TECHNICAL UNIVERSITY – SOFIA

BRANCH PLOVDIV

FACULTY OF ELECTRONICS AND AUTOMATION

FINAL YEAR PROJECT

BACHELOR'S DEGREE

TITLE: Autonomous data logger

STUDENT: Ilko Milkov Masaldzhiyski
MAJOR: Industrial Engineering
FACULTY NUMBER: 508251
SUPERVISOR: Assoc. Prof. Rumen Popov

SUPERVISOR: **STUDENT:**

PLOVDIV 2013

Table of Contents

Introduction

The aim of this thesis is to introduce an autonomous data logging device and the ability to use it as an embedded system. Some basic terms and concepts, were used, associated with embedded systems, their description and application. A short presentation, is shown, of the historical development of data loggers and examples of their applications. It is important to demonstrate the necessity to use data loggers and their importance to all aspects of manufacturing and control.

Exposition:
Chapter 1 introduction to the term data logger, types of data loggers and logging. Chapter 2 shows the current state of the-art and trends in embedded systems logging. Chapter 3 explains the formulation of the typical design structure of an autonomous data logger. Chapter 4 is the chapter in which is explained the basic theory behind data loggers, analysis of their purpose, synthesis of hardware and software behind the embedded system, applied diagrams and justification of the used methodology. Chapter 5 shows experimental data and followed analysis. Chapter 6 contains the conclusions. Chapter 7 appendixes.

Ilko Milkov Masaldzhiyski, No.508251

Chapter 1. Introduction

1.1. Data logger, description.

A data logger (also “datalogger” or data recorder) is an electronic device that records data over time or in relation to location either with a built in instrument or sensor or via external instruments and sensors. Increasingly, but not entirely, they are based on a digital processor (or computer). They generally are small, portable, and equipped with a microprocessor, internal memory for data storage, and sensors. Some data loggers interface with a personal computer and utilize software to activate the data logger and view and analyze the collected data, while others have a local interface device (keypad, LCD) and can be used as a stand-alone device.

Data loggers vary between general purpose types for a range of measurement applications to very specific devices for measuring in one environment or application type only. It is common for general purpose types to be programmable; however, many remain as static machines with only a limited number or no changeable parameters. Electronic data loggers have replaced chart recorder in many applications.

One of the primary benefits of using data loggers is the ability to automatically collect data on a 24-hour basis. Upon activation, data loggers are typically deployed and left unattended to measure and record information for the duration of the monitoring period. [1]

Data loggers can be battery powered or powered via the electrical network. The advantage of a battery powered data logger is that it can be situated at a remote location which is not accessible, or has no electrical network dispatched. Thus, they are capable of collecting data in really harsh conditions. A drawback is that being battery dependent can lead to a shorter data acquisition. If logging in harsh conditions a supplementary protective case is required in order to protect the data logger itself. Such a protective case can be quite expensive.

A mains powered data logger is almost always directly connected to a personal computer. Advantages are that it is not battery dependent, the measurements can be displayed and analyzed almost instantaneously. A drawback is the immobility of such a system.

1.2 Types of data loggers.

The differences between various data loggers is based on the way that data is recorded and stored. The basic difference between the two data logger types is that one type allows the data to be stored in a memory, to be retrieved at a later time, while the other type automatically records the data on paper, for immediate viewing and analysis. [2]

Both of the types are still used. The data logger with memory offers a larger amount of data to be logged and subsequently analyzed in different manners. It is cheaper than paper, does not require interchange, is smaller in size, can easily trigger and log an alarm and transfer large amounts of data easily.

1.3 Application of data loggers

Depending on the application of data loggers they can be used to log different processes, have their own sensors or simply wait for something to trigger its logging process. Some of the applications are:

- Temperature & humidity
- Voltage, current and power
- Pressure
- Acceleration and speed
- pH and conductivity
- Pressure, strain and force
- Flow and level

Each of these applications requires different hardware and software approach for the data logger realization. Some signals may require additional amplification or rectifying. Depending on the desired application the sensors may actually be placed inside the casing of the data logger.

Chapter 2. State-of-the-Art and trends

2.1 History

The earliest form of data logging involved taking manual measurements from analog instruments such as thermometers and manometers. These measurements were recorded into a written log, along with the time of observation. To view trends over time, people manually plotted their measurements on graph paper. In the late 19th century, it became possible to begin automating this process with machines, and strip chart recorders evolved. Strip chart recorders are analog instruments that translate electrical impulses from sensors into mechanical movement of an arm. A pen is attached to the arm, and long rolls of paper are moved at a constant rate under the pen. The result is a paper chart displaying the parameters measured over the course of time. Strip chart recorders were a great leap over manual data logging, but still had drawbacks. For example, translating the traces on the paper into meaningful engineering measurements was tedious at best, and the data recorded took up reams and reams of paper.

With the development of the personal computer in the 1970s and 80s, people began to use computers for analysis of data, data storage, and report generation. The need to bring data into the PC brought about data loggers – a new special-purpose device for data logging. Data loggers are stand-alone, box instruments that measure signals, convert to digital data, and store the data internally. This data must be transferred to the PC for analysis, permanent storage, and report generation. Data is typically transferred either by manually moving a storage device (such as a floppy disk) from the data logger to the computer or by connecting the data logger to the PC through some communications link such as serial or Ethernet.

In the 1990s, a further evolution in data logging took place as people begin to create PC-based data logging systems. These systems combine the acquisition and storage capabilities of stand-alone data loggers with the archiving, analysis, reporting, and display capabilities of modern PCs. PC-based logging systems finally brought about full automation of the data logging process. The move to PC-based data logging systems was enabled by three technological enhancements:

- Increasing reliability of PCs.
- Steadily decreasing cost of hard drive space on PCs.
- PC-based measurement hardware that could meet or exceed measurement capabilities of stand-alone data loggers. [3]

2.2 Current trends

Today the widest amount of data loggers used, are based on embedded systems.
Embedded systems can easily be implemented as a data logger unit due to number of reasons. They can be completely autonomous, very efficient, handle logging even in an idle state (sleep mode), they have a large amount of inputs/outputs which can be used for a variety of applications.

2.2.1 Low-cost portable data loggers

They provide a cost-effective and easy-to-use solution for data-logging applications. These data loggers easily take measurements with no programming required and display results. They are affordable enough for student use yet powerful enough for more sophisticated measurement applications. [4]

2.2.2 Multichannel data loggers

They provide multiple channels of temperature, load, pressure, voltage, current, or acceleration measurements. They are available in several different form factors and cost more. Often, they are provided with alarm outputs that can signal that a certain condition has happened.

2.2.3 Modular data-logging systems

With these systems, the user can mix different modules into one fully operational unit. Thus, a customized data can be logged with these systems.

2.2.4 Wireless data loggers

They extend measurement capabilities to applications where cables are inconvenient or impractical. Wireless technology can dramatically reduce costs by eliminating cables and installation time. Many applications extend beyond the controlled environment of a lab or research facility. Wireless data loggers are often located in harsh outdoor or industrial environments. In such applications, IP enclosures can protect sensitive measurement equipment. [4]

2.3 Trends in communication

Data loggers can be connected in different ways to a personal computer or a different data logger. Today several standards are used:

2.3.1 USB

USB (Universal Serial Bus) was originally designed to connect peripheral devices such as keyboards and mice, however, it has proven useful for other applications including connecting to measurement devices. USB is easier to use than many other PC buses because computers automatically detect the devices. [4] Nowadays it is hard to find a computer without USB ports.

2.3.2 ETHERNET

Ethernet is the backbone of almost every corporate network and, therefore, is widely available. Although Ethernet requires more configuration and networking knowledge than USB, it provides longer-distance measurements up to 100 m, or further if used with a hub, switch, or repeater. [4]

2.3.3 WIRELESS

Wireless technology extends measurement capabilities to applications where cables are inconvenient or impractical. It can also dramatically reduce costs by eliminating cables and installation time. [4] Different standards for wireless communication can be used. The most common ones are Bluetooth (close distance), Wi-Fi, Infra-Red (objects must be visible), ZIGBEE, XBEE etc. Cost-wise wireless modules are the most expensive due to the necessity of telecommunication which requires antennas (internal or external) and a very strict and complex methodology of communications.

2.4 Trends in storage

2.4.1 Multi Media Card

Currently the most used type of storage is the SD (Secure Digital) card. The MMC/SD/SDHC cards offer large amounts of storage space, while their cost remains low. Communication is done over SPI (Serial Peripheral Interface Bus) which offer high speed transfers. MMC (Multi Media Card) is about the size of a postage stamp: 24 mm × 32 mm × 1.4 mm. It originally used a 1-bit serial interface, but newer versions of the specification allow transfers of 4 or 8 bits at a time. It has been superseded by the SD card, but can still be used in most devices that support SD cards. [5]

2.4.2 Flash memory

Flash memory is an electronic non-volatile computer storage device that can be electrically erased and reprogrammed. In addition to being non-volatile, flash memory offers fast read access times, as fast as dynamic RAM (Random Access Memory), although not as fast as static RAM or ROM (Read Only Memory). Its mechanical shock resistance helps explain its popularity over hard disks in portable devices, as does its high durability, being able to withstand high pressure, temperature, immersion in water, etc. [6] Flash memory is preferable for its fast transfer speeds and its small form factor.

2.5 Trends in connectivity

Nowadays the so called “smart” data loggers come with various numbers of input/output ports. One input can handle different data like Pt100 (Platinum 100), Cu100 (Cuprum 100), thermocouple “J”, thermocouple “K”, thermocouple “S”, thermocouple “R”, linear voltage 0 – 50mV, linear current 0 – 20 mA, linear current 4 – 20 mA. The outputs can be configured as relay outputs, SSR (Solid State Relay) outputs, output for external SSR, analog output, alarm output etc. A specific setup can be given to each input pin through the menu of the data logger or if a personal computer connection is available.

Except these connectivity ports, almost each input has an embedded digital filter, which allows the logger to correctly “see” and handle signals.

The calibration of the device can be manual, given by the user, or can be automatic, where the system handles the processes.

Chapter 3. Formulation of design structure

The 8 main components of data loggers are:

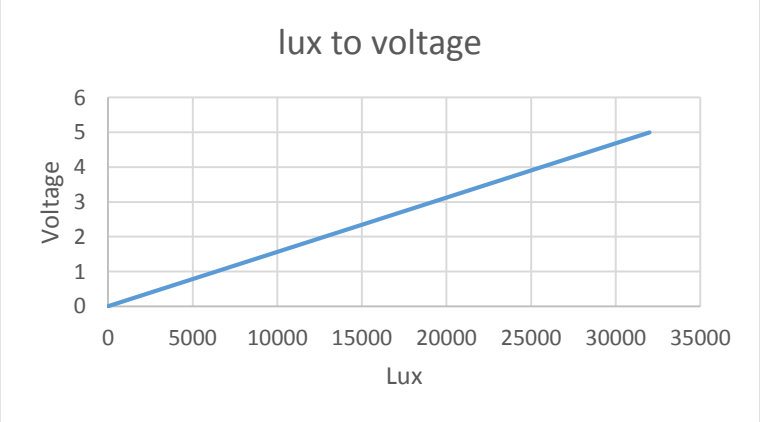
- Input channels
- A/D converters
- Microprocessor
- Memory
- Power supply
- Data output port
- Weatherproof enclosure
- Software [7]

3.1Input channels

The sensor data is send to the data logger via the input channels. They are mostly current or voltage constructed, which means that the input value is actually a scaled voltage/current representation of the measureable value. Data loggers can be single channel or multi-channel depending on the amount of channels they have.
Each channel can be connected to one sensor, thus for two or three sensors we will respectively need two or three channels data logger. Some data loggers do not have input channels accessible to the user due to their internal installation and connection to a sensor. The main type of channels that can be found on a data logger unit are:

3.1.1 Analog inputs

These channels acquire analog data. Almost each data logger is equipped with such channels which, based on their electrical circuitry, can be current or voltage controlled.
Analog signal can be defined as: An analog or analogue signal is any continuous signal for which the time varying feature (variable) of the signal is a representation of some other time varying quantity, i.e., analogous to another time varying signal. [8]
So from the above explanation we can say that an analog signal is continuous in time. An analog sensor will output an electrical signal, which typically is in mA, mV or V depending on the internal structure of the sensor, and the technique used to create its electrical circuitry. Thus if an analog sensor is going to measure light intensity within input range: $10^{-4}\ lux \div 32\,000\ lux$, and it has an output from $0 \div 5$ volts the graphical relationship would be linear as is shown in Figure1



(Figure 1) Linear relationship between lux and voltage in a light sensor

Thus, if the sensor is directed to a moonless, overcast night sky, the voltage would be 0 volts, and if the sensor is pointed towards direct sunlight the voltage would be 5 volts. If the voltage is truly linear the output voltage would be scalar to the input light intensity. Typical analog sensors are used to measure:

- Temperature
- Relative humidity
- Solar radiation
- Barometric pressure
- Wind direction
- Leaf wetness
- Dew point
- Pressure/level
- pH
- Conductivity
- Dissolved oxygen
- Turbidity
- Air pollution

3.1.2 Digital inputs

Digital inputs are the second most common type of input channel found on multi-channel data loggers. [8] They can be defined as a physical signal that is a representation of a sequence of discrete values (a quantified discrete-time signal), for example of an arbitrary bit stream, or of a digitized (sampled and analog-to-digital converted) analog signal. [9]

Graphically the 'curve' of a digital signal would increase or decrease in a 'staircase' like pattern. The signal output has an on/off characteristic and is not continuous. For example, a wind speed sensor may contain a contact switch that closes each time the wind sensor does a revolution. Therefore, if the sensor does only 3/4 of a revolution the switch will not close, only when it does a full revolution it closes. After a second full revolution, it closes again. Graphically, the relationship between contact closures and wind speed would be in steps and could or could not be linear. Knowing the sensor characteristics, or calibration factor, and if the closures are measured over a fixed time interval, the wind speed can be calculated. For instance 100 revolutions per minute may equal a wind speed of 5 km/hr and 200 revolutions per minute may equal a wind speed of 10 km/hr. The data logger accumulates, or totalizes, the discrete closures over a fixed time interval. [7]

Typical digital sensors are used to measure:

- Flow
- Speed
- Level

Nowadays data loggers can be configured to receive PWM signals and measure their duty cycle. Pulse-width modulation (PWM), or pulse-duration modulation (PDM), is a modulation technique that conforms the width of the pulse, formally the pulse duration, based on a modulator signal information. Although this modulation technique can be

used to encode information for transmission, its main use is to allow the control of the power supplied to electrical devices, especially to inertial loads such as motors. [10]

3.2 A/D converters

An analog-to-digital converter (abbreviated ADC, A/D or A to D) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude. This is done using the operations discretization and quantization.

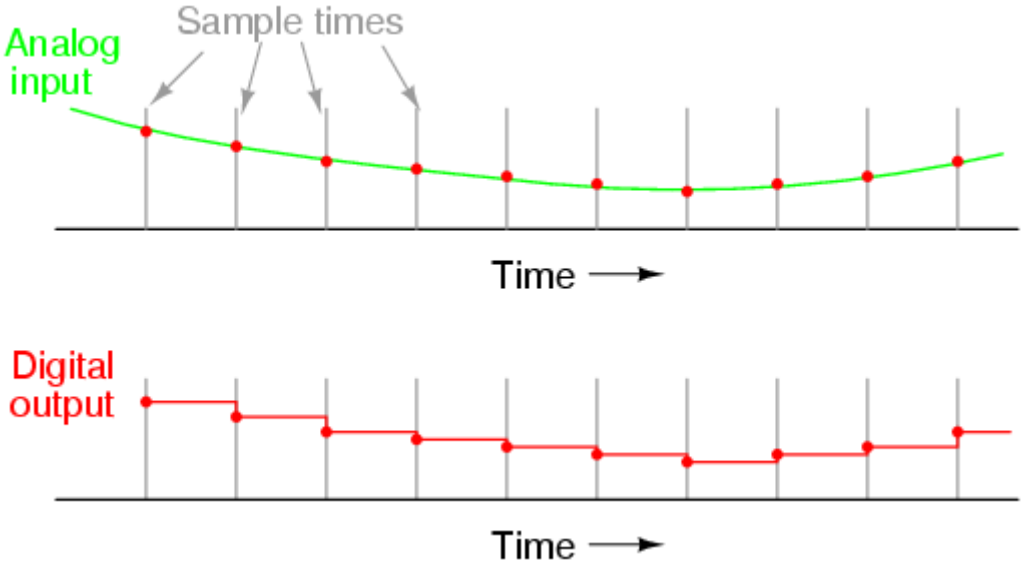
In mathematics, discretization concerns the process of transferring continuous models and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementation on digital computers. [11] When this step is carried out, the frequency of discretization should be correctly chosen. The discretization frequency should be at least 2 times higher than the frequency of the applied signal. This is known as the Nyquist frequency. The Nyquist frequency, named after the Swedish-American engineer Harry Nyquist, is ½ of the sampling rate of a discrete signal processing system. It is sometimes known as the folding frequency of a sampling system. [12] Choosing the correct frequency of discretization can lead to minimizing the cost of the device and optimizing the results achieved

Quantization, in mathematics and digital signal processing, is the process of mapping a large set of input values to a smaller set – such as rounding values to some unit of precision. A device or algorithmic function that performs quantization is called a quantizer. The round-off error introduced by quantization is referred to as quantization error. [13] Quantization can be defined as dividing the maximal amplitude (peak to peak) value of the signal Um to N parts. Each part is defined as:

$$\Delta U = \frac{U_m}{N} \qquad (1)$$

Where N is number of bits.

Figure 2[15] shows the process of discretization and quantization:



(Figure 2) The digital signal can be closer to the original if higher resolution is applied

Each discrete value can be represented with a digital number representing the quantity's amplitude.

The main parameters of an ADC are:

- Frequency of discretization
- Time for conversion
- Strobe time
- Quantization error
- Resolution

Typical types of ADCs that are being used:

- Flash ADC – very fast
- Sigma-delta ADC – High precision
- Dual slope converter- Input signal is averaged
- Successive approximation converter – Good tradeoff between speed and cost.

The resolution of the ADC is varying depending on the type of ADC chosen. That is because some ADCs can be quite expensive if higher resolution is needed. The resolution is defined as the number of output levels it can quantize a signal to. The values are usually stored electronically in binary form, so the resolution is usually expressed in bits. Today's ADCs that are widely used are 8-bit, 10-bit, 12-bit, and 16-bit. Except being used as a stand-alone device, they can be integrated inside a microprocessor.

Normally, the number of voltage intervals is given by (2):

$$N = 2^M - 1 \qquad (2)$$

Where M is the resolution of the ADC in bits [14]
ADC values are sent in the form of bytes. A byte is 8 bits, and it can represent 256 values (from 0 to 255). If the ADC is 10 bit, 12 bit or 16 bit two bytes will be sent. One byte should be read as a high order byte and the other as a low order byte. For example:

8 bit binary representation	8 bit decimal representation
00000000	0
00000001	1
00000010	2
00000011	3
10000000	128
11111110	254
11111111	255

16 bit binary representation	16 bit decimal representation
00000000 00000000	0
00000000 11111111	255
00000001 00000000	256
01111111 11111111	32 767
10000000 00000000	32 768
11111111 11111110	65 534
11111111 11111111	65 535

3.3 Microprocessor

A processor is the logic circuitry that responds to and processes the basic instructions that drive a computer or data logger. A microprocessor is a computer processor on a microchip. It's sometimes called a logic chip. A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called registers. Typical microprocessor operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another. [7] Microprocessors are the core of all embedded systems. Apart from the logical and mathematical operations that they can perform, one of their main advantages is the ability to go in idle state, also called sleep mode.

3.3.1 Sleep mode

The sleep mode of microprocessor is putting the chip in a state with very low current draw, around ~10 nA (nano amperes). Waking up the chip can be achieved using the interrupt vectors that reside inside the microprocessors.

Interrupt vectors can be internal, external, maskable or non-maskable. Based on their type they can be used in various applications.

3.3.2 Internal interrupts

Internal interrupts are triggered inside the microprocessor and can wake the microprocessor up, or pause the program in its current state, handle the interrupt and continue with the program. Internal interrupts can be triggered from the overflow of a timer, send byte acknowledgement, real-time-calendar/clock or user created interrupt flag.

3.3.3 External interrupts

External interrupts are triggered outside of the microprocessor. They also can wake the microprocessor up, pause the program and continue further after being handled. External interrupts can be signals arriving on the input ports (buttons, alarm conditions, incoming data, real-time-clock etc.).

Before an interrupt can be processed the programmer must enable interrupts and create functions that handle these interrupts. When an interrupt happens, and the programmer has enabled it, the interrupt vector shows the location of the function handling that interrupt. Register values are saved, program counter is saved, and are moved inside the stack. The program continues with the completion of the interrupt sub-routine (function). After its completion the register values are read from the stack, so is the program counter, and the program continues from its previous state, or goes back again to sleep mode.

A remark should be made here that if the programmer sets the interrupt to go in an endless loop, the program will never continue from its previous state, and a reset may be required.

3.3.4 Clock frequency

Microprocessors have an internal clock that can operate at high frequencies (up to 8 MHz). For higher clock frequencies an external oscillator must be used and the correct programming techniques must be implemented in order for its configuration. New embedded microprocessors can reach up to GHz clock frequencies. With such frequencies data logging is made easy, but this does not offer just easy logging but also the solution to real-time analyzing, imaging and post-processing.

3.3.5 Internal ADC/DAC/Compare/PWM/Capture

The microprocessors are supplied with internal modules that facilitate the input and output of signals.

- Internal ADCs can reach up to 16 bit resolution with low errors.
- DAC (Digital-to-analog-converter) modules can give the ability to output analog voltage.
- Capture modules can count incoming signals easily and if they are combined with timer modules, data like RPM and Speed can be measured.
- PWM (Pulse Width Modulation) modules offer precise power control to any peripheral device connected.
- Newer microprocessors have SPWM (sinusoidal PWM) which easily incorporates sine-wave output.
- Compare modules give the ability to compare signals fast and easy.

3.3.6 Implemented communication protocols

Microprocessors have built-in communication modules like:

- 3-wire/4-wire SPI
- I^2C with Multi-Master/Slave
- UART modules (Universal Asynchronous Receive/Transmit)
  - RS-232
  - RS-485
  - LIN 1.2
  - IrDA
- Parallel Master Slave Port (PMP/PSP)

3.3.7 Internal memory

On-board FLASH program memory and EEPROM (Electrically Erasable Programmable Read Only Memory) offers various approaches to handle different tasks. The high amounts of on-board storage makes possible the creation of complex programs and keeping data without the requirement of external memory. If needed, external memory can be connected to the microprocessor and without hassle enable it to continue logging or simply completing the given tasks.

3.3.8 C ready

Microprocessors are now not only programmable on Assembler but they can be programmed in higher level languages like C and C++. Their instruction set architecture is C compiler optimized, which can leady to overall performance over older microprocessors.

3.4 Memory

Two types of memory are used in data loggers:

- RAM (Random Access Memory)

Unlike a PC's RAM which is used as a 'workshop area', a data logger can use RAM to store data (readings from the input channel). RAM chips are inexpensive but must be battery backed up in order to retain the data. RAM chips are downloaded via the serial port of a PC. [7] Even though that RAM is faster than EEPROM it is not so widely used given the fact that it is a volatile memory.

- EEPROM

Developed for data loggers in the late 1980's EEPROM memory does not need to be backed up by a battery. Many data loggers use EEPROM chips for both storing the operating system of the microprocessor, as well as for data storage. An EEPROM chip can be programmed, read (stored data) and erased via the serial port of a PC. [7]

- SD Card

Secure Digital or (SD) is a non-volatile memory card format for use in portable devices, such as mobile phones, digital cameras, GPS navigation devices, and tablet computers. [16] The SD card has some advantages compared to the EEPROM which are important for logging:

- Accessibility – An SD card is much easier and faster to read from and access. Taking it out from the device, putting it in a card reader and reading the data is convenient and easy.
- Capacity – SD cards have enough free space to log data for years ahead.
- Ease of implementation – many commercial and open source applications use FAT drivers that can be installed for many microcontrollers.

3.5 Power supply

A feature that clearly distinguishes data loggers from PC's is the low power requirements of data loggers. Data loggers are designed to operate in remote locations for long periods of time void of main AC power. Most data loggers require a 12 VDC power source. Battery capacities are measured in Milli-Amp hours (mAh) which determines the length of time that the battery can provide power for a given load. Increased capacity requires greater battery size and weight. [7]

3.5.1 Non-rechargeable batteries

- Lithium batteries – Lithium batteries are disposable (primary) batteries that have lithium metal or lithium compounds as an anode. They stand apart from other batteries in their high charge density (long life) and high cost per unit. Depending on the design and chemical compounds used, lithium cells can produce voltages from 1.5 V (comparable to a zinc–carbon or alkaline battery) to about 3.7 V.[17]
- Alkaline batteries – Alkaline batteries are a long-lived dry cell with an alkaline electrolyte that decreases corrosion of the cell. There are many uses for alkaline batteries. Since they come in many sizes, they can be useful in many portable electronic devices. [18]

3.5.2 Rechargeable batteries

Sealed lead acid batteries - A sealed lead acid battery or gel cell is a lead acid battery that has the sulfuric acid electrolyte coagulated (thickened) so it cannot spill out. They are partially sealed, but have vents in case gases are accidentally released for example by overcharging. They can be used for smaller applications where they are turned upside down. They are more expensive than normal lead acid batteries, but they are also safer. [19]

3.6 Data output port

Most data loggers communicate with a PC via a serial port, which allows data to be transmitted in a series (one after the other). The RS-232 interface has been a standard for decades as an electrical interface between data terminal equipment, such as a PC, and data communications equipment employing serial binary data interchange, such as a data logger or modem. Data can be sent in both directions, and many loggers use 9600 baud as a standard communication speed. Since the RS-232 is so popular, many modems are available that can be connected to a data logger to retrieve data remotely or to program the data logger. [7] Newer loggers communicate via USB or wireless communications (WI-FI, BLUETOOTH, IR, etc.).

3.7 Weatherproof enclosures

Since data loggers are completely remote and are left on places that can have harsh conditions, a protective enclosure is required to keep the electronics safe. The enclosure must be weatherproof and can also be EMI (electromagnetic interference) resistant.

3.8 Software

Proprietary software is usually required to program and download data from a data logger. Data logging functions such as sensor scan rate and scaling, log interval, communication protocol and output format (Excel, ASCII, plot, etc.) are programmed using software loaded on a PC. Most loggers available today have a Windows software package and many manufactures have developed software to run on a Palm or similar device. Usually only one license is required per user regardless of the number of loggers they operate. [7]

Chapter 4. Basic theory, analysis, synthesis, hardware and software development, diagrams, justifications

4.1Hardware solution, basic theory, analysis, synthesis, diagrams and justifications

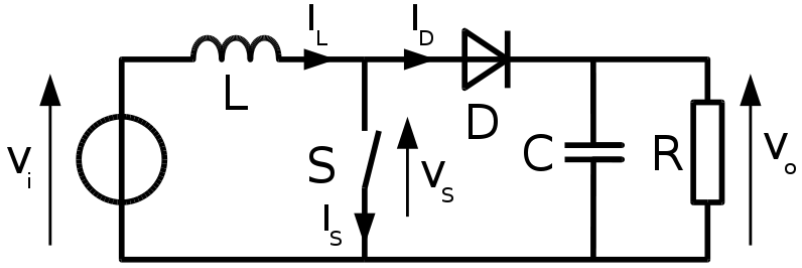
The hardware solution will cover choosing and configuring the main elements of a data logger:

- Power supply
- Microprocessor
- Analog inputs protection circuitry
- Digital inputs protection circuitry
- Digital outputs protection circuitry
- LCD
- SD card
- PCB

4.1.1 Power supply

The requirements of the power supply are input of 2.4 volts (battery powered) and output of 3.3 volts. The efficiency should be above 90% and the output must be filtered. Effectively ensuring continuous supply of energy to the microprocessor.
The battery will be a 2.4 volts Lithium battery. The cost is higher but the effective power capacity is much larger and will provide a greater lifespan of the data logger.
In order to achieve 3.3 volts from a 2.4 volts a DC-DC converter must be used. DC-to-DC converters are used to convert one DC (direct current) voltage to another. The main types of DC-DC converters are Buck and Boost converters. Buck converters offer a step-down voltage converter, which lowers the initial level. Boost converters offer a step-up DC-DC conversion, which increases the initial voltage level. Thus the power supply must be a
A boost converter (step-up converter) is a DC-to-DC power converter with an output voltage greater than its input voltage. It is a class of switched-mode power supply (SMPS) containing at least two semiconductor switches (a diode and a transistor) and at least one energy storage element, a capacitor, inductor, or the two in combination. Filters made of capacitors (sometimes in combination with inductors) are normally added to the output of the converter to reduce output voltage ripple. [20]

The typical schematic of a boost converter is shown on Figure 3.



(Figure 3) Boost converter

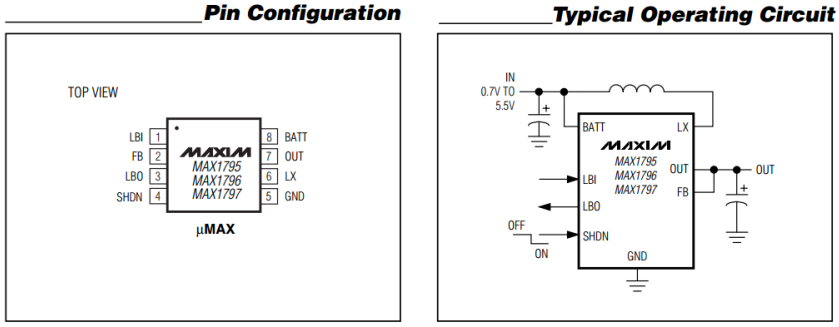
The basic principle of the Boost converter can be explained with its two switching states:

- During the On-state, the switch S is closed, which increases the current in the inductor. Current cannot pass through the diode due to its higher resistance than the line.
- During the Off-state, the switch is open and the only path for the current to discharge is through the diode D, the capacitor C and the load R. This results in transferring energy accumulated during the On-state into the capacitor.

If the switch is cycled fast enough, the inductor will not discharge fully in between charging stages, and the load will always see a voltage greater than that of the input source alone when the switch is opened. Also while the switch is opened, the capacitor in parallel with the load is charged to this combined voltage. When the switch is then closed and the right hand side is shorted out from the left hand side, the capacitor is therefore able to provide the voltage and energy to the load. During this time, the blocking diode prevents the capacitor from discharging through the switch. The switch must of course be opened again fast enough to prevent the capacitor from discharging too much. [20]

The company Maxim Integrated offers a big variety of boost converters which are easily configured and implemented.

MAX1796 is a Boost converter which offers over 95% efficiency and a Low-Noise feature. The circuit has reduced EMI which can enable the circuit to operate in noise-sensitive applications. The pin configuration and typical operating circuits are shown on Figure 4.



(Figure 4) MAX1796 pin-out and typical operating circuit

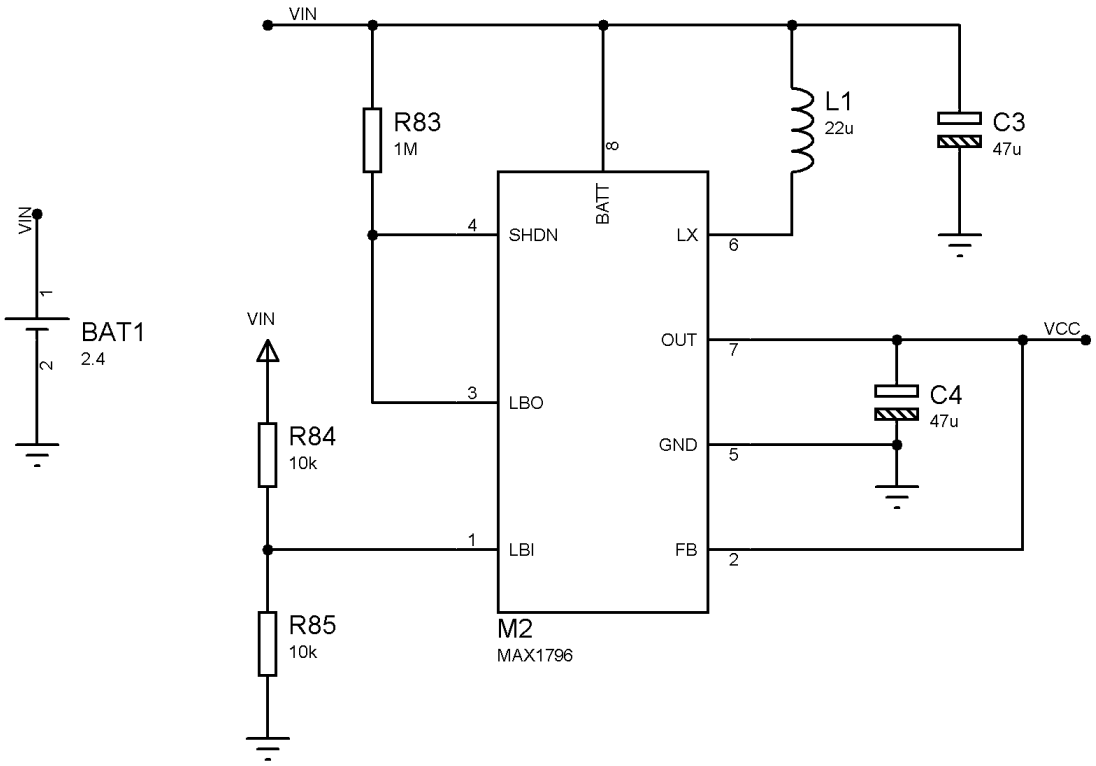
MAX1796 offers shutdown mode if the battery discharges and gets bellow 0.85V. This is done using the comparator LBI/LBO (Low-battery comparator input and Low-battery comparator output). For further understanding on how MAX1796 works the datasheet is reviewed. The pin description is shown on Figure 5.

Pin Description

PIN	NAME	FUNCTION
1	LBI	Low-Battery Comparator Input. Internally set to trip at +0.85V. This function remains operational in shutdown.
2	FB	Dual-Mode™ Feedback Input. Connect to GND for preset 5.0V output. Connect to OUT for preset 3.3V output. Connect a resistive voltage-divider from OUT to GND to adjust the output voltage from 2V to 5.5V.
3	LBO	Low-Battery Comparator Output, Open-Drain Output. LBO is low when V_{LBI} < 0.85V. This function remains operational in shutdown.
4	SHDN	Shutdown Input. If SHDN is high, the device is in shutdown mode, OUT is high impedance, and LBI/LBO are still operational. Connect shutdown to GND for normal operation.
5	GND	Ground
6	LX	Inductor Connection
7	OUT	Power Output. OUT provides bootstrap power to the IC.
8	BATT	Battery Input and Damping Switch Connection

(Figure 5) Pin description of the MAX1796

Thus the chosen Power Supply circuit will be Figure 6

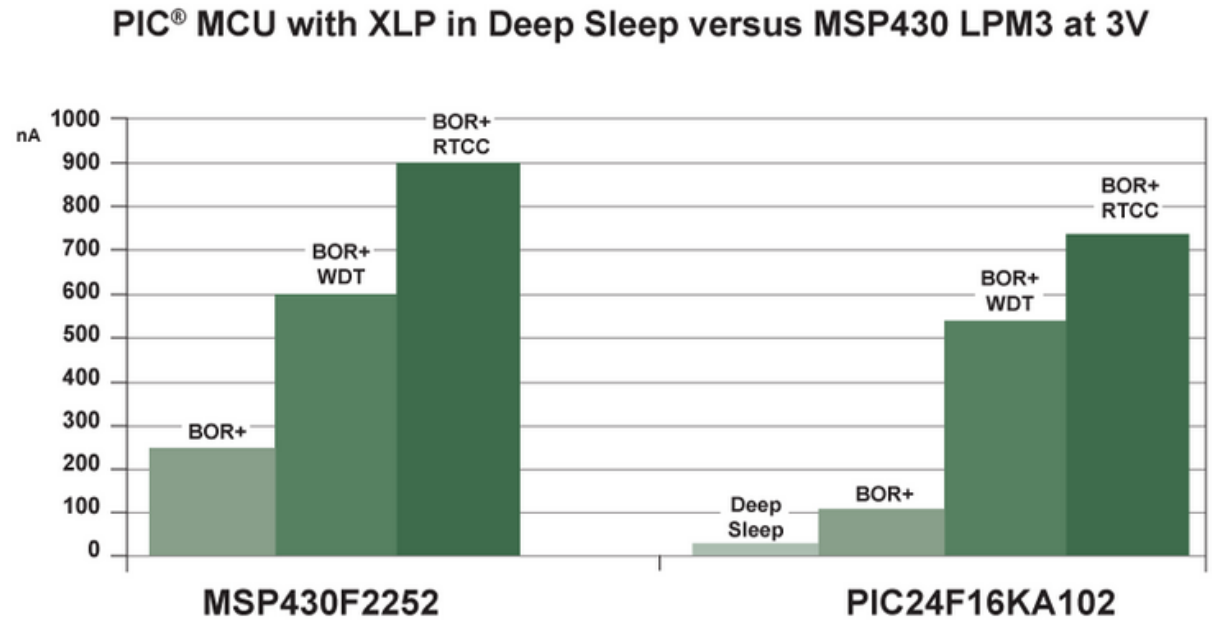


(Figure 6) Used power supply circuit

As a conclusion should be stated that the choice of this circuit gives a steady, filtered output of 3.3 volts with an input of 2.4 volts.

4.1.2 Microprocessor

The requirements on the microprocessor are low power consumption, large amount of input/output ports, internal 8-channel ADC with 10-bit resolution, high operational frequency, internal and external interrupts, the presence of a RTCC, and the ability to communicate via the standard communication protocols (UART, SPI).
Microchip provides solutions for the entire performance range of 8-bit, 16-bit, and 32-bit microcontrollers, with a powerful architecture, flexible memory technologies, comprehensive easy-to-use development tools, complete technical documentation and post design in-support through a global sales and distribution network. [21] The chips that they offer use the nanowatt XLP Technology which is unrivaled with its competitors as is shown in Figure 7.



(Figure 7) PIC MCU versus competition MCU

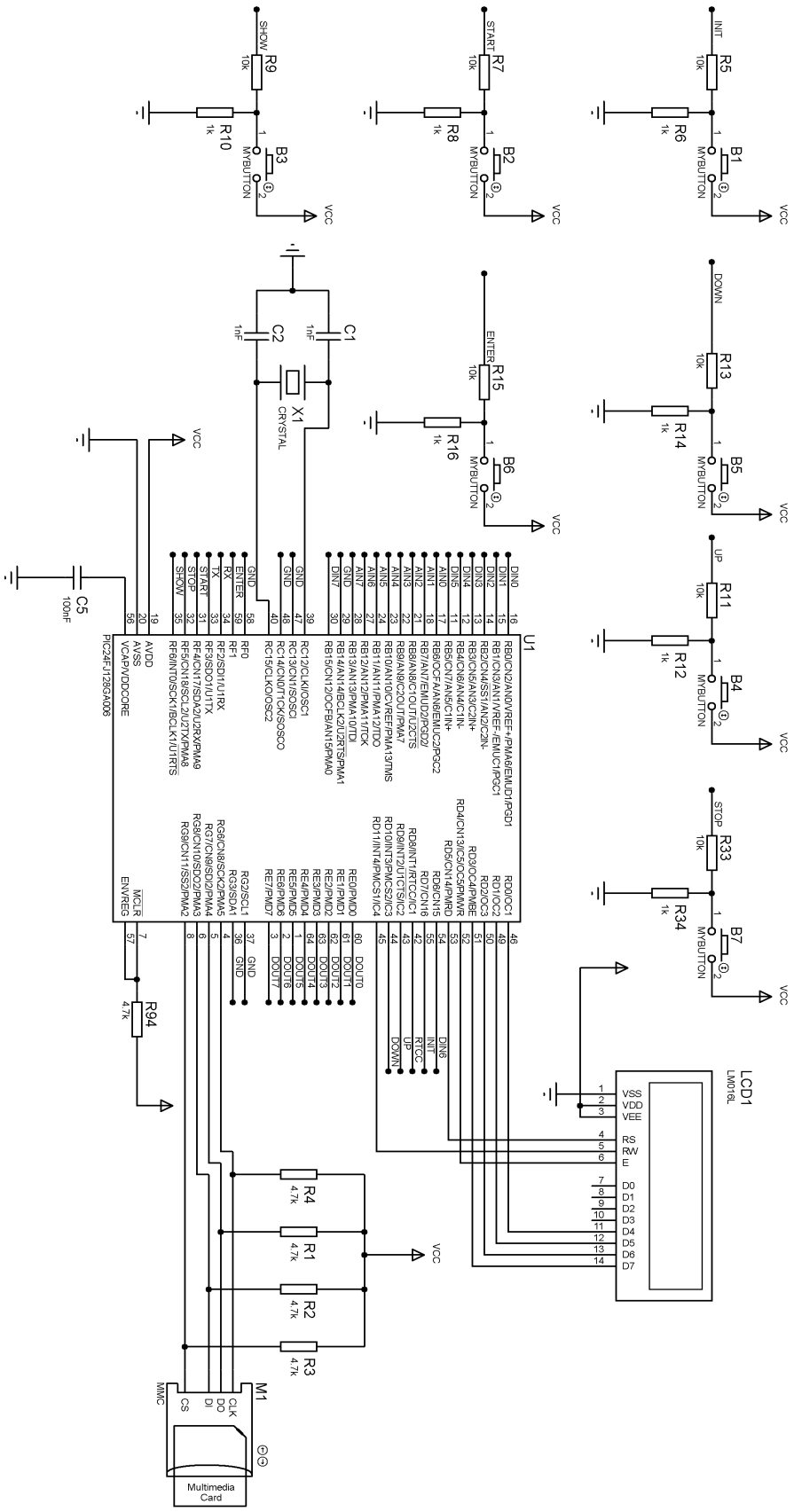
The constructed data logger requires as little as possible energy to be consumed. Thus the choice of PIC microprocessors is evident.
A chip that meets all of the requirements for the data logger assembly is PIC24FJ128GA006.
PIC24FJ128GA006 has:

- 16 ADC channels that are 10-bit
- 53 Input/Output pins
- 2 UART modules
- 2 SPI modules
- 1 Hardware RTCC
- low operating voltage (3,3 volts)
- nanowatt Technology

The microprocessor has enough pins to handle an LCD, has 8 digital inputs, 8 analog inputs, 8 digital outputs, and service buttons.
After carefully investing the datasheet of the microprocessor, the circuit was designed in accordance to the specifications:

- MCLR and ENVREG must be connected to the supply voltage for a normal start and operation of the microprocessor.
- VCAP must be grounded through a capacitor with value of 100nF.
- Analog ground and analog power should be connected to VDD and VSS respectively.
- The oscillator must be configured with the crystal being connected to OSC1 and OSC2 with capacitors grounding each of the pins. The desired frequency of operation is 20 MHz which requires the two capacitors to be with values of 22 pF.
- The interrupt-activating buttons are connected to Interrupt-On-Change pins.
  - Initialization button is connected to CN16.
  - Start of logging button is connected to CN17.
  - Stop of logging button is connected to CN18.
  - Show digital inputs button is connected to INT0 which is an external interrupt pin.
  - Enter button is connected to pin RF0.
  - Up button is connected to pin RD9
  - Down button is connected to pin RD10
  - Buttons Up, Down and Enter do not trigger an interrupt.
- The digital inputs are also connected to Interrupt-On-Change pins. They are located on pins CN2, CN3, CN4, CN5, CN6, CN7, CN15 and CN12. The pins are marked as DIN0 – DIN7
- The analog inputs are connected to the analog channels, starting from pin AN6 to pin AN13, located at PORTB. They are marked as AIN0 – AIN7
- The digital outputs (alarm) are connected on PORTE from RE0 to RE7. They are marked as DOUT0 – DOUT7
- The SD card will be connected to the second SPI channel located on pins RG6 - Clock, RG7 – Data In, RG8 – Data Out, RG9 – Chip select
- The LCD will be connected at RD0, RD1, RD2, RD3, RD4, RD5, RD11
- RTCC pin is an alarm pin which can be connected. Every time an interrupt occurs this pin will go HIGH.
- An RS-232 debug console can be connected to pins RF2 - RX and RF3 – TX. The microprocessor will output current status at baud 9600.
- All other unconnected pins will be grounded for EMI resistance.

It should be stated that the connection diagrams of the buttons are as follows: one resistor should be connected to ground and to pin 1 of the button, taking as granted that pin 1 of the button is connected to the microprocessor. This is done because we need to ensure that if the button is not pressed we will have a logic LOW on the inputs of the microprocessor. Another resistor must be connected in series to pin 1 and the microprocessor to ensure current limiting. The value chosen is 1k for grounding and 10k for the series resistor, which are standard values picked for button configurations.
The schematic of the microprocessor connections is shown on Figure 8.



(Figure 8)Microprocessor and peripherals connected

4.1.3 Analog inputs protection circuitry

The analog inputs of the on-board microprocessor are 5V tolerant. Thus the analog input voltage should not exceed that value. An easy and cost efficient way is to add a Zener diode. A Zener diode is a diode which allows current to flow in the forward direction in the same manner as an ideal diode, but will also permit it to flow in the reverse direction when the voltage is above a certain value known as the breakdown voltage, "zener knee voltage", "Zener voltage" or "avalanche point". [22]
The Zener diode acts as a voltage limiter. It is picked with the closest standard value of 5.1 volts. A current limiting resistor must be put before the Zener diode, which limits the current. When calculating the current limiting resistor we take as granted that the maximum analog voltage that the user can apply is 12 volts, even though that the assignment is 5 volts. This would be a mistake by the user which can happen. The maximum current for the Zener diode is 49 mA (from the datasheet), thus:

$$I = \frac{U}{R} = \frac{12}{R} = 0.049\,mA \quad (3)$$

$$R = \frac{12}{0.049} = 244\,\Omega \quad (4)$$

The resistance should be chosen above 244 Ohms. The closest higher standard value is 270 Ohms but in order to back up the Zener diode we increase the resistance to 330 Ohms. The power factor of the resistor is calculated based on the resistance and the maximum voltage applied:

$$P = \frac{U^2}{R} = \frac{12^2}{330} = 0.43\,W \quad (5)$$

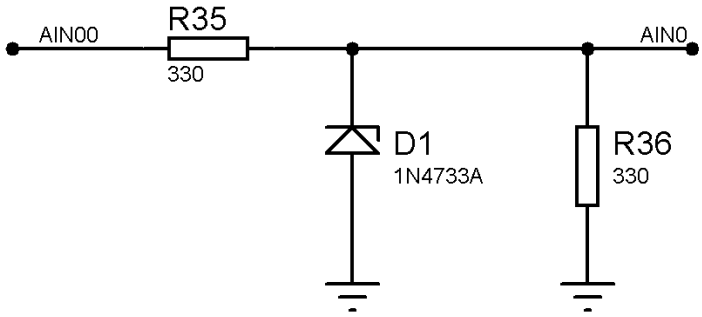
The power factor of these resistors must be a standard value. The closest higher standard value is 0.6W.
The microprocessor analog pins are 5 volts tolerant but since the power supply is 3.3 volts a voltage divider must be set up after the analog input in order to lower the value based on maximum analog voltage measurements. The voltage is going to be divided by 2 to ensure that even signals with voltage raiting of 5 volts are going to be properly captured. This means that after the voltage divider the maximum voltage would be approximately 2.5 volts. In order to properly calculate the output voltage we use (6).

$$Vout = Vin\frac{R2}{R1+R2} \quad (6)$$

Replacing the values in the formula we receive that R2 must be equal to R1 in order to achieve a 50% divider which is proven bellow:

$$2.5 = 5 * \frac{330}{330+330} \quad (7)$$

The configuration of the analog inputs protection circuitry is shown on Figure 9



(Figure 9) Analog inputs protection

4.1.4 Digital inputs protection circuitry

The digital inputs of the on-board microprocessor are 3.3V tolerant. Thus the digital input voltage should not exceed that value. An easy and cost efficient way is to follow the previous protection circuitry design with adding a Zener diode. The digital inputs though require GND (ground) separation from the input source, due to the possibility of high voltage peaks. After the Zener diode, an optocoupling device must be used. In electronics, an opto-isolator, also called an optocoupler, photocoupler, or optical isolator, is a component that transfers electrical signals between two isolated circuits by using light. Opto-isolators prevent high voltages from affecting the system receiving the signal. Commercially available opto-isolators withstand input-to-output voltages up to 10 kV and voltage transients with speeds up to 10 kV/μs. A common type of opto-isolator consists of an LED and a phototransistor in the same package. [23] The chosen optocoupling device is PC817. It covers the necessary characteristics of the input and output signals needed, which are:

- Possibility of two separate grounds
- Low current powered LED (5 mA)
- High output currents (50 mA)

The Zener diode acts as a voltage limiter. It is picked with the closest standard value of 5.1 volts. The purpose of the Zener diode is to protect the LED inside of the optocoupler by limiting the voltage applied. A current limiting resistor must be put before the Zener diode, which limits the current of both the Zener and the LED of the optocoupler. When calculating the current limiting resistor we take as granted that the maximum digital voltage that the user can apply is 12 volts. The optocoupler's phototransistor is going to be fed by the VCC, so the output of the optocoupler is going to be 3.3 volts. The maximum current for the Zener diode is 49 mA (from the datasheet), and the maximum current for the LED of the optocoupler is 50 mA and the forward voltage is 1.2 volts thus:

$$I = \frac{U}{R} = \frac{5}{R} = 0.049 + 0.05 = 0.099\,mA \qquad (6)$$

$$R = \frac{12}{0.099} = 121\,\Omega \qquad (7)$$

The resistance should be chosen above 121 Ohms. The closest higher standard value is 150 Ohms but in order to have less power dissipation as heat and given the fact that the minimum current for the LED is 5mA the value of 1000 Ohms is chosen.

The power factor of the resistor is calculated based on the resistance and the maximum voltage applied:

$$P = \frac{U^2}{R} = \frac{12^2}{1000} = 0.144\,W \qquad (8)$$

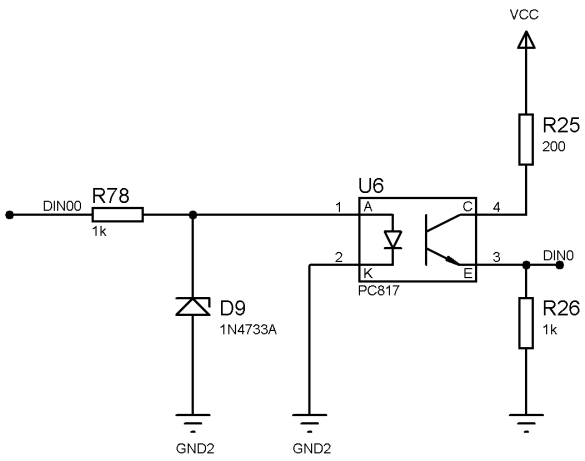
The power factor of these resistors must be a standard value. The closest higher standard value is 0.25W.
The microprocessor digital pins are 3.3 volts tolerant, thus they can be connected to the phototransistor of the optocoupler. Despite the voltage levels being okay, the current must be limited. Since the emitter is going to be used as the output, resistance must be added to the collector line of the phototransistor. The maximum sink current of the microprocessor is 18 mA. Given these values it can easily be calculate that the resistor value should be atleast:

$$I = \frac{U}{R} = \frac{3.3}{0.018} = 183\,Ohms \qquad (9)$$

The closest higher standard value is 200 Ohms. The power factor of the resistor is calculated based on the resistance and the maximum voltage applied.

$$P = \frac{U^2}{R} = \frac{3.3^2}{200} = 0.054\,W \qquad (10)$$

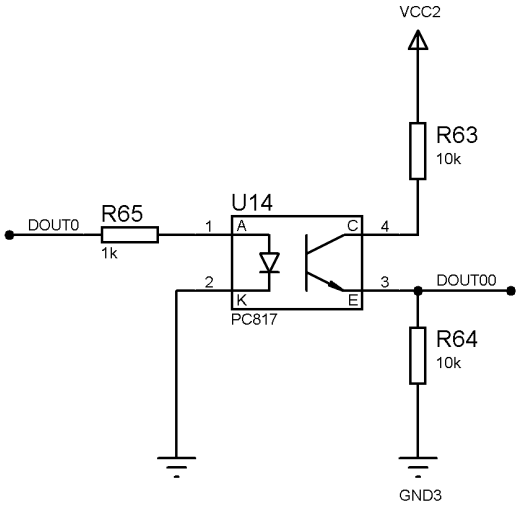
The closest higher standard value chosen is 0.125 W
A grounding resistor should be used from the emitter. His purpose is to ground the input of the microprocessor, when no input signal is given. A typical value is 1000 Ohms in 0.125W package.
The configuration of the digital inputs protection circuitry is shown on Figure 10



(Figure 10) Digital inputs protection

4.1.5 Digital outputs protection circuitry

When constructing the digital outputs protection circuitry we must take in consideration what was discussed for the digital inputs protection circuitry. The grounds must be separated with an optocoupler.
The output voltage of the microprocessor is 3.3 volts and the maximum source current is 18mA. Resistor with value of 1000 Ohms is placed after the output of the microprocessor. After the optocoupler two resistors with standard values of 10k are put on the emitter and on the collector. The phototransistor should be supplied with its own VCC and GND signals in order to operate.
The configuration of the digital outputs protection circuitry is shown on Figure 11.



(Figure 11) Digital outputs protection

4.1.6 LCD

A liquid-crystal display (LCD) is a flat panel display, electronic visual display, or video display that uses the light modulating properties of liquid crystals. [24]The LCD choice should be made on the basis of low current consumption and effectiveness for calibration and monitoring. LM016L is a low-powered LCD display that can operate with 3.3 volts. It is 16 columns with 2 rows and can be used with 4 data communication lines which is ideal for the I/O connections of the microprocessor

4.1.7 SD card

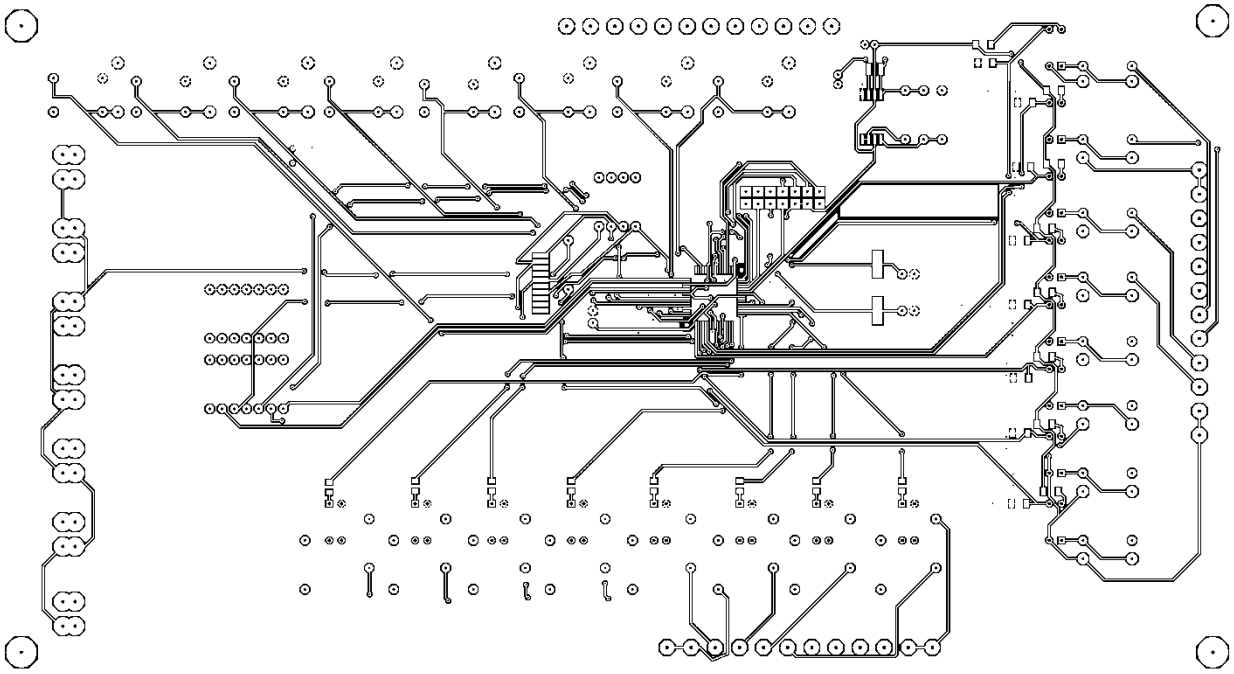
The choice of the SD card is based on several factors. The logged data can be logged on an SD card formatted (with file system) or unformatted (disk dump). For easier post-access the data will be formatted. For the operation of the autonomous data logger, a 2 GB (gigabytes) SD card is enough for logging. For example if every 1 minute we log, and one log takes 512 bytes, 365 days have exactly 525 600 minutes (logs). 525 600 logs of 512 bytes each is equal to around 256 MB (megabytes). So 256 megabytes are worth one year of logging every minute. Thus 2 GB (gigabytes) are worth around 8 years of autonomous logging. Size of 2 GB can easily be implemented with the file system FAT16, which is supported on all Windows versions starting from XP and continuing to the latest versions. Linux can also handle the operation of such files. FAT16 will be further explained in the software part of this chapter.
The SD card will communicate with the microprocessor via 4 lines, while using the SPI communication protocol. Data In, Data Out, Clock and Chip Select are the lines that are

used. They should be Pulled-Up with resistors to the VCC voltage levels, to ensure proper communication. The SD card will be put in an SMD SD-card socket.

4.1.8 PCB

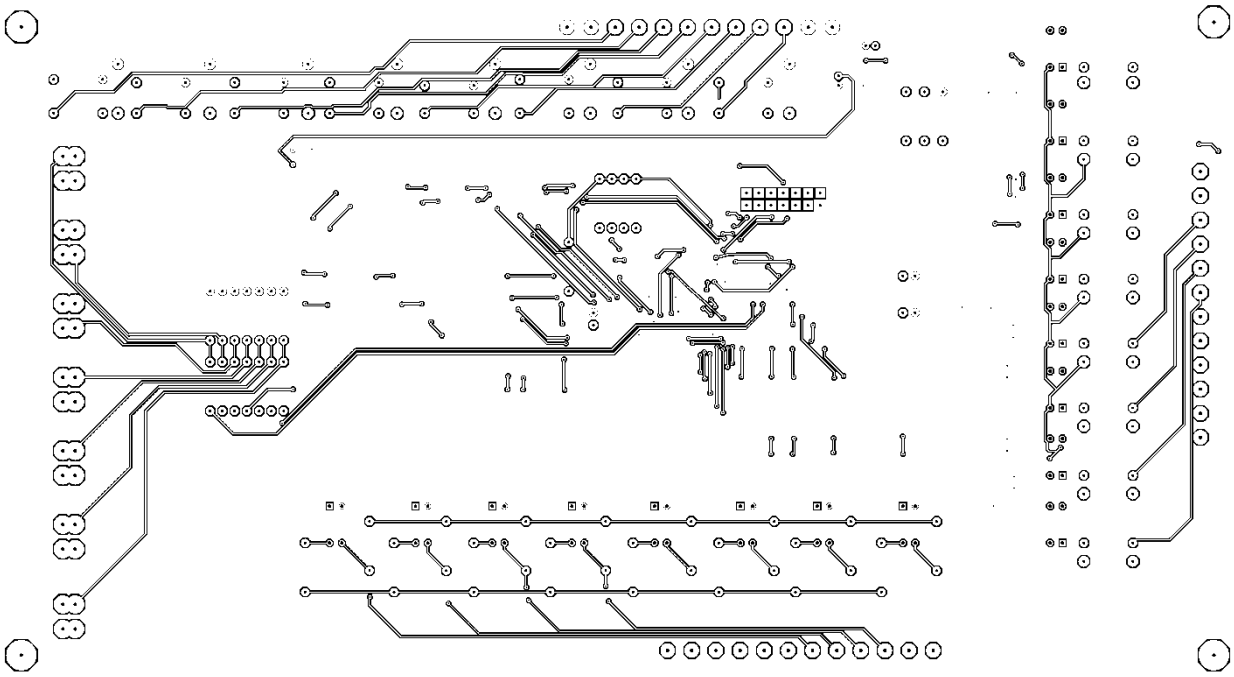
For the construction of the PCB was used a special software tool. ARES Proteus v.7.10. The board was developed following all the criteria on correct placement and positioning of elements, weight and I/O ports. Thus top layer, bottom layer, top silk and 3d visualizations were created:

- Top layer



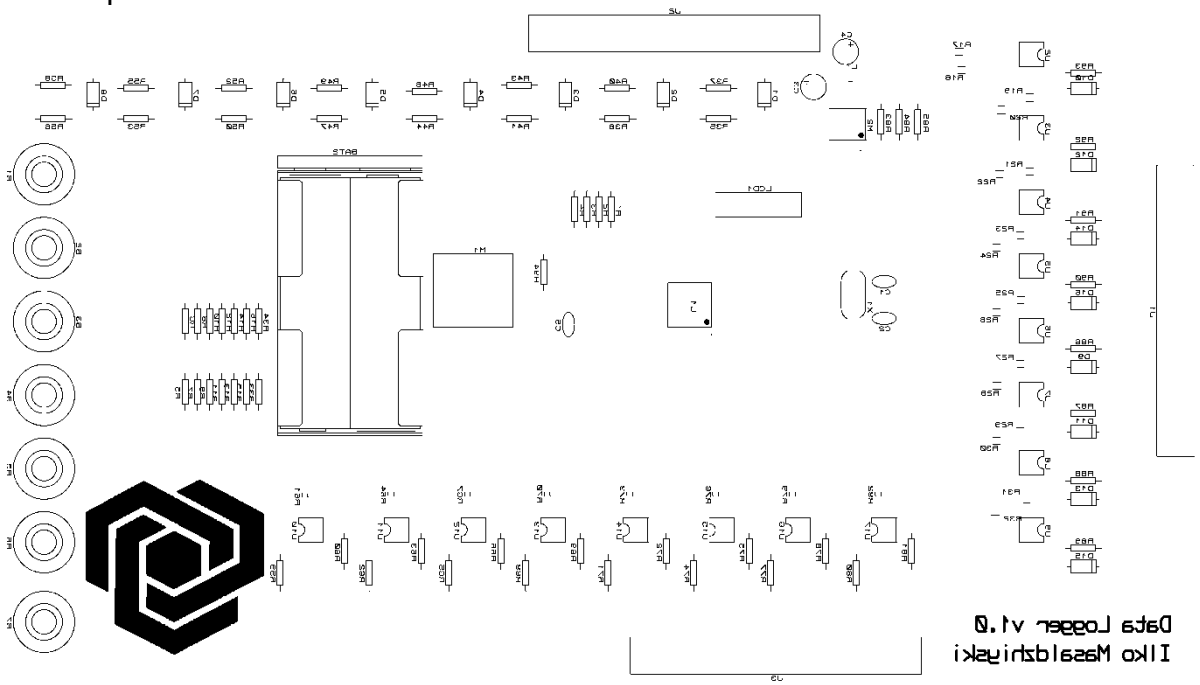
(Figure 12) Top layer of PCB

- Bottom layer



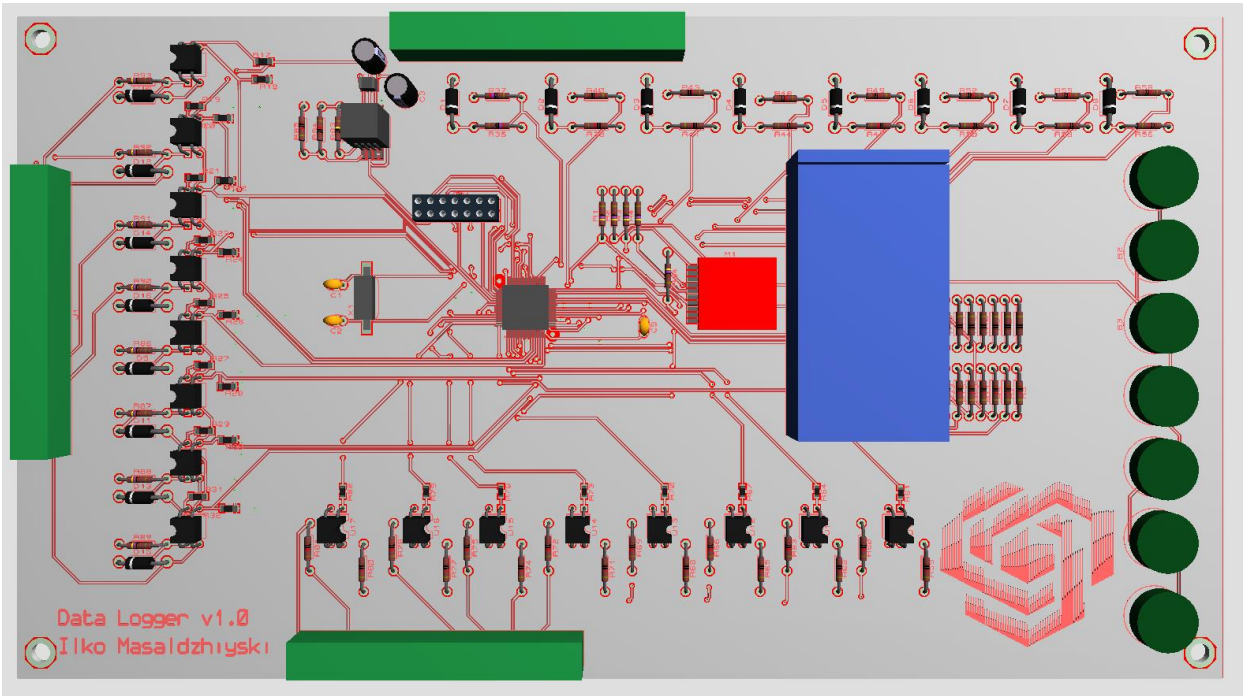
(Figure 13) Bottom layer of PCB

- Top silk



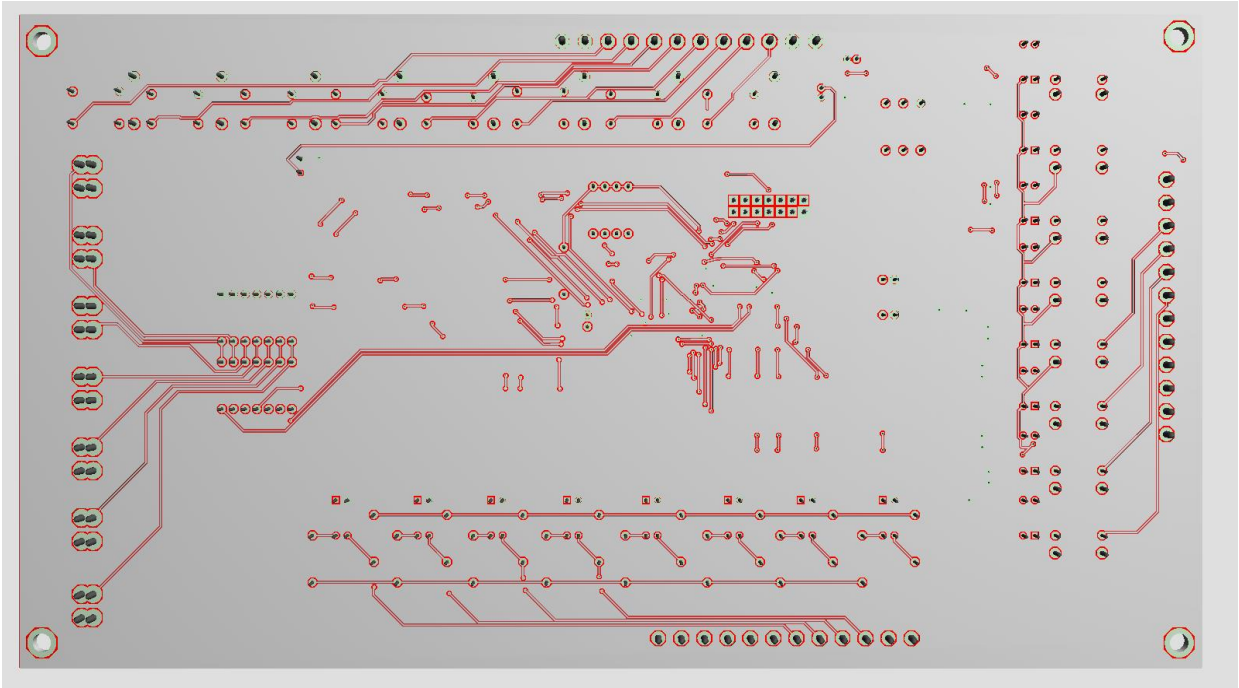
(Figure 14) Top silk layer of PCB

- 3D Front



(Figure 15) 3D front view. Image generated with ARES PROTEUS 7.10

- 3D Back



(Figure 16) 3D back view. Image generated with ARES PROTEUS 7.10

4.2Software solution, basic theory, analysis, synthesis, diagrams and justifications

The software solution will cover the main techniques used to program a data logger.

4.2.1 Choice of programming language and compiler

When choosing a programming language for microprocessors one should consider Assembler and a C-like language. Despite the speed and optimization drawbacks the C-like language would best fit the solution. The microprocessor itself is architecturally and instruction-wise C-ready. Yes assembly is quite faster and not so memory heavy but the benefits of writing on C is the use of functions, pointers, arrays, and easy definable algorithms.

When choosing a C-compiler for programming a PIC Microchip microprocessor it is not necessary to use their own C compilers. First they are expensive and even though they provide excellent code optimization it is not of such necessity due to the large amount of storage space. Other than that of Microchip's compiler, the CCS C compiler offers a large variety of pre-built libraries, examples and very good overall functionality and code optimization. A drawback of that compiler is that it is not ANSI-C thus some of the standard C coding may not work as it should. Despite that, the manual offers full coverage of everything that one might need in the process of programming.

The programming environment will be MPLAB X IDE, and the programmer itself will be PICKit v.2.0.

4.2.2 FAT16

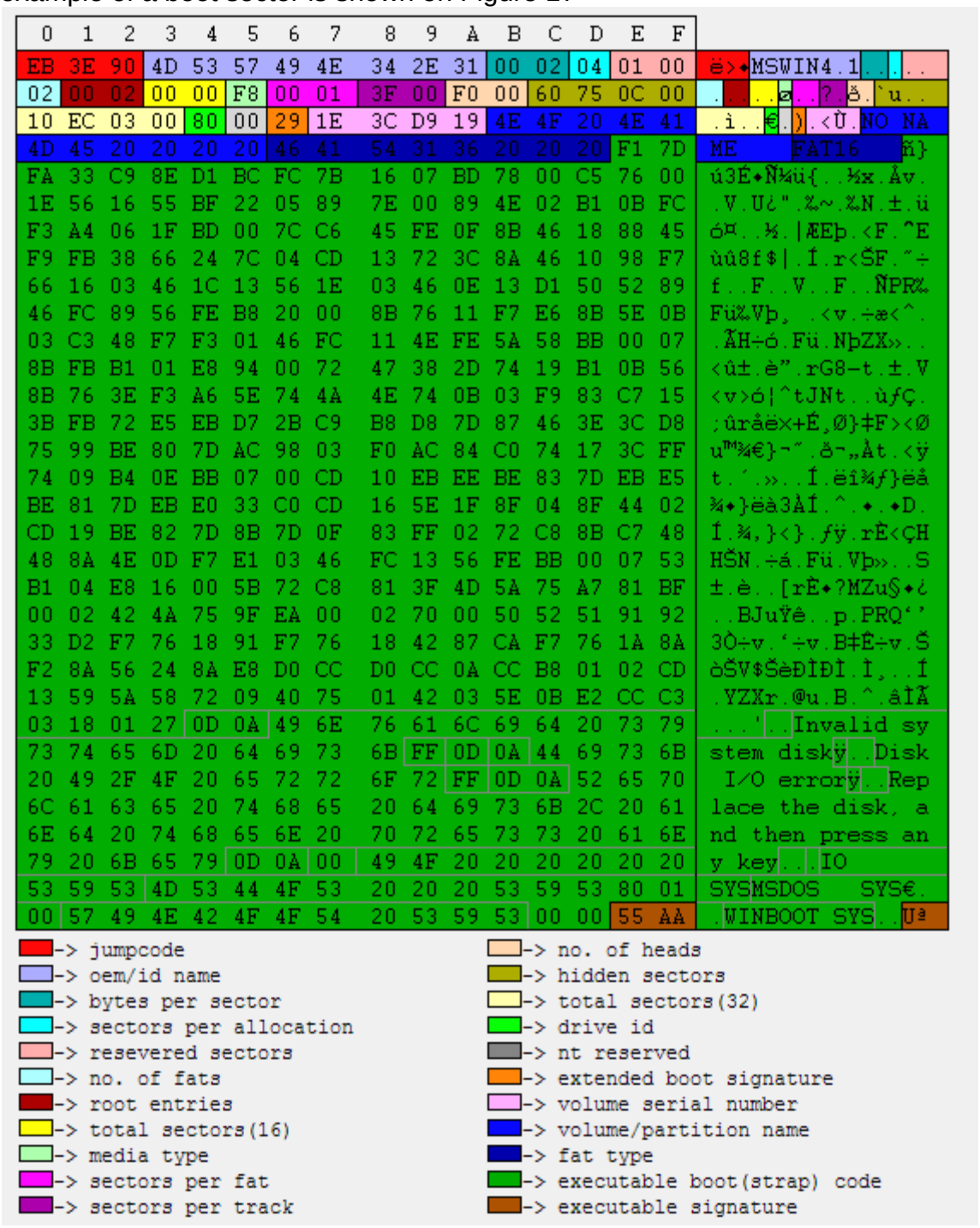
The traditional DOS file system types are FAT12 and FAT16. FAT stands for File Allocation Table: the disk is divided into *clusters*, the unit used by the file allocation, and the FAT describes which clusters are used by which files.

Let us describe the FAT file system in some detail. The FAT12/16 type is important, not only because of the traditional use, but also because it is useful for data exchange between different operating systems, and because it is the file system type used by all kinds of devices. [25]

FAT16 Layout

- The storage carrier's internal memory starts with the boot sector (at relative address 0). This is a reserved sector. Usually the boot sector is the only reserved sector.
- After the boot sector are located the FAT tables. They are following the reserved sectors; the number of reserved sectors is given in the boot sector, and can be taken from bytes number 14-15. The length of a sector is found in the boot sector, takes place in bytes 11-12). Usually there are two copies of the FAT tables which have exactly the same content.
- Then the Root Directory which is after the FATs. The number of FATs is given in the boot sector at byte position 16. Each FAT has a number of sectors which are given in the boot sector with bytes position 22-23.
- At the end is the Data Area. It is after the root directory and the distance can be found if we know the number of root directory entries. The number of the root directory entries is given in the boot sector with bytes at position 17-18, and each directory entry takes exactly 32 bytes.

An example of a boot sector is shown on Figure 17



(Figure 17) Boot sector of a FAT16 file system [26]

As can be seen from the figure each byte or set of bytes represents data about the SD card. It should be noted that the type of saving bytes is in Little Endian order. This means that the least significant byte is on the left, and the most significant byte is on the right. The byte representation is in Hexadecimal form.

Jumpcode (=3 bytes)
The offset jump to the boot(strap) executable code plus a NOP instruction

Id name (=8 bytes)
Some indication to what system formatted the partition, not checked, but set for compatibility.

Bytes per sector (=2 bytes)
Normally set to 512 bytes; from the disk: 0x00, 0x02 (flip) -> 0x02, 0x00 convert to decimal = 512 bytes. 1024, 2048 and 4096 are also valid, but are not generally used.

Sectors per cluster (=1 byte)
States the number of sectors per cluster. This will vary depending on the size of the partition.

Reserved sectors (=2 bytes)
States the number of reserved sectors. On FAT16: 0x01,0x00 (flip) -> 0x00, 0x01 convert to decimal = 1 sector. This is the boot sector.

Number of FAT tables (=1 byte)
States the number FATs used. Normally set to 2 in case of bad sectors, which could lead to data errors.

Root entries (=2 bytes)
States the maximum number of 32byte entries in the root directory. For example if on a FAT16 it is: 0x00, 0x02 (flip) -> 0x02, 0x00 convert it to decimal = 512. 512 * 32(bytes) = 16384 bytes - data is stored after this point.

Total sectors (=2 bytes)
Set if the partition is less than 33,554,432 bytes (32mb) in size - mainly for a floppy disk: 0x40,0x0B (flip) -> 0x0B,0x40 convert to decimal = 2880 sectors; FAT16 partitions may also use this entry. If the number of sectors is above 0xFF,0xFF this will be set to 0x00, 0x00

Media type (=1 byte)
States the physical media type; 0xF8 = hard drive; 0xF0 = (standard) floppy drive

Sectors per fat (=2 bytes)
Number of sectors in one fat; for a floppy: 0x09, 0x00 (flip) -> 00,09 covert to decimal = 9 * 512(bytes per sector) = 4608; there are two copies of the fat thus 4608 * 2 = 9,216 bytes - the jump to the root directory.

Sectors per track (=2 bytes)
sSates the number of sectors per track.

Number of heads (=2 bytes)
Number of disk drive heads; e.g. on a floppy disk: 0x02,0x00 flip -> 0x00, 0x02 convert to decimal = 2 heads. on a hard disk this will be much more; e.g. 0xF0, 0x00 (flip) -> 0x00, 0xF0 convert to decimal = 240 heads

Hidden sectors (=4 bytes)
This will matchup with the starting sector stated in the partitions' entry in the MBR. It states the number of hidden sectors from the beginning of the drive to the boot record of the partition. Normally set to 0x3F, 0x00, 0x00, 0x00 (flip) -> 0x00, 0x00, 0x00, 0x3F convert to decimal = 63, for a primary partition. note that sector 63 will contain the boot record, as sector numbers start at zero.

Total sectors (=4 bytes)
Number of sectors in the partition e.g. 0x10, 0xEC, 0x03, 0x00 (flip) -> 0x00, 0x03, 0xEC, 0x10 convert to decimal = 257040 sectors (normally 512 bytes each) this entry is used if total sectors is set to 0x00, 0x00

Drive id (=1 byte)
Set to 0x00 for floppy disks and 0x80 for hard disks. also refered to as the logical drive number.

NT reserved (=1 byte)
Set at format to 0x00 and not checked thereafter.

Extended boot signature (=1 byte)
Set to 29 indicating that the serial, label and type data is present.

Volume serial number (=4 bytes)
When a partition is formatted; quick or full, it will display the newly assigned serial such as: 0x15, 0xE7, 0x2A, 0x35. This is written in reverse on the disk as: 0x35, 0x2A, 0xE7, 0x15. Calculated by combining the date and time at the point of format, it is an unique identifier to keep track of drives in use. It is not possible to retrieve the date and time from the serial number.

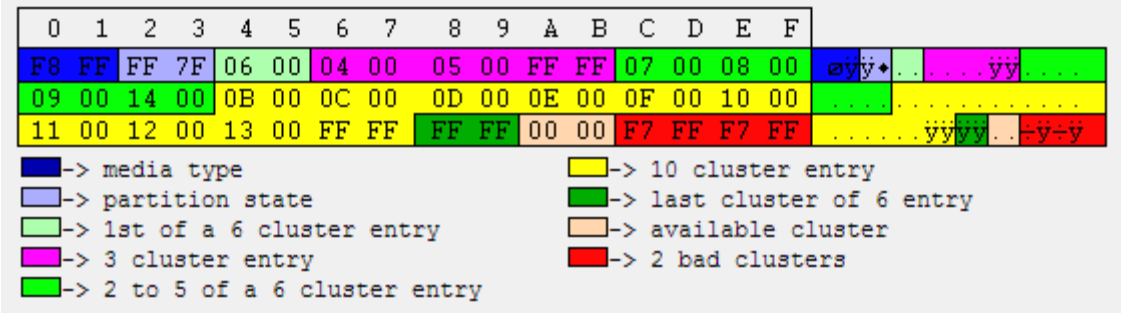
Volume/partition name (=11 bytes)
The volume label can be up to 11 characters. It also has to be referenced in the root directory as a 32 byte entry with the volume attribute to be displayed. "no name" will normally be used when the label is unused.

FAT type (=8 bytes)
Normally set to: FAT12, FAT16 and FAT32; this is not used to determine the FAT type and can be changed, though some non-MS drivers use it. It is useful as a quick reference of the FAT type; though not always accurate.

Executable boot code (=448 bytes)
Sets out the aforementioned partition information and checks for the first file to start the system; normally io.sys. If this is not present; the stated error message (or modified one) will appear.

executable signature (=2 bytes)
The MBR checks for this signature: 0x55, 0xAA - if absent: "missing operating system" error. [26]

After the boot sector and rest of reserved sectors start the FAT tables. FAT tables are sequences of 16-bit (2 byte) values that show which file clusters a file takes in the data area. An example of a FAT table is shown on Figure [18]



(Figure 18) FAT table

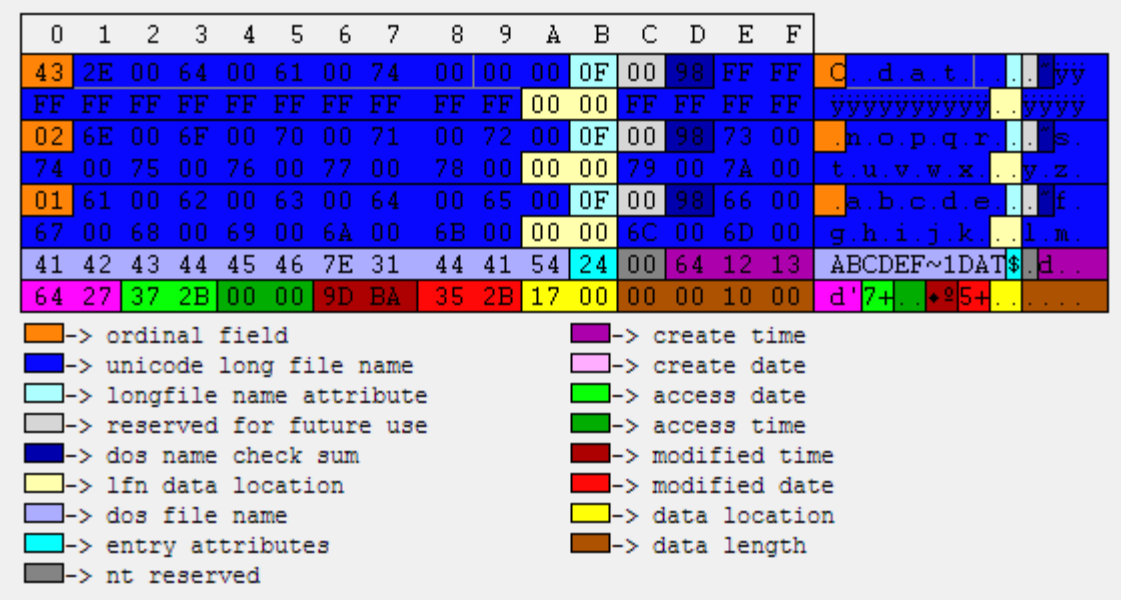
The file allocation tables (FAT) are positioned between the boot record and the root directory. There are normally two identical copies made of the FAT; FAT1 and FAT2; designed to try and prevent storage errors when disks were less reliable.

Media type (=2 bytes)
The first 2 bytes of FAT16 state the media type; 0xF8 = hard disk; floppy disks are 0xF0; this value must match up with the media type stated in the boot record.

Partition state (=2 bytes)
Set to 0xFF, 0xFF at format and at shut down; it means that the partition is "clean" not mounted or hasn't been written to. If data or an entry is renamed or saved these two bytes will change to 0xFF, 0xF7 indicating that the partition is "dirty" mounted or in use. If a system starting up finds that the partition is still mounted, it will know that the system did not shut down properly and may run scandisk. This only applies to hard disks.

FAT attributes (varies in size)
Data clusters starts at cluster 2 right after the root directory. On FAT16 one cluster is represented by 2 bytes in the fat. If the cluster is part of chain of clusters, occupying more than one cluster, they will be numbered in hex order each showing the next one. The numbers range from 0x03, 0x00 to 0x99, 0xFF, in little endian order. If the cluster is the last in the chain or if the entry only takes up one cluster it will be marked as 0xFF, 0xFF. If a file is resaved and it exceeds its allocated cluster, the file or directory list will be split onto two, and the extra clusters located elsewhere on the drive. The fat will be changed so that the last cluster in the first chain will point to the start of the next cluster chain. In the diagram above the first cluster (relative 16-bit address: 2) points to cluster 6, (though labeled as 0x07, 0x00 it is cluster 6). This is known as fragmenting, and can slow the system down a little as these pieces need to be reassembled before use. Defragmenting will move the data around so that all the chains of clusters are grouped together. If there is no data stored or data has been deleted the FAT will be marked as 0x00, 0x00. If the FAT entries do not completely fill the last sector of the fat the remaining space will be left empty and scandisk will not check it. Bad clusters are marked with 0xF7, 0xFF. data will not be written to these areas and a thorough scandisk will not recover these clusters, only report that they are there. [26]

After the FAT tables start the Root Directory Entries. Their size is constant 32 bytes and the number of entries is stated in the boot sector. An example is shown in Figure 19



(Figure 19) Root directory entries

If the entry is deleted the first byte is changed to 0xE5; and the sectors are marked as free.

DOS file name (=11 bytes)

DOS names can be >=1 and <=11 characters. Any remaining bytes will be filled with spaces.

Entry attributes (=1 byte)

Documented attribute values

None: 0x00

Archive: 0x20

Read-only: 0x01

System: 0x04

Hidden: 0x02

Directory: 0x10

Volume: 0x28 (HDD)

ntreserved (=1 byte)

Set to 00 at entry creation and never modified or checked thereafter. Reserved for windows nt. Copying the file resets the attribute, but renaming doesn't.

Create time (=3 bytes)

Minimum time: 00:00:00 = 0x00, 0x00, 0x00 if zero, no time in properties

Maximum time: 23:59:59 = 64 7D BF

If the values on the disk are: 0x64, 0x90, 0xA6 and properties: 8:52:33pm

First we must flip the bytes: 0x64, 0x90, 0xA6 -> 0xA6, 0x90, 0x64

Convert to binary: 0xA6, 0x90, 0x64 -> 10100110,10010000,01100100

Divide up the sections and convert back to decimal: (from left to right)
(5bits;hour) 10100 -> = 20hrs
(6bits;mins) 110100 -> = 52 minutes
(5bits;secs) 10000 -> 16; * 2 = 32 seconds
(8bits;mili) 01100100 -> = 100 milliseconds

Create date (=2 bytes)
Minimum date: 1/1/1980 = 0x21, 0x00
Maximum date: 2/7/2106 = 0x46, 0xFC
If the values on the disk are: 0x14, 0x2B and properties: 20th august 2001
First we must flip the bytes: 0x14, 0x2B -> 0x2B, 0x14
Convert to binary: 0x2B, 0x14 -> 00101011,00010100
Divide up the sections and convert back to decimal: (from left to right)
(7bits;yr) 0010101 -> 21; + 1980 = 2001
(4bits;mt) 1000 -> = 8 or august
(5bits;dy) 10100 -> = 20th

Access date (=2 bytes)
Minimum date: 1/1/1980 = 0x21, 0x00
Maximum date: 2/7/2106 = 0x46, 0xFC
This date is highly changeable, just right clicking on it will reset to current date, however it can be correctly queried programmatically.
This is calculated the same way as the Create date (see above)

Access time (=2 bytes)
Minimum/Maximum: 0x00, 0x00
This is calculate in the same way as creation time without the last byte of milliseconds.

Modified time (=2 bytes)
00:00:00 = 0x00, 0x00 if zero, no time in properties
23:59:58 = 0x24, 0x28
this is calculated the same way as the creation time (see above) except that it does not have the same accuracy; one less byte. Thus the modified time is to the nearest 2 seconds.

Modified date (=2 bytes)
Minimum date: 1/1/1980 = 0x21, 0x00
Maximum date: 2/7/2106 = 0x46, 0xFC
This is calculated the same way as the create date (see above)

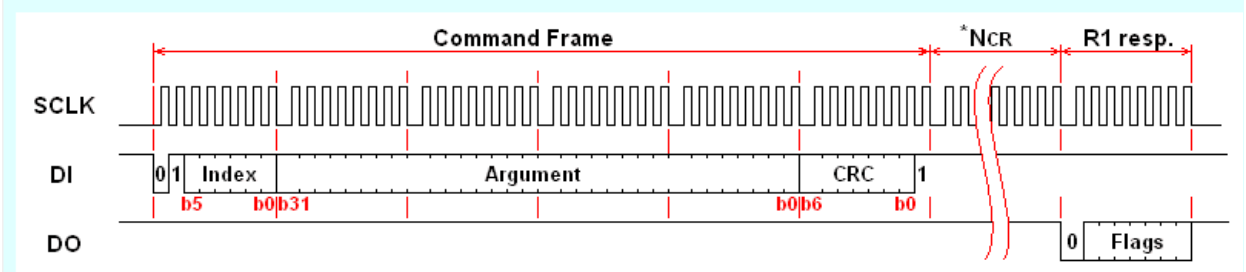
Data location (=2 bytes)
The minimum can be 0x00, 0x00 and the maximum 0xFF, 0xFF
The correct value will point the OS to the starting cluster of the data.
Example: 0x03, 0x00, flip these values: 0x00, 0x03, convert to decimal: 3. Data starts at the beginning of the third cluster.

Data length (=4 bytes)
The minimum can be 0x00 and the maximum 0xFF,0xFF,0xFF,0xFF [26]

4.2.3 SD card communication

The MMC/SDC can be attached to the most microprocessors via a generic SPI interface or GPIO ports.

In SPI mode, the data direction on the signal lines are fixed and the data is transferred in **byte oriented** serial communication. The command frame from host to card is a fixed length packet that shown below. The card is ready to receive a command frame when it drives DO high. After a command frame is sent to the card, a response to the command (R1, R2, R3 or R7) is sent back from the card. Because the data transfer is driven by serial clock generated by host controller, the host controller must continue to read data, send a 0xFF and get received byte, until a valid response is detected. The DI signal must be kept high during read transfer (send a 0xFF and get the received data). The response is sent back within command response time (NCR), 0 to 8 bytes for SDC, 1 to 8 bytes for MMC. The CS signal must be driven high to low prior to send a command frame and held it low during the transaction (command, response and data transfer if exist). The CRC feature is optional in SPI mode. CRC field in the command frame is not checked by the card. [27] Figure 20 shows the command packet.



(Figure 20) Command packet for SPI communication with SD card

Some of the commands supported by the SD cards are shown on Figure 21.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.
*1:ACMD<n> means a command sequense of CMD55-CMD<n>.					
*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]					
*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]					

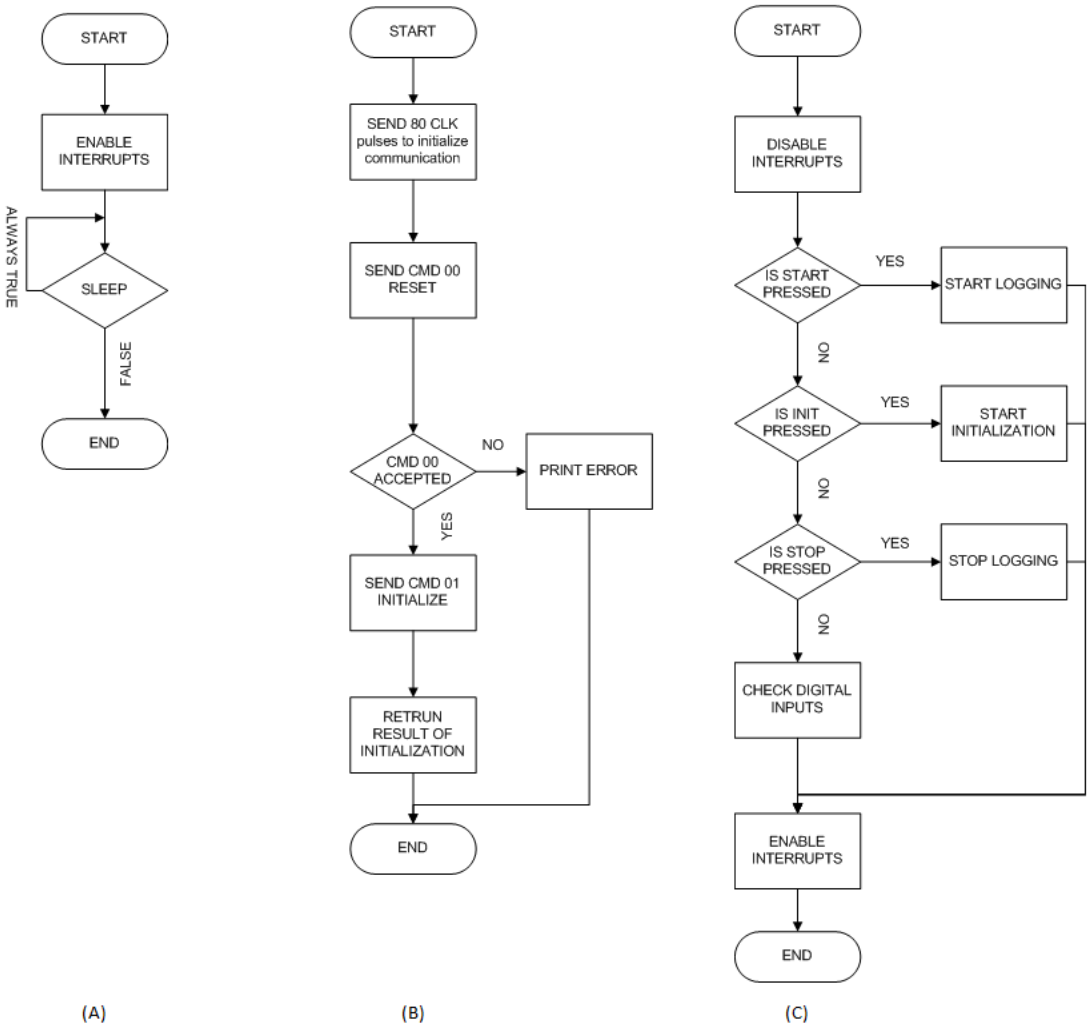
(Figure 21) Most used commands by SD cards

After supply voltage reached 2.2 volts, one millisecond at least must pass before further communications. SPI clock should be set between 100kHz and 400kHz. DI and CS must go HIGH and 74 or more clock pulses should be applied to SCLK. The card will enter its native operating mode and go ready to accept native command.

To reset the SD card a CMD0 with CS low must be sent. If the CS signal is low, the card enters SPI mode and responds R1 with In Idle State bit (0x01). Since the CMD0 must be sent as a native command, the CRC field must have a valid value. When once the card enters SPI mode, the CRC feature is disabled and the CRC is not checked by the card, so that command transmission routine can be written with the hardcoded CRC value that valid for only CMD0.

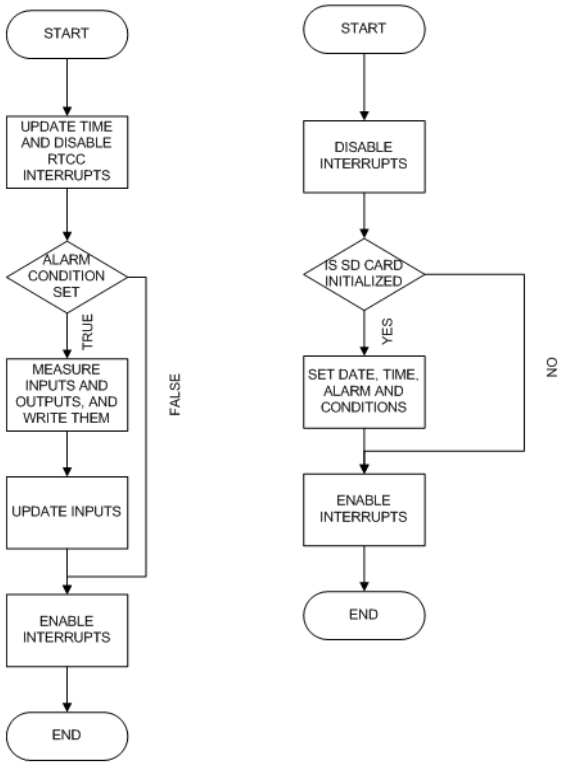
4.2.4 Software flowcharts

This part will introduce the basic software techniques used to develop the software of the microprocessor. The flowcharts are simple representation of the processed code. For full code assessment, please turn to the appendixes chapter, all of the source code and its explanation can be found there.
The main function flowchart is shown on figure 22(A).



(Figure 22)(A) – Main function, (B) – SD card initialization function, (C) Interrupt-on-change function.

The main function enables the interrupts, initializes the LCD, and puts the microprocessor in sleep mode. The SD card initialization function is shown on figure 22 (B). This function initializes the SD card by sending two commands, reset and initialize. The Interrupt-on-change function is shown on figure 22 (C). This function is active when the interrupt-on-change has been triggered. This is done by pressing a button. Figure 23 shows the RTCC interrupt routine (A). This function is triggered each second by the RTCC alarm. The initialization routine (B) happens when the INIT button is pressed. This routine prompts the user for information on the date, time, alarm conditions etc. All of the routines are explained in detail in chapter 7 Appendixes.

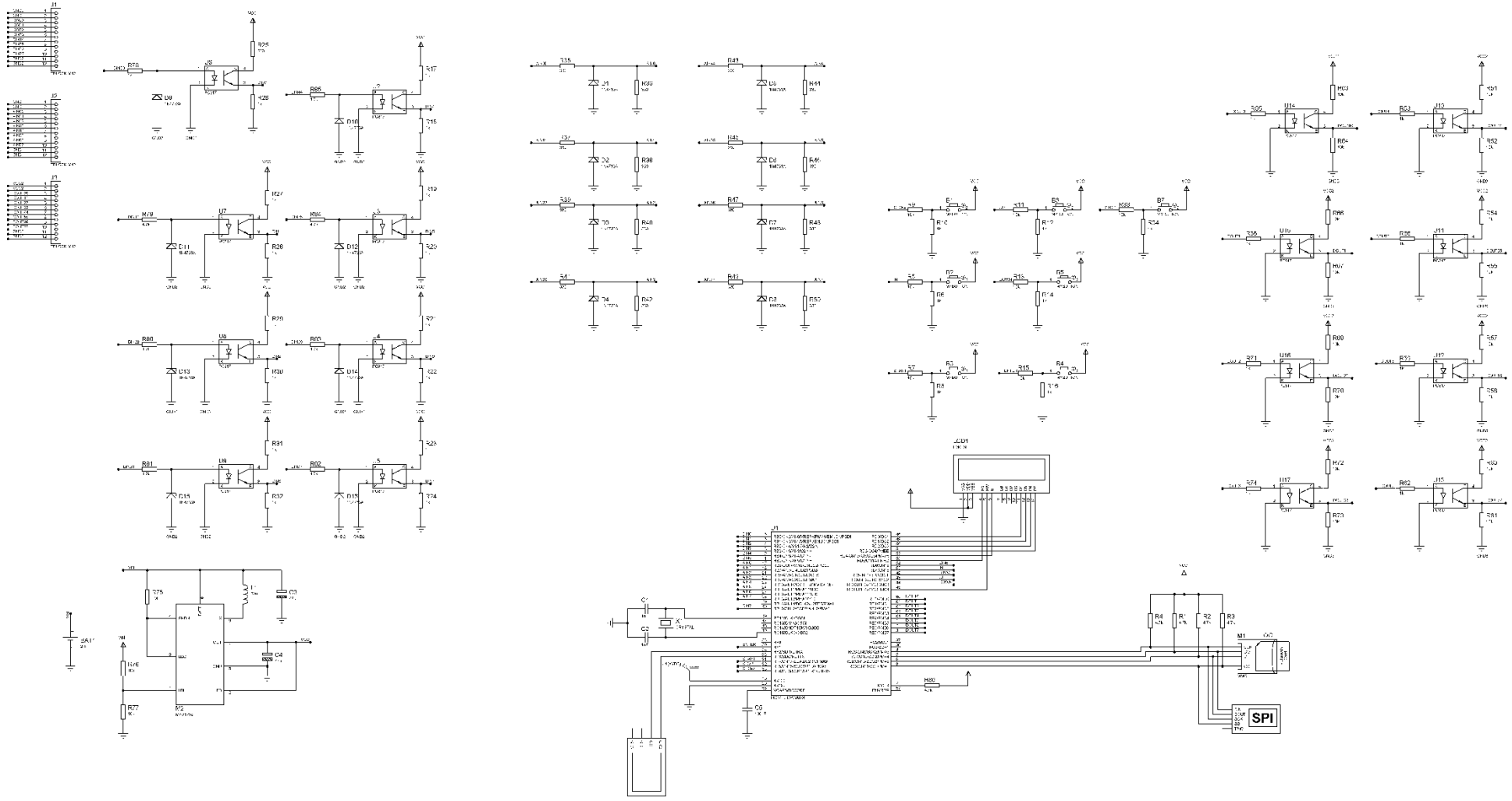


(Figure 23)(A) – RTCC interrupt routine, (B) – Initialization routine

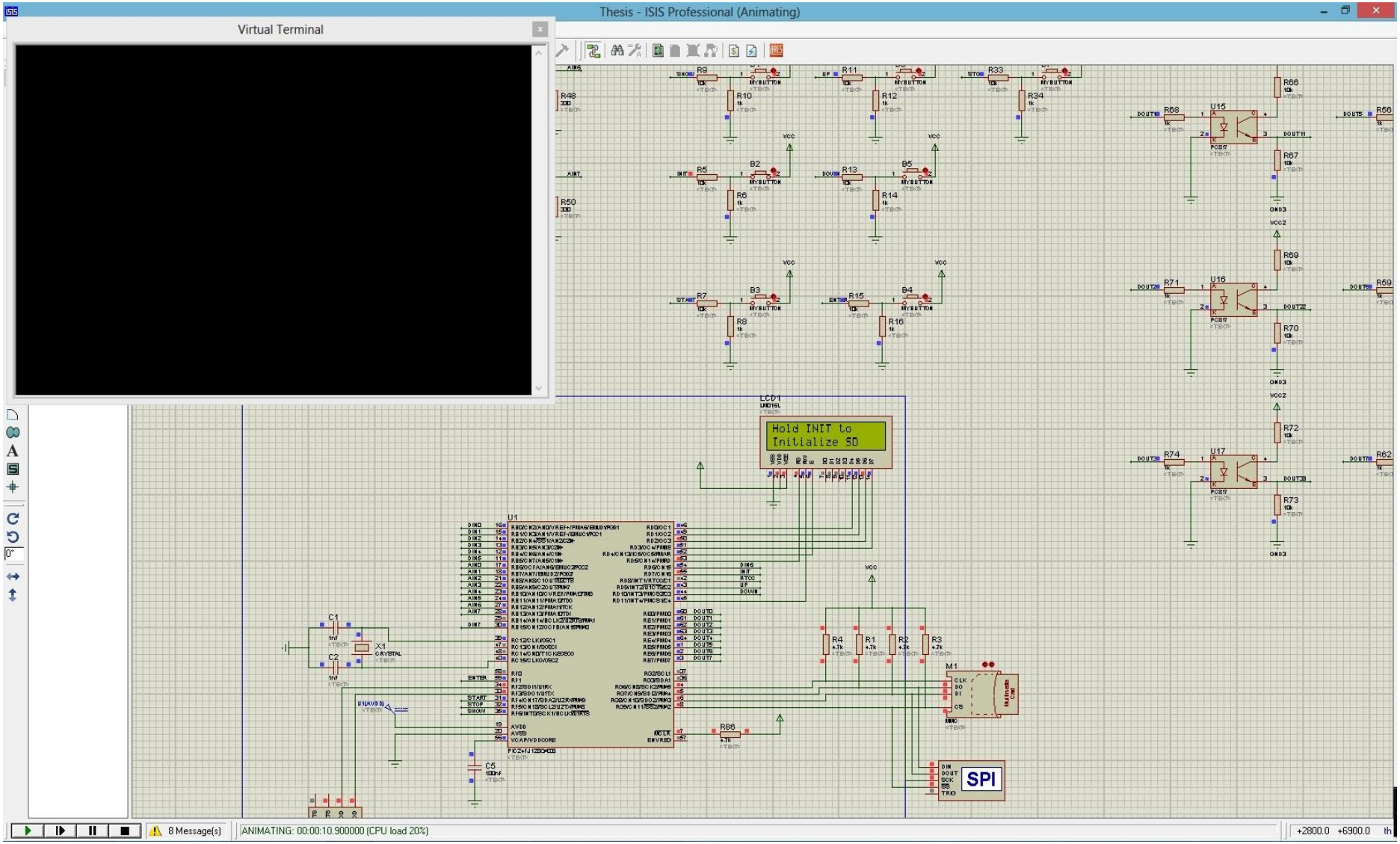
Chapter 5. Experiments and analysis

Simulations will be carried with the use of PROTEUS ISIS v.7.10. The software can simulate microprocessors, SD cards and all kinds of digital and analog signals. For the experiment the circuit shown on Figure 25 has been assembled.
For the purpose of the test an image file was created using WINIMAGE. The image file will act as a storage card.
ISIS Proteus offers RS232 monitors and SPI debuggers which can be used to follow the correct program operation.
When the program is loaded into the microprocessor and the simulation starts all of the functions and interrupts work as expected and programmed. The RS232 Feedback is enough to understand at which point is currently the program, or if the microprocessor is sleeping. The figures attached bellow demonstrate part of the process of configuring and proper working of the whole process. From initialization of the microprocessor, initialization of the SD card, interrupt handling, ADC logging, digital inputs checking, digital outputs alerting based on the conditions, reading the boot sector, modifying FAT tables, reading and creating new root directory entries, adding data, plotting and CSV creation.
Guide to properly installing and using the data logger:

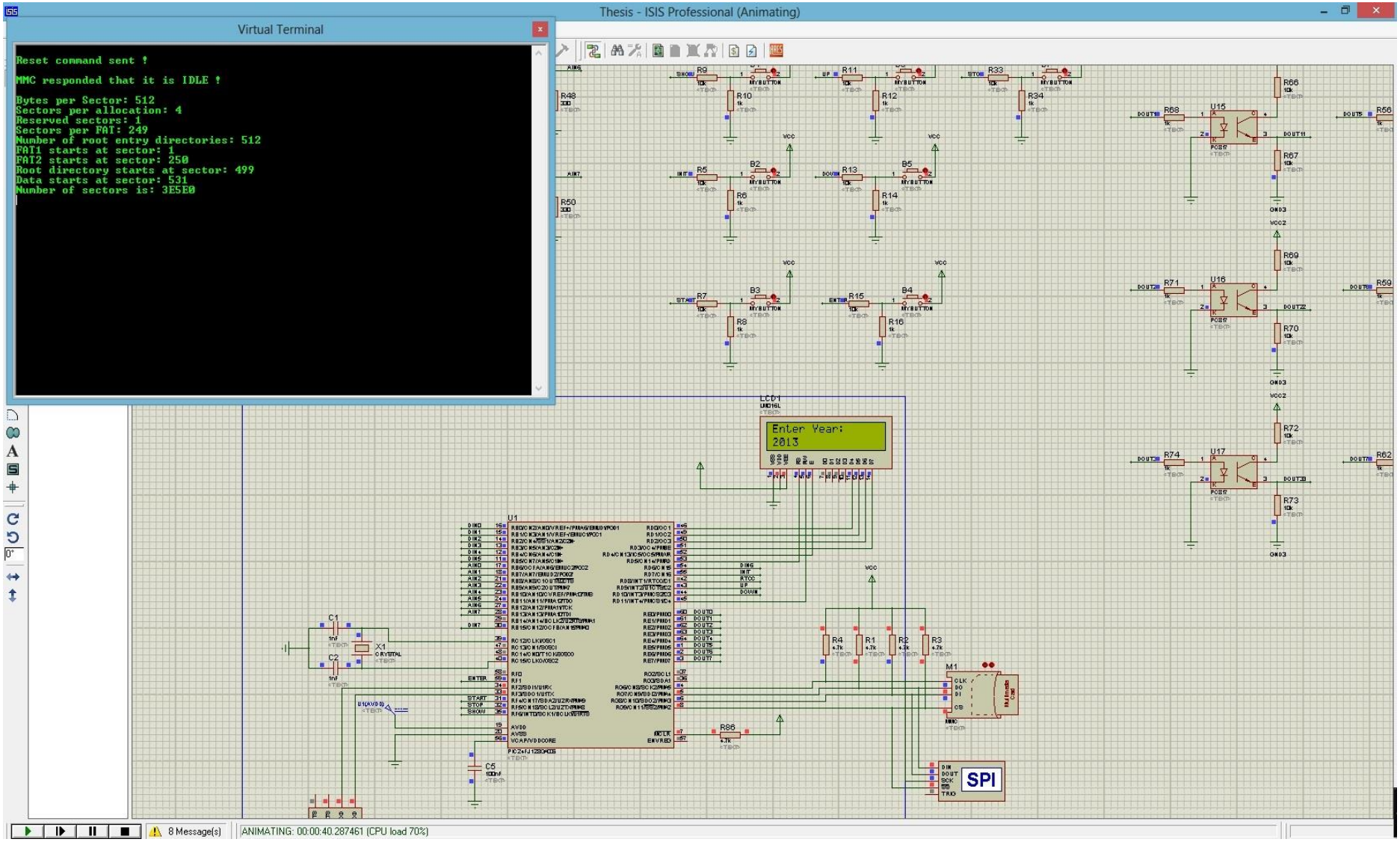
- Before placing the SD card in the data logger unit, format it as a FAT16 file system. The SD card should not exceed 4 GB of storage, preferably 2 GB.
- Do not apply power to the data logger unit during the time of the SD card connection. This can lead to damaging of certain sectors, or completely destroying it.
- Do not apply negative voltage to the analog inputs of the Logger. They are specifically designed to operate under the assignment of this Thesis.
- Do not apply 220VAC to the analog inputs or any voltage above the maximum voltage ratings discussed in this thesis. This can lead to the destruction of the microprocessor and the whole unit itself.
- Before unplugging the SD from the SD socket, be sure to hold the STOP button for at least 5 to 6 seconds.



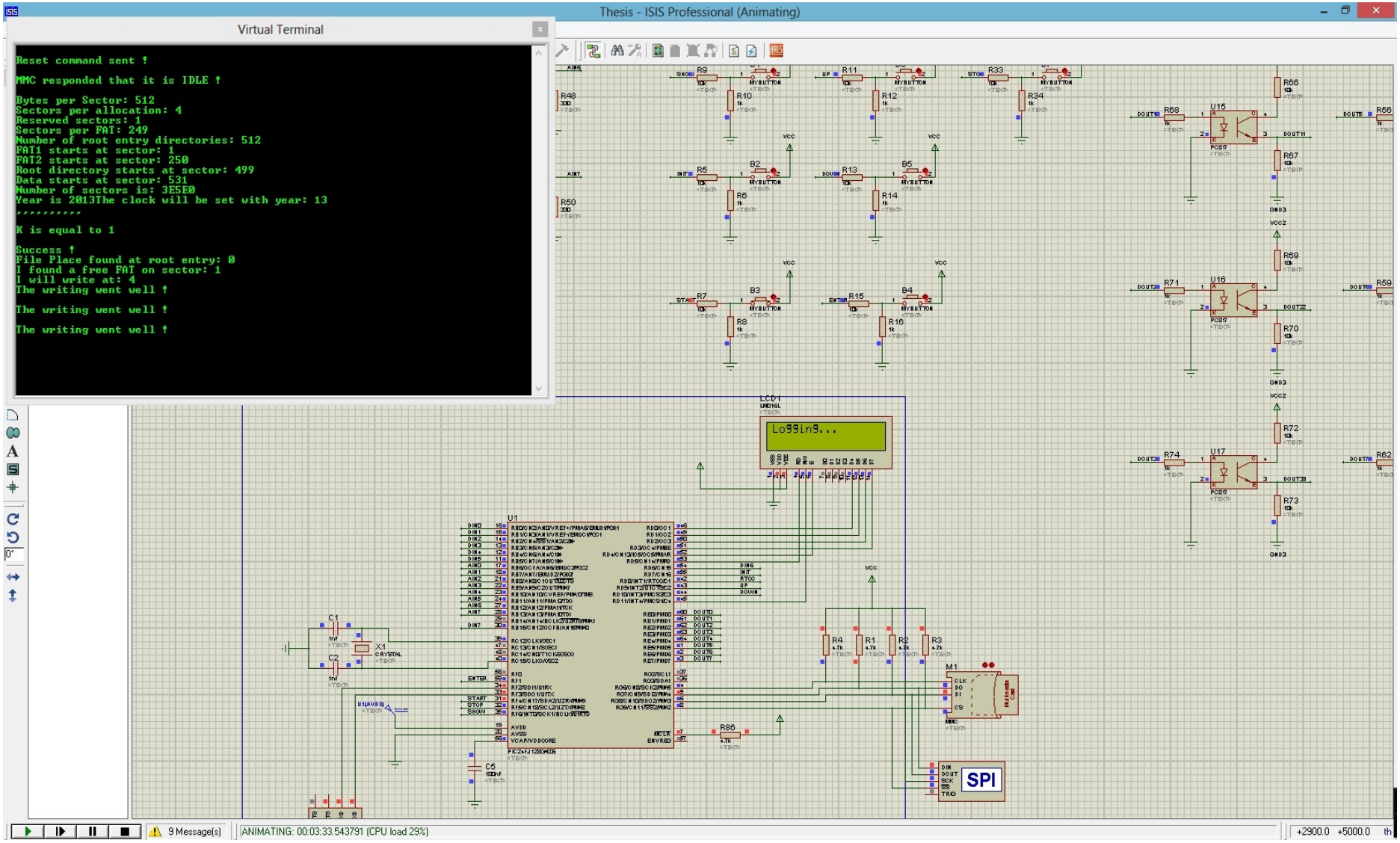
(Figure 24) Circuit of the autonomous data logging unit



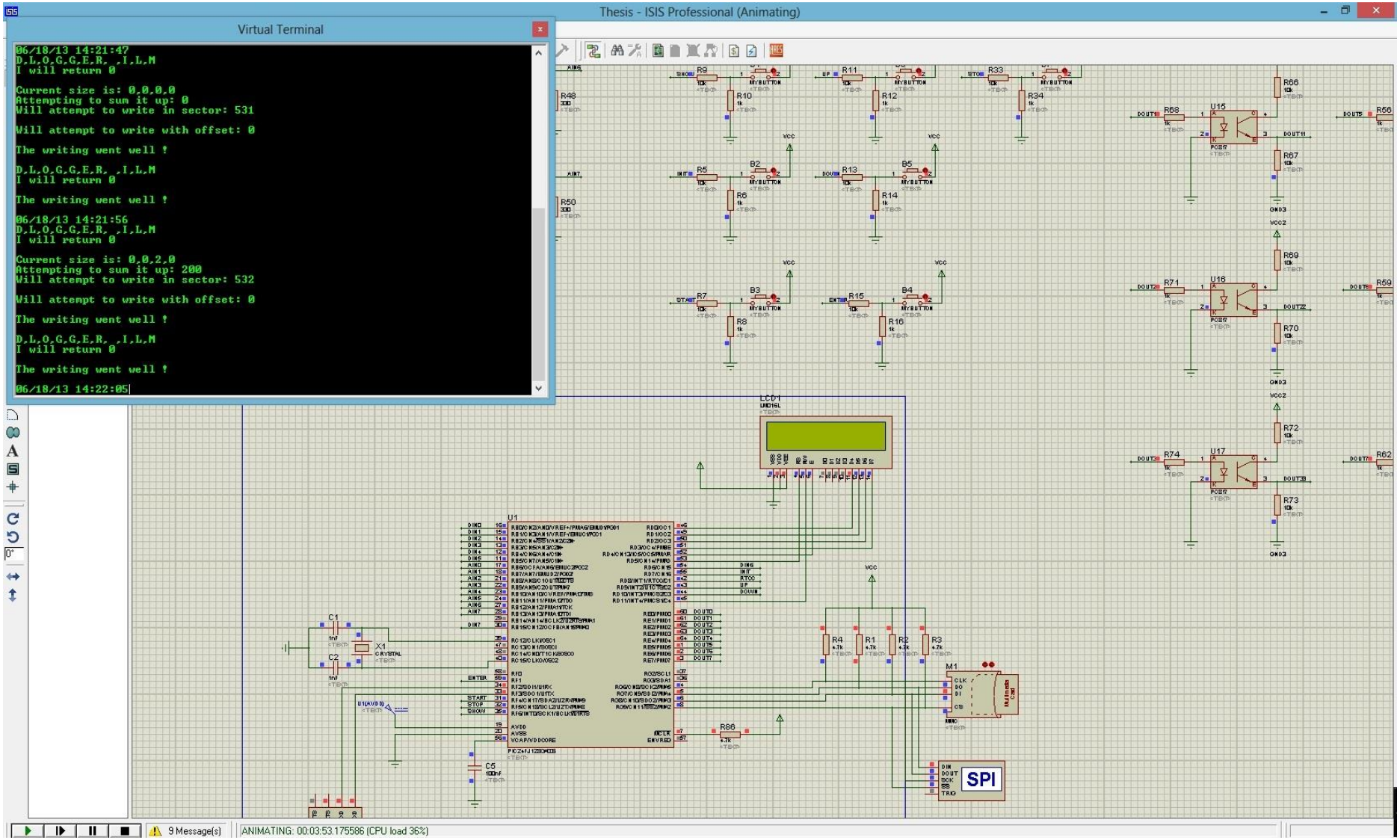
(Figure 25) SD card INIT button



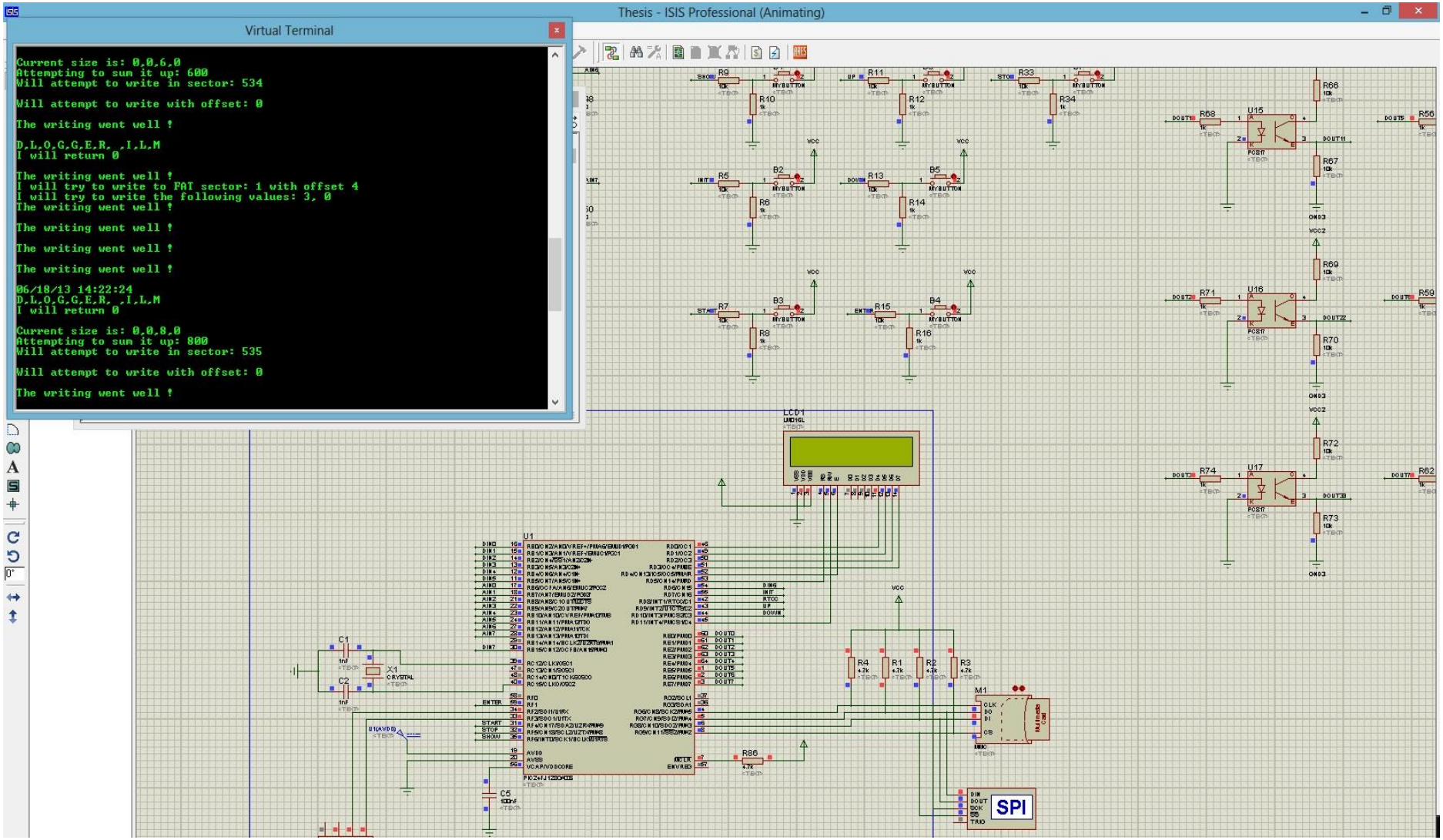
(Figure 26) SD card boot sector and customization



(Figure 27) Initialization complete, logging starting



(Figure 28) Real time clock interrupts and data logging



(Figure 29) Extending the file to other free clusters

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
000000000	EB	58	90	57	49	4E	49	4D	41	47	45	00	02	04	01	00	ëX WINIMAGE
000000016	02	00	02	00	00	F8	F9	00	20	00	40	00	20	00	00	00	øù @
000000032	E0	E5	03	00	80	00	29	50	0E	C3	5D	20	20	20	20	20	àå ▌)P Ã]
000000048	20	20	20	20	20	20	46	41	54	31	36	20	20	20	00	00	FAT16
000000064	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000080	00	00	00	00	00	00	00	00	00	00	FA	33	C0	8E	D0	BC	ú3À▌Ð¼
000000096	00	7C	B8	B0	07	8E	D8	8E	C0	B9	00	01	8B	F1	BF	00	|¸° ▌Ø▌À¹ ▌ñ¿
000000112	03	F3	A5	B8	D0	07	50	8E	D8	8E	C0	B8	80	01	50	CB	ó¥¸Ð P▌Ø▌À¸▌ PË
000000128	FB	BE	13	02	E8	3A	00	B8	01	02	B9	01	00	BA	80	00	û¾ è: ¸ ¹ º▌
000000144	33	DB	8E	C3	BB	00	7C	06	53	CD	13	72	0A	26	81	3E	3Û▌Ã» | SÍ r & >
000000160	FE	7D	55	AA	75	01	CB	BE	D0	01	E8	14	00	B4	01	CD	þ}Uªu Ë¾Ð è ´ Í
000000176	16	74	06	32	E4	CD	16	EB	F4	32	E4	CD	16	33	D2	CD	t 2äÍ ëô2äÍ 3ÒÍ
000000192	19	FC	AC	0A	C0	74	08	56	B4	0E	CD	10	5E	EB	F3	C3	ü¬ Àt V´ Í ^ëóÃ
000000208	43	61	6E	6E	6F	74	20	6C	6F	61	64	20	66	72	6F	6D	Cannot load from
000000224	20	68	61	72	64	64	69	73	6B	2E	0D	0A	49	6E	73	65	harddisk. Inse
000000240	72	74	20	53	79	73	74	65	6D	64	69	73	6B	20	61	6E	rt Systemdisk an
000000256	64	20	70	72	65	73	73	20	61	6E	79	20	6B	65	79	2E	d press any key.
000000272	0D	0A	00	44	69	73	6B	20	66	6F	72	6D	61	74	74	65	Disk formatte
000000288	64	20	77	69	74	68	20	57	69	6E	49	6D	61	67	65	20	d with WinImage
000000304	36	2E	35	30	20	28	63	29	20	31	39	39	33	2D	32	30	6.50 (c) 1993-20
000000320	30	34	20	47	69	6C	6C	65	73	20	56	6F	6C	6C	61	6E	04 Gilles Vollan
000000336	74	0D	0A	73	65	65	20	68	74	74	70	3A	2F	2F	77	77	t see http://ww
000000352	77	2E	77	69	6E	69	6D	61	67	65	2E	63	6F	6D	0D	0A	w.winimage.com
000000368	42	6F	6F	74	73	65	63	74	6F	72	20	66	72	6F	6D	20	Bootsector from
000000384	43	2E	48	2E	20	48	6F	63	68	73	74	61	74	74	65	72	C.H. Hochstatter
000000400	0D	0A	0D	0A	4E	6F	20	53	79	73	74	65	6D	64	69	73	No Systemdis
000000416	6B	2E	20	42	6F	6F	74	69	6E	67	20	66	72	6F	6D	20	k. Booting from
000000432	68	61	72	64	64	69	73	6B	2E	0D	0A	00	00	00	00	00	harddisk.
000000448	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000464	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000480	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000496	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	Uª
000000512	F8	FF	FF	FF	03	00	FF	FF	00	00	00	00	00	00	00	00	øÿÿÿ ÿÿ

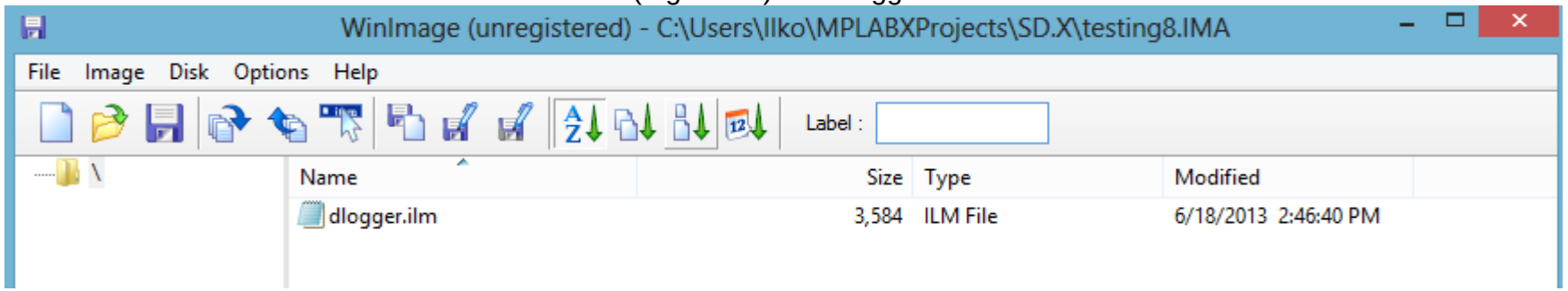
(Figure 30) Boot sector + first FAT entry

```
000255488 44 4C 4F 47 47 45 52 20  49 4C 4D 00 00 BC EE 8B DLOGGER ILM  ¼î▌
000255504 C5 42 00 00 00 00 D4 75  D2 42 02 00 00 0E 00 00 ÅB    ÔuÒB
000255520 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255536 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255552 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255568 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255584 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255600 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255616 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255632 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000255648 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

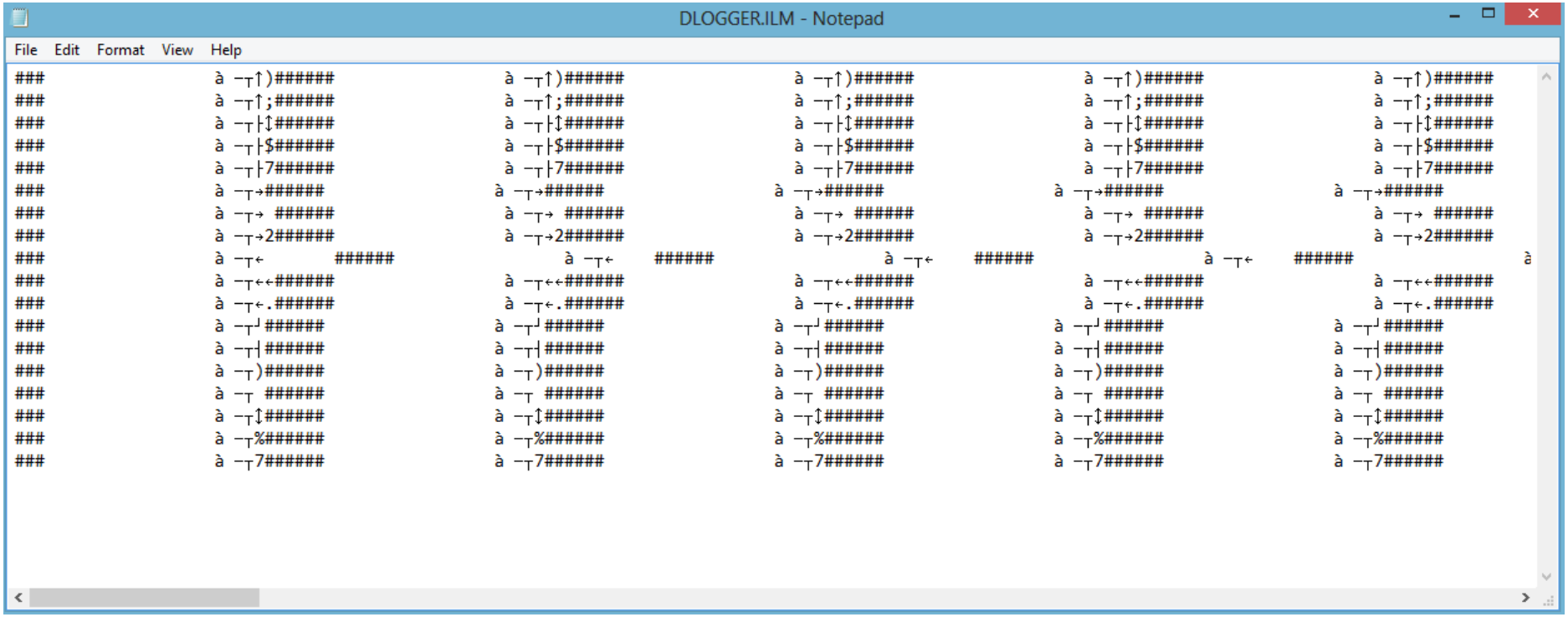
(Figure 31) Root directory entry

```
000299520 23 23 23 FF C0 00 00 00  00 00 00 00 00 00 00 00 ###ÿÀ
000299536 00 00 00 00 FF 80 00 0D  07 08 0E 14 30 23 23 23     ÿ▌       0###
000299552 23 23 23 00 00 00 00 00  00 00 00 00 00 00 00 00 ###
000299568 00 00 00 00 7F 80 00 0D  07 08 0E 14 30 23 23 23       ▌       0###
000299584 23 23 23 00 00 00 00 00  00 00 00 00 00 00 00 00 ###
000299600 00 00 00 00 7F 80 00 0D  07 08 0E 14 30 23 23 23       ▌       0###
000299616 23 23 23 FF C0 00 00 00  00 00 00 00 00 00 00 00 ###ÿÀ
000299632 00 00 00 00 FF 80 00 0D  07 08 0E 14 30 23 23 23     ÿ▌       0###
000299648 23 23 23 FF C0 00 00 00  00 00 00 00 00 00 00 00 ###ÿÀ
000299664 00 00 00 00 FF 80 00 0D  07 08 0E 14 30 23 23 23     ÿ▌       0###
000299680 23 23 23 00 00 00 00 00  00 00 00 00 00 00 00 00 ###
```

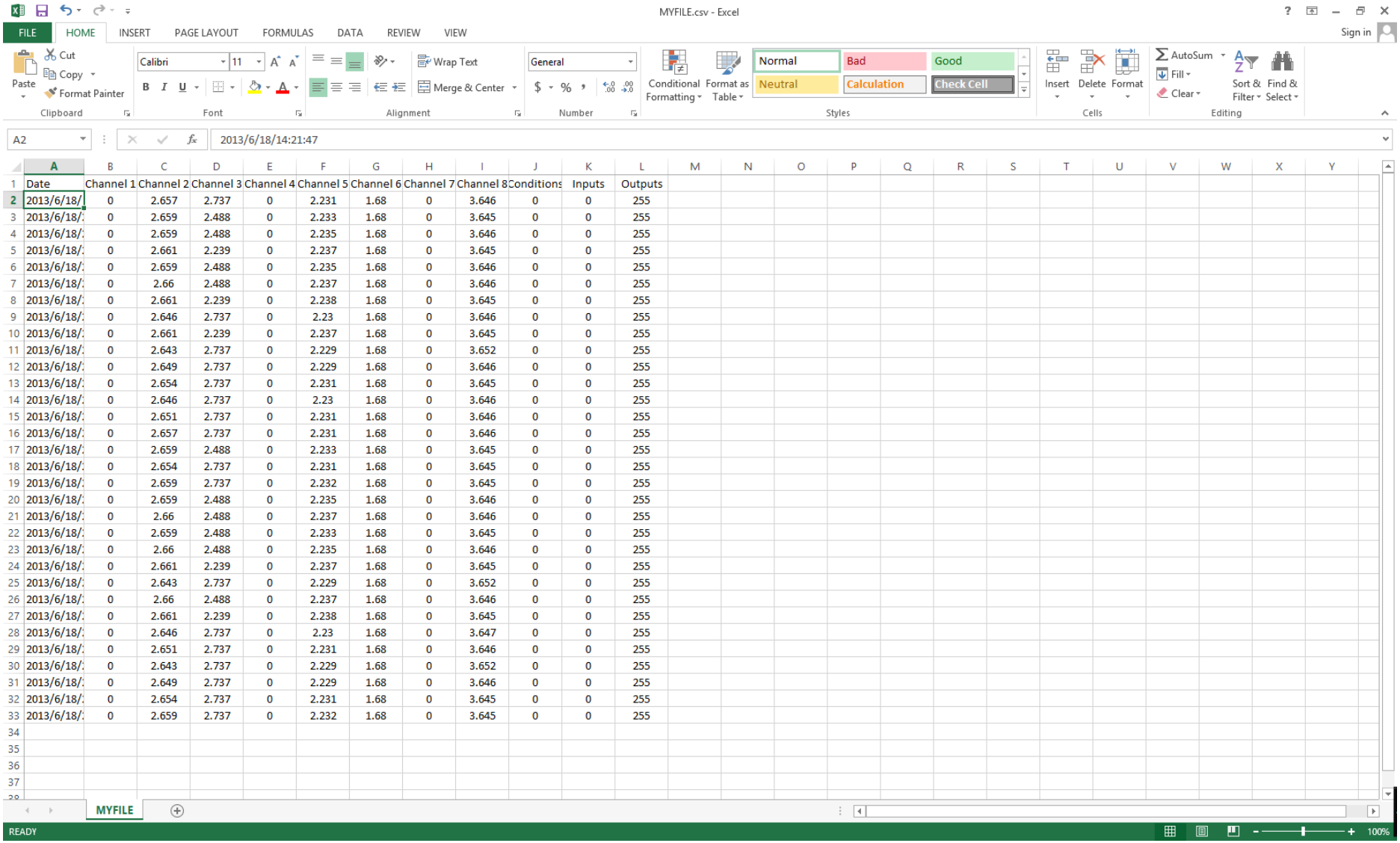
(Figure 32) Data logged



(Figure 33) Image file after logging. Date and size have been written.

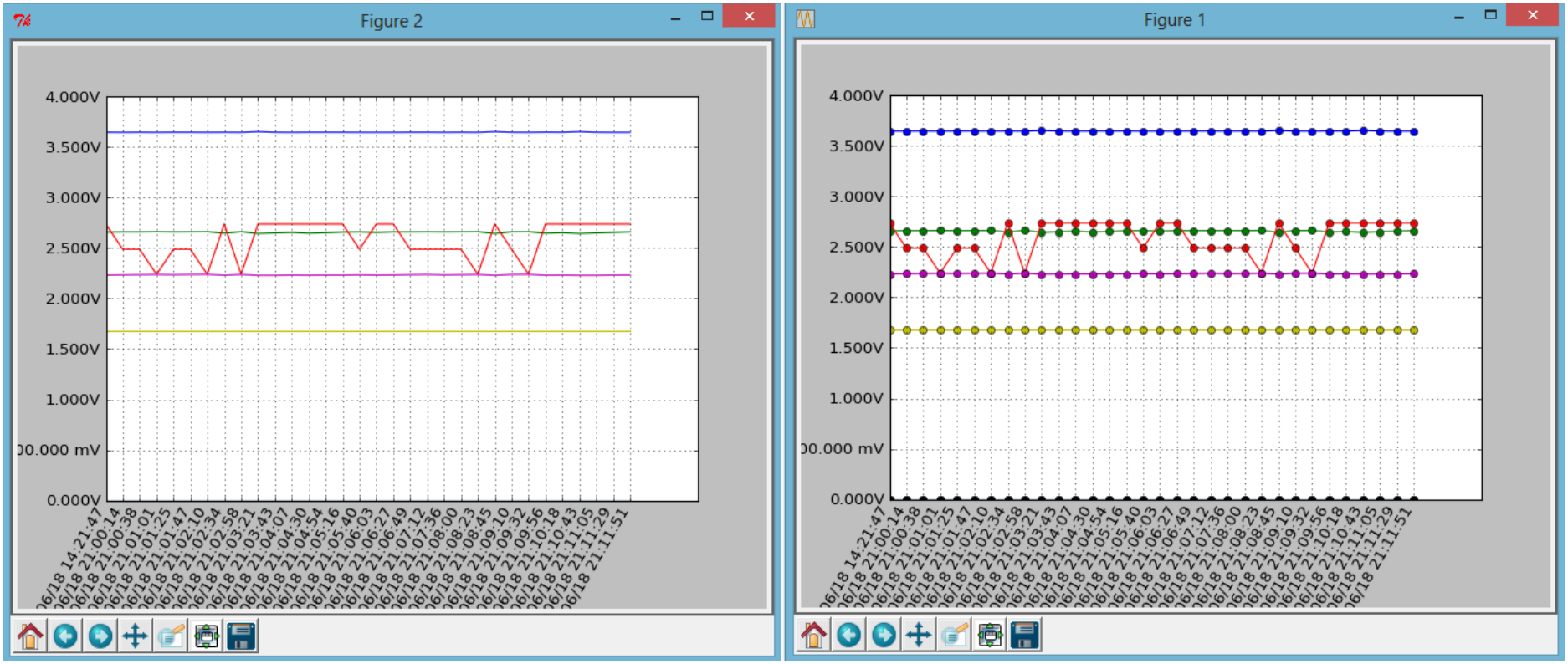


(Figure 34) Logging file opened with a text editor.



	A	B	C	D	E	F	G	H	I	J	K	L
1	Date	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7	Channel 8	Conditions	Inputs	Outputs
2	2013/6/18/	0	2.657	2.737	0	2.231	1.68	0	3.646	0	0	255
3	2013/6/18/	0	2.659	2.488	0	2.233	1.68	0	3.645	0	0	255
4	2013/6/18/	0	2.659	2.488	0	2.235	1.68	0	3.646	0	0	255
5	2013/6/18/	0	2.661	2.239	0	2.237	1.68	0	3.645	0	0	255
6	2013/6/18/	0	2.659	2.488	0	2.235	1.68	0	3.646	0	0	255
7	2013/6/18/	0	2.66	2.488	0	2.237	1.68	0	3.646	0	0	255
8	2013/6/18/	0	2.661	2.239	0	2.238	1.68	0	3.645	0	0	255
9	2013/6/18/	0	2.646	2.737	0	2.23	1.68	0	3.646	0	0	255
10	2013/6/18/	0	2.661	2.239	0	2.237	1.68	0	3.645	0	0	255
11	2013/6/18/	0	2.643	2.737	0	2.229	1.68	0	3.652	0	0	255
12	2013/6/18/	0	2.649	2.737	0	2.229	1.68	0	3.646	0	0	255
13	2013/6/18/	0	2.654	2.737	0	2.231	1.68	0	3.645	0	0	255
14	2013/6/18/	0	2.646	2.737	0	2.23	1.68	0	3.646	0	0	255
15	2013/6/18/	0	2.651	2.737	0	2.231	1.68	0	3.646	0	0	255
16	2013/6/18/	0	2.657	2.737	0	2.231	1.68	0	3.646	0	0	255
17	2013/6/18/	0	2.659	2.488	0	2.233	1.68	0	3.645	0	0	255
18	2013/6/18/	0	2.654	2.737	0	2.231	1.68	0	3.645	0	0	255
19	2013/6/18/	0	2.659	2.737	0	2.232	1.68	0	3.645	0	0	255
20	2013/6/18/	0	2.659	2.488	0	2.235	1.68	0	3.646	0	0	255
21	2013/6/18/	0	2.66	2.488	0	2.237	1.68	0	3.646	0	0	255
22	2013/6/18/	0	2.659	2.488	0	2.233	1.68	0	3.645	0	0	255
23	2013/6/18/	0	2.66	2.488	0	2.235	1.68	0	3.646	0	0	255
24	2013/6/18/	0	2.661	2.239	0	2.237	1.68	0	3.645	0	0	255
25	2013/6/18/	0	2.643	2.737	0	2.229	1.68	0	3.652	0	0	255
26	2013/6/18/	0	2.66	2.488	0	2.237	1.68	0	3.646	0	0	255
27	2013/6/18/	0	2.661	2.239	0	2.238	1.68	0	3.645	0	0	255
28	2013/6/18/	0	2.646	2.737	0	2.23	1.68	0	3.647	0	0	255
29	2013/6/18/	0	2.651	2.737	0	2.231	1.68	0	3.646	0	0	255
30	2013/6/18/	0	2.643	2.737	0	2.229	1.68	0	3.652	0	0	255
31	2013/6/18/	0	2.649	2.737	0	2.229	1.68	0	3.646	0	0	255
32	2013/6/18/	0	2.654	2.737	0	2.231	1.68	0	3.645	0	0	255
33	2013/6/18/	0	2.659	2.737	0	2.232	1.68	0	3.645	0	0	255
34												
35												
36												
37												

(Figure 35) CSV file after python post-processing



(Figure 36) Graphics after python post-processing

Chapter 6. Conclusions

Based on the finished Thesis, the student believes that the objectives have been fulfilled completely, namely:

- Understanding the functionality of autonomous data loggers, their types and applications.
- Comprehensive analysis of the different methodologies of constructing such devices.
- Implementation and construction of working model of an autonomous data logger unit with the following specifications:
    - Battery supply voltage – 2.4 V DC;
    - Controller supply voltage – 3.3V DC;
    - Number of analog input channels – 8;
    - Analog input voltage 0 … 5V
    - Number of digital inputs 8;
    - Number of digital outputs 8

Future thesis development will introduce:

- Wireless logging features, Touchscreen and a new file system standard (FAT32)

Chapter 7. Appendixes and used literature

7.1 Appendixes

7.1.1 Configuring the microprocessors configuration registers

```
#INCLUDE <24FJ128GA006.H>            //This chooses the current microprocessor and adds all of the available internal registers
#FUSES HS, NOWDT, NOPROTECT      //Sets the configuration bits to High Speed Clock, No WatchDog Timer, No Code protect
#USE DELAY(OSCILLATOR=20M,CLOCK=20M)                    //Sets the clock frequency of the microprocessor to 20 MHz
```

7.1.2 Configuring the ADCs

```
#device adc = 10                                                    //This instruction states that the used ADC is 10-bit
{
….
SETUP_ADC_PORTS(SAN6|SAN7|SAN8|SAN9|SAN10|SAN11|SAN12|SAN13,VSS_VDD); //This instruction states which pins are
                                                                                    // analog inputs, and reference power
SETUP_ADC(ADC_CLOCK_DIV_128);                    //This instruction sets the clock speed of the internal ADC oscillator
….
SET_ADC_CHANNEL(6);                    //This instruction sets the current measurement to be made on analog channel #6
DELAY_US(60);                  //60 micro seconds must pass between setting the channel, and actually measuring the value
VALUE = READ_ADC();                                                                 //Reading the value of the ADC
}
```

7.1.3 Configuring the interrupts

Before an interrupt happens it must be enabled.
For example

```
{
…
enable_interrupts(INTR_CN_PIN|PIN_B0);    //Enables interrupt-on-change at pin B0. If the value changes it will trigger an interrupt.
enable_interrupts(intr_global);                                                     //Enables interrupt flag receiving
```

```
while(1)                                                    //Puts the microprocessor in an endless loop
        sleep();                                            //Puts the microprocessor to sleep (low-power level)
}
```

After this is done the interrupt vector should receive an address pointing to a sub-routine:

```
#INT_CNI                                                    //Interrupt vector
Void cni_isr(void)                                          //Address of a sub-routine that will be executed when interrupt happens
{
…
}
```

7.1.4 Configuring the RTCC

```
{
…
setup_rtc(RTC_ENABLE|RTC_OUTPUT_ALARM,0x00);      //Enables the RTCC, at alarm interrupt the RTCC pin will go //HIGH, no
                                                  //additional calibration will be used
setup_rtc_alarm(RTC_ALARM_ENABLE,RTC_ALARM_SECOND,1);      //Enables the alarm, alarming each second, doing it 1 time
set_clock(write_clock);                                    //The structure write_clock is set with Time & Date
rtc_write(&write_clock);                                   //These values are sent to the RTCC
}
```

7.1.5 Entering sleep mode

Sleep mode is entered with the function

```
{
Sleep();
}
```

7.1.6 Configuring the UART/SPI

```
#use RS232(baud=9600, bits = 8, parity = N, xmit = PIN_F3, rcv = PIN_F2, ERRORS) //This sets the RS232 to be active with baud
                                                                                 //9600, 8-bit communication, parity is disabled,
                                                                  //transmit pin is F3, receive pin is F2, No errors should be handled
```

```
SETUP_SPI2(SPI_MASTER | SPI_XMIT_L_TO_H | SPI_SS_ENABLED | SPI_MODE_8B | SPI_CLK_DIV_128);
//This instruction sets the SPI2 to be master, low-to-high edge clock, control over chip select, 8-bit byte, and the clock value of SPI
```

7.1.7 C libraries include

```
#include <flex_lcd.c>          //
#include <stdlib.h>            //These libraries are needed by certain mathematical and input/output functions
#include <stdio.h>             //
#include <math.h>              //
```

7.1.8 Global variables

```
#define CS              PIN_G9          //Chip select is on pin G9
#define INIT            PIN_D7          //INIT button is on pin D7
#define DOWN            PIN_D10         //DOWN button is on pin D10
#define UP              PIN_D9          //UP button is on pin D9
#define START           PIN_F4          //START button is on pin F4
#define STOP            PIN_F5          //STOP button is on pin F5
#define SHOW            PIN_F6          //SHOW button is on pin F6
#define ENTER           PIN_F1          //ENTER button is on pin F1
static int Position[16];                //16 bytes intArray containing conditions and their percentage values
char Buffer[512];                       //512 bytes char Array which will contain SD card reading results
char sBuffer[512];                      //512 bytes char Array which will contain SD card writing values
char BootSector[512];                   //512 bytes char Array containing BootSector values
static int sData[512];                  //512 bytes int Array containing the bytes that will be sent to data sectors
rtc_time_t write_clock, read_clock;     //write_clock, read_clock are instances of the structure rtc_time_t
char Logger[32] = {0x44,0x4C,0x4F,0x47,0x47,0x45,0x52,0x20,0x49,0x4C,0x4D,0x00,  // This is the root entry that will be
        0x00,0xBC,0xEE,0x8B,0xC5,0x42,0x00,0x00,0x00,0x00,0xF8,0x8B,             // created in the SD card root directory
        0xC5,0x42,0x02,0x00,0x000,0x00,0x00,0x00};                               //
int32 ResSec,FilePA,SecPF,NumRE,RDStart,FATStart,DataStart,NumSec;              //BootSector variables for easier readings
int Month=6,Day=18,Hours=14,Minutes=20,Seconds = 0,Weekday=3;                    //RTCC variables
int16 BytesPS,interrupts=0,alarm=1,Year=2013&0xFFFF,LogDate,LogTime;             // Bytes per sector and RTCC variables
unsigned int8 Status=0;                                                         //Digital inputs will be put here
```

7.1.9 Main function

```
void main(void)
{
   lcd_init();                    // LCD library function required before sending commands to the LCD
   delay_ms(1);                   // Before talking to the MMC/SD 1 mS must pass
   SETUP_ADC_PORTS(sAN6|sAN7|sAN8|sAN9|sAN10|sAN11|sAN12|sAN13,VSS_VDD);     // ADC pins setup
   SETUP_ADC(ADC_CLOCK_DIV_128);              //ADC clock setup by dividing oscillator (20 MHz) to 128
   enable_interrupts(INTR_CN_PIN|START);      //START button triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|INIT);       //INIT button triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|STOP);       //STOP button triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B0);     //Digital input 0 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B1);     //Digital input 1 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B2);     //Digital input 2 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B3);     //Digital input 3 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B4);     //Digital input 4 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B5);     //Digital input 5 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_D6);     //Digital input 6 triggers interrupt-on-change
   enable_interrupts(INTR_CN_PIN|PIN_B15);    //Digital input 7 triggers interrupt-on-change
   enable_interrupts(INT_EXT0);               //SHOW button pressed triggers external interrupt
   enable_interrupts(intr_global);            //Un-masking and enabling all of the interrupts
   while(1){
      sleep();                                //Puts the microprocessor to sleep, after interrupt again will be set to sleep
   }
}
```

7.1.10 SD card initialization function

```
int mmc_init()                          //This function initializes the SD card
{
   int i;
   SETUP_SPI2(SPI_MASTER | SPI_XMIT_L_TO_H | SPI_SS_ENABLED | SPI_MODE_8B | SPI_CLK_DIV_128); //SPI setup
   OUTPUT_HIGH(CS);                                  // Sets chip select to Logic 1 (off)
   for(i=0;i<10;i++)                                // Initializes the SD card into SPI mode by sending more than 74 CLK signals
   {
         SPI_WRITE2(0xFF);                           //One byte is 8 bits, each bit sent requires a clock signal thus 80 CLK are sent
   }
   printf("\r\nReset command sent !\r\n");          //Sends feedback by the RS232
   OUTPUT_LOW(CS);                                   // set chip select to Logic 0 (on)

   SPI_WRITE2(0x40);                                 // Send reset command
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x95);
   SPI_READ2(0xFF);
   if(SPI_READ2(0xFF)==1)                            // SD card must respond with 1 if IDLE
      printf("\r\nMMC responded that it is IDLE !\r\n");  // RS232 feedback if all is okay
   else {
      printf("\r\nSPI_READ2 returned %d\r\n",i);     // RS232 feedback if not initialized SD card
      return;
   }
   SPI_WRITE2(0x41);                                 // Send init command
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
   SPI_WRITE2(0x00);
```

```
    SPI_WRITE2(0x00);
    SPI_READ2(0xFF);

    return SPI_READ2(0xFF);                    // Should return 0 if init command was received and went okay
}
```

7.1.11 SD_Read function

```
void SD_read(unsigned long sector, unsigned int offset, unsigned int len) { // Function takes as arguments which sector to read, at
    int i;                                                                  // what offset to read it, and what length to read
    OUTPUT_LOW(CS);                                                         // Enables communication with SD card
    SPI_WRITE2(0x00);
    SPI_WRITE2(0x51);                                                       //Send READ command (CMD17|0x40)
    SPI_WRITE2(sector>>15); // sector*512 >> 24
    SPI_WRITE2(sector>>7);  // sector*512 >> 16
    SPI_WRITE2(sector<<1);  // sector*512 >> 8
    SPI_WRITE2(0x00);         // sector*512
    SPI_WRITE2(0x00);

    for(i=0; i<1000 && SPI_READ2(0xFF) != 0x00; i++) {}                     // wait for 0 – this means command received okay

    for(i=0; i<1000 && SPI_READ2(0xFF) != 0xFE; i++) {}                     // wait for data start, data starts after 0xFE received

    for(i=0; i<offset; i++)
       SPI_WRITE2(0xFF);                                                    // will skip bytes and start after the offset value

    for(i=0; i<len; i++)                                                    // read len bytes and put them in the reading Buffer
       Buffer[i] = SPI_READ2(0xFF);

    for(i+=offset; i<512; i++)                                              // "skip" again
       SPI_WRITE2(0xFF);
```

```
    SPI_WRITE2(0x00);                                   //
    SPI_WRITE2(0x00);                                   // skip checksum
    SPI_WRITE2(0x00);                                   //
    SPI_WRITE2(0x00);                                   //

    OUTPUT_HIGH(CS);                                    // Deselect SD card, end of communication
}
```

7.1.12 SD_Write function

```
void SD_write(unsigned long sector, unsigned int offset,unsigned int len) { //Function takes as arguments which sector to write,
    int i,j;                                            //at what offset to write, what length to write
    if (offset > 512 || len > 512){                     //If the offset or length is bigger than 512 an error occurs
        printf("\r\n\n\nImpossible to write !\n\n\n");   //because SD card space is in 512 byte sectors, if we try
        break;                                          //to write/offset 513 bytes the SD card will signal for error
    }
    SD_Read(sector,0,512);                              //Calls SD_read which reads the sector we want to write
    OUTPUT_LOW(CS);                                     //enables communication with the SD card
    SPI_WRITE2(0x00);
    SPI_WRITE2(0x58);                                   //Sends write command
    SPI_WRITE2(sector>>15);                             // sector*512 >> 24
    SPI_WRITE2(sector>>7);                              // sector*512 >> 16
    SPI_WRITE2(sector<<1);                              // sector*512 >> 8
    SPI_WRITE2(0x00);                                   // sector*512, these actions transform sector to address
    SPI_WRITE2(0x00);                                   //
    SPI_WRITE2(0x00);                                   //Skipping CRC check
```

```
   if (SPI_READ2(0xFF)==0){                                  //If the command was accepted
      SPI_WRITE2(0xFE);                                      //Send 0xFE which means start of writing
      for (i=0;i<offset;i++)
         SPI_WRITE2(Buffer[i]);                              //Until the offset is reached write old data
      for (i=0;i<len;i++)
         SPI_WRITE2(sBuffer[i]);                             //Write new data for the length of data given
      for (i+=offset;i<512;i++)
         SPI_WRITE2(Buffer[i]);                              //Again write old data if there is length left
      SPI_WRITE2(0xFF);                                      // CRC low byte skip
      SPI_WRITE2(0xFF);                                      // CRC high byte skip
      SPI_READ2(0xFF);                                       // NRC
      SPI_READ2(0xFF);                                       // skip byte
      for(i=0;i<2048;i++){
         j = SPI_READ2(0xFF);
         if(j==0){                                           //waiting for 0x00 meaning all went well
            delay_us(100);
            printf("\r\nThe writing went well !\r\n");
            return;
         }
      }
      printf("THE WRITING DIDNT GO WELL !!!\r\n");           //If 0x00 was not received indicate error at RS232
   }
   OUTPUT_HIGH(CS);                                          //End communication
}
```

7.1.13 getBoot function

```
void getBoot(void){                                  //The function reads the Boot Sector and loads the global variables
   int32 temp;
   int i;
   SD_read(0,0,512);                                 //The sector starts at address 0,for 512 bytes. This instructions reads it.
   for (i=0;i<512;i++)                               // Read receive buffer and transfer to boot array
      BootSector[i]=Buffer[i];    //
   (char)temp = BootSector[12];                      //
   temp<<=8;                                         //Take bytes 12 and 11, transform them to BIG endian and put in the
   (char)temp += BootSector[11];                     //variable BytesPS. This is the value of bytes per sector. Usually is 512
   BytesPS = temp;                                   //
   FilePA = BootSector[13];                          //Take byte 13 and put it in FilePA. Value is sectors per allocation
   temp = 0;
   (char)temp = BootSector[15];                      //
   temp<<=8;                                         //Take bytes 15 and 14, transform them to BIG endian and put in the
   (char)temp += BootSector[14];                     //variable ResSec. This is the value of reserved sectors. Usually is 1
   ResSec = temp;                                    //
   temp = 0;
   (char)temp = BootSector[18];                      //
   temp <<=8;                                        //Take bytes 18 and 17, transform them to BIG endian and put in the
   (char)temp += BootSector[17];                     //variable NumRe. This is the value of root entries. Usually is 512
   NumRE = temp;                                     //
   temp = 0;
   (char)temp = BootSector[23];                      //
   temp<<=8;                                         //Take bytes 23 and 22, transfom them to BIG endian and put in the
   (char)temp+=BootSector[22];                       //variable SecPF. This is the value of sectors per FAT.
   SecPF = temp;                                     //
   FATStart = ResSec;                                // FAT starts after the reserved sectors.
   RDStart = FATStart + SecPF*2;                     //Root Directory starts after the two FAT tables
   DataStart = NumRE*32;                             //
   DataStart >>=9;                                   //
```

```
    DataStart += RDStart;                                   // Data Start is after the root directory entries
    (char)temp = BootSector[20];                            //
    temp <<= 8;                                             //Take bytes 23 and 22, transfom them to BIG endian and put in the
    (char)temp += BootSector[19];                           //variable NumSec. This is the value of number of sectors.
    NumSec = temp;                                          //
    temp = 0;                                               //
    if (NumSec == 0){                                       //If the file is above 32 megabytes, the address of the number of sectors
       for (i=0;i<4;i++){                                   //is at bytes 32,33,34,35.
          temp <<= 8;                                       //
          (char)temp += BootSector[35-i];                   //
       }                                                    //
       NumSec = temp;                                       //
    }
    printf("\r\nBytes per Sector: %d",BytesPS);                         //RS232 feedback
    printf("\r\nSectors per allocation: %d"FilePA);                     //RS232 feedback
    printf("\r\nReserved sectors: %d", ResSec);                         //RS232 feedback
    printf("\r\nSectors per FAT: %d",(int16)SecPF);                     //RS232 feedback
    printf("\r\nNumber of root entry directories: %d", (int16)NumRE);   //RS232 feedback
    printf("\r\nFAT1 starts at sector: %d", FATStart);                  //RS232 feedback
    printf("\r\nFAT2 starts at sector: %d", (int16)FATStart+SecPF);     //RS232 feedback
    printf("\r\nRoot directory starts at sector: %d",(int16)RDStart);   //RS232 feedback
    printf("\r\nData starts at sector: %d",(int16)DataStart);           //RS232 feedback
    printf("\r\nNumber of sectors is: %X\r\n",NumSec);                  //RS232 feedback
}
```

7.1.14 find_free_fat function

```
unsigned long find_free_fat(void){              //This function returns the offset of a free FAT entry in the FAT table
   int32 I;
   int j;
   unsigned long r=0;
   for (i=0;i<SecPF;i++){                       //Will look until the end of the FAT table
      SD_read(FATStart+i*FATStart,0,512);
      for (j=0;j<512;j+=2){                     //Will check 16-bit values
         if(Buffer[j] == 0 && Buffer[j+1] == 0){ //If they are cleared will return the address
            r = i*512 + j;
            return r;
         }
      }
   }
   return 0;                                    //If no free fat found will return 0, which is an error. FAT 0 can never be free.
}
```

7.1.15 is_file_there function

```
int is_file_there(void)
{
    int i,j,k;
    for (i = 0; i < (NumRE*32)>>8; i++){
        SD_read(RDStart+i,0,512);
        for(j = 0; j < 16; j++){
            printf("\r\n%C,%C,%C,%C,%C,%C,%C,%C,%C,%C,%C\r\n",Buffer[j*32],Buffer[j*32+1],          //RS232 feedback that prints
                    Buffer[j*32+2],Buffer[j*32+3], Buffer[j*32+4], Buffer[j*32+5], Buffer[j*32+6],   //the name of the current root
                    Buffer[j*32+7], Buffer[j*32+8],Buffer[j*32+9], Buffer[j*32+10]);                 //entry
            if(Buffer[j*32] == 0x44 && Buffer[j*32+1] == 0x4C && Buffer[j*32+2] == 0x4F && Buffer[j*32+3] == 0x47              //Checks
                    && Buffer[j*32+4] == 0x47 && Buffer[j*32+5] == 0x45 && Buffer[j*32+6] == 0x52 && Buffer[j*32+7] == 0x20   //if file is
                    && Buffer[j*32+8] == 0x49 && Buffer[j*32+9] == 0x4C && Buffer[j*32+10] == 0x4D){                          //there
                for(k=0;k<32;k++)
                    Logger[k]=Buffer[j*32+k];                                                 //If the file is there will return
                printf ("I will return %u\r\n",i*16+j);                                       //the position in the root entry
                return i*16+j;                                                                //directory
            }
            else if(Buffer[j*32] == 0x00 || Buffer[j*32+1] == 0x00 || Buffer[j*32+2] == 0x00 || Buffer[j*32+3] == 0x00        // Check if last
                    || Buffer[j*32+4] == 0x00 || Buffer[j*32+5] == 0x00 || Buffer[j*32+6] == 0x00 || Buffer[j*32+7] == 0x00    // empty or
                    || Buffer[j*32+8] == 0x00 || Buffer[j*32+9] == 0x00 || Buffer[j*32+10] == 0x00)                           // last entry
                return 7;                                                                     //If the last entry is reached will return number 7
            }                                                                                 //The user must format the SD card before using it
        }                                                                                     //Which will always lead to the return of 0 in the previous
    return 7;                                                                                 //condition
}
```

7.1.16 Find_end_cluster function

```
int find_end_cluster(void){                    //This function returns the end cluster of a file in the FAT table
   int16 temp = 0;
   temp = (Logger[26] + (Logger[27]<<8))*2;    //Reads the Logger root entry and locates the starting address
   SD_read(FATStart + temp/512,temp,2);        //Reads the FAT table at the given address and reads only 2 bytes
   while (Buffer[0] != 0xFF && Buffer[1] != 0xFF){   //Looks if this is the end of the cluster which is marked with 0xFF,0xFF
      temp = (Buffer[0] + (Buffer[1]<<8))*2;   //If not will go to the next FAT entry
      SD_read(FATStart + temp/512,temp,2);
   }
   return temp;                                //returns cluster entry
}
```

4.2.18 write_to_log function

```
void write_to_log(int i){                      // i is the sector in the data entry at which should be written
   unsigned int32 size=0,temp,sizing1=0,sizing2=0;;
   int j,k;
   for (j=4;j>0;j--){                          //
       size <<= 8;                             //Reads the Logger register and sums up the size
      (char)size += Logger[27+j];              //
   }                                           //
   sizing1 = (size+512)/(float)BytesPS;        //Calculates if the new size will be
   sizing2 = NumSec - DataStart;               //bigger than the total size of the SD card
   if (sizing1 > sizing2){                     //
      printf ("\r\nSIZE PROBLEM !\r\n");       //RS232 feedback that the SD card is full
      return;
   }
   temp = size;                                //
   if (temp >= 512){                           //
```

```
    temp /= 512;                                    //Calculating of the offset that the next data should be written
    temp = size-temp*512;                           //
}
printf("\r\nCurrent size is: %X,%X,%X,%X\n",Logger[31],Logger[30],Logger[29],Logger[28]); //RS232 feedback on the current size
printf("\r\nAttempting to sum it up: %X\n",size);                                        //RS232 feedback on new size
for (j=0;j<512;j++)
    sBuffer[j]=sData[j];                                                                //Taking the Data to the write buffer
printf("\r\n\nWill attempt to write in sector: %d\r\n",DataStart+size/512);             //RS232 feedback on writing sector
printf("\r\n\nWill attempt to write with offset: %d\r\n",temp);                         //RS232 feedback on writing offset
SD_Write(DataStart + size/512,0,512);                                                   //Writing the data
for (j=0;j<512;j++)
    sBuffer[j]=0;                                                                       //Clearing the write buffer
size += 512;                                                                            //Increasing the current size
Logger[23] = (LogTime & 0xFF00)>>8;                                                     //Sets the access time
Logger[22] = (LogTime & 0xFF);                                                          //
Logger[25] = (LogDate & 0xFF00)>>8;                                                     //Sets the access date
Logger[24] = (LogDate & 0xFF);                                                          //
Logger[28]=size & 0xFF;                                                                 //
Logger[29]=(size & 0xFF00)>>8;                                                          //Sets the new size
Logger[30]=(size & 0xFF0000)>>16;                                                       //
Logger[31]=(size & 0xFF000000)>>24;                                                     //
for (j=0;j<32;j++)                                                                      //Writes the new Logger values to the
    sBuffer[j] = Logger[j];                                                             //write buffer
j = is_file_there();
if (j != 7){                                                                            //If the file is there
    SD_Write(RDStart + j/512,j*16,32);                                                  //Rewrite the logger values
    for (j=0;j<512;j++)
        sBuffer[j]=0;                                                                   //Clear the write buffer
    (int32)size /= (float)BytesPS;
    if (size % FilePA == 0 && size != 0){                                               //Check if a new cluster is required for
        k = find_free_fat();                                                            //the next data writing
```

```
        j = k/512;                                                          //Checks if the new FAT is at distance 512
        sBuffer[0]=(k & 0xFF)/2;                                            //Loads the buffer with the free FAT address
        sBuffer[1]=((k & 0xFF00)/2)>>8;                                     //
        printf("I will try to write to FAT sector: %u with offset %u\r\n",FATStart+j,i);   //RS232 feedback
        printf("I will try to write the following values: %u, %u",sBuffer[0],sBuffer[1]);  //RS232 feedback
        SD_write(FATStart + j,i,2);                                         //Writes the new pointer FAT to the first FAT
        sBuffer[0]=(k & 0xFF)/2;
        sBuffer[1]=((k & 0xFF00)/2)>>8;
        SD_write((FATStart+j)+SecPF,i,2);                                   //Writes the new pointer FAT to the 2nd FAT
        sBuffer[0]=0xFF;
        sBuffer[1]=0xFF;
        SD_write(FATStart + j,(k & 0xFF),2);                                //Writes end of file at the free entry in FAT1
        sBuffer[0]=0xFF;
        sBuffer[1]=0xFF;
        SD_write((FATStart + j)+SecPF,(k & 0xFF),2);                        //Writes end of file at the free entry in FAT2
      }
    }
}
```

7.1.17 create_log_file function

```
void create_log_file(void){                          //This function creates the Log File DLOGGER.ILM
   int i,j,free,k=0;
   Logger[28]=0x00;
   Logger[29]=0x00;
   Logger[30]=0x00;
   Logger[31]=0x00;
   for (i=0;i<((NumRE*32)>>8) && k==0;i++){
      SD_read(RDStart+i,0,512);
      for(j=0;j<16;j++){
         if(Buffer[j*32]==0x00){
            k=1;
            printf("\rK is equal to 1\r\n");
            Logger[26] = (find_free_fat()/2) & 0x00FF;      //Looks for empty root entry
            Logger[27] = ((find_free_fat()/2) & 0xFF00)>>8;
            break;
         }
         else if(Buffer[j*32]==0xE5){                   //Or for a deleted one
            k=2;
            printf("\rK is equal to 2\r\n");
            Logger[26]= (find_free_fat()/2) & 0x00FF;
            Logger[27]= ((find_free_fat()/2) & 0xFF00)>>8;
            break;
         }
      }
   }
   free = j + 16*(i-1);
   if (k == 1 || k == 2){
      printf("\r\nSuccess !");
      printf("\r\nFile Place found at root entry: %d\r\n",free);
      sBuffer[0]=0xFF;
```

```
        sBuffer[1]=0xFF;
        i = find_free_fat() / 512;
        printf("I found a free FAT on sector: %u\r\n\n",i+FATStart);
        printf("I will write at: %u",Logger[26]*2);
        SD_write(FATStart + i,Logger[26]*2,2);                    //Creating Log file and writing FAT table
        SD_write((FATStart + i)+SecPF,Logger[26]*2,2);
        for (i=0;i<32;i++)
            sBuffer[i]=Logger[i];
        SD_write(RDStart+(free/16),free*32,32);
    }
    else
        printf("\r\nFailure !");
}
```

7.1.18 RTCC interrupt function

```
#INT_RTC
void RTC_isr(void){                                   //This function will be performed on each RTCC interrupt
   int16 value[8],curYear,curValue;
   int i,j,k,l,curMonth,curDay,curHours,curMinutes,curSeconds;
   int32 temp;
   read_clk();
   curValue = Year+20;
   curValue <<= 4;                                    //Taking current values of Date and Time
   curValue += Month;
   curValue <<= 5;
   curValue += Day;
   logDate = curValue;
   curValue = Hours;
   curValue <<= 5;
   curValue += Minutes;
   curValue <<= 6;
   curValue += Seconds*2;
   LogTime = curValue;
   curYear = Year;
   curMonth = Month;
   curDay = Day;
   curHours = Hours;
   curMinutes = Minutes;
   curSeconds = Seconds;
   disable_interrupts(int_rtc);                       //Disabling interrupts
   disable_interrupts(intr_global);
   interrupts++;
   if(interrupts == alarm){
      for (l = 0; l < 16; l++){
         for (j=0;j<3;j++)                            // Start bytes
```

```
    sData[j+l*32]="#";
for (i=0;i<8;i++){                                  // Analog inputs
    set_adc_channel(6+i);
    delay_ms(60);
    value[i] = read_adc();
    sData[j+l*32] = (value[i] & 0xFF00)>>8;
    sData[j+1+l*32]=(value[i] &0xFF);
    j+=2;
}
sData[j+l*32]=Status;                               // Digital inputs
j++;
k=0;
for(i=0;i<16;i+=2){
    k<<=1;
    temp = value[i/2];
    temp &= 0xFFFF;
    temp *= 100;
    temp /= 65472;
    if (Position[i]==0){
        if (temp < Position[i+1])
            k+=1;
    }
    if (Position[i]==1){
        if (temp > Position[i+1])
            k+=1;
    }
}
output_e(k);
sData[j+l*32] = k;                                  //Digital outputs
j++;
k = 0;
```

```
    for(i=0;i<16;i+=2){
        k<<=1;
        if (Position[i]==0){
            k+=0;
        }
        if (Position[i]==1){
            k+=1;
        }
    }
    sData[j+l*32]=k;                                    //Conditions
    j++;
    sData[j+l*32]=(curYear & 0xFF00)>>8;
    j++;
    sData[j+l*32]=curYear&0xFF;
    j++;
    sData[j+l*32]=curMonth;                             //Date and time
    j++;
    sData[j+l*32]=curDay;
    j++;
    sData[j+l*32]=curHours;
    j++;
    sData[j+l*32]=curMinutes;
    j++;
    sData[j+l*32]=curSeconds;
    j++;
    for(i=0;i<3;i++,j++)                                // End bytes
        sData[j+l*32]="#";
}
i = is_file_there();
if (i!=7)
    write_to_log(find_end_cluster());
```

```
      interrupts = 0;
   }
   //printf(lcd_putc,"\f");
   status = 0;                                    //Updating status register(Inputs)
   if (input(PIN_B0)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B1)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B2)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B3)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B4)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B5)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_D6)!=0)
      Status|=1;
   status*=2;
   if (input(PIN_B15)!=0)
   Status|=1;
   setup_rtc(RTC_ENABLE|RTC_OUTPUT_ALARM,0x00);
   setup_rtc_alarm(RTC_ALARM_ENABLE,RTC_ALARM_SECOND,1);    //Enabling interrupts
   enable_interrupts(int_rtc);
}
```

7.1.19 INIT function

```
void SD_init(void){                          //Initialization process of the microprocessor and RTCC
   disable_interrupts(intr_global);
   int flag = 0,i;
   while (flag == 0){
      lcd_putc("\fPlease wait\n");
      lcd_putc("...");
      if(mmc_init() == 0){
         lcd_putc("\fSuccess !\n");          //LCD feedback that all went okay
         lcd_putc("SD initialized");
         getBoot();
         flag = 1;
      }
      else{
         lcd_putc("\fError ! SD card\n");     //LCD feedback with a problem
         lcd_putc("not found !");
         delay_ms(3000);
         return;
      }
   }
   flag = 0;
   while (flag == 0){
      lcd_putc("\fEnter Year:\n");
      while (1){
         printf(lcd_putc,"%u\r\n",Year);
         if (input(ENTER)!=0){
            delay_ms(350);
            if(input(ENTER)==1){
               printf("Year is %u",Year);          //Setting Year
               break;
            }
```

```
    }
    if (input(UP)!=0){
        delay_ms(200);
        if(input(UP)==1)
            Year++;
    }
    if (input(DOWN)!=0){
        delay_ms(200);
        if(input(DOWN)==1)
            Year--;
    }
}
lcd_putc("\fEnter Month:\n");
while (1){
    printf(lcd_putc,"%u\r\n",Month);                    //Setting Month
    if (input(ENTER)!=0){
        delay_ms(350);
        if(input(ENTER)==1)
            break;
    }
    if (input(UP)!=0){
        delay_ms(200);
        if(input(UP)==1){
            Month++;
            if(Month>12)
                Month = 1;
        }
    }
    if (input(DOWN)!=0){
        delay_ms(200);
        if(input(DOWN)==1){
```

```
            Month--;
            if(Month<1)
               Month = 12;
         }
      }
   }
   lcd_putc("\fEnter Day:\n");
   while (1){
      printf(lcd_putc,"%u\r\n",Day);
      if (input(ENTER)!=0){                                  //Setting day
         delay_ms(350);
         if(input(ENTER)==1)
            break;
      }
      if (input(UP)!=0){
         delay_ms(200);
         if(input(UP)==1){
            Day++;
            if(Day>28){
               if(Month == 2){
                  if((Year-2012)%4 == 0){
                     if(Day>29)
                        Day = 1;
                  }
                  else
                     Day = 1;
               }
               if(Month == 4 || Month == 6 || Month == 9 || Month == 11){          //checking days of month
                  if(Day>30)
                     Day = 1;
               }
```

```
            if(Month == 1 || Month == 3 || Month == 5 || Month == 7 || Month == 8 || Month == 10 || Month == 12){
               if(Day>31)
                  Day=1;
               }
            }
         }
      }
      if (input(DOWN)!=0){
         delay_ms(200);
         if(input(DOWN)==1){
            Day--;
            if(Day<1){
               if(Month == 2){
                  if((Year-2012)%4 == 0)
                     Day = 29;                                          //checking if leap year
                  else
                     Day = 28;
               }
               if(Month == 4 || Month == 6 || Month == 9 || Month == 11)
                  Day = 30;
               if(Month == 1 || Month == 3 || Month == 5 || Month == 7 || Month == 8 || Month == 10 || Month == 12)
                  Day=31;
            }
         }
      }
   }
   lcd_putc("\fEnter Weekday:\n");
   while (1){
      printf(lcd_putc,"%u\r\n",Weekday);                              //Setting weekday
      if (input(ENTER)!=0){
         delay_ms(350);
```

```
        if(input(ENTER)==1)
            break;
    }
    if (input(UP)!=0){
        delay_ms(200);
        if(input(UP)==1){
            Weekday++;
            if(Weekday>7)
                Weekday = 1;
        }
    }
    if (input(DOWN)!=0){
        delay_ms(200);
        if(input(DOWN)==1){
            Weekday--;
            if(Weekday<1)
                Weekday = 7;
        }
    }
}
lcd_putc("\fEnter Hours:\n");
while (1){
    printf(lcd_putc,"%u\r\n",Hours);                    //Setting hours
    if (input(ENTER)!=0){
        delay_ms(350);
        if(input(ENTER)==1)
            break;
    }
    if (input(UP)!=0){
        delay_ms(200);
        if(input(UP)==1){
```

```
            Hours++;
            if(Hours>23)
               Hours = 0;
         }
      }
      if (input(DOWN)!=0){
         delay_ms(200);
         if(input(DOWN)==1){
            Hours--;
            if(Hours<0)
               Hours = 23;
         }
      }
   }
   lcd_putc("\fEnter Minutes:\n");
   while (1){
      printf(lcd_putc,"%u\r\n",Minutes);                    //Setting minutes
      if (input(ENTER)!=0){
         delay_ms(350);
         if(input(ENTER)==1)
            break;
      }
      if (input(UP)!=0){
         delay_ms(200);
         if(input(UP)==1){
            Minutes++;
            if(Minutes>59)
               Minutes = 0;
         }
      }
      if (input(DOWN)!=0){
```

```
            delay_ms(200);
            if(input(DOWN)==1){
                Minutes--;
                if(Minutes<0)
                    Minutes = 59;
            }
        }
    }
    for(i=0;i<16;i+=2){
        Position[i+1]=1;
        printf(lcd_putc,"\fAlert Ch%d if:\n",i/2);
        while (1){
            if (Position[i]==0)
                printf(lcd_putc,"Below\r\n");                   //Setting alert outputs
            if  (Position[i]==1)
                printf(lcd_putc,"Above\r\n");
            if (input(ENTER)!=0){
                delay_ms(350);
                if(input(ENTER)==1)
                    break;
            }
            if (input(UP)!=0){
                delay_ms(200);
                if(input(UP)==1){
                    Position[i]=1;
                }
            }
            if (input(DOWN)!=0){
                delay_ms(200);
                if(input(DOWN)==1){
                    Position[i]=0;
```

```
            }
        }
    }
    printf(lcd_putc,"\fAlert Ch%d if:\n",i/2);
    while (1){
        printf(lcd_putc,"%u\r\n",Position[i+1]);
        if (input(ENTER)!=0){
            delay_ms(350);
            if(input(ENTER)==1)
                break;
        }
        if (input(UP)!=0){
            delay_ms(200);
            if(input(UP)==1){
                Position[i+1]++;
                if (Position[i+1]>99)
                    Position[i+1]=1;
            }
        }
        if (input(DOWN)!=0){
            delay_ms(200);
            if(input(DOWN)==1){
                Position[i+1]--;
                if (Position[i+1]<1)
                    Position[i+1]=99;
            }
        }
    }
}
printf(lcd_putc,"\fMeasure every\n");
while (1){
```

```
    printf(lcd_putc,"%3u minutes\r\n",alarm);                    //Setting measurement period
    if (input(ENTER)!=0){
       delay_ms(500);
       if(input(ENTER)==1)
          break;
    }
    if (input(UP)!=0){
       delay_ms(200);
       if(input(UP)==1){
          alarm++;
          if(alarm>999)
             alarm = 1;
       }
    }
    if (input(DOWN)!=0){
       delay_ms(200);
       if(input(DOWN)==1){
          alarm--;
          if(alarm<1)
             alarm = 999;
       }
    }
 }
 setup_rtc(RTC_ENABLE|RTC_OUTPUT_ALARM,0x00);                    //Setup of RTCC and enabling of interrupts
 setup_rtc_alarm(RTC_ALARM_ENABLE,RTC_ALARM_SECOND,1);
 set_clock(write_clock);
 rtc_write(&write_clock);        //writes new clock setting to RTCC
 printf(lcd_putc,"\fInitialization\n");
 printf(lcd_putc,"Complete\r");
 delay_ms(2000);
 flag = 1;
```

```
    }
}
```

7.1.20 Interrupt-on-change function

```
#INT_CNI
void cni_isr(void){                          //This function handles interrupt-on-change
   disable_interrupts(intr_global);
   int i;
   if (input(START)!=0){
      delay_ms(3000);
      if(input(START)==1){
         lcd_putc("\fLogging...\n");         //If start is pressed for 3 seconds logging will start
         delay_ms(1000);
         lcd_putc("\f");
         enable_interrupts(INT_RTC);
         enable_interrupts(intr_global);
         interrupts = 0;
         return;
      }
   }

   else if (input(INIT)!=0){                 //If init is pressed the logger will be initialized
      lcd_putc("\fHold INIT to\n");          //Button must be pressed for 2 seconds in order to start initialization
      lcd_putc("Initialize SD");
      if (input(INIT)==1){
         delay_ms(2000);
         if (input(INIT)==1){
            SD_init();
            i = is_file_there();
            if (i==7)
               create_log_file();
```

```
            delay_ms(3000);
            lcd_putc("\f");
            return;
         }
   }
}
   else if (input(STOP)!=0){
      disable_interrupts(intr_global);                  //If stop is pressed for 3 seconds the logger will stop and disable all interrupts
      delay_ms(3000);
      if(input(STOP)==1){
         setup_rtc(RTC_DISABLE,0);                           //STOP of RTCC
         setup_rtc_alarm(RTC_ALARM_DISABLE,RTC_ALARM_YEAR,1);
         disable_interrupts(INT_RTC);
      }
      return;
   }
   status = 0;                           //Updating status (Digital inputs)
   if (input(PIN_B0)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B1)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B2)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B3)!=0)
      Status|=1;
   Status*=2;
   if (input(PIN_B4)!=0)
      Status|=1;
```

```
    Status*=2;
    if (input(PIN_B5)!=0)
        Status|=1;
    Status*=2;
    if (input(PIN_D6)!=0)
        Status|=1;
    status*=2;
    if (input(PIN_B15)!=0)
    Status|=1;
}
```

7.1.21 SHOW interrupt function

```
#INT_EXT0
void ext0_isr(void){                    //This function shows on the LCD the current status of the Digital inputs
    if (input(SHOW)!=0){                //If the button is pressed for more than 3 seconds will continue
        delay_ms(3000);

        if (input(SHOW)==1){            //Status (digital inputs) update
            Status = 0;
            if (input(PIN_B0)!=0)
                Status|=1;
            Status*=2;
            if (input(PIN_B1)!=0)
                Status|=1;
            Status*=2;
            if (input(PIN_B2)!=0)
                Status|=1;
            Status*=2;
            if (input(PIN_B3)!=0)
                Status|=1;
            Status*=2;
```

```
        if (input(PIN_B4)!=0)
           Status|=1;
        Status*=2;
        if (input(PIN_B5)!=0)
           Status|=1;
        Status*=2;
        if (input(PIN_D6)!=0)
           Status|=1;
        status*=2;
        if (input(PIN_B15)!=0)
        Status|=1;
        lcd_putc("\fThe value of\n");
        printf(lcd_putc,"i is: %u\r",Status);          //LCD feedback with digital inputs that are ON
        delay_ms(2000);
        lcd_putc("\f");
     }
  }
}
```

7.1.22 set_clock function

```
void set_clock(rtc_time_t &date_time)                    //This function sets the clock at the desired date and time
{
   Year -= 2000;
   printf("The clock will be set with year: %d",Year);
   date_time.tm_year=Year;
   date_time.tm_mon=Month;
   date_time.tm_mday=Day;
   date_time.tm_wday=Weekday;
   date_time.tm_hour=Hours;
   date_time.tm_min=Minutes;
   date_time.tm_sec=Seconds;
}
```

7.1.23 read_clk function

```
void read_clk(void)                                      //This function reads the clock and puts the values in the global variables
{
   rtc_read(&read_clock);
   Year = read_clock.tm_year;
   Month = read_clock.tm_mon;
   Day = read_clock.tm_mday;
   Weekday = read_clock.tm_wday;
   Hours = read_clock.tm_hour;
   Minutes = read_clock.tm_min;
   Seconds = read_clock.tm_sec;
   printf("\r%02u/%02u/%02u %02u:%02u:%02u",Month,Day,Year,Hours,Minutes,Seconds);
}
```

7.1.24 Post-processing in Python and creation of CSV file

This program was written in Python. It takes the data from the DLOGGER.ILM and creates a CSV file. The CSV file Is a structured file that can be opened and further examined with powerful data analysis tools like Excel. Even though graphics can be made with Excel, this program creates 2 graphics of the analog inputs versus Time/Date.

```
from __future__ import print_function
import matplotlib.pyplot as plt
from matplotlib.ticker import EngFormatter                    //Libraries importing
import csv
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import matplotlib.ticker as ticker
from matplotlib import dates
import matplotlib.dates as md
import datetime as dt
import time
f = open('DLOGGER.ILM','rb')
A = f.read()
stringlist=[]
ch1Value=[]
ch2Value=[]
ch3Value=[]
ch4Value=[]                                   //Variables
ch5Value=[]
ch6Value=[]
ch7Value=[]
ch8Value=[]
Date = []
Inputs=[]
Outputs=[]
```

```
ch1Values=[]
ch2Values=[]
ch3Values=[]
ch4Values=[]
ch5Values=[]
ch6Values=[]
ch7Values=[]
ch8Values=[]
curValue=0
curConditions=0
curInputs=[]
curOutputs=[]
curSize=0
curYear=0
curMonth=0
curDay=0
curHours=0
curMinutes=0
curSeconds=0
curDate=[]
i = 0
j = 0
k = 0
string = ''
while k < (len(A)/512):
    for i in range(16):
        while (j < 32):
            if j == 3:
                ch1Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])          //Retrieving Analog channels from logger file
                j+=2
                ch2Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
```

```
j+=2
ch3Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
ch4Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
ch5Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
ch6Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
ch7Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
ch8Values.append((A[i*32 + j + k*512]<<8) + A[i*32+1+j + k*512])
j+=2
curInputs.append(A[i*32 + j + k*512])                    //Retrieving inputs
j+=1
curOutputs.append(A[i*32 + j + k*512])                   //Retrieving outputs
j+=1
curConditions = A[i*32+j+k*512]                          //Retrieving conditions
j+=1
curYear = 2000 + (A[i*32 + j + 1 + k*512])               //Retrieving date and time
j+=2
curMonth = (A[i*32 + j + k*512])
j+=1
curDay = (A[i*32 + j + k*512])
j+=1
curHours = (A[i*32 + j + k*512])
j+=1
curMinute = (A[i*32 + j + k*512])
j+=1
curSeconds = (A[i*32 + j + k*512])
j+=1
```

```
            else:
                j+=1
        j=0
    f.close()
    for i in range(len(ch1Values)):
        curValue += ch1Values[i]
    curValue /= len(ch1Values)
    curValue *= (6.6/65472)
    curValue = round(curValue,3)
    ch1Values = []
    ch1Value.append(curValue)
    curValue = 0
    for i in range(len(ch2Values)):
        curValue += ch2Values[i]
    curValue /= len(ch2Values)
    curValue *= (6.6/65472)
    curValue = round(curValue,3)
    ch2Values = []
    ch2Value.append(curValue)
    curValue = 0
    for i in range(len(ch3Values)):
        curValue += ch3Values[i]
    curValue /= len(ch3Values)
    curValue *= (6.6/65472)
    curValue = round(curValue,3)
    ch3Values = []
    ch3Value.append(curValue)
    curValue = 0
    for i in range(len(ch4Values)):
        curValue += ch4Values[i]
    curValue /= len(ch4Values)
```

```
curValue *= (6.6/65472)
curValue = round(curValue,3)
ch4Values = []
ch4Value.append(curValue)
curValue = 0
for i in range(len(ch5Values)):
    curValue += ch5Values[i]
curValue /= len(ch5Values)
curValue *= (6.6/65472)
curValue = round(curValue,3)
ch5Values = []
ch5Value.append(curValue)
curValue = 0
for i in range(len(ch6Values)):
    curValue += ch6Values[i]
curValue /= len(ch6Values)
curValue *= (6.6/65472)
curValue = round(curValue,3)
ch6Values = []
ch6Value.append(curValue)
curValue = 0
for i in range(len(ch7Values)):
    curValue += ch7Values[i]
curValue /= len(ch7Values)
curValue *= (6.6/65472)
curValue = round(curValue,3)
ch7Values = []
ch7Value.append(curValue)
curValue = 0
for i in range(len(ch8Values)):
    curValue += ch8Values[i]
```

```
    curValue /= len(ch8Values)
    curValue *= (6.6/65472)
    curValue = round(curValue,3)
    ch8Values = []
    ch8Value.append(curValue)
    curValue = 0
    for i in range(len(curInputs)):
        curValue |= curInputs[i]
    curInputs = []
    Inputs.append(curValue)
    curValue = 0
    for i in range(len(curOutputs)):                    //Creation of structured lists for easier CSV file handling
        curValue |= curOutputs[i]
    curOutputs = []
    Outputs.append(curValue)
    curValue = 0
    curValue += curYear
    curValue *=100
    curValue += curMonth
    curValue *=100
    curValue += curDay
    curValue *=100
    curValue += curHours
    curValue *=100
    curValue += curMinute
    curValue *=100
    curValue += curSeconds
    Date.append(curValue)
    curValue = 0
    k+=1
curYear = 0
```

```
curMonth = 0
curDay = 0
curHours = 0
curMinute = 0
curSeconds = 0
temp = ['Date','Channel 1','Channel 2','Channel 3','Channel 4','Channel 5','Channel 6','Channel 7','Channel 8','Conditions','Inputs','Outputs']
stringlist.append(temp)
for i in range(len(Date)):
    stringlist.append(temp)
    temp = []
    curValue = Date[i]/pow(10,10)
    curYear = int(curValue)
    curValue -= curYear
    curValue *= 100
    curMonth = int(curValue)
    curValue -= curMonth
    curValue *= 100
    curDay = int(curValue)
    curValue -= curDay
    curValue *= 100
    curHours = int(curValue)
    curValue -= curHours
    curValue *= 100
    curMinute = int(curValue)
    curValue -= curMinute
    curValue *= 100
    if curValue >= 59:
        curValue = 0
        curMinute += 1
    curSeconds = int(curValue)
```

```
    temp.append(str(curYear) + '/' + str(curMonth) + '/' + str(curDay) + '/' + str(curHours) + ':' + str(curMinute) + ':' + str(curSeconds))
    temp.append(str(ch1Value[i]))
    temp.append(str(ch2Value[i]))
    temp.append(str(ch3Value[i]))
    temp.append(str(ch4Value[i]))
    temp.append(str(ch5Value[i]))
    temp.append(str(ch6Value[i]))
    temp.append(str(ch7Value[i]))
    temp.append(str(ch8Value[i]))
    temp.append(str(curConditions))
    temp.append(str(Inputs[i]))
    temp.append(str(Outputs[i]))
stringlist.append(temp)
file = open("MYFILE.csv", "a",newline='')          //CSV file creation
c = csv.writer(file,delimiter=',')
for row in range(1,len(stringlist)):
    c.writerow(stringlist[row])
file.close()
print (stringlist)
datafile = cbook.get_sample_data('myfile.csv', asfileobj=False)                //Reading the CSV file
print ('loading %s' % datafile)
r = mlab.csv2rec(datafile)
formatter = EngFormatter(unit='V', places=3)
r.sort()
N = len(r)
ind = np.arange(N)
def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y/%m/%d %H:%M:%S')
fig = plt.figure()
ax = fig.add_subplot(111)
```

```
ax.plot(ind,r.channel_1,'o-')
ax.plot(ind,r.channel_2,'o-')
ax.plot(ind,r.channel_3,'o-')
ax.plot(ind,r.channel_4,'o-')                    //Plotting channels on graph 1
ax.plot(ind,r.channel_5,'o-')
ax.plot(ind,r.channel_6,'o-')
ax.plot(ind,r.channel_7,'o-')
ax.plot(ind,r.channel_8,'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax.yaxis.set_major_formatter(formatter)
fig.autofmt_xdate()
plt.xticks(rotation=60)
ax.set_xticks(ind)
ax.grid(color='k', linestyle=':')
fig2 = plt.figure()
ax2 = fig2.add_subplot(111)
ax2.plot(ind,r.channel_1,'-')
ax2.plot(ind,r.channel_2,'-')
ax2.plot(ind,r.channel_3,'-')
ax2.plot(ind,r.channel_4,'-')
ax2.plot(ind,r.channel_5,'-')                    //Plotting channels on graph 2
ax2.plot(ind,r.channel_6,'-')
ax2.plot(ind,r.channel_7,'-')
ax2.plot(ind,r.channel_8,'-')
ax2.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
ax2.yaxis.set_major_formatter(formatter)
fig2.autofmt_xdate()
plt.xticks(rotation=60)
ax2.set_xticks(ind)
ax2.grid(color='k', linestyle=':')
plt.show()                                       //Showing graphs
```

7.2 Used literature

[1] http://en.wikipedia.org/wiki/Data_logger
[2] http://www.omega.com/techref/pdf/LOGGERINTRO.pdf
[3] http://www.ni.com/white-paper/2693/en/
[4] http://www.ni.com/data_logger/
[5] https://en.wikipedia.org/wiki/MultiMediaCard
[6] http://en.wikipedia.org/wiki/Flash_memory
[7]http://www.geoscientific.com/technical/tech_references_pdf_files/Data_Logger_Fundamentals.pdf
[8] http://en.wikipedia.org/wiki/Analog_signal
[9] https://en.wikipedia.org/wiki/Digital_signal
[10] http://en.wikipedia.org/wiki/Pulse-width_modulation
[11] https://en.wikipedia.org/wiki/Discretization
[12] http://en.wikipedia.org/wiki/Nyquist_frequency
[13] http://en.wikipedia.org/wiki/Quantization_(signal_processing)
[14] http://en.wikipedia.org/wiki/Analog-to-digital_converter
[15] *Lessons In Electric Circuits* copyright (C) 2000-2013 Tony R. Kuphaldt
[16] http://en.wikipedia.org/wiki/Secure_Digital
[17] http://en.wikipedia.org/wiki/Lithium_battery
[18]http://suite101.com/article/alkaline-battery-uses-and-other-information-on-alkaline-batteries-a279227
[19] http://simple.wikipedia.org/wiki/Sealed_lead_acid_battery
[20] http://en.wikipedia.org/wiki/Boost_converter
[21] http://www.microchip.com/pagehandler/en-us/products/picmicrocontrollers
[22] http://en.wikipedia.org/wiki/Zener_diode
[23] http://en.wikipedia.org/wiki/Opto-isolator
[24] http://en.wikipedia.org/wiki/Liquid-crystal_display
[25] http://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html
[26] http://www.beginningtoseethelight.org/fat16/index.htm
[27] http://elm-chan.org/docs/mmc/mmc_e.html