz']

            # X-X' Phase Space for Synch Particles
            self.ax3.set_xlabel('x (mm)', fontsize=8)
            self.ax3.set_ylabel("x'(mrad)", fontsize=8)
            self.ax3.scatter(synch_particles['x']*1000, x_prime*1000,
                                 s=1, c='red', alpha=0.6)

            # Y-Y' Phase Space for Synch Particles
            self.ax4.set_xlabel('y (mm)', fontsize=8)
            self.ax4.set_ylabel("y'(mrad)", fontsize=8)
            self.ax4.scatter(synch_particles['y']*1000, y_prime*1000,
                                 s=1, c='red', alpha=0.6)

            # Statistics and time display
            stats = self.calculate_statistics(beam)
            self.display_statistics(stats)
            self.display_time(time)

            self.fig.canvas.flush_events()
            self.fig.canvas.draw()

        plt.close()

if __name__ == "__main__":
    path = "C:\\Users\\ardau\\OneDrive\\Desktop\\MasterThesis\\BeamSimulation\\beam_sim_python\\finalSim\\electricField
    simulation = ParticleDynamicsSimulator(path)
    simulation.run_simulation()
```

Figure 6.29. Plot loop part 2

APPENDIX F: MARX GENERATOR OUTPUT WAVEFORMS

Some of the results of the Marx Generator are added here.

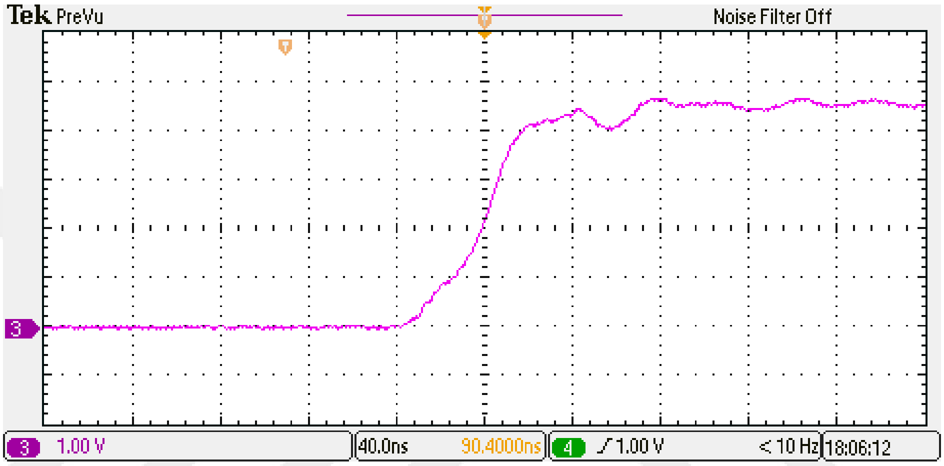

Figure 6.30. 500 V output waveform rising edge

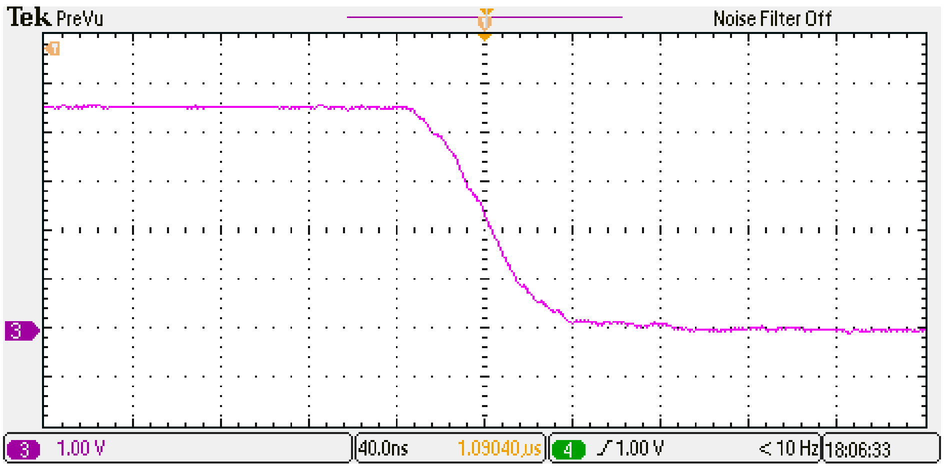

Figure 6.31. 500 V output waveform falling edge

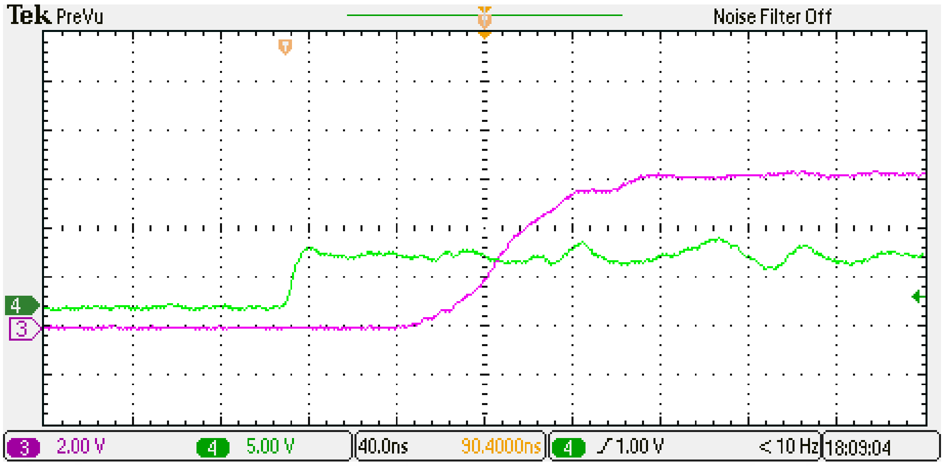

Figure 6.32. 700 V output waveform rising edge

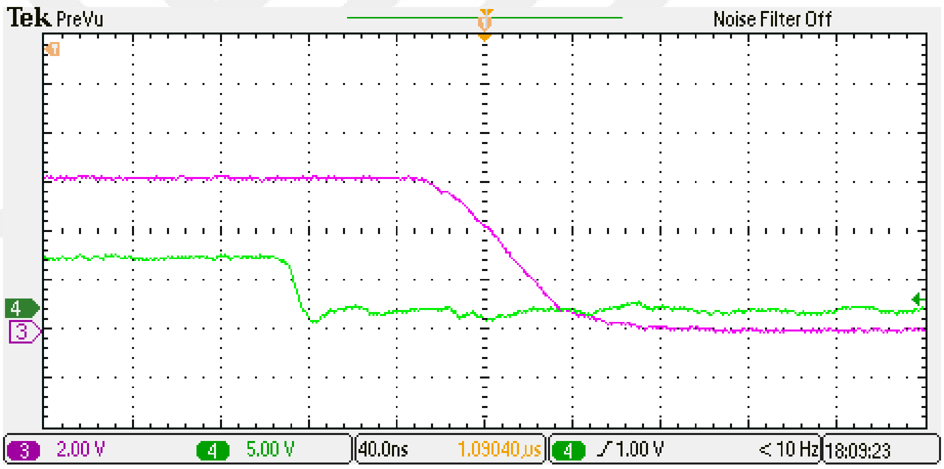

Figure 6.33. 700 V output waveform falling edge

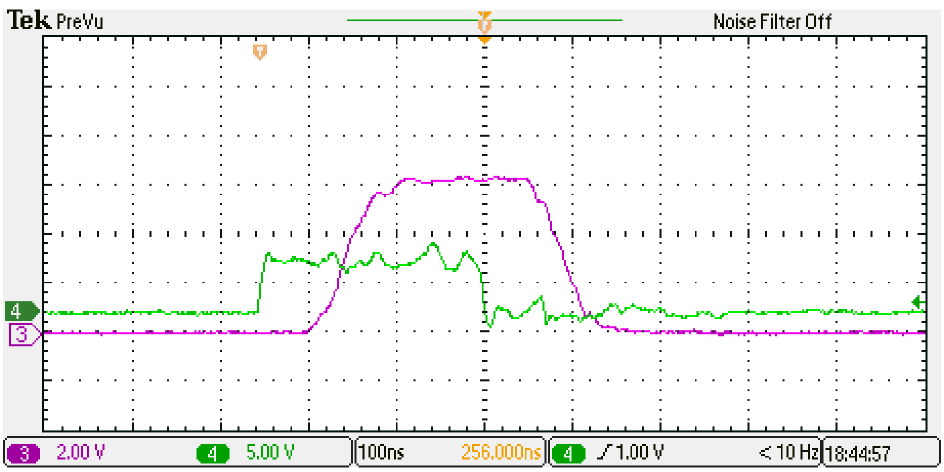

Figure 6.34. 700 V output waveform with 250 ns pulse width

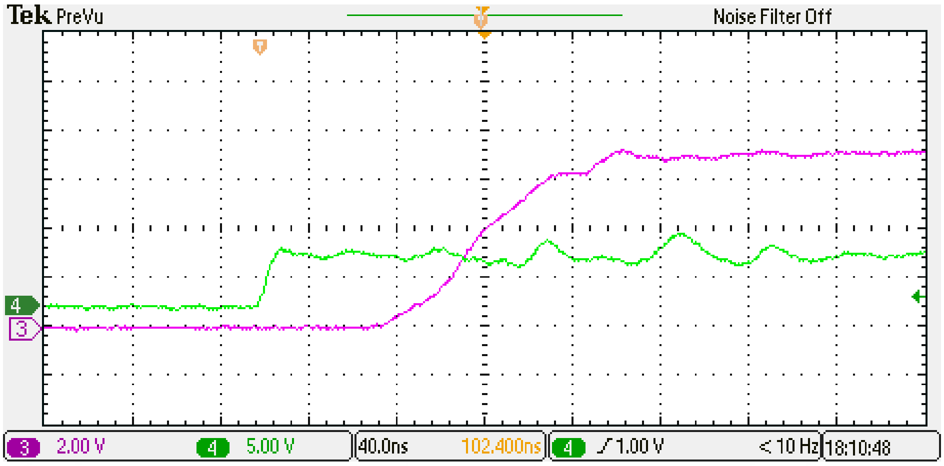

Figure 6.35. 800 V one PCB output waveform rising edge

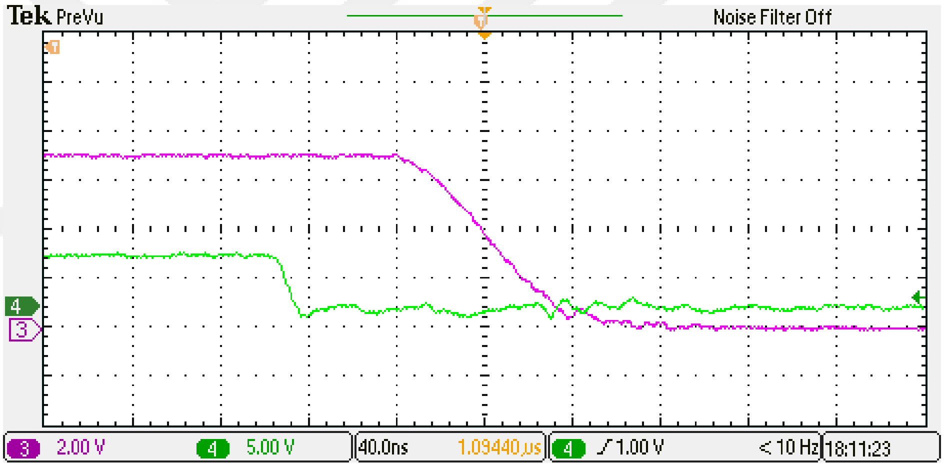

Figure 6.36. 800 V one PCB output waveform falling edge

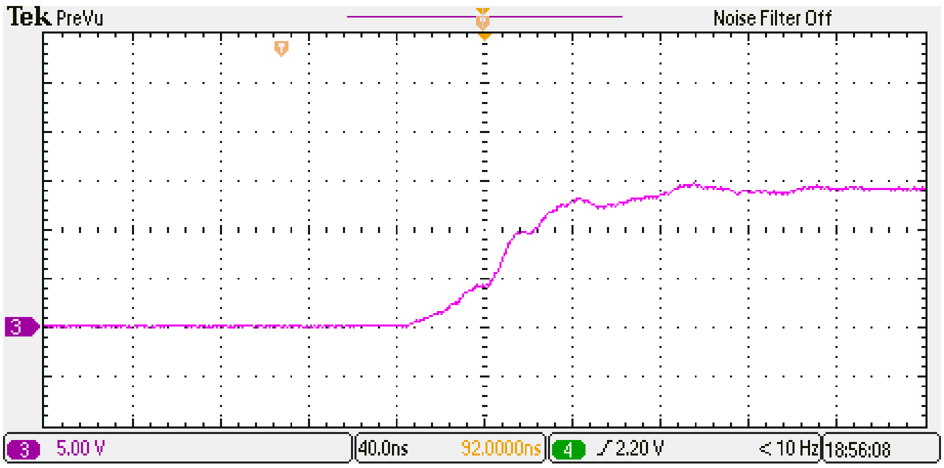

Figure 6.37. 800 V two PCBs rising edge of the output waveform

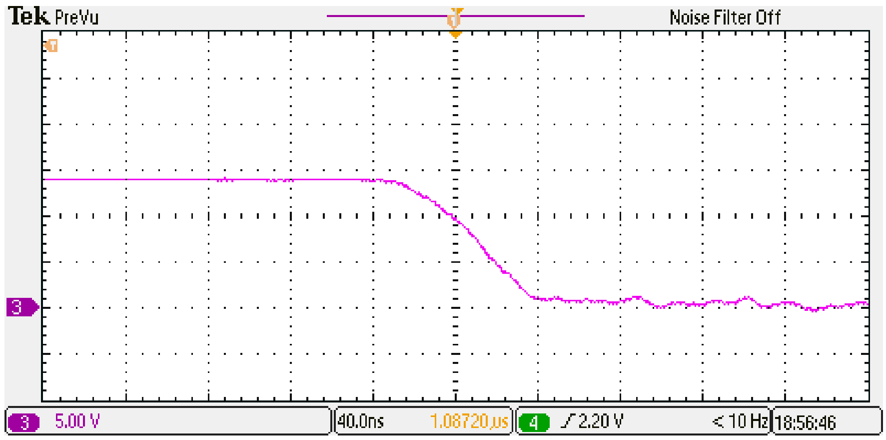

Figure 6.38. 800 V two PCBs falling edge of the output waveform

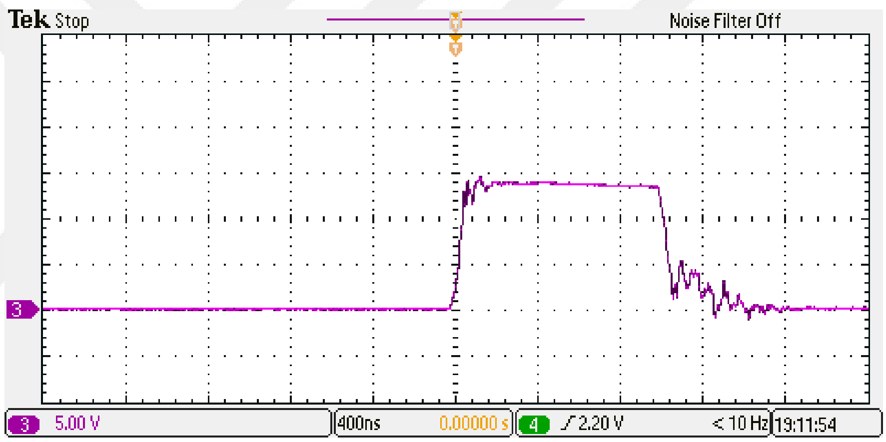

Figure 6.39. Output waveform of the three cascaded Marx generator units supplied by 500 V

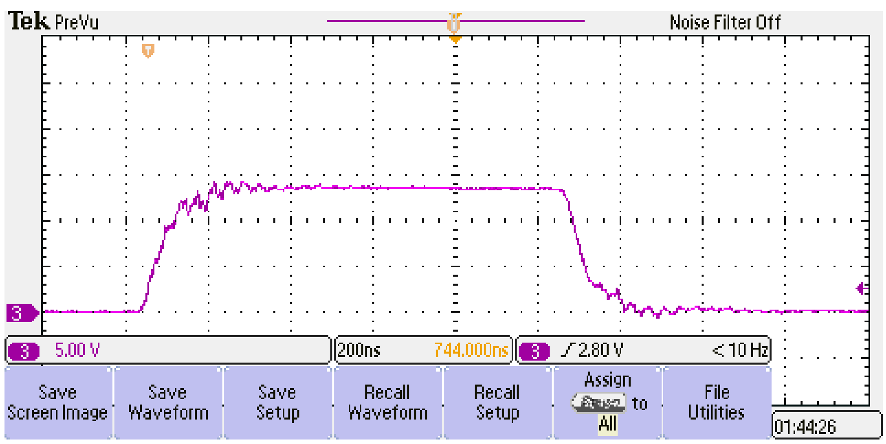

Figure 6.40. Another example of the output waveform of the three cascaded Marx generator units supplied by 500 V