

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
M'Hamed BOUGARA University – Boumerdes



Institute of Electrical and Electronic Engineering
Department of Power and Control

Final Year Project Report Presented in Partial Fulfilment of
The Requirements for the Degree of

MASTER

In Control

Option: Control

Title:

**Design and Implementation of PLC using
Arduino Due**

Presented by:

- **Dahmane Roumaissa**
- **Rabia Zohra**

Supervisor:

Dr.Ouadi

Registration Number:...../2018

Abstract

In this project we will present an Arduino based PLC, the PLC uses Arduino Due board as communication path between the input and output interfaces. A new extension of the Arduino framework was used; it introduces multitasking support and allows running multiple concurrent tasks in addition to the single execution cycle provided by the standard Arduino framework.

To design Programmable Logic Controller using Arduino there exist two approaches. The first one is to write the program in ladder logic then use compilers directly and compile it to Arduino board. The second approach is to create a plc library inside the Arduino libraries folder, the library that we implemented in this project take advantage of the digital I/O of the Arduino Due. The plc library was coded using c language; it comprises the Basic functions of the PLC respecting the CEI 61131-3 standards.

The development of PLC hardware has been designed and improved. This Arduino based PLC is embedded with I/O module such as normally open push buttons, 24VDC power supply and output interface of 5V Relays with LEDs as indicators. I/O field devices are connected using optocouplers installed in order to protect the PLC from any electrical damages in addition to the Darlington Sink Driver (ULN2803) as current amplifier since the current generated by the Arduino pins is so small to drive a relay.

Finally to test the functionality of our plc two applications were implemented, the first one is the single task operating conveyor and the second is dual-task Motor.

Acknowledgement

In the name of Allah, the Most Gracious and the Most Merciful Alhamdulillah, all praises to Allah for the strengths and His blessings in completing this dissertation.

We are grateful to the God for the good health and wellbeing that were necessary to complete this work.

We wish to express our sincere thanks and gratitude to our supervisor Dr.Ouadi, for providing us with all the necessary facilities.

We would like to express our deepest gratitude to him, for his excellent guidance, without his assistance and dedicated involvement in every step throughout the process; this Thesis would have never been accomplished

We would also like to thank all those people who have been associated with this work for their passionate participation and help because without it this project could not have been successfully conducted.

Dedication

This thesis is dedicated to:

The sake of Allah, my Creator and my Master,

My great teacher and messenger, Mohammed (May Allah bless and grant him), who taught us the purpose of life.

My beloved father who leads me through the valley of darkness with light of hope and support, who has been a source of encouragement and inspiration to me throughout my life, a very special thank you for working hard for me and for my brothers throw all the passing years, you have actively supported me in my determination to find and realize my dream.

My dearest mother, the source of unconditional love in my life. Who was my sister, my friend and my all. Today I can proudly say thank you mom for being there for me whenever I needed support, for listening to me whenever life became unbearable and so hard, thank you for showing me that a dream can be reachable with hard work and strong faith.

My aunt Mofida, my second mother. This great woman who cares for me and loves me as a daughter of hers and who taught me to be patient and strong;

My beloved brothers: kheireddine and Mustapha Amine, whom I can't force myself to stop loving. These two are the spices of my life and without then it would be meaningless. I wish them luck and success in their exams.

To all my family, the symbol of love and giving, particularly my three angles: Fatima, Abderrahmane and the new coming baby;

My friends who encourage and support me,

All the people in my life who touch my heart, I dedicate this work.

Dahmane Roumaissa

Dedication

This thesis is dedicated to:

The sake of Allah, my Creator and my Master,

first and foremost, I have to thank my mother for her love and support throughout my life I would like to thank her for giving me strength to reach for the stars, chase my dreams and own the best education so far.

This thesis is dedicated also to the best people I have in my life who always loved me unconditionally, Moez, Radja, Ikram;

Who have always been constant sources of support and encouragement during the challenges of my whole college life and my final year project, Also to my sister who was ready all time to give me help when I need it.

It is dedicated also to those who are good examples who have taught me to work hard for the things that I aspire to achieve

Zohra Rabia

Table of content	
<i>Abstract</i>	<i>II</i>
<i>Acknowledgement</i>	<i>III</i>
<i>Dedication</i>	<i>IV</i>
<i>Dedication</i>	<i>V</i>
<i>Table of content</i>	<i>VI</i>
<i>List of Tables</i>	<i>IX</i>
<i>List of Figures</i>	<i>X</i>
<i>General Introduction</i>	<i>XIII</i>
<i>Chapter one: PLC Overview</i>	<i>1</i>
1.1 Introduction	1
1.2. Definition	1
1.3. Historical Background	2
1.4. Hardware Components	3
1.4.1 CPU (Central Processing Unit)	3
1.4.2. Memory	3
1.4.3. Input and Output Modules (I/O)	4
1.4.4. Power Supply	4
1.4.5. Programming Devices	4
1.4.6. System Busses	5
1.4.7. The Scan Cycle	5
1.5. Basic Operation	6
1.6. PLC standard EN 61131 (IEC 61131)	6
1.7. PLC Programming Languages	7
1.7.1. Ladder Diagram (LD)	7
1.7.2 Function Block Diagram (FBD)	8
1.7.3. Structured Text (ST)	8
1.7.4. Instruction List (IL)	9
1.7.5. Sequential Function Chart (SFC)	9
1.8. PLC Configurations	10
1.8.1. An Integrated or Compact PLC	10

1.8.2. A Modular PLC	10
1.10. PLC Applications.....	11
1.11. Plc Advantages	12
1.12. Plc Manufactures	13
Chapter 2: Real Time Operating System	14
2.1. Introduction to Operating System.....	14
2.2. Types of Operating System	15
2.2.1 Batch Operating System.....	15
2.2.2 Time-sharing Operating Systems	15
2.2.3 Distributed Operating System	15
2.2.4. Network Operating System	15
2.2.5. Real-Time Operating System	16
2.3. Types of RTOS (real time operation system)	16
2.3.1. Hard real-time	16
2.3.2. Firm real-time.....	16
2.3.3. Soft real-time.....	16
2.4. Tasks & Functions	17
2.4.1. What is a Task?	17
2.4.2. Typical RTOS Task Model	17
2.4.3. Task Classification	18
2.5. What is an Interrupt?.....	18
2.6. Features of RTOS	18
2.7. RTOS architecture	19
2.7.1. Kernel	19
2.7.2. Task management.....	20
2.7.3 Scheduler.....	21
2.7.4. Synchronization and communication	21
2.7.5. Timer Management	21
2.7.6. Device I/O Management	21
Chapter3: System Description	22
3.1. Introduction.....	22
3.2. Arduino	22
3.3. Arduino Features.....	22

3.4. Arduino IDE	23
3.5. Libraries	23
3.6. Arduino Due	24
3.7. Real time Multitask Arduino	25
3.7.1 What is ARTe?.....	26
3.7.2. Erika Enterprise.....	26
3.7.3. ARTE Design Goals.....	26
3.7.4. The ARTe Architecture.....	27
3.7.5. The ARTe Programming Model	28
3.7.6. ARTe Builds Process	29
Chapter 4:PLC Implementation	32
4.1. Introduction.....	32
4.2. Introduction to The plcLib Library	32
4.3. The Default Hardware Configuration	33
4.4. Command References	34
4.4.1. General Configuration.....	34
4.4.2. Single Bit Digital Input/output.....	34
4.4.3. Combinational Logic.....	35
4.4.4. Setting and Resetting.....	35
4.4.5. Timers	36
4.4.6. Counters	36
4.5. Building a Simple PLC Hardware	36
4.5.1. Hardware Required to Build a Simple Arduino-based PLC	37
4.5.2. Building 24V DC Input Modules.....	38
4.5.3.Building the Relay Output Modules	39
4.6. Writing PLC-Style Applications with plcLib	39
4.6.1. Single Task Application: Conveyor Driver.....	39
4.6.2. Multi-Task Application: Dual-Task Motors Driving.....	42
4.7. Results and Discussion.....	44
Conclusion	45
Bibliography	
Appendix	

List of Tables

Table4.1: plcLib Input/Output Mapping with Arduino Due.	33
Table4.2: plcLib General Configuration.	34
Table4.3: plcLib Digital I/O comands.	34
Table 4.4: plcLib Basic Logical Commands	35
Table4.5: plcLib Setting and Resetting Commands	35
Table4.6: plcLib different timers	36
Table4.7: plcLib Counter function.....	36

List of Figures

Figure 1.1: Typical PLCs.	1
Figure1.2: PLC System.	3
Figure 1.3: PLC Scan cycle	5
Figure 1.4: Example of ladder logic programming	7
Figure 1.5: FBD program	8
Figure 1.6: PLC structure text example	8
Figure 1.7: Instruction list example.....	9
Figure 1.8: SFC example	9
Figure 2.1: Layered view of computer system.....	14
Figure 2.2: Task	17
Figure2.3: Task Typical model.....	17
Figure 2.4: Kernel process.....	20
Figure 2.5: States of the task	21
Figure3.1: Arduino IDE.....	23
Figure3.2: Arduino Due.....	24
Figure3.3: Arduino Due technical information.....	25
Figure3.4: ARTe Architecture.....	27
Figure3.5: Arduino Programming Model.....	28
Figure3.6: ARTe Build Process.....	29
Figure 4.1: From Electrical Circuit to Ladder Diagram then to Simple Program.....	33
Figure 4.2: CQY80 Optocoupler.....	37
Figure 4.3: 8 Pins 5V DC Relay.....	37
Figure 4.4: The ULN2803 Pin Connection	38
Figure 4.5: Input Module of an Arduino-Based PLC.....	38
Figure 4.6: Output Module for an Arduino-Based PLC.....	39

Figure 4.7: Operating the Conveyor according to the Sensors Data.....	40
Figure 4.8: Arduino code of a Conveyor Operation.....	41
Figure 4.9: Driving 3 Motors Sequentially.....	42
Figure 4.10: Driving 2 Motors Simultaneously	42
Figure 4.11: Arduino code of dual-task Motor.....	43

General Introduction

A programmable logic controller (PLC), or programmable controller is an industrial digital computer, the main difference from most other computing devices is that PLCs are intended-for and therefore tolerant-of more severe conditions (such as dust, moisture, heat, cold), while offering extensive input/output (I/O) to connect the PLC to sensors and actuators.

The automotive industry in the USA made the PLC born regarding to the need of its use. Early PLCs were designed to replace relay logic systems. Before the PLC, control, sequencing, and safety interlock logic for manufacturing automobiles was mainly composed of relays, cam timers, drum sequencers, and dedicated closed-loop controllers. The process for updating PLC facilities was time consuming and expensive, as electricians needed to individually rewire the relays to change their operational characteristics.

The PLC (Programmable Logic Controller) has been and still the basic component of the industrial automation world. The Industrial application made the PLC systems very expensive, both to buy and repair, and also because of the highly specific skills requested to software designers to extract the maximum potentials from. To bridge the disadvantages above, many industries start using Arduino as a kind of universal programmable controller for small applications as it is cheaper, it has a simple and open source software which makes it possible for update, also because it needs a little of external hardware to make it more practical than others software.

In this project we will explain how to convert an Arduino board into a PLC-like controller using the Arduino real time extension framework (ARTE) and the appropriate interfaces for I/O.

1.1. Introduction

In this chapter a brief overview of plc will be introduced; to know the basic information needed about programmable logic controllers starting from its history to know about its architecture, hardware components, basic operation, programming languages and its advantages.

1.2. Definition

A Programmable Logic Controller, or PLC, is more or less a small digital computer with a built-in operating system (OS). This OS is highly specialized and optimized to handle incoming events in real time, i.e., at the time of their occurrence [1].

PLC is used for the automation of various electro-mechanical processes in industries. It is specially designed to survive in harsh situations and shielded from heat, cold, dust, and moisture etc. PLC consists of a microprocessor which is programmed using the computer language .The program is written on a computer and is downloaded to the PLC via cable [2].

PLC was first developed in the automobile industry to provide flexible, ruggedized and easily programmable controllers to replace hard-wired relays, timers and sequencers. Since then it has been widely adopted as high-reliability automation controllers suitable for harsh environments [3].



Figure 1.1: Typical PLCs.

1.3. Historical Background

In older days the control panels of machines were generally made with relays, contactors, cam timers, drum sequencers, and dedicated closed-loop controllers, for the automation purpose and to increase the productivity of the machine which had reduced the human interference or human dependency for the machine handling process. Relays which were used are in numbers in the hundreds or even thousands, the process for updating such relay based control panel for change-over was very time consuming and expensive, as electricians needed to individually rewire the relays to change their operational characteristics. As days passed there is a lots of changes started involving in technology in all ways including automation. In recent years people started using PLC for the automation purpose instead of Relays and contactors as PLC has many advantages over Relay based control panel [4].

In 1968 GM Hydramatic (the automatic transmission division of General Motors) issued a request for proposal for an electronic replacement for hard-wired relay systems. The winning proposal came from Bedford Associates of Bedford, Massachusetts. The first PLC, designated the 084 because it was Bedford Associates' eighty-fourth project, was the result, Dick Morley worked on this project and is being considered as the Father of PLC [5]. It was not a simple straightforward process to convince people that a small box of software can replace thousands of relays, cam timers, drum sequencers and many closed loop controllers, and even do the exact same job of them [6]. The wiring required for PLC based system is very less almost 80% reduction in wiring of control panel as that of relay based control panel. The system is very flexible in terms of introducing any new changes.

1.4. Hardware Components

PLC is designed with some basic hardware components, that each adds its own function to the PLC. Below is a diagram of the system overview of PLC.

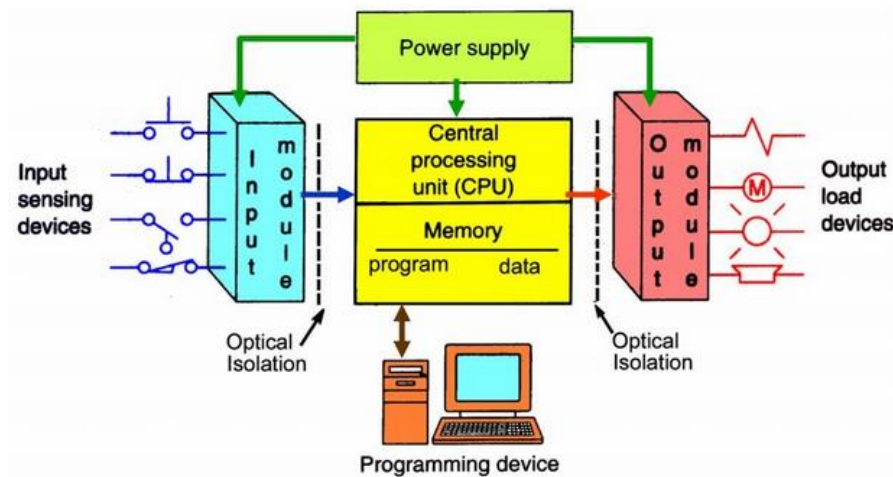


Figure 1.2: PLC Systems [7]

1.4.1. CPU (Central Processing Unit)

The Central Processing Unit provides all the logic functions as well as directing communication flow with other components; both internal and external. It is the “brain” of the PLC. Internally, the CPU provides data space to store the program instructions. It also performs math, counting and timing functions based on the program logic. Externally, the CPU updates the input file status from field devices and executes the appropriate output function [7].

1.4.2. Memory

The PLC has memory like the computer, where it saves all the information: System (ROM) to give permanent storage for the operating system and the fixed data used by the CPU. RAM for data; this is where information is stored on the status of input and output devices and the values of timers and counters and other internal device. PROM is for ROM's that can be programmed and then the program made permanent [7].

1.4.3. Input and Output Modules (I/O)

Inputs and Outputs ports are based on Reduced Instruction Set Computer (RISC), it can be digital or analog, depending on the field device needed. I/O field devices are connected to an I/O card which has opto-isolators installed in order to protect the PLC from any electrical damages [7]. An input is an uncontrolled field device of the PLC; it only reads the status of the process running and reports the data back to the PLC. An input field device could be a pushbutton switch, pressure sensor, or photo sensor to name a few [7].

An output is a command supplied by the CPU Logic to control an output field device. These field devices may include: relays, timers, motor starters, lights, counters or displays. The output command is driven by the updated input tables, the CPU logic, and a signal sent to the output field device to be controlled [7].

1.4.4. Power Supply

The Power Supply provides different voltage sources that are required for the different field devices, CPU, and internal memory operations [7]. Most PLC controllers work either at 24 VDC or 220 VAC.

1.4.5. Programming Devices

Programming devices are a way for the programmer to write and load the program. Programming Devices may also be used to monitor the I/O and the process in real time and for troubleshooting when the process doesn't function correctly. There are different types of Programming devices [7], whereas the PC or laptop is the most widely used programming device. Multiple lines of code may be viewed on one screen. Human Machine Interface's (HMI's) are becoming very popular for process monitoring and troubleshooting on the shop floor.

1.4.6. System Busses

Buses are the paths through which the digital signal flows internally of the PLC. The four system buses are [2]:

- ✓ Data bus is used by the CPU to transfer data among different elements.
- ✓ Control bus transfers signals related to the action that are controlled internally.
- ✓ Address bus sends the location's addresses to access the data.
- ✓ System bus helps the I/O port and I/O unit to communicate with each other.

1.4.7. The Scan Cycle

A PLC executes an initialization step when placed in run mode, and then repeatedly executes a scan cycle sequence. The basic PLC scan cycle consists of three steps

- ✓ An input scan
- ✓ A user program scan
- ✓ An output scan

The total time for one complete program scan is a function of processor speed, I/O modules used, and length of user program typically, hundreds of complete scans can take place in 1 second [8] .

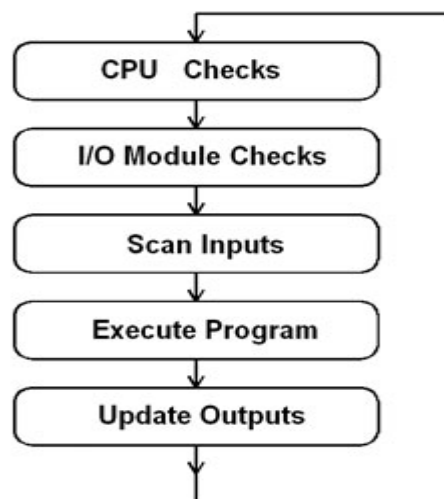


Figure1.3: PLC scans cycle.

1.4.7.1. Input Scan

During the input scan, data is taken from all input modules in the system and placed into an area of PLC memory referred to as the input image area [8].

1.4.7.2. User program scan

During the program scan, data in the input image area is applied to the user program, the user program is executed and the output image area is updated [8].

1.4.7.3. Output Scan

During the output scan, data is taken from the output image area and sent to all output modules in the system [8].

1.5. Basic Operation

The operation of the PLC system is simple and straightforward. The input/output (I/O) system is physically connected to the field devices that are encountered in the machine or that are used in the control of a process [9].

1.6. PLC standard EN 61131 (IEC 61131)

Previously, no equivalent, standardized language elements existed for the PLC developments and system expansions made in the eighties, such as processing of analogue signals, interconnection of intelligent modules, networked PLC systems etc. Consequently, PLC systems by different manufacturers required entirely different programming. Since 1992, an international standard now exists for programmable logic controllers and associated peripheral devices (programming and diagnostic tools, testing equipment, man-to-machine interfaces etc.). In this context, a device configured by the user and consisting of the above components is known as a PLC system [10].

The new EN 61131 (IEC 61131) standard consists of five parts:

- ✓ Part 1: General information
- ✓ Part 2: Equipment requirements and tests
- ✓ Part 3: Programming languages
- ✓ Part 4: User guidelines (in preparation with IEC)
- ✓ Part 5: Messaging service specification (in preparation with IEC)

The new standard takes into account as many aspects as possible regarding the design, application and use of PLC systems. The extensive specifications serve to define open, standardized PLC systems [10].

1.7. PLC Programming Languages

As PLCs have developed and expanded, programming languages have developed with them. Programming languages allow the user to enter a control program into a PLC using an established syntax. Today's advanced languages have new, more versatile instructions, which initiate control program actions. These new instructions provide more computing power for single operations performed by the instruction itself. For instance, PLCs can now transfer blocks of data from one memory location to another while, at the same time, performing a logic or arithmetic operation on another block. As a result of these new, expanded instructions, control programs can now handle data more easily [2].

Five PLC languages are available according to the IEC 11313 standard, each of is best suited to certain types of applications. The following is a list of programming languages specified by this standard:

1.7.1. Ladder Diagram (LD)

Ladder diagram is a graphical programming language; it is programmed with a contact that simulates the opening and closing of relays. Ladder logic includes many basic functions and it is used in industry for facility of its graphical use.

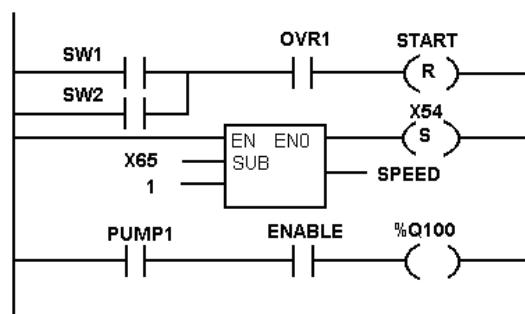


Figure 1.4: Example of a Ladder Logic Program.

1.7.2. Function Block Diagram (FBD)

It is another graphical language for depicting signal and data flows through re-usable function blocks. FBD is very useful for expressing the interconnection of control system algorithms and logic.

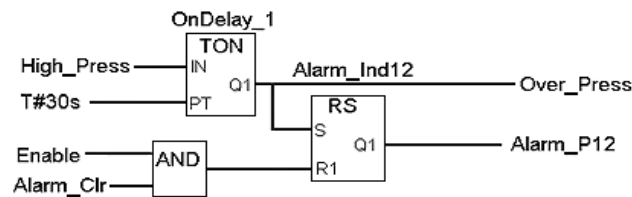


Figure 1.5: FBD Program.

1.7.3. Structured Text (ST)

Structured Text is PLC programming language defined by PLC open in IEC 61131-3 . The programming language is text-based, compared to the graphics-based ladder diagram. At first, it may seem better to use a graphical programming language for PLC programming. But that is only true for smaller PLC programs. By using a text-based PLC programming language, the program will take up much smaller space, and the flow/logic will be easier to read and understand. Another advantage is that you can combine the different programming languages, You can even have function blocks containing functions written in Structured Text [11].

```

    If Speed1 > 100.0 then
    Flow_Rate: = 50.0 + Offset_A1;
    Else
    Flow_Rate: = 100.0; Steam: = ON
    End_If;

```

Figure 1.6: PLC Structure Text program example [5].

1.7.4. Instruction List (IL)

An Instruction list (IL) is composed of a series of instructions. Each instruction begins on a new line and consists of:

- ✓ An Operator
- ✓ One or more Operands.

Each instruction uses or alters the current content of the accumulator (a form of internal cache). IEC 61131 refers to this accumulator as the "result". For this reason, an instruction list should always begin with the LD operand ("Load in accumulator command") [12]

Instruction list

```

LD      X001
SET     Y000
LDI     X001
OUT     T0      K300
LD      T0
RST     Y000
    
```

Figure 1.7: Instruction List Example.

1.7.5. Sequential Function Chart (SFC)

Sequential function chart (SFC) is a graphical language, which makes it possible to depict sequential behavior. One of the most important aspects of SFC is that it shows the main states of a system, all the possible changes of state and the reasons why those changes would occur. It can be used at the top level to show the main phases of a process, but it can also be used at any other lower level [13].

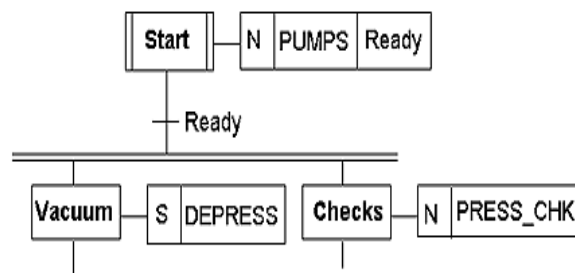


Figure 1.8: SFC example.

1.8. PLC Configurations

There are two basic configurations for Programmable Logic Controllers (PLCs) first category is the integrated PLC where the second type is the single or modular units.

1.8.1. An Integrated or Compact PLC

It is built by several modules within a single case. Therefore, the I/O capabilities are decided by the manufacturer, but not by the user. Some of the integrated PLCs allow to connect additional I/Os to make them somewhat modular [14].

1.8.2. A Modular PLC

It is built with several components that are plugged into a common rack or bus with extendable I/O capabilities. It contains power supply module, CPU and other I/O modules that are plugged together in the same rack, which are from same manufacturers or from other manufacturers. These modular PLCs come in different sizes with variable power supply, computing capabilities, I/O connectivity, etc [14]. The modular PLC is divided according to its size:

1.8.2.1. Small PLC

Is a mini-sized PLC that is designed as compact and robust unit mounted or placed beside the equipment to be controlled, it has less than 500 analog and digital I/Os. This type of PLC is used for replacing hard-wired relay logics, counters, timers, etc [14]

1.8.2.2. Medium-sized PLC

It is mostly used PLC in industries which allows many plug-in modules that are mounted on backplane of the system [14], some hundreds of input/ output points are provided by adding additional I/O cards [14].

1.8.2.3. Large PLCs

These systems are used wherein complex process control functions are required. Mostly, these PLCs are used in supervisory control and data acquisition (SCADA) systems, larger plants, distributed control systems, etc [14].

1.10. PLC Applications

The original task of a PLC involved the interconnection of input signals according to a specified program and, if "true", to switch the corresponding output. Boolean algebra forms the mathematical basis for this operation, which recognizes precisely two defined statuses of one variable: "0" and "1". Accordingly. However, the tasks of a PLC have rapidly multiplied: Timer and counter functions, memory setting and resetting, mathematical computing operations all represent functions, which can be executed by practically any of today's PLCs [15].

The demands to be met by PLC have continued to grow in line with their rapidly spreading usage and the development in automation technology. Visualization, the representation of machine statuses such as the control program being executed, via display or monitor. Hence a master computer facilitates the means to issue higher-level commands for program processing to several PLC systems the networking of several PLCs as well as that of a PLC and master computer is affected via special communication interfaces. At the end of the seventies, binary inputs and outputs were finally expanded with the addition of analogue inputs and outputs, since many of today's technical applications require analogue processing. At the same time, the acquisition or output of analogue signals permits an actual/set point value comparison and as a result the realization of automatic control engineering functions, a task, which widely exceeds the scope suggested by the name (programmable logic controller). Yet further PLCs are able to process several programs simultaneously – (multitasking) [15].

As PLCs became more advanced, methods were developed to change the sequence of ladder execution, and subroutines were implemented. This simplified programming and could also be used to save scan time for high-speed processes; parts of the program used, for example, only for setting up the machine could be segregated from those parts required to operate at higher speed.

Finally, PLCs are coupled with other automation components, thus creating considerably wider areas of application.

1.11. Plc Advantages

PLCs are evolving and continue to be the best option for a variety of industrial automation applications because of the great features it offers:

- ✓ Good Flexibility, it is possible to use just one model of a PLC to run multiple machines [8].
- ✓ Large Quantities of Contacts, The PLC has a large number of contacts for each coil available in its programming [8].
- ✓ Lower Cost, Increased technology makes it possible to condense more functions into smaller and less expensive packages [8].
- ✓ Visual Observation, A PLC circuit's operation can be seen during operation directly on a CRT screen [8].
- ✓ Ladder or Boolean Programming Method, The PLC programming can be accomplished in the ladder mode by an engineer, electrician or possibly a technician. Alternatively, a PLC programmer who works in digital or Boolean control systems can also easily perform PLC programming [8].
- ✓ High Reliability and Maintainability, Solid-state devices are more reliable, in general, than mechanical systems or relays and timers [8].
- ✓ Quality and Strong Easy Operation: it is very easy to edit or modify the program in plc by a computer offline. As it is effortless to know where the fault is located by only the graphical user interface where it can be shown and diagnoses, which make the maintenance easier.

1.12. Plc Manufactures

The need of the plc in multiple area of the industry made it a very competitive for some companies to produce and manufacture them. We can find different automation manufactures now days that offer the latest best PLC's for different usage and area of industry.

A post was written in 2012 on the top 50 automation companies which used data from controlglobal.com's 2011 list. This placed Siemens at the top spot worldwide with other major PLC manufacturers as follows:

- 1.Siemens
2. ABB
4. Schneider (Modicon)
5. Rockwell (Allen-Bradley)
7. Mitsubishi
8. GE

2.1. Introduction to Operating System

An Operating System (OS) is an interface between computer user and computer hardware. It is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. While each OS is different, most provide a graphical user interface, or GUI, that includes a desktop and the ability to manage files and folders. They also allow user to install and run programs written for the operating system [16]. Following are some of important functions of an operating System.

- ✓ Memory Management
- ✓ Processor Management
- ✓ Device Management
- ✓ File Management
- ✓ Security
- ✓ Control over system performance
- ✓ Job accounting
- ✓ Error detecting aids
- ✓ Coordination between other software and users

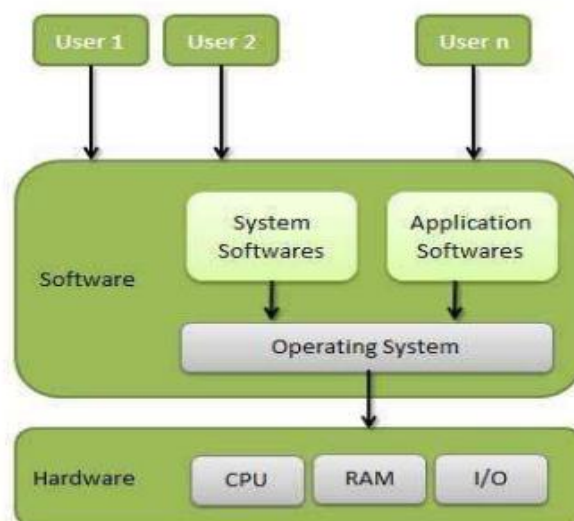


Figure 2.1: Layered view of computer system [16].

2.2. Types of Operating System

The followings are the important types of operating systems which are most commonly used.

2.2.1 Batch Operating System

Batch Operating System the users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches [16].

2.2.2 Time-sharing Operating Systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing [16].

2.2.3 Distributed Operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly. The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines) [16].

2.2.4. Network Operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD [16].

2.2.5. Real-Time Operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, scientific experiments, medical image systems, industrial control systems, weapon systems, robots, air traffic control systems, etc [16].

2.3. Types of RTOS (Real Time Operation System)

RTOS specifies a known maximum time for each of the operations that it performs. Based upon the degree of tolerance in meeting deadlines, RTOS are classified into following categories:

2.3.1. Hard real-time

A hard RTOS is distinguished by its strict adherence to the deadline or limits of the task stipulated. A missed deadline can result in catastrophic failure of the system.

2.3.2. Firm real-time

Missing a deadline might result in an unacceptable quality reduction but may not lead to failure of the complete system.

2.3.3. Soft real-time

Deadlines may be missed occasionally, but system doesn't fail and also, system quality is acceptable.

2.4. Tasks & Functions

2.4.1. What is a Task?

A task is a process that repeats itself in Loop it is an Essential building block of real time software systems. A function is a procedure that is called. It runs, then exits and may return a value. For example, `process_data();` `int add_two_numbers(int x, int y)` [17].

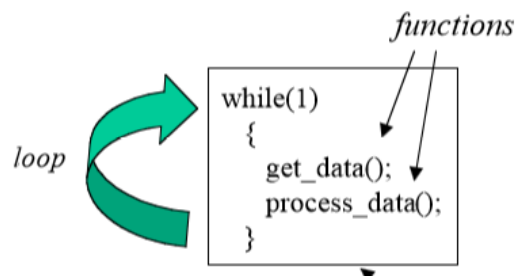


Figure 2.2: Task.

2.4.2. Typical RTOS Task Model

Each task a triple: (execution time, period, deadline)

- Execution time also called computation time, is the time necessary for execution without interruption, [18]
- Usually, deadline = period, it is the time before which task has to complete its execution [18]
- Can be initiated any time during the period [18]

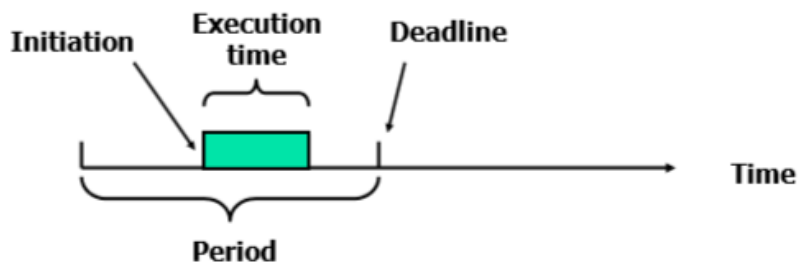


Figure2.3: Task Typical model

2.4.3. Task Classification

1. **Periodic Tasks:** also called Time-driven tasks, their activation is generated by timers. At a fixed frequency, these tasks can be characterized by 3 parameters: computations time, deadline, and period. Generally the deadline is equal to the period but it can be more or less [19].
2. **Non-Periodic or Aperiodic Tasks:** also called event-driven tasks, their activation may be generated by external interrupts [19].
3. **Sporadic Tasks:** aperiodic tasks with minimum interval time T_{\min} (often with hard deadline) [19].

2.5. What is an Interrupt?

A hardware signal that initiates an event Upon receipt of an interrupt, the processor: completes the instruction being executed save the program counter (so as to return to the same execution point) loads the program counter with the location of the interrupt handler code executes the interrupt handler In practice, real time systems can handle several interrupts in priority fashion Interrupts can be enabled/disabled Highest priority interrupts serviced first [17].

2.6. Features of RTOS

A basic RTOS will be equipped with the following features [20]

1. **Multitasking and Perceptibility:** An RTOS must be multi-tasked and perceptible to support multiple tasks in real-time applications. The scheduler should be able to preempt any task in the system and allocate the resource to the task that needs it most even at peak load.
2. **Task Priority Preemption:** defines the capability to identify the task that needs a resource the most and allocates it the control to obtain the resource. In RTOS, such capability is achieved by assigning individual task with the appropriate priority level. Thus, it is important for RTOS to be equipped with this feature.
3. **Reliable and Sufficient Inter Task Communication Mechanism:** For multiple tasks to communicate in a timely manner and to ensure data integrity among

each other, reliable and sufficient inter-task communication and synchronization mechanisms are required.

4. **Priority Inheritance** to allow applications with stringent priority requirements to be implemented, RTOS must have a sufficient number of priority levels when using priority scheduling.

2.7. RTOS architecture

For simpler applications, RTOS is usually a kernel but as complexity increases, various modules like networking protocol stacks debugging facilities, device I/Os are included in addition to the kernel [20].

2.7.1. Kernel

Kernel of operating system generally consists of two parts: kernel space (kernel mode) and user space (user mode). Kernel is the smallest and central component of an operating system. Its services include managing memory and devices and also to provide an interface for software applications to use the resources. Additional services such as managing protection of programs and multitasking may be included depending on architecture of operating system. There are three broad categories of kernel models available, namely [20]:

- **Monolithic kernel** It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) in kernel space. As such, monolithic kernels provide rich and powerful abstractions of the underlying hardware.
- **Microkernel** it runs only basic process communication (messaging) and I/O control. The other system services (file system, networking, etc) reside in user space in the form of daemons/servers. Thus, micro kernels provide a smaller set of simple hardware abstractions.
- **Exokernel** The concept is orthogonal to that of micro- vs. monolithic kernels by giving an application efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems (libOSes) and leaving the management to the application.

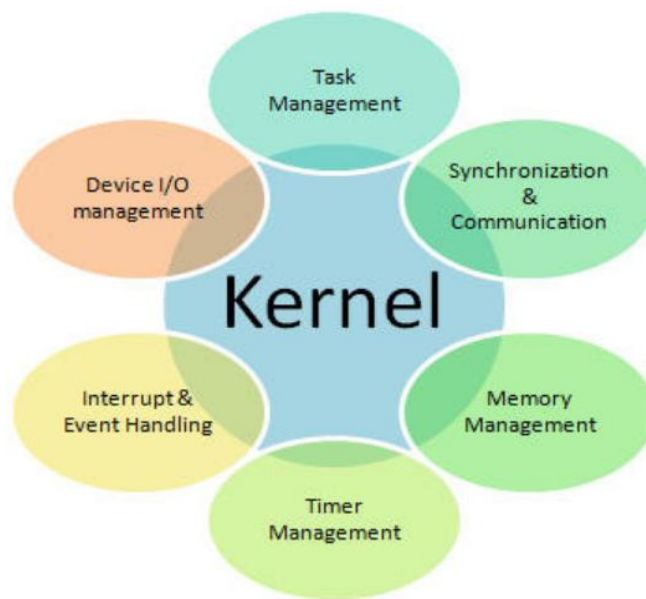


Figure 2.4: Kernel process [20].

2.7.2. Task management

A task also called a thread is a simple program that thinks it has the CPU all to itself. The design process for a real time application involves splitting the work to be done in different tasks.

Each task is assigned a priority, its own set of CPU registers and its own stack area. Each task is typically in an infinite loop that can be in any of the five states: Dormant, Ready, Running, Waiting and Interrupted. The first step in designing of the RTOS is to decide on the number of tasks that the CPU of the embedded application can handle [20].

Each task may exist in following states:

- ✓ Dormant: Task doesn't require computer time
- ✓ Ready: Task is ready to go active state, waiting processor time
- ✓ Active: Task is running
- ✓ Suspended: Task put on hold temporarily
- ✓ Pending: Task waiting for resource.

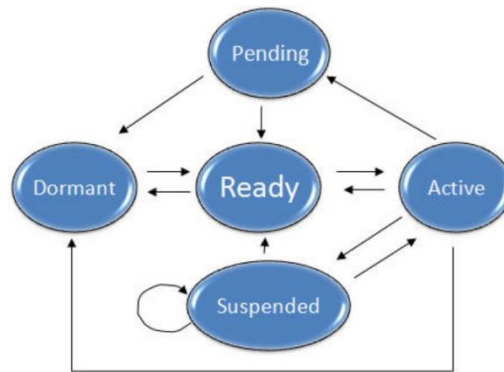


Figure 2.5: States of the task

2.7.3 Scheduler

The scheduler keeps record of the state of each task and selects from among them that are ready to execute and allocates the CPU to one of them. Various scheduling algorithms are used in RTOS [20].

2.7.4. Synchronization and communication

Task Synchronization & inter-task communication serves to pass information amongst tasks [21].

2.7.5. Timer Management

Tasks need to be performed after scheduled durations. To keep track of the delays, timers- relative and absolute- are provided in RTOS [20].

2.7.6. Device I/O Management

RTOS generally provides large number of APIs (application programming interface) to support diverse hardware device drivers [20].

3.1. Introduction

In this chapter we will list the software and hardware system used to build our plc, first the Arduino due will be introduced further more we will use the concept explained in the previous chapter RTOS to present an extension to the Arduino framework that introduces multitasking support and allows running multiple concurrent tasks in addition to the single execution cycle provided by the standard Arduino framework.

3.2. Arduino

Arduino is an open source computer hardware and software company, project, and user community that designs and manufactures single microcontrollers and microcontroller kits for building digital devices and interactive objects that can sense and control objects in the physical and digital world [21] and its platform is based on a simple input/output (I/O) board and a development environment that implements the Processing language.

3.3. Arduino Features

Arduino is different from other platforms on the market because of many features such as [22]:

- It is a multiplatform environment; it can run on Windows, Macintosh, and Linux.
- It is based on the Processing programming IDE, an easy-to-use development environment used by artists and designers.
- You program it via a USB cable, not a serial port. This feature is useful, because many modern computers don't have serial ports.
- It is open source hardware and software.
- The hardware is cheap.
- There is an active community of users.
- The Arduino Project was developed in an educational environment.

3.4. Arduino IDE

The Arduino integrated development environment (IDE) is a cross-platform application (for Windows, macOS, Linux) that is written in the programming language Java. It originated from the IDE for the languages *Processing* and *Wiring*. It includes a code editor with features such as text cutting and pasting, searching and replacing text, automatic indenting, brace matching, and syntax highlighting, and provides simple *one-click* mechanisms to compile and upload programs to an Arduino board. It also contains a message area, a text console, a toolbar with buttons for common functions and a hierarchy of operation menus [21].

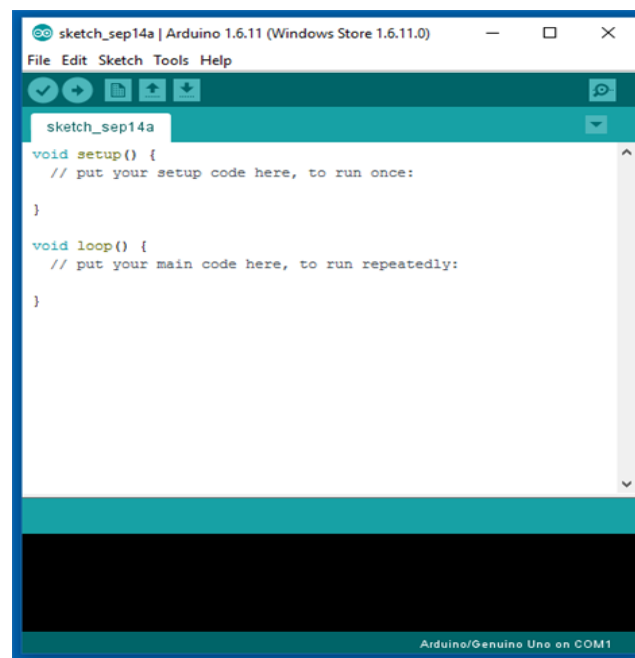


Figure3.1: Arduino IDE

3.5. Libraries

The Arduino IDE comes with a set of standard libraries for commonly used functionality. These libraries support all the examples included with the IDE. Standard library functionality includes basic communication functions and support for some of the most common types of hardware [23].

Arduino libraries can be created for different purposes, in this project we created a new library to serve the need of plc instructions. A library needs two files to be valid: a header file and a source file. The header file consists of a descriptive

comment, constructs and a class that contains the functions and variables to be used in the library. To use the library, (*plcLib.h* and *plcLib.cpp*) must be inside a folder that should be inside the Arduino libraries folder.

3.6. Arduino Due

We used in this project an Arduino Due, The Arduino Due is a microcontroller board based on the Atmel SAM3X8E ARM Cortex-M3 CPU. It is the first Arduino board based on a 32-bit ARM core microcontroller. It has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), a 84 MHz clock, an USB OTG capable connection, 2 DAC (digital to analog), 2 TWI, a power jack, an SPI header, a JTAG header, a reset button and an erase button [21].

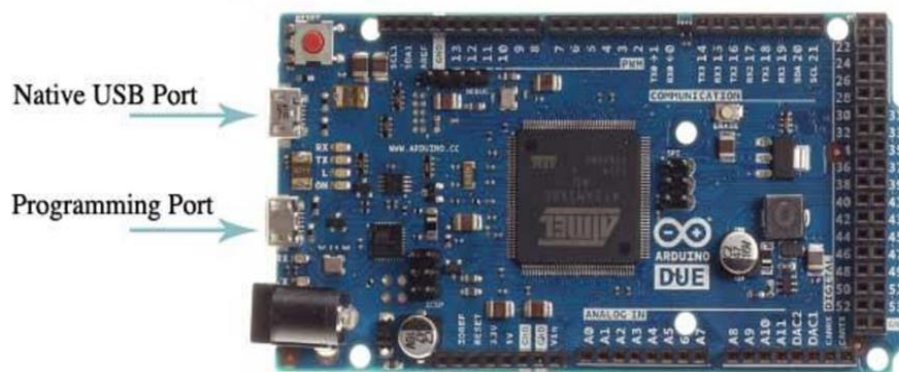


Figure3.2: Arduino Due [21]

The SAM3X has 512 KB (2 blocks of 256 KB) of flash memory for storing code. The bootloader is preburned in factory from Atmel and is stored in a dedicated ROM memory. The available SRAM is 96 KB in two contiguous banks of 64 KB and 32 KB. All the available memory (Flash, RAM and ROM) can be accessed directly as a flat addressing space. It is possible to erase the Flash memory of the SAM3X with the onboard erase button. This will remove the currently loaded sketch from the MCU[24].

Microcontroller	AT91SAM3X8E
Operating Voltage	3.3V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 12 provide PWM output)
Analog Input Pins	12
Analog Outputs Pins	2 (DAC)
Total DC Output Current on all I/O lines	130 mA
DC Current for 3.3V Pin	800 mA
DC Current for 5V Pin	800 mA
Flash Memory	512 KB all available for the user applications
SRAM	96 KB (two banks: 64KB and 32KB)
Clock Speed	84 MHz

Figure3.3: Arduino Due technical information.

3.7. Real time Multitask Arduino

Arduino is used since many years taking advantage of its simplicity; it consists of a physical programmable embedded board and an integrated development environment (IDE) that runs on a personal computer as it is mentioned in the previous sections.

In spite of its simplicity and effectiveness, Arduino framework has many limitations [28], it does not support concurrency and a program execution is limited to a single block of instructions cyclically repeated, no period can be expressed. Such a limitation prevents a full exploitation of the computing platform and in several situations forces the user to adopt tricky coding solutions to manage activities with deferent timing requirements within a single execution cycle [25].

To overcome the previous mentioned issue multiple solutions were found such as adding the scheduler library that contain a Support for multiple “concurrent” loops; Cooperative Scheduler each task is responsible to “pass the baton” this solution contains many drawback as it does not provide periodic activities, the library is just an Experimental Library till now. Neither real time nor preemption is available by this library. A very efficient solution that was recently found is the Arduino Real-Time extension or ARTe, the new Arduino framework will be more detailed in the coming sections.

3.7.1. What is ARTe?

Erika enterprise offered a new solution to overcome the one task loop problem in Arduino framework. **ARTe** (Arduino Real-Time extension) is an extension to the Arduino framework that supports multitasking and real-time preemptive scheduling.

Thanks to ARTe, the users can easily specify and run multiple concurrent loops each one scheduled with a Real-Time OS.

ARTe is supported for real-time multitasking and periodic activities; it maintains a very simple programming interface compliant with the Arduino philosophy with Minimum amount of differences with respect to the original Arduino programming model.

3.7.2. Erika Enterprise

ERIKA Enterprise (ERIKA for short) is an open-source real-time kernel that allows achieving high predictable timing behavior with a very small run-time overhead and memory footprint (in the order of one kilobyte). ERIKA is an OSEK/VDX certified RTOS that uses innovative programming features to support time sensitive applications on a wide range of microcontrollers and multi-core platforms, in addition to the OSEK/VDX standard [25].

3.7.3. ARTE Design Goals

ARTE has been conceived according to the following design objectives:

- **Simplicity:** although deferent works have been proposed to integrate a multitasking support in Arduino, thus making all the new programming features provided by ARTE ease of use. This has been achieved by designing the ARTE programming model as similar as possible to the original Arduino programming model, hence limiting the additional effort required to the user to implement concurrent applications [25].

- **Real-Time Multitasking support:** Arduino is generally used to build embedded systems that interact with the environment through sensors, actuators and communication devices. For this reason, any delay introduced in the computational activities may affect the overall system performance. Bounding the execution delays in all the concurrent activities programmed by the user is therefore crucial for ensuring a desired system performance [25].
- **Integration with standard Arduino Libraries:** The huge number of libraries provided with Arduino is one of the key strength points that determined its widespread use. To this purpose, ARTE has been conceived to enable the use of all existing Arduino libraries inside a multi programmed application [25].
- **Efficiency:** To preserve the performance of the Arduino computing platforms, ARTE has been designed to have a minimal impact on resource usage, in terms of both footprint and run-time overhead [25].

3.7.4. The ARTE Architecture

In addition to the single loop present in the standard Arduino approach, the user can easily specify a number of concurrent loops to be executed at specific rates. Concurrency and real-time scheduling is provided by the ERIKA Enterprise open-source real-time kernel

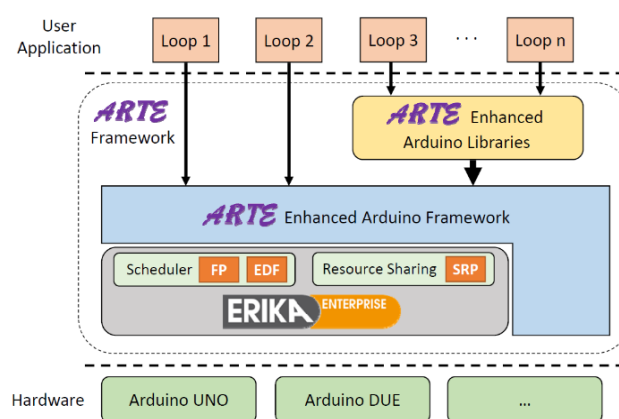


Figure3.4: ARTE Architecture

3.7.5. The ARTe Programming Model

As explained before, the ARTE programming model has been designed to result as similar as possible to the original Arduino programming model. Each periodic loop defined by the user is specified as follows:

```
void loop_i(int period)
{
    ...
}
```

Figure3.5: Arduino Programming Model

Where $i = 1, 2, 3, \dots$ and period represents the time interval (in milliseconds) with which the loop is executed. As in the original Arduino programming model, the `setup ()` function is also available under ARTE with the same syntax and semantics. Similarly, the original `loop ()` function can also be used under ARTE [25].

3.7.6. ARTe Builds Process

The whole ARTE build process flow is shown in Figure3.7. The original Arduino framework includes a sketch processing phase, denoted as Arduino processing, which is implemented inside the Arduino IDE. The main part of the ARTE build process consists in extending the Arduino IDE with two additional processing phases (shown inside the dashed box) [25]:

- (i) ARTE pre-processing
- (ii) ARTE post-processing

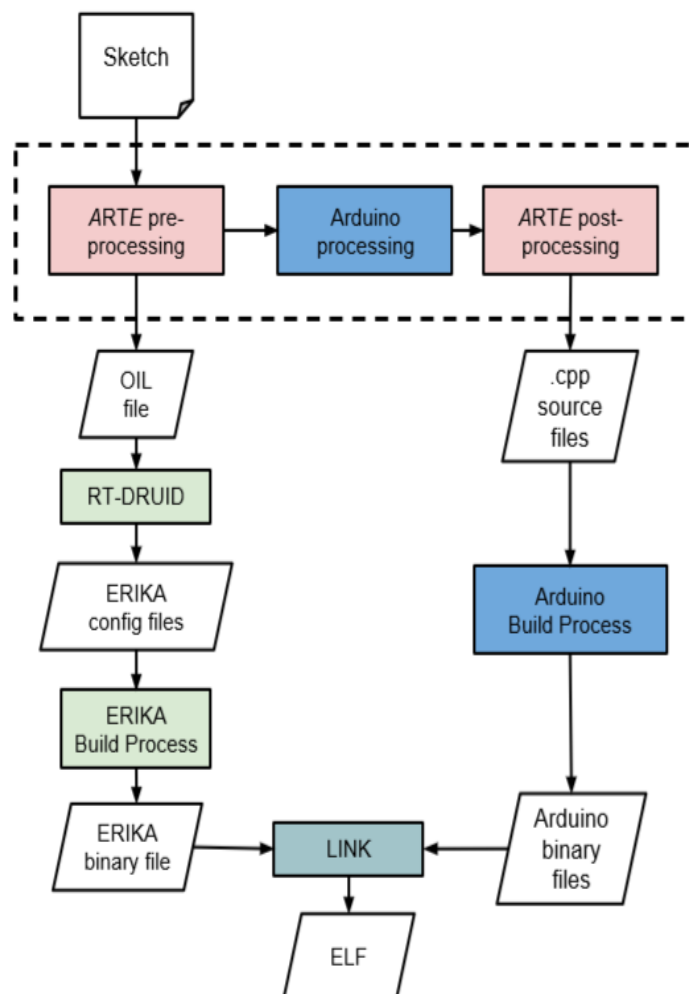


Figure3.6: ARTe Build Process [25]

3.7.6.1.ARTe Pre-Processing

During this phase, the sketch is processed to extract the structure of the application, that is, the identification of the loops with their periods, in order to automatically generate the ERIKA configuration supporting the execution of the user application. For each identified loop, an ERIKA task configuration is generated in an OIL file and then associated to the code inside the loop. In addition, the period of the loop is extracted and used to configure an OSEK alarm, which is the OSEK standard mechanism conceived to trigger periodic activities. The remaining part of the ERIKA configuration consists in an OIL section that specifies the underlying hardware platform; this section is selected from a set of predefined OIL templates [25].

3.7.6.2.Arduino processing

This phase consists in the default Arduino transformation needed to produce a compiler-compatible code. In particular, the original sketch (in .pde or .ino formats) is converted to a standard .cpp file (i.e., C++ code); any additional files beside the main one are appended to it [25].

3.7.6.3.ARTe post processing

This phase is responsible for transforming the sketch into an ERIKA application and modifies the .cpp file produced in the previous step to make it compiler-compatible. Specifically, each ARTE loop declaration is transformed into an OSEK compliant task declaration, in the form TASK (loop i). Also, since Arduino sketches are written in C++, while Erika is written in C, the ERIKA code has to be wrapped into an extern "C" declaration to avoid errors when the code is linked together. At this point, the sketch is ready to be compiled, but it still requires additions to make it fully functional. In particular, all the ERIKA initialization functions are added in the setup () function (i.e., before any user-defined code is executed), and each OSEK alarm automatically generated in the ARTE pre-processing phase is activated. In this way, task activations will be completely transparent to the user [25].

3.7.6.4.Linkin

As shown in Figure 8, the ARTE pre-processing phase produces as output the ERIKA configuration consisting in an OIL file. This file is given as input to the RT-DRUID tool, which generates the specific files of ERIKA describing its configuration. At this time, the ERIKA build process is executed to obtain the RTOS binary. Note that this binary file is an RTOS image specifically configured for the user application needs that are automatically derived from the ARTE sketch. On the other side, the user code is built by means of the standard Arduino build process, enhanced to have the visibility of ERIKA C headers, so obtaining the object files of the user application. Finally, the LINK phase puts together the ERIKA binary with the object files resulted from the Arduino build process, generating the final ELF binary file ready to be loaded into the microcontroller [25].

4.1. Introduction

In the previous chapter we had a look at the Arduino and we learnt more about the real time extension and the concept of RTOS, in this chapter we will gather all these information to build our PLC based Arduino, and we are going to test the PLC using small applications.

One approach to turn Arduino into a Programmable Logic Controller is to use an Arduino library called plcLib as we mentioned before, it provides a host of functions that can do a similar job as PLC functions. It allows us to write its code with a language similar to instruction list (instructions: LD, AND, OR ...) having the control on timers, counters and other functions.

4.2. Introduction to The plcLib Library

plcLib is a simple C/C++ code Arduino library that can be used to write control-oriented PLC software applications for Arduino boards. The library provides a host of functions to write applications for control devices in industrial environments.

The software is supplied as an installable Arduino library, which is included in the normal way at the start of the program. A range of text-based PLC-style commands then become available for use in the programs. Also, unlike a modern commercial PLC, the software does not currently support graphical program entry, simulation, or runtime monitoring. Programs must be entered using the standard Arduino IDE.

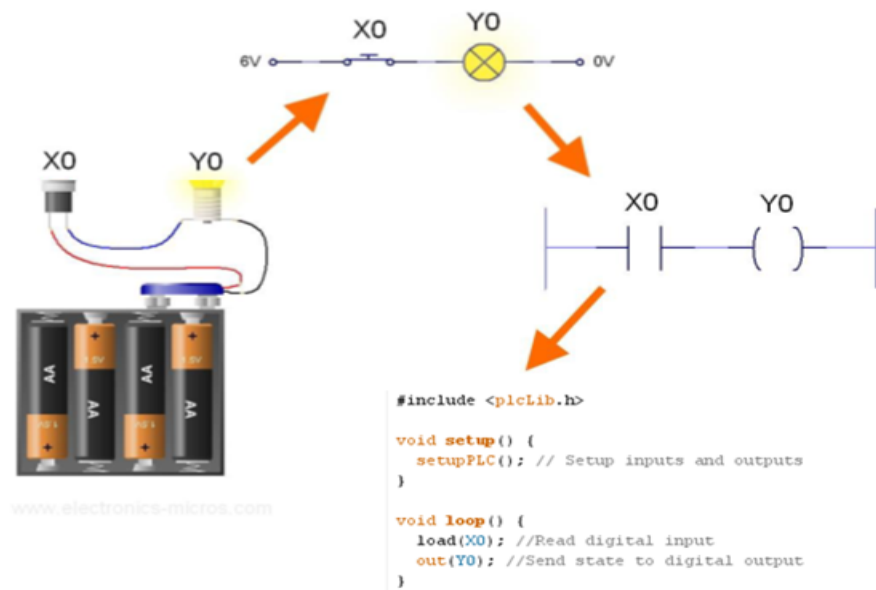


Figure 4.1: From Electrical Circuit to Ladder Diagram then to Simple Program.

4.3. The Default Hardware Configuration

The plcLib library provides software defined inputs and outputs. We select this default configuration by firstly include the plcLib file (`#include <plcLib.h>`) and then by calling the `setupPLC()` function from within the `setup()` section of the Arduino sketch.

Table 4.1: plcLib Input/Output Mapping with Arduino Due

plcLib Inputs	Arduino due Input pins	plcLib Outputs	Arduino due Output pins
X0	2	Y0	8
X1	3	Y1	9
X2	4	Y2	10
X3	5	Y3	11
X4	6	Y4	12
X5	7	Y5	13

The main features of this hardware layout are explained below:

- Inputs are capable of reading digital values.
- Outputs can produce digital values.
- Arduino pins with duplicate functions have been avoided wherever possible, to minimize hardware conflicts.
- Data directions of inputs and outputs are automatically configured and outputs are initially disabled (based on the assumption that 0='OFF' and 1 = 'ON')
- The default inputs and outputs mapping can be used with a different Arduino board with equal or less number of pins.

4.4. Command References

This section lists all commands supported by the plcLib software.

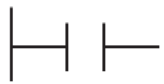
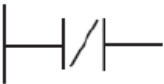
4.4.1. General Configuration


Table 4.1: plcLib General Configuration

Command	Description
setupPLC()	A command which configures data directions for the default set of inputs and outputs, together with initial output values as discussed in the Configuring the Hardware section.

4.4.2. Single Bit Digital Input/output





Table 4.2: plcLib Digital I/O Commands

PLC		Arduino		
Symbol	Instruction	Command	Description	Example
	LD	load (input)	Reads a digital input	load(X0);
	LDI	loadInv (input)	Reads an inverted digital pin	loadInv(X0);

	OUT	out (output)	Outputs to the digital output	out(Y0);
---	-----	--------------	-------------------------------	----------



4.4.3. Combinational Logic

Table 4.3: plcLib Basic Logical Commands

PLC		Arduino		
Symbol	Instruction	Command	Description	Example
	AND	andLogic (input)	Logical AND with a digital input	load(X0); andLogic(X1); out(Y0);
	ANI	andInv (input)	Logical AND with an inverted digital pin	load(X0); andInv(X1); out(Y0);
	OR	orLogic (input)	Logical OR with a digital input	load(X0); orLogic(X1); out(Y0);
	ORI	orInv (input)	Logical OR with an inverted digital input	load(X0); orInv(X1); out(Y0);

4.4.4. Setting and Resetting

Table 4.4: plcLib Setting and Resetting Commands

PLC		Arduino		
Symbol	Instruction	Command	Description	Example
	SET	set (output)	Set the digital output HIGH	load(X0); set(Y0);
	RST	reset (output)	Clear the digital output	load(X1); reset(Y0);

4.4.5. Timers

Table 4.5: plcLib different timers

Command	Description	Example
timerOn (timer_variable , timer_period)	Set output HIGH for certain duration of time in milliseconds after the output is LOW.	load(X0); timerOn(T0, 3000); out(Y0);
timerDelay(timer_variable , timer_period)	Delays activate the output until the input has been continuously active for the specific period of time in milliseconds.	load(X0); timerDelay (T0, 2000); out(Y0);
m8013 (timer_variable1, timer_variable2)	Creates a repeating 1second pulse waveform when the input is HIGH.	m8013 (T0, T1); out(Y0);

4.4.6. Counters

Table 4.6: plcLib Counter functions

Command	Description	Example
counter (counter_variable, current_State, previous_State, maxtimes)	Counts up, if counter_variable is less then maxtimes	load(X0); counter(C0,Curr , Prev,10); out(Y0);
counterState(counter_variabl e, current_State, previous_State, maxtimes)	Set the state of the counter HIGH when counting is finished	counterState(C0,Curr,Prev,10); andLogic(X1); out(Y1);

Note: counterState () function is used to activate next stage of the task required in some applications.

4.5. Building a Simple PLC Hardware

As previously mentioned in chapter 1, section 3, the main hardware components of any PLC are the power supply, the input and outputs modules and the CPU. In this section we try to build all these components.

4.5.1. Hardware Required to Build a Simple Arduino-based PLC

1. Optocoupler CQY80: A phototransistor Optocoupler device used to provide electrical isolation between an input source and an output load using light. It consists of an LED that produces infra-red light and a semiconductor photo-sensitive transistor that is used to detect the emitted infra-red beam. One application of Optocouplers is to switch a range of other large electronic devices such as transistors providing the required electrical isolation between a lower voltage control signals, in our case, one signal is from the Arduino and the other is from a much higher voltage (24V DC) or higher current output signal (relays).

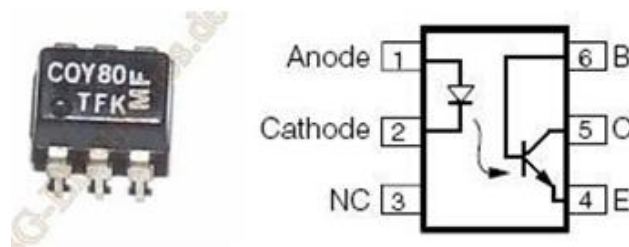


Figure 4.2 : CQY80 Optocoupler

2. Power Supply 24V DC and 5VDC.

3. Push Buttons are used as input modules.

4. LEDs to display the output.

5. 5V DC Relay: an ideal solution to drive AC or DC loads which are rated with high voltage and high power (12V DC, 24V DC, 120V AC or 240V AC).

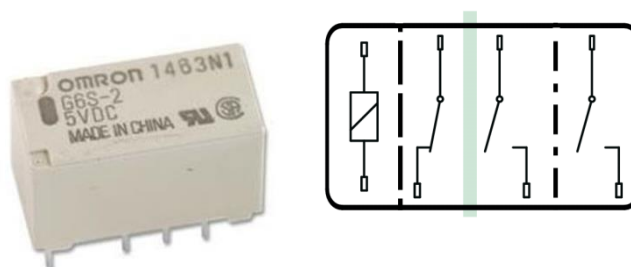


Figure 4.3: 8 Pins 5V DC Relay

6. ULN2803: is a High voltage, high current Transistor Array IC used especially with Microcontrollers where we need to drive high power loads. This IC consists of 8 NPN Darlington connected transistors with common Clamp diodes for switching the loads connected to the output. This IC is widely used to drive high loads such Lamps, relays, motors etc.

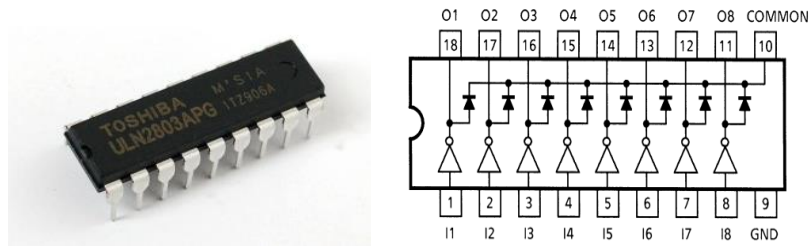


Figure 4.4 :The ULN2803 Pin Connection

7. Arduino Due: we have already introduced the Arduino due in chapter 3, section 6, in this implementation it represents the CPU of our PLC that will provide all logic functions. It is the space where the program instructions will be stored.

4.5.2. Building 24V DC Input Modules

To build the PLC input modules an optocoupler was required to provide the required isolation between the 24V DC power supply and the 3.3V DC supplied by the Arduino Due pins.

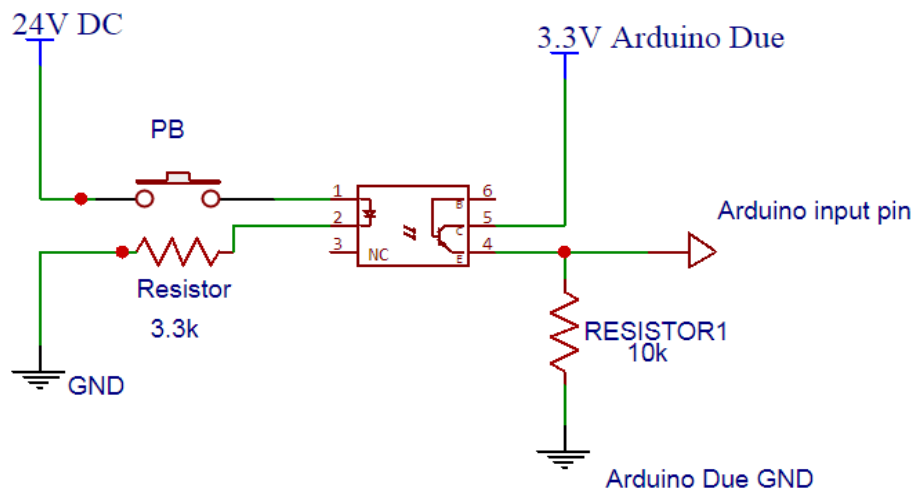


Figure 4.5: Input Module of an Arduino-Based PLC

4.5.3. Building the Relay Output Modules

The lowest voltage relay i.e 5V relay needs about 200mA current at 5V. So it is quite easy to know that the digital pins cannot drive the relay directly. Therefore a very practical solution to get a little more power out of the outputs is to use a Darlington ULN2803 since it can source up to 500mA of current out of each pin. Also the CQY 80 optoisolator is used to provide the required isolation.

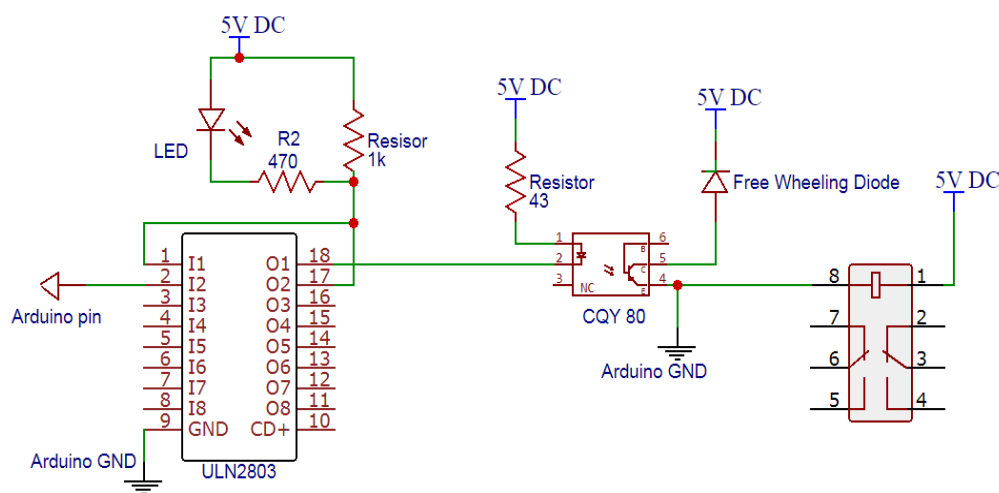


Figure 4.6: Output Module for an Arduino-Based PLC

4.6. Writing PLC-Style Applications with plcLib

To test the functionality of our Arduino-based PLC we have tried to implement two different PLC applications one with single task and the other with multi-task by enabling the ARTE.

4.6.1. Single Task Application: Conveyor Driver

Product packaging is one of the most frequent cases for automation in industry. It can be encountered with small machines (ex. packaging grain like food products) and large systems such as machines for packaging medications. The Example we are showing in (Fig.4.7) solves the classic packaging problem with few elements of automation. Small number of needed inputs and outputs provides for the use of ladder PLC controller which represents simple and economical solution.

Chapter 04: PLC Implementation

By pushing START key (X0) motor of a conveyor for boxes is activated (Y0). The conveyor takes a box up to the limit switch, using the optical sensor X1, and then the motor stops. Condition for starting a conveyor with apples is actually a limit switch for a box. When a box is detected, a conveyor with apples (Y1) starts moving. Presence of the box allows counter to count 10 apples through a sensor used for apples (X2). The counting process is displayed by the red light on the wall (Y2). Once the box is full the blue light on the wall (Y3) pulses 5 times (1 sec ON, 1 sec OFF) indicating that the box is full. Also, the motor of the conveyor for apples stops and the one of the boxes moves again. Operations repeat until STOP key (X3) is pressed.

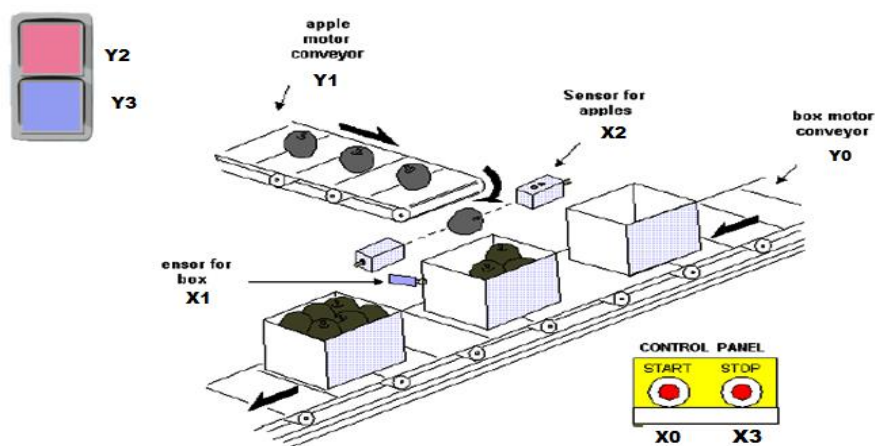


Figure 4.7: Operating the Conveyor according to the Sensors Data.

The associated sketch is giving bellow:

```
#include <plcLib.h>

unsigned long T1=0;    //variable to hold ON time of m8013
unsigned long T2=0;    //variable to hold OFF time of m8013
int C0=0;             //variable to hold counting procedure for counter 0
int C1=0;             //variable to hold counting procedure for counter 0
int cur=0;            //variable to hold current State of counter 0
int pre=0;            //variable to hold previous State of counter 0
int curl=0;           //variable to hold current State of counter 1
int prel=0;           //variable to hold previous State of counter 1

void setup() {
    setupPLC();        // Setup inputs and outputs
    Serial.begin(9600); //start the counter
}

void loop() {

    load(X0); //read start PB
    set(Y0);  // activate motor of a conveyor for boxes

    load(X1); //read sensor for boxes (PB)
    orLogic(Y1); // OR with motor of a conveyor for apples to make it move continuously
    out(Y1);    // activate motor of a conveyor for apples
    reset(Y0);  //stop motor of a conveyor for boxes

    load(X2); //read sensor for apples
    counter(C0,cur,pre,10); //count 10 apples
    out(Y2);  //Output to red LED to see the counting process

    counterState(C0,cur,pre,10); // use state of counter to activate next stage
    m8013(T1,T2);                //pulse 1sec ON, 1 sec OFF
    counter(C1,curl,prel,5); //pulse 5 times
    out(Y3);  //Output to blue LED to indicate the the box is full
    set(Y0);  //restart motor of conveyor for boxes
    reset(Y1); //stop motor of conveyor for apples

    load(X3); //read reset PB
    reset(Y0); //reset procedure
}
```

Figure 4.8: Arduino code of a Conveyor Operation

4.6.2. Multi-Task Application: Dual-Task Motors Driving

In an Oil manufacture, there are many tasks running at the same time. In this example we choose two samples about Oil Pump Motors Driving.

The 1st task: Starting 3 Motors Sequentially with delay i.e. starting the oil pump motor (Y0) immediately when START PB is pressed (X0). The main motor (Y2) will be started after a 10 sec delay and then the auxiliary motor (Y1) after a 5 sec delay, in addition stopping all motors immediately when STOP PB (X1) is pressed.

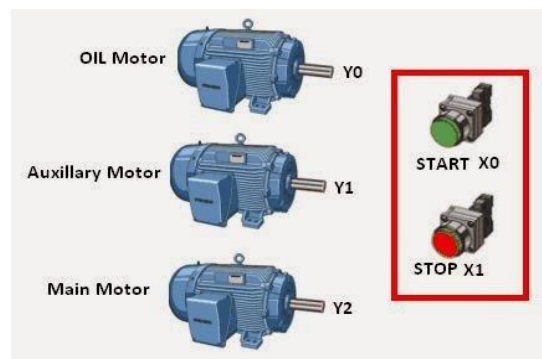


Figure 4.9: Driving 3 Motors Sequentially

The 2nd task: Starting 2 Motors Simultaneously each for a specific period of time i.e. starting When START PB is pressed (X2) the oil pump motor (Y3) is activated for 3 sec. Instantly, The main motor (Y4) will be activated for 8 sec.

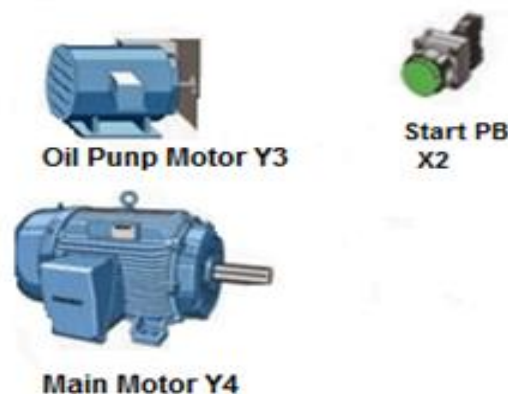


Figure 4.10: Driving 2 Motors Simultaneously

The associated sketch is giving bellow:

```
#include <plcLib.h>

unsigned long T0=0; //variable to hold delay time of timer 0
unsigned long T1=0; //variable to hold delay time of timer 1
unsigned long T2=0; //variable to hold active time of timer 2
unsigned long T3=0; //variable to hold active time of timer 3
void setup() {
    setupPLC(); // Setup inputs and outputs
}

// Task 1:Driving 3 Motors Sequentiall with DelayDriving 3 Motors Sequentiall with Delay
void loop() {
    load(X0); //read start PB X0
    set(Y0); //activet Oil Motor
    timerDelay(T0,10000); //wait 10 sec befor activating Main motor
    set(Y2); //activate Main motor
    timerDelay(T1,15000); //wait more 5 sec befor activating Auxiliary Motor
    set(Y1); //activate Auxiliary Motor

    load(X1); //read reset PB X1
    reset(Y0); //Stop Oil Motor
    reset(Y1); //Stop Auxiliary Motor
    reset(Y2); //Stop Main motor
}

// Task 2: Driving 2 Motors Simultaneously for spesific period
void loop1(100) {
    load(X2); //read start PB X2
    timerOn(T2,3000); //keep Oil motor active for 3 sec after input is low
    out(Y3); //activate Oil Motor
    timerOn(T3,5000); //keep Main motor active for 8 sec after input is low
    out(Y4); //activate Main Motor
}
```

Figure 4.11: Arduino code of dual-task Motor

4.7. Results and Discussion

After implementing the Arduino-based PLC and testing it we were able to implement simple applications within the limitation of our functions, and we came up with these results:

- We were able to implement the basic logic operations such as AND, OR, NOT and the basic timer and counter functions.
- The Arduino Due was a good choice for such operation since it offers a large number of I/O and thanks to its fast clock the communication between hardware and software was done with less interrupts and delays; however the other Arduino boards can be used with no problems such as Arduino mega which is less sensitive than the due .
- The use of ARTe was very efficient and improves the performance of our Arduino-based PLC and makes it more like a real PLC since it offers the multi-tasking feature.
- The Arduino-based PLC is a good choice to implement a PLC with a minimal budget.
- Since the software was written in c language it can be improved in order to cover larger number of instructions like implementing registers and special relays or logic blocks.
- We can make use of the Arduino features like reading analogue values or using PWM signal or serial monitor.
- To make the Arduino-based PLC more like a PLC it is good to create a Ladder graphical interface compatible with the Arduino board being used and the plcLib implemented.

Conclusion

This project aim was to implement an Arduino-based PLC controller, a new way used in industry for controlling products based on Open Source technology. This new technology offers smart and flexible installations, in addition to the low cost comparing to a real programmable logic controller (PLC).

The Arduino-based PLC controller system is divided into two main sections: The software is supplied as an installable Arduino library, plcLib, a simple C/C++ code Arduino library that was used to write control-oriented PLC software applications for Arduino boards. The library provides a host of functions to write applications for control devices in industrial environments. Also **ARTe** (Arduino Real-Time extension) made our system more realizable and efficient as it offers the multitasking and real-time preemptive scheduling. Thanks to ARTe, we were able to easily specify and run multiple concurrent loops at different rates, in addition to the single execution cycle provided by the standard Arduino framework. The second part was the hardware design that we made according to the industrial PLC hardware specifications, where a 24V DC power supply was used with Arduino Due board as a CPU (brain) of our PLC, in addition to 5V Relays with LEDs as indicators. Also, Optocouplers (CQY80) and Darlington Sink Driver (ULN2803) were used to insure the I/O isolation and the voltage and current range switching.

For future work, it would be useful to develop more control functions, such as block Logic operations, relays, registers and comparators, this make the library able to implement SFC (Sequential Function Charts) for parallel applications. Another recommendation is to make use of the Arduino functions like using Analogue pins, PWM signal and serial monitor for display. The last improvement that may be done to make this project very similar to a real PLC is to create a Ladder graphical interface that can be compiled to the Arduino to be used as monitoring tool.

The interface must contain the same functions declared in the plcLib to make sure that it will be read by the Arduino after uploading the program.

Bibliography

- [1] Mrs. Pooja.S.Puri, "Advancement in Home Appliance Automation Using PLC," D. K. T. E. Society's Textile & Engineering Institute, India, June-2016.
- [2] Manisha Hooda Niharika Thakur, "A Review Paper on PLC & Its Applications in Robotics and Automation," Dept. of ECE, Manav Rachna University, India, 4, August 2016.
- [3] "Plant, Automains Failure System Control for Power," *International Journal of Advanced Research in*, vol. Volume 7, no. Issue 2, February 2017.
- [4] Toni M. & Kinner, Russell H. P.E Harms, "Enhancing PLC Performance with Vision Systems," in *18th Annual ESD/HMI International Programmable Controllers Conference Proceedings*, 1989, pp. 387-399.
- [5] D. J. Warne (ed) M. A. Laughton, *Electrical Engineer's Reference book*, 16th ed., 2003.
- [6] M. M. I. Norashikin M. Thamrin, "Development of Virtual Machine for Programmable Logic Controller (PLC) by Using STEPS Programming Method," Malaysia, 2011.
- [7] Electrical A2Z. [Online]. <http://electricala2z.com/motors-control/plc-programmable-logic-controller-hardware-components-plc-hardware-basics/>(Accessed 20 April 2018).
- [8] Dr. D. J. Jackson, *Programmable Logic Controllers, plc Basics*.
- [9] S. F. B. a. D. J. Packs, *Microcontrollers Fundamentals For Engineers and Scientist*, Morgan & Claypool, Ed., 2006.
- [10] F.Ebel, C.Löffler, B. Plagemann, H.Regber, E.v.Terzi, A. Winter R. Bliesener, "Programmable Logic Controllers Basic Level- Textbook TP 301," Germany, 2002.

- [11] Peter. Structured Text (ST) PLC Programming with IEC-61131-3.
- [12] Paulo Jorge Oliveira. (2011-2013) Programming Languages PLC Programming Languages Instruction List.
- [13] Martin Bruggink. (1999, January) Programming PLCs using Sequential Function Chart.
- [14] Tarun Agarwal. Applications, What is a PLC System – Different Types of PLCs with. [Online]. <https://www.elprocus.com/programmable-logic-controllers-and-types-of-plcs/>(Accessed 1 March 2018).
- [15] Sadegh vosough and Amir vosough, "International journal of multidisciplinary science and engineering," *PLC and its Applications*, vol. 02, no. 08, p. 42, 2011.
- [16] (2016) operating system fundamental os concepts. Tutorials Point.
- [17] Real Time Operating Systems Lecture. MIT 16.07.
- [18] G.Buttazzo, *Hard Real-Time Computing Systems ,Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Ed.
- [19] Edinburgh The school of Informatics at the University, *RTOS course*. UK.
- [20] Mr. Mahesh P.Gaikwad, "The Design of Real Time Operating Systems for Embedded Systems," *International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)*, vol. 02, no. 02, February 2013.
- [21] arduino official website. [Online]. <https://www.arduino.cc>(Accessed 10 April2018).
- [22] Massimo Banzi, *Getting Started with Arduino*, 2nd ed. USA, 2011.
- [23] Adafruit Industries William Earl, *All About Arduino Libraries.*, 2017.
- [24] Arduino due Datasheet by the manufacturer.

- [25] Alessandro Biondi, Marco Pagani, Mauro Marinoni, Giorgio Buttazzo Pasquale Buonocunto, "ARTE: Arduino Real-Time Extension for Programming Multitasking Applications," in *SAC*, Italy, 2016.
- [26] Alessandro Biondi, Pietro Loreface Pasquale Buonocunto, *Real-Time Multitasking in Arduino*, information and perception technologies of communication, Ed. Italy.

Arduino I/O pin configuration

```
#include "Arduino.h"

#include "plcLib.h"

extern int scanValue = 0; //variable to store inputs and outputs state

// Define default pin directions and initial output levels.

void setupPLC() {

    // Basic digital input pins

    pinMode(X0, INPUT);
    pinMode(X1, INPUT);
    pinMode(X2, INPUT);
    pinMode(X3, INPUT);
    pinMode(X4, INPUT);
    pinMode(X5, INPUT);

    // Basic digital output pins

    pinMode(Y0, OUTPUT);
    pinMode(Y1, OUTPUT);
    pinMode(Y2, OUTPUT);
    pinMode(Y3, OUTPUT);
    pinMode(Y4, OUTPUT);
    pinMode(Y5, OUTPUT);

    // Default output port values

    digitalWrite(Y0, LOW);
    digitalWrite(Y1, LOW);
    digitalWrite(Y2, LOW);
    digitalWrite(Y3, LOW);
    digitalWrite(Y4, LOW);
    digitalWrite(Y5, LOW);

}
```

PLC code:

// Read an input pin (pin number supplied as integer)

```
unsigned int load (int input) {  
    scanValue = digitalRead (input); // read input pin (HIGH=1, LOW=0)  
    return (scanValue);              //return input state (1 or 0)  
}
```

// Read an inverted input (pin number supplied as integer)

```
unsigned int loadInv (int input) {  
    if (digitalRead (input) == 1) { // if input is HIGH  
        scanValue = 0; }           // set output LOW  
    else {                          //if input is LOW  
        scanValue = 1; }           // set output HIGH  
    return (scanValue);             // return result state (HIGH=1 or LOW=0)  
}
```

// Output to a digital output pin

```
unsigned int out (int output) {  
    if (scanValue == 1) {           // if input is HIGH  
        digitalWrite(output, HIGH);} //set output HIGH  
    else {                          //if input is LOW  
        digitalWrite(output, LOW);} //set output LOW  
    return(scanValue);              //return result state  
}
```

// AND scanValue with input (pin number supplied as integer)

```
unsigned int andLogic (int input) {  
    scanValue = scanValue & digitalRead(input);  
    return (scanValue);  
}
```

// AND scanValue with inverted input (pin number supplied as integer)

```

unsigned int andInv (int input) {
    scanValue = scanValue & ~digitalRead(input);
    return(scanValue); }

// OR scanValue with input (pin number supplied as integer)
unsigned int orLogic (int input) {
    scanValue = scanValue | digitalRead(input);
    return(scanValue);
}

// OR scanValue with inverted input (pin number supplied as integer)
unsigned int orInv (int input) {
    if (scanValue == 1) { // if previous input is HIGH do nothing
    }
    else { //if previous input is LOW
        if (digitalRead(input) == 0) { // if 2nd input is LOW
            scanValue = 1;} // set output HIGH
        else { // if 2nd input is HIGH
            scanValue = 0;} // set output LOW
        }
    return(scanValue); //return result
}

// Set the output HIGH (output pin number supplied as integer)
unsigned int set (int output) {
    scanValue = scanValue | digitalRead(output); // Self latch by ORing with Output pin
    if (scanValue == 1) { //if input is HIGH
        digitalWrite(output, HIGH);} // set output HIGH directly
    return(scanValue); //return result
}

// reset (or clear) the output (output pin number supplied as integer)

```



```

unsigned int reset (int output) {
    if (scanValue == 1) {    //if unput is HIGH
        digitalWrite(output, LOW); } // set output LOW
    return(scanValue); //retutn result
}

// Set output ON for certain duration (timerPeriod)

unsigned int timerOn (unsigned long & timerState, unsigned long timerPeriod) {
    if (scanValue == 0) {      // Timer input is off (scanValue = 0)
        if (timerState == 0) { // Timer is not started so do nothing
            }
        else {                  // Timer is active and counting
            if (millis() > (timerState + timerPeriod)) { // Timer has finished
                scanValue = 0; } // Result = 'turn-off output '(0)
            else {              // Timer has not finished
                scanValue = 1; } } // Result = 'turn-On output' (1)
        else {                  // Timer input is high (scanValue = 1)
            timerState = millis();} // Set timerState to current time in milliseconds
        return (scanValue); // Return result
    }

    // time delay befor LED is ON

unsigned int timerDelay (unsigned long & timerState, unsigned long timerPeriod) {
    if (scanValue == 0) {      // timer is disabled
        timerState = 0; }      // Clear timerState (0 = 'not started')
    else {                     // Timer is enabled
        if (timerState == 0) { // Timer hasn't started counting yet
            timerState = millis(); // Set timerState to current time in ms
            scanValue = 0; } // Result = 'output OFF' (0)

        else {                  // Timer is active and counting

```

```

        if (millis() <= (timerState + timerPeriod)) { // Timer has not
                                                    //finished
            scanValue = 0; } // Result = 'output OFF' (0)
        else { // Timer has finished
            scanValue = 1; } } // Result = 'output ON' (1)

    return (scanValue); // Return result
}

// creates a repeating 1second pulse waveform when enabled
// (timer1State,timer2State are unsigned long values - 32 bit)
unsigned int m8013 (unsigned long &timer1State, unsigned long &timer2State) {
    if (scanValue == 0) { // Enable input is off (scanValue = 0)
        timer2State = 0; // Ready to start HIGH pulse period when enabled
        timer1State = 1; }
    else{
        if (timer2State == 0 ) { // HIGH pulse Active
            if (timer1State == 1) { // HIGH pulse period starting
                timer1State = millis(); } // Set timerState to current time
                //in milliseconds
            else if (millis() - timer1State >= 1000) { // HIGH pulse period
                // has finished
                timer1State = 0;
                timer2State = 1; } // Ready to start LOW pulse period
            scanValue = 1; } // Result = 'Pulse HIGH' (1)
        if (timer1State == 0) { // LOW pulse Active
            if (timer2State == 1) { // LOW pulse period starting
                timer2State = millis(); } //Set timerState to current time in
                //milliseconds
            else if (millis() - timer2State >= 1000) { // LOW pulse has

```

```

//finished
timer2State = 0;
timer1State = 1; } // Ready to start HIGH pulse period
scanValue = 0; }} // Result = 'Pulse LOW' (1)
return(scanValue); //return result
}

//Counts number of pulses received on its input

unsigned int counter (int & count, int & currentState, int & previousState, int
maxtimes){

    if (scanValue == 1 && count<maxtimes ) { // check if the input is HIGH and counter
                                                //didn't reach maximum

        scanValue=1; // turn output HIGH

        currentState = 1; } // Set output state to HIGH

    else {

        scanValue=0; // turn output LOW

        currentState = 0; } // set output state LOW

    if (currentState != previousState && count<maxtimes){ // Start loop of counting

        if(currentState == 1){

            count = count + 1; //increment counter

            Serial.print("counter:"); Serial.println(count); }} // print counter

        previousState = currentState; //save OUTPUT state

        delay(50); //used for serial monitor

        return (scanValue); // return output state (ON/OFF)

    }
}

```

```

// Set the state of the counter HIGH when counting is finished

unsigned int counterState (int & count,int & currentState, int & previousState, int
maxtimes) {

if (scanValue == 1 && count<maxtimes ) { // check if the input is HIGH and counter
                                         //didn't reach maximum

    currentState = 1;                    // Set output state to HIGH
    scanValue=0; }                      // don't display output
else {
    currentState = 0; }                 // set output state LOW
if (currentState != previousState && count<maxtimes){ // Start loop of counting
    if(currentState == 1){
        count = count + 1; }           //increment counter
    previousState = currentState;      //save OUTPUT state
    if(count == maxtimes){             // if counter reaches is finished
        scanValue=1; }                 //set output HIGH
    return (scanValue); // return output state (ON/OFF)
}

```