NTNU

Norwegian University of
Science and Technology

A Light-Weight Operating System for Internet of Things Devices

Emekcan Aras

Embedded Computing Systems
Submission date: June 2016
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

A Light-weight Operating System for Internet of Things Devices

Emekcan Aras

June 2015

PROJECT / MASTER THESIS

Department of of Engineering Cybernetics

Norwegian University of Science and Technology

Supervisor: Associate Professor Amund Skavhaug

Preface

This master thesis was written the spring of 2016 at the Norwegian University of Technology and Science, Departmen of Engineering Cybernetics. It has been a part of my master program EMECS(European Master in Embedded Computing Systems) and this thesis concludes my Master's degree.

This Master's thesis is a contribution and new way of look to embedded operating systems. It is not the continuation of the specialization project of the fall semester 2015 even if the overall goal of developing the robust wireles networks still stands. This thesis has not been submitted previously and has been made by independent work. In addition, a conference paper is being written based on this paper which focus more on IOT service layer.

The thesis can be divided into two important parts. The first part is developing a very small "microscopic" operating system. The design steps and background lies behind the design is explained in detail in this particular paper.

The second part is radio communication protocol and IOT service layer which is presented as the novelty of the project. Since there are many different operating system and architecture can be found in the market, we have revealed that a specific embedded operating system designed for IOT and wireless sensor networks that enables resource and information sharing, would be a good research topic. Therefore, the particular project can be considered as the collection of the related background education, embedded software skills, hours of discussions and sparks of new ideas about embedded operating systems.

This particular paper has been written for three types of readers. The one who would like to read and learn something about operating systems and embedded system, the students who would like to continue the project and the professors who will asses this paper.

Trondheim, June 2016

Emekcan Aras

Acknowledgment

I have now spent two years in abroad since I left Istanbul for this Joint Degree program. I have lived in two beautiful cities Southampton and Trondheim and I have found to chance to meet with wonderful people in these years. I will soon be graduated and leave Trondheim but I will never forget this period of my life.

I would like to thank my friends in EMECS program who have worked hard with me and never hesitate to help me for anything. In addition, I would like to thank my beloved friends in Istanbul, Glasgow, Berlin and Milano who gave me the inspiration to apply this program and supported me during it, we may live away from each other but they will never be forgotten.

I would like to thank my supervisor professor Amund Skavhaug for the help, guidance and invaluable ideas he has given during the semester.

Lastly, I would like to thank my beloved father and mother who always supported me during this program and taught me that another world without any war or discrimination is possible.

E.A.

Summary and Conclusions

Nowadays, all around of us has been surrounded by wireless devices. More than a decade, engineers have been using these wireless devices in industry to measure environmental data. Large networks, which are formed by hundreds or thousands sensor nodes, are not easy to build, maintain and observe. Moreover, with the trending concept called Internet of Things, devices have gained different functionalities and the different type of networks started to develop. In order to design an efficient and low cost sensor and IOT network, a different type of architecture in both network side and low-level software side must be developed.

Designing a specific operating system motivated us earlier part in this project. After detailed investigation, it is revealed that resource-constrained embedded devices need a new type of embedded operating system specifically developed for sensor and IOT network. Moreover, in the market, there are many choice for embedded operating systems. However, an open-source operating system for embedded devices in the sensor and IOT networks is needed and could be useful for educational purposes as well as commercial purposes.

In the light of the motivations that is mentioned at the paragraph above, a light-weight embedded operating system has been designed and developed from scratch in this thesis. In addition to that, a radio protocol and an IOT service layer which act as middleware in the network have been designed and implemented into the embedded operating system. All the background lies behind this project and the designing steps are explained in detail in this thesis. In addition to this, a conference paper about the operating system particularly IOT service layer is being written.

Keil uVision5 software development platform was used to develop the mentioned operating system in this project. Detailed investigation and research were done about embedded systems, operating systems, distributed systems and wireless communication before the project. Afterwards, entire software libraries for the kernel and hardware drivers were developed from scratch by using C and Assembly language. Since the project is focused on embedded devices, one of the low-power processor architectures called ARM Cortex-M0 architecture has been used as the target platform. An SOC from Nordic Semiconductor called NRF51822, which contains this processor architecture, is used inside of the devices in this project. In order to test and validate the

results and functionality of the system, two different development kits have been used provided by the vendor company. Detailed tests scenarios have been tried and a small working sensor network/cluster has been formed by using these devices and the operating system.

At the end of the project, a small size operating system which requires around 8Kb of flash memory and 7Kb of RAM, has been created. Tasks which is given to devices has been efficiently executed without causing any overhead or failure. In addition to this, functionality which is provided by the IOT layer has been achieved as well. Two types of device are introduced with the proposed IOT layer. A standard component for sensor and IOT networks called “node device” and supervisor component of the network called “manager device” are tested. After tests and verifications, it has been seen that resource and information sharing across the network is achieved.

In this thesis project, resource-constrained low profile embedded devices and architectures selected as the target device group. It is revealed that using a specifically design operating system, these types of devices could be used in many applications with reasonable prices. Moreover, the proposed operating system layer could bring a new way of thinking on the sensor and IOT device networks and could be used as the solution for problems and challenges. Furthermore, with the user-friendly design particular operating system could be very useful in educational purposes.

List of Abbreviations and Symbols

IOT Internet of Things

RAM Random Access Memory

CORBA Common Object Request Broker Architecture

SOA Service Oriented Architecture

OSI Open System Interconnection

OS Operating System

API Application Porgramming Interface

HAL Hardware Abstraction Layer

CPU Central Processing Unit

TCP IP Transmission Control Protocol-Internet Protocol

SOC System on Chip

TI Texas Instruments

RTOS Real-Time Operating System

ROM Read-Only Memory

GUI Graphical User Interface

IPC Inter-process Communication

RF Radio Frequency

VLF Very Low Frequency

EHF Extremely High Frequency

ASK Amplitude Shift Keying

PSK Phase Shift Keying

FSK Frequency Shift Keying

PRN Pseudo-Random-Noise

DSSS Direct Sequence Spread Spectrum

FHSS Frequency Hopping Spread Spectrum

ISM Industrial Scientific Medical

IO Input-Output

GPIO General Purpose Input-Output

PCB Printed Circuit Board

RSSI Received Signal Strength Indication

AES Advanced Encryption Standard

CRC Cyclic Redundancy Check

DMA Direct Memory Access

I2C Inter-Integrated Circuit

SPI Serial Peripheral Interface

PSP Process Stack Pointer

MSP Main Stack Pointer

ISR Interrupt Service Routine

SVC Service Call

RISC Reduced Instruction Set Computing

MIPS Microprocessor without Interlocked Pipeline Stages

List of Figures

1.1 Prediction about IOT devices [5]. . . . 3
1.2 A survey about Internet of Things 1 [8]. . . . 6
1.3 A survey about Internet of Things 1 [8]. . . . 7

2.1 A general structure of Embedded Systems [9]. . . . 12
2.2 Monolithic System Architecture [12]. . . . 14
2.3 Layered System Architecture [13]. . . . 15
2.4 Microkernel Architecture Structure [12] 16
2.5 Difference between Client-Server and Multi-Tier Architecture 20
2.6 Implementation of CORBA 21
2.7 Structure of SOA 22
2.8 OSI model [19] 23
2.9 Frequency Ranges [20] 24
2.10 Types of Digital Modulation [21] 25
2.11 Spread Spectrum Communication [22] 26
2.12 Overview of Wireless Technologies in the market [24] 29

3.1 Kernel Structure 37
3.2 Structure of OS Services with API and Application Layer 39
3.3 Proposed Operating System Architecture 40
3.4 Structure of IOT Service 42
3.5 Block Diagram of NRF51822 [34] 45
3.6 Nrf51822 Radio Module Block Diagram [35] 47

4.1 Structure of Hardware Drivers 49
4.2 Interaction between HAL and Hard Drivers and Structural Design 54

5.1 Multitasking and Context Switching[37] 56
5.2 Context Switching from one task to another[37] 57
5.3 Context Switch without PendSV[37] 59
5.4 Flow diagram of system tick handler and dispatcher 61
5.5 Transition of Task States 64
5.6 The flow chart of priority-driven scheduler during the context switching 65

6.1 Structure of Linked List [38] 68
6.2 Main memory after the memory allocation [39] 70
6.3 The structures and functionality of API 71

7.1 Structure of Soft-Device[41] 77
7.2 On-air packet layout [35] 78
7.3 Structure of Radio Library and Interactions 80
7.4 Packet Layout of the New Radio Protocol 81
7.5 The stack structure of the radio protocol based on OSI model 82

8.1 Classical Cluster Approach 86
8.2 Cluster with Main Cluster 87
8.3 Cluster with Nodes Net 88
8.4 IOT service software structure 91
8.5 Structure of Mailing System and Yellow Pages 93
8.6 Flow Chart of Registration Process 98

9.1 NRF518222 Development Kit and NRF51822 Dongle 100
9.2 Structure of Test Scenario 101

List of Tables

2.1 Evaluation of Embedded Operating Systems 33
2.2 Comparison of Operating Systems Embedded 34

8.1 Pros and Cons of Suggested Strcutres. 89
8.2 Possible Problems and Solutions of Suggested Structures 90

9.1 Size of the Embedded Operating System 102
9.2 Execution Time of IOT Service Layer 102

Listings

4.1 Hal structure for GPIO 52

4.2 HAL structure of communication drivers 53

5.1 Task Control Structure 58

5.2 Mutex Structure 60

6.1 Configuration File of Particular Operating System 74

8.1 IOT node resources 93

8.2 IOT node resources 94

Contents

Preface i

Acknowledgment ii

Summary and Conclusions iii

1 Introduction **2**

1.1 Internet of Things and Embedded Systems 2

1.2 Example Applications 4

1.3 Challenges and Problems in IOT 5

1.4 A specific Operating System for Internet of Thing Devices 8

2 Background Research **11**

2.1 Operating Systems and Real-Time Systems 12

2.2 Distributed Systems 18

2.3 Wireless Communication 22

2.4 Related Work 29

3 Details and Features **35**

3.1 Simple and Light-weight Design 36

3.2 Two Dimensional Kernel Architecture 39

3.3 Internet of Things Service and Resource Sharing 41

3.4 3.4. Development Platform 43

4 Hardware Abstraction Layer and Drivers **48**

5 Kernel Architecture **55**

5.1 5.1. Multitasking and Context Switching 56
5.2 Priority-Driven Scheduler and Real-Time Compatibility 62

6 Other System Components **67**
6.1 Utilities Library 67
6.2 Application Programming Interface 69
6.3 User defined functions 73

7 Radio Library, Stack and Protocol **76**

8 IOT Service and Security **84**
8.1 Network Structure 85
8.2 IOT Service 89
8.3 Security Issue 95

9 Tests and Results **99**

10 Discussion, Future Work, and Conclusion **103**
10.1 Discussion 103
10.2 Future Work 109
10.3 Conclusion 112

11 Bibliography **116**

Chapter 1

Introduction

1.1 Internet of Things and Embedded Systems

In last twenty years, many technologic achievements have been happened in the Integrated Circuit industry. These have led to develop small, energy efficient and faster processors and microcontrollers. Nowadays even small microcontrollers which cost ten to twenty dollars, have enough CPU power and memory space to achieve complicated tasks such as managing with TCP/IP stack or Bluetooth communication. Even some operating systems can be supported by these kind of low cost microcontrollers. Because of these, every kind of embedded systems have started to connect with each other and internet. This concept is called Internet of things (IOT).

Internet of Things represents the devices, which communicates and shares data among each other using different protocols [1]. In 1993, a system developed in University of Cambridge to help people working in other parts of the building, avoid pointless trips to the coffee pot by providing, on the user's desktop computer, a live 128×128 greyscale picture of the state of the coffee pot. Since it was real time and online application [2], this system is considered as first example of internet of things. However the name of the concept actually used by Kevin Ashton in 1999 in a presentation, which was given by him to a private cooperation [3]. However, this concept has gained broader meaning during the years. Nowadays Internet of Things is defined as "a pervasive and ubiquitous network which enables monitoring and control of the physical environment by collecting, processing, and analyzing the data generated by sensors or smart objects." [4]

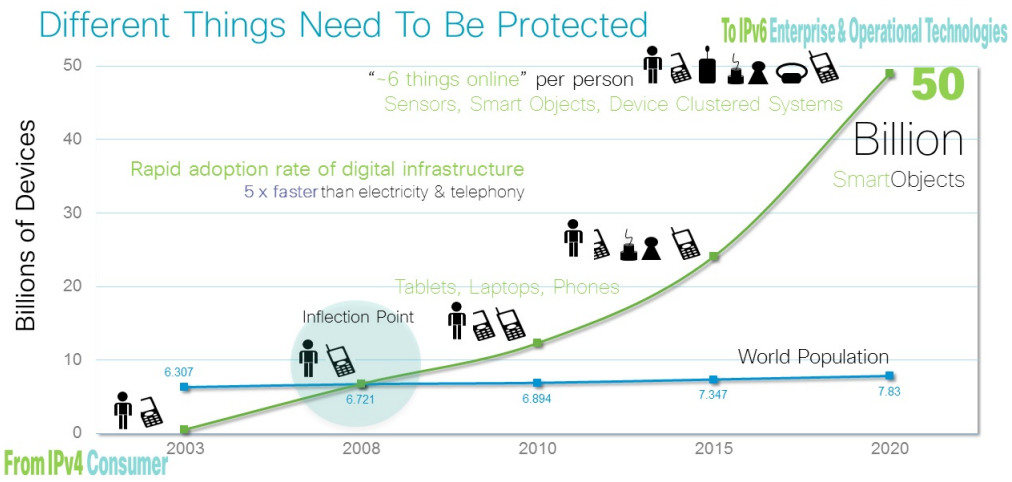

Figure 1.1: Prediction about IOT devices [5].

As it can be seen from the figure 1.1, for almost 15 years, we as human kind have been trying to connect everything to each other and to Internet. The estimations say that in 2020 there will have been 50 billion IOT object in the World. That means there will be six device per person. Therefore, it has already become dominant concept for the industry. In order to implement this concept, new approaches in communication and software had been started to investigate and develop.

The semiconductor vendors have already started to produce architectures desired by the market. System on Chip devices are becoming more popular day by day. A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio frequency functions all on a single chip substrate. The contrast with a microcontroller is one of degree. Microcontrollers typically have under 100 kB of RAM (often just a few kilobytes) and often really are single-chip-systems, whereas the term SoC is typically used for more powerful processors, capable of running software such as the desktop versions of Windows and Linux, which need external memory chips (flash, RAM) to be useful, and which are used with various external peripherals. In short, for larger systems, the term system on a chip is hyperbole, indicating technical direction more than reality: a high degree of chip integration, leading toward reduced manufacturing costs, and the production of smaller systems. Many systems are too complex to fit on just one chip built with a processor optimized for just one of

the system's tasks. SOCs are very common in the mobile electronics market because of their low power consumption. [6] These developments in both silicon industry and wireless network industry have led to new communication protocols and new low cost small processor in the market. System on Chip devices with the radio chip has been introduced to the market. In addition, the protocol stacks for ZigBee, Bluetooth Low Energy, ANT, and Thread have been implemented inside of those SOCs.

1.2 Example Applications

Together with embedded (resource constrained) devices, many different applications have been developed and implemented on many different fields.

- Automotive Industry: Consumers want the digital experiences in their vehicles to align with the ones they enjoy everywhere else. When tied to the IoT, the car is an integral part of the interdependent web of information flow, turning data into actionable insight both inside the car and in the world around it
- Energy Industry: Through the IoT, the power grid's countless devices can share information in real time to distribute energy more efficiently. Consumers, businesses, and utility providers get the information they need to better manage their energy-connected things to consume less energy.
- Fitness and Environment: Fitness bands, watches, and even smart clothes are able to monitor and transmit data on your daily activity levels through step counting, heart rate and temperature.
- Health: These wearables monitor crucial health factors like oxygen saturation, heart rate and more, and can communicate any results outside of a programmed range to the patient and to her physician.
- Retail: Retailers use the IoT to provide personalized and immersive experiences that keep shoppers coming back. Gathering and organizing data is only part of the challenge.

- Smart Manufacturing: The benefits of IoT products include software and hardware side that ease and accelerate design time for smart manufacturing application.
- Smart Buildings: The IoT is enabling a transformation in building efficiency and management.
- Home Automation: From enhancing security to reducing energy and maintenance costs, there are a many of innovative IOT technologies for monitor and control of smart homes.

1.3 Challenges and Problems in IOT

As it was mentioned earlier in this paper, novel architectures in digital design and networks protocols make the development of Internet of Things networks easier. All around us surrounded by smart objects. Moreover, most of the people have started to work in smart buildings, live in smart cities. In our daily life, we use devices to measure and track our health related most private data such as heart rate, blood pressure and even our location.

On the other hand, the industry has realized the importance of IOT objects as well. A concept called Industry 4.0 is becoming more important day by day for the industry. Industry 4.0 is a collective term embracing a number of contemporary automation, data exchange and manufacturing technologies. It had been defined as a collective term for technologies and concepts of value chain organization, which draws together Cyber-Physical Systems, the Internet of Things and the Internet of Services. [7] In industry, large sensor networks and smart objects have started to use in their production line. Demand of IOT devices in industry has been increasing for last ten years. At the figure 1.2 , needs of industry can be seen.

Secure devices is the most important need together with easy integration. In addition, sharing device resource/data and remote monitoring/management of the devices can be considered as second most important features as well. In order to develop embedded IOT devices according to needs of industry, we as developers need to identify the problems and challenges and find to way to overcome.

The figure 1.3 gives the perspective from developer and producer point of view. According to developers integration of the devices and total cost are the most difficult hurdles to overcome.

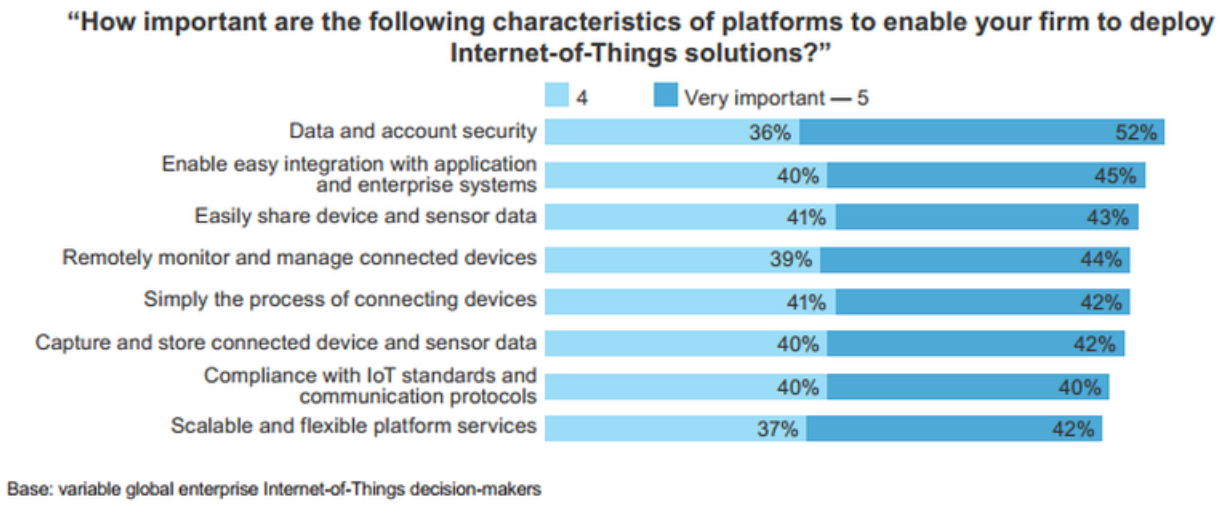

Figure 1.2: A survey about Internet of Things 1 [8].

Moreover, to develop hundred percent secure devices is also one of the noticeable challenges. Furthermore, since low cost embedded devices have turned out to be the best solution for internet of things as well as the sensor networks, large networks, which may be formed by hundreds or thousands of nodes, can be created easily with these devices. However, in the application side, this new trend brought some new problem to the networks and software. Since wireless communication not reliable as wired communication, there are always some minor errors such as missing packet etc. in the wireless network. In addition to that, most of the sensor network are used in critical measurements in the harsh environments. Sensor nodes may become unavailable because of lots of different reasons. Sometimes these failure may not create big problems. In addition, we started to use devices, which can collect our private data such as heart rate, blood pressure, location etc. We make everything accessible. That also caused problems in security side. Together with other problems, internet of things devices and networks could be unsecure and inefficient. In order to prevent devices and network such scenarios, a new approach just for the internet of things devices must be developed in both the operating system , software and network side. Challenges must be detailed and identified better to develop a novel embedded Internet of Things devices. Challenges and difficulties can be divided into three part. These parts are embedded software/operating system, hardware/resource constrained and network related challenges.

- Software/Operating System Challenges;

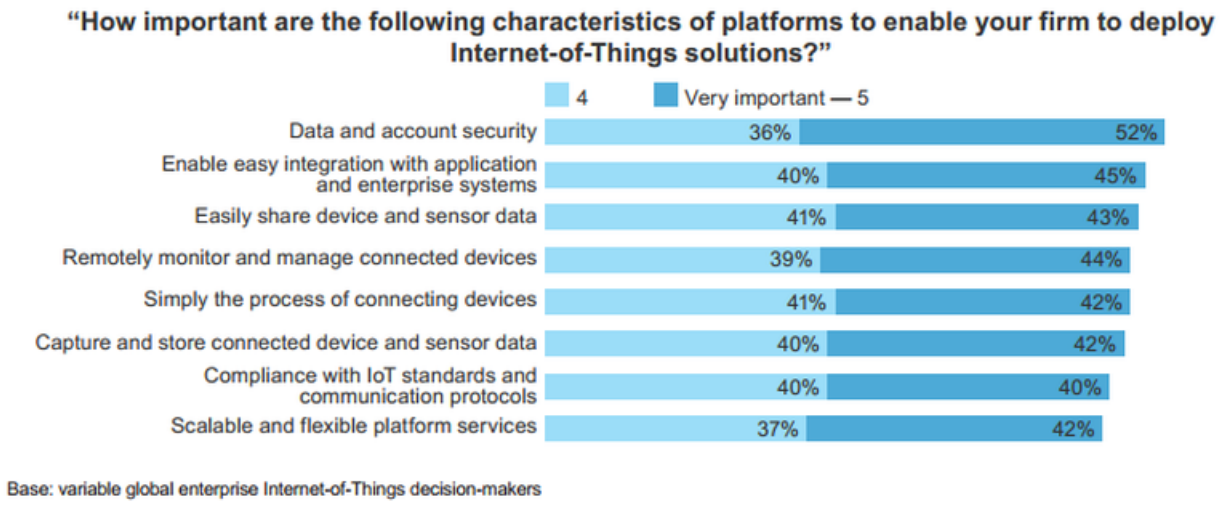

Figure 1.3: A survey about Internet of Things 1 [8].

  - Integration and implementation
  - Development of Real Time Kernel
  - Resource sharing
  - Secure and upgradable Operating System

- Hardware Challenges;
  - Memory Constrain(hard to build an Operating System inside of small microcontroller)
  - Limited Encryption Capabilities
  - Energy Consumption and total cost

- Network Challenges;
  - Implementation of Large Networks(hard to set-up thousands of nodes)
  - Lack of global standard
  - Fault detection and avoiding network failure design for Internet of Things

In this particular project, a specific Operating System design for Internet of Things devices is proposed as a solution to overcome many of these challenges and problems. Specifications and an overview of the proposed operating system are mentioned in the following chapter.

1.4 A specific Operating System for Internet of Thing Devices

After identifying problems and challenges about embedded devices on Internet of Things networks, a specific operating system for IOT devices had been decided to be designed. Since low power and low cost microcontroller has become most suitable architecture for embedded IOT devices, proposed architecture specifically will be developed in this kind of platform. More information about platform and microcontroller, which has been used during this project, is detailed later on this paper.

Since our focus is on memory constrained embedded devices, firstly the kernel has to be minimized to have a successful implementation. In order to achieve minimization, the dispatcher and the scheduler should be minimized. In addition, the operating system services must be reduced to save memory space. Most of the operating system have large footprints because of the number of services implemented on operating system. However specific applications for sensor networks and Internet of Things do not require complicated operating systems. Secondly, it should have priority driven scheduler to make them real time compatible. Real-time tasks could have higher priority to not miss the deadlines. In addition, low power consumption should be taken in consideration. Different power modes must be introduced. Moreover, idle function must be designed carefully to achieve this purpose.

However doing optimization in classical design in embedded operating systems is not enough to overcome all challenges and problems earlier specified in this paper. A new perspective is needed to achieve to develop such an operating system. A new kernel architecture specific for IOT devices could help us to achieve easy integration and the user-friendly operating system. In this particular project a new two dimensional kernel architecture is proposed. This architecture is also planned to be a light weight and implemented with priority-driven scheduler. A new operating system service called IOT service is proposed. The aim of the service is to allow resource sharing and easy integration with other devices without using any other library. This service is planned to be out on top of every layer even application layer in the operating system architecture. Therefore, users would need only develop an application for specific node. Since IOT service would handle all integration and sharing, application development would be like developing a basic firmware. Details about design and development of particular embedded

operating system is mentioned another chapter in this thesis.

In addition to a new kernel architecture and IOT service, device security must be achieve in operating system level. Security should not be thought of as an add-on to a device, but rather as integral to the devices reliable functioning. Software security controls need to be introduced at the operationg system level, take advantage of hardware security capabilities. Most of the embedded device use old operating systems such as TinyOS, FreeRTOS etc. However, none of them was developed specific for IOT. A detailed comparison of embedded operating systems in investigated later on this paper. Feature of proposed operating system can be seen at list below.

- Microscopic(light-weight) Kernel Architecture
  - Reduced OS services
  - Small footprint and Efficient Task execution
- Two dimensional Operating System
  - User friendly
  - Gives two different perspective and different system layers
- A specific Operating System service for Internet of Things application
  - Enables resource sharing and distribute the workload of the network
  - Easy integration
  - Specifically developed for IOT devices
- Real-Time compatible
  - Priority-driven scheduler
- Low energy consumption
  - Different power modes for different applications
- Security in Operating System Level
  - A new system layer for security

  - Separated Communication functionally from application layer

The main contributions of this thesis are:

- A new kernel architecture for embedded operating system is proposed.
- Detailed Background Research about Operating Systems, Distributed Systems and Wireless Communication is done.
- Hardware Libraries, Kernel Functions and OS services are developed from scratch.
- A new radio protocol is and implemented.
- A new OS service called IOT service which acts as middleware and enables resource and information sharing across the network, is introduced and developed.
- Using this embedded operating system, a cluster is formed and all the system functionalities are tested in real world.

The remaining Chapters of this thesis are organized as follows. Chapter 2 provides a literature and technologies review related with the operating and distributed systems, wireless communication. In addition to that, it contains related work about the project topic. Chapter 3 details and features about proposed operating system and kernel architecture. Chapter 4 describes the implementation of the hardware abstraction layer. Additionally, it gives information about hardware drivers, which are used by HAL. Chapter 5 focuses on the kernel architecture and functionality. In addition to that, in chapter 6 information about other small system components are given. In chapter 7 design and structure of a small radio library, stack and protocol which is developed and implemented specifically for this operating system is explained. Our novel operating system service called IOT service and network structure lies behind it are explained in detail in Chapter 8. The results are shown at Chapter 9 and lastly, the ideas discussed along this thesis work are concluded in Chapter 10. Additionally, it provides the future work and lessons learned during the development of this work. References and materials used during this project can be found at chapter 11.

Chapter 2

Background Research

To develop such an embedded operating system, which is mentioned in earlier section, many disciplines in computer science and electronics, must be investigated and researched. In this section, related background research and topics are detailed. Firstly brief explanation about embedded systems is given. Afterwards operating systems and real time system concepts are explained in detail. In addition, since resource and information sharing is one of the features of the particular operating systems distributed system topic is investigated. Moreover, to create reliable, secure and fast communication between devices, fundamentals of wireless communication is mentioned from the embedded software and electronics point of view. At the last part of the chapter, research has been done about related work and comparison of the embedded operating system which are currently in the market, can be found.

"Last few decades have seen the rise of computers to a position of prevalence in human affairs. It has made its mark in every field ranging personal home affairs, business, process automation in industries, communications, entertainment, defence etc..." [9]. Embedded systems can be defined as a computer system which is designed and optimized to execute a dedicated task and where the computer itself is not necessarily the purpose of the system. Any device that contains a programmable computer which introduce some degree of intelligence to the system, but which itself is not a general purpose computer (such as PCs, workstations, laptops, tablets). An embedded system is generally a system within a larger system. Modern cars and trucks contain many embedded systems. One embedded system controls anti-lock brakes, another monitors and controls vehicle's emission and a third displays information on the dashboard. Even

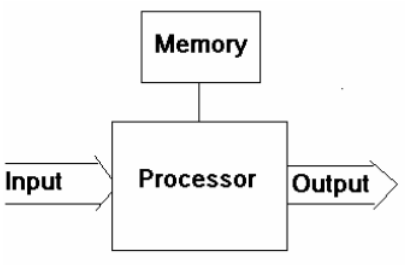

Figure 2.1: A general structure of Embedded Systems [9].

the general-purpose personal computer itself is made up of numerous embedded systems. Keyboard, mouse, video card, modem, hard drive, floppy drive and sound card are each an embedded system. For almost twenty years we as human being are surrounded by specific embedded systems like smartwatches, cars, temperature sensors. Right now, even coffee machines, washing machines and other assets in our daily life, become smart embedded system with the help of Internet of Things concept.

2.1 Operating Systems and Real-Time Systems

Every embedded platforms contain a processor and software to run the processor. Mostly there must be a place to store the executable code and temporary storage for run-time data to run the software. These take the form of ROM and RAM respectively. If memory requirement is small, it may be contained in the same chip as the processor. However, there are also external memory chips which one or both types of memory can be placed. In addition to memory also every embedded system must have some kind of input and outputs to collect data from real world and manipulate it accordingly [9]. For instance, in a washing machine the inputs are the buttons on the front panel and temperature sensor or probe in water tank and the outputs are the human readable display and rpm. Generally, inputs to the embedded system are in the form of sensors and probes, communication signals, or control knobs and buttons. Outputs are generally displays, different kind of signals and changes to the real world. General structure of an embedded system can be seen at the figure 2.1.

1. Processor Management: An operating system must ensure that every single process and task receives same amount of execution time from processor while using maximum pro-

cessor cycles to not miss hard deadlines. In addition, it is also responsible for switching processes/task in multi-tasking environment [9].

2. Memory and Storage Management: Every task needs amount of memory to perform specific tasks properly. An operating system also allocates enough memory for task/processes and uses different types of memory efficiently.

3. Device Management: All the hardware on the system is managed by the operating system. The hardware drivers form a way for the user applications to use of hardware functionality without the need for details about hardware operation [9]. "The driver's function is to be the translator between the electrical signals of the hardware subsystems and the high-level programming languages of the operating system and application programs "[9]. Drivers are abstracted from the operating system kernel for upgradability of devices.

4. Providing Common Application Interface: In order to achieve easy upgradable operating system and security, abstraction layers is used in operating system. Hard drivers is the first abstraction layer in the operating system and application program interface is second and most important one. With the help of APIs, programmers for particular operating system can use the functions/services of operating system without considering CPUs other operation. Usually operating system deals with the details about particular service/functions when programmer/developer uses an API functions [10].

5. Providing Common User Interface: Usually in order to achieve interaction between user and system there is a user interface. Although most of the large operating system like Macintosh or Microsoft have graphical user interface, in embedded operating system mostly no need to have such a GUI since embedded devices usually perform specific tasks and no need to have user interaction.

In addition, an operating system is a compound structure of software. Therefore architecture of the operating system can be built in many different ways. Some architecture about operating systems design have been currently used in popular embedded operating system in market, are mentioned the lines below.

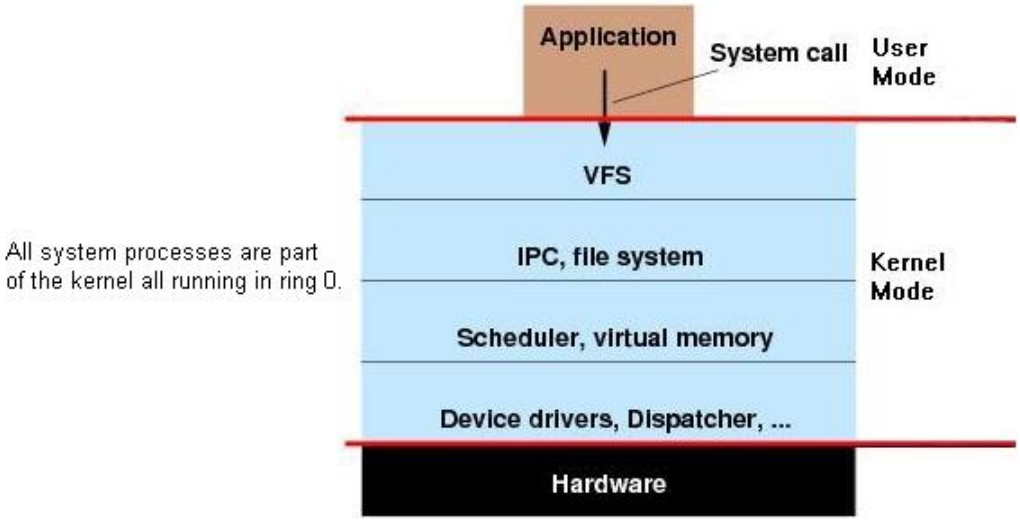

Figure 2.2: Monolithic System Architecture [12].

1. Monolithic systems are the first and one of the oldest architecture. However, it is still popular in the market especially very small real time applications for embedded systems. Because it is very simple and easy to implement and it has very low processor and memory overhead. [11]

   In this architecture, all of the system runs in privileged mode. “The only internal structure is usually induced by the way operating system services are invoked” [11]. In application layer, which is run in user mode, can request operating system services using a special instruction, mostly it is called system call instruction [11]. This “superior” call provide privileged mode and transfers control to system call dispatcher of the operating system. System call dispatcher controls and decides which service must be carried out and transfers control to suitable procedure.

   Handling of interrupts are done directly in the kernel (at least most of the part) and handlers are not fully authorized processes or functions. Therefore, overhead which can be caused by interrupt handling is very small since there is no full task switching at interrupt arrival. However, interrupt handler (according to its priority) cannot run most of the operating system services. In addition, scheduler of operating system is not running while interrupt handler is running or in progress and also priority according the hardware inter-

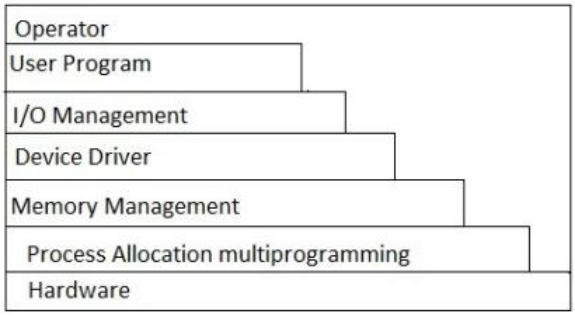

Figure 2.3: Layered System Architecture [13].

rupt requests is in effect, therefore handler functions or task is run at a priority higher than the priority of all other task and it may have higher priority than some system services. At the figure 2.2, you can see the structure of Monolithic Systems.

2. Layered systems. "A refinement and generalization of the monolithic system design consists of organizing the operating system as a hierarchy of layers at system design time." [11] Every single layer is built on top of the operating system services given by the one layer below, hence mostly richer and well-described set of services to the layer above it. Memory and processor overheads are similar to monolithic systems, since interrupt handling and system interface implemented similarly to it.

   Since structural design and modularity is the key part of this architecture, maintenance is easier. Operating system code is easy to read and understand. Moreover, since structural design can be changed without changing other layers, interlayer interfaces does not need to be changed. Structural form of layered system can be seen at the figure 2.3 .

3. Microkernel systems. This design is based on to have smallest kernel as possible in the operating system. Therefore most of the operating system services, they run in user mode and only a few amount of core service can be run in privileged mode. There is a messaging system between kernel and other services and tasks. All the processes and services in user mode send a request message for core services to suitable operating system server and wait for a reply. [11]

   The key concept drives microkernel architecture to handle all the communication between applications and server is security. This kind of architecture enforce users to have

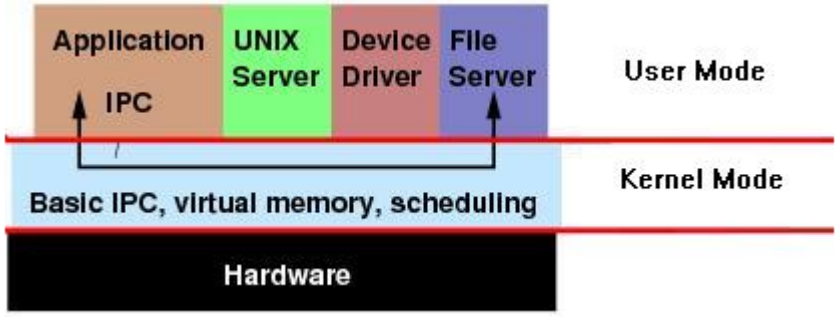

Figure 2.4: Microkernel Architecture Structure [12]

proper security policy on such communication since it would be difficult or inefficient to perform some critical operating system functions in user-mode processes.

Moreover, also it help to make operating system easier to manage and maintain. In addition, using that kind of message passing system between processes and operating system components makes system modular and it leads to have clear understanding on operating system components.

There are some ways to design embedded and real-time software applications. One of the convenient and easy to understand way to design it is organizing them like sequential set of processes which cooperates with each other [11]. Processes are the key concept in every operating system." A process is basically a program in execution. Associated with each process is its address space, a list of memory locations from 0 to some maximum, which the process can read and write" [10]. In addition to that, every process usually associated with set of resources, including registers and all information needed to execute a program.

Moreover, usually operating systems supports to control multiple threads within the same process, sharing the same address space. In the implementation of threads, user mode can used for most of the part without the kernel intervention. Also the address space does not change its contents and must not be switched to achieve very fast switching between threads with respect to switching between processes. Besides, as it was mentioned before all the threads within a process share the same address space, therefore there is limited protection among threads. Thus for instance, a thread can block or disable or may cause error by mistake another thread registers or data and for operating system, it is impossible to detect this kind of errors [14].

Thereupon, most of the embedded and small operating systems only implement threads

which can keep overheads and hardware requirements to a minimum, while operating system for complex applications supports single and multiple process model to enhance the reliability.

Another important part of embedded and real-time operating systems is the scheduler. Task of scheduler is to decide which runnable threads or task available in processor and for how long. Schedulers can schedule threads and task beforehand that is called static schedulers and they can perform scheduling during runtime which is called dynamic schedulers. There are several different algorithms for dynamic schedulers however, since a priority driven scheduler is implemented in particular project, dynamic scheduler algorithms are not mentioned in this paper.

However sometimes different task and processes may need to work synchronize to each other or use to same resources. To achieve this task an essential function of multi-programmed operating system known as Inter-Process Communication must be implemented. Many inter-process synchronization and communication mechanisms have been proposed and were objects of extensive theoretical study in the scientific literature. Since semaphore concept has been used in this project, only this concept is detailed.

"A semaphore, first introduced by Dijkstra in 1965, is a synchronization device with an integer value, and on which the following two primitive, atomic operations are defined"[11]:

- The P operation, often called DOWN or WAIT, checks if the current value of the semaphore is greater than zero. If so, it decrements the value and returns to the caller; otherwise, the invoking process goes into the blocked state until another process performs a V on the same semaphore.

- The V operation, also called UP, POST, or SIGNAL, checks whether there is any process currently blocked on the semaphore. In this case, it wakes exactly one of them up, allowing it to complete its P; otherwise, it increments the value of the semaphore. The V operation never blocks.

The brilliance of semaphores is simplicity in implementation. In addition, this concept is a very low level IPC mechanism. Therefore it helps to have a low processor and memory overhead especially on uniprocessor systems. On the other hand, its low level feature causes difficult usage in complex application. Especially they can easily cause to mutual exclusion which leads

the problem of priority inversion. If a higher priority process or task is stopped or waited by lower priority process or task for shared resources, this leads undesired behaviour in scheduler and called priority inversion. In order to avoid from this problem, most of embedded and real-time operating system uses different kind of algorithms like priority inheritance, priority-ceiling protocols, etc.

2.2 Distributed Systems

As it was mentioned before, purposed operating system enables resource and information sharing between embedded devices. In order to achieve this task, distributed systems concept must be investigated and implemented. A distributed systems is a software and hardware system in which components located on some kind of network and communicate and coordinate their actions by passing messages. There are three important characteristics of a distributed system :

1. Concurrency: Tasks or workload on the network can be done at the devices separately. They can shared devices over network when they needed. Therefore, system power or scale can be increased easily by just adding more devices to distributed network [15].

2. No global clock: Since all the communication done among devices is made by message, there is no reason for a global clock to synchronize devices. Most of the other networks or environments a global clock needed to synchronize devices [15].

3. Independent failures: Every device in the network (distributed systems) may fail and this does not affect other components of the system. Therefore, distributed system can keep running independent from device failure [15].

These characteristics are fundamental to the understanding of distributed systems. These make a distributed system what it is. In addition, there are several reason to use this concept in this particular project. These main reasons can be classified into six section.

1. Resource sharing: In distributed systems, resources can be shared easily. Not only software resources but also hardware resources or data can be shared among the devices. This is one the main factor to implement distributed system concept in this project.

2. Openness: Making system or network cluster components public to other networks is also important purpose of distributed systems. This leads systems to be more extensible [16].

3. Scalability: An Internet of Things network based system must be scalable. With the help of the concepts, adding more resources to the network becomes less costly and simple [16].

4. Fault-tolerance: Every task, device or network may fail independently of the other system component. Shared resource can be installed on many devices in a distributed system thus, handling with the loss of that particular resource if one device fail, is important. Network failures or loss of a resource in the network can be detected using different algorithms in middleware to make other components of the network more reliable [17].

5. Concurrency: Since there is no global clock and dependencies, each devices connected to system can run concurrently. System can execute different tasks in different devices by using messaging system [16].

6. Transparency: Last reason to implement distributed system concept is make system more transparent to the user. This can be divided into three sub section.

   (a) Location transparency: Local and remote resources can be accessed seamlessly.

   (b) Failure transparency: Masking of failures can be made.

   (c) Replication transparency: Allows duplicate resources in multiple components invisibly. [16]

As it was mentioned earlier in this section, in distributed systems, components are presented on different platforms and several components can cooperate with one another over the network. In literature, there are several different architectures in distributed systems. In this paper client-server model, multi-tier architecture and implementation of broker architecture CORBA and the Service-Oriented Architecture (SOA) have been detailed.

1. Client-Server Model: This model or architecture is the most commonly used distributed system architecture. This architecture classify devices into two sub-system. These are client and server. In this model, server device have some resources or services which are

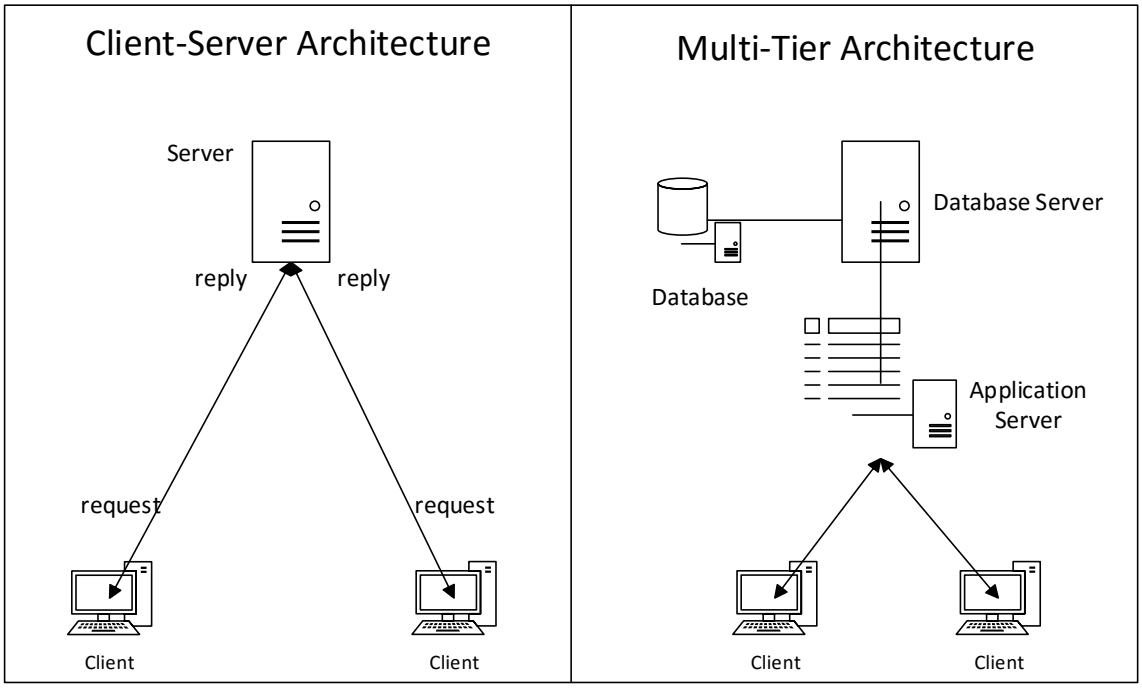

Figure 2.5: Difference between Client-Server and Multi-Tier Architecture

ready to use by client [18]. Client sends request to the server to use these services provided by server. Therefore, client needs to know information about servers to communicate. On the other hand, server just provides requested service and does not need to know information about client. This leads easy integration and scalability in distributed architecture.

2. Multi-Tier Architecture: This architecture is a type of client-server architecture. Difference between two architecture is the separation of different functionalities in the system. For instance application processing, data management etc. are physically separated. Therefore, it provides developers to easy integration and option of changing specific layer without changing the entire application. It makes distributed systems more flexible and reusable. Differences and similarities between these two architecture in structural level can be seen at the figure 2.5.

3. Broker Architectural Style: This architecture style is for middleware systems used in distributed systems. It provides reliable communication between servers and clients. In

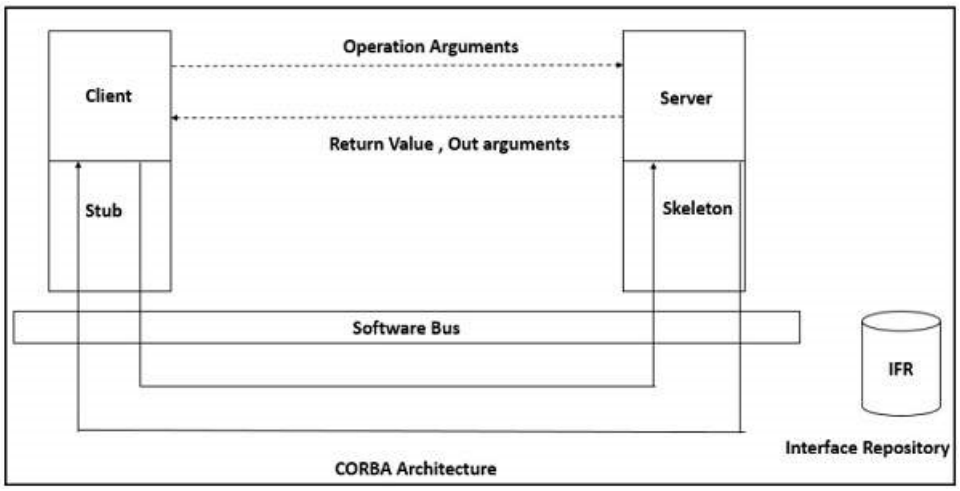

Figure 2.6: Implementation of CORBA

this architecture, a communication object called object request broker does communication. In this section, one of the implementation of broker architectural style called CORBA (Common Object Request Broker Architecture) has been investigated. CORBA can be investigated in four main components. These are broker, stub, skeleton and bridge.

Broker coordinates communication between devices and handles the results and exceptions. “This can be either an invocation-oriented service, a document or message - oriented broker to which clients send a message [18]” Moreover it organizes the service request, locates suitable servers and handles with requests and responses. Also it provides APIs for clients to use the services and send and receive messages. In order to use as a proxy for the client, there is a component called stubs, which is generated at the static compilation time and deployed to the client. This client-side proxy components acts like an arbiter between the client and the broker. Moreover it makes a remote object appears like a local object in client [18].

Since there is a proxy in client side, there must be an another proxy in the server side. This proxy in server side which is created by the service interface called skeleton. It is responsible for encapsulating low-level networking functions and providing high-level APIs to negotiate between the server and the broker. Moreover it is responsible for handling with communication and calling available service [18]. In addition to that, in order to work with different brokers and different communication protocols such as DCOM, Java

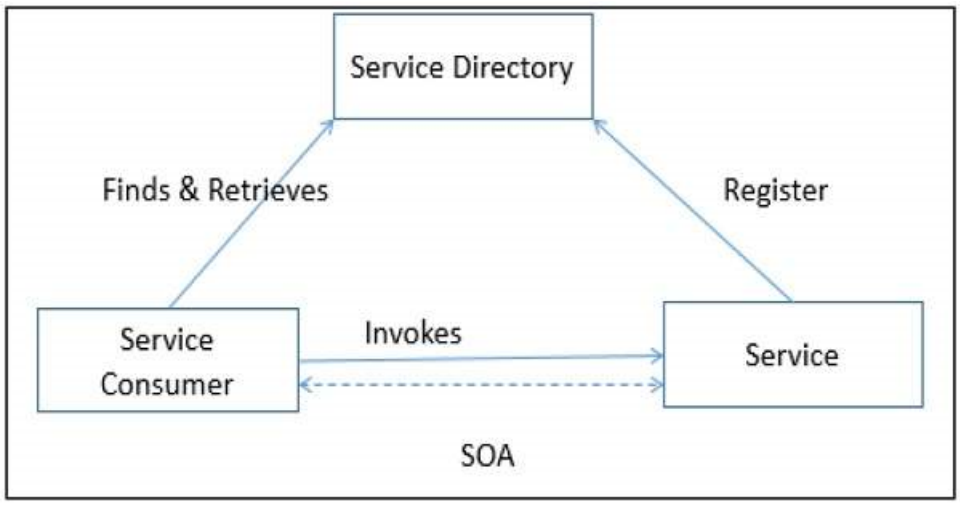

Figure 2.7: Structure of SOA

CORBA brokers, there must a translator component. In CORBA architecture, it is called bridge. These are optional components, which encapsulates the implementation details when two brokers operate together and handles with communication and translates one to another [18]. CORBA is an international standard and implementation of it in structural level can be seen at the figure 2.6.

4. Service-Oriented Architecture: In this architecture, each service is a component of well-defined, independent business functionality which can be used via a standard programming interface. Connections among the services are provided by common and universal message-oriented protocol. It is responsible for delivering requests and responses between services. This architecture is based on client-server architecture. However, it supports business-driven IT approach. That means an application consists of the software services and its consumers [18]. Structure of this architecture can be seen at the figure 2.7.

2.3 Wireless Communication

In order to develop reliable operating system for Internet of Things devices, wireless communication must be investigated in detail. In this section, an overview has been given about wireless communication and protocols. To better explanation and understanding, a conceptual model

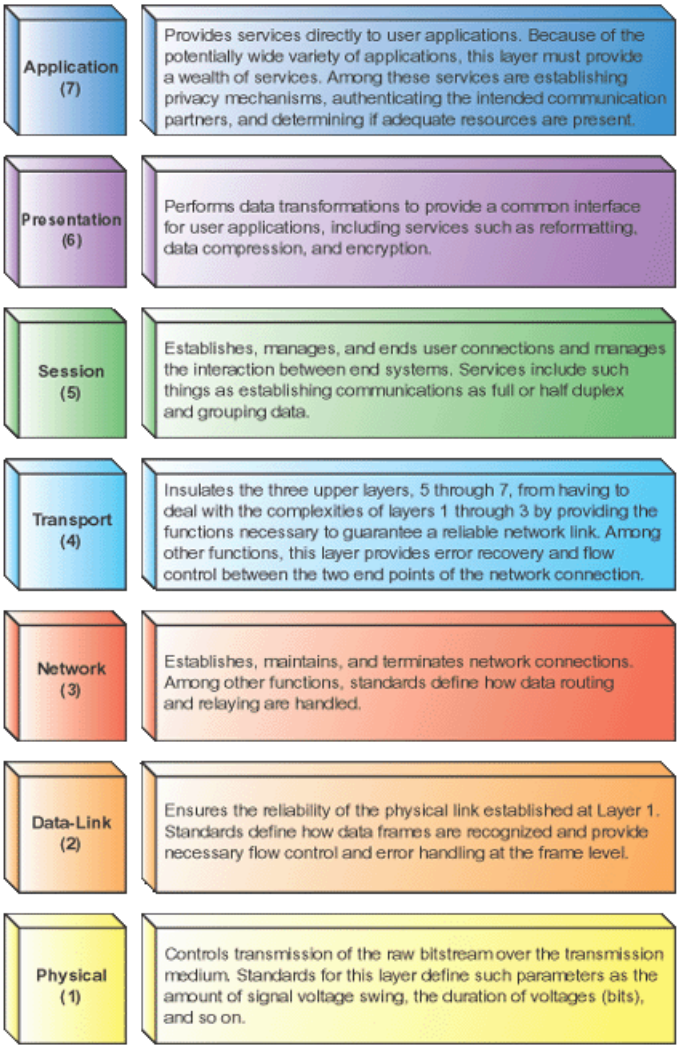

Figure 2.8: OSI model [19]

was created called Open Systems Interconnection model (OSI). This is a conceptual model that characterizes and standardise the communication functions without regard to their underlying internal structure and technology. OSI model and explanation of each layer can be seen at the figure 2.8.

To understand how wireless communication works, firstly physical layer and theory lies behind it must be explained. The physical layer translates transmission and reception of wireless waveforms and processes it. “It is a commonly acknowledged truth that the properties of the transmission channel and the physical-layer shape significant parts of the protocol stack” [20]. Wireless communication fundamentals can be divided into three sub-section. These are fre-

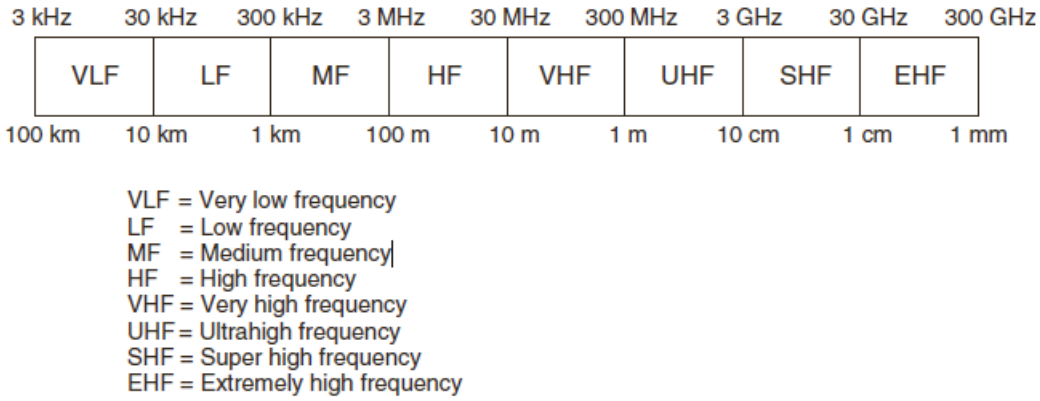

Figure 2.9: Frequency Ranges [20]

quency allocation, modulation and demodulation and spread spectrum.

1. Frequency allocation: Every radio frequency based system has carrier frequency which determines the propagation characteristics – for example, how well are obstacles like walls penetrated – and the available capacity. A single frequency does not support any capacity. Therefore frequency band which is a finite portion of the electromagnetic spectrum, is used for communication purposes. In RF communication these frequencies starts from Very Low Frequency(VLF) ends at Extremely High Frequency(EHF). Exact values of this ranges can be seen at the figure 2.9 [20].

2. Modulation and Demodulation: In order to transmit data wirelessly, a digital signal must be converted to a radio-, optical- or sonic signal that contains the same information as the original signal. In digital electronics when systems communicate, they transfer digital data to each other which are the sequences of symbols and each symbol comes from the channel alphabet are mapped to one a finite number of waveforms of the same finite length. This process is called modulation. On the other hand, modulated signal must be translated to meaningful data after it receives the signal. Demodulation is extracting the original information-bearing signal from a modulated carrier wave. A demodulator is an electronic circuit (or computer program in a software-defined radio) that is used to recover the information content from the modulated carrier wave

   There several types of modulation and demodulators. The signal output from a demodulator may represent sound (an analog audio signal), images (an analog video signal) or

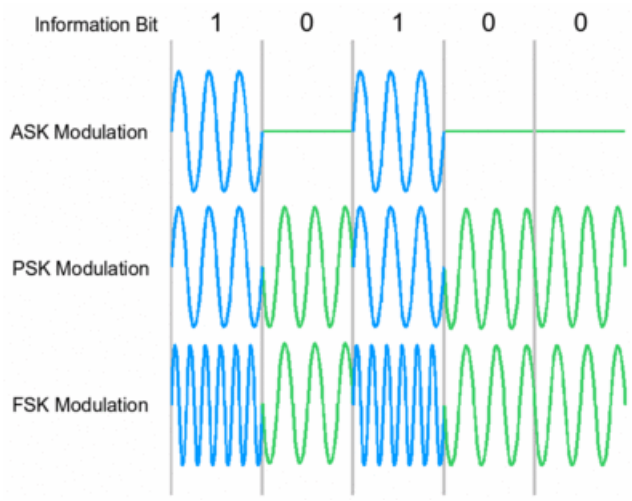

Figure 2.10: Types of Digital Modulation [21]

binary data (a digital signal).These terms are usually used in connection with radio receivers, but many other systems use different kinds of demodulators. For instance, in a modem, which is a contraction of the terms modulator/demodulator, a demodulator is used to extract a serial digital data stream from a carrier signal, which is used to carry it through a telephone line, coaxial cable, or optical fiber.

In digital modulation, which is used in that particular project, there are three types. These are Amplitude Shift Keying(ASK), Phase Shift Keying(PSK), Frequency Shift Keying (FSK). The differences among them can be seen at the following figure 2.10.

3. Spread-Spectrum Communication: This is a dominating modulation and coding technique in wireless communication and networks. In this technique , the transmitted waveforms are way larger than needed to transmit the given data thus it occupies all the bandwidth. The sender spreads the energy in the signal (baseband) over a wide frequency band, for instance the bandwidth of the original signal is much narrower than that of the transmitted signal. The receiver concentrates the energy in the broadband signal back to the original baseband. Sender and receiver employ a special high frequency code sequence (key) to spread and concentrate the signal respectively. The code sequence is denoted as pseudo-random-noise (PRN) and is only known by the sender and receiver. For others, it will be experienced as white noise. The receiver restores the original signal by synchronizing/correlating the incoming signal with the PRN sequence. In addition,

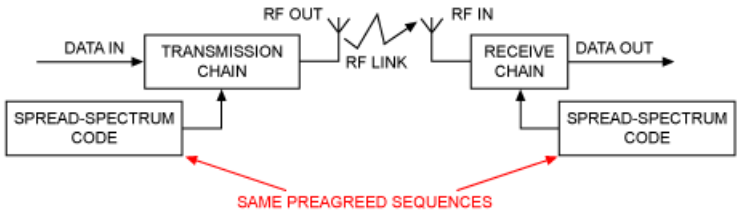

Figure 2.11: Spread Spectrum Communication [22]

the effects of narrowband noise and interference are reduced by using a wideband signal. Therefore, robustness of the system is increased and avoiding from multipath effect can be achieved, though receiver for that technique is more complex than conventional receivers [20]. It is illustrated at the figure 2.11.

There are two popular techniques in spread-spectrum communication

- Direct Sequence Spread Spectrum (DSSS)
  - PRN code modulates the data signal directly (XOR)
  - Phase shift modulation of baseband signal (BPSK, QPSK)
- Frequency Hopping Spread Spectrum (FHSS)
  - PRN code basis for random hopping through a set of valid transmitter frequencies
  - Frequency modulation of baseband signal (FSK)

So far, in this section theory behind the wireless communication has been detailed from transceiver side to the receiver side. However, many devices are capable of communicating wirelessly and without some regulations that would not be reliable communication type. Therefore, some radio band (portions of radio spectrum) are reserved or defined internationally for the use of radio frequency communication for industrial, scientific and medical applications. These frequencies is called ISM radio bands. Example applications of these bands contain RF process heating, microwave ovens and medical diathermy machines. Since these kind of devices can create electromagnetic interference and disrupt radio communication in same frequencies, these devices are limited to certain band in radio spectrum. Thus, communication devices must

handle and tolerate any interference caused by ISM applications. Moreover, particular restrictions exist with respect to transmission power and duty cycle for equipment that operate in these frequency bands. The license-free radio bands ISM (Industrial, Scientific, Medical) make up the basis for a range of different standards within wireless data communication. Most popular frequencies in this band are 315MHz, 433MHz, 868MHz, 2.4GHz, and 5.8GHz.Radio modems for simple digital wireless point-to-point communication in the ISM bands are readily available and quite easy to use (relatively low data rates). Typical applications: cable replacements, remote control, access control, telemetry.

There exists an almost bewildering choice of connectivity options for electronics engineers and application developers working on products and systems for the Internet of Things (IoT). Depending on the application, factors such as range, data requirements, security and power demands and battery life will dictate the choice of one or some form of combination of technologies. In this section, only the major communication protocols/technologies in embedded systems have been mentioned.

1. Bluetooth: For at least fifteen years in computing and many consumer products in the market, Bluetooth has become quite popular and important short-range communication protocol. As especially for wearable products is the key technology because of its specification. In IOT devices, it is used via a smartphone in most of the application. Moreover, it is expected to increase its popularity and usage in IOT applications with the new Bluetooth low-energy (also known as Bluetooth Smart) protocol. The new protocol offers similar range to Bluetooth and significantly reduced power consumption thus it suits well to embedded devices and applications [23].

    - Standard: Bluetooth 4.2 core specification
    - Frequency: 2.4GHz (ISM)
    - Range: 50-150m (Smart/BLE)
    - Data Rates: 1Mbps (Smart/BLE)

2. Zigbee: Widely used protocol especially in industrial application. It has different profiles for different applications. Particularly ZigBeePRO and ZigBee Remote Control protocols

are based on the IEEE 802.15.4 protocol (its and industrial standard for wireless technology), therefore, it is suitable for industrial applications which need infrequent data exchanges at low data-rates. In addition to that, it offers 100m range that makes it more suitable for home or building applications [23].

- Standard: ZigBee 3.0 based on IEEE802.15.4
- Frequency: 2.4GHz
- Range: 10-100m
- Data Rates: 250kbps

3. Wi-Fi: Wi-Fi is the leading technology in electronics and computer systems after the proliferation of the internet. However, it used to be used in personal computing rather than embedded or electronics application. It requires bigger stack and well-built infrastructure. The technological developments in IC design and electronics, led to more processing power with bigger memory for the same price. Therefore, it has become a reliable and easy solution for embedded applications as well. It offers fast and reliable data transfer and the ability to handle high volume of data [23].

   - Standard: Based on 802.11n (most common usage in homes today)
   - Frequencies: 2.4GHz and 5GHz bands
   - Range: Approximately 50m
   - Data Rates: 600 Mbps maximum, but 150-200Mbps is more typical, depending on channel frequency used and number of antennas (latest 802.11-ac standard should offer 500Mbps to 1Gbps)

4. NFC: Also known as Near Field Communication enables very simple communication between two electronic devices with safe two-way interaction. Especially along with the digital revolution, it has started to find wide area of usage in contactless payment transactions, accessing digital content and connecting electronic devices. Mainly, it has increased the capability and quality of contactless card technology with providing secure communication at a distance that is less than 4cm [23].

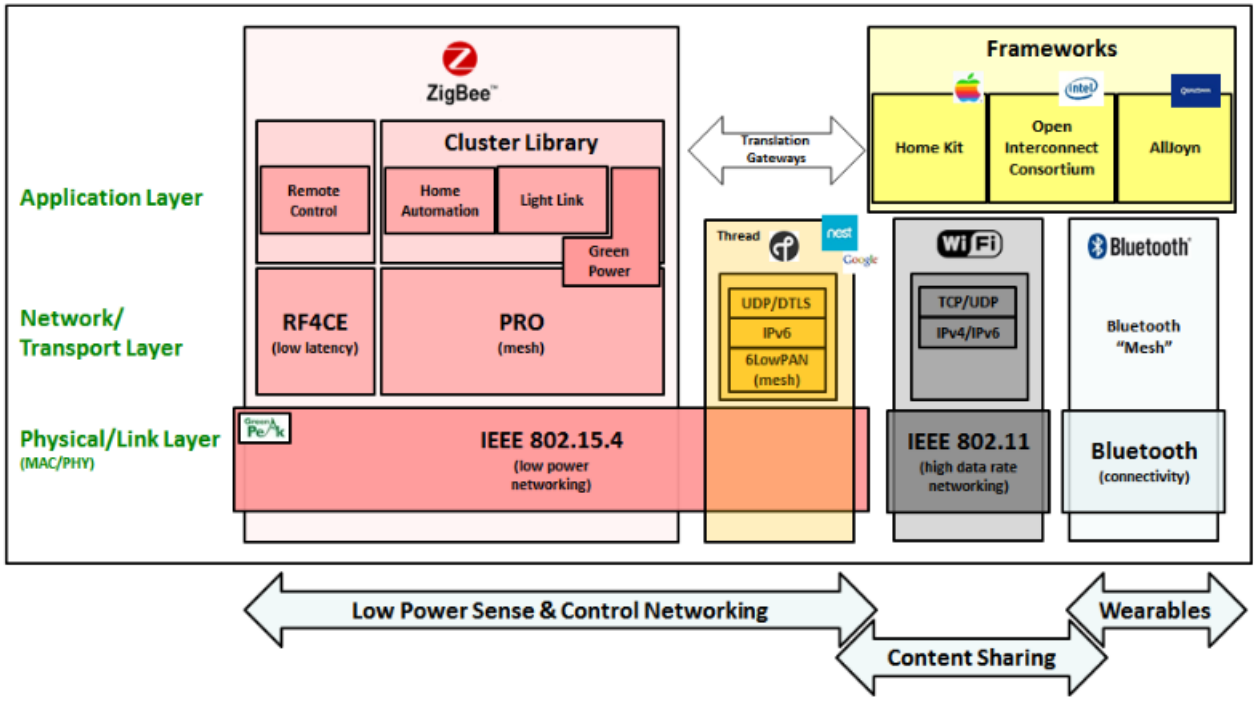

Figure 2.12: Overview of Wireless Technologies in the market [24]

- Standard: ISO/IEC 18000-3
- Frequency: 13.56MHz (ISM)
- Range: 10cm
- Data Rates: 100–420kbps

Overview of most of the short-range wireless technology popular in nowadays can be seen at the figure 2.12.

2.4 Related Work

In order to develop an efficient and novel design in operating system, related works and embedded operating systems which are widely used currently in industry and academy must be investigated in detail. Therefore, specification about particular operating system must be decided according to market needs, current market situation and related work or similar products.

In this section six operating system that are inspired us to develop such an operating system have been mentioned. These are most popular operating system currently both in the industry and academy.

1. FreeRTOS: It is a real time operating system as it can be seen from its name and specifically designed for embedded devices. It is an open source operating system in addition, it provides compiler toolchains and has support for many different architectures. Designing purpose of this operation system is to be “small, simple and easy to use” [25].

   Same as all operating systems, main task of FreeRTOS is to run tasks reliably. Therefore, most of the code is related with scheduling, prioritizing and running user-specific tasks. Since the simplicity of its design, the kernel contains only three or four C files , thus it makes the all system portable and easy to maintain. Although its simplicity, multiple threads or tasks, mutexes, semaphores and software timers are provided for users. Moreover it provides four memory allocation schemes . These are allocate only, allocate and free with a very simple, fast, algorithm, a more complex but fast allocate and free algorithm with memory coalescence and C library allocate and free with some mutual exclusion protection [26].

   Operating systems like Linux or Microsoft Windows, more features like device drivers, advanced memory management, user accounts, and networking can be found. However, the emphasis of FreeRTOS is on compactness and speed execution thus, it can be seen as a thread library rather than operating system although it provides command line interface and POSIX-like I/O abstraction add-ons.

2. Contiki-OS: Contiki-OS firstly introduced to the market in 2003. Like FreeRTOS, it has widely usage in resource-limited systems. It requires 10kb of Ram and 30kb of Rom to function and it defines itself as an ultimate OS for network devices. Contiki mainly designed to connect low-power microcontrollers to the Internet and Wide-Area Networks. It provides three popular network mechanism and these are the uIP TCP/IP stack, which provides IPv4 networking, the uIPv6 stack, which provides IPv6 networking, and the Rime stack, which is a set of custom lightweight networking protocols designed specifically for low-power wireless networks. The IPv6 stack was contributed by Cisco and was, at the

time of release, the smallest IPv6 stack to receive the IPv6 Ready certification [27].

Furthermore, to achieve multitasking in embedded systems, programming model of Contiki is based on protothreads. A protothread is a memory-efficient programming abstraction that shares features of both multi-threading and event-driven programming to attain a low memory overhead of each protothread [28].

3. Micro C/OS: "It is a public domain real-time operating system which is representative of current commercial RTOSes which employ static priority based scheduling algorithms" [29]. It supports 64 different priority level for user-defined functions. Main purposes of the design are determinism, modularity and scalability. MicroC/OS allows defining several functions in C, each of which can execute as an independent thread or task. Each task runs at a different priority, and runs as if it owns the CPU. Lower priority tasks can be pre-empted by higher priority tasks at any time. Higher priority tasks use operating system services (such as a delay or event) to allow lower priority tasks to execute. Moreover, it provides standard operating services such as IPC mechanism, task management, memory management and timing. It has two version called pre-emptive and non-preemptive version.

4. Tiny OS: One of the oldest open source operating system is TinyOS. It is the inspiration for all the other operating system after it. It is the first embedded operating system that is designed for low-power embedded devices and wireless sensor networks. It introduce a term called mote which means a microcontroller based node in wireless sensor network and , capable of reading sensory information, processing and exchanging its data with other nodes [30]. TinyOS applications are developed in the programming language nesC, a dialect of the C language optimized for the memory limits of sensor networks. The extra tools of the system are mainly in the form of Java and shell script front-ends. Every program are built as components and those components communicates among them via interfaces provided by operating systems. Moreover, TinyOS provides interfaces and components for common abstractions such as packet communication, routing, sensing, actuation and storage. All the I/O operations that last longer than a few hundred microseconds are asynchronous and have a call-back since the systems is full-non blocking and has only

one call stack. To enable the native compiler to better optimize across call boundaries, TinyOS uses nesC's features to link these call-backs, called events, statically. While being non-blocking enables TinyOS to maintain high concurrency with one stack, it forces programmers to write complex logic by using many small event handlers. To support larger computations, TinyOS provides tasks, which are similar to a Deferred Procedure Call and interrupt handler bottom halves. A TinyOS component can post a task, which the operating system will schedule to run later. Tasks are non-pre-emptive and run in first in, first out order. This simple concurrency model is typically sufficient for I/O based applications, but its difficulty with CPU-heavy applications has led to developing a thread library for the particular system, called TOSThreads.

5. QNX: This operating system is one of the most popular commercial operating system in the market. It is also most traditional Real-Time Operating System. It is based on the micro-kernel architecture and it supports POSIX standards. In this micro-kernel implementation as it is discussed earlier in this paper, only four operating system services are implemented and these are task scheduling, inter task communication, low level network communication and interrupt handling. All the remaining parts of the system (device drivers included) are implemented as user tasks, making the kernel fast, reliable and small [31]. All this kernel services known as servers in QNX. Therefore, it is different from monolithic kernel. The operating system gives user to turn off and on any operating system service without needing any change in the operating system itself. QNX Neutrino has been ported to a number of platforms and now runs on practically any modern CPU that is used in the embedded market. QNX neutrino supports ARM, MIPS, PowerPC, SH4 and the PC architecture.

6. Embedded Linux(Yocto Project) : In recent years, Linux has started to become popular on embedded applications and devices especially consumer gadgets, telecom routers and switches, Internet appliances and automotive applications. "Because of the modular nature of Linux, it is easy to slim down the operating environment by removing utility programs, tools, and other system services that are not needed in an embedded environment " [32]. The Yocto Project has the aim and objective of attempting to improve the lives of de-

	FreeRTOS	Contiki OS	Micro C/OS	TinyOS	QNX	Embedded Linux(Yocto Project)
Footprint	Small(5kb to 10kb)	Relatively Small(10kb to 30kb)	Small(5kb to 24kb)	Relatively Small(20/ 30kb to several hundred	Big size(however it is configurable)	Big Size
Kernel architecture	Microkernel	Event/driven kernel	Microkernel	Monolithic	Microkernel	Linux Kernel(monolithic)
Platforms	More than 30 different architecture	AVR,MSP430, ARM7, PIC32	More than 30 different architecture	MSP430 family, Atmega128 family, and the Intel px27ax	Intel 8088, x86, MIPS, PowerPC, SH-4, ARM, StrongARM, XScale	ARM,32 and 64bit x86,PowerPC, MIPS(all major embedded architectures)

Table 2.1: Evaluation of Embedded Operating Systems

velopers of customised Linux systems supporting the ARM, MIPS, PowerPC and x86/x86 64 architectures. A key part of this is an open source build system, based around the Open-Embedded architecture, which enables developers to create their own Linux distribution specific to their environment. This reference implementation of Open-Embedded is called Poky. As well as building Linux systems, there is also an ability to generate a toolchain for cross compilation and a Software Development Kit (SDK) tailored to their own distribution, also referred to as the Application Developer Toolkit (ADT). The project tries to be software and vendor agnostic.

Detailed comparison and evaluation about these six operating systems is given in the tables 2.1 and 2.2.

	Positive Points	Negative Points	Network or/and IOT feature
Free RTOS	▪ Open Source ▪ Large Support and Community ▪ Optional preemptive Scheduling and other scheduling algorithms ▪ Device Support	▪ Overhead may occur in particular configurations. ▪ The instruction documents are not free.	▪ No specific support for Networking or IOT devices
Contiki OS	▪ Optional preemptive Scheduling ▪ GUI ▪ Downloading code at run time ▪ Light weight	▪ Not too many architectures(especially arm cortex m series) ▪ Relatively Poor documentation	▪ Network Simulator ▪ Sensor/Network Specific OS ▪ GUI
Micro C/OS	▪ Embedded System Tools ▪ Allow stack growth of tasks to be monitored ▪ Preemptive multitasking real-time kernel	▪ Expensive License	▪ Supports networking technology IPV4
TinyOS	▪ Flexibility ▪ Low power ▪ Small footprint	▪ Lack preemptive real-time scheduling ▪ Lack of flexibility ▪ Lack of Virtual Memory	▪ Limited Physical Parallelism and Controller Hierarchy ▪ Specifically developed for wireless sensor applications
QNX	▪ Can run Linux applications with no modifications ▪ Cross-compile advantage ▪ Has a solid, stable API, which is very well documented. ▪ QNX has only one graphical toolkit supplied (Photon)	▪ Expensive Licence ▪ Limited choice of hardware platform	▪ Supports networking technologies include IPv4, IPv6, IPSec, FTP, HTTP, SSH and Telnet. ▪ Has configuration for Resource Sharing
Embedded Linux(Yocto Project)	▪ Linux Kernel ▪ Big community and good support	▪ Big size Linux kernel	▪ Supports networking technology IPV4

Table 2.2: Comparison of Operating Systems Embedded

Chapter 3

Details and Features

After detail research about the operating systems in the market, the features and the details about particular operating systems have been decided. In addition, the needs of industry and past works in academy have been considered. The features about this project are mentioned in earlier sections. However, after consideration of time and workload some changes in features must have been done to achieve the implementation of all the features. Therefore, implementation of different power modes and security layer had been changed as possible future works. The particular operating system provides only one power mode that is designed to be low-power mode. In addition to that, instead of implementing completely new security layer, a new communication protocol for IOT service has been designed and implemented.

In this particular section, you will find more information about details and features. Firstly, designing steps of how to achieve small footprint and efficient task execution are mentioned. In addition, information about reduced-size operating system services and real-time compatible priority-driven scheduler can be found. Later on, our novel design for Kernel architecture is detailed and how it makes operating system more user-friendly is explained. At the end of this chapter, the proposed IOT service and how it enables resource sharing between devices can be found.

3.1 Simple and Light-weight Design

As it was discussed earlier in this paper, technological development in silicon industry does not have same effects on CPU power and memory sizes. Since the beginning of 80s, CPU speed of the microcontroller has been increasing more than memory size. With the help of new powerful inside of embedded systems, developers demand more task from a microcontroller. This leads complicated and big operating system and libraries. Therefore, most of the operating system have started to suffer from lack of memory.

The term of embedded devices can also be defined as memory-constrained or resource-constrained systems. It is because usually at the best case microcontrollers have a few hundred MHz of CPU and few hundreds of ram and flash. Therefore, an embedded operating system must be small footprint in order to work efficiently in embedded devices. In addition to that, there must be enough space in the ram and flash for run-time and user-defined applications.

This project is based on the idea of simplicity and smallest size that can be achieved. Complicated operating systems require lots of memory space. Although memory requirements can be satisfied buying external flash memory or selecting bigger microcontroller, it is very expensive. Since the concept of IOT is spreading around our daily life, this kind of cost could get very high in mass production. However, simple and efficient operating system can be achieved if this operating specifically developed and optimized for IOT concept. In order to develop such an operating system, the needs from an operating system must be known and system components must be defined well.

Firstly, kernel function of the operating system was divided into two sections. These are portable kernel and native kernel libraries. Moreover, portable kernel also can be divided into three section. Firstly, a component called dispatcher in the particular operating systems are written in assembly language. Dispatcher is the part of kernel that is triggered by an interrupt and receives the full register set of the program that was running at the time of interrupt (with the exception of the program counter, which is presumably propagated through a mutually agreed upon 'volatile' register or some such). Thus, the dispatcher must be carefully written to store the current state of register banks as its first operation upon being triggered. In short, the dispatcher itself has no immediate context and thus does not suffer from the same problem.

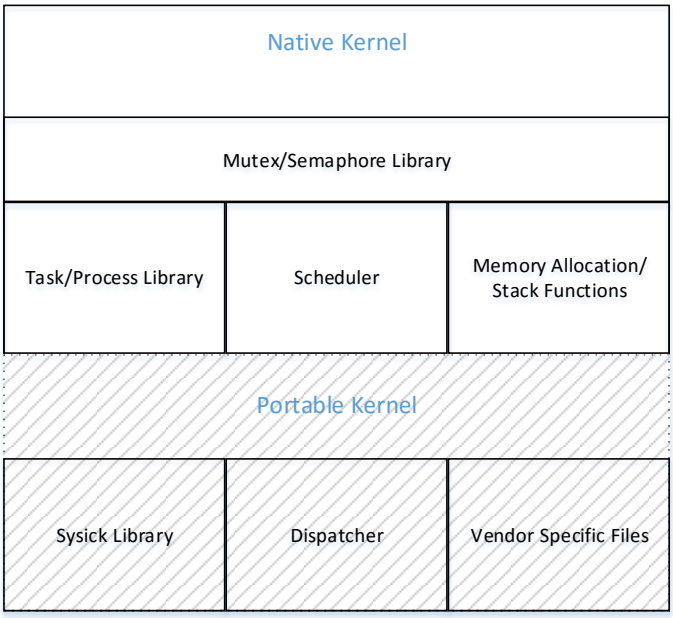

Figure 3.1: Kernel Structure

The second part of portable kernel which can trigger dispatcher when it is necessary, is system tick. A tick is an arbitrary unit for measuring internal system time. There is usually an operating system internal counter for ticks; the current time and date used by various functions of the operating system are derived from that counter. How many milliseconds a tick represents depends on the operating system, and may even vary between installations. Use the OS's mechanisms to convert ticks into seconds. The third part of the portable kernel is interrupt handlers for exceptions such as system-level service and supervisor call. This two function handles the registers during context switch or exception handling. Detailed information about native kernel and portable kernel will be mentioned in section four.

As it was mentioned in the paragraph below, portable kernel has some devices specific definition and functions and also device definition files provided by vendor of particular device(processor or microcontroller). Therefore, since this operating system is designed to support many different processor and architecture, this part was separated from native kernel to make operating system more portable.

First optimization in particular operating system was made in kernel side. All the components of the kernel can be seen at the figure 3.1. Most of the embedded operating system

contains a large kernel which has more capabilities. For example, usually except scheduler and dispatcher, they support operating system event to provide privileges to operating system services. In addition to that, they provide software timer for upper layer of the system. Complicated mutex and semaphore infrastructure is also common component of other operating system. Furthermore, memory protection layer is implanted for the memory management for more complicated CPUs. However, most of the embedded system contains low-power and small size processor. In this particular architecture, as it can be seen, kernel tried to develop as small as possible. Major task of kernel is to achieve reliable execution of task and provide required infrastructure for semaphores and mutexes. APIs and other service of the operating system can handle other extra features and these can be configured from a file. Therefore, even for very basic embedded application, this operating system can be implemented with smaller footprint.

However, optimization in kernel is not enough to reduce to size of the operating system. Generally, operating system services require lots of memory and since most of them are executed periodically, require CPU power as well. An Operating System provides services to both the users and to the programs. Common services provided by operating systems are program execution, I/O operations, file system manipulation, communication, error detection, resource allocation, protection [10]. However, in embedded systems and application most of them are implemented in vain. In embedded applications, user must decide I/O operation, since devices are application and I/O specific. Memory allocation can be done calling kernel functions, there is no need to have a services running for it. Moreover, usually file system is not a necessity in embedded applications due to memory restriction. Therefore, implementing service for that would be a pointless act. In this project, needs of a typical embedded application were investigated in detail. As a result, operating system services reduced to two services. One of them is booting service, which just is executed only once at the beginning to set and configure required hardware. The other one is idle service, which puts operating system to sleep state, if there is no task/process to schedule. User can be controlled all the other operating system services by using APIs for specific part of the system. Moreover, there is IOT service that is implemented on top of everything in the system and it has the higher priority among the task. However, since it is not implemented inside of operating system service component and has different concept than traditional operating system services, it does not count as operating system service in this

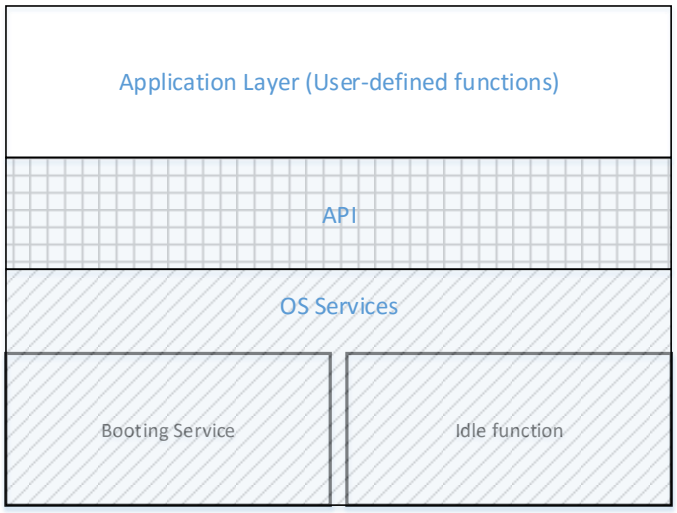

Figure 3.2: Structure of OS Services with API and Application Layer

particular operating system. Details about operating system services and API can be found later on this paper. In addition, structure of operating system services with API and application layer can be seen at the figure 3.2.

3.2 Two Dimensional Kernel Architecture

As it was mentioned before, one of the feature of proposed operating system is a novel and simple architecture. Operating system has been divided into seven components. From bottom to top these are hardware driver, hardware abstraction layer, scheduler and dispatcher, operating system services, application layer, communication layer and IOT service layer. Proposed architecture of the embedded operating system can be seen at the figure 3.3.

There are currently many types of kernel architecture for operating systems in the market and academy. As it was discussed before, two of must popular ones are monolithic and microkernel architecture. However, for instance also hybrid kernel architecture is widely used. Especially systems based on Microsoft Windows has this architecture. They are similar to micro kernels, except they include some additional code in kernel-space to increase the performance.

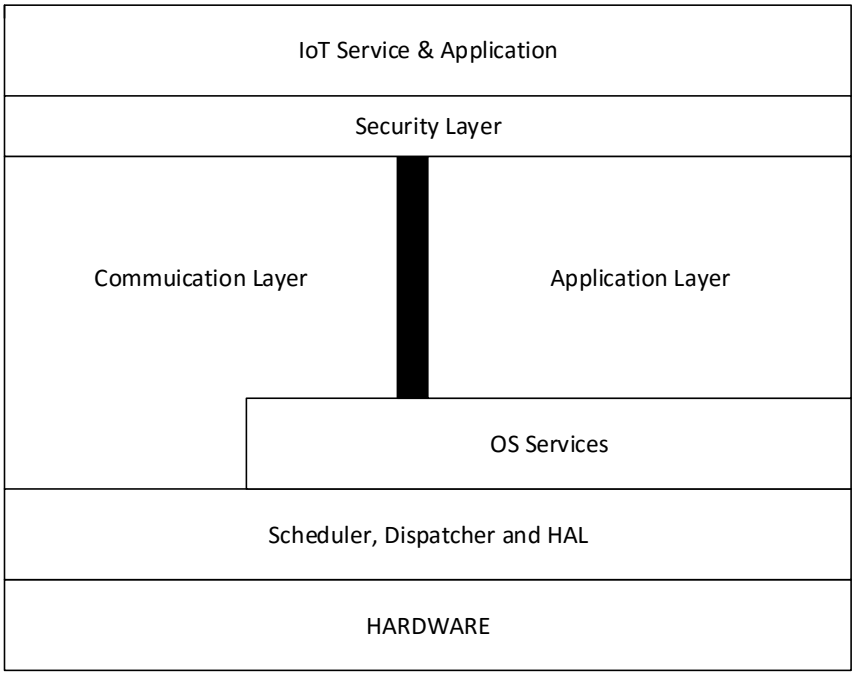

Figure 3.3: Proposed Operating System Architecture

These kernels represent a compromise that was implemented by some developers before it was demonstrated that pure micro kernels could provide high performance. These types of kernels are extensions of micro kernels with some properties of monolithic kernels. In addition, they are similar to micro kernels, except they include some additional code in kernel-space to increase performance. In addition, there is nanokernel, which delegates virtually all services (including even the most basic ones like interrupt controllers or the timer) to device drivers to make the kernel memory requirement even smaller than a traditional microkernel. Moreover exokernels which is very different from other traditional kernels. Exokernels are extremely small. However, they are accompanied by library operating systems providing application developers with the functionalities of a conventional operating system.

As it can be seen, there is no need to have a different kernel architecture unless it is based on a novel idea. In this project, a kernel architecture is proposed as a solution that cannot be solved using other kernels. Firstly, our embedded operating system is based on simplicity and small size. All of those kernel architectures are not specifically developed for small and low power microcontroller. Since the particular operating system is developed specifically for this purpose,

most of the operating system parts can be configurable to reduce to size for different applications. Basic configuration of the operating system provides multitasking, memory allocation and real-time task execution. Since even basic configuration has the same basic capabilities with other kernel, this architecture is called microscopic kernel architecture.

Second purpose to develop new architecture is to achieve user-friendly and secure operating systems. Most of the embedded developers tend to develop basic C functions and libraries for the specific application. Usually using an operating system for this kind of application it is not the first choice of the developer due to complexity of software and necessity of learning new software. Therefore, idea behind this kernel architecture is to create a user space (application layer) for developer. Developers can control hardware like they did before, and they do not need to develop or learn new software to implement wireless networks or deal with other communications in the systems. That is the first dimension of our proposed architecture. User dimension just need to develop their code like there is no operating system. This can be seen from the figure above, our first dimension is start from right hand side until the communication abstraction. In addition, also developers who want to contribute or change some kernel functions according to their need in application, there is another dimension from bottom the top called supervisor dimension in the structure of the operating system. Therefore, this architecture actually called as microscopic two-dimensional kernel. With the help of two dimension, implementation of the operating system is easier. Moreover, development of operating system for future work became easier and more collaborative. Details about kernel operation and functionality can be found later on this paper.

3.3 Internet of Things Service and Resource Sharing

Last feature and purpose of the particular project is to achieve reliable resource and information sharing. Earlier in this paper, six different operating systems for embedded devices have been mentioned. Moreover, during the development of the project, many other operating systems were investigated. Although most of them have some feature or libraries specifically provided for Internet of things, none of them has an OS service for this kind of applications.

Our main idea is to develop an embedded operating system not only for internet of things

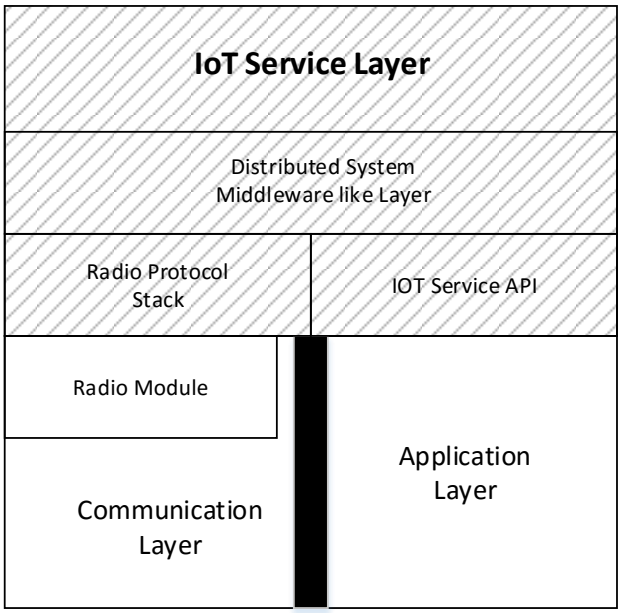

Figure 3.4: Structure of IOT Service

devices but also for all kind of wireless network. Large networks which may be formed by hundreds or thousands of nodes, can be created easily. However, in the application side, this new trend brought some new problem to the networks and software. Since wireless communication not reliable as wired communication, there are always some minor errors such as missing packet and etc. in the wireless network. In addition to that, most of the sensor network are used in critical measurements in harsh environments. Sensor nodes may become unavailable because of lots of different reasons. Sometimes this failure may not create big problems. However sometimes these kind of failures may cause of shutting down whole network. To overcome these problems, different kind of approaches can be used. Most of these approaches requires reliable communication and information sharing. Therefore, in this particular project, a novel operating system called Internet of Things service has been introduced as an infrastructure. Structure of proposed architecture can be seen at the figure 3.4.

Earlier period during this project, Bluetooth protocols was the base of this service. The IOT service was implemented on top Bluetooth smart protocol. However, Bluetooth stack has a large footprint. Since our focus is to develop, a very small size embedded operating system that was not an optimal solution. Bluetooth stack was tried to be reduced but even optimized stack for

specific application was too big. Therefore, a new radio library and a specific radio protocol for wireless sensor networks and Internet of Things decided to develop. Lightweight protocol stack for this particular protocol is implemented into IOT service.

Like every other operating system service, IOT service periodically does particular task to share data between other devices in the network. Actually, this IOT service can be considered as distributed system middleware. While developing this services, distributed system concepts such as CORBA and SOA architecture is used as conceptual base. In addition to that, a basic mail architecture has been added to this service. Therefore, this service can be seen as hybrid of all these three architecture. On the other hand, this service handles all wireless communication and in the meantime, it does not affect the execution of other user applications or tasks. User can interact with the service using IOT service API. Moreover two device types have been introduced for the network structure, thus, user needs to select device type before the development. However as it was mentioned before, user does not need to develop and library to enable resource or information sharing. More information about Internet of Things Services and radio library can be found later on this paper.

3.4 3.4. Development Platform

In this project, a small embedded operating system is developed for low-cost and small (memory-constrained) microcontrollers. Although this project aim to support all this kind of microcontrollers, for development one platform must have been selected. Therefore, a system-on chip from Nordic Semiconductor called nRF51822 was chosen for this particular project.

This integrated circuit is a flexible and powerful system on chip, which contains a microcontroller and radio module ideally suited for Bluetooth applications. The nrf518221 is based on 32-bit ARM cortex M0 CPU with 256kB/128kB flash + 32kB/16kB RAM for improved application performance. In addition, embedded transceiver works at 2.4 GHz and supports both Bluetooth Smart and the Nordic Gazell 2.4 GHz protocol stack [33].

The SOC contains a rich option for analog and digital peripherals that can be executed without interrupting CPU through Programmable Peripheral Interconnect system. “A flexible 31-pin GPIO mapping scheme allows I/O like serial interfaces, PWM and quadrature demodulator to

be mapped to any device pin as dictated by PCB requirements" [33]. Therefore, this leads to have more flexible designs with pin-out location and function.

For CPU architecture, ARM offers large variety of processor architecture for different types of embedded application. As it was mentioned before, in this SOC contains ARM cortex M0 CPU with 16-bit instruction set with 32-bit extension (Thumb-2 technology) which provides a small-memory-footprint in high-density code. In addition to that, the processor uses a single-cycle 32-bit multiplier, a 3-stage pipeline, and a Nested Vector Interrupt Controller (NVIC) to achieve simple and efficient program execution [34]. Furthermore, ARM provides a hardware abstraction layer called The ARM Cortex Microcontroller Software Interface Standard (CMSIS) to achieve further compatibility with other architecture such as ARM Cortex M3 based devices [34].

"The nRF51 series power management system is orthogonal and highly flexible with only simple ON or OFF modes governing a whole device." [35]. When it is set to System OFF mode, sections in the RAM can be retained although everything is powered down. System can be easily set to System ON mode through reset or it can be waken up from all GPIOs. Once system is on, all functional blocks can be used again and switch to the IDLE mode [35].

The two pin Serial Wire Debug interface (provided as a part of the Debug Access Port, DAP) offers a flexible and powerful mechanism for non-intrusive program code debugging. This includes adding breakpoints in the code and performing single stepping [35]. The block diagram of the SOC can be seen at the figure 3.5.

The reasons of the choice of particular processor architecture as follows;

- The smallest ARM processor: Right now in the market, it is the smallest ARM architecture can ever be found. Not only code density and energy efficiency but also forward compatibility for other bigger architecture such as Cortex-M3 and Cortex-M4 processors makes this architecture more suitable for this project.

- Low power: The Cortex-M0 processor, which consumes as little as 12.5µW/MHz (90LP process, minimal configuration) in an area of under 12 K gates is the leader microcontroller in the market now.

- Simplicity: With just 56 instructions, it is possible to master quickly the entire Cortex-

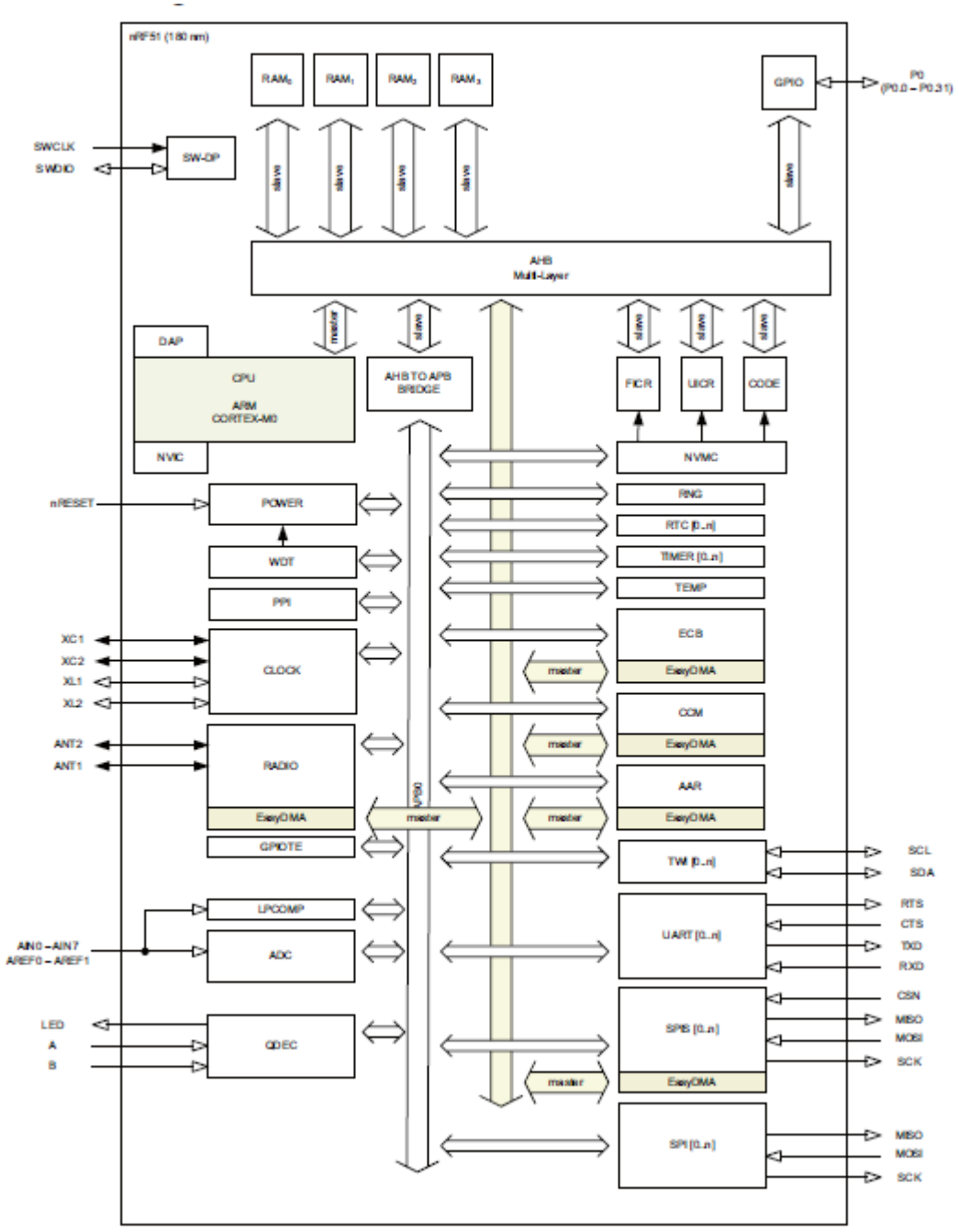

Figure 3.5: Block Diagram of NRF51822 [34]

M0 instruction and its C friendly architecture, making development simple and fast. The option for fully deterministic instruction and interrupt timing makes it easy to calculate response times.

- Optimized connectivity: Most important feature of this architecture is to be optimized for connectivity. It is specifically design to be used in application with wireless protocols such as Bluetooth Low Energy (BLE), IEEE 802.15 and Z-wave [36].

On the other hand, other components of SOC that is radio transceiver is worth to mention as well. Nordic Semiconductor designed and optimized this transceiver specifically to operate in the worldwide ISM frequency band at 2.400 to 2.4835 GHz. Moreover, it its highly configurable transceiever, thus radio modulation modes and packet structures can be configured to support large variety of protocols such as Bluetooth low energy, ANT, Enhanced ShockBurst [34].

The transceiver receives and transmits data directly to and from system memory for flexible and efficient packet data management. The on chip transceiver has the following features:

- General modulation features
  - GFSK modulation
  - Data whitening
  - On-air data rates
    - 250 kbps
    - 1 Mbps
    - 2 Mbps
- Transmitter with programmable output power of +4 dBm to -20 dBm, in 4 dB steps
- Transmitter whisper mode -30 dBm
- RSSI function (1 dB resolution)
- Receiver with integrated channel filters achieving maximum sensitivity
  - -96 dBm at 250 kbps

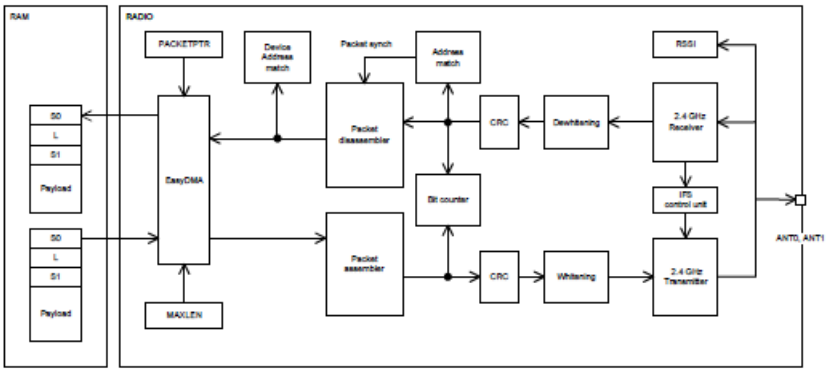

Figure 3.6: Nrf51822 Radio Module Block Diagram [35]

  - -93 dBm at 1 Mbps BLE
  - -90 dBm at 1 Mbps
  - -85 dBm at 2 Mbps

- RF Synthesizer
  - 1 MHz frequency programming resolution
  - 1 MHz non-overlapping channel spacing at 1 Mbps and 250 kbps
  - 2 MHz non-overlapping channel spacing at 2 Mbps
  - Works with low-cost ± 60 ppm 16 MHz crystal oscillators
- Baseband controller
  - EasyDMA RX and TX packet transfer directly to and from RAM
  - Dynamic payload length
  - On-the-fly packet assembly/disassembly and AES CCM payload encryption
  - 8 bit, 16 bit, and 24 bit CRC check (programmable polynomial and initial value) [34]

The block diagram of on chip transceiver can be seen at figure 3.6 .

Chapter 4

Hardware Abstraction Layer and Drivers

In order to use the capabilities of the microcontroller peripherals, a piece of software which is specifically developed for this purpose, what an operating system needs. Driver provides a software interface to hardware devices, enabling operating systems and other programs to access hardware functions without requiring details of the hardware being used.

A driver communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface. As it was mentioned before in this particular project, nRF51822 SOC has been used as target platform. Therefore, all the hardware drivers in this project are written for that microcontroller. Since this SOC contains a microcontroller, also it comes with lots of peripherals to use. Some of them are implemented in this particular project and the others can be developed later on and integrated easily into the operating system.

In this project, two type of development board which contains particular SOC have been used to test and verify the operating system. Thus, firstly board specific libraries was developed. The purpose of these libraries are to create common ground for development between boards. Since there are two type of development board, all the pins for LED and button connection, UART pins for J-link connection are different. All the libraries are implemented on top of board

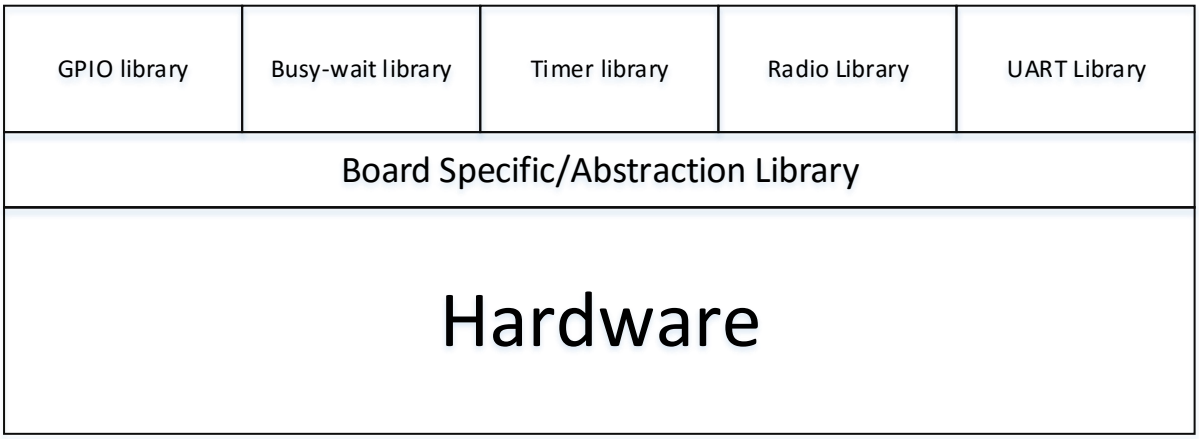

Figure 4.1: Structure of Hardware Drivers

libraries. Therefore, there is no need to develop or use two different driver libraries for two different boards. Structure of hardware drivers can be seen at the figure 4.1 .

One of the most important library in hardware drivers for most of the embedded operating system is general-purpose input-output libraries or widely known as GPIO. This library is used to communicate or interact with other components, actuators or devices. Therefore, even to light a small LED, this library must be used. To reduce the size of the GPIO library, only basic functionality is implemented. Most of the cases, main tasks of this library set particular pins as input or output, set output pin values to logic high or low and configure pull-up and pull-down of pins. Therefore, only three functions are developed and implemented. Firstly, configure function needs three argument. These are pin number, direction and pull-up/down option. User can configure desired pin according to needs. Pin-set function sets desired pin to logic high and pin-clear function sets desired pin to logic low.

Timer functionality of device is a crucial for many applications but especially for real-time applications. In this particular SOC, there are three timers. One of the can be configured up to 32bit and other tow can be configured up to 16bit. Designing of the timer library is based on very simple architecture. Initialize, start and stop functions are implemented according to nature of timers. User can set interval and the address of call-back functions. In computer programming, a call-back is a piece of executable code that is passed as an argument to other code, which is

expected to call back the argument at some convenient time. The invocation may be immediate as in a synchronous call-back, or it might happen at later time as in an asynchronous call-back. In all cases, the intention is to specify a function or subroutine as an entity that is, depending on the language, more or less similar to a variable. In this library call-back functions provides handling with timer interrupt to user without dealing with the timer registers. In addition, also the SOC contains a real time counter. This timer has been used as SYSTICK timer. This library cannot be used for other purposes or changed by user and is implemented into portable kernel.

Busy-waiting, busy-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available. Bust-wait can also be used to generate an arbitrary time delay, a technique that was necessary on systems that lacked a method of waiting a specific length of time. Processor speeds vary from computer to computer, especially as some processors are designed to dynamically adjust speed based on external factors, such as the load on the operating system. As such, spinning as a time delay technique often produces unpredictable or even inconsistent results unless code is implemented to determine how quickly the processor can execute a "do nothing" loop, or the looping code explicitly checks a real-time clock. To create a common wait/delay function for all the applications, bust-wait library is implemented in this level. Basically user can delay execution for desired time. In addition, with the help of two functions, this time can be set in microseconds or milliseconds precision. These functions executes no operation instruction of the processor several time to reach one microsecond and then using a loop, this delay is extended. While busy-wait functions are being executed, the operating system keeps handling with interrupts and other operating system services.

A universal asynchronous receiver/transmitter, abbreviated UART is a computer hardware device that translates data between parallel and serial forms. UARTs are commonly used in conjunction with communication standards such as TIA (formerly EIA) RS-232, RS-422 or RS-485. The universal designation indicates that the data format and transmission speeds are configurable. The electric signalling levels and methods (such as differential signaling etc.) are handled by a driver circuit external to the UART. A UART is usually an individual (or part of an) integrated circuit (IC) used for serial communications over a computer or peripheral device serial port. UARTs are now commonly included in microcontrollers. In this particular project

UART library is implemented for the debugging issues in real time applications. Simple UART library is formed by only three functions. UART -configure function configures communication for desired baud-rate and sets corresponding bits to prepare physical layer of communication correctly. UART -send function needs avoid pointer, and length of the UART library to send. UART -Receive function works and needs same arguments. In order to keep UART library simple, none of other feature or interrupt of UART module were not implemented in this particular project.

Wireless communication particularly most important part of the project. In order to achieve relible communication between devices, library must be developed carefully. Moreover that since a small protocol has been created for this project, library for radio module must be developed accordingly the stack of the protocol. Mainly this library provides some functionality as other communication libraries. Configuration, send and receive function works like the functions in UART library. However, radio library has more features, handles with interrupts, and supports other functionalities. Details about this library is given in the section called “Radio Library, Stack and Protocol” due to make this parts of paper more understandable. Choice of the implement other modules and peripherals inside of the microcontroller has left to user. Since for the test scenarios implementation of particular peripherals was enough, libraries such as I2C, SPI or ADC were not implemented.

Another part of the system which will be explaining rest of the section is hardware abstraction layer. Hardware abstraction layer also known as HAL is formed by routines and functions in the operating system which emulates hardware and platform specific functionalities and provides direct access layer to hardware resources for the application layer.

At the past, the computer or embedded systems did not have any kind of software layer, which works as the hardware abstraction layer. Therefore, developers working with those devices would have to know how each hardware device communicated with the rest of the system. This was very challenging for the software developers since every part of the hardware in the system had to be known to ensure the software’s compatibility. With HAL, rather than the program communicating directly with the hardware device, it communicates to the operating system what the device should do, which then generates a hardware-dependent instruction to the device. For that reason, the software started to be compatible with any device without need

for the information and details about the operation of hardware.

An example of this might be a "Joystick" abstraction. The joystick device, of which there are many physical implementations, is readable/writable through an API which many joystick-like devices might share. Most joystick-devices might report movement directions. Many joystick-devices might have sensitivity-settings that can be configured by an outside application. A Joystick abstraction hides details (e.g., register formats, I2C address) of the hardware so that a programmer using the abstracted API needn't understand the details of the device's physical interface. This also allows code reuse since the same code can process standardized messages from any kind of implementation which supplies the "joystick" abstraction. A "nudge forward" can be from a potentiometer or from a capacitive touch sensor that recognizes "swipe" gestures, as long as they both provide a signal related to "movement".

In this embedded operating system, hardware abstraction layer used as a very thin layer that is responsible from translation of hardware drivers. Upper layers of operating system and hardware drivers can be seen as two different group of people who speak different languages. HAL acts like a translator between two layers and translates driver function into easier functions. Basically HAL is divided into seven basic type of drivers which can be found in microcontrollers. Translation is done using specific structures for all types of device.

```
typedef struct {
        void (*configure)(int,hal_gpio_dir_t,hal_gpio_pull_t);
        void (*set)(uint8_t);
        void (*clear)(uint8_t);
        }hal_gpio_dev_t;
        static hal_gpio_dev_t  dev_gpio = {
        &gpio_pin_configure,
        &gpio_pin_set,
        &gpio_pin_clear
};
```

Listing 4.1: Hal structure for GPIO

As it can be seen from the listing 4.1, GPIO structure has been designed accordingly to driver.

Then device defined as static global structure and drivers function are assigned. With the help of that, device can be used by user easily. Moreover it increases readability of the code.

```
typedef struct {
    void (*open)(void);
    void (*configure)(void *);
    void (*send)(void * ,uint32_t);
    void (*receive)(void * ,uint32_t);
    void (*close)(void);
    void (*change_channel)(uint8_t);
}hal_comm_dev_t;
static hal_comm_dev_t dev_radio = {
    &radio_open,
    &radio_configure,
    &radio_send,
    &radio_receive,
    &radio_close,
    &radio_change_channel
};
static hal_comm_dev_t dev_uart = {
    NULL,
    &uart_configure,
    &uart_send,
    &uart_receive,
    NULL,
    NULL
};
```

Listing 4.2: HAL structure of communication drivers

The listing 4.2, communication structure of HAL library can be seen. The struct type called `hal_comm_dev_t` is defined for all communication modules inside of the SOC. After investiga-

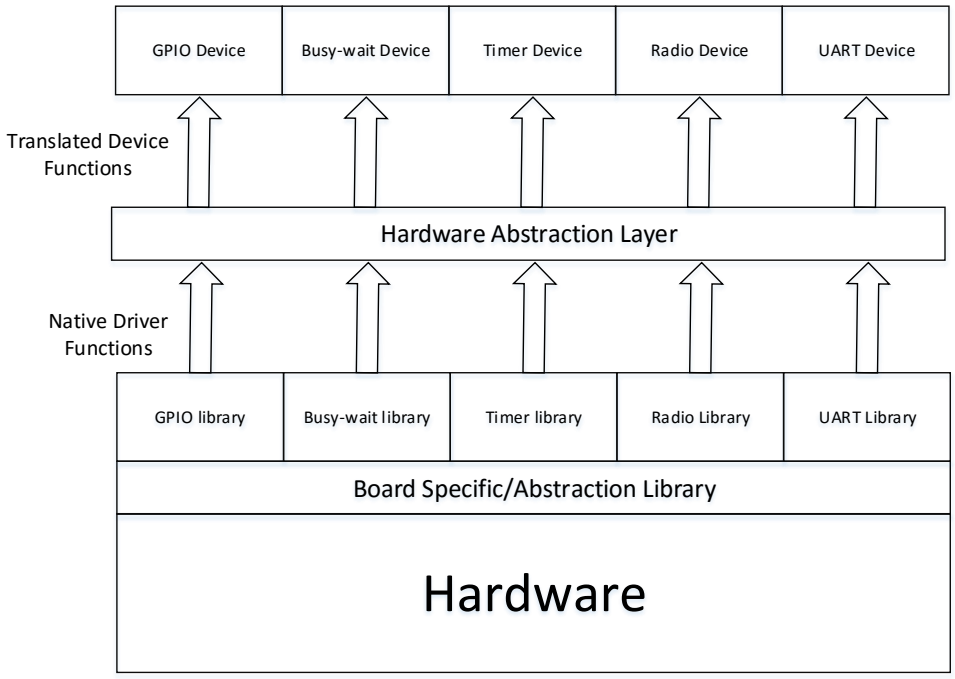

Figure 4.2: Interaction between HAL and Hard Drivers and Structural Design

tion about communication protocols, model about most of the communication type has been decided. According to this model, every communication module either have open, close functionality such as radio module or just uses send and receive functions such as SPI, I2C and UART. Also most of the communication chips allows or needs user to configure and some of them they do not allow or require this. Therefore, as it can be seen the example below, if one of the functions is unrelated or unimplemented for that communication protocols, this can be easily discarded giving NULL value to particular field of struct. On the other hand, new hard drivers of sensors or module must be designed and implemented according to HAL. Once it is developed accordingly to that, it can be easily implemented and used inside of the operating system. The other part of HAL are implemented as same way as communication and GPIO structs. Interaction between HAL and hard drivers and structural design of them can be seen at below.

Chapter 5

Kernel Architecture

In this chapter, designing steps of kernel architecture, theory lies behind it and how kernel operates inside of the particular operating system are explained in detail. Kernel has two main task in this system. To control context switching and thus, achieve multitasking and schedule given task according to their priority.

Kernel is the part of operating systems, which contains the central core of operating system functionality. It has complete control over everything that occurs in the system. Usually it is the first program loaded on start-up, and then manages the remainder of the start-up, as well as input/output requests from software, translating them into data processing instructions for the central processing unit. It is also responsible for managing memory, and for managing and communicating with computing peripherals.

Target platform (cortex-M0) has a number of features targeting at embedded operating system support. These are:

- System Tick Timer: It has a special timer called System Tick (SysTick) timer. It is 24-bit counter that can be used to create needed system tick for some operation in kernel especially to control context switching. Since this is an optional feature, Nordic semiconductor, which developed our target platform, decided not to implement this timer. Therefore, a real time counter is used instead of this special counter.

- Two stack pointers: There are two stack pointer in the processor architecture called The Main Stack Pointer (MSP) and the Process Stack Pointer (PSP). With the help of having two

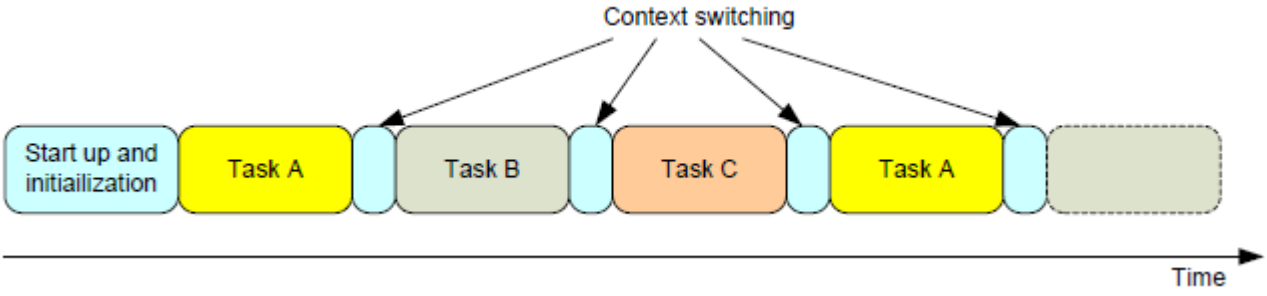

Figure 5.1: Multitasking and Context Switching[37]

stack pointers, OS kernel and application stack can be separated [37].

- A SVCall exception and SVC instruction: This exception or call can be used by applications to access OS services and make development of OS easier [37].
- A PendSV exception: PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

5.1 5.1. Multitasking and Context Switching

Operating systems that are used in embedded applications are design to handle with multiple tasks. They achieve this dividing the processor execution time into numbers of tasks and perform each task in different slot. The length of these time slots differs from one operating system to another and depends on hardware as well. Scheduler of the operating system works at the end of each time slot to change the task or to decide changing of the task. That switching of tasks is called context switching [37]. During the context switching, execution context (state pf a process, task or thread) is stored and restored to resume the execution of switched process later. Therefore, multiple tasks can share a single CPU and this is the essential part of multitasking operating systems. Performing context switch and multitasking in a single processor system can be seen at the figure 5.1.

To achieve context switching in periodic way, the task execution should be interrupted by a hardware like a timer. When the timer asserts an interrupt, an interrupt handler or exception handler must deal with task scheduling. The handler might also carry out other OS maintenance

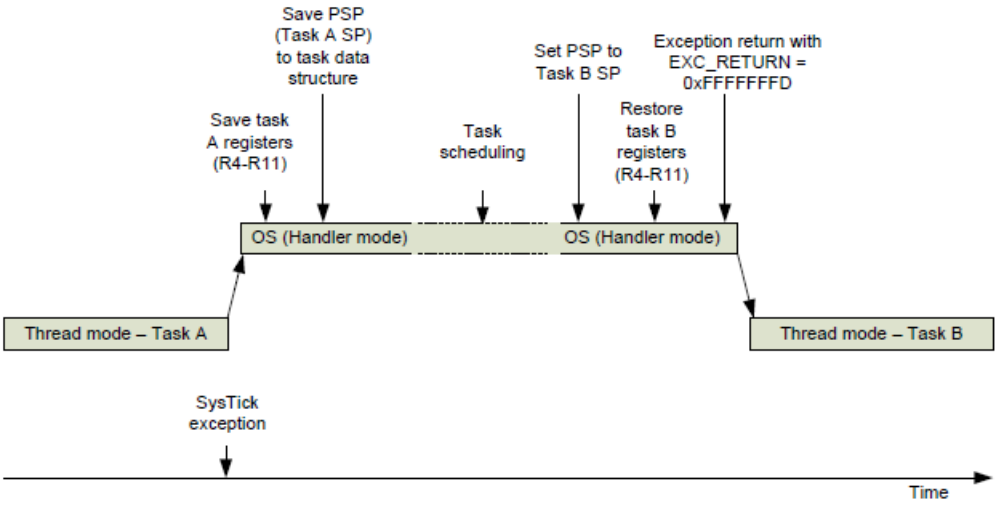

Figure 5.2: Context Switching from one task to another[37]

tasks. In this particular operating system, systick library is developed for that purpose. User can configure the interval of system tick timer from configuration file and default value is 400 Hz. With default value, operating system can change task for four hundred times per second. In order to keep overhead as low as possible, execution time of this handler must be very small. Therefore, this handler only checks need for context switching and set some registers to maintain the whole system. To achieve context switching two stack pointers provided by Cortex-M0 architecture were used. One of them called MSP that can be used at start-up and in exception handlers, including OS operations. Other one is known as PSP that is typically used by application tasks in a multitasking system Both of them are 32-bit registers and but only one of them can be used at once according to the vale in CONTROL special register and current mode. The MSP is the default stack pointer and initialized after the reset by value from the first word of the memory. Only MSP can be used in simple applications. However, multiple stack regions are needed for embedded operating systems and high reliability. With the help of separation, one stack could be used for OS kernel and exceptions and the other one for tasks. In addition, this separation also leads to reduce the chance of stack error. Also using memory protection unit can limit stack usage of task. However, in this project to keep simplicity and follow lightweight design, it is not implemented. The kernel has to keep track of the stack pointer values for each task during context switching and switch back to the PSP value to provide stack for each task. During context switching, the SP for the exiting application task in the PSP will have to be saved

and the PSP will then change to the SP location for the next task. This process is illustrated at the figure 5.2.

```
typedef struct {
        volatile stack_type_t    *p_top_of_stack;
        base_type_t              priority;
        stack_type_t             *p_stack;
        char                     task_name[3];
        task_state_t             state;
        uint8_t                  blocked_by_mutex_id;
        list_node_t              p_node;
}task_control_t;
```

Listing 5.1: Task Control Structure

In this operating system, kernel has native libraries for task creation. Task library is responsible from main task functionalities. In addition to that, stack function library provides memory allocation and initialization functions for each task. The task control structure can be seen at the listing 5.1. When a task is being created, task create function is called. Firstly it initialises the lists for task states. Afterwards it calls one of stack functions called allocate stack and control block to allocate required memory space and stack for particular task if there is enough space. If there is not enough space for the task, it returns an error code and creation of task is cancelled. After successfully allocate the memory for the stack and control block, other fields of control structure has been filled with relevant values. Then if there is no running task currently, current task pointer value assigned as particular task pointer. In order to reduce to size of the code, for allocating memory to control pointer, our own simple memory allocation function was written.

After implementation of tasks, dispatcher function for context switching is implemented. Dispatcher function is implemented into PendSV handler. It is asserted by setting a pending status bit in the system control block in cortex-M0. Thus, it can be set even during the execution of higher priority exception handler. PendSV exception is used to achieve context switching as it is mentioned before. In addition, this exception can be used for the following:

- Separating an interrupt processing task into two halves.

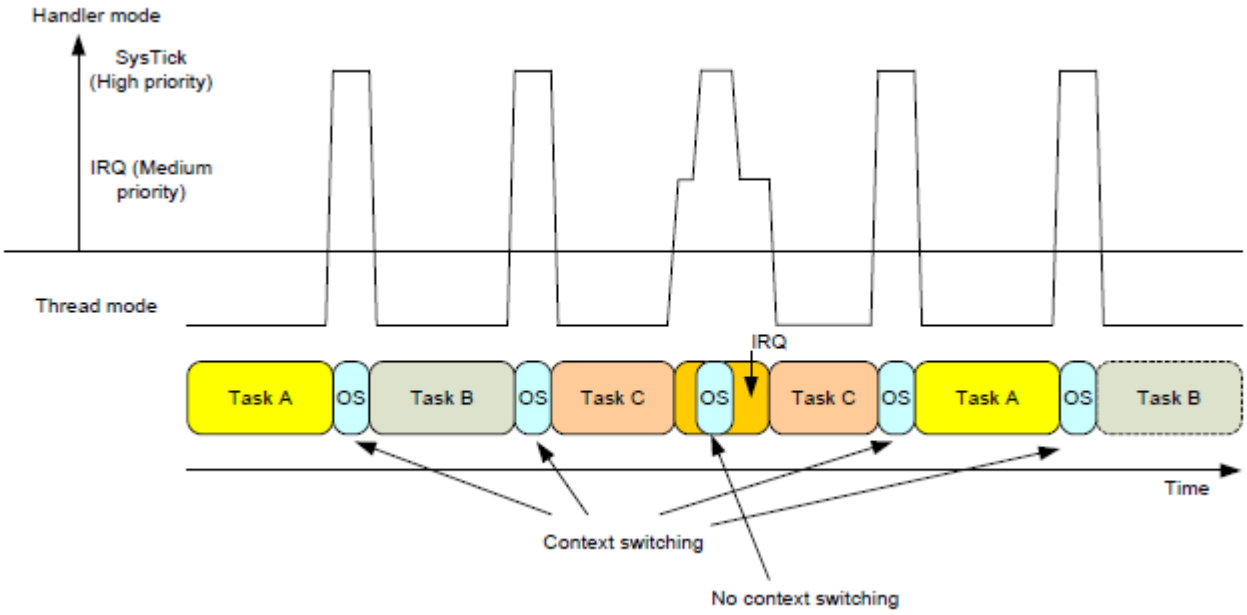

Figure 5.3: Context Switch without PendSV[37]

- The first half must be executed quickly and is handled by a high-priority interrupt service routine (ISR)
- The second half is less-timing critical and can be handled by a deferred PendSV handler with a lower priority. Therefore, it allows other high-priority interrupt requests to be processed quickly.

The SysTick exception is set up as a medium priority exception. Therefore, the SysTich handler can be called even during the execution of another interrupt handler. However, the context switching should not be discarded while an interrupt service routine is running. In that case, the interrupt service could be broken into multiple parts. Thus, it leads to increase of overhead and this can be clearly seen at the figure 5.3. In this particular system, systick check the need of context switch and if context switch must be called, it sets PendSV exception and processor immediately perform context switching since it has higher priority than system tick timer handler.

There are three native functions in dispatcher library. Dispatcher first kick function provides the necessary first kick for the scheduler. It obtains the location of the current task control block. Then saves the first item in control block as top of the stack. Afterwards it discards everything up to register 0 (R0) and makes PSP as the new top of the stack to use in the task. Moreover, it

manipulates PSP stack and copy the contents. Finally, it pops out the program counter to jump to the user defined task code.

Dispatcher switch handler, as it can be understood from its name, handles with contents during the context switching. PendSV handler calls that function to switch between tasks. It works same as first kick function. Only difference is saving and restoring registers for switched tasks.

Last function of dispatcher is task yield function, which asserts PendSV exception. It is used at the end of each task to indicate the end of task. With the help of yield function, scheduler can choose the new task to execute. Flow chart of context switching and functional algorithm of dispatcher and system tick timer can be seen at the figure below.

To complete the kernel of particular operating system, software interrupt mechanism is needed to allow tasks to trigger specific kernel functions. That can be achieved using Supervisor Call (SVCall) in Cortex-M0 architecture. This exception can be triggered using provided instruction for SVC. . "Typically, when the SVC instruction is executed, the SVCall exception is triggered and the processor will execute the SVCall exception handler immediately, unless an exception with a higher or same priority arrived at the same time and is being served first" [37]. In some systems, SVC is used as access point of some operating system functions which do not require any address information. Thus, it provides complete separation between application and operating system. In this system, SVC is used as a gateway for using mutex and IPC.

Inter-process communication or commonly known as IPC is a mechanism or services, provided by operating service, which allows processes to share data with each other. In this particular project, IPC capability is limited due to light-weight architecture. Therefore, to achieve this mechanism only mutex concept is implemented. Mutex is used abbreviation of mutual exclusion object. In computer programming a mutex object allows tasks and process to share the same resource not at the same time. When a program is started, a mutex is created with a unique name. After this stage, any task/process that needs the resource must lock the mutex from other task/process while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished. In this system, native mutex function is implemented inside of the kernel since also other kernel functions may need to use mutex mechanism. User needs to declare the mutex object as a structure to use it. Mutex structure can be seen at the figure 5.2.

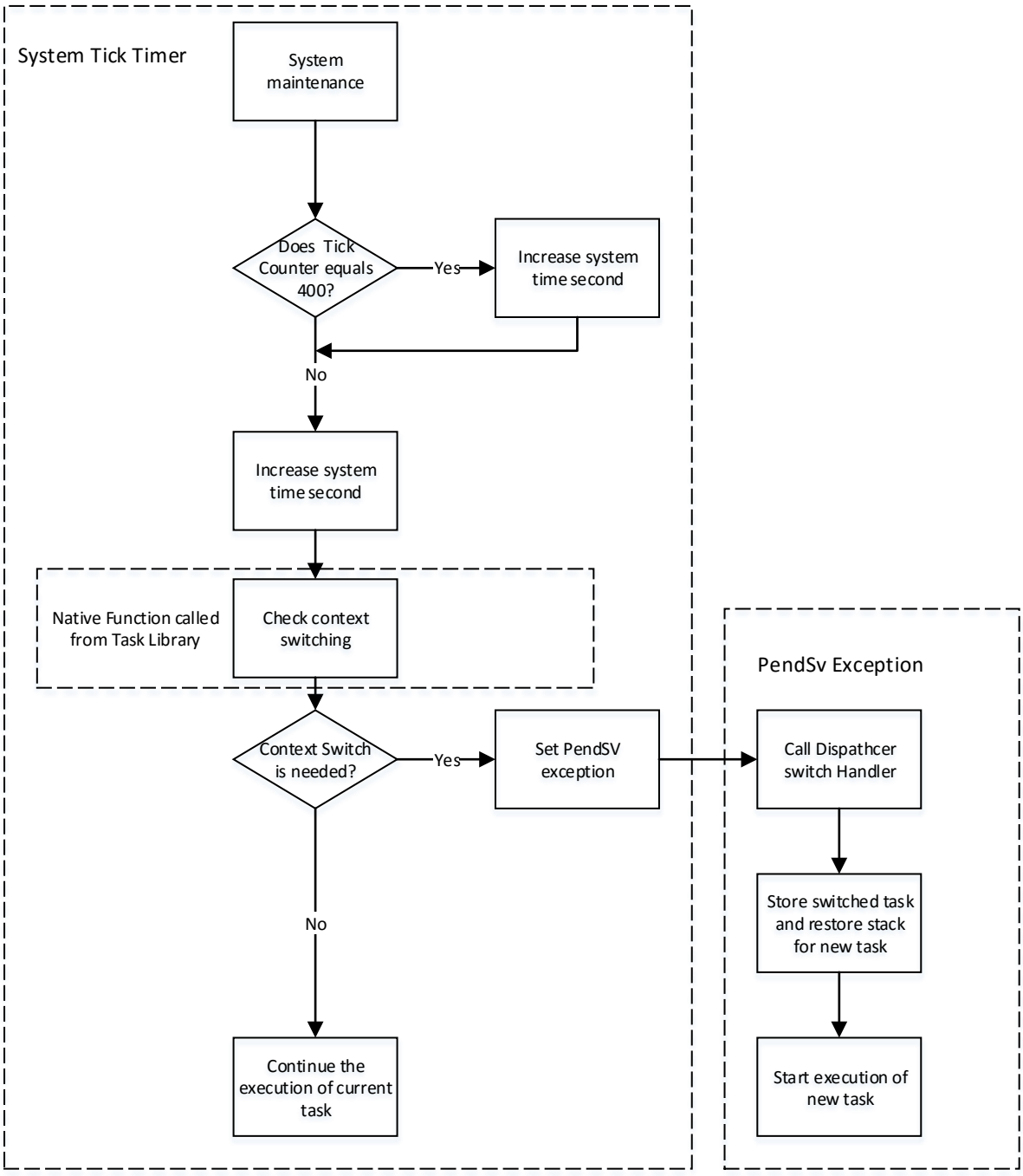

Figure 5.4: Flow diagram of system tick handler and dispatcher

```
typedef struct {
        mutex_state_t    state;
        mutex_id_t       mutex_id;
        bool             initialized;
}mutex_struct_t;
```

Listing 5.2: Mutex Structure

The mutex object has very simple design. There are two possible states for mutex object and these are locked and unlocked states. Other fields of mutex structure (`mutex_id` and initialized) are set by operating system during the creation of mutex element. In order the lock or unlock the mutex object, two functions with the same name are implemented. User just needs to give specific mutex as the argument of the function to use them. In order the lock and unlock mutex even from top level without having issue with other kernel operation, SVC exception is used. Once the function is called to lock/unlock the mutex, this immediately asserts an SVC exception. In order pass arguments to the this exception, master stack pointer(MSP) and process stack pointer(PSP) are used as it was mentioned earlier in this section.

5.2 Priority-Driven Scheduler and Real-Time Compatibility

Scheduling is an important concept in operating systems. It means assigning of a work/task or processes to a resource, which completes that. The work may be computation elements such as task, process or thread which are in turn scheduled onto specific hardware such as processor, peripheral unit or server. Schedulers allows user to use and share system resource effectively and efficiently. Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer or an embedded system; the concept of scheduling makes it possible to have computer multitasking with a single processor systems.

In different systems and application, schedulers may have different goals. For instance, maximizing throughput (the total amount of work completed per time unit), minimizing response time (time from work becoming enabled until the first point it begins execution on resources), or minimizing latency (the time between work becoming enabled and its subsequent completion). During the implementation of scheduler, usually these goals conflict with each other. Therefore

suitable compromise for different application must be done. Preference is given to any one of the concerns mentioned above, depending upon the user's needs and objectives. In real time application like embedded systems for control industry, the scheduler must ensure that all the task or process can meet deadlines. Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end. Since this particular project is based on lightweight design for embedded systems, the main goals of the scheduler are simple and real-time compatible task execution.

Priority levels can be defined for each task to execute high-priority task before lower priority tasks. In addition, the operating system run the task for a number of time slots until it reaches an idle state, if the task has a higher priority than others do. On the other hand, exception priority and priority definition in the operating system is completely different than each other. The definition of task priority is based on application, user and sometime Operating System design and varies between different application and operating systems [37].

Fixed-priority pre-emptive scheduling is the base of particular scheduler. In this scheduling schemes, the scheduler ensures that the highest priority task is executed among the all tasks that are currently ready to execute in scheduler. Since it is also pre-emptive scheduling, system tick provides the scheduler to switch after the task has had a given period to execute. With that scheduler ensure that no task is executed in processor for any time longer than system tick. Nevertheless, this leads system to lockout since priority is given to higher-priority tasks, the lower-priority tasks could wait an indefinite amount of time. Therefore, user must know the application and assign priorities carefully. Most of the embedded operating system use pre-emptive schedulers.

In this particular project, scheduler can be considered as the conductor of an orchestra. Scheduler does not provide lots of functionalities to user or the other layers of operating system. However, it orchestrates the system and task. Thus, this system has very small and efficient scheduler using pre-emptive scheduling. Most of the priority and task management functionalities are provided by task library.

Every task has a state to be executed properly by scheduler. This system a task may have six different state to ensure reliable execution of task. These are running, ready, waiting, sleep, suspended and blocked. Task library also provides lists about task states. Whenever a task changes

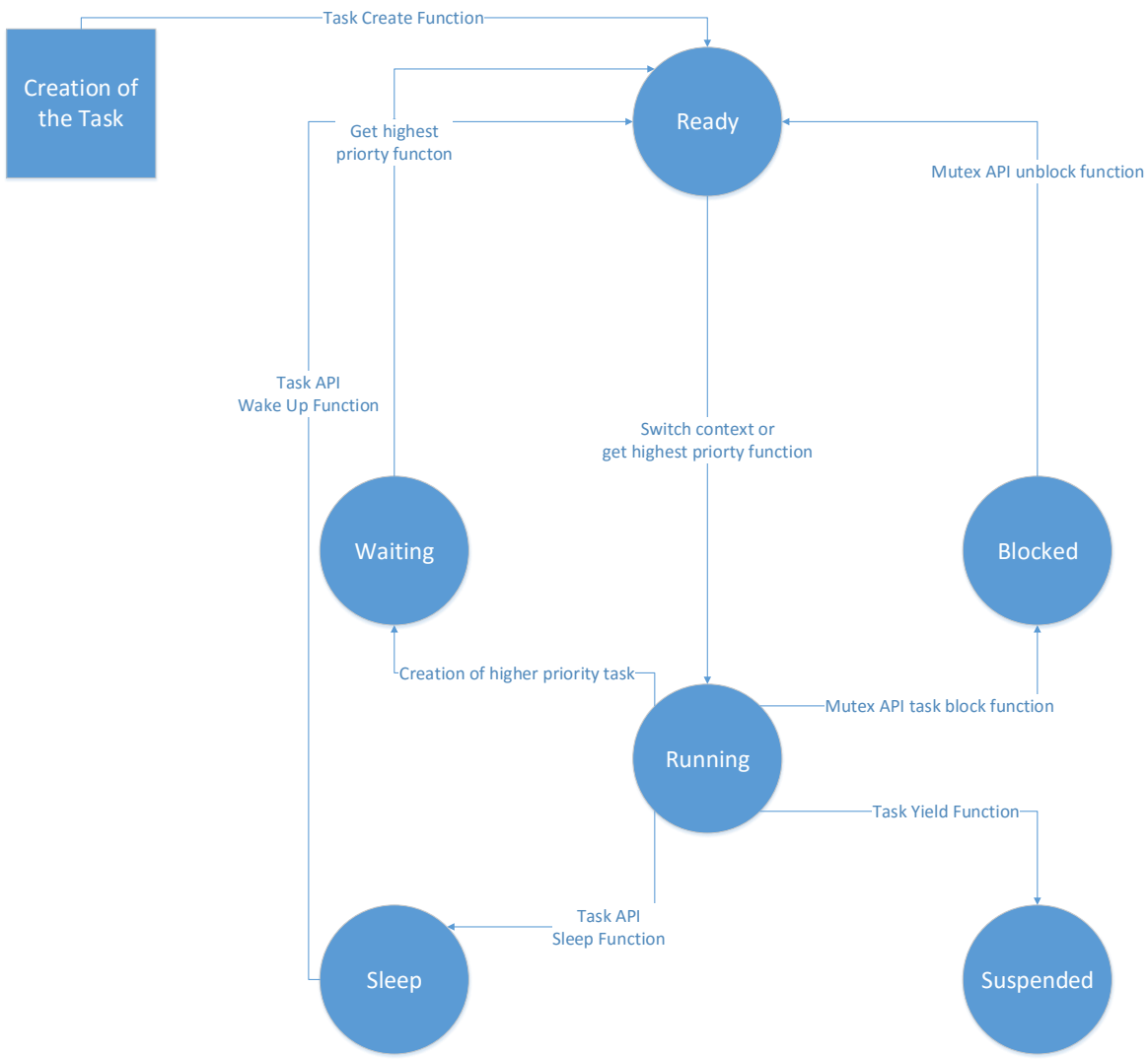

Figure 5.5: Transition of Task States

its state, it is removed from previous state's list and added to current state's list. Transition between states can be seen at figure 5.5. When task is created, its state is set to ready state. In the same creation function, priority control is done and if it is the only task to be scheduled or it has the higher priority than current task, its state is changed immediately to running state.

During the context switching relevant functionality is also provided by task library. If context switching occurs, current task's state is changed to waiting and it is added to waiting list. After the execution of highest priority task, task yield function must be used. With yield function, current state is declared as suspended state and highest priority task can be chosen. Whenever a new task must be chosen after the execution of highest priority task, firstly ready list is checked

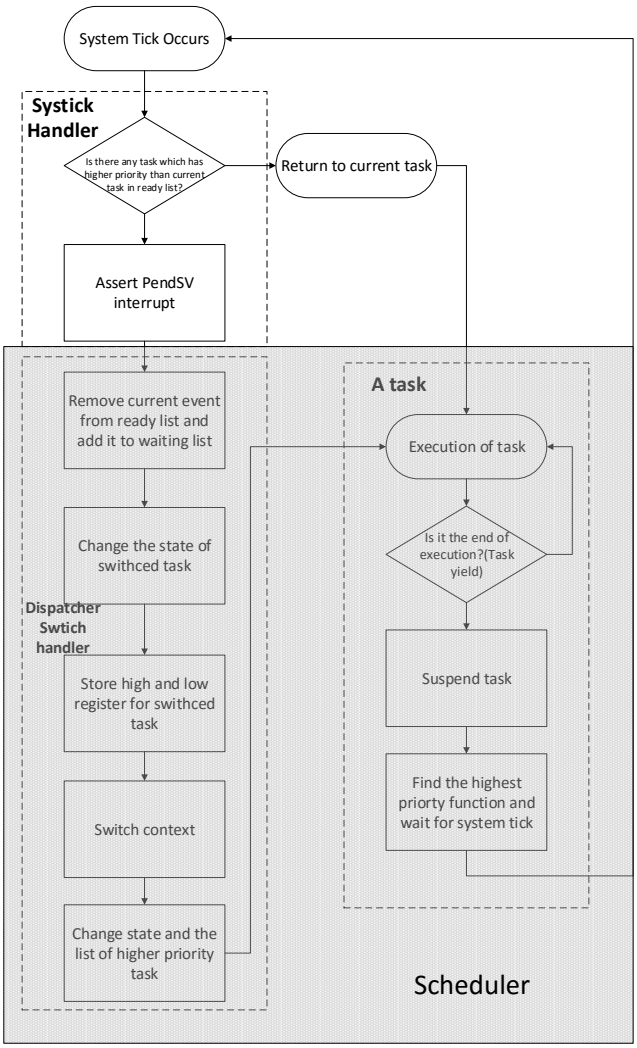

Figure 5.6: The flow chart of priority-driven scheduler during the context switching

for same priority. If there is no task in ready list same priority with the highest priority task, then waiting list could be started to check with the same priority. Whenever a suitable task is found to execute, its states changed to running again and all process repeats after the execution of every task. The flow chart of priority-driven scheduler during the context switching can be seen at the figure 5.6.

In order to create more reliable semaphores and prevention of deadlock, block state and list is added to task library. Task block function is called by mutex function, if particular mutex which is tried to locked by task is already lock. Whenever the mutex object is unlocked, the functions unlock the task, which was blocked by mutex. This block and unblock functions is

called by SVC as it was earlier mentioned in this paper.

Moreover, in order to sleep and wake the task according to an external event or interrupt, sleep task and wake up task functions are implemented. Task is put to sleep state whenever sleep function is called and can be woken up using wake up function and add immediately to ready list. However, when it wakes up, if there is a higher priority task that is being executed by scheduler, the task waits the end of higher priority task's execution.

Chapter 6

Other System Components

In this particular section, other small components of operating systems are explained. As it was mentioned before, operating systems are formed by many components. Although hard drivers, hardware abstraction layer and kernel form the main part of the operating system, there are some components, which are implemented in order to develop software application using this particular operating system. These components are utilities library, application-programming interface and user defined function (application layer).

6.1 Utilities Library

In order to make development easier, particular operating system provides some basic functionalities for basic software concepts. The provided libraries are linked list and memory allocation libraries. In computer science, a linked list is a linear collection of data elements, called nodes pointing to the next node by means of a pointer. This structure formed by a group of nodes that together represent a sequence. Each node contains data and a reference to next node and it provides efficient insertion and removal of elements from any position in an easy way. The structure of basic linked list is illustrated at the figure 6.1

This structure is the one of the simplest and most common structure in computer science. It can be used to implement abstract data types, including list, stacks, queues, associative arrays, and S-expressions.

Linked lists are among the simplest and most common data structures. They can be used

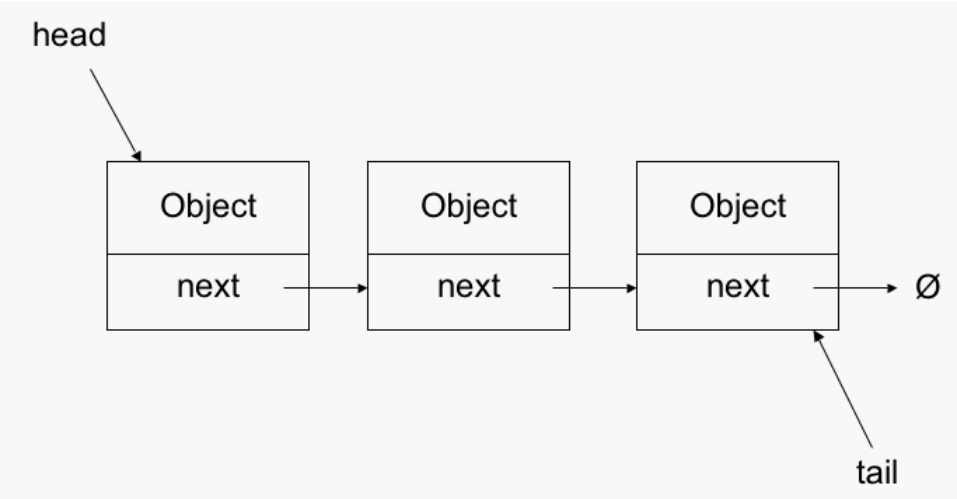

Figure 6.1: Structure of Linked List [38]

to implement several other common abstract data types, including lists (the abstract data type), stacks, etc. The purpose of implementing a linked list rather than an array is the insertion and removal of the element can be done easily without needing reallocation or reorganization of the entire structure since the data items does not need to be stored in the memory. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link before the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, may require sequential scanning of most or all of the list elements.

Linked list library provides four function to user to create reliable linked lists. Every list must be initialized before adding new nodes and `list_init` function is used for that purpose. All the content in the particular list can be deleted by using `list_clear` function. In order to add a node to initialized list, a node must be declared using node data structure in the library. Afterwards particular node can be added or removed by`list_node_add` and `list node_remove` functions. In some kernel functions in this system, linked list library has been used and it can be used also in application layer for user specific functions.

Memory management is the concept of managing system memory in kernel or system level.

Essentially providing functions to allocate portions or sections of the memory for the programs at their request and free that section after the usage is required.

There are several methods and concepts which is implemented in computers. Virtual memory systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the effectively available amount of RAM using paging or swapping to secondary storage. However in this particular project none of them are used since using this concept would be overkill for embedded oriented operating system. Therefore, basic memory allocation function is implemented for developers to use it in the application which memory allocation is needed.

Allocation request contains locating a block of unused memory of sufficient size .These request are done by allocating section from the heap. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations.

On the other hand, there are several issues which makes implementation complicated. For instance external fragmentation may occur when there are many small gaps between allocated memory blocks. Allocations must be tracked to ensure that they do not overlap and no memory is lost as memory leak.

Since standard memory allocation library has large footprint, a simple memory allocation utility library was developed. This library has basic functionality for memory allocation. Memory allocation can be achieved using `malloc_alligned` function. It allocates memory and returns the pointer of the stack. In addition, allocated memory can be freed using mfree function. These functions are used in creation of the task for the operating system in kernel level and can be used in user applications as well. At the figure 6.2, memory sections can be seen after the allocation.

6.2 Application Programming Interface

Application Programming Interface commonly known as API is concept, which contains set of routines, protocols and tools to develop software applications. Therefore, APIs can be seen translator for a software or hardware component in terms of their implementations that allows definitions and implementations to vary without compromising the interface. Therefore, a good

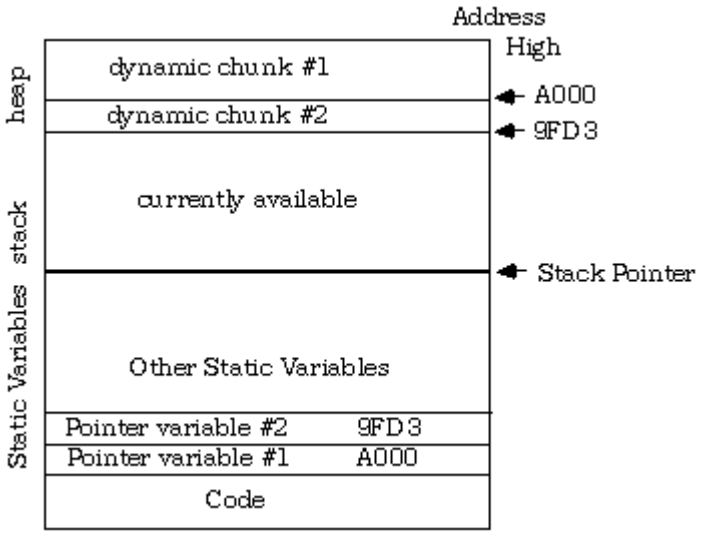

Figure 6.2: Main memory after the memory allocation [39]

API provides all the building blocks of particular component and makes software development easier.

An API can be developed for the component of operating system, web-based system or database system, and provides functionalities to user to create application for that particular system by using specific programming language. Depends on the programming language, usually an API comes in the form of a library, which contains specifications for routines, data structures, object classes, and variables. In the operating systems, API allows user/developers to use predefined functions to use or interact with the operating system component instead of writing them from scratch.

Almost all the operating system in the market such as Windows, Unix and the Mac OS provide an API for developers. Moreover, it can be used by game consoles, other hardware specific devices or embedded systems which can execute software programs. For instance, in order to develop application for Android can use an Android API to use the front camera of an Android-based device. Although developer can develop library for camera from scratch using native Android libraries, it would take lots of effort and would not be compatible with every android device.

This particular operating system provides five different APIs to user for development usage. These are boot, communication, IOT service, IPC and task APIs. In order to keep simplicity and achieve lightweight design, only the most needed APIs have been implemented to ease the pain

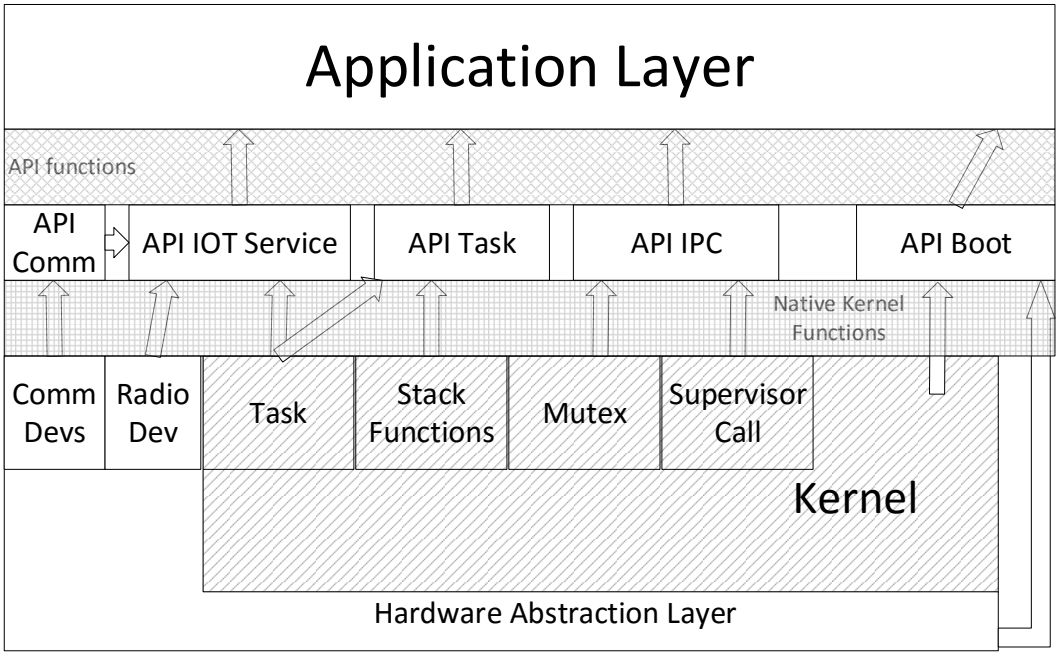

Figure 6.3: The structures and functionality of API

of development process. For other functionalities of hardware, user can directly interact with hardware abstraction layer. Since it is an embedded operating system, further development of APIs is not required. The structures and functionality of API can be seen at the figure 6.3.

Booting or commonly known as booting up is the initialization routines of the electronics and computer systems. This process can be executed after CPU is switched from off the on (hard) to detect hardware error or can be executed by a software command (soft). When booting is complete, the normal, operative and runtime functionality of operating system is attained. The piece of software, which is responsible from booting-up, is called boot-loader. It loads the operating system or some system software. On modern personal computers, this process may take tens of seconds and usually helps to perform a self error test, locate and initialize peripheral devices. However in the embedded systems, the booting sequence is not that complicated. Simply, it sets hardware registers and run operational programs that are stored in ROM. Therefore, also in that particular operating system booting sequence is short and small. API boot provides boot start function to user. User needs to call that function to set the operating system and the platform correctly. Boot start function creates a task and executes it before the user defined tasks. It sets GPIO, communication block and other small components in the system and

it returns error message if there is a hardware error in the platform.

The other API library provided by OS is API communication. Actually, this is the only API which is not provided for user. It is provided for IOT service API to make the development of that component easier and compatible with further release of the operating system. User can use other communication protocol or modules such as SPI,I2C and etc. by using hardware abstraction layer. However the privilege to use protocols or modules which can allow device to communicate with the other devices, is not given to user. User can interact with IOT service to have basic functionality in this issue. IOT service and its API are explained in detail in the section called “IOT Service and Security”.

Since mutex element is created by using the native functions and supervisor call of the kernel, API IPC is implemented to secure the kernel from user-defined task errors or mistakes. API task provides basic functionality of the mutex object. User must need to declare a mutex object using the given data structure type in the `api_ipc.h`. After the declaration of the object, user can lock and unlock the mutex element by using basic functions with the same name. API mutex functions are executed in the same way with mutex native kernel functions. Only difference is using API, the important details are hidden from users/developers and basic functionality is provided.

The most important and the biggest API in the particular system is API task. As it was discussed before, the most important feature for embedded operating systems is task execution. The task execution needs to have real time compatibility and should be performed reliably and efficiently. Therefore, the API for task creation and scheduling must be user-friendly. In addition to that, it needs to give the full functionality to user with a small size design. The native library of the task was explained earlier section in this paper. The API task gives user the every aspect of task scheduling and creation. It provides functions for creation and deletion of tasks. Before the creation of the task, initialization function must be called to set the operating system for this API. User needs to use yield function if scheduler wants to be run after the execution of the task. Moreover, user can start scheduling of the user tasks by using `api_task_start` function. These functions may return error message, if unexpected error occurs during the task creation, deletion or starting scheduler. Furthermore, this API provides sleep function depends on specific event such as timer event in the operating system.

6.3 User defined functions

Application program means a piece of code or software, which is designed to perform a group of functions, task or activities to achieve specific user task with using particular operating system. For instance in personal computer worlds, a word processor, an accounting application or a web browser are considered as applications. The concepts refers to all kind of applications running in specific operating systems. Applications may be bundled with the computer and its system software or published separately, and may be coded as proprietary, open-source or university projects.

An application is designed to help user perform an activity. Therefore, it differs from an operating system (which runs a computer or embedded system) or a utility (which performs maintenance or general-purpose functionality). Depending on the activity and targeted operating system for which it was designed, an application can manipulate text, numbers, hardware, or a combination of these elements.

The delineation between system software such as operating systems and application software is not exact, however, and is occasionally the object of controversy [40] .For instance; there are debates about whether Microsoft's Internet Explorer web browser was part of its Windows operating system or a separable piece of application software. As another example, the GNU/Linux naming controversy is, in part, due to disagreement about the relationship between the Linux kernel and the operating systems built over this kernel. In the embedded operating systems, the application can be indistinguishable to the user, since user can be also developer and they can develop software to control piece of hardware.

However in this particular operating system, is specifically designed for low-level applications. Therefore, every other piece of code which is written using operating system API is considered as application software. Moreover, user needs to do programming in C language since there is no internal compiler or provided graphical user interface. Thus, the main function is left for user applications. User needs to include specific API header files and call some functions to run the platform. Examples about simple application layer can be found in appendix. Moreover, in order to configure operating system specifically for application, a configuration file is implemented. It allows enabling and disabling functionalities of operating system to user to provide

flexibility for different applications. The configuration file can be seen at the listing 6.1.

```
#define XOS_H

#include <stdint.h>
#include <stdlib.h>

//memory configuration
#define configMINIMAL_STACK_SIZE     ((unsigned short)600)
#define configTOTAL_HEAP_SIZE        (( size_t )(3000))
//tasks configuration
#define TASK_PRIORITY_NUMBER         16
#define IDLE_TASK_PRIORITY           15
#define IOT_SERVICE_PRIORITY         1
//system tick configuration
#define RTC_FREQUENCY                (400UL)
//API configuration
//api_task configuration
#define TASK_STACK_DEPTH             200
//api_iot_service configuration
#define IOT_SERVICE                      (1)
#define IOT_SERVICE_MANAGER              (1)
#define IOT_SERVICE_NODE                 (0)
#define IOT_SERVICE_SUPPORTED_NODE_NUMBER    (255)
//interrupt handlers
#define xPortSysTickHandler          RTC0_IRQHandler
#define dispatcher_switch_handler    PendSV_Handler
//hal configuration
#define BUSY_WAIT_DEV                (1)
#define GPIO_DEV                     (1)
#define I2C_DEV                      (0)
```

```
#define SPI_DEV                 (0)
#define UART_DEV                (1)
#define RADIO_DEV               (1)
#define ADC_DEV                 (0)
#define TIMER_DEV               (1)
#define COM_DEVS                (1)

extern volatile uint32_t system_time_second;
#endif
```

Listing 6.1: Configuration File of Particular Operating System

Chapter 7

Radio Library, Stack and Protocol

For this particular project, in order to provide wireless communication, a new simple protocol has been design. In addition to that, in order to have efficient communication while using that protocol, radio library is designed from scratch. Library and Protocol design are explained in detail in this section. Implementation and purpose of developing a new library, stack and protocol is mentioned as well.

As it was explained before targeted platform(NRF51822) has a radio module in the same SOC together with an arm Cortex-M0 microcontroller. Nordic Semiconductor provides a software in order to control radio module by using Bluetooth Smart protocol. The piece of software is called Soft-Device, which is precompiled and linked binary software implementing a Bluetooth 4.1 low energy protocol stack for the nRF51 series of chips.

They also provide API which is written in C language and which provides the application complete compiler and linker independence from implementation. This allows user to develop their project or application as using a standard arm Cortex-M0 device without needing to integrate with proprietary chip-vendor software frameworks [41]. Structure of Soft-Device can be seen at the figure 7.1.

Besides the advantages of Soft-Device in Nordic products, this Soft-Device is not implemented in this particular project. Firstly, Bluetooth stack has a very large footprint. Since the main purpose of project to develop a small size embedded operating system, implementing the large Bluetooth stack would be inappropriate. In addition, there is no need to have different profiles for different applications as Bluetooth protocol provides. This particular project, pro-

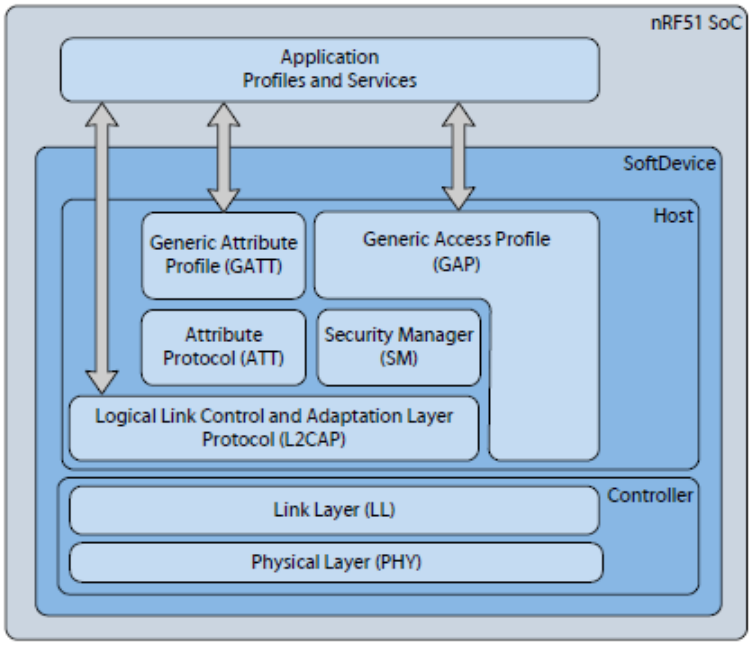

Figure 7.1: Structure of Soft-Device[41]

vides functionality and task execution for wireless sensor networks and internet of things devices. User or developer can develop their own project easily on top of this particular operating system. On the other hand, since this soft-Device comes with pre-compiled way and API, there is hard to see what is going on inside of that pre-compiled software. They may be undesired behavior for developers in some specific applications.

As it was detailed before, particular SOC contains very flexible and reliable 2.4 GHz Radio transceiver. In order to develop specific library for this module features and details of radio modules must be investigated. First of all, a DMA module which is called Easy-DMA, is implemented by vendor into radio module. DMA module provides reading and writing data packets from and to the RAM sections without interrupting CPU. It has PACKETPTR pointer for receiving and transmitting packets. Processor needs to reconfigure this pointer each time before starting of module via START task command. Accessing the RAM will have been finished by DMA module when radio DISABLE task is executed. If the PACKETPTR is not pointing to the Data RAM region, an Easy-DMA transfer will result in a Hard-Fault [35].

A Radio packet contains the following fields: PREAMBLE, ADDRESS, LENGTH, S0, S1, PAYLOAD and CRC as illustrated in Figure X. When radio module sends a packet, order of the packet

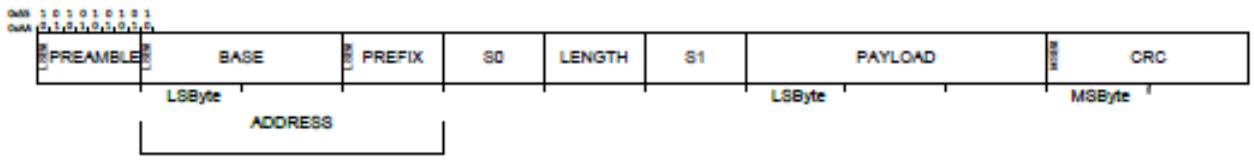

Figure 7.2: On-air packet layout [35]

is described from left to right in the figure 7.1. The pre-amble bits is sent firstly on-air. It is always one byte long for all the modes provided by MODE register.

Radio module also has CRC generator to calculate cyclic redundancy check over the whole packet excluding preamble. Therefore, no need to implement an external CRC library in order to check CRC errors. In addition to that, CRC error is detected by radio module automatically. After setting packet and radio register, series of command must be followed to send the packet to other device. The module must first ramp-up in TX mode. This mode can be set when TXEN task command is triggered .After the successfully ramping up of radio module, it generates the READY event to indicate that transmission can be started [35]. However, event is not used in this library as an interrupt in order to keep reliability. System simply busy-waits while radio is getting ready. Afterwards, transmission is started by using START task.

Like in transmitting, radio module must be ramped-up in RX mode to be able to receive a packet. Once the RXEN task is executed, RX ramp-up sequence is started. After the successfully ramping up of radio module, it generates the READY event to indicate that packet reception can be started [35]. In this particular event called packet-received event, radio module asserts an interrupt. System does not wait to get a new packet; it performs other task or applications. Once it receives a packet, it asserts and interrupt and radio stack and library will handle with packet. Packet handling is implemented on IOT service and more information about the service can be found in the following chapter. After asserting packet receive interrupt, radio module enters RX-IDLE state and radio module is closed/disabled by radio library to reduce energy consumption. Particular radio modules supports communication between 2.400 GHz and 2.483 GHz since it is specifically designed for Bluetooth Smart protocol. Therefore, new protocol and library use same physical layer with Bluetooth protocol.

In this particular protocol, defines two types of device. These are manager and node de-

vices. User needs to assign the device one of them. Manager communicates every device in the cluster and analyze their need and enable to share resource and information inside the cluster. Every cluster can only have one manager. On the other hand, node device is the other device which performs specific task such as computing, sensing or actuating. Nodes do not communicate with each other. According to protocol every device that is connected to cluster, uses different channel to communicate with manager. A node is registered by manager at the beginning. Manager has a choice to reject the node or accept to the cluster. This communication between unregistered node and manager is done at the 2.44 GHz. After successfully registration of the node, manager assigns specific id and channel for every particular node/device. Each new channel is given with 1 MHz difference from previous assigned channel. Therefore, this protocol only support 82 device inside of one cluster. Bandwidth for every channel is 1 Mb/s. More information about manager and node devices functionality are explained in detail in next section.

The radio library has simple and small architecture. It provides six different function to hardware abstraction layer for basic functionalities of radio module. These are open, close, send, receive, configure and change channel functions. Radio module is configured by the operating system at the start-up of the platform using boot function. Before sending and receiving packet `radio_open` function must be called to start ramping up task for transmission or packet reception. `Radio_change_channel` function is responsible from changing channel frequency for particular device. User needs to give message pointer and length as argument before using send and receive functions. Only receive function creates an external interrupt which is handled in IOT service layer. Radio module has a register called state register to use radio state in callback functions. However Nordi Semiconductor did not implement this register yet [34] . In order to achieve state tracking in this system a state type is declared separately and a variable is declared from it. Every function of radio module, this variable is changed according to datasheet to track state of radio module and send it to the high-level callback function. The structure of radio library and interaction between radio library and IOT service can be seen at the figure 7.3.

A communication protocol means a group of rules that provides two or more devices to transmit and receive information to each other via any kind of variation of physical and electronics quantity in tele communication. The set of rules defines the syntax, semantics and syn-

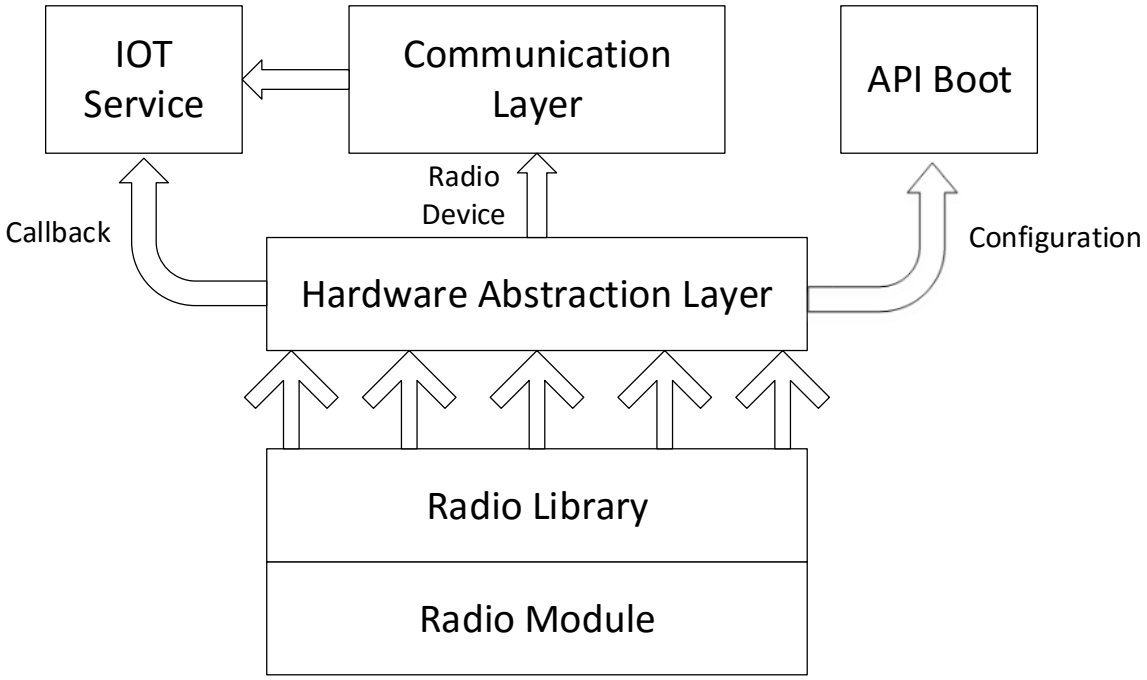

Figure 7.3: Structure of Radio Library and Interactions

chronization of communication. Protocols may be implemented by hardware, software, or a combination of both [42].

Systems use defined formats for exchanging messages. Each message has an exact meaning intended to elicit a response from a range of possible responses pre-determined for that particular situation. Implementation is independent from specified behavior of protocol. In addition, there is a close analogy between protocols and programming languages since protocols are to communications as programming languages are to computations [43].

In computer and embedded systems, this protocols is expressed by algorithms or different data structures. Expressing the algorithms in a portable programming language makes the protocol software operating system independent. Typically, a set of processes, which manipulates shared data to communicate with each other, is provided by operating systems. This communication is achieved by using well implement protocols which may be implemented into the process code itself. On the other hand, since there is no shared memory among devices, devices needs to communicate with each other using a shared transmission medium. Each device or system may use different hardware or operating system while using same protocol.

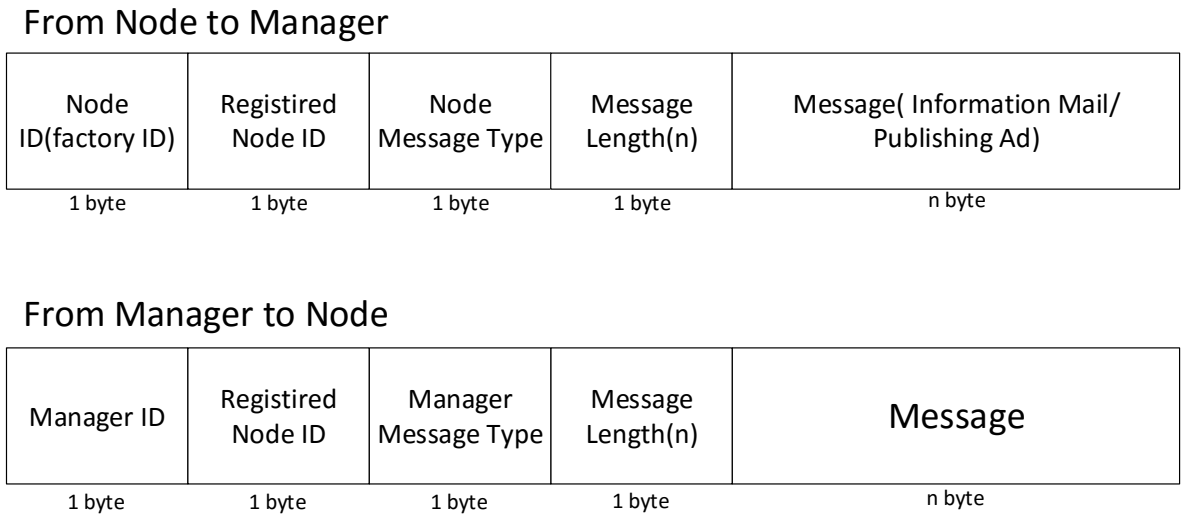

Figure 7.4: Packet Layout of the New Radio Protocol

This particular protocol is developed in order to enable resource and information while keeping protocol stack footprint as small as possible. This protocol basically is based on two different concept and these are yellow pages and mailing system. Every node creates a mail periodically and interval of that period can be configured by user/developer. Manager collects these mail whenever it Is suitable for it and create yellow pages according to need of the nodes. Moreover manager can calculate the workload of the cluster or particular node. All relevant information to analyze particular node is included to the mail with timestamp. To achieve this conceptual communication, the protocol is designed based on this. Every node has one byte node id which is given by manager and that is the first byte of the message from node to manager. Afterwards one byte device specific id (factory id) is included. Then one byte message type and one byte of message length comes. Message Length * 1byte long message comes afterwards. The protocol from manager to node is similar as well. Only first byte of message contains manager id and second byte of message contains node id which is given by manager. The packet layout of particular protocol can be seen at the figure 7.4.

A protocol stack means an implementation of a computer networking protocol suite. The communication protocols can be considered as definition and protocol stack can be considered the implementation of particular rules in software.

Protocol stacks are often designed as layered software. This modularization allows user to

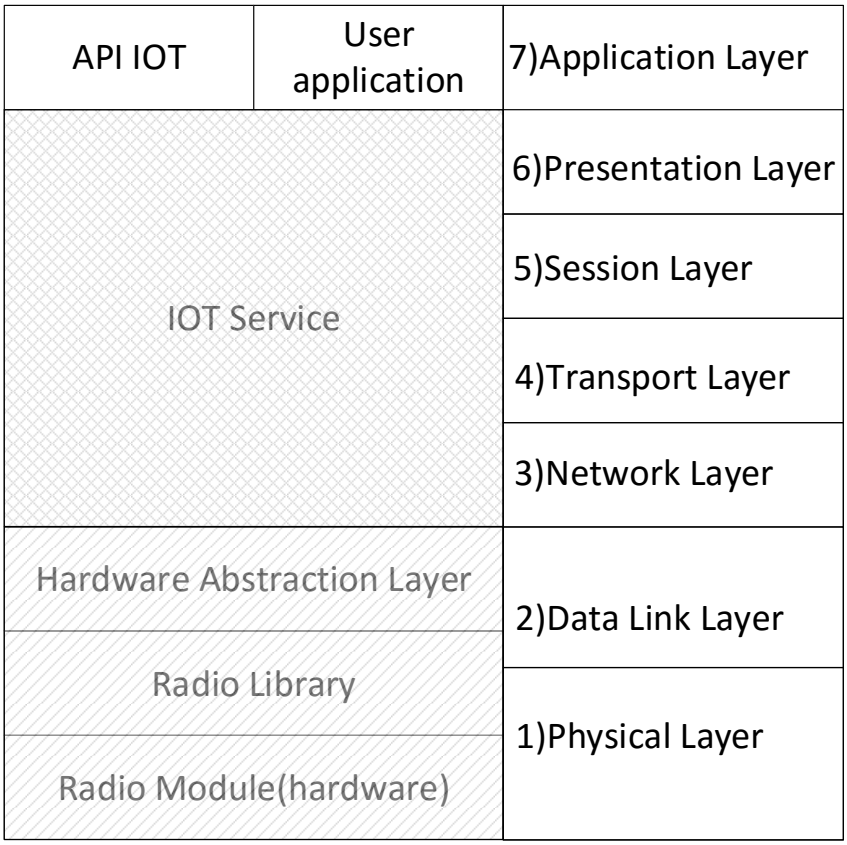

Figure 7.5: The stack structure of the radio protocol based on OSI model

design and evaluation easier. The lowest layer of protocols usually handles with low-level, physical interaction of hardware. In this case (this particular operating system), lowest layer of protocol stack is radio library. Higher layers adds more features and most of the user applications only deal with top layers.

There are three major section usually in every protocols stack. These are media, transport and application layer. In addition to that, mostly two interfaces are implemented between those layers by a particular operating system or platform.

This particular protocol is implemented using three different components of the system. If seven layer OSI model considered as the base of this protocol, physical layer and data link layer is performed by radio library. IOT Service handles with network, transport, session and presentation layers. User can use the stack using IOT API and that forms application layer of the protocol. The stack structure of the radio protocol based on OSI model is illustrated in the figure

7.5.

As it was discussed earlier in this section, main reason of the developing a specific radio library and protocol is large footprint of provided Bluetooth soft-device. This pre-compiled soft-device needs 92 kb space in the flash memory. Moreover when it is enabled, it needs minimum 6 kb and regularly it requires around 8 kb space in the RAM. Furthermore, its maximum call stack usage at run time is 1536 bytes. On the other hand, after successfully implementation of particular radio library and protocol, all the implementation takes 1,86 Kb of RAM and 2,67 Kb of Flash memory. This also includes the size of IOT service, which enables resource and information sharing. Since a running task handles with radio module and protocol, its maximum call stack usage at run time is 600 bytes. Performance of the protocol and implementation and developing radio library and protocol from scratch is discussed in the section called “Discussion”.

Chapter 8

IOT Service and Security

In this chapter, designing steps of internet of things service, theory lies behind it and how this service operates inside of the particular operating system are explained in detail. Since IOT service enables wireless communication and resource and information sharing among the cluster, security vulnerability caused by this and how the security issue has been solved using this particular service have been mentioned.

In last twenty years, many technologic achievements have been happened in the Integrated Circuit industry. These have led to develop small, energy efficient and faster processors and microcontrollers. Nowadays even a small microcontrollers which cost ten to twenty dollars, have enough CPU power and memory space to achieve complicated tasks such as managing with TCP/IP stack or Bluetooth communication. Even some operating systems can be supported by this kind of low cost microcontrollers. Because of that lots of availability of low cost devices, every kind of embedded systems have started to connect with each other and internet.

This technological achievements lead industry to use low cost embedded devices to create large sensor network or IOT devices. Therefore even smallest failure or attack from outside have turned out to be the most crucial problem since these devices either store private data or doing very crucial task in harsh environment. Failures may not create big problems. However sometimes these kind of failures may cause of shutting down whole network. In order to prevent network such a scenarios, a new approach just for sensor networks must be developed in both network layer and software layer.

As it was discussed earlier in this paper, a new operating system service called Internet of

Things Service has been introduced in this project to solve particular problems that are mentioned at the paragraph above. This service sits top of everything in this particular architecture ad enables resource sharing and device connectivity. Application layer (user defined task and functions) cannot reach communication layer without using IOT service. This service or architecture can be investigated from three different point of view and these are network and protocol details, IOT service functionality and security issue.

8.1 Network Structure

Many different approaches have been tried and implemented so far in academy and industry for sensor networks. However almost all of them had tried to improve the power consumption or tried to allocate sensor automatically. On the other hand, our aim is creating solid sensor network which can detect network faults, fix them and in the meantime it should keep the latency as low as possible [44].

To implement and develop such a network, computer clustering can be the base of the infrastructure. Computer cluster consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. Unlike grid computers, computer clusters have each node set to perform the same task, controlled and scheduled by software. The computer clustering approach usually connects a number of readily available computing nodes via a fast local area network. The activities of the computing nodes are orchestrated by "clustering middleware", a software layer that sits atop the nodes and allows the users to treat the cluster as largely one cohesive computing unit [44].

In this particular project, different type of architectures for sensor networks and IOT devices are investigated and suggested. Three different approach has been suggested and investigated in detail. For all these three approaches, computer clustering can be considered as the base of them. These structures are;

1. Classical Cluster Approach:

   In classical approach for sensor network, sensor nodes perform their task coordinately and send their data to main computer or to another node of the cluster which is responsible for communication with main computer. Classical cluster approach can be seen at

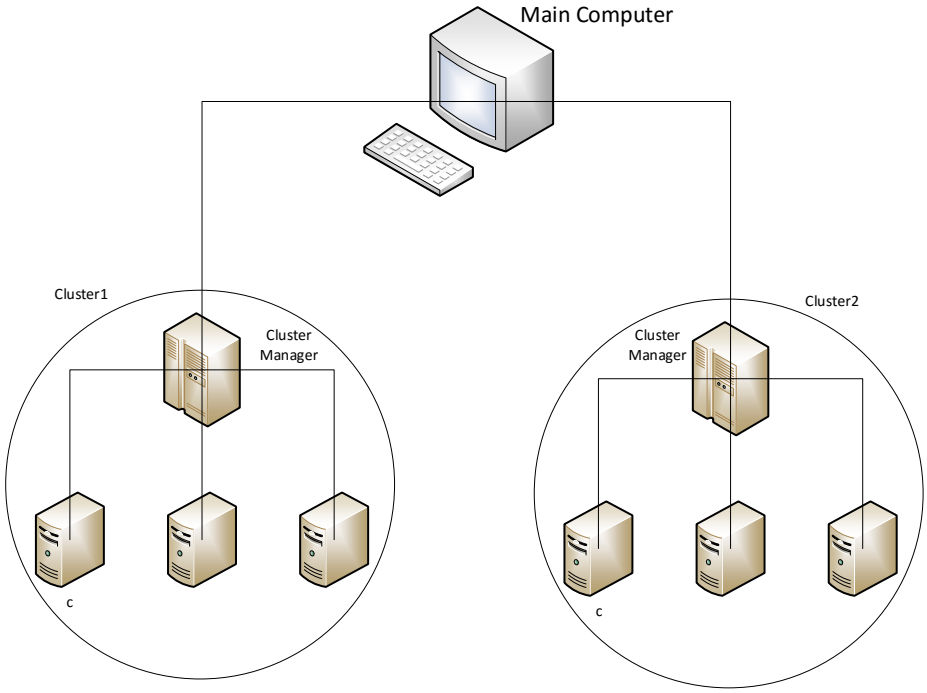

Figure 8.1: Classical Cluster Approach

figure 8.1.

2. Cluster with Main Cluster Approach:

   Classical cluster approach has some problems. Especially it is vulnerable some network faults. These problems will discuss later on. Therefore, this approach needs to be improved. Therefore, a node that should act as a kind of manager can be added. This manager can analyze network health, rule the other nodes and send all data of cluster to main computer. In addition to that, a manager cluster can be added to network which works like manager node as well. Nodes may be connected to each other consecutively. This would create communication fault. With these manager nodes, this problem will be solved. This structure can be seen at figure 8.2 [45].

3. Cluster with Nodes Net Approach:

   Different concept can be implemeted (collective intelligence, self-awareness, etc.) in order to deal with every kind of network problem for wide variety applications. These con-

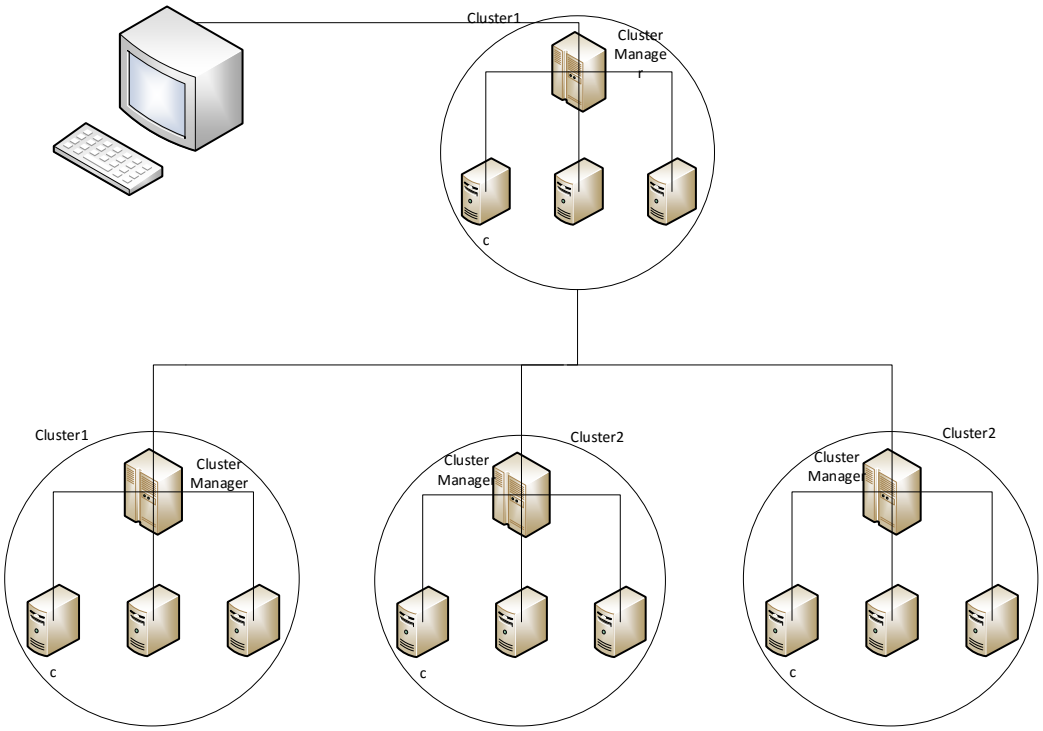

Figure 8.2: Cluster with Main Cluster

cepts requires different network structure. Also for hardware and software parts will be challenging. However, hardware and software parts have not discussed yet. In order to implement that kind of system, information about network and the other nodes must be shared with each other. With shared information, nodes will get information about environment and network. In addition, they can analyse the network and predict future errors. Moreover, they can change their tasks or priority. That kind of system will act more like an intelligent network. Structure of this approach can be seen at figure 8.3 [20].

Pros and cons of suggested structures have been investigated. In addition, possible problems and solutions of them are mentioned at the tables 8.1 and 8.2.

After investigation of cluster approaches, it can be easily seen that “classical approach” is vulnerable to some problems. On the other hand, the approach called “Cluster with Nodes Net” may be seen as the best approach among three of them. However, in the implementation part, it would take more time for development and require large footprint in the software. Moreover, nodes communicate with each other and manager in this approach. That would lead to increase

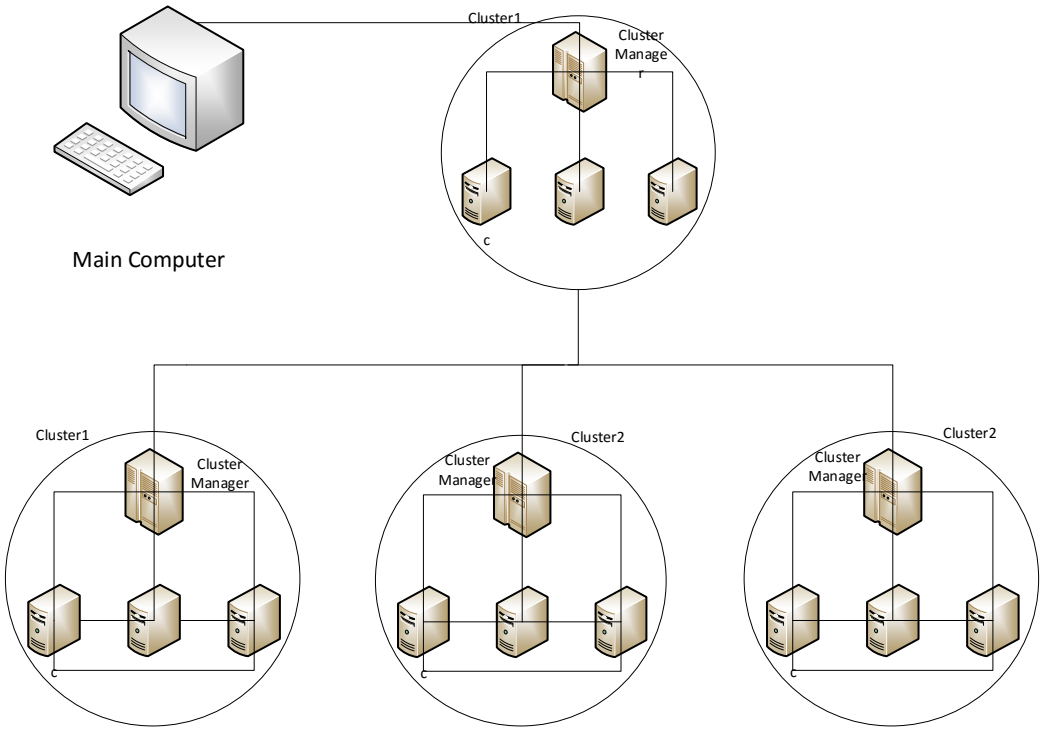

Figure 8.3: Cluster with Nodes Net

power consumption in each device. Since particular embedded operating system is focused on low power consumption as well, implementation of this approach would be inappropriate. Therefore, “Cluster with a Manager Approach” was decided to implement. The disadvantages caused by this approach is tried to be solved by developing a reliable IOT service task and manager device.

Implementation of this approach was divided in to two part. These are protocol which provides reliable and compatible communication between devices and IOT service which provides efficient execution of wireless communication and resource and information sharing. There are two types of device type in this particular approach and this also is implemented into protocol and device configuration. These are called manager and node device types. Each device has different execution sequence and algorithms according to its device configuration and as it was explained before they have slightly different message layout in the protocol.

	Approaches		
	Classical Approach	With Main Cluster Approach	Main Cluster + Node Net Approach
Pros	-Simple and Solid -Can be easily implemented to large scale -Doesn't require too much computation or process power -Secure for outside network attack	-Relatively simple to implement -Immune for some network problems -Secure for outside network attacks	-Shared data can lead to collective intelligent and self-awareness. -Reconfigurable -Immune for network problems -May decrease the latency
Cons	-Not reconfigurable (self healing) -Simple but not optimized -Vulnerable to network problems -Not an intelligent system	-Breaking down of the manager node or cluster create big big system problem. -Overhead may occur. -Not reconfigurable	-Hard to implement -Overhead may occur -Needs more processor computing power -May need different hardware for cluster managers or special nodes(for nowadays)

Table 8.1: Pros and Cons of Suggested Strcutres.

8.2 IOT Service

During the background research for this particular paper, problems and missing concept in the embedded operating systems for sensor networks and IOT have been investigated and listed at the earlier in this paper. Most of the embedded device use the old TinyOS or FreeRTOS. However, none of them is developed specific for IOT or sensor networks. They may have lots of supported architecture and they are good at real time applications. However, classical way of thinking and developing OS and firmware caused those problems. As Einstein said «we cannot solve our problems with same thinking we used when we created the». A new kernel architecture and secure API must be achieved.

This particular service uses different parts of components and different module form hardware. It is the biggest layer of the operating system after kernel. As it was discussed before it allows device to communicate with the other devices. In order to achieve this purpose, it uses radio module which is provided by communication layer. In addition to that, it has right to manipulate input and output of the device by using GPIO device provided by hardware abstraction

Problems	Solutions
-It must be compatible for every application.	-Deployment of clusters will be automated.Also application aware middleware may be solution for this problem
-Breaking down of manager node might cause fatal error in the system.	-Vice/Second manager can be assigned. On the other hand, every node in side of that cluster can get manager different priority at the beginning. If manager breaks down, according to their priority they can their hierarchical position.
-All nodes can start to breakdown at same time.	-Health report can be sent to main pc periodically. If connection goes down, node can be restarted
-Latency may increase if some nodes go down.	-Cluster can be reconfigurable while network is working

Table 8.2: Possible Problems and Solutions of Suggested Structures

layer. Moreover, since it is an operating system service, it is created by kernel as a task. Although it is executed as regular task by kernel, since kernel contains a priority-driven scheduler, it has highest priority. Therefore, it cannot be pre-empted by other task created by user. The software structure of IOT service can be seen at the figure 8.4.

The particular approach called “Cluster with a Manager Approach” introduces two different device types to network. Thus, this service comes with two different configuration. One for manager devices and another one for node devices. Both of the device types have different functionalities and execution procedure to follow. User must carefully configure IOT service by using configuration file.

A cluster that is formed by using this protocol and operating system supports only one device manager. Therefore, a device manager is unique for every cluster. Cluster manager communicates with every device in the cluster. It is the only device in cluster, which can communicate with the devices outside of the cluster or connect directly to Internet. It collects information from every device and asses this information to calculate cluster and network workload. Moreover, it ensures that every node in cluster keeps its connectivity. It can also control specific node using special function to control node resource such as computing power, memory or actuators

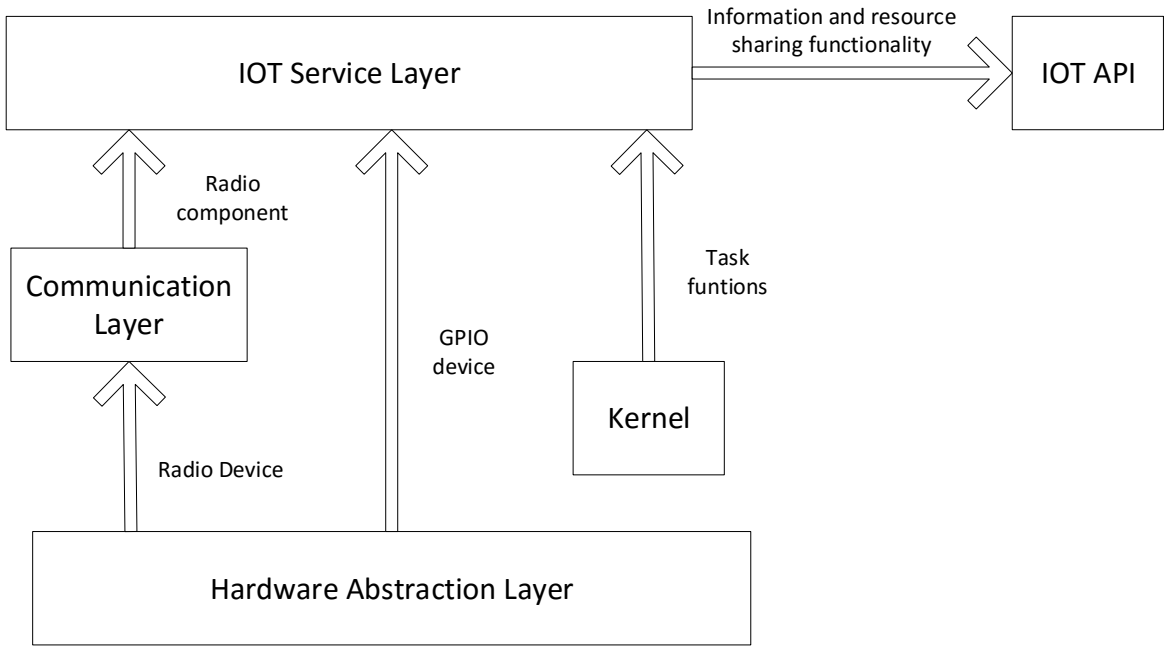

Figure 8.4: IOT service software structure

like motors or LEDs. In addition, checking device connectivity and information collected from nodes, a manager can check the problems in the cluster and do maintenance task for network. Although its massive workload, since efficient execution and usage of task and components of OS, it can also perform user specific tasks.

On the other hand, node devices act differently than manager devices. First of all, the IOT service which runs in node devices, has less workload. Therefore, these devices can be used for real-time application and they will provide better performance for meeting hard deadlines. A cluster can have many nodes. However, the protocol sit top of this IOT service, support 82 nodes excluding manager device for each cluster. It creates information about itself periodically and it sends to manager whenever manager ask to send information. It is able to communicate only with manager. During the registration, node is assigned to a channel between 2.400 and 2.483 and only manager can change this assigned channel. Since it can only communicate with manager, a node device consumes less power than manager device. Manager can use it as a slave. However, this slavery usage of a node does not affect execution of other tasks. In addition, IOT service of node devices has slightly smaller footprint than manager device's IOT service, since it has less functionality.

There are two concepts based on distributed systems and implemented into this structure. These are yellow pages and mailing system. The name of the yellow pages is choose to describe of the structure in better way. Yellow pages can be considered as advertisement page called yellow pages in newspapers. In this particular system, yellow pages are implemented into only manager devices. Manager devices retrieves information from the nodes and creates these pages. Manager can check yellow pages and decide which node needs what kind of resource. Only needs of nodes are written in these pages. Therefore, one of the task of the manager is to provide required source by nodes. Required source could be provided directly from manager or it can be provided one of the node, which is not using particular resource currently. Yellow pages are implemented as array of structures. Therefore, it can be easily send to another computer, device, or server easily. Size of yellow pages is configurable. If there is no space to write in yellow pages, it overwrites the oldest advertisement/message by using newest advertisement/message. Once manager provides needed resource for particular node, the advertisement that is published by that node, will be deleted.

The mailing system is implemented to collect information from nodes continuously. It can be considered as offline messaging system as well. It is called offline system since the messages between node and manager are not sent immediately and even there is not any connectivity between them, it saved and sent it later once get the connectivity again. A node creates a mail and registers it into its mailbox. Manager checks mailbox of nodes regularly and collects these mail from them. There are two types of mail that can be registered by nodes. These are information and publish ad mails. Nodes register information mail periodically. Information mail contains seven different information about particular node. These are needed, available and total resources, number of ready and total task, battery percentage and timestamps. Mailbox is implemented as array of structure as well so there is a configurable limit. A node overwrites the oldest information mail with new one if there is no space in the mailbox. However, publish ad mails are sent immediately since they contains information about the need of the node. This mailing system is implemented together with the yellow pages to share information and resource along the network with the low power consumption. The structure of mailing system and yellow pages can be seen at the figure 8.5.

This service handles with radio callbacks. This particular operating system has two different

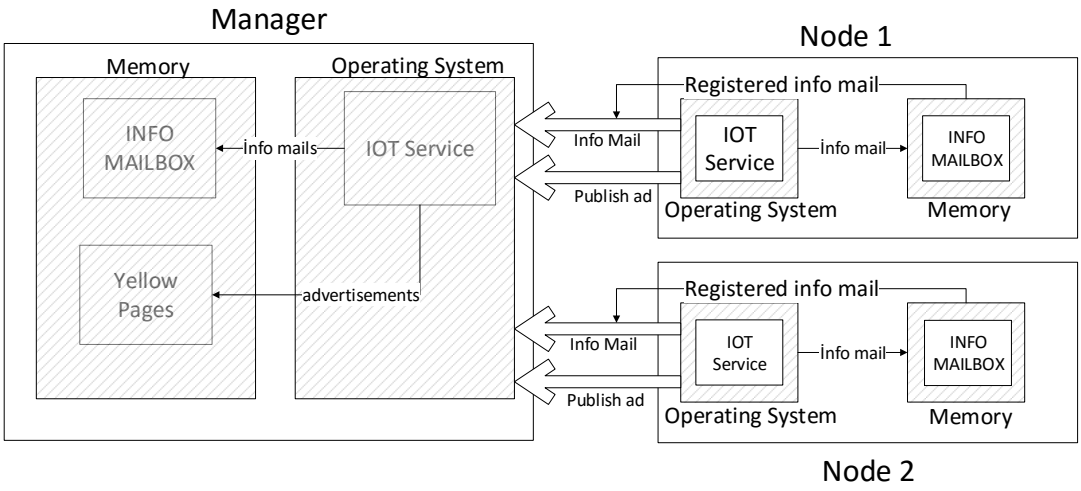

Figure 8.5: Structure of Mailing System and Yellow Pages

point of view as it was explained before. One of them is user point which can be thought as writing firmware without dealing with any communication library and protocol. The IOT service I is implemented for that purpose. This service handles with all wireless communication including handling with radio module callback. Call-back function for radio module is defined and implemented into this service. Radio library sends only radio state to higher-level callback. A buffer is implemented as well to write received data. According to radio state and device state, service extract relevant data from the message. The purpose of handling radio callback in this service is to not block other kernel services or OS exceptions or other hardware interrupts with wireless communication. Since this is an embedded operating system, it needs catch if there is a hard deadline to catch. In addition, the IOT service simply can be degraded by giving lower priority if user needs to execute more important task.

```
typedef enum{
    IOT_SERVICE_RESOURCE_TYPE_TIMER,
    IOT_SERVICE_RESOURCE_TYPE_LED,
    IOT_SERVICE_RESOURCE_TYPE_ACTUATOR,
    IOT_SERVICE_RESOURCE_TYPE_COMPUTATION,
    IOT_SERVICE_RESOURCE_SENSOR_TEMPERATURE,
    IOT_SERVICE_RESOURCE_SENSOR_HUMIDITY,
```

```
    IOT_SERVICE_RESOURCE_SENSOR_PRESSURE,
    IOT_SERVICE_RESOURCE_SENSOR_PROXIMITY,
}iot_service_resource_type_t;
```

Listing 8.1: IOT node resources

A node may have various type of resources. These type of resource is implemented as a special type called `iot_service_resource_type_t` and it can be seen at the listing 8.1. For different kind of applications, node would have different modules and might need different types of resources. For instance, a group of node inside of a cluster may have the task to measure temperature while other group measures humidity and assume that they have both sensor module mounted on the devices. Nodes use only one sensor at the time(humidity or temperature) to reduce power consumption. If one of the nodes has failure in its temperature sensor, it can publish an advertisement about the need of temperature sensor and manager can control and use the temperature sensor on the other node on behalf of the needed node.

```
typedef enum{
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_LED_ON,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_LED_OFF,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_LED_TOGGLE,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_ACTUATOR_ON,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_ACTUATOR_OFF,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_ACTUATOR_TOGGLE,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_SENSOR_READ,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_SENSOR_WRITE,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_TIMER_START,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_TIMER_STOP,
    IOT_SERVICE_RESOURCE_SLAVERY_COMMANDS_TIMER_SET,
}iot_service_resource_slavery_commands;
```

Listing 8.2: IOT node resources

There are many ways to control node by manager inside of a cluster. This command called slavery command in the particular protocol. Basically, every node opens radio module in receive

mode periodically to check whether there is any request from manager or not. At the figure 8.2, the type of slavery command can be seen. Sensors, timers, actuators and LEDs can be control easily by manager using these commands. More functionalities are considered and discussed in the future work section later on this paper.

IOT service task has also provides an API to user to use the IOT service. User can start and stop IOT service using this API. In addition to that, API tracks the number of the send packet and received packets. IOT service registers every node in the cluster inside of one list. API provides information about each node and that list to user as well. Moreover, user can interfere IOT manager task and can start slavery with the entire node in the cluster regardless their workload. However, it is not recommended to interfere most of the time, if application of the other node is not known well.

8.3 Security Issue

The security issue of large sensor networks and IOT devices have been becoming more important day by day. In addition to that, according to estimations each person in the world will have had six devices. Even now all around us is surrounded by personal devices, smart buildings, even cities and factories. We as developers use these devices to measure our heartbeat, temperature, location, blood pressure. We make everything accessible. Even our most sensitive and private data. For instance when you set your Wi-Fi-enabled devices, you add something to your local area network without proper security. That means you put a device after your security or firewall. That could be used as a key to your more private data even to your computer.

The risks are obvious as we discussed earlier in this paper. Rogue/Mock/Invalid firmwares, invisible backdoors, eavesdropping, having malware in upgrade. In addition, devices called botnets are already a major threat. Number of attack against routers, smart TVs, game consoles are increasing. Limited encryption capabilities of device one of the main issue. In addition, upgradable software is the key concept. It is hard to find backdoors or bugs without putting particular product to the field. Linux, Windows, Android and IOS all the other big OS they always fix their bugs and backdoors with patches.

Security should not be thought of as an add-on to a device, but rather as integral to the

device's reliable functioning. Software security controls need to be introduced at the operating system level, take advantage of the hardware security capabilities now entering the market, and extend up through the device stack to continuously maintain the trusted computing base. Building security in at the OS level takes the onus off device designers and developers to configure systems to mitigate threats and ensure their platforms are safe. Therefore, it can be easily seen that specific OS for IOT is needed. Thus, main purpose of this service and protocol to propose a new way of thinking to IOT and large sensor networks and security issue.

As it was explained before, in this particular protocol, each node registered to the cluster uses a different channel to communicate with the manager. The information about which node communicate in which channel is only stored in manager. Therefore even node is accessed by unauthorized user, there is no way to get the knowledge of the channel information regarding to every node. On the other hand, the node needs to keep creating new information mails periodically. Therefore, any missing or suspicious information coming from node side can be considered as threat and node can be banished from the cluster.

One of the component that also provide security in operating system level is communication layer. Communication layer provides abstraction between radio module and application. Therefore even unauthorized user somehow access to a node, it can only use radio module with limited functionality provided by API. As it was discussed before, it can only publish advertisement about needed resource and it needs to be approved by manager too. Thus, there is no way to access manager or radio module except this.

IOT service is defined as user tasks in the scheduler. Its only privilege than regular user tasks is having higher priority than all the other user task. Therefore IOT service can use some specific functionalities provided by kernel but cannot be run in kernel mode or privilege. Moreover, other functionalities of operating system, exception and interrupt handling has different priorities than scheduler's task. Thus, kernel and other components of particular operating system still keep working even IOT service controlled by unauthorized access.

In this particular network structure, manager has many rights on network and nodes. It can collect information, use their hardware and can assign task to do. On the other hand, node device type has very limited right or functionalities over the network and devices. A node can only access to the manager using assigned channel. Manager device decide to accept the node

to the cluster and registered it, or send rejection message to the node. Moreover, manager can assess all the collected information from nodes and can perform a health check of entire network or particular node. Since any suspicious act of node will degrade the network health point, manager can shut down the entire cluster or send emergency signal to another trusted device.

As it was mentioned before, a registration system is implemented in this network structure. After booting-up of particular node, it starts to send registration message in fixed frequency called registration frequency, which is 2.44GHz. It performs that task periodically to reduce the power consumption. Same functionality is also performed by manager device. It periodically checks whether there is new device try to register in registration frequency or not. If there is a node which is trying to register, it checks the protocol and extract the message. After the protocol check, manager registers and stores the information about particular node and send an acknowledge message about registration process. If it is successful , particular node stops searching new manager, starts performing other user tasks while creating information messages periodically. This registration process make registration of mock/invalid or unauthorized device harder. Moreover, any suspicious act and timeout of node is being observed by manager and these kind of acts can be considered as invalidation of particular node, thus, it can be banished from cluster. The flow chart of the registration process for manager and node side can be seen at the figure 8.6.

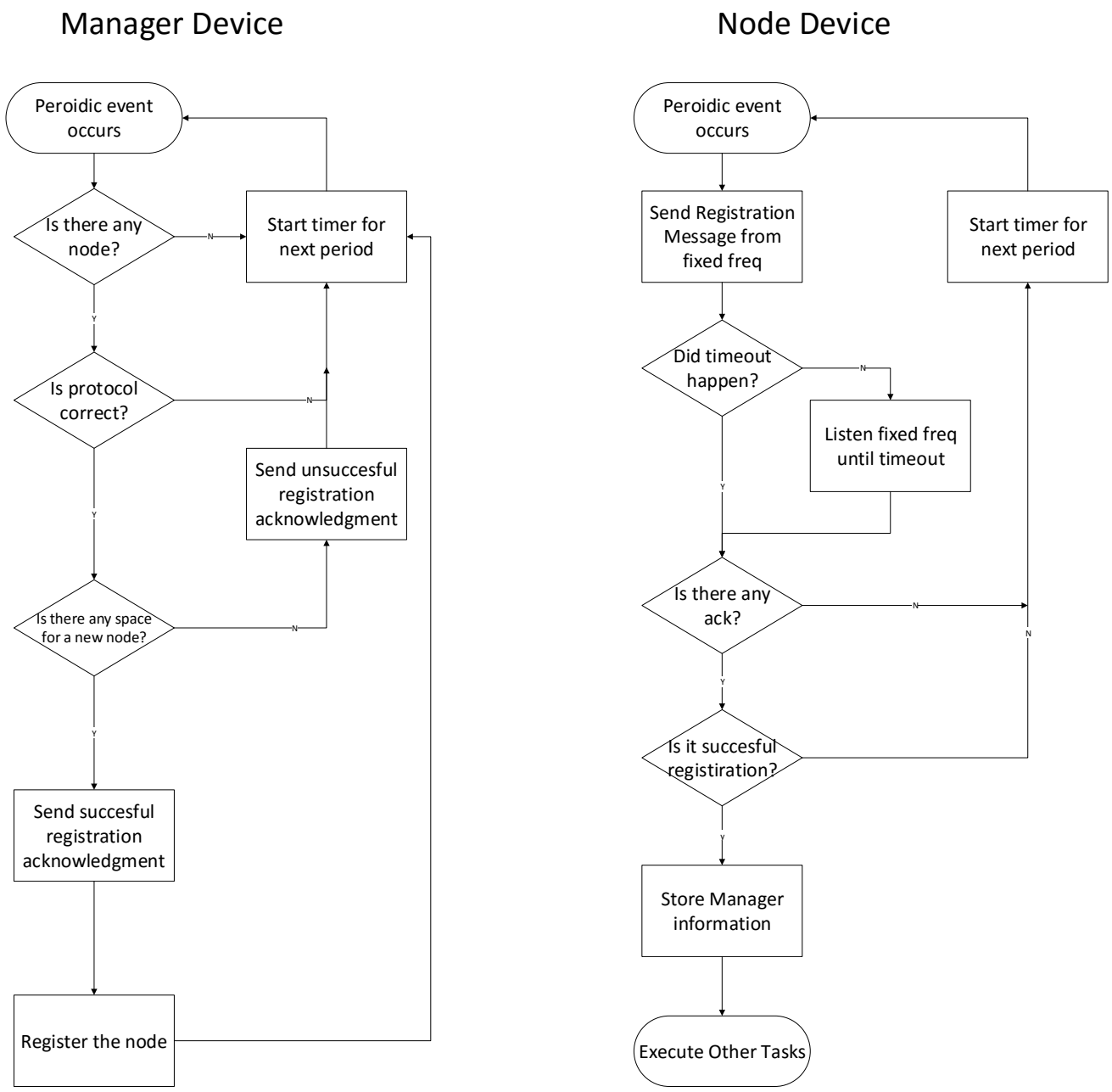

Figure 8.6: Flow Chart of Registration Process

Chapter 9

Tests and Results

After the every software development, exhausting tests must be done to validate the functionality of the system. Verification and validation is the process of checking a software system meets specification and perform expected functionality. Software verification is ensuring that the product has been built according to the requirements and design specifications. On the other hand, verification in software development checks that functionally of devices whether as it is expected or not.

In addition, while testing a particular software of software related product, terms should be used in order to explain the type of the problem. Therefore, from test point of view, fault means wrong or missing function in the code. The manifestation of a fault during execution is called failure and malfunction of software or device can be described as according to its specification the system does not meet its specified functionality. In this particular section, tests and test scenarios for particular operating system are explained in detail. In addition to that, results, which were obtaining after the tests are presented.

In order to test the operating system in real world, embedded devices (a circuit-board or development kit) were needed to implement particular project in it. For that purpose, two types of devices have been used provided by vendor company Nordic Semiconductor. As it can be seen from figure 9.1, these are NRF518222 Development Kit and NRF51822 Dongle.

The selected SOC NRF51822 has two type of development or evaluation boards provided by vendor. One of them, which is used in the test as node device, is nRF51 Dongle. It is a low-cost, versatile USB development dongle for Bluetooth Smart, ANT and 2.4GHz proprietary applica-

Figure 9.1: NRF518222 Development Kit and NRF51822 Dongle

tions. The kit provides I/O interface with 6 solder pads and has programmable RGB LED and programmable button for user applications. The other development kit that is used in this particular project is NRF51DK. It is used as manager node during the tests and it is a low-cost, versatile single-board development kit for Bluetooth Smart, ANT and 2.4GHz proprietary applications. The kit provides I/O interface via connectors and has 4 programmable LEDs and 4 programmable button for user applications. Both of the devices provide Program/Debug options with Segger J-Link OB for standard tool-chain.

Basic functionality of operating system has been tested and verified by using NRF51822DK device. After achieving the satisfactory result from kernel functions and hardware drivers, IOT service layer and radio infrastructure had been started to develop. In order to test radio library and functionalities, a system formed by a NRF51822 DK board and a NRF51822 board, was created and ping-pong test was done between two devices. Afterwards, a test has been formed by same devices but this time with IOT service layer and new radio protocol implemented. After we ensured that IOT service layer performs communication between devices and all the functionalities related with communication is working, resource and information sharing part of the layer was started to develop.

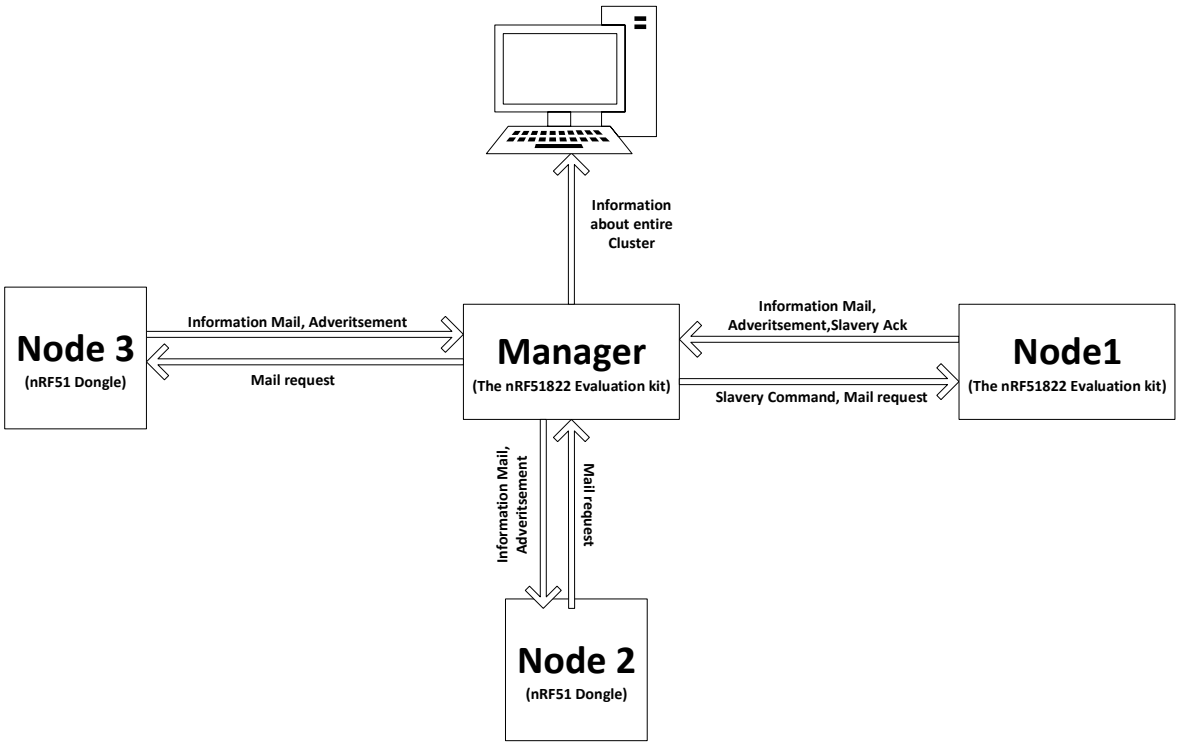

Figure 9.2: Structure of Test Scenario

As it is illustrated in figure 9.2, this cluster was formed to test all the system functionality and timing specification for IOT service layer. In this test scenario[1] , a user program has been developed in the manager device, to communicate with personal computers and send the information sharing inside of the cluster. In addition, same program also was controlling Node1 LEDs by using IOT service API. Computational tasks were implemented all the devices in the cluster including manager to see the effect of multitasking over IOT service layer. In this test all the nodes issued mails in the memory and manager collected those mail periodically and information about each node was seen on from personal computer serial terminal. After the tests, it has seen that all the functionalities are implemented such as resource and information sharing, task execution, hard-drivers etc. In addition to that, size of the code and execution time of particular IOT service layer are obtain as a results of the test. Particular results are shown in table 9.1 and 9.2 .

[1] The example cluster which is formed by three nodes and one manager, is also submitted with this thesis

	Memory Partition		
Device Type	FLASH SIZE	RAM SIZE	IOT SERVICE LAYER SIZE
Node Device	6.43kB	7.58kB	1.375kB
Manager Device	6.95kB	8.24kB	1.785kB

Table 9.1: Size of the Embedded Operating System

	Communication Type	
Device Type	1-1 Communication	Cluster (4 Devices)
Node Device	207.5ms	210ms
Manager Device	267.5ms	866ms

Table 9.2: Execution Time of IOT Service Layer

Chapter 10

Discussion, Future Work, and Conclusion

10.1 Discussion

The main motivation of the thesis was to design a simple kernel and a small embedded operating system particularly for low-power resource-constrained devices with the additional layer called IOT layer and make development easier for the trending concept. To achieve this, some research had to be done. However, before designing such a system, requirements fo the system should have been decided. Hence, detailed investigation about different embedded operating systems and distributed system architectures was done. Since the competition in the embedded OS market fierce, proposed architecture should introduce new features to that area. Main focus while designing the operating system was to keep the simplicity of the design as much as possible while providing different functionalities to users with the operating system.

On the other hand, the footprint of the OS was also the other main concern. Therefore, creating own kernel architecture was seemed quite promising. This task was indeed time consuming but eventually a “microscopic” kernel was designed which satisfies our needs and requirements. Moreover, a very simple and new protocol for wireless communication is desired to be implemented to enable wireless communication and information sharing. This protocol is implemented together with the IOT service layer which is the novelty of the project. Relevant tests about the functionality and performance have been done and the results are shown earlier in this thesis.

As it was mentioned before, this operating system is designed specifically for the low cost,

low power and resource-constraint embedded devices. Currently in the market many different embedded devices and CPU architectures can be found. Especially single-board computers started to dominate the market. This device type is a complete computer built on single board with input/outputs, memory and microprocessor. Moreover, single-board computers such as raspberry pi or beagleboard often have features like graphical-user interface, wireless shields to enable wireless communication and other popular communication protocols such as USB, HDMI, etc. Mostly they are used for every kind of applications and educational purposes. Moreover, most of them contains large processor with lots of features and large memories.

During the development of this particular project using these single-board computers would be easier and faster. However, these devices are not good fit for sensor nodes or low-power IOT devices. Firstly, these devices consumes more power than small microcontrollers since they have many module in it, thus it is not convenient to use those devices with battery and most of the IOT and sensor network application requires low-power consumption and mobility. Moreover, during our research we have seen that, most of the challenges in those devices are going around industry and user needs. Therefore, we could not find academic challenges which motivate us to research on single-board computers.

In contrast, resource-constrained devices have many challenges to overcome since it is always hard to perform complicated tasks with resource-constrained devices. In addition to this, with the growing IOT and sensor network concept, many new problems and challenges have appeared such as security, efficiency, power consumption and functionality.

In this project, we particularly focused on the task execution and scheduling in the embedded operating system. Our target architecture for development was low-power single core processors especially ARM Cortex-M series and as it was explained before a cortex M0 architecture has been used as a target during the development. The purposed kernel architecture can support multitasking. It can switch between task and run them according to their priority.

However, using memory management units and algorithms, multi-thread behavior could have been achieved. With this method two task can be run concurrently. However, this method would cause performance problems in entire system .Moreover, it might affect the execution of real-time task. Since particular operating system is designed for embedded system and devices, real-time compatibility is an important feature.

On the other hand, a true concurrent kernel could have been developed and implemented on the multi-core processor architecture. However, multi-core systems are completely different topic. Developing a kernel for multicore processor would be very time consuming. In addition to this, developing such an operating system has its own academic and technical challenges. Therefore, the project related with a multicore processor would not be related with our main focus which is a small and efficient operating system for sensor and IOT device network.

In order to achieve small footprint for the operating system in this project, optimization must be done in the most of the operating system components. Usually, one of the largest part of the operating system is OS services. Despite their large size, OS services are the component which makes operating systems out of piece of software and kernel functions. Not only commercial operating systems for personal computer such as Windows, Linux or Macintosh but also in embedded OSes such as FreeRTOS, TinyOS, Contiki. There are many OS services running in the background while the user application is running.

The number and types of OS servces in embedded operating system are very controversial topics. During our research, we have seen that most of the embedded devices uses only task scheduling and communication drivers in sensor networks and IOT devices. Therefore, it has been decided that particular operating system would provide task scheduling/execution and communication drivers as an operating system services. All the other services that can be needed, depends on application such as file management can be added by the user. In addition to that, this operating system has been designed to provide an infrastructure to enable information and resource sharing across the sensor or IOT device network. Because of its user-friendly design, system can be extended easily from user layer for more demanding applications.

The kernel is the most important part of the operating system and can be imagined as the heart of every operating system. The kernel design has been being very popular topic both in the industry and academy. Different operating systems use different kernels. Monolithic, micro, hybrid, nano and exo kernels are the widely used ones in the industrial and academic projects. Moreover, there was a big debate called Tanenbaum-Torvald debate about microkernel and monolithic kernel architecture.

All these kernel types have different advantages, challenges and disadvantages. However, a strict structure like monolith or microkernel is not a suitable architecture for embedded appli-

cations. These structures are very good in a system with powerful CPU and many resources to manage. Nevertheless, most of the non-constrained embedded systems in sensor and IOT networks have low profile CPU and limited resource to manage. Therefore, to implement such an architecture in this project, would be definitely in vain.

On the other hand, efficiency of the task execution appeared to be more important than number of services provided by OS according to background research. In addition to hat, we believe that a new kernel design is needed for the resource-constrained device in the sensor and IOT device networks. Thus, a new kernel architecture has been proposed and implemented in this thesis.

A priority driven scheduler is implemented inside of proposed kernel architecture. This type of schedulers are easy to implement. The scheduler does not need to know the information on the release times and execution times of the jobs. In addition, the run-time overhead due to maintaining a priority queue of ready jobs can be made small. The context switching does not occur often in this type of systems thus also overhead caused by context switching does not cause any problem. The priority-driven algorithms are on-line scheduling algorithms. The scheduler makes decision without having information about jobs released in the future. Moreover, on-line scheduling is very suitable option for a system which future workload is unpredictable such as embedded systems.

In contrast, priority-driven schedulers are non-deterministic, thus the timing behavior of the system is not predictable. In addition to that, it is difficult to validate that all jobs scheduled in a priority-driven manner meet their deadlines when the jobs parameters vary. There are better scheduling algorithms for real-time systems such as earliest-deadline first, value-based scheduling or deadline monotonic priority algorithms. Moreover, priority-ceiling algorithms could have been implemented with dynamic scheduler to avoid priority inversion and deadlocks during the execution of the real-time task. However, operating system needs to control the tasks all the time and know everything about the tasks in these algorithms. That means a bigger task library thereby a bigger kernel and bigger footprint of the operating system. Additionally, this would be completely opposite of the main idea of the particular project that is simplicity.

After research on the sensor networks and the IOT concept, it is revealed that intelligent

networks for these kind of networks could be the solution to overcome the challenges. In order to achieve that purpose, each network should have conscious to investigate what is going on the network and act accordingly. This could be possible with information and resource sharing.

As it was discussed before, these kind of networks are full of resource-constrained devices. Since achieving information sharing across network would increase the communication rate per device per second, this would lead increasing of power consumption. However, one of our main motivation for this project was a specifically designed operating system and OS service. This could be successfully enable resource sharing and keep the power consumption as low as possible.

However, during the development, we have seen that not only the OS and OS services but also hardware drivers and communication infrastructure should be designed specifically. To implement communication infrastructure for particular IOT service, Bluetooth protocol could have been used. It is a compatible, low-power protocol and all the libraries are provided by vendors. However, the Bluetooth stack is too large to implement on such an operating system. Bluetooth stack must be optimized or reduced. Even it would not have had resource-sharing feature. Therefore, although it is indeed time consuming to develop a specific protocol, a library and a service for this purpose, a simple structure was purposed and implemented into the operating system in this thesis.

While developing the project, some decision must have been made to increase development efficiency. Since this project includes complete development of an operating system from scratch including device drivers, software platform and the compiler that are used in the development are very important. After deep investigation about development environments for targeted platform, in this particular project, Keil uVision5 software development platform has been decided to use. This environment specifically developed for wide range of Cortex-M, and Cortex-R based microcontroller devices. In addition, it provides the µVision IDE/Debugger, ARM C/C++ Compiler, and essential middleware components.

As a compiler, Keil provides a complete tool chain called the ARM compiler toolchain. This includes following components:

- The ARM C/C++ Compiler (armcc)

- Microlib
- The ARM Macro Assembler (armasm)
- The ARM Linker (armLink)
- ARM Utilities (Librarian and FromELF)

With the help of these tools developers can easily write applications for the ARM family microcontrollers in C or C++. Moreover, after compilation this tools ensures to provide speed and efficient of assembly language. The tool chain translate C/C++ files into relocatable object modules which contain full symbolic information for debugging with the µVision Debugger or an in-circuit emulator. Further more the compiler generates a listing file which contains symbol table and cross-reference information.

These development tools for the ARM family of microcontrollers allow you to write ARM applications in C or C++ that, once compiled, have the efficiency and speed of assembly language.The ARM Compiler toolchain translates C/C++ source files into relocatable object modules which contain full symbolic information for debugging with the µVision Debugger or an in-circuit emulator. In addition to the object file, the compiler generates a listing file which may optionally include symbol table and cross-reference information.

As it was discussed before cortex-M0 processor architecture has been decided as targeted architecture. It is because this particular architecture is a very low gate count, highly energy efficient processor that is intended for microcontroller and deeply embedded applications that require an area optimized processor. In addition to that its power consumption is lower than other competitor architectures.

The configurable, multistage, 32-bit RISC processor ARM Cortex-M0 processor consumes small silicon area and low power and also minimal code footprint. Due to all these, developers can achieve higher performance than old 8-bit micro controllers. Therefore, it is also very suitable architecture for embedded system learners and developers in academy. In addition to that, this particular project is also based on wireless communication and designing a new infrastructure for sensor networks and IOT devices. Therefore, this particular architecture was the most suitable architecture for this project since it is specifically optimized connectivity to support low power protocols such as as Bluetooth Low Energy (BLE), IEEE 802.15 and Z-wave.

Detailed investigation has been done to choose most suitable microcontroller/SOC and its development kit to implement the software and test it. NRF51 series from Nordic Semiconductor is the most suitable for this project in price and performance manner. NRF51 is a system-on-chip with with a Cortex M0 and a BLE radio chip all in one. With provided libraries it is easy to learn and develop software. NRF51822 has been selected among NRF51 series. The particular SOC has two type of development boards provided by vendor. One of them, which is used in the test as node device, is nRF51 Dongle. The other development kit that is used in this particular project is NRF51DK and it is used as manager device.

In this particular project a kernel designed has been proposed for embedded devices specifically sensor and IOT networks. The kernel architecture is implemented together with a new radio protocol and library into an embedded operating system which has specific IOT service. All the software which was develop during the project is open source and anyone from industry or academy can access and use it easily. In addition, this particular paper would lead the way those who would use this embedded operating system. As it was explained before purpose of proposed kernel architecture is to create user-friendly environment. Moreover that this particular operating system is fully functional currently. A group of node has been controlled by manager device and a cluster has been created as it was mentioned in “test and results” section.

On the other hand, the particular operating system is open to develop further by those who would like to continue this project in the academy. Although, future works is explained in detail in the next section, there are many other ideas to extend this project. This system is not only introduced as user-friendly but also it is developer friendly because of its two dimensional architecture. Furthermore, this software can be turned out to be an industrial or commercial product with few changes. Although it is a master project and developed with an amateur spirit, after necessary tests, validation and improvements it would be a competitive product in the market.

10.2 Future Work

The software development in this project was done from scratch. It was tested and the result of it is ensured that the system functionality works as it was expected. This project also proved that our concept ideas on sensor and IOT network could be implemented and used in embedded

devices. However, during the background research and development of the project, new ideas have appeared about the project and further development of the project. Since, the time was limited while developing this project, this ideas could not be implemented into this project. Therefore in this section, recommendations for the possible extension of the project are given. These recommendations can be classified as short, medium and long-term future work.

For short term;

- Although this particular operating system designed to achieve low-power consumption, no power modes is implemented for different type of application. Therefore different power modes with different specification can be added and the power management can be optimized in future works.

- One of the concern while developing IOT layer and communication infrastructure that lies behind it, was security. In architectural design of this particular layer and abstraction of communication library helps user to avoid security issues. In addition to that, most of the microcontrollers have AES encryption modules. Communication between nodes and manager can be encrypted by using this modules and more secure platform can be developed.

- Although, many hardware libraries were developed during the project, some of the libraries, which is commonly used in embedded application, could not be implemented due to limited time. Libraries such as flash memory, SPI, I2C can be developed and integrated easily.

- This system and communication infrastructure is specifically developed for sensor and IOT device networks. Particular network infrastructure can be configured to use specific application such as server application, distributed system applications or robotics applications.

For medium term;

- As it was explained before, manager node contains all the information about the cluster and control resource of nodes as its own resource. For end user a user interface can developed for Windows or Linux via USB port since all of the personal computer has USB

ports and communication between PC and manager node is already tested by using UART library with converter module from UART to USB.

- Information and resource sharing features are implemented and tested in this project. However, assign task sharing and using other nodes computational power could not be implemented due to limited time.

- None of the memory management concepts such as paging or caching were not desired to developed to not lose the focus of the project. However, for high profile CPUs, memory management support can be implemented to increase the total system performance.

- At the beginning of the project, security layer between IOT service layer and application layer was thought to be one of the feature of the embedded operating system. However, it is cancelled to implement after the careful consideration about workload and time. A registration and validation system can be implemented for user application to user IOT service. With that kind of system, unauthorized accessed node can isolated from the cluster and possible attack can be prevented.

- In addition, a tool for yellow pages can implemented inside of manager node which can work as database. This data can be compressed while storing in database and this would lead smaller footprint as well as having more functionality on yellow pages.

For long term;

- This particular project is designed to implement many different processor architectures. For that purpose, device specific components such as hard drivers and portable kernel functions have been abstracted. Therefore, to provide support for different processor architecture would be easy. However, since there are many different processor architectures in the embedded system market, that task indeed would be time consuming.

- Producing multicore processors with low power consumption feature is trending concept in integrated circuit industry. Despite the technical challenges of multicore system, particular kernel can be extended to support these kind of architectures.

- During the background research, problems and challenges in the networks caused by number of device and capabilities of devices have been revealed as most difficult challenges and problems to overcome. However, we proposed earlier in this paper, these challenges and problems could be overcome implementing machine learning or AI algorithms to network infrastructure and service. In addition to that, since particular project achieves wireless communication thus, it could be used to implement those algorithms to sensor or IOT device networks.

- The particular operating system provides real-time compatibility for real-time applications. However, since a priority-driven scheduler is implemented into the kernel, it may degrade the performance or may cause some problems in hard real-time systems. In order to solve this problem, different scheduling algorithms can be implemented. Yet, task library needs to be extended to implement those algorithms.

- This particular operating system has been developed for engineers and developers. Therefore, end-user cannot use particular system without having any knowledge about computer science, operating systems and electronics. However, a graphical-user interface can be developed as it is in end-user operating systems. With a simple GUI, user could see the network elements of particular network and manipulate network structure and devices in the cluster.

10.3 Conclusion

The software development for embedded devices such as low-power and small microcontroller has been significant topic for last 10-15 years. Many research have been going on both in the academy and industry about efficiency and functionality of embedded devices. Therefore, many different operating system optimized specifically for embedded devices, have been developed. Together with wireless connectivity, embedded devices have started to appear everywhere in our daily life. Nowadays, it is an important concept for not only in the academy and industry but also in regular life. Therefore, embedded devices will have been important concept for next years. However, since wearable technologies and Bluetooth low energy devices have started to

be used in the most private and crucial applications, the importance of failures in these devices and networks have increased. On the other hand, the old technology has started to be thrown away such as old 8-bit microcontrollers because of the trending SOCs with wireless communication modules. These devices should be reused to prevent the world from being microelectronic garbage. Moreover, because of the funding problems, most of the universities in non-developed countries cannot afford embedded operating systems to teach the basics of computer science and electronics.

Although, the developments in IC technology and software tools has provided improved functionalities to the small microcontrollers, it also has started to cause new problems. Therefore, both in the academy and industry a new point of view for embedded operating systems is very important. In this thesis, we have proposed ideas and solutions and we have shown that these problems could be solved even in limited-time with limited labour. After the detailed research and dedicated software development, an embedded operating system was designed in the light of new ideas. Moreover, during this project, problems and challenges of the new types of networks which are formed by hundreds of sensor and IOT devices are revealed. In order to solve these problems, a radio library were specifically developed for this project. Moreover, a wireless communication protocol was proposed and implemented. Furthermore, to provide improved functionality such as information and resource sharing for elements of the networks, a new operating system service and layer called IOT service layer was developed and implemented.

In order to develop such an embedded operating system, some specific tasks must be defined and done. First task was to develop required hardware drivers for the targeted platforms. Although, the vendor company of the targeted platform provides libraries for the hardware, they must have been developed from scratch to have efficient, optimized and suitable hardware drivers for the operating system. Moreover, one of the important feature of the project was the proposed kernel architecture. This kernel is designed to achieve not only small size but also to be a real-time kernel. In addition, the OS services must be optimized and implemented on the operating system to make the operating system functional. However, the most difficult task in this project was developing the specific radio library, a protocol and a service, which depends on a new communication infrastructure. Detailed research about wireless communication and

distributed system has been done to achieve these tasks. Every software project must be tested and the proposed features and functionalities must be verified. Therefore, a test scenario has been created and the entire system features and functionalities has been tested by using development kits provided by the vendor.

This operating system has features and characteristics, which are different from other embedded operating systems in the market. The system has very small footprint. It can be implemented even on modern small microcontrollers in the market without need an external memory. Simplicity of the system provides users an efficient task execution. Moreover, the proposed two-dimensional kernel architecture gives users a simplified point of view for the development thus; it leads the system to be more user-friendly. Last but not least, the operating system and communication protocol specifically designed for the sensors and IOT device networks. It can be configured easily to use in such networks.

In this particular project, a new architecture for embedded operating systems has been proposed and a small embedded operating system with distributed system functionality has been designed and implemented. In addition to this, the operating system was implemented on different resource-constrained devices with same processor architecture and detailed tests about system have been done. Currently, all the features and requirements, which were mentioned in the introduction part of this thesis, have been met and the embedded operating system is ready to use. Moreover, during the development of the project, detailed background research were done about operating systems, distributed systems and wireless communication and explained in this thesis. Therefore, this paper can be considered as a resource in these topics as well as the user-manual of the embedded operating system. Furthermore, a conference paper about the operating system particularly IOT service layer is being written.

Lastly, the current developments in the technology are increasing the importance of the embedded systems and resource-constrained systems in our daily life. The application field of the embedded systems has been growing wider thus; the stored data and handling with system failures have been becoming more important as well. Therefore, the operating systems or network protocols, which are used in these applications, should be enhanced. More efficient and secure systems are needed. Since the vast wireless sensors and IOT device networks surround us all over the world, even the slight improvements in the operating systems or network architectures

will produce significant results.

Chapter 11

Bibliography

[1] ITU, "Internet of Things Global Standars Initative," [Online]. Available: http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx. [Accessed 1 April 2016].

[2] D. Gordon, "The Trojan Room Coffee Machine," University of Cambridge, [Online]. Available: http://www.cl.cam.ac.uk/coffee/coffee.html. [Accessed 1 April 2016].

[3] A. Wood, "theguardian," [Online]. Available: http://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must. [Accessed 1 April 2016].

[4] J. Apcar, "IP Routing in Smart Object Networks and the Internet of Things," Melbourne , 2012.

[5] UN Economic and Social Affairs, "World Population to 2300," Cisco IBSG projections, 2004.

[6] P. Bennett, "EE Times," [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1276973. [Accessed 2 April 2016].

[7] H. Mario, T. Pentek and O. Boris, "Design Principles for Industry 4.0 Scenarious: A literature Review," Technische Universität Dortmund, 2015.

[8] Z. Technologies, "Variable global enterprise Internet of Things decision makers," 2014.

[9] S. P. M, "Embedded Operating Systems for Real-Time Applications," M.Tech. credit seminar report, Bombay, 2002.

[10] A. S. Tanenbaum, Modern Operating Systems, Amsterdam: Pearson Education International, 2009.

[11] I. C. Bertolotti, "Real-Time Embedded Operating Systems: Standards and Perspectives," in Embedded Systems Handbook, Taylor and Francis Group, LLC, pp. 4-10.

[12] BrokenThorn Entertainment„ "Operating Systems Development - Kernel: Basic Concepts Part 2," BrokenThorn Entertainment, [Online]. Available: http://www.brokenthorn.com/Resources/OSDev13.html. [Accessed 06 05 2016].

[13] S. Poudel, ""Operating System Structure"Science HQ," Ed. Rod Pierce, 2013 February 18. [Online]. Available: http://www.brokenthorn.com/Resources/OSDev13.html. [Accessed 6 May 2016].

[14] A. Holt and H. Chi-Yu, Embedded Operating Systems, Bristol: Springer, 2014.

[15] G. Coulouris, Distributed Systems Concept and Design, Boston: Pearson, 2012.

[16] J. WEIJIA, Distributed Network Systems from Concepts to Implementations, Boston: Springer, 2005.

[17] D. Kinneryd and A. Mäkitalo, "Distributed applications: A journey to platform independency," Luleå University of Technology, Luleå , 2007.

[18] tutorialspoint.com, "Tutorials Point," tutorialspoint.com, [Online]. Available: www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm. [Accessed 8 May 2016].

[19] Micro Focus, "Networking Primer," Novell, [Online]. Available: https://www.novell.com/info/primer/prim05.html. [Accessed 8 May 2016].

[20] H. Karl and W. Andreas, Protocols and Architectures for wireless sensor networks, Chichester: John Wiley and Sons, 2005.

[21] Magna Design Net, "Digital Modulation," Magna Design Net, 2014. [Online]. Available: http://www.magnadesignnet.com/en/booth/technote/ofdm/page2.php. [Accessed 8 May 2016].

[22] Maxim Integrated, "An Introduction to Spread-Spectrum Communications," Maxim Integrated, 18 February 2003. [Online]. Available: https://www.maximintegrated.com/en/app-notes/index.mvp/id/1890. [Accessed 2016 May 8].

[23] RS Components, "11 Internet of Things (IoT) Protocols You Need to Know About," RS Components, 2015. [Online]. Available: http://www.rs-online.com/designspark/electronics/knowledge-item/eleven-internet-of-things-iot-protocols-you-need-to-know-about. [Accessed 2016 May 8].

[24] C. Links, "Wireless Communication Standards for the Internet of Things," Green Peak Technologies, 2015.

[25] C. Svec, "The Architecture f Open Source Applications Volume II," Creative Commons: Attribution, 2012.

[26] S. Kolesnik, "Comparing microcontroller real-time operating systems," embedded.com, 2013.

[27] A. Dunkels, "Full TCP/IP for 8-Bit Architectures," in The First International Conference on Mobile Systems, Applications, and Services, San Francisco, 2003.

[28] A. Dunkels, O. Schimdt, T. Voigt and A. Muneeb, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in SenSys'06, Boulder, 2006.

[29] B. Ganesh, "Architectural Support for Embedded Operating Systems," University of Maryland,, Maryland, 2002.

[30] H. Galzner and M. Hartmann, "TinyOS, an Embedded Operating," Faculty of Informatics, TU Wien, Wien, 2012.

[31] V. R. Aroca and G. Caurin, " A Real Time Operating Systems (RTOS) Comparison," University of Sau Paulo, Sao Paulo, 2009.

[32] C. L. Wang, B. Yao, Y. Yang and Z. Zhu, "A Survey of Embedded Operating System," University of California, San Diego, 2001.

[33] Nordic Semiconductor, "nRF51822," Nordic Semiconductor, [Online]. Available: https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRF51822. [Accessed 10 May 2016].

[34] Nordic Semiconductor, "nRF51822 Product Specification v3.1," Nordic Semiconductor, Trondheim, 2014.

[35] Nordic Semiconductor, "nRF51 Series Reference Manual," Nordic Semiconductor, Trondheim, 2014.

[36] ARM Ltd., "Cortex-M0 Processor," ARM Ltd., 2012. [Online]. [Accessed 10 May 2016].

[37] J. Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, Elsevier, 2015.

[38] S. Rollins, "Linked Lists," University of San Francisco, 01 October 2007. [Online]. Available: http://www.cs.usfca.edu/ srollins/courses/cs112-f08/web/notes/linkedlists.html. [Accessed 2016 May 15].

[39] R. J. Sutcliffe, Modula-2: Abstractions for Data and Programming Structures, Arjay Enterprises, 2005.

[40] W. Ulrich, "Application Package Software: The Promise Vs. Reality," Business Technology and Digital Transformation Strategies , 31 August 2006.

[41] Nordic Semiconductor, "S110 nRF51 Bluetooth® low energy Peripheral SoftDevice," Nordic Semiconductor ASA, Trondheim, 2014.

[42] R. Perlman, "network design folklore," Interconnections: Bridges, Routers, Switches, and Internetworking Protocols, vol. 18, 1999.

[43] Comer , "They (protocols) are to communication what programming languages are to computation," The Need For Multiple Protocols, vol. XI, no. 2, p. 177, 2000.

[44] Y. Fan, C. lvin, L. Songwu and Z. Lixia, "A Scalable Solution to Minimum Cost Forwarding in Large Sensor," UCLA Computer Science Department, Los Angeles, 2001.

[45] Y. Jennifer, M. Biswanath and G. Dipak, "Wireless sensor network survey," Elvesier:Computer Networks, no. 52, p. 2292–2330, 2008.