Cape Peninsula
University of Technology

Design of an energy aware real-time application development technique

by

Henoc Ndala

Thesis submitted in fulfilment of the requirements of the degree

Master of Engineering in Electrical Engineering

Department of Electrical Engineering, Electronics and Computer Engineering

in the Faculty of Engineering and the Built Environment

at the Cape Peninsula University of Technology

Supervisor: Dr Laban Mwansa

Belville campus

Date submitted: February 2022

CPUT copyright information

The thesis may not be published either in part (in scholarly, scientific or technical journals), or as a whole (as a monograph), unless permission has been obtained from the University

DECLARATION

I, Henoc Ndala, declare that the contents of this proposal represent my own work, and that the work has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.

[illegible]

03/03/2022

Signed **Date**

ABSTRACT

The utilization and management of energy currently is a focus area in embedded systems design. Most embedded systems devices operate with limited power budget, or they are battery-powered devices. Software has a notable influence on the global system energy consumption. The evaluation of software energy consumption of application software is vital for lowering the energy consumption to extend the life span of these embedded systems. Two approaches were used. This thesis examined the effect of nesting loop structures (For and While) in C language on energy consumption during the computation of these structures on the STM32F303RE microcontroller (MCU).

In the first approach, the average energy consumption per each level of For loop was about 109.2 mJ from level 0 to level 2, and then the increase per each level was 5.25 mJ from level 2 to level 5. The average energy consumption per each level of While loop was about 78.9mJ from level 0 to level 2, and the average energy consumption was about 9.6 mJ from level 2 to level 5. The first approach demonstrated that the STM32F303 RE executing the While loop consumes about 62.41 mJ less per each level than executing the For loop structure.

The second approach indicated an exponential increase from 7.39 nJ to 408.98 J when executing the For loop structure. The energy consumption increased proportionally while executing the While loop structure. This implies an increase of 0.362 mJ per each level for the While loop. The second approach also demonstrated that the STM32F303RE consumes about 81.80 J less executing the While loop structure than executing the For loop structure.

The second approach also demonstrated an increase in the execution time in relation to the levels in the nesting fashion. The results demonstrated that the nesting loop structures increase the energy consumption in the MCU. The execution time to execute 40000 iterations in the While loop at 8 MHz increases by 25 ms from level 0 to level 2 per each nesting levels and 35 ms from level 2 to level 5 per each nesting levels of the While loop and the execution time to execute 40000 iterations in the For loop at 8 MHz increases by 10 times per each nesting levels of the For loop.

Furthermore, it is observed that the nesting levels of a For loop structures present a higher power consumption than the nesting level of While loop structures. The methodology used to achieve this research was discussed and the outcomes were addressed, including the work to be conducted in the future as well.

DEDICATION

In loving memory of my late mom, Yvette Mpukuta who believed in my ability to be successful in life and continuously supported and encouraged me in all aspects. Her belief in me made this journey possible, she will be deeply missed.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my God for the strength, wisdom and good heath, good care He provided for me on the course of this journey and for the successful journey.

With a grateful heart, I would also want to thank my supervisor Dr L. Mwansa for his guidance, availability throughout this entire research as well as to the CIR laboratory staff for their assistance on the course of the work.

A special thanks to my dad, Willy Ndala who dedicated his efforts for my success, for the unfailing support (spiritually, morally, physically, and financially), encouragement and motivation throughout all my studies since the beginning and made this achievement possible. I also express my gratitude to the Ndala brothers (Ariel, Joel, Lionel and Winner) as well as the Kapia's family for their support and encouragement.

Also, a big shoutout to my lovely Laetitia Kakese who has been of a great assistance and support throughout this journey. She kept me company and kept on reminding me of the goal and purpose even when days were hard.

I would also like to extend my gratitude to all my friends and colleagues who also advised and encouraged me throughout this amazing journey.

Acronyms and Abbreviations

Abbreviations	**Description**
A/D	Analog to Digital
ACO	Ant Colony Optimization
ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
ARM	Advanced Reduced Instruction Set Computer Machines
BPS	Bits Per Second
CAN	Control Area Network
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semi-conductor
CPU	Central Processor Unit
DSP	Digital Signal Processing
EPROM	Erasable Programmable Read Only Memory
FPU	Floating Point Unit
FSM	Finite Simple Machine
GPIO	General Purpose Input/Output
I/O	Input/Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization of Standardization
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
MCU	Microcontroller Unit
MIPS	Million Instructions Per Seconds
OS	Operating System
Pc	Program counter
PC	Personal Computer
PCB	Printed Circuit Board.
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTOS	Real Time Operating Systems
SELV	Safety Extra Low Voltage

SP	Stack pointer
SPI	Serial Peripheral Interface
SR	Status Register
SWD	Serial Wire Debug
SWIM	Single Wire Interface Module
TCB	Task Control Block
USART	Universal Synchronous Asynchronous Receiver/Transmitter

TABLE OF CONTENTS

LIST OF FIGURES.

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Embedded systems refer to a specific type of computer that executes some specific predefined programs. They include a specific processor that executes a specific software application. (V. Tiwari, S. Malik and A. Wolfe, 1994) The tremendous growth in computer technology and the internet of things have led to their development. They are predominantly used because of their promising features like low cost, high speed, or ease of control. These embedded systems find their application in smart transport systems, aerospace, smart home, smart grid, medical applications, and providing and monitoring healthy environments and structural integrity in buildings.

The relevance of the technology is found in the capacity of embedded systems to provide functionality within the commercial sphere through clear and efficient communication operationally, made possible by maximal high-level, optimized, reliable computer performance. (P. Ruberg, K. Lass and P. Ellervee, 2015) Numerous new embedded spheres have emerged in sensor networks and mobile devices over the past years specifically. They operate under various physical requirements such as time constraints, voltage levels or extreme temperatures. Power consumption has emerged as a critical consideration in the design process as batteries are utilized as the sole source to generate power. (P. Ruberg, K. Lass and P. Ellervee, 2015), (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)

During computation, the performance of the system varies rapidly and broadly in relation to the work output. In other words, the more intense the performance, the greater the power influence.

This thesis aims to present the idea of optimizing a C software program for reducing the energy consumed due to software execution during active computation in the STM32F303RE MCU of the Nucleo-64 board to diminish the global energy consumption of the embedded systems in order to improve their durability and portability. The optimization is done by evaluating the levels of nesting of the For and While loop structures commonly used in embedded system applications.

1.2 PROBLEM STATEMENT.

1.2.1 What is the problem?

Embedded systems, unlike a general-purpose computer, execute predefined tasks under very specific requirements and constraints. Due to their application allocated to a given task, embedded systems can be optimized in lessening the dimensions and the value of the product. (Majid Sarrafzadeh, 2006) Current techniques of embedded system design have been developed to seek new improvements for maximum reliability. (Juan Castillo, 2004) One of the most common solutions is platform-based design. These platforms include standard and specific hardware devices run by software applications processed via microcontrollers. The software design in these platforms comprises in excess of 80% of the expenditure of creative energy in the development of the design system. (Juan Castillo, 2004)

Energy efficiency can be achieved in numerous ways with the invention of specific fit-for-purpose hardware in the past years, as well as through the structure of the software and the nature of its interface with the relevant hardware. Despite focused study and research into the problem a universal recipe to program energy-aware software has not resulted. (T. Rauber and G. Rünger, 2018)

The recurring execution of the software conducts the activities of hardware which is also responsible for the energy consumption of the systems. The resultant energy consumption is a progressively significant non-functional property of program codes and a crucial component in the development of varied structural designs. Furthermore, these systems must meet not only the functional requirement but also the timing requirements. The correct behaviour of these systems also depends on their timeliness on the accuracy of computation while energy consumption is low. (Juan Castillo, 2004)

The research addresses the issue of energy consumption caused by the execution of software in an embedded system.

1.2.2 Why is energy consumption a problem?

In many embedded system designs, power(energy) consumption is one of the most significant constraints as most of these systems are primarily designed for mobile applications with a restricted power budget or limited lifetime battery. (Juan Castillo, 2004) Reducing the energy consumption at the software level will assist these systems to improve their battery life span, portability, and durability as a result

1.2.3 How would the problem be solved?

The research addresses the optimization of the energy consumption due to the execution of embedded software application by evaluating the nesting level of For and While loop executed by the STM32F303RE microcontroller unit (MCU) and suggest adopting an optimal loop structure so that the consumption of energy due to computation is minimized, thus decreasing the overall energy consumption of the embedded systems. The study will evaluate the impact of nested loops on energy consumption and derive a technique to optimize the energy consumption suitable for embedded systems. The technique will help embedded systems developers to develop software applications which are energy-aware.

1.3 AIM OF RESEARCH.

This research aims to come up with a development technique to diminish the energy consumption of the MCU due to the computation of an application.

1.4 RESEARCH OBJECTIVES.

The above aim will be achieved by fulfilling the research objectives below:

- To set up an MCU platform for running nesting loops.
- Implementation of the nesting loops in STM32F303 RE MCU.
- Evaluate the energy consumption of STM32F303RE MCU when running the nesting loops.
- Determine the correlation between Energy Utilization and nesting level of loops.

1.5 RESEARCH QUESTIONS.

The following research questions will be addressed:

- What is an embedded system, and what are the sources of energy consumption?
- What are the existing techniques to reduce the energy consumption of a processing unit at the software level?
- How to measure the energy consumption of the processing unit?
- What benefits are there in reducing the processing unit's energy consumption?

1.6 SIGNIFICANCE AND CONTRIBUTIONS OF THE RESEARCH.

As stated earlier, the study will evaluate the impact of nested loops on energy consumption and derive a technique to optimize the energy consumption suitable for embedded systems. The technique will help embedded systems developers to develop software applications which are energy-aware. The work done in this research will contribute to decreasing the energy consumed by the processing unit when running an application. The lower energy consumption in processing applications implies a concomitant decrease in the total consumption of energy of the embedded systems. This decrease in consumption of energy will guarantee the durability and portability of the embedded systems due to the improved battery life of mobile devices.

1.7 RESEARCH DELINEATION.

The research investigates the power(energy) consumption in MCU of embedded systems due to software execution of nesting loops only. Therefore, it will not investigate the overall energy consumption of the whole embedded systems.

1.8 RESEARCH METHODOLOGY.

This study employs the following research tools and approaches to ensure efficiency and reliability in the intended outcomes:

Literature review: reliable internet sites, journals and books provide the sources for a review of the techniques relating to reducing energy consumption at the software level.

Implementation: different levels of loops are implemented on STM32F303RE MCU. Then the energy consumption will be evaluated as the MCU executes these loops. In order to measure energy consumption, two different approaches will be used.

In the first approach, the average current consumption is measured using a Fluke Desktop multimeter set as an ammeter, and the execution time is measured using a stopwatch as this execution time is the duration of operation of the embedded systems.

In the second approach, the average current consumption is measured using the shunt resistor and the oscilloscope TDS3024 B, and the execution time is measured using the oscilloscope to determine the time taken by the MCU to execute the loops.

In both approaches, the MCU is powered by the external power supply.

1.9 THESIS OUTLINE.

Chapter 1 presents an overview of the thesis, the problem statement, the aim, the research objectives, the research questions, the significance of the research, the delineation of the research, and the methodology.

Chapter 2 provides a background on embedded systems, real-time systems, ARM microcontrollers, loops and nesting loops in C language, measurements methods of energy consumption, and the work done at the software level to reduce energy consumption.

Chapter 3 addresses the experiment methodology to be followed in order to carry out the research.

Chapter 4 presents and discusses the obtained results.

Chapter 5 provides a summary of the study with conclusions and recommendations for further research and investigation.

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION.

This chapter engage in a comprehensive survey of the publications which address the issue of stepping down the consumption of energy in embedded systems at the level of the software, focusing on the benefits of stepping down the overall system's energy consumption, the background on loop structures in C, the description of embedded and real-time systems, and the Nucleo-64 STM32F303RE board. Measurement's methodology of energy consumption on the MCU will also be discussed.

2.2 SOURCES OF ENERGY CONSUMPTION IN MCU.

According to Chen et al. (Mittal, 2014), the energy (power) consumption of embedded systems microprocessors originates from Dynamic Power due to switching activity and leakage current. In Complementary Metal Oxide Semiconductor (CMOS) technology, the dynamic power consumption is significant due to the increased processing cores on a chip.

In addition, intensive resource applications such as multimedia processing are currently executed by embedded processors. These intensive applications were initially designed for general-purpose processors. Current embedded processors use many complex features like multi-level caches and multi-cores to meet these performance requirements. This scenario has impacted the design of embedded systems to be optimized for higher performance with lower power consumption. (Mittal, 2014)

2.3 RELATED WORK - TECHNIQUES TO REDUCE ENERGY CONSUMPTION.

Several options exist to reduce the energy consumption in a microcontroller system. These approaches are convenient for some architectures and design targets. (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)In addition, there exist several research papers on optimizing hardware platforms, but not as many for software power optimization.

First, Tiwari and Wolfe (V. Tiwari, S. Malik and A. Wolfe, 1994) described a power analysis technique for embedded software. Their research aimed to produce a methodology and certify an instruction level power for any given processor. They hypothesized that by measuring the current drawn by the processor as it repetitively processes some or short instructions sequences, it is feasible to acquire the necessary data to calculate the power cost of a program. (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)

Their methodology is established on an instruction level model that measures the energy cost of single instructions and the several inter-instruction's effects. The authors, in their work, split up the assembly or machine program into blocks and evaluate the amount of consumption of these basic blocks and then compute the overall cost of energy consumption by summing up all the cost or the basic blocks. Tiwari et al. (V. Tiwari, S. Malik and A. Wolfe, 1994) also claimed that the methodology can be useful in verifying if an embedded design meets its energy constraints, and it can also be used as a guide for the design of embedded software.

Russel and Jacome (J. T. Russell and M. F. Jacome, 1998) applied an easier model. The authors discovered that numerous items such as condition codes and registers were irrelevant and were thus dispensable in the experimentation. Furthermore, it was discovered that immediate values within instructions had a massive impact and needed to randomize the values to achieve actual measurement. Finally, it was concluded that, for the individual instructions, the average energy was approximately equal. (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)

Utilising run time, average power, and frequency, the authors were able to accurately determine the utilization of power by the programme.

The work studied by Tiwari *et al*. (V. Tiwari, S. Malik and A. Wolfe, 1994) and by Russel et Jacome (J. T. Russell and M. F. Jacome, 1998) established the basis for all power (energy consumption) measurements and power estimators. (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)

Dalal and Ravikumar (V. Dalal and C. P. Ravikumar, 2001), in their paper:" Software Power Optimizations in An Embedded System", analysed the energy consumption due to software by adopting some optimization techniques like loop unrolling, loop blocking. The authors were able to confirm that energy consumption was effectively reduced through the utilization of source of code level software optimization techniques.

Chang *et al*. (C. Chang, S. Muftic and D. J. Nagel, 2007) conducted the study to determine the energy consumption of running algorithms of RC5, the data encryption standard with cypher block chaining.

They exhibited the capacity of the measurement system to determine energy consumption by the processor while running the algorithms. The energy consumption of the algorithm was obtained by means of the voltages drop measured across a shunt resistor circuit. These voltages were then helpful to compute the current from Ohm’s law. This obtained current was then used along with the supply voltage to compute the power consumption during the cryptographic algorithm.

Guimaraes *et al*. (G. Guimaraes; E. Souto; D. Sadok; J. Kelner, 2005), after determining the energy consumption measurements for the processor before including the cryptographic algorithms, the authors noted that the energy consumption would rise with the added and execution and transmission time because of the cryptographic processes. Therefore, to determine the execution time interval of the algorithms, an oscilloscope was used by observing the logical state in a General-Purpose Input/Output (GPIO) pin.

Pritt *et al*. (P. Ruberg, K. Lass and P. Ellervee, 2015) introduced an energy consumption estimation method for microcontrollers. Their approach made use of C programs only with no lower-level abstraction. Their method is founded on measuring the energy consumption per each C instruction program. The authors in their experiments used the PIC32MX460F512L MCU. Their work is founded on the fact that the total energy is obtained by adding the sum of C instructions.

The energy consumption of each instruction is measured in a loop to calculate the average and then utilized on the benchmark program to compute the energy estimation. It is important to note that the authors only used the microcontroller core without considering the Analog to Digital Converter (ADC), watchdog, timers, and dynamic scaling.

The work done by Rauber and Rünger (T. Rauber and G. Rünger, 2018) presented loop transformations to optimize the performance and energy behaviour. The authors demonstrated that loop transformations significantly influence energy consumption and performance, which differs for the different factors such as processor architecture chosen, parallelism in terms of the number of threads, application to be solved, and size of the application data. In addition, Rauber and Rünger noticed that an increased number of threads decreases the execution time.

Wu *et al*.in (Chi Ta Wu, Ang-Chih Hsieh and Ting Ting Hwang, 2006) observed that small programme loops consume a significant percentage of the execution time of a number of embedded applications. Their experiments demonstrate that dynamic instruction counts account for 70% to 80% of such loops. Furthermore, the energy consumed by the on-chip instruction cache can contain as high as 27% of the Central Processing Unit (CPU) energy. Therefore, a small loop buffer was recommended to store the regularly retrieved instructions to save access to the large instruction cache.

This will reduce the power as the total number of cycles to execute the instructions is constant, the energy is then reduced. They proposed a stack-based controller to perform instruction buffering. Their scheme deals with the nested-loop and if-then-else structures.

The outcomes show the energy consumption using this technique improves the energy consumption up to 36% and up to 25% improvement compared to instruction buffering with inner-most loop only.

Lehlogonolo and Ledwaba (Lehlogonolo P. I. Ledwaba, 2018), in their paper, “Performance Costs of Cryptography in Securing New-Generation Internet of Energy Endpoint Devices”, also evaluated the energy consumption of the Cryptography algorithm with the Cortex M microcontrollers. Their work discovered the Cortex M4 as the best suitable general-purpose processor for executing cryptographic services. As a result, they could run the algorithm with no diminution of their memory resources and disproportionate power (energy) consumption. (Lehlogonolo P. I. Ledwaba, 2018)

The works done by authors in (P. Ruberg, K. Lass and P. Ellervee, 2015), (T. Rauber and G. Rünger, 2018) and (Chi Ta Wu, Ang-Chih Hsieh and Ting Ting

Hwang, 2006) as the rest of the works do not evaluate the energy consumption consumed by the MCU with respect to the levels of loops in nesting fashion. This lays an opportunity for research in this area.

Loops are common in numerous real-time applications. These kinds of applications are detected in image processing and the field of digital processing as well. For instance, an image is represented as a two-dimensional array where each cell of the array stores one pixel of the image. The image is processed, then transformed to do some processing on the array using “for loops” to access the elements of the array. (Chabini and Wolf , 2003)

Another illustration of loop intensive real application is multimedia applications. Multimedia applications require a high processor speed. As some of them are mobile applications, decreasing the power consumption is necessary to increase the battery life. Extending the life of the battery has become a product differentiator in the market. Therefore, loops are considered as computational-and-data intensive applications. (Chabini and Wolf , 2003)

2.4 BENEFITS OF REDUCING THE ENERGY CONSUMPTION.

Reducing the power consumption for embedded systems is critical and offer the benefits below:

2.4.1 Limited Size and Battery.

As stated earlier, energy supply on battery-powered mobile embedded systems is a critical constraint. Furthermore, power consumption generates heat which is not desired in applications of mobile embedded systems. Moreover, resulting from the comparatively small sizes of the devices, management of heat transfer is limited. Decreasing the energy (power) consumption offers the usage of smaller power supplies and reduces the heat transfer, lowering cost, mass, size and area of the systems. Another benefit is that reducing power consumption leads to a simpler system design. (Mittal, 2014)

2.4.2 Ensuring Longevity.

The failure rates of the device are increased close to a factor of 2 due to a temperature increase of 15℃. The power dissipation causes a harmful impact on the reliability of the embedded systems, which is vital for mission-critical systems and devices used for medical purposes. This implies that reducing energy consumption prolongs the longevity of the system. (Mittal, 2014)

2.4.3 Addressing inefficiency resulting from the over-provisioning of resources in embedded systems.

Idle intervals in these systems increase for diverse causes as pessimistic estimates of worst-case execution time and inherent slack due to relaxed deadlines. Notwithstanding this, the designers need to accommodate resources to meet the worst-case performance requirement, which causes energy loss. Therefore, dynamic energy consumption reduction techniques make use of runtime adaption to trade performance over saving energy. Additionally, embedded systems generally find their usage in well-defined applications; static techniques can be used for per application tuning of resources. (Mittal, 2014)

2.4.4 Power Challenges Posed by CMOS Scaling.

The development in CMOS technology has exponentially grown the on-chip transistor speeds and densities. These advancements led to a technology-imposed utilization wall which prevents the portion of the chip that can be concurrently necessary at full speed within the available power. Hence, nowadays, the performance of processors is mainly limited by energy efficiency. It has been reported that if this scenario is not addressed, power constraints could impede future performance scaling. (Mittal, 2014)

2.4.5 Trends in Usage Pattern.

Recently, portable computing devices have become the key platform for web browsing, imaging, and video streaming. These tendencies have led the embedded systems to be used globally. The large-user embedded systems consume higher power consumption, while a mobile system has low power consumption than a server in the data centre. (Mittal, 2014)

2.4.6 Enabling Green Computing.

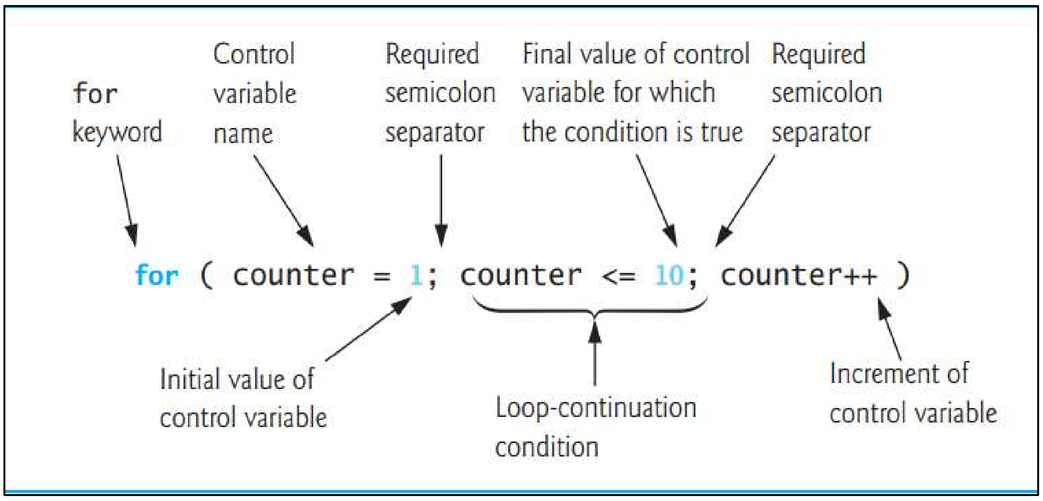
Information and Communication Technology (ICT) has contributed around 3% to the overall carbon footprint. Therefore, reducing energy consumption in embedded systems also present the advantage of achieving green computing. (Mittal, 2014)

2.5 LOOPS AND NESTING LOOPS IN C.

The majority of the program involves repetition or looping. A looping is defined as a set of instructions that the computer implements continually while the condition stays true. (Paul Deitel, Harvey Deitel, 2010)

Two kinds of the loop could be addressed. (Paul Deitel, Harvey Deitel, 2010):

- Counter-controlled repetition is also referred to as definite repetition since the number of times the loop will be executed is determined in advance. A control variable is needed to count the number of repetitions. This control variable is incremented (often by 1) each time the group of instructions is executed. When the control variable reaches the correct number of repetitions to be executed, the loop terminates, and the computer continues executing with the statement after the repetition. (Paul Deitel, Harvey Deitel, 2010)
- A counter-controlled repetition necessitates the following: the name of a control variable (or loop counter), the initial value of the control variable, and the increment (or decrement) by which the control variable is altered every moment in the loop as well as the condition that tests for the final value of the control variable. (Paul Deitel, Harvey Deitel, 2010) Figure 1 below depicts the general structure of a counter-controlled repetition. (Παυλ Δειτελ, Ηαρϖεψ Δειτελ, 2010)



```
for ( counter = 1; counter <= 10; counter++ )
```

for keyword
Control variable name
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
Initial value of control variable
Loop-continuation condition
Increment of control variable

Figure 1: General structure of a counter-controlled loop.

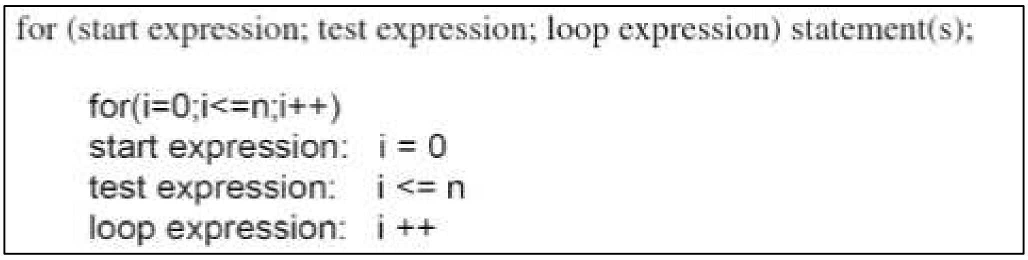
Sentinel-controlled repetition is also known as indefinite repetition, as the number of times the loop will be executed is unknown. This type of loop is used when the number of repetitions is unknown in advance and contains statements that acquire data whenever the loop is executed. The sentinel value indicates “end of data”. The sentinel is entered after all regular data items have been provided to the program. Therefore, sentinels must be different from regular data items. (Paul Deitel, Harvey Deitel, 2010)

There are three sorts of repetition or looping structures in C languages: For, While, and Do…while loops. (Paul Deitel, Harvey Deitel, 2010)

2.5.1.1 For loop.

The For loop is a highly flexible version of the conventional For statement in other high-order languages. The typical main goal of a For statement is to process a loop a defined number of times. The For loop in C can handle any number of counters and have multiple conditions to exit the loop. (Siegesmund, 2014)

The For loop general expression and its application for illustration as well as the flowchart is depicted below in Figure 2 and Figure 3, respectively (Siegesmund, 2014)



```
for (start expression; test expression; loop expression) statement(s);

     for(i=0;i<=n;i++)
     start expression:   i = 0
     test expression:    i <= n
     loop expression:    i ++
```

Figure 2: General expression of For loop.

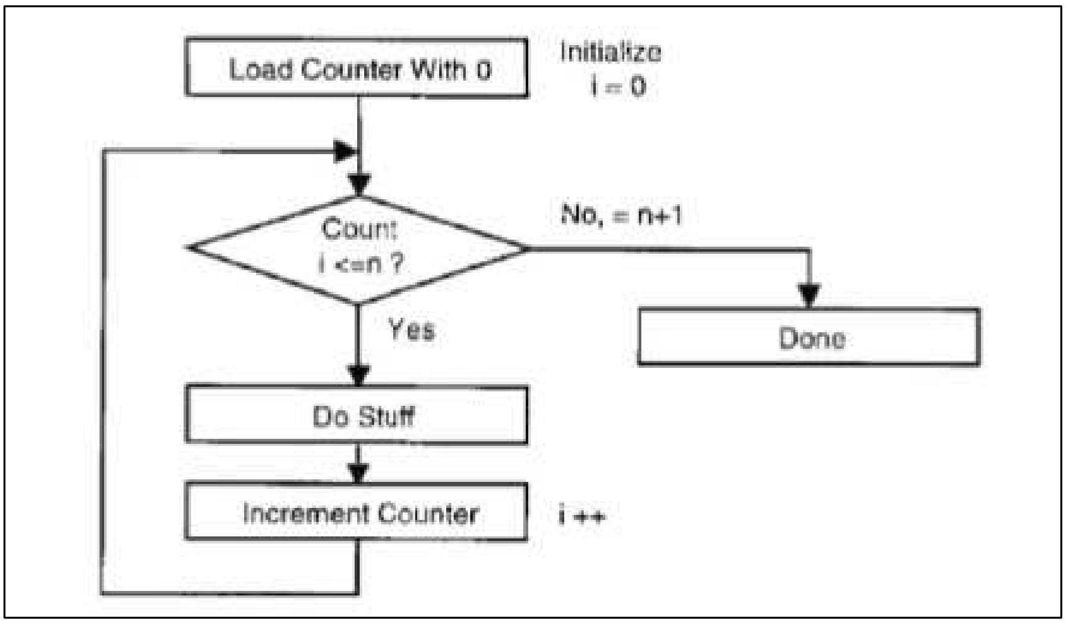


Figure 3: Flow chart of For loop.

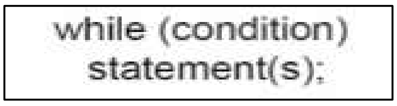
The loop evaluates the start expression only once prior to the beginning of the loop. The outcome of the start expression is disregarded. The test expression is assessed at the top of every loop, and if found not equal to 0, the loop is implemented. Alternatively, the statement ends. (Siegesmund, 2014)

The loop expression is assessed at the bottom of the loop, and the outcome is unnoticed. (Siegesmund, 2014)

2.5.1.2 While loop.

The While loop repeats the code until the condition is false. (Siegesmund, 2014)

The general expression of the while loop and its flowchart is depicted in Figure 4 and Figure 5, respectively below (Siegesmund, 2014):



```
while (condition)
  statement(s);
```

Figure 4: While loop structure.

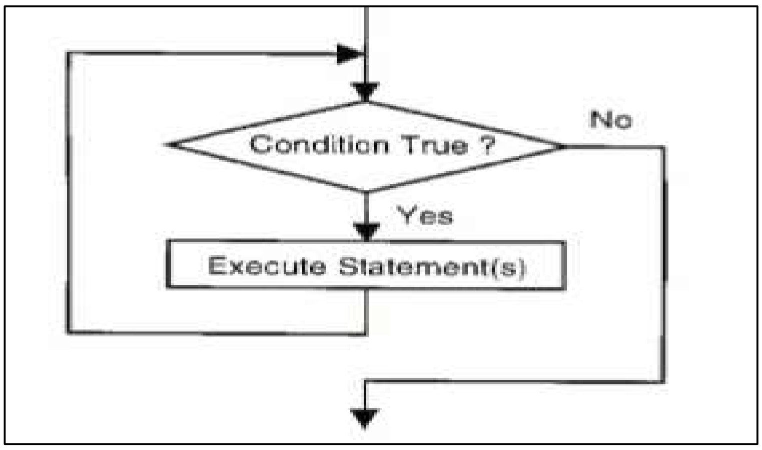


Figure 5: Flowchart of While loop.

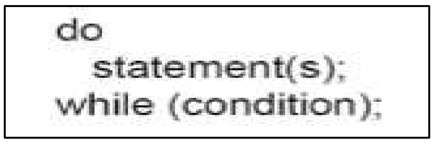
In this instance, the loop is processed as below (Siegesmund, 2014):

- The condition is first evaluated. (Siegesmund, 2014)
- If the testing condition is met (true), the statements within the loop are processed, and the execution loops back. (Siegesmund, 2014)
- If the result of the condition is false, the while loop ends, and the execution goes to the statement after the while loop if available. (Siegesmund, 2014)
- The statement after the While loop may not be executed while this (while loop) condition remains true. (Siegesmund, 2014)

2.5.1.3 Do while loop.

Unlike the while loop in which the statements may never be processed if the condition is initially not true, the do while loop is applied when the instructions are always computed at least once before testing the condition. (Siegesmund, 2014)

The general expression of the do while loop and its flowchart are depicted in Figure 6 and Figure 7 respectively below (Siegesmund, 2014):



```
do
  statement(s);
while (condition);
```

Figure 6: Do while loop structure.

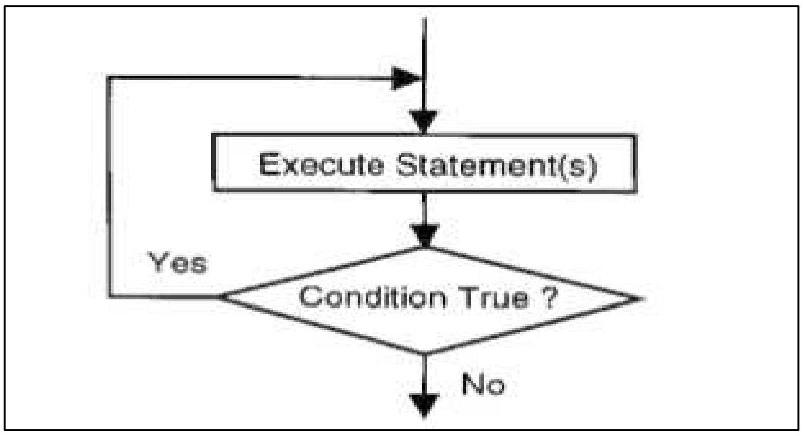


Figure 7: Flowchart of Do while loop.

In the Do while loop (Siegesmund, 2014):

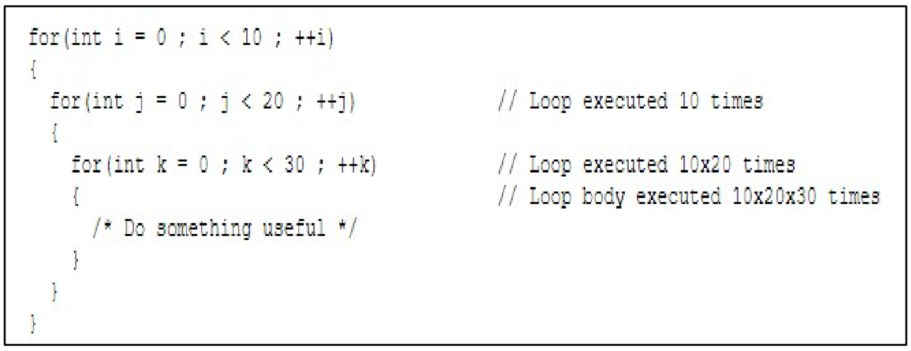
- The statements are processed first. (Siegesmund, 2014)
- The condition is verified. (Siegesmund, 2014)
- If the condition is met (true), the processing goes back to the loop and instructions are processed again. (Siegesmund, 2014)
- If the condition is not true, the do while statement ends, and the processing jumps to the instructions after this loop. (Siegesmund, 2014)
- The statements in braces will be executed at least once. (Siegesmund, 2014)

2.5.1.4 Nesting loops.

The C language allows the use of statements within a statement, and there is no limitation in the number of levels (nesting) that can be done. (Siegesmund, 2014)

The statements could be conditionals or loops. Nesting statements could be complex and confusing to read to the human reader; identification is used for the human reader to identify where the branches of conditions and loops are located (Siegesmund, 2014). Generally, indentation change occurs when the instruction forms the new condition or loop. (Siegesmund, 2014)

Below is an example of nesting loops depicted in Figure 8. (Horton, 2013)



```
for(int i = 0 ; i < 10 ; ++i)
{
  for(int j = 0 ; j < 20 ; ++j)           // Loop executed 10 times
  {
    for(int k = 0 ; k < 30 ; ++k)         // Loop executed 10x20 times
    {                                     // Loop body executed 10x20x30 times
      /* Do something useful */
    }
  }
}
```

Figure 8: Nesting loops.

The inner loop dictated by the variable j is executed once for each iteration of the outer loop regulated by the variable, i.e., the innermost k is executed once for every iteration of the loop dictated by j. In this instance, the body of the innermost loop is computed 6000 times. (Horton, 2013) Nested loops are critical since numerous embedded codes of image and video processing domains manipulate large arrays using many nested loops. (J. Ramanujam, Jinpyo Hong, M. Kandemir, A. Narayan and A. Agarwal, 2006)

2.6 EMBEDDED SYSTEMS.

2.6.1 System description.

Embedded systems operate around a microcontroller or a microprocessor that is embedded within them to undertake the control. (Wilmshurst, 2009) Figure 9 depicts a block diagram of common embedded systems. (Sutter, 2002)

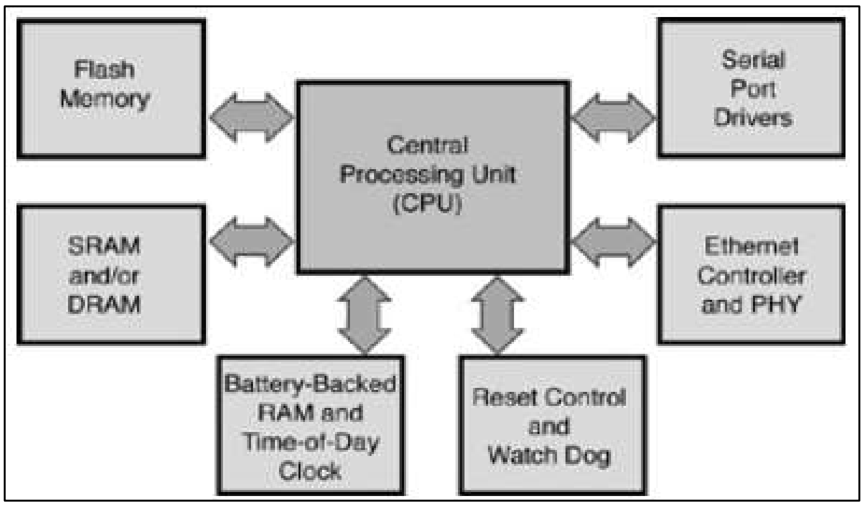

Flash Memory
Central Processing Unit (CPU)
Serial Port Drivers
SRAM and/or DRAM
Ethernet Controller and PHY
Battery-Backed RAM and Time-of-Day Clock
Reset Control and Watch Dog

Figure 9: Block diagram of Embedded Systems.

Every embedded system utilized some sort of non-volatile storage such as flash memory, Erasable Programmable Read-Only Memory (EPROM), Read-Only Memory (ROM) and include a sort of Random-Access Memory (RAM). The majority of these systems contain channels to be used to communicate with a development host (Ethernet port, Joint Test Action Group (JTAG) port or a serial port). (Sutter, 2002)

An example of such a system is the domestic fridge, as illustrated in Figure 10 below (Wilmshurst, 2009):

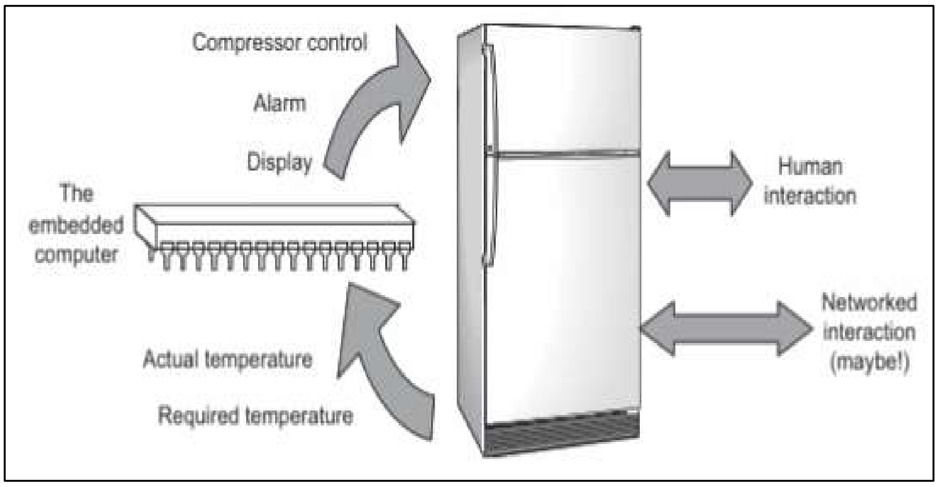


Figure 10: Domestic fridge.

A domestic refrigerator, as shown above, is functional when maintaining a steady, consistent internal low temperature. This is possible preserving the temperature at a level as set by the operator of the refrigerator by means of sensors which evaluate and adjust the temperature accordingly. The temperature is lowered by activating a compressor when it is detected that the temperature is rising above the required setting. One or more sensors are necessary for temperature measurement and wherever signal conditioning, and data acquisition circuitry is required. The data processing compares the output of the sensor indicating the actual temperature in relation to the required temperature and activates the process to achieve the desired outcome. (Wilmshurst, 2009)

The control of the compressor is facilitated via an electronic interface that transmits a low-level input control signal to the required electrical drive that will activate the compressor, resulting in its switching on or off, depending on the desired outcome—to decrease or increase the temperature in relation to the setting of the thermostat. (Wilmshurst, 2009)

A conventional electronic circuit can activate the control processor can be or, alternatively, via a small, embedded computer. The embedded computer can simplify the control process previously outlined. Furthermore, the embedded computer employs a digital signal to facilitate the processing power. Other benefits are the easier incorporation of additional advanced control features, intelligent displays, and more efficient control mechanisms (Wilmshurst, 2009).

Moreover, once an embedded computer is available, it is able to network with other computers which are embedded or not. This facilitates a smaller submodule   interacting with a larger system, with information-sharing between the systems, as in domestic products such as refrigerators and more sophisticated items. (Wilmshurst, 2009)

While figure 10 is specific to the refrigerator, it conceptualises the operation of an embedded system. This process is committed to computing internal variables, resulting in messaging that maintains the system's performance. This may include human or networked interaction. In general, consumer or operator is clueless about the presence of a computer inside the fridge. (Wilmshurst, 2009)

2.6.2 Real-time Embedded systems.

Systems that respond to external events timeously can be described as real-time systems. Such a system provides a guaranteed response time synchronously or asynchronously. The response to the external action involves recognising the time that the event occurs, the execution of the required computation as per the event, and the result of the required outcomes within a limited time. Such time comprises either the time of completion or both commencement and cessation of the event. (Qing Li, Caroline Yao, 2003)

A simpler method of conceptualising real-time and embedded systems is by picturing two overlapping circles, illustrated in Figure 11 below (Qing Li, Caroline Yao, 2003).

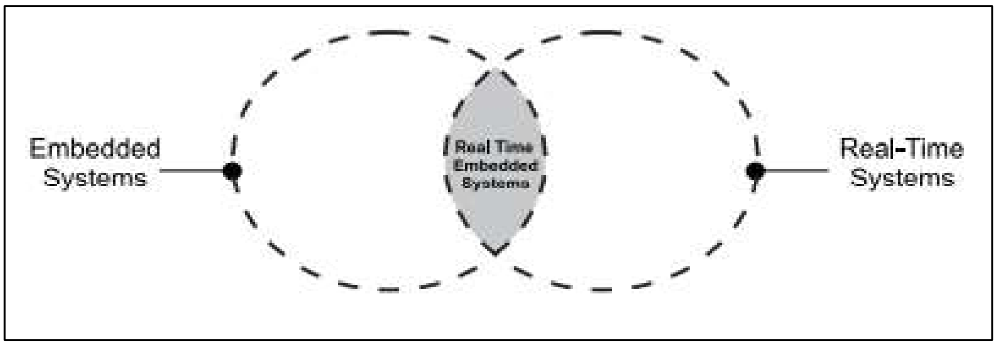


Figure 11: Connection between real-time systems and embedded systems.

The illustration shows that some embedded systems do not have real-time behaviours, and also, not all real-time systems are embedded. Nevertheless, the aforementioned systems are reciprocally exclusive. (Qing Li, Caroline Yao, 2003)

The intersecting region leads to combined systems referred to as real-time embedded systems. (Qing Li, Caroline Yao, 2003)

2.6.2.1 Real-time Systems.

The environment of real-time systems generates external events, which are sensed by a single or multiple elements of the real-time system. The output of the real-time system is subsequently fed into the environment utilizing a single or many elements. A single view of the real-time system is indicated in Figure 12 below. (Qing Li, Caroline Yao, 2003)

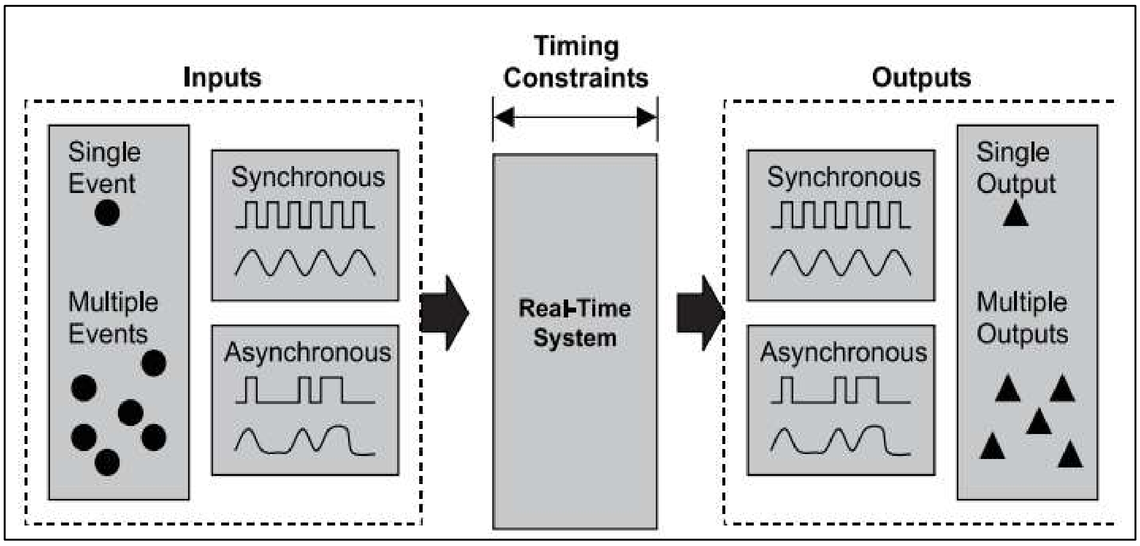


Figure 12: Overview of Real-time System.

The diagram above can be decomposed into a typical real-time system structure. A real-time system's standard structure consists of a control system and at least one controlled system. Different methods are used by the controlling system to communicate with the controlled system (Qing Li, Caroline Yao, 2003):

Firstly, the communication can be periodic, implying that the controlling system instantiates this communication to the controlled system. This means that the communication is evident and happens at set intervals. (Qing Li, Caroline Yao, 2003)

Secondly, It is possible for the controlling system and the controlled system to communicate on an ad hoc basis. This communication starts from the controlled system to the controlling system unobtrusively, defined by arbitrary external events occurring in the ambit of the controlled system. (Qing Li, Caroline Yao, 2003)

Lastly, this iteration may be a hybrid of the two. In other words, the controlling system must compute and respond to events and information generated by the controlled system within a specific time frame. (Qing Li, Caroline Yao, 2003)

Figure 13 illustrates the image of the common structure of a real-time system.

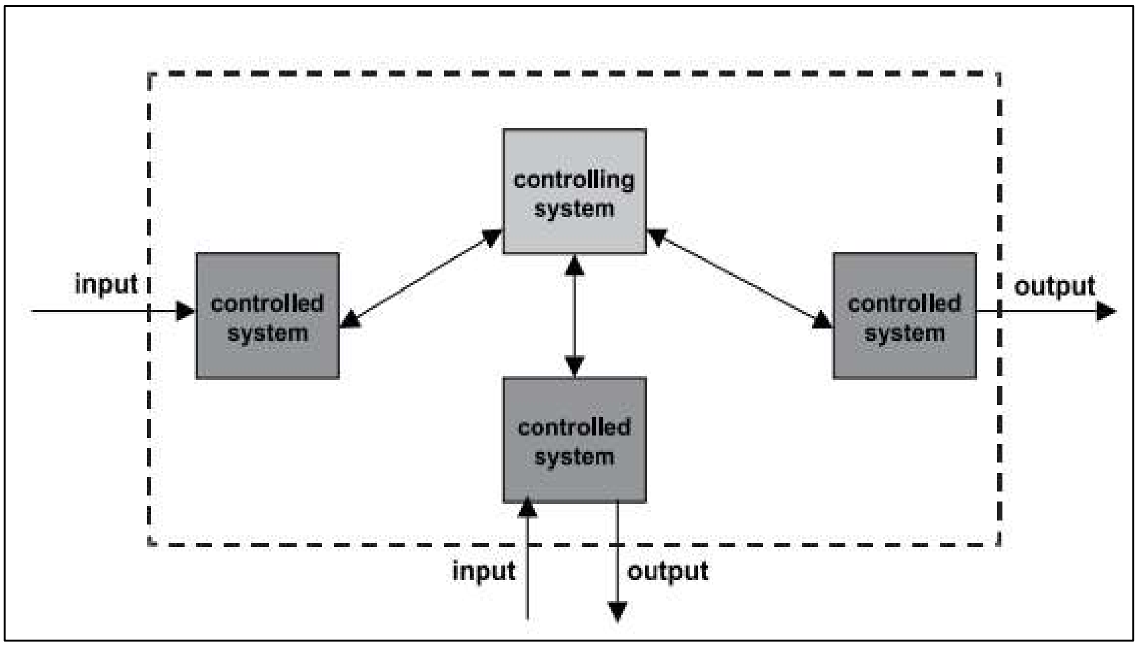


Figure 13: Common structure of a Real-time system.

2.6.2.2 Characteristics of Real-time systems.

Real-time systems exhibit two crucial features. These two features are that real-time systems have to generate correct computational responses named as functional or logical correctness. It is also necessary that these computations be complete within a prescribed time known as timing correctness. In some real-time systems, timing accuracy is at least as important as logical accuracy; there is a trade-off between the logical accuracy and the timing accuracy.

The functional correctness is often forfeited for timing correctness. (Qing Li, Caroline Yao, 2003) Real-time systems are aware of the controlled system's environment and the applications that run on it, just like embedded systems. As a result, because their response time to a perceived event is limited, several real-time systems are considered as deterministic. (Qing Li, Caroline Yao, 2003)

The required actions performed in response to an event is named as a priori. A deterministic real-time system requires that each component exhibit deterministic behaviour in order to influence the system's overall determinism. As a result, a deterministic real-time system is less adaptable to changes in the environment. As a result of this situation, the system is less robust. The levels of robustness and determinism must be balanced, and the methods for doing so vary depending on the system. (Qing Li, Caroline Yao, 2003)

2.6.2.3 Real-Time Operating System.

A real-time operating system (RTOS) is essential for many embedded systems nowadays as RTOS offers a software platform on which applications are built. However, not all embedded systems applications are based on RTOS. While some embedded systems have simple hardware with a limited amount of software application code with no RTOS requirement, many embedded systems run software applications ranging in size from medium to large, which needs some scheduling on the other hand. Hence the need for RTOS for such systems. (Qing Li, Caroline Yao, 2003)

An RTOS defines software that provides scheduling for timely execution, system resource management, and providing a consistent foundation for developing application code. Application code designed on an RTOS can be different from a simple application such as a digital stopwatch to a more complex aircraft navigation application. A suitable RTOS should therefore be scalable so that various sets of requirements for different applications are met. (Qing Li, Caroline Yao, 2003) For instance, an RTOS is made up entirely of a kernel, which is the primary supervisory software that provides applications with the bare minimum of logic, scheduling, and resource management algorithms.

An RTOS always include a kernel and can combine various components comprising the kernel, a file system, networking protocol stacks and other modules necessary for a specific application as depicted in Figure 14 below (Qing Li, Caroline Yao, 2003):

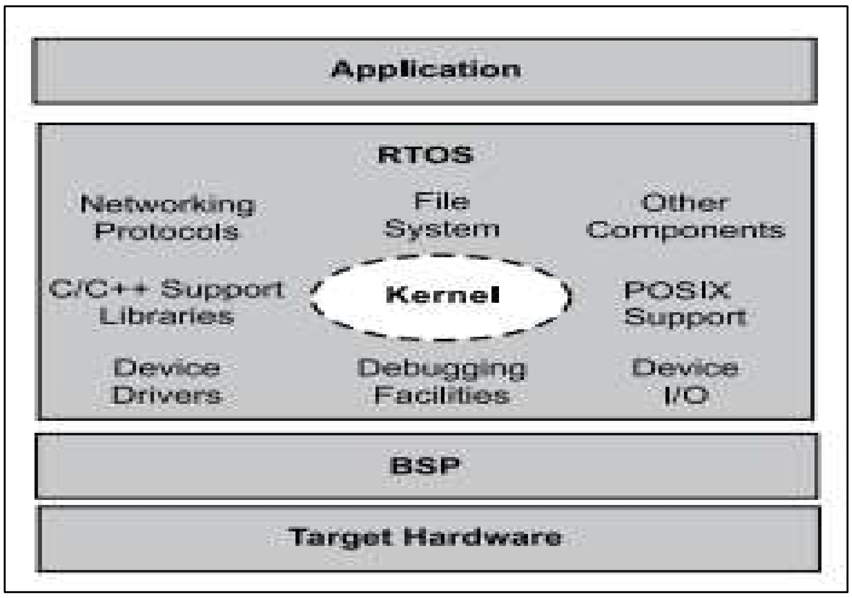


Figure 14: High-level view of RTOS, its Kernel and other modules.

In order to meet application requirements, most RTOSes can scale up or down.

The majority of RTOS kernels include the following modules (Qing Li, Caroline Yao, 2003):

Scheduler: included in every kernel and follows a set of algorithms that defines which task and when to execute. Among the scheduling algorithms, there are round-robin and pre-emptive scheduling. (Qing Li, Caroline Yao, 2003)

Objects: specifies specific kernel constructs that aid developers in developing applications for real-time embedded systems. Tasks, semaphores, and message queues are all examples of kernel objects. (Qing Li, Caroline Yao, 2003)

Services: all operations performed by the kernel on an object or operations like timing interrupt handling and resource management. (Qing Li, Caroline Yao, 2003)

Figure 15 illustrates the different modules of the kernel as found in some kernels as not all RTOS conforms precisely with the objects, scheduling algorithms and services as described below (Qing Li, Caroline Yao, 2003):

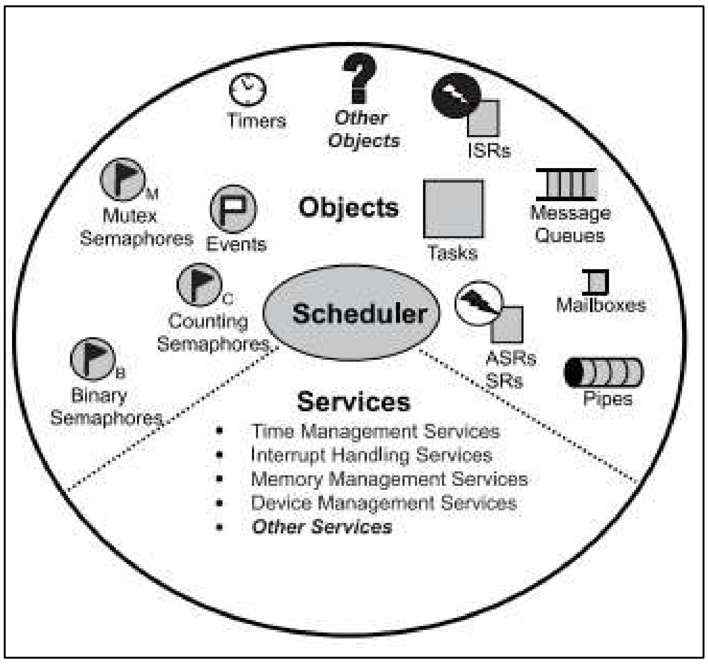


Figure 15: Common modules of Kernel.

A. The Scheduler.

Every kernel has a scheduler at its core that provides the algorithms needed to designate the tasks to be performed and when they should be completed. In order to understand the scheduling operation, Li and Yao suggest that the points below need to be discussed:

- “Schedulable entities
- Multitasking.
- Context switching.
- Dispatcher.
- Scheduling algorithms” (Qing Li, Caroline Yao, 2003).

A.1. Schedulable entities.

A schedulable entity is a kernel object that can compete for execution tiles on a specific system using a predefined scheduling algorithm. Within most kernels, tasks and processes are examples of schedulable entities in the majority of kernels. (Qing Li, Caroline Yao, 2003)

A task is defined as

an independent thread of execution that contains a [series] of schedulable instructions. Some kernels provide [a different] type of schedulable object called process. Processes are similar to tasks in that they can compete for CPU execution time [independently]. Processes, [on the other hand,] differ from tasks in that they [offer enhanced] memory protection features at the expense of performance and memory overhead. (Qing Li, Caroline Yao, 2003)

Message queues and semaphores cannot be classified as schedulable entities. They are the “inter-task communication objects required for synchronization and communication”. (Qing Li, Caroline Yao, 2003)

A.2. Multitasking.

Multitasking is the aptitude of the Operating System (OS) to manage various activities within a predefined time. A real-time Kernel can possess numerous tasks that require scheduling in order to be executed. Figure 16 below illustrates the scenario of multitasking. (Qing Li, Caroline Yao, 2003)

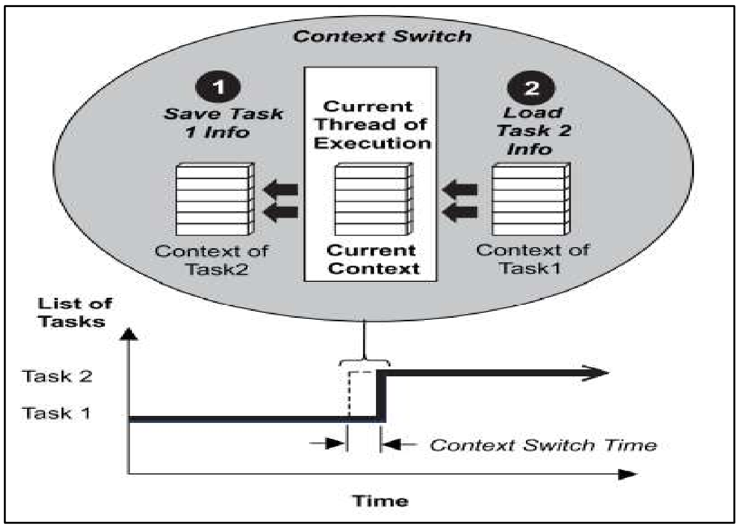


Figure 16: Scenario of multitasking.

The scenario described above indicates that the kernel runs multiple tasks in a manner that numerous paths (threads) of execution appearing to be running concurrently, while the kernel is interleaving execution chronologically, depending on a predefined scheduling policy. The scheduler guarantees that the suitable task is executed at the appropriate time. (Qing Li, Caroline Yao, 2003)

A crucial point to consider is that tasks are scheduled using the kernel's scheduling algorithm. Meanwhile, Interrupt Service Routine (ISR) is fired to run due to hardware interrupts and their defined priorities. The number of tasks to schedule is directly proportional to the CPU performance requirements; as the number of tasks to schedule rises, the CPU performance requirements also arise. This is caused by the increased switching activities between contexts of the various threads of execution. (Qing Li, Caroline Yao, 2003)

A.3. Context switch.

The context is the state of the CPU registers required each time the task is scheduled to be implemented. Every task owns its own context. A context switch happens at the instance the scheduler changes from one task to another. (Qing Li, Caroline Yao, 2003)

When a new task is created, the kernel also creates and maintains a related Task Control Block (TCB). A TCB refers to system data structures utilized by the kernel in order to maintain the specific task information. TCBs includes all information that the kernel needs to be aware of for a specific task. The task context is highly dynamic every time the task is running. (Qing Li, Caroline Yao, 2003)

TCB maintains the dynamic context. On the other hand, a task context is frozen within the TCB when the task is not running, and it is restored at the next instance the task runs. (Qing Li, Caroline Yao, 2003) A typical context switch scenario is illustrated in Figure 16.

At the instance the Kernel’s scheduler indicates that it has to stop the executing task1 and begin executing task 2, the following steps need to be taken (Qing Li, Caroline Yao, 2003):

- Firstly, The Kernel saves the context information of task1 within its TCB. (Qing Li, Caroline Yao, 2003)
- Secondly, task2’s context information is loaded from its TCB, which is now the current thread of execution. (Qing Li, Caroline Yao, 2003)

- Thirdly, task 1’s context is frozen while task 2 runs. However, once the scheduler needs to execute task 1 again, task 1 resumes where it left off nearly before the context switch. The time for the scheduler to shift from one task to another is known as the context switch time. This context switching time is negligible as opposed to the majority of operations performed by a task. (Qing Li, Caroline Yao, 2003)

However, the frequent existence of context switching in the application's design can suffer unsolicited performance overhead. Hence, applications need to be designed with limited context switching. Each time an application makes a system call, the scheduler determines whether the switch contexts are needed. If the scheduler determines that a context switch is necessary, it depends on the associated module named dispatcher to make the switch occur. (Qing Li, Caroline Yao, 2003)

A.4. The dispatcher.

The dispatcher is a scheduler component that switches contexts and alters the execution flow. When an RTOS is running, the execution flow, also known as the control flow, passes through one of three regions: an application task, the kernel, or an ISR. (Qing Li, Caroline Yao, 2003)

When a task or ISR makes a system call, the control flow is routed through the kernel to one of the kernel's system routines. When the user's application exits the kernel, the dispatcher is responsible for passing control to one of the tasks available in the user's application. The task may not be the same as the one that made the system call. This is accomplished through the scheduling policy, which specifies the next task to be carried out. (Qing Li, Caroline Yao, 2003)

The dispatcher, however, actually handles context switching and passing execution control. Based on the kernel's entry and dispatching can occur in a variety of ways. When a task makes system calls, the dispatcher has the option of exiting the kernel at the end of each system call. In this instance, the dispatcher is useful on a call-by-call basis because it can coordinate task state transitions caused by any of the system calls. (one or more tasks are ready to run, for instance). (Qing Li, Caroline Yao, 2003)

On the contrary, If an ISR issues system calls, the dispatcher is ignored until the ISR has completed its execution. This is factual even though some resources have become available, typically starting a context switch between tasks. This prevents the context switches to occur as the ISR must be complete without interruption from tasks. At the end of the ISR execution, the kernel exits via the dispatcher to dispatch the correct task. (Qing Li, Caroline Yao, 2003)

A.5. Scheduling algorithms.

As previously stated, the scheduler determines the next task to run by employing a scheduling algorithm (scheduling policy). Nowadays, most kernels support two widely used scheduling policies: pre-emptive priority-based scheduling and round-robin scheduling. These scheduling algorithms are typically predefined by the RTOS manufacturer. Nevertheless, software developers can generate and define their own scheduling policies in some cases.. (Qing Li, Caroline Yao, 2003)

A.5.1. Pre-emptive Priority-Based Scheduling.

The pre-emptive priority-based scheduling is used in most real-time kernels by default. Figure 17 below illustrates the priority-based scheduling (Qing Li, Caroline Yao, 2003):

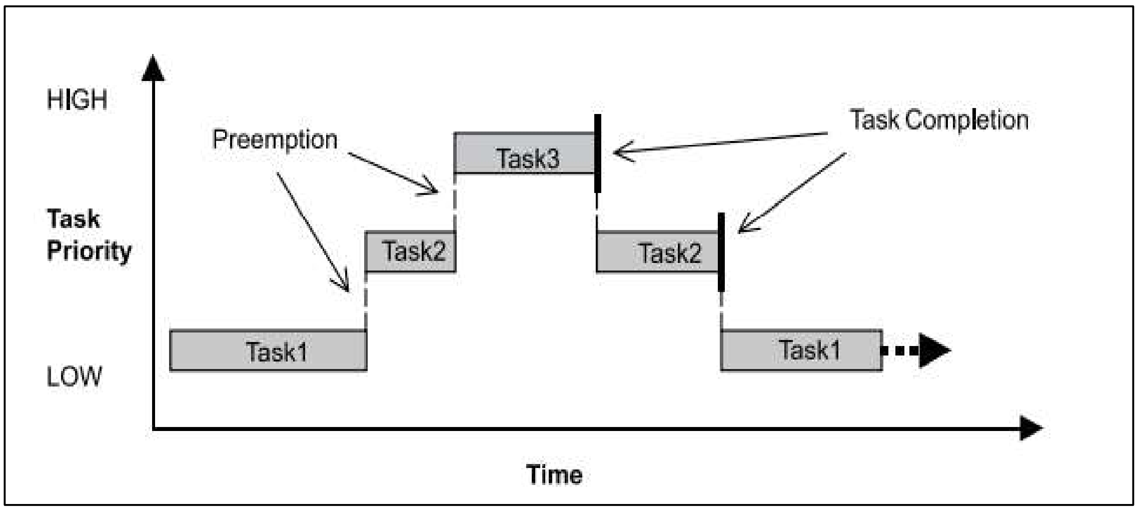


Figure 17: Pre-emptive Priority Based Scheduling.

In this scheduling policy, the task that needs to be run is the one that has the highest priority amongst all other tasks available for running in the system. Real-time kernels typically support up to 256 priority levels, with 0 being the most important and 255 being the least important. Other kernels have the priorities in reverse order, whereas 255 is the highest and 0 is the lowest. Irrespective, the principles remain the same. With the pre-emptive priority-based scheduler, each task is prioritised, and the highest priority takes precedence to run first.

A task with a priority greater than the current one is ready to be executed; the kernel directly saves the current task's context in its TCB and switches to a greater priority task. (Qing Li, Caroline Yao, 2003) As illustrated in Figure 17, task 1 is pre-empted by the greater priority task 2, which is in turn pre-empted by task 3. Upon completing task 3, task 2 resumes and similarly, when task 2 ends, task 1 resumes execution.

Although tasks are prioritised when they are created, the priority of a task can be dynamically changed using kernel-provided calls. This possibility of changing task priorities dynamically enables embedded applications the flexibility to respond to external events as they occur, creating a true real-time, responsive system. Nevertheless, the mismanagement of this feature can cause inversions, deadlock, and possible system failure. (Qing Li, Caroline Yao, 2003)

A.5.2. Round-Robin Scheduling.

Round-robin scheduling gives every task a similar slice of the CPU execution time. As a result, pure round-robin scheduling cannot satisfy real-time system requirements since, in real-time systems, tasks execute work changing degrees of importance. Instead, (Qing Li, Caroline Yao, 2003) pre-emptive priority-based scheduling can be improved with round-robin scheduling which utilizes time slicing (repartition) to accomplish an equal share of the CPU for tasks having identical priority as shown in figure 18 below (Qing Li, Caroline Yao, 2003):

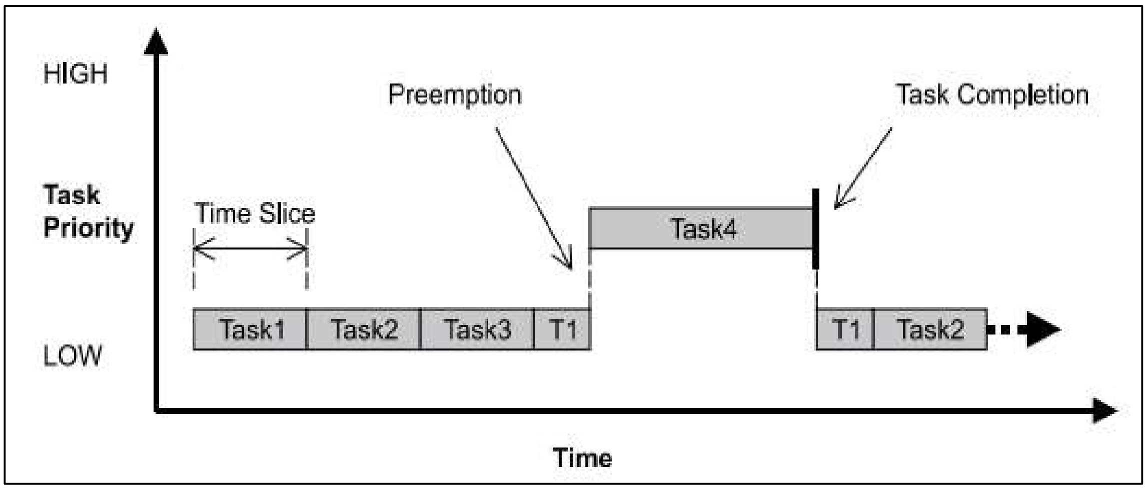


Figure 18: Round Robin scheduling.

With time slicing, every task performs in an ongoing round-robin scheduling cycle for a set period or time slice. The time-slicing for each task is tracked by a run-time counter which increments on each clock tick. When the task time slice ends, the counter is reset, and the task is moved at the tail of the cycle with their run-time reset to 0. In this instance, a round-robin cycle is preceded by a higher-priority task; its run-time count is saved and returned when the interrupted task is again available for processing. In the figure, task 1 is preceded by task 4 as it has a higher priority and restart where it was interrupted upon the completion of task4. (Qing Li, Caroline Yao, 2003)

B. Objects.

Kernel objects are particular constructs that are the building for application development for real-time embedded systems. The most familiar RTOS kernel objects are (Qing Li, Caroline Yao, 2003):

Tasks: are parallel and independent threads of execution that can run for CPU execution time. (Qing Li, Caroline Yao, 2003)

Semaphores: are tokens similar to objects which can be increased or decreased by tasks for synchronization or mutual exclusion. (Qing Li, Caroline Yao, 2003)

Message queues: are buffer like data structures useful for synchronization, mutual exclusion, and data communication by exchanging messages between tasks. Software designers designing real-time embedded applications can associate these basic kernel objects with resolving common real-time design issues such as concurrency, activity synchronization and data communication. (Qing Li, Caroline Yao, 2003)

C. Services.

With objects, many kernels offer services that assist software designers to design and develop applications for real-time embedded systems. These services include sets of API calls that can be used to perform operations on kernel objects or can be utilized in general to ease timer management, interrupt handling, device Input/Output (I/O) and memory management. (Qing Li, Caroline Yao, 2003)

Also, additional services might be provided; these services are the most common in RTOS kernels. (Qing Li, Caroline Yao, 2003)

2.6.2.4 Characteristics of the RTOS.

The application requirements determine the underlying RTOS of an application. Reliability, predictability, performance, compactness, and scalability are the main characteristics of an RTOS. However, the RTOS attribute of an application requirement is dependent on the kind of application being developed. (Qing Li, Caroline Yao, 2003)

A. Reliability

Embedded systems need to be reliable. Depending on the application, the system requires to operate for a long time with no human intervention. Various levels of reliability are required. For instance, a digital solar-powered calculator might self-reset if there is insufficient light, but the calculator remains acceptable. On the contrary, a telecom switch cannot self-reset while operating without experiencing highly related costs for downtime. (Qing Li, Caroline Yao, 2003)

Hence for these applications, different levels of reliability are required. Even though these various levels of reliability could be considered, a reliable system generally carries on providing service without failing. While the RTOS must be reliable, the system's reliability is not solely dependent on the RTOS. This reliability depends on the association of all elements in the system comprising the hardware, RTOS, BSP and application, which determines the reliability of a system. (Qing Li, Caroline Yao, 2003)

B. Predictability.

As numerous embedded systems are real-time systems, satisfying deadlines is the main point of guaranteeing proper operation. The required RTOS needs to be predictable in this case to some extent. The word deterministic identifies an RTOS with predictable performance, in which the end of OS calls happens at a predefined period. Simple benchmark programs written by developers can validate the determinism of an RTOS. The result depends on timed responses to particular RTOS calls. In a good deterministic RTOS, the difference between response times for every type of system call is negligible. (Qing Li, Caroline Yao, 2003)

C. Performance.

Performance defines how fast an embedded system is obliged to perform in order to meet deadline constraints. In other words, meeting more deadlines with a shorter interval of time between them is faster than the system's CPU is. Though the fundamental hardware influences the processing power, the software also contributes to system performance. (Qing Li, Caroline Yao, 2003)

Generally, the performance of the processor is given by million instructions per second (MIPS). With the combination of hardware and software, the overall performance of a system is measured by throughput. Throughput is defined as the rate at which the system produces output related to the incoming inputs. It is also the quantity of data transferred over the time taken to transfer them. Data transfer throughput is expressed in multiples of bits per second (BPS). (Qing Li, Caroline Yao, 2003)

Sometimes, a call-by-call basis is used by developers to measure RTOS performance. Written benchmarks produce timestamps when a system call starts and when it ends. Although this step might be helpful in the analysis stages of design, the actual performance testing is obtained at the measurement of the entire system performance. (Qing Li, Caroline Yao, 2003)

D. Compactness.

Application design constraints and cost constraints are useful to determine the compactness of an embedded system. A cell phone, for instance, clearly has to be small, portable, and cost-effective. The system memory is limited with these design criteria, leading to the application's size limitation and the OS. The RTOS has to be small and efficient in this embedded system, where there are size and cost constraints. In such cases, the RTOS memory footprint might be a crucial factor. The static and dynamic consumption of the RTOS and the running application ought to be understood by designers to meet the total system requirements. (Qing Li, Caroline Yao, 2003)

E. Scalability.

Since RTOSes are useful in different embedded systems, they have to be apt to scale up or down to meet application particular needs. An RTOS should add or delete modular parts based on the required functionality, including file systems and protocol stacks. If an RTOS fails to scale up appropriately, developers might have to purchase or construct the missing pieces if a developer wants to incorporate an RTOS for designing a cell phone project and a base station project. The RTOS should be useful for both projects at a successful scale instead of two different RTOSes, saving considerable time and cost. (Qing Li, Caroline Yao, 2003)

2.6.2.5 Tasks.

Simple software applications are generally developed to run by sequence, one Instruction at once, in a pre-defined chain of instructions. However, this structure is unsuitable for real-time embedded applications that deal with numerous inputs and outputs within limited time constraints. Real-time embedded software applications need to be developed for concurrency. Developers need to split an application into small, schedulable, and sequential program units to achieve this concurrency. The concurrent design enables system multitasking to adhere to performance and time constraints for a real-time system when correctly implemented. Most RTOS kernels offer tasks objects and task management services to ease concurrency design in an application. (Qing Li, Caroline Yao, 2003)

A task is referred to as an autonomous thread of execution that can compete with other concurrent tasks for processor execution time, as described earlier

(Qing Li, Caroline Yao, 2003). A task is schedulable. It needs to be able to compete for execution time on a system depending on a scheduling policy. A task is defined by its specific set of parameters and supporting data structures. When created, each task has a related name, a unique ID, a priority (in case of pre-emptive scheduling policy), a TCB, a stack and a task routine. These sets of parameters are referred to as task objects. (Qing Li, Caroline Yao, 2003)

Figure 19 below depicts an illustration of a task structure.

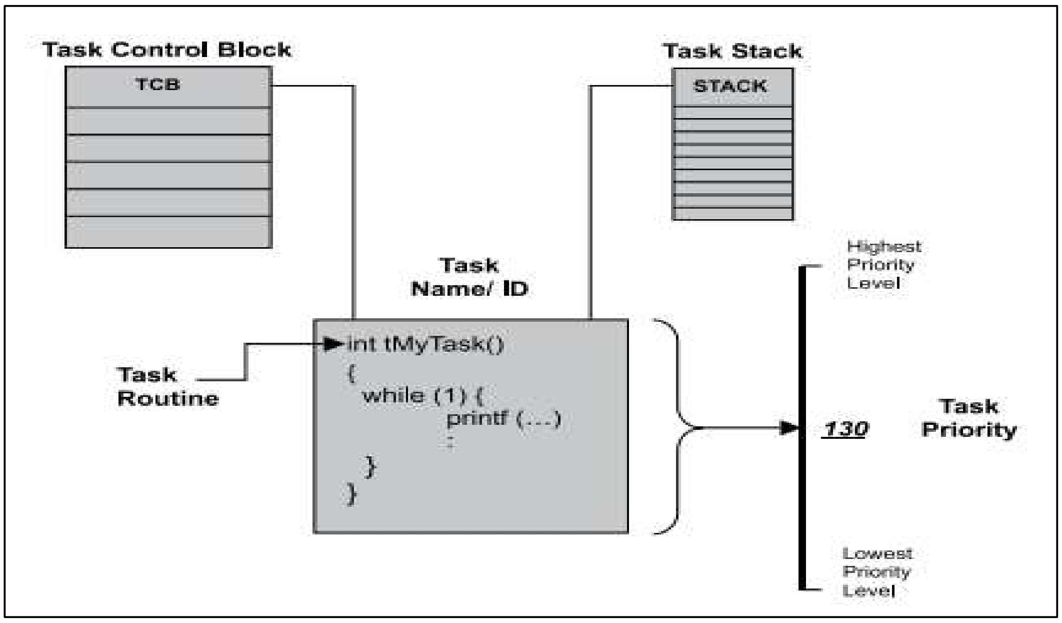


Figure 19: Structure of a Task.

At the start of the kernel, it produces its own set of system tasks and dedicates suitable priority for every system task from a set of reserved priority levels. The reserved priority level means the internal priorities used by the RTOS for its system tasks. An application should not use them for its tasks since running application tasks at such a level can influence the global system behaviour (performance). Many RTOSes do not enforce these reserved priorities. The kernel requires its system tasks and their reserved priority levels to function, and these priorities may not be changed. For instance, some illustrations of system tasks are (Qing Li, Caroline Yao, 2003):

Initialization or start-up task initializes the system, generates, and begin system tasks. (Qing Li, Caroline Yao, 2003)

Idle task: makes use of processor idle cycles in the absence of other activity. (Qing Li, Caroline Yao, 2003)

Logging task: records system messages. (Qing Li, Caroline Yao, 2003)

Exception handling task holds exceptions. (Qing Li, Caroline Yao, 2003)

Debug agent task enables debugging with a host debugger. (Qing Li, Caroline Yao, 2003)

Other system tasks may be generated during the initialisation based on the other components inserted within the kernel. (Qing Li, Caroline Yao, 2003)

The idle task generated at the kernel start-up is one system task that holds mention and may not be overlooked. It is set to the lowest priority, generally executing in an infinite loop and runs in the absence of a passing running task or at the inexistence of other tasks; this is for the sake of using idle processor cycles. The idle task is necessary as the processor runs instructions that the program counter register indicates when running. (Qing Li, Caroline Yao, 2003)

Except the processor is suspended, the program counter has still to point to valid instructions even in the absence of a task or when no task can run. Hence the idle task guarantees that the processor program counter is always available in the absence of a running task. (Qing Li, Caroline Yao, 2003)

In some instances, however, the kernel can enable a user-configured routine to operate in lieu of idle tasks to implement essential requirements for a specific application. An illustration of an essential requirement is power conservation. When no other tasks can run, the kernel can change control to the user-supplied routine instead of the idle task. The user-supplied routine, in this case, performs like the idle task and initiates power conversation code instead, like system suspension, after a period of idle time. (Qing Li, Caroline Yao, 2003)

After the kernel's initialization and creation of all necessary tasks, the kernel goes to a predefined entry point (like a predefined function) that helps in effect as the beginning of the application. Other application tasks, including other kernel objects required by the application design, could be initialized and created by the developer from the entry point. (Qing Li, Caroline Yao, 2003)

As the developer creates new tasks, it is mandatory for the developer to assign each task name, priority, stack size, and a task routine. The kernel does the rest by assigning each task a single ID and creating a related TCB and task space in memory (Qing Li, Caroline Yao, 2003).

2.6.2.6 Task states and scheduling.

A system task or an application task, at any given time, they are available in one of the states, namely: ready, running or blocked. Each task moves from one state to another regarding the logic of simple Finite Simple Machine (FSM) as the real-time embedded system runs. Figure 20 below is the illustration of FSM for task execution states with brief details of state transitions. (Qing Li, Caroline Yao, 2003)

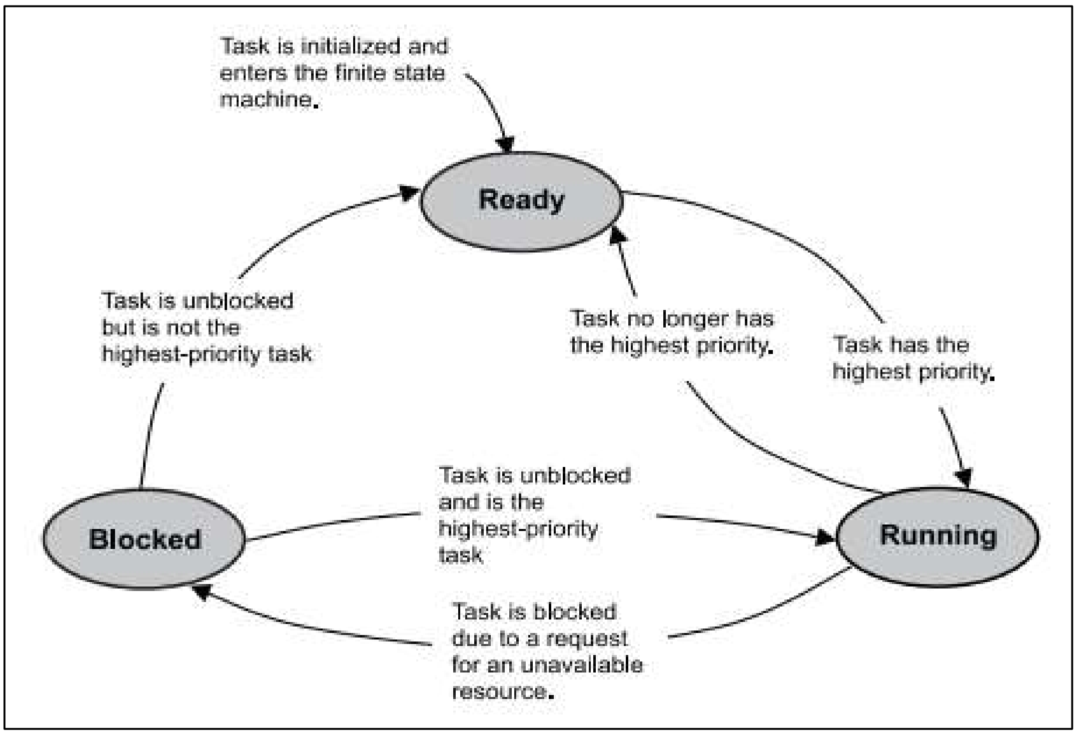

Task is initialized and enters the finite state machine.
Ready
Task is unblocked but is not the highest-priority task
Task no longer has the highest priority.
Task has the highest priority.
Task is unblocked and is the highest-priority task
Blocked
Running
Task is blocked due to a request for an unavailable resource.

Figure 20: FSM task execution states.

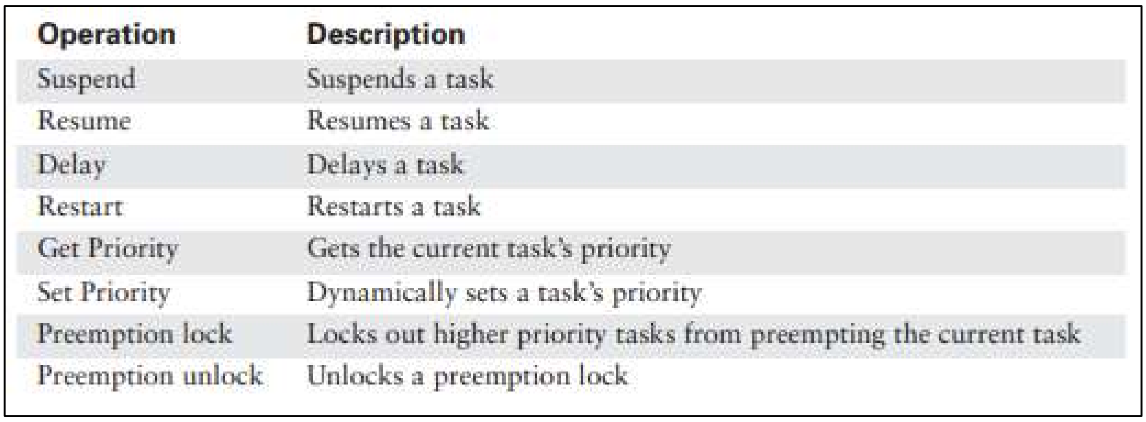
In general, three primary states are mostly used in common pre-emptive scheduling kernels (Qing Li, Caroline Yao, 2003):

- **Ready state**: the task is ready to run; however, it cannot run as a task with a higher priority is busy running. (Qing Li, Caroline Yao, 2003)

- **Blocked state:** the task has requested an unavailable resource or has requested for the occurrence of some event or has delayed itself for some time. (Qing Li, Caroline Yao, 2003)
- **Running state:** the task has the highest priority and is being executed. (Qing Li, Caroline Yao, 2003)

From the creation of a task to the time the task is deleted, the task runs through various states as a result of program execution and kernel scheduling. Even though the state changes automatically, many kernels offer a set of API calls that enable developers to track when the task moves to a changed state, as described in table 1 below. This is referred to as manual scheduling. (Qing Li, Caroline Yao, 2003)

Table 1: Different states of task operation.



Operation	Description
Suspend	Suspends a task
Resume	Resumes a task
Delay	Delays a task
Restart	Restarts a task
Get Priority	Gets the current task's priority
Set Priority	Dynamically sets a task's priority
Preemption lock	Locks out higher priority tasks from preempting the current task
Preemption unlock	Unlocks a preemption lock

Employing manual scheduling, developers can suspend and resume tasks within an application. This is probably crucial for debugging purposes or suspending a task with a high priority to allow a task that has a low priority to run, as described earlier.

A developer can opt to delay or block a task to enable manual scheduling to hold for an external event that does not include a related interrupt. The delay of a task enables the task to surrender the CPU and permits another task to execute. At the expiration of the delay, the task is sent back to the task-ready list, postal other ready tasks with its priority level. A delayed task expecting an external condition can wake up after a set time to check if a particular condition or event has happened, referred to as polling. (Qing Li, Caroline Yao, 2003)

A developer can opt to restart a task as opposed to resuming a suspended task. When restarting the task, the task starts as if its execution was not done previously. When suspended, the internal state owned by the task during this time is lost when a task is restarted. (Qing Li, Caroline Yao, 2003)

On the contrary, when a task is resumed, it keeps the same internal state as before its suspension. Restarting a task is necessary while debugging or when reinitializing the task after a major fault. While debugging, a developer can restart a task to jump into its code again from the beginning to the end. In case of a major fault, the developer can restart a task and ensure the system runs without a complete reinitialization. (Qing Li, Caroline Yao, 2003)

2.6.2.7 Embedded System resource's criteria.

Embedded systems ought to be resource-aware. The resources below need to be considered (Marwedel, 2018):

Energy: embedded systems need electrical energy to process information. This electrical energy needed is referred to as the consumed energy. Electrical energy is converted into a different form of energy, generally thermal energy. The available power and energy (the integral of power over time) remain a decisive factor for embedded systems since energy is regarded as a crucial challenge (Marwedel, 2018).

Run-time: embedded systems need to take advantage of the existing hardware architecture as much as possible. The ineffective usage of execution time, such as wasting processing cycles, need to be avoided. Hence, optimising execution time is needed at all levels, from algorithms to hardware execution (Marwedel, 2018).

Code size: generally, code needs to be stored on the system itself in some embedded systems. The storage capacity could be a tough challenge. This constraint is applicable for systems on chips, as these systems include all the information processing circuits on a single chip. If memory is inserted onto a chip, efficient usage is required as this could be implanted in a human body. Given the size and the communication constraints of these devices, the code needs to be very compact. Nevertheless, the significance of this design achievement is relative when the loaded code becomes reasonable or when the memory densities exist. (Marwedel, 2018)

Weight: all mobile systems must be light since the light weight is commonly a deciding factor for purchasing the device. (Marwedel, 2018)

Cost: competitiveness in the market is vital for high volume embedded systems. Efficient usage of hardware components and software development budget is necessary. A low number of resources could be used for implementing the necessary functionality. (Marwedel, 2018)

Given the resource awareness targes, software designs cannot be done separately from the fundamental hardware. Hence, software and hardware have to be considered at the conception step. (Marwedel, 2018)

2.6.3 MCU.

Single-chip computers with at least a microprocessor and input/output module are microcontrollers. A Microcontroller is referred to as a single-chip computer. The term Micro implies that a device is small. The term controller means that the device is utilized in control applications. Microcontrollers are also referred to as embedded controllers since they are incorporated (or embedded) into devices they control (Ibrahim, 2014).

Based on the complexity, additional components like timers, counters, interrupt control circuits, serial communication modules, ADC, digital signal processing modules are included in microcontrollers. A microcontroller ranges from a small single-chip incorporated (embedded) controller to an extensive computer system with a keyboard, monitor, printer, hard disk, etc. A microprocessor differs from a microcontroller in various manners.

The principal difference is that a microprocessor needs several extra external support chips like memory and I/O circuits before being used as a digital controller. Meanwhile, a microcontroller comprises all these support chips on one chip, hence the name of a single-chip computer. Multiple chip microprocessor-based computer systems have a higher energy consumption than microcontroller-based systems as a result. Furthermore, the OS is another difference between microprocessors and microcontrollers.

Commonly, modern microprocessors would be found without Linux, MacOS or Windows OS, for instance. On the contrary, microcontrollers are unlikely to find their use with an OS at all. The program is to run directly on the hardware without extra support. If an OS is required, it is unlikely from that on a desktop computer. This is because microcontrollers are very frequently used in real-time systems where the necessity of responding to external events within a defined period (time) is needed. (Davies, 2008)

In this case, an RTOS is utilized. The connection between a user's software and an RTOS is not the same as between software and the OS on a desktop computer. For instance, when switching on a desktop computer, the operating system must be loaded first before doing anything on the desktop. Meanwhile, a microcontroller begins with the user's software which starts the RTOS first, set its configurations and then execute the desired operations. An additional evident difference is that an RTOS may need a memory of a few hundred bytes and not megabytes. (Davies, 2008)

A single-chip microcontroller system is more cost-effective than multiple chip-based microprocessor systems. (Ibrahim, 2019) Microprocessors and microcontrollers function by processing a set of instructions (user programs) saved in the device's program memory and comprises instructions that can be read and executed by the microcontroller or microprocessor. (Ibrahim, 2019)

The instructions from the program memory are retrieved one by one, decoded, and then carried out the necessary operations by the microcontroller. (Ibrahim, 2014) Data is entered from external input modules (inputs), computed as needed and then output by an external module (outputs) under the control of a user program. (Ibrahim, 2019) In general, the assembly programming language of the target microcontroller has been used for programming microcontrollers. (Ibrahim, 2014) the assembly language is made of numerous mnemonics where each of them is an instruction to be processed by the microcontroller. (Ibrahim, 2019)

Regardless of the assembly language being very fast, it presents many drawbacks (Ibrahim, 2014) (Ibrahim, 2019). Primarily, it is challenging to learn the assembly language due to its syntax. Then, there are different sets of assembly language instructions developed by various manufacturers of the microcontrollers, which requires the programmer to learn different sets of language at each instance a different processor needs to be used. The same is true for the devices manufactured by the same manufacturers. (Ibrahim, 2019)

Finally, the maintenance of a program written in the assembly language is not easy. Even though some real-time applications are written in assembly language, nowadays, numerous applications are programmed employing High-level language like BASIC, C, C++, C#, Visual BASIC, PASCAL, JAVA … (Ibrahim, 2019)

These High-level languages are easier to learn than the assembly language as a benefit. Another benefit is that huge and complex applications can be simply and quickly developed by means of High-level languages as opposed to the assembly language. (Ibrahim, 2014), (Ibrahim, 2019) For instance, a piece of code in assembly language for multiplying two floating numbers can take more time and contain several mistakes. Meanwhile, the two numbers can easily be multiplied by High-level language. (Ibrahim, 2019)

Another benefit, the maintenance of a program written in High-level languages is made easier (Ibrahim, 2019). High-level languages offer the advantage of having the same user program that can easily be taken to function on a different device with few or no modifications. (Ibrahim, 2019) Additionally, numerous built-in libraries that simplify the development of very large programs in short times are also supported by High-level languages. (Ibrahim, 2019)

Also, the other advantage is that a program written in High-level language can be tested, which reduces the development time. (Ibrahim, 2019) Generally, a single chip is the only requirement to have an embedded system running. However, other components might be necessary to enable the MCU to interact with its environment in practical applications. (Davies, 2008) A typical example is shown below in Figure 21 (Ibrahim, 2010):

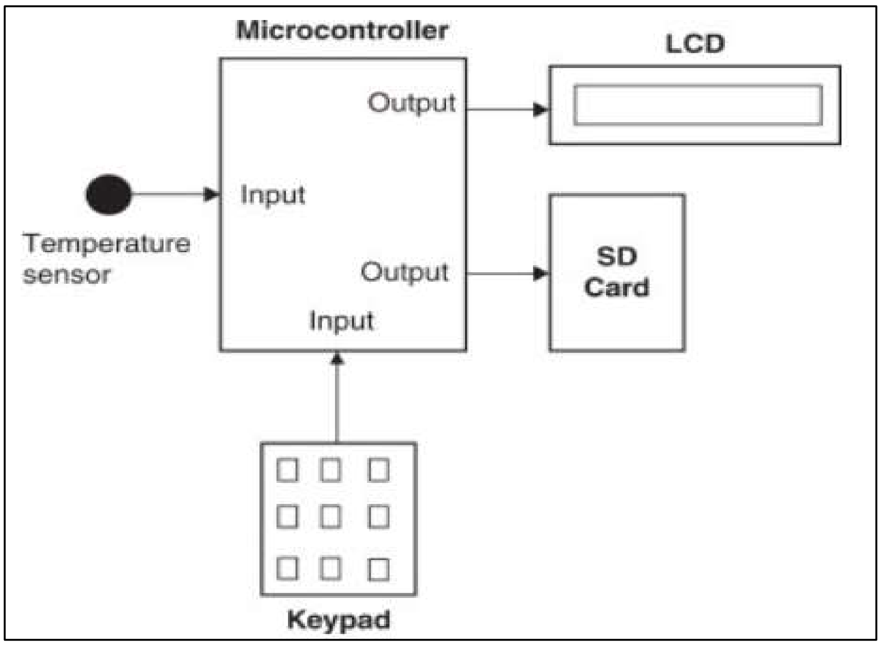


Figure 21: Temperature Data Logger System with a Keypad and LCD.

2.6.3.1 Typical anatomy of MCU.

The standard features found in different MCUs are listed below (Davies, 2008):

Central Processing Unit: this unit includes (Davies, 2008):

- Arithmetic Logic Unit (ALU): which computes. (Davies, 2008)
- Registers required for the basic operation of the CPU, such as the program counter (Pc), stack pointer (SP), and status register (SR). (Davies, 2008)
- Further registers to hold temporary results. (Davies, 2008)
- Instruction decoder and other logic to control the CPU, handle resets, and interrupts, etc. (Davies, 2008)

Memory for the program: this is a non-volatile memory, known as ROM; this stores the data where there is no power supply. (Davies, 2008)

Memory for data: This is a volatile memory, referred to as RAM. (Davies, 2008)

Input and output ports: to enable the digital communication of the MCU with the outside environment. (Davies, 2008)

Address and data buses: allow the internal connection between subsystems to exchange data and instructions. (Davies, 2008)

Clock: Maintain the entire system synchronized. The clock signal can be generated internally or externally from an external source or a crystal. Current MCUs allow a substantial selection of clocks. (Davies, 2008)

Figure 22 below illustrates the anatomy of most MCUs (Davies, 2008).

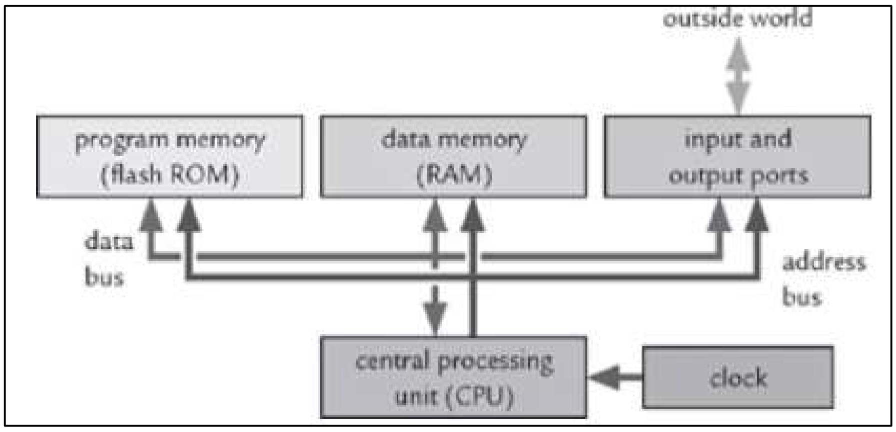


Figure 22: Anatomy of MCU.

Most processors would likely have these features, even though their implementation may be significantly different. A huge difference between MCUs originates from the variety of peripherals available. Some time ago, these features required distinct pieces of equipment. However, with the growth of technology, the features could be contained on the same Printed Circuit Board (PCB) with the processor. Most of these peripherals are currently included on the same Integrated Circuit (IC) with the processor and a different classification. (Davies, 2008)

2.6.3.2 Architecture of MCU.

In general, two sorts of architecture are used in MCU: Von Neumann architecture and Harvard architecture. Most MCUs make use of the Von Neumann architecture. In this architecture, the same bus is used for memory space, and the instruction and data also utilize the same bus. In the Harvard architecture, however, the software code and data are located on distinct buses. This enables the code and data to be retrieved concurrently and provide improved performance as a result. Figures 23 and 24 below illustrate the two common architectures for the MCU. (Ibrahim, 2010)

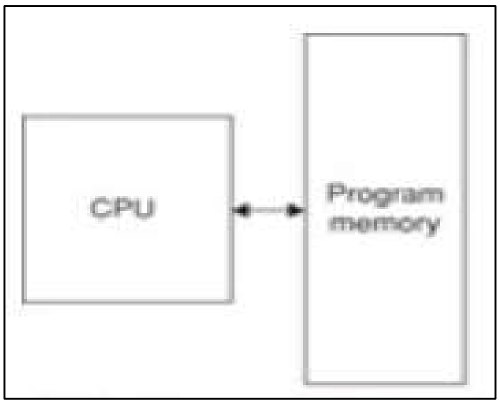


Figure 23: Von Neumann Architecture.

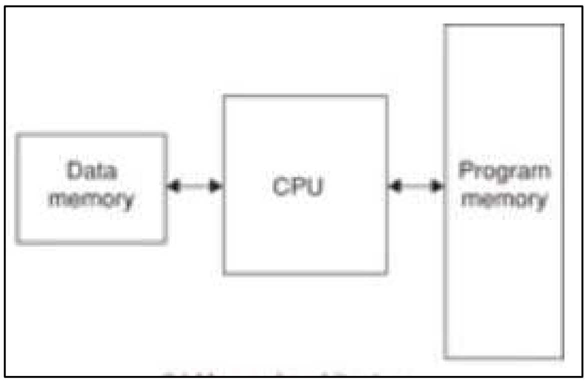


Figure 24: Harvard Architecture.

2.6.3.3 Instruction set of MCU.

Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) are MCU instruction sets. For an 8-bit RISC MCU, data is 8 bits wide while the instruction words are larger than 8 bits wide and can typically be 12,14 or 16 bits). Moreover, the instructions fill one word in the program memory allowing the instructions to be retrieved and computed in a single cycle. This results in improved performance. On the other hand, for a CISC MCU, the data and instructions are both 8 bits wide. Also, CISC MCU generally contains more than 200 instructions. The same bus contains the data and the code which prevent them from being fetched concurrently. (Ibrahim, 2010)

2.6.3.4 Bits MCU (8, 16,32).

Deciding between the bits of MCUs is generally confusing. It is critical to notice that the number of bits simply means the size (width) of data handled by the CPU. The precision of mathematical computations carried out by the processor is restricted by the number of bits, even though emulating high-order mathematics in software or by the utilization of special hardware is feasible. (Ibrahim, 2015)

8-bit MCUs have existed since the early days of MCU development. They are affordable, simple to use as they are in small package size, low speed, and necessary in most general-purpose control and data manipulation computations. For instance, it is very effective to design low-to medium-speed control systems such as fluid level control, temperature control or robotics applications with 8-bit MCUs. The low cost is more vital than the high speed. Furthermore, numerous industrial and commercial applications belong to this category and can simply be designed utilizing normal 8-bit MCUs. (Ibrahim, 2015)

In contrast, 16 and 32-bit MCUs are generally expensive but give higher speeds and higher precision in mathematical computations. These MCUs are generally cased in a bigger package such as 64 or 100 pins, for instance, and present much more features like larger data and program memories, more and faster Analog to Digital (A/D) channels, more timer/counter modules, more I/O ports, etc. 32-bit MCUs found their applications generally in high-speed, real-time, and digital signal processing applications which require high precision.

These applications are digital image processing and digital audio processing, for example. Numerous consumer products like mobile phones and electronic games are designed on 32-bit MCUs as they need high-speed real-time computation, colour graphical displays, and touchscreen panels. Furthermore, additional high-speed applications such as image filtering, video capturing, video streaming, speech recognition, video editing, and speech processing necessitate high-speed 32-bit processors with more data and program memories. They also require high precision while executing the digital signal processing algorithms. (Ibrahim, 2015)

2.6.3.5 ARM MCUs.

A. History.

ARM processor history is fascinating. In 1981, the British Broadcasting (BBC) called the computer industries to produce a computer for educational projects. Acorn Computers Ltd won, and they designed a home computer around the 8-bit MCU 6502, which the firm MOS Technology manufactured. Though in nowadays standards, the 6502 was not a powerful microprocessor.it presented a high speed in the early days and became very popular in the 1980s. (Ibrahim, 2019)

Christopher Curry and Herman Hauser established Acorn Computers Ltd. Christopher worked closely with Clive Sinclair of Sinclair Radionics Ltd for more than ten years. Sinclair encountered some financial issues and founded another company named Sinclair Research Ltd where Christopher was one of the key people in Sinclair Research Ltd. (Ibrahim, 2019)

After a disagreement with Sinclair, Christopher left the firm to start Acorn Computers Ltd with the Austrian physicist Herman Hauser. In the 1980s, IBM manufactured their first Personal Computer (PC), established on 16-bit microprocessor 8088. IBM became very famous in the PC industry with the primitive MSDOS OS, and many small competitors vanished. The 8-bits Acorn 6502 was not powerful enough for graphics-based applications. (Ibrahim, 2019)

Acorn decided that they required a new architecture with a higher speed processor and contemplated inventing their own processor. In 1990, Acorn realized that their future depended not on selling computers alone but on developing new computer architectures. So, in 1990 a new company called Advanced Reduced Instruction Set Computer (RISC) Machines (ARM) Ltd was founded by Acorn Computers Ltd. This new company was a joint venture with Apple Computer and VLSI technology. (Ibrahim, 2019)

Apple invested cash, VLSI supplied the needed technology tools, and Acorn supplied experienced design engineers in this venture. By the end of the year, ARM turned into a £ 26.6 million company with a net income of £ 2.9 million. In 1998, currently named ARM Holdings appeared on the London Stock Exchange and NASDAQ list. Many years later, ARM designed new processor architectures and sold the intellectual property rights (licences)to the companies who desired to include ARM designs in their products. In the year 2016, ARM was bought by SoftBank for $ 31 billion. For over two decades, ARM has been designing 32-bits. (Ibrahim, 2019)

ARM started also developing 64-bit designs in the last few years. ARM is a firm dedicated to designing processor architecture and does not produce and sell processor chips. ARM earns money by selling its license designs to chip manufacturers. The core ARM processors are used and integrated with their peripherals to end up in a whole microcontroller chip. Each chip manufactured by third-party companies gives royalty fees to ARM. Apple, Atmel, Cypress, Broadcom, Cypress Semiconductors, Analog Devices, Nvidia, NXP, Samsung Electronics, Freescale Semiconductors, Texas Instruments, Qualcomm, Renesas, for instance, use the ARM core processors. (Ibrahim, 2019)

The concept of ARM became famous in mobile applications. In 2005, nearly 98% of all mobile phones sold made use of at least one ARM processor. In 2011, the 32-bit architecture was widely distributed and became the architecture of choice for mobile devices. In 2013, over 10 billion processors were produced. The ARM architecture presents the best MIPS to Watts ratio and the MIPS to US Dollard’s ratio in the industry. It also offers the smallest size of CPU. (Ibrahim, 2019)

Over the years, many cores ranging from 32-bits to 64-bits, such as the ARMv1, ARMv2, ARMv8, etc., have been designed. Mobile devices mostly made use of the 32-bit ARMv7-A in mobile devices in 2011. Architectures like Cortex-A5, Cortex-A7, Cortex-A8 were also widely used. The licence fees are proportional to the cores' performance; the lower performing cores cost a lower license fee than higher performing cores. Lower performing cores use The ARMv6-M and ARMv7E-M, and the higher performing cores use the Cortex-A5, Cortex-A7 and Cortex-A8 (Ibrahim, 2019).

B. ARM processor architecture.

Nowadays, most processors are known as RISC since they contain limited instructions sets, as described above. ARM processors are based on RISC designs and are commonly used, specifically in mobile devices. RISC processors use few transistors and reduce the power consumption due to their more straightforward design. These processors offer high-speed processing as they execute one instruction per cycle. (Ibrahim, 2019)

However, more instructions are required for a given task. The instructions are retrieved from the memory and are processed using one cycle each at high speed. (Ibrahim, 2019)

RISC-based processors designs enable many pipelining levels, whereas the next instructions are retrieved from the memory while the present instruction is being computing, resulting in higher throughput. (Ibrahim, 2019)

With their more straightforward design, these RISC-based processors have low power consumption, critical for battery-powered mobile applications. thus, the ARM architecture is commonly used in mobile applications with very high-speed processing times and low power consumption. (Ibrahim, 2019)

C. Selection of ARM.

The selection of an MCU for a specific task application lies on many factors like power consumption, cost, speed, size, number of digital and analogue input-output ports, digital input-output current capacity, antilog port resolution and accuracy, program and data memory sizes, interrupt support, timer support, Universal Synchronous/Asynchronous Receiver/Transmitter (USART) support, bus support (USB, CAN, SPI and I2c), ease of system development (programming), operating voltage. (Ibrahim, 2019)

As an illustration, to develop a battery-powered mobile device such as mobile phones or game consoles, the high-clock speed and long battery life are the principal criteria. However, a high-speed clock or low power consumption is not the main requirement for a liquid level control system. Generally, the power consumption is proportional to the power consumption, and this imposes a trade-off to be made in selecting an MCU for a given application. (Ibrahim, 2019)

ARM-based processors are established on Thumb (an instruction set). This instruction set with clever designs gets 32-bit instructions and compresses to 16-bits. Hence the hardware size is compacted (decreased), which reduces the total cost and the power consumption. (Ibrahim, 2019) Additionally, the ARM processor uses multistage pipelined architecture that is simpler to learn, build and program. (Ibrahim, 2019)

ARM's core architecture is strictly a processor and does not comprise graphics, input-output ports, serial communication, USB, wireless connectivity, or any other form of peripheral modules. Systems are built around ARM core by chip manufacturers. This implies the reason why different manufacturers present various sorts of ARM-based MCUs. (Ibrahim, 2019)

D. ARM Family.

Several 32-bit processors have been developed by ARM over the last two decades. ARM decided around 2003 to advance their market share as they developed new series of high-performing processors specifically for MCUs based applications like embedded control and monitoring applications. This led to the creation of the Cortex family of processors. This family is made of three processor families, which are: Cortex-M, Cortex-R, and Cortex-A. Figure 25 below gives an overview of some of the ARM processor families. (Ibrahim, 2019)

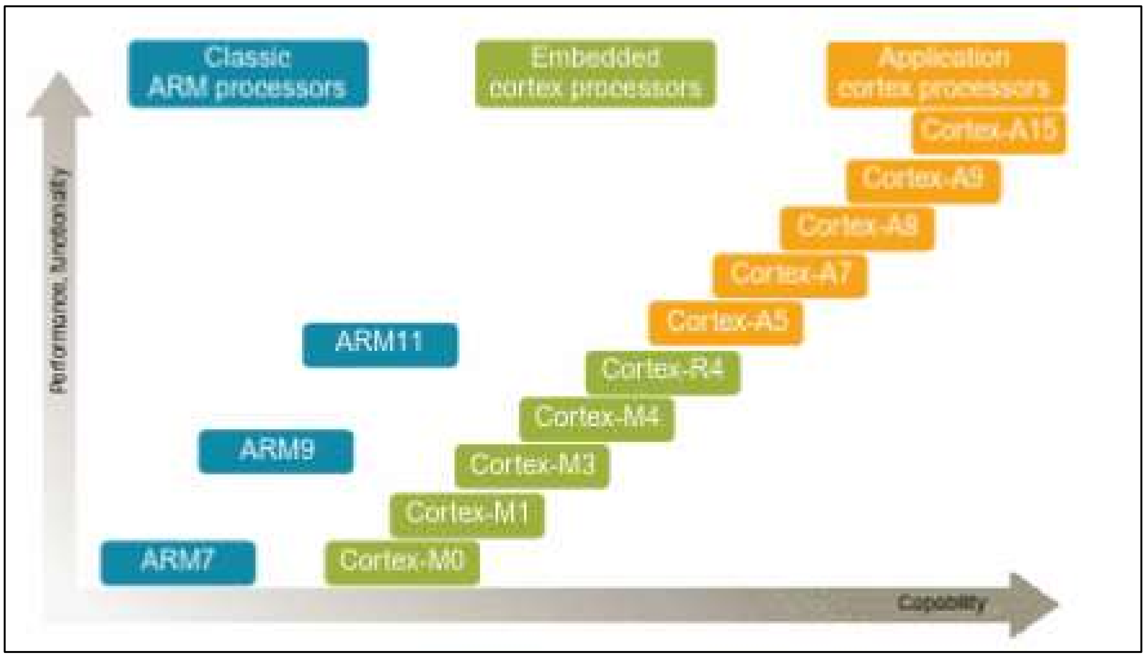


Figure 25: ARM Processor Family.

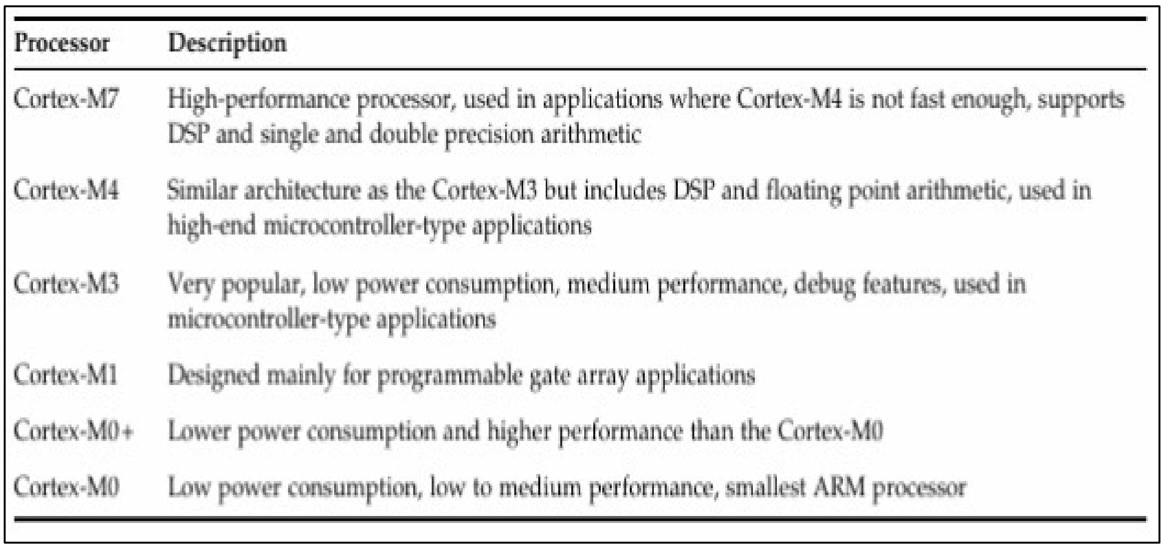
1. Cortex-M.

The Cortex-M families are designed around the ARMv6-M architecture (Cortex-M0 and Cortex-M0 +) and the ARMv7-M architecture (Cortex-M3 and Cortex-M4). Their primary purpose is for the MCU market, presenting fast and deterministic interrupt responses, low cost, low power consumption, reasonably high performance, and ease of use. The Cortex-M3 and Cortex-M4 have identical architecture and the same instruction sets (Thumb 2). However, the Cortex-M4 differs from the Cortex-M3 as it presents Digital Signal Processing (DSP) capabilities and includes an optional Floating-Point Unit (FPU). With the DSP and FPU capabilities, the Cortex-M4 is an ideal processor for the Internet of Things and mobile applications. The Cortex-M0 or the Cortex-M0+ could be used for cost-sensitive and lower performance applications. (Ibrahim, 2019)

The Cortex-M0 processor contains a small gate count (12k gates) with a power consumption of 12.5 µW/MHz only. The cortex-M0+ series has a power consumption of only 9.85 µW/MHz and is established on a subset of the Thumb 2 instruction set. It presents a performance higher than the one of Cortex-M0 and under that of the Cortex-M3 and Cortex-M4. Cortex-M7 series are high-performance processors that can allow fast DSP and single or double precision floating point operations. Their applications are primarily found where higher performance than of the Cortex-M4 is needed. (Ibrahim, 2019)

Table 2 below gives a summary of the Cortex-M families. (Ibrahim, 2019)

Table 2: Summary of Cortex-M families.



Processor	Description
Cortex-M7	High-performance processor, used in applications where Cortex-M4 is not fast enough, supports DSP and single and double precision arithmetic
Cortex-M4	Similar architecture as the Cortex-M3 but includes DSP and floating point arithmetic, used in high-end microcontroller-type applications
Cortex-M3	Very popular, low power consumption, medium performance, debug features, used in microcontroller-type applications
Cortex-M1	Designed mainly for programmable gate array applications
Cortex-M0+	Lower power consumption and higher performance than the Cortex-M0
Cortex-M0	Low power consumption, low to medium performance, smallest ARM processor

The Cortex-M0 and Cortex-M0+ are for low speed and low-power applications. The Cortex-M1 is enhanced for programmable gate array utilizations. The Cortex-M3 and Cortex-M4 (medium-power processors) found their usage in MCU applications, and the Cortex-M4 enables DSP and FPU arithmetic computations. The Cortex-M7 is a high-performance processor and found its applications where higher performance than the Cortex-M4 is necessary. (Ibrahim, 2019)

Figure 26 below indicates a simple block diagram of the Cortex-M4 processor architecture. (Ibrahim, 2019)

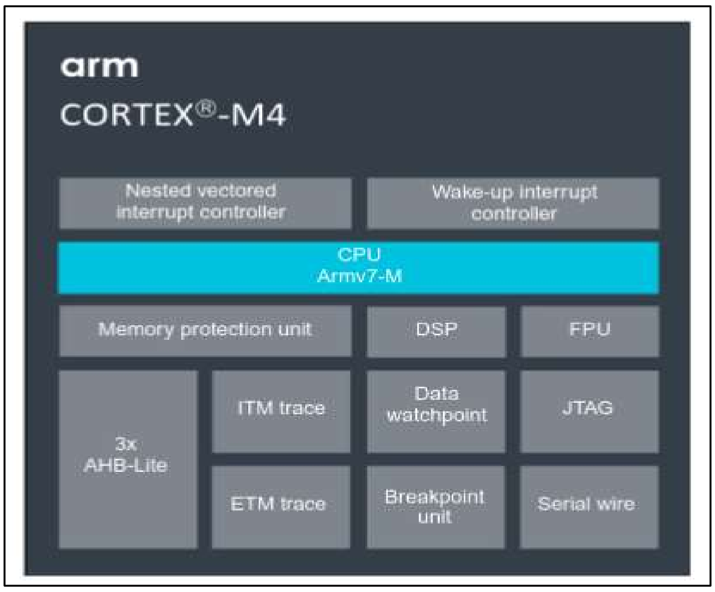


Figure 26: Simple block diagram of Cortex-M4 processor.

Next to the top left-hand side, a CPU contains the ALU, register banks, instruction decoder, DSP module and the memory interface. The DSP module is absent in the Cortex-M0 or Cortex-M3 processors. Also, the memory does not belong to the Cortex-core, and it is provided by the company manufacturing the processor. (Ibrahim, 2019)

The nested vectored interrupt controller is comprised at the top side of the picture. These nested vectored interrupt controllers are external interrupt inputs that are necessary for interrupt-driven real-time applications. An FPU is included in the core for speeding the floating point during mathematical computations. The test and debug are represented at the bottom right side of the illustration. The bus matrix, memory protection unit, and peripheral interfaces are also situated at the bottom right side of the image. The core does not have the inputs-outputs ports. (Ibrahim, 2019)

2. Cortex-R.

The Cortex-R families offer real-time high-performance processors than the Cortex-M. Some processors in the Cortex-R families are designed to operate high-clock rates exceeding 1 GHz. These processors are found in hard-disk controllers, network devices, automotive applications and specialized in high-speed MCU applications. (Ibrahim, 2019)

Cortex-R4 and Cortex-R5 are the primitive members of this family and operate at a clock speed of up to 600 MHz. The latest Cortex-R7 includes 11 stage pipelines for high-performance and can operate exceeding 1 GHz. Despite the high performance of Cortex-R processors, they present a complex architecture, and their processors have a high-power consumption which prevents their applications in wearable battery-powered devices. (Ibrahim, 2019)

3. Cortex-A.

This family has the highest performance ARM processors designed for RTOS in mobile applications. These processors provide advanced features for OS like Linux and Android, for instance. Furthermore, the Cortex-A family supports advanced memory management with virtual memory. The Cortex-A5 to Cortex-A17 processors were the first processors of this family. (Ibrahim, 2019)

They were based on the ARMv7-A architecture. The Cortex-A50 and Cortex-A72 series are the newest processors designed for low-power and high-performance mobile applications. They also built with the ARMv8-A architecture, which presents 64-bits energy-efficient processing and offers more than 4 GB of physical memory. (Ibrahim, 2019)

2.6.3.6 ARM core based MCU.

Many ARM-core-based development boards exist; this section only considers the Cortex-M-based development boards used in MCU-based control and monitoring applications and are Mbed compatible. Figure 27 below illustrates the picture of an ARM-core (Cortex-M) MCU. (Ibrahim, 2019)

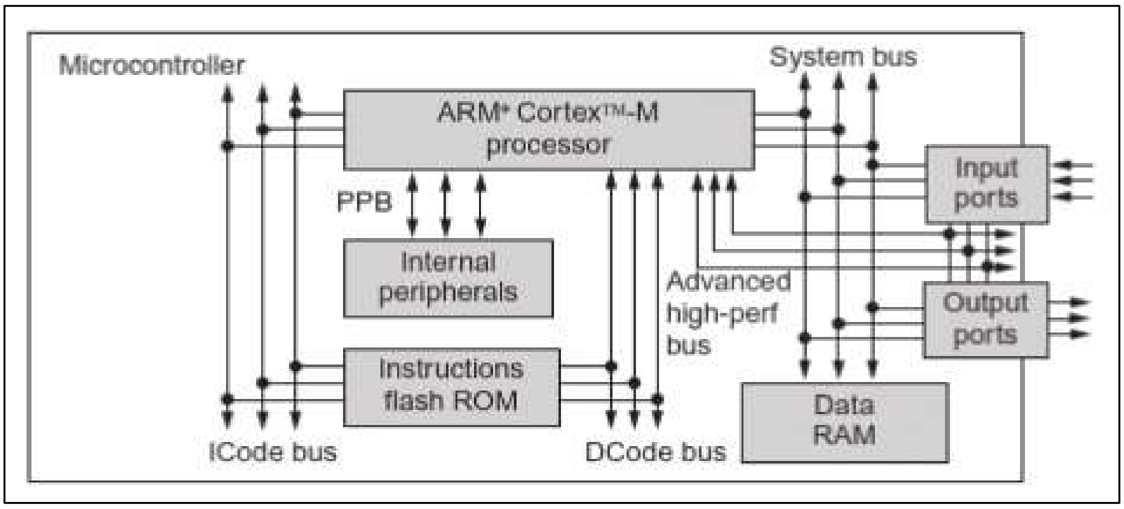


Figure 27: ARM - Core (Cortex-M) MCU.

2.6.4 STM32 Family.

The STM32 Nucleo boards consist of high-performance and low-cost development boards based on cutting-edge architecture of the 32-bit ARM Cortex-M. These boards are widely used by learners (students), hobbyists, and professional engineers at every levels. These boards can suit the famous Mbed, Arduino, ST-LINK, and ST Morpho, enabling users to familiarize themselves with the boards. Also, despite the additional hardware extension modules, The STM32 Nucleo family is compatible with many software tools development and Integrated Development Environments (IDEs) like debuggers, professional compilers, and in-circuit programming tools. (Ibrahim, 2019)

This family contains over 30 various boards to satisfy the needs of the users. Generally, the Nucleo boards are in various sizes, namely Small (Nucleo-32), Short (Nucleo-64) and Long (Nucleo-144), where the numbers indicate the number of pins of the MCUs used on the boards. (Ibrahim, 2019). Furthermore, these three types are categorized into three groups which are: the ultra-low power (green colour), the mainstream (blue colour) and the high performance (magenta colour). (Ibrahim, 2019)

The ultralow power boards are found on the STM32 L family and used for small power applications like watches and smart meters. NucleoL433RC-P, Nucleo-L432KC, Nucleo-L031K6 and Nucleo-L011K4 are among the STM32 L family. The STM32 L family is subdivided into three groups: L0 based on ARM Cortex-M0, L1 based on ARM Cortex-M3 and L4 based on ARM Cortex-M4. (Ibrahim, 2019)

The mainstream class consists of nearly around half of the STM32 Nucleo boards: Nucleo-F303RE, Nucleo-F042K6 and Nucleo-F303K8 are examples of this class. The mainstream is subdivided into three categories: F3 based on ARM Cortex-M4, F1 based on ARM Cortex-M3 and F0 based on ARM Cortex-M0. (Ibrahim, 2019)

The high-performance boards include huge memory sizes along with high-speed MCUs. Nucleo-F722ZE, Nucleo-F401RE and Nucleo-F410RB are examples of high-performance boards. The high-performance category is also subdivided into three categories: F2 based on ARM Cortex-M3, F4 based on ARM Cortex-M4 and F7 based on ARM Cortex-M7. (Ibrahim, 2019)

Figure 28 below illustrates the STM32 Nucleo development boards.

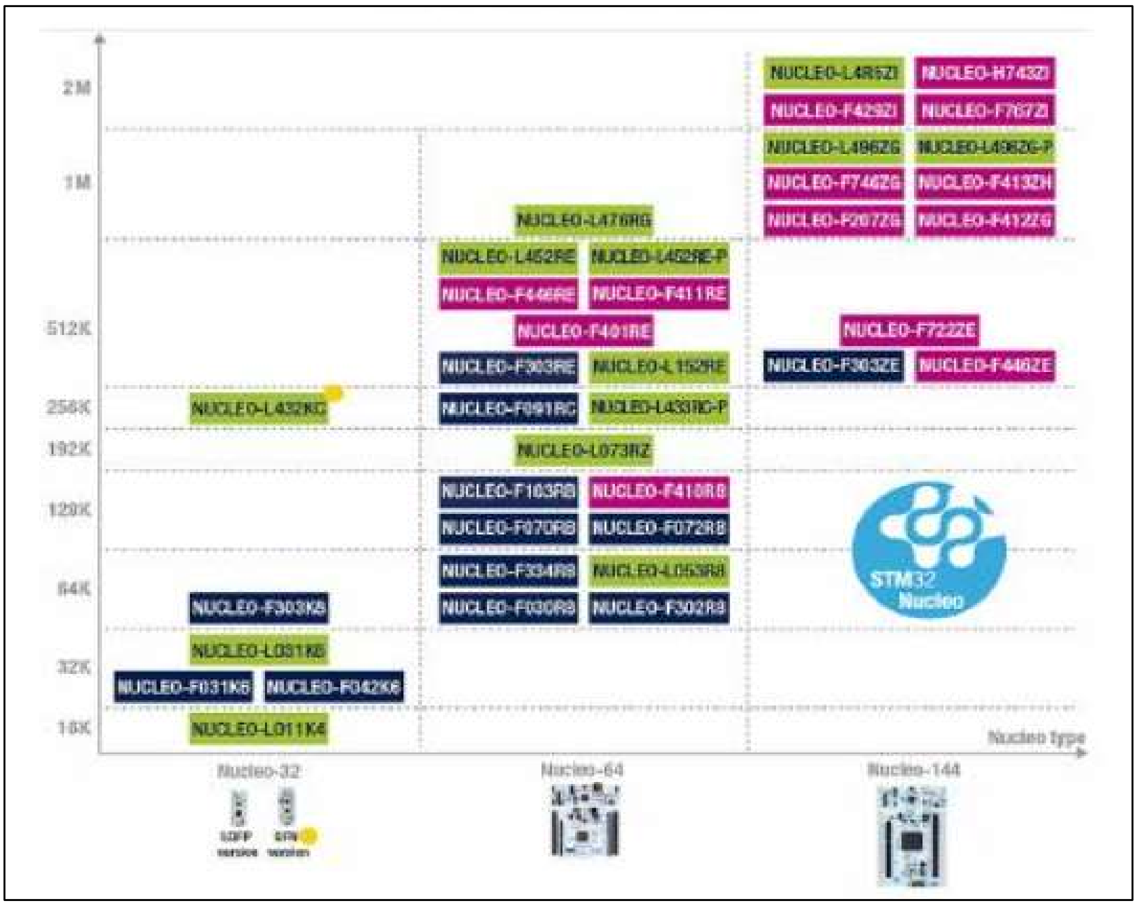


Figure 28: STM32 Nucleo development boards.

A. Nucleo-32 Board Development.

The Nucleo-32 board is nano compatible with many Arduinos nano shields utilized when using the board. The Nucleo-32 board contains the features below (Ibrahim, 2019):

- 32MHz Cortex M0+ microcontroller in 32-pin package. (Ibrahim, 2019)
- 32KB flash memory. (Ibrahim, 2019)
- 8KB RAM. (Ibrahim, 2019)
- 1KB EEPROM. (Ibrahim, 2019)
- Real-time clock. (Ibrahim, 2019)

- Serial interfaces (USART, SPI, and I2C). (Ibrahim, 2019)
- 3 LEDs (USB communication, power, and user). (Ibrahim, 2019)
- Push-button Reset. (Ibrahim, 2019)
- Flexible power-supply options: ST-LINK USB VBUS or external sources. (Ibrahim, 2019)
- Arduino Nano compatible expansion connector. (Ibrahim, 2019)
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port. (Ibrahim, 2019)
- Support for IDE software (IAR, Keil, ARM Mbed, and GCC-based IDEs). (Ibrahim, 2019)

Figure 29 illustrates a Nucleo-32 development board:

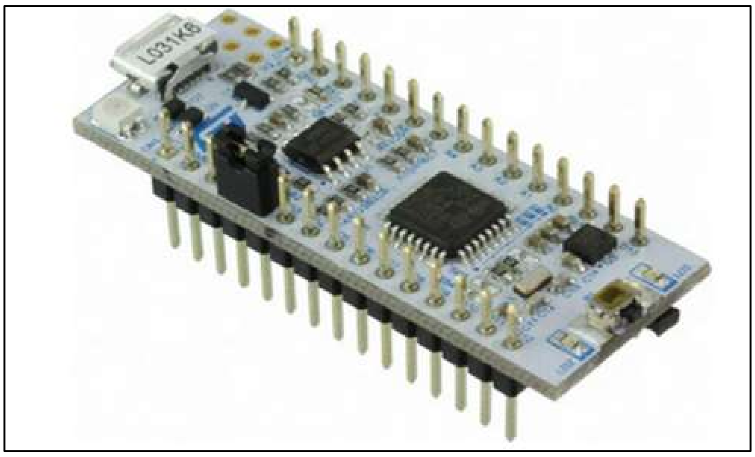


Figure 29: Nucleo-32 development board.

B. Nucleo-64 Development board.

Figure 30 below illustrates a sample of a nucleo-64 development board, the Nucleo-F091RC. The board is referred to as a mainstream board with a 64-pin MCU. This board is compatible with Arduino Uno, and a considerable amount of Arduino Uno shields can be utilized with this board. The board includes the following features (Ibrahim, 2019):

- 1 User LED. (Ibrahim, 2019)
- 1 User push-button switch. (Ibrahim, 2019)
- 32.768 kHz crystal oscillator. (Ibrahim, 2019)
- ST morpho connecter. (Ibrahim, 2019)

- Arduino Uno expansion socket. (Ibrahim, 2019)
- Flexible power-supply options: ST-LINK USB VBUS or external sources. (Ibrahim, 2019)
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port. (Ibrahim, 2019)
- Comprehensive free software libraries. (Ibrahim, 2019)
- Support of a wide selection of IDE software such as IAR, Keil, ARM Mbed, and GCC-based IDEs. (Ibrahim, 2019)

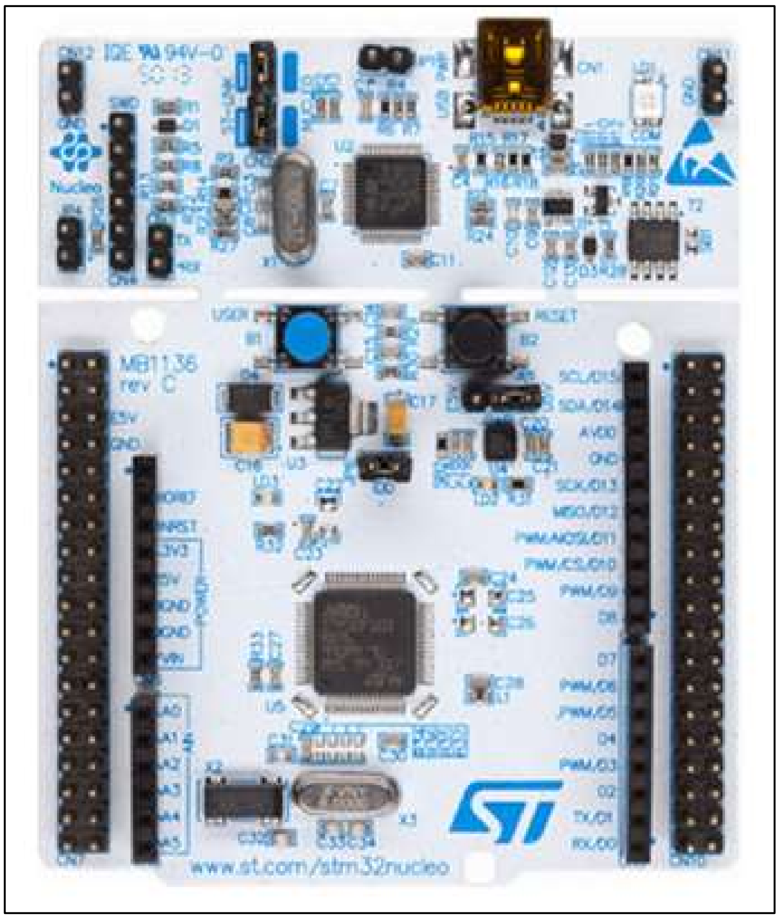


Figure 30: Nucleo-64 Development board.

C. Nucleo-144 Development Board.

The Nucleo-144 board development board is a high-performance board containing a 144-pin MCU. The board includes the features below (Ibrahim, 2019):

- Ethernet compliant with RJ45 connector. (Ibrahim, 2019)
- ST morpho connector. (Ibrahim, 2019)
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port. (Ibrahim, 2019)
- ST Zio connector. (Ibrahim, 2019)
- Three user LEDs. (Ibrahim, 2019)
- Two push-button switches. (Ibrahim, 2019)
- 32.768 kHz crystal oscillator. (Ibrahim, 2019)
- Flexible power-supply options: ST-LINK USB VBUS or external source. (Ibrahim, 2019)
- Comprehensive free software libraries. (Ibrahim, 2019)
- Support of a wide choice of IDE software (IAR, Keil, ARM Mbed, and GCC-based IDEs). (Ibrahim, 2019)

Figure 31 below illustrates a sample of a Nucleo-144 development board. (Ibrahim, 2019)

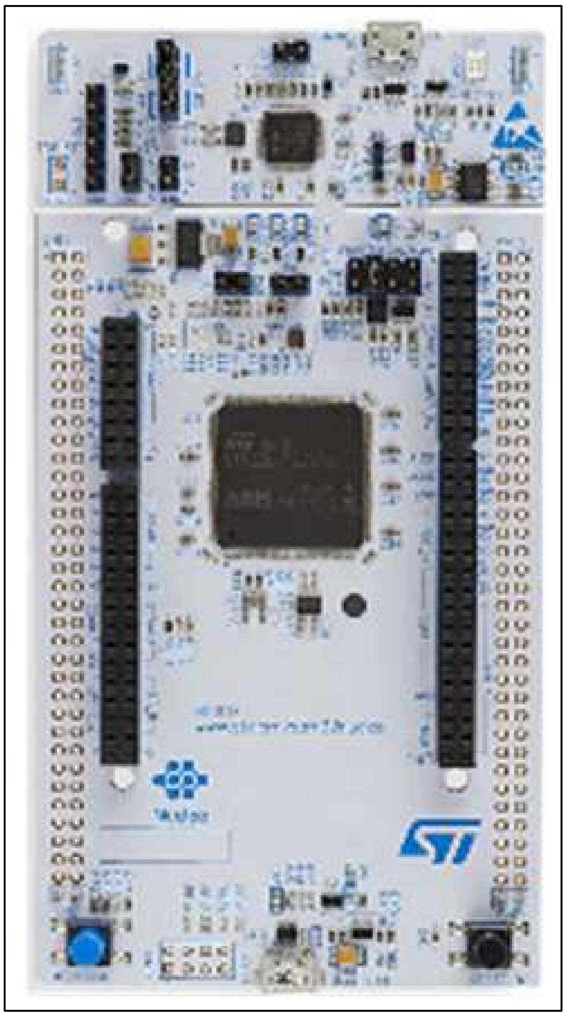


Figure 31: Nucleo-144 Development board.

2.6.4.1 Selection of the Nucleo-64 STM32F303RE development board.

The emphasis on ARM Cortex-M MCUs is justified since devices based on this core presents a useful set of features. These features allow students, hobbyists, and professionals to design medium to high complex real-time embedded systems using RTOS to develop solutions to have modules that are portable to other projects. (Amos, 2020)

Among the ARM Cortex-M family, the ARM Cortex-M4 core presents DSP capabilities and comprise FPU. These capabilities enable the Cortex-M4 for the internet of things and mobile applications, as stated earlier. (Ibrahim, 2019)

Furthermore, the Cortex-M4 core is designed to solve the digital signal control markets that require efficient and simple control and signal processing features. These high-efficiency features combined with the low power, low price and ease of use to satisfy many fields like motor control, power management, industrial automation. (Arm, 2020)

Since the thesis specifically addresses the energy consumption in real-time embedded systems, the Cortex-M4 has been opted for choosing the STM32 development board. The STM32 mainstream development board offers a wide range of high-volume applications, which is cost-effective and performance balanced.

Among this family, the STM32 Nucleo-64 board is an affordable and flexible way for experimenting with new designs and build prototypes based on a variety of combinations of performance and power consumption features offered by this family of boards (STMicroelectronics, 2020).

The STM32F303RE Nucleo-64 board available at the time of this experiment belongs to the mainstream family. This board includes the STM32F303RE MCU based on an ARM-Cortex M4 core suitable for real-time applications with high performance at low power consumption. (Ibrahim, 2019), (STMicroelectronics, 2020), (STMicroelectronics, 2020).

2.6.4.2 Nucleo-64 STM32F303RE Development Board.

The Nucleo-64 STM32F303RE is made of two sections: the ST-LINK section and the STM32F303RE MCU section. The ST-LINK section includes the micro-USB port and the programming /debugging interface. In order to reduce the board size, the ST-LINK section of the PCB could be desired if necessary. In this case, the MCU section is powered by VIN, E5V and 3.3 V on the CN7 connector of the ST Morpho or by VIN and 3.3 V connector CN6 (user manual). On the other hand, the MCU section includes the MCU, two pushbuttons, LEDs, Arduino, ST morpho connectors, power controller and the crystal. (STMicroelectronics, 2020)

When the ST-Link is removed, the MCU can be programmed by connecting wires between the connector CN4 on the ST-LINK board and the Serial Wire Debug (SWD) signals present on the CN7 (ST Morpho connector). Figure 32 below illustrates the two parts of the STM32F303RE Nucleo board, while figure 33 depicts the physical Nucleo-64 STM32F303RE. (STMicroelectronics, 2020)

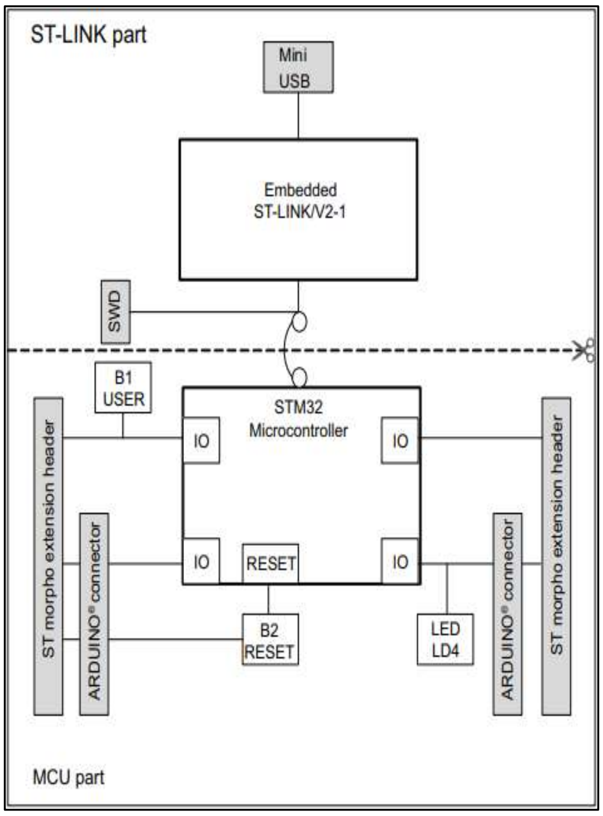


Figure 32: Nucleo-STM32F303RE.

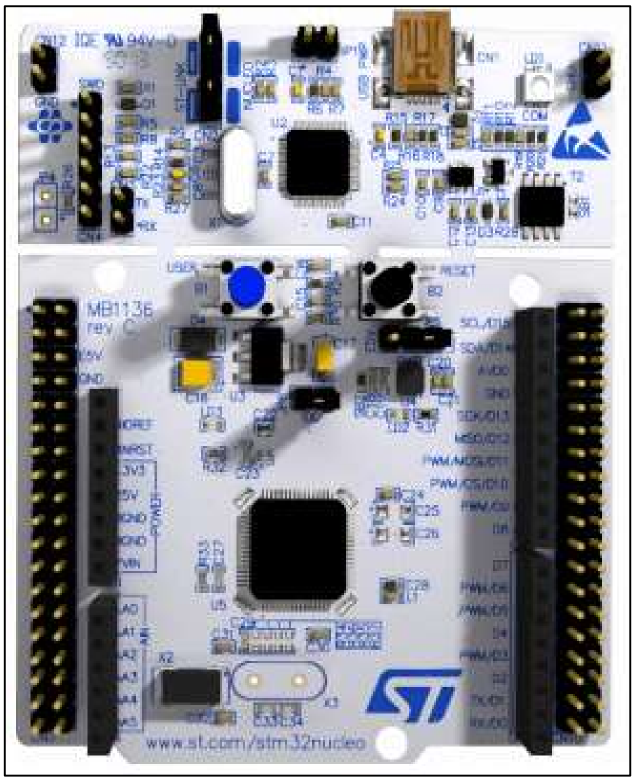


Figure 33: Nucleo-64 STM32F303RE.

The figures (Figure 34 and Figure 35) below describe the components in the top and bottom views of the Nucleo-64 STM32F303 RE board. (STMicroelectronics, 2020)

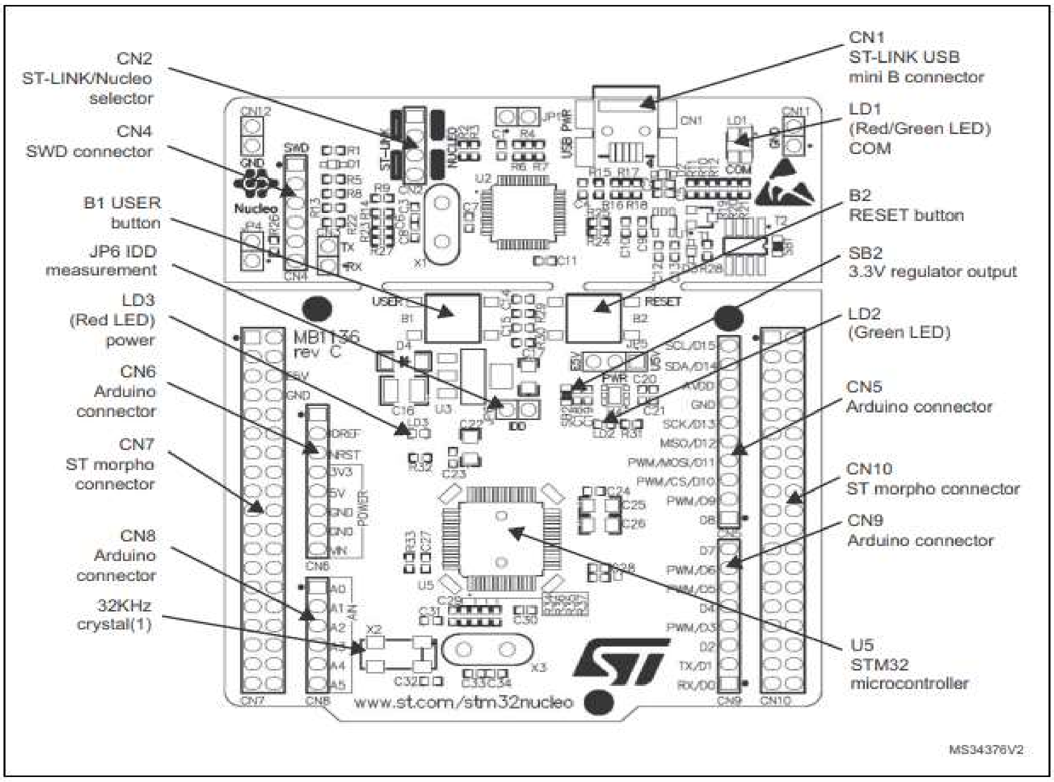


Figure 34: Top view of Nucleo-64 STM32F303RE.

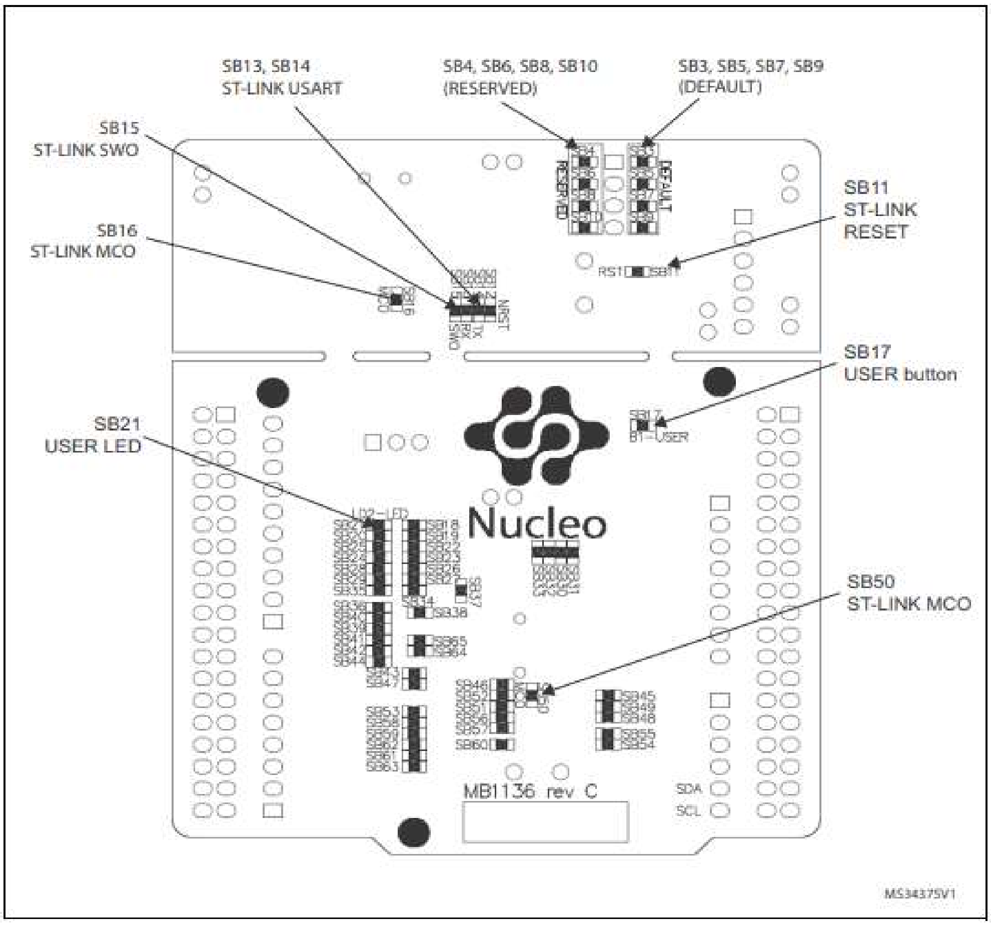


Figure 35: Bottom view of Nucleo-64 STM32F303RE.

A. The power supply.

The power supply of the Nucleo-64 STM32F303 RE is provided by the PC using a USB cable or by using an external source connected in VIN pin for 7-12 V, E5V pin for 5 V or via +3.3 V pins on connectors CN6 or CN7. An external DC power supply unit is necessary when the VIN, E5V or +3.3 V is used to supply power to the board. The power supply unit must comply with the EN-60950-1: 2006+A11/2009 standard and ought to be Safety Extra Low Voltage (SELV) with limited power capability. (STMicroelectronics, 2020)

B. Power supply by USB.

The ST-LINK/V2-1 supports USB power management. This allows the board to draw more than 100 mA current from the host PC. The ST-LINK USB connector CN1 (U5V or VBUS) can power all STM32 Nucleo-board and shield parts. Before the USB enumeration, the host PC supplies 100 mA to the board. This causes only the ST-LINK part of the STM32 Nucleo board to be power supplied at that time. (STMicroelectronics, 2020)

The STM32 Nucleo-board needs to draw 300 mA of current from the host PC At the USB enumeration. If the host PC can provide the required current, the STM32 MCU is power supplied and the red LED (Light Emitting Diode) LD3 is switched ON. In this case, the STM32 Nucleo board and its shield draw a current not exceeding 300 mA. On the contrary, the red LD3 remains switched OFF as the STM32 MCU, and the MCU part are not power supplied. In this event, an external power supply is necessary. (STMicroelectronics, 2020)

A jumper is mandatory to be connected between pins 1 and 2 of the JP5, as shown in Figure 36 below (STMicroelectronics, 2020):

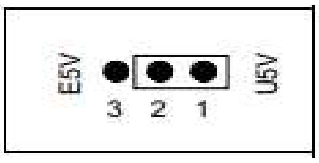


Figure 36: Jumper connection between 1 and 2 of JP5.

JP1 is set as per the maximum current consumption of the board when supplied by USB (U5V). The jumper JP1 can be configured when the board is supplied by USB, and the maximum current consumption is not more than 100mA (including an eventual extension board or Arduino shield). In this instance, the USB enumeration will always be successful as a current not exceeding 100 mA is drawn from the host PC. (STMicroelectronics, 2020)Table 3 below summarizes the Jumper connections for powering the board via USB for jumper JP1. (STMicroelectronics, 2020)

Table 3:JP1 connections for powering the board via USB.

Jumper state	Power supply	Allowed current
JP1 jumper OFF	USB power through CN1	300 mA max
JP1 jumper ON		100 mA max

C. Power supply through external sources.

An external power supply connected on VIN or E5V pins is necessary, whereas the current consumption of the board and extension boards surpasses the acceptable current on USB. In this case, using the USB for communication, programming or debugging only remains possible. However, it is compulsory to supply the board using VIN or E5V first and then connect the USB to the PC. This enhances the enumeration that occurs by the external power source. (STMicroelectronics, 2020)

The user manual (STMicroelectronics, 2020) details the sequence power procedure to adhere to when using VIN or E5V as external power sources. (STMicroelectronics, 2020)

The jumper's configuration needed for powering the board by VIN or E5V is illustrated below (STMicroelectronics, 2020):

- The Jumper JP5 needs to be on pin two and pin 3. (STMicroelectronics, 2020)
- The Jumper JP1 needs to be removed. (STMicroelectronics, 2020)

Figure 37 below illustrates the jumper's configuration of JP5. (STMicroelectronics, 2020)

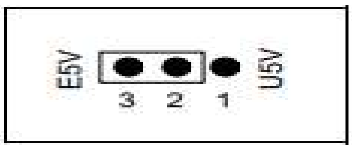


Figure 37: Connections between pins 2 and 3 of JP5.

Table 4 below summarizes the jumper configurations when using VIN or E5V pins connectors. (STMicroelectronics, 2020)

Table 4: Jumper configurations for using VIN or E5V pin connectors.

Input power name	Connectors pins	Voltage range	Max current	Limitation
VIN	CN6 pin 8 CN7 pin 24	7 V to 12 V	800 mA	From 7 V to 12 V only and input current capability is linked to input voltage: 800 mA input current when Vin=7 V 450 mA input current when 7 V<Vin (< or =) 9 V 250 mA input current when 9 V<Vin (< or =) 12 V
E5V	CN7 pin 6	4.75 V to 5.25 V	500 mA	-

D. Power supply through 3.3 V pin.

When an extension board delivers a voltage of 3.3 V, the +3.3 V pin, pin 4 of connector CN6 or pin 12 and pin 16 of connector CN7 can be directly used. In this instance, the power supply is provided by the + 3.3; the ST-LINK is not supplied, disabling the programming and debug features. (STMicroelectronics, 2020)

The configurations for using + 3.3 V pin are given in Table 5 below (STMicroelectronics, 2020):

Table 5: Pin connector for using +3.3V to supply the board.

Input power name	Connectors pins	Voltage range	Limitation
+3.3V	CN6 pin 4 CN7 pin 12 and pin 16	3 V to 3.6 V	Used when ST-LINK part of PCB is cut or SB2 and SB12 OFF

There are two different set-ups available for using the +3.3 V pin to provide power to the board: the ST-LINK is disconnected (PCB cut-off), or the solder bridge SB2 (3.3 V regulator) and SB12 (NRST) are OFF. (STMicroelectronics, 2020)

E. External power supply output.

The Nucleo board can provide an output power supply for an Arduino shield or an extension board when supplied via the USB, Pins VIN or E5V, +5V (pin 5 of connector CN6 or pin 18 of connector CN18). This implies it is necessary to adhere to the maximum current rating of the power source provided above. (STMicroelectronics, 2020) Moreover, the +3.3 V can also provide power supply output. The maximum current capability of the regulator U4 (500 mA max) limits the output current. (STMicroelectronics, 2020)

F. LEDs.

F.1. LD1.

The information regarding the ST-LINK communication status is given by the tricolour LED (green, orange, red) LD1 (COM). The colour red is the default colour, and the green colour shows that the communication between the PC and the ST-LINK/V2-1 is underway. The LED LD1 presents the status below (STMicroelectronics, 2020):

- **Slow blinking Red/Off:** occurs at switch-ON prior to USB initialization. (STMicroelectronics, 2020)
- **Fast blinking Red/Off:** occurs after the first successful communication between the PC and ST-LINK/V2-1 (enumeration). (STMicroelectronics, 2020)
- **Red LED ON**: occurs on the completion of the initialization between the PC and ST-LINK/V2-1. (STMicroelectronics, 2020)
- **Green LED ON:** occurs subsequently a successful target communication initialization. (STMicroelectronics, 2020)
- **Blinking Red/Green:** occurs when communication with a target is occurring. (STMicroelectronics, 2020)
- **Green ON:** occurs at complete and successful communication. (STMicroelectronics, 2020)
- **Orange ON:** occurs when communication fails. (STMicroelectronics, 2020)

F.2. User LD2.

The green colour LED is a user LED which connects to Arduino signal D13 (STM 32 pin 21, I/O PA5 on Pin 34, PB13) according to the STM32 target. The table below describes the states of the LED. When the I/O is High, the LED goes ON, and when the I/O is Low, the LED goes OFF. (STMicroelectronics, 2020)

F.3. LD3 PWR.

This LED shows the MCU part is power supplied and that the +5V is present. (STMicroelectronics, 2020)

F.4. Pushbuttons.

The push-button B1 USER is a user button that connects to pin 2 (I/O PC13) of the STM32 MCU (STMicroelectronics, 2020), while the push-button B2 RESET connects to NRST and is needed to reset the STM32 MCU. These pushbuttons are covered by blue and black plastic hats that can be taken off if required. For instance, when an application or a shield is superposed on the Nucleo board to prevent pressure on the pushbuttons, which can cause the STM32 MCU to RESET at all times. (STMicroelectronics, 2020)

F.5. Current measurement.

The Jumper JP6 referred to IDD and is necessary for the current consumption of the MCU measurement. This measurement is performed by connecting an ammeter to the jumper pins in place of the jumper. By default, the jumper is connected to the pins to provide the board with a disabled current measurement at shipment. (Ibrahim, 2019) When the jumper is available, the STM 32 MCU receives power by default, and when the jumper is removed, an ammeter is required to be connected for the current consumption measurement of the STM 32 MCU. The STM 32 MCU is powerless without the ammeter. (STMicroelectronics, 2020)

F.6. The ST-LINK /V2-1.

The ST-LINKV2-1 is a necessary tool for programming and debugging. This tool is integrated into the Nucleo boards and makes it to be compatible with Mbed. The ST-LINK/V2-1 supports only the Serial Wire Device for STM32 devices. The tool also supports the virtual COM port interface on USB, USB software renumeration, mass storage interface on USB, and USB power management requests exceeding 100 mA power on USB. However, the SWIM is not supported. Moreover, the minimum supported application voltage is set to 3 V. (Ibrahim, 2019)

There are two separate ways available to use the ST-LINKV2-1 according to the jumper's state. These states allow for programming(debugging) the onboard STM32 and programming (debugging) the MCU in an external application through a cable connected to SWD connector CN4. The table below illustrates the summary of the state of jumpers. (STMicroelectronics, 2020)

Table 6: Jumper states summary.

Jumper state	Description
Both CN2 jumpers ON	ST-LINK/V2-1 functions enabled for on board programming (default)
Both CN2 jumpers OFF	ST-LINK/V2-1 functions enabled for external CN4 connector (SWD supported)

The configurations for connector CN2 for programming/debugging the onboard STM32 and the programming/debugging the board by an external application connecting a cable to SWD connector CN4 are depicted respectively in Figure 38 and Figure 39 below (STMicroelectronics, 2020):

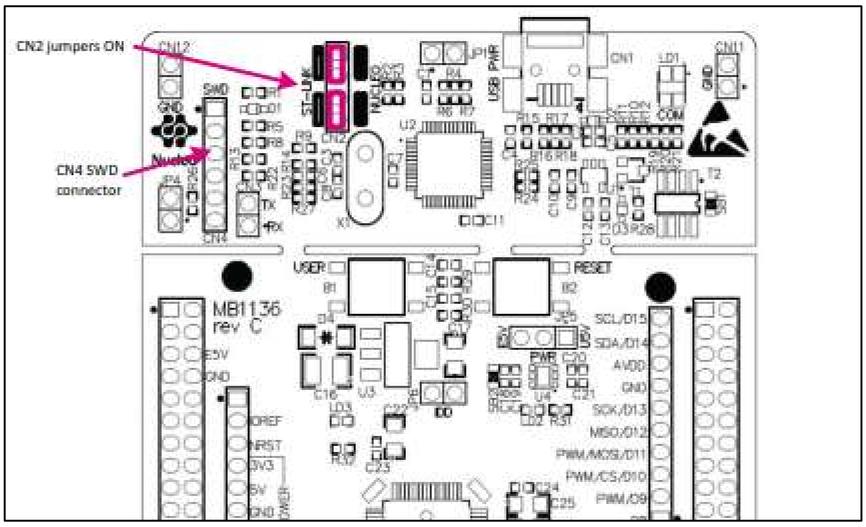


Figure 38: STM32 connection to program the onboard MCU.

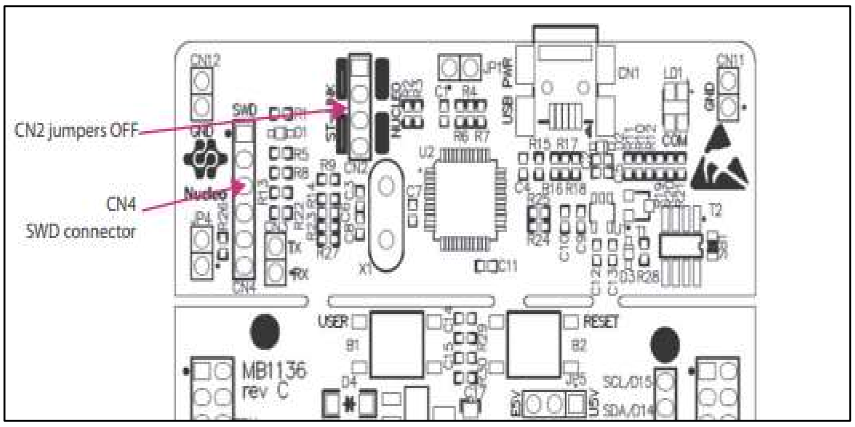


Figure 39: Using ST-LINK to program the MCU via an external application.

F.7. Clock circuit of STM32F303RET6 MCU.

The clock circuit is a vital part of the MCU**.** (Ibrahim, 2019) the selection of the system clock is made on start-up. Nevertheless, the internal RC 8 MHz oscillator is selected as a CPU clock on reset by default. An external 4 – 32 MHz is also an option; the CPU is monitored for failure in the instance. If a failure is detected, the system changes back to the internal RC 8 MHz oscillator. When enabled, a software interrupt is produced. Likewise, the full interrupt management of the PLL clock entry is present when required, as in a case of the failure of an indirectly used external oscillator, for instance. (STMicroelectronics, 2016)

Many prescalers are available to enable the configuration of the AHB frequency, the high-speed APB (APB2) and the low-speed APB (APB2) and the low-speed APB (APB1) domains. On the one hand, 72 MHz is the maximum frequency of the AHB and the high-speed APB domains, and on the other hand, 36 MHz is the maximum allowed frequency of the low-speed APB domain. (STMicroelectronics, 2016)

Figure 40 below depicts the clock tree of the STM32F303RET6. (STMicroelectronics, 2016)

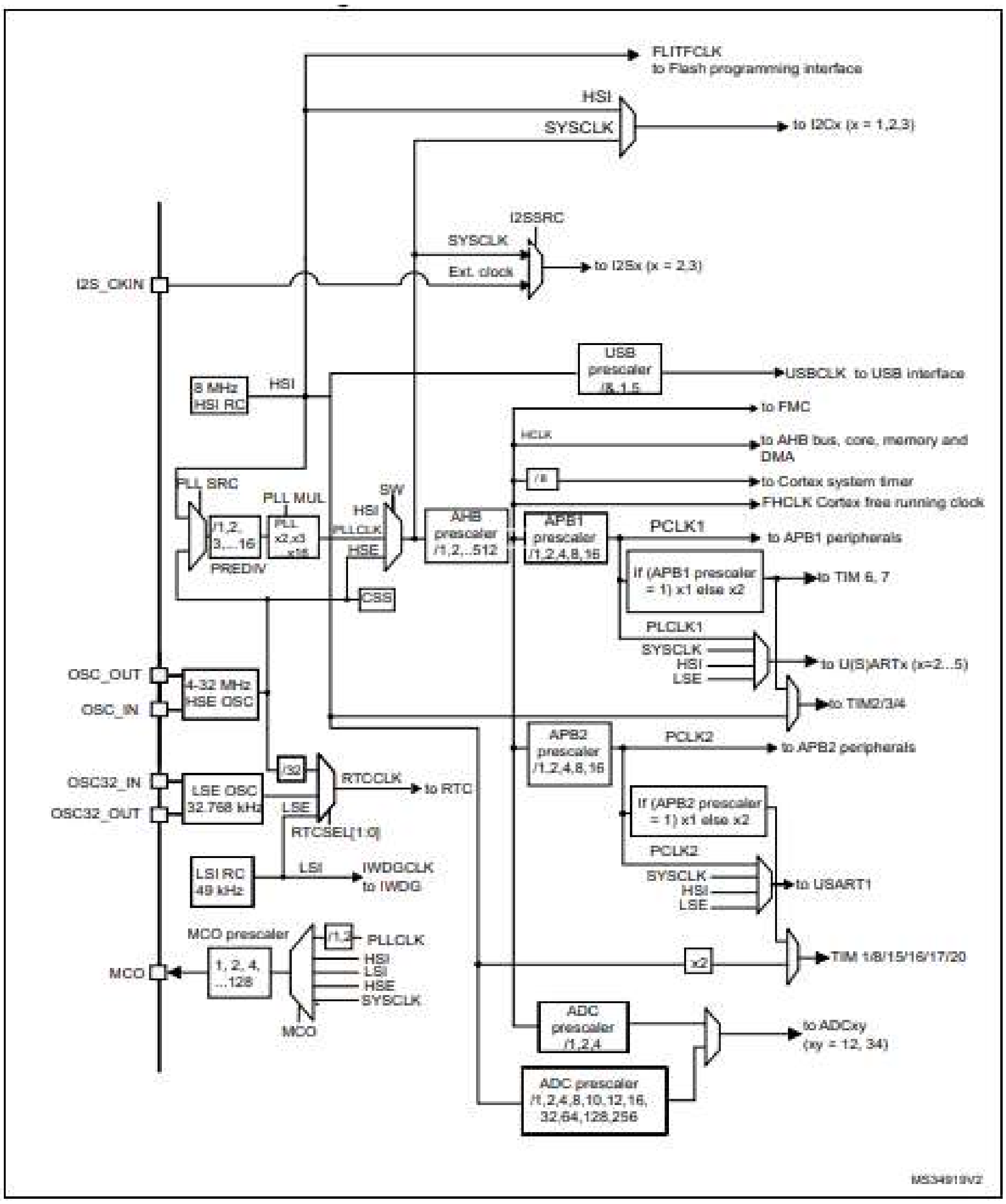


Figure 40: Clock Tree of STM32F303RET6.

2.7 MEASUREMENTS OF ENERGY CONSUMPTION.

2.7.1 Power and energy measurements.

In general, the power consumption of a device is the amount of energy consumed at a given moment, while the sum of the power consumption over time is the energy consumption (Haulin, 2018). As an example, the flash camera has a very high-power consumption when ON. However, the total energy consumption is mild as the camera flash is ON for a very short while. (Haulin, 2018)

The total energy consumption E of a device can be expressed mathematically by (Haulin, 2018):

$$E = \int_0^T P(t)\, dt \tag{1}$$

With E: Energy consumption

T: Execution time

$\mathrm{P(t)}$: the instantaneous power.

In electrical devices, the power consumption P is proportional to the current consumption I and the voltage U given by Joule's law below (Haulin, 2018):

$$P = U \, . I \tag{2}$$

On systems with constant supply voltage U, the power consumption depends solely on the current consumption I. This implies that reducing the current consumption will cause the power consumption to be reduced. (Haulin, 2018) Since an embedded system application often runs continually, the energy and the time would rise without limit. It is, therefore, more practical to use the mean (average) power P mean, as opposed to the total energy since, this P mean is a limited quantity given by (Haulin, 2018):

$$Pmean = \frac{E}{T} \tag{3}$$

A physical measurement must be made so as to determine the power consumption for each instruction, a physical measurement needs to be made. Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, (2012), give the energy consumption as:

$$E = P \times T \tag{4}$$

In Equation *(4)*, P is referred to as the average power over the time period T. Thus, the time period of our measurement will be the clock frequency of the microcontroller. (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012)

2.7.2 MCU energy consumption measurements.

2.7.2.1 Power and Current consumption measurements.

Power consumption measurements can be achieved in several different ways. (Haulin, 2018) The most common way to measure the power consumption of a device is to use a shunt resistor. (Haulin, 2018) For instance, in (Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012), (C. Chang, S. Muftic and D. J. Nagel, 2007), (Lehlogonolo P. I. Ledwaba, 2018), the authors used this method in their work to establish the power consumption of the processing unit.

This shunt resistor is connected in series with the device to convert the current into a voltage. The obtained voltage is then utilized to compute the current from Ohm’s law. the obtained current values and the supply voltage are used to compute the power, as stated earlier. (Haulin, 2018), (Lehlogonolo P. I. Ledwaba, 2018) The shunt resistor is commonly used as most engineers are familiar with and does not necessitate the use of advanced circuits. (Haulin, 2018)

2.7.2.2 Execution time measurements.

Many various methods are available in order to measure the execution time of a piece of code. However, not a single best technique exists. Instead, each technique is a trade-off between different attributes, such as resolution, accuracy, granularity, and difficulty. (Stewart, September 2006) Furthermore, the use of the methods depends on the hardware features such as digital output port, while other techniques necessitate specific software and measurement instrumentation tools to be available. (Stewart, September 2006)

The hardware and tools are not always affordable, and their non-availability can render using a particular method impossible. However, contrastingly, access to the correct equipment can greatly facilitate obtaining the requisite measurements—a mitigating factor in investing in procuring the appropriate tools (Stewart, September 2006).

According to the attributes mentioned above, these measurement methods are summarized (Stewart, September 2006) according to the attributes as mentioned above. Among these measurements' methods, the stopwatch is easy to use and suitable for non-interactive programs. A stopwatch can measure the time of things like numerical code, which may take minutes or hours to execute and when the measurements only require to be approximate (to the nearest second). (Stewart, September 2006)

This method simply requires using a chronograph feature or digital wristwatch (or other equivalent timing devices). The watch is started when the program starts, and when the program ends, the watch is stopped to read the time. (Stewart, September 2006) The oscilloscope and logic analyser are among the techniques used nowadays. (Andreas Ermedahl, Jakob Engblom, 2007)

These methods consider the observable behaviour of a system in operation while not influencing the functioning of the system. For example, an oscilloscope involves adding a bit-flip on an externally accessible pin of the processor to the program segments of interest and observe the obtained waveform to determine the periodicity and thus the execution time. A logic analyser looks at the data or address bus of the system and can see when the instructions are being fetched. However, the logic analyser needs relevant memory transactions to reach the bus, which is not necessarily the case with a cache system. (Andreas Ermedahl, Jakob Engblom, 2007)

CHAPTER 3

EXPERIMENTAL METHODOLOGY

3.1 INTRODUCTION.

The two approaches used to measure the energy consumption at the MCU due to the computation of the loop structures (For and While) in a nesting fashion are described in this chapter. The main difference between the two approaches is the execution time concept. In the first approach, the execution time is the system's operation duration; meanwhile, on the second approach, the execution time is the time taken by the STM32F303RE MCU of the Nucleo-64 board to execute the different levels of loops. The different setups for these approaches are detailed below.

3.2 FIRST APPROACH.

3.2.1 Software.

The microcontroller is programmed via the ST-Link using the USB cable, as shown in Figure 41.

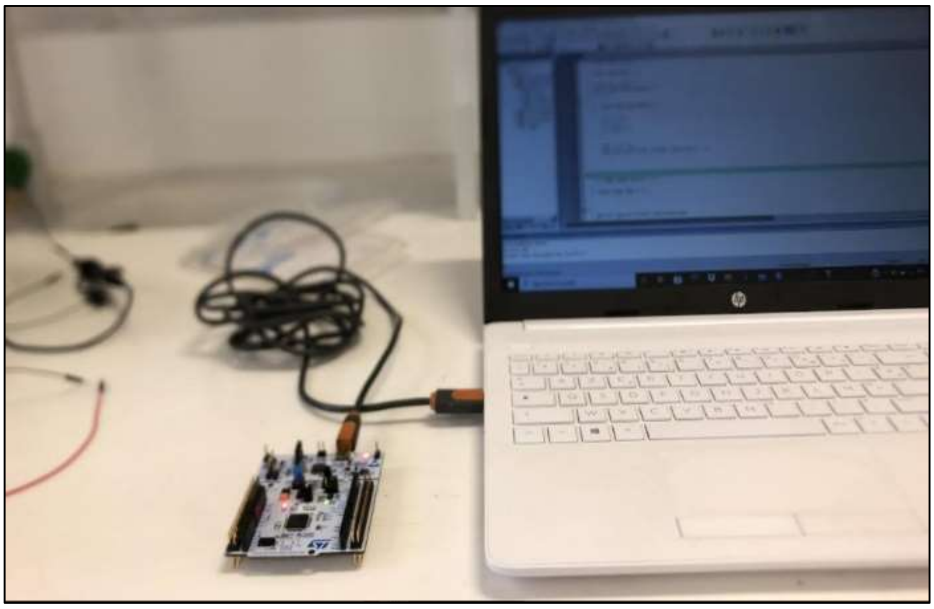
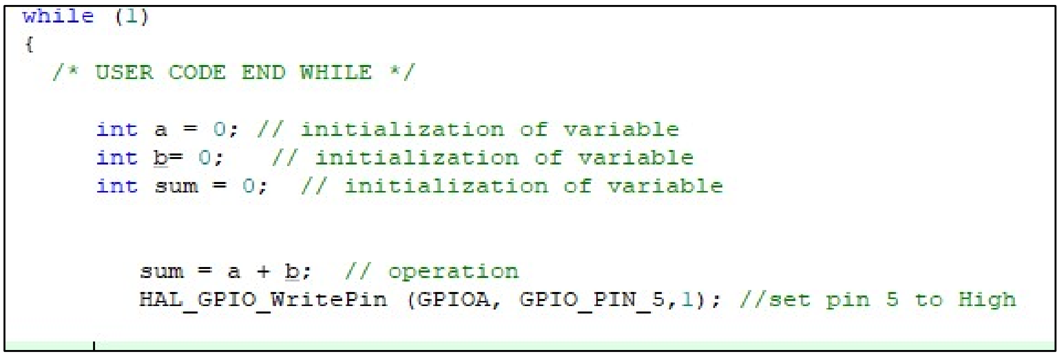


Figure 41: Programming of STM32F303RE MCU.

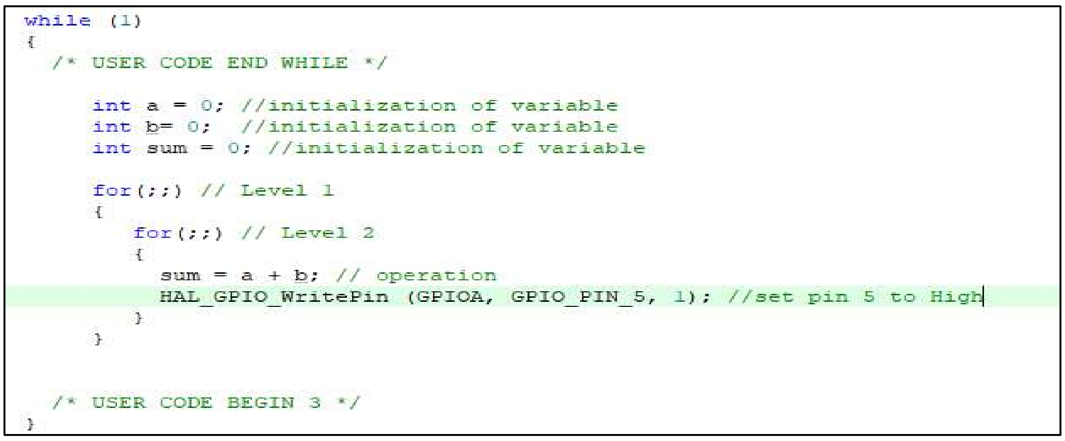
The software loaded in the STM32F303RE is an infinite loop. Then, the USB cable is unplugged. Since an embedded system runs the application endlessly, different levels of infinite For and While loops have been loaded on the MCU. As every code runs within an infinite loop, the initial code is generated and considered as the reference point. The initial code includes the command for setting the GPIO pin high to indicate the level of operation of the inner loop with a delay function to pause the MCU operation. Figure 42 depicts the initial code.



```
while (1)
{
  /* USER CODE END WHILE */

     int a = 0; // initialization of variable
     int b= 0;   // initialization of variable
     int sum = 0;  // initialization of variable


        sum = a + b;  // operation
        HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,1); //set pin 5 to High
```

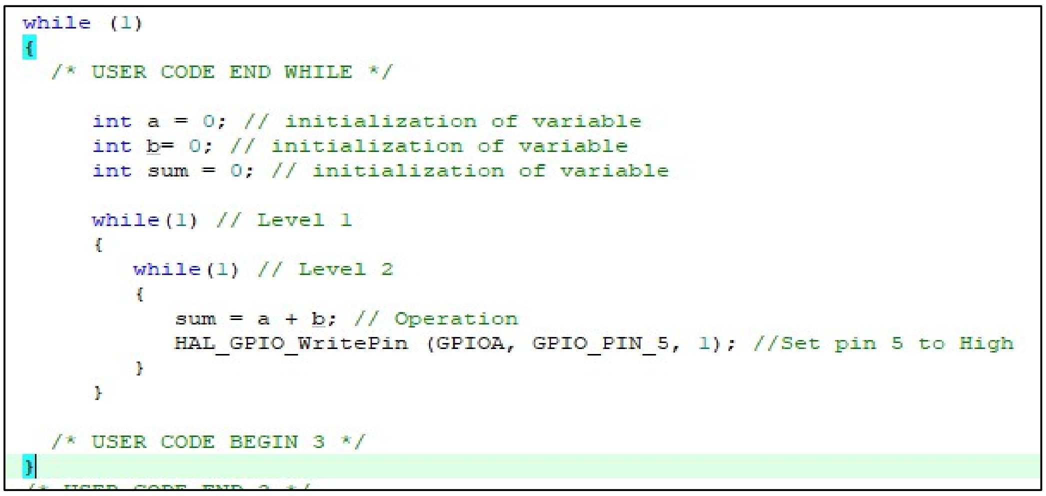
Figure 42: Initial code.

The MCU operates at 8 MHz for the different levels of loops programmed. In the software, the different levels of infinite loops of For and While structures have been implemented with the command for setting the GPIO pin high during the computation of the inner loop. Figure 43 and Figure 44 illustrate the For and While loops for different nesting levels, respectively as an illustration.



```
while (1)
{
  /* USER CODE END WHILE */

     int a = 0; //initialization of variable
     int b= 0;  //initialization of variable
     int sum = 0; //initialization of variable

     for(;;) // Level 1
     {
        for(;;) // Level 2
        {
          sum = a + b; // operation
          HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5, 1); //set pin 5 to High
        }
     }


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

Figure 43: For loop (nesting level 2).



```
while (1)
{
  /* USER CODE END WHILE */

     int a = 0; // initialization of variable
     int b= 0; // initialization of variable
     int sum = 0; // initialization of variable

     while(1) // Level 1
     {
        while(1) // Level 2
        {
           sum = a + b; // Operation
           HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5, 1); //Set pin 5 to High
        }
     }

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

Figure 44: While loop (nesting level 2).

3.2.2 Execution time measurements.

A stopwatch is used to measure the operation time of the MCU. The current consumption is measured at the interval of 1 minute for a period of 5 minutes from the time the power supply is ON. The stopwatch is started at the time the power is ON and stopped at 5 minutes. The picture of the stopwatch is depicted in Figure 45 below.

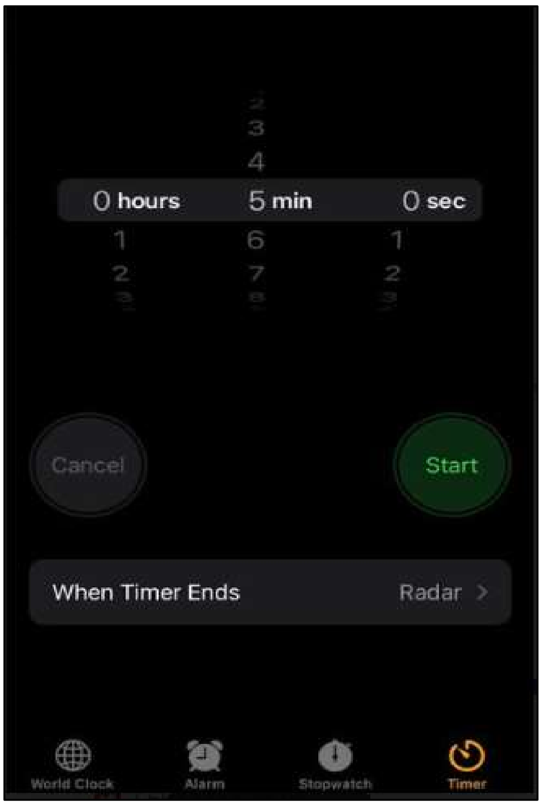


Figure 45: Stopwatch application

3.2.3 Current and Energy Consumption.

Using the FLUKE DESKTOP multimeter set as an ammeter, the current consumed by the MCU is measured as the application runs at an interval of each minute for a period of 5 minutes of operation. These measurements are recorded manually. Then the average currents for the 5 minutes are computed. The FLUKE DESKTOP set as an ammeter is depicted in Figure 46 below:

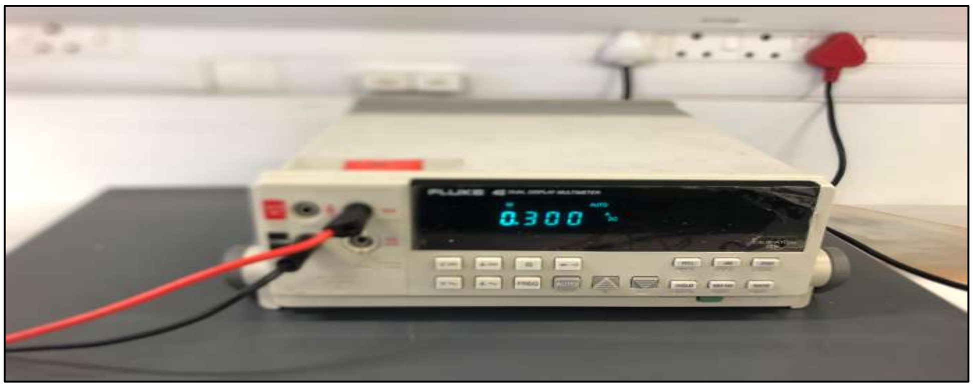


Figure 46: Fluke Multimeter Desktop set as an ammeter.

An external power source DELTA ELEKTRONICA SM52-30 is used to power the

MCU with a fixed 3.3 V as illustrated in Figure 47 below:

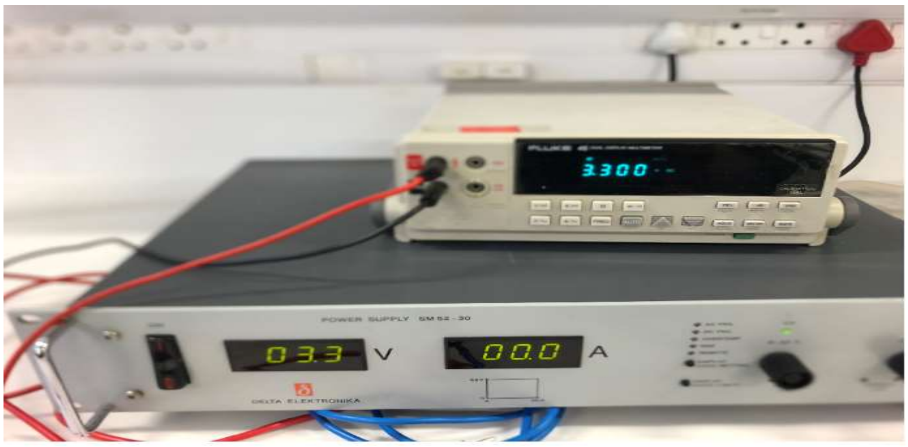


Figure 47: Delta ELEKTRONICA SM52-30 power supply.

The power supply is set to supply a voltage of 3.3 V and a current of 0.300 mA acceptable for the MCU powered by an external 3.3 V. Furthermore, before each measurement, the power supply was calibrated to provide the set voltage and current required as shown in the above pictures. Using equations (2) and (4), the average power and energy consumption are calculated respectively for every minute, and the power and energy consumption for 5 minutes is also determined. Figure 48 illustrates the setup to measure the energy consumed by the STM32F303RE MCU.

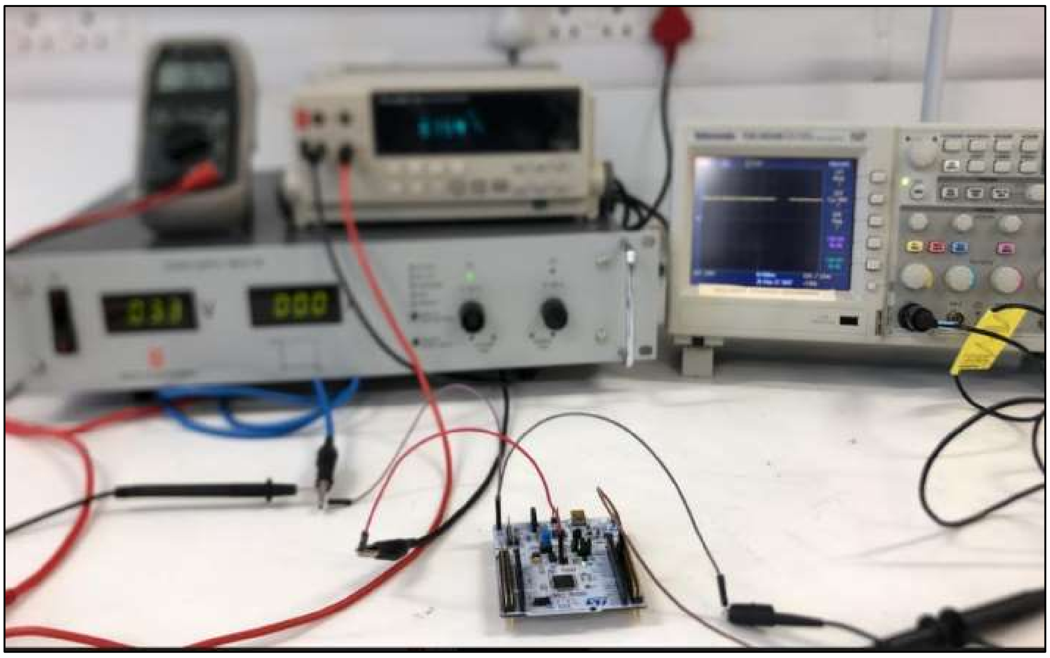
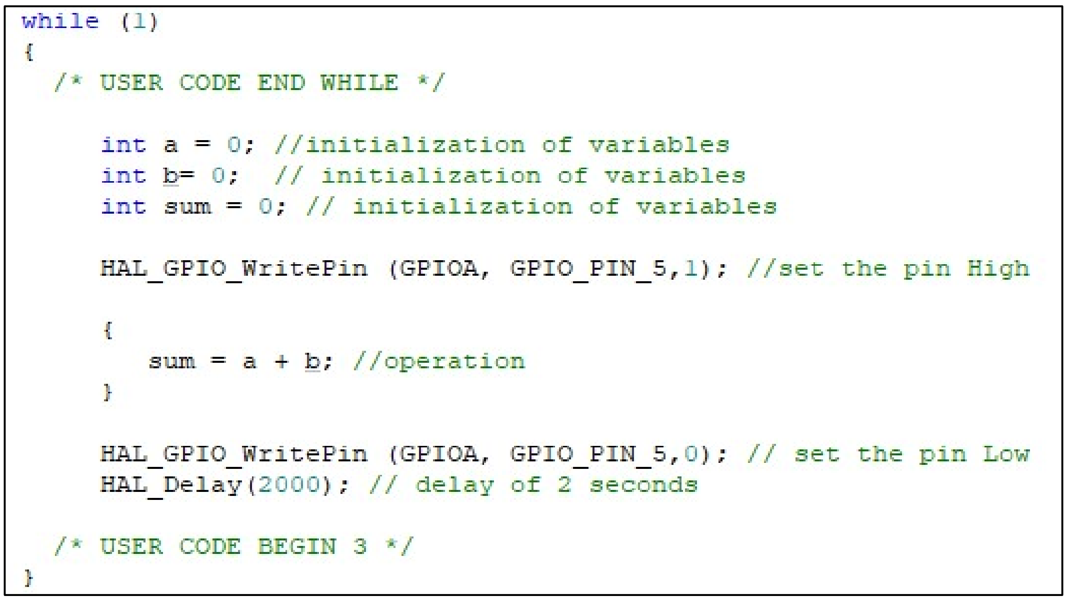


Figure 48: Energy Consumption measurement setup.

3.3 SECOND APPROACH.

3.3.1 Software.

At the start, the microcontroller is programmed, then the USB is disconnected to eliminate the impact of circuit consumption of the ST-Link as previously. After that, the programmed software on the STM32F303RE is processed in an endless loop. As every code runs within an infinite loop, an initial code is considered the measurements' reference point. This initial code includes the command for setting the GPIO pin high and low with the delay function. The different levels of loops will then be implemented within the initial code. The picture in Figure 49 depicts the setup of the initial code.

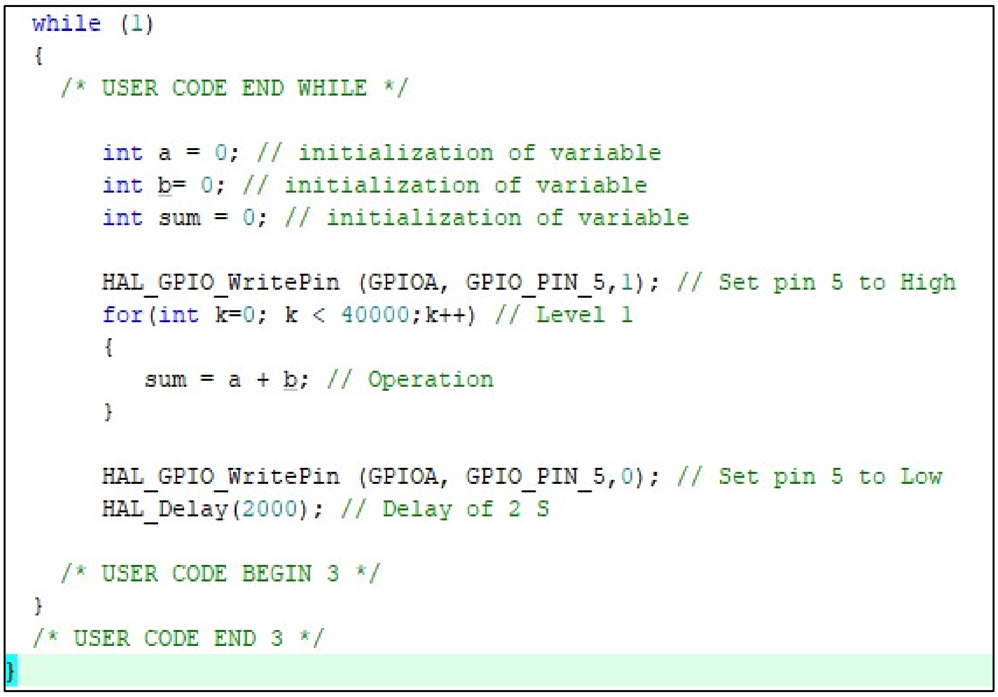


```
while (1)
{
  /* USER CODE END WHILE */

     int a = 0; //initialization of variables
     int b= 0;  // initialization of variables
     int sum = 0; // initialization of variables

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,1); //set the pin High

     {
        sum = a + b; //operation
     }

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,0); // set the pin Low
     HAL_Delay(2000); // delay of 2 seconds

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

Figure 49: Initial code.

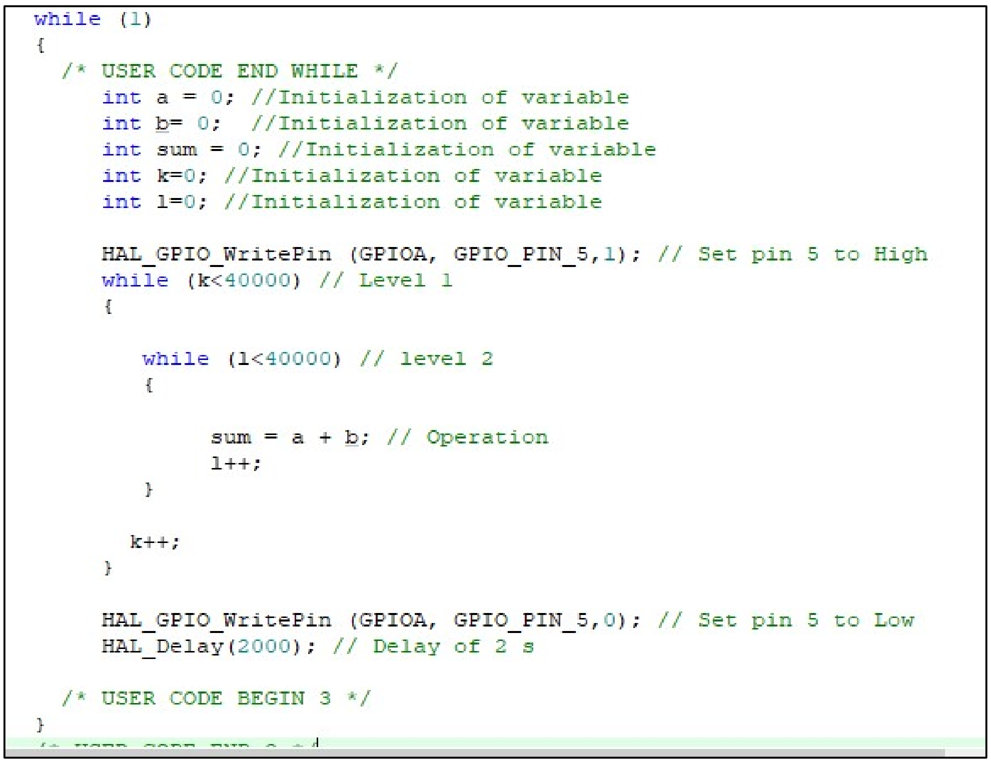
The 32-bit MCU runs at an 8 MHz internal clock frequency; this implies that each instruction is processed at 125 nanoseconds within the MCU as set in the program.

In the code, loops include a fixed counter value. This number is chosen according to the maximum time of 20 milliseconds delay between the two states of the GPIO pin at level 1. Then 1 level of a loop (For or While respectively) is added up to 4 levels in nesting fashion. Finally, the inner loops' values are chosen to conveniently display the waveforms on the oscilloscope for the four different levels of nesting loops. Figure 50 and Figure 51 illustrate different nesting levels of for and while loops.



```c
while (1)
{
  /* USER CODE END WHILE */

     int a = 0; // initialization of variable
     int b= 0; // initialization of variable
     int sum = 0; // initialization of variable

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,1); // Set pin 5 to High
     for(int k=0; k < 40000;k++) // Level 1
     {
        sum = a + b; // Operation
     }

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,0); // Set pin 5 to Low
     HAL_Delay(2000); // Delay of 2 S

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

Figure 50: For-Level 1.



```c
while (1)
{
  /* USER CODE END WHILE */
     int a = 0; //Initialization of variable
     int b= 0;  //Initialization of variable
     int sum = 0; //Initialization of variable
     int k=0; //Initialization of variable
     int l=0; //Initialization of variable

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,1); // Set pin 5 to High
     while (k<40000) // Level 1
     {

        while (l<40000) // level 2
        {

             sum = a + b; // Operation
             l++;
        }

       k++;
     }

     HAL_GPIO_WritePin (GPIOA, GPIO_PIN_5,0); // Set pin 5 to Low
     HAL_Delay(2000); // Delay of 2 s

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

Figure 51: While-level 2.

The power supply needs to be set to provide a 3.3 V voltage and 300 mA current to supply the MCU for the measurements to be consistent. Therefore, the power supply needs to be set before taking each measurement.

3.3.2 Execution time measurements.

The execution time measurements were performed by changing the selected GPIO pin and using a Tektronix TDS3024B oscilloscope to determine the waveform width time between the rising and falling edges of the nested loops being executed. The physical measurement setup is illustrated in Figure 52 below.

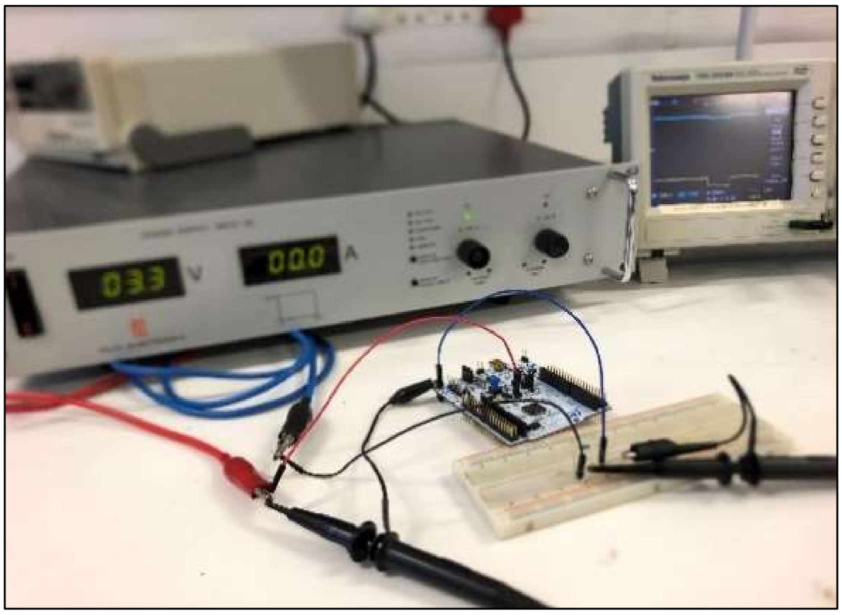


Figure 52: Setup.

Illustrations of the execution time measured by the oscilloscope using channel 1 (CH1) and channel 2 (CH2) are shown in Figure 53, Figure 54, and Figure 55 below:

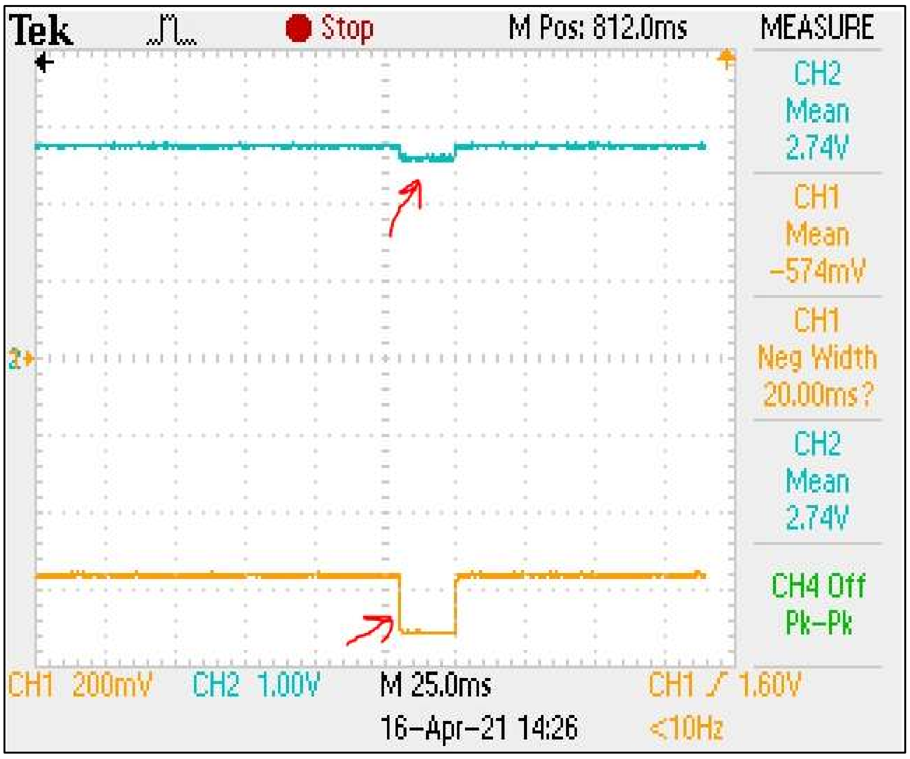


Figure 53:For loop level 1 Execution time measurement.

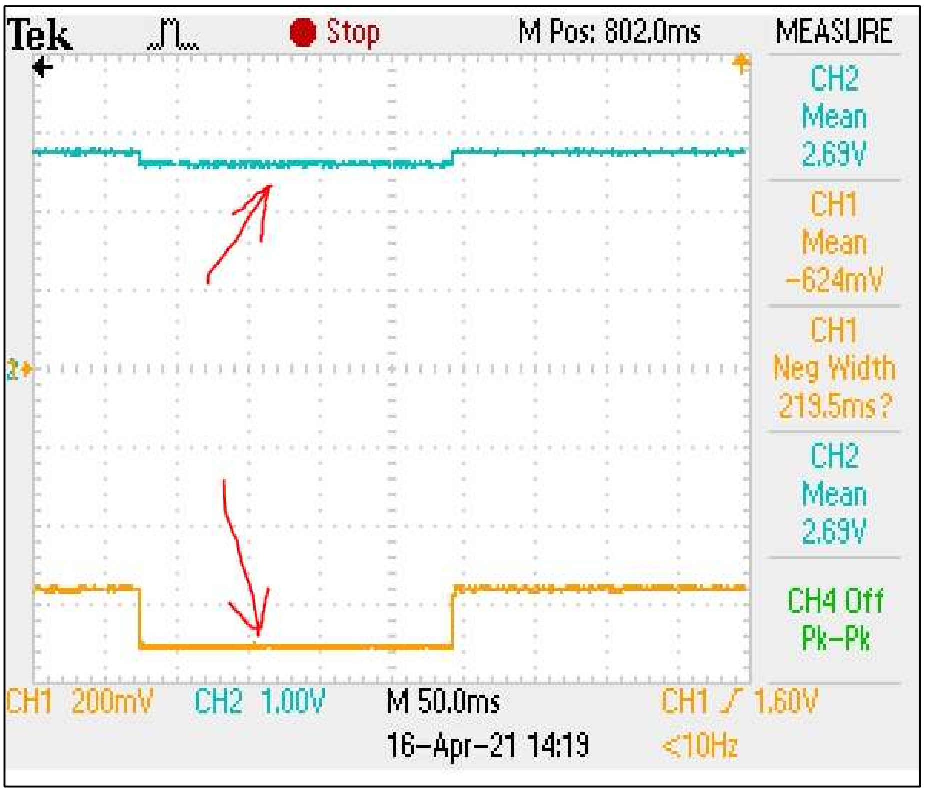


Figure 54: For Level 2 Execution time.

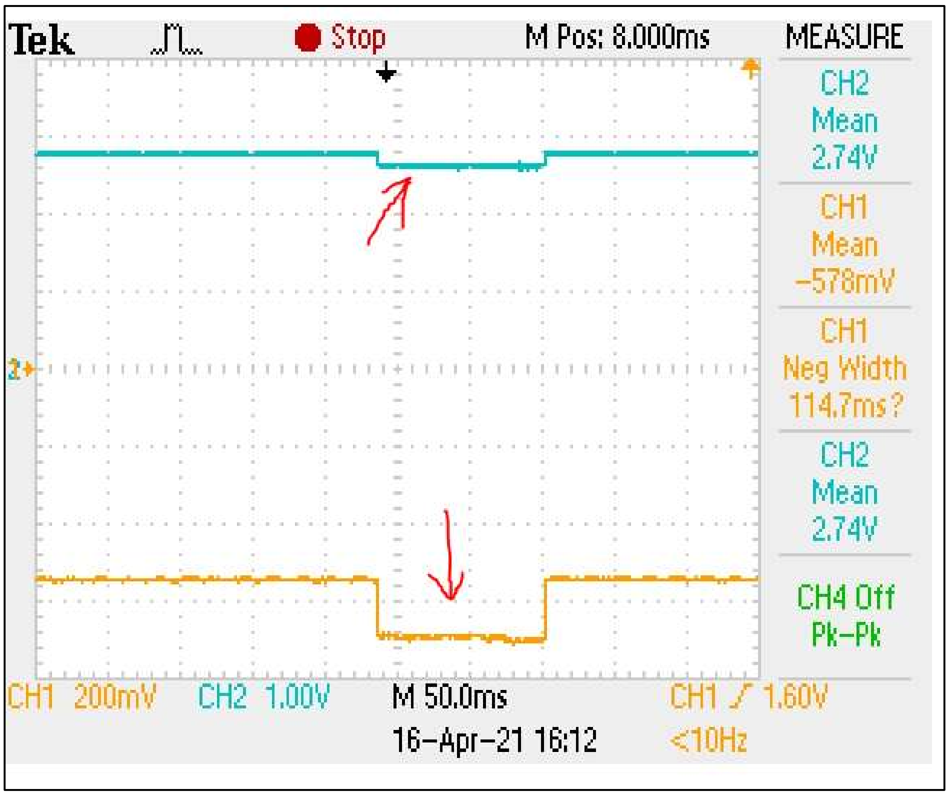


Figure 55: While level 4 Execution time measurements.

The time to execute the running nested loops was measured with the oscilloscope, and the readings were manually recorded. The GPIO pin was set high at the beginning of the nested loops and was set low as the nested loops ended. The square wave was displayed at the oscilloscope, and the time between the rising and falling edge was measured manually in some by counting the number of divisions or by reading the width (pulse) of the obtained waveform given by the oscilloscope where possible to determine the time needed to execute the nested loops. The negative value simply means that the polarity is swapped around. This is intended to be as such so that the user can visualise the measurement signal.

While CH1 in yellow measures the execution time of the loops as the execution starts and when the execution ends by reading the obtained pulse, CH2 in blue also indicates the supply voltage of the MCU during the time of execution displayed in CH1. The area indicated in CH1 and CH2 has the same duration.

3.3.3 Power and Energy consumption measurements.

In order to determine the power (energy) consumed while the various levels of loops are being executed, the current consumption needed to be measured by inserting a 100-ohm shunt resistor in the current consumption line of the STM32F303RE development board by removing the IDD jumper of the board as shown in Figure 56 below:

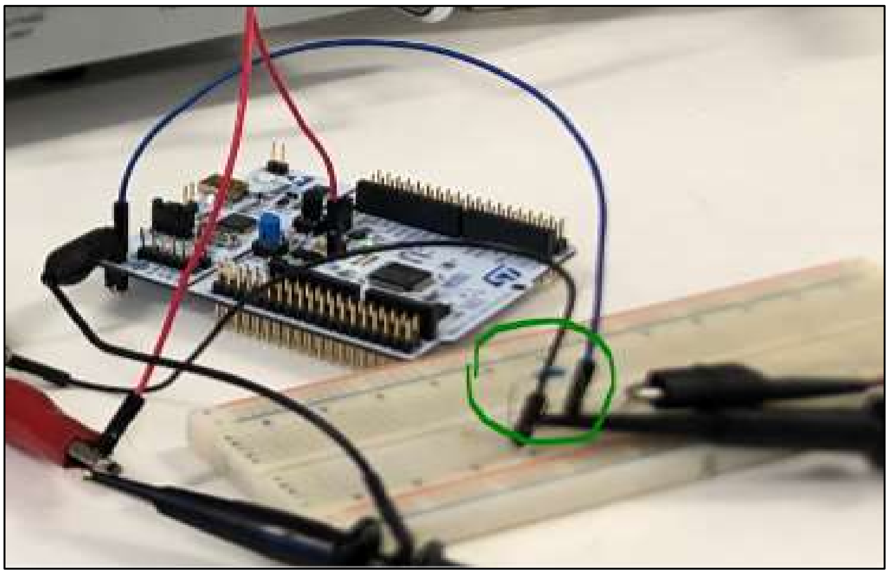


Figure 56: Shunt Resistor Set-up.

The resistor value has been measured by the Fluke desktop multimeter set as an Ohmmeter before the insertion in the circuit, as illustrated in Figure 57 below:

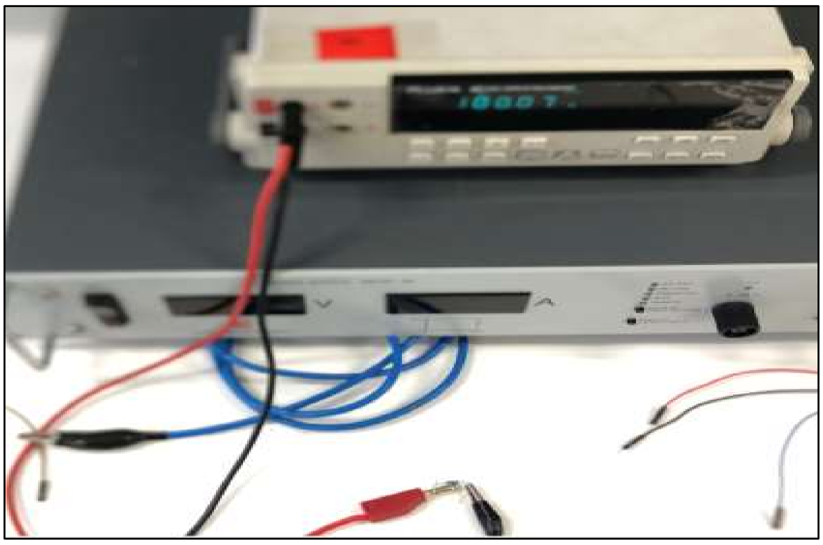


Figure 57: Shunt resistor measurements.

The shunt resistor is 100 ohms with 1% tolerance; the resistor has been measured by the Fluke Desktop multimeter set as an Ohmmeter as per the above. Its value is 100.07 Ohm. (Since the decimal point is insignificant, the 100 Ohm value is used for the current calculations). The measured voltage drop across the shunt resistor during execution was used in the Ohms law and power equation to determine the power consumed by the MCU seen during the execution of the levels of the loop. The available current consumed is activated by the removal of the IDD jumper.

For the experiments, two channels of the oscilloscope TEKTRONIX TDS2024B was used. CH 1 in yellow was utilised to measure the voltage drop across the resistor Which waveform, while CH 2 in blue was utilised to monitor the supplied voltage of the MCU, as shown in Figure 54. In addition, the GPIO output pin driving an LED was used to physically see the LED's blinking as the pin toggles before and after execution of the level of loops to be tested. With the two waveforms displayed, the voltage drop was then matched with the supplied voltage during the execution of the level of loops, allowing the detection of the portion of the MCU consumption that happened due to the processing of these loops.

An external power source, DELTA ELEKTRONICA SM52-30 supplying 3.3 V, was chosen to supply the MCU only since the power supply via USB was removed after loading the codes in the target board. Figure 58 indicates the setup of the MCU with a shunt resistor, the power supply, and the oscilloscope.

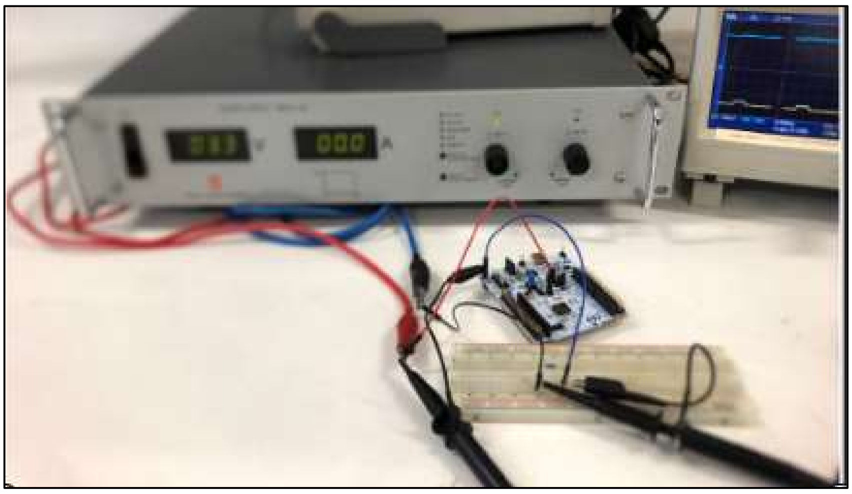


Figure 58: Measurement's setup.

The average voltage across the shunt resistor was used to compute the MCU current consumption. Along with the supply voltage measured, the MCU current consumption was used to calculate the power and energy consumption of the MCU.

CHAPTER 4

EXPERIMENTATION RESULTS

In the first approach, the results of the initial code and five levels of For and While loops in nesting fashion are summarized in Table 7 and Table 8, respectively.

Table 7: Results of For Loop measurements.

Levels	Time					Average	Average	Average
	1 min	2 min	3 min	4 min	5 min	Current	Power	Energy
0	I = 8.82mA	I= 8.67 mA	I= 8.512 mA	I= 8.33 mA	I= 8.32 mA	I= 8.530 mA	P= 28.149 mW	E =8444.7 mJ
1	I = 8.83 mA	I= 8.75 mA	I= 8.67 mA	I= 8.59 mA	I= 8.59 mA	I= 8.686 mA	P= 28.664 mW	E =8599.2 mJ
2	I = 9.00 mA	I= 8.92 mA	I= 8.84 mA	I= 8.764 mA	I= 8.78 mA	I = 8.861 mA	P= 29.241 mW	E =8772.3 mJ
3	I = 9.002 mA	I= 8.933 mA	I= 8.851 mA	I= 8.776 mA	I= 8.792 mA	I = 8.871 mA	P= 29.274 mW	E =8782.2 mJ
4	I = 9.024 mA	I= 8.92 mA	I= 8.816 mA	I= 8.812 mA	I= 8.83 mA	I= 8.880 mA	P= 29.304 mW	E =8791.2 mJ
5	I = 9.03 mA	I= 8.935 mA	I= 8.832 mA	I= 8.796 mA	I= 8.815 mA	I= 8.882 mA	P= 29.311 mW	E =8793.3 mJ

Table 8: Results of While Loop measurements.

Levels	Time					Average	Average	Average
	1 min	2 min	3 min	4 min	5 min	Current	Power	Energy
0	I = 8.82 mA	I= 8.67 mA	I= 8.512 mA	I= 8.33 mA	I= 8.32 mA	I= 8.530 mA	P= 28.149 mW	E =8444.7 mJ
1	I = 8.75 mA	I= 8.66 mA	I= 8.59 mA	I= 8.53 mA	I= 8.46 mA	I= 8.598 mA	P= 28.373 mW	E =8511.9 mJ
2	I = 8.89 mA	I= 8.825 mA	I= 8.755 mA	I= 8.668 mA	I= 8.707 mA	I = 8.769 mA	P= 28.938 mW	E =8681.4 mJ
3	I = 8.900 mA	I= 8.832 mA	I= 8.758 mA	I= 8.69 mA	I= 8.705 mA	I = 8.777 mA	P= 28.964 mW	E =8689.2 mJ
4	I = 8.949 mA	I= 8.838 mA	I= 8.736 mA	I= 8.748 mA	I= 8.761 mA	I= 8.806 mA	P= 29.06 mW	E =8718 mJ
5	I = 8.98 mA	I= 8.883 mA	I= 8.787 mA	I= 8.69 mA	I= 8.698 mA	I= 8.808 mA	P= 29.066 mW	E =8719.8 mJ

The average current consumed by the MCU is the average of the current measured at each minutes for the 5 minutes of operation. As an illustration the averages of current when executing a For loop of level 5 and While loop of level 5 are respectively obtained as below:

Average Current = (9.03 mA + 8.935mA+ 8.832mA + 8.796mA + 8.815mA)/5

= 8.882 mA.

Average Current = (8.98 mA + 8.883mA+ 8.787mA + 8.69mA + 8.698mA)/5

= 8.808 mA.

The average power consumption of the MCU is obtained using the average current consumed and the fixed voltage of 3.3V (MCU supply voltage) in equation (2) to obtain the average power consumption. As an illustration, the averages of power when executing a For loop and While loop of level 5 are respectively obtained as below:

Average Power = (3.3V) X (8.882 mA)

= 29.311 mW.

Average Power = (3.3V) X (8.808 mA)

= 29.056 mW.

The average energy consumption of the MCU is obtained by using average power consumption and the total time of operation (5 minutes) in equation (4) to obtain the average energy consumption. As an illustration, the averages of energy when executing a For loop and While loop of level 5 are respectively obtained as below:

Average Energy = (29.311 mW) X (300 s)

= 8793.3 mJ.

Average Energy = (29.311 mW) X (300 s)

= 8793.3 mJ.

The obtained average current and average energy consumption of the MCU summarized in Table 7 for the For loops in the nesting fashion (from level 0 to level 5) are represented in graphs as shown in Figure 59 and Figure 60, respectively.

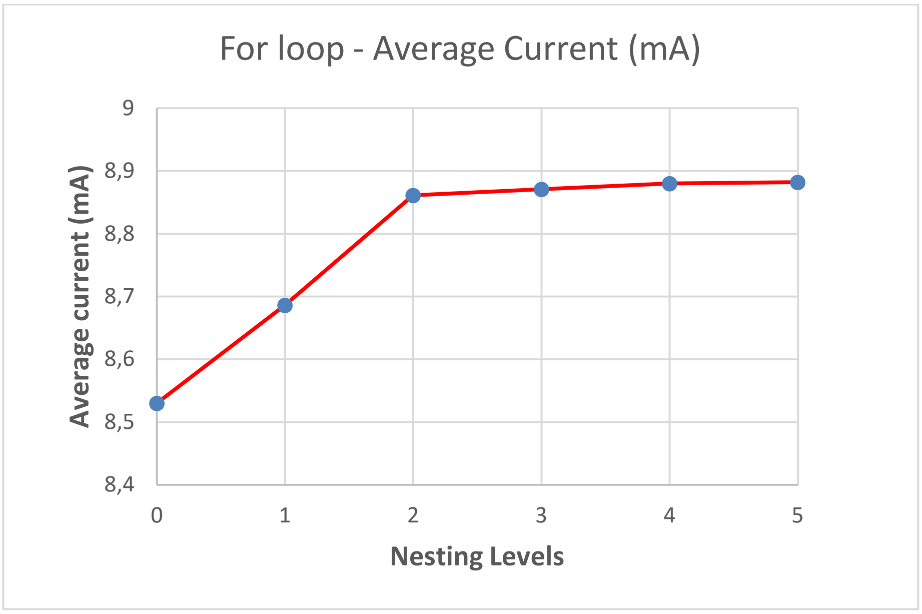


Figure 59: For- Average Current vs nesting levels of loops.

In Figure 59, the image shows a steep rise of the current consumption concerning the nesting level of loops from level 0 to level 2; then, it later saturates from level 2 to level 5. From level 0 to level 2, there was an increase of about 0.110 mA (8.861 mA – 8.53 mA, divided by 3) per level. Then from level to level 5, there was an increase of 0.005 mA (8.882 mA – 8.861 mA, divided by 4) per each level.

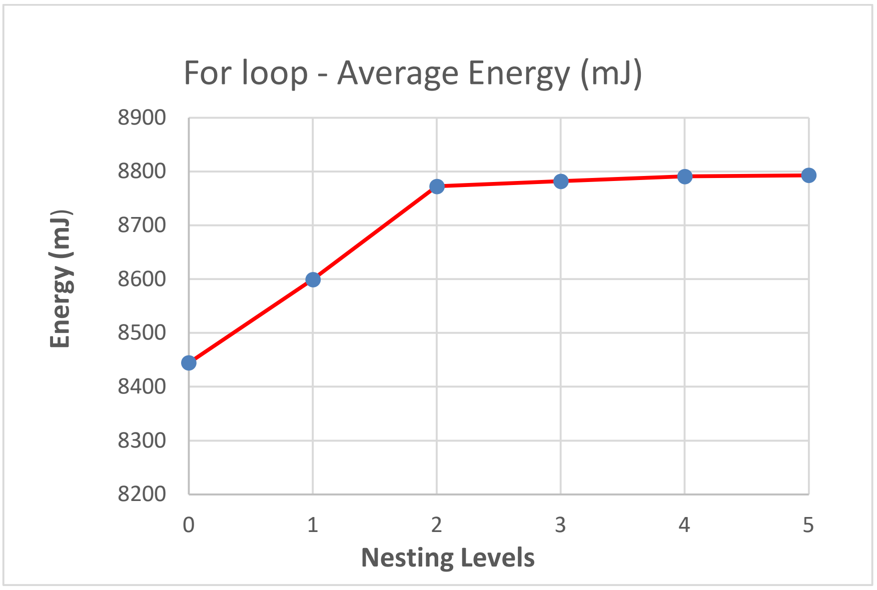


Figure 60: For - Average Energy vs nesting levels of loops.

In Figure 60, the representation of the average energy consumption of the STM32F303RE MCU with respect to the nesting level of loops is depicted. The graph shows a steep rise from level 0 to level 2 and a saturation from level 2 to level 5. From level 0 to level 2, there was an increase in energy consumption of about 109.2 mJ (8772.3 mJ – 8444.7 mJ, divided by 3 levels) per level. Then from level 2 to level 5, there was an increase in energy consumption of about 5.25 mJ (8793.3 mJ – 8772.3 mJ, divided by 4 levels) per level.

For the While loop, the obtained average current and average energy consumption of the STM32F303RE MCU summarized in table 8 for the While loops in the nesting fashion (from level 0 to level 5) are represented in graphs shown in Figure 61 and Figure 62 below, respectively.

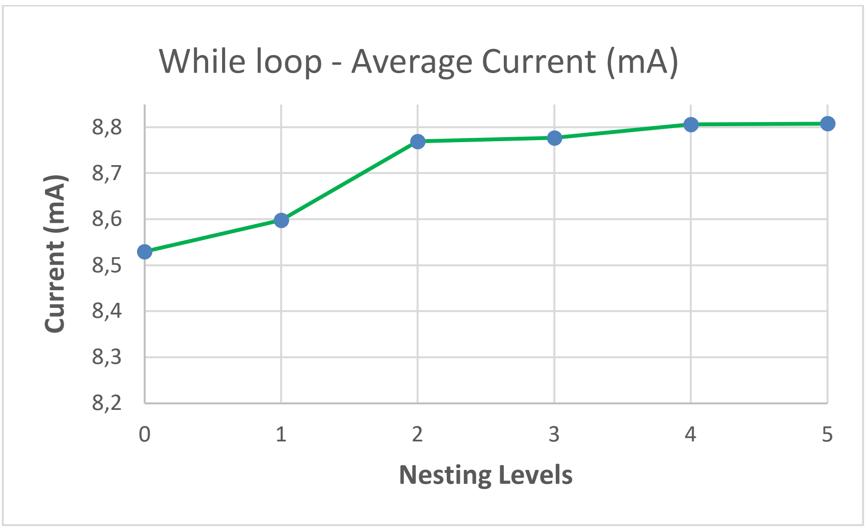


Figure 61: While - Average Current vs nesting levels of loops.

In Figure 61, the image shows a proportional rise (steep rise) of the current consumption of STM32F303RE MCU concerning the nesting level of loops from level 0 and level 2. Then from level 2 to level 5, there was saturation.

There was an increase in current consumption of about 0.080 mA (8.769 mA – 8.53mA, divided by 3 levels) per level from level 0 to level 2 and an increase of about 0.01 mA (8.808 mA – 8.769 mA, divided by four levels) per each level from level 2 to level 5.

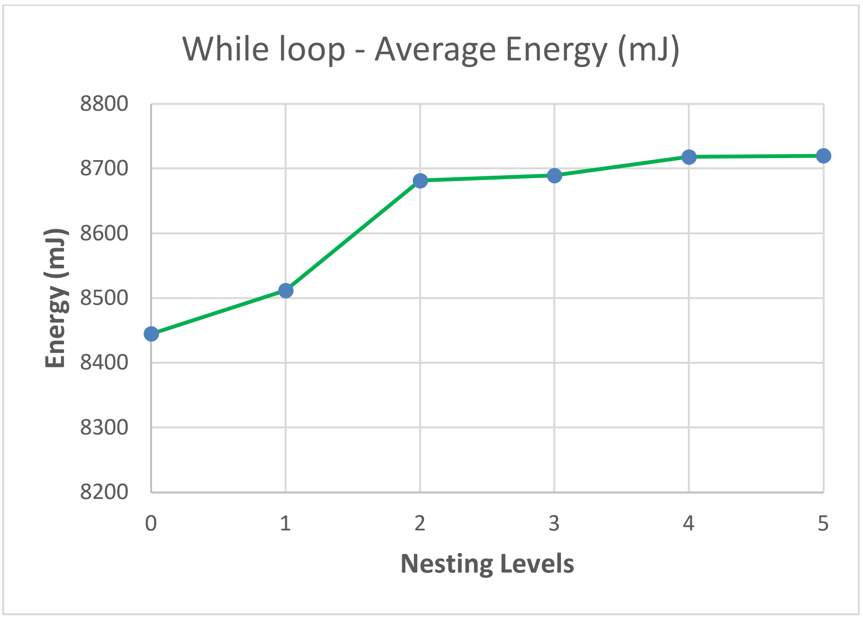


Figure 62: While - Average Energy vs nesting levels of loops.

In Figure 62, the representation of the average energy consumption of the STM32F303RE MCU with respect to the nesting level of loops is depicted. The graph shows a steep rise in the energy consumption of about 78.9 mJ (8681.4 mJ-8444.7 mJ, divided by three levels) per level from level 0 to level 2. Then there was an increase in energy consumption of about 9.6 mJ (8719.8 mJ - 8681.4mJ, divided by 4 levels) from level 2 to level 5.

Furthermore, a comparative evaluation of the energy consumption of the STM32F303 RE MCU executing the For and While loops in nesting fashion are depicted in Figure 63 below:

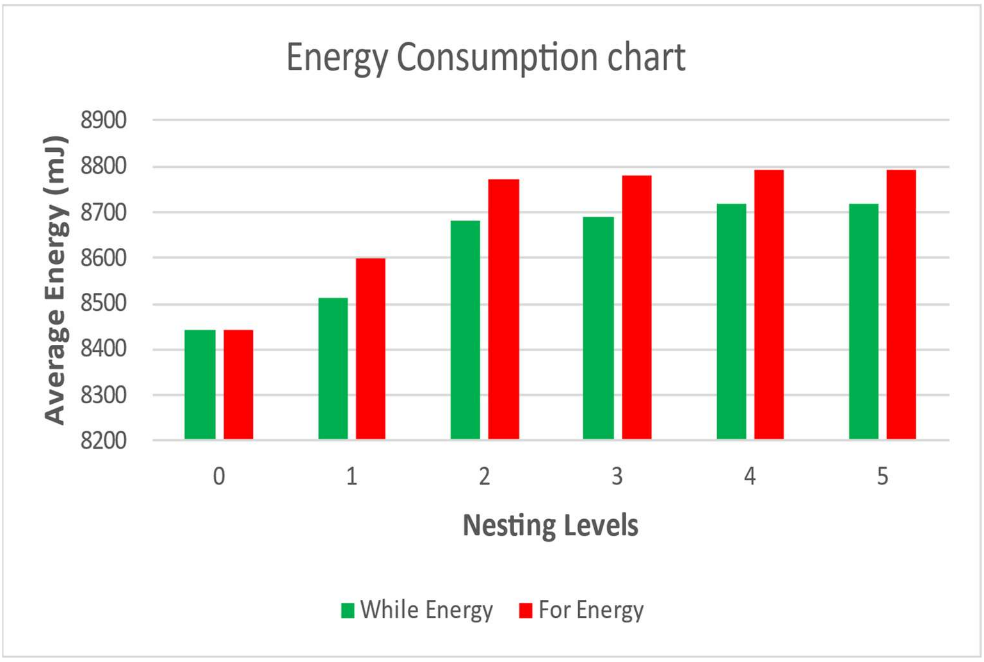


Figure 63: Energy Consumption comparison.

In Figure 63, the average energy consumption of MCU for the While loop in green and the energy consumption of the For loop in red have the same level at level 0 since this is the initial code.

- At level 1, the average energy consumption of the For loop is 8599.2 mJ, and the average energy consumption of the While loop is 8511.9 mJ. This presents a difference of 87.3 mJ (8599.2 mJ – 8511.9 mJ), which means the energy consumption of the STM32F303RE executing the While loop is lower than its energy consumption when executing the For loop.
- At level 2, the average energy consumption of the For loop is 8772.3 mJ, and the average energy consumption of the While loop is 8681.4 mJ. This presents a difference of 90.9 mJ (8772.3 mJ – 8681.4 mJ), which means the energy consumption of the STM32F303RE executing the While loop is lower than its energy consumption when executing the For loop.
- At level 3, the average energy consumption of the For loop is 8782.2 mJ, and the average energy consumption of the While loop is 8689.2 mJ. This presents a difference of 93 mJ (8782.2 mJ – 8689.2 mJ), which means the energy consumption of the STM32F303RE executing the

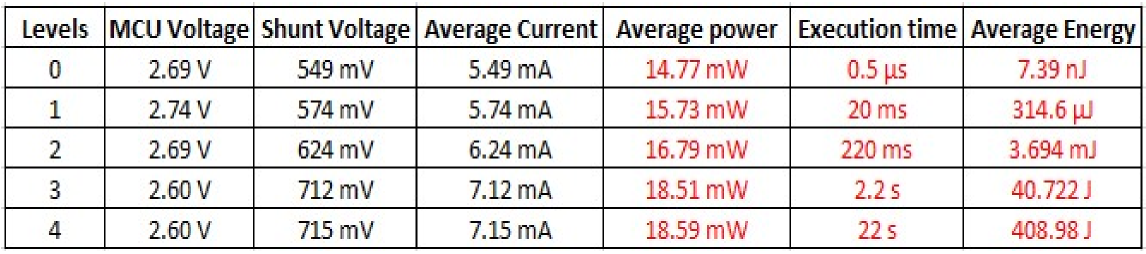
While loop is lower than its energy consumption when executing the For loop.

- At level 4, the average energy consumption of the For loop is 8791.2 mJ, and the average energy consumption of the While loop is 8718 mJ. This presents a difference of 73.2 mJ (8791.2 mJ – 8718 mJ), which means the energy consumption of the STM32F303RE MCU executing the While loop is lower than its energy consumption when executing the For loop.
- At level 5, the average energy consumption of the For loop is 8793.3 mJ, and the average energy consumption of the While loop is 8719.8 mJ. This presents a difference of 73.5 mJ (8793.3 mJ – 8719.8 mJ), which means the energy consumption of the STM32F303RE MCU executing the While loop is lower than its energy consumption when executing the For loop.

This comparison indicates that the STM32F303RE MCU executing the While loops in nesting fashion has a lower energy consumption (about 62.41 mJ average) than executing the For loops in nesting fashion.

In the 2nd approach, the execution time for the various levels has been measured to determine the duration (period) taken by the STM32F303RE MCU to execute the different loop structures. In addition, the effect of these loop structures on energy consumption is also determined. The obtained results are summarized in Table 9 and Table 10.

Table 9: Results of For Loop measurements.



Levels	MCU Voltage	Shunt Voltage	Average Current	Average power	Execution time	Average Energy
0	2.69 V	549 mV	5.49 mA	14.77 mW	0.5 μs	7.39 nJ
1	2.74 V	574 mV	5.74 mA	15.73 mW	20 ms	314.6 μJ
2	2.69 V	624 mV	6.24 mA	16.79 mW	220 ms	3.694 mJ
3	2.60 V	712 mV	7.12 mA	18.51 mW	2.2 s	40.722 J
4	2.60 V	715 mV	7.15 mA	18.59 mW	22 s	408.98 J

Table 10: Results of While loop measurements.

Levels	MCU Voltage	Shunt Voltage	Average Current	Average power	Execution time	Average Energy
0	2.69 V	549 mV	5.49 mA	14.77 mW	0.5 µs	7.39 nJ
1	2.72 V	596 mV	5.96 mA	16.21 mW	20 ms	324.2 µJ
2	2.72 V	598 mV	5.98 mA	16.27 mW	55 ms	894.85 µJ
3	2.70 V	618 mV	6.18 mA	16.69 mW	90 ms	1.502 mJ
4	2.74 V	578 mV	5.78 mA	15.84 mW	115 ms	1.822 mJ

The system was powered by an external supply at 3.3 V. The insertion of the shunt resistor creates a shunt voltage across the resistor (100 Ohm) which is measured with the oscilloscope. In addition, the STM32F303RE MCU supplied voltage is also measured by the oscilloscope.

The average current is the average current consumed by the MCU is then determined using the shunt voltage and the resistor value according to Ohm's law.

The average power is the average power consumed by the MCU during the execution time of the nesting level of loops. This average power is obtained using the MCU supply voltage and the average current consumption in equation (2). As an illustration, the Average Power of the STM32F303RE when executing the For loop level 4 and While loop level 4 are respectively calculated as below:

Average Power = (2.60 V) X (7.15 mA)

= 18.59 mW.

Average Power = (2.74 V) X (5.78 mA)

= 15.84 mW.

The execution time is the time taken by the CPU when executing the level of loops. It is measured by the oscilloscope using the toggling pin.

The average energy is the average energy consumed by the MCU during the time taken to execute the nesting loop level. This average energy is then obtained using the equation's average power and execution time (4). As an illustration, the Average Energy of the STM32F303RE when executing the For loop level 4 and While loop level 4 are respectively calculated as below:

Average Energy = (18.59 mW) X (22 s)

= 408.98 J.

Average Energy = (15.84 mW) X (115 ms)

= 1.822 mJ.

In this instance, the execution time is exponential to the number of levels for the For and While loops structure as per Figure 64 and Figure 65, respectively.

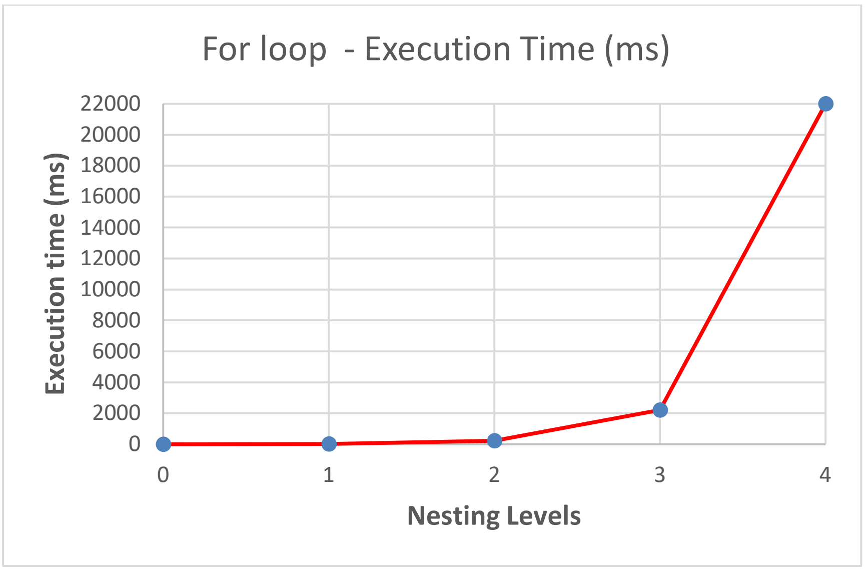


Figure 64: For - Execution time vs nesting levels of loops.

Figure 64 indicates that the execution time of the CPU increases exponentially with respect to the nesting level of loops. For example, with the same number of iterations, the graph shows that the STM32F303RE MCU takes about 20 ms to execute a For loop level 1 and takes about 2.2 s to execute For loop level 4. This implies that the execution time to execute 40000 iterations in the For loop at 8 MHz increases by ten times per nesting level of the For loop. This is due to the number of instructions to execute as the nesting level of For increases.

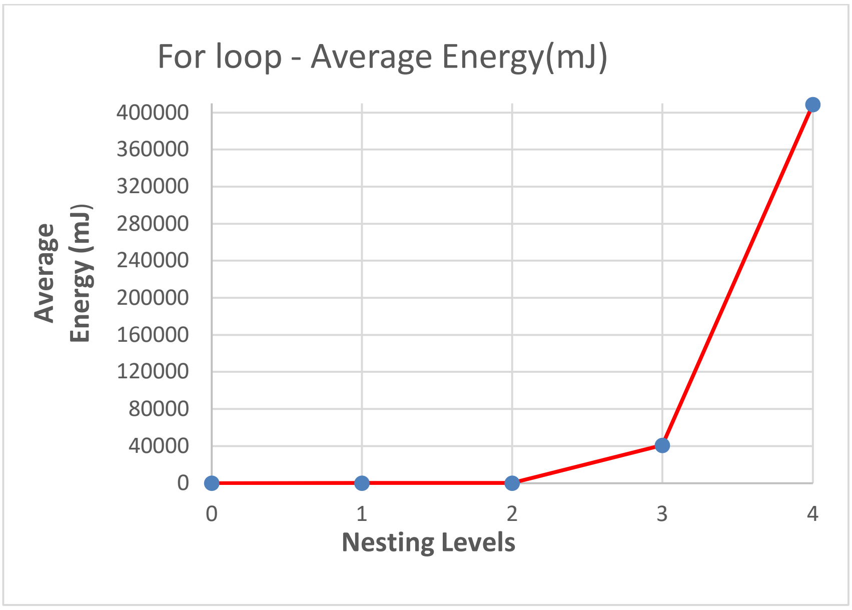


Figure 65: For Loop - Average Energy vs nesting levels of loops.

In figure 65 indicates that energy consumption increases exponentially from 7.39 nJ at the initial code. For example, the For loop level 1 causes an energy consumption of 314.6 nJ. 408.98 J when executing the For loop level 4. Since the energy consumption is a function of the execution time, the energy consumption also increases at about ten times per each level of nesting level of For loops.

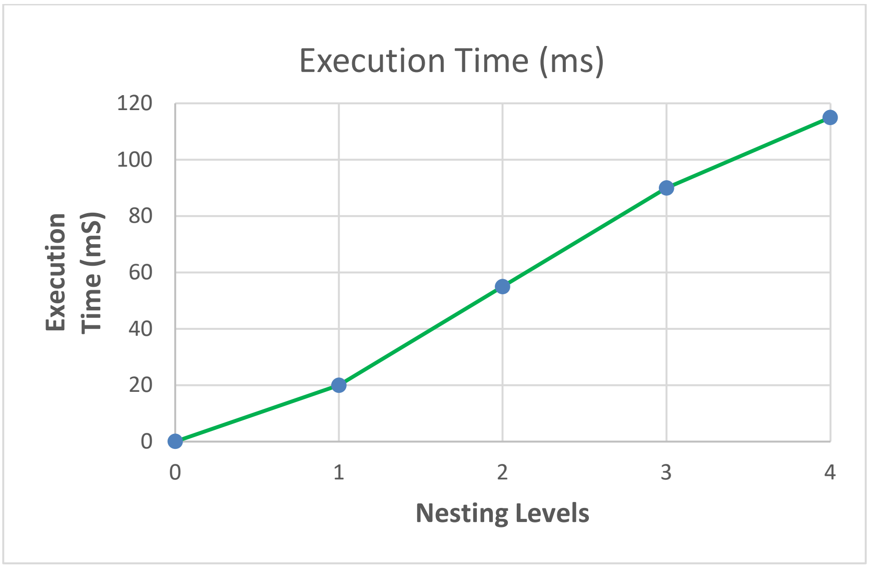


Figure 66: While Loop - Execution time vs nesting levels of loops.

Figure 66 indicates that the execution time of the CPU increases proportionally with respect to the nesting level of loops. For example, with the same number of iterations, the graph shows that the STM32F303RE MCU took about 20 ms to execute a While loop level 1 and takes about 115 ms to execute While loop level 4. This implies that the execution time to execute 40000 iterations in the While loop at 8 MHz increases by 25 from level 0 to level 2 and 35 ms per each nesting level of the While loop from level 2 to level 4.

As in the first approach, the energy consumption increases with the number of nesting levels of the For loop and While loop structures, as shown in Figure 66 and Figure 67, respectively. Again, this is due to the longer execution time of the nesting levels as the nesting level increases.

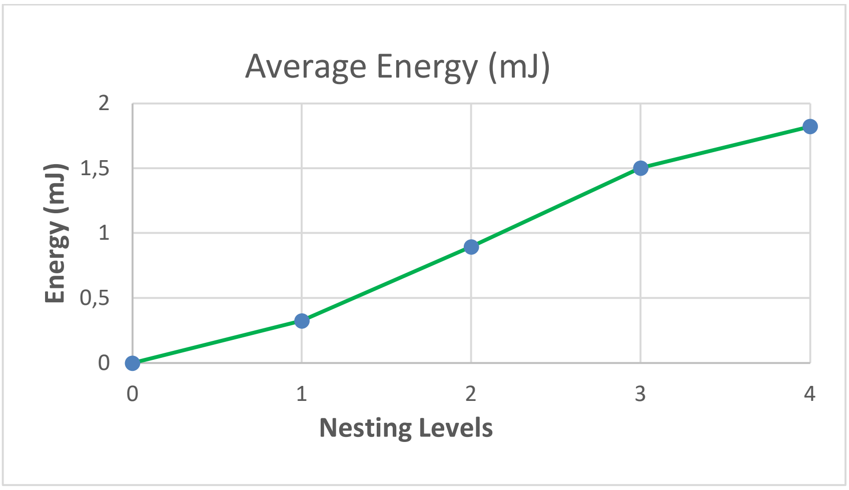


Figure 67: While Loop - Average Energy consumption vs nesting levels of loops.

In figure 67, the average energy consumption of the STM32F303RE MCU is 7.39 nJ when executing the initial code and 1.822 mJ when the STM32F303RE MCU executes the While loop level 4. This is justified by the execution of time, which also increases as the nesting level increases.

Furthermore, the results also indicate that the STM32F303RE MCU consumes more energy when executing the For-loop structures than running the While loop structures, as per Figure 68.

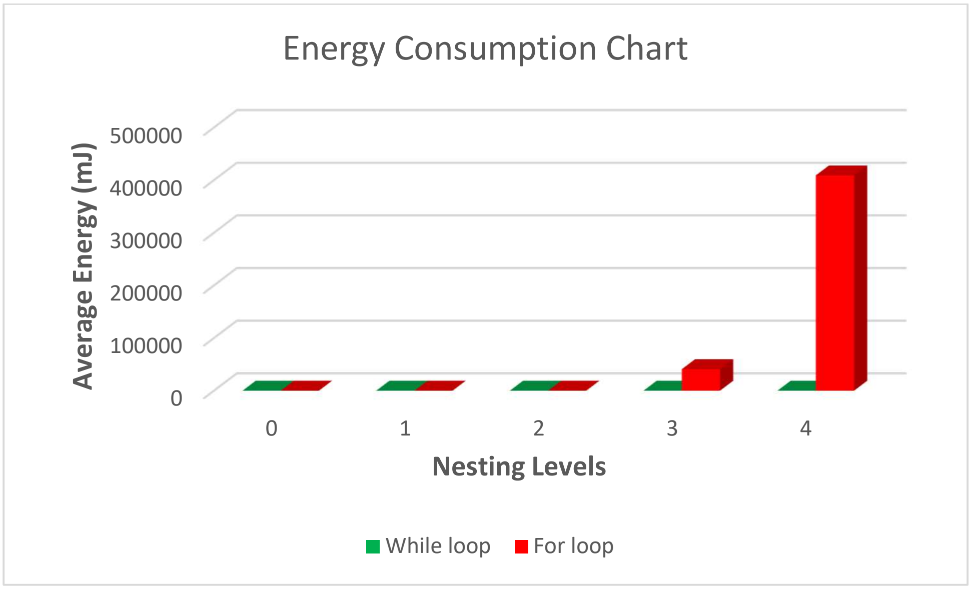


Figure 68: Energy Consumption comparison.

The above graph indicates that the energy consumption of the STM32F303 RE MCU executing the While loop ranges from 7.39 nJ to 1.822 mJ. This implies an increase of energy consumption at an average of about 0.362 mJ per nesting level of the While loop. Meanwhile, the energy consumption of the STM32F303 RE MCU executing the For loop ranges from 7.39 nJ to 408.98 J. This implies an increase of energy consumption at an average of about 81.80 J per nesting level of For Loop.

For battery-powered embedded systems, the higher energy consumption of the MCU causes an embedded system's overall total energy consumption to increase. This causes the battery life of these systems to be reduced.

While with the first approach, it is indicated that the current (Power and energy) consumption of MCU rises as the nesting level of the loop is increased. The second approach also shows that the current (power and energy) consumption of MCU rises. In the second approach, the nesting level of loops also increases the time taken by the processor to execute this loop (i.e., execution time). The rising of the current consumption of the MCU impacts the design of the embedded systems as these systems operate under a limited power budget or depend on batteries as the sole source of energy, as stated earlier.

Reducing the current consumption extends the battery life of these devices and increase their lifespan.

Furthermore, these embedded systems are also designed as real-time systems where the tasks need to meet deadlines which are time constraints. Therefore, an increase in execution time can cause these real-time systems to fail to meet the time requirements when running.

It is also observed that the For loop nesting levels has a higher power (energy) consumption than the While loop nesting levels. This is because the For loop structures begin by initializing the counter variable, testing the condition, executing the instructions, and incrementing the counter variable. This process is repeated until the condition is no longer valid. Meanwhile, the While loop only tests the condition before execution of the instructions within the loop.

By reducing the number of loops in nesting fashion, the current (power and energy) consumption of MCU is also reduced. The execution time also is reduced. This approach allows the implementation of energy-aware systems, and it is appropriate to consider them in real-time systems. The work showed that the aim of this research was achieved as embedded systems developers should be mindful of the use of nesting loop structures for their applications where possible.

CHAPTER 5

CONCLUSION AND FUTURE WORK.

In conclusion, this work aimed to develop an energy-aware real-time application technique to reduce the energy consumption due to computation. The nesting loop structures (For and While) were executed by the Nucleo-64 STM32F303RE MCU, which executed the different levels of loops. The energy consumption of the For loop and While loop structures in nesting fashion up to 5 levels of loops have been measured in the first approach, then up to 4 levels in the second approach. The measurements were performed on the STM32F303RE MCU of the Nucleo – 64 board.

The obtained results demonstrated that the energy consumption rises with the number of loops in nesting fashion. In the first approach, the average energy consumption per level of For loop was about 109.2 mJ from level 0 to level 2, and then the increase per level was 5.25 mJ from level 2 to level 5. The average energy consumption per level of While loop was about 78.9mJ from level 0 to level 2, and the average energy consumption was about 9.6 mJ from level 2 to level 5. The first approach demonstrated that the STM32F303 RE executing the While loop consumes about 62.41 mJ less per level than executing the For loop structure.

The second approach indicates an increase exponentially from 7.39 nJ to 408.98 J when executing the For loop structure. The energy consumption increased proportionally while executing the While loop structure. This implies an increase of 0.362 mJ per level for the While loop. The second approach also demonstrated that the STM32F303RE consumes about 81.80 J less executing the While loop structure than executing the For loop structure. The second approach also indicated that the execution time increases with the number of levels in the nesting fashion. The execution time to execute 40000 iterations in the While loop at 8 MHz increases proportionally by 25 ms from level 0 to level 2 and 35 ms from level 2 to level 4 per each nesting level of the While loop. The execution time to execute 40000 iterations in the For loop at 8 MHz increases by ten times per nesting level of the For loop.

Both experiments showed that the MCU consumes more energy executing the For loop structures than executing the While loop structures. As this work aimed to come out with a technique in order to develop energy-aware applications for embedded systems, these measurements indicate that by reducing the level of nesting loops where applicable will result in decreasing the energy consumption during computation as well. Therefore, embedded software developers must minimise the level of loops where possible and select the proper structure when developing applications. This will decrease the energy consumed by the MCU while executing these applications and the overall energy consumed by the embedded system. Furthermore, reducing the energy consumption extends the battery life span of mobile and battery-powered embedded devices.

In future, this work can be extended for other C language structures such as Do while, if -Statement. The combination of the For loop and While loop structure can also be evaluated to see if optimal power consumption can be achieved. Additionally, different MCUs can be used for this work. Furthermore, more advanced measurement methods can also be used in the experiments.

REFERENCES

Amos, B., 2020. *Hands-On RTOS with Microcontrollers.* s.l.:Packt.

Andreas Ermedahl, Jakob Engblom, 2007. *Execution Time Analysis for Embedded Real-Time Systems.* [Online] Available at: https://www.academia.edu/24443794/Execution_Time_Analysis_for_Embedded_Real-Time_Systems
[Accessed 16 May 2021].

Arm, 2020. *Arm Cortex-M4 Datasheet,* s.l.: s.n.

Attakorn Lueangvilai, Christina Robertson, and Christopher J. Martinez, 2012. A Dynamic Frequency Controlling Technique for Power Management in Existing Commercial Microcontrollers. *Journal of Computing Science and Engineering,* 6(2), pp. 79-88.

C. Chang, S. Muftic and D. J. Nagel, 2007. *Measurement of Energy Costs of Security in Wireless Sensor Nodes.* s.l., s.n., pp. 95-102.

Chabini and Wolf , 2003. *Minimizing Variables' Lifetime in Loop-Intensive Applications.* Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-45212-6_8, s.n.

Chakrabarty, V. S. a. K., n.d. *Real-Time Task Scheduling for Energy-Aware Embedded Systems.* Durham, s.n.

Chi Ta Wu, Ang-Chih Hsieh and Ting Ting Hwang, 2006. *Instruction buffering for nested loops in low-power design.* s.l., s.n.

Davies, J. H., 2008. *MSP430 Microcontroller Basics.* s.l.:Newnes.

G. Guimaraes; E. Souto; D. Sadok; J. Kelner, 2005. *Evaluation of security mechanisms in wireless sensor networks.* s.l., s.n.

Haulin, L., 2018. *A state-based method to model and analyze the power consumption of embedded systems,* s.l.: s.n.

Horton, I., 2013. Chapter 4 - Loops. In: *Beginning C, Fifth Edition.* s.l.:Apress. © 2013. Books24x7 http://library.books24x7.com.libproxy.cput.ac.za/toc.aspx?bookid=52439> (accessed May 16, 2021).

Ibrahim, D., 2010. *SD Card Projects Using the PIC Microcontroller.* s.l.:Newnes.

Ibrahim, D., 2014. Chapter 1 - Microcomputer Systems. In: *Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC..* s.l.:Elsevier Science & Technology.

Ibrahim, D., 2015. *PIC32 Microcontrollers and the Digilent chipKIT: Introductory to Advanced Projects..* s.l.:Newnes.

Ibrahim, D., 2019. *ARM-based microcontroller projects using mbed First edition.* s.l.:Oxford, England ; Cambridge, Massachusetts : Newnes.

J. Ramanujam, Jinpyo Hong, M. Kandemir, A. Narayan and A. Agarwal, 2006. Estimating and reducing the memory requirements of signal processing codes for embedded systems. *IEEE Transactions on Signal Processing doi: 10.1109/TSP.2005.855086,* 54(1), pp. 286-294.

J. T. Russell and M. F. Jacome, 1998. *Software power estimation and optimization for high performance, 32-bit embedded processors.* s.l., s.n.

Juan Castillo, H. P. E. V. M. M., 2004. Energy Consumption Estimation Technique in Embedded Processors with Stable Power consumption based on Source-Code Operator Figures.

Lehlogonolo P. I. Ledwaba, G. P. H. S. V. S. J. I., 2018. Performance Costs of Software Cryptography in Securing New-Generation Internet of Energy Endpoint Devices. *IEEE Access,* Volume 6, pp. 9303-9323.

Majid Sarrafzadeh, F. D. R. J. T. M. A. N., 2006. *Low power light-weight embedded systems.* Tegemsee, s.n.

Marwedel, P., 2018. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems,and the Internet of Things Third Edition..* Dortmund: Springer.

Mittal, S., 2014. A Survey of Techniques For Improving Energy Efficiency in Embedded Computer Systems. *International Journal of Computer Aided Engineering and Technology,* 6 (4)(10.1504/IJCAET.2014.065419ff. ffhal-01101854f), pp. 440-459.

P. Ruberg, K. Lass and P. Ellervee, 2015. *Microcontroller energy consumption estimation based on software analysis for embedded systems.* s.l., s.n.

Paul Deitel, Harvey Deitel, 2010. *C HOW TO PROGRAM.* Upper Saddle River, New Jersey 07458: Pearson Education, Inc.

Qing Li, Caroline Yao, 2003. In: *REAL-TIME CONCEPTS FOR EMBEDDED SYSTEMS.* San Francisco: CMP Books.

Qingchen Zhang, M. L. L. T. Y. ,. Z. C. ,. a. P. L., 2019. Energy-Efficient Scheduling for Real-Time Systems Based on Deep Q-Learning Model. *IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING,* 4(1), pp. 132-138.

Siegesmund, M., 2014. Chapter 6: Statements. In: *Embedded C programming: Techniques and Applications of C and PIC MCUS.* s.l.:Elsevier Science & Technology, 2014, pp. 73-94.

Siegesmund, M., ProQuest Ebook Central, https://ebookcentral.proquest.com/lib/cput/detail.action?docID=1802024.
Chapter 6: Statements. In: *Embedded C programming: Techniques and Applications of C and PIC MCUS.* s.l.:Elsevier Science & Technology, 2014, pp. 73-94.

Stewart, D. B., September 2006. *Measuring Execution Time and Real-Time Performance.* Boston, s.n.

STMicroelectronics, 2016. *Datasheet-STM32F303xD STM32F303xE,* s.l.: s.n.

STMicroelectronics, 2020. *Data brief-STM32 Nucleo-64 boards,* s.l.: s.n.

STMicroelectronics, 2020. *UM1724 User Manual,* s.l.: s.n.

Sutter, E., 2002. *Embedded Systems Firmware Demystified.* Kansas: CMP Books Lawrence.

T. Rauber and G. Rünger, 2018. *How do loop transformations affect the energy consumption of multi-threaded Runge-Kutta methods?.* s.l., s.n.

V. Dalal and C. P. Ravikumar, 2001. *Software power optimizations in an embedded system.* s.l., s.n.

V. Tiwari, S. Malik and A. Wolfe, 1994. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 2(4), pp. 437-445.

Vishnu Swaminathan, K. C., n.d. Real-Time Task Scheduling for Energy-Aware Embedded Systems. *Duke University.*

Vlad Radulescu, S. A. A. M. C., 2014. *A Heuristic-Based Approach for Reducing the Power Consumption of Real-Time.* s.l., s.n.

Wilmshurst, T., 2009. *Designing Embedded Systems with PIC Microcontrollers : Principles and Applications.* s.l.:Elsevier Science & Technology.

X. Zhou, B. Guo, Y. Shen and Q. Li, 2009. *Design and Implementation of an Improved C Source-Code Level Program Energy Model.* s.l., s.n.