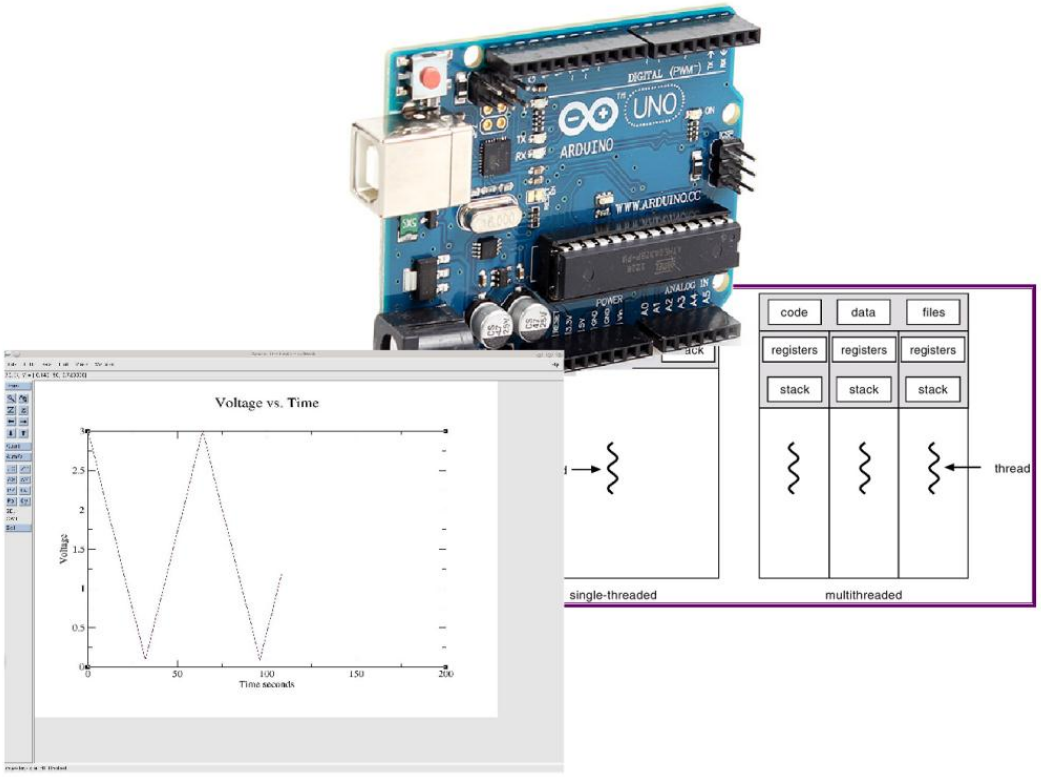
Technological Educational Institute Of Crete

Electrical Engineering Department

Bachelor Thesis

Remote Surveillance of Power Systems Using Arduino Platform



George Magoulakis (3459)

jmagoylakis@gmail.com

Supervisor: Prof. Dr. Grammatikakis Miltiades

Evaluation Committee: Dr. Sfakiotakis Michael

Dr. Vasilakis Konstantinos

date of presentation: 14/07/2014

Acknowledgements

By the completion of this thesis, I would like to thank all those who supported and encouraged me in this thesis preparation and for my entire course at the T.E.I. of Crete and particularly my parents and my friends who have always been my support.

In particular, I would like to thank my supervisor prof. Dr. Grammatikakis Miltiades for his guidance, his patience and the help he provided me throughout this thesis.

Abstract

The topic of this thesis is the study and evaluation of the following three programming models implemented on an Embedded System: the Serial, the Asynchronous Parallel and the Synchronous Parallel. The parallel programming models are based on Protothreads, which are lightweight threads, designed for severely memory constrained embedded systems. This study provides the opportunity to see and understand in depth how these models work and to comprehend the precise meaning and operation of an embedded system. The embedded system which was used for the implementation of this thesis is the Arduino board, along with some of its peripherals (sensors, actuators etc.) for collecting and processing the collected data of the project.

For the accomplishment of the above objective, individual steps were followed which constitute the methodology followed in this work. These are summarized below.

At first, concerning the methodology followed in this thesis, we studied issues which relate to theoretical computer engineering concepts, such as embedded systems, in particular real-time embedded systems, the Arduino microcontroller and sensor networks, as well as programming models. We examined several types of threads (Linux threads, POSIX threads, Protothreads etc.) as well as the ability to control and monitor data using specialized programs, such as the Processing IDE and Grace for visualization.

Next, we describe the project, which is separated in three parts, one for each programming model. The case study is related to the design and implementation of a Remote Monitoring/Control Power Strip device (RC Power Strip). We used C-C++ (Arduino IDE) and the Linux OS (OpenSUSE 12.3) and we examined how these models and their operating parameters affect application performance and time behaviour.

Based on this methodology, this thesis provides the opportunity to draw important remarks and conclusions on the efficiency of parallel programming models using Protothreads.

ΣΥΝΟΨΗ

Το θεματικό αντικείμενο της παρούσας πτυχιακής εργασίας είναι η μελέτη τριών μοντέλων προγραμματισμού: του Σειριακού, Ασύγχρονου Παράλληλου και Σύγχρονου Παράλληλου, τα οποία μελετήθηκαν και αξιολογήθηκαν σε ενσωματωμένο σύστημα. Τα δύο παράλληλα μοντέλα βασίστηκαν στην τεχνολογία των Protothreads, τα οποία εντάσσονται στην κατηγορία των lightweight threads, και η χρήση τους ενδείκνυται σε ενσωματωμένα συστήματα με πολύ περιορισμένη μνήμη. Η μελέτη αυτή δίνει την δυνατότητα στον ενδιαφερόμενο να δει και να κατανοήσει σε βάθος πώς λειτουργούν αυτά τα μοντέλα, καθώς και να κατανοήσει την ακριβή σημασία και λειτουργία των Ενσωματωμένων Συστημάτων. Το ενσωματωμένο σύστημα που χρησιμοποιήθηκε είναι η πλατφόρμα Arduino μαζί με κάποια περιφερειακά του (αισθητήρες κ.α.), τα οποία χρησιμοποιήθηκαν για την συλλογή και την επεξεργασία των δεδομένων.

Για την επίτευξη του παραπάνω στόχου, ακολουθήθηκαν επιμέρους βήματα τα οποία συνοψίζονται παρακάτω και τα οποία συνιστούν και την μεθοδολογία που ακολουθήθηκε στην παρούσα εργασία.

Πρώτα από όλα, στο μεθοδολογικό μέρος της πτυχιακής εργασίας μελετήθηκαν θέματα που αφορούν τις απαραίτητες θεωρητικές έννοιες που αφορούν τα ενσωματωμένα συστήματα, τα ενσωματωμένα συστήματα πραγματικού χρόνου, τον μικροελεγκτή Arduino, τα δίκτυα αισθητήρων, όσο και τα ανάλογα μοντέλα προγραμματισμού. Εξετάσθηκαν δηλαδή τύποι νημάτων (Linux threads, POSIX threads, Protothreads), όπως και η δυνατότητα ελέγχου και παρακολούθησης δεδομένων με την χρήση διαφόρων προγραμμάτων (Processing IDE, Grace).

Στην συνέχεια, αναπτύξαμε το project σε τρία σκέλη, ένα για κάθε μοντέλο προγραμματισμού. Μελετήσαμε την χρονική συμπεριφορά των μοντέλων αυτών και συγκρίναμε τα χαρακτηριστικά τους με βάση εφαρμογή που αφορά απομακρυσμένη πρόσβαση και έλεγχος συσκευών παροχής ενέργειας (RC Power Strip). Χρησιμοποιήσαμε C – C++ (Arduino IDE) και λειτουργικό σύστημα Linux (OpenSUSE 12.3) και εξετάσαμε πώς αυτά τα μοντέλα και οι αντίστοιχες παράμετροι λειτουργίας τους επηρεάζουν την απόδοση της εφαρμογής.

Με βάση τη μεθοδολογία αυτή, η πτυχιακή εργασία δίνει την δυνατότητα να εξάγουμε σημαντικά συμπεράσματα ως προς την αποδοτικότητα των παράλληλων μοντέλων προγραμματισμού και των Protothreads.

Table of contents

Table of Pictures

1. Embedded Systems

1.1 Definition of the Embedded Systems

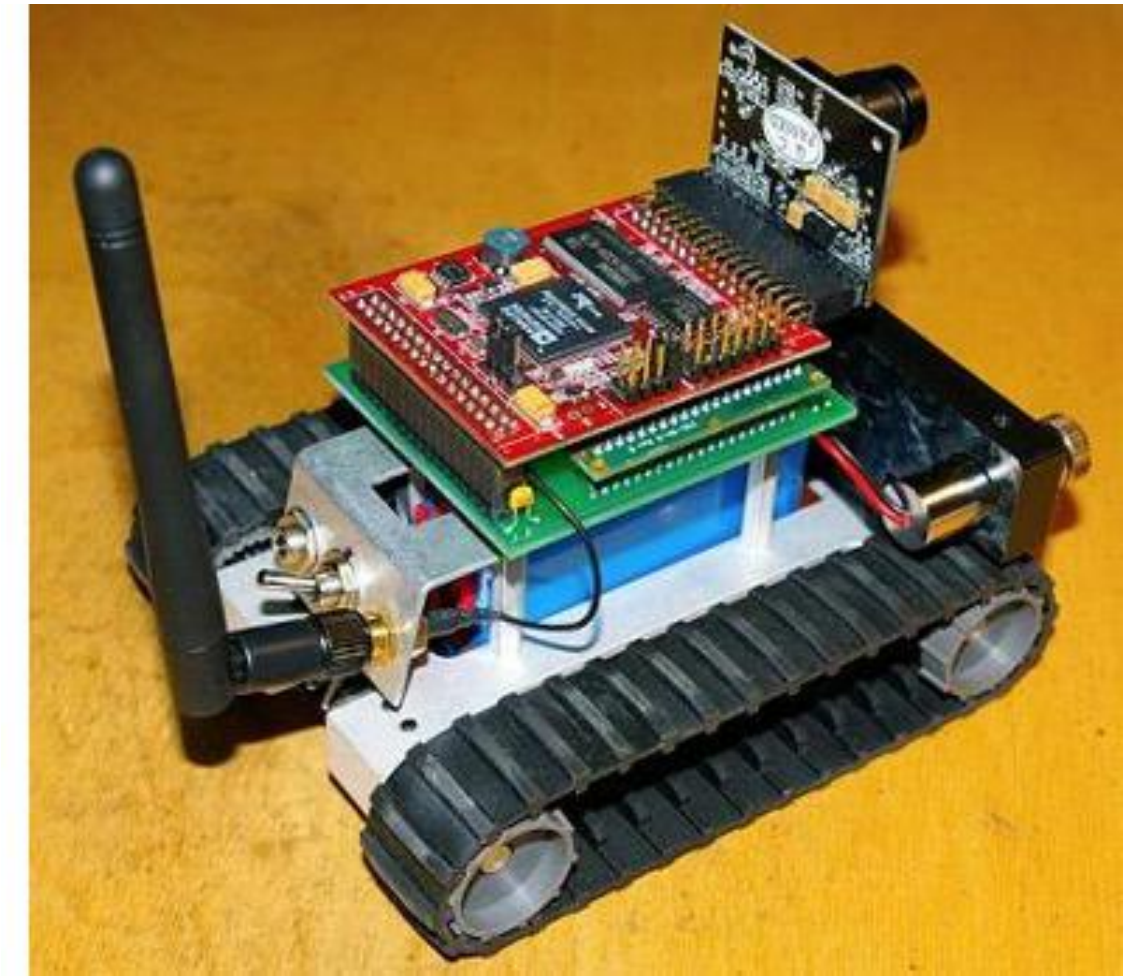


Figure 1 – 1: A typical autonomous tank vehicle

What are the Embedded Systems?

Although it is really difficult to define the term "embedded systems", we can initially describe them as computer systems that do not use a monitor, keyboard or mouse (general purpose computers). According to this general definition of the embedded systems, we can easily imagine where we can find applications of them in our lives.

Consumer applications (mp3 players, video cameras etc.), telecommunications (such as mobile and smart phones), cooking, industrial (sensors and thermostats etc.), automotive, medical (for vital sign monitoring), military applications and games (e.g. see Figure 1 – 1) are just a few examples. One of the first recognizably modern embedded systems was the Apollo Guidance Computer [1] (see Figure 1 – 2) which was developed by Charles Stark Draper at the MIT Instrumentation Laboratory in 1966.

Figure 1 – 2: Apollo Guidance Computer

But, what is exactly an (Embedded) **System**? System is a way to organize, work and perform one or many tasks according to rules, or schedules of a program. Thus, Embedded Systems are software, embedded into hardware. This means that systems can be used for implementing dedicated computing, communication and synchronization protocols, often related to a specific schedule associated with **real-time computing** constraints.

1.1.1 Characteristics of Embedded Systems

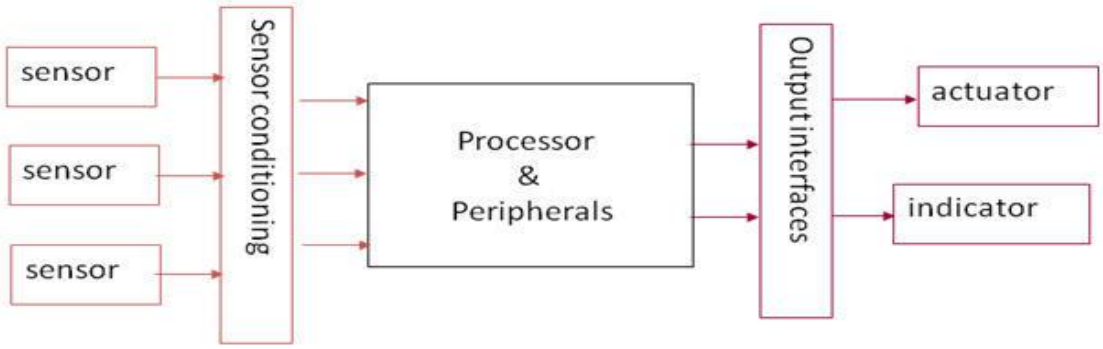


Figure 1 – 3: Block diagram of a typical embedded system.

Embedded Systems have the following characteristics.

- They are **application specific** and **specialized**. This means that they are designed for a specific application and programs are running repeatedly.
- They are optimized for performance (execution time, latency), energy efficiency, code size, dimensions and cost.
- They are typically designed to meet **real-time constraints**. This means that they are designed to react to stimuli from the object that they control, within the given time interval. Slower reaction than the time schedule is a problem for Real-time systems.
- They are interacting with the environment through sensors and actuators (see Figure 1 – 3). Therefore they are typically reactive systems.
- They generally have minimal or no user interface.

1.1.2 Elements of Embedded Systems

The elements which comprise an embedded system are **Hardware** and **Software**.

1. Focusing first on **Hardware**, the core element of an embedded system is the processor who is programmed to perform specific tasks using a variety of options. Processors are divided into General Purpose Microprocessors (μP) and Microcontrollers – Embedded Processors.

    a. General Purpose Microprocessors

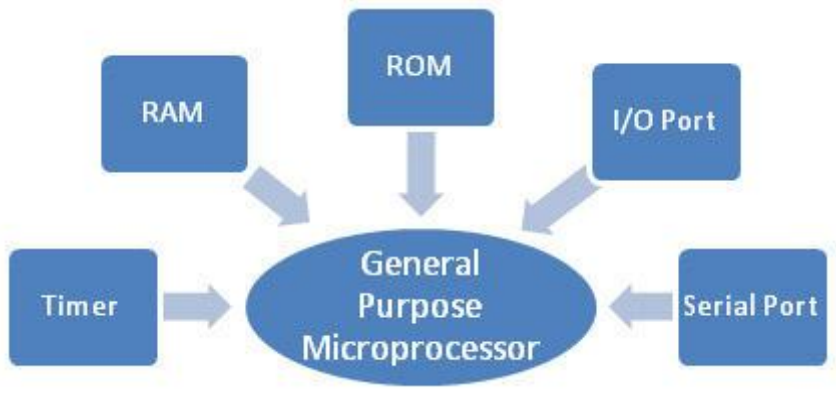


Figure 1 – 4: A General Purpose Microprocessor

Although a general purpose μP contains on a single chip an ALU, a Program Counter, a Stack Pointer, registers, a clock and interrupt circuits, these elements do not compose a complete computer. For this reason, as shown in Figure 1 – 4, ROM and RAM, memory decoder, oscillator and a number of serial and parallel ports are also necessary.

General purpose μPs are designed to run a large number of applications and are used to prototype embedded system, offering a short design time since the only thing that needs to be developed is the software. The characteristics of a general purpose μP are:

- High cost
- High speed
- High Power consumption
- Large-scale architecture
- Large memory size
- Extra capabilities, e.g. onboard flash and cache
- External bus interface

b. Microcontrollers – Embedded Processors

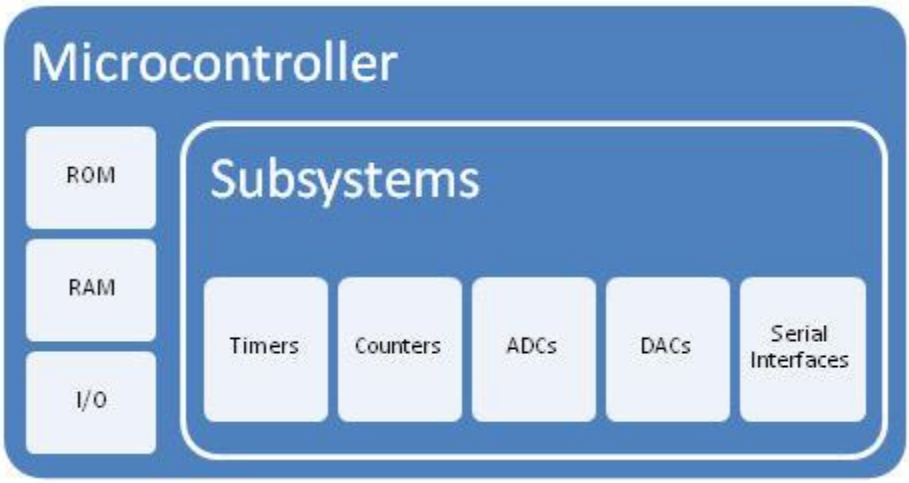


Figure 1 – 5: A typical Microcontroller

Microcontroller is a functional computer system on a chip, which is not expandable since it has no external bus interface. It has a processor, memory and several peripheral devices, all in one chip. This results in low-power consumption and compactness. The characteristics of a microcontroller are

- Low cost
- Low speed
- Low Power
- Small architecture
- Small memory size
- Onboard flash
- Limited I/O

Software is the soul of embedded systems. We can use a variety of languages and libraries according to the tasks that we want to run, depending on the nature of the application. For example, a particular approach (language, library or protocol) which is good for control-dominated applications might not be as good in other applications. In fact, several languages are involved in system design. While Hardware Design Languages (HDLs), such as Verilog [2] or VHDL [3] (or even SystemC [4]) are used to describe hardware components, General Programming Languages, especially High Level Languages (C, C++, Java, ADA etc.) or Assembly (symbolic and difficult to understand) are ioften used for embedded software. Other specialized Languages are better for specific application domains, e.g. dataflow or streaming languages for digital signal processing (DSP) applications, Esterel for real-time systems and SDK framework for embedded architecture development in specialized (Xilinx - based) platforms.

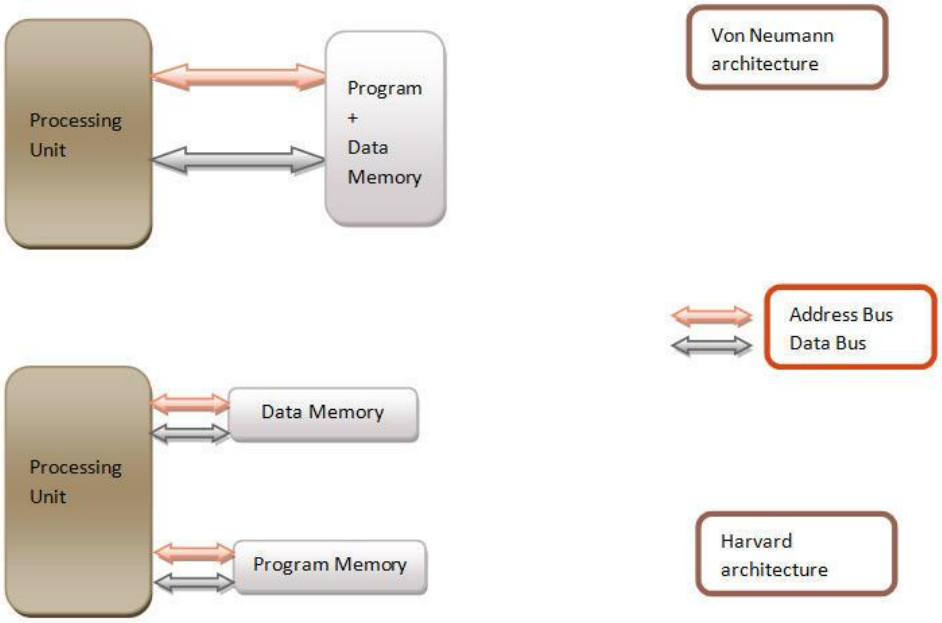


Figure 1 – 6: Traditional versus Harvard Architecture

Another related issue concerns the computer architecture. While in the von Neumann model, program and data memory is shared between them including address bus and data bus, in the Harvard architecture, the data memory and the program memory are separate, and thus the data bus and the address bus are separate for each memory (cf. Figure 1 – 6).

1.1.3 Embedded Systems - Interfacing to Physical World

Previously we mentioned that embedded systems are using sensors and actuators in order to interact with the external environment. Sensors are like "senses" for the embedded systems and actuators are like "limbs". Some of the physical parameters that are used for this purpose are light, temperature, acceleration, speed, mass, distance etc. and the communication interfaces that are used for transducing these sets of physical parameters to an embedded system, are:

- Synchronous or Asynchronous Serial Communication Interfaces

- Universal Serial Bus (USB) or other Peripheral Buses
- Networks (Ethernet, Controller Area Network etc.)
- Analog to Digital and Digital to Analog Converters (ADC/DAC).

1.1.4 Designing an Embedded System

The design of an embedded system consists of several stages (which may overlap in time):

- Requirement analysis
- Defining system specifications
- Modeling the system under design. Examining different architectures and algorithms and performing preliminary simulations for concept validation. Factoring the task into smaller subtasks and modeling their interactions.
- HW-SW partitioning. Task allocation the tasks into hardware or software (called HW/SW Co-design) proceeds in parallel with modeling.
- Detailed VHDL simulation and choice of process technology.
- Resource Analysis, in terms of performance, energy, cost, time-to-marker, manpower etc.
- Identification of components and development tools (routing & placement)
- Circuit design and Schematic Capture, PCB layout design or custom ASIC fabrication
- Firmware development - debugging & testing
- System Integration
- Testing – functional, environmental
- Certifications, if required
- Final documentation

1.2 Real-time Embedded Systems

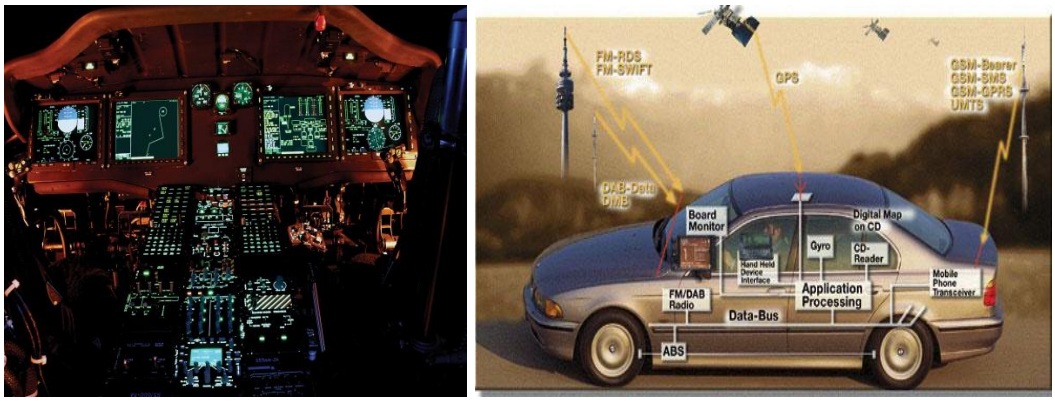


Figure 1 – 7: Real-time embedded systems

As mentioned in the previous section, real-time systems are used for specialized work based on a specific application which exhibits **real-time computing constraints.** This is common in many application domains, such as chemical and nuclear plants, space applications and transportation (see Figure 1 – 7).

But, what is actually a real-time system? Real-time systems produce a result of a process which is executed, and this result, **not only must be logically correct**, but it must become available **within a certain time period.**

In general, we consider two kinds of tasks: periodic and aperiodic tasks. In periodic tasks, a period is defined as the amount of time between iterations of the repeated task. In contrast to Periodic tasks, aperiodic tasks respond to randomly arriving events.

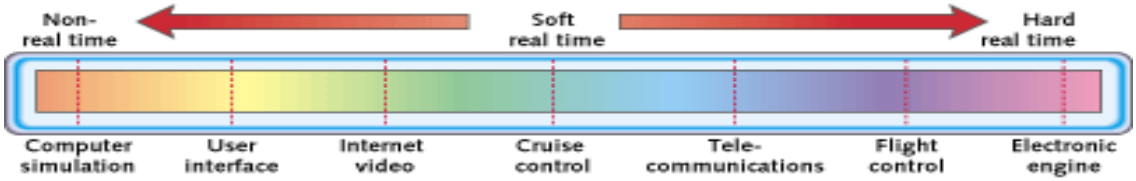


Figure 1 – 8: Real-time embedded systems time spectrum

Reflecting the importance of time in a real-time system, we can say that a real time system does not need to be very fast, but it **must return correct results predictably, i.e. always before a deadline**. Hence, **deadline** is a **time limit** for a system operation, after which any result may be worthless, depending on how important is the time limit of the specific application. If we consider as a scale the importance of keeping the deadlines on individual applications, we can divide real-time systems into non real-time systems, soft real-time systems and hard real-time systems. This scale is illustrated in Figure 1 – 8.

As we can see in the time spectrum, at one end, where non-real-time systems are located, there are no deadlines or deadlines are not important. At the other end, where hard-real-time

systems are located, all deadlines must not be missed, otherwise this event results critical system failure. Between these two ends, there are soft real-time systems.

Missing a deadline on a hard-real-time system, it results in system failure, e.g. causing potential injuries or even death to system users. One example of a hard-real-time system is the flight controller. If the controller is not responding in the proper time limits and misses the deadline, it is possible that the aircraft will crash to the ground. So, there are no mistakes that are allowed for a controller (part of the aircraft navigation system).

From the opposite side, soft-real-time systems often have time limits, but occasional violation does not result in critical system failure. But under no circumstances this means that it is acceptable to have a continuous violation. For example, considering the cruise control application, if the system fails to measure the current velocity in time, it can use the previous value of it, without resulting to an error as the difference between the old and the current value of the speed is very small. But if it misses several consecutive values, it might leads to a problem, because the cruise control would likely stop meeting application requirements for the current speed.

Predictability is a term that is used to describe real-time systems. When we say that a real-time system is predictable, it means that its timing behaviour is within an acceptable time range. In other words, a real-time system must behave in a way that can be predicted mathematically. Thus, in order to design a predictable real-time system, we need to know the period, the deadline and the worst-case execution time of each individual application. Taking into consideration these parameters, a system can be designed with the most appropriate scheduling algorithm, to ensure that it meets all timing constraints for predictability.

A special property of a predictable real-time system is determinism. Determinism represents the ability to ensure the execution of an application without concerns that outside factors may upset the execution in an unpredictable way. That means its timing behaviour can be predetermined. For example, we can consider a router (device) with pre-allocated slots (in time and space) for certain types of packets (tasks). Execution for these packets only occurs during those time slots. Therefore, upper bounds on the latency for every packet can be computed precisely and we avoid anomalies in system predictability [5].

1.3 Sensors and Sensor Networks

1.3.1 Sensors

Sensors are devices which can measure a physical quantity and convert it into a signal which can be read by an observer, such as an electronic instrument. In other words, a sensor is a device which responds to an input physical quantity (e.g. temperature) by generating a functionally related output function usually in a form of an electrical or an optical signal. As For example, by using a mercury-in-glass thermometer, we can easily read the temperature of the environment in a calibrated glass tube. Another way to measure temperature is to use a thermocouple which converts temperature differences into voltage which we can measure using a voltmeter.

Figure 1 – 9: Examples of Sensors and Sensor Networks

Nowadays, sensors are having numerous applications in our lives (see Figure 1 – 9). They are used in simple or complicated applications in every embedded system today, such as automotive, train and aerospace industry, computer monitoring systems, fire detection, electricity distribution, industrial automation, automated and smart homes, audio/image/video surveillance, traffic monitoring, medical device monitoring, weather/climate monitoring, air traffic control and robot control.

Actually, without sensors, it would be difficult to design or even imagine a control, surveillance or security system. An airplane without sensors will not have the ability to measure the distance between the aircraft and the ground. Hence, the idea of flying would remain as primitive as in the era of the Wright brothers [6].

1.3.2 Ideal and Real Sensors

A sensor needs to follow these three rules.

- First rule is that the sensor must be sensitive only to the physical quantity for which the sensor has been created.
- Second rule is that the sensor must not be influenced by any other kind of physical quantity that might influence the system.
- Third rule is that the sensor must not influence the measured physical quantity (minimal intrusiveness), so that in it provides correct values of the measured quantity.

If all previously mentioned rules are obeyed, the sensor would be ideal. An ideal sensor is typically a linear or logarithmic mathematical function of the measurement. Its output is an analogue signal, related to the value of the measured quantity. If a sensor needs to be used by a digital system, then its analogue output must be converted into digital, using an Analogue to Digital Converter.

In real sensors, deviations are observed. The resolution of a sensor is the smallest deviation in the measured physical quantity that it can detect. It obviously relates on the accuracy of the sensor.

Sometimes the sensitivity might not be appropriate for the application in which a sensor is used. Other times deviations due to non-linearity, hysteresis, noise and many other conditions e.g., related to the sampling frequency are observed. These deviations can be categorized as

systematic errors and random errors. Systematic errors can be reduced by calibrating the sensitivity of the sensors while random errors (e.g. noise) interfering with the hardware can usually be reduced by placing filters.

1.3.3 Sensor Networks

Previously it was mentioned that sensors are used for monitoring environmental or physical quantities. However, some of these measurements are really difficult to perform, since they must be placed under extreme conditions. For example, using instruments inside the volcano crater is required in order to predict a volcanic eruption or an earthquake. Similarly, placing sensors on satellites before sending them in deep space to measure the magnetic field of the earth, or the ultraviolet radiation that comes from the Sun is an extreme task. This leads us to create sensor networks which can monitor an environment from a safe distance and transmit data far away from their location. Such achievements can help predict physical disasters or understand our planet and our galactic neighbourhood even better.

So, a sensor network is a group of specialized transducers [7] with a communication infrastructure intended to monitor and record conditions at diverse locations. Commonly monitored parameters are temperature, humidity, pressure, wind direction and speed, illumination intensity, vibration intensity, sound intensity, power-line voltage, chemical concentrations, pollutant levels, and vital body functions.

A sensor network consists of multiple detection stations called sensor nodes, each of which is small, lightweight and portable. Every sensor node is equipped with a transducer, microcomputer, transceiver and power source. The transducer generates electrical signals based on sensed physical effects and phenomena. The microcomputer processes and stores the sensor output. The transceiver, which can be hardwired or wireless, receives commands from a central computer and transmits data back to that computer. The power source for each sensor node is derived from the electric utility or from a battery.

2. Arduino

Figure 2 – 1: Several types of Arduino Boards

According to the official site [8], the Arduino is an open-source electronics prototyping platform based on flexible, lightweight easy-to-use hardware and software. In other words, the Arduino is an Embedded Computing Platform which can be viewed as an interactive system.

Arduino platforms can be used for developing stand-alone, possibly distributed applications. Arduino can also be accessed from a computer in order to support data exchanges, e.g. monitoring information. It is intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments.

The Arduino board is always designed with an Atmel AVR microprocessor, a crystal oscillator and a 5-volt linear regulator. However, depending of the type of use, different Arduino boards can support a USB connector, Pins and others elements.

Today, a plethora of Arduino boards exists (see Figure 2 – 1), depending on the embedded application or of interest. In **Table 1**, official Arduino boards are presented [9]. Although there are many official Arduino versions, there are also several unofficial Arduino based platforms (e.g. Funduino [10], Sainsmart [11] etc.) because both, the manufacturing process and the programs which are being developed, are Open source. So, anyone who wishes to build his own Arduino, he can freely make one without paying royalties to the designer.

Name	Processor		Dimensions	Host	Voltage	Flash	EEPROM	SRAM	I/O			Release Date
	Processor	Frequency	(mm)	Interface	(V)	(kB)	(kB)	(kB)	Digital I/O	Digital I/O with PWM (Pins)	Analog Input (Pins)	
Arduino Leonardo	Atmega32u4	16 MHz	68.6 × 53.3	USB 32u4	5	32	1	2.5	14	6	12	23/7/2012
Arduino UNO R3	ATmega328P	16 MHz	68.6 × 53.3	USB 16u2	5	32	1	2	14	6	6	24/9/2010
Arduino UNO R1&R2	ATmega328P	16 MHz	68.6 × 53.3	USB 8u2	5	32	1	2	14	6	6	24/9/2010
Arduino DUE	AT91SAMX8E	84 MHz	101.6 × 53.3	USB 16u2 + native host	3.3	512	0	96	54	12	12	22/10/2012
Arduino Mega2560	ATMega2560	16 MHz	101.6 × 53.3	USB 16u2	5	256	4	8	54	14	16	24/9/2010
Arduino Ethernet	ATmega328	16 MHz	68.6 × 53.3	Ethernet Serial interface - Wiznet Ethernet	5	32	1	2	14	4	6	13/7/2011
Arduino Fio	ATmega328P	8 MHz	66.0 × 27.9	Xbee Serial	3.3	32	1	2	14	6	8	18/3/2010
Arduino Nano	ATmega328	16 MHz	43.18 × 18.54	USB FTDI	5	16/32	0.5/1	1.0/2	14	6	8	15/5/2008
LilyPad Arduino	ATmega168V or ATmega328V	8 MHz	51 mm		2.7-5.5	16	0.5	1	14	6	6	17/10/2007
Arduino Mega ADK	ATmega2560	16 MHz	101.6 × 53.3	8U2 MAX3421E USB Host	5	256	4	8	54	14	16	13/7/2011
Arduino Esplora	Atmega32u4	16 MHz	165.1 × 61.0	32u4	5	32	1	2.5				10/12/2012
Arduino Micro	ATmega32u4	16 MHz	17.8 × 48.3		5	32	1	2.5	20	7	12	8/11/2012
Arduino (Pro) Mini	ATmega168[30] (Pro uses ATMega328)	8 MHz (3.3 V model) or 16 MHz (5 V model)	17.8 × 33.0		5 or 3.3	16	0.5	1	14	6	6	23/8/2008

Table 1: List of all the official Arduino boards in 2014 [9]

2.1 Hardware

2.1.1 Arduino UNO R3 Microcontroller

Figure 2 – 2: The Arduino Uno R3 Board

The Arduino Uno Rev.3 microcontroller (see Figure 2 – 2) has the following characteristics according to the official site of Arduino:

Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recom.)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM [12] output)
Analogue Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

Arduino board can be powered via USB, or via an external power adapter of 9 Volts DC and 1 Ampere. The power source is chosen automatically. It has power PINS which are: VIN (the input pin of the Arduino board which is connected to an external power supply), 5V (it gives a regulated 5V from the regulator on the board), 3V3 (it gives 3.3 volts), the IOREF (it provides the voltage reference with which the microcontroller operates) and the GND.

Arduino has a Serial communication (RX & TX), external interrupts, PWM Pins (3, 5, 6, 9, 10, and 11), SPI Pins (10, 11, 12, and 13), and TWI [13] (A4 or SDA pin and A5 or SCL pin.

Support TWI communication using the Wire library), AREF (Reference voltage for the analogue inputs), Reset (Bring this line LOW to reset the microcontroller). The ATMega328 PIN mapping is shown in Figure 2 – 3:

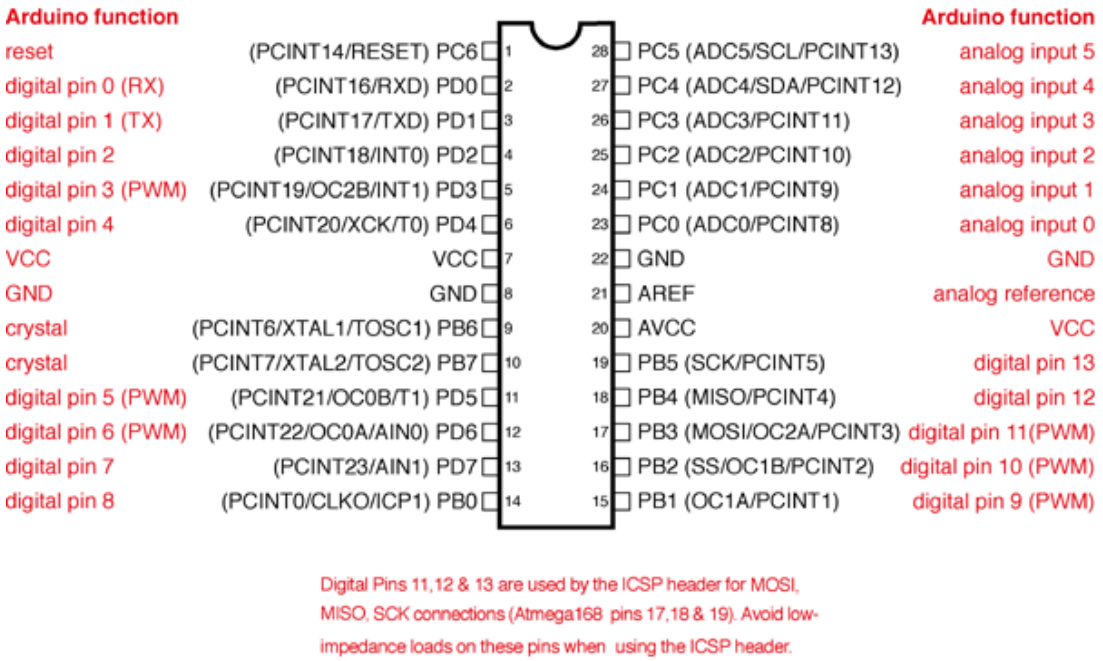


Figure 2 – 3: ATMega328 PIN Mapping

The Arduino UNO can communicate with a computer, with another Arduino, or with other microcontrollers. The ATmega328 provides UART TTL (5V) serial communication, which is available on digital pins 0 (RX) and 1 (TX).

The Arduino software includes a serial monitor which allows simple text to be sent to and from the Arduino board. The RX and TX LEDs on the board will flash when data is being transmitted via the USB-to-serial chip and USB connection to the computer (but not for serial communication on pins 0 and 1). The ATmega328 also supports I2C (TWI) and SPI communication. The Arduino software includes a Wire library to simplify use of the I2C bus.

Finally, the Arduino can be reset by a button that is assembled on its board. But if the Arduino is powered by USB, every time it starts, it does a reset. It also has the ability to reset automatically by a program running from a PC. One of the hardware flow control lines (DTR) of the ATmega8U2/16U2 is connected to the reset line of the ATmega328 via a 100 nanofarad capacitor. When this line is asserted (taken low), the reset line drops long enough to reset the chip.

2.1.2 Arduino Sensors

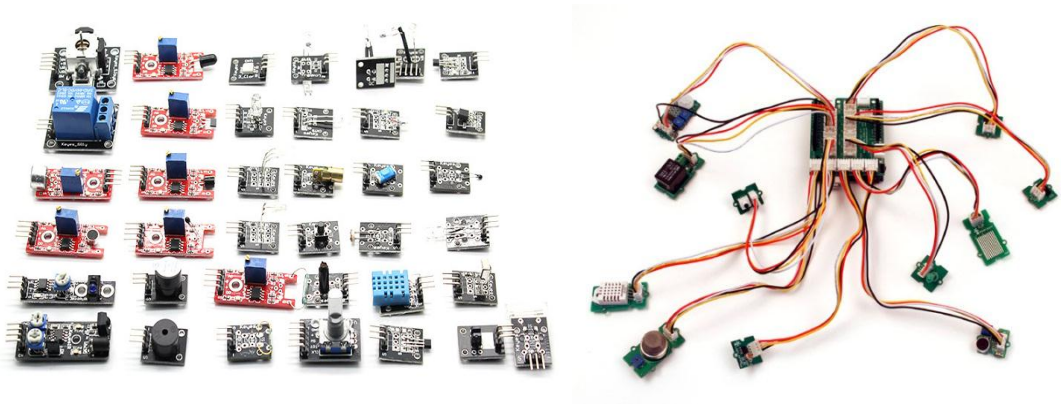


Figure 2 – 4: Several types of Arduino Sensors

As it was foretold in a previous section of this thesis, Arduino belongs to embedded systems, which means that it communicates with its environment through sensors. As the Arduino became known and spread all over the world, new sensors were designed. At the same time, the need of measuring different physical quantities led to design better control and monitoring systems. That led to the design of a plethora of sensors which, as the years were passing, were appropriate for measuring each physical quantity at different levels of accuracy. In the following link [14] a variety of Arduino sensors is shown with their current price (18/11/2013).

Throughout this project different sensors have been used, for which we provide below a deeper analysis of their physical characteristics.

- **Photosensitive Detection Switch Light Sensor Module**

Compact light sensor module with on-board photoresistor, with three pins for 5V power supply, TTL electrical level (or SCM) and GND power respectively. This light sensor module comes with indicator for output signal and the sensitivity is adjustable.

Main Chip: LM393, photoresistor

Working Voltage: DC 3V ~ 5V

Single-way output signal with indication

Low-level output for effective signal

Sensitivity is adjustable

Perfect for light control applications

Overall Size: Approx. 65 x 11 x 13 mm

- **5V 4-Channel Relay Module Switch Board For Arduino**

5V 4-Channel Relay interface board, and each one needs 15-20mA Driver Current.

Equipped with high-current relay, AC250V 10A; DC30V 10A.

Standard interface that can be controlled directly by microcontroller (Arduino, 8051, AVR, PIC, DSP, ARM, ARM, MSP430, TTL logic).

Indication LED's for Relay output status.

- **1602 LCD module for Arduino**

LCD display module with blue backlight.

Wide viewing angle and high contrast.

Built-in industry standard HD44780 equivalent LCD controller.

Commonly used in: copiers, fax machines, laser printers, industrial test equipment, networking equipment such as routers and storage devices.

LCM type: Characters

Can display 2-lines X 16-characters.

Operate with 5V DC.

Module dimension: 80mm x 35mm x 11mm.

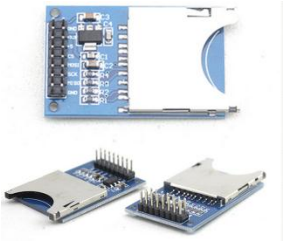
Viewing area size: 64.5mm x 16mm.

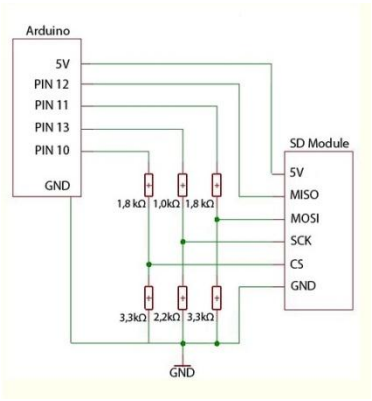
Along with these sensors, we have also used various electronic components such as photoresistor, potentiometers, resistors of various values, LED, breadboard etc.

2.1.3 Memory – SD Card Module

In this design work, we had to record different runtime measurements that were carried out in Arduino, mostly for measuring time or voltage, as well as the states of the relay. The reasons for which these measurements were made will be explained in a next unit of this thesis.

For recording the above values, we choose an SD card of 1 GB, along with an SD Card module.





The module adopts a pop-up SD card interface, designed with a double interface, convenient for pinhole connection.

Features:

All SD SPI pins output, MOSI, SCK, MISO and CS.

Support 5V/3.3V input

It is easily interfaced as a peripheral to Arduino sensor shield module.

Through programmable, i.e. we can read and write to the SD card by using the Arduino.

SD Card is more commonly used with MP3 Player, digital cameras, MCU/ARM system control.

Specifications:

Item size:51*30*4mm

Net weight: 8g

Colour: Blue

2.1.4 Ethernet Shield

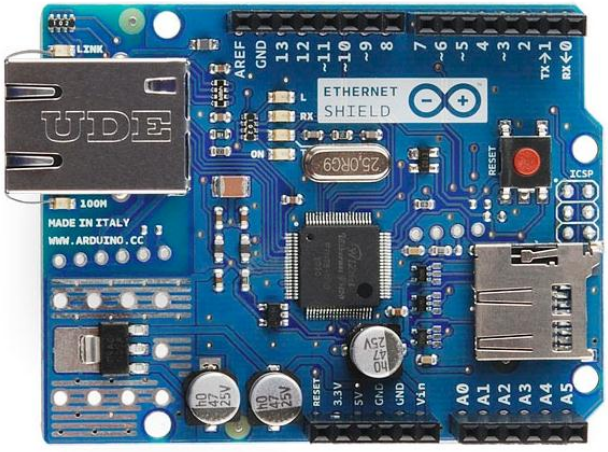


Figure 2 – 5: Ethernet Shield

The Ethernet Shield (see Figure 2 – 5) allows an Arduino or compatible board to connect to the internet. It is based on the Wiznet W5100 Ethernet chip providing a simplified network

(IP) stack capable of both TCP and UDP. Its data sheet is attached at the end of this project. The Ethernet Shield supports up to four simultaneous socket connections. Using the Ethernet library we can write appropriate software sketches that can connect to the internet using the shield.

The Ethernet shield connects to an Arduino - or compatible board using long wire-wrap headers which extend through the shield. This keeps the pin layout intact and allows for another shield to be stacked on top.

The latest revision of the shield adds a micro-SD card slot, which can be used to store files for serving over the network. An SD card library is not yet included in the standard distribution. The Shield includes a reset controller, to ensure that the W5100 Ethernet module is properly reset on power-up. Previous revisions of the shield were not compatible with the Mega and needed to be manually reset after power-up.

The Shield communicates with both the W5100 and SD card using the SPI bus (through the ICSP header). This is on digital pins 11, 12, and 13 on the Arduino Duemilanove and pins 50, 51, and 52 on the Arduino Mega. On both boards, pin 10 is used to select the W5100 and pin 4 for the SD card. These pins cannot be used for general I/O. On the Mega, the hardware SS pin, 53, cannot be used to select either the W5100 or the SD card, but it must be kept as an output or the SPI interface won't work.

Note that because the W5100 and SD card share the SPI bus, **only one can be active** at a time. If you are using both peripherals in your program, this should be taken care of by the corresponding libraries. If you are not using one of the peripherals in your program, however, you will need to explicitly deselect it. To do this with the SD card, set Pin 4 as an output and write a high to it. For the W5100, set digital pin 10 as a high output.

The shield provides a standard RJ45 Ethernet jack. The reset button on the shield resets both the W5100 and the Arduino board.

The shield contains a number of informational LEDs:

PWR: indicates that the board and shield are powered

LINK: indicates the presence of a network link and flashes when the shield transmits or receives data

FULLD: indicates that the network connection is full duplex

100M: indicates the presence of a 100 Mb/s network connection (as opposed to 10 Mb/s)

RX: flashes when the shield receives data

TX: flashes when the shield sends data

COLL: flashes when network collisions are detected

The solder jumper marked "INT" can be connected to allow the Arduino board to receive interrupt-driven notification of events from the W5100, but this is not supported by the Ethernet library. The jumper connects the INT pin of the W5100 to digital pin 2 of the Arduino.

2.2 Software

2.2.1 Arduino IDE

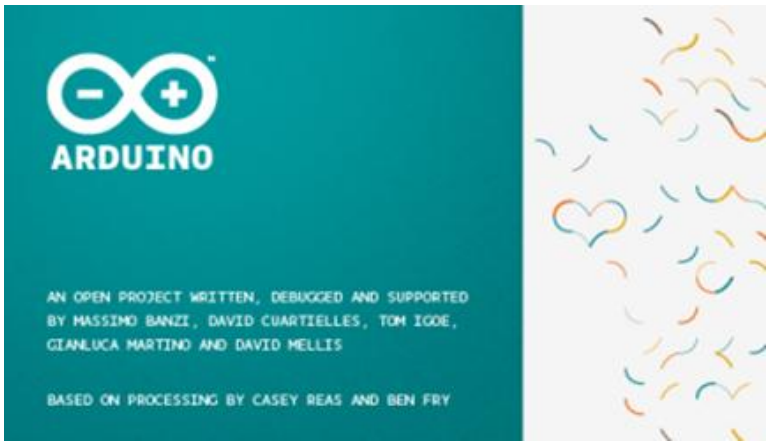


Figure 2 – 6: Loading picture of the Arduino IDE

In the previous section of this thesis, it was mentioned that the Arduino is a platform, based on easy-to-use hardware and software. So, as the hardware of the Arduino has already been considered, we proceed to investigate the software of the microcontroller.

According to Wikipedia, the Arduino IDE (Integrated Development Environment) [15] is described as a cross-platform [16] application written in Java, and is derived from the IDE for the (programming language) Processing and the Wiring projects. In simpler words, the Arduino IDE is a free to download program and it is used to program the Arduino to do design work or run an application. The IDE runs on Windows, mac OS and Linux platforms.

2.2.1.1 Installing the Arduino IDE on OpenSUSE v.12.3

There are a couple of ways to install the Arduino IDE on every version of openSUSE that is currently maintained. One of them, is by visiting the site of Arduino [17] and pressing the green button “Install Software via 1-click” as it is shown in Figure 2 – 7.

Figure 2 – 7: Install Software via 1-click button

Then, the installer will automatically choose packages of the same architecture (32 or 64 bit) as those used on your system. After installation, you will need to make every user a member of the groups: **"dialout", "lock"** and **"uucp"**.

To do this in YaST, select the **Security and Users** section, open the **User and Group Management** module and make the changes required here.

To do this from the command line, enter the following as root:

 # usermod -A dialout,lock,uucp <USER_NAME>

Then log out and log in again.

Now run arduino in your favourite terminal.

The other way to install Arduino IDE on openSUSE, is through the Terminal.

Enter the following commands as root from the command line, replacing the "<NN.N>" part of the URI with the version number required, eg. "12.3":

 # zypper ar -f
http://download.opensuse.org/repositories/CrossToolchain:/avr/openSUSE_<NN.N>
'CrossToolchain:avr'

 # zypper ref

 # zypper in arduino

Still as root, add the users to the required groups by entering the following:

 # usermod -A dialout,lock,uucp <USER_NAME>

Then log out and log in again.

Now run arduino in the terminal.

The Arduino package from arduino.cc works well too. Make sure the packages avrdude, rxtx-java, avr-libc, (cross-)avr-binutils and (cross-)avr-gcc are also installed. For further instructions the official site of Arduino provides usefull information.

2.2.1.2 Environment of the Arduino IDE

The Arduino development environment contains a text editor, a message area (the black area at the bottom of the Arduino environment), a menu and a few buttons (Toolbar) (see Figure 2 – 8).

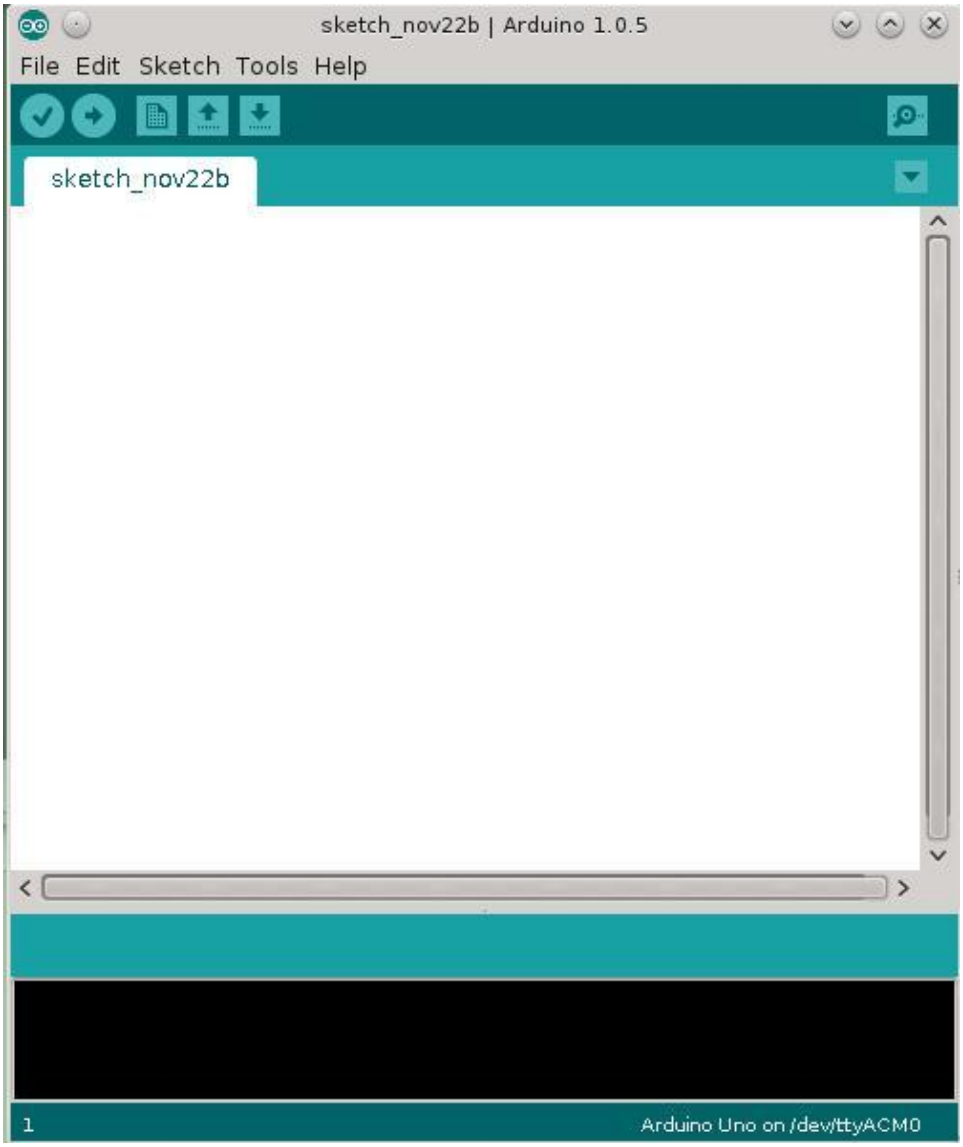


Figure 2 – 8: Arduino IDE

The Arduino IDE can be downloaded by the official site of Arduino [18]. The latest stable release is the 1.0.5 and it runs on Windows, mac OS, and the Linux operating system. In this site, older releases of the Arduino IDE can also be found, as well as releases which support the newer members of the Arduino family, the Arduino YUN, the Arduino Due and the Arduino Intel Galileo, available in 32bit and 64bit versions.

When the Arduino program is running the interface looks as shown in Figure 2 – 8. Its title has the name sketch as the programs under development are by default named. Programs are saved with the extension .ino; in some versions of the Arduino IDE (v.1.0), it is possible to open a .pde file format but the user is prompted to save as .ino file format on save. The message area gives feedback while saving and exporting and also displays errors. The console displays text output by the Arduino environment including complete error messages and other information. At the right bottom corner of the application the type of Arduino used and the serial port (through which the board is connected to the computer) are shown.

Regarding the toolbar of the Arduino, the following buttons exist:

 It checks if the code has any error.

 It compiles and uploads the program to the Arduino.

 It creates a new sketch.

 It opens an existed sketch.

 It saves the current sketch.

 It opens the Serial Monitor that displays serial data being sent from the Arduino board. This is shown in Figure 2 – 9.

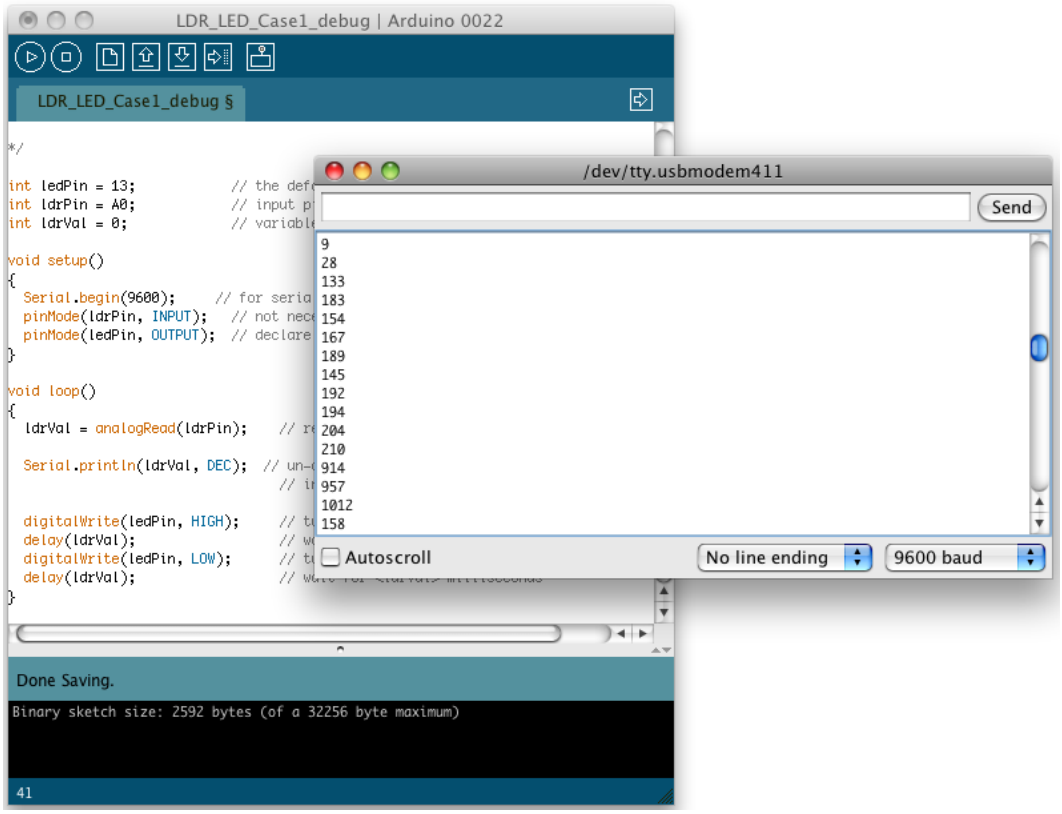


Figure 2 – 9: The Serial Monitor of the Arduino IDE.

In order to communicate with the board we can type text and press enter or the send button on the serial monitor as it is shown in the picture. The Baud rate from the drop-down matches the rate passed to **Serial.begin** in the sketch. Note that on Mac or Linux, the Arduino board resets (reruns the sketch from the beginning) when it connects with the serial monitor.

The menu of Arduino environment has different commands and options:

- **Edit**
  - *Copy for Forum*

It copies the code of your sketch to the clipboard in a form suitable for posting to the forum, complete with syntax colouring.

- *Copy as HTML*

  It copies the code of your sketch to the clipboard as HTML, suitable for embedding in web pages.

- **Sketch**
  - *Verify/Compile*

    It checks your sketch for errors.

  - *Show Sketch Folder*

    It opens the current sketch folder.

  - *Add File...*

    Adds a source file to the sketch (it will be copied from its current location). The new file appears in a new tab in the sketch window. Files can be removed from the sketch using the tab menu.

  - *Import Library*

    Adds a library to your sketch by inserting #include statements at the start of your code.

- **Tools**
  - *Auto Format*

    This formats your code nicely: i.e. indents it so that opening and closing curly braces line up, while statements inside curly braces are indented.

  - *Archive Sketch*

    It archives a copy of the current sketch in .zip format. The archive is placed in the same directory as the sketch.

  - *Board*

    Select the board that you're using.

  - *Serial Port*

    This menu contains all serial devices (real or virtual) on your machine. It should automatically refresh every time you open the top-level tools menu.

  - *Programmer*

    It is useful for selecting a hardware programmer when programming a board or chip and not using the on-board USB-serial connection. It is used in the case of burning a bootloader to a new microcontroller.

- *Burn Bootloader*

  The items in this menu allow you to burn a bootloader onto the microcontroller on an Arduino board.

The Arduino platform has many, ready to use libraries as well as examples which help the user get an early start in designing an application according to his needs. To import a library to a sketch, we select **Sketch→ Import Library…** as it is shown in the Figure 2 – 10.

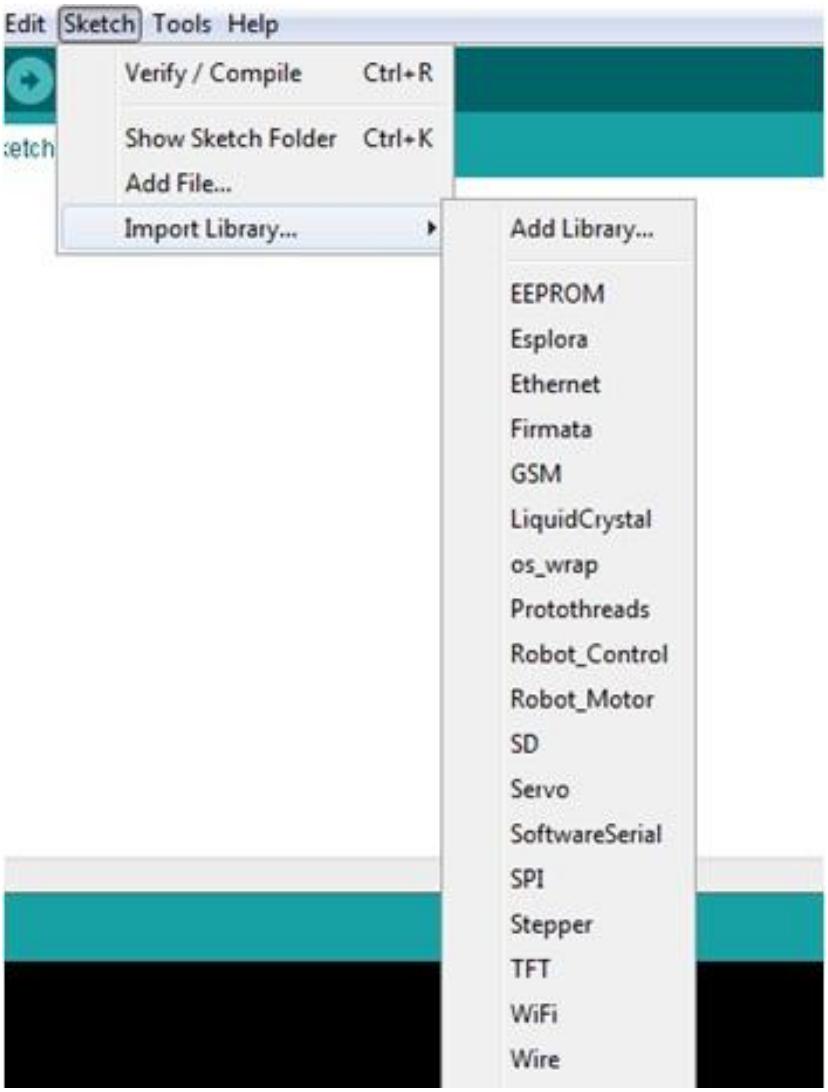


Figure 2 – 10: The Libraries menu of the Arduino

Also, any user who has created his own library and wishes to add it into Arduino environment, he can easily add it by clicking the "**Add Library…**" option and use it immediately.

The examples which accompany the Arduino IDE and relate to different application domains are easy to use by choosing **File →Examples**.

2.2.2 Threads

2.2.2.1 Processor Threads

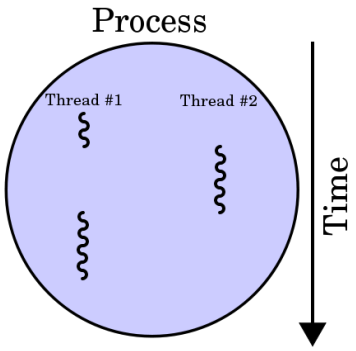


Figure 2 – 11: Processor Threads

According to Wikipedia, in computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. The scheduler itself is a light-weight process.

A typical UNIX process is thought to work as a thread as it is doing one thing at a time. Adding more threads in a process has as a result increasing the concurrency of the code. This means that the user-level threads:

1. Can manage in a better way code which deals with asynchronous events, handling the asynchronous events in parallel.
2. Can easily share system resources automatically, unlike processes which need complex mechanisms like pipes, shared memory, FIFO etc. to do that.
3. Can share the load of the problem between them, in order to achieve better performance.
4. Can greatly reduce program response time, since each thread undertakes to complete each task of the program independently.

Figure 2 – 12 illustrates the way in which the threads work within the process.

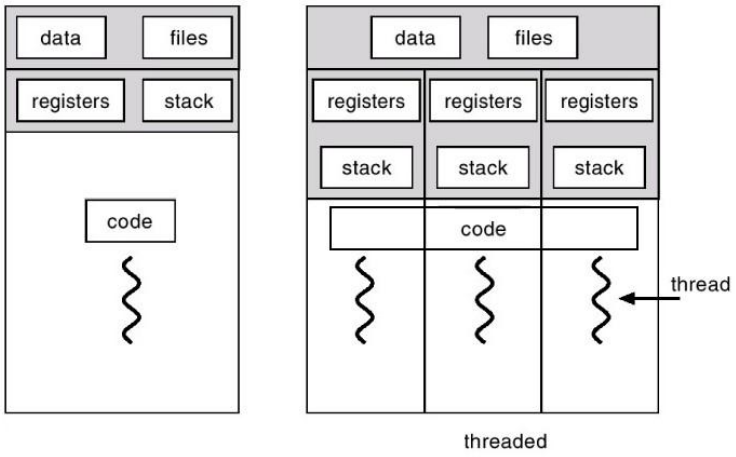


Figure 2 – 12: Threads vs. Processes

Sometimes, a multithreading program is confused with a multicore system, as its benefits are obvious even in a single-core system. Furthermore, when some threads in a program have been blocked, others can run unhindered, providing results to the user.

The information which a thread needs to have in order to run correctly in a program is its **ID,** which has significance within the context of the process to which belongs, a set of **register values**, a **stack**, its **scheduling information**, a **signal mask**, an **errno variable** which gives additional information about errors which have occurred in the UNIX system and **thread-specific data**. Everything else within a process is sharable, including the code text of the thread, the memory, the data etc.

2.2.2.2 Linux Threads

The threads are divided into the user-level threads and the kernel-level threads.

The kernel-level threads (or lightweight processes) are created and scheduled by the kernel. So, the kernel knows and manages the threads, by using a thread table that keeps track of all the threads of the system. The advantages of the kernel-level threads are:

1. Depending of the number of threads that exist in a process, the scheduler may give more time to a process which has an enormous number of threads to execute, instead of another which does not has many threads, as the kernel has full knowledge of all threads of the OS.
2. Kernel-level threads are especially good for applications that frequently block.

The disadvantages are:

1. They are extremely slow and inefficient. For instance, thread operations are hundreds of times slower than that of user-level threads.
2. Since kernel must manage and schedule threads as well as processes, it requires a full thread control block (TCB) for each thread to maintain information about threads. As a result there is a significant overhead and increased in kernel complexity.

The Linux threads belong to kernel-level threads. Previously it was mentioned that Linux-level threads are also named as lightweight processes. This means that differences between Linux threads and other threads are mostly differences between the processes and the threads, which are:

1. Processes are not sharing their resources while threads do.
2. Since there is no sharing of the same resources to the processes, it is quite difficult for the communication between them to be achieved, while the threads which are being executed within the same process, can easily communicate with each other as they have their resources in common e.g. memory etc.
3. As the processes are being executed independently of each other, their synchronization is taken care by kernel functions, while the thread's synchronization is being carried out by the process in which they are executed.
4. Context switching between threads is fast as compared to context switching between processes.

The Linux threads belong to kernel-level threads category. They can be created by using specific system calls.

2.2.2.2.1 Fork ()

On UNIX systems, the fork() process [19] is a procedure in which a process creates a copy of itself in order to run other programs etc. The new generated process is called child-process and the original parent-process. The child-process, calls the exec system call; it ceases execution of its former program in favour of the other.

The child-process is an exact duplicate of the parent-process except from the following points, which are specified in POSIX:

1. The child has its own unique process ID, and this PID does not match the ID of any existing process group.
2. The child's parent process ID is the same as the parent's process ID.
3. The child does not inherit its parent's memory locks.
4. Process resource utilizations and CPU time counters are reset to zero in the child.
5. The child's set of pending signals is initially empty.
6. The child does not inherit semaphore adjustments from its parent.
7. The child does not inherit record locks from its parent.
8. The child does not inherit timers from its parent.
9. The child does not inherit outstanding asynchronous I/O operations from its parent, nor does it inherit any asynchronous I/O contexts from its parent.

Also parent and child differ in the following Linux-specific process attributes:

1. The child does not inherit directory change notifications from its parent.
2. The child does not receive a signal when the parent terminates.
3. The default timer slack value is set to the parent's current timer slack value.
4. Memory mappings are not inherited across a fork().
5. The port access permission bits are not inherited by the child; the child must turn on any bits that it requires.

2.2.2.2.2 Clone ()

The clone() function [20] creates processes in similar way to the fork() function, with the difference that the clone() function allows sharing of memory, file descriptor tables and signal handlers tables.

Also one more difference between them, is that the child-process, which created by calling the fork() function continues to execute from the point of which the fork() function called. This is in contrast with the clone() called child-process, where the call of a function which is defined by the fn(arg) pointer, is done at the start of the execution of the child-process.

The child-process terminates when the function returns an integer. Also the child might terminate when a fatal signal arrives of by the exit() command.

Any further information can be found by visiting the open Linux manuals link: [21].

2.2.2.3 Posix Threads (Pthreads)

Historically, the programmers used their own type of threads in order to create a parallel program. This resulted in portability issues when developing parallel applications. This reason was the cause which led to the need of the creation of the POSIX standard. POSIX, is an abbreviation for "Portable Operating System Interface" and it has determined by the standard of IEEE POSIX 1003.1c (1995).

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library [22] [23].

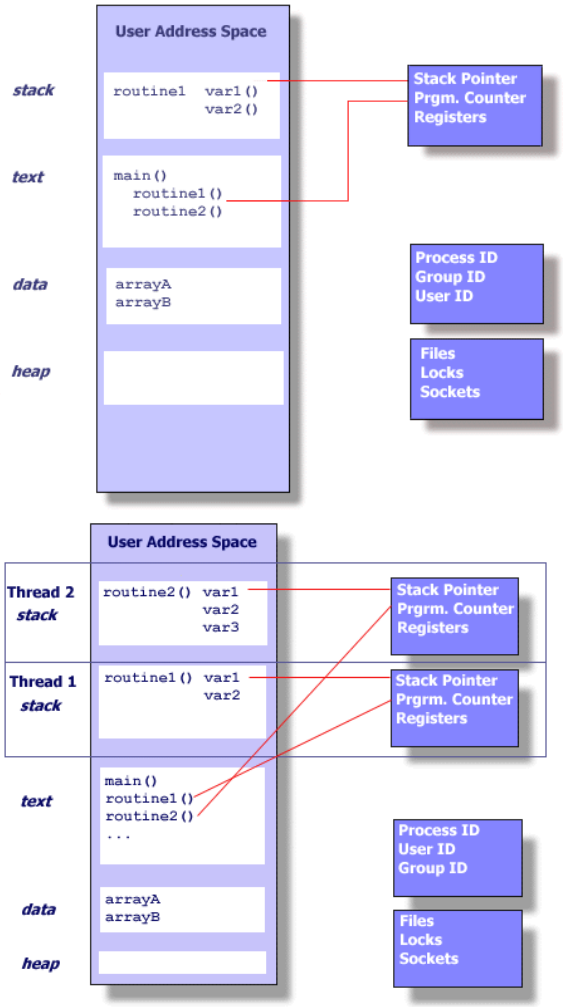


Figure 2 – 13: UNIX Process vs. Threads within a UNIX Process

The advantages of using Pthreads instead of processes are remarkable. They can provide to the user of an application high performance by achieving faster execution. In other words, performance can be achieved in the following ways.

1. By overlapping CPU processing with I/O or communication. While a program is waiting for an input for a long time, a thread can run a part of the process in which it is within, at the same time for time saving.
2. Using priority/real-time scheduling. Higher execution priority for the threads which are more important than others in the application.
3. Via synchronous event handling.

Also, when compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

The primary motivation for considering the use of Pthreads on SMP architecture is to achieve optimum performance. In particular, if an application is using MPI for point-to-point communications, there is a potential that performance could be greatly improved by using Pthreads.

Pthreads can also be used for serial applications, to emulate parallel execution and/or take advantage of spare cycles.

There are some considerations which need to be made by an application developer in order to design a parallel program.

1. Type of parallel programming model
2. Problem partitioning
3. Load balancing
4. Cache coherence
5. Communications
6. Data dependencies
7. Synchronization and race conditions
8. Memory issues
9. I/O issues
10. Program complexity
11. Programmer effort/costs/time etc.

But also a program should have the following characteristics to be suited for pthreads:

1. Work that can be executed, or data that can be operated on, by multiple tasks simultaneously
2. Blocking for potentially long I/O waits
3. Use many CPU cycles in some places but not others
4. Must respond to asynchronous events
5. Some work is more important than other work (Priority interrupts)

Depending on the characteristics of the developing application, there are different patterns for thread programming:

1. Manager and Worker model: The manager thread assigns works to worker threads and coordinates them.
2. Pipeline model: A program is broken into a series of tasks, each of which is handled in series but concurrently, by a different thread.
3. Peer model: Similar to Manager/Worker model but after the main thread creates other threads, it participates in the work.

The threads need to interact with each other. So, specifying the way they do that is the definition of Threads API (Application Programming Interface). There are four groups of routines which comprise the pthreads API:

1. Thread management

2. Mutexes
3. Condition variables
4. Other synchronization

The thread management routines work directly on threads. This means that threads can join, create, or detach etc. without the use of others conditions or mutexes.

Mutexes are an abbreviation for ‘’mutual exclusion’’ and they are functions used for thread synchronization. In computer programming, a mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

As for condition variables, these are subroutines which address communications between the threads that share a mutex. The conditions involve wait and signal actions as specified by the programmer. The synchronization routines manage read/write locks and barriers.

Some of the objects and function of the pthreads API are shown in the Table 2.

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Table 2: Definitions for the pthreads API

The pthreads API contains more than 100 subroutines. Always the pthread.h header file should be included in the program.

2.2.2.3.1 Thread Management

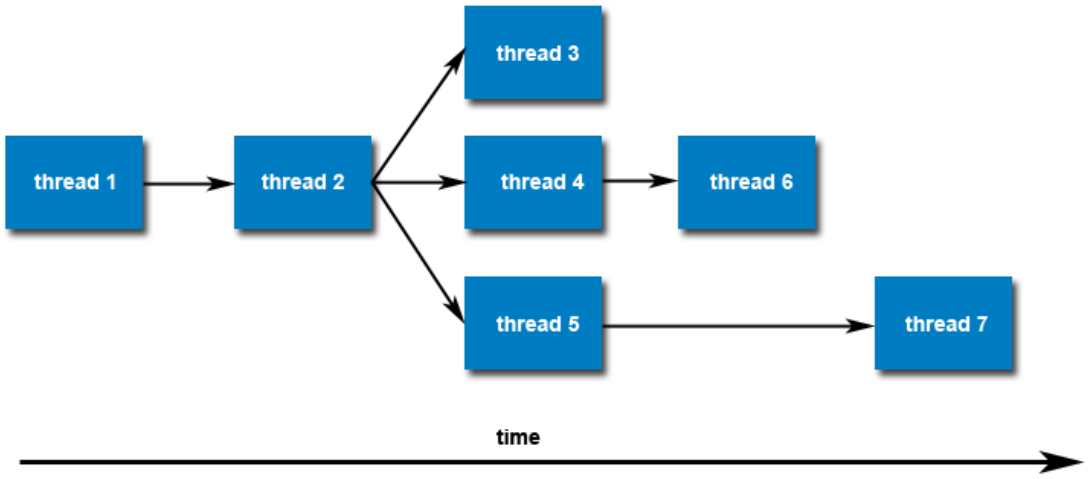


Figure 2 – 14: Threads

The main program of an application is considered separately and threads must be explicitly created by the programmer. So the routine `pthread_create` creates a new thread and makes it executable. After that, the newly created threads are peers and may create other threads with no dependency between them.

The threads are created with certain attributes. But some of them can be changed by the programmer. The routine `pthread_attr_init` initializes the thread attribute object and the `pthread_attr_destroy` destroys it. However, there are other important attributes of the thread that can be initialized. Some of them are:

- Detached or joinable state
- Scheduling inheritance. If we decide to use scheduling, we don't need to individually set the scheduling attributes of each thread we create. Instead, we can specify that each thread should inherit its scheduling characteristics from the thread that created it.
- Scheduling policy. A thread's scheduling policy is a way of expressing how threads of the same priority run and share the available CPUs.
- Scheduling parameters. This sets the priority of a thread to be scheduled. In more details, the priority is an integer value. The higher the value the higher a thread's priority for scheduling [24].
- Scheduling contention scope. Contention scope is the POSIX term for describing bound and unbound threads. A bound thread is said to have system contention scope i.e., it contends with all threads in the system. An unbound thread has process contention scope i.e., it contends with threads in the same process.
- Stack size
- Stack address
- Stack guard (overflow) size

The definition of Thread Management is about:

1. Creating Threads.

While the main program it is considered to be as a single thread, all the other threads need to be created by the programmer. By using the command `pthread_create`, a new thread is created and it becomes executable. The maximum number of the created threads is implementation dependent.

Concerning the scheduling of pthreads, if there is no scheduling mechanism, it is up to OS or/and to the implementation to decide which threads will execute first and when.

2. Terminating Threads
   The following list, describes several ways of termination of pthreads:

   a. When the thread completes its job.
   b. When the `pthread_exit` subroutine is called. This routine, allows the programmer to specify his own termination parameters. Also, any file which has opened inside the thread will remain open after the thread terminates.
   c. The thread cancelled by another thread via the `pthread_cancel` routine.
   d. The entire process is terminated due to making a call to either `exec()` or `exit()`.
   e. The main program finishes first.

3. Passing arguments to Threads
   One argument can pass arguments to the thread when the `pthread_create` routine is called. But sometimes it is necessary to pass more arguments in it. This can be achieved by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread_create` routine. But, in order to pass it safely to the thread, the passed data must be thread safe, i.e. it cannot be changed directly by other threads.

4. Joining Threads

```
pthread_join (threadid,status)

pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)
```

Figure 2 – 15: Joining and Detaching Routines

One way to accomplish synchronization between threads is the ‘’joining’’ routines (Figure 2 – 15), because the `pthread_join` subroutine blocks the calling thread until the called thread is terminated.

A thread can be joinable only if it is created as joinable.

5. Detaching Threads

Calling the `pthread_detach` subroutine, can be used to explicitly detach a thread even though it was created as joinable.

Depending on the implementation, a thread must be explicitly considered as joinable if it must be joinable or as detached if some system resources must be freed by that thread.

6. Stack Management

```
pthread_attr_getstacksize (attr, stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr, stackaddr)
pthread_attr_setstackaddr (attr, stackaddr)
```

Figure 2 – 16: Stack Management Routines

The POSIX standard does not dictate the size of a thread's stack. This is implementation dependent and varies.

7. Miscellaneous Routines

```
pthread_self ()
pthread_equal (thread1,thread2)
```

Figure 2 – 17: Miscellaneous Routines

The pthread_self returns the unique, system assigned thread ID of the calling thread.

The pthread_equal compares two thread IDs.

2.2.2.4 Protothreads

There are some lightweight threading approaches with limited stacks such as TOSThreads under TinyOS (similarly for MANTIS) [25], and TinyThreads under Linux [26]. These approaches enable applications to be developed using multithreading, where each thread has its own stack to store its context. This approach, while greatly improving the expressivity of programs, is not too attractive when considering memory efficiency.

Protothreads provide a better alternative in terms of memory efficiency; each thread only requires 2 bytes of memory. However, protothreads are not as expressive as regular threads.

Protothreads is a programming model invented by Adam Dunkels [27] of the Swedish University of Computer Science [28]. They are extremely lightweight stackless type of threads, designed for severely memory constrained embedded systems.

The comparison of Protothreads with fast event-driven implementations can show that the programs which are written with an event-driven model typically have to be implemented as

explicit state machines, in contrast with the Protothreads based programs, which can be written in a sequential fashion without having to design explicit state machines.

Comparing the Protothreads with other threads, their main advantage is that Protothreads are very lightweight, as they do not require a stack, which makes them proper for low memory embedded systems. Another point that they differ, is that the Protothreads runs only within a single C function and cannot span over others. In other words, a protothread may call other functions but cannot block inside a called function. So, unlike threads, Protothreads makes blocking explicit. Table 3 summarizes the features of Protothreads and compares them with the features of discrete events and threads based on Adams Dunkels work [29].

Feature	Events	Threads	**Proto-threads**
Control structures	No	Yes	**Yes**
Debug stack retained	No	Yes	**Yes**
Implicit locking	Yes	No	**Yes**
Preemption	No	Yes	**No**
Automatic variables	No	Yes	**No**

Table 3: Qualitative comparison between events, threads and Protothreads [30]

1. **Control Structure**
   A control structure is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. In event-driven models, control structures must be broken in smaller pieces in order to implement continuation structures in contrast to protothreads which allow blocking statements to be used together with control structures.
2. **Debug stack retained**
   Because the manual stack management and the free flow of control in the event-driven model, debugging is difficult as the sequence of calls is not saved on the stack. With both threads and Protothreads, the full call stack is available for debugging.
3. **Implicit locking**
   All the yield points are immediately visible in event-driven models and in protothreads with the use of `pt_wait`, instead of threads in which it is not always evident that a particular function call yields.
4. **Pre-emption**
   The semantics of the threaded model allows for pre-emption of a running thread: the thread's stack is saved, and execution of another thread can be continued. Because both the event-driven model and protothreads use a single stack, pre-emption is not possible within either of these models.
5. **Automatic Variables**
   Automatic Variables are allocated and deallocated automatically when program flow enters and leaves the variable's context. Local variables, are usually synonymous with automatic variables, but local is more general.
   With threaded model, the automatic variables are retained even if the thread blocks as it uses a single shared stack for all programs and it rewinds it every time a program blocks. In contrast, with protothreads, automatic variables are not saved across a blocking wait.

2.2.2.4.1 Limitations of the Protothreads

Although protothreads have many advantages which are the same as the multithreaded programing model, they impose some limitations of the event-driven model. As it was foretold, such a limitation is that they do not retain automatic variables while the protothread blocks. Therefore the automatic variables must explicitly be saved somewhere before protothreads go into standby state. However, some of the compilers of C like gcc provide the user with warnings for automatic variables which is might not be saved after the execution of the program.

The programmer can use the state objects method similar to the event-driven model in order to save the state of an automatic variable. State objects are memory locations in which the automatic variables are saved.

2.2.2.4.2 Implementation of Protothreads

Protothreads are based on a low-level mechanism: the *local continuation mechanism*, which is implemented in a variety of ways. One possible way is based on the C switch statement.

A local continuation has two operations; *set* or *resume.* At the set operation, the condition of the function is captured, except from its stack. At the resume operation, the condition of the function is reset to what it was when the local continuation was set.

Implementation of protothreads imposes a restriction not to use the C switch statement in programs using protothreads. If they are used together, in some cases the C compiler will detect an error but in the most cases the error will not be traced by the compiler which makes it hard to debug. Executing the program which has an untraced error, will result an erroneous mixture of one particular implementation of protothreads and switch statements.

Protothreads are based on C macros rather than C functions, because protothreads are altering the flow of control. This is typically difficult to do with C functions since such an implementation would require low-level assembly code to work.

As Adam Dunkels, who is the Protothreads creator, recommends, we will explain the use of protothreads and local continuations with the following simple example [31]:

```
1  #include "pt.h"
2  static int counter;
3  static struct pt example_pt;
4  static
5  PT_THREAD(example(struct pt *pt))
6  {
7    PT_BEGIN(pt);
8    while(1) {
9      PT_WAIT_UNTIL(pt, counter == 1000);
10     printf("Threshold reached\n");
11     counter = 0;
12   }
13  PT_END(pt);
```

```
14  }
15  int main(void)
16  {
17    counter = 0;
18    PT_INIT(&example_pt);
19    while(1) {
20      example(&example_pt);
21      counter++;
22    }
23    return 0;
24  }
```

This program, waits for a counter to reach a certain threshold, prints out a message, and resets the counter. This is done in a `while()` loop that runs forever. The counter is increased in the `main()` function.

It is important to explain the protothreads macros. The following definition (originated from Adam Dunkels) is a combined version of the protothreads macros and the local continuation macros implemented with the C switch statement.

```
struct pt                              { unsigned short lc; };
#define PT_THREAD(name_args)           char name_args;
#define PT_BEGIN(pt)                   switch(pt->lc) {case 0:
#define PT_WAIT_UNTIL(pt, c)           pt->lc=__LINE__; \
                                       case __LINE__:   \
                                       If(!(c)) return 0
#define PT_END(pt)                     }pt->lc = 0; return 2
#define PT_INIT(pt)                    pt->lc = 0
```

We see that the `struct pt` consists of a single unsigned short (2 bytes) called `lc`, short for local continuation. This unsigned short variable is the source of the "two byte overhead". Furthermore, we see that the `PT_THREAD` macro simply puts a char before its argument. Also, we note how the `PT_BEGIN` and `PT_END` macros open and close a C switch statement, respectively. The `PT_WAIT_UNTIL` macro is the most complex one. It contains one assignment, one case statement, one if statement and a return statement. Also, it uses the built-in `__LINE__` macro twice. The **`__LINE__` macro** is a special macro that the C preprocessor will expand to the **line number at which the macro is issued**. Finally, the `PT_INIT` macro simply initializes the lc variable to zero.

Many of the statements used in the protothread macros are not commonly used in macros. The return statement used in the `PT_WAIT_UNTIL` macro breaks the flow of control in the function the macro is used. The `PT_BEGIN` macro opens a switch statement, but does not close it. The `PT_END` macro closes a compound statement that it has not itself opened. These things do look weird when looked at without the perspective of protothreads. However, in the context of protothreads these things are absolutely essential for correct operation of protothreads: the macros essentially change the flow of control in the C function in which they are used. This is indeed the whole essence of protothreads.

In order to understand how protothreads really work, we will see how the protothread in the example above looks when expanded by the C preprocessor:

The original protothread of the above program:

```
4  static
5  PT_THREAD(example(struct pt *pt))
6  {
7    PT_BEGIN(pt);
8    while(1) {
9      PT_WAIT_UNTIL(pt, counter == 1000);
10     printf("Threshold reached\n");
11     counter = 0;
12   }
13   PT_END(pt);
14 }
```

is now expanded by the C preprocessor:

```
4  static
5  char example(struct pt *pt)
6  {
7    switch(pt->lc) { case 0:
8    while(1) {
9      pt->lc = 9; case 9: if(!(counter == 1000)) return 0;
10     printf("Threshold reached\n");
11     counter = 0;
12   }
13   } pt->lc = 0; return 2;
14 }
```

At the first line of the code we see how the `PT_THREAD` macro has expanded so that the `example()` protothread has been turned into a regular C function that returns a char. The return value of the protothread function can be used to determine if the protothread is blocked waiting for something to happen or if the protothread has exited or ended.

The `PT_BEGIN` macro has expanded to a PT(switch) statement with an open brace. If we look down to the end of the function we see that the expansion of the `PT_END` macro contains the closing brace for the switch. After the opening brace, we see how the `PT_BEGIN` expansion contains a `case 0:` statement. This is to ensure that the code after the `PT_BEGIN` statement is the first to be executed the first time the protothread is run. Recall that `PT_INIT` set `pt->lc` to zero.

In the `while(1)` line, we see that the `PT_WAIT_UNTIL` macro has been expanded into something that contains the **number 9**. The `pt->lc` variable is set to 9, and a `case 9:` statement follows right after the assignment. After this, the counter variable is checked to see if it has reached 1000 or not. If not, the `example()` function now executed an explicit return.

The next time the `example()` function is called from the main() function, the `pt->lc` variable will not be zero but 9, as it was set in the expansion of the PT_WAIT_UNTIL macro. This makes the `switch(pt->lc)` jump to the `case 9:` statement. This statement is just before the `if` statement where counter variable is checked to see if it has reached 1000. So the counter variable is checked again. If it has not reached 1000, the `example()` function returns again. The next time the function is invoked, the switch jumps to the `case 9:` again and re-evaluates the `counter == 1000` statement. It will continue to do so until the counter variable reaches 1000. Then, the `printf` statement is executed and the counter variable is set to zero, before the `while(1)` loop is executed again.

The number 9 finally, is the line number of the `PT_WAIT_UNTIL` statement. The nice thing with line numbers are that they are monotonically increasing. In simpler words, if we put another `PT_WAIT_UNTIL` statement later in our program, the line number will be different from the first `PT_WAIT_UNTIL` statement. Therefore, the `switch(pt->lc)` knows exactly where to jump.

We can notice something strange with the code above. The `switch(pt->lc)` statement jumped right into the `while(1)` loop. The `case 9:` statement was inside the `while(1)` loop. This feature of the C programming language was probably first found by Tom Duff in his wonderful programming trick named **Duff's Device**. The same trick has also been used by Simon Tatham to implement **coroutines** in C.

Duff's Device is an optimized implementation of a serial copy that uses a technique widely applied in assembly language for loop unwinding. Its discovery is credited to Tom Duff in November 1983 [32]. Loop unwinding (or loop unrolling) is a loop transformation technique that attempts to optimise a program's execution speed at the expense of its binary size. Its goal is to increase a program's speed by reducing instructions that control the loop [33].

Coroutines are similar to threads in the sense of multithreading. They have a line of execution, with their own stack, local variables, and instruction pointer, however, sharing global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended [34].

Coroutines are computer program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes [35].

2.3 Visualization and Data Monitoring

It is very essential in industrial and experimental cases to setup, monitor and control some physical quantities, such as voltage and temperature. The efficient solution for this problem is to develop a data logger.

Earlier development of data logger was done through manual measurements from analogue instruments such as thermometers. Unfortunately this type of data logger cannot fulfil the current requirements in terms of time and accuracy. Since 1990 further development in data logging has taken place as people begin to create PC-based data logging systems. In a later stage of development it has been found that microcontrollers (integration of microprocessors and certain peripherals including memory on single chip) are more reliable as well as efficient. Use of microcontrollers in embedded design has not only increased but also has brought a revolutionary change. At the same time competition requires manufacturers to expand their product functionality and provide differentiation while maintaining or reducing the cost.

Monitoring and controlling physical parameters by embedded systems using microcontrollers are very effective in industrial and research-oriented requirements. The nature of monitoring information, e.g. temperature, pressure, humidity, and light, is ever-changing. The parameters may be exposed to stimuli from their operating environment. For example, temperature can be monitored through a variety of sensors; one should adhere to utmost care in selecting sensors due to different levels of complexity associated with the calibration process. If calibration is not implemented properly, output of the embedded system may vary from actual temperature values measured through standard instruments. Similarly for the case of light photoresistors (LDR), calibration in Lumens is difficult due to easy unavailability of Lux-meter. Hence in general, the ADC reference voltage can be related with some precaution to light intensity [36].

In this section, we will demonstrate a case study focusing on visualising and dynamically imaging a physical quantity, such as voltage. This parameter will be used an embedded system (Arduino Uno R3) and specific software packages such as Processing [37] and Grace [38]. The Processing programming language will be used to read data from the Serial Port of the computer connected to the Arduino board and store it into a .dat file, as long as the Arduino sends data. This .dat file will be used by Grace for reading and visualising data into a graph. We will operate the Grace program through specific linux commands (pipes) of the OpenSUSE 12.3 terminal. The collected data are the virtual triangular voltage (3 – 0 – 3 Volts) versus time (seconds).

A brief description of the Processing IDE and Grace software follows.

1. Processing IDE.
   According to the official site, Processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software technology within the visual arts. Initially created as a software sketchbook for teaching computer programming fundamentals within a visual context, Processing evolved into a development tool for professionals. Today, there are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production. Processing can easily be downloaded and installed on Linux, Mac OS X, and Windows operating systems [39].

   After the installation of Processing we notice that its environment (see Figure 2 – 18) is almost the same as the Arduino IDE. In fact, the Arduino IDE is based on Processing. Furthermore, the Processing Environment includes a text editor, a compiler, and a display window. It enables the creation of software within a carefully

designed set of constraints. Further information about Processing can be found on its official site.

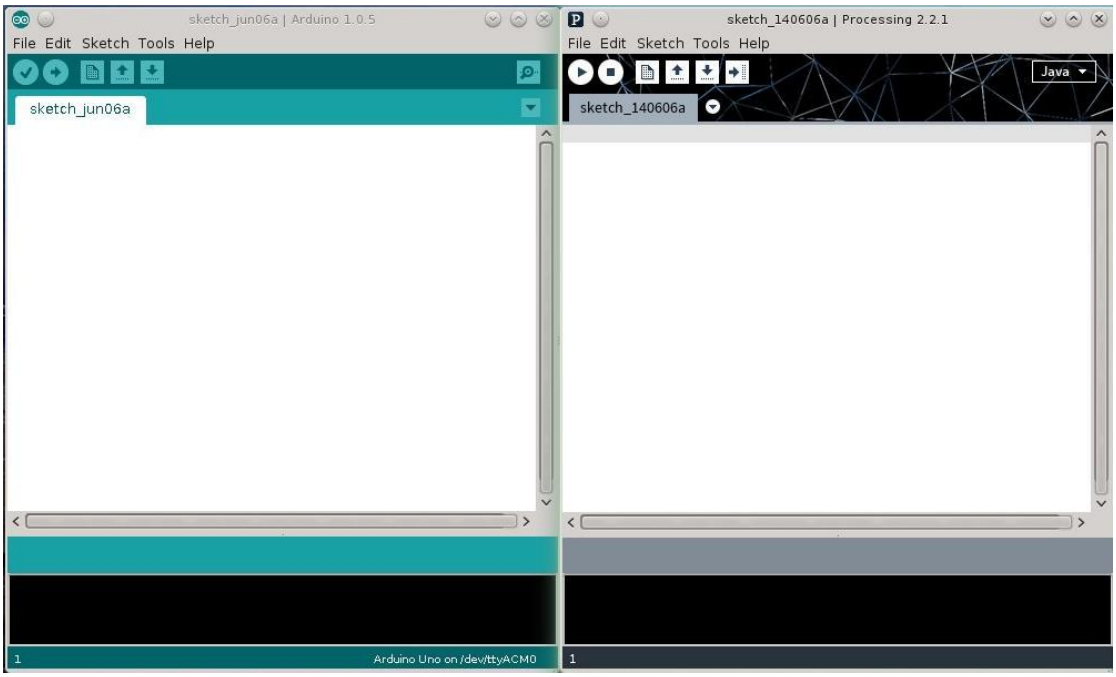


Figure 2 – 18: Comparison of the Arduino IDE and Processing PDE

2. Grace software (see Figure 2 – 19).
   Grace is a WYSIWYG [40] 2D plotting tool for the X Window System [41] and M*tif as described on Grace's official site. Grace runs on practically any version of Unix-like OS. Grace features at a glance:

   - General
     a. WYSIWYG design
     b. Convenient point-and-click graphical user interface
     c. Precise control of graph features
     d. True publication quality
     e. Instant plot refresh
   - Export Options
     a. Exports vector graphics to (E)PS, PDF, MIF, and SVG formats
     b. Supports cross-platform PNG, PNM, and JPEG formats
   - Graphing Flexibility
     a. Unlimited number of graphs
     b. Unlimited number of curves on a graph
     c. Up to 256 customizable colours
     d. 9 dashed line styles
     e. 32 fill patterns
     f. 10 built-in marker symbols; plus, any character glyph from any font can be used as a marker
     g. Colour/fill markers

h. Text annotations with subscripts, superscripts, mixed fonts, styles and colours and more complex typesetting

- Curve Fitting
    a. Linear and nonlinear least-squares
    b. Calculation and display of residuals
    c. Arbitrarily complex user-defined fitting functions, including dynamically loadable C/Fortran/... modules
    d. Fitting with constraints
    e. Region restrictions
    f. Fitting with arbitrary weight functions
- Analysis Capability
    a. FFT
    b. Integration and differentiation
    c. Histograms and statistics
    d. Splines, including Akima splines
    e. Interpolation and smoothing
    f. Convolution, correlation, and covariation
    g. Sorting
- Data Formats
    a. Unlimited data size; up to six dimensions plus an optional array of strings
    b. Reads text data input files
    c. Reads 1D netCDF files
- Programmability
    a. Built-in programming language
    b. Math functions manipulate entire array
    c. Variables, including arrays (1D)
    d. User-definable functions via loadable modules
    e. All aspects of plot outlook can be programmed
    f. Controllable by external programs

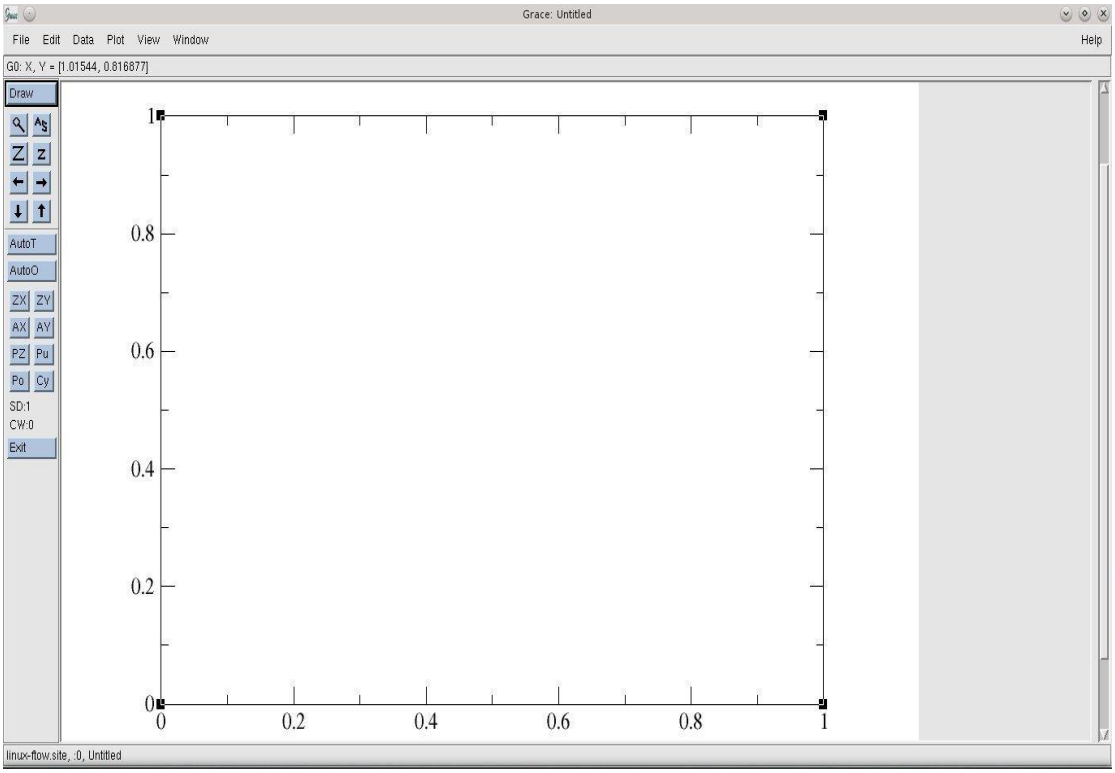


Figure 2 – 19: The Grace Environment

A user's guide can be found on the official site of Grace [42] along with other useful tutorials and documents.

2.3.1 Using Arduino, Processing and Grace together for Visualizing Data

The purpose of this section is to show practically how to achieve online monitoring and visualization of a physical quantity (voltage) based on collected data continuously stored in a file. In our experiments, we use the Arduino Uno R3 board as the hardware platform (embedded system), and the Arduino IDE, Processing PDE, Grace Plot tool and OpenSUSE 12.3 terminal as the software part,. We developed two programs, one for each software package, which we describe below.

1. The developed program for Arduino IDE.
   This program is developed to run on the Arduino board. It produces a virtual triangular voltage of 3 – 0 – 3 Volts versus time in seconds. It continuously sends voltage vs. time data to serial port of the computer in order to be used from the Processing PDE. The code of the Arduino IDE follows next with line numbers and comments wherever they needed.

```
1        #define MAX_NO_ITERATIONS 8
2        #define MAX_VOLTAGE_SCALE 32

3        float voltage;  //-->The virtual triangular voltage
4        int iterations, VL, j;
```

```
5       //Setting up the Serial port
6       void setup()
7       {
8         Serial.begin(9600); //-->Starts the serial port at 9600
9       }

10      //This function, creates the virtual triangular voltage
11      void triangVoltage()
12      {
13        for (iterations = 0; iterations < MAX_NO_ITERATIONS;
14        iterations++)
15              for (VL = 0; VL < MAX_VOLTAGE_SCALE; VL++)
16                                      //   waveform of 0-3-0 volts
17            {
18              if (iterations %2 != 0)
19              {
20                j = MAX_VOLTAGE_SCALE - VL;
21              }
22              else
23              {
24                j = VL;
25              }
26              //The next code sends the voltage and time data
27               from Arduino to Computer through Serial Port
28              voltage = 3 - j * (1023 / MAX_VOLTAGE_SCALE) *
29               (3.0 / 1023.0);
30              Serial.print(voltage);
31              Serial.print(" ");
32              int time = millis()/100;
33              Serial.println(time);
34              delay(100);
35            }
36      }
37      //The infinite loop
38      void loop()
39      {
40        triangVoltage ();
41      }
```

2. The developed Processing IDE program.
   This program, instead of the one which runs on the microcontroller, runs on the computer and its purpose is to read data from the computer's Serial port and save into a .dat file, which will be used from Grace. The code for Processing IDE follows next with line numbers and comments where needed.

```
1   import processing.serial.*;
2   import java.text.*;
3   import java.util.*;
4   PrintWriter output;
5   String fileName;
6   Serial myPort; // Create object from the serial port
7   short portIndex = 0; //Select the com port, 0 is the
8           //first port
9   void setup ()
10  {
11    size(200, 200); // set the window size:
12    println(Serial.list()); // List all the available
13                            //serial ports
14    //If Arduino is in the first port in the serial list,
15    open Serial.list()[0].
16    myPort = new Serial(this, Serial.list()[0], 9600);
17    // don't generate a serialEvent() unless you
18    //get a newline character
19    myPort.bufferUntil('\n');
20    background(50); // set initial background

21    fileName = "dataFile";
22    output = createWriter(fileName + ".dat");
23  }

24  void draw ()
25  {

26  }

27  void serialEvent (Serial myPort)
28  {
29    String inString = myPort.readStringUntil(' '); //this
30                                  //is the space character
31    String tmString = myPort.readStringUntil('\n'); //this
32                          //is the change line character
33    if (inString != null && tmString != null)
34    {
35      inString = trim(inString); // trim off any whitespace
36      tmString = trim(tmString);
37      float Voltage = float(inString); //Arduino Voltage to
38                                       //a float Voltage
39      int Time = int(tmString); // Arduino Time to int Time
40      output.print(Time);
41      output.print("  ");
42      output.println(Voltage);
43      output.flush();  // Writes data to the file
44    }
```

```
45        }
46        void keyPressed()
47        {
48          output.flush(); //Writes the remaining data to the file
49          output.close(); // Finishes the file
50          exit(); //Stops the program
51        }
```

The above two programs are saved as **triang_volt.ino** and **processing_triang.pde**, and when we make sure that they run correctly, i.e the Arduino program gives pairs of values (Voltage – Time in its serial monitor and the Processing stores them into a **.dat** file on the computer, we proceed to invoke Grace.

The terminal of the OpenSUSE 12.3 is one of the ways we can achieve this purpose. The Grace Plot tool has many commands used from terminal to control the visualization process [43]. If we assume that we save this experiment to **/home/User/proc_grace** folder, we can use the following commands in a terminal.

Next command opens Processing from its installation folder.

./processing /home/User/proc_grace/processing_triang/processing_triang.pde

Next, we change the current directory of the terminal to:

>cd /home/User/proc_grace/processing_triang

This is necessary in order to operate correctly the Grace program. Now in the terminal we should see the following:

User@linux-ftow: ~/proc_grace/processing_triang>

After that, we are going to use a named pipe [44] to invoke Grace from terminal.

> mkfifo /home/User/proc_grace/processing_triang/nameofpipe

Next, we open Grace and connect it with the pipe:

> xmgrace -npipe nameofpipe&

Next, we configure the graph by giving it a title and labels to its x, y axis:

> echo title \''Voltage vs. Time '' > nameofpipe
> echo xaxis label \''Time seconds\'' > nameofpipe
> echo yaxis label \''Voltage\'' > nameofpipe

After that, we create a while loop in terminal to help Grace to read the .dat file continuously and draw its data to the graph:

> while true
> do
> echo ''read \''dataFile.dat\'' '' > nameofpipe

> echo autoscale > nameofpipe
> echo redraw > nameofpipe
> done

Before we press enter key on terminal to run the while loop, we must run the Processing program for reading the Serial input from Arduino and start to store data on the .dat file. After that, we press the enter key on terminal.

The result we see on Grace looks like the Figure 2 – 20:

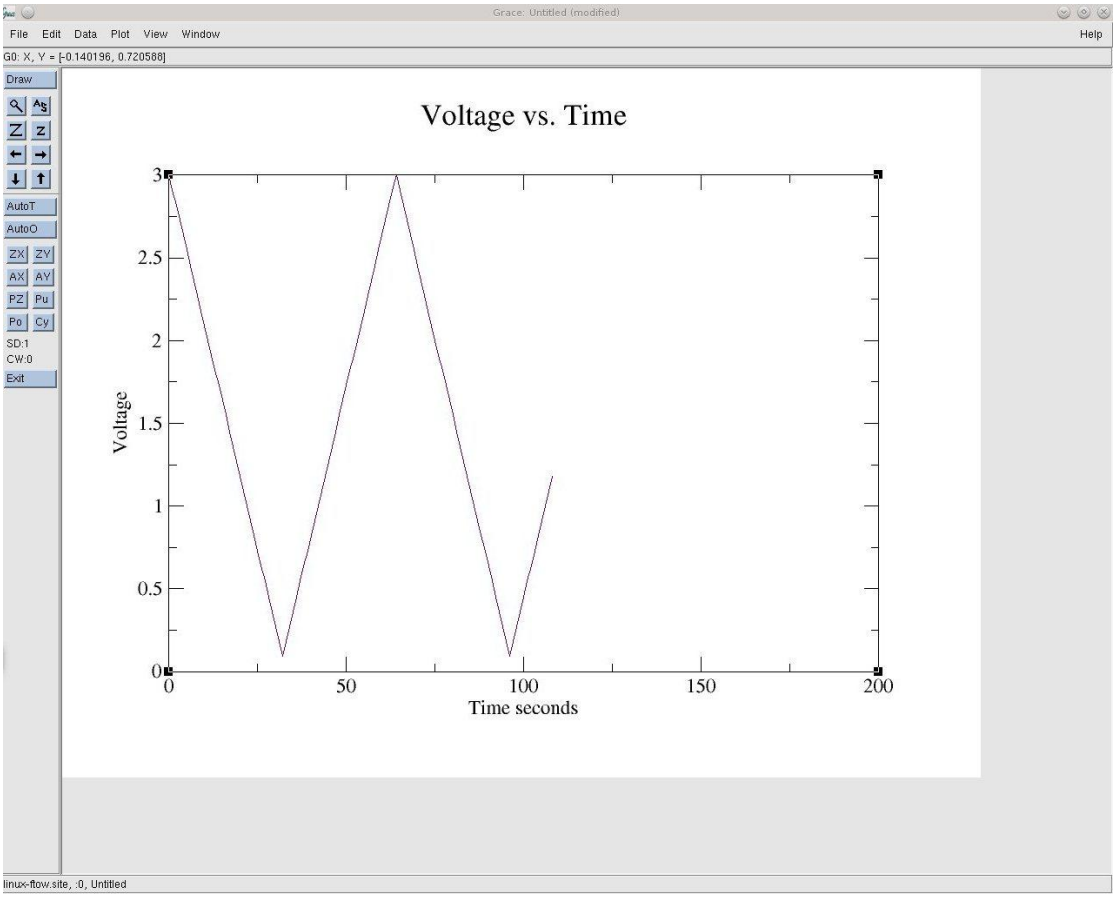


Figure 2 – 20: The result on Grace

We notice that the curve is updated continuously online, as long the Processing program runs and the Arduino board is connected on the Serial port.

In the future, we can expand this experiment to show more plots, one for each data series monitored on the Arduino board. We conclude that data monitoring is of high importance because we can monitor every change of the measured values visually. This way of monitoring is much easier than monitoring values without a graph, since online monitoring gives us the opportunity to know at any time what happens in our device and detect any erroneous behaviour.

3. Project Design

3.1 Energy Management

3.1.1. Definition of Energy Management

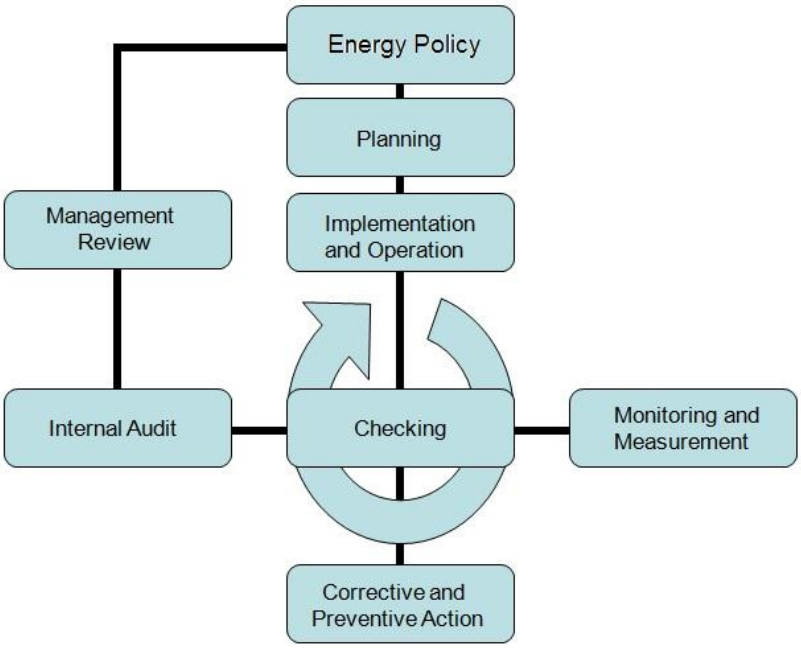


Figure 3 – 1: Energy Management Model

Energy Management has the effect of producing goods with minimal economic costs, as well as minimal negative environmental effects. In fact, according to Capehart, Turner and Kennedy Guide to Energy Management Fourth Edition [45] the Energy Management is defined as the judicious and effective use of energy to maximize profits – or to minimize costs – in order to enhance competitive positions. Another comprehensive definition of Energy Management, states "The strategy of adjusting and optimizing energy, using systems and procedures so as to reduce energy requirements per unit of output while holding constant or reducing total costs of producing the output from these systems" [46].

The economic cost is minimized by managing energy consumption as follows:

1. With increased energy efficiency and reduced energy use
2. By cultivating the interest in energy matters through dissemination of issues related to them
3. By developing better methods in monitoring, reporting and managing the energy
4. By facilitating research and investment in research projects, which searching ways for energy savings
5. By increasing the interest in energy issues through education

Proper energy management has many benefits, not only individually but socially as well. As it was foretold, the economical profits could be significantly increased, improving the quality of life, reducing poverty and creating new works for the unemployment. In particular one can achieve:

1. The increase of national wealth and security as the adequacy of oil rises
2. Growth of national economic competition

Regarding the environment, the energy management offers remarkable benefits for environmental protection. Specifically:

1. Reduction of the acid rain
2. Limitation of global climate change
3. Increase of the amount of ozone in the upper atmosphere.

3.1.2. Designing an Energy Management System

The design of an Energy Management System (EMS) is essential in order to achieve an effective energy saving. Designing an energy management system must follow the rules of Figure 3 – 1. Such a system is based on the following processes:

1. Measuring the energy consumption and collecting data. A detailed interval of energy consumption report provides more information about the current energy consumption plan in order to redesign it.
2. Developing an energy use profile. An energy use profile will demonstrate how energy use is distributed in an occasionally system.
3. Setting reduction targets. In order to create a better energy management system, it is needed to predetermine the conditions and the terms under which that system will work.
4. Developing strategic action plans for improvement.
5. Tracking, measuring and reporting the energy consumption.

3.2 Hardware

3.2.1 Hardware Development

Considering the theoretical part (Section one and two) of this thesis, which is related to the design and implementation of the Remote Control Power Strip device (RC Power Strip), implementation was divided into the following sub-stages:

1. Requirement analysis. The first stage was to set the problem which the RC Power Strip device has to solve. More specifically, the problem relates to *remote control of devices based on energy monitoring*.
2. Definition of system specifications and use-cases of the RC Power Strip device. In particular, we define its minimum requirements as well as other characteristics. The system which will be designed needs to have the following specifications:
    a. Remote control of devices, via internet or Local Area Network,
    b. Measuring AC current and AC voltage in order to protect the other devices from overvoltage and short circuits,
    c. Temperature and humidity measurement of the environment in which the RC Power Strip device will operate.
3. Modelling the system under design, experimenting with different algorithms and preliminary evaluation. In this step, the algorithm which will be used to perform the previously mentioned tasks was developed. Different kinds of programming models were used and a comparison between them was performed:

a. Serial Programming Model
b. Asynchronous Parallel Programming Model using Protothreads
c. Synchronous Parallel Programming Model using Protothreads

4. Selection of components and development tools. We have selected:
   a. An Arduino Uno R3 for the development of the algorithm used by the device
   b. An Arduino Mega 2560 R3 for the implementation of the device
   c. An Arduino Ethernet Shield for connecting the Arduino board into a local network or internet over the Ethernet
   d. An ARM MCU SD Card module
   e. A Songle one-relay module used as a (preliminary) experimental energy-related component
   f. A Songle four-relay module used for the implementation of the energy-related component
   g. A 1602 LCD hd44780 display module
   h. Several electronic components, such as resistors of various values, LEDs, potentiometers etc.
   i. Jump wires, 3 breadboards of 830 tie points.
5. Schematic designing of the device by using Fritzing [47].
6. Construction of the device
7. Test of the device

3.2.1.1 Arduino Mega 2560 R3

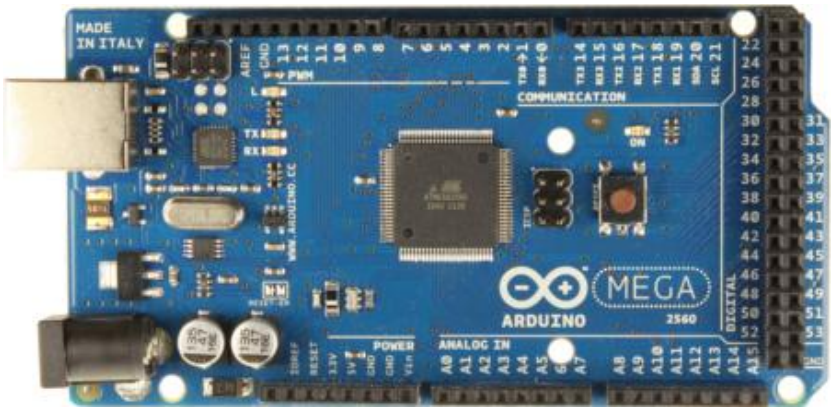


Figure 3 – 2: Arduino Mega R3

The Arduino Mega 2560 Revision 3 (see Figure 3 – 2), is a microcontroller board based on ATmega2560 microprocessor whose pin mapping is shown in Figure 3 – 3.

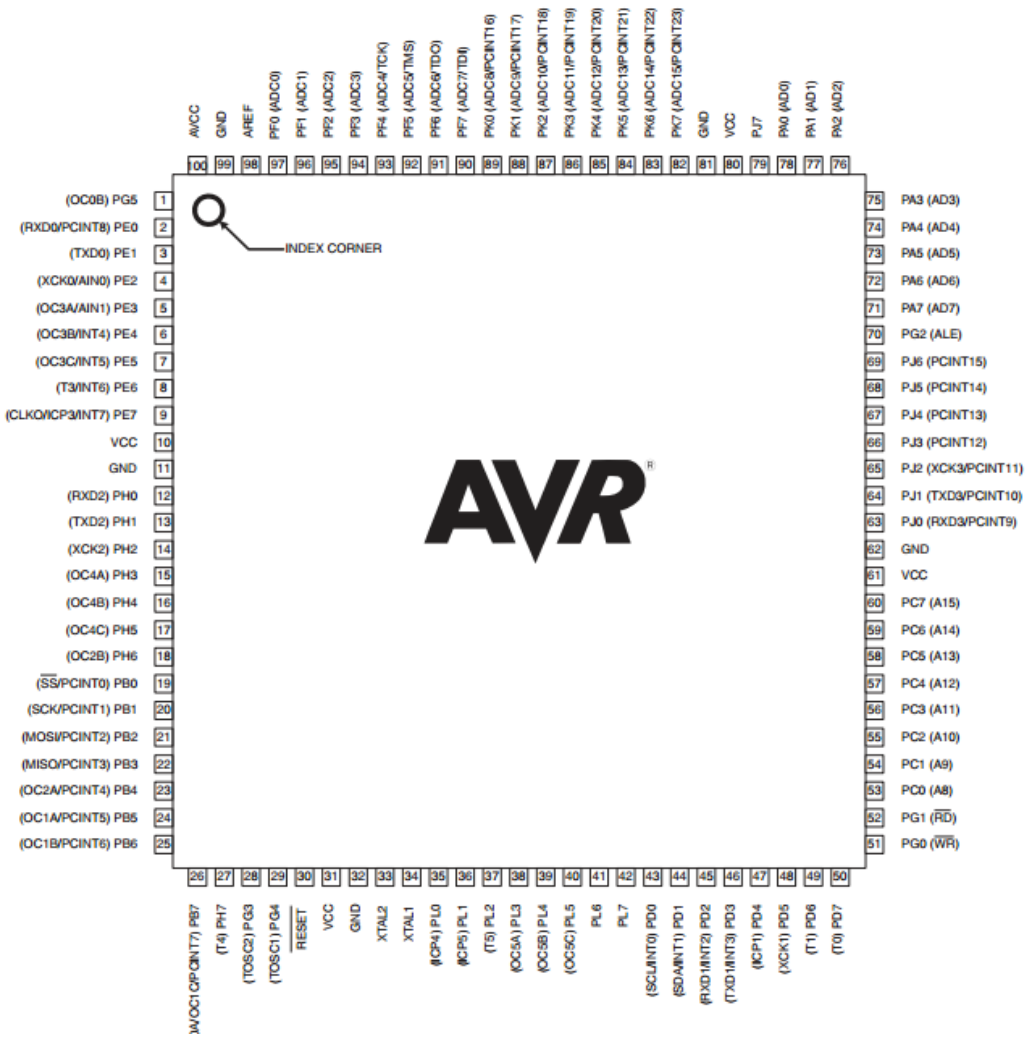


Figure 3 – 3: ATmega2560 Microprocessor

The microprocessor has:

1. 55 input/output pins
2. 16 analogue inputs
3. 4 UARTs (hardware serial ports)
4. A 16 MHz crystal oscillator
5. A USB connection
6. A power jack
7. An ICSP header
8. A reset button

All of the above elements are shown in the Figure 3 – 4:

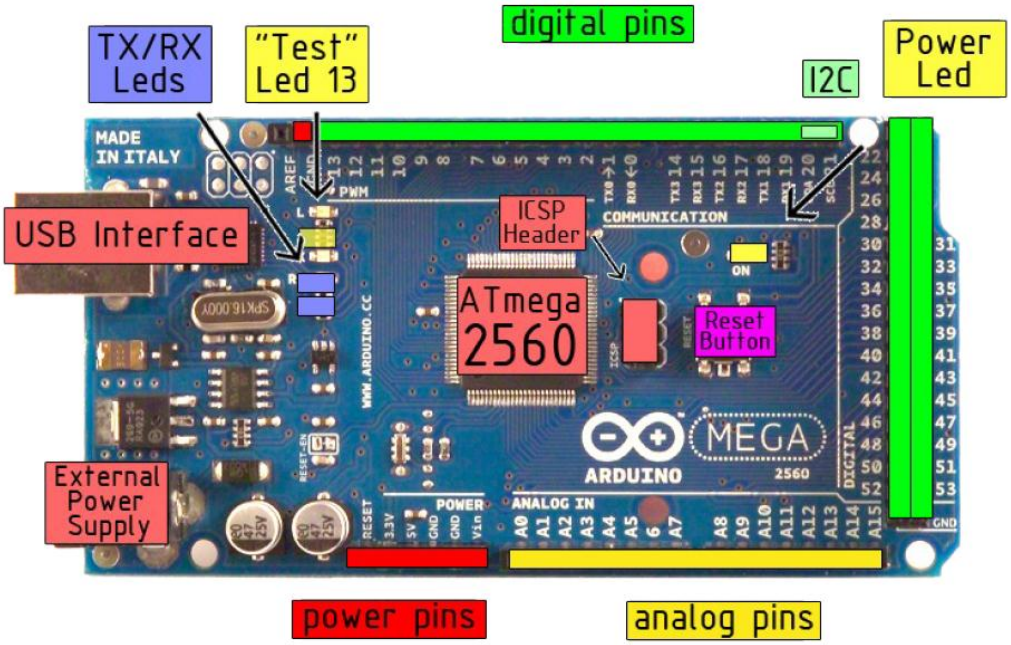


Figure 3 – 4: The Elements of the Arduino Mega 2560 R3

3.2.1.2 Characteristics of Arduino Mega 2560 R3

According to the official site of Arduino [48], the characteristics of the Arduino Mega2560 are the following:

1. Operating Voltage: 5V
2. Input Voltage (recommended): 7 – 12V
3. Input Voltage (limits): 6 – 20V
4. Digital I/O Pins: 54(of which 15 provide PWM output)
5. Analogue Input Pins: 16
6. DC Current per I/O Pin: 49mA
7. DC Current for 3.3V Pin: 50mA
8. Flash Memory: 256KB
9. SRAM: 8KB
10. EEPROM: 4KB
11. Clock Speed: 16MHz

The Arduino Mega can be powered up either from USB or an external power supply. Its power pins are the following:

1. VIN which is the input voltage when an external power supply is used,
2. 5V Pin which outputs 5V regulated voltage
3. 3V3 Pin which supplies 3.3V regulated voltage
4. GND Pins
5. IOREF Pin which provides the voltage reference with which the microcontroller operates.

Some of the I/O Pins of the microcontroller have specialized functions, which are:

1. Serial: 0 (RX) and 1 (TX); Serial 1: 19 (RX) and 18 (TX); Serial 2: 17 (RX) and 16 (TX); Serial 3: 15 (RX) and 14 (TX). Used to receive (RX) and transmit (TX) TTL serial data. Pins 0 and 1 are also connected to the corresponding pins of the ATmega16U2 USB-to-TTL Serial chip.
2. External Interrupts: 2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), and 21 (interrupt 2).
3. PWM: power pins 2 to 13 and 44 to 46.
4. SPI: 50 (MISO), 51 (MOSI), 52 (SCK), 53 (SS). These pins support SPI communication using the SPI library.
5. LED: 13. There is a built-in LED connected to digital pin 13.
6. TWI: 20 (SDA) and 21 (SCL). Support TWI communication using the Wire library.

3.2.1.3 1602 LCD HD44780 LCD Screen

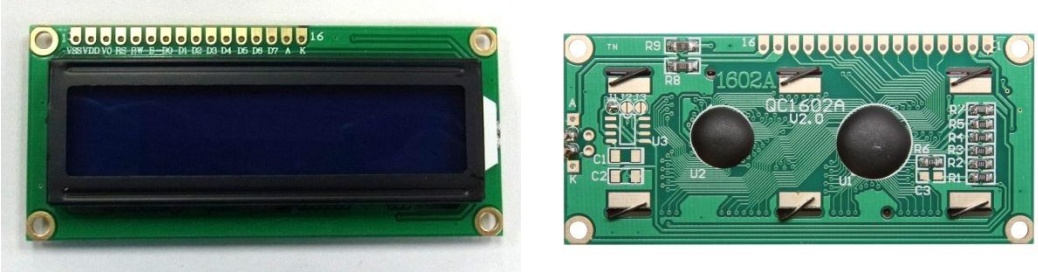


Figure 3 – 5: The front and the back side of 1602 LCD Screen

The main features of the LCD screen of the Figure 3 – 5 are the following:

1. Display format: 16 character x 2 lines
2. Input data: 4-Bits or 8-Bits interface available
3. Display Font: 5 x 8 Dots
4. Power Supply: 5V ± 10%
5. Driving Scheme: 1/16Duty, 1/5Bias
6. Backlight: Blue
7. Operating Temperature: 0 – 50°C

Pin assignment of the LCD Screen (Table 4):

No.	Symbol	Level	Function	
1	Vss	--	0V	Power Supply
2	Vdd	--	+5V	
3	V0	--	for LCD	
4	RS	H/L	Register Select: H:Data Input L:Instruction Input	
5	R/W	H/L	H--Read L--Write	
6	E	H,H-L	Enable Signal	
7	DB0	H/L	Data bus used in 8 bit transfer	
8	DB1	H/L		
9	DB2	H/L		
10	DB3	H/L		
11	DB4	H/L	Data bus for both 4 and 8 bit transfer	
12	DB5	H/L		
13	DB6	H/L		
14	DB7	H/L		
15	BLA	--	BLACKLIGHT +5V	
16	BLK	--	BLACKLIGHT 0V-	

Table 4: Pin Assignment of the LCD Screen

3.2.1.4 Experimental part

In the experimental part of this thesis, the following components were used:

1. Breadboards
2. Resistors of several values
3. 1 potentiometer of 5.5K
4. 1 LED Bright blue
5. 1 Photosensitive light sensor module
6. 1 single Relay module
7. 1 SD card Module
8. 1 Arduino Uno R3
9. 1 1602 LCD Screen

The Light Sensor simulates a possible sensor which will detect errors and damage which arise in each device that is plugged with the developing device; in a production stage, other alternatives could use a more elaborate, but definitely more expensive sensor that transmits power data. Those errors might be:

1. Over voltage
2. Under voltage
3. Faulty device

The LCD Screen was used to indicate the measurements of the light sensor and a flag "working/not working" device depending on the last monitoring status info of the device.

As for the LED light, it represents the device which is connected to the RC power strip. The device is powered independently from an external power source; in this case by a 9VDC battery. For this reason the light sensor was selected to do the monitoring of the device.

Experimental simulation data was saved in a .csv file on SD card, for further data processing.

In Figure 3 – 6, we show the electronic components and the wiring for the implementation of the experiment. The design was made using the Fritzing designing program.

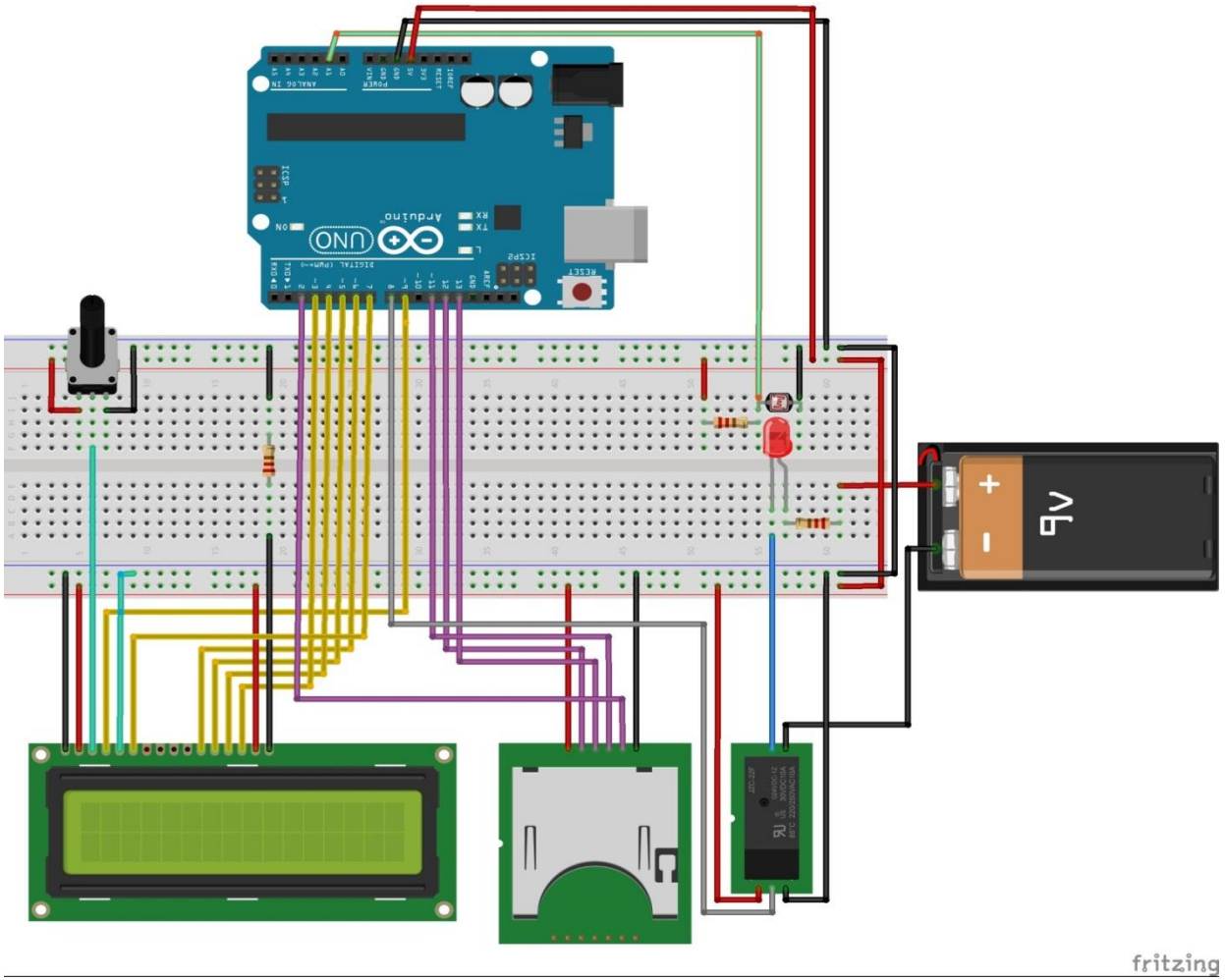


Figure 3 – 6: Connections in the Experimental Platform

3.3 Software

In this Section, we study the Serial and Parallel programming models. For the parallel programming model, both asynchronous and synchronous implementations of the developing program are examined. The models are compared by recording the execution time of specific parts of the program involving processing or sensor communication, or of the whole program. In particular, the measured values are the following:

1. Execution timings
    a. of some parts of the program in microseconds (μS)
    b. of the whole program in microseconds (μS)
2. The voltage of the LDR sensor in Volts.

3.3.1 Sequential

The serial model program with line numbers and detailed comments is provided next.

```
1  //importing of LCD library and SD library
2  #include <LiquidCrystal.h>
3  #include <SD.h>

4  /*Follows the definition of the variables in line, to create a
5    triangular waveform.*/
6  #define MAX_NO_ITERATIONS 8
7  #define MAX_VOLTAGE_SCALE 32

8  const int lightRes = A1; //This is a sensor (LDR) that checks
9                              if the Load indeed works
10 int ldr = 0; //definition of an ldr integer which is useful to
11                measure the analogue voltage from LDR Sensor
12 const int myRelay = 8; //The pin on which the Relay is
13                          connected
14 LiquidCrystal lcd(9, 7, 6, 5, 4, 3); //The pins on which the
15                                        LCD is connected
16 static int sensorFlag = 0;  //The flag that is used by the LDR
17                                Sensor, is low (0)
18 float voltage, ldrVolt;  //Definition of triangular voltage
19                           and LDR Sensor's Voltage as float
20 int iterations, VL, j; //This also helps in the produce of the
21                          triangular waveform
22 /* The following variables is used to measure the time between
23 the parts of the program and of the whole program */
24 unsigned long ldrTimerOn = 0, ldrTimerOff = 0,
25 timeRelayOnStart = 0, timeRelayOnEnd = 0, timeRelayOffStart =
26 0, timeRelayOffEnd = 0, overAllStart = 0, overAllEnd = 0;
27 /* The following variables is used to count how many times the
28 relayControl and ldrSensor functions are executed */
29 unsigned long thread1Count =0, thread2Count = 0;
30 File noThread; //Definition of the file which will be used to
31                 save data on SD card
32 boolean relayStatus = 0; //On 0 = normally off
33 long id = 1;    //This counts how many times the whole program
34                 is executed
```

```
/* Setting up the relay, the LCD and the Serial Communication
  */
void setup()
{
      Serial.begin(9600);
      lcd.begin(16, 2); //16 columns and 2 rows of the LCD
                         Screen
      pinMode (myRelay, OUTPUT);  //Declaration of the Relay
                                    pin as an Output
      while (!Serial)  //While there is not any serial signal
  {
    ; //just waiting...
  }
  Serial.print("Initializing SD Card...");
  pinMode(10, OUTPUT);  //This PIN, should always be stated as
                        an output, or else the SD will not
                        work.
  digitalWrite(2, HIGH);
  if (!SD.begin(2))  //If there is NOT begin the SD on pin 2
  {
    Serial.println("Card failure!"); //print this on Serial
    return;
  }
//Or print the following messages
  Serial.println("Card ready!");
  Serial.println("");
  Serial.println("Writing data...");
  Serial.println("");
  SD.remove("noThread.csv"); //When the Arduino restarts, the
                             noThread.csv file will be
                             deleted
  noThread = SD.open("noThread.csv", FILE_WRITE);     //and
                               recreated for writing data.
  if (noThread) //As long as the noThread file exists
  {
//write the following messages in the noThread file
noThread.println("S/N,      Device,      Voltage,      Status,
timeRelayOffStart,     timeRelayOffStop,     timeRelayOnStart,
timeRelayOnStop,  Sensor,     Device,  Voltage,  timeLDRStart,
timeLDRStop, allStart, allStop");
    noThread.close();    //Always close the file after any
                         addition
  }
  else
  {
    Serial.println("Error opening noThread.csv");
  }
}
//This function activates the relay
void relayOn()
{
      delay(20); //Wait for 20 mS
      digitalWrite(myRelay, HIGH); //Activate the relay
      lcd.setCursor(15, 0); //locating the cursor on LCD
```

```
89        lcd.print("T"); //And writing “T” (this means On) on LCD
90                        Screen
91    }
92    /* This function deactivates the relay in the same way as the
93    previous function */
94    void relayOff()
95    {
96          delay(20);
97          digitalWrite(myRelay, LOW); //Deactivates the relay
98          lcd.setCursor(15, 0);
99          lcd.print("F"); //writes “F” (Off) on LCD Screen
100   }
101   /* The following function writes down all the measuring data
102   from relayControl function to noThread.csv file on SD Card: */
103   void printControlStatus()
104   {
105     noThread = SD.open("noThread.csv", FILE_WRITE);
106     if (noThread)
107     {
108       noThread.print(id); //The count of execution of the
109                           program
110       noThread.print(",Relay1,");
111       noThread.print(voltage); //The triangular waveform
112       noThread.print(",");
113       noThread.print(relayStatus); //The relay status
114       noThread.print(",");
115       noThread.print(timeRelayOffStart);  //The  start  execution
116                                     time   of   the   relayOff
117                                     function
118       noThread.print(",");
119       noThread.print(timeRelayOffEnd); //The stop execution time
120                                     of the relayOff function
121       noThread.print(",");
122       noThread.print(timeRelayOnStart);  //The  start  execution
123                                     time   of   the   relayOn
124                                     function
125       noThread.print(",");
126       noThread.print(timeRelayOnEnd);  //The stop execution time
127                                     of the relayOn function
128       noThread.print(","); //This means an empty column
129       noThread.print(",");
130       noThread.print(",");
131       noThread.print(",");
132       noThread.print(",");
133       noThread.print(",");
134       noThread.print(overAllStart);  //The  start  execution  time
135                                     of the program
136       noThread.print(",");
137       noThread.println(overAllEnd); //The stop execution time of
138                                     the program
139       noThread.close(); //Closing the noThread file
140     }
141   }
142   /* The following function writes down in the same way as the
143   relayControl  function,  all  the  measuring  data  which  are
```

```
144  measured by the ldrSensor function to noThread.csv file on SD
145  Card: */
146  void printSensorStatus()
147  {
148    noThread = SD.open("noThread.csv", FILE_WRITE);
149    if (noThread)
150    {
151      noThread.print(",");
152      noThread.print(",");
153      noThread.print(",");
154      noThread.print(",");
155      noThread.print(",");
156      noThread.print(",");
157      noThread.print(",");
158      noThread.print(",S1,LDR Sensor,");
159      noThread.print(ldrVolt); //The analogue voltage of the LDR
160                               Sensor
161      noThread.print(",");
162      noThread.print(ldrTimerOn); //The start execution time of
163                                    the sensor
164      noThread.print(",");
165      noThread.print(ldrTimerOff); //The stop execution time of
166                                    the sensor
167      noThread.print(",");
168      noThread.print(",");
169      noThread.print(",");
170      noThread.println(",");
171      noThread.close(); //Closing of the noThread file
172    }
173  }
174  /* This function, controls the relay */
175  void relayControl()
176  {
177  /* The following "for" command, repeats the next command lines
178  of it, as long as the iterations are less than
179  MAX_NO_ITERATIONS number */
180      for (iterations = 0; iterations < MAX_NO_ITERATIONS;
181  iterations++)
182  /* The following "for" command, repeats the following code
183  lines of it, as long as the VL variable is less than
184  MAX_VOLTAGE_SCALE number */
185         for (VL = 0; VL < MAX_VOLTAGE_SCALE; VL++)
186         {
187           overAllStart = micros(); //Start counting executing
188                  time of all the program in uSec
189              if (iterations %2 != 0) //Checks if the remainder
190                                      of division is not 0
191                  {
192                      j = MAX_VOLTAGE_SCALE - VL;
193                  }
194              else
195                  {
196                      j = VL;
197                  }
```

```
198  /* The following code calculates the voltage which will
199  produce the triangular waveform: 3 is the maximum voltage
200  value, 1023 is the analogue reading. */
201                voltage = 3 - j * (1023 / MAX_VOLTAGE_SCALE) *
202  (3.0 / 1023.0);
203                id++; //this is inside the for loop as it is
204                  needed to count in every change of the voltage
205                lcd.setCursor(0, 0);
206                lcd.print("R1=");
207                lcd.setCursor(3, 0);
208                lcd.print(thread1Count); //the LCD shows how many
209                times the relayControl function is executed
210                lcd.setCursor(0, 1);
211                lcd.print("V1=");
212                lcd.setCursor(3, 1);
213                lcd.print(voltage); //Also shows the voltage
214  /* The following "if" statement checks if the voltage's values
215  are between 2.1 and 2.4 (operating range) and also checks if
216  the LDR Sensor has detected any error to the connected device
217  (LED) */
218                if (voltage >= 2.1 && voltage <= 2.4 &&
219  sensorFlag == 0)
220                 {
221                     thread1Count++; //counts how many times the
222                                     "if" statement is true
223                     timeRelayOnStart    =    micros();    //Starts
224                                       counting time in uSec
225                     relayOn(); //Activates the relay
226                     timeRelayOnEnd = micros(); //Stops counting
227                                       time in uSec
228                     relayStatus = 1; //Relay On
229                     printControlStatus();
230                 }
231                 else
232                 {
233                     timeRelayOffStart    =    micros();    //Starts
234                                       counting time in uSec
235                     relayOff(); //Deactivates the relay
236                     timeRelayOffEnd = micros(); //Stops counting
237                                       time in uSec
238                     relayStatus = 0; //Relay Off
239                     printControlStatus();
240                 }
241         }
242  }
243  /* The ldrSensor function, checks if the connected device
244  (LED) is functional, or if under voltage or high voltage is
245  detected */
246  void ldrSensor()
247  {
248        ldr = analogRead(lightRes); //Reading the analogue input
249                                    from PIN A1
250        ldrTimerOn = micros(); //Starts counting time in uSec
251        ldrVolt = ldr * (5.0 / 1023.0); //Transmute the analogue
252                                  reading into voltage
253        lcd.setCursor(8, 0);
```

```
254         lcd.print("S1=");
255         lcd.setCursor(11, 0);
256         thread2Count++; //counts how many times the ldrSensor
257                             function has executes
258         lcd.print(thread2Count); //the LCD shows how many times
259                           the ldrSensor function executes
260         lcd.setCursor(8, 1);
261         lcd.print("V2=");
262         lcd.setCursor(11, 1);
263         lcd.print(ldrVolt); //Prints on LCD the voltage of the
264                                 LDR Sensor
265   /* The following "if" statement, checks the voltage of the LDR
266   Sensor. If it is lower than 0.30 volts, or equal to this
267   value, it keeps the flag down (0). This means that the
268   connected device (LED) has no problem or there is not any
269   under voltage or high voltage. But if the LDR voltage is more
270   than 0.30 volts, it raises the flag which means there is a
271   problem on the connected device and so it alerts the
272   relayControl function not to activate the relay. */
273         if (ldrVolt <= 0.30)
274       {
275           sensorFlag = 0;
276       }
277       else
278       {
279           sensorFlag = 1;
280       }
281       ldrTimerOff = micros(); //Stops counting time in uSec
282       printSensorStatus();
283       overAllEnd = micros(); //Stops counting executing time of
284                                 all the program in uSec
285   }
286   //The infinitive loop
287   void loop()
288   {
289       relayControl(); //call of the relayControl function
290       ldrSensor(); //call of the ldrSensor function
291   }
```

3.3.2 Protothreads Asynchronous

We provide the asynchronous parallel programming model, with line numbers and comments where needed.

```
1  //importing of SD, protothreads and LCD library
2  #include <pt.h> //This is the protothreads library
3  #include <LiquidCrystal.h>
4  #include <SD.h>
5  /* Follows the definition of the variables in line, to create
6    a triangular waveform.*/
7  #define MAX_NO_ITERATIONS 8
8  #define MAX_VOLTAGE_SCALE 32
9  const int lightRes = A1; //This is a sensor (LDR) that checks
10                           if the Load indeed works
11 int ldr = 0; //definition of an ldr integer which is useful to
12               measure the analogue voltage from LDR Sensor
13 const int myRelay = 8; //The pin on which the Relay is
14                          connected
15 LiquidCrystal lcd(9, 7, 6, 5, 4, 3); //The pins on which the
16                                       LCD is connected
17 static struct pt pt1, pt2;    //Definition of the two
18                                protothreads
19 static int sensorFlag = 0;  //The flag that is used by the LDR
20                               Sensor, is low (0)
21 float voltage, ldrVolt;  //Definition of triangular voltage
22                           and LDR Sensor's Voltage as float
23                           variable
24 int iterations, VL, j; //This also helps in the produce of the
25                          triangular waveform
26 /* The following variables is used to measure the time between
27 the parts of the program and of the whole program */
28 unsigned long ldrTimerOn = 0, ldrTimerOff = 0,
29 timeRelayOnStart = 0, timeRelayOnEnd = 0, timeRelayOffStart =
30 0, timeRelayOffEnd = 0, overAllStart = 0, overAllEnd = 0;
31 /* The following variables is used to count how many times the
32 relayControl and ldrSensor threads are executed */
33 unsigned long thread1Count =0, thread2Count = 0;
34 File noSync; //Definition of the file which will be used to
35                save data on SD card
36 boolean relayStatus = 0; //On 0 = normally off
37 long id = 1; //This counts how many times the whole program is
38                executed
39 /* Setting up the relay, the LCD, the Serial Communication and
40   the protothreads */
41 void setup()
42 {
43   Serial.begin(9600);  //Starts the serial port at 9600
44   lcd.begin(16, 2); //16 columns and 2 rows of the LCD Screen
45   pinMode (myRelay, OUTPUT);   //Declaration of the Relay pin
46                                  as an Output
47   PT_INIT(&pt1);  //Initialization of
48   PT_INIT(&pt2);  //the two protothreads
49   while (!Serial)  //While there is not any serial signal
50   {
```

```
51      ; //just waiting...
52    }
53    Serial.print("Initializing SD Card...");
54    pinMode(10, OUTPUT); //This PIN, should always be stated as
55                         an output, or else the SD won't work.
56    digitalWrite(2, HIGH);
57    if (!SD.begin(2)) //If there is NOT begin the SD on  pin 2
58    {
59      Serial.println("Card failure!"); //print this on Serial
60      return;
61    }
62  //Or print the following messages
63    Serial.println("Card ready!");
64    Serial.println("");
65    Serial.println("Writing data...");
66    Serial.println("");
67    SD.remove("noSync.csv");  //When  the  Arduino  restarts,  the
68                                noSync.csv file will be deleted
69    noSync  =  SD.open("noSync.csv",  FILE_WRITE); //and recreated
70                                       for writing data.
71    if (noSync) //As long as the noSync.csv file exists
72    {
73      noSync.println("S/N,      Device,      Voltage,      Status,
74  timeRelayOffStart,      timeRelayOffStop,      timeRelayOnStart,
75  timeRelayOnStop,  Sensor,     Device,  Voltage,  timeLDRStart,
76  timeLDRStop, allStart, allStop");
77      noSync.close();     //Always  close  the  file  after  any
78                            addition
79    }
80    else
81    {
82      Serial.println("Error opening noSync.csv");
83    }
84  }
85  //This function activates the relay
86  void relayOn()
87  {
88    delay(20); //Wait for 20 mS
89    digitalWrite(myRelay, HIGH); //Activate the relay
90    lcd.setCursor(15, 0); //locating the cursor on LCD
91    lcd.print("T");  //And  writing  "T"  (this  means  On)  on  LCD
92                          Screen
93  }
94  /* This function deactivates the relay in the same way as the
95  previous function */
96  void relayOff()
97  {
98    delay(20);
99    digitalWrite(myRelay, LOW); //Deactivates the relay
100   lcd.setCursor(15, 0);
101   lcd.print("F"); //writes "F" (Off) on LCD Screen
102 }
103 /* The following function writes down all the measuring data
104 from relayControl thread to noSync.csv file on SD Card: */
105 void printControlStatus()
106 {
```

```
107     noSync = SD.open("noSync.csv", FILE_WRITE);
108     if (noSync)
109     {
110       noSync.print(id); //The count of execution of the program
111       noSync.print(",Relay1,");
112       noSync.print(voltage); //The triangular waveform
113       noSync.print(",");
114       noSync.print(relayStatus); //The relay status
115       noSync.print(",");
116       noSync.print(timeRelayOffStart);   //The   start   execution
117                                          time   of   the   relayOff
118                                          function
119       noSync.print(",");
120       noSync.print(timeRelayOffEnd);  //The  stop  execution  time
121                                          of the relayOff function
122       noSync.print(",");
123       noSync.print(timeRelayOnStart); //The start execution time
124                                          of the relayOn function
125       noSync.print(",");
126       noSync.print(timeRelayOnEnd); //The stop execution time of
127                                          the relayOn function
128       noSync.print(","); //This means an empty column
129       noSync.print(",");
130       noSync.print(",");
131       noSync.print(",");
132       noSync.print(",");
133       noSync.print(",");
134       noSync.print(overAllStart); //The start execution time of
135                                          the program
136       noSync.print(",");
137       noSync.println(overAllEnd); //The stop execution time of
138                                          the program
139       noSync.close(); //Closing the noSync.csv file
140     }
141   }
142   /* The following function writes down in the same way as the
143   relayController thread, all the measuring data which are
144   measured by the ldrSensor thread to noSync.csv file on SD
145   Card: */
146   void printSensorStatus()
147   {
148     noSync = SD.open("noSync.csv", FILE_WRITE);
149     if (noSync)
150     {
151       noSync.print(",");
152       noSync.print(",");
153       noSync.print(",");
154       noSync.print(",");
155       noSync.print(",");
156       noSync.print(",");
157       noSync.print(",");
158       noSync.print(",S1,LDR Sensor,");
159       noSync.print(ldrVolt); //The analogue voltage of the LDR
160                                  Sensor
161       noSync.print(",");
```

```
      noSync.print(ldrTimerOn);  //The start execution time of
                                      the sensor
      noSync.print(",");
      noSync.print(ldrTimerOff);  //The stop execution time of
                                      the sensor
      noSync.print(",");
      noSync.print(",");
      noSync.print(",");
      noSync.println(",");
      noSync.close(); //Closing of the noThread file
    }
}
//This function (protothread), controls the relay
static int relayController(struct pt *pt)
{
  PT_BEGIN(pt);  //From this point on, is the code of the
                  relayController Protothread
/* The following "for" command, repeats the next command lines
of it, as long as the iterations are less than
MAX_NO_ITERATIONS number */
  for (iterations = 0; iterations < MAX_NO_ITERATIONS;
iterations++)
/* The following "for" command, repeats the following code
lines of it, as long as the VL variable is less than
MAX_VOLTAGE_SCALE number */
      for (VL = 0; VL < MAX_VOLTAGE_SCALE; VL++)
{
      overAllStart = micros(); //Start counting executing time
                               of all the program in uSec
      if (iterations %2 != 0) //Checks if the remainder of
                                      division is not 0
      {
        j = MAX_VOLTAGE_SCALE - VL;
      }
      else
      {
        j = VL;
      }
/* The following code calculates the voltage which will
produce the triangular waveform: 3 is the maximum voltage
value, 1023 is the analogue reading. */
      voltage = 3 - j * (1023 / MAX_VOLTAGE_SCALE) * (3.0 /
1023.0);
      id++; //this is inside the for loop as it is needed to
               count in every change of the voltage
      lcd.setCursor(0, 0);
      lcd.print("R1=");
      lcd.setCursor(3, 0);
      lcd.print(thread1Count); //the LCD shows how many times
             the relayController thread is executed
      lcd.setCursor(0, 1);
      lcd.print("V1=");
      lcd.setCursor(3, 1);
      lcd.print(voltage); //Also shows the voltage
```

```
/* The following "if" statement checks if the voltage's values
are between 2.1 and 2.4 (operating range) and also checks if
the LDR Sensor has detected any error to the connected device
(LED) */
      if (voltage >= 2.1 && voltage <= 2.4 && sensorFlag == 0)
      {
        thread1Count++;  //counts how many times the "if"
                                    statement is true
        timeRelayOnStart = micros(); //Starts counting time in
                                              uSec
        relayOn(); //Activates the relay
        timeRelayOnEnd = micros(); //Stops counting time in
                                     uSec
        relayStatus = 1; //Relay On
        printControlStatus();
      }
      else
      {
        timeRelayOffStart = micros(); //Starts counting time
                                    in uSec
        relayOff(); //Deactivates the relay
        timeRelayOffEnd = micros(); //Stops counting time in
                                    uSec
        relayStatus = 0; //Relay Off
        printControlStatus();
      }
    }
  PT_END(pt); //Here stops the code of the relayController
               protothread
}
/* The ldrSensor thread, checks if the connected device (LED)
is functional, or if under voltage or high voltage is detected
*/
static int ldrSensor(struct pt *pt)
{
  PT_BEGIN(pt); //Here starts the code for ldrSensor Thread
    ldr = analogRead(lightRes); //Reading the analogue input
                                      from PIN A1
  ldrTimerOn = micros(); //Starts counting time in uSec
  ldrVolt = ldr * (5.0 / 1023.0); //Transmute the analogue
                                  reading into voltage
  lcd.setCursor(8, 0);
  lcd.print("S1=");
  lcd.setCursor(11, 0);
  thread2Count++;  //counts how many times the ldrSensor
                  function has executes
  lcd.print(thread2Count); //the LCD shows how many times the
                        ldrSensor function executes
  lcd.setCursor(8, 1);
  lcd.print("V2=");
  lcd.setCursor(11, 1);
  lcd.print(ldrVolt); //Prints on LCD the voltage of the LDR
                              Sensor
  /* The following "if" statement, checks the voltage of the
LDR Sensor. If it is lower than 0.30 volts, or equal to this
value, it keeps the flag down (0). This means that the
```

```
272   connected device (LED) has no problem or there is not any
273   under voltage or high voltage. But if the LDR voltage is more
274   than 0.30 volts, it raises the flag which means there is a
275   problem on the connected device and so it alerts the
276   relayControl function not to activate the relay. */
277     if (ldrVolt <= 0.30)
278     {
279       sensorFlag = 0;
280     }
281     else
282     {
283       sensorFlag = 1;
284     }
285     ldrTimerOff = micros(); //Stops counting time in uSec
286     printSensorStatus();
287     overAllEnd = micros(); //Stops counting executing time of
288                            all the program in uSec
289     PT_END(pt); //The end of code of the ldrSensor protothread
290   }
291   //The infinitive loop
292   void loop()
293   {
294     relayController(&pt1); //call of the relayController thread
295     ldrSensor(&pt2); //call of the ldrSensor thread
296   }
```

3.3.4 Protothreads with Synchronization

We examine the synchronous parallel programming model, with line numbers and comments where necessary.

```
1  //importing of SD, protothreads and LCD library
2  #include <pt.h>
3  #include <LiquidCrystal.h>
4  #include <SD.h>
5  /* Follows the definition of the variables in line, to create
6    a triangular waveform.*/
7  #define MAX_NO_ITERATIONS 8
8  #define MAX_VOLTAGE_SCALE 32
9  const int lightRes = A1; //This is a sensor (LDR) that checks
10                           if the Load indeed works
11 int ldr = 0; //definition of an ldr integer which is useful to
12               measure the analogue voltage from LDR Sensor
13 const int myRelay = 8; //The pin on which the Relay is
14                          connected
15 LiquidCrystal lcd(9, 7, 6, 5, 4, 3); //The pins on which the
16                                       LCD is connected
17 static struct pt pt1, pt2;    //Definition of the two
18                                protothreads
19 static int riseFlag = 0, sensorFlag = 0; //In this program,
20                           were added two flags, one for the
21                           relayController thread and one for
22                           the ldrSensor thread
23 float voltage, ldrVolt;  //Definition of triangular voltage
24                           and LDR Sensor's Voltage as float
25                           variable
26 int iterations, VL, j; //This also helps in the produce of the
27                          triangular waveform
28 /* The following variables is used to measure the time between
29 the parts of the program and of the whole program */
30 unsigned long ldrTimerOn = 0, ldrTimerOff = 0,
31 timeRelayOnStart = 0, timeRelayOnEnd = 0, timeRelayOffStart =
32 0, timeRelayOffEnd = 0, relayWaitOn = 0, relayWaitOff = 0,
33 overAllStart = 0, overAllEnd = 0;
34 /* The following variables is used to count how many times the
35 relayControl and ldrSensor threads are executed */
36 unsigned long thread1Count =0, thread2Count = 0;
37 File syncFile; //Definition of the file which will be used to
38                  save data on SD card
39 boolean relayStatus=0; //On 0 = normally off
40 long id = 1; //This counts how many times the whole program is
41                executed
42 /* Setting up the relay, the LCD, the Serial Communication and
43   the protothreads */
44 void setup()
45 {
46   Serial.begin(9600);  //Starts the serial port at 9600
47   lcd.begin(16, 2); //16 columns and 2 rows of the LCD Screen
48   pinMode (myRelay, OUTPUT);   //Declaration of the Relay pin
49                                  as an Output
50   PT_INIT(&pt1);  //Initialization of
```

```
51    PT_INIT(&pt2);  //the two protothreads
52    while  (!Serial)  //While  there  is  not  any  serial  signal
53  {
54      ; //just waiting...
55  }
56    Serial.print("Initializing SD Card...");
57    pinMode(10, OUTPUT);  //This PIN, should always be stated as
58                        an output, or else the SD won't work.
59    digitalWrite(2, HIGH);
60    if (!SD.begin(2))  //If there is NOT begin the SD on  pin 2
61    {
62      Serial.println(" Card failure!"); //print this on Serial
63      return;
64    }
65  //Or print the following messages
66    Serial.println(" Card ready!");
67    Serial.println("");
68    Serial.println("Writing data...");
69    Serial.println("");
70    SD.remove("syncFile.csv"); //When the Arduino restarts, the
71                               syncFile.csv   file   will   be
72                               deleted
73  syncFile   =   SD.open("syncFile.csv",   FILE_WRITE);      //and
74                                        recreated  for  writing
75                                        data.
76    if (syncFile) //As long as the syncFile.csv file exists
77    {
78      syncFile.println("S/N,     Device,     Voltage,     Status,
79  timeRelayOffStart,       timeRelayOffStop,       timeWaitStart,
80  timeWaitStop, timeRelayOnStart, timeRelayOnStop, S1,   Device,
81  Voltage,     timeWaitStart,     timeWaitStop,     overAllStart,
82  overAllEnd");
83      syncFile.close();    //Always  close  the  file  after  any
84                           addition
85    }
86    else
87    {
88      Serial.println("Error opening syncFile.csv");
89    }
90  }
91  //This function activates the relay
92  void relayOn()
93  {
94    delay(20); //Wait for 20 mSec
95    digitalWrite(myRelay, HIGH); //Activate the relay
96    lcd.setCursor(15, 0); //locating the cursor on LCD
97    lcd.print("T");  //And  writing  "T"  (this  means  On)  on  LCD
98                        Screen
99  }
100 /* This function deactivates the relay in the same way as the
101 previous function */
102 void relayOff()
103 {
104   delay(20);
105   digitalWrite(myRelay, LOW); //Deactivates the relay
106   lcd.setCursor(15, 0);
```

```
107     lcd.print("F"); //writes “F” (Off) on LCD Screen
108   }
109   /* The following function writes down all the measuring data
110   from relayControl thread to noSync.csv file on SD Card: */
111   void printControlStatus()
112   {
113     syncFile = SD.open("syncFile.csv", FILE_WRITE);
114     if (syncFile)
115     {
116       syncFile.print(id);  //The count of execution of the
117                            program
118       syncFile.print(",Relay1,");
119       syncFile.print(voltage); //The triangular waveform
120       syncFile.print(",");
121       syncFile.print(relayStatus); //The relay status
122       syncFile.print(",");
123       syncFile.print(timeRelayOffStart);  //The start execution
124                                      time of the relayOff
125                                      function
126       syncFile.print(",");
127       syncFile.print(timeRelayOffEnd); //The stop execution time
128                                      of the relayOff function
129       syncFile.print(",");
130       syncFile.print(relayWaitOn);  //The relayWaitOn and the
131                           relayWaitOff variables help to count how
132                           much time in uSec the relayController
133                           waits for the ldrSensor to confirm that
134                           the connected device has no problem
135       syncFile.print(relayWaitOff); //The same as the previous
136                                      command
137       syncFile.print(","); //This means an empty column
138       syncFile.print(timeRelayOnStart);  //The start execution
139                                      time of the relayOn
140                                      function
141       syncFile.print(",");
142       syncFile.print(timeRelayOnEnd); //The stop execution time
143                                      of the relayOn function
144       syncFile.print(",");
145       syncFile.print(",");
146       syncFile.print(",");
147       syncFile.print(",");
148       syncFile.print(",");
149       syncFile.print(",");
150       syncFile.print(overAllStart); //The start execution time
151                                       of the program
152       syncFile.print(",");
153       syncFile.println(overAllEnd); //The stop execution time of
154                                      the program
155       syncFile.close(); //Closing the syncFile.csv file
156     }
157   }
158   /* The following function writes down in the same way as the
159   relayControl thread, all the measuring data which are measured
160   by the ldrSensor thread to noSync.csv file on SD Card: */
161   void printSensorStatus()
162   {
```

```
163     syncFile = SD.open("syncFile.csv", FILE_WRITE);
164     if (syncFile)
165     {//1
166       syncFile.print(",");
167       syncFile.print(",");
168       syncFile.print(",");
169       syncFile.print(",");
170       syncFile.print(",");
171       syncFile.print(",");
172       syncFile.print(",");
173       syncFile.print(",");
174       syncFile.print(",");
175       syncFile.print(",S1,LDR Sensor,");
176       syncFile.print(ldrVolt); //The analogue voltage of the LDR
177                                Sensor
178       syncFile.print(",");
179       syncFile.print(ldrTimerOn); //The start execution time of
180                                    the sensor
181       syncFile.print(",");
182       syncFile.print(ldrTimerOff); //The stop execution time of
183                                    the sensor
184       syncFile.print(",");
185       syncFile.print(",");
186       syncFile.println(overAllEnd); //The finish execution  time
187   of the program
188       syncFile.close();
189     }
190   }
191   //This function (protothread), controls the relay
192    static int relayController(struct pt *pt)
193   {
194     PT_BEGIN(pt);  //From this point on, is the code of the
195                    relayController Protothread
196     {
197       /* The following "for" command, repeats the next command
198   lines of it, as long as the iterations are less than
199   MAX_NO_ITERATIONS number */
200       for (iterations = 0; iterations < MAX_NO_ITERATIONS;
201   iterations++)
202   /* The following "for" command, repeats the following code
203   lines of it, as long as the VL variable is less than
204   MAX_VOLTAGE_SCALE number */
205           for (VL = 0; VL < MAX_VOLTAGE_SCALE; VL++)
206   //   waveform of 0-3-0 volts
207         {
208           overAllStart = micros(); //Start counting executing
209                                    time of all the program in uSec
210            if (iterations %2 != 0) //Checks if the remainder of
211                                    division is not 0
212           {
213             j = MAX_VOLTAGE_SCALE - VL;
214           }
215           else
216           {
217             j = VL;
218           }
```

```
/* The following code calculates the voltage which will
produce the triangular waveform: 3 is the maximum voltage
value, 1023 is the analogue reading. */
        voltage = 3 - j * (1023 / MAX_VOLTAGE_SCALE) * (3.0 /
1023.0);
        id++; //this is inside the for loop as it is needed to
               count in every change of the voltage
        lcd.setCursor(0, 0);
        lcd.print("R1=");
        lcd.setCursor(3, 0);
        lcd.print(thread1Count);  //the LCD shows how many
              times the relayController thread is executed
        lcd.setCursor(0, 1);
        lcd.print("V1=");
        lcd.setCursor(3, 1);
        lcd.print(voltage); //Also shows the voltage
/* The following "if" statement checks if the voltage's values
are between 2.1 and 2.4 (operating range) */
        if (voltage >= 2.1 && voltage <= 2.4)
        {
          riseFlag = 1; //It rises the flag to inform the
                        ldrSensor thread that the voltage is
                        between 2.1 and 2.4 volts
          thread1Count++; //counts how many times the "if"
                                statement is true
          relayWaitOn = micros(); //The start time in uSec of
                           how many time the relayController
                           thread waits the ldrSensor thread
                           to confirm that the connected
                           device (LED) has no problem
          PT_WAIT_UNTIL(pt,    sensorFlag    !=    0);    //The
                           relayController waits until the
                           sensorFlag raises
          relayWaitOff = micros(); //The stop time
          sensorFlag = 0; //downhaul of flag
          if (ldrVolt <= 0.30) //This "if" statement checks if
                            the voltage of the ldrSensor
                            is less than 0.30V
          {
              timeRelayOnStart = micros(); //Starts counting
                                          time in uSec
              relayOn(); //Activates the relay
              timeRelayOnEnd = micros(); //Stops counting time
                                          in uSec
              relayStatus = 1; //Relay On
              printControlStatus();
          }
          else
          {
            timeRelayOffStart = micros(); //Starts counting
                                       time in uSec
            relayOff(); //Deactivates the relay
            timeRelayOffEnd = micros(); //Stops counting time
                                       in uSec
            relayStatus = 0; //Relay Off
            printControlStatus();
```

```
275             }
276             riseFlag = 1; //Rises the flag
277           }
278           else
279           {
280             timeRelayOffStart = micros(); //Starts counting time
281                                           in uSec
282             relayOff();//Deactivates the relay
283             timeRelayOffEnd = micros(); //Stops counting time in
284                                           uSec
285             relayStatus = 0; //Relay Off
286             printControlStatus();
287             riseFlag = 1; //Rises the flag
288           }
289         }
290     }
291     PT_END(pt);  //Here  stops  the  code  of  the  relayController
292                   protothread
293   }
294   /* The ldrSensor thread, checks if the connected device (LED)
295   is functional, or if under voltage or high voltage is detected
296   */
297    static int ldrSensor(struct pt *pt)
298   {
299     PT_BEGIN(pt); //Here starts the code for ldrSensor Thread
300     ldr  =  analogRead(lightRes);  //Reading  the  analogue  input
301                                           from PIN A1
302     ldrTimerOn = micros();//Starts counting time in uSec
303     ldrVolt  =  ldr  *  (5.0  /  1023.0);  //Transmute  the  analogue
304                                     reading into voltage
305     lcd.setCursor(8, 0);
306     lcd.print("S1=");
307     lcd.setCursor(11, 0);
308     thread2Count++;  //counts  how  many  times  the  ldrSensor
309                     function has executes
310     lcd.print(thread2Count); //the LCD shows how many times the
311                                   ldrSensor function executes
312     lcd.setCursor(8, 1);
313     lcd.print("V2=");
314     lcd.setCursor(11, 1);
315     lcd.print(ldrVolt); //Prints on LCD the voltage of the LDR
316                                 Sensor
317     sensorFlag = 1; //Rises the sensor flag
318     PT_WAIT_UNTIL(pt,  riseFlag  !=  0);  //The  ldrSensor  thread
319                                   waits  until  the  relayController
320                                   thread  rise  the  flag,  which
321                                   means  that  the  relayController
322                                   has  entered  in  the  operation
323                                   range of voltage
324     riseFlag = 0; // downhaul of flag
325     ldrTimerOff = micros(); //Stops counting time in uSec
326     overAllEnd  =  micros();  //Stops  counting  executing  time  of
327                                 all the program in uSec
328     printSensorStatus();
329     PT_END(pt); //The end of code of the ldrSensor protothread
330   }
```

```
331  //The infinitive loop
332  void loop()
333  {
334    relayController(&pt1); //call of the relayController thread
335    ldrSensor(&pt2); //call of the ldrSensor thread
336  }
```

4. Results

4.1 Metrics

In this Section, the experimental values of the three programming models are presented and analysed. The programs ran for one hour, collecting a large amount of data which is shown in charts in order to examine the behaviour of the microprocessor during the execution of each of the three types of programming models.

4.1.1. NoThread.csv file

For the serial programming model, six tabs of Excel sheets are presented, with measured and calculated data. These tabs are the following:

a. **NoThread.** This tab contains all measured data of the experiment. They are presented in 15 columns of data, which are:

1. S/N: Serial Number
2. Device: Identifies the relay used
3. Voltage: The values of the virtual triangular waveform (3 – 0 – 3V)
4. Status: The state of the relay; 1 = ON, 0 = OFF
5. timeRelayOffStart: Capture the start execution time of the `relayOff()` function
6. timeRelayOffStop: Capture the stop execution time of the `relayOff()` function
7. timeRelayOnStart: Capture the start execution time of the `relayOn()` function
8. timeRelayOnStop: Capture the stop execution time of the `relayOn()` function
9. Sensor: ID of the sensor
10. Device: Type of the sensor
11. LDR Voltage: Capture of the voltage from the light sensor from the analogue input of Arduino
12. timeLDRStart: Capture of the start execution time of the ldrSensor thread
13. timeLDRStop: Capture of the stop execution time of the ldrSensor thread
14. allStart: Capture of the start execution time of the whole program
15. allStop: Capture of the stop execution time of the whole program

b. **Voltage and Error.** This tab contains some of the data of the NoThread tab as well as one column of evaluated data in order to create the Voltage and Status chart, as long as the Error flag in Generating Triangular Voltage Distribution is not set. This tab contains the following columns:

1. Voltage: Data were taken from the corresponding column of the NoThread tab
2. allStart: Data was taken from the corresponding column of the NoThread tab
3. Status: Data was taken from the corresponding column of the NoThread tab
4. allStop: Data was taken from the corresponding column of the NoThread tab

5. Error: This shows the time which is needed for the program to be executed in a period (the complete execution of the program)

The following charts were exported:

1. The virtual triangular waveform: Represents the virtual voltage 0 – 3 – 0 volts, versus start execution time of the program (allStart).
2. The Status of the relay: It is combined in the previously mentioned waveform, in a second y axis and it represents the On or Off statements of the relay through time versus voltage
3. The diagram named "Error in Generating Triangular Voltage Distribution": Represents graphically the stop execution time of every period of the execution of the program. It results from the allStop column versus the Error column.

c. Time per Period. This tab shows how many times the program was executed in the total execution time of the 1 hour. The following data columns were used:

1. Voltage: Data was taken from the corresponding column of the NoThread tab
2. allStart: Data was taken from the corresponding column of the NoThread tab
3. allStop: Data was taken from the corresponding column of the NoThread tab
4. Time per Period (uSec): Shows the average execution time of each period of the program in microseconds
5. Time per Period (Sec): Shows the average execution time of each period of the program in seconds
6. Period: Shows the number of iterations of the execution of the program, in the total time of one hour. It resulted from counting the number 3 of the voltage column, by checking all the values of the voltage column.
7. LDR Voltage: The voltage of the LDR sensor is recorded each time the ldrSensor thread executes
8. Period of the ldrVoltage: The number of executions of the ldrSensor thread is indicated.

d. Status Error. In this tab, we checked if there was any incorrect state of the relay module. It was checked if the relay was in an ON state, while the virtual triangular voltage and the sensor's voltage was out of the predefined operational range and vice versa. It is recalled that the operating range of the virtual triangular voltage is from 2.1 Volts to 2.4 Volts and the operational range of the LDR's voltage is less than 0.30 Volts.

The following columns of data were used:

1. Voltage: Data was taken from the corresponding column of the NoThread tab
2. ldrVoltage: Data was taken from the corresponding column of the NoThread tab
3. allStart: Data was taken from the corresponding column of the NoThread tab
4. Theoretical Status: Represents the state of the relay as it should work, in accordance to the virtual triangular voltage and the LDR Sensor's voltage. The formula which were used in excel, was the following:
   $=IF(AND(A_{x1}>=2.1;A_{x1}<=2.4;B_{x1}<=0.3);1;0)$

5. Practical Status: Data taken from the Status column of the NoThread tab
6. Error: In this data column, the Theoretical Status and the Practical Status were compared by using the formula:
   $=IF(D_{x1}=E_{x1};0;1)$.

   If the relay module has any wrong state, then 1 should be written in the Error column or else should be written 0, which indicates that the relay has the correct state, according to the given parameters.

7. allStart: Same values as previously.
8. Error: Same values as previously.

The allStart and the Error columns were copied in order to be used together in the exportation of the diagram which is presented on this tab. In its x axis are assigned the allStart data column and in its y axis the Error data column. This diagram, named Error, shows graphically the possibly state errors of the relay, during the execution time of the program, which is the time of 1 hour.

e. RelayOn Delay. This tab calculated the variation of the `relayOn()` function and exported it into a graph versus the timeRelayOnStart data column, which is the start execution time of the `relayOn()` function of the program.
The following data columns were used:

1. AllStart: Data was taken from the corresponding column of the NoThread tab
2. Time: This data column clarifies at what time of the column allStart, the relay is activated and deactivated, when i.e. the `relayOn()` function runs. The values of this column are resulted by using the following excel formula:
   $=IF(AND(Cx_2=Cx_1;Dx_2=Dx_1);0;Ax_2)$

   This formula shows that if both of the equations which are in the brackets, are true, then the value of 0 will be written in the corresponding cell, in which the formula is calculated. Otherwise, the value of the corresponding Ax cell will be written in the formula cell.

3. timeRelayOnStart: Data was taken from the corresponding column of the NoThread tab
4. timeRelayOnStop: Data was taken from the corresponding column of the NoThread tab
5. RelayOn Time Variation. This data column shows the duration of the `relayOn()` function. Its values are resulted by using the formula:
   $=Dx_1-Cx_1$

   This formula results the time duration of the `relayOn()` function; A subtraction of the stop execution time of the `relayOn()` function from the start execution time of the same function is made, in order to export the duration time result.

Next, the previously mentioned columns (allStart, RelayOn Time Variation, timeRelayOnStart, timeRelayOnStop, Variation RelayOn) were sorted in ascending order based on the RelayOn Time Variation data column.

From the previously mentioned classification, the values of the timeRelayOnStart and the Variation RelayOn data columns were used, but only those which start from the point on which the relay changes state, i.e. as calculated in Time data column. So, those data are copied in two new columns:

1. timeRelayOnStart sorted
2. Variation RelayOn sorted.

Consequently, the two new data columns are used in order to export the "Variation RelayOn Sorted" graph in this tab. The x-axis represents the start execution time of the `relayOn()` function and the y-axis corresponds to the variation of the `relayOn()` function.

f. RelayOff Delay. In the same way as the RelayOn Delay tab was calculated, the variation of the `relayOff()` function was exported into a graph versus the timeRelayOffStart data column, which is the start execution time of the `relayOff()` function of the program.
The following data columns were used:

1. AllStart: Data was taken from the corresponding column of the NoThread tab
2. Time: This data column clarifies at what time of the column allStart, the relay is activated and deactivated. When i.e. the `relayOff()` function runs. The values of this column are resulted by using the following excel formula:
   $=IF(AND(Cx_2=Cx_1;Dx_2=Dx_1);0;Ax_2)$

   This formula shows that if both of the equations which are in the brackets, are true, then the value of 0 will be written in the corresponding cell, in which the formula is calculated. Otherwise, the value of the corresponding Ax cell will be written in the formula cell.

3. timeRelayOffStart: Data was taken from the corresponding column of the NoThread tab
4. timeRelayOffStop: Data was taken from the corresponding column of the
5. NoThread tab
6. RelayOff Time Variation. This data column shows the duration of the `relayOff()` function. Its values are resulted by using the formula:
   $=Dx_1-Cx_1$

   This formula results the time duration of the `relayOff()` function; A subtraction of the stop execution time of the `relayOff()` function from the start execution time of the same function is made, in order to export the duration time result.

In the same way as the previous tab was created the two new columns:

1. timeRelayOffStart sorted
2. Variation RelayOff sorted.

Consequently, the two new data columns are used in order to export the "Variation RelayOff Sorted" graph in this tab. The x-axis represents the start execution time of the `relayOff()` function and the y-axis corresponds to the variation of the `relayOff()` function.

4.1.2. NoSync.csv File

This file corresponds to asynchronous parallel programming model. The procedure was similar to the previous one, as the data types were the same with those of Section 4.1.1. Therefore, we evaluated same things and were arisen charts which correspond to those of the Section 4.1.1.

4.1.3. SyncFile.csv File

This file corresponds to synchronous parallel programming model. The procedure was also similar to the two previous programming models, but there were some important differences which will be analysed below:

a. **SyncFile tab**: In this tab, measurements and timings of the experiment was the same as the previous two sections except from:

1. timeWaitStart: Capture of the start time of the relayController thread waiting, which is achieved by using the PT_WAIT_UNTIL command of the Protothreads library
2. timeWaitStop: Capture of the stop time of the relayController thread waiting, which is also achieved by using the PT_WAIT_UNTIL command of the Protothreads library
3. timeWaitStart: Capture of the start time of the ldrSensor thread waiting, which is achieved by using the PT_WAIT_UNTIL command of the Protothreads library
4. timeWaitStop: Capture of the stop time of the ldrSensor thread waiting, which is also achieved by using the PT_WAIT_UNTIL command of the Protothreads library

b. **Voltage and Error.** In this tab we followed the same process as we did in the correspond tab of the two previous sections. But we have differences in the chart "Error in Generating Triangular Voltage Distribution" which is analysed below.

The first chart which exported (Virtual triangular Waveform and Relay Status) is the same as the corresponding one of the two previous sections. But in the second diagram named "Error in Generating Triangular Voltage Distribution", there are some differences between the current programming model and the two others. This lies in the fact that the two threads are synchronized between them by using the `PT_WAIT_UNTIL` command. An important effect that the synchronization results are that the execution time is differently managed, in comparison to the other programming models. Also, the time behaviour of the program is different.

Analysing furthermore the previously mentioned data, the following remarks and conclusions are arising:

1. While the virtual triangular voltage is out of the range of 2.1 – 2.4 Volts, the program is on hold i.e. never reach its termination limit, until the virtual triangular voltage's range is within the above range. This behaviour results from the use of the PT_WAIT_UNTIL command. Thus, the following execution timings are resulted:

a. t_1 Time limit: The virtual triangular voltage is within the range of 3 – 2.41 Volts
b. t_2 Time limit: The virtual triangular voltage is within the range of 2.4 – 2.1 Volts and 2.1 – 2.4 Volts
c. t_3 Time limit: The virtual triangular voltage is within the range of 2.09 – 0 – 2.09 Volts
d. t_4 Time limit: The virtual triangular voltage is within the range of 2.41 – 3 Volts

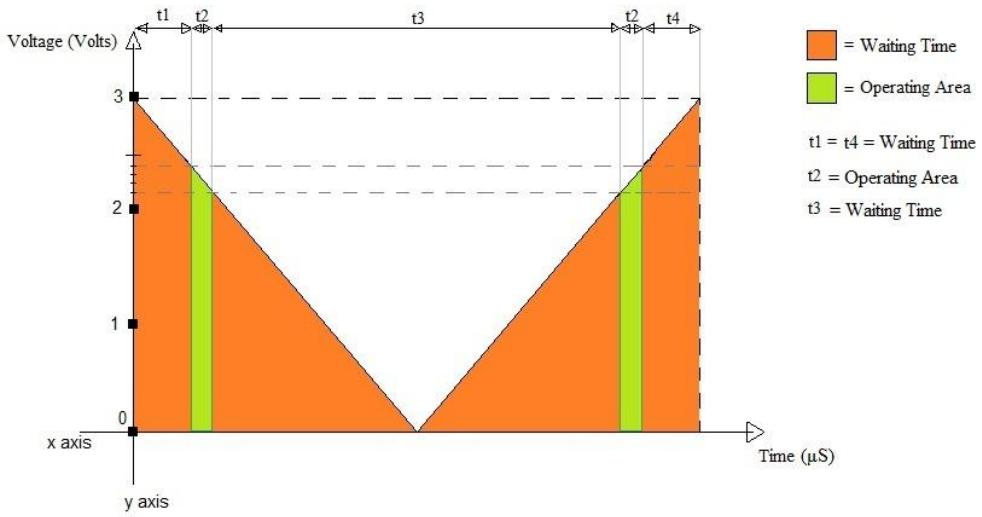


Figure 4 – 1: Graphical View of the Timetable of the Program in one period

4.1.4 Comparing the results

After analysing the collected data, it is important to compare results from the different programming models. The file "**Models Comparison.xlsx**" was created in order to show this comparison.

In this file four tabs of data are presented, which are:

a. Voltage and Error Comp. In this tab, we present collected data ("**allStop**" column and "**Error**" column of "**Voltage and Error**" tab) of the three models: the serial programming model (in blue), the asynchronous parallel programming model (in red) and finally, the synchronous parallel programming model (in green). Figure 4 – 20 compares execution time, as analysed in the next Section. In addition, Table 6 compares the percentage difference of the execution speed of the three programming models.

We also present the Table 7, titled as "**Execution Time Variation for each Model (µS)**", which shows the variation between the execution timings of the programming models. In simpler words, with the help of this table, we show the difference between

the **final end** execution time of the program, minus its **first end** execution time of the same program, for each programming model, in μS.

b. **TimePerPeriod Comp.** In this tab, we present the "**Comparison of the Period of the Program and of the ldrSensor Thread**" table, which presents the average execution time of one period of the program in μS and in S, the periods of the program in the time of 1 hour and the periods of the ldrSensor in the same time, for the three programming models.

c. **RelayOn Delay Comp.** In this tab, we present the "**timeRelayOnStart Sorted**" and the "**Variation RelayOn Sorted**" data of the **RelayOn Delay** tab of the three files. The highlighted colours are the same as previous. Also the "**Comparison of the RelayOn Variation of the three programming models**" diagram is resulted. It will be analysed in the next Section.

d. **RelayOff Delay Comp.** In the same way as the previous tab, we present the **timeRelayOffStart Sorted** and the **Variation RelayOff Sorted** data of the **RelayOff Delay** tab of the three files. The highlighted colours are also the same and the "**Comparison of the RelayOff Variation of the three programming models**" diagram is resulted.

4.2 Graphics and Evaluations

In this Section, we present charts and diagrams based on data analysis from the experimental part of this thesis, at first examining each programming model separately.

4.2.1 NoThread.csv file's diagrams

By processing certain tabs of the spread sheet (see Section 4.1), we obtain the following charts:

1. Voltage and Error Tab Diagrams.

2. Error in Generating Triangular Voltage Distribution

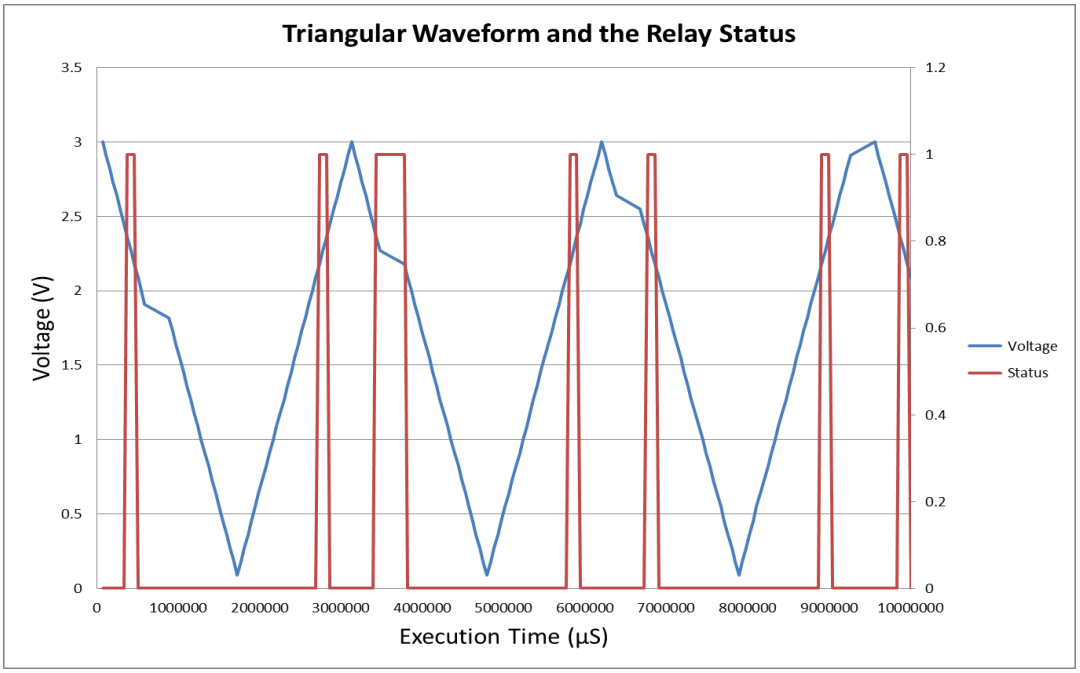


Figure 4 – 2: Virtual Triangular Waveform and the Relay Status

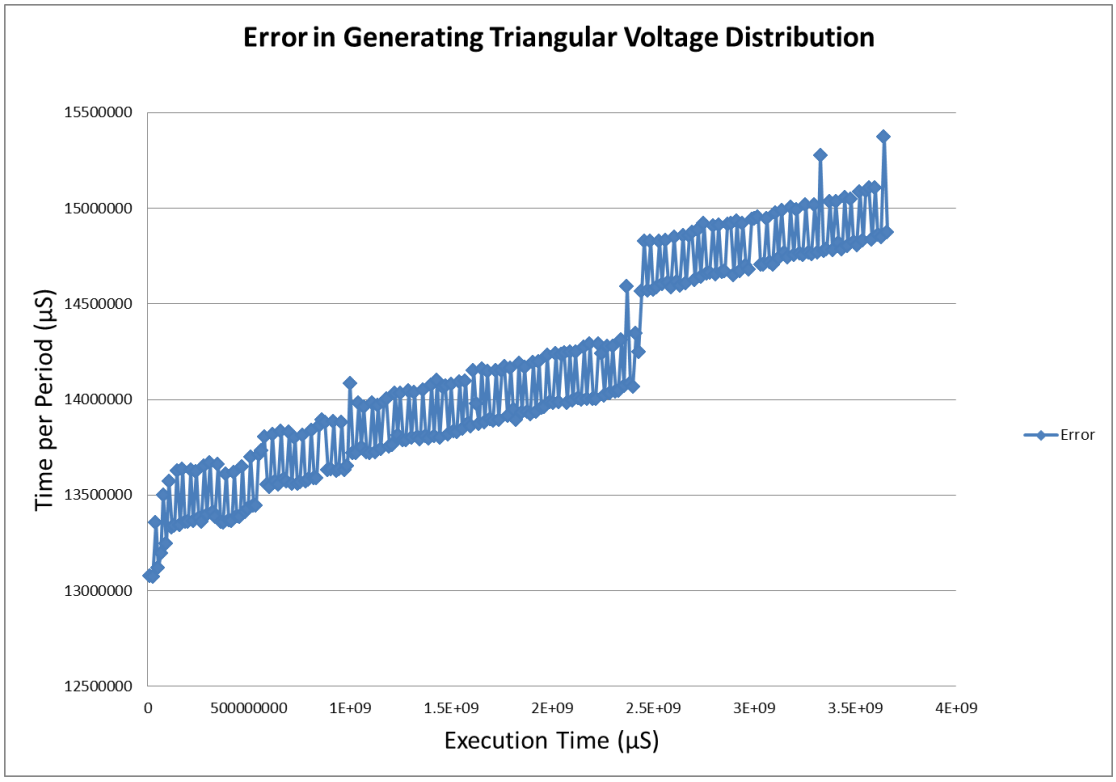


Figure 4 – 3: Error in Generating Triangular Voltage Distribution

- Figure 4 – 2 is a combination of the virtual triangular voltage sweep and of the Relay status, versus the total execution time of the program, which is about 1 hour. The triangular voltage is illustrated in the y_1 axis (left vertical), the Relay status in the y_2 axis (right vertical) and eventually, the total execution time in the x axis (horizontal).
- Figure 4 – 3 presents differences between the current and the previous execution time of the program versus the total execution time. In order to understand better what this diagram illustrates, it is necessary to focus on the following facts:
  - The triangular voltage is produced using two for loops as the Figure 4 – 4 shows.

```
for (iterations = 0; iterations < MAX_NO_ITERATIONS; iterations++)
   for (VL = 0; VL < MAX_VOLTAGE_SCALE; VL++)
```

Figure 4 – 4: The two for loops in the Program

The first for loop produces the number of iterations of one period of

the waveform, i.e. how many periods of 3 – 0 – 3 volts will be produced in one execution of the program. The default iterations are 8, thus typically when the "for loop" finishes its execution, the virtual triangular voltage will have done 8 periods.
The second for loop, which is inside the previous for loop, produces the steps by which the triangular voltage changes its values, in order to create one period. These steps were predefined to be 32 in a single period of the triangular voltage. Hence, the triangular voltage steps are as follows:

$$Vstep = \frac{3\,Volts}{32\,Steps} = 0.09375\,Volts \cong 0.094\,Volts$$

Therefore, the first step of change is

$$Vnew = 3\,Volts - Vstep = 2.91\,Volts$$

In the same way all subsequent steps can be constructed until the triangular voltages returns to 3 volts.

- When the previous processes are completed, the program stores all the measured data, along with its stop execution time, in the noThreads.csv file in the SD card module terminates and then it goes again. When the program terminates once more, it stores the new execution time of the program. The difference between the new execution time and the previous execution time, gives the first point in Figure 4 – 2. When this process is repeated for all data points, Figure 4 – 2 is produced.

Figure 4 – 3 reveals an upward trend in the difference of the total execution time of the program. This is due to the gradual increase of the recorded data on the SD card module, as the process of the Arduino for data storing in SD Card (opening of the file, finding the last storing, storing the new data, closing of the file), creates an increasing delay of the execution time of the program (see Table 7). In conclusion, we notice that the hardware of the Arduino (the microprocessor and the varying response times of the Arduino peripherals) is influencing the execution time, and thus time behaviour of the program. In real-time systems, this variance could be unacceptable.

3. Status Error tab Diagram.
   In Figure 4 – 5 we examine if the relay module is ever at a wrong state, as mentioned in Section 4.1.

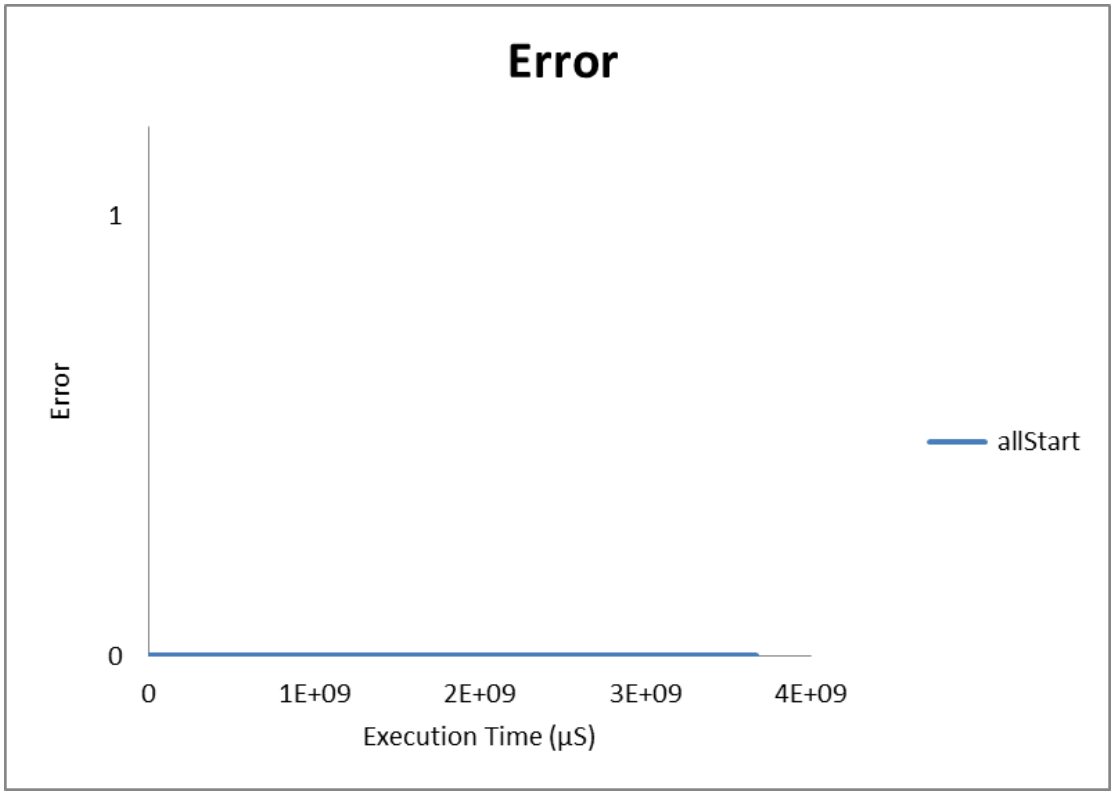


Figure 4 – 5: Status Error of the Relay Module

The x axis corresponds to the total execution time of the program (1 hour) and the y axis corresponds to the possible error status of the Relay. It is emphasized that if the relay has a wrong status while the program executes, it will return the value of 1. Otherwise, it will return the value of 0.
By observing the Figure 4 – 5, we notice that the relay does not have any wrong state for the whole duration of the execution, thus the program behaves consistently.

4. RelayOn Delay Tab Diagram.
   In Figure 4 – 6, we show the "**Variation RelayOn Sorted**" Diagram which illustrates the time duration of the `relayOn()` function, in comparison with the start time of the same function. The duration time of this function in µS is computed via the difference between its stop execution time minus its start execution time.

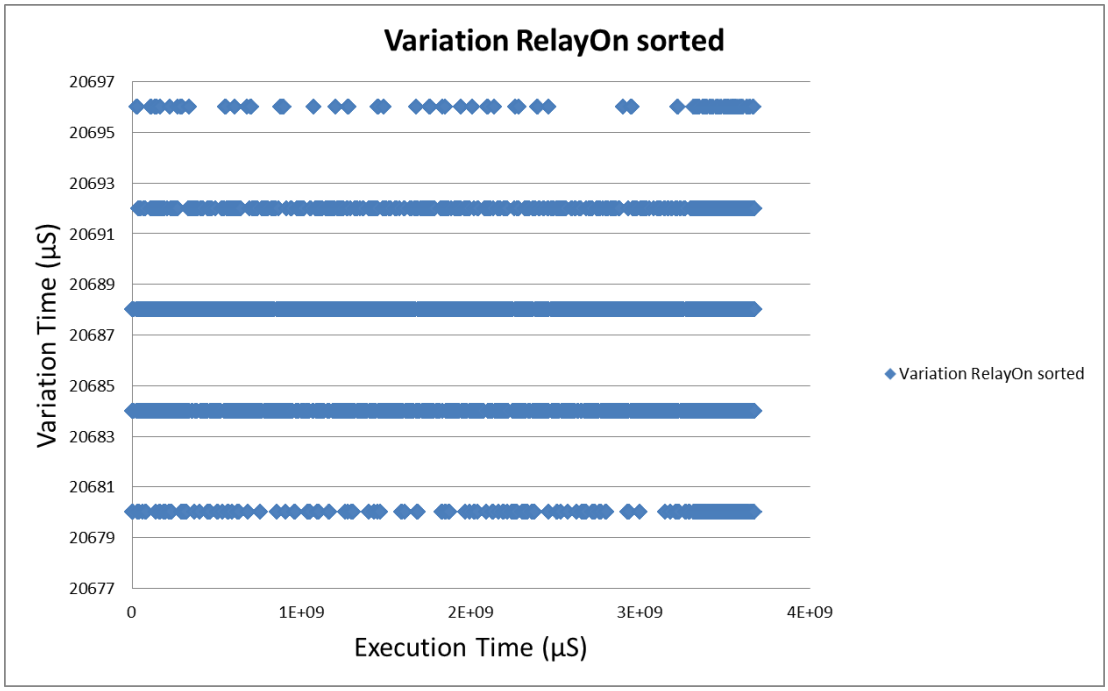


Figure 4 – 6: Variation RelayOn Sorted Diagram

Analysing this diagram, there is a slight variation of the duration of the execution time of the `relayOn()` function which varies from 0 to 16 µS. Possible reasons for this variation are non-deterministic response rate of the relay module or of the microcontroller. Also another reason for this behaviour might be arithmetic rounding of timing values when they come to many decimal digits.
However, there is no increase of the runtime of the function as happened in the Figure 4 – 3. This is an indication that the SD card module is the cause for the increase in the runtime of the program that generates the triangular voltage sweep during the 1 hour execution.

5. RelayOff Delay Tab Diagram.
   In Figure 4 – 7, the "**Variation RelayOff Sorted**" is presented. This figure illustrates the time duration of the relayOff() function, in comparison to the start time of the same function.

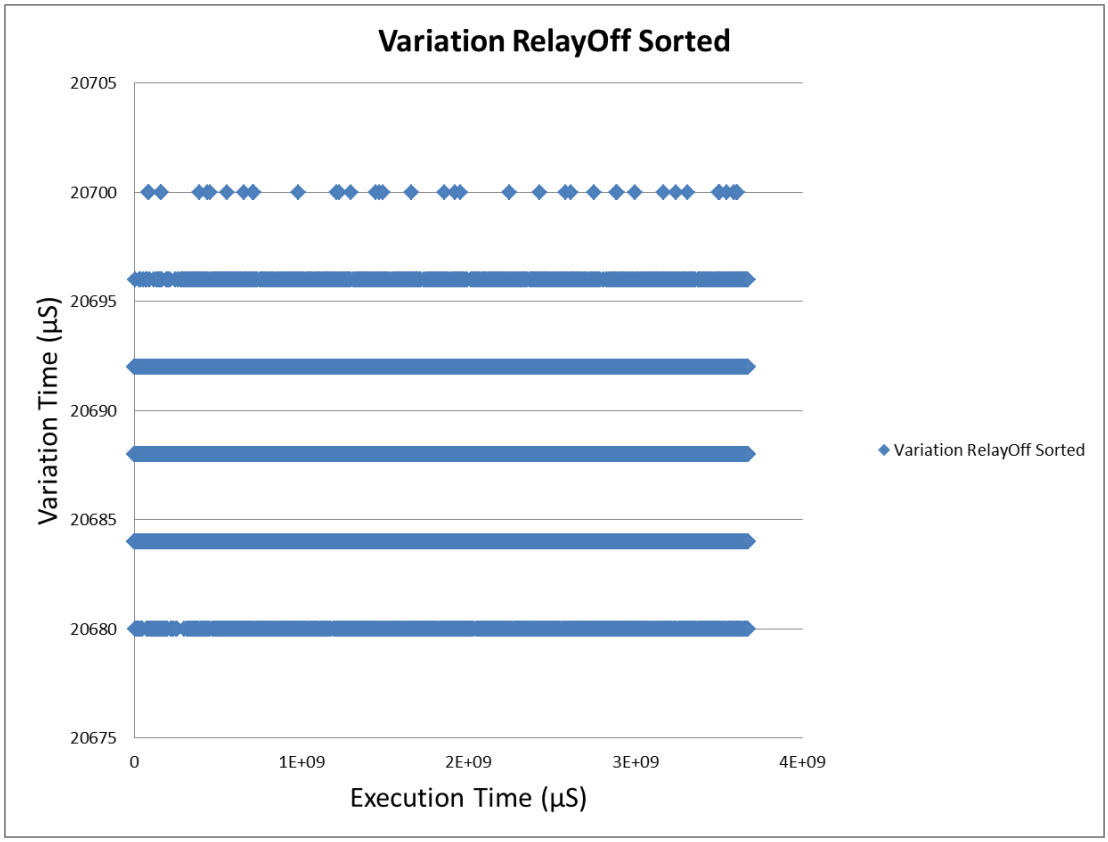


Figure 4 – 7: Variation RelayOff Sorted Diagram

Notice that there is slightly greater variation of the runtime of the `relayOff()` function, which is about 20 uSec, in relation to the variation in Figure 4 – 6.

4.2.2 NoSync.csv file's diagrams

We proceed in a similar way with the noThreads.csv file.

1. Voltage and Error Tab Diagrams (Figures 4 – 8 and 4 – 9).
   Figures 4 – 8 and 4 – 9 are similar to Figures 4 – 2 and 4 – 3, and thus the remarks are also similar to the ones in Section 4.2.1.

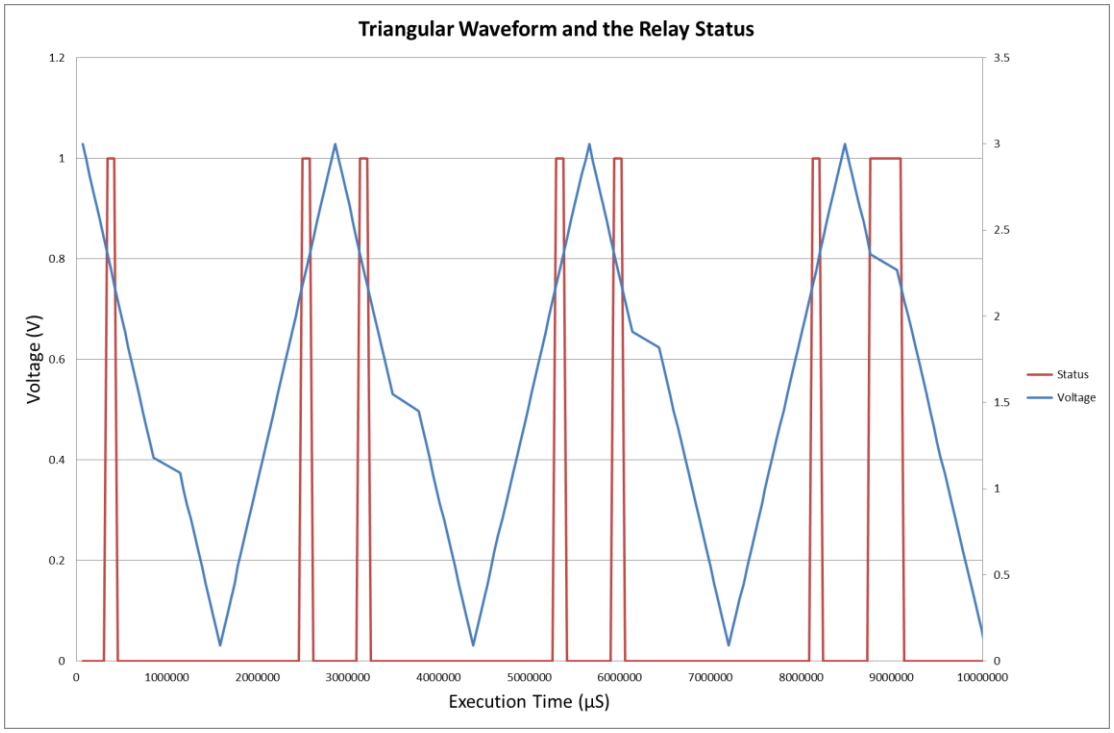


Figure 4 – 8: Virtual Triangular Waveform and the Relay Status

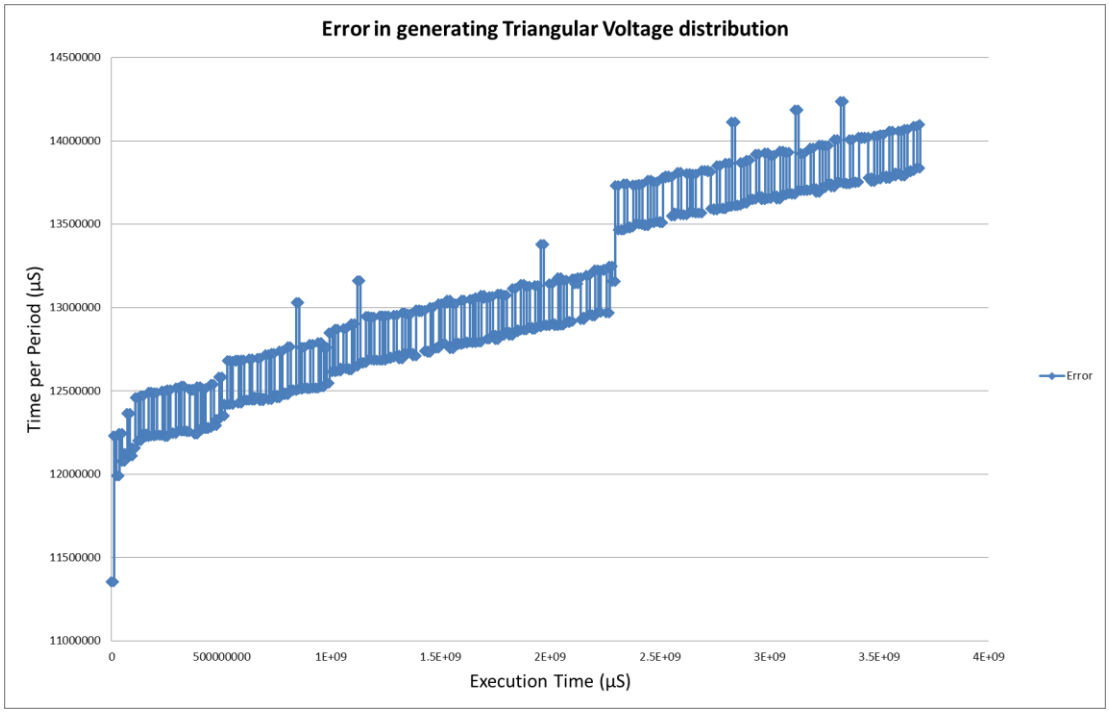


Figure 4 – 9: Error in Generating Triangular Voltage Distribution

2. Status and Error Tab Diagram.
   Figure 4 – 10 is similar to Figure 4 – 5, and the remarks are also similar to Section

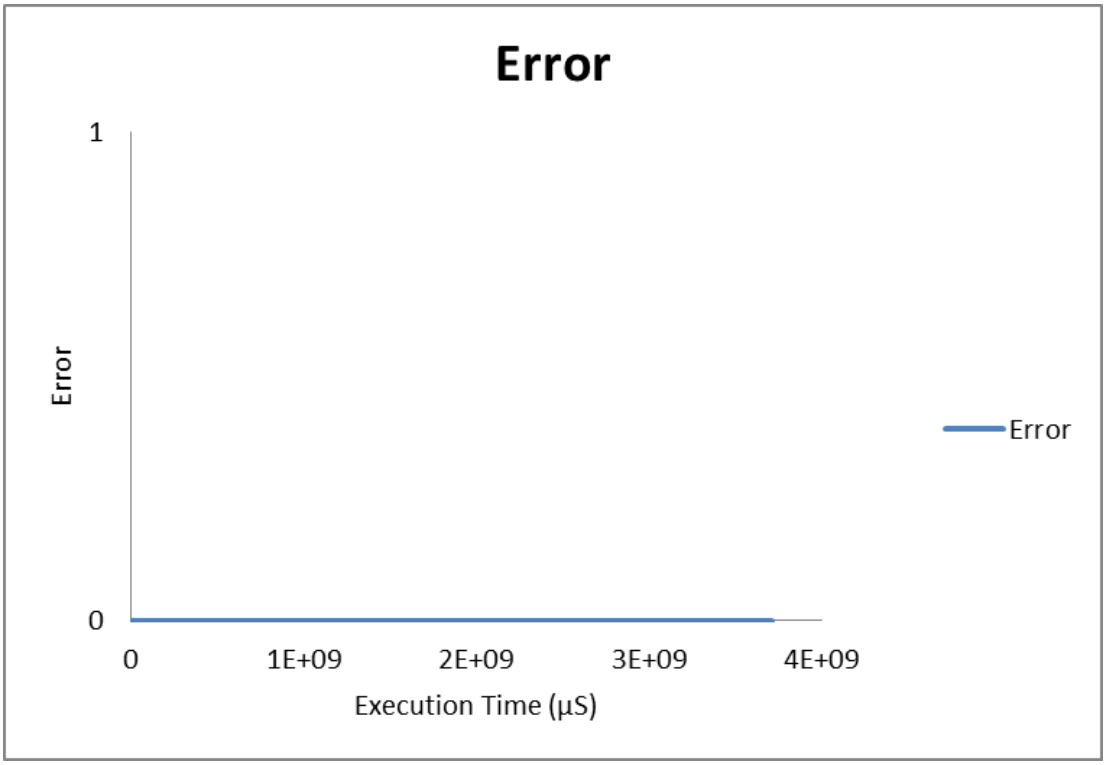


Figure 4 – 10: Status Error of the Relay Module

3. RelayOn Delay Tab Diagram.
   Figure 4 – 11 presents the Variation in the time duration of the `relayOn()` function, in comparison with the start time of the same function.

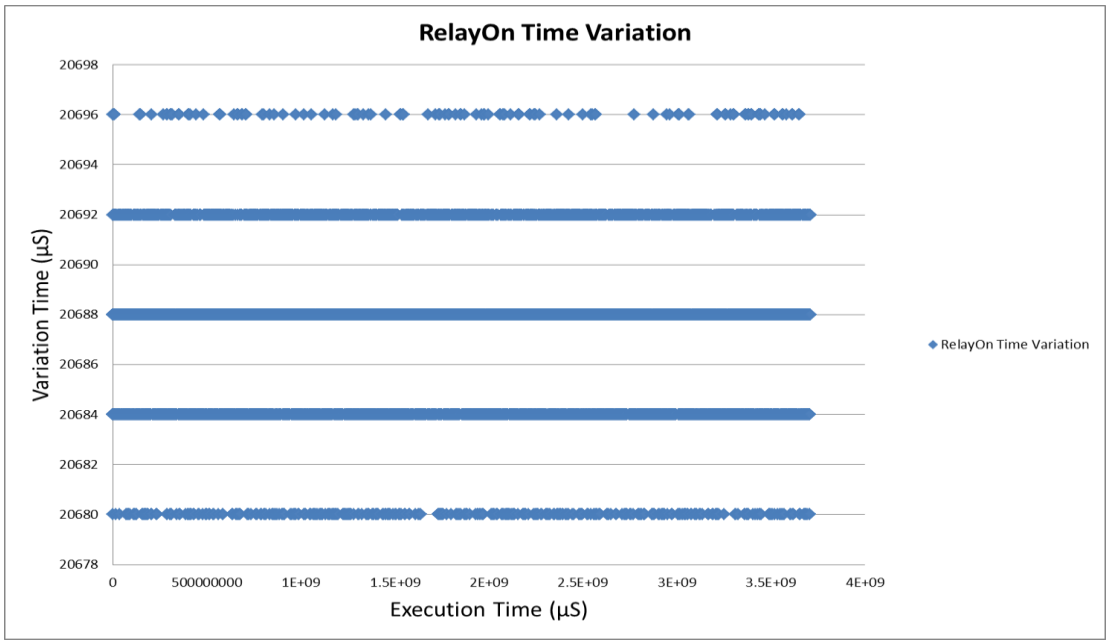


Figure 4 – 11: Variation RelayOn Sorted Diagram

The variation of the duration of the execution time of the `relayOn()` function is

from 0 to 16 μS. The reasons are similar to those of Figure 4 – 6.

4. RelayOff Delay Tab Diagram.
   Figure 4 – 12 presents he variation of the `relayOff()` function, in comparison with the start time of the same function. The same remarks and conclusions can be made for this diagram, as Figure 4 – 7 of the noThreads.csv file.

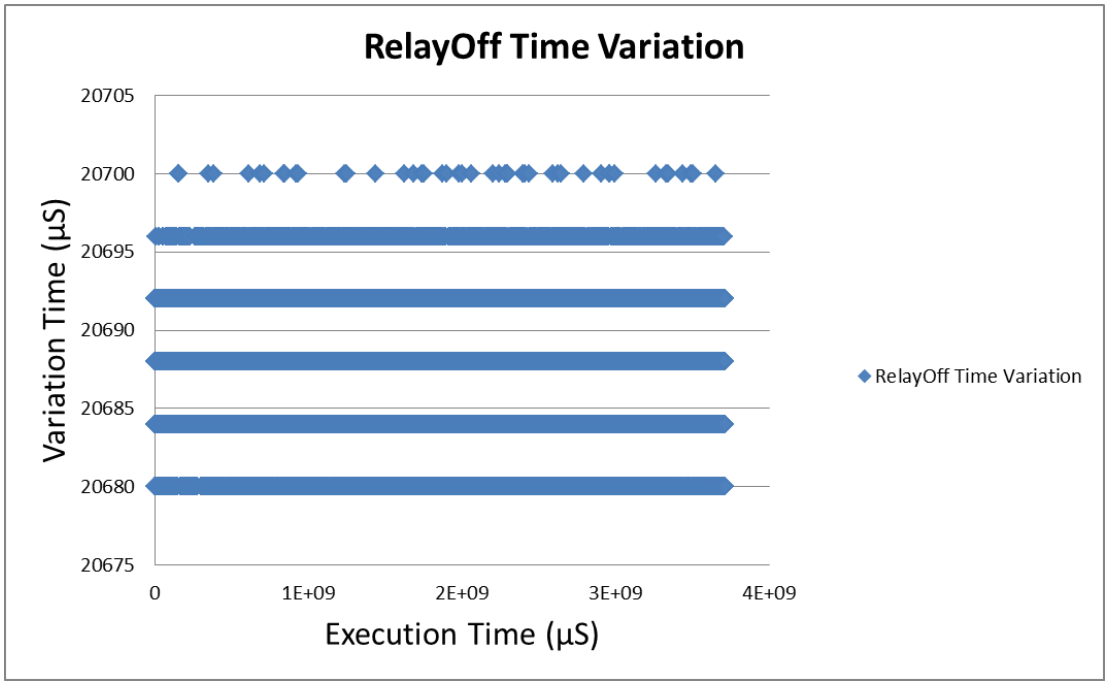


Figure 4 – 12: Variation RelayOff Sorted Diagram

4.2.3 syncFile.csv file's diagrams

Most of the diagrams in this Section were the same as previous ones, but there were also some differences between them, which are explained below.

1. Voltage and Error Tab Diagrams.
   Figures 4 – 13 and 4 – 14 are similar to the ones in Sections 4.2.1 and 4.2.2, except that the differences in timing stem from the synchronization between threads.

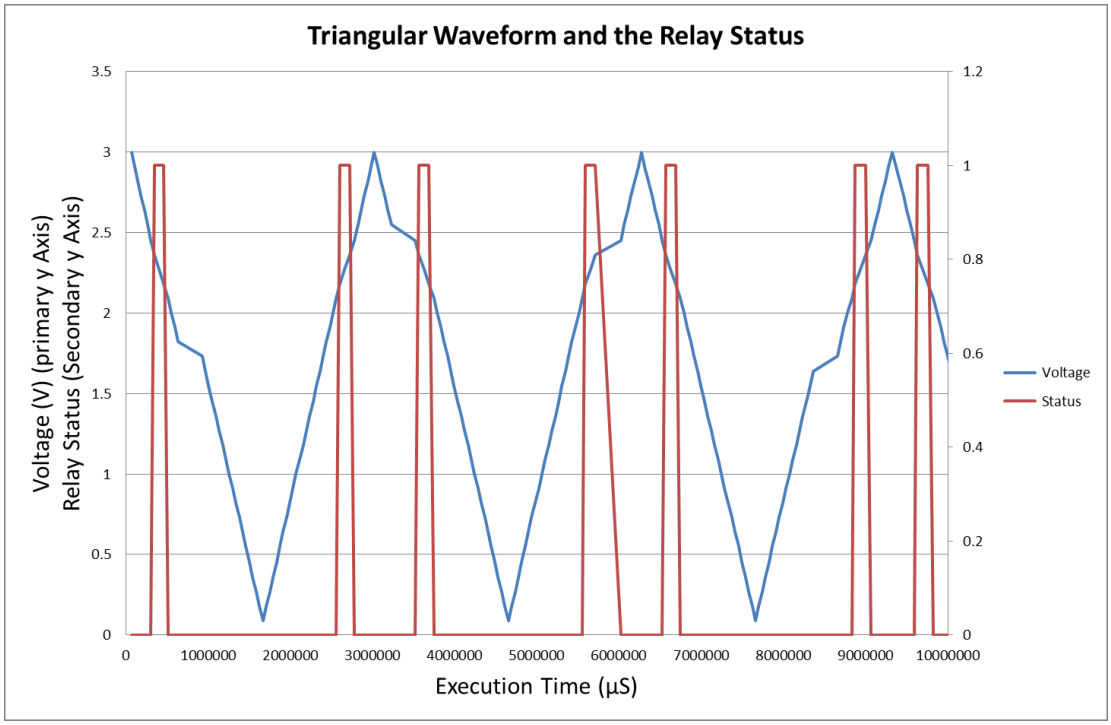


Figure 4 – 13: Virtual Triangular Waveform and the Relay Status

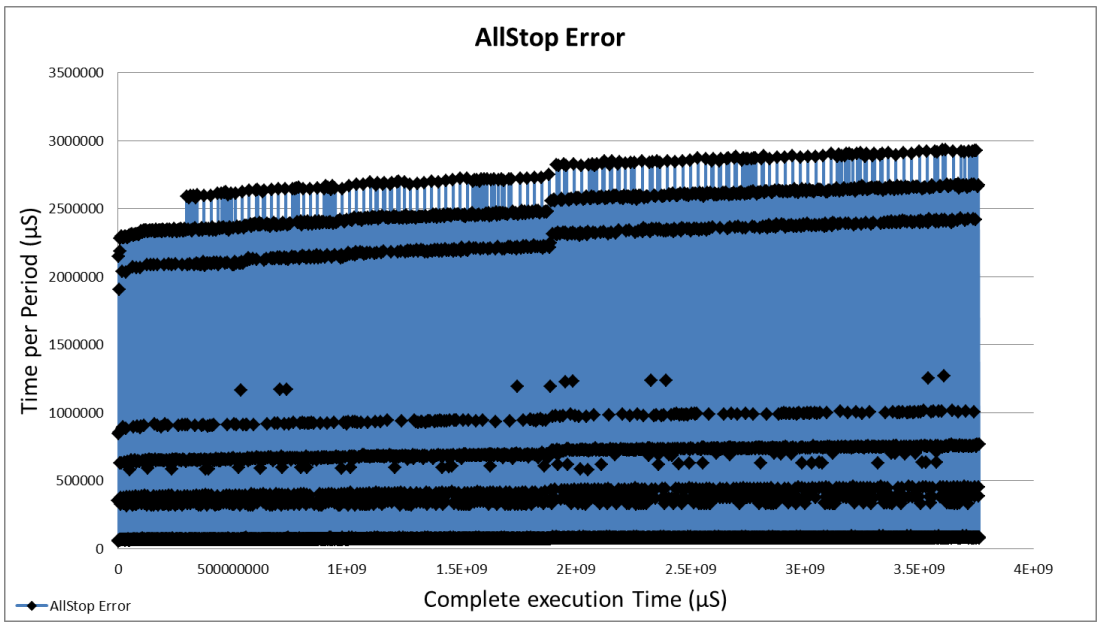


Figure 4 – 14: Error in Generating Triangular Voltage Distribution

A focused "Error in Generating Triangular Voltage Distribution" graph is presented next (Figure 4 – 15).

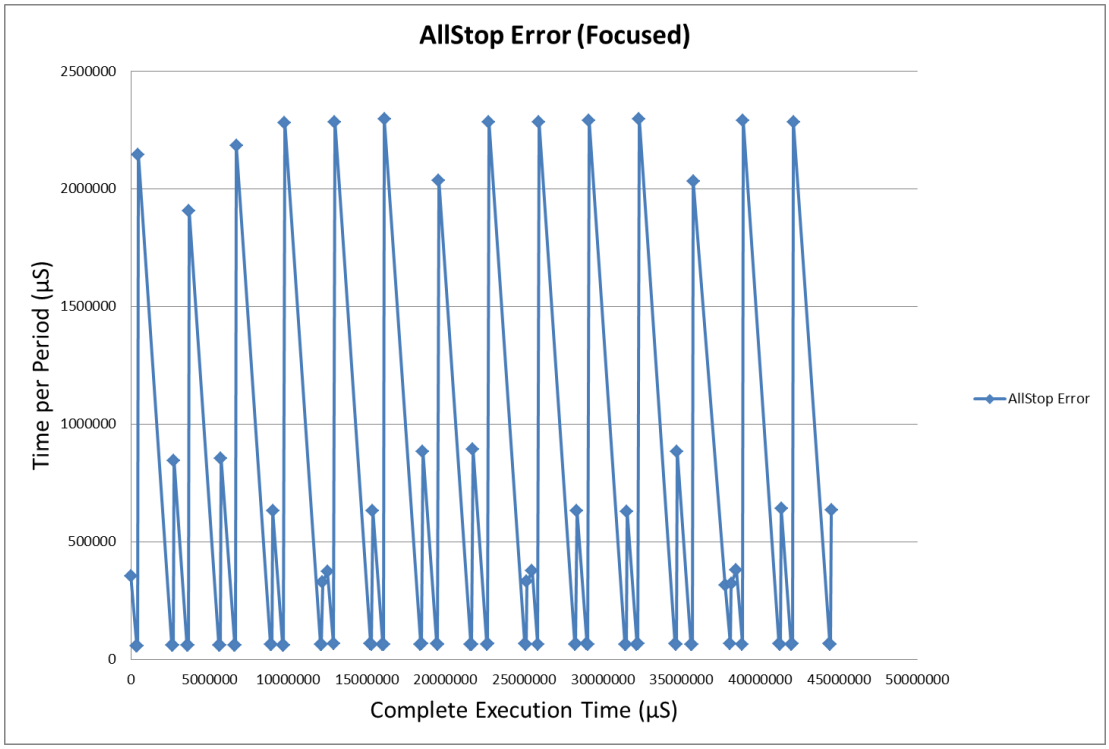


Figure 4 – 15: Error in Generating Triangular Voltage Distribution (Focused Graph)

In Figure 4 – 14 we illustrate the t_1, t_2, t_3 and t_4 time limits as they were described in Section 4.1. As shown in Figure 4 – 16a, the t_1 and t_4 time limits, correspond to 3 – 2.4 Volts (t_1) and 2.4 – 3 Volts (t_4) of the virtual triangular voltage (orange highlighted area), i.e. two intervals equal in duration.

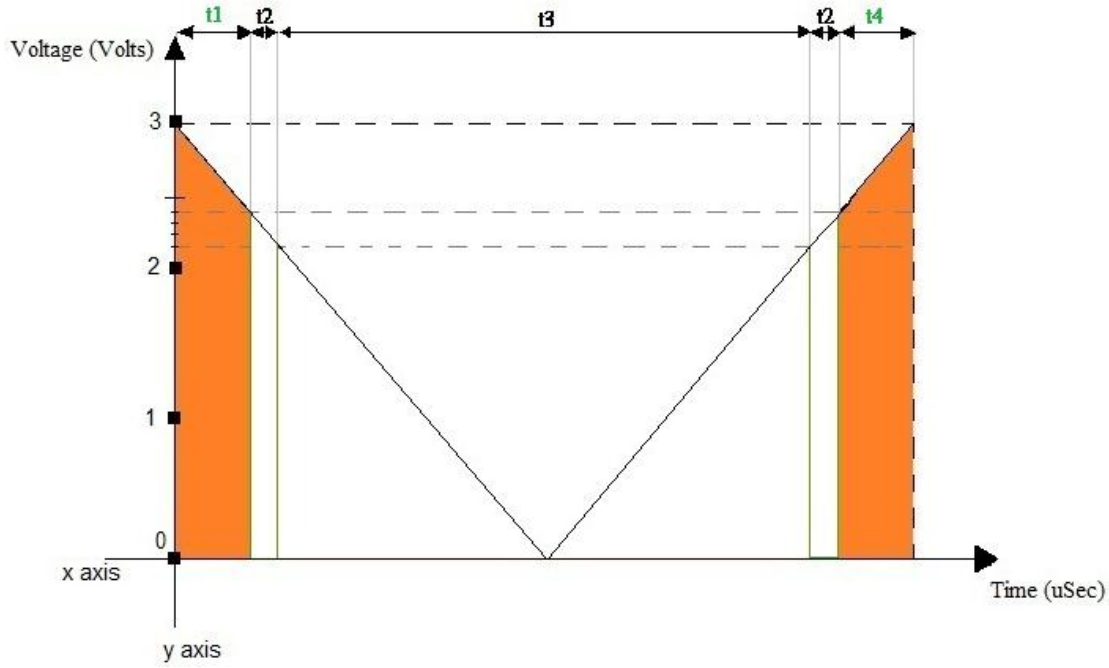


Figure 4 – 16a: t_1 and t_4 Time Limits

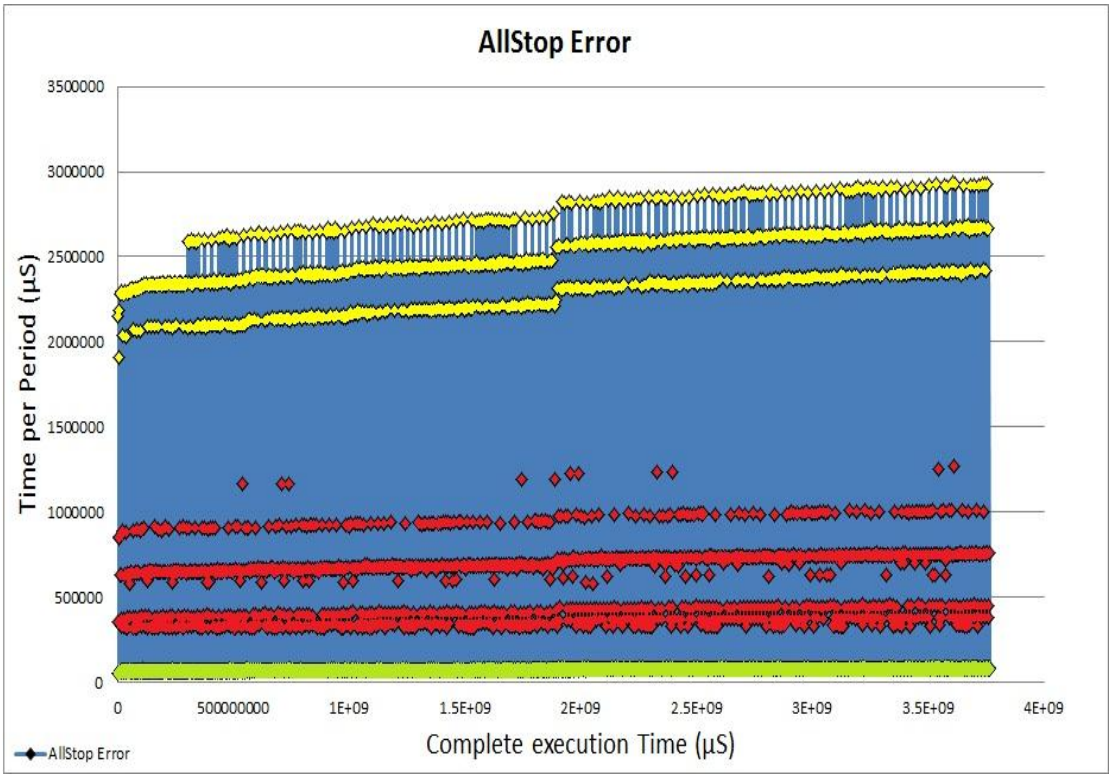


Figure 4 – 16b: Time limits highlighted with different colors

Figure 4 – 16b illustrates the time limits which were described in Section 4.1, as they were depicted by the experiment in this graph, but emphasized with different colour for each time limit. In more details, the **green spots** of the curve represent the **t_2 time limit**, the **red spots** the **t_1 and t_4 time limits** and finally the **yellow spots** represent the **t_3 time limit**.

As in the other two programming models, we notice that there is a gradual increase of the program execution time during the complete system execution time of 1 hour, which is due to storing an increasing amount of data on the SD card module. This increase is compared to the other programming models in Table 7, cf. next section.

- t_1 and t_4 Time Limits (red spots on the 4 – 16b graph)
  These time limits essentially concern the waiting time of the program until the triangular voltage values are within the operation limits of the experimental device (i.e. between 2.1 – 2.4 Volts). This extra wait in the program results from use of the protothreads `PT_WAIT_UNTIL` command.

- t_2 Time Limit.
  As shown in Figure 4 – 16c, the t_2 time limit corresponds to the operating range of the experimental device, which is 2.4 – 2.1 Volts and 2.1 – 2.4 Volts of the triangular voltage (orange highlighted area). Therefore, this time limit is essentially the execution time of the program, without any wait from synchronization. In simple terms, in this time interval, the program executes in parallel both threads in every step of the triangular voltage and checks all

parameters for the correct operation of the device (i.e. sensor values). If everything is in order, the program terminates and starts again, doing the same process, until the operation limits of the triangular voltage are **out of the working range**, or the sensor outputs eventually report an **unacceptable value**. In the latter, the program enters again in standby mode by calling `PT_WAIT_UNTIL`.

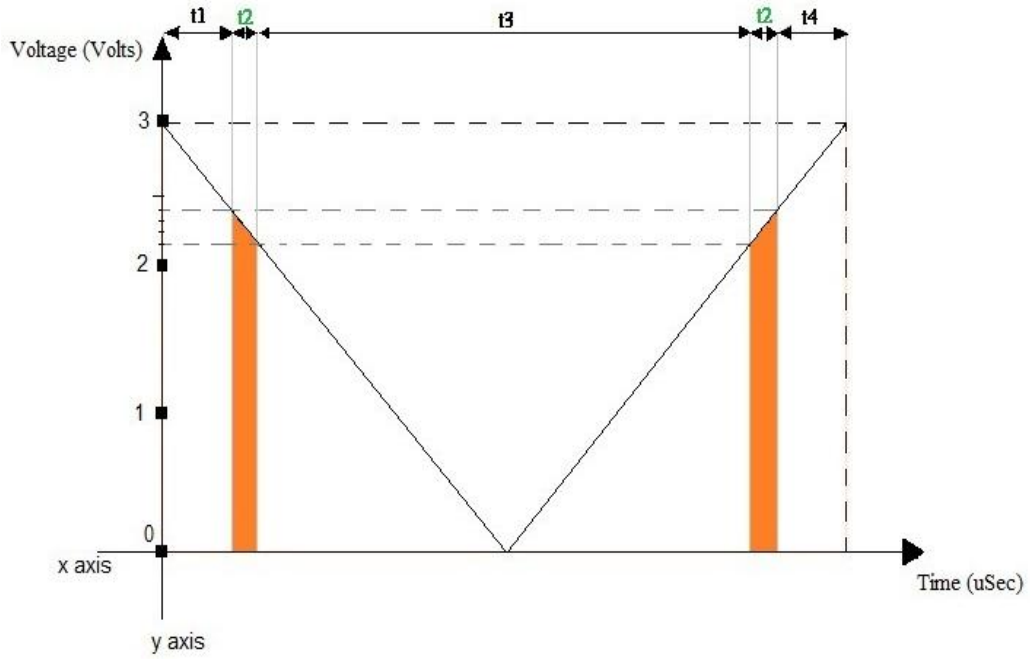


Figure 4 – 16c: t_2 Time Limit

- t_3 Time Limit.
  As it is shown in Figure 4 – 16d, the t_3 time limit corresponds to 2.1 – 0 – 2.1 volts of the triangular voltage (Orange highlighted area).

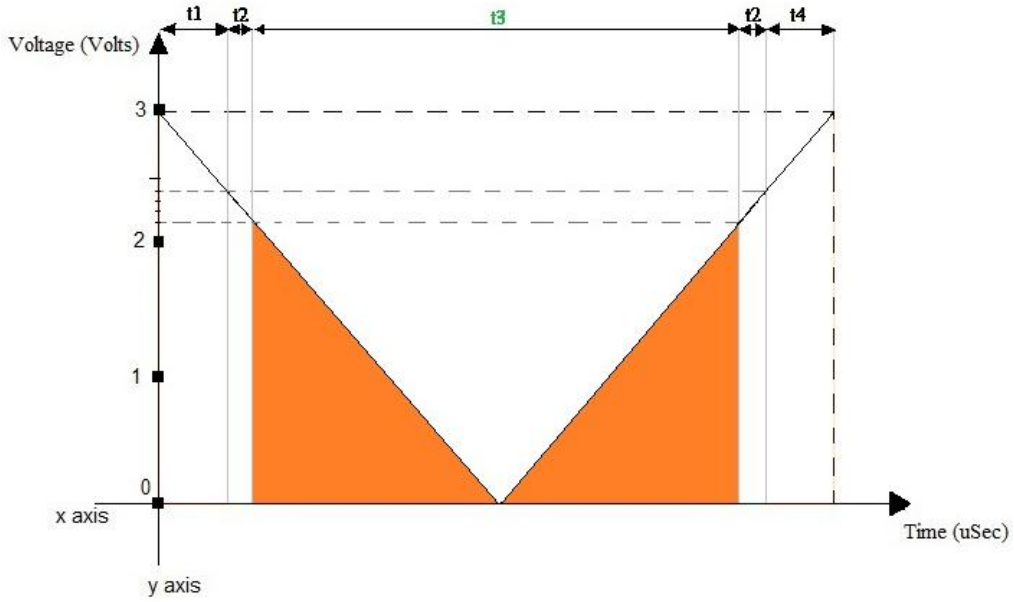


Figure 4 – 16d: t3 Time Limit

The t_3 time limit concerns another waiting time of the program until the triangular voltage's values are within the operation limits of the experimental device. This program wait also results from use of the protothreads `PT_WAIT_UNTIL` command.

2. Status and Error tab Diagram.
   Figure 4 – 17 reveals consistent operation, as mentioned in Section 4.1.

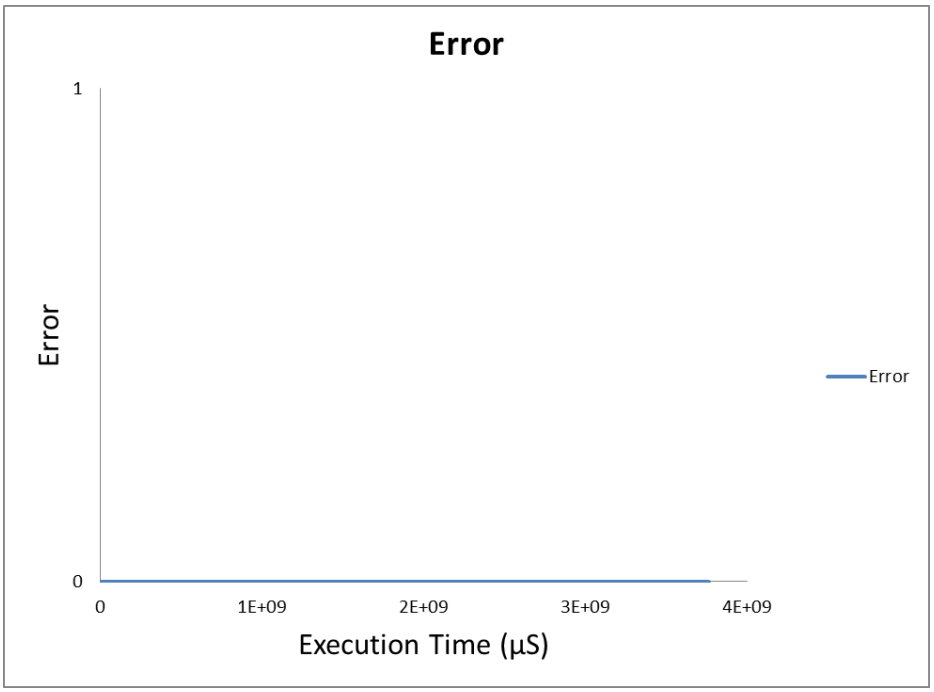


Figure 4 – 17: Status Error of the Relay Module

3. RelayOn Delay Tab Diagram.
   Figure 4 – 18 presents a variation in the time duration of the `relayOn()` function in comparison to the start time of the same function. The variation of the duration of the execution time of the `relayOn()` function is again from **0 to 20 µS** and the reasons for this variation have been described before.

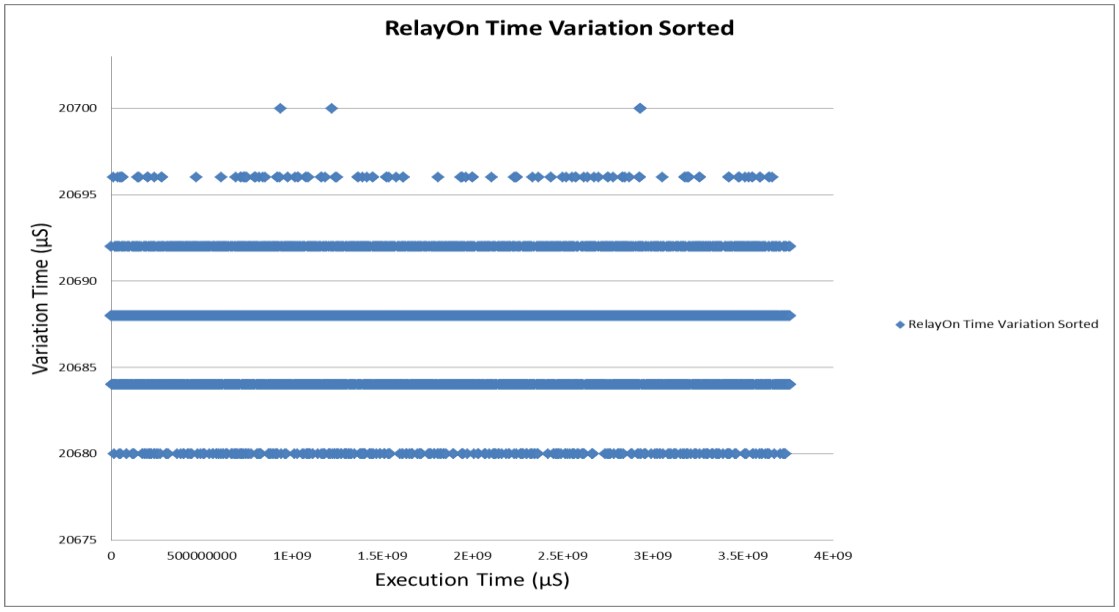


Figure 4 – 18: Variation RelayOn Sorted Diagram

4. RelayOff Delay Tab Diagram.
   In Figure 4 – 19, we show variation of the time duration of the `relayOff()` function, in comparison with the start time of the same function. The same remarks and conclusions can be drawn as previous Sections.

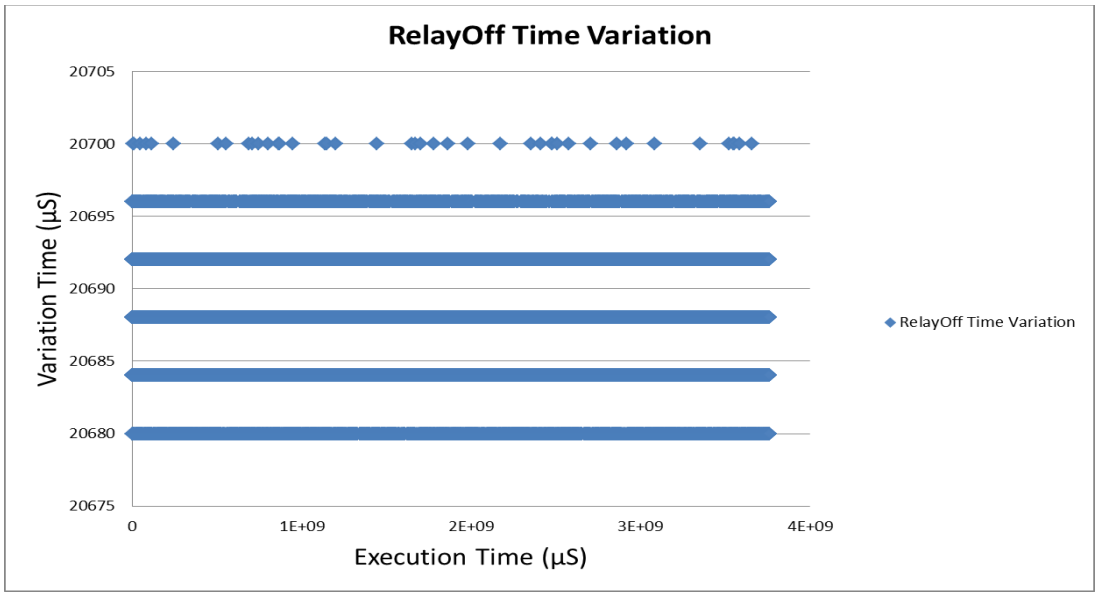


Figure 4 – 19: Variation RelayOff Sorted Diagram

4.2.4 Comparing the Results

After the data analysis made in the previous sections, we focus on comparing results of the three programming models against each other, in order to make conclusions and remarks about how parallelism affects program execution time on an Arduino microcontroller. As

foretold, the "**Models Comparison.xlsx**" file has been created to process comparison data.

1. Voltage and Error Comp. Tab.

   In this tab, we have combined the "**Error in Generating Triangular Voltage Distribution**" graphs of the three programming models into Figure 4 – 20.

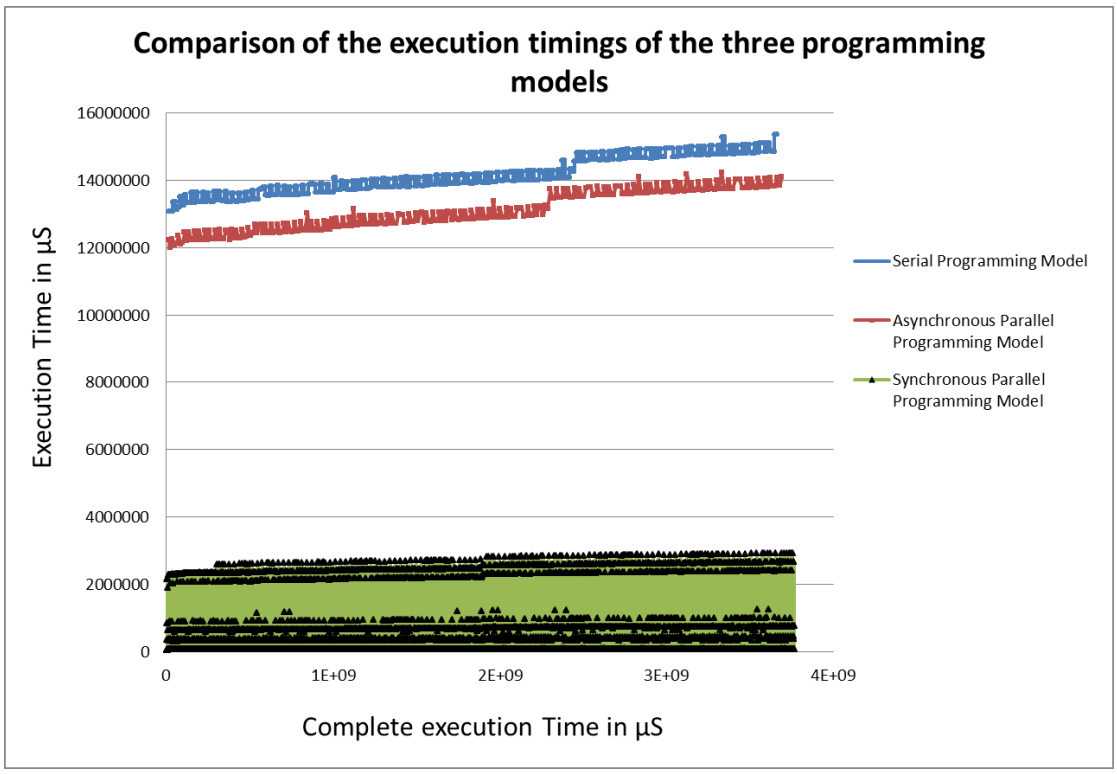


Figure 4 – 20: Comparison of Execution Timings of the Three Programming Models

Figure 4 - 20 shows variation in execution speed for each programming model during system operation of 1 hour. While the Asynchronous Parallel programming model has almost the same behaviour as the Serial programming model, it is slightly faster, while the Synchronous model is always much faster. There is also a tendency for the execution time to increase, which is most likely due to delay in SD card data logging as more data is recorded (see Section 4.2.1). In real-time systems, this variance could be unacceptable.

Average Execution Time (µS)	
Serial Model	14138110.0
Asynchronous Parallel Programming Model	13088806.0
Synchronous Parallel Programming Model	593082.2

Table 5: Average Execution Time (per loop) of the Three Programming Models

Relative Speedup vs. Serial Model	
Asynchronous Model	1,08
Synchronous Model	23,84

Table 6: Speedup of Parallel Programming Models

Table 5 shows the average execution time (per loop) of the programming models. We can see that the Parallel programming models are both faster than the Serial model. In fact, as Table 6 shows, the Asynchronous Parallel programming model has a small speedup factor (1.08), while the Synchronous model achieves a significant speedup (23.84) over the Serial model. This big improvement is due to synchronization between the two threads of the program, i.e. use of `PT_WAIT_UNTIL`. Notice that if the maximum number of iterations currently set to 8 (cf. `MAX_NO_ITERATIONS` in lines 6 (7 or 7) in the sequential (resp. asynchronous and synchronous) is decreased to a bare minimum of 1, the relative speedup decreases to 5.64 (instead of 23.84). However, the overhead of the sequential and asynchronous implementations can never be smaller than 5.64, without the use of proper lightweight thread synchronization mechanisms.

Execution Time Variation for each Model (μS)		
Serial Model	**complete execution**	1796280
Asynchronous Parallel Programming Model	**complete execution**	1868424
Synchronous Parallel Programming Model	**t_2 Time Limit**	35184
	t_1 + t_4 Time Limits	951564
	t_3 Time Limit	1027708
	Total	2014456

Table 7: Execution Time Variation for each model

Table 7 shows gradual increase in the execution time (drift) for each model, by comparing duration of the last execution of the program with its first during the 1-hour experiment. As it can be easily observed in this table the execution time increases during the 1-hour experiment of the Synchronous Parallel programming model. For the synchronous model each specific time limit has been defined in a previous Section. The drift appears to be similar for all models.

2. TimePerPeriod Comp. tab.

   In this tab, Table 8 compares the average execution time per each period in the program for all three models. In addition, we compare how many times the program loop and the ldrSensor thread has been executed within the 1 hour system run.

Comparison of the Period of the Program and of the ldrSensor Thread			
Programming Models	**Average Time Per Period (μS)**	**No. Periods (loop)**	**No. Periods (ldrVoltage)**
Serial Programming Model	3606568.318	1019	260
Asynchronous Parallel Programming Model	3366347.819	1102	283
Synchronous Parallel Programming Model	3711874.690	1014	6345

Table 8: Comparison of the Periods of the Programs

By comparing the results we notice that:

a. The average time per period of the program is almost the same between the three models. The same is true for the number of periods (loop) executed in the program.

b. Concerning the Serial and Asynchronous Parallel programming models, we notice that the ldrSensor thread runs almost the same number of times, while for the Synchronous Parallel programming model we notice a huge increase of the execution of the ldrSensor thread. This means that the ldrSensor thread was executed 24 (or 22) times more than the Serial programming model (resp. asynchronous model). This indicates that if we use synchronization in a parallel program based on protothreads, in the same time in which this program needs to run in serial mode, the synchronous parallel programming model executes much more updates of the ldrVoltage. Therefore, better system monitoring and control can be achieved.

3. RelayOn Delay Comp. tab

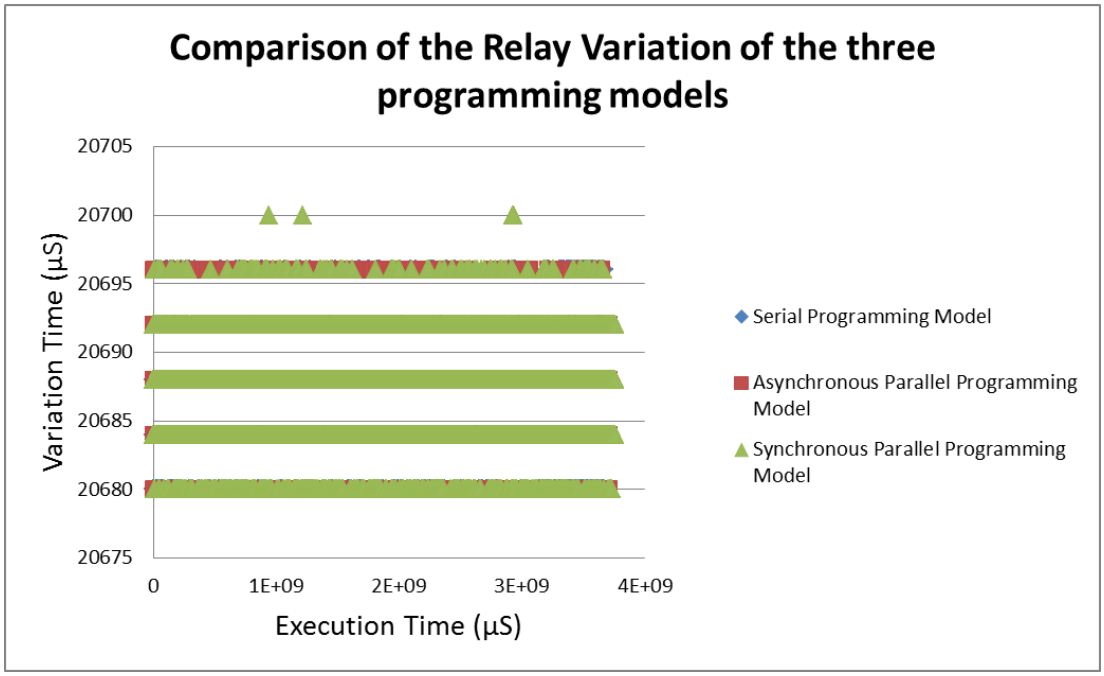


Figure 4 – 21: Comparison of RelayOn Variation in the three programming models

In this tab we examine if the `relayOn()` function runs in the same way in the three programming models. As we notice in Figure 4 - 21, the `relayOn()` function runs in the same way in the three programming models.

4. RelayOff Delay Comp. tab

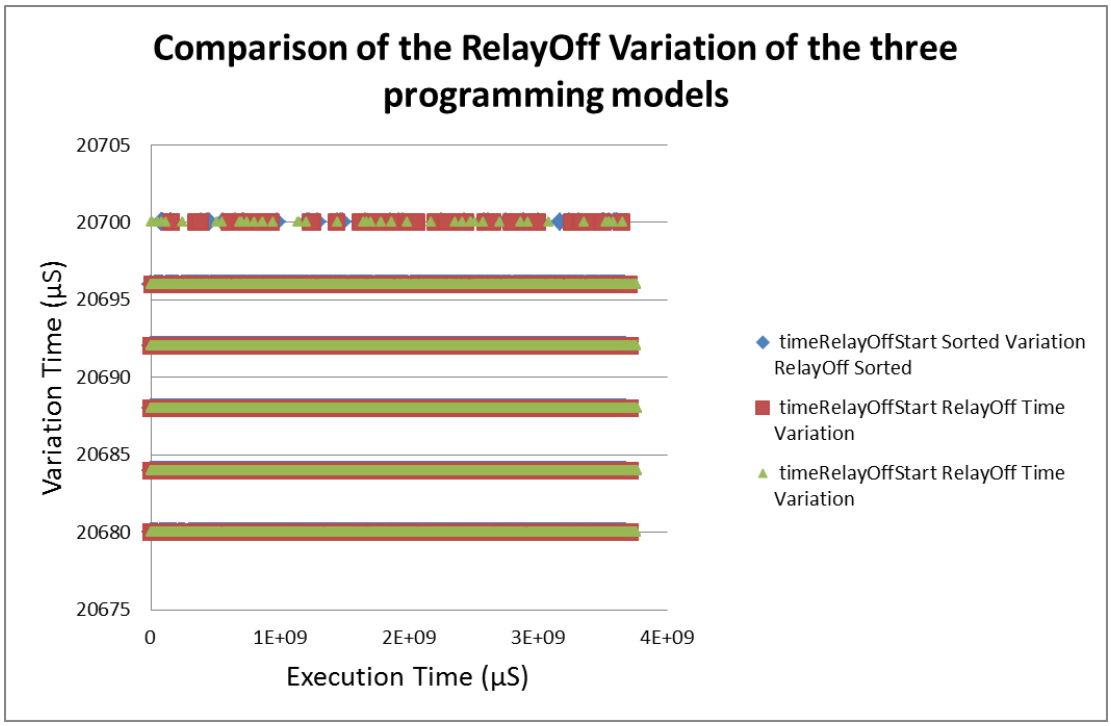


Figure 4 – 22: Comparison of RelayOff Variation in the three programming models

Similarly to the `relayOn()` function, we examine if the `relayOff()` function runs in the same way in the three programming models. As we see in Figure 4 - 22, the `relayOff()` function runs in almost the same time for all three programming models.

5. Future Work

Through the experimental study and analysis in this thesis, we have studied and evaluated the execution time behaviour of three different programming models (serial, asynchronous parallel and synchronous parallel programming) on an Arduino board design focusing on energy monitoring. In order to make accurate comparisons among the three programming models, all designs use the same monitors, equivalent execution environment, as well as identical control flow. Moreover, during experimentation, all designs compute the same performance metrics.

The objective of this study is to demonstrate experimentally whether a parallel programming model based on lightweight protothreads is as reliable as the serial programming model, but faster.

During the course of this experiment some concerns were raised related to whether parallelism is achieved with protothreads while the asynchronous programming model is running, or not. More specifically, it was noticed that the asynchronous parallel programming model operates exactly as the serial programming model does. So, one of the results is that the protothreads library requiring further development in order that true parallelism can be achieved, even when there is no synchronization between the parallel execution of the threads.

Extensions of the protothreads library can consider synchronization and especially mutual exclusion of collected data. For example, when Thread1 attempts to increase the value of x by 1, at the same moment, Thread2 (which is faster than Thread1) for some reason, e.g. because of the input signal of a sensor, attempts to reduce x by 1, (while Thread1 is already attempting to do that), the x variable may result with an error value unless locks are used.

Along with execution differences in the programming models, we examined the way to manage their execution time. Specifically, by taking into consideration the execution speed of the synchronous programming model, one possible future extension of this experiment would be to use it in a real-time system environment. Possible ways to convert the current setup into a real-time system require reprogramming of the code depending on the time response of the device, the creation of new libraries based on interrupt signals, the simplification of both hardware and software of the device etc. These conversions would make possible the use of the device in an environment which has specific operation time-frames and deadlines. It can also be examined whether the Arduino board is reliable enough for use in commercial real time systems.

This is particularly important in cases that dangerous (hazardous) situations may arise during the execution of the program. In this case, the code must safeguard and protect the human factor and equipment from potential damage or injuries. Therefore, an additional future extension of this application would be the ability to predict these conditions. In other words, it is necessary to design smart devices that will have the ability to decide what action to take, according to data received from sensors, in combination with a real-time library, in order to act immediately, before there is any damage or injury.

6. References

[1] http://en.wikipedia.org/wiki/Apollo_Guidance_Computer - Apollo Guidance Computer
[2] http://www.fullchipdesign.com/verilog_tutorial.htm - Verilog
[3] http://en.wikipedia.org/wiki/VHDL - VHDL
[4] http://embedded.eecs.berkeley.edu/research/hsc/class/ee249/lectures/l10-SystemC.pdf - System C
[5] http://www.informit.com/articles/article.aspx?p=1352549&seqNum=2 - Predictability and determinism
[6] http://en.wikipedia.org/wiki/Wright_brothers - Wright Brothers
[7] http://searchcio-midmarket.techtarget.com/definition/transducer - Transducers
[8] http://arduino.cc/ - Arduino Official Site
[9] http://arduino.cc/en/Main/arduinoBoardUno - Arduino Uno Rev 3
[10] http://www.funduino.cn/ - Funduino
[11] http://www.sainsmart.com/ - Sainsmart
[12] http://arduino.cc/en/Tutorial/PWM - PWM
[13] http://web.engr.oregonstate.edu/~traylor/ece473/lectures/twi.pdf - TWI
[14] http://www.trossenrobotics.com/c/arduino-sensors.aspx - List of Arduino Sensors with their price
[15] http://el.wikipedia.org/wiki/Arduino - Arduino IDE Wikipedia
[16] http://en.wikipedia.org/wiki/Cross-platform - Cross-platform applications
[17] http://playground.arduino.cc//Linux/OpenSUSE - Installing Arduino IDE on OpenSUSE
[18] http://arduino.cc/en/Main/Software - Download link of the Arduino IDE
[19] http://man7.org/linux/man-pages/man2/fork.2.html#section_dir – fork()
[20] http://man7.org/linux/man-pages/man2/clone.2.html#section_dir – clone()
[21] http://man7.org/linux/man-pages/man2/clone.2.html#COLOPHON - Further information about Linux Threads
[22] https://computing.llnl.gov/tutorials/pthreads/ - Pthreads
[23] http://www.yolinux.com - Pthreads
[24] http://www.cs.cf.ac.uk/Dave/C/node30.html#SECTION00306200000000000000 - Scheduling Parameters
[25] Threads under specialized OS, http://www.tinyos.net/tinyos-2.x/doc/html/tep134.html - TOSThreads http://www.cs.berkeley.edu/~prabal/teaching/cs294-11-f05/slides/day8c.pdf MANTIS OS
[26] Thread libraries under Linux, http://tinythreadpp.bitsnbites.eu
[27] http://dunkels.com/adam/ - Adam Dunkels CV
[28] https://www.sics.se/media/news/adam-dunkels-receives-prestigious-chester-carlson-prize - Swedish University of Computer Science
[29] http://dunkels.com/adam/pt/ - Adam Dunkels work: Protothreads
[30] Dunkels A. Schmidt O. Voigt T. *Using Protothreads for Sensor Node Programming*, 2005
[31] http://dunkels.com/adam/pt/expansion.html - How protothreads really work
[32] http://en.wikipedia.org/wiki/Duff's_device - Duff's Device
[33] http://en.wikipedia.org/wiki/Loop_unwinding - Loop Unwinding
[34] http://www.lua.org/pil/9.html - Coroutines
[35] http://en.wikipedia.org/wiki/Coroutine - Coroutines definition: Coroutines wiki
[36] Goswami A. Bezboruah T. Sarma K.C. *Design of An Embedded System For Monitoring and Controlling Temperature and Light*, Research India Publications, India, 2009
[37] http://processing.org/ - Processing
[38] http://plasma-gate.weizmann.ac.il/Grace/ - Grace
[39] https://processing.org/download/?processing - Download Processing
[40] http://en.wikipedia.org/wiki/WYSIWYG - WYSIWYG
[41] http://en.wikipedia.org/wiki/X_Window_System - X Window System

[42] http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html - Grace's User Guide
[43] http://plasma-gate.weizmann.ac.il/Xmgr/doc/commands.html - Grace's terminal commands
[44] http://www.tldp.org/LDP/lpg/node15.html - Named pipe
[45] Capehart B. Turner W. Kennedy W. *Guide to Energy Management Fourth Edition,* The Fairmont Press, Inc. United States of America, 2003
[46] http://www.ihu.edu.gr/gateway/expertise/research-areas/energy_management.html - Definition of Energy Management
[47] http://fritzing.org/home/ - Fritzing Program
[48] http://arduino.cc/en/Main/arduinoBoardMega2560 - Official site of Arduino Mega 2560

Pictures

[1 – 1]: http://www.engineersgarage.com/articles/embedded-systems
[1 – 2]: http://en.wikipedia.org/wiki/Apollo_Guidance_Computer
[1 – 3]: http://www.engineersgarage.com/articles/embedded-systems
[1 – 4]: http://www.engineersgarage.com/articles/embedded-systems?page=2
[1 – 5]: http://www.engineersgarage.com/articles/embedded-systems?page=2
[1 – 6]: http://www.engineersgarage.com/articles/embedded-systems?page=4
[1 – 7]: http://www.ercim.eu/publication/Ercim_News/enw52/donhoffer.html , http://www.thinkdefence.co.uk/2013/10/game-spydin-barcelona/
[1 – 8]: http://www.embedded.com/electronics-blogs/beginner-s-corner/4023859/Introduction-to-Real-Time
[1 – 9]: http://cert.ics.uci.edu/sesa2011/Schedule.html , http://www.sbbindia.com/relays.php
[2 – 1]: http://accitron.blogspot.gr/
[2 – 2]: http://arduino.cc/en/Main/arduinoBoardUno
[2 – 3]: http://arduino.cc/en/Hacking/PinMapping168
[2 – 4]: http://www.buyapi.ca/product/37-in-1-arduino-compatible-shield-mega-kit/ , http://www.wiregarden.org/
[2 – 5]: http://arduino.cc/en/Main/ArduinoEthernetShield
[2 – 7]: http://playground.arduino.cc/Main/WikiSandbox
[2 – 11]: http://en.wikipedia.org/wiki/Thread_(computing)
[2 – 12]: http://web.cs.miami.edu/home/gallo/CSC322/Content/UNIXProgramming/UNIXThreads.shtml
[2 – 13]: https://computing.llnl.gov/tutorials/pthreads/
[2 – 14]: https://computing.llnl.gov/tutorials/pthreads/
[2 – 15]: https://computing.llnl.gov/tutorials/pthreads/
[2 – 16]: https://computing.llnl.gov/tutorials/pthreads/
[2 – 17]: https://computing.llnl.gov/tutorials/pthreads/
[3 – 1]: http://www.neuralenergy.info/2011_08_01_archive.html
[3 – 2]: http://arduino.cc/en/Main/arduinoBoardMega2560
[3 – 3]: http://www.datasheetdir.com/ATMEGA2561+AVR-microcontrollers