A masters thesis

Department of Industrial Electrical Engineering and Automation

Lund Institute of Technology

Design Of A Current Controlled Defibrillator

By
Magnus Jonsson, e99
Filip Jörgensen, e99

Supervised by:
Per Karlsson

Abstract

This report describes the design procedures and underlying theory needed in order to engineer a current controlled defibrillator. Electric circuits are presented and explained together with the theory for some of the more important components like the transformer and the IGBT. The need for protective circuitry regarding overvoltage is investigated and safety issues are discussed.

The high voltage is achieved by using a flyback converter to charge a capacitor. This capacitor is discharged via an H-bridge and the current is controlled using a hardware tolerance band controller. Three types of discharge are available, monophasic, biphasic and triphasic.

The entire system is controlled by an ATMega8 processor monitoring the charging and creating the necessary PWM.

Aknowledgement

During this masters thesis much help has been recieved from several people and companies. The following is an alphabetical list of the people and companies we would like to thank.

- *Mats Alaküla*, professor at the department of industrial electrical engineering and automation, LTH
- *Coilcraft*, manufacturer of coils and transformers, supplying us with free samples
- *Getaschew Darge*, engineer and technichian at the department of industrial electronics and automation, LTH
- *Fairchild semiconductors*, manufacturer of semiconductors, supplying us with free samples
- *Eilert Johansson*, engineer at Profec AB, manufacturer of custom made transformers
- *Per Karlsson*, our supervisor at the department of industrial electrical engineering and automation, LTH
- *Maxim IC*, manufacturer of semiconductors, supplying us with free samples
- *Audrius Paskevicius*, our supervisor at Xenodenice AB
- *Johan Åkesson*, tech. lic. at the department of automatic control providing us with knowledge of discrete digital controllers.

Contents

1 Introduction

Today, very few people survive a cardiac arrest (about 3 %) [8]. The problem with cardiac arrests will most likely escalate since the western lifestyle increases the risk of heart and vascular diseases.

In order for the survival rate of people suffering from cardiac arrest to increase, extensive and often immediate cardiac massage is vital. At present, this represents a problem since any normal person has trouble supplying sufficient cardiac massage for more than two minutes due to the excessive stamina required.

A machine called LUCAS (Figure 1) has been developed to provide the necessary cardiac massage. If a patient suffering from cardiac arrest receives immediate treatment with LUCAS, the survival rate from cardiac arrests could increase as much as 30%.

Figure 1: LUCAS

The commercial version of LUCAS is currently powered by compressed air and is clamped around the patient. LUCAS has been tested in some ambulances in Skåne läns andsting, Sweden, and has worked very well.

The emergency personnel working with LUCAS are very satisfied, but has requested more functionality in the machine. Today, emergency personnel have to carry both LUCAS and a portable defibrillator. This is rather heavy since LUCAS, being powered by compressed air, has to be accompanied by an air tank. A new and improved version of the LUCAS should contain a defibrillator, a pacemaker and an ECG[1]. Of these three the defibrillator is the most desired enhancement.

---

[1] Electro CardioGram

2 Scope and purpose

The purpose of this master thesis is to investigate the possibility of integrating a fully functional medical defibrillator into LUCAS. As previously described this is a requested functionality from a user point of view and therefore an important step forward in the development process for LUCAS.

There are a number of activities that has been conducted in order to reach the desired goal. First of all, there are a number of medical requirements that has to be taken into account. These requirements are not carved into stone since this is an area of intensive development and discussions on how to achieve the best clinical results. This leads to the fact that the defibrillation unit initially has to have an extensive set of adjustable parameters. These parameters will later be fixed when elaborative tests show what settings seems to give the best results.

The next step is to investigate component selection and different circuit layouts that will meet the maximum ratings for the previously defined requirements. This is a difficult area since cost, weight and electrical specifications has to be taken into account. Simulations and calculations are needed to determine which solutions that are feasible.

A coarse schematics has been developed and a number of prototype boards have been built. These prototypes serve as a guide to which theoretical presumptions that meet specifications and which have to be reworked. After thorough tests in a power laboratory the prototype is now ready for some clinical tests. At this point medical expertise is required in order to evaluate acquired results.

If the results are satisfactory the development process moves forward to a different route which unfortunately is out of the scope for this master thesis. This route includes cost and weight reduction, medical certification, further clinical studies among other things.

3 Defibrillators and LUCAS

A defibrillator is a medical device that, in layman terms, resets the heart of a person with ventricular fibrillation[2]. This is done by sending large amounts of electrical energy through the heart.

Defibrillation is normally carried out by placing two pads on the chest of the patient, thus allowing a path for the current leading through the heart. In LUCAS, one of the defibrillation pads will most likely be placed in the moving coil and the other on the backplate. This provide a direct path through the heart and very small amounts of current will go around the heart. This should be compared with commercial stand alone defibrillators, where much of the current flows on the skin and small amounts of the totally applied current flows through the heart.

3.1 Different types of defibrillators

Defibrillators today use a large capacitor (around 200 μF) which is charged to high voltages (around 2 kV). This energy is applied across the patients heart as described in the previous section. Depending of how the discharge of the capacitor is done, different types of defibrillation are achieved:

- Monophasic defibrillation - the current never changes directions during discharge.
- Biphasic defibrillation - the current changes directions after approximately half the time of discharge.
- Triphasic defibrillation - the current changes directions twice after usually one third and two thirds of the time of discharge.

The second, biphasic defibrillation, is becoming increasingly common since clinical studies has showed that the total amount of energy needed to defibrillate is lower in this case [3]. A lower amount of energy means that the defibrillators can be smaller since the size of the capacitor and the power supply (battery) can be greatly reduced. Since LUCAS is supposed to be portable, it is essential that the increase in weight due to an added defibrillator is kept at a minimum; thus choosing the biphasic defibrillator is a natural step. Triphasic defibrillation is still a field of research, but studies has shown that it should be possible to further lower the required energy without a decrease in success rate when using triphasic defibrillation compared to biphasic defibrillation [11].

There are different ways of controlling the discharge in biphasic defibrillators. All methods described below are assumed to be biphasic, even if it is not mentioned, i.e. the current changes directions after about half the time of discharge. For a discharge, there are three main parameters that can be controlled, *voltage* (V), *current* (I) and *time* (T). It is also possible to control the total energy discharged, but since this is limited by the charging voltage and capacitance of the capacitor this is rarely an option — complete discharge of the capacitor is most common. However, in many commercial defibrillators the energy is the only parameter the user can set, but this just means that the defibrillator controls one or several of the parameters above.

---

[2] Irregular heartbeats

The easiest way to control the defibrillation is to control the time. This is done by simply charging the capacitor to the desired voltage and then discharging it directly through the patient creating an RC-circuit with its characteristic discharging behavior. One problem with this method is that the resistance of the patient has to be measured prior to defibrillation, however this is not a major problem since most defibrillators usually have some kind of protection from short circuit and open air discharge which means that measuring the resistance prior to defibrillation in most cases is a must. The major problem is instead the high current at the start of the defibrillation since high currents might result in burn-marks on the patient.

Voltage controlled defibrillation is not considered a good idea. If the voltage was to be controlled, one has to charge the capacitor to a voltage much higher than the one desired in the discharge since the voltage delivered from the capacitor will decrease as the discharge commences. This could be solved by constantly charging the capacitor from the power source, but this would result in a transformer that is capable of delivering large amounts of current on the secondary side which in turn would render the capacitance obsolete leaving the defibrillator very large and heavy due to the overly dimensioned transformer and power source.

Current controlled defibrillation is becoming increasingly common. The current is allowed to swing between preset values and if the amount of energy stored in the capacitor is sufficient, both phases of the biphasic defibrillation can be made current controlled. Current controlled defibrillation is an area under research and is believed to be the future choice.[9]

3.2 The defibrillator in LUCAS

The defibrillator to be placed in LUCAS has to be very energy efficient and has to have low weight, small size and low cost. Factors like ease of use and the possibility to alter currents, voltages and discharge times has to be considered.

3.2.1 Energy efficiency

The energy efficiency of the defibrillator is essential since this will reduce the size of the required battery. With low energy efficiency it is likely that power dissipation in the device will cause the device to heat up and overly dimensioned heat sinks might be needed.

The entire design of the LUCAS defibrillator need to be imbued with energy efficient thoughts. The switching losses need to be kept at a minimum and great care has to be taken regarding energy consumption when choosing the components.

3.2.2 Weight, size and cost

As mentioned, the weight of LUCAS must not be significantly increased by the integration of a defibrillator. This means that the final product must be optimized when it comes to weight and size. However, the weight issue is somewhat out of the scope of this masters thesis since the battery and capacitor are the main contributers to the increased weight.

The size of the defibrillator can, and should, be optimized as early as possible in the design process. This is accomplished by minimizing the footprint[3] of the defibrillator. Components need to be chosen as small as possible and a tight schematic layout is of great importance. However, at this stage, trade-offs have to be considered, small and tight components are (usually) more expensive than larger ones and a certain amount of common sense need be used in order to optimize size/cost. Additionally, the voltage level is rather high so isolation distances need to be considered in order to achieve reliable and safe operation.

3.2.3 Variable variables

Since the concept of defibrillation through the human body (chest to back) is not as well documented as normal defibrillation (chest to chest) much research is needed before a suitable configuration can be found. This means that the discharging and charging options should be maximized. The defibrillator in LUCAS should be a biphasic defibrillator, but should it be possible to defibrillate with monophasic and triphasic waveforms? Current controlled defibrillation should be considered, but how much current is needed when defibrillation is performed through the body? These parameters and many more need to be investigated before the device is taken into production. For this reason the outcome of this masters thesis need to be a prototype providing freedom to set these variables so that medical personnel can perform clinical tests and decide the features of the final product.

The possibility to vary the following parameters should be considered in a prototype:

- Monophasic-/biphasic-/triphasic defibrillation
- Discharge current/-voltage
- Current control on/off
- Time of discharge
- Idle time[4]

---

[3] Size on PCB

[4] Time between the positive and negative defibrillation pulse

4 Voltage transformation

4.1 Energy and voltage levels

A crucial part of the defibrillating system is that a high voltage source is needed. Since LUCAS is a portable unit, the highest voltage available is 12 *volts* from the built-in battery. This voltage level is not sufficient to drive any larger currents into the human body since clinical measurements show that 95% of the human population have a body resistance in the range of 30 to 90 Ω [9].

Hence, the idea is to use a transforming circuit boosting the battery voltage to a defined voltage level which is then stored in a high voltage capacitor. The capacitor, which acts like a fuel tank during the defibrillation sequence, has a stored energy level which can be calculated using the formula

$$W = \frac{C \cdot U^2}{2} \quad Joule \tag{1}$$

In many commercial defibrillation systems the energy level is the predominant, and sometimes the only setting, that the user can alter. These energy levels are usually predefined in the range of 200 to 360 *Joule*. In order to store this kind of energy, the market for high voltage capacitors was investigated. There are a few capacitors that are specifically intended for use in defibrillation systems, and a capacitor with the electrical characteristics 196 μF and 3 kV was selected. If a maximum energy level of 360 *Joule* is desired, the capacitors maximum voltage should at least exceed 1.9 kV according to (1).

Creating this voltage is possible, but care has to be taken that the components of the charging and discharging circuits can cope with such high voltage levels.

4.2 The IGBT and the MOSFET

High power switching applications has not always been easy to perform without high losses in switching. Until recently, (1980's) BJT's[5] where used since they have lower power dissipation compared with the faster MOSFET[6]. In 1984 [6] the IGBT[7] was introduced combining the best of two worlds. The IGBT has the fast switching of the MOSFET and the lower on state losses of the BJT. The IGBT allows high current, high voltage switching with low losses.

The IGBT, just like the MOSFET, is voltage controlled, i.e. there is a capacitive coupling between the gate and the emitter (C_{ge}) and between gate and collector (C_{gc}) of the transistor. The total gate capacitance is the sum of C_{gc} and C_{ge} and will hereafter be refered to as the gate capacitance C_g. This means that the switching time is proportional to the equivalent capacitance of the gate ($C_g = C_{gc} + C_{gc}$) and the resistance connected to the gate (R_G). The schematic symbol of the IGBT is shown in Figure 2 [10].

Since high speed switching may cause voltage and current spikes through the transistor the feature with R_g is very useful. The same problem with spikes arise when the transistor is turned off, which is solved by the same resistor. The

---

[5] Bipolar Junction Transistor

[6] Metal Oxide Semi Conductor

[7] Insulated Gate Bipolar Transistor

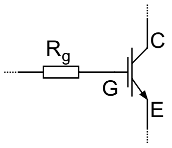

Figure 2: Schematic symbol of the IGBT with gate resistor R_g

rise time of the voltage over the gate-emitter capacitor is calculated as:

$$\tau = R_g \cdot (C_{gc} + C_{gc}) = R_g \cdot C_g \tag{2}$$

Equation 2 shows that a larger R_g will increase the rise time of the gate-emitter voltage and thus lowering the voltage and current spikes due to switching. However, R_g need to be chosen carefully since a long rise time causes the transistor to pass through its active region slowly, which will cause higher switching losses.

4.3 Converter topologies

Considering the voltage transformation there are a vast number of available converter topologies. The application that the converter is going to be used in determines which topology that is most suited in the design. In this case the flyback converter, which is derived from the buck converter, is examined.

The flyback converter is attractive due to the fact that it provides current control and isolation in one conversion stage. The galvanic isolation is a key factor by itself since the voltage used to perform a defibrillation, as shown above, well exceeds $1kV$ and thus it is important to protect the low voltage control electronics from damages related to over-voltage.

The current control feature also plays an important role. First of all it is vital not to saturate the transformer core (more in Section 4.4.2) since this will degrade the performance or even worse, damage the windings. Secondly it is necessary not to draw energy from the battery too quickly since this could shorten the battery life time or even damage the battery cells.

4.4 The flyback converter

This section gives a brief introduction to the flyback converter starting with an analysis of the primary side.

4.4.1 Primary side of the flyback converter

Considering Figure 3, it is possible to determine the voltage drop across, and the current through the primary winding of the transformer (when the transistor is on).

The voltage drop across the primary winding of the transformer is easily described by Equation 3.

$$L\frac{di}{dt} = V_{bat} - V_{ds} - R_L i, \tag{3}$$

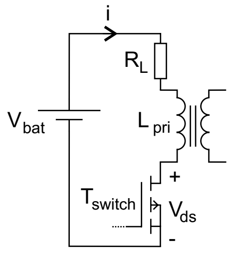

Figure 3: Primary side of the flyback converter

where R_L is the parasitic resistance of the coil and V_{ds} is the voltage drop across the transistor. From Equation 3 the current i can be calculated:

$$\begin{cases} i(0) = 0, \\ L\frac{di}{dt} + R_L i = V_{bat} - V_{ds} \Leftrightarrow \frac{d}{dt}\left(ie^{\frac{R_L}{L}t}\right) = \frac{V_{bat}-V_{ds}}{L}e^{\frac{R_L}{L}t} \quad \Leftrightarrow \end{cases}$$
$$i(t) = \frac{V_{bat}-V_{ds}}{R_L}\left(1 - e^{-\frac{R_L}{L}t}\right) \tag{4}$$

In (4) it is evident that the current through the winding increases exponentially until it reaches it's maximum value calculated from:

$$i_{max} = \lim_{t\to\infty}\left(\frac{V_{bat} - V_{ds}}{R_L}\left(1 - e^{-\frac{R_L}{L}t}\right)\right) = \frac{V_{bat} - V_{ds}}{R_L} \tag{5}$$

The transformer specifications limit the maximum peak current to 5 A. Using (5) with a voltage drop V_{ds} across the transistor of 2 V and an inductance of the primary winding equal to 10 μH gives a maximum current of 1 kA ($R_L = 0.2\ \Omega$). This is of course if a duty cycle of 100% is applied. However, the equations give a clue of what duty cycle to use. Using (4) to calculate t results in

$$i(t) = \frac{V_{bat} - V_{ds}}{R_L}\left(1 - e^{-\frac{R_L}{L}t}\right) \Leftrightarrow t = -\frac{L}{R_L}ln\left(1 - \frac{R_L i(t)}{V_{bat} - V_{ds}}\right) \tag{6}$$

With values from the transformer and transistor data sheets ($L = 24.5\ \mu H$, $R_L = 0.2\ \Omega$ and $R_{ds} = V_{ds}/I = 0.4\ \Omega$) the maximum time that the transistor should be on is calculated to 12.9 μs. With a switching frequency of 16 kHz this would correspond to a duty cycle of about 20 %. This is an estimate of what duty cycle to use but the modell used to derive the duty cycle is insufficient. Firstly, the primary winding has been considered to be an almost ideal coil. The resistance of the coil has been considered but the fact that the secondary side will be the load of the primary side has been ignored. Furthermore, it was assumed that there were no parasitic inductances (stray inductances) in the circuit, i.e. all wires were ideal. But one of the most incorrect assumptions made, when this modell was derived, was the switching of the transistor. The transistor is assumed to have ideal switching behaviour, that is, the rise and fall times are zero. The assumptions above are considered to be common practice.

With this brief discussion it is clear that one factor is essential when it comes to driving a current through the primary side of the transformer, short rise and fall times of the transistor (di/dt large). Furthermore, leakage inductances from the transformer (L_λ) and stray inductances from the wires (L_δ) will cause the current through the winding to rise more slowly. Hence two factors need to be considered:

- Short rise and fall times of the transistor (di/dt large).
- Minimize leakage inductances in the circuit.

These two factors control different parts of the current pulse. In Figure 4 below the basic behavior of the current pulse can be seen.

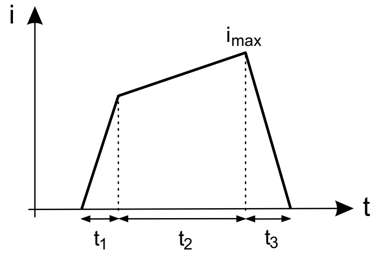

Figure 4: Current pulse

The different times in Figure 4, t_1, t_2 and t_3, are finite due to the factors mentioned above. The times t_1 and t_3 are due to the switching time of the transistor and t_2 is due to the inductances of the circuit, L_δ, L_λ but mainly the main inductor L_{pri}.

When it comes to minimizing the switching times of the current (t_1 and t_3) there are not many factors that can be altered. The main thing that can be controlled is the gate resistance of the switching transistor (see Section 4.2).

The PWM[8] signal controlling the switching of the switching transformer has its origin in a ATMega8 processor from Atmel. This means that the PWM signal is TTL logic (alternating between 0 and 5 V). The output of the ATMega8 is not strong enough to deliver sufficient amounts of energy to charge the gate of the MOSFET. This is solved by placing a small driver circuit between the TTL signal and the MOSFET, Figure 5. The driver circuit is a push-pull circuit where a combination of PNP and NPN BJT transistors is used. When the PWM signal is set (5 V), the NPN transistor (T_1) saturates and provides a current path allowing the gate of the MOSFET (T_3) to be charged via the resistor R_g. If the voltage drop across the NPN is neglected, the gate of the MOSFET will be equivalent to a RC-circuit with R_g and the gate capacitance (C_g) of the MOSFET. Therefore, the MOSFET gate is charged directly from the 12 V supply according to Figure 2. When the PWM signal is low (0 V) the NPN transistor is turned off and the PNP transistor ((T_2) is turned on since the voltage of the gate of the PNP is lower than that of the emitter ($V_{be,PNP} < 0$ due to the charged capacitor C_{gs}). This allows the gate to discharge in a controlled manner via R_g to ground.

---

[8] Pulse Width Modulation

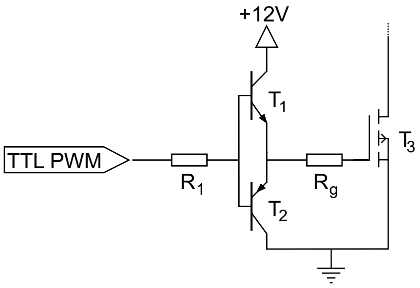

Figure 5: MOSFET driver circuit

The resistor R_1 in Figure 5 provides the PWM output of ATMega8 with a resistive load reducing the power consumption of the processor. If this resistor is neglected the ATMega8 would have to cope with the complete voltage drop ($V_{TTL} - V_{be}$) directly on the PWM output. This is an important factor since this is a common problem when a signal is transformed from TTL levels to analogue voltages.

4.4.2 The transformer

The initial thought regarding this project was to manufacture the transformer for voltage transformation by hand. However this became a bit too much to handle since the transformer is extremely sensitive and complex. This section intends to describe why.

Figure 6 below describes a simplified schematic view of a transformer. For a start, assume that the transformer in Figure 6 is supplied with an ideal sinusoidal input voltage V_p and that there are no losses in the transformer core. Furthermore, the permeability of the core (μ) is infinite, and all windings are lossless. If this is true, then the magnetic flux (ϕ) in the core will be the same both on the primary and on the secondary side of the transformer. Faraday's law states that an alternating magnetic flux induces an EMF[9], (e_p and e_s) according to (7)[4].

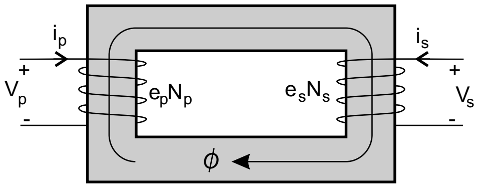

Figure 6: Ideal iron core transformer

$$e_p = N_p \frac{d\phi}{dt}$$
$$e_s = N_s \frac{d\phi}{dt} \tag{7}$$

---

[9] Electro Motive Force

Since the magnetic flux is the same on both sides, the time derivative of the magnetic flux has to be the same too. Additionally the induced EMF should, ideally, be equal on both sides of the transformer. This makes it possible to rewrite Equation 7.

$$\frac{e_p}{N_p} = \frac{e_s}{N_s} \Leftrightarrow \frac{V_p}{V_s} = \frac{N_p}{N_s} = \frac{e_p}{e_s} \Leftrightarrow V_s = V_p \frac{N_s}{N_p} \tag{8}$$

Sadly, real transformers are not ideal. In a real transformer there are leakage inductances, losses in the core, parasitic resistors and the magnetic permeability, μ, is far from infinity. A more accurate way to describe the transformer behavior with a non ideal transformer ($\mu < \infty$) is using Equation 9,

$$N_p i_p - N_s i_s = \frac{l}{\mu S}\phi, \tag{9}$$

where l is the length of the ferromagnetic core and S is the cross-sectional area of the core [4].

When developing a transformer one has to take several factors into account:

- Core material, reducing eddy currents
- Size of the core
- Wire thickness
- Isolation voltage
- Efficiency of the transformer

One of the most important losses in a transformer core are the losses due to eddy currents or Focault[10] currents. Eddy currents are local currents induced by the magnetizing flux and flow in the normal direction of the flux. These currents produce ohmic power loss and cause local heating of the core. The losses due to eddy currents can be reduced by using core materials with high permeability (μ) but low conductivity (σ). For low-frequency applications the most common way to reduce the currents is by using laminated cores. The laminated cores are put together with an electrically isolating compound forcing the eddy currents to propagate in each 'sheet' of the core. As the frequency of the eddy currents rise, the laminated cores behaves more and more like a solid core. For high frequency applications a ferrite core is a better solution. The ferrite material can be described as made up of very small balls, where each ball is electrically and magnetically isolated from its neighbors minimizing the area for the eddy currents to propagate in.

The size of the core is essential since this determines the amount of magnetic flux the core can hold without saturation. When a magnetic field intensity H is applied to a ferromagnetic material a resulting magnetic field density, B, arises in the material according to Equation 10.

$$B = \mu H \tag{10}$$

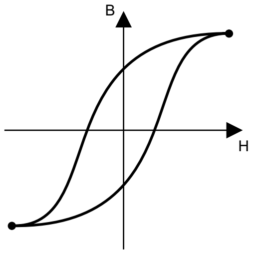

Figure 7: Hysteresis loop in the B-H plane for a ferromagnetic material

However, Equation 10 is not linear due to the fact that μ changes with the field intensity. This is due to remanent flux and saturation of the core, illustrated in Figure 7.

As seen in Figure 7 the magnetic flux density does not return to zero when the applied magnetic field intensity is zero. The remaining flux density in the material is called the residual or remanent flux density. It is also evident that a large applied magnetic field intensity will drive the core into saturation. If the core is saturated, its magnetization does not increase with an increase in applied magnetic field intensity.

Wire thickness of the primary and secondary windings of the transformer need to be considered in order for the transformer to work as intended. If the thickness of the wires is too thin, the parasitic resistance and inductance of the wires will increase. If the wires are much too thin they might not be able to withstand the necessary currents. If, on the other hand, the wires where to be chosen too large, the efficiency of the transformer will decrease (if the transformer is not chosen unnecessarily large). This is due too the decrease of the effective winding area, (Equation 11).

$$Effective\ winding\ area = \frac{\sum I_e}{A_e} \leq 1 \tag{11}$$

The variables I_e and A_e are illustrated in Figure 8 below.

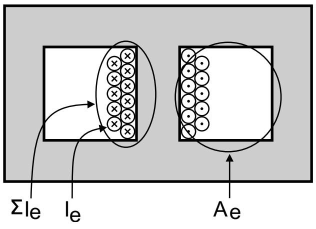

Figure 8: Cross section of a transformer, illustrating I_e and A_e

Using Equation 11 and Figure 8, one can see that a thicker wire will leave larger air gaps between the windings, causing the effective copper fill factor to

---

[10] Jean Benard Leon Focault, 1819-1868, French Scientist who proved the existence of eddy currents

decrease since the number of turns has to be decreased in order for the windings to fit in the available winding window A_e. This could be solved by using Litz wire or polarized wires (square wires). The inductance of a winding n of a transformer can be written as:

$$L = N_n^2 \frac{\sum I_e}{A_e} \mu_0 \mu_r \tag{12}$$

The voltage drop over a coil with inductance L can be written as:

$$V_L = L \frac{di_L}{dt} \tag{13}$$

Combining Equation 12 with Equation 13 result in Equation 14 [5].

$$V_L = N_n^2 \frac{\sum I_e}{A_e} \mu_0 \mu_r \frac{di_L}{dt} \tag{14}$$

The voltage drop across the transformer should be maximized, which means that the effective winding quota, $\sum I_e / A_e$, should be as close to one as possible. This also clarifies the need of accurate winding if the winding is done by hand. The wires should be as closely packed as possible and ideally no empty space should be available in the winding window.

A good transformer also provides galvanic isolation between the primary and the secondary winding. This isolation has to be taken into account when choosing wires. Since many cores have both the secondary and primary winding in the same winding window it might be necessary to add an isolating sheet between the windings, even though this will reduce the effective winding quota. The isolation voltage also need to be experimentally verified before the transformer can be used in good faith.

All the parameters above affect the efficiency of the transformer. Neglecting to optimize any of them will result in increased leakage inductances and resistance, thus making the transformer ill fit for medical applications. Like mentioned in the beginning of this section, the initial thought of the project was to engineer the transformer by hand. However, this resulted in very poor performance due to high leakage inductance and many headaches due to the breaking of the thin secondary winding wire. The process of manufacturing a transformer can be seen in Figure 9.

In Figure 9 the roll of wire and the bobbin (circled) of the transformer is connected to a turning lathe providing a reasonable rotation of approximately 30 *rpm*. One person had to gently steer the wire correctly whilst the other counted the number of windings. All the effort resulted in the ordering of a custom made transformer from Profec AB. It should be mentioned that the hand made transformer might have been more successful today when the knowledge and know-how regarding transformer design has increased.

4.4.3 Secondary side of the flyback converter

Since the voltage transfered from the primary side has the same frequency as the PWM signal (there might be differences in harmonics), the output voltage of the transformer, V_s, has to be rectified, Figure 10.

The secondary side of the flyback transformer is fairly simple. As explained in Section 4.4.2 the secondary side of the transformer will, ideally, only induce

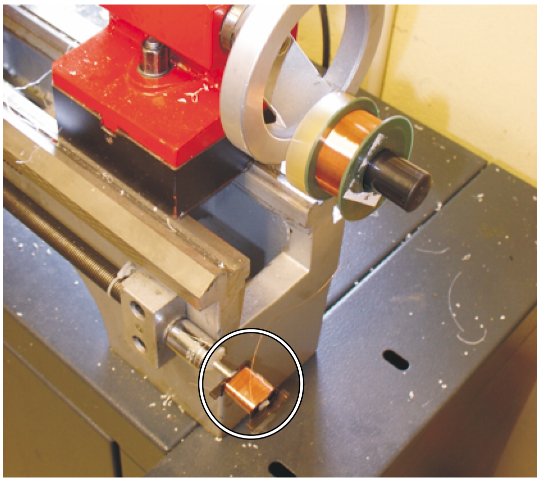

Figure 9: Manufacturing a transformer by hand.

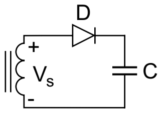

Figure 10: Simplified secondary side of the flyback transformer

current during the time the primary side of the transformer is off. What happens is that the magnetically charged core of the transformer induces a current in the secondary winding. This current is used to charge the main capacitor C. When the voltage of the capacitor C rises, the diode D will block the current, forcing the voltage V_s to build up (Lentz' law) until it is larger than that of the main capacitor plus the voltage drop across the diode, thus increasing the voltage of the main capacitor additionally. The higher the voltage of the main capacitor becomes, the more energy needs to be transfered via the transformer in order for the voltage to build up. This means that the PWM signal, switching the transistor of the primary side, needs to be changed as the voltage rises. When the voltage of the main capacitor is low, very little energy is needed to increase the voltage significantly, this means that a very short duty cycle of the PWM is sufficient at the start of the charging but as the voltage rises, the duty cycle needs to be increased.

The most important component on the secondary side is the diode D. This diode has to be fast enough in order to follow the switching frequency of the PWM on the primary side and it has to be able to block very high voltages, since the entire voltage of the capacitor must be blocked by the reverse biased diode.

Another important characteristic of the diode is the reverse recovery current.

The reverse recovery current is the current that flows through the diode during the time it takes for the diode to switch from conducting to blocking. The faster the diode, the higher (but shorter) the peak of the reverse recovery. This very high current will give rise to a voltage drop across the diode which might, if it is high enough, cause the diode to break. However, there are solutions to the problem, a snubber circuit could limit the effect of the reverse recovery current, Figure 11.

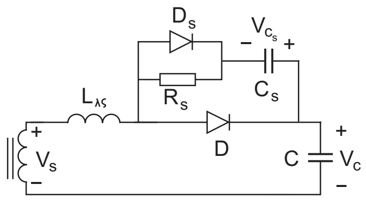

Figure 11: A turn off RCD snubber

The snubber in Figure 11 is a RCD (Resistor, Capacitor and Diode) snubber. It works during the turning off of the diode D. When V_s has dropped below V_c, D will, for a short period of time, conduct in its reverse direction (reverse recovery current). When the diode D blocks, the leakage and stray inductances on the secondary side will give rise to a voltage, $V_{L_{\lambda\delta}}$, that is proportional to the time derivative of the diode current. Without a snubber circuit, this voltage will rise to infinity and eventually break D since both the capacitor and the secondary side of the transformer are inert to sudden voltage changes. With the snubber circuit the alternative path is through C_s and R_s where R_s will limit the maximum peak current and thus the voltage across D. [6]. It is essential that the values of R_s and C_s are chosen carefully. The snubber should only be active during the switching on / switching off time of the diode D. [6] Another important thing to remember is the fact that the snubber capacitance must be able to hold the entire voltage of the main capacitor C plus the voltage of the primary side multiplied by the winding quota, Equation 15. This makes the selection of possible capacitors fairly limited.

$$V_{C_s} = V_C + V_{cc,primary} \cdot \frac{N_s}{N_p} \tag{15}$$

4.5 Controlling the charging

In order for the charging to work satisfactory, certain factors need to be controlled. The most important factor is the voltage on the secondary side i.e. the voltage of the capacitor. This voltage needs to be measured by the ATMega8 processor and compared to the reference value. If the voltage level is sufficient, the charging should seize. Since the high voltage of the secondary side of the transformer is galvanically isolated from the primary side and control electronics, the voltage of the main capacitor needs to be measured galvanically isolated. The voltage is measured like shown in Figure 12.

There are some important features to remember when studying Figure 12. The resistor R_{vm} need to be chosen very large in order for the capacitor C not

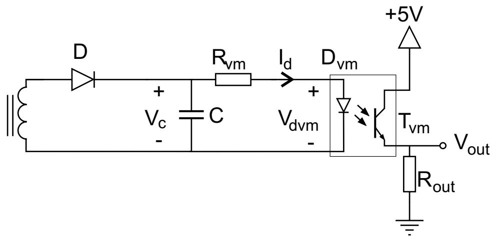

Figure 12: Measuring the voltage of the capacitor

to discharge via this resistor. But, if R_{vm} is chosen too large, the current I_d will not be able to drive the diode D_{vm}. The optocoupler, consisting of D_{vm} and T_{vm}, should have a linear transfer function. If it has, the voltage on the output, V_{out} can be written as Equation 16.

$$V_{out} = I_d \beta R_{out} = \frac{V_c - V_{D_{vm}}}{R_{vm}} \beta R_{out} \approx \frac{V_c}{R_{vm}} \beta R_{out}, \tag{16}$$

where β ($<< 1$) is the transfer factor of the optocoupler. The output voltage, V_{out}, need to be in the range $0 < V_{out} < 5\ V$ in order for the ATMega8 to be able to A/D convert it properly. The A/D converter of the ATMega8 needs an input impedance of less than 10 $k\Omega$ in order to successfully perform an A/D conversion within reasonable time. This is due to the RC circuit on the input of the A/D converter working as a sample and hold circuit. The resistor R_{out} is much larger than 10 $k\Omega$, meaning that the A/D conversion will take unnecessary long time. Since it is very difficult to get the voltage V_{out} in the proper voltage range; an amplifier in series with V_{out} solves both the amplitude problem and the impedance problem. In Figure 13 the complete circuit for measuring the voltage is shown.

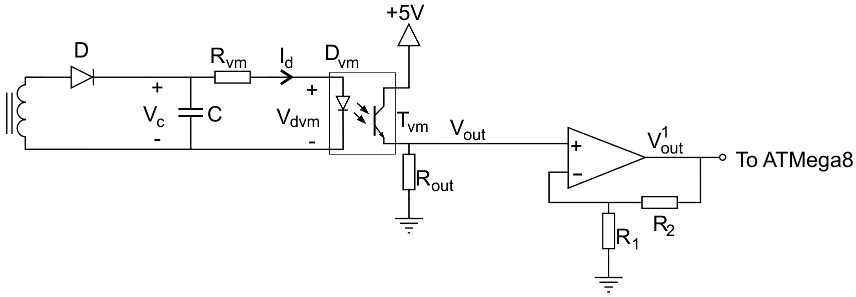

Figure 13: Voltage measurement with amplifier

In Figure 13, V_{out} is amplified according to Equation 17.

$$V_{out}^1 = \left(1 + \frac{R_2}{R_1}\right) V_{out} \tag{17}$$

This means that the voltage to be A/D converted, V_{out}^1, can, using Equation 16 and Equation 17, be written as:

$$V_{out}^1 \approx \left(1 + \frac{R_2}{R_1}\right) \frac{V_c}{R_{vm}} \beta R_{out} \tag{18}$$

The factor between V_{out}^1 and V_c can be controlled by changing the factor R_2/R_1.

It was mentioned in Section 4.4.1 that the peak current through the transformer must not be larger than 5 A. The highest current is on the primary side of the transformer which means that this is where problems might occur. The current on the primary side is most easily determined using a very small ($< 1\ \Omega$) in series resistor and measuring the voltage drop over this. The resistor can be fitted in two places, either before the switching transistor (Figure 14 a) or after (Figure 14 b).

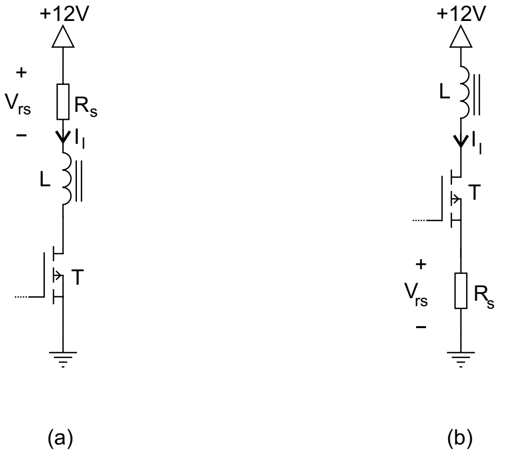

Figure 14: Different placements of the current measurement resistor

If the resistor is positioned like shown in Figure 14 a, the voltage drop over the resistor will be in the interval $R_s I_L < V_{R_s} < 12$. This means that the voltage drop over R_s needs to be measured with a differential amplifier or instrument amplifier. The other placement of R_s, Figure 14 b, will result in a voltage V_{R_s} in the range of $0 < V_{R_s} < R_s I_L$. This voltage can be directly A/D converted meaning that no extra components need to be added, resulting in a smaller footprint.

There are some problems when it comes to measuring the current in a circuit like this. There will only be a current flowing during the time when the PWM pulse is high and since the PWM pulse is very short the ATMega8 can not trigger on this pulse to start the A/D conversion. This problem could be solved by using a stand alone hold circuit, see Figure 15.

An important problem with the circuit in Figure 15 is the switching transistor T_s. This transistor needs to be fully conducting and blocking in order for this circuit to work, hence the push pull circuit connected to the gate of T_s. Another factor that disrupts the use of the circuit in (15) is the capacitor. The capacitor will present a mean value of the current flowing through the primary side during the time the main swithing transistor is on. This is not the desired

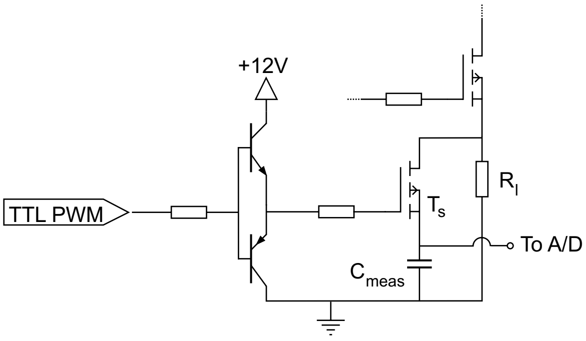

Figure 15: Voltage hold circuit

behavior, much uncertainty will be introduced when estimating the peak value of the current from the mean value.

Another approach is to measure the current using a simple peak hold circuit shown in Figure 16.

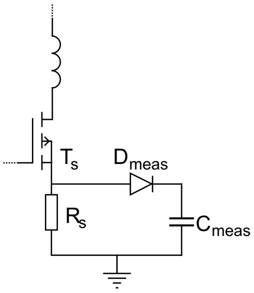

Figure 16: Peak hold circuit

The peak hold circuit has one crucial component, the diode, D_{meas}. This diode has to be fast in order for it to block the negative spikes that arise due to stray inductances in the main circuit. Furthermore, the forward voltage drop across the diode has to be much smaller than the voltage to be measured. The latter problem can be solved by using the circuit in Figure 17.

The output voltage across C_{meas} in Figure 17 should be the same as the voltage over R_s since the OP will compensate for the voltage drop across D_{meas}. In order for the output voltage to be less fluctuating, a low pass filter is introduced to filter the voltage across R_s with an RC-circuit, Figure 18.

This approach proved to be somewhat useful since the filter capacitor C_f removes the spikes in the measured voltage. Nevertheless, the output voltage across C_{meas} still could not be directly associated with the current through the transformer.

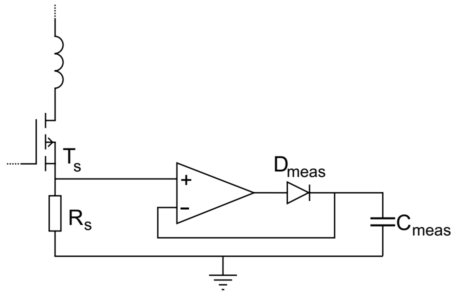

Figure 17: Peak hold circuit with amplifier

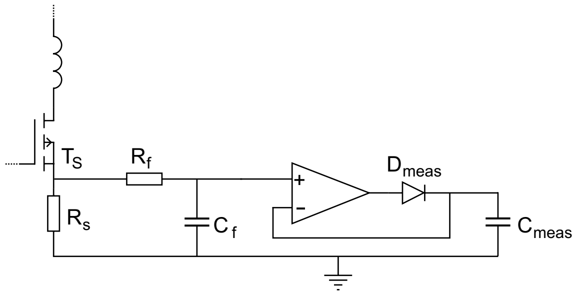

Figure 18: Peak hold circuit with amplifier and filter

4.6 Safety

When doing a voltage transformation like the one in this project it is always important to consider the safety of the circuit designed. In previous sections the galvanic and optical isolation is motivated by means of protecting the control electronics, however extra precautions need to be taken in order to protect the surrounding environment and the user. Important features are:

- Automatic discharge
- High voltage automatic shutdown

4.6.1 Automatic discharge

The automatic discharge is needed to avoid the main capacitor being left at high voltages when not operated. For instance, if the capacitor is charged and the medical personnel finds it unnecessary to complete the defibrillation, the capacitor should be discharged without connecting the load (the patient). This protection is most easily achieved by letting the ATMega8 monitor the time passed since the charging stopped. When the maximum idle time is achieved, the ATMega8 produces a control signal that is used to discharge the main capacitor. However, this raises some new problems. The idea is to use the control signal to drive a transistor, forcing the main capacitor to discharge via a large resistor, Figure 19. Since the circuit in Figure 19 disrupts the whole idea of electric isolation between high voltage and control electronics, the control signal needs

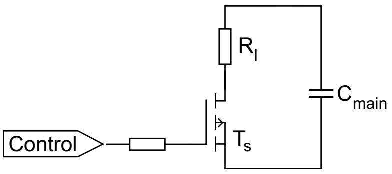

Figure 19: Basic circuit for the safety discharge

to be transfered to the high voltage side without electrical connections. As described in previous sections, this can be done either by an optocoupler or a transformer. The first alternative, the optocoupler, transfers DC signals in a straight forward manner, but it requires power supply on the secondary side. This makes it complicated to use since the voltage level on the secondary side is varying. The latter alternative, the transformer, can not transfer DC signals but does not require any power supply on the secondary side. The problem with DC signals can be solved by modulating the control signal with a high frequency signal (carrier). This is achieved by using a fast AND gate with the control signal and the high frequency signal as inputs, Figure 20.

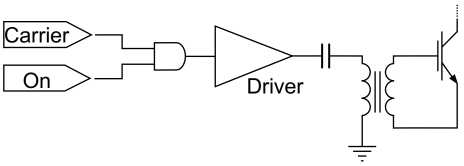

Figure 20: Simplified galvanic isolation between discharge transistor and control electronics

The output of the AND gate is too weak to be able to deliver the necessary amount of current to drive the coil. Consequently, a driver circuit is connected in series with the AND gate. The capacitor in series with the driver and the transformer removes the DC component of the signal avoiding saturation of the transformer. However, the carrier is still present on the secondary side of the transformer and needs to be removed. If it is not removed, the discharge transistor will try to switch with the carrier frequency. The carrier signal is removed by rectifying the signal on the secondary side, Figure 21.

As shown in Figure 21, the transformer chosen for the task has a center tapping that creates the reference voltage on the secondary side. If a transformer without center tapping is to be used, a complete rectifying bridge could be used instead of the two diodes D_1 and D_2. The third diode, D_3, is used in order for the PNP transistor T_1 to start conducting when the control signal is turned off. One important thing to keep in mind is that the voltage drop across D_3 has to be larger than the voltage drop V_{be} of the transistor. If this is not considered, the PNP transistor will not start conducting when the control signal is turned off. However, the PNP transistor gives the ability to choose the gate resistors, R_{off} and R_{on}, individually (see Section 4.2). The load resistor, R_{pd}, is needed

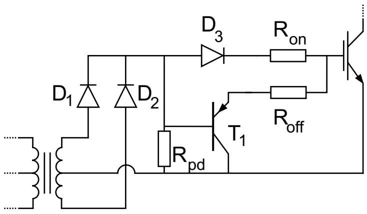

Figure 21: Rectifying the current

to create a well defined voltage on the base of the PNP transistor.

4.6.2 Over voltage shutdown

The second safety issued that need to be addressed is the overvoltage automatic shutdown. This is implemented in order to protect the user and electronics if the voltage on the secondary side becomes too high. The voltage on the secondary side could keep rising if, for instance, the ATMega8 hangs up and does not stop the PWM or if the voltage measurement circuit is faulty. This means that the voltage protection circuit needs to be made completely in hardware (if the ATMega8 hangs up). The idea is to measure the voltage on the secondary side and then, if the voltage is too high, transfer a signal to the primary side. The voltage measurement should not be done using the same hardware as described in Section 4.5, since this would make the entire design dependent on this single optocoupler. Another optocoupler needs to be introduced in the circuit, but some of the hardware from the normal voltage measurement can be used, Figure 22.

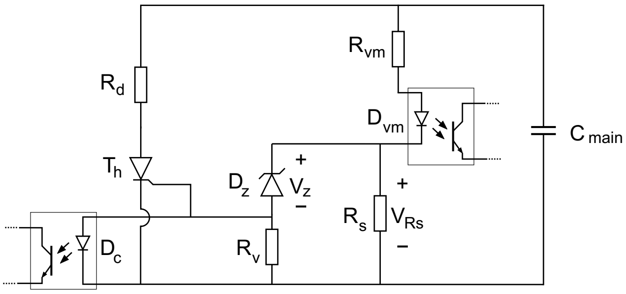

Figure 22: Circuit for over voltage protection

As seen in Figure 22, the resistor R_{vm} and the optocoupler diode D_{vm} are the components that make up the voltage measurement circuit. The voltage of the capacitor C_{main} will be divided between the resistors R_s and R_{vm} according to Equation 19.

$$V_{R_s} = \frac{R_s}{R_s + R_{vm}}(V_{C_{main}} - V_{D_{vm}}),\ V_{R_s} < V_z \tag{19}$$

If this voltage V_{R_s} is higher than the zener voltage, V_z, the diode will open in its reverse direction, allowing a current to create a voltage across the resistor

R_v. This voltage is the same as the driving voltage of the thyristor, T_H, which will cause this to open and discharging the capacitor C_{main} via R_d. Since the diode of the high voltage protection control signal optocoupler, D_c, is current driven, the current through D_c will not be linear but dependent on the voltage of the main capacitor according to Equation 20.

$$I_{D_c} = I_s \left(e^{\frac{V_{D_c}}{V_T}} - 1 \right), V_{D_c} = V_{R_s} - V_z, V_{R_s} > V_z \tag{20}$$

In Equation 20, V_T is the threshold voltage of D_c and I_s is the saturation current [1]. The voltage drop over the diode, V_{D_c}, is determined by the voltage drop over the resistor R_s which in turn is determined by the current through the diode when $V_{R_s} > V_z$, Equation 21.

$$V_{R_s} = I_{R_{vm}} - I_{D_c} = I_{R_{vm}} - I_s \left(e^{\frac{V_{R_s} - V_z}{V_T}} - 1 \right) \tag{21}$$

Where $I_{R_{vm}}$ is a function of the voltage drop across D_{vm}:

$$I_{R_{vm}} = \frac{V_{C_{main}} - V_{R_s} - V_{D_{vm}}}{R_{vm}} \tag{22}$$

Equation 21 is not very easy to solve since the voltage drop over D_{vm} is dependent of the current through the resistor R_{vm}. This makes it hard to determine what value of R_s to use, hence experimental verification will be needed.

The high voltage shutdown should have discrete levels, either on or off, and stay that way until the appropriate actions have been completed by the user. This will be a problem since the voltage of the main capacitor will drop, due to the discharging via R_d, causing the voltage over R_s to be lower than the zener voltage. Consequently, the current I_{D_c} falls to zero and D_z will block, thus returning the control signal to it's normal state. This can be solved by using a S/R latch on the secondary side, Figure 23. The S/R latch sets the output, *out*, depending on the levels of S and R combined with the previous state of the latch. For instance, if the latch is powered up with [S,R] equal to [1,1] the output is low (0 V). If this is followed by setting S low the output becomes high. Now the output remains high regardless of the signal level on S. This is the sought for behavior needed to solve the problem with declining I_{D_c} and is illustrated in Figure 24. In Figure 24, the R port is constantly high and the S port is controlled from the optocoupler. This means that when the optocoupler starts conducting, the S port is low, producing the inverse of the S port (high) on the output. The resistor R_{p_u} (Figure 24) is used as a

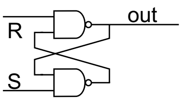

Figure 23: S/R latch

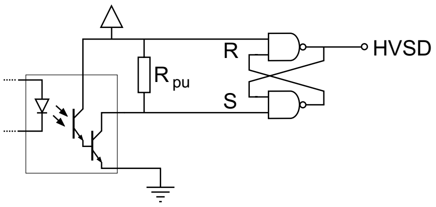

Figure 24: S/R latch connected to output of optocoupler

pull up resistor to ensure a known voltage level on the collector of the output transistor. It should also be mentioned that the output side of this specific optocoupler (Figure 24) consists of two transistors (Darlington connection) in order to provide extra gain, making it able to switch the output at very low input currents. The optocoupler also has a pin connected to the base of the output transistor, allowing the user to bias the transistor if it should work in its active region. However, the circuit in Figure 24 has one great disadvantage; once the output is set it is not to be lowered until the power of the circuit is toggled. This is not a good solution since the user should have the ability to reset the system without rebooting after a high voltage safety shutdown. If a reset switch is introduced in the schematic the problem is solved by the circuit in Figure 25, since the S/R latch returns to its original state if S is low at the same time as R is low. The switch SW in Figure 25 is connected to the base

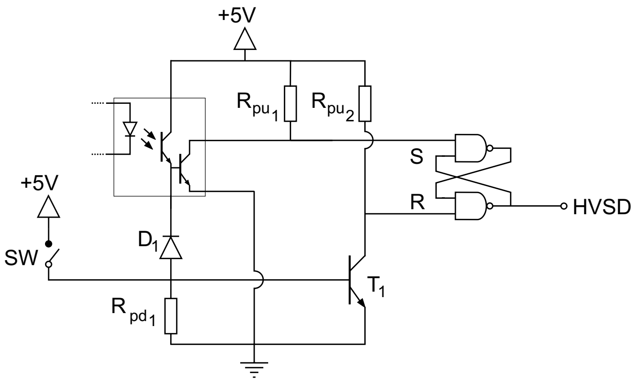

Figure 25: S/R latch with user reset

of T_1 and the base of the internal output transistor of the optocoupler. This means that when SW is closed, T_1 starts to conduct, setting the R port low. Simultaneously, the output transistor in the optocoupler will open, setting the S port low. The resistors $R_{pu_{1,2}}$ and R_{pd_1} are pull up / pull down resistors to ensure a well defined voltage level on the nodes they are connected to. The diode D_1 is used to avoid the base current of the output transistor from flowing via the pull down resistor R_{pd_1}. The circuit in Figure 25 produces an output signal, $HVSD$ (High Voltage Safety Discharge), that is high from the moment the voltage on the secondary side exceeds its maximum level, and stays high until the user toggles the switch SW.

When the voltage on the secondary side becomes too high, there are three things that need to be done on the primary side:

- Turn off the PWM
- Start the safety discharge
- Send interrupt to the ATMega8

The first item, turning off the PWM is achieved by adding a pull down transistor after the PWM series resistor, Figure 26. The resistor R_1 in Figure 26 was originally (Figure 5) chosen rather small ($< 200\ \Omega$) since it in the original circuit was used to allow the voltage level to drop, not to limit the current because the transistors themselves limit the base currents. This means that the resistor need to be increased in order for the PWM to have a reasonable load.

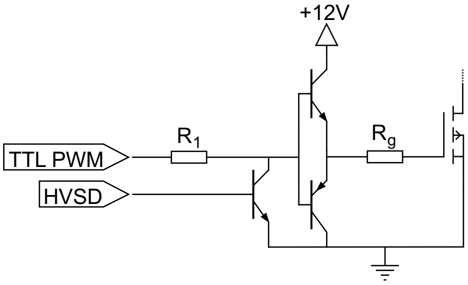

Figure 26: High voltage safety turn off of the PWM signal

Starting the high voltage safety discharge on the primary side is rather simple, Figure 27, and is achieved by adding a pull up transistor to the ON signal from the ATMega8.

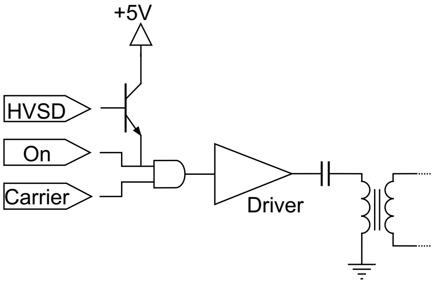

Figure 27: High voltage safety discharge on the primary side

The final task, sending an interrupt to the ATMega8, is simply a matter of connecting the $HVSD$ signal to an interrupt port of the ATMega8, allowing the software to handle the interrupt (if the processor is still running).

5 Discharge

As mentioned numerous times before the behavior of the defibrillator should during discharge (defibrillation) be biphasic. Applying the whole voltage of the main capacitor directly to the load (patient) would result in a current flowing constantly from anode to cathode of the capacitor until it was completely discharged. In this section it is described how the development of the discharging module was conducted and the underlying theory.

5.1 The four quadrant converter

The four quadrant converter (sometimes called H-bridge), Figure 28, is the main building block of the discharge module.

The converter allows the current to flow in both directions through the load depending on which transistors that are conducting. This means that the transistors should conduct according to the following list:

1. $T1$ and $T3$ are on
2. All transistors off
3. $T2$ and $T4$ are on
4. All transistors off

In the first stage the current i_l flows in its positive direction according to the definition in Figure 28. However this will lead to the charging of the inductive parts of the load (including wires and such) making it hard to change the direction of the current immediately. This explains the need of stage two where the current is presented with a path back to the capacitor via the freewheeling diodes mounted across T_2 and T_4 thus allowing the inductive load to discharge. This time is called idle time. Stage three means conducting the current in the opposite way compared to stage one, meaning that $i_l < 0$. The fourth and final stage is the same as stage two to avoid leaving the load inductively charged.

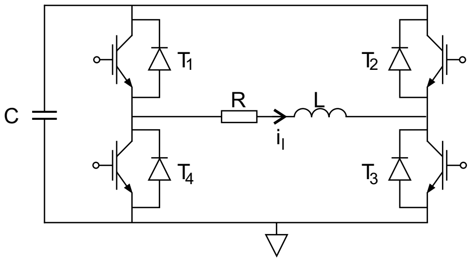

Figure 28: Four quadrant converter with RL load

One important detail regarding the converter is that the emitter potentials of T_1 and T_2 are dependent on the switch states of the transistors. This implies that the power supplies for the driver circuits of both transistors T_1 and T_2 must be galvanically separated from the one used for driving T_3 and T_4 [2]. The galvanic isolation is achieved in the same way as for the safety discharge, described in

Section 4.6.2. Since all transistors need to be controlled independently, four different circuits like the ones described in Figure 21 are needed.

5.2 Controlling the discharge

The need for some kind of controller for the discharge is evident. If one is not used (old defibrillators) the entire voltage of the capacitor will be applied across the load and the resistive and inductive parts of the load will be the only factors limiting the current. It was mentioned in earlier sections that current control is an important feature that should be implemented in the device.

In order to simplify the description of the current control circuit, the discussion will start assuming that monophasic behavior of the defibrillator is desired, Figure 29. In order for any current control to work properly the current through the load needs to be measured. Measuring the voltage drop across a series resistor would not be a good solution since the resistor would disspipate energy and the need for an extremely fast and accurate optocoupler would introduce much uncertainty in the measurement. Instead, a LEM module is used. The LEM module consist of a ferrite core coil with a Hall element, measuring the magnetic flux of the core, producing a voltage on the secondary side corresponding (linearly) to the current through the coil. The starting point is to turn T_1 and T_3 on in Figure 29 charging the inductance L. When the current, i_l reaches the highest value allowed, the transistors are turned off allowing the current to flow via the diode D_4, through the load and through D_2. As the current flows through the diodes, the energy stored in the inductance will decrease, thus reducing the current i_l until it reaches its minimum value allowing the turning on of T_1 and T_3 to restart the cycle. This type of controller is called a tolerance band controller since the current is allowed to vary in a band around the current reference, Figure 30. In Figure 30, the value i_{rv} is the reference value. The difference between the maximum value, i_{max}, and the minimum value i_{min} is referred to as the current ripple. The bandwidth of the controller is dependent on the size of the inductance L. A large coil would reduce the needed bandwidth but add to the size and weight of the device, whereas a small coil would increase the needed bandwidth with little increase in weight and size. It is evident that the size of the inductance is of great importance since the driver circuits and the IGBT's used in the discharging circuits have a limited bandwidth ($\approx 30\ kHz$), and one of the most important sought after features of the defibrillator is low

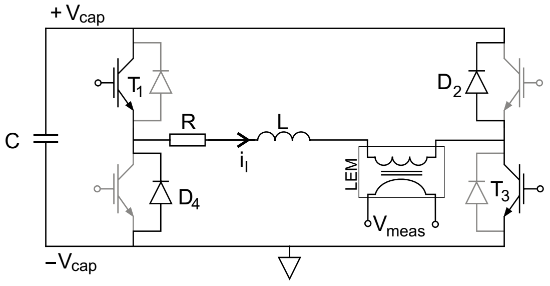

Figure 29: Circuit for monophasic discharge

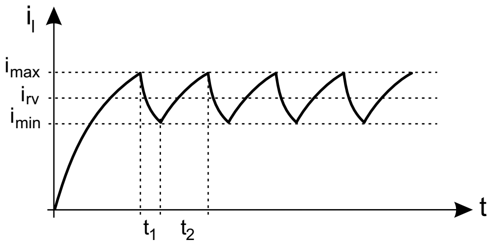

Figure 30: Current through the load when using tolerance band current control

weight.

The initial thought was to use the ATMega8 to A/D convert the measured voltage, V_{meas}, but the bandwidth of the current control loop needs to be much higher than that achievable using the ATMega8. The ATMega8 can be used to A/D convert at approximately 50 kHz, but this figure get radically reduced when actions are to be performed between the conversions. Knowing these limitations of the ATMega8 it was decided to implement a hardware tolerance band controller.

The hardware controller is made up of two building blocks, an adder and a comparator with a hysteresis that determine the current ripple, Figure 31. In Figure 31, the signal i_l is the current through the load measured using the LEM module and the signal i_{rv} is the reference value. The error e is calculated as $i_l - i_{rv}$ and is sent to the comparator.

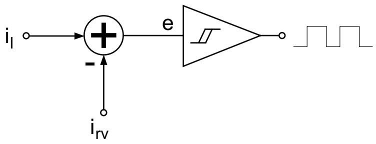

Figure 31: Block diagram of the tolerance band current controller

The adder is designed using an OP-amplifier with negative feedback, Figure 32. Note that the notations V_l and V_{rv} refer to voltages corresponding to the actual current through the load and the current reference value.

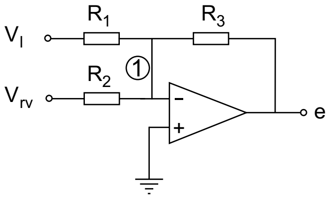

Figure 32: OP-amplifier adder

Applying Kirchoffs current law in node (1) the signal e can be described as:

$$\frac{0 - V_l}{R_1} + \frac{0 - V_{rv}}{R_2} + \frac{0 - e}{R_3} = 0 \Leftrightarrow e = -R_3 \left(\frac{V_l}{R_1} + \frac{V_{rv}}{R_2} \right) \tag{23}$$

Choosing the resistors $R_{1,2,3}$ equal gives:

$$e = -\left(V_l + V_{rv}\right) \tag{24}$$

From (24) it is clear that the reference value must be of opposite polarity compared to that of the current through the load. The adder produces the error e which is connected to the tolerance band controller.

The tolerance band controller is made up of an OP-amplifier with positive feedback, Figure 33. The feedback is positive since the output should be an alternating signal with the discrete levels $-V_{cc}$ and $+V_{cc}$. In order to derive an expression for the hysteresis of the controller a node analysis using Kirchoffs current law is performed in node (1):

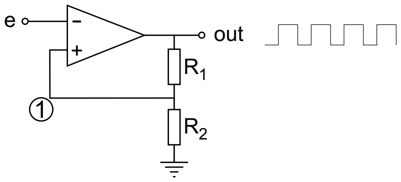

Figure 33: OP-amplifier tolerance band controller

$$\frac{e - 0}{R_2} + \frac{e - out}{R_1} = 0 \Leftrightarrow e = \frac{R_2}{R1 + R_2} \cdot out \tag{25}$$

Knowing that the positive feedback will produce an output equal to $\pm V_{cc}$ makes it possible to rewrite (25) as:

$$e = \begin{cases} \frac{R_2}{R1+R_2} \cdot V_{cc}, \ e > 0 \\ -\frac{R_2}{R1+R_2} \cdot V_{cc}, \ e < 0 \end{cases} \tag{26}$$

From (26) it can be seen that the output is of opposite polarity compared to the wanted. When the error is positive, i.e. the current through the load is too high, the output from the controller is high ($+V_{cc}$) and vice versa. This means that the output needs to be inverted before it is allowed to control the transistors. Another problem is that the output is $\pm V_{cc}$ ($\pm 12\ V$) when it should be TTL levels (0 or $+5\ V$). This is solved by simply inserting a diode in series with the output and placing two resistors as a voltage divider in order to lower the voltage, Figure 34.

When it comes to actually creating the reference value this is done simply by connecting the ATMega8 to a digital potentiometer with I^2C communication, Figure 35. By connecting the digital potentiometer in Figure 35 between GND and $-V_{cc}$, a negative reference value is created. Using the ATMega8 to control the digital potentiometer gives the ability to set the reference value in software, which allows the user the freedom to choose this value.

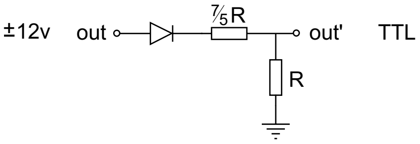

Figure 34: Conversion from ±12 V to TTL levels

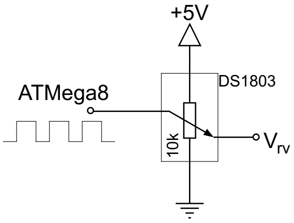

Figure 35: Creating the reference value

During the discussions above it has been assumed that monophasic defibrillation should be used. However, this is not the case in the final device; it should be possible to use both biphasic and triphasic defibrillation if the energy of the capacitor is high enough. Consequently, the current control should work for both positive and negative load currents. The controller above needs some minor modifications before this task can be fulfilled:

- Ability to change the sign of the reference value
- Switch between inverted and non-inverted output

The ability to change the sign of the reference value could be achieved by connecting the digital potentiometer between $+V_{cc}$ and $-V_{cc}$, which allows the ATMega8 to change the reference to its positive equivalent. This is not a good solution since this would set a lower limit of the idle time equal to the time it would take to change the reference value. Another, and better way is to use two reference values and using an analogue switch to select the desired value. This would introduce an extra digital potentiometer and add an expensive component, the analogue switch, to the schematic. The simplest, and perhaps most efficient way is by using three simple OP-amps and a MOSFET, Figure 36. The circuit in (36) is actually rather simple, it is just two adders (B and C) where the lower adder, B, is switched in by setting $Sign$. The first OP, A, is just a voltage follower which makes the impedance of the digital potentiometer of little importance since the ideal input impedance of the follower is infinite. The final OP, C, is an adder, just like the one described using Equation 23, with the inputs of the voltages in (1) and (2) ($V_{(1)}$, $V_{(2)}$). $V_{(1)}$ is, due to the voltage follower, equal to V_{rv} regardless of the $Sign$ input. This means that the output, V'_{rv}, will be the inverse of V_{rv} if $Sign$ is low ($V_{(2)} = 0\ V$), Equation 24. If $Sign$ is set, the lower OP, B, will have an input voltage equal to V_{rv}. With the resistor relations according to Figure 36 this means that the output from B, $V_{(2)}$, will

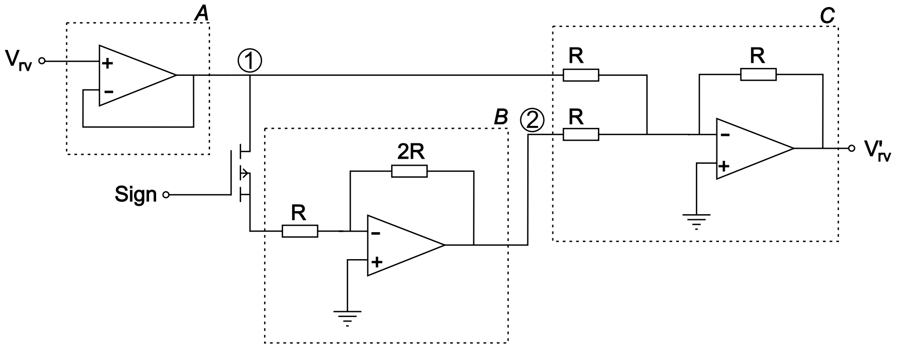

Figure 36: Changing signs of the reference value

be $-2V_{rv}$. Adding this to the voltage of node (1) produces an output, V'_{rv}, of $+V_{rv}$. Or written in a more comprehensive manner:

$$V'_{rv} = \begin{cases} -V_{rv}, \ Sign \leq 0 \\ V_{rv}, \ Sign > 0 \end{cases} \tag{27}$$

The only real problem with the circuit in (36) is that the resistor values need to be chosen carefully if the absolute values of the output should be exactly the same regardless of *Sign*.

The ability to switch between inverted and non inverted output might need further explanation before presentation. If the current should change directions, so should the reference value. This means that the error (e) will be calculated using two different formulas depending on the value of *Sign*:

$$e = \begin{cases} |V_l| - |V_{rv}|, \ Sign \leq 0 \quad (a) \\ |V_{rv}| - |V_l|, \ Sign > 0 \quad (b) \end{cases} \tag{28}$$

It was described earlier that the first expression, (28 a), leads to the need of inverting the output. But the second expression, (28 b), leads to a positive error if the current through the load is lower than the reference value. This is the opposite behaviour compared to that of Equation 28 a, leading to the need of inverting the output compared to that of the monophasic case, i.e. no inverting should be performed when *Sign* is set. In summary, the output should be inverted when *Sign* is low and not inverted when *Sign* is set. This behavior is achieved by inverting the *Sign* signal and then making a modulus 2 addition with the output signal, Figure 37.

The *Sign* signal is the signal controlling the direction of the current through the load, and in order for the system to work this signal needs to be connected to the correct transistors in the H-bridge. The solution described above makes it possible to control the H-bridge leg-wise instead of controlling each transistor individually. This reduces the needed control signals from the ATMega8. Two AND gates are needed to control the bridge, one for each leg, Figure 38. In (38), the boxes $DC_{1..4}$ are the driver circuits and transformer for each transistor. These circuits were described in detail in Section 4.6.2. The two new signals in (38), Leg_1 and Leg_2, are the control signals from the ATMega8 that decide whether the H-bridge leg should be active or not. Note that these signals are not

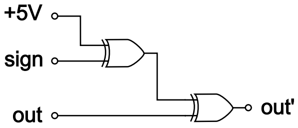

Figure 37: Inverting the output

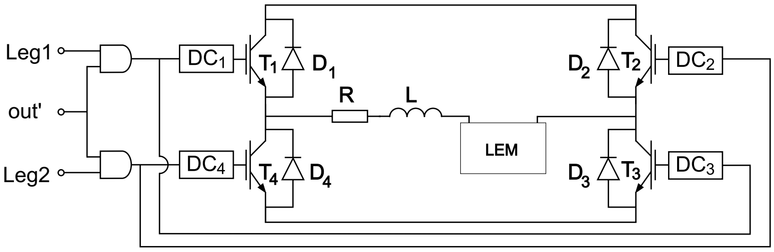

Figure 38: H-bridge connected to control signals

the same as *Sign* and $\overline{Sign}$ since this would cause one of the legs to be constantly active, thus spoiling the possibility to set the idle time. A better solution would be to replace the Leg_1 and Leg_2 signals with the *Sign* and $\overline{Sign}$ signals. This would require two three-inputs AND gates and an additional signal, *Discharge*, where the *Discharge* signal would decide whether the H-bridge should be active or not and the *Sign* would decide in which direction the current should flow. The final solution described would only require two ports of the ATMega8 whilst the previous solution requires three ports.

6 Software

The operational software running on the defibrillation system has been developed from scratch. No libraries or other 3rd party tools have been used to finalize the code. A development environment from ImageCraft has been used to edit, compile and finally to download the defibrillation program to the system.

6.1 ATMega8

Many products today contain some kind of central processing unit in order to provide the user with a specified functionality. Depending on factors such as processing power, cost, physical package, power consumption, electrical interfaces, etc, there are a number of different product lines that could be considered. Two different paths appear, microprocessors and microcontrollers. The difference between the two is that the microprocessor is dependent on peripheral circuits in order to work, whereas the microcontroller has the necessary subsystems such as memory etc. integrated on the chip.

When considering the design of a defibrillation system for LUCAS it was quite apparent that extreme processing power was not a key factor. This quickly led to the conclusion that a DSP[11], which is part of the microprocessor family, would not be a wise selection since they provide far too much arithmetic processing power. Programmable logic on the other hand, is more geared toward lightweight designs that require minimum processing power and thus programmable logic would not fit the bill either. As a parenthesis, it could be mentioned that this picture is about to change, since there are products on the market that combines programmable logic with a DSP core.

The next step was to examine the microcontroller market. Microcontrollers are convenient to work with since they usually contain many features, apart from onboard memory, such as ADCs[12], PWMs and digital input and output ports. There are many vendors of microcontrollers on the market, each having a number of different models, which can make the selection process rather difficult.

It was decided that a good idea would be to investigate microcontrollers from well known vendors such as Atmel, Philips, Microchip and Motorola. Another important part to consider was the software support available. Since a development tool for the AVR family of microcontrollers from Atmel was available it was decided to look closer at what Atmel had to offer.

The AVR product family is a RISC[13] microcontroller, which implies that it has a reduced number of instructions. Most of these 130 instructions are executed in just one clock cycle, which effectively means that the microcontroller can execute almost 4 MIPS[14] with a 4 MHz core clock. Among the 40 different versions of AVR that Atmel has for sale, the choice fell on the part called ATMega8 since it has an attractive package (28 pin DIL capsule) combined with the features that were anticipated to be needed. The main features of the ATMega8 are:

- 8 kBytes of flash (for program storage)

---

[11] Digital Signal Processor

[12] Analogue to Digital Converter

[13] Reduced Instruction Set Computer

[14] Million Instructions Per Second

- 512 bytes of EEPROM[15] (for permanent data storage)
- 1024 bytes of SRAM[16] (for program variables)
- 23 IO[17] pins (used for digital input and output)
- 2 external interrupts (used for trigging on external events)
- I^2C bus (for communicating with external circuits)
- 8 channel 10-bit ADC (used for measuring analogue signals)
- 2 timers (used for accurate timing needs)
- 3 PWM channels (used for controlling motors etc)

6.2 Overview

Since ATMega8 has a limited memory and the final size of the defibrillation program was difficult to foresee, it was decided that an RTOS[18] was going to be difficult to fit into the 8 kBytes of available flash memory. It would also have led to an increased complexity and thus the project was designed using state machines instead. A state machine is basically a global variable that keeps track of the state that the software is currently in. An example of a simple state machine is illustrated in Figure 39.

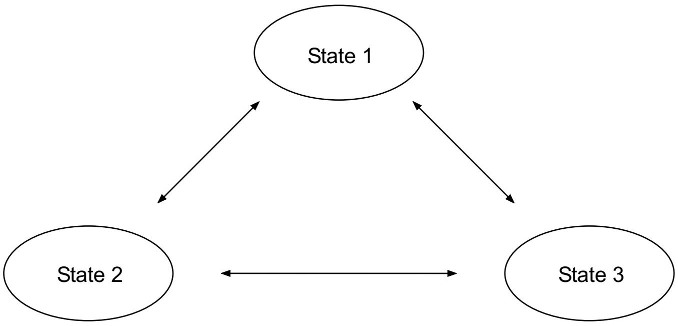

Figure 39: Example of a state machine

Transitions between different states are results of either external triggers or internal events. The defibrillation program has two state machines, one for the menu system and one for the defibrillation sequence. Both state machines react to external stimuli, but the defibrillation state machine is more autonomous in the sense that it runs through a predefined sequence once it has been started.

---

[15] Electrical Erasable Programmable Read Only Memory

[16] Static Random Access Memory

[17] Input Output

[18] Real Time Operating System

6.3 Menu system

The menu system is handled by a menu state machine which present a UI[19] on a small 2 by 16 character display, connected to the defibrillation system via a serial interface. The UI, shown in C.1, enables the user to change important defibrillation parameters as well as initiate a defibrillation. This is done by using the four menu buttons +, -, Enter and a Cancel which together mimics a menu behavior similar to that of early mobile phones.

The parameters and their functions are described using an example of a triphasic discharge shown in Figure 40.

6.3.1 Charge voltage

This is the voltage that the main capacitor is charged to. This gives the user an idea of how much energy that is available for defibrillation. The voltage level is selectable from 100 to 1350 V in steps of 50 V.

6.3.2 Discharge type

This menu option enables the user to determine which kind of defibrillation that should take place. The options are *monophasic*, *biphasic* and *triphasic* defibrillation. Depending on the selected discharge type, the menu options *Phase length* and *Idle time* have different number of submenus.

6.3.3 Current control

This menu option determines if the defibrillation should be current controlled or not. If the user selects *Yes* the current setting in menu *Discharge curr.* is used. If *No* is selected, the menu option *Discharge curr.* is not visible and a maximum discharge current of 25 A is used.

6.3.4 Phase length

This menu has different submenus depending on the selected *Discharge type*. If a *triphasic* defibrillation has been selected as in Figure 40, this submenu will enable the user to specify the length of A_1, A_2 and A_3 in quarters of milliseconds.

6.3.5 Safety time

When the user initiates a charge of the main capacitor and the specified voltage selected by *Charge voltage* is reached, the *Safety time* value determines the number of seconds that this voltage will be maintained before an automatic safety discharge will take place.

6.3.6 Idle time

This menu option contains a submenu where the user can edit the time interval between a positive and a negative discharge. If a triphasic defibrillation is performed, as done in (40), this sub menu enables the user to edit B_1 and B_2 in milliseconds. If a monophasic defibrillation is selected, this menu has no editable parameters.

---

[19] User Interface

6.3.7 Discharge current

The menu option enables the user to select the mean value of the discharge current, shown as C in (40).

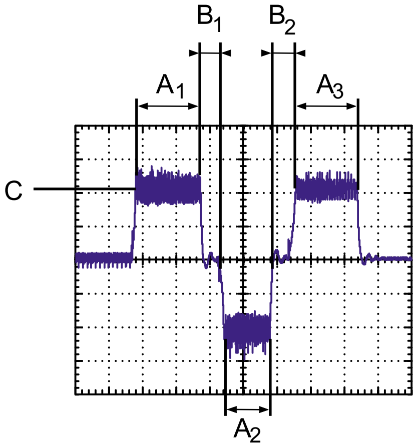

Figure 40: A triphasic defibrillation with editable time parameters

6.3.8 User banks

In order to aid the user to evaluate results of different parameter settings, the UI contains six complete parameter banks. These banks are stored to (using the save option) and retrieved (using the recall option) from the EEPROM which means that the user can quickly recall previously stored setups even if power is cycled. If the user wants to have a specific setting which is always loaded at power up, bank 1 should be selected since it is loaded automatically when the system is initiated.

6.4 Defibrillation system

The defibrillation system is handled by the defibrillation state machine. This state machine makes sure that charging, maintenance charging, discharging, safety discharging and other events takes place in a logical order to prevent hazardous conditions to occur for the defibrillation hardware as well as the user.

Figure 41 shows the major states of the defibrillation state machine. In order to describe the internal operation of this state machine a defibrillation scenario is described. References to the numbers in the figure are shown in brackets.

When the defibrillation system is powered, the *Startup* state is entered during which initialization of variables and display is performed. When done, the state advances to the *Idle* state where the user menu is shown. If the user selects to charge the capacitor, PWM is initialized and the state transition (1) is performed. If the user presses Cancel during the charging process, the charging is stopped and the energy in the capacitor is safely discharged (8). When the requested voltage has been reached, the *Maintenance Charging* state is entered (2). This state makes sure that the voltage across the capacitor is kept constant. A timer is also started and before this timer expires the user could choose to defibrillate (5) by pressing Enter or by applying an external trigger interrupt to defibrillate (4). The *Waiting to discharge* states waits an extra second before performing the actual defibrillation (7). If there is remanent energy in the capacitor after the defibrillation has taken place, this energy is automatically safely discharged through a resistor (6). If however, the timer expires before an external trigger or a user interaction has occured, the system automatically safety discharges the energy in the capacitor (3) so that the user is not exposed to hazardous conditions. When either a discharge or a safety discharge has been performed the state machine reverts back to an idle state via (9) or (10).

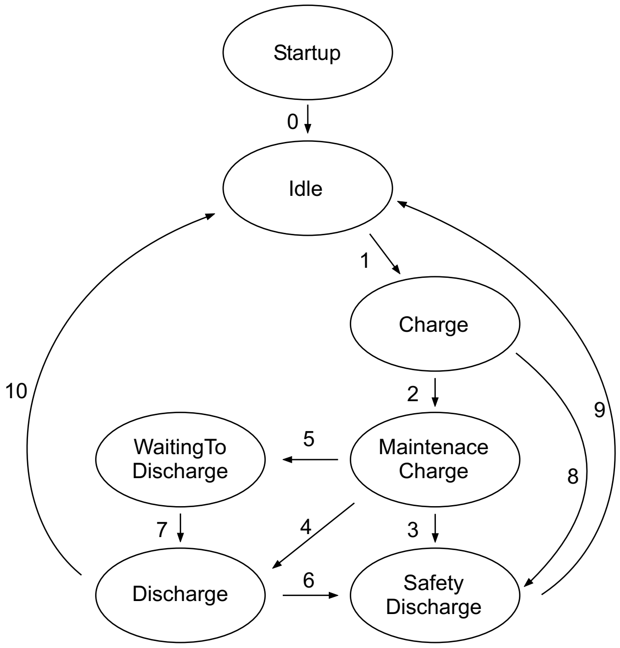

Figure 41: Defibrillation state machine

7 Result

This section describes the results and final solutions regarding the defibrillator and could be read somewhat separately by those who have little interest in the underlying electrical theory. The final solution consists of three separate PCB's:

- Charge and over voltage protection circuitry
- Discharge circuitry
- ATMega8 and current control circuitry

All the PCB's were created using the Eagle software from Cadsoft and manufactured at the department of industrial electrical engineering and automation. All plots of measured results were achieved using a Tektronix TDS2002 oscilloscope together with the WaveStar software.

7.1 Charging

The final version of the charging module consists of one PCB containing the charging electronics and the over voltage protection circuit described in Section 4.6. A complete schematic of the charging board is presented in B.2.1 and the PCB layout in B.2.2.

The only major difficulty encountered when designing the charging circuitry was the current measurement of the primary side. In the beginning of the project the idea was to measure this current and optimize the duty cycle of the switching transistor, so that the current through the primary windings of the transformer would never exceed 5 A. Despite extensive research and experimenting, the current could never be accurately measured with the system processor generating the PWM pulse. However, if a fast stand alone circuit for generating the PWM and measuring the current was to be used, the bandwidth of this circuit might be enough to measure the current directly without having to use mean value approximation or peak hold circuits.

The need for safety distances between signals on the high voltage side of the PCB was learned the hard way and no compromises have been made in the final design regarding isolation distances.

The final version of the charging PCB is capable of charging a 44.1 μF capacitor from 12 V DC to about 1350 V DC in 12 seconds, Figure 42. This equals a total energy of $W = C \cdot U^2/2 \approx 43.2$ $Joule$. Note that the plot in (42) only goes as far as 1200 V. This is a known measurement error and the actual peak voltage of the capacitor is 1350 V. It is possible to achieve higher voltage levels but since the IGBT controlling the safety discharge is limited to $V_{ce} \leq 1500$ V, higher secondary voltage is not recommended with this transistor. The charging time mentioned above will of course increase if a larger capacitor was to be charged, but this would be the only difference in respect of the charging electronics.

The 16 kHz PWM signal used to boost the voltage is of incremental duty cycle, since a constant duty cycle would cause charging time for the last hundreds of volts to be very long. The incremental duty cycle algorithm utilizes fixed values dependent on the voltage of the capacitor. These values has been experimentally deduced and optimized so that the average current consumption is below 600 mA during charging.

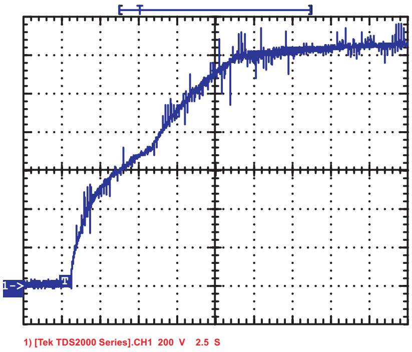

Figure 42: Measured capacitor voltage versus time

Something worth to notice regarding the switching transistor is that it has to be able to block much larger voltages than the supply voltage. The minimum voltage this transistor should be able to block is the supply voltage plus the voltage on the secondary side transfered via the transformer winding qouta, in this case:

$$V_{max,trans} = V_{cc} + V_{sec} \cdot \frac{n_p}{n_s} = 12 + 1400 \cdot 0.033 \approx 58.7\ V \tag{29}$$

The over voltage protection circuit implemented on the charging PCB works very well and is set to about 1420 V. However, it should be mentioned that no suitable thyristor was found, meaning that no safety discharge is performed directly on the secondary side but this is instead initiated via the same safety discharge circuit as used by the ATMega8.

7.2 Discharging and current control

The final version of the discharging module is composed of two PCB's, one containing the four quadrant converter, 5.1, and one that contains the current control circuitry together with the AVR. The schematics are viewed in B.3.1 (H-bridge) and in B.4.1 (AVR and current control). The PCB layouts used to manufacture the boards are shown in B.3.2 (H-bridge) and in B.4.2 (AVR and current controller).

The quadrant converter presented no major problems during the design phase. The largest problem encountered was that of driving the transistors galvanically isolated from the control electronics, but this was solved as described in Section 4.6.1.

Designing the control logic proved to be somewhat more troublesome. The tolerance band controller (described in Section 4.5) together with the biphasic behavior of the discharging requires the ability to change the sign of the current reference. This was solved using an inverting amplifier which amplified the reference value two times. If the current should change direction, the amplified and inverted reference value is added to the original reference value. This means that it is very hard to achieve the exact same absolute value of the current in both directions.

Perhaps the most important component when it comes to applying current control is the inductance placed in series with the load. This inductance has to be large enough in order to maintain reasonable fall times for the current through the load. The inductor core has to be large enough so that it does not saturate when large currents are flowing through the load. The final version of the discharging module consists of a ferrite core coil with an inner diameter of 4.8 cm and an inductance of around 3.5 mH. A smaller and lighter inductance might have been possible to use. However, this might cause the switching of the H-bridge transistors to rise above their maximum switching frequency causing them to heat up and eventually break.

Since the energy available in the capacitor is limited to around 43 $Joule$ no extensive measurement has been made regarding the behavior of the module during long discharging times. In Figure 43 a discharge is performed through a 22+27 Ω load and the voltage is measured across the 22 Ω resistor. The reference value for the current is 6 A (from the ATMega8 UI) and the capacitor is charged to 1.3 kV. In Figure 43, it is evident that current control works properly and the average current through the 22 Ω resistor is $I_l \approx 130/22 = 5.9\ A$. This corresponds to an error of approximately 1.67 %. A close up of the ripple is shown in Figure 44 below. The current ripple is about 44 $V/22\ \Omega = 2\ A$ or $\pm 1\ A$. The voltage spikes seen in Figure 44 are due to unfortunate noise in the control logic. The most probable noise sources are the driver circuits on the H-bridge PCB creating disturbances on the positive supply voltage causing the transistors to start conducting for a short period of time. Although extensive time and energy has been invested in the minimization of these disturbances the plot in (44) illustrates the best result achieved by the time of writing this report. The system is designed in such a way that it should be possible to set the maximum current ripple from the ATMega8, but this feature was canceled due to the fact that changing the current ripple changed the average current through the load. The problem is most certainly hardware related and should be possible to work around if more time was available for troubleshooting.

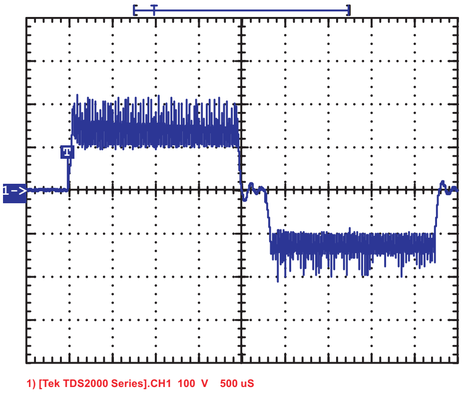

Figure 43: Current controlled biphasic defibrillation through a 22 Ω load

The differences in current between the positive and negative pulse are visible from Figure 45 which is a close up of Figure 43. Close inspection of (45) reveals a small difference in absolute values of the currents between the two directions,

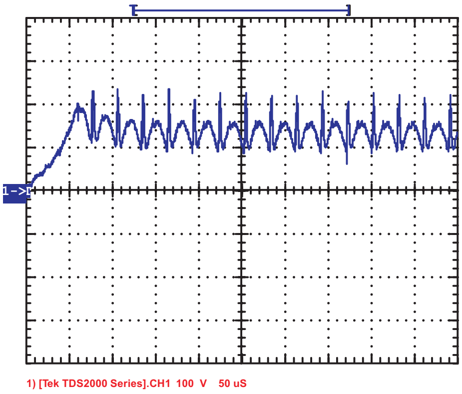

Figure 44: Close up of the current ripple

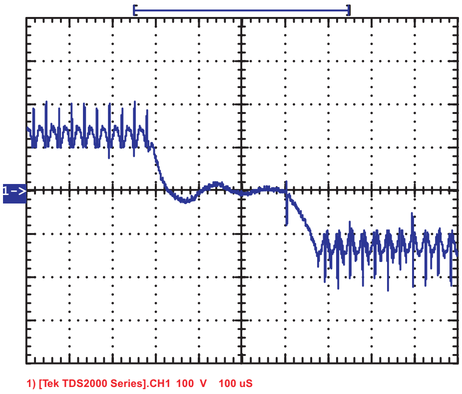

Figure 45: Close up of the idle time

$\approx 2/22\ A \approx 91\ mA$ which corresponds to 1.5 %. Considering the difficulty to accurately measure the current error, an error of 1.5 % could be neglected.

It was mentioned in Section 7.1 that the energy of the capacitor is limited. This becomes obvious when a current controlled defibrillation is performed since the current control requires a minimum voltage of $R_{patient} \cdot I_{reference}\ [A]$ across the capacitor in order to control the current. When the voltage of the capacitor is below this minimum value no current control is performed and the capacitor will discharge directly through the load. The output from the control circuitry when the voltage of the capacitor becomes too low can be seen in Figure 46. From Figure 46 it is seen that the frequency of the control signal is reduced as the capacitor voltage drops. At the point when the capacitance voltage is below the minimum controllable voltage the output signal will have a frequency of $0\ Hz$ with a $5\ V$ DC offset.

Including the one described above, there are six different modes of discharge available in the device. It is possible to perform a monophasic, biphasic or triphasic discharge, all with or without current control. The waveforms for all types of discharge are seen in Section B.1.

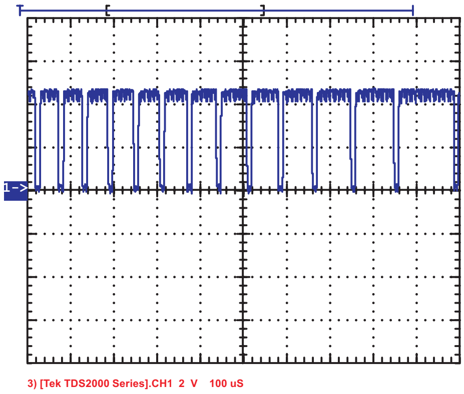

Figure 46: Output from control circuitry with low capacitor voltage

7.3 Meeting the standards

The initial idea presented in this masters thesis is closely related to the LUCAS device and is mainly focusing on investigating the possibilities of integrating a defibrillator in LUCAS. However, the thesis presented is not primarily a study of a defibrillator in LUCAS, but rather a design procedure for engineering a defibrillator. This has led to a stand alone defibrillator which needs much optimization before it can be integrated in LUCAS. Nevertheless the result of the thesis is a working prototype which can be further improved and optimized when it comes to weight and size in order to fit inside the LUCAS device.

Another important purpose of this thesis was to engineer or investigate a defibrillator with *all variables variable*. This goal has without a doubt been met, all factors that effect the defibrillation are variable except for the current ripple.

A current controlled defibrillator was desired and biphasic discharge was the preferred mean of defibrillation. Not only is the engineered prototype current controlled, it is also possible to perform triphasic defibrillation.

All in all, the goals of the thesis have been met. Even though further work is needed in order for the system to be completely compatible with LUCAS, a working prototype that can be tested together with LUCAS has been engineered.

7.4 Future improvements

The first priority when designing the defibrillator described in this report was to determine whether a defibrillator could be built and secondly to create working prototype. Working with this approach some trade offs have been made in order to produce the prototype on schedule. Many of the factors mentioned below are hardware related and should be solved by simply ordering better components, but since the delivery times of some of these components are very long it was decided to mention the factors here rather than actually waiting for the components to arrive.

One of the most obvious drawbacks of the defibrillator designed is that the energy of the capacitor is rather low. This is solved by changing the capacitor to one with higher capacitance. Doing this would result in an increase of stored energy proportional to that of the increase in capacitance. In order to further increase the energy the IGBT's should be replaced with components capable of enduring higher collector emitter voltages. The increase in stored energy would be proportional to the square of the increase in capacitor voltage. Both these actions should be performed before any real clinical testing of the device is commenced.

The microcontroller used to control the entire design should be replaced with a larger one from the same family before the projects develops further. The 8 kB of flash memory in the current controller (ATMega8L) is 94 % full and the replacement controller should have at least 16 kB of flash. Added features may cause the flash of the processor to fill completely thus resulting in malfunction.

Short-circuit protection circuitry needs to be added before the device is used clinically without the supervision of electrically competent people. Other safety factors that needs to be considered are shielding of the high voltage parts since these parts of the defibrillator should be protected from human touch.

Currently, no control of the PWM signal in order to keep the voltage at the desired value during the time between charging and discharging is implemented. Rough sketches of a digital PI controller has been produced, but the lack of available memory in the ATMega8 made it impossible to implement this.

The coil placed in series with the load is a bit overly dimensioned due to the lack of available core sizes. An alternative, if reducing the coil cause the core of the coil to saturate, is to increase the allowed ripple or the switching frequency of the current controller. However, the latter would require faster transistors than the ones currently used.

The present prototype is, as mentioned, built on three different PCB's. This means that each of these boards have its own voltage controller in order to convert the 12 V DC needed for the OP-amps to 5 V DC needed for the TTL logic. Furthermore, there are two 4 MHz oscillator circuits consuming approximately 20 mA each. So an obvious improvement would be to manufacture the entire design on one PCB, thus reducing the needed footprint and power consumption radically. The three PCB's that the prototype consists of are the first PCB's manufactured, this means that some of the PAD's of the PCB's have lifted of the board and several soldered wires had to replace the originally routed wires. A new version of the PCB's would be more stable and less sensitive to shock.

During the design of the three PCB's several compromises have been made in order to fit the layout onto the boards. This has led to a layout that is not completely optimized when it comes to EMC and the disturbances generated on the PCB's need to be carefully investigated and minimized. Furthermore, no tests regarding sensitivity to disturbances have been conducted and this is an important safety issue that needs to be considered when introducing the device in a clinical environment.

References

[1] Agnvall, Clas G., *Analog Elektronik, Kompendium 2000*, department of applied electronics, Lund Institute of Technology

[2] M. Olsson and M. Alaküla, *Elmaskinsystem*, IEA/LTH Lund, 2002

[3] Bardy, et al., 1996

[4] Cheng, David K., *Field and Wave Electromagnetics*, Addison Wesley, 1989, Second Edition

[5] ELFA, *catalogue ELFA 51*,2003

[6] Per Karlsson, *Kraftelektronik*, IEA/LTH Lund, Oct. 1998

[7] Per Karlsson, *Power Electronics Laboratory Exercises*, department of Industrial Electrical Engineering and Automation, Lund Institute of Technology

[8] Studies presented by Lombardi, Gallagher, & Gennis, 1994

[9] J. W. Machin, J. Brownhill and A Furness, *Design for a Constant Peak Current Defibrillator*, IEEE transactions on biomedical engineering, Vol. 37, No 7, July 1990

[10] Mohan N., Underland T., Robbins W., *Power Electronics, Converters, Applications, and Design*, John Wiley & Sons Inc, 1995, Second Editon

[11] Zhang Y, Ramabadran RS, Boddicker KA, Bawaney I, Davies LR, Zimmerman MB, Wuthrich S, Jones JL, Kerber RE., *Triphasic waveforms are superior to biphasic waveforms for transthoracic defibrillation: experimental studies*, J Am Coll Cardiol. 2003 Aug 6;42(3):568-75

A Technical specifications

A.1 Ports

The table below describes some of the ports available to the user.

Inputs		
Type	*Value*	*Description*
$\overline{Ext\ trig}$	$GND,\ 5\ V$	When set to GND, discharge will commence
$+V_{cc}$	$+12\ V$	Positive supply
$-V_{cc}$	$-12\ V$	Negative supply

A.2 Buttons

The table below describes the buttons on the defibrillator.

Buttons	
Name	*Description*
+	Menu button '+'
-	Menu button '-'
Enter	Menu button 'enter'
Cancel	Menu button 'cancel'
Reset	Resets the processor
OV reset	Resets the over voltage protection circuit

Note: pressing the reset button may set some of the outputs undefined thus causing malfunction

A.3 Current consumption

The following table presents the current consumption of the system during different stages. Note that currents listed are all avarage currents and the peak consumption could be as high as 5 A momentarily.

Current consumption	
Mode	*Current*
Idle	90 mA
Charging max	0.6 A
Charging min	0.3 A
Safety discharge	190 mA

B Hardware

This appendix provides additional information regarding the schematics and PCB layouts used in the project. It also presents some plots of different types of discharging.

B.1 Discharging plots

All discharging sequences in this section are performed through a 49 Ω load consisting of two, in series connected, resistors (22 Ω and 27 Ω). All plots show the voltage drop over the 22 Ω resistor.

B.1.1 No current control

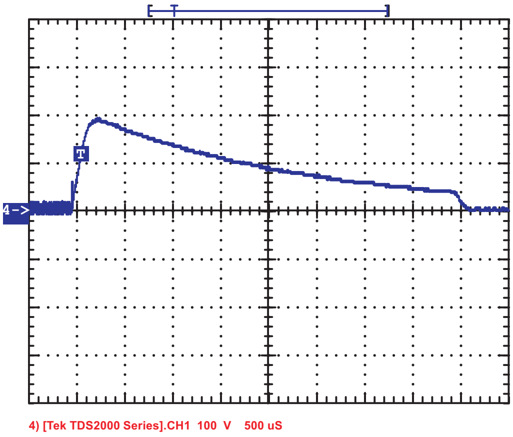

Figure 47: Monophasic discharge from 500 V

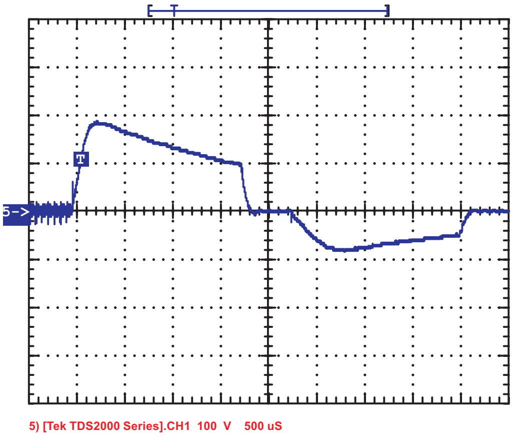

Figure 48: Biphasic discharge from 500 V

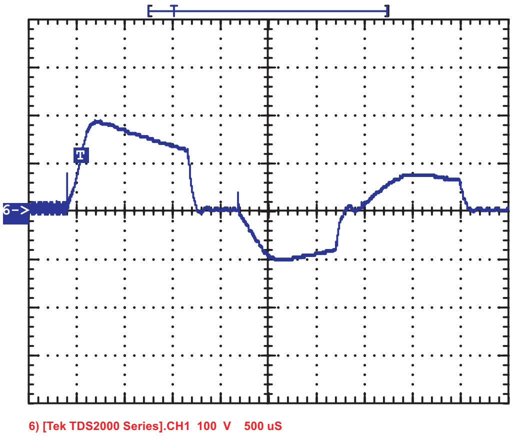

Figure 49: Triphasic discharge from 500 V

B.1.2 Current control

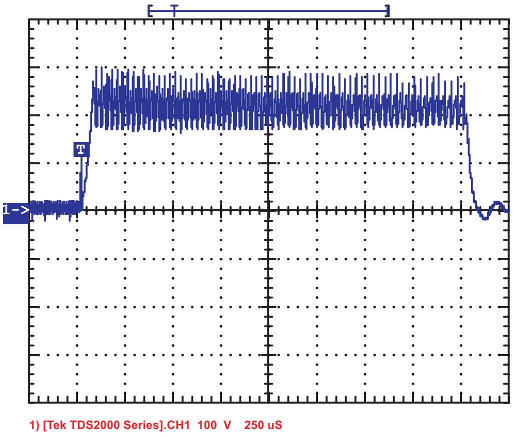

Figure 50: Monophasic discharge with 10 A

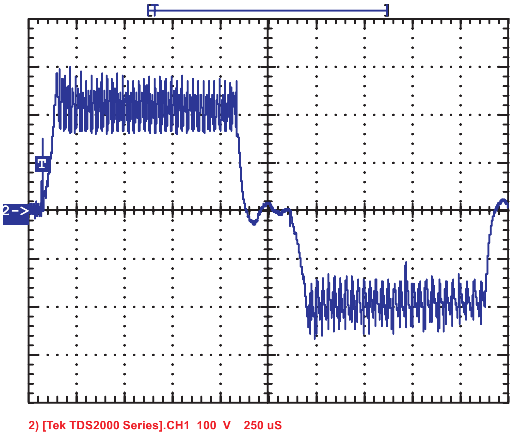

Figure 51: Biphasic discharge with 10 A

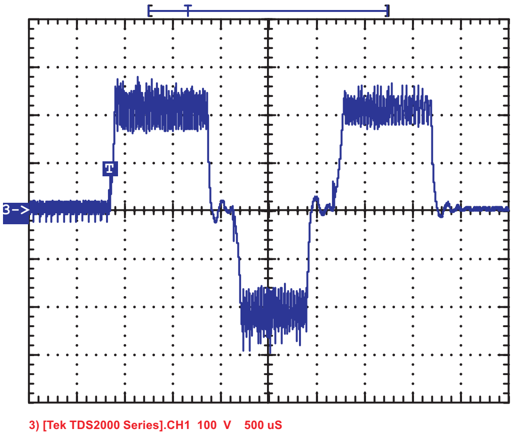

Figure 52: Triphasic discharge with 10 A

B.2 Charging

B.2.1 Schematic

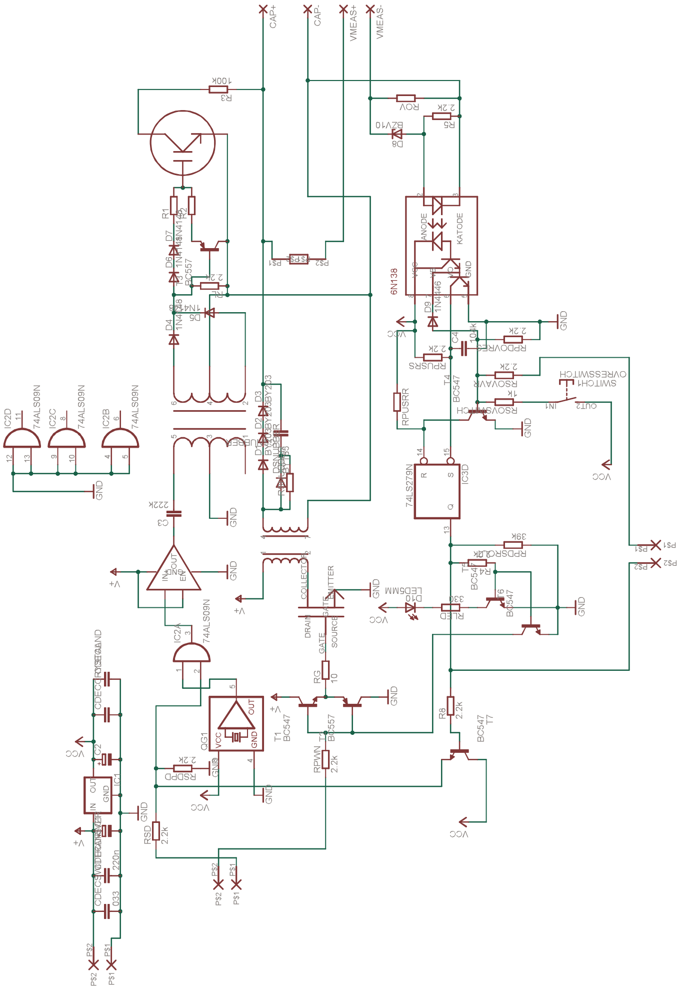

Figure 53: Schematic for the charging circuit

B.2.2 PCB

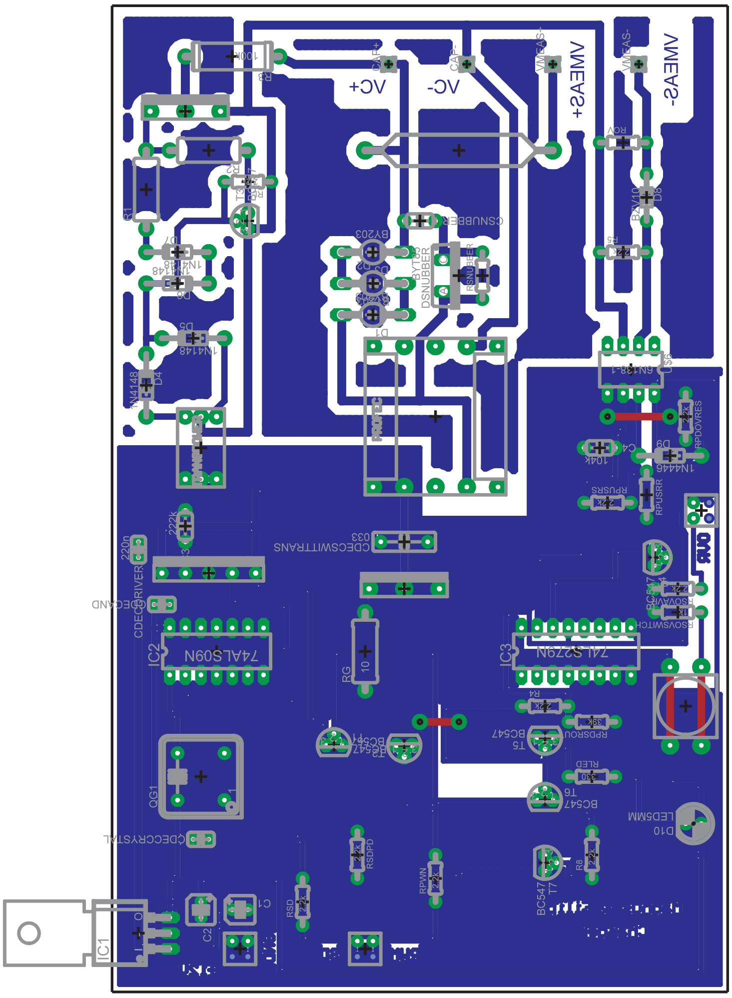

Figure 54: PCB layout for the charging board

B.3 Discharge

B.3.1 Schematic

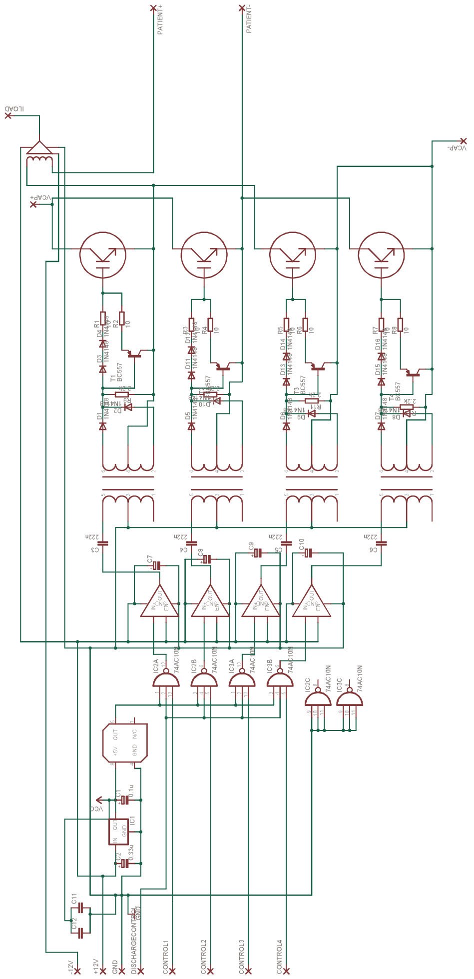

Figure 55: Schematic for the discharge circuit

B.3.2 PCB

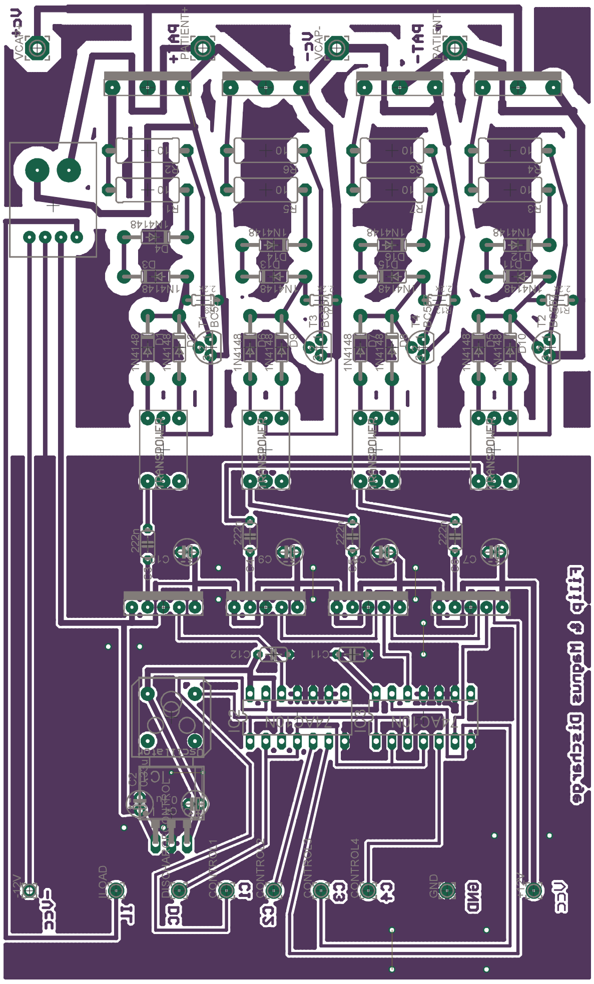

Figure 56: PCB layout for the discharge board

B.4 ATMega8 and control

B.4.1 Schematic

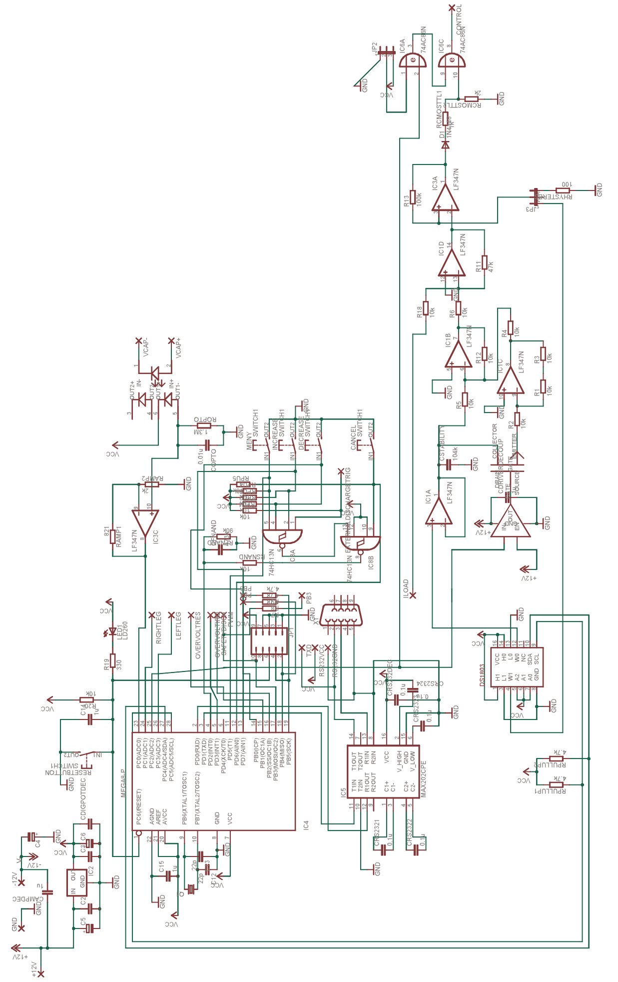

Figure 57: Schematic for the control and AVR circuit

B.4.2 PCB

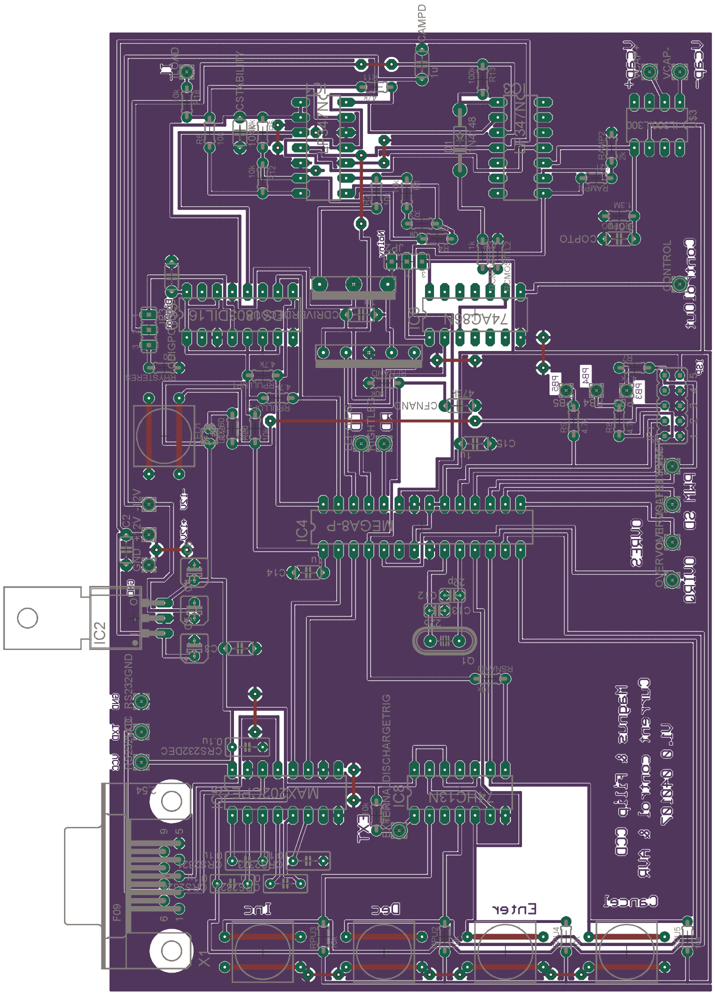

Figure 58: PCB layout for the AVR and control board

C Software

C.1 Graphical User Interface

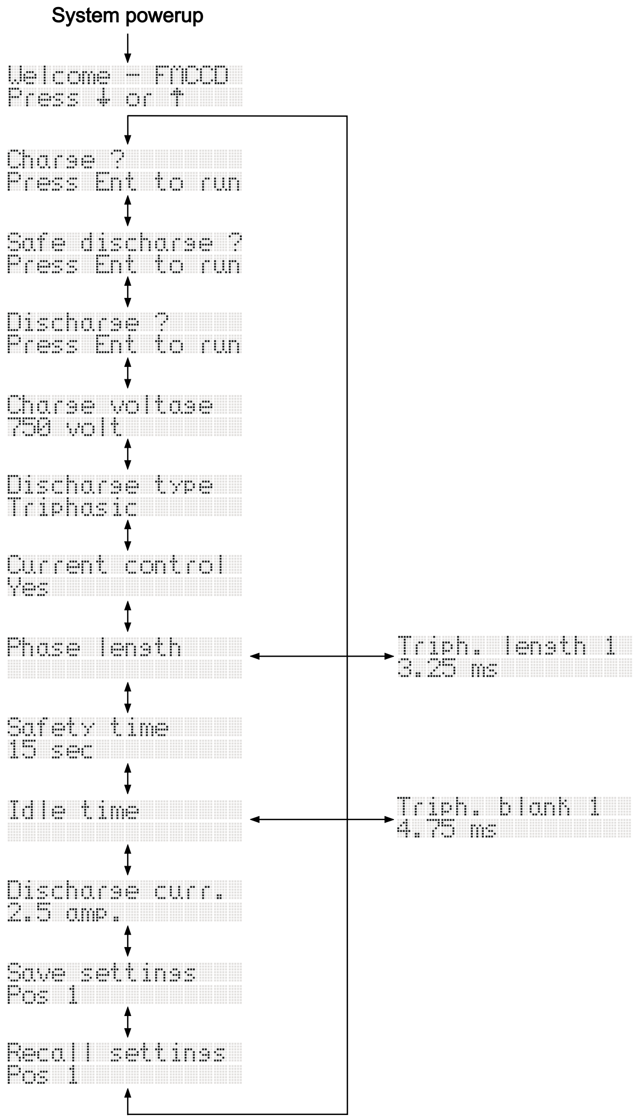

Figure 59: The menu system

C.2 Program listing

The program listed below is compiled using ICCAVR[20] from ImageCraft. This software environment enables the programmer to develop C programs and downloaded the compiled code to the AVR directly from within ICCAVR.

```
// This is the complete source code file for FMCCD
// (Filip Magnus Current Control Defibrillator)
// It has support for both a small display
// as well terminal program connected to the
// serial port of a PC. The input to the program
// is always buttons on the defibrillator, but
// it is also possible to receive commands from
// the serially connected PC.
// The PC option is switched on via compiler directives.

// The source code is written to fit an ATMega8L running
// at 4.0 MHz. If the crystal is changed, the source code
// needs to be modified as well.



#include <iom8v.h>
#include <macros.h>
#include <Stdlib.h>
#include <eeprom.h>
#include "FMCCD.h"

// This first passage is to describe what should be
// connected to the AVR
//
// What should be connected to the AVR when it comes
// to Ports, PWMs, ADCs etc
//
// --- PORTB ---
#define DDRPortB 0xFE
// PB0 (Pin 14) Connected to decrease (-) button (input)
// PB1 (Pin 15) Connected to overvoltage reset
// PB2 (Pin 16) Connected to Safety Discharge
// PB3 (Pin 17) Connected to PWM charge (OC2)
// PB4 (Pin 18) Extra option (a diode shows the current charge status)
// PB5 (Pin 19) Extra option (a diode shows the current charge status)
// PB6 (Pin  9) Used by XTAL
// PB7 (Pin 10) Used by XTAL
//
// --- PORTD ---
#define DDRPortD 0x02
// PD0 (Pin  2) RXD (Input)
// PD1 (Pin  3) TXD
// PD2 (Pin  4) INT0 Connected to buttons and external def. request
// PD3 (Pin  5) INT1 IRQ from overvoltage protection circuit
// PD4 (Pin  6)
// PD5 (Pin 11) Connected to Enter button (Input)
// PD6 (Pin 12) Connected to Cancel button (Input)
// PD7 (Pin 13) Connected to inc (+) button (Input)
//
// --- PORTC ---
#define DDRPortC 0xFE
// PC0 (Pin 23) Connected to measurement of sec voltage (ADC 0)
// PC1 (Pin 24) Connected to IGBT 2 and 3 (Right leg)
// PC2 (Pin 25) Connected to IGBT 1 and 4 (Left leg)
// PC3 (Pin 26) Enable. Connected to transistor to alter reference value polarity
//              This signal is also used in the XOR circuity to alter
//              the output signal that is used as enable from the current control
// PC4 (Pin 27) SDA
// PC5 (Pin 28) SCL
// PC6 (Pin 1 ) RESET


// First some defines

// Is display of terminal program used ?
// If UseTerminalProg is used it is presumed that there is
// a terminal program running on the PC and that the serial
// port is attached to the device.

//#define UseTerminalProg
#define UseSerialDisplay

// Input device used
// If a PC is connected to the serial device this define
// should be on
// #define UsePCAsInputControl

// IGBT connections
#define CIGBTLeftLeg  0x04
#define CIGBTRightLeg 0x02
#define CIGBTLeftLegPort PORTC
#define CIGBTRightLegPort PORTC
```

---

[20] version 6.29

```
// Safety discharge
#define CSafetyDischargePort PORTB
#define CSafetyDischarge 0x04
#define CSafetyDischargeLowLevelADC 0

// Charging PWM define
#define CChargingPWMPort OCR2
#define CChargingPWMOff 0xFF

// ADC defines
#define CSecondaryVoltageADC 0x00

// Diode for charge completion
#define CDiodeCompletionPort PORTB
#define CDiodeCompletionA 0x10
#define CDiodeCompletionB 0x20


// Discharge polarity
#define CDischargeWantedPolarityPort PORTC
#define CDischargeWantedPolarity 0x08

// Overvoltage
#define COvervoltageResetPort PORTB
#define COvervoltageReset 0x02

// External button defines
#define CButtonEnterPort PIND
#define CButtonEnter 0x20
#define CButtonCancelPort PIND
#define CButtonCancel 0x40
#define CButtonIncreasePort PIND
#define CButtonIncrease 0x80
#define CButtonDecreasePort PINB
#define CButtonDecrease 0x01


// Program defines

// These two defines are used for calibrating the ADC
// This has to be done since the optocoupler is not
// linear
// #define CalibrateVoltageLevels
// #define TestVoltageLevels

// TWI defines not defined in include files
#define START 0x08
#define MT_SLA_ACK 0x18
#define MT_DATA_ACK 0x28

// Digital potentiometer DS1803
// Selection bits for DS1803 are the 4 MSB
// which are defined as 0b0101 i.e.0x50
// (See page 4 of the DS1803 data sheet)
// The consequtive 3 bits is the address
// which is defined (by us) in hardware to be 0b111
#define DS1803_AdressRead 0x5F
#define DS1803_AdressWrite 0x5E
#define DS1803_Pot0Command 0xA9
#define DS1803_Pot1Command 0xAA
#define DS1803_Pot0and1Command 0xAF

// Misc defines
#define CMaxNoOfUserBanks 6
#define CEEPromDataValidMarker 0x4519
#define CMaxChargingVoltage 1350
#define CMinChargingVoltage 150
#define CMinimumBlankingTime 1
#define CHysterValue 5
#define CTicsToQuarterMilliSeconds 124

enum Bool {false = 0, true = 1};

// TProcessStep is an enum used in the main state machine
// which keeps track of what is currently happening
enum TProcessStep {psStartup,
                   psIdle,
                   psEnterIdleState,
                   psInitCharge,
                   psCharge,
                   psInitMaintenanceCharge,
                   psMaintenanceCharge,
                   psStopCharge,
                   psInitForHumanControlledDischarge,
                   psWaitingToDischarge,
                   psPreDischarge,
                   psDischarge,
                   psStopDischarge,
                   psInitSafetyDischarge,
                   psSafetyDischarge,
                   psStopSafetyDischarge,
                   psShutDown};
// TCurrentMenuMode is an enum that is used for the menu
// state machine. It keeps track of what is shown
// currently
enum TCurrentMenuMode {mmIdle = 1,
                       mmCharge = 2,
                       mmSafetyDischarge = 3,
                       mmDischarge = 4,
```

```
                       mmSetChargingVoltage = 5,
                       mmSetDischargeType = 6,
                       mmCurrentControlled = 7,
                       mmSetPhaseLength = 8,
                       mmSetSafetyTime = 9,
                       mmSetBlankingTime = 10,
                       mmSetDischargeRefVoltage = 11,
                       mmSaveParamToEEProm = 12,
                       mmRestoreParamFromEEProm = 13,

                       // The menu modes below are used
                       // for sub menu settings

                       mmSetPhaseLength_Mono1 = 14,
                       mmSetPhaseLength_Bi1 = 15,
                       mmSetPhaseLength_Bi2 = 16,
                       mmSetPhaseLength_Tri1 = 17,
                       mmSetPhaseLength_Tri2 = 18,
                       mmSetPhaseLength_Tri3 = 19,
                       mmSetBlankingTime_Bi = 20,
                       mmSetBlankingTime_Tri1 = 21,
                       mmSetBlankingTime_Tri2 = 22
                       };
// These constants are used for the calibrated ADC
// in order to convert an ADC value to a voltage level.
// There is also a PWM lookup table. The intention
// with this table is that a known voltage will be
// kept with this PWM value
#define LookupSize 23
const unsigned int ADCLookup[LookupSize] =     {  7,  16,  57, 110, 172, 243, 314, 354, 394, 435, 472, 518, 560, 609,
                                                652,  700,  745,  796,  842,  895,  940,  895, 1022};
const unsigned int VoltageLookup[LookupSize] = { 60, 100, 200, 300, 400, 500, 600, 650, 700, 750, 800, 850, 900, 950,
                                               1000, 1050, 1100, 1150, 1200, 1250, 1300, 1350, 1400};
const unsigned int PWMLookup[LookupSize] =     {254, 254, 253, 250, 244, 242, 240, 239, 238, 238, 237, 237, 236, 235,
                                                235,  234,  233,  232,  229,  222,  218,  215,  215};

// For showing a status (via a led) for the user
enum TDiodeStatus {dsLedOff = 1,
                   dsLedColor1 = 2,
                   dsLedColor2 = 3};
enum TParamStoragePos {psCurrentValue = 0,
                       psOldValue = 1};
// All editable parameters are stored in this vector.
// The reason for this is that is easy to revert to an old
// value if the user decides to press Cancel during an editing
// process
int ParameterStorage[2][mmSetBlankingTime_Tri2 + 1];
// Menu texts. These are stored in flash memory only
// and are not copied to SRAM by the compiler
const char *MenuFirstLineText[] = {"",
                                   "Welcome - FMCCD",
                                   "Charge ?",
                                   "Safe discharge ?",
                                   "Discharge ?",
                                   "Charge voltage",
                                   "Discharge type",
                                   "Current control",
                                   "Phase length",
                                   "Safety time",
                                   "Idle time",
                                   "Discharge curr.",
                                   "Save settings",
                                   "Recall settings",
                                   "Monophasic len.",
                                   "Biph. length 1",
                                   "Biph. length 2",
                                   "Triph. length 1",
                                   "Triph. length 2",
                                   "Triph. length 3",
                                   "Biph. blank",
                                   "Triph. blank 1",
                                   "Triph. blank 2",
                                   ""};

// TDischargeType is an enum describing what kind of
// discharge that will be performed. Single is equal to
// a monophasic, Double is a bipashic, Triple is tripahisc
// discharge
enum TDischargeType {dtSingle = 1,
                     dtDouble = 2,
                     dtTriple = 3};
// TIGBTSequence describes which IGBT leg (that is which
// pair of IBGT transistors) that is switched on
enum TIGBTSequence {ig_LeftLegOn,
                    ig_RightLegOn,
                    ig_LegsOff};
// TDischargeWantedPolarity describes which polarity the
// reference signal the controller should work with.
enum TDischargeWantedPolarity {wpPositive,
                               wpNegative};
// TTWIAction describes what action that should take place
// on the TwoWireInterface buss.
enum TTWIAction {TWI_StartCommand,
                 TWI_SlaveAddress,
                 TWI_SlaveData,
                 TWI_StopCommand};
// TDischargeParameter desribes which digital potentiometer
// that should be used. dpWantedValue controls the
// reference voltage for the controller and dpHysteresis
```

```
  // controls the hysteresis voltage for the controller.
  enum TDischargeParameter {dpWantedValue = 1,
                            dpHysteresis = 2};
  #ifdef UseSerialDisplay
    // TDisplayControlCodes is an enum which describes control
    // codes for the display.
    enum TDisplayControlCodes {dcClearScreen = 1,
                               dcMoveCursorHome = 2,
                               dcInvisibleCursor = 12,
                               dcBlinkingBlockCursor = 13,
                               dcMoveCursorOneLeft = 16,
                               dcMoveCursorOneRight = 20,
                               dcMoveToFirstLine = 128,
                               dcMoveToSecondLine = 192};
    enum TDisplayCustomChars {ccCursorUp = 3,
                              ccCursorDown = 4};
  #endif



  // Some variable definitions
  unsigned char MaintenacePWM = 248;
  #ifdef CalibrateVoltageLevels
    unsigned char CurrCalibPWMValue;
  #endif
  #ifdef TestVoltageLevels
    unsigned char CurrCalibPWMValue;
  #endif

  // Buffer used for printing
  char PrintStrBuff[20];

  unsigned char CurrentProcessStep = psStartup;
  unsigned char CurrentMenuMode = mmIdle;
  unsigned char EditingCurrentParameter = false;
  unsigned char NbrOfOvervoltageIRQ = 0;
  unsigned char IgnoreOverVoltageIRQ = true;
  unsigned char OkeyToShowMenu = true;
  int SafetyDischargeCounter = 0;

  //signed long   Ik; // This is used for the integral part of the PI-reg


// Initialize default values to paramterers
void InitializeParameterStorage()
{
  ParameterStorage[psCurrentValue][mmSetChargingVoltage] = CMinChargingVoltage;
  ParameterStorage[psCurrentValue][mmSetDischargeType] = dtDouble;
  ParameterStorage[psCurrentValue][mmCurrentControlled] = true;
  ParameterStorage[psCurrentValue][mmSetSafetyTime] = 15;
  ParameterStorage[psCurrentValue][mmSetBlankingTime] = 200;
  ParameterStorage[psCurrentValue][mmSetDischargeRefVoltage] = 25;
  ParameterStorage[psCurrentValue][mmSaveParamToEEProm] = 1;
  ParameterStorage[psCurrentValue][mmRestoreParamFromEEProm] = 1;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Mono1] = 20;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Bi1] = 20;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Bi2] = 20;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri1] = 20;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri2] = 20;
  ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri3] = 20;
  ParameterStorage[psCurrentValue][mmSetBlankingTime_Bi] = 2;
  ParameterStorage[psCurrentValue][mmSetBlankingTime_Tri1] = 2;
  ParameterStorage[psCurrentValue][mmSetBlankingTime_Tri2] = 2;
}


// Initialise the IO ports
void port_init(void)
{
  DDRB  = DDRPortB;
  PORTB = 0x00;

  DDRC  = DDRPortC;
  PORTC = 0x00;

  DDRD  = DDRPortD;
  PORTD = 0x00;
}


//TWI (I2C) initialisation
void twi_init(void)
{
  TWCR= 0X00; //disable twi
  TWBR= 0x03; //set bit rate
  TWSR= 0x03; //set prescale
  TWAR= 0x00; //set slave address
  TWCR= 0x45; //enable twi
}


// UART0 initialisation
// Baud rate: 9600,8 bit, no parity
void uart0_init(void)
{
  UCSRB = 0x00; //disable while setting baud rate
  UCSRA = 0x00;
  UCSRC = 0x86;
  UBRRL = 0x19; //set baud rate lo
```

```
  UBRRH = 0x00; //set baud rate hi
  UCSRB = 0x98;
}

// ADC initialisation
// Conversion time: 26uS
void adc_init(void)
{
  ADCSR = 0x00; //disable adc
  ADMUX = 0x20; //select adc input 0, left adjust
  ACSR  = 0x80;
  ADCSR = 0xC3;
}

// This timer is used for creating the charging PWM signal
// Timer 2 initialisation - prescale:1
// WGM: CTC
// desired value: 15KHz
// actual value: 15,038KHz (0,3%)
void timer2_init(void)
{
  TCCR2 = 0x00; //stop
  ASSR  = 0x00; //set async mode
  TCNT2 = 0x7C; //setup
  CChargingPWMPort  = CChargingPWMOff;
  TCCR2 = 0x79; //start
}



// Call this routine to initialise all peripherals
void init_devices(void)
{
  // Stop errant interrupts until set up
  CLI(); // Disable all interrupts
  port_init();
  timer2_init();
  uart0_init();
  adc_init();
  twi_init();

  MCUCR = 0x0A;
  GICR  = 0xC0;
  TIMSK = 0x00;
  SEI(); //re-enable interrupts
  // All peripherals are now initialised
}

// Delays a number of tics
void DelayTime(unsigned int Delay)
{
  while(--Delay > 0) {};
}

// Delay in milliseconds
void DelaymS(unsigned int Delayms)
{
  // This gives us pretty close to millisecond resolution
  Delayms = Delayms * 4;
  while(Delayms > 0) {
    DelayTime(CTicsToQuarterMilliSeconds);
    --Delayms;
  }
}

// Delays in seconds
void DelaySeconds(unsigned char DelaySec)
{
  while (DelaySec > 0) {
    DelaymS(1000);
    --DelaySec;
  }
}


#ifdef UseSerialDisplay
  // Sends a control code to the display
  void SendControlCharToDisplay(unsigned char ControlCode)
  {
    putchar(254);
    putchar(ControlCode);
    switch (ControlCode) {
      case dcClearScreen:
      case dcMoveCursorHome:
        DelayTime(5000);
        break;
      case dcMoveToFirstLine:
        break;
      case dcMoveToSecondLine:
        break;
    }
  }

  // Initializes the display
  void InitDisplay()
  {
    unsigned char m;
    // We need to wait almost 1 sec in order for the display to
    // power up.
```

```
    DelaymS(900);
    // We want to make character 0 (position 80 in the display
    // RAM) enmpty. This enables us to use the puts function
    // which terminates with a carriage return that shows up
    // as character 0
    putchar(254);  // Tell display to
    putchar(80);   // point to RAM (ASCII code 2)
    for (m = 0; m < 8; m++)
      putchar(0);
    // Then design an up and down arrow for showing that
    // it is possible to edit the current value
    // First the up arror
    putchar(254);  // Tell display to
    putchar(88);   // point to RAM (ASCII code 3)
    putchar(4);    // Data
    putchar(14);   // Data
    putchar(21);   // Data
    for (m = 0; m < 5; m++)
      putchar(4);  // Data
    putchar(254);  // Tell display to
    putchar(96);   // point to RAM (ASCII code 4)
    for (m = 0; m < 5; m++)
      putchar(4);  // Data
    putchar(21);   // Data
    putchar(14);   // Data
    putchar(4);    // Data
    putchar(254);  // Tell display to
    putchar(128);  // move cursor back to screen
  }


  // We override the normal library routine sine
  // it prints a CR in the end which looks bad on
  // our display
  int puts(char *s)
  {
    while (*s != '\0') {
      putchar (*s);
      s++;
    }
  }

#endif

// This routine writes data to the serial port
void WriteDataToScreen(char s[])
{
  puts(s);
}

// This method writes data that resides only in flash
void WriteConstDataToScreen(const char s[])
{
  char pTempChar[] = {"1234567890123456"};
  // We need to copy the const declared string from flash to a
  // RAM variable in order to be able to print it out
  strcpy(pTempChar, MenuFirstLineText[CurrentMenuMode]);
  WriteDataToScreen(pTempChar);
}

// This method shows an error message and then enters an endless
// loop preventing the system from running again until system reset
void ErrorHalt(char Msg1[], char Msg2[])
{
  SendControlCharToDisplay(dcClearScreen);
  WriteDataToScreen(Msg1);
  SendControlCharToDisplay(dcMoveToSecondLine);
  WriteDataToScreen(Msg2);
  while (true);
}


// This method shows a user message and then waits for
// two seconds before continuing
void WriteDelayedUserMessage(char Msg1[], char Msg2[])
{
  SendControlCharToDisplay(dcClearScreen);
  WriteDataToScreen(Msg1);
  SendControlCharToDisplay(dcMoveToSecondLine);
  WriteDataToScreen(Msg2);
  DelaySeconds(2);
}



// This routine writes data to the serial port
void WriteIntToScreen(char PreStr[], int Value, char PostStr[])
{
  // First the leading string
  char m;
  if (PreStr != NULL) {
    for (m = 0; PreStr[m] != '\0'; m++)
      putchar (PreStr[m]);
  }

  // Then check if this is a negative number
  // (itoa will print -10 as 65526 if an int)
  if (Value < 0) {
    Value = abs(Value);
```

```
  putchar(’-’);
  }

  // Write the value to serial port
  itoa(PrintStrBuff, Value, 10);
  puts(PrintStrBuff);

  if (PostStr != NULL) {
    for (m = 0; PostStr[m] != ’\0’; m++)
      putchar (PostStr[m]);
  }
}

// This method is used for notifying the user of the current
// status. Two ports are used to either light a green or a
// red diode
void SetDiodeStatus(unsigned char Status)
{
  switch (Status) {
    case dsLedOff:
      CDiodeCompletionPort = CDiodeCompletionPort & ~CDiodeCompletionA;
      CDiodeCompletionPort = CDiodeCompletionPort & ~CDiodeCompletionB;
      break;
    case dsLedColor1:
      CDiodeCompletionPort = CDiodeCompletionPort | CDiodeCompletionA;
      CDiodeCompletionPort = CDiodeCompletionPort & ~CDiodeCompletionB;
      break;
    case dsLedColor2:
      CDiodeCompletionPort = CDiodeCompletionPort & ~CDiodeCompletionA;
      CDiodeCompletionPort = CDiodeCompletionPort | CDiodeCompletionB;
      break;
  }
}

// This method saves all parameters to EEProm so that
// the user later can recall them if so desired
void SaveParametersToEEProm(unsigned char BankNo)
{
  // Each bank is 40 * 2 bytes big (i.e. 40 integers)
  // On the first address we store if the data is valid
  // If a specific value found, data is valid

  // Since BankNo is a value from 1 to 6
  // we reserve the space below 32 for
  // future program usage
  // This leads us to the following (EEPROM is 512 bytes)
  // if the maximal bankno is 6 :
  // (6 - 1) * 40 * 2 + 32 = 432
  // from 432 to 512 there is exactly 80 bytes to save for the last bank
  int Address = (BankNo - 1) * 40 * 2 + 32;
  unsigned char M;
  int CheckValue = CEEPromDataValidMarker;
  EEPROM_WRITE(Address, CheckValue);
  Address += sizeof(int);
  for (M = mmSetChargingVoltage; M <= mmSetBlankingTime_Tri2; M++) {
    EEPROM_WRITE(Address, ParameterStorage[psCurrentValue][M]);
    Address += sizeof(int);
  }
}

// This method recalls all parameters from EEProm.
// This is also done at power up if the contents of the
// EEProm is valid
void RestoreParametersFromEEProm(unsigned char BankNo)
{
  // Each bank is 40 * 2 bytes big (i.e. 40 integers)
  // On the first address we store if the data is valid
  // If a specific value found, data is valid

  // Since BankNo is a value from 1 to 6
  // we reserve the space below 32 for
  // future program usage
  // This leads us to the following (EEPROM is 512 bytes)
  // if the maximal bankno is 6 :
  // (6 - 1) * 40 * 2 + 32 = 432
  // from 432 to 512 there is exactly 80 bytes to save for the last bank
  int Address = (BankNo - 1) * 40 * 2 + 32;
  unsigned char M;
  int CheckValue;
  EEPROM_READ(Address, CheckValue);
  Address += sizeof(int);
  if (CheckValue == CEEPromDataValidMarker) {
    for (M = mmSetChargingVoltage; M <= mmSetBlankingTime_Tri2; M++) {
      EEPROM_READ(Address, ParameterStorage[psCurrentValue][M]);
      Address += sizeof(int);
    }
  } else
    ParameterStorage[psCurrentValue][mmSetChargingVoltage] = CMinChargingVoltage - 50;
}



// This method converts the inparameter to a time
// ms (the value is supposed to be shifted left by 2)
// which gives it a 0.25 resolution
void ConvertTimeToMilliSecondsAndPrint(int Time)
{
  // Time is here a value that has two decimals,
  // i.e. it is of fixed point type.
```

```
  unsigned char Temp = 0;
  WriteIntToScreen(NULL, Time >> 2, ".");
  if ((Time & 0x01) == 0x01)
    Temp = 25;
  if ((Time & 0x02) == 0x02)
    Temp += 50;
  WriteIntToScreen(NULL, Temp, " ms");
}


// Shows the main menu
void ShowMenu()
{
  if (OkeyToShowMenu) {
    SendControlCharToDisplay(dcClearScreen);
    WriteConstDataToScreen(MenuFirstLineText[CurrentMenuMode]);
    SendControlCharToDisplay(dcMoveToSecondLine);

    switch (CurrentMenuMode) {
      case mmIdle:
        WriteDataToScreen("Press ");
        putchar(ccCursorUp);
        WriteDataToScreen(" or ");
        putchar(ccCursorDown);
        break;

      case mmDischarge:
      case mmCharge:
      case mmSafetyDischarge:
        WriteDataToScreen("Press Ent to run");
        break;

      case mmSetChargingVoltage:
        WriteIntToScreen(NULL, ParameterStorage[psCurrentValue][CurrentMenuMode], " volt");
        break;
      case mmSetDischargeType:
        switch (ParameterStorage[psCurrentValue][CurrentMenuMode]) {
          case dtSingle:
            WriteDataToScreen("Monophasic");
            break;
          case dtDouble:
            WriteDataToScreen("Biphasic");
            break;
          case dtTriple:
            WriteDataToScreen("Triphasic");
            break;
        }
        break;
      case mmCurrentControlled:
        if (ParameterStorage[psCurrentValue][CurrentMenuMode])
          WriteDataToScreen("Yes");
        else
          WriteDataToScreen("No");
        break;

      case mmSetPhaseLength:
      case mmSetBlankingTime:
        // Do nothing. This menu contains a submenu
       break;

      case mmSetSafetyTime:
        WriteIntToScreen(NULL, ParameterStorage[psCurrentValue][CurrentMenuMode], " sec");
        break;

      case mmSetPhaseLength_Mono1:
      case mmSetPhaseLength_Bi1:
      case mmSetPhaseLength_Bi2:
      case mmSetPhaseLength_Tri1:
      case mmSetPhaseLength_Tri2:
      case mmSetPhaseLength_Tri3:
      case mmSetBlankingTime_Bi:
      case mmSetBlankingTime_Tri1:
      case mmSetBlankingTime_Tri2:
        ConvertTimeToMilliSecondsAndPrint(ParameterStorage[psCurrentValue][CurrentMenuMode]);
        break;

      case mmSetDischargeRefVoltage:
        WriteIntToScreen(NULL, ParameterStorage[psCurrentValue][CurrentMenuMode] / 10, ".");
        WriteIntToScreen(NULL, ParameterStorage[psCurrentValue][CurrentMenuMode] % 10, " amp.");
        break;
      case mmSaveParamToEEProm:
      case mmRestoreParamFromEEProm:
        WriteIntToScreen("Pos ", ParameterStorage[psCurrentValue][CurrentMenuMode], NULL);
        break;
      default:
       break;
    }
    if (EditingCurrentParameter) {
      putchar(32);
      putchar(ccCursorUp);
      putchar(ccCursorDown);
      SendControlCharToDisplay(dcBlinkingBlockCursor);
    } else
      SendControlCharToDisplay(dcInvisibleCursor);
  }
}


// Edit a settings depending on the current menu
```

```
// The inparameter describes which way (up or down)
// that we a re going
void AlterSetting(unsigned char ButtonPressed)
{
  // First find out which direction we are going. Up or down ?
  signed char Dir = (ButtonPressed == wpPositive) ? 1 : -1;
  unsigned char TempChar;
  switch (CurrentMenuMode) {

    // These are parameters that are adjustable
    case mmSetChargingVoltage:
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir * 50;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < CMinChargingVoltage)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = CMinChargingVoltage;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] > CMaxChargingVoltage)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = CMaxChargingVoltage;
      break;

    case mmSetDischargeType:
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < dtSingle)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = dtTriple;
      else if (ParameterStorage[psCurrentValue][CurrentMenuMode] > dtTriple)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = dtSingle;
      break;

    case mmCurrentControlled:
      if (ParameterStorage[psCurrentValue][CurrentMenuMode])
        ParameterStorage[psCurrentValue][CurrentMenuMode] = false;
      else
        ParameterStorage[psCurrentValue][CurrentMenuMode] = true;
      // Make sure that the digital potentiometer is set correctly
      SetCurrentDischargingParameters();
      break;

    case mmSetPhaseLength_Mono1:
    case mmSetPhaseLength_Bi1:
    case mmSetPhaseLength_Bi2:
    case mmSetPhaseLength_Tri1:
    case mmSetPhaseLength_Tri2:
    case mmSetPhaseLength_Tri3:
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < 0)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 0;
      break;

    case mmSetBlankingTime_Bi:
    case mmSetBlankingTime_Tri1:
    case mmSetBlankingTime_Tri2:
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < CMinimumBlankingTime)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = CMinimumBlankingTime;
      break;

    case mmSaveParamToEEProm:
    case mmRestoreParamFromEEProm:
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] > CMaxNoOfUserBanks)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 1;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < 1)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = CMaxNoOfUserBanks;
      break;

    case mmSetSafetyTime:
    ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < 1)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 1;
      break;

    case mmSetDischargeRefVoltage:
      // The user is about to change the value of the digital
      // potentiometers. We will address the DS1803 chip right
      // away to reflect the change !
      // If we fail, we will set the current value to zero in order
      // to show that something went wrong...
      ParameterStorage[psCurrentValue][CurrentMenuMode] += Dir * 5;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] > 250)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 250;
      if (ParameterStorage[psCurrentValue][CurrentMenuMode] < 10)
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 10;
      if (!SetDischargingParameter(dpWantedValue, ParameterStorage[psCurrentValue][CurrentMenuMode]))
        ParameterStorage[psCurrentValue][CurrentMenuMode] = 0;
      break;

    default:
    break;
  }
}



// This method sets both the digital potentiometers to
// the value determined by the menu system.
boolean SetCurrentDischargingParameters()
{
  boolean Result = false;
  unsigned char TempDischargeSetting;

  // Do the user want a current controlled def. ? If not
```

```
  // we set the wanted value (the ref voltage) to 0xFF
  if (ParameterStorage[psCurrentValue][mmCurrentControlled])
    TempDischargeSetting = ParameterStorage[psCurrentValue][mmSetDischargeRefVoltage];
  else
    TempDischargeSetting = 0xFF;

  // Make sure that the digital potentiometers have the correct value
  Result = SetDischargingParameter(dpHysteresis, CHysterValue);
  if (Result)
    Result = SetDischargingParameter(dpWantedValue, TempDischargeSetting);
  return Result;
}




// This method sets the wipers (used to set
// a predefined voltage level to the discharge
// logic) to a value from 0 to 255.
// Param is the parameter (i.e. the wiper)
// that is to be changed
// Value is the wanted output setting, a value
// from 0 to 255.
boolean SetDischargingParameter(unsigned char Param,
                                unsigned char Value)
{
  boolean Result = false;
  switch (Param) {
    case dpWantedValue:
      Result = WriteToDS1803(DS1803_Pot0Command, Value);
      break;
    case dpHysteresis:
      Result = WriteToDS1803(DS1803_Pot1Command, Value);
      break;
  }
  return Result;
}

// This method sets an IO port to control if the wanted
// value (reference voltage to the summator) should be
// negative or positive. The IO signal is fed to a
// PNP transistor.
void SetDischargeWantedPolarity(unsigned char Pol)
{
  if (Pol == wpPositive)
    CDischargeWantedPolarityPort = CDischargeWantedPolarityPort | CDischargeWantedPolarity;
  else if (Pol == wpNegative)
    CDischargeWantedPolarityPort = CDischargeWantedPolarityPort & ~CDischargeWantedPolarity;
}


// Sends a reset signal to the overvoltage circuit
void ResetOvervoltageCircuit()
{
  COvervoltageResetPort = COvervoltageResetPort | COvervoltageReset;
  DelayTime(500);
  COvervoltageResetPort = COvervoltageResetPort & ~COvervoltageReset;
}

// This method does all the setting of IO signals
// in order to do a proper discharge
void DoDischarge()
{
  CLI();
  switch (ParameterStorage[psCurrentValue][mmSetDischargeType]) {
    case dtSingle:
      // Set the IO signal for the tolerance band regulator
      SetDischargeWantedPolarity(wpNegative);
      // Switch on one leg of IGBT transistors
      EnableIGBTLeg(ig_RightLegOn);
      // Let the pulse last for a the desired time
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Mono1]);
      // Switch off the IGBT leg
      EnableIGBTLeg(ig_LegsOff);
      break;
    case dtDouble:
      SetDischargeWantedPolarity(wpNegative);
      EnableIGBTLeg(ig_RightLegOn);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Bi1]);
      EnableIGBTLeg(ig_LegsOff);
      // Change the polarity for the tolerance band regulator
      SetDischargeWantedPolarity(wpPositive);
      // Wait for the specified blanking time
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetBlankingTime_Bi]);
      EnableIGBTLeg(ig_LeftLegOn);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Bi2]);
      EnableIGBTLeg(ig_LegsOff);
      break;
    case dtTriple:
      SetDischargeWantedPolarity(wpNegative);
      EnableIGBTLeg(ig_RightLegOn);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri1]);
      EnableIGBTLeg(ig_LegsOff);
      SetDischargeWantedPolarity(wpPositive);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetBlankingTime_Tri1]);
      EnableIGBTLeg(ig_LeftLegOn);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri2]);
      EnableIGBTLeg(ig_LegsOff);
```

```
      // Change the polarity for the tolerance band regulator back again
      SetDischargeWantedPolarity(wpNegative);
      // Wait for the specified blanking time again
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetBlankingTime_Tri2]);
      EnableIGBTLeg(ig_RightLegOn);
      DelayTime(CTicsToQuarterMilliSeconds * ParameterStorage[psCurrentValue][mmSetPhaseLength_Tri3]);
      EnableIGBTLeg(ig_LegsOff);
      break;
    default:
      break;
  }
  SEI();
}



// This method does the actual setting
// of the IO signal to switch on and off
// the IGBT transistors

// The physical layout of the IGBTs are as
// follows
//
//   1     2
//      X
//   3     4
//
// LeftLeg diagonal is 1 and 4
// RightLeg diagonal is 2 and 3
//
void EnableIGBTLeg(unsigned char IGBTLeg)
{
  switch (IGBTLeg) {
    case ig_LeftLegOn:
      CIGBTRightLegPort = CIGBTRightLegPort & ~CIGBTRightLeg;
      CIGBTLeftLegPort = CIGBTLeftLegPort | CIGBTLeftLeg;
      break;
    case ig_RightLegOn:
      CIGBTLeftLegPort = CIGBTLeftLegPort & ~CIGBTLeftLeg;
      CIGBTRightLegPort = CIGBTRightLegPort | CIGBTRightLeg;
      break;
    case ig_LegsOff:
      CIGBTRightLegPort = CIGBTRightLegPort & ~CIGBTRightLeg;
      CIGBTLeftLegPort = CIGBTLeftLegPort & ~CIGBTLeftLeg;
      break;
    default:
      break;
  }
}



// Get an ADC reading from the selected ADC
unsigned int GetADCValue(unsigned char ADCNumber)
{
  int Value;
  ADMUX = ADCNumber;
  ADCSRA = ADCSRA | 0x40; // Start ADC
  while ((ADCSRA & 0x40) == 0x40); // wait for ADC to finnish
  Value = ADCL;
  Value += ((ADCH << 8));
  return Value;
}

// This method determines if the cap contains any energy, or rather
// can we see any voltage residing in the cap...
boolean IsCapacitorCharged()
{
  return (GetADCValue(CSecondaryVoltageADC) > CSafetyDischargeLowLevelADC);
}



// This method performs an ADC measurement
// and returns the calculated voltage
unsigned int MeasureVoltage()
{
  return CalcVoltage(GetADCValue(CSecondaryVoltageADC));
}


// Method that calclates the current voltage
// as a function of the ADC value
unsigned int CalcVoltage(unsigned int ADCValue)
{
  // This method contains two versions of code.
  // First one that does a coarse but fast
  // linear approximation.
  // Then one version which has certain fixpoints
  // that the code linearises between

  // Version 1
  // This is the short version that is not very exact
  //return (ADCValue * 2) + 350;

  // Version 2
  // This is the long version with a lookuptable
  // Unfortunately it is also a bit slower
```

```
  int Cnt;


  Cnt = 0;
  // First iterate through the ADC table to find out where we are
  while ((Cnt <= LookupSize) && (ADCValue >= ADCLookup[Cnt]))
    ++Cnt;
  --Cnt;
  // If we are 'above' the table, calc with the last known value
  if (Cnt == LookupSize)
    return ((ADCValue - ADCLookup[LookupSize - 2]) * 2 + VoltageLookup[LookupSize - 2]);
  // If we are below, just return zero voltage
  if (Cnt == -1)
    return 0;
  // Otherwise return the calulated value in the table
  return ((ADCValue - ADCLookup[Cnt]) * 2 + VoltageLookup[Cnt]);
}


// This method returns a PWM value for the inparameter (in volts)
// so that the voltage over the CAP can be made constant
unsigned char GetPWMForCurrentVoltage(unsigned int CurrentVoltage)
{
  // Since we do not measure the current through the transformers
  // primary side we have a lookup table that gives us a reasonable
  // duty cycle depending on the current voltage

  int Cnt;
  Cnt = 0;
  // First iterate through the Voltage table to find out where we are
  while ((Cnt <= LookupSize) && (CurrentVoltage >= VoltageLookup[Cnt]))
    ++Cnt;
  --Cnt;
  // If we are 'above' the table, just return the last known value
  if (Cnt == LookupSize)
    return PWMLookup[LookupSize - 2];
  // If we are below, just return the lowest value
  if (Cnt == -1)
    return PWMLookup[0];
  // Otherwise just return the PWM value for the current span
  return PWMLookup[Cnt];
}




// This method writes a wiper setting to the Dallas / Maxim
// potentiometer IC. The communication is handled over the
// I2C (TwoWireInterface) protocol.
boolean WriteToDS1803(unsigned char PotAddress, unsigned char Value)
{
  // First send a start command
  boolean Result = TWICommunication(TWI_StartCommand, 0);
  // then send the slave identifier (hardcoded in the DS1803)
  if (Result)
    Result = TWICommunication(TWI_SlaveAddress, DS1803_AdressWrite);
  // then send the address of the potentiometer to be used (0, 1 or both)
  if (Result)
    Result = TWICommunication(TWI_SlaveData, PotAddress);
  // then send the actual potentiometer setting
  if (Result)
    Result = TWICommunication(TWI_SlaveData, Value);
  // and finally send the stop command
  if (Result)
    Result = TWICommunication(TWI_StopCommand, 0);
  return Result;
}



// This method sends either a command or data over the I2C (TWI) buss.
boolean TWICommunication(unsigned char Command, unsigned char Data)
{
  boolean Result = true;
  switch (Command) {
    case TWI_StartCommand:
      TWCR = 0xA4;
      Result = WaitForTWIFlag();
      if (Result)
       Result = ((TWSR & 0xF8) == START);
      break;

    case TWI_SlaveAddress:
    case TWI_SlaveData:
      TWDR = Data;
      TWCR = 0x84;
      Result = WaitForTWIFlag();
      if (Result) {
        if (Command == TWI_SlaveAddress)
          Result = ((TWSR & 0xF8) == MT_SLA_ACK);
        else
          Result = ((TWSR & 0xF8) == MT_DATA_ACK);
      }
      break;

    case TWI_StopCommand:
      TWCR = 0x94;
      break;
```

```
    default:
      return false;
  }
  return Result;
}



boolean WaitForTWIFlag()
{
  unsigned int LoopCounter = 0;
  // We will now wait for the TWI interrupt flag to be set
  // This is done by hardware when the current TWI operation
  // has finished. Note that we will not reset the flag since this
  // implies a new TWI operation.
  // We have a afety counter just in case the device does not
  // respond
  while (((TWCR & 0x80) == 0x00) && (LoopCounter < 5000))
    ++LoopCounter;
  return ((TWCR & 0x80) == 0x80);
}



// This method is called when we receive an
// interrupt for incoming characters
#pragma interrupt_handler uart0_rx_isr:12
void uart0_rx_isr(void)
{
  #ifdef UsePCAsInputControl
    switch (UDR) {
      case 'q':
        CurrentProcessStep = psEnterIdleState;
        break;
      case 'w':
        CurrentProcessStep = psInitCharge;
        break;
      case 'e':
        CurrentProcessStep = psInitSafetyDischarge;
        break;
      case 'r':
        CurrentProcessStep = psInitForHumanControlledDischarge;
        break;

      // Below are things for debugging
      case 'a':
        SetDischargingParameter(dpWantedValue, 2);
        break;

      case 's':
        SetDischargingParameter(dpWantedValue, 22);
        break;

      case 'd':
        SetDischargingParameter(dpWantedValue, 5);
        break;

      case 'z':
        EnableIGBTLeg(ig_LeftLegOn);
        break;

      case 'x':
        EnableIGBTLeg(ig_RightLegOn);
        break;

      case 'c':
        EnableIGBTLeg(ig_LegsOff);
        break;

      case 'v':
        SetDischargeWantedPolarity(wpPositive);
        break;

      case 'b':
        SetDischargeWantedPolarity(wpNegative);
        break;

      default:
        break;
    }
  #endif
}



#pragma interrupt_handler twi_isr:18
void twi_isr(void)
{
  // We get an interrupt every time a TWI event is completed
  // However, we don't use the interrupt currently due to the
  // fact that we are not doing multiple things while using
  // the TWI line. This implies that we are waiting for a flag
  // to be set by polling anyway and an interrupt driven scenario
  // would not give us any benefits.
}
```

```
#pragma interrupt_handler int0_isr:2
void int0_isr(void)
{
  // Okay, so we have seen an interrupt
  // This could be due to a number of different reasons

  // 1. A key has been pressed
  //    All keys are wired so that when a key is pressed
  //    an interrupt is sent. We then read the ports to
  //    find out which key it was.
  //
  // 2. An external discharge signal was sent to us
  //    This means that we should defibrillate. How do we know?
  //    Well, we got an interrupt and none of the buttons were
  //    pressed...

  // First check the status of buttons
  //
  // Execute, Cancel, Increase and Decrease buttons


  if ((CButtonEnterPort & CButtonEnter) == 0x00) {

    switch (CurrentProcessStep) {
      case psCharge:
        break;
      case psMaintenanceCharge:
        // Discharge is the only thing we can do
        CurrentProcessStep = psInitForHumanControlledDischarge;
        break;

      default:
        // Are we in a menu that requires immediate
        // action instead of changing a parameter ?
        switch (CurrentMenuMode) {
          case mmIdle:
            break;
          case mmCharge:
            CurrentProcessStep = psInitCharge;
            break;
          case mmSafetyDischarge:
            CurrentProcessStep = psInitSafetyDischarge;
            break;
          case mmDischarge:
            CurrentProcessStep = psInitForHumanControlledDischarge;
           break;

          case mmSaveParamToEEProm:
          case mmRestoreParamFromEEProm:
            if (EditingCurrentParameter) {
              if (CurrentMenuMode == mmSaveParamToEEProm)
                SaveParametersToEEProm(ParameterStorage[psCurrentValue][CurrentMenuMode]);
              else {
                InitializeParameterStorage();
                RestoreParametersFromEEProm(ParameterStorage[psCurrentValue][CurrentMenuMode]);
              }
              EditingCurrentParameter = false;
            } else {
              EditingCurrentParameter = true;
              ParameterStorage[psOldValue][CurrentMenuMode] = ParameterStorage[psCurrentValue][CurrentMenuMode];
            }
            break;

          case mmSetPhaseLength:
            // The user want to alter the length of the current
            // discharge type. Depending on which one that is selected
            // that sub menu will be displayed.
            switch (ParameterStorage[psCurrentValue][mmSetDischargeType]) {
              case dtSingle:
                CurrentMenuMode = mmSetPhaseLength_Mono1;
                break;
              case dtDouble:
                CurrentMenuMode = mmSetPhaseLength_Bi1;
                break;
              case dtTriple:
                CurrentMenuMode = mmSetPhaseLength_Tri1;
                break;
            }
            break;

          case mmSetBlankingTime:
            // The user want to alter the blanking time for the current
            // discharge type. Depending on which one that is selected
            // that sub menu will be displayed.
            switch (ParameterStorage[psCurrentValue][mmSetDischargeType]) {
              case dtSingle:
                // We can not set a banking time for a monophasic (single) pulse
                break;
              case dtDouble:
                CurrentMenuMode = mmSetBlankingTime_Bi;
                break;
              case dtTriple:
                CurrentMenuMode = mmSetBlankingTime_Tri1;
                break;
            }
            break;
          default:
           // We are in a menu that has an editable
```

```
          // parameter
         EditingCurrentParameter = !EditingCurrentParameter;
          ParameterStorage[psOldValue][CurrentMenuMode] = ParameterStorage[psCurrentValue][CurrentMenuMode];
          break;
      }
      break;
 }

} else if ((CButtonCancelPort & CButtonCancel) == 0x00) {
  // The user has pressed the cancel button. This either means
  // that we want to abandon the current editing process
  // or that we want to go back from the current submenu
  // to the main menu or cancel a charging process

  // First check if we are in the process of charging
  // or maintenace charging the capacitor. If this is
  // the case we enter idle state, but we leave the
  // capacitor charged. This might be subject to change
  // later on...
  switch (CurrentProcessStep) {
    case psCharge:
      #ifdef CalibrateVoltageLevels
        CChargingPWMPort = CChargingPWMOff;
        DelaySeconds(3);
      #endif
    case psMaintenanceCharge:
      CurrentProcessStep = psStopCharge;
      break;
    default:
      // If we are not editing a value and we are in a submenu,
      // Cancel means that we want to go up a menu level
      if (!EditingCurrentParameter) {
        switch (CurrentMenuMode) {
          case mmSetPhaseLength_Mono1:
          case mmSetPhaseLength_Bi1:
          case mmSetPhaseLength_Bi2:
          case mmSetPhaseLength_Tri1:
          case mmSetPhaseLength_Tri2:
          case mmSetPhaseLength_Tri3:
            CurrentMenuMode = mmSetPhaseLength;
            break;
          case mmSetBlankingTime_Bi:
          case mmSetBlankingTime_Tri1:
          case mmSetBlankingTime_Tri2:
            CurrentMenuMode = mmSetBlankingTime;
            break;
        }
      } else
        // We were editing a value, and hence pressing Cancel
        // means that we want the old value back without
        // storing the newly edited one
        ParameterStorage[psCurrentValue][CurrentMenuMode] = ParameterStorage[psOldValue][CurrentMenuMode];
      EditingCurrentParameter = false;
      break;
    }

} else if ((CButtonIncreasePort & CButtonIncrease) == 0x00) {
  // The increase (+) button has been pressed.
  // This either means that we are editing a value or
  // that we are browsing the menu system. If we are
  // browsing the menu system we have to take special care
  // if we are in a submenu.

  switch (CurrentProcessStep) {
    case psMaintenanceCharge:
    case psCharge:
      #ifdef CalibrateVoltageLevels
        ++CurrCalibPWMValue;
      #endif
      #ifdef TestVoltageLevels
        ++CurrCalibPWMValue;
      #endif
      break;

    default:
      if (EditingCurrentParameter)
        AlterSetting(wpPositive);
      else {
        // Okay, we are not editing a value. Are we in a submenu?
        // The submenus have different 'length' so we must take this
        // into account.
        switch (CurrentMenuMode) {
          case mmSetPhaseLength_Mono1:
            break;
          case mmSetPhaseLength_Bi1:
            CurrentMenuMode = mmSetPhaseLength_Bi2;
            break;
          case mmSetPhaseLength_Bi2:
            CurrentMenuMode = mmSetPhaseLength_Bi1;
            break;
          case mmSetPhaseLength_Tri1:
          case mmSetPhaseLength_Tri2:
            ++CurrentMenuMode;
            break;
          case mmSetPhaseLength_Tri3:
            CurrentMenuMode = mmSetPhaseLength_Tri1;
            break;
          case mmSetBlankingTime_Bi:
            break;
```

```
          case mmSetBlankingTime_Tri1:
            CurrentMenuMode = mmSetBlankingTime_Tri2;
            break;
          case mmSetBlankingTime_Tri2:
            CurrentMenuMode = mmSetBlankingTime_Tri1;
            break;
          default:
            // Okay, we were not in a submenu. Just iterate
            // around in the top level menu system then as usual.
            ++CurrentMenuMode;

            // If we have a non current controlled option switched on
            // the current menu should not be visible
            if ((!ParameterStorage[psCurrentValue][mmCurrentControlled]) &&
                (CurrentMenuMode == mmSetDischargeRefVoltage))
              ++CurrentMenuMode;

            // If there is voltage in the CAP we should not be able
            // to change settings. Then we should just be allowed
            // to charge, discharge and safety discharge
            if (IsCapacitorCharged() && (CurrentMenuMode > mmDischarge))
              CurrentMenuMode = mmCharge;
            else if (CurrentMenuMode > mmRestoreParamFromEEProm)
              CurrentMenuMode = mmCharge;
            break;
        }
      }
      break;
  }
} else if ((CButtonDecreasePort & CButtonDecrease) == 0x00) {
  // The decrease (-) button has been pressed.
  // This either means that we are editing a value or
  // that we are browsing the menu system. If we are
  // browsing the menu system we have to take special care
  // if we are in a submenu.

  switch (CurrentProcessStep) {
    case psMaintenanceCharge:
    case psCharge:
      #ifdef CalibrateVoltageLevels
        --CurrCalibPWMValue;
      #endif
      #ifdef TestVoltageLevels
        --CurrCalibPWMValue;
      #endif
      break;

    default:
      if (EditingCurrentParameter)
        AlterSetting(wpNegative);
      else {
        // Okay, we are not editing a value. Are we in a submenu?
        // The submenus have different ’length’ so we must take this
        // into account.
        switch (CurrentMenuMode) {
          case mmSetPhaseLength_Mono1:
            break;
          case mmSetPhaseLength_Bi1:
            CurrentMenuMode = mmSetPhaseLength_Bi2;
            break;
          case mmSetPhaseLength_Bi2:
            CurrentMenuMode = mmSetPhaseLength_Bi1;
            break;
          case mmSetPhaseLength_Tri2:
          case mmSetPhaseLength_Tri3:
            --CurrentMenuMode;
            break;
          case mmSetPhaseLength_Tri1:
            CurrentMenuMode = mmSetPhaseLength_Tri3;
            break;
          case mmSetBlankingTime_Bi:
            break;
          case mmSetBlankingTime_Tri1:
            CurrentMenuMode = mmSetBlankingTime_Tri2;
            break;
          case mmSetBlankingTime_Tri2:
            CurrentMenuMode = mmSetBlankingTime_Tri1;
            break;
          default:
            // Okay, we were not in a submenu. Just iterate
            // around in the top level menu system then as usual.
            --CurrentMenuMode;

            // If we have a non current controlled option switched on
            // the current menu should not be visible
            if ((!ParameterStorage[psCurrentValue][mmCurrentControlled]) &&
                (CurrentMenuMode == mmSetDischargeRefVoltage))
              --CurrentMenuMode;

            // If there is voltage in the CAP we should not be able
            // to change settings. Then we should just be allowed
            // to charge, discharge and safety discharge
            if (IsCapacitorCharged() && (CurrentMenuMode < mmCharge))
              CurrentMenuMode = mmDischarge;
            else if (CurrentMenuMode < mmCharge)
              CurrentMenuMode = mmRestoreParamFromEEProm;
            break;
        }
      break;
```

```
      }
    }
  } else {
    // We got ourselves an interrupt even though no key was
    // pressed...
    // This must be an external interrupt
    // The user wants to either charge or discharge
    // the cap

    switch (CurrentProcessStep) {
      case psIdle:
        // We are in idle mode. An interrupt here means that
        // we want to charge the capacitor
        CurrentProcessStep = psInitCharge;
        break;

      case psMaintenanceCharge:
        // We are currently in a charged mode. An interrupt here
        // means that we want to discharge
        CurrentProcessStep = psPreDischarge;
        break;

      default:

        break;
    }
  }
  ShowMenu();
}




#pragma interrupt_handler int1_isr:3
void int1_isr(void)
{
  // Okay, we have seen an interrupt
  // Reason is that we have an overvoltage from the overvoltage logic.

  // We might get interrupts during power-up. We want to
  // avoid this from getting the system into a locked mode.
  if (!IgnoreOverVoltageIRQ) {
    // First stop the PWM pulse
    CChargingPWMPort = CChargingPWMOff;
    ++NbrOfOvervoltageIRQ;
    // then start discharging if this is the first time
    if (NbrOfOvervoltageIRQ == 1) {
      WriteDelayedUserMessage("Overvoltage err!", "Discharging");
      CurrentProcessStep = psStopCharge;
    } else {
      CSafetyDischargePort = CSafetyDischargePort | CSafetyDischarge;
      ErrorHalt("OVERVOLTAGE ERR!", "System halted!");
    }
  }
}


// The intention with this code was to create a simple
// PI controller which kept the voltage constant
// It turns out however that the compiler creates much
// larger code if the type long is used. Long was used
// due to the fact that we need decent resolution
// for our discrete time calculations.
// We did not have time to solve this problem since
// the code already takes up 94% of the available
// flash memory
/*

unsigned char PIReg(int CurrentCAPVoltage)
{
  // We have a PI regulator that tries to keep the voltage
  // over the CAP constant.
  // Inparameter is the current voltage measured by the
  // ADC and the menu system supplies us with the wanted value

  // The returned result from this method is a 'new' PWM value

  // This define tells us how many bits that should be used for
  // the decimal part and how many for the integer part
  // i.e. a long is 32 bits. We need
  // SignBit + IntegerBits + DecimalBits
  #define IntegerCutOffPos 10;

  signed long VoltageErr;
  signed long uk;
  signed long a1; //(20 <<  IntegerCutOffPos);
  signed long a2; // = (10 <<  (IntegerCutOffPos - 3)); // Should be 0.010

  // First calculate the error
  VoltageErr = (CurrentCAPVoltage - ParameterStorage[psCurrentValue][mmSetChargingVoltage]);

  // Calculate the new output :    uk = a1*ek + Ik;
  uk = (a2); // >> IntegerCutOffPos) + Ik;

  // then calculate a new integral part for the next run
  // Ik(+) = Ik + a2*e
  //Ik = Ik + (((a2 << IntegerCutOffPos) * VoltageErr) >> IntegerCutOffPos);

  // The last thing we need to do is to rescale the output
```

```
  //return (uk >> IntegerCutOffPos);
  return 5;
}

*/


// Main method
void main(void)
{
  unsigned int LoopCount = 0;
  unsigned char TempChar;
  int TempValue;
  int TempVoltage;
  int DischargeCounter = 0;

  // First init all devices
  IgnoreOverVoltageIRQ = true;
  init_devices();
  InitializeParameterStorage();

  while(1 == 1) {

    LoopCount++;

    switch (CurrentProcessStep) {
      case psStartup:
        #ifdef UseSerialDisplay
          InitDisplay();
        #endif
        #ifdef UseTerminalProg
          WriteDataToScreen("System initialised, version 0.1");
          #ifdef UsePCAsInputControl
            WriteDataToScreen("Use keys:");
            WriteDataToScreen("q - Idle");
            WriteDataToScreen("w - Charge");
            WriteDataToScreen("e - Safety discharge");
            WriteDataToScreen("r - Discharge");
          #endif
          WriteDataToScreen("");
          WriteDataToScreen("System idle");
          WriteDataToScreen("");
        #endif

        // In case the overvoltage reset happens to be switched on
        // during power up (spikes ?) we will just reset it
        ResetOvervoltageCircuit();

        // Get the settings from EEProm. The settings retreived are those
        // in position 1
        RestoreParametersFromEEProm(1);

        // Initialize digital potentiometers
        SetCurrentDischargingParameters();

        IgnoreOverVoltageIRQ = false;
        CurrentProcessStep = psEnterIdleState;
        break;
      case psEnterIdleState:
        CChargingPWMPort = CChargingPWMOff;
        CSafetyDischargePort = CSafetyDischargePort & ~CSafetyDischarge;
        CurrentProcessStep = psIdle;
        OkeyToShowMenu = true;
        ShowMenu();
        SetDiodeStatus(dsLedOff);
        break;
      case psIdle:
        // Do nothing right now
        break;
      case psInitCharge:
        OkeyToShowMenu = false;
        SetDiodeStatus(dsLedColor1);
        #ifdef UseTerminalProg
          WriteIntToScreen("Upper voltage  = ", ParameterStorage[psCurrentValue][mmSetChargingVoltage], NULL);
          WriteDataToScreen("Starting to charge...");
        #endif
        #ifdef UseSerialDisplay
          SendControlCharToDisplay(dcClearScreen);
          WriteIntToScreen("Charging to ", ParameterStorage[psCurrentValue][mmSetChargingVoltage], NULL);
        #endif
        #ifdef CalibrateVoltageLevels
          CurrCalibPWMValue = 255;
        #endif
        #ifdef TestVoltageLevels
          CurrCalibPWMValue = 255;
        #endif
        CurrentProcessStep = psCharge;
        break;
      case psCharge:
        // During the charging process we measure the voltage over the cap
        // and depending on that voltage we set different PWM values

        #ifdef CalibrateVoltageLevels
          TempValue = GetADCValue(CSecondaryVoltageADC);
          CChargingPWMPort = CurrCalibPWMValue;
        #else
          TempValue = MeasureVoltage();
          #ifdef TestVoltageLevels
            CChargingPWMPort = CurrCalibPWMValue;
```

```
        #else
          if (TempValue < 800)
            CChargingPWMPort = 216;
          else
            CChargingPWMPort = 207;
        #endif

      #endif

      if (LoopCount % 1000 == 0) {
        #ifdef UseTerminalProg
          WriteIntToScreen("Voltage  = ", TempValue, NULL);
        #endif
        #ifdef UseSerialDisplay
          #ifdef CalibrateVoltageLevels
            SendControlCharToDisplay(dcClearScreen);
            WriteIntToScreen("PWM : ", CurrCalibPWMValue, NULL);
            SendControlCharToDisplay(dcMoveToSecondLine);
            WriteIntToScreen("DAC : ", TempValue, " steps");
          #else
            #ifdef TestVoltageLevels
              SendControlCharToDisplay(dcClearScreen);
              WriteIntToScreen("ADC : ", GetADCValue(CSecondaryVoltageADC), NULL);
            #endif
            SendControlCharToDisplay(dcMoveToSecondLine);
            WriteIntToScreen("Voltage : ", TempValue, " V  ");
          #endif
        #endif
      }
      if (TempValue > ParameterStorage[psCurrentValue][mmSetChargingVoltage]) {
        CurrentProcessStep = psInitMaintenanceCharge;
        CurrentMenuMode = mmDischarge;
      }
      break;
    case psInitMaintenanceCharge:
      SetDiodeStatus(dsLedColor2);
      CChargingPWMPort = GetPWMForCurrentVoltage(ParameterStorage[psCurrentValue][mmSetChargingVoltage]);
      SafetyDischargeCounter = ParameterStorage[psCurrentValue][mmSetSafetyTime];
      #ifdef UseTerminalProg
        WriteDataToScreen("Maintenance charging...");
      #endif
      CurrentProcessStep = psMaintenanceCharge;
      break;

    case psMaintenanceCharge:
      if (LoopCount % 1000 == 0) {
        TempVoltage = MeasureVoltage();
      }
      // Check if we should safety discharge
      if (LoopCount % 9000 == 0) {
        SafetyDischargeCounter--;
        #ifdef UseSerialDisplay
          SendControlCharToDisplay(dcClearScreen);
          WriteConstDataToScreen(MenuFirstLineText[mmDischarge]);
          SendControlCharToDisplay(dcMoveToSecondLine);
          WriteIntToScreen("Ent in ", SafetyDischargeCounter, " secs");
        #endif
        #ifdef UseTerminalProg
          WriteIntToScreen("Safetycounter = ", SafetyDischargeCounter, BULL);
        #endif
        if (SafetyDischargeCounter == 0)
          CurrentProcessStep = psStopCharge;
      }
      break;

    case psStopCharge:
      CChargingPWMPort = CChargingPWMOff;
      #ifdef UseTerminalProg
        WriteDataToScreen("Charging stopped...");
      #endif
      #ifdef UseSerialDisplay
        SendControlCharToDisplay(dcClearScreen);
        WriteDataToScreen("Charging stopped...");
      #endif
      CurrentProcessStep = psInitSafetyDischarge;
      break;

    case psInitSafetyDischarge:
      OkeyToShowMenu = false;
      #ifdef UseTerminalProg
        WriteDataToScreen("Starting safety discharge...");
      #endif
      #ifdef UseSerialDisplay
        SendControlCharToDisplay(dcClearScreen);
        WriteDataToScreen("Safety discharge");
        SendControlCharToDisplay(dcMoveToSecondLine);
        WriteDataToScreen("Please wait !");
      #endif
      CSafetyDischargePort = CSafetyDischargePort | CSafetyDischarge;
      CurrentProcessStep = psSafetyDischarge;
      break;

    case psSafetyDischarge:
      if (!IsCapacitorCharged())
        CurrentProcessStep = psStopSafetyDischarge;
      break;

    case psStopSafetyDischarge:
      // We wait some extra time just to be sure
```

```
        DelaySeconds(2);
        CSafetyDischargePort = CSafetyDischargePort & ~CSafetyDischarge;
        #ifdef UseTerminalProg
          WriteDataToScreen("Safety discharge done!");
        #endif
        #ifdef UseSerialDisplay
          WriteDelayedUserMessage("Safety discharge", "Done !");
        #endif
        CurrentProcessStep = psEnterIdleState;
        break;

      case psInitForHumanControlledDischarge:
        OkeyToShowMenu = false;
        #ifdef UseTerminalProg
          WriteDataToScreen("Stand by for discharge");
        #endif
        #ifdef UseSerialDisplay
          SendControlCharToDisplay(dcClearScreen);
          WriteDataToScreen("Discharging...");
        #endif
        DischargeCounter = 2;
        CurrentProcessStep = psWaitingToDischarge;
        // This is due to the fact that the overvoltage circuit
        // might react during a discharge

      case psWaitingToDischarge:
        if (LoopCount % 10000 == 0) {
          DischargeCounter--;
          #ifdef UseTerminalProg
            WriteIntToScreen("Time to discharge = ", DischargeCounter, NULL);
          #endif
          #ifdef UseSerialDisplay
            SendControlCharToDisplay(dcClearScreen);
            WriteDataToScreen("Discharging...");
            SendControlCharToDisplay(dcMoveToSecondLine);
            WriteIntToScreen("Wait...  ", DischargeCounter, NULL);
          #endif
          if (DischargeCounter == 0)
            CurrentProcessStep = psPreDischarge;
        }
        break;
      case psPreDischarge:
        IgnoreOverVoltageIRQ = true;
        CChargingPWMPort = CChargingPWMOff;
        CurrentProcessStep = psDischarge;
        break;

      case psDischarge:
        #ifdef UseTerminalProg
          WriteDataToScreen("Discharging");
        #endif
        DoDischarge();
        CurrentProcessStep = psStopDischarge;
        break;
      case psStopDischarge:
        #ifdef UseTerminalProg
        WriteDataToScreen("Discharging done");
        #endif

        DelayTime(1000);
        ResetOvervoltageCircuit();

        if (IsCapacitorCharged()) {
          WriteDelayedUserMessage("Discharge done", "Now safety dis.");
          CurrentProcessStep = psInitSafetyDischarge;
        } else {
          WriteDelayedUserMessage("Too low energy", "in capacitor");
          CurrentProcessStep = psEnterIdleState;
        }
        IgnoreOverVoltageIRQ = false;
        CurrentMenuMode = mmCharge;
        break;

      case psShutDown:
        // Well, currently we never get here.... :-)
        break;
  }
  }
}
```