T.C.

ISTANBUL SABAHATTIN ZAIM UNIVERSITY

INSTITUTE OF SCIENCE AND TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

DESIGNING A FRAMEWORK FOR WSN AND IOT BASED APPLICATIONS BY STATE MACHINES

MASTER OF SCIENCE THESIS

SAJJAD NEMATZADEH MIANDOAB

Istanbul

January - 2019

T.C.

ISTANBUL SABAHATTIN ZAIM UNIVERSITY

INSTITUTE OF SCIENCE AND TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

DESIGNING A FRAMEWORK FOR WSN AND IOT BASED APPLICATIONS BY STATE MACHINES

MASTER OF SCIENCE THESIS

SAJJAD NEMATZADEH MIANDOAB

Thesis Advisor

Asst. Prof.Dr. FARZAD KIANI

Istanbul

January - 2019

Fen Bilimleri Enstitüsü Müdürlüğüne,

Bu çalışma, jürimiz tarafından Bilgisayar Mühendisliği Anabilim Dalı, Bilgisayar Bilimi ve Mühendisliği (%30 İngilizce) Bilim Dalında YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

Danışman	Dr. Öğr. Üyesi Farzad KIANI	.....
Üye	Dr. Öğr. Üyesi Aydın Tarık ZENGİN	.....
Üye	Doç. Dr. Tahir Çetin AKINCI	.....

Onay

Yukarıdaki imzaların, adı geçen öğretim üyelerine ait olduğunu onaylarım.

.....

Prof. Dr. Ahmet Korhan BİNARK

Enstitü Müdürü

BİLİMSEL ETİK BİLDİRİMİ

Yüksek lisans tezi olarak hazırladığım "**DESIGNING A FRAMEWORK FOR WSN AND IOT BASED APPLICATIONS BY STATE MACHINES**" adlı çalışmanın öneri aşamasından sonuçlandığı aşamaya kadar geçen süreçte bilimsel etiğe ve akademik kurallara özenle uyduğumu, tez içindeki tüm bilgileri bilimsel ahlak ve gelenek çerçevesinde elde ettiğimi, tez yazım kurallarına uygun olarak hazırladığımı, bu çalışmamda doğrudan veya dolaylı olarak yaptığım her alıntıya kaynak gösterdiğimi ve yararlandığım eserlerin kaynakçada gösterilenlerden oluştuğunu beyan ederim.

Sajjad Nematzadeh Miandoab

ÖNSÖZ

Araştırmamdaki her aşamada bana yardımcı olan değerli tez danışmanım Dr. Öğr. Üyesi Farzad KIANI'ya, eğitim alanında dersleriyle bize vizyon katan çok değerli eşim Mahsa TORKAMANIAN AFSHAR ve aileme teşekkürlerimi sunarım.

Sajjad Nematzadeh Miandoab

İstanbul - 2019

ÖZET

DURUM MAKİNELERİ İLE KAA VE IOT TEMELLİ UYGULAMALAR İÇİN BİR FRAMEWORK TASARIMI

Sajjad NEMATZADEH MIANDOAB

Yüksek Lisans, Bilgisayar Bilimi ve Mühendisliği

Tez danışmanı: Dr. Öğr. Üyesi. Farzad Kiani

Ocak-2019, 142 Sayfa

Bu tez, durum algılayıcıya bağlı olarak Kablosuz Algılayıcı Ağlar (KAA) ve Nesnelerin İnterneti (IoT) aygıtları gibi özerk sistemler için mantık birimini sağlayan bir çerçeve (Framework) tanımlamaktadır. Önerilen bu çerçeveye SMORITHM ismi verilmiştir. Kablosuz algılayıcı ağlar ve nesnelerin interneti alanları günlük yaşamımızda modern teknolojiler olarak çeşitli uygulamalarda yaygın olarak kullanılmaktadır. Ayrıca, kararları kendileri tarafından alabildikleri için bir tür ad-hoc ve akıllı sistemler olarak da adlandırılabilirler. Makine durumu kavramına dayanan önerilen çerçeve, çeşitli uygulama alanları için çok faydalı olabilir ve geliştiriciler için kolaylıklar sağlayabilir. Ayrıca, çevresel olayları algılayabilen ve parametrelerin değerine ve önceden belirlenmiş hedeflere bağlı olarak uygun eylemleri gerçekleştirebilecek çeşitli cihazlardan oluşan bu sistemler için de uygun bir çözüm olabilir. Diğer yandan, aynı mantık biriminin "durum makine katmanı" ile "donanım katmanından" ayrılması mümkün olup farklı platformlarda kullanılması muhtemeldir ve ilgili bir exporter, hedef bir platform için gereken çıktıyı üretir. Bu tezde exporterlerin bir örneği olarak Arduino mikro-denetleyicisi için bir C++ kod üreten bir çerçeve geliştirilmiştir. Geliştiriciler ise, uzun süreli uygulamalar için yüksek okunabilirlik ve yüksek performans verimliliği sağlayan mantık birimleri sağlayan bu çerçeveyi kullanabileceklerdir.

Anahtar Kelimeler: Akıllı Sistemler, Durum Makineleri, Ad-hoc Ağlar, Kablosuz Algılayıcı Ağlar, Nesnelerin İnterneti, Arduino, Akıllı Makineler, Çerçeve.

ABSTRACT

DESIGNING A FRAMEWORK FOR WSN AND IOT BASED APPLICATIONS BY STATE MACHINES

Sajjad Nematzadeh Miandoab

Master of Science, Computer Science and Engineering

Supervisor: Asst. Prof. Dr. Farzad Kiani

January 2019, 142 Pages

This thesis represents a framework, which is called SMORITHM, to provide the logic unit for Wireless sensor networks (WSN) and Internet of Things (IoT) devices based on the state machine. WSNs and IoT are widely used in several aspects of modern technologies in our daily life. They consist of various devices that are capable to sense the environmental phenomena and perform appropriate actions depending on the value of parameters and preset goals. They are a kind of autonomous and intelligent systems because decisions are made by themselves. The proposed framework, which is based on machine state concept, can be very useful for various application areas and can provide facilities for developers. It may also be a suitable solution for these systems, which can detect environmental events and consist of various devices that can perform appropriate actions depending on the value of the parameters and the predetermined phenomena. Moreover, the same logic unit is probable to be used on different platforms which is possible by separated “state machine layer” from “hardware layer”; a related exporter generates required output for a target platform. A C++ code generator for Arduino microcontroller has been implemented as an instance of the exporter in this thesis. Via this framework, developers are able to provide logic units with high readability and high-performance efficiency for long term applications.

Keywords: Smart Systems, State Machine, Ad-hoc Network, Wireless Sensor Networks, Internet of Things, Arduino, Intelligent Machines, and Framework.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Highlights

- Goals of the thesis
- Summary of the proposed framework
- Road map of the thesis

This thesis is about creating a framework to manage Wireless Sensor Networks (WSNs) using state machine concept. Furthermore, as the concept of WSN applications is similar to other autonomous system types such as the Internet of Things (IoT), Robotics and various ad hoc systems, this framework can be used for them too.

Considering there are several types of applications, each type of them may have its own behavior depending on requirements. Implementing these applications are not too easy. On the other hand, the WSN and similar systems have specific characteristics (such as dynamic topology, self-configuration and managing and etc.), so creating a proper infrastructure for different applications is more difficult. In the literature has been proposed many solutions, one of them is logic unit based methods.

These behaviors are controlled by a unit that called logic unit. Typically, implementing logic unit for these applications are important; But because of their complex natures, designing a well-working logic is difficult. Despite most of these systems that have similar logic units, developers have to design and implement totally different structures depending on target platforms.

The proposed framework in this thesis suggests a solution to design a logic unit for WSNs and IoT devices more convenient and higher readability than common programming develops. In addition, reusing same logic units for different platforms is possible by separating the logic layer from the hardware layer. Thanks to this framework, developers of various systems will be able to perform whatever application they want, regardless of sensor nodes and other parameters. This framework, which is named “SMORITHM”, has been developed on state machine concepts to design intelligence systems for WSN and IoT.

All sequential logics have the capability to be designed with state machines. State machines have excellent properties to design algorithms in an easy and efficient way. They have widely used the background designing of in automaton machines. This framework handles the logic unit of them directly by presenting state diagram designer tools and generates outputs automatically based on diagrams. Generally, these systems are composited from different sensor nodes. Each sensor node includes various components, for example in a WSN, nodes have four main

components: the sensing unit, the processing unit, the transmission unit, and the power supply unit. Sensor nodes are a kind of autonomous systems that make decisions by themselves. The successes of a network are performed by appropriate action of its nodes. In other words, correct and efficient behavior of nodes can guarantee the achieving expected goals of a network. State machines are appeared more optimized and consume less processing resources than linear and sequential approaches. More details are mentioned in chapter 5.

Several goals and reasons motivated us to propose this thesis with its related framework. Realizing state machine and visual scripting at the same time is the primary goal of this framework (SMORITHM). It is defined to reduce the illegibility of software that is used in devices, to increase the easing of transferring experiences between members of develop-team and to create an Integrated Development Environment (IDE) to present a logic development environment for non-expert developers in programming. In addition, extending the current application or reusing previous modules are provided in a better way. The other goal is distinguishing logic layer from the hardware layer. Beside of performing administrative instructions and obligations, sensor nodes act based on streams of environmental events, so many of their reactions should be real-time and they have to react immediately and same as each other. The SMORITHM provides a unique behavior in both single and multiprocessors. Moreover, especially in heterogeneous structures, different types of nodes have to collaborate with each other. Proposing a standard structure for communicating them is the other goal of the SMORITHM. Creating a standard IDE to prevent common development mistakes (syntactic and/or semantic errors) is the last goal of the SMORITHM. The SMORITHM generates outputs (raw codes or compiled objects) based on the target platform. In this stage, an exporter is implemented to generate C++ code for Arduino microprocessors which are capable to be written directly on target devices. It is possible to generate output for other platforms and devices by implementing various generators and exporters. In result, As a result, the proposed framework in this thesis can generate outputs according to the requirements of a wide range of applications, regardless of the system kinds and processor of the devices. More details about the proposed framework are represented in chapter 4.

Chapter 2 is focused on logical inferences, intelligent systems, and their general attributes. In this chapter, decision structures, the state machine in smart applications development and intelligent machine types and autonomous systems are explained. Chapter 3 describes the wireless sensor network and IoT briefly. Chapter 4 introduces the proposed framework as WSN Manager Framework (SMORITHM). It explains the goals of SMORITHM, architecture and other available researches in related to our thesis topic as named related works. Evaluation and simulation of the SMORITHM are shown in chapter 5 which include some scenarios.

CHAPTER 2

LOGICAL INFERENCES AND INTELLIGENT SYSTEMS

Highlights

- Explanation to logical decision making structures
- Describing the state machine in smart applications development
- Survey on intelligent machine types and autonomous systems

Generally, an intelligence system is a system that is created with some abilities to act properly in various situations. It should use collected data in its decision making structures (Veres, et al., 2011). Decisions are Boolean functions that provide doing or ignoring a set of actions. Although they have many types, all of them work on logical statements and conditions. The result of a circuit of conditional statements which is provided by expert designers or machine learning methods leads to making a decision to do actions. So, decisions and their conditional statements are the most important parts of intelligence systems.

The state machine is a proper method to provide an efficient decision system. It includes one diagram or more in the case of concurrent systems which can be checked simultaneously. The current state of these diagrams can be bound into a set of actions. Therefore, when a current state is changed in a diagram, its related actions should be run. In other words, state machines are the mappers that connect input parameters to different states and actions.

In this chapter logical decision making, state machine and intelligence machines are described.

2.1. Logical Decision Making

To control a logical flow, calling or not calling a function or instruction is essential. In computer science, it is provided by the conditional statements which are evaluated as a true or a false answer. The conditional statements have a nested structure, however, each statement only can be a Boolean member. A conditional statement can include mathematical and logical operators and operands (Horvitz, Breese and Henrion, 1988). For example consider a conditional system with three variable inputs (Temperature, Humidity, and weather) and three constant values (22, 50 and sunny) and five operators (AND, OR,>, < and =). Figure 2.1 is represented as a visual sample for a little bit more complex circuit of conditional statements.

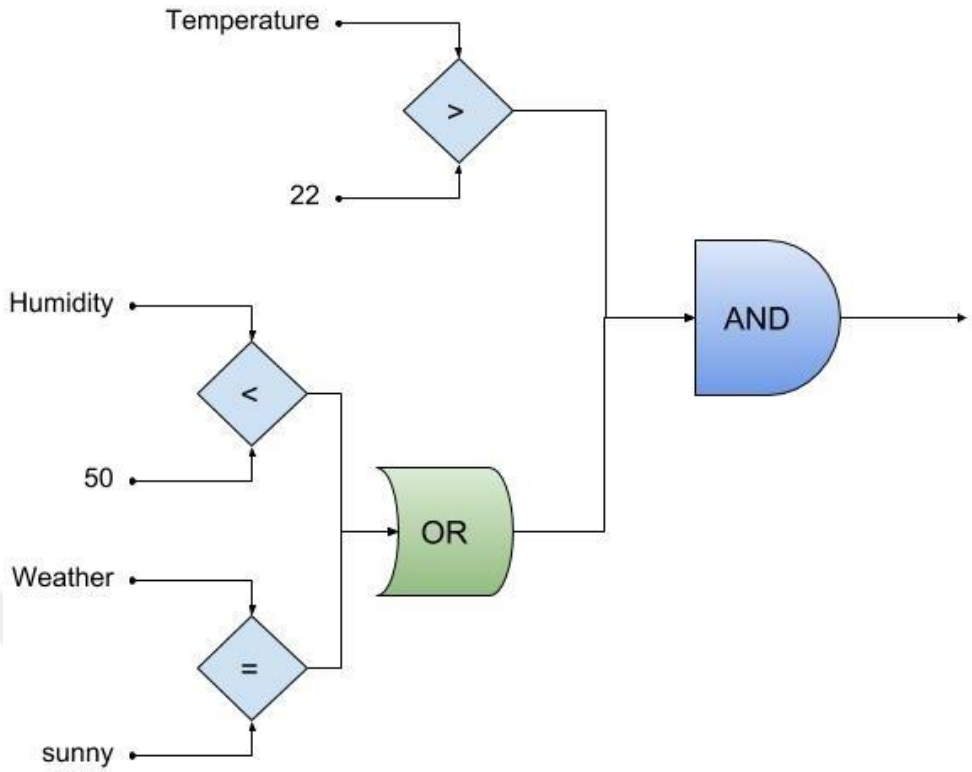


Figure 2.1: An example of circuit of conditional statements

For instance, if the current temperature is 22-degree Celsius, the outcome of temperature > 20 is true and for temperature < 0 is false. Assume that the current humidity is 60% and the weather is sunny. These nested statements samples can be valid similar the Figure 2.2. The result of conditional statements is (temperature > 20 && (humidity<50 || weather==sunny)) = true.

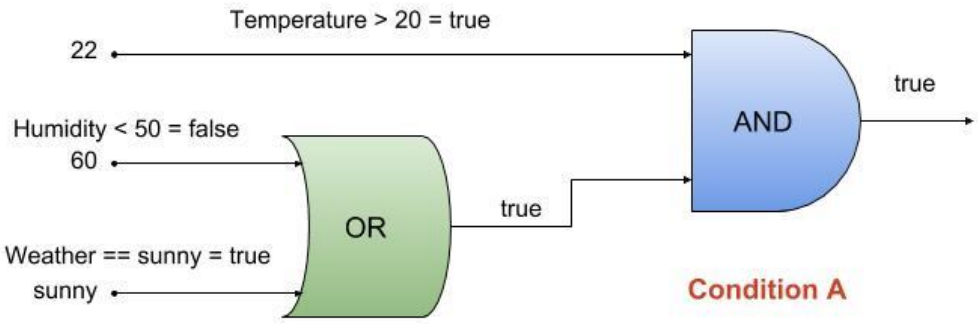


Figure 2.2: The logical statement of Condition "A"

Each condition statement can be titled for a better realization. For above example let's name it as Condition A and give another example as a Condition B which is shown in Figure 2.3. The result is (temperature < 0 || humidity > 80 || weather==cloudy) = false.

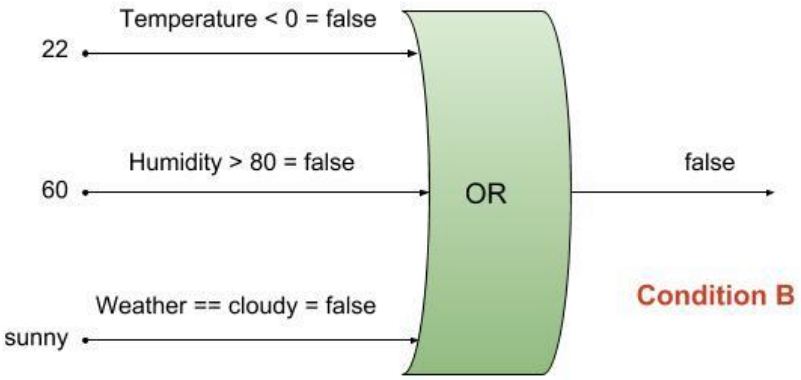


Figure 2.3: The logical statement of Condition "B"

A decision is made when the outcome of its condition is true. In the condition part, the truth function is called and a Boolean output is calculated. Despite a condition is able to accept multiple inputs, its only one output determines a binary result to do or not doing a function; the result can be true or false. In a logical problem, practically mentioning all Boolean expressions is impossible. Therefore the conditional expressions definition should be limited to the problem's scope and the evaluating system must be able to calculate the result of each. But in making decisions, a logical decision system must evaluate all possible branches. It means that a decision system has not to know anything, but it has to act all possible actions based on any condition it has.

When a decision system is defined, the designer wants to handle various functions in a specific application. For example, as it is represented in Figure 2.4 with considering above examples of conditional statements, a simple decision system contains three functions (Go to picnic, stay at home and study) are mapped to A and B conditions.

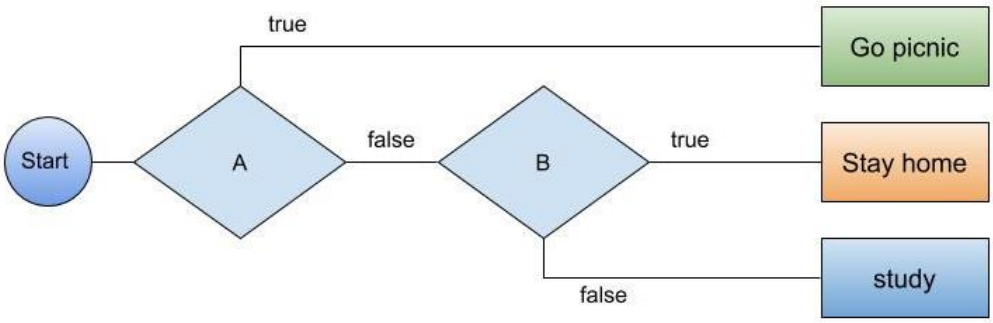


Figure 2.4: An example of a decision system

Generally, in programming languages (or compilers) these structures are represented by “if”, “then” and “select case” keywords; instructions and functions are executed between related blocks. Coding and programming for a large logical system can generate lots of condition statements as instructions and code blocks that can confuse the developers. There is a mathematical model of computation for these cases that called state machine.

2.2. State Machine

The state machine or finite automaton is an abstract machine to model a mathematical computation for designing algorithms. This machine can changes its current state to another state response to external inputs (Balle, 2013). The state changing action is called transition which is fired when the condition of the current state is raised. State machines’ behavior is observable in many modern devices that provide predefined actions depending on the series of events as inputs. Vending machines, elevators and traffic lights are the instances of these devices that we visit every day.

Each machine has a pointer to identify the current state and accepts inputs via specific ports and passes them to its state managers. Afterward, the exit transitions of the current state are checked and the proper state is selected as the next state. Each transition of current state that evaluates the result of the conditional statement as false is not eligible to be a right candidate for next state and state machine ignores that. The set of parameters that a state machine can accept and read

(alphabet) must be known for the machine; therefore the non-recognized entries are rejected. The first state that is selected by state machine automatically is called start state and the last state is called exit state. Intermediate states are the states that can be chosen as the current state frequently. A general schematic of a state diagram is shown in Figure 2.5.

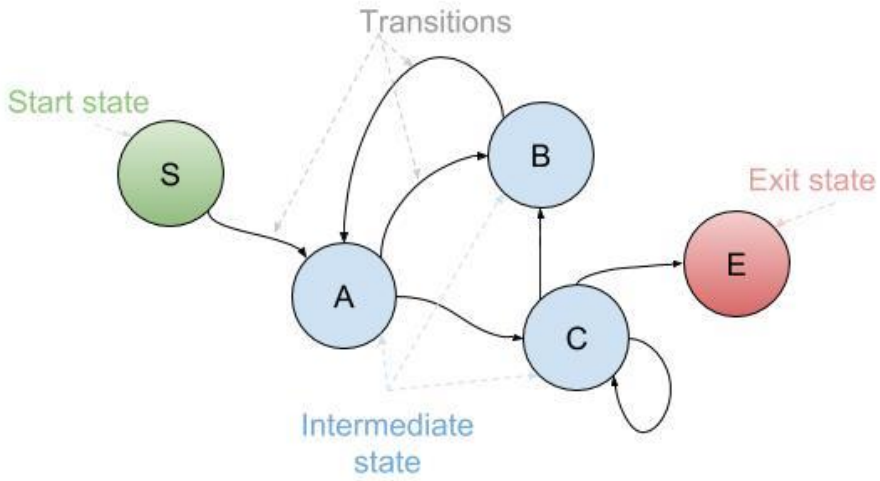


Figure 2.5: Schematic of a state diagram

Mathematical model

- Σ: input alphabet (a finite and non-empty set of symbols)
- S: Set of states (finite and non-empty)
- S_0: Start and initial state
- δ: transition function: $\delta: S \times \Sigma \rightarrow S$
- F: Exit and final states a (possibly empty) subset of S

A simple vending machine accepts money and request then after validating delivers an item or shows a proper error message.

A simple vendor state diagram is shown at Figure 2.6. Typically, the clarity of the state diagram is more than solid codes and it represents more information than those. Also, it is traceable more quickly.

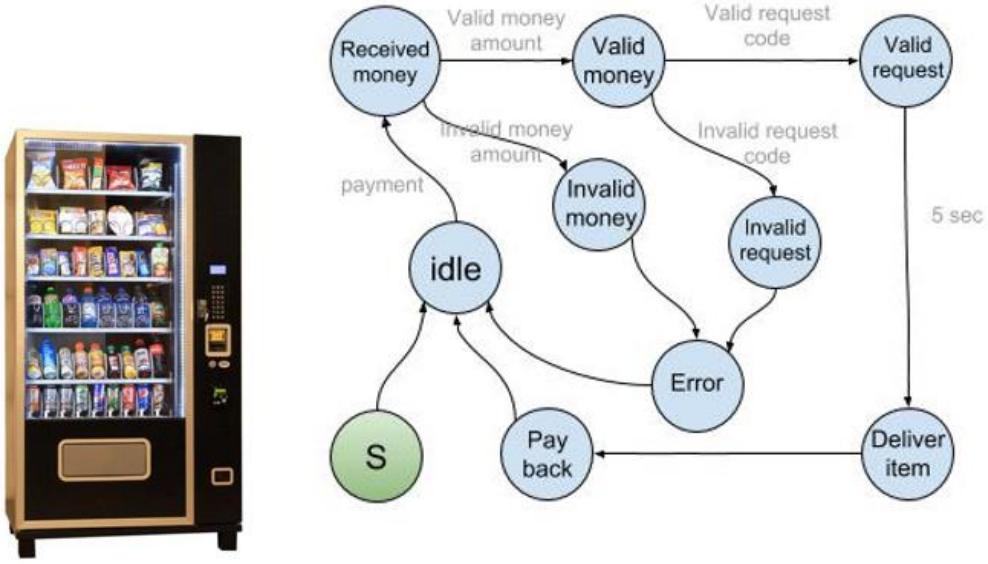


Figure 2.6: Sample state diagram for a vendor machine

The initialing input sequences for the previous example can be something like that: 10$, 34 (code of a drink that costs 3 $), finish button press. In the following, the output is a drink and 7$ remaining money payback.

For each input, a set of proper functions are anticipated and it is expected that all standard inputs have a corresponded response.

For consuming resources properly, achieving goals and completing tasks, doing right action at the right moment is an important attribute for a machine. A system can be an intelligent system by the abilities is created with or can be a smart system by the abilities that earns itself. At the following, a brief explanation about intelligence systems is available.

2.3. Intelligence machines

Some bodies believe that we are nothing else than energy. Earth, humans, cells, and atoms are kind of managed energy units, and they are made of each other. If this theory is emphasized, all instances of energy are frequently converted to other types of them. In some cases, materials are converted to actions and in some other cases actions are converted to other actions. The universe is one and many at the same time. No temporal creature is the first and nothing will be the last, everything is transformed from meaning to another concept.

The light of a bulb is a bunch of photons created from the kinetic energy of water behind a dam and in a process, it is transformed. Maybe we expect anything to be a controllable transformer for us. As humans recognize their limitations, they use more assisting artifacts than other creatures to achieve their goals. One of the most famous of them is machines. The machines are everywhere and they assist us. Usually, we want to perform our works quickly, easily and accurately. In some cases, the level of a risk forces us to use machines instead of a real human. As the result, we have a primary expectation from machines: they should be our successor.

In the classic definition, a machine is a structure does action. At the complex instances, machines are made of other machines, for example, a vehicle has a motor and the motor has some other subparts. To have a well-working machine, every part of that should works on-time and correctly. A machine needs to be monitored and controlled, and somebody should manage various parts of a machine based on logic. A machine should do specific actions in proper times. But how and when? The answers to these questions can help us to design and create useful machines. A machine should represent two abilities: the power of action and the tact of decision making (Garvey and Lesser, 1994).

This chapter of the thesis is focused on decision structures by explaining some of the common methods in this field.

2.3.1. Systems classification based on intelligence factors

An intelligence system is a set of connected components and elements that are organized for a specified purpose and it has the capability of analyzing and making decision too. If these systems are man-made, they are called artificial. Robots which are accepted as intelligence machines are controlled by computational and/or learning software. As the point view of how the machine can make a decision, there are four major types that are explained in this section.

2.3.1.1. Reactive machines

To understand this kind of systems, a brief explanation about reactive programming might be useful. It is a declarative programming paradigm that

follows data streams and reacts the base on data attributes and the set of conditions and functions are constant and predefined instructions.

Reactive machines are the most basic type of Artificial Intelligence systems. They work on the same logic and all decisions and actions are anticipated in the analyzing and design step. The machine tracks the data propagation and decides and acts properly (Perez, Bärenz and Nillson, 2018). These data streams can be originated from external devices like sensors or internal data providers such as an internal clock. The most famous machine belongs to this type is the Deep Blue which beat Garry Kasparov (international chess grandmaster) in 1990 (Figure 2.7).

Because of using the direct computational and mathematics functions on current data stream without any overhead of extra processes, the most optimal software running and high-performance operations execution appear here. Ease of implementation is another advantage of these systems, which is possible by clear condition and function structures. Another advantage is reliability which is provided by observable data values and corresponding responses.

Figure 2.7: ACM Chess Challenge Garry Kasparov vs Deep Blue

A large scale of the decision structure's data that should be configured by developers is one of the disadvantages of designing these machines. Also typically, these machines use current streaming data and they lose previous data streams and experiences.

2.3.1.2. Limited memory

In this type of machines, the machine observes data streams and tries to remember some information related to its task. These results are temporary and they can't be used as experiences for future generations. Although the targets are dynamic but similar to reactive machines, generally the conclusion and decision structure of these machines are mostly static. A good example of these systems is self-driving cars which are observable by an example in Figure 2.8. These systems recognize road lanes, other cars, humans and other important elements and decide to avoid collisions.

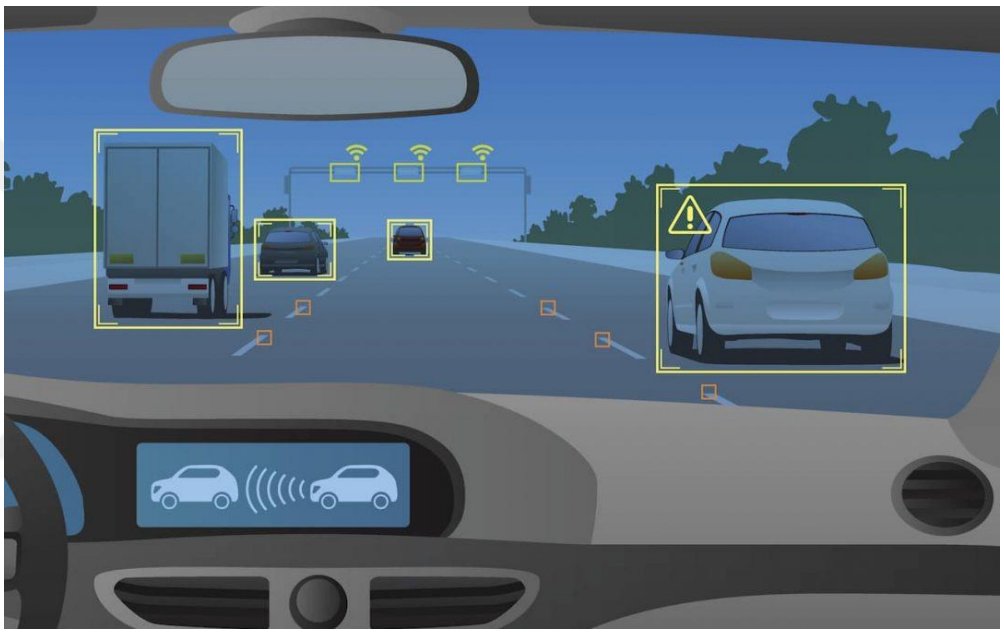


Figure 2.8: A vision of a self-driving car

Limited memory machines which are the extended generation of Reactive machines decide and act not only by current data but previously observed data effect on following actions and decisions. Instances of this type need memories with more capacity and higher processing resources to respond in a reasonable duration.

2.3.1.3. Theory of Mind

The "Theory of Mind" in psychology and philosophy is a term that represents a mental state imputation from an individual for itself or others by imaging the position and situation of them (Rabinowitz, et al., 2018). The parameters of this

state have an essential role in making decisions for the members of these machines. The methods of this type that anticipate the state and behavior of recognized agents assist machines to have a better interaction with the external world. Some researchers suggest that these machines should have a simulation of a version of their targets to predict their requirements. As it is shown in Figure 2.9, research (Blum, Winfield and Hafner, 2018) discusses the safety of mobile machines by simulating via an internal simulator within them.

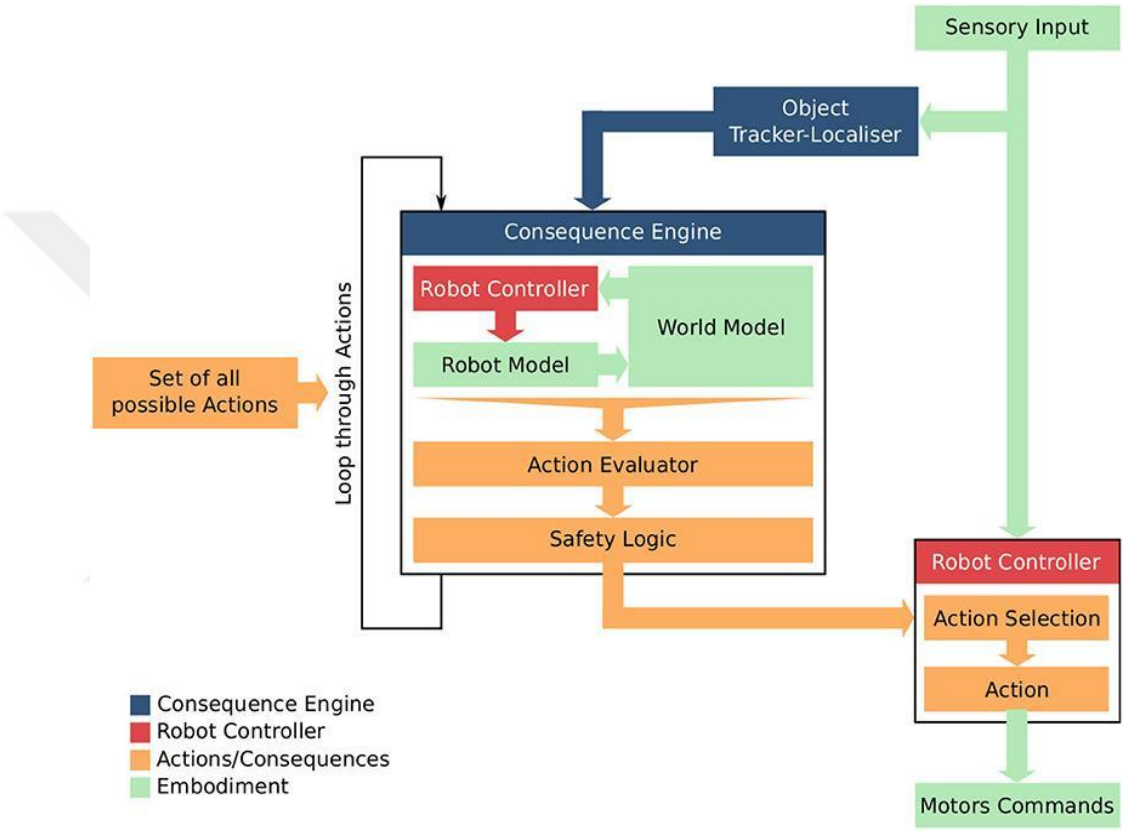


Figure 2.9: The safety of mobile machines' diagram

Since the only perceivable space for machines is discrete, these machines need an individual part for each agent that need to be simulated. Indeed as the machines have limited resources, so that is necessary to know the limitation of machines or at least has a concrete strategy for ignoring over-resourced targets.

2.3.1.4. Self-awareness

Being aware of self-abilities and limitations and having an accurate prediction about the result of possible decisions and actions are brilliant abilities of an

intelligent machine. Even Though this is a big claim but this type of machines wants to replace better choices with the best choices. If all perceptions of an environment are accessible for a machine and it is able to perceive and conclude them to make decisions, a fault-free machine has been created, but we know that is impossible, so sensible information should be limited in a controllable scope. The next solution to increase the efficiency is to distribute the processes in multiple nodes and creating a proper way to communicate between them. Especially in the swarm and the colony of machines, communicating between nodes can significantly increase the performance and lead to achieving the overall goals in a better way. A schematic architecture for these systems is represented in Figure 2.10. The further information is available in this paper (Lewis, et al., 2015).

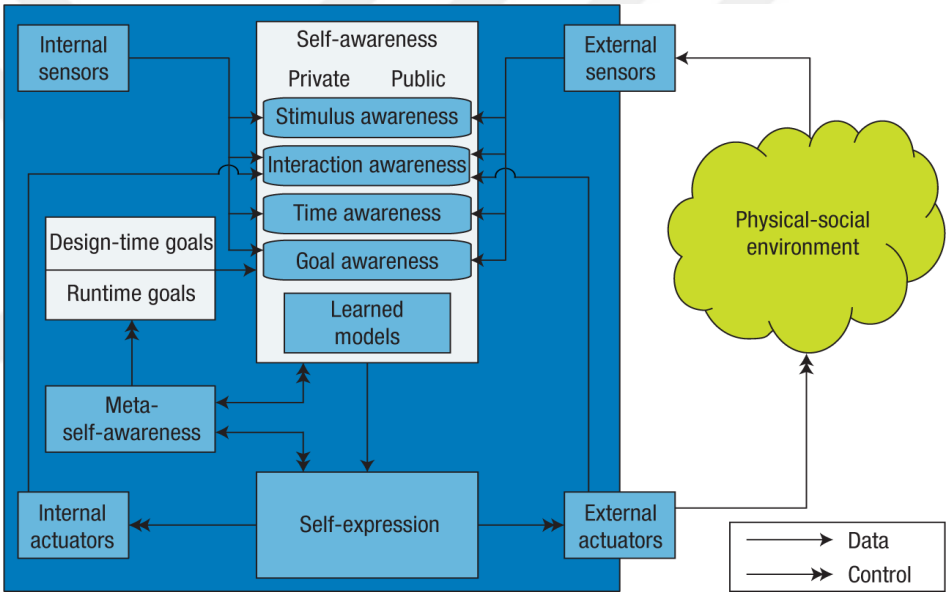


Figure 2.10: A schematic architecture for a self-awareness system

Until there is not an appropriate tool to represent feeling and awareness, this type will remain an indefinite concept. Also gathering, designing and processing conditions are crucial; because an intelligent system decides base on stored conditions and experiences. Determining the level and the type of intelligence factors for a system can be varied by different applications` requirements. There are many methods to managing knowledge and experiences and to convert them into understandable information for synthetic brains in data science concepts; such as decision trees, Bayesian and neural networks, reinforcement and deep learning methods, regression methods and etc. The most important point that we have to mind is that: a decision is a definite action, whether providing the conditions` part

lasts 10 days or 1 millisecond, or the condition set depends on one input parameter or thousands, the machine decides definitely one decision: to do or not to do. Decision system has the full coverage attribute in its nature and all machines have a decision for all parameters. In other words, if there is an unknown situation for a decision structure, it is ignored by that so the decision system decides to be passive against unknown situations. A better decision system is a system that decides more correct decisions than others that close it to its goal.

Due to some situations, guiding and controlling machines by humans is impossible. In the scenarios with the lack of stable connectivity or the situations that require rapid reactions, a high degree of distribution, calculating the large amount and executing lots of parallel actions, using human abilities directly seems inefficient. Therefore in these cases, a system that has the capability to act independently is more useful.

2.3.2. Autonomous systems

Autonomy is the capacity to make independent and aware decisions. An intelligent agent that decides without asking and acquiring permission is called an autonomous agent. A tangible example for autonomous systems is the robot and a more conceptual instance of these systems is a wireless sensor node. They can be a single machine with individual decisions or a member of a swarm which each member can effect on other members' decisions. Typically autonomous systems are supervised and managed by an organization or a single entity which they are employed to assume specific obligations; hence they are pre-programmed agents and they are responsible for their decisions as successors on behalf of their employers (Chen, Wang and Li, 2009).

In the cases that an application needs to has many of these agents to guarantee the quality of its services; by the way, a network of agents has amazing advantages for its owner. Beside the increasing the coverage and reliability by distributing commodity agents, the cost and fault probability are decreased. One of the most essential concepts in these networks is connectivity and transferring data and commands between agents and supervising centers.

These days, autonomous systems appear everywhere in our life and they are extending their place more firmly. Smart cities and buildings, security systems, mobile communication networks, ISPs and etc. are using autonomous infrastructures and concepts daily. All of these systems have a common attribute; in a defined scope, they decide independently and automatically perform actions to help their owners. Moreover, to reach a better performance, they start to help each other. These collaborations are anticipated to reduce the risks and cost and to increase efficiency. For example in a smart building management system, the collaboration between a local temperature sensing device, residents' preferences analyzer and a global weather forecasting system can lead to saving energy in total for the owner and increasing accuracy and efficiency for each system separately.

2.4. Conclusion

Typically, intelligence systems perform logical inferences by their decision systems with circuits of conditional statements. Depending on the result of inferences, a system does a proper action that bound to a corresponded decision. An autonomous system is a type of intelligence systems; therefore, to make a well-working autonomous system, designing a good logic unit with decision systems are necessary.

Next chapter has been focused on Wireless Sensor Networks and IoT as distributed and autonomous systems.

CHAPTER 3

WIRELESS SENSOR NETWORK AND IOT

Highlights

- Introduction to network
- Ad hoc, Wireless Sensor network concepts and IoT
- Applications and use cases for IoT and WSN
- Hardware components of WSN
- Characteristics of WSN

Since the “data” term has been born and it is used in machines and devices, sharing data between devices becomes more important day by day. This sharing can be seen in a television broadcast, a telephone communication, a Bluetooth speaker or an online market website. The advantages of being the member of a network are non-negligible, therefore many methods are provided for devices to connect them to networks.

3.1. Network

In computer science, a network is a collection of connected devices (nodes) that allow the sharing of data. The connection’s type of nodes which generally modeled by graph is called topology. The nodes are equivalent of vertices and connections are represented by edges. Most common topologies are shown in Figure 3.1.

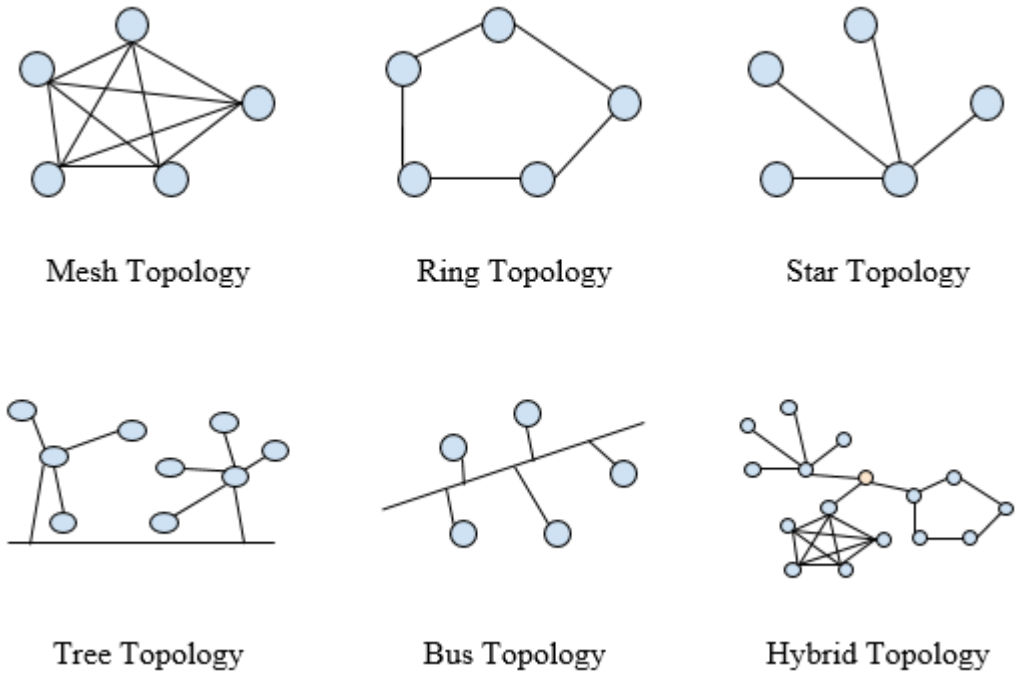


Figure 3.1: Topologies of the network.

Each topology has its specific properties and use-cases. Data are transferred between nodes via several types of links which they are generally grouped into two main categories: wired and wireless. Moreover, both of them can be organized with various topologies. In wired networks, nodes are connected via physical cables and in wireless networks, the relation is provided by radio waves, optical rays,

ultrasound and etc. which is fully dependent on expectations from a network and environmental features like the density of space's material and so on.

Networks are categorized with several aspects and views. Furthermore, there is another method to categorize networks by their scales which is shown at Table 3.1.

Table 3.1: Network classification by scale

Row	Name	Scale
1	Personal Area Network (PAN)	Home (between personal devices)
2	Local Area Network (LAN)	A building (e.g. company)
4	Campus Area Network (CAN)	Multiple buildings (Multiple LANs)
5	Metropolitan Area Network (MAN)	In a city scale
6	Wide Area Network (WAN)	Country / world scale

Sharing data has its problems like security and resource management. The validation and access control generally is done with one of these main groups: centralized and decentralized. In a centralized structure, nodes are coordinated by a managing system and they have to be authenticated by that system. The data which should be transferred are routed by a set of known rules and protocols from a target node to a source node or multiple nodes. Conversely, due to the peer to peer attribute of nodes in decentralized structures, each node assumes the responsibility of connectivity by itself and it gives permissions or denies a connection. Centralized networks like internet are more reliable than decentralized networks like ad hoc networks. On the other hand, generally decentralized

networks are cheaper and more flexible than centralized because decentralized networks do not need to enterprise infrastructures.

3.2. Wireless Sensor Networks (WSN) and Internet of Things (IoT)

Physical independence of wireless nodes from a static location enables them to have mobility while wired network nodes cannot move easily. Wireless ad hoc network is a type of wireless networks that the nodes are able to communicate directly with each other and they don't rely on pre-existing devices like routers or access points (Akyildiz, et al., 2002). This feature is important when we want to create a fast and an easy-to-install network. For example, two mobile phones can transfer files via Bluetooth directly without dependency on any third party device or infrastructure. Typically in these devices, the transceivers support global standard protocols so these devices can be used in a wide range.

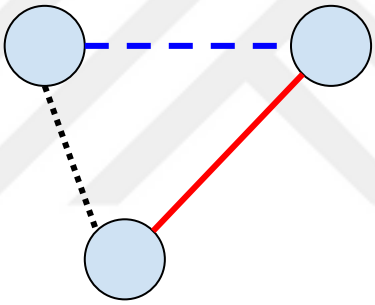


Figure 3.2: An ad hoc network sample with three independence connections

WSN is a type of networks that sense the environment and send them to a central base station. These networks have a semi-centralized structure which is inherited the direct negotiating between nodes and mobility from ad hoc networks (Figure 3.2) additionally the coordinating and routing attributes from centralized networks (Figure 3.3).

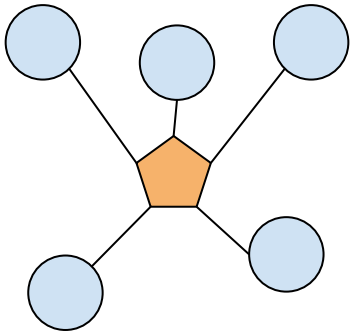


Figure 3.3: A centralized network

The nodes in the Wireless Sensor Network (Figure 3.4) are able to communicate with each other to perform a bigger task which is gathering information from nodes to base stations and commands from base stations to nodes. Some nodes are assigned to the sensing tasks, receive their packets and send their data. Some other nodes also may accept the data that do not belong to them, but they pass those packets to other nodes to deliver them to the target node; In other words, they act as bridges and routers.

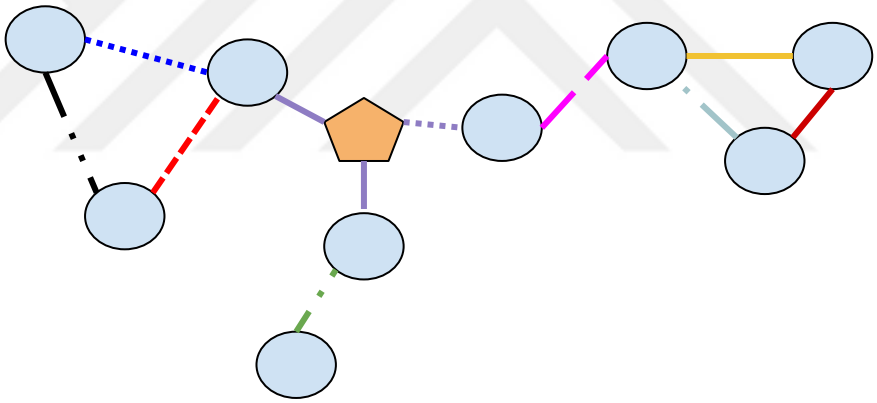


Figure 3.4: A schematic for WSN with ad hoc connections

Plenty of protocols have been designed for wireless networks to enhance the efficiency based on global protocols (like ZigBee and TCP/IP) or fully customized and internal protocols (Al-Karaki and Kamal, 2004).

IoT

IoT is the wide network of various type of devices. The Internet of Things (IoT) involves internet connectivity beyond devices (are shown in Figure 3.5) that have standard network interface and enough infrastructures to connect to the internet.

IoT is a young sibling of WSN that uses TCP/IP as a communication protocol to communicate between nodes and central servers. By the nature of TCP/IP, its topology is the star, but by combining WSN concepts with IoT, hybrid structures are considerable (Figure 3.6).

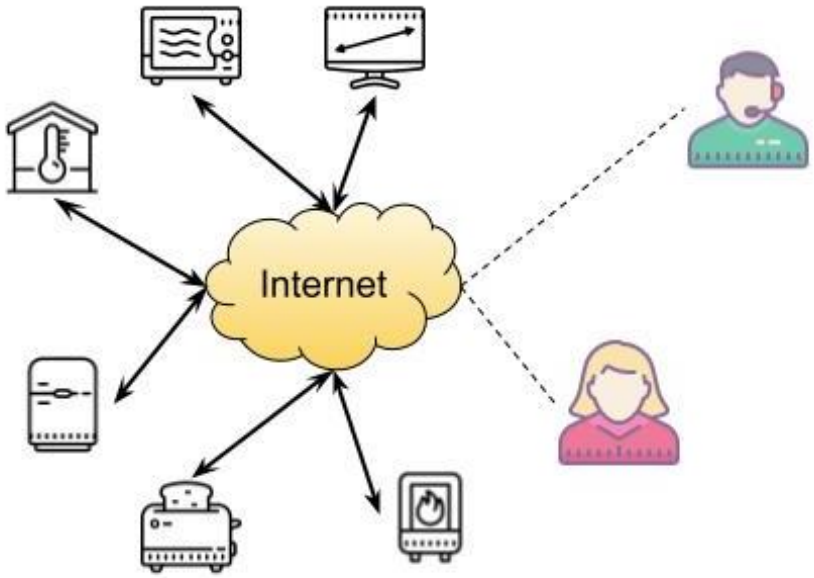


Figure 3.5: A schematic for IoT

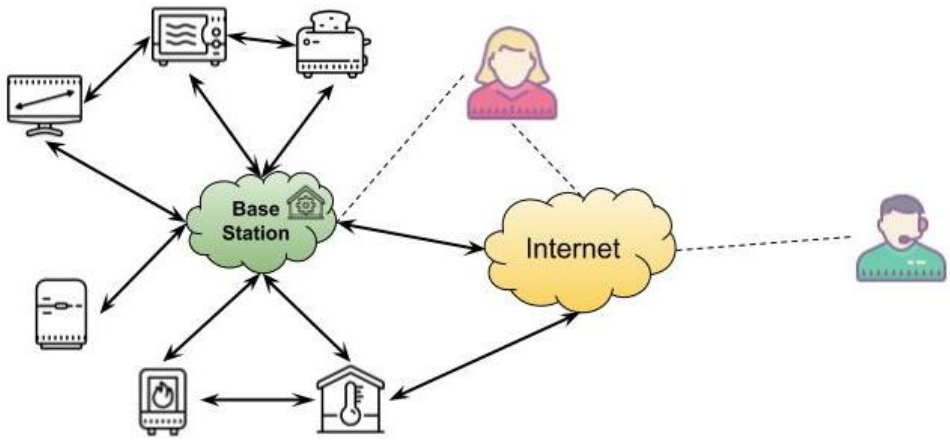


Figure 3.6: A schematic for WSN and IoT hybrid

Remotely controlling and monitoring devices are feasible by IoT. There are vast and fast-growing use cases and applications for WSN and IoT in commercial, industrial, infrastructure, and consumer spaces. Additionally, other types of

grouping are described such as smart metering, asset tracking, connected vehicles, fleet management and so on.

3.3. Applications and use cases for IoT and WSN

Smart sensing and remote controlling machines are useful where performing these kinds of actions are difficult directly by humans. There lots of categories and instances for related applications that are mentioned some of them in this sections.

3.3.1. Area monitoring

Continuously monitoring area parameters like temperature, motion, vibration, specific particles and gases, humidity and etc. can be implemented by these networks. Generally, area monitoring is divided into indoor and outdoor monitoring. Nodes that programmed to recognize specified phenomena are deployed over an area and detected events are sent to base stations; On the other hand, devices can get commands to execute (Ganesh, 2017; Gubbi, et al., 2013; Bellavista, et al., 2013; Iqbal, Kim and Lee, 2017). An example of a sensor node with a light sensor is shown in Figure 3.7.

Figure 3.7: A sample of light sensor node

3.3.2. Health care monitoring

Pervasive and easy to use health care gadgets have an irreplaceable position when saving lives is the subject. Usually, treatment actions should be applied in a limited

deadline and seconds are vital. Differently, measuring clinical trials may be annoying and costly for patients in short intervals. Therefore, mounting small sensors on patients' bodies to send their status continuously or triggered by events is a good solution. Generally, two types of body sensors are on-body sensors which sense and are connected to body and off-body sensors which transfers data from on-body sensors to the external world. On-body sensors can be wearable or mountable on skin or organs and the off-body sensor can be an ordinary device like smartphones (Lo, et al., 2005; Bangash, et al., 2014). An example of body sensor network is represented in Figure 3.8.

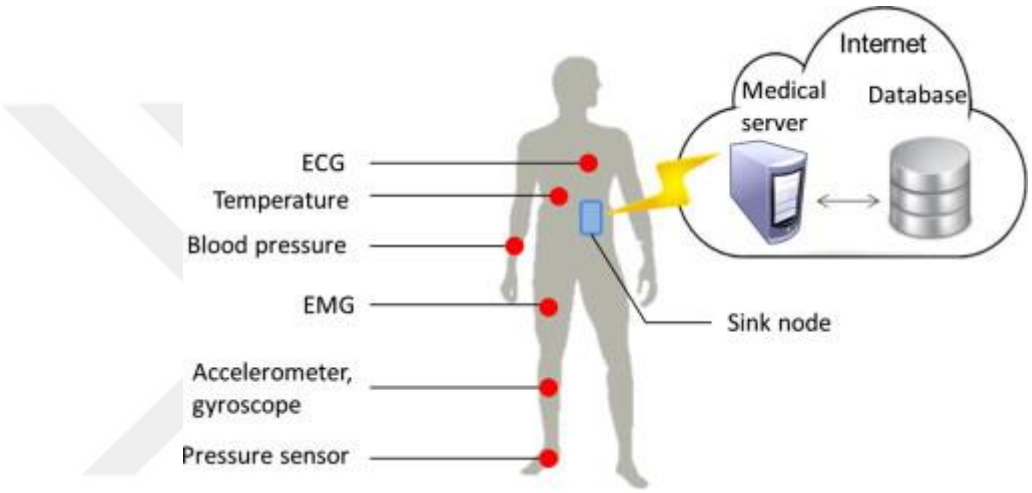


Figure 3.8: Body sensor network

3.3.3. Environmental/Earth sensing

Measuring environmental phenomena can be risky and dangerous, but to get accurate anticipation they are very important. Naturally, the geographical scopes of phenomena are so wide that maintaining measurement devices are difficult and time-consuming. A few instances of these networks' applications are explained below.

Air pollution monitoring

Different types of air pollution monitoring systems are performed to data gathering from various sources and the collected data are used in different usages. Four instances of them are described in this part.

- **Ambient monitoring:** By this method, overall quality of an area is measured and it involves the whole city, state or country. The collected data are used for long-term spatial trend analysis in time intervals.
- **Mobile and on-road monitoring:** If the mobile version of sensing instruments similar to ambient monitoring instruments are equipped on a mobile container (such as a vehicle that is shown in Figure 3.9, drone or ship), a linear or local area can be measured. Especially roads, tunnels and places without the possibility of permanent nodes' montage are the target of this method.
- **Emission monitoring:** If a specific point is spotted to be monitored, proper detecting devices are mounted in appropriate place to detect the partial pollution. This method is useful for monitoring pollution of factories, industrial machines and so on.
- Satellite monitoring: Generally, this method is used to compare pollution changes in time periods by devices that are embedded on satellites.

Except for satellite monitoring, all three methods are feasible to be implemented by WSN and IoT nodes with gas and particle sensors. To better area coverage, nodes can collaborate with each other to reduce redundancy and lack of forgotten locations data.

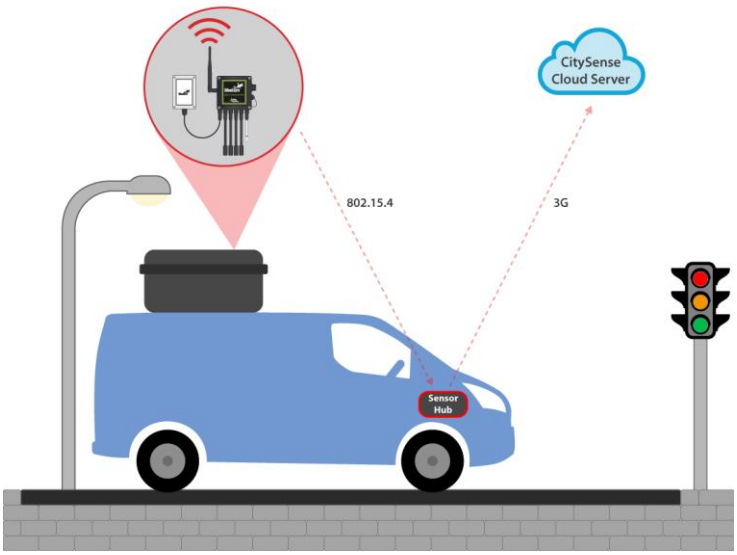


Figure 3.9: A mobile air pollution monitoring system

Forest fire detection

Most places of forests are vast and empty of human, but a small fire can be extended rapidly. Detecting fire events immediately can help the firemen and extinguishers to prevent fire expanding. As mentioned in the paper x (A Review on Forest Fire Detection Techniques) fire detection techniques by authorities are fire weather forecast and estimates of moisture and fuel, watchtowers, optical smoke detection, lightning geographical coordinates detectors, infrared, spotter planes and smartphone/mobile calls. Sensing is an important subject, but informing them to authorities has the same importance (Alkhatib, 2014). Flowing data streams to deciding centers can be provided by WSN and IoT in reasonable cost and complexity.

Landslide detection

Landslides can be caught by various methods (like inertial measurement unit sensors, image processing and etc.); monitoring and warning them can prevent big disasters. Usually, if enough amount of data exits, landslides are predictable; WSN and IoT are capable to collect these data. By machine learning methods, time and scale of landslides can be estimated (Giri and Phillips, 2018; Ramesh, 2009).

The water quality monitoring, natural disaster prevention, and some other instances are the other types of environmental/earth sensing applications (Pule, Yahya and Chuma, 2017; Fernandes, et al., 2013).

3.3.4. Industrial monitoring

Typically in industrial manufacturing pipelines, the machines of factories are configured to produce in a high-performance process. Therefore the time gaps are minimized and in a hierarchical productive structure, the start of a process immediately is begun after finishing the previous process. Sometimes fault occurrence in some machines leads to appearing problems in related machines. In some industrial units, machines are arranged with minimum spaces and roaming between them is difficult for maintainers. Generally, three types of actions can be done in industrial spaces and they are described in this section.

- **Fault detection:** Machines and resources can be monitored continuously and supervising team can be aware when and where a fault has occurred. For example, an alarm is raised when the temperature of a rack in a data center is increased more than a threshold; by using WSN administrators are able to know the exact point of fault without roaming between shelves and cables.
- **Fault correction:** in some cases, there are automatic and predefined instructions to do by other machines to fix a particular fault in a machine. For example in the case of previous section fault, until fixing the main problem by maintainers, an alternative cooling system can temporarily be entered to the circuit. Usually, to reduce the risk, alternative systems can start their jobs autonomously.
- **Fault avoidance:** Sometimes by knowing fault occurrence or unusual behavior in a machine, upcoming other errors are predictable; so, changing current activity to a safe level can prevent destroying other parts. In the following previous example, if the alternative cooling system is not successful to reduce the temperature, the electric power of racks is shut down to prevent the fire.

The use cases of WSN and IoT are not limited to these instances. They can be used for military purposes, smart city, agriculture, security, education, sport, and many other purposes. Depending on applications' requirements, proper software and hardware should be provided for network nodes and devices (Khariunniza-Bejo, Ramli and Muharam, 2018; Othman and Shazali, 2012; Đurišić, et al., 2012; Afshar and Manzuri, 2011; Kiani and Seyyedabbasi, 2018; Afshar, Manzuri and Latifi, 2011). At the following section, a brief description of hardware components and their specifications are explained.

3.4. Hardware components of WSN

Typically, a wireless sensor network is composed of members that called node. In a common grouping, nodes are divided into normal nodes and base stations so that base stations supervise normal nodes and collect data from them. Base stations own more powerful resources than normal nodes, whereas normal nodes are configured to consume minimum resources in order to have a longer operational lifetime. Both sets of conceptual behaviors and physical components are required

to create a wireless sensor node. Processing and transmitting capabilities are the minimum expectations from a node. Furthermore, in some applications nodes must provide extra controlling capabilities like performing turn on, turn off and other mechanical commands. Also, normal nodes (based on ZigBee standards) can be categorized into coordinators, end nodes and routers too. While end nodes sense the environmental phenomena, router nodes route collected data to base stations. Titles of nodes depending on their roles and can be different in various scenarios.

The tradeoff between costs and capabilities has always been an important issue as well as finding a balance point is a key to design a good network. Four primary components of a normal node are the processing unit, transceiver unit, sensing unit, and a power supply unit (Kiani, 2014).

3.4.1. Processing unit

The processing unit comprises processor and memory. At least one processing unit is mounted on nodes to execute commands and to perform algorithms like routing, sensing and lightweight data processing. Usually, based on light process characteristic of these nodes, microcontrollers are used which are better in energy consumption. Therefore, this kind of processors has a smaller memory and slower processing power than enterprise processors. Moreover, they have a single core and time-sharing techniques are suggested to avoid the deadlock in the case of concurrent processes.

3.4.2. Transceiver unit

This unit includes a receiver, a transmitter and some supportive components like antennas. As data and information can be defined as energy sequences (electrical currents) in order to store and transfer them, a dataset can be converted to a series of electrical signals in a predefined format. Wireless communication gives the capability of connectivity in the movement to nodes. By considering several attributes of physics, wireless communications are divided into four main categories.

Electromagnetic radiation

By referring to the physics, photons can be propagated from a source in a direction (or multiple directions) with the speed of light (a little bit less than 300,000 kilometers per second); this radiation is called electromagnetic radiations which consists of electromagnetic waves. These waves have properties like frequency and wavelength and have got names by several ranges that are shown in Table 3.2.

Table 3.2: Electromagnetic waves classification

Name	Frequency min (Hz)	Frequency max (Hz)	Wave length min	Wave length max	Photon energy (eV)
Gamma ray	30 EHz		0	0.01 nm	124 keV – 300+ GeV
X-Ray	30 PHz	30 EHz	0.01 nm	10 nm	124 eV – 120 keV
Ultraviolet	750 THz	30 PHz	10 nm	400 nm	3.1 eV – 124 eV
Visible	428.5 THz	750 THz	400 nm	700 nm	1.7 eV – 3.1 eV
Infrared	300 GHz	428.5 THz	700 nm	1 mm	1.24 meV – 1.7 eV
Micro wave	300 MHz	300 GHz	1 mm	1 m	1.24 μeV – 1.24 meV
Radio	3 kHz	300 MHz	1 m	100 km	12.4 feV – 1.24 μeV

Depending on frequency and wavelength, bandwidth and effective range and subsequently usage domain of each can be changed. The higher density of the

transferring environment causes to increase the collision between photons by environment particle. In the result, increasing density of the environment can lead to energy waste and data loss.

Radio waves

Radio waves are a kind of electromagnetic radiation with wavelengths which has the electromagnetic spectrum longer than infrared with frequencies from 3 KHz to 300 GHz. Data are modulated to radio waves and are sent by an antenna of a transmitter and at the other side, the receiver demodulates the radio waves to data again. As it is shown in Table 3.2 and Figure 3.10, radio waves are the lowest-frequency, lowest-energy and longest-wavelength; in addition, they are safer for live creatures. So they are used for general communication like radio and television broadcast, mobile and wireless networks and etc.

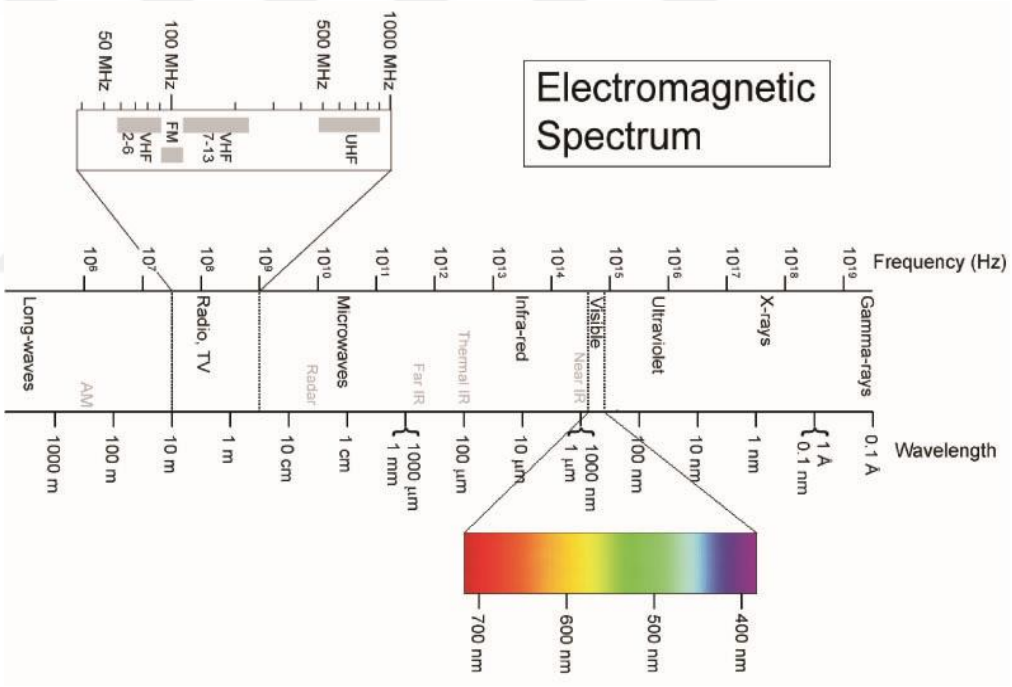


Figure 3.10: Electromagnetic spectrum

There is a protocol (various sub-versions of IEEE 802.11) that represents the WLAN (Wireless Local Area Network) frequency standards to choose a common frequency for devices.

Sonic

The other solution for data transfer is based on vibration which is known for us as sound. In high-density environments, radio waves are inefficient but in some cases, sonic which its logical concept is same as radio waves, acts better. The most used technique of sonic is ultrasound. Ultrasound is sequence of sound waves with higher frequencies than the upper audible limit of human hearing. Unlike the radio waves, Ultrasound needs a dense environment to travel the waves by sequential vibrations toward the materials. A study (Graphene electrostatic microphone and ultrasonic radio) expressed that the Ultrasound has a better performance in environments where Radio waves do not work well (like underwater). Naturally, dolphins and whales use this technique to communicate with other biotypes and profit it like radars to find baits (Zhou, et al., 2015).

Electromagnetic induction

Electromagnetic induction is the production of electromotive force across an electrical conductor in a changing magnetic field. Because of short range and power, this technique has been used in short-range RFID tags and biomedical devices such as pacemakers.

3.4.3. Sensing unit

A sensing unit includes an ADC and sensors. Each sensor node is capable to sense analog perceptions from phenomena via its related sensors and converts it to digital data by ADC. Sensing and measuring environmental parameters and events give valuable knowledge to make the right decisions (Sudevalayam and Kulkarni, 2011).

3.4.3.1. Sensed value calculation

To calculate and process the phenomena, a metric value range for each phenomenon has been mapped by scientists; in following interpreting them is possible. For example, the temperature has been measured in Celsius too and a +48 degree Celsius weather means too hot for humans. Typically phenomena have continuous values but at the specific time, it can be recorded as a single primitive value, vector or more complex object.

Detecting the value more than an expected threshold in phenomena is called event. Logically, events are caught to be handled by systems; these handles can be a controlling or monitoring approach. To catch an event, at least a listener or tracker should senses between start until finish of an event. Sensors have a detecting interval and choosing the right sensor for any scenario is important to avoid missing events. Referring to major types of sensors which are Digital and analog, an output voltage is translated to a number; zero or one for digital values and for example zero to 1023 for the analog 16-bit sensor. If a temperature sensor has sensitivity range between -50 and 150-degree Celsius, the functionality and gauge system can be interpreted like Figure 3.11 and Formula 3.1.

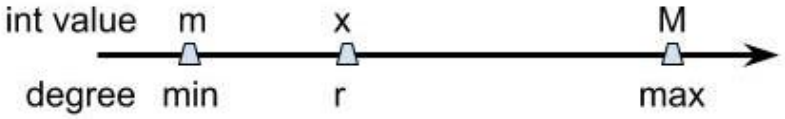


Figure 3.11: One dimensional vector of sensor

$$r = \frac{x(max - min)}{M - m} - min \tag{3.1}$$

For example, if the sensor shows 375 integer value as an output, the temperature is equal to $r = \frac{375 \times (200)}{1023} - 50 \simeq 23.3$ degree Celsius. Figure 3.12 shows a corresponding temperature vector.

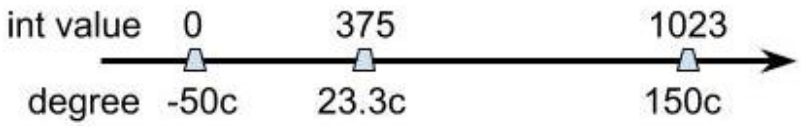


Figure 3.12: An example for a temperature vector

3.4.3.2. Sensors classification

Typically, sensors are categorized by their phenomenon. As this section is one of the most important parts, the categorized list of sensors is presented in this section.

Acoustic, sound, vibration

Geophone	Hydrophone	Microphone	Seismometer
Sound locator			

Automotive Sensors

Air flow meter	Air–fuel ratio meter	AFR sensor	Blind spot monitor
Crankshaft position sensor	Engine coolant temperature sensor	Hall effect sensor	Wheel speed sensor
Airbag sensors	Automatic transmission speed sensor	Brake fluid pressure sensor	Camshaft position sensor
Cylinder Head Temperature gauge	Crankshaft position sensor	Engine crankcase pressure sensor	Exhaust gas temperature sensor
Fuel level sensor	Fuel pressure sensor	Knock sensor	Light sensor
MAP sensor	Mass airflow sensor	Oil level sensor	Oil pressure sensor
Oxygen sensor (o2)	Parking sensor	Radar sensor	Speed sensor

Throttle position sensor	Tire pressure sensor	Torque sensor	Transmission fluid temperature sensor
Turbine speed sensor	Variable reluctance sensor	Vehicle speed sensor	Water-in-fuel sensor
Wheel speed sensor			

Chemical

Breathalyzer	Carbon dioxide sensor	Carbon monoxide detector	Catalytic bead sensor
Chemical field-effect transistor	Chemiresistor	Electrochemical gas sensor	Electronic nose
Electrolyte–insulator–semiconductor sensor	Energy-dispersive X-ray spectroscopy	Fluorescent chloride sensors	Holographic sensor
Hydrocarbon dew point analyzer	Hydrogen sensor	Hydrogen sulfide sensor	Infrared point sensor
Ion-selective electrode	Nondispersive infrared sensor	Microwave chemistry sensor	Nitrogen oxide sensor

Olfactometer	Optode	Oxygen sensor	Ozone monitor
Pellistor	pH glass electrode	Potentiometric sensor	Redox electrode
Smoke detector	Zinc oxide nanorod sensor		

Electric current, electric potential, magnetic, radio

Current sensor	Daly detector	Electroscope	Electron multiplier
Faraday cup	Galvanometer	Hall effect sensor	Hall probe
Magnetic anomaly detector	Magnetometer	Magnetoresistance	MEMS magnetic field sensor
Metal detector	Planar Hall sensor	Radio direction finder	Voltage detector

Environment, weather, moisture, humidity

Actinometer	Air pollution sensor	Bedwetting alarm	Ceilometer
Dew warning	Electrochemical gas sensor	Fish counter	Frequency domain sensor

Gas detector	Hook gauge evaporimeter	Humistor	Hygrometer
Leaf sensor	Lysimeter	Pyranometer	Pyrgeometer
Psychrometer	Rain gauge	Rain sensor	Seismometers
SNOTEL	Snow gauge	Soil moisture sensor	Stream gauge
Tide gauge			

Flow, fluid velocity

Air flow meter	Anemometer	Flow sensor	Gas meter
Mass flow sensor	Water meter		

Ionizing radiation, subatomic particles

Cloud chamber	Geiger counter	Geiger-Muller tube	Ionisation chamber
Neutron detection	Proportional counter	Scintillation counter	
Semiconductor detector		Thermoluminescent dosimeter	

Navigation instruments

Air speed indicator	Altimeter	Attitude indicator	Depth gauge
Fluxgate compass	Gyroscope	Inertial navigation system	Inertial reference unit
Magnetic compass	MHD sensor	Ring laser gyroscope	Turn coordinator
Variometer	Vibrating structure gyroscope		Yaw rate sensor

Position, angle, displacement, distance, speed, acceleration

Auxanometer	Capacitive displacement sensor	Capacitive sensing	Flex sensor
Free fall sensor	Gravimeter	Gyroscopic sensor	Impact sensor
Inclinometer	Integrated circuit piezoelectric sensor	Laser rangefinder	Laser surface velocimeter
LIDAR	Linear encoder	Linear variable differential transformer (LVDT)	Liquid capacitive inclinometers

Odometer	Photoelectric sensor	Piezoelectric accelerometer	Position sensor
Position sensitive device	Angular rate sensor	Rotary encoder	Rotary variable differential transformer
Selsyn	Shock detector	Shock data logger	Tilt sensor
Tachometer	Ultrasonic thickness gauge	Ultra-wideband radar	Variable reluctance sensor
Velocity receiver			

Optical, light, imaging, photon

Charge-coupled device	CMOS sensor	Colorimeter	Contact image sensor
Electro-optical sensor	Flame detector	Infrared sensor	Kinetic inductance detector
LED as light sensor	Light-addressable potentiometric sensor	Nichols radiometer	Fiber optic sensors
Optical position sensor	Thermopile laser sensors	Photodetector	Photodiode
Photomultiplier tubes	Phototransistor	Photoelectric sensor	Photoionization detector

Photomultiplier	Photoresistor	Photoswitch	Phototube
Scintillometer	Shack-Hartmann	Single-photon avalanche diode	Wavefront sensor
Superconducting nanowire single-photon detector		Transition edge sensor	Visible light photon counter

Pressure

Barograph	Barometer	Boost gauge	Bourdon gauge
Hot filament ionization gauge	Ionization gauge	McLeod gauge	Oscillating U-tube
Permanent downhole gauge	Piezometer	Pirani gauge	Pressure sensor
Pressure gauge	Tactile sensor	Time pressure gauge	

Thermal, heat, temperature

Bolometer	Bimetallic strip	Calorimeter	Exhaust gas temperature gauge
Flame detection	Gardon gauge	Golay cell	Heat flux sensor
Infrared thermometer	Microbolometer	Microwave radiometer	Net radiometer

Quartz thermometer	Resistance thermometer	Silicon bandgap temperature sensor	Special sensor microwave/imager
Temperature gauge	Thermistor	Thermocouple	Thermometer
Pyrometer			

Proximity, presence

Alarm sensor	Doppler radar	Motion detector	Occupancy sensor
Proximity sensor	Passive infrared sensor	Reed switch	Stud finder
Triangulation sensor	Touch switch	Wired glove	

Speed sensors

Wheel speed sensors	Speedometers	Pitometer logs	Pitot tubes
Airspeed indicators	Piezo sensors	LIDAR	Ground speed radar
Doppler radar	ANPR	Laser surface velocimeter	

Many types of sensors with different sensitivity various costs (financial, energy, and reliability) are available. Some types of them are a simple circuit and some other more complex types have an internal processor. But all of them can report their values via analog buses and pins.

3.4.4. Power supply unit

A power supply is an electrical device that supplies electric power to an electric load. There are many categories for power supplies but to concentrate on the subject of this thesis, static power supplies, and mobile power supplies are emphasized. Due to the very low consumption of nodes in WSN, the static power supplies (like urban electricity) are defined as infinitive power resources, but consumers require wire and cables to conducting the electricity power. So they are useless for mobile nodes. On the other hand, these kinds of resources are proper for steady equipment like servers and base stations that need a lot of energy to perform heavy processes.

Considering the characteristic of WSN nodes, the mobility has an excellent position between nodes' properties. Batteries are light-weight power supplies that can be carried easily beside of mobile nodes. Batteries have limited capacity and lifetime. Therefore they should be changed after they are exhausted; some types of them are rechargeable such as lithium batteries.

The current energy level of a battery can be calculated by voltage and its remaining power for a future life-time are predictable; the operational duration is depended on usage and the physical structure of battery. In smartphones and portable devices, a number in percentage format is popular to show the remaining battery level. To estimate metrics of the batteries, manufacturers put some data such as Battery Voltage Chart and Battery Discharge Curve in data sheets.

3.5. Characteristics of WSN

WSNs have common characteristics that empower them to appear successful in their operational domains. Having enough knowledge about them helps developers

to design and implement optimize networks (Yick, Mukherjee and Ghosal, 2008; Sundani, 2011; Sudevalayam and Kulkarni, 2011; Kim and Hong, 2004). Some of these cases are described in this subsection.

3.5.1. Power Consumption

Using resources is the main reason for consuming power in WSN. It is a popular point in WSN that reaching physically to nodes can be difficult, so having long-life nodes are necessary for the nodes with limited power supply. If a certain capacity for power supply (battery in most cases) is assumed, better power consumption means longer operational lifetime. Each component consumes electricity in its own way.

Processing unit: Depending on processor architecture and hardware issues, the processing unit consumes a specified amount of power per each process; therefore writing optimized codes and using algorithms with less complexity can reduce the instructions calling and lead to consuming less power. In addition, transferring large block of processes in a more powerful external processing unit (servers for example) can be helpful.

Transceiver unit: A connection without sending and/or receiving data is meaningless, exactly like a communication without connection is useless. In wireless networks based on radio, the term of sending data means propagation a bunch of waves and the receiving data means demodulating hit waves to the antenna; moreover sending and receiving need different amounts of energy which sending needs more than receiving. Generally, routing algorithms take care of this besides other important parameters. Covering vaster range needs more powerful sending, so finding a balanced point via choosing proper transceiver with reasonable range and frequency fits the scenario and application can prevent energy wasting in nodes.

Sensing unit: The other component is the sensor part. Analog sensors conduct a current inside their circuits to sense. Similar to processors, each sensor consumes electricity to sense; furthermore, the interval of sensing is effective in energy consumption. To have better power efficiency, choosing proper sensors with better

performance and using software methods to configure the sensing intervals are two solutions.

Power supply unit: Always, choosing more power with bigger containers has not better effect on performance. In some cases (like flying ad hoc networks) weight of batteries can increase the power consumption; especially in the case of using a shared power source for both nodes and mobile machine. The parameters such as rechargeable, durability, self-discharge rate, weight, size, persistence in a harsh environment (like inappropriate humidity, PH, temperature and etc.), and cost and so on are essential for choosing the best one for each application.

3.5.2. Flexible node failures handling

WSN nodes are cheaper and smaller compared with devices of other types of common networks. Portability, having more alternative nodes and the capacity of autonomy give more features in designing the networks with high degree of network failure flexibility. In other words, a failed node can be replaced by another one automatically without the need to administrators manipulating directly.

3.5.3. Mobility

Nodes support wireless communication and they carry their power supplies with themselves, so technically nothing can limit their movement.

3.5.4. Heterogeneity and Homogeneity of nodes

In the homogeneous WSN, all members are same. That is fully supported to have nodes that act as an alternative to each other when needed. On the other hand in heterogeneous WSN due to existing standard protocols in communicating between nodes, a WSN has not to have the same set of nodes. They can be configured based on applications' requirements. For example, some nodes can have more powerful transmitters and some other may have the batteries ten times heavier and more capable than others. Consider that there are thousands of cheap nodes and only two expensive and well-protected nodes. The diversity of features can be provided by hardware options or software limitations.

3.5.5. Scalability

Wireless Sensor Networks use their wireless communication and mobility properties to deploy nodes in a large-scale rapidly. Also, they are very talented to collapse the network in smaller scales. Nodes don't need any third party infrastructure to make connectivity and they are flexible to expand.

3.5.6. Ability of withstand harsh environmental conditions

Naturally, WSN nodes can be modular systems; therefore components of them can be chosen or designed based on environmental conditions. Losing a cheaper device are more reasonable than the expensive instances, so they can be used with less caution and they can work until last moment they can.

3.5.7. Ease of use

Nodes can be programmed easily and as they are autonomous, putting them in proper place seems enough.

3.5.8. Virtual layers

Without needing any physical facilities, creating logical and virtual layers is possible. Increasing security and utilization are two motivations to use virtual layer concepts. Layers can be fully separated or can have controlled communication with cross-layer techniques.

3.6. Conclusion

Communicating between two nodes without any third party is called ad hoc. This structure is suitable for mobile nodes with dynamic topologies. WSN is a sample of ad hoc networks that each node can perform its tasks by itself. These autonomous nodes create networks with flexible structure and various usages that static structures are useless there. A WSN node can be configured by choosing specifications of the processing unit, sensing unit, transceiver unit, and power

supply unit depending on applications' requirements. Knowing components' abilities and limitations help developers and administrators to manage a successful network for various environments, scenarios, and goals.

To create an autonomous system (Node) which is capable to collaborate with other systems (WSN Nodes), a set of hardware and software components should work together. Changing logical software on standard hardware are more common. At the next chapter, methods of making logic for an Arduino based node are explained.

CHAPTER 4

SMORITHM
(A FRAMEWORK FOR IMPLEMENTING WSN AND IOT BASED APPLICATIONS BY STATE MACHINES)

Highlights

- Goals, reasons and motivations to create the SMORITHM
- Architecture and software components of framework
- Related works

These days the machines are our new family members. They assist us to perform our works more accurately and more quickly. They appear in all aspects of life and consolidate their positions more than before. Everyday novel ideas come to brains and they become bridges for next ideas. By analyzing scientific topics, scientists resolve questions in their specialty fields. Sometimes these solutions lead to create an automated machine. For example in the medical, a tiny device is able to detect blood pressure and body temperature of a patient in his house and send them to a medical care center.

Plug and play devices which they are prepared to use quickly by ordinary people, get popular rapidly. They are available in online markets and customers can order them easily. This is a pleasant idea, Gather what you want and mount them in your custom machine. This solution represents a brilliant, economic and swift approach to create a prototype and feasibility proof in research stages.

One of the most fundamental subjects in a commercial or industrial problem is that how a requirement is mapped to an implementation. Sometimes developers need to develop a solution at a reasonable duration and sometimes they need to review and redevelop the implementation frequently. In logical problems, tracking both syntactic and semantic issues concurrently is vital. Fortunately, in this case, it has been developed a few tools and methods last years. Two of the most favorite concepts are visual scripting for syntactic problems and state machine for semantic and decision-based problems.

When a logical and smart machine is assessed, it is expected to have a machine that is able to make decisions. A set of decisions that a smart machine can make, represents the abilities of that machine. Generally, an autonomous system makes decisions based on inputs and pre-initialized values. So if in a machine, some inputs are mapped to some operations, an automated machine has been created. But usually, problems and requirements in real life are more complex than a simple binding between inputs and operations. In other words, some input values may affect some decisions and defining proper reactions seems necessary.

Many examples of autonomous machines are present in our daily life, like a coffee machine or a smart house system. The mutual aspects of them are a decision system and operating devices. They get information from the environment, process them,

connect with other machines, decide and perform some operations. In technology terms, they are named wireless sensor network. The nodes of a WSN have sensors, minimal processors, radio transceiver, interface ports, and power supply. By the nature of these networks, power consumption is a hot point to discuss. Another feature of these networks is offering inexpensive and economic devices.

Related to the topology and design of them, data-process is performed at various levels. Some processes need to execute in nodes, some other may be in base stations or servers. But avoiding from resources' overconsume is very important. Reducing the costs and prices is a great goal of todays' develops. This is interesting that anyone able to create a machine with minimum cost. Developers can focus on their specialty fields instead of learning the deep concepts of electronic and software programming.

Using modular hardware is popular now. Creating a modular and easy to understand modular software is our primary goal. This thesis was defined to realize a framework to create autonomous systems with Wireless Sensor Network and IoT approach. The framework was designed base on state machine and output generator for embedded systems too.

In result, state machine was our best choice to handle the decision making system of autonomous nodes of WSN. Moreover, as it is mentioned in following parts, a framework has been developed for this thesis was named "SMORITHM". It is composed of several modules and subsystems like diagram designer, state machine handler, code generator and etc. There good references for each of this parts in the literature and some of them are described in related works.

The goals of this thesis and reasons are explained in this part.

4.1. Goals of the Proposed Framework and Reasons

The primary goal of this thesis is making a framework that it can realize state machine and visual scripting at the same time. This goal planned for resolving these problems:

Semantic modeling of a problem can be illegible. Similar to programming, semantic modeling should retain its own language. This model can be in a text

document or in a markup oriented drawing. When the topic goes around a small problem, developers are talented to achieve dominance in all aspects of their problem. But when a problem grows as well as a complex problem, they need to examine work and data flow to track the correctness of that problem. Among plenty of comments, conditions, and codes and to do lists, performing that is extremely hard. Notice that in these cases a great deal of energy is consumed to organize the "ifs" and "conditions".

Writing code and creating software programs can be difficult for non-programmer experts. These days, smart devices with various usages in diverse fields are being invented. Many researchers are a specialist in their own fields but developing a smart device needs some more knowledge in the computer and electronic. One of the most important of them is programming for those devices. Each hardware architecture needs its proper structure, but the logic of all is the same. This is so good for developers that get outputs those are independent of the platform. This framework generates the standard output for defined platforms without coding.

Understanding and efficiently transferring the practical knowledge of an innovative solution can be time-consuming for various members of a developing team. Probably the development team of a device or a network is typically made of different members in various particular professions. The members in these teams focus on their fields and they establish their style to documenting the plans. But the final product should work properly with the plans of members. A common language can synchronize them, and anyone can evaluate the logic in a standard modeling. For new members, studying the state diagrams and visual scripts are easier and with minimum learning, they can review and understand the project's logic individually.

Extending and recoding a solution through a significant number of codes can be hard. Consider a rather extensive application with many items. In the case of extending or semantic debugging, the developers have to carefully track long-tailed flows codes. Typically, some parts of codes are related to other parts of that, and if a part is changed, it may affect other parts. Therefore altering a part may change a queue of codes. As a result, it takes exponential time.

The logical layer should be separated from the implementation layer. In the modern age, developing a successful product never is stopped. Everyday people face a new version of products. In some cases, changes have been seen in analysis and design and in some other cases, modifications in implementation are observable. Marketing motivations may force the developers to release cheaper versions with light configurations. It means the same logic in different bodies. When developers want to generate various outputs for various devices, providing a separate logical layer from implementation layers with appropriate output generators is useful for them.

A stable and equivalent out coming behavior should be seen for real-time and on-demand systems at both single and multiprocessing systems. Cheaper processors typically have more simple architecture. Reasonable expectations from a smart device are concurrent execution and on-demand response. While a part of tasks is watching and listening to the new statuses, the other part may have to run concurrently. Ordinarily, costs of processes in these scenarios are so minimal, but tasks desire to seize possession exclusively. It deprives the possible chance of execution from other tasks due to long delays. So regardless of having one or more processing cores in hardware, framework splits task executions into atomic blocks. Then the framework tries to run them fairly one after one by time sharing in a loop cycle, and in result utilization of resources is increased appreciably.

A standard protocol for communicating various nodes with different attributes and same interface formats should be designed. Naturally in Wireless Sensor Networks and IoT networks using heterogeneous nodes are popular. Each node may have its own logic but they want to communicate with each other or base stations. In IoT, HTTP has been applied. But in WSN transferring the raw data is common. This framework provides a tool to customize the data template. In a network, it is possible to reserve some templates and share them between others to make connectivity with anyone who implements those templates outside of the network.

The structure of SMORITHM, prevents some common mistakes in software development phase. Also, the performance of different parts is assured by predefined modules.

Besides a bunch of features and abilities for managing the autonomous systems, this thesis is focused on developing a platform that is convenient and customized for wireless sensor network autonomous nodes.

4.2. Architecture of SMORITHM

It is typically a significant feature that the final implemented layer does not depend on the logic layer. This independence guarantees that the minimum ordinary standards have been provided.

In addition to the available layers and modules, a three layer architecture which is presented in Figure 4.1 and mentioned below is designed and implemented for this framework.

- diagram base
- state machine manager
- exporter manager

Diagram base gives a capability to the framework to use visual abilities like design by drag and drop. State machine manager includes all logic parts and manages all state machine subparts. In the exporter layer, by selecting a target platform, the framework generates proper output. This output can be transferred as a program for a device or a feed for a simulator. Each generator is implemented separately and the output of each generator may contain codes, compiled objects or data files.

All layers of the current version of this framework were written with C# .net and WPF. An Arduino generator was implemented too. Because of resources` limitation and requirement to consume them as less as possible due to the nature of WSN and IoT, this exporter generates pure c++ code to copy directly on a device. The reason for this approach is that we want to eliminate the overhead of the middle layer in Arduino devices.

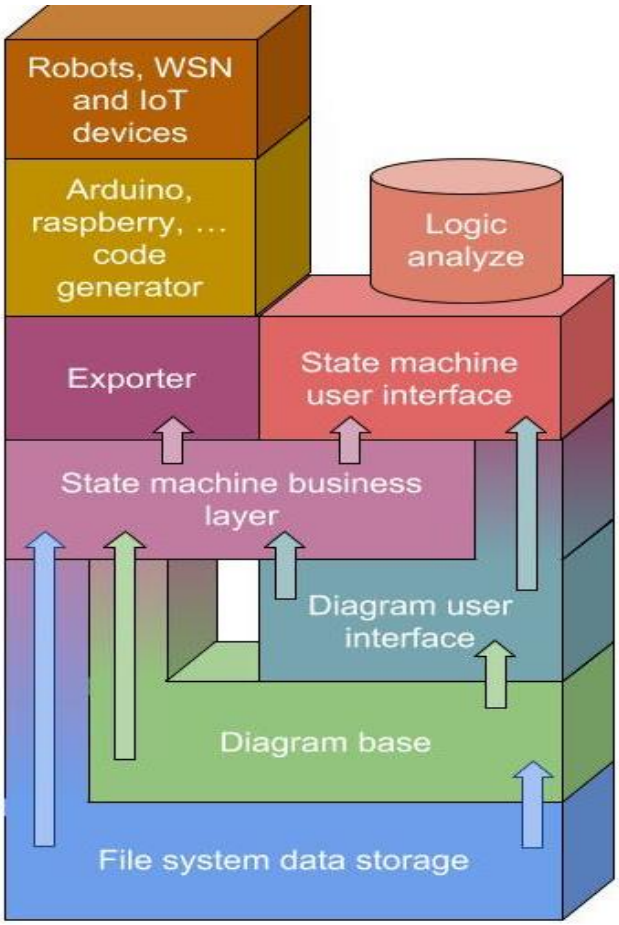


Figure 4.1: SMORITHM architecture layers

The main workflow of the work is explained here. Any working output should be held in a logic unit. At least one or more diagrams should be designed inside that. States and transitions are appended. The framework generates an output. This output is transferred into a target device or platform.

There are a few definitions and concepts. These are implemented as components and parts in the framework; the main home page of the SMORITHM application is shown in Figure 4.2. For better using, it is recommended that to have a background knowledge about state diagram, sensors and processor main concepts (Arduino in this case).

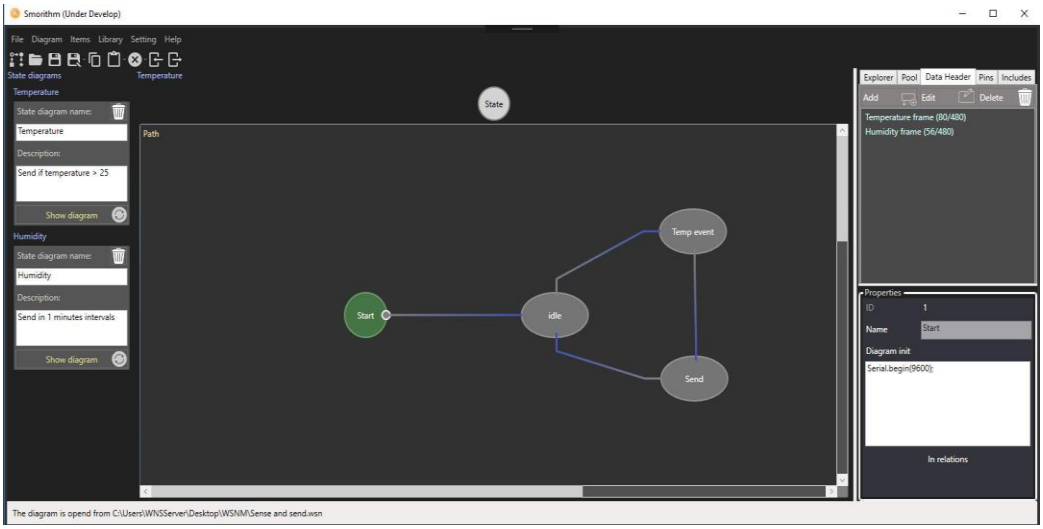


Figure 4.2: Main page of SMORITHM

In the following sections, a brief description is explained about main concepts and components.

4.2.1. Logic Unit

The concept of logic unit has taken place on top of all concepts of SMORITHM framework. A logic unit is used as a container for all parts of a solution and it can contain a few state diagrams. All of state diagrams in a logic unit (as an example is represented in Figure 4.3) should be run concurrently. If the processor has not enough processor cores (multi thread support) to execute them simultaneously, the state machine framework splits them into smaller time-shared slices and runs them one after other.

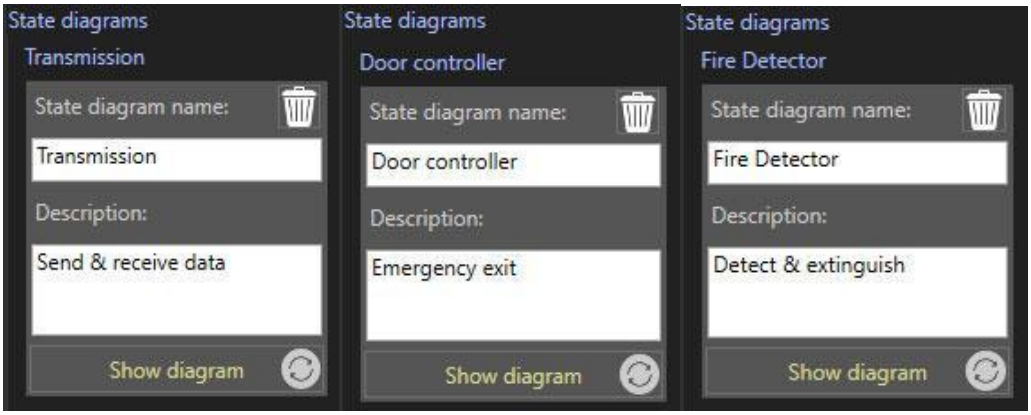


Figure 4.3: State diagrams in a Logic unit

4.2.2. State Diagram

To design a logic, the state diagram is the right place and contains all required elements. A state diagram start its work from a state and the state machines decides what is the current state of running algorithm for each state diagram. Each state diagram can contains some states. Figure 4.4 represents an example of a state diagram to extinguish fire.

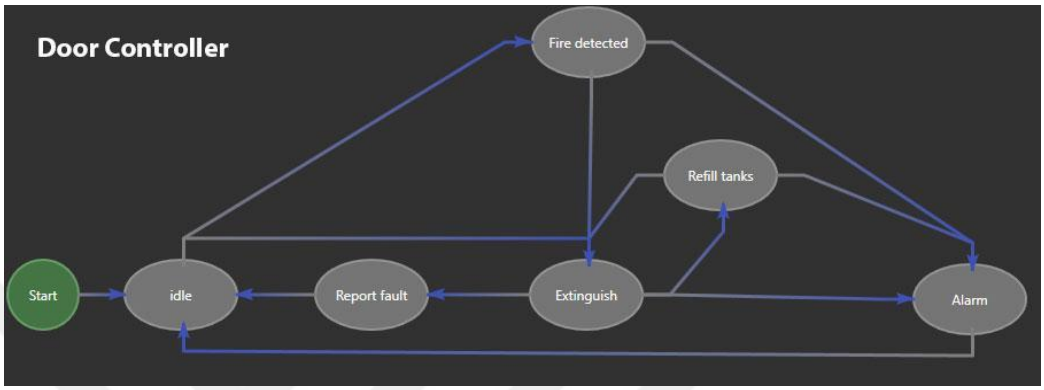


Figure 4.4: State diagram

State

A state is the smallest processing block in this framework. There are two kind of states: “Start states” and “intermediate states” (shape of them are shown in Figure 4.5). Each diagram has only one start state which is the entry point of diagram. In addition the start state can has only one output transition. In a diagram at runtime, the framework specifies a current state and checks output transitions` conditions at the beginning of each process loop cycle to ensure that is it needed to change current state or not.

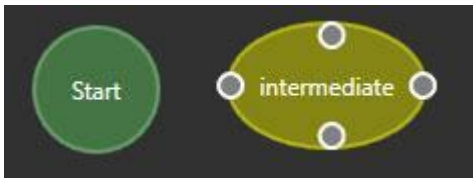


Figure 4.5: State types

Intermediate states has three default functions and are described in following part,

- **OnStateEnter**: this function will be called immediately after selecting a state as a current state. In other word, this function is the first method of a state that

will be called after getting turn for that state. Each time the framework decides to change the current state to a target state, the OnStateEnter function of target state is called only once. But when a state loses the turn and recapture it again, it has a chance to run it again.

- **OnStateUpdate**: this function will be called one time for each processing cycle. Until a state is the current state of a diagram, in each cycle it will be called. If we have more than one diagram in a logic unit, the OnStateUpdate method of current state of each one will be called sequentially. It is a proper place to check some events` raises. E.g. listening to an incoming received data from a radio transceiver and assign it to a data packet variable.
- **OnStateExit**: just before exiting from a state, this method will be called.

These three functions which are shown in Figure 4.6 are available for all intermediate states separately. The content of these functions for each state can be empty or fulfilled by some instructions. Each function is executed in its own turn. The start state of each diagram has only one function; these function are called just only once immediately after device is started and logic unit is initialized.

Figure 4.6: Functions of the intermediate state

4.2.3. Pool

There is a variable pool to hold shared variables. These variables is common and accessible for all diagrams. Any diagram of a logic unit has permission to read and write into them. Developers can define a variable in a type (single or array) and use it in diagrams. A screenshot of Pool panel is represented in Figure 4.7.

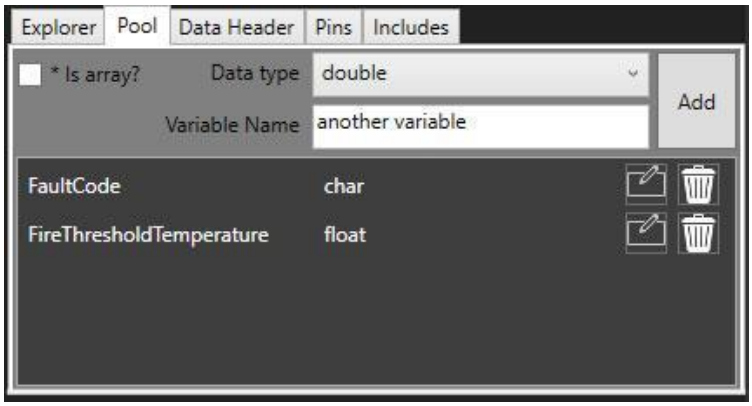


Figure 4.7: Pool management panel

The primitive data types those are used in this framework are listed in the following Table 4.1.

Table 4.1: Pool variables' data types

Data Type	**Size (bites)**	**Example**
bool	8	0
char	8	-124
unsigned char	8	251
double	64	$1.7977x10^{308}$
long double	96	...
float	32	$3.4028x10^{+38}$
int	16	-32768
long int	32	-2147483647

unsigned long int	32	4294967296
short int	16	32768
short unsigned int	16	65536

4.2.4. Data headers

Sometimes it is required to define a custom and a little more complex data types. Especially if developers want to define a custom data structure or a data format for send or receive it via serial port or radio transmitter, this part (Figure 4.8) is so interesting for them. For easier use, there are 4 eight bites predefined variables in data templates. The value of these variable are filled by framework.

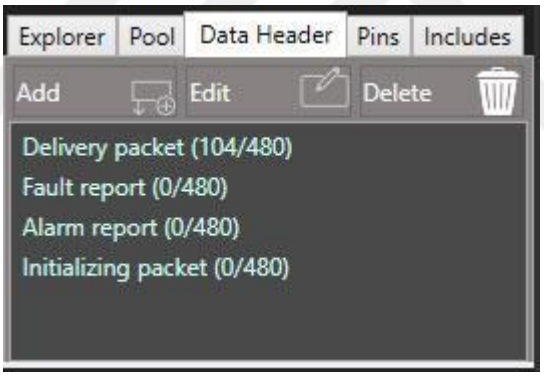


Figure 4.8: Data header management panel

Preset variables include four following types.

1. LQI (Link Quality Indicator): the value of this variable is a metric of the quality of received signal from radio transceiver.
2. RSSI (Received Signal Strength Indication): Checking the value of this variable is useful to detect the free wireless channel and etc.
3. Mode: This variable has been created for distinguishing between various data templates those are designed by the developer.
4. Data length: This variable represents the length of the data packet.

Figure 4.9: An example for data header structure

The configuration of this section is set on release time and has a dynamic structure to determine the limits. Similar to Pool, it is possible to design a custom data templates with primitive data type (an example is shown in Figure 4.9). Once a data template is created, it is available for developer to define a variable from new generated data type in Pool. Also, the content of this template is recognizable for other devices with the same Mode ID and template format even though if they are not in the same logic unit.

4.2.6. Pins

One of the most useful interfaces to read or write between a microprocessor and an electrical device is pin. By changing voltage, the levels of voltage is mapped to a numerical value. Analog pins and digital pins are two type of pins. Digital pin can take only two values as HIGH and LOW. This type of pins is proper for logical circuits, for example a button. The other type of pins is Analog. The voltage of this type can be changed between a min and a max value. This type is appropriate to get a numerical value, for example a value from a temperature sensor. In this example, the detail of sensor can be found in its data sheet. Similar to an example that is represented in Figure 4.10, if a sensor can measure between -10 degree Celsius and 150 degree Celsius, the value of this pin can be changed between 0 and 1023. So it is concluded that 0 value of pin is equal to -10 degree Celsius, 1023 means 150 degree Celsius and 511 means 70 degree Celsius.

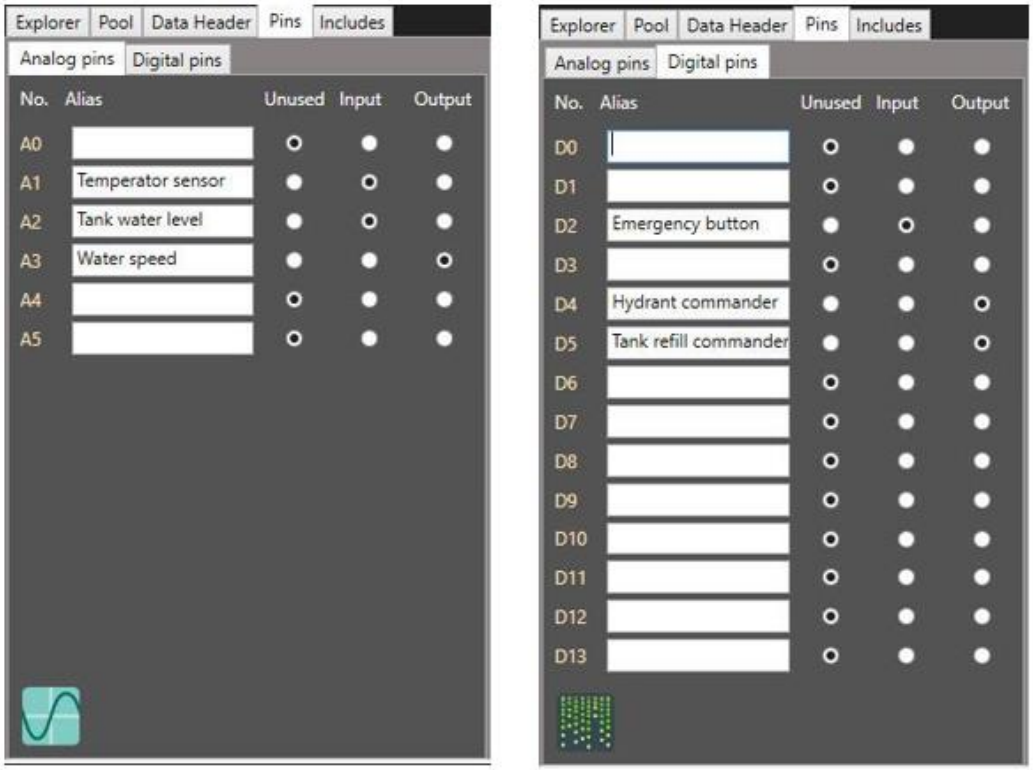


Figure 4.10: Analog and Digital pins configuration panel.

This framework`s pins is designed for unidirectional pins. Each pin can be set as input or output pin. Output pins can be set by processor to send a value to an external device and input pins can be read to get a value from external device. Also they can be taken alias names and can be used in transition diagrams.

4.2.5. Transitions

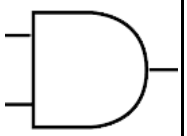
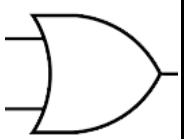
A transition is a directional relation between two states that connects a state (source state) to another state (target state). As pointed previously, the state machine must decide which state has enough rights to be chosen as the next current state. The start state is a special type of states and it is the first current state of all diagrams that is initialized by the framework at the beginning. The transition that exit from a start state determines the next current state after start by following the target state of that.

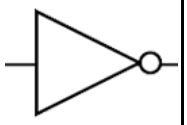
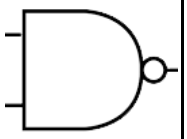
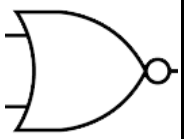
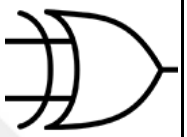
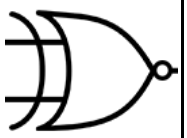
Each transition is capable to hold some conditions. If the consequence of all conditions is true, then the transition tells to the framework that the target state of the transition is the proper candidate to be the next current state of its diagram.

Transitions have a special diagram to contain their conditions. Transition diagrams have an exit point. All outputs of a transition diagrams must be led up to this point (directly or indirect).

An exit point, all pool variables, all output pins, constant values and a set of operators are appeared in transition diagram toolbox. These elements has an output data type. Generally these data types are divided into two logical and numeric sets. Before providing a transition, the framework ensures that the output data type of source state is compatible with the input data type of target state. Therefore each operator between listed operators in Table 4.2 permits to specific elements to connect to them.

Table 4.2: SMORITHM operators' list

	Operator name	**Category**	**Input data type**	**Output data type**	**Icon**
1	Exit	Exit point	bool	-	
2	AND	Logical gates	bool	bool	
3	OR	Logical gates	bool	bool	

4	NOT	Logical gates	bool	bool	
5	NAND	Logical gates	bool	bool	
6	NOR	Logical gates	bool	bool	
7	XOR (not equal)	Logical gates	bool	bool	
8	XNOR (equal)	Logical gates	bool	bool	
9	Bigger	Mathematical gates	Numeric	bool	
10	Bigger or equal	Mathematical gates	Numeric	bool	

11	Equal	Mathematical gates	Numeric	bool	
12	Not equal	Mathematical gates	Numeric	bool	
13	Less	Mathematical gates	Numeric	bool	
14	Less or equal	Mathematical gates	Numeric	bool	
15	Sum	Mathematical operators	Numeric	Numeric	
16	Subtract	Mathematical operators	Numeric	Numeric	
17	Multiply	Mathematical operators	Numeric	Numeric	

18	Divide	Mathematical operators	Numeric	Numeric	
19	Exponent	Mathematical operators	Numeric	Numeric	
20	Root	Mathematical operators	Numeric	Numeric	
21	Mod	Mathematical operators	Numeric	Numeric	

Unlike the state diagram, which all states begin their work from “start state”, in the transition diagram all elements attend and lead up to the “end point”. If any element is not connected with a direct or indirect wire to the “exit point” (unlike Figure 4.11), then that is not counted in the decision system.

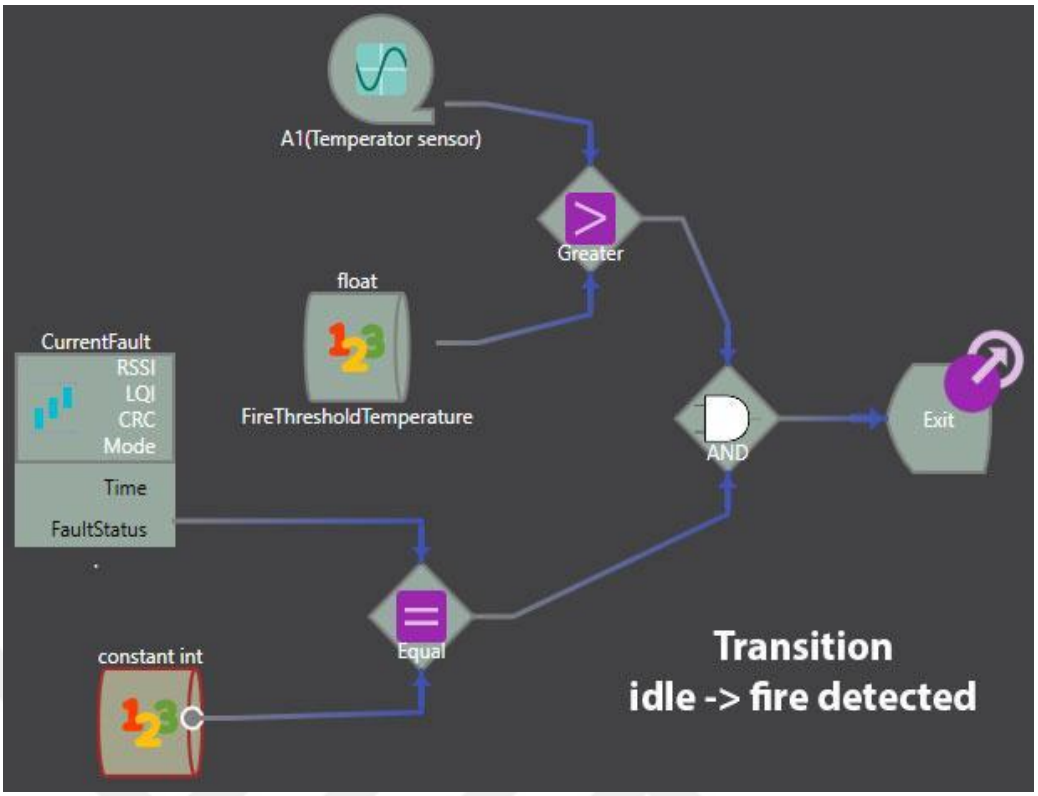


Figure 4.11: An example for transition diagram

4.2.6. Exporter

By developing a proper Exporter, wide range of outputs can be extract from this framework. Generally two types of Exporters can be developed. The first category includes the code generators that generate raw source codes to copy in target devices directly. Second type of Exporters compile the Logic unit and export objects in preset formats. In target device a version of SMORITHM driver must be installed. The output object is imported to device and the driver executes that.

The Exporter is the brain of SMORITHM. Depending on target platform, all items of logic unit are translated to corresponding outputs by the Exporter. Because of being isolated state machine manager from Exporter manager, having several Exporters for different target platforms is possible. By the time this thesis was completed, an Arduino Exporter has been developed to generate C++ codes from Logic unit. The other version of the Exporter has been planned to develop for Raspberry PI with Linux and Windows IoT.

4.3. Related works

Using WSN and IoT are growing rapidly and typically these kind of networks have includes a vast range of technologies with many features. There are a lots of studies and products individually for each of those technologies or mutually for collaboration between some of them; meanwhile, they are observable from several aspects (Kim and Hong, 2004; Miao, et al., 2012; Gray and Scherer, 2014; Aslan, 2010). Studies on operating frameworks for WSN and related theories, visual scripting and state machine based applications are focused in this part.

4.3.1. A Programming Language for Networked Embedded Systems

The nesC is the name of a programming language for networked embedded systems. The TinyOS which has a component-based architecture has been designed with nesC. Furthermore, it provides a simple event-based model and split-phase operations. nesC design descriptions including component specification, implementation, concurrency, and atomicity have been explained in the paper (Gay, et al., 2014). As it been claimed in this paper, nesC whose most of the components have been written as small finite state machines, obtained a better performance than other sequential languages.

4.3.2. Hierarchical Finite State Machines

A hierarchical finite state machine based implementing is the main subject of study (Krämer, Bader and Oelmann, 2013) was focused on optimizing interrupts. This framework has a three-layer architecture which is observable in Figure 4.12: Application Layer, Support Layer and Hardware Layer.

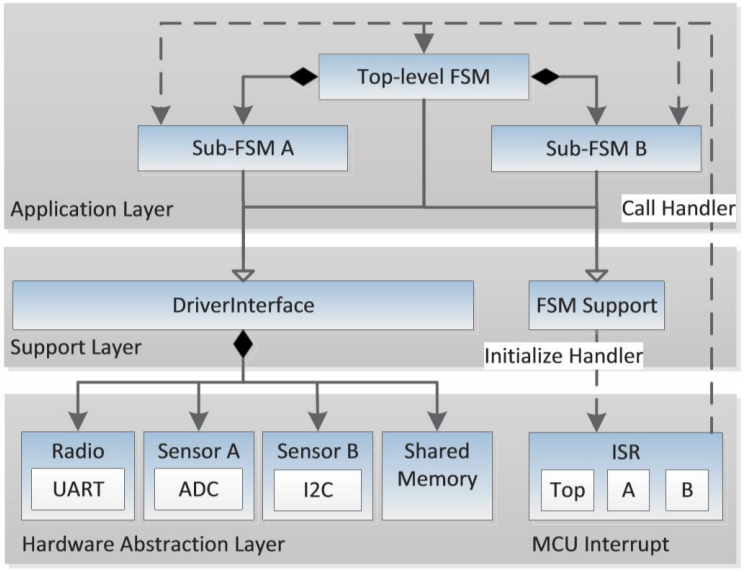


Figure 4.12: Three-layer architecture for HFSM

4.3.3. A Reconfigurable Interface for Industrial WSN in IoT

The study (Chi, et al., 2014) explained a solution to decrease the complexity and difficulty of writing data collection programs in an industrial WSN in IoT environment. The main idea of this work was on combining CPLD (Complex Programmable Logic Device) and the application of wireless communication based on the IEEE 1451 protocol. An interface device was anticipated to make a connection between phenomena by sensors and other nodes via serial port and ZigBee wireless communication.

4.3.4. Learning Finite-State Machine

“Learning Finite-State Machine - Statistical and Algorithmic Aspects” is the title of a Ph.D. thesis in Universitat “Polit`ecnica de Catalunya” which was written by “Borja de Balle Pigem” that divided into State-merging algorithms and Spectral methods and the mathematical theories were discussed. The first chapter included State-Merging with Statistical Queries, Implementation of Similarity Oracles, Learning PDFA from Data Streams Adaptively and State-Merging on Generalized Alphabets. These four parts represented a set of methods that are useful for reduction and minimizing different types of state diagrams. The second part

included A Spectral Learning Algorithm for Weighted Automata, Sample Bounds for Learning Stochastic Automata, Learning Transductions under Benign Input Distributions, The Spectral Method from an Optimization Viewpoint and Learning Weighted Automata via Matrix Completion (Balle, 2013). Learning methods were pointed out in this chapter.

Machine learning has a unique role in Artificial Intelligence which empowers logic units of intelligent machines. Considering the efficiency of state machines in a real-time system with streaming input data, machine learning is useful for both automatically designing and/or optimizing manual designed state diagrams. This thesis can be a nice inspiration for future works of SMORITHM.

4.3.5. State machine providers (YAKINDU State-chart Tools)

By the definition of this application in its website, YAKINDU State-chart Tools (www.itemis.com, 2017) provides an integrated modeling environment for the specification and development of reactive, event-driven systems based on the concept of state machines. This application has been developed by “itemis” company and has a few examples about Robot control, microcontroller support, system simulator in Dynamic Probabilistic Safety Assessment, Python code generator, Java code generator and so on. This application provides for features for state machine based programs which is shown in Figure 4.13.

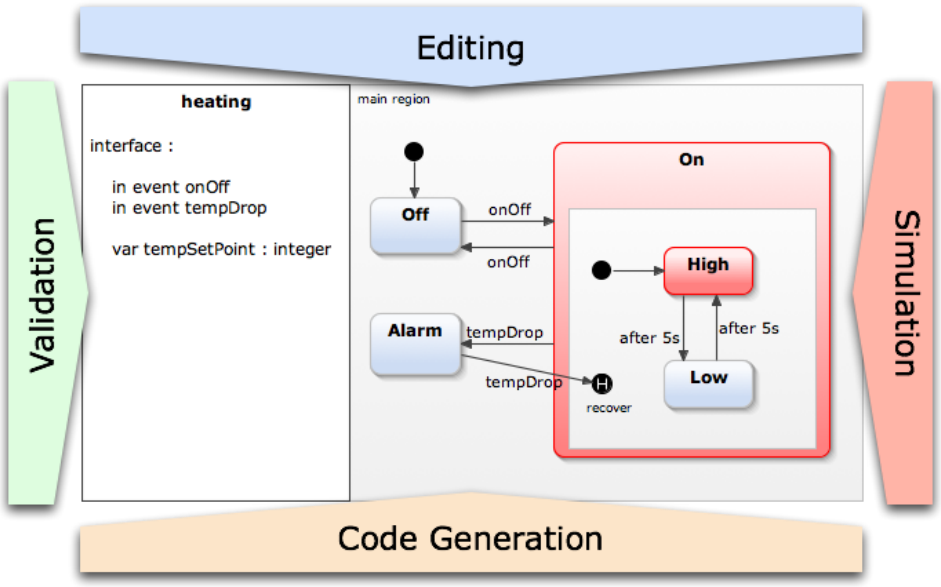


Figure 4.13: YAKINDU primary features

YAKINDU state-chart tools is a general purpose application for state charts with various code generators. Other applications like Sttcl (makulik.github.io/sttcl, 2013), Ragel state machine compiler (www.colm.net, 2017) and State forge (www.stateforge.com, 2015) are the similar applications and framework for handling state machines.

4.3.6. Visual scripting components

An interesting alternative for a text programming IDE is visual scripting applications that have been popular recent years. One of the best examples is Tinkercad (www.tinkercad.com, 2018) with an internal simulator. The Ardu block (ardublock.com, 2016) is a well-designed application to make programs by dragging and dropping. By using these kinds of applications, the readability of programs is increased; especially for who they are not familiar with coding. The other instances that support visual scripting Mini bloq visual scripting for Arduino (blog.minibloq.org, 2016). Also, some game engines like Unity3d (unity3d.com, 2018), Unreal Engine (www.unrealengine.com, 2018), and Cryengine (www.cryengine.com, 2018) provide visual scripting facilities for their developers. The schema of SMORITHM's state diagram designer is similar to the Animator Controller component of Unity3D.

4.4 Conclusion

The SMORITHM is a framework to design logic units, especially this is so convenient to design and develop that developers will able to use it for bigger projects than before. This potential is provided by implementing logic via state machines and export standard outputs depending on target platforms. Created diagrams are used to generate outputs by a different layer that is called Exporter.

CHAPTER 5

EVALUATION AND SIMULATION OF SMORITHM

Highlights

- Background and use cases for SMORITHM
- Introducing a sample hardware device for WSN
- Examples of implemented algorithms/protocols by SMORITHM
- Sample scenarios

5.1. Background and suggested use cases

The explained framework in this thesis is focused on creating logic for autonomous systems. Naturally, an autonomous system makes decisions and acts based on pre-initialized conditions and environmental information which usually are streams of data. For example, an automatic fire extinguisher senses the environmental temperature continuously and starts to spray the water when the temperature exceeds from a threshold. As mentioned in chapter 2, a decision performs a set of actions when the overall result of its conditions is true. With these backgrounds, here is a proper place to answer an important question:

-Why state machine is suitable for WSN?

Autonomous nodes of WSN sense the environment and communicate with other members; it is possible to exist many conditions for many decisions for a node to make. Checking conditions in a cycle with a linear processing method maximizes the count of conditional statements' calling. This cost can be increased if the operands of conditions need to be calculated. For example, detecting a specific individual by recognizing its voice consumes time a little bit more than simple instruction. Plus, the increment of decisions' count is effective. One solution is using the hierarchical structure. It means that the checking order of conditional statements is dependent on the priority of them. But in the best situation, the logic unit has to check all root conditional statements. The most optimized method to check the conditional statements is state machine; because at first, the problem is divided into diverse states and the state machine takes and checks only the statements of the current state in each cycle. In the worst scenario of performance, states have a design corresponds to a hierarchical structure, so a state machine at the worst situation delivers an efficiency approximate to the hierarchical method. Three probable statuses for decisions in various methods are presented in Figure 5.1.

During the operational run-time, an autonomous system has to check conditions continuously meanwhile the processing unit never stops. Checking too many statements of conditions even though a node is idle can exhaust the power resource. Optimizing logic unit for long term tasks (same as most of WSN tasks) is vital. So state machine is one of the best choices for WSN.

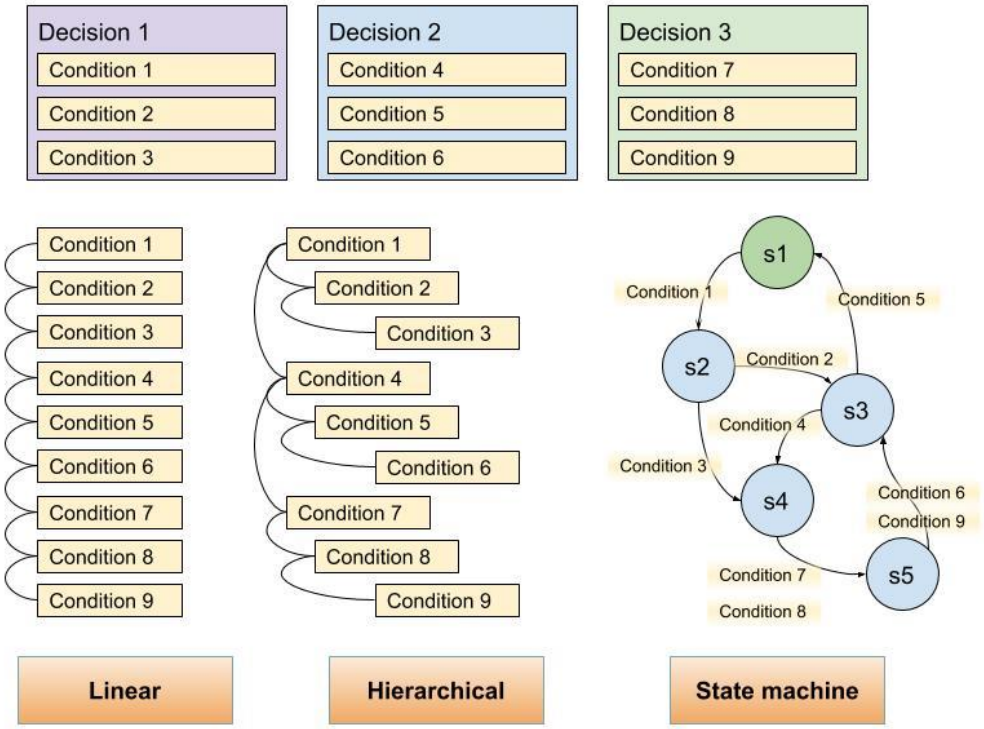


Figure 5.1: Methods of checking conditional statements

All WSN scenarios that can be implemented by sequential programming and possess an automation logic with finite states in their nature are suggested to be restudied with this framework. Also, via the capacity of data frame template management, developers can implement routing and sensing algorithms easily.

5.2. Handling a sample scenario

For example, consider a simple scenario that has been designed for a physical security system. The scenario is that: if a sensor detects any motion in front of a door, and if it is daytime, sends a warning signal to the security room, else if it is night, the system raises an alarm in the courtyard.

Two kind of sensors is needed:

1. Motion detector sensor (Detecting any motion in a sight range)
2. Photocell (illumination level detector)

Two kind of actions is required:

1. Sending warning signal to security room
2. Raising alarm in courtyard

This machine`s decision depends to more than one input.

The **pseudocode** of this scenario is shown below:

```
1   if (MotionSensorValue>Threshold)
2           if(IsDay==true)
3                   result=SendWarning()
4                   if(result==false)
5                           GoTo 3
6                   end if
7           end if
8           else if(IsDay==false)
9                   SetAlarm(true)
10                  delay(3)
11                  SetAlarm(false)
12          end if
13  end if
```

The state diagram of this scenario are shown in Figure 5.2.

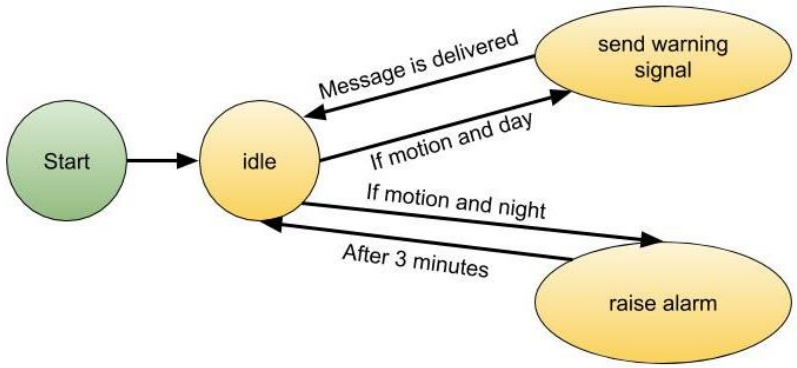


Figure 5.2: A sample state diagram

It is obviously conceived that this scenario is vulnerable in front of some exceptions like these:

- What will happen if the connection between our machine and has been lost? What is alternative?
- What will happen if we have a secondary motion while the machine is waiting for alarm ends?
- …

If these exceptions need to be handled, this logical blueprint has to be extended. The key point is that: increasing requirements may cause increasing complexity exponentially. On other hands, the amount of the code, understanding logic and debug probable semantic faults need a lot of time and effort.

Using the power of the state machine to implement logic in machines and devices is the main idea that enlightens our way. The state machine is suitable for dividing a big problem into smaller states with related transitions. The SMORITHM is a framework that converts diagrams to standard output. This standard output can be used as inputs to generate codes and programs for final devices. To avoid extra overhead of intermediate driver layers, a C++ code generator has been implemented and tested for an Arduino embedded device. Finally, the pure C++ classes and headers are copied to the device.

5.3. An introduction for sample hardware device for WSN

By the time this thesis was being written, a node was ordered by the university to sixfab (www.sixfab.com, 2018) group and a wireless sensor node was designed and made. This product contains an Arduino Nano microprocessor, a CC1101 transceiver with PCB 868 Mhz Antenna, two embedded temperature and humidity sensors, a 3.7 Li-Ion Battery with some interfaces.

5.3.1. A sample WSN node hardware specification

To represent a solid example of WSN nodes and the device is chosen as a pilot hardware for this thesis and some algorithms are implemented and tested on it,

detailed features of Wireless Sensor Network Development Board are shown in Figure 5.3.

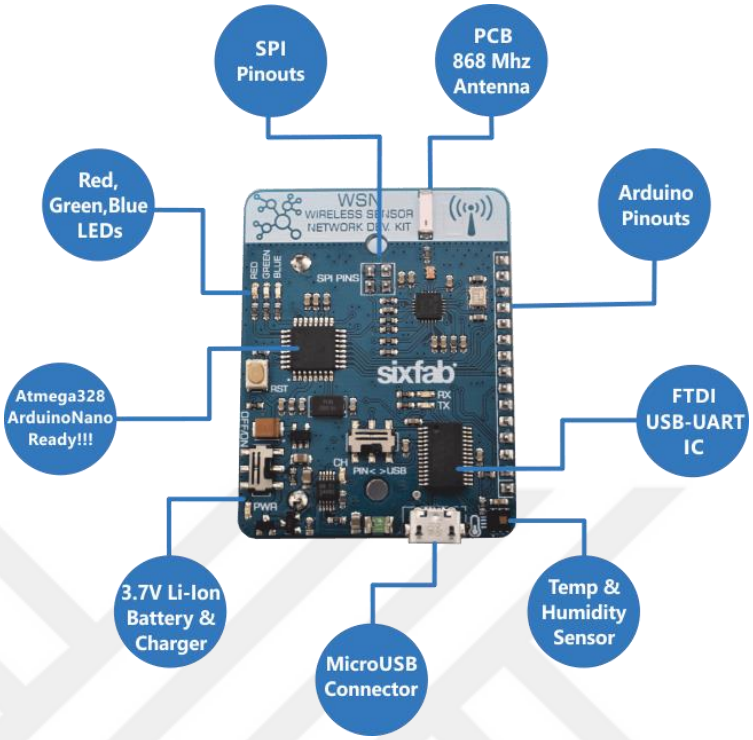


Figure 5.3: Detailed features of WSN device

- 868 MHz Wireless Communication (Up to 100 meters outdoor)
- Rechargeable 1200 mAh Li-on Battery
- Built-in temperature sensor (HDC1080) (-40 +125 C)
- Built-in humidity sensor (HDC1080) (0% 100%)
- Easy charge with micro USB socket
- Communication and programming via micro USB socket
- 3 user LED’s
- Capable to connect to external sensors via SPI, I2C, Analog Pins and General IO Pins
- Usage without battery via micro USB socket (as a static node)
- SMD chip antenna
- Uart switching (USB<>Open pins)

There are two analog and five digital pins (is shown in Figure 5.4) to connect external devices like sensors.

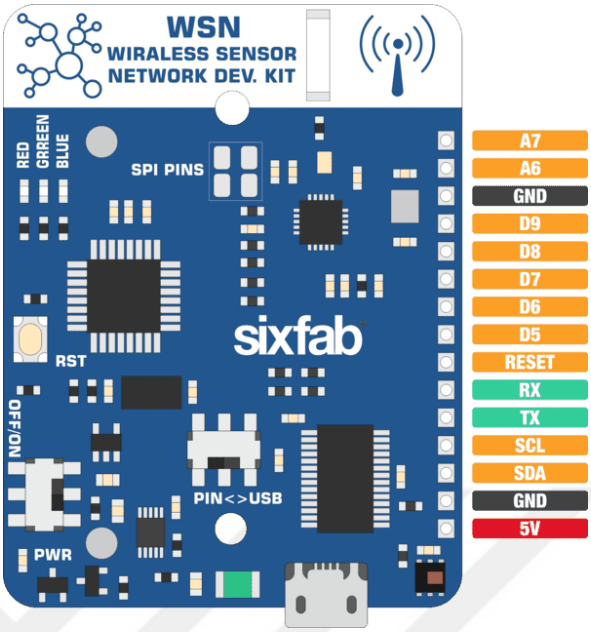


Figure 5.4: WSN device pins schematic

5.3.2. Layouts

A schematic layouts' presentation of device is shown in Figure 5.5.

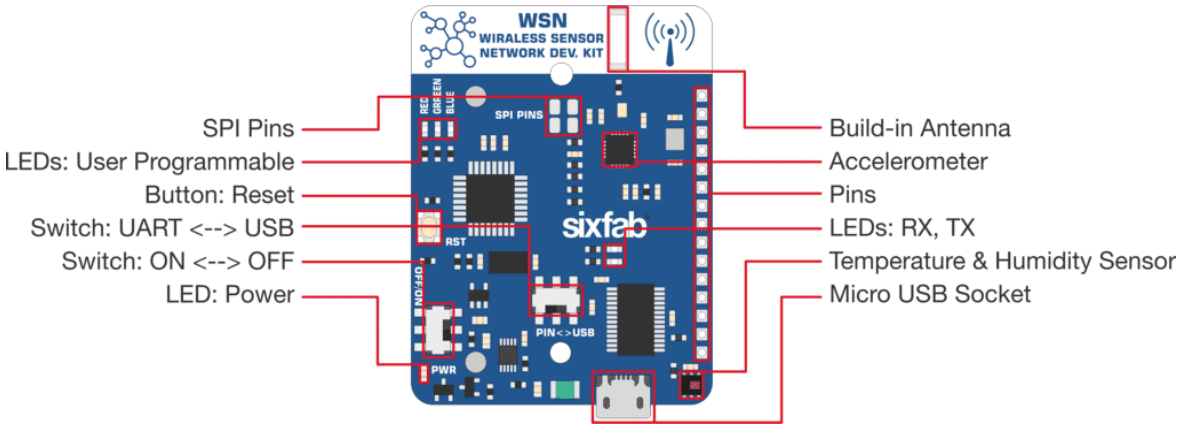


Figure 5.5: WSN device's layout schematic

5.3.3. Dimensions

This device is designed so small that be able to movement easily. The dimensions are mentioned in Figure 5.6.

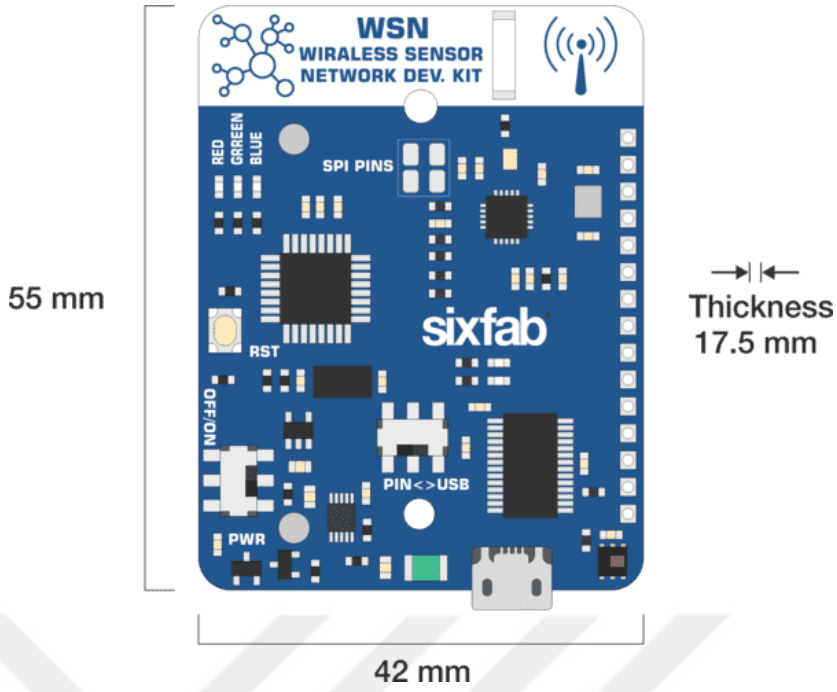


Figure 5.6: WSN device's dimensions schematic

The Arduino microprocessors are used widely in academic and learning Laboratories; also, this microcontroller is programmable with C and C++ with appropriate libraries and support forums.

5.4. Sample algorithms/protocols

A series of simple examples for the device that is mentioned in section 5.3 of this thesis are explained in this part. All examples include diagram figures and generated source codes in the related appendix.

5.4.1. A simple scenario for sequential application

Generally, each processing core is able to run one instruction at the time. The processor is allocated to instruction in its turn and is released after running. Sequential (it is also called linear) applications are composited from sequences of instructions as lines of codes that are run one after other. In the case of single-processor-core architectures, the execution of only one instruction is possible at

the same time and there are not extra to manage priorities; in the result, sequential programs have the most performance where instructions' orders are predefined.

A sample scenario is defined so that two blue and red LEDs are turned on and off repeatedly in fixed time intervals. This example shows that LEDs controller uses delay function to make a simple timing and how blocking instructions manipulate each other. In this example, Pins A1 and A3 are assigned to a blue LED and a red LED which are shown in Figure 5.7.

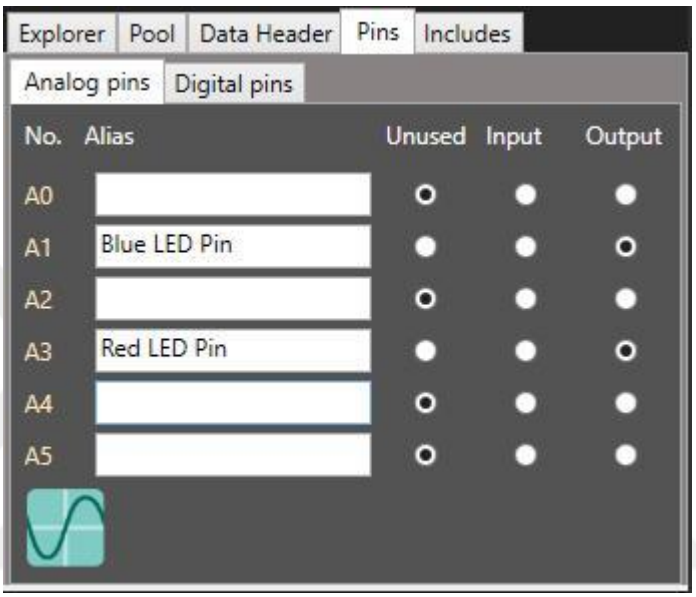


Figure 5.7. Pins configuration panel

Also, two diagrams are designed for blue LED and red LED separately. These diagrams have similar structures including two intermediate states to turn on and off LEDs which are observable in Figure 5.8.

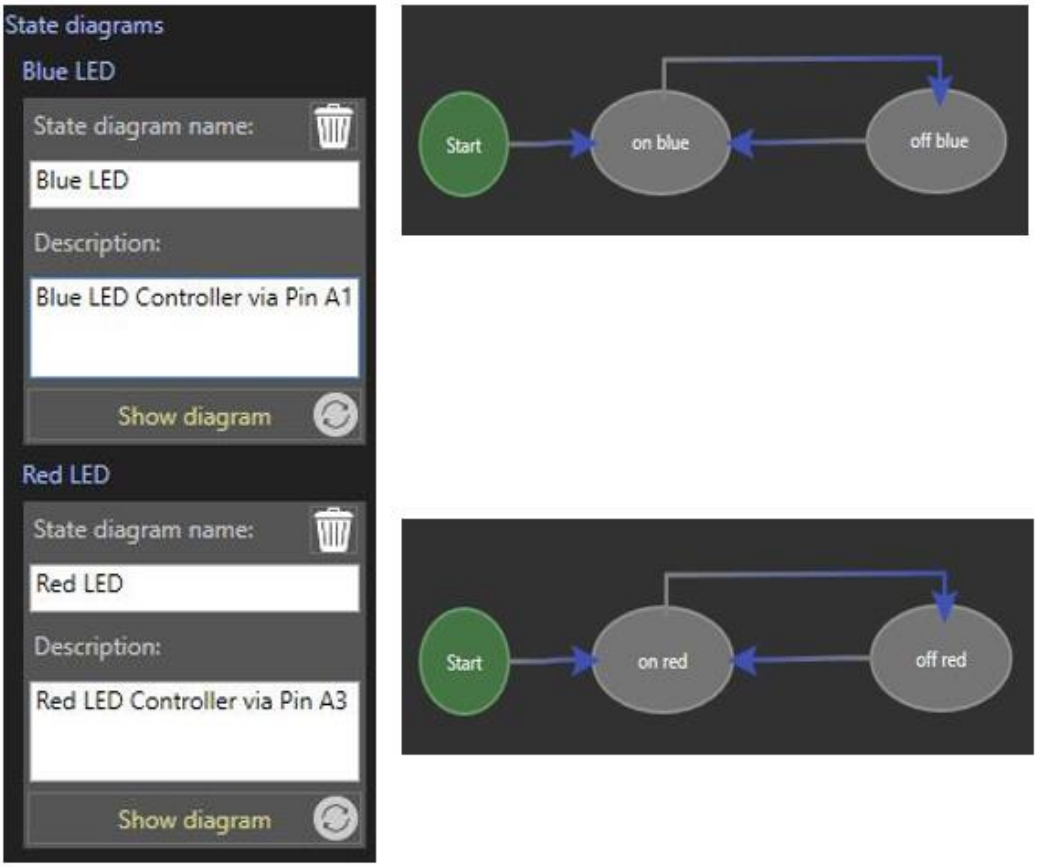


Figure 5.8. State diagrams of sequential example scenario

As empty transitions are ignored by the framework, a constant Boolean with true value is appended for each transition to pass the current state without condition in their transition diagram such as figure 5.9.

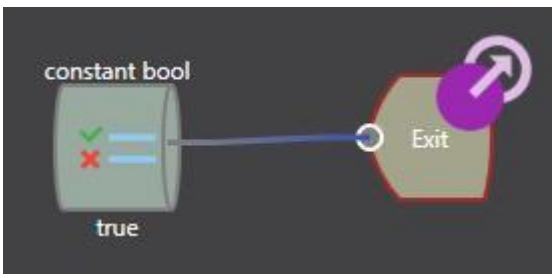


Figure 5.9. Transition diagram for all transitions

The “on blue” and “on red” write HIGH value in A1 and A3 pins switch corresponding LEDs on and other two “off blue” and “off red” switch LEDs off write LOW value. A 1000 milliseconds interval for the blue LED and a 750

milliseconds interval for the red LED are provided by delay functions immediately after pins write. Functions of these states are presented in figure 5.10.

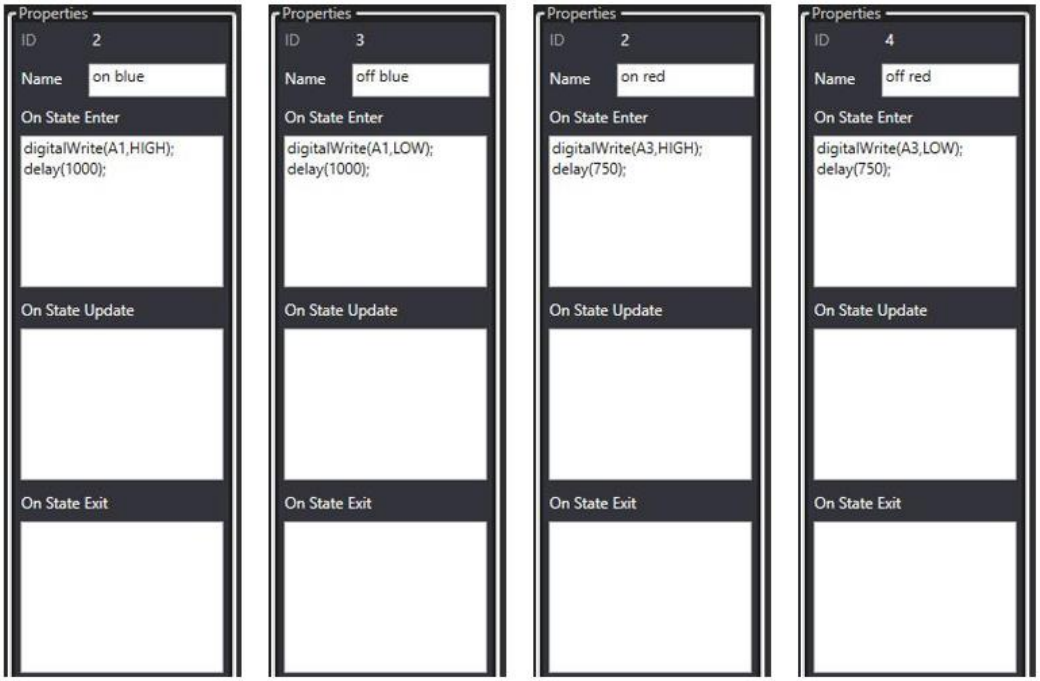


Figure 5.10. Functions of a sequential scenario

Despite LEDs have only 250 milliseconds difference in their intervals, but actually, LEDs wait until the delay function of other one been finished. But, in concurrent applications, this method cause to block the processor, especially in the cases that are required to instructions must be run in time intervals. The generated code is available in Appendix A.

5.4.2. A simple scenario for a concurrent application

To solve the concurrency problems, time-sharing algorithms are used. Instead of allowing blockage by delay functions, a simple round robin algorithm can be implemented by state timers. All parts of this example are similar to the previous example except transition diagrams and functions of states. As it is shown in Figure 5.11, in the transition diagrams the value of state timer is compared with constant numbers which are 1000 and 750 in this case.

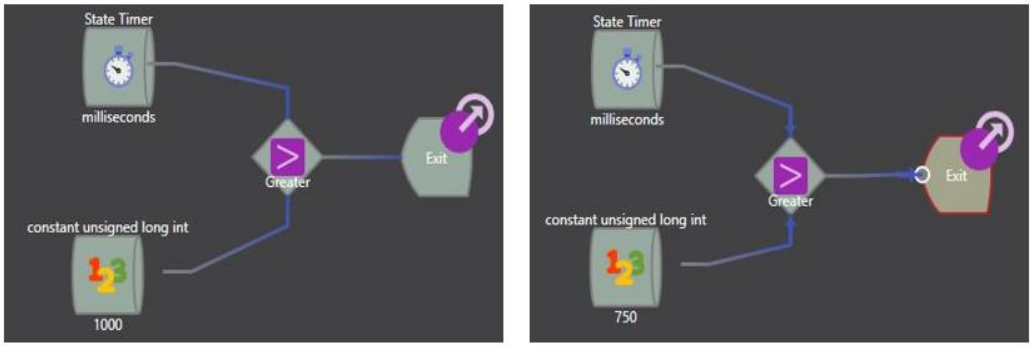


Figure 5.11. Transition diagrams

Also, the delay functions are not required anymore and removed in figure 5.12.

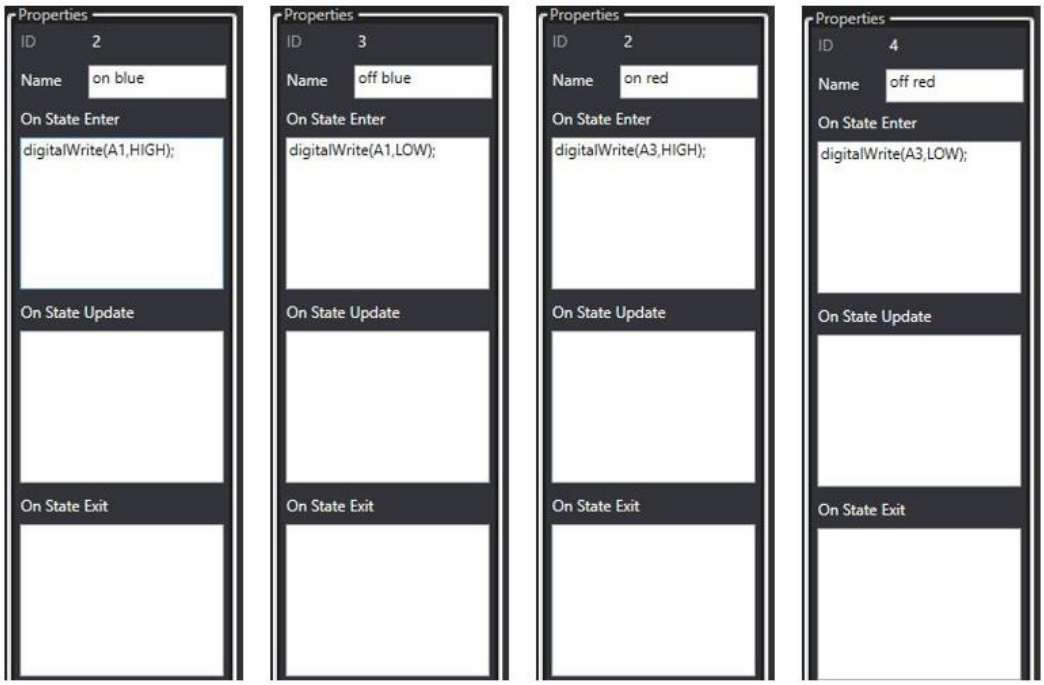


Figure 5.12. Concurrent decision controlling functions

To design on-demand systems, understanding and using concurrency is vital. Because devices wait to interrupts and events, concurrency can increase the utilization of processing resources. The generated source code is available in Appendix B.

5.4.3. A simple sensor node with the broadcast method

In this scenario, nodes are programmed to sense the humidity in fix 15-second intervals and they are sensitive to temperatures higher than 32 degree Celsius. As is represented in Figure 5.13, the logic unit of this example contains two diagrams for detecting temperature event and reporting humidity.

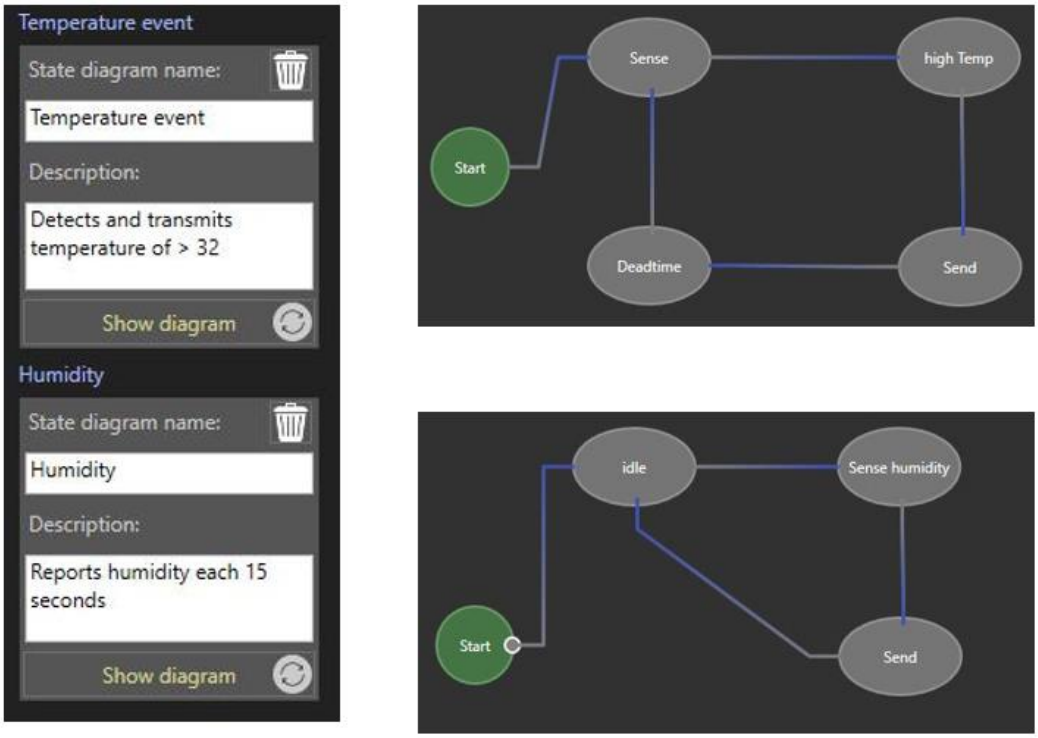


Figure 5.13. Diagrams of a sample sensor transmitter

Two data frame templates are designed to send data packets via radio transmitter and are shown in Figure 5.14.

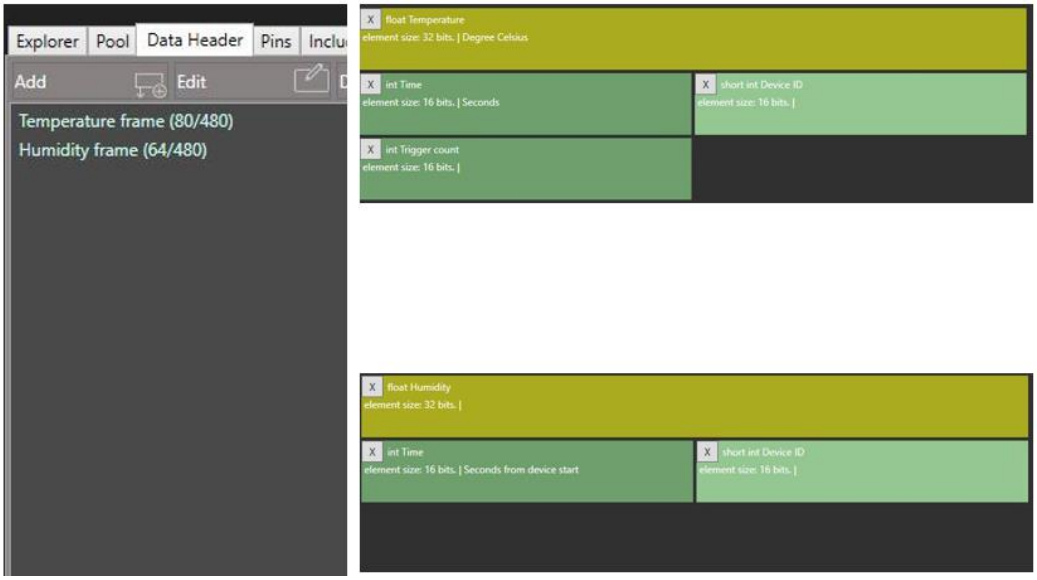


Figure 5.14. Data frame templates

Timing is applied by state timer which is shown in Figure x for the transition between “sense state” and “high Temp state” in the Temperature diagram. To prevent sending lots of temperature event, a 10 seconds dead time is anticipated to put a delay before sending again. This is similar to Figure 5.15 with 10000 (10 seconds) delay.

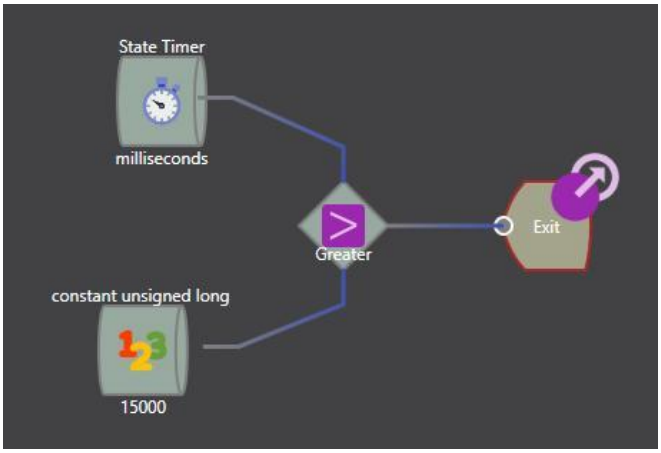


Figure 5.15. The timing transition diagram

Functions of states which are represented in Table 5.1 are used to sense and send actions. Serial print functions are used to inform the developer via serial port (USB) and can be ignored.

Table 5.1. Functions of transmitter's states

Diagram	State	Function	Codes
Temperature event	Start	Custom code	#include "ClosedCube_HDC1080.h" ClosedCube_HDC1080 hdcT; byte syncWord[2] = {199, 10};

		initializer	Pool::v_2.dfi_2=1;
			hdcH.begin(0x40);
			hdcH.setResolution(HDC1080_RESOLUTION_11BIT, HDC1080_RESOLUTION_11BIT);

	Sense	Enter	Serial.println("Temperature sensing started...");
		Update	Pool::v_1.dfi_0=hdcT.readTemperature();
	high Temp	Enter	Pool::v_1.dfi_3++; Pool::v_1.dfi_1=millis()/1000; Serial.print(" A high temperature was detected: "); Serial.print(Pool::v_1.dfi_0);
	Send	Enter	Pool::SendRadio((char*)&Pool::v_1, sizeof(Pool::v_1),1); Serial.print(" degree Celsius and was sent at: "); Serial.println(Pool::v_1.dfi_1);
	Dead time	Exit	Pool::v_1.dfi_0=hdcT.readTemperature();
Humidity	Start	Custom code	#include "ClosedCube_HDC1080.h" ClosedCube_HDC1080 hdcH;
		initializer	Pool::v_2.dfi_2=1;

			hdcH.begin(0x40); hdcH.setResolution(HDC1080_RESOLUTION_11BIT, HDC1080_RESOLUTION_11BIT);
	idle	start	Serial.println("Humidity sensing started... ");
	Sense humidity	start	Pool::v_2.dfi_0=hdcH.readHumidity(); Serial.print("Sensed humidity is: "); Serial.print(Pool::v_2.dfi_0); Pool::v_2.dfi_1=millis()/1000;
	Send	start	Pool::SendRadio((char*)&Pool::v_2, sizeof(Pool::v_2),2); Serial.print("%. The humidity was sent at: "); Serial.println(Pool::v_2.dfi_1);

The first index of buffer specifies the identification number of data headers which are selected and assigned by the network administrator. Also, a static Device ID is added for each device. The generated source code is available in Appendix C.

5.4.4. A simple monitoring base station scenario

The applications of this scenario receive data packets, and recognize their data templates and show proper message via the serial port. The data header templates are the same in this example with the previous example. The only diagram of this logic unit and its outgoing transitions are shown in Figure 5.16.

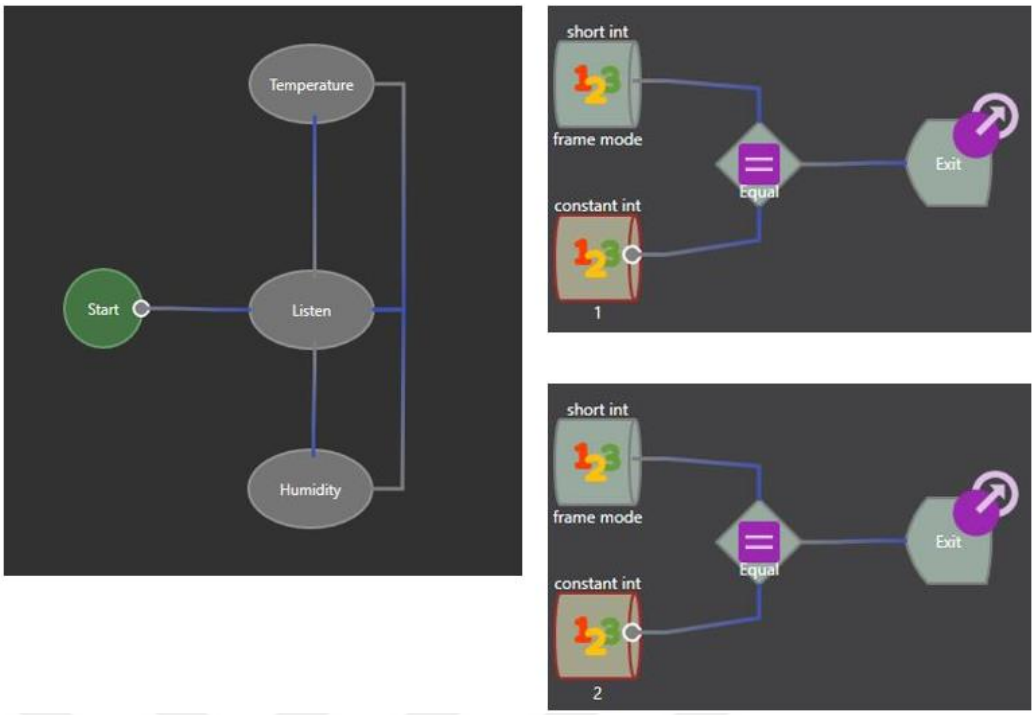


Figure 5.16. A diagram for a simple base station

The Pool manager contains three variables which are shown in Figure 5.17 and the functions of states are shown in Table 5.2.

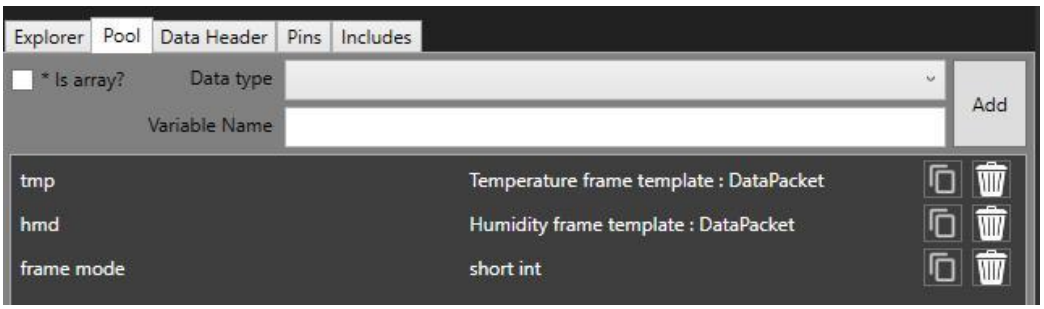


Figure 5.17. Pool manager of the base station

Table 5.2. Functions of a base station's states

Diagram	State	Function	Codes
Packet listener	Start	Custom code	byte syncWord[2] = {199, 10}; CCPACKET packet;

		initializer	Serial.begin(9600); Pool::InitRadio(CFREQ_868, PA_LongDistance, syncWord);
	Listen	Update	packet = Pool::ReceiveRadio(); Pool::v_3=packet.data[0];
	Temperature	Enter	memcpy(&Pool::v_1, packet.data+1, sizeof(Pool::v_1)+1); Serial.print("Temperature "); Serial.print(Pool::v_1.dfi_0); Serial.print(" from device "); Serial.print(Pool::v_1.dfi_2); Serial.print(" at time "); Serial.print(Pool::v_1.dfi_1); Serial.print(" ,event number: "); Serial.println(Pool::v_1.dfi_3);
		Exit	Pool::v_3=0;
	Humidity	Enter	memcpy(&Pool::v_2, packet.data+1, sizeof(Pool::v_2)+1); Serial.print("Humidity "); Serial.print(Pool::v_2.dfi_0); Serial.print("% from device "); Serial.print(Pool::v_2.dfi_2); Serial.print(" at time: "); Serial.print(Pool::v_2.dfi_1);
		Exit	Pool::v_3=0;

The source code is available in Appendix D. Common files contents are not appended in Appendix B to D.

5.5. Conclusion

In this chapter, most of the essential and fundamental concepts that required to design and manage a WSN sensor node are explained. These concepts include sequential and concurrent scenarios, writing on interfaces (pins and serial port), timing, logic unit design, state diagram designing, transitions configuration, custom data headers and templates, custom functions, reading sensors, transmitting and receiving. Many of algorithms and protocols can be implemented by these concepts.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

6.1. Conclusion

Intelligent and autonomous systems enable societies to benefit resources with less wasting and more utilization. These systems can be proper alternatives for human resources where the harsh environment increases risk, a high degree of accuracy is needed for a specific phenomenon sensing, accurate timing is required, many of resources should be deployed in an area and other reasons that determine the using of machines in some cases is preferred to humans.

The most popular autonomous and intelligent systems are WSN and IoT devices which make decisions by themselves and act automatically. They are widely used to data gathering, monitoring environments and controlling other devices and machines. A decision can be inferred from external sensed data and internal predefined logical statements. The nature of input data for sensors is continuous, and devices have to catch streams of data.

State machines divide problems into several states and only focus on a current state. Therefore, they check the minimal conditional statements of transitions to make proper decisions. Also, transitions allow states to pass current state to other states and represent better readability for developers. Different parts of a logic unit can be designed as multiple state diagrams that can be run with concurrent algorithms. Furthermore, reusing different modules is possible for the intelligent agent by saving several state diagrams of its logic unit and reloading related state diagrams in other logic units. The state machine has been implemented in their individual and independent layers.

Different devices with various platforms require their adapted software, but they may follow the same logical concepts. Also, existing a standard protocol independent from developers and manufacturers provides a vast and more flexible design. State machine layer explains what is the logic and another layer create

proper output for target platform. This means a logical unit can be used for multiple types of devices with the same behavior.

In result, we developed a framework that provides an IDE for WSN, IoT and autonomous systems' developers to design their required state diagrams in the logic unit. The final output is accessible via the corresponding exporter. This version includes an embedded C++ code generator for Arduino microcontrollers.

6.2. Future works

A successful prototype of the framework developed in this thesis. Some other developments can empower this framework and turn it into a professional solution. Some of these future works are explained in this section.

Importer

Even though the framework is capable to save and load diagrams by its format, but an open-source and standard data formats like JSON, XML or CSV is required to import diagrams from other applications.

Other Exporters

Implementing other exporters can extend the support range of hardware devices by this framework. Python is an example of common languages in recent years that is supported by many platforms.

Dynamic logic unit

In addition to raw source code generation by the framework, the other type of outputs can be a compiled object that is transferred to the target device. It can be run on a driver layer in the target device. Instead of copying source code, the logic manager can download the object of logic unit and import it. In other words, logic unit can be changed dynamically at run time. There are a few platforms for manager driver such as WinIoT and Linux which are installable on devices like Raspberry PI.

Experience sharing

This theory comes after dynamic logic unit. When a structure is made for managing distributed logic unit, subparts can be shared with other logic units. The strategy of choosing subpart priorities, merging and replacing should be determined.

State diagram from Machine learning

The learning automata is a new concept that provides state diagrams based on real feature values. State diagrams can be generated by learning methods. Also, their states and transitions can be reconfigured by learned model from datasets.

Simulator

To observe values of internal state diagrams (such as current state, Pool variables' value, data packets values, sensors and pool values, remaining power and other parameters) a simulator can be designed for one or multiple target devices with various hardware components.

Monitoring

The expectation from a monitoring system for this framework is the presenting data flow and device statuses. To implement this module, capturing some extra information from devices may be required.

Web IDE

This version of framework is implemented by ".Net" for the Microsoft Windows operating system. Other versions such as a web IDE can be developed for better access.

REFERENCES

Afshar, M. T., & Manzuri, M. T. (2011). A Novel Technique to Control the Traffic of Wireless Ad-hoc Network by Fuzzy Systems and Prediction with Neural Network. *Third International Conference on Computational Intelligence, Modelling & Simulation*, 18-21.

Afshar, M. T., Manzuri, M. T., & Latifi, N. (2011). A model for traffic prediction in wireless ad-hoc networks. *In Innovative Computing Technology, Springer*, 328-335.

Akyildiz, I. F., et al. (2002). A survey on sensor networks. *IEEE communications magazine*, *40*(8): 102-114.

Al-Karaki, J. N., & Kamal, A. E. (2004). Routing techniques in wireless sensor networks: a survey. *IEEE wireless communications*, *11*(6): 6-28.

Alkhatib, A. A. (2014). A review on forest fire detection techniques. *International Journal of Distributed Sensor Networks*, *10*(3): 597368.

Aslan, Y.E. (2010). *A Framework for the Use of Wireless Sensor Networks in Forest Fire Detection and Monitoring* **(Master Thesis). Bilkent University, Ankara, Turkiye**.

Balle, P. B. D. (2013). *Learning finite-state machines: statistical and algorithmic aspects* **(PhD Thesis). Polytechnic University of Catalonia, Barcelona, Spain**.

Bangash, J., et al. (2014). A survey of routing protocols in wireless body sensor networks. *Sensors*, *14*(1): 1322-1357.

Bellavista, P., et al. (2013). Convergence of MANET and WSN in IoT urban scenarios. IEEE *Sensors Journal*, *13*(10): 3558-3567.

Blum, C., Winfield, A. F., and Hafner, V. V. (2018). Simulation-based internal models for safer robots. *Frontiers in Robotics and AI*, *4*, 74.

Breslow, L. A., & Aha, D. W. (1997). Simplifying decision trees: A survey. *The Knowledge Engineering Review*, *12*(1): 1-40.

Chen, H., Wang, X. M., and Li, Y. (2009). A survey of autonomous control for UAV. *International Conference on in Artificial Intelligence and Computational Intelligence, AICI'09, 2*: 267-271.

Chi, Q., et al. (2014). A reconfigurable smart sensor interface for industrial WSN in IoT environment. *IEEE transactions on industrial informatics, 10*(2): 1417-1425.

Durišić, M. P. et al. (2012). A survey of military applications of wireless sensor networks. *In Embedded Computing (MECO)*, 196-199.

Fernandes, M. A., et al. (2013). A framework for wireless sensor networks management for precision viticulture and agriculture based on IEEE 1451 standard. *Computers and Electronics in Agriculture, 95*: 19-30.

Iqbal, Z., Kim, K., and Lee, H. N. (2017). A Cooperative Wireless Sensor Network for Indoor Industrial Monitoring. *IEEE Trans. Industrial Informatics, 13*(2): 482-491.

Ganesh, E. N. (2017). IoT Based Environment Monitoring using Wireless Sensor Network. *International Journal of Advanced Research 5*(2): 964-970.

Garvey, A., & Lesser, V. (1994). A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems, 6*(3): 317-347.

Gay, D., et al. (2014). The nesC language: A holistic approach to networked embedded systems. *ACM Sigplan Notices, 49*(4): 41-51.

Giri, P., Ng, K., and Phillips, W. (2018). Wireless Sensor Network System for Landslide Monitoring and Warning. *IEEE Transactions on Instrumentation and Measurement, 99:* 1-11.

Gray, M. A., & Scherer, P. N. (2014). Web services framework for wireless sensor networks. *In Service Computation, The Sixth International Conferences on Advanced Service Computing, IARIA*, 15-23.

Gubbi, J., et al. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems, 29*(7): 1645-1660.

Horvitz, E. J., Breese, J. S., and Henrion, M. (1988). Decision theory in expert systems and artificial intelligence. *International journal of approximate reasoning, 2*(3): 247-302.

Karaboga, D., et al. (2014). A comprehensive survey: artificial bee colony (ABC) algorithm and applications. *Artificial Intelligence Review, 42*(1): 21-57.

Khairunniza-Bejo, S., Ramli, N., and Muharam, F. M. (2018). Wireless Sensor Network (WSN) Applications in Plantation Canopy Areas: A Review. *Asian Journal of Scientific Research, 11*(2): 151-161.

Kiani, F. (2014). *Designing New Routing Algorithms Optimized for Wireless Sensor Network*, LAP LAMBERT Academic Publishing, Dusseldorf, Germany.

Kiani, F., Seyyedabbasi, A. (2018). Wireless Sensor Network and Internet of Things in Precision Agriculture. *International Journal of Advanced Computer Science and Applications (IJACSA), 9*(6): 99-103.

Kim, T. H., & Hong, S. (2004). State machine based operating system architecture for wireless sensor networks. *In Parallel and Distributed Computing: Applications and Technologies*, 803-806.

Krämer, M., Bader, S., and Oelmann, B. (2013). Implementing Wireless Sensor Network applications using hierarchical finite state machines. *10th IEEE International Conference on in Networking, Sensing and Control (ICNSC)*, 124-129.

Kulkarni, R. V., Forster, A., and Venayagamoorthy, G. K. (2011). Computational intelligence in wireless sensor networks: A survey. *IEEE communications surveys & tutorials, 13*(1): 68-96.

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, *13*(1): 32-35.

Lewis, P. R., et al. (2015). Architectural aspects of self-aware and self-expressive computing systems: From psychology to engineering. *Computer*, *48*(8): 62-70.

Liu, C., Wu, K., & Pei, J. (2007). An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. *IEEE transactions on parallel and distributed systems, 18*(7): 1010-1023.

Lo, B. P., et al. (2005). Body sensor network–a wireless sensor platform for pervasive healthcare monitoring, 77-80.

Lu, K., et al. (2008). A framework for a distributed key management scheme in heterogeneous wireless sensor networks. *IEEE transactions on wireless communications, 7*(2): 639-647.

Miao, Y., et al. (2012, October). Research on power consumption weighted state machine of farmland WSN node. *International Conference on in Cloud Computing and Intelligent Systems (CCIS), 3*: 1141-1144.

Murthy, S. K. (1998). Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery, 2*(4): 345-389.

Othman, M. F., & Shazali, K. (2012). Wireless sensor network applications: A study in environment monitoring system. *Procedia Engineering, 41*: 1204-1210.

Perez, I., Bärenz, M., and Nilsson, H. (2018). Functional reactive programming, refactored. *ACM SIGPLAN Notices, 51*(12): 33-44.

Pule, M., Yahya, A., and Chuma, J. (2017). Wireless sensor networks: A survey on monitoring water quality. *Journal of Applied Research and Technology, 15*(6): 562-570.

Rabinowitz, N. C., et al. (2018). Machine Theory of Mind. arXiv preprint arXiv:1802.07740.

Ramesh, M. V. (2009). Real-time wireless sensor network for landslide detection. In Sensor Technologies and Applications, 2009. *Third International Conference on IEEE SENSORCOMM'09*, 405-409.

Rault, T., Bouabdallah, A., and Challal, Y. (2014). *Energy efficiency in wireless sensor networks: A top-down survey. Computer Networks, 67*: 104-122.

Sundani, H., et al. (2011). Wireless sensor network simulators a survey and comparisons. *International Journal of Computer Networks, 2*(5): 249-265.

Sudevalayam, S., & Kulkarni, P. (2011). *Energy harvesting sensor nodes: Survey and implications. IEEE Communications Surveys & Tutorials, 13*(3): 443-461.

Veres, S. M., et al. (2011). Autonomous vehicle control systems—a review of decision making. *Proceedings of the Institution of Mechanical Engineers, Journal of Systems and Control Engineering, 225*(2), 155-195.

Yick, J., Mukherjee, B., and Ghosal, D. (2008). Wireless sensor network survey. *Computer networks, 52*(12): 2292-2330.

Zhou, Q., et al. (2015). Graphene electrostatic microphone and ultrasonic radio. *Proceedings of the National Academy of Sciences, 112*(29): 8942-8946.

APPENDIX A – A simple scenario for sequential application

```
//Smorithm
#include "LogicUnit.cpp"

LogicUnit logicUnit;

void setup()
{
        logicUnit = LogicUnit();
        logicUnit.Setup();
}

void loop()
{
        logicUnit.UpdateLogicUnit();
}

//Code generated by WSN Manager - izu
```

```
//Smorithm
#include "Arduino.h"

//state machine handler includes
#include "StateMachineHandler1.h"
#include "StateMachineHandler2.h"


//custom includes


class LogicUnit
{
private:
        //State mchine handlers definition
        StateMachineHandler1 sm1;
        StateMachineHandler2 sm2;


public:
        void Setup()
        {
                //Pins
                pinMode(A1,OUTPUT);//Blue LED Pin
        pinMode(A3,OUTPUT);//Red LED Pin



                //State mchine handlers allocation
                sm1 = StateMachineHandler1();
                sm2 = StateMachineHandler2();


                //State machine Setup
                sm1.stMachine.Setup();
                sm2.stMachine.Setup();


                //State machine handlers Setup
```

```
        sm1.Setup();
        sm2.Setup();


        }
        void UpdateLogicUnit()
        {
                //State mchine handlers Update
                sm1.stMachine.CheckCurrentState();
                sm2.stMachine.CheckCurrentState();


        }
};
```

```
//Smorithm
#include "Pin.h"
#include "Arduino.h"

Pin::Pin(){}
Pin::~Pin(){}

static void Pin::Setup()
{

}

static bool Pin::GetDigital(int id){return digitalRead(id);}
static int Pin::GetAnalog(int id){return analogRead(id);}
```

```
//Smorithm
#ifndef PIN_H
#define PIN_H

class Pin
{
public:
        Pin();
        ~Pin();
        static void Setup();
        static bool GetDigital(int);
        static int GetAnalog(int);
};

#endif
```

```
//Smorithm
#include "Pool.h"

Pool::Pool(){}
Pool::~Pool(){}
```

```
//Smorithm
#ifndef POOL_H
#define POOL_H
```

```
//Data frame structures


class Pool
{
public:

        Pool();
        ~Pool();

};

#endif
```

```
//Smorithm
#include "State.h"

State::State(){}
State::~State(){}

void State::StateEnter(){}
void State::StateUpdate(){}
void State::StateExit(){}

void State::SetTransitionsCount(int val){TransitionsCount = val;}
int State::GetTransitionsCount(){return TransitionsCount;}
void State::SetTransitions(Transition** transitions){Transitions =
transitions;}
Transition** State::GetTransitions(){return Transitions;}

int State::NextState(){for (int i = 0; i < TransitionsCount; i++){if
(Transitions[i]->IsTransitionFired(time)){return Transitions[i]-
>GetTargetState();}}return -1;}
```

```
//Smorithm
#ifndef STATE_H
#define STATE_H

#include "Transition.h"

class State
{
private:
  int TransitionsCount;
  Transition** Transitions;
public:
  unsigned long time;

  State();
  ~State();
  int GetTransitionsCount();
  void SetTransitionsCount(int);
  void SetTransitions(Transition**);
  Transition** GetTransitions();
  virtual void StateEnter();
  virtual void StateUpdate();
  virtual void StateExit();
  virtual int NextState();
};
#endif
```

```
//Smorithm
#include "StateMachine.h"

StateMachine::StateMachine(){}
StateMachine::~StateMachine(){}

void StateMachine::Setup(){}
State StateMachine::GetState(int index){return *States[index];}
void StateMachine::SetStates(State** states){States = states;}
void StateMachine::SetStatesCount(int count){StatesCount = count;}
void StateMachine::CheckCurrentState()
{
  int ns = States[CurrentState]->NextState();
  if (ns > -1)
  {
    States[CurrentState]->StateExit();
    CurrentState = ns;
    States[CurrentState]->StateEnter();
  }
  else{States[CurrentState]->StateUpdate();}
}

void StateMachine::InitStateEnter()
{
    States[CurrentState]->StateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINE_H
#define STATEMACHINE_H

#include "State.h"
#include "Transition.h"
#include "Arduino.h"

class StateMachine
{
private:
        int StatesCount;
        State** States;
public:
        int CurrentState;
        int StartState;
        int AnyState;
        StateMachine();
        ~StateMachine();
        void Setup();
        void CheckCurrentState();
        void SetStatesCount(int); int GetStatesCount();
        void SetStates(State**);
        void InitStateEnter();
        State** GetStates(); State GetState(int);
};
#endif
```

```
//Smorithm
#include "StateMachineHandler1.h"

//Custom code
```

```
//States decleration
//Smorithm
/*---- on blue -------*/
class st_2_1 : public State
{
public:
        st_2_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A1,HIGH);
delay(1000);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_2_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};
//Smorithm
/*---- off red -------*/
class st_3_1 : public State
{
public:
        st_3_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A1,LOW);
delay(1000);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};
```

```
//Smorithm

class tr_7_3_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};



StateMachineHandler1::StateMachineHandler1()
{
        //States count
        State** st = new State*[2];

        //States initialization

                st[0] = new st_2_1();
                st[0]->SetTransitionsCount(1);
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_6_2_1();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_3_1();
                st[1]->SetTransitionsCount(1);
                Transition** trans_3 = new Transition*[1];

                trans_3[0]=new tr_7_3_1();
                trans_3[0]->SetTargetState(0);
                st[1]->SetTransitions(trans_3);


        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
}

void StateMachineHandler1::Setup()
{

        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER1_H
#define STATEMACHINEHANDLER1_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_1;
class st_3_1;
```

```
//Transitions
class tr_6_2_1;
class tr_7_3_1;


class StateMachineHandler1
{
public:
        StateMachineHandler1();
        StateMachine stMachine;
        void Setup();
};

#endif
```

```
//Smorithm
#include "StateMachineHandler2.h"

//Custom code


//States decleration
//Smorithm
/*---- on red -------*/
class st_2_2 : public State
{
public:
        st_2_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A3,HIGH);
delay(750);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_5_2_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};
//Smorithm
/*---- off red -------*/
class st_4_2 : public State
{
```

```
public:
        st_4_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A3,LOW);
delay(750);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_4_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};



StateMachineHandler2::StateMachineHandler2()
{
        //States count
        State** st = new State*[2];

        //States initialization

                st[0] = new st_2_2();
                st[0]->SetTransitionsCount(1);
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_5_2_2();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_4_2();
                st[1]->SetTransitionsCount(1);
                Transition** trans_4 = new Transition*[1];

                trans_4[0]=new tr_6_4_2();
                trans_4[0]->SetTargetState(0);
                st[1]->SetTransitions(trans_4);


        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
}
```

```
void StateMachineHandler2::Setup()
{

        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER2_H
#define STATEMACHINEHANDLER2_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_2;
class st_4_2;


//Transitions
class tr_5_2_2;
class tr_6_4_2;


class StateMachineHandler2
{
public:
        StateMachineHandler2();
        StateMachine stMachine;
        void Setup();
};

#endif
```

```
//Smorithm
#include "StateMachine.h"

Transition::Transition(){}
Transition::~Transition(){}

bool Transition::IsTransitionFired(unsigned long time){return
false;}
void Transition::SetTargetState(int val){TargetState = val;}
int Transition::GetTargetState(){return TargetState;}
```

```
//Smorithm
#ifndef TRANSITION_H
#define TRANSITION_H

class Transition
{
private:
```

```
  int TargetState;
public:
  Transition();
  ~Transition();
  void SetTargetState(int);
  int GetTargetState();
  virtual bool IsTransitionFired(unsigned long);
};

#endif
```

APPENDIX B - A simple scenario for a concurrent application

```
 //Smorithm
#include "StateMachineHandler1.h"

//Custom code


//States decleration
//Smorithm
/*---- on blue -------*/
class st_2_1 : public State
{
public:
        st_2_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A1,HIGH);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_2_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 1000))
{return true;}return false; }};
//Smorithm
/*---- off blue -------*/
class st_3_1 : public State
{
public:
        st_3_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A1,LOW);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
```

```
        {

        }
};

//Smorithm

class tr_7_3_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 1000))
{return true;}return false; }};



StateMachineHandler1::StateMachineHandler1()
{
        //States count
        State** st = new State*[2];

        //States initialization

                st[0] = new st_2_1();
                st[0]->SetTransitionsCount(1);
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_6_2_1();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_3_1();
                st[1]->SetTransitionsCount(1);
                Transition** trans_3 = new Transition*[1];

                trans_3[0]=new tr_7_3_1();
                trans_3[0]->SetTargetState(0);
                st[1]->SetTransitions(trans_3);


        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
}

void StateMachineHandler1::Setup()
{

        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER1_H
#define STATEMACHINEHANDLER1_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
```

```
class st_2_1;
class st_3_1;


//Transitions
class tr_6_2_1;
class tr_7_3_1;


class StateMachineHandler1
{
public:
        StateMachineHandler1();
        StateMachine stMachine;
        void Setup();
};

#endif
```

```
//Smorithm
#include "StateMachineHandler2.h"

//Custom code


//States decleration
//Smorithm
/*---- on red -------*/
class st_2_2 : public State
{
public:
        st_2_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A3,HIGH);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_5_2_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 750))
{return true;}return false; }};
//Smorithm
/*---- off red -------*/
```

```
class st_4_2 : public State
{
public:
        st_4_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                digitalWrite(A3,LOW);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_4_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 750))
{return true;}return false; }};



StateMachineHandler2::StateMachineHandler2()
{
        //States count
        State** st = new State*[2];

        //States initialization

                st[0] = new st_2_2();
                st[0]->SetTransitionsCount(1);
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_5_2_2();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_4_2();
                st[1]->SetTransitionsCount(1);
                Transition** trans_4 = new Transition*[1];

                trans_4[0]=new tr_6_4_2();
                trans_4[0]->SetTargetState(0);
                st[1]->SetTransitions(trans_4);


        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
```

```
}

void StateMachineHandler2::Setup()
{

        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER2_H
#define STATEMACHINEHANDLER2_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_2;
class st_4_2;


//Transitions
class tr_5_2_2;
class tr_6_4_2;


class StateMachineHandler2
{
public:
        StateMachineHandler2();
        StateMachine stMachine;
        void Setup();
};

#endif
```

APPENDIX C - A simple sensor node with the broadcast method

```
//Smorithm
#include "Pool.h"

//temperature Packet. |
        static df_0 Pool:: v_1;

//humidityPacket. |
        static df_1 Pool:: v_2;

Pool::Pool(){}
Pool::~Pool(){}

static CC1101 Pool::radio;
static bool Pool::packetWaiting;

static void Pool::InitRadio(byte freq, uint8_t paLevel, byte*
syncWord)
{
  radio.init();
  radio.setSyncWord(syncWord);
  radio.setCarrierFreq(freq);
  radio.disableAddressCheck();
  radio.setTxPowerAmp(paLevel);

  attachInterrupt(0, messageReceived, FALLING);
}

static bool Pool::SendRadio(char* data, int length,char mode)
{
  CCPACKET packet;
  packet.length = length + 2;
  memcpy(packet.data+1, data, length);
  packet.data[0]=mode;
  packet.data[length]='\0';
  return radio.sendData(packet);
}
static void Pool::messageReceived() {packetWaiting = true;}

static CCPACKET Pool::ReceiveRadio()
{
  CCPACKET packet;
  packet.length = 0;
  if (packetWaiting) {
    detachInterrupt(0);
    packetWaiting = false;
    radio.receiveData(&packet);
    if (!packet.crc_ok)packet.length = 0;
  }
  attachInterrupt(0, messageReceived, FALLING);
  return packet;
}
```

```
//Smorithm
#ifndef POOL_H
#define POOL_H
#include <cc1101.h>
#include <ccpacket.h>
```

```
//Data frame structures

//Temperature frame :
struct df_0
{
        //Temperature , Degree Celsius
        float dfi_0;

        //Time , Seconds
        int dfi_1;

        //Device ID ,
        short int dfi_2;

        //Trigger count ,
        int dfi_3;

};

//Humidity frame :
struct df_1
{
        //Humidity ,
        float dfi_0;

        //Time , Seconds from device start
        int dfi_1;

        //Device ID ,
        short int dfi_2;
};

class Pool
{
  private:
    static bool packetWaiting;
    static CC1101 radio;
    static void messageReceived();

public:

        Pool();
        ~Pool();


//temperature Packet. |
        static df_0  v_1;

//humidityPacket. |
        static df_1  v_2;

        static void InitRadio(byte , uint8_t , byte*);
    static bool SendRadio(char*,int length,char mode);
    static bool IsPacketWaiting();
    static CCPACKET ReceiveRadio();
};
#endif
```

```
//Smorithm
#include "StateMachineHandler1.h"

//Custom code
#include "ClosedCube_HDC1080.h"

ClosedCube_HDC1080 hdcT;
byte syncWord[2] = {199, 10};



//States decleration
//Smorithm
/*---- Sense -------*/
class st_2_1 : public State
{
public:
        st_2_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Serial.println("Temperature sensing started... ");
                time=millis();
        }

        void StateUpdate()
        {
                Pool::v_1.dfi_0=hdcT.readTemperature();
        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_2_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((Pool::v_1.dfi_0 > 32))
{return true;}return false; }};
//Smorithm
/*---- high Temp -------*/
class st_3_1 : public State
{
public:
        st_3_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Pool::v_1.dfi_3++;
Pool::v_1.dfi_1=millis()/1000;
Serial.print(" A high temperature was detected: ");
Serial.print(Pool::v_1.dfi_0);
                time=millis();
        }
```

```
        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_7_3_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};
//Smorithm
/*---- Send -------*/
class st_4_1 : public State
{
public:
        st_4_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Pool::SendRadio((char*)&Pool::v_1,
sizeof(Pool::v_1),1);
Serial.print(" degree Celsius and was sent at: ");
Serial.println(Pool::v_1.dfi_1);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_10_4_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};
//Smorithm
/*---- Deadtime -------*/
class st_9_1 : public State
{
public:
        st_9_1()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
```

```
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {
                Pool::v_1.dfi_0=hdcT.readTemperature();
        }
};

//Smorithm

class tr_11_9_1 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 10000))
{return true;}return false; }};


StateMachineHandler1::StateMachineHandler1()
{
        //States count
        State** st = new State*[4];

        //States initialization

                st[0] = new st_2_1();
                st[0]->SetTransitionsCount(1);
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_6_2_1();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_3_1();
                st[1]->SetTransitionsCount(1);
                Transition** trans_3 = new Transition*[1];

                trans_3[0]=new tr_7_3_1();
                trans_3[0]->SetTargetState(2);
                st[1]->SetTransitions(trans_3);

                st[2] = new st_4_1();
                st[2]->SetTransitionsCount(1);
                Transition** trans_4 = new Transition*[1];

                trans_4[0]=new tr_10_4_1();
                trans_4[0]->SetTargetState(3);
                st[2]->SetTransitions(trans_4);

                st[3] = new st_9_1();
                st[3]->SetTransitionsCount(1);
                Transition** trans_9 = new Transition*[1];

                trans_9[0]=new tr_11_9_1();
                trans_9[0]->SetTargetState(0);
                st[3]->SetTransitions(trans_9);
```

```
        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
}

void StateMachineHandler1::Setup()
{
          hdcT.begin(0x40);
  hdcT.setResolution(HDC1080_RESOLUTION_11BIT,
HDC1080_RESOLUTION_11BIT);
Serial.begin(9600);
Pool::InitRadio(CFREQ_868, PA_LongDistance, syncWord);
Pool::v_1.dfi_2=4;
Pool::v_1.dfi_3=0;


        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER1_H
#define STATEMACHINEHANDLER1_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_1;
class st_3_1;
class st_4_1;
class st_9_1;


//Transitions
class tr_6_2_1;
class tr_7_3_1;
class tr_10_4_1;
class tr_11_9_1;


class StateMachineHandler1
{
public:
        StateMachineHandler1();
        StateMachine stMachine;
        void Setup();
};

#endif
```

```
//Smorithm
#include "StateMachineHandler2.h"

//Custom code
#include "ClosedCube_HDC1080.h"

ClosedCube_HDC1080 hdcH;



//States decleration
//Smorithm
/*---- idle -------*/
class st_2_2 : public State
{
public:
        st_2_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Serial.println("Humidity sensing started... ");
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_6_2_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if((millis()-time > 15000))
{return true;}return false; }};
//Smorithm
/*---- Sense humidity -------*/
class st_4_2 : public State
{
public:
        st_4_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Pool::v_2.dfi_0=hdcH.readHumidity();
Serial.print("Sensed humidity is: ");
Serial.print(Pool::v_2.dfi_0);
Pool::v_2.dfi_1=millis()/1000;
                time=millis();
        }

        void StateUpdate()
```

```
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_7_4_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};
//Smorithm
/*---- Send -------*/
class st_5_2 : public State
{
public:
        st_5_2()
        {
                SetTransitionsCount(1);
        }
        void StateEnter()
        {
                Pool::SendRadio((char*)&Pool::v_2,
sizeof(Pool::v_2),2);
Serial.print("%. The humidity was sent at: ");
Serial.println(Pool::v_2.dfi_1);
                time=millis();
        }

        void StateUpdate()
        {

        }

        void StateExit()
        {

        }
};

//Smorithm

class tr_8_5_2 : public Transition{public:bool
IsTransitionFired(unsigned long time) { if(true) {return
true;}return false; }};



StateMachineHandler2::StateMachineHandler2()
{
        //States count
        State** st = new State*[3];

        //States initialization

                st[0] = new st_2_2();
                st[0]->SetTransitionsCount(1);
```

```
                Transition** trans_2 = new Transition*[1];

                trans_2[0]=new tr_6_2_2();
                trans_2[0]->SetTargetState(1);
                st[0]->SetTransitions(trans_2);

                st[1] = new st_4_2();
                st[1]->SetTransitionsCount(1);
                Transition** trans_4 = new Transition*[1];

                trans_4[0]=new tr_7_4_2();
                trans_4[0]->SetTargetState(2);
                st[1]->SetTransitions(trans_4);

                st[2] = new st_5_2();
                st[2]->SetTransitionsCount(1);
                Transition** trans_5 = new Transition*[1];

                trans_5[0]=new tr_8_5_2();
                trans_5[0]->SetTargetState(0);
                st[2]->SetTransitions(trans_5);


        //Start state
        stMachine.StartState = 0;
        stMachine.CurrentState = 0;

        stMachine.SetStates(st);
}

void StateMachineHandler2::Setup()
{
        Pool::v_2.dfi_2=4;
  hdcH.begin(0x40);
  hdcH.setResolution(HDC1080_RESOLUTION_11BIT,
HDC1080_RESOLUTION_11BIT);
        stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER2_H
#define STATEMACHINEHANDLER2_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_2;
class st_4_2;
class st_5_2;


//Transitions
class tr_6_2_2;
class tr_7_4_2;
class tr_8_5_2;
```

```
class StateMachineHandler2
{
public:
        StateMachineHandler2();
        StateMachine stMachine;
        void Setup();
};

#endif
```

APPENDIX D - A simple monitoring base station scenario

```
//Smorithm
#include "Pool.h"

//tmp. |
        static df_0 Pool:: v_1;

//hmd. |
        static df_1 Pool:: v_2;

//frame mode. |
        static short int Pool:: v_3;

Pool::Pool(){}
Pool::~Pool(){}

static CC1101 Pool::radio;
static bool Pool::packetWaiting;

static void Pool::InitRadio(byte freq, uint8_t paLevel, byte*
syncWord)
{
  radio.init();
  radio.setSyncWord(syncWord);
  radio.setCarrierFreq(freq);
  radio.disableAddressCheck();
  radio.setTxPowerAmp(paLevel);

  attachInterrupt(0, messageReceived, FALLING);
}

static bool Pool::SendRadio(char* data, int length,char mode)
{
  CCPACKET packet;
  packet.length = length + 2;
  memcpy(packet.data+1, data, length);
  packet.data[0]=mode;
  packet.data[length]='\0';
  return radio.sendData(packet);
}
static void Pool::messageReceived() {packetWaiting = true;}

static CCPACKET Pool::ReceiveRadio()
{
  CCPACKET packet;
  packet.length = 0;
  packet.data[0]=0;
  if (packetWaiting) {
    detachInterrupt(0);
    packetWaiting = false;
    radio.receiveData(&packet);
    if (!packet.crc_ok)packet.length = 0;
  }
  attachInterrupt(0, messageReceived, FALLING);
  return packet;
}
```

```
//Smorithm
#ifndef POOL_H
#define POOL_H
#include <cc1101.h>
#include <ccpacket.h>

//Data frame structures

//Temperature frame template :
struct df_0
{
        //Temperature ,
        float dfi_0;

        //Time , Seconds
        int dfi_1;

        //Device ID ,
        short int dfi_2;

        //Event count ,
        int dfi_3;

};

//Humidity frame template :
struct df_1
{
        //Humidity , %
        float dfi_0;

        //Time ,
        int dfi_1;

        //Device ID ,
        short int dfi_2;

};


class Pool
{

  private:
    static bool packetWaiting;
    static CC1101 radio;
    static void messageReceived();

public:

        Pool();
        ~Pool();


//tmp. |
        static df_0  v_1;

//hmd. |
        static df_1  v_2;

//frame mode. |
```

```
        static short int  v_3;


        static void InitRadio(byte , uint8_t , byte*);
    static bool SendRadio(char*,int length,char mode);
    static bool IsPacketWaiting();
    static CCPACKET ReceiveRadio();
};

#endif
```

```
//Smorithm
#include "StateMachineHandler1.h"

//Custom code
byte syncWord[2] = {199, 10};
CCPACKET packet;

//States decleration
//Smorithm
/*---- Listen -------*/
class st_2_1 : public State
{
  public:
    st_2_1()
    {
      SetTransitionsCount(2);
    }
    void StateEnter()
    {

      time = millis();
    }

    void StateUpdate()
    {
      packet = Pool::ReceiveRadio();
      Pool::v_3 = packet.data[0];

    }

    void StateExit()
    {

    }
};

//Smorithm

class tr_6_2_1 : public Transition {
  public: bool IsTransitionFired(unsigned long time) {
      if ((Pool::v_3 == 2)) {
        return true;
      } return false;
    }
};
//Smorithm

class tr_10_2_1 : public Transition {
```

```
  public: bool IsTransitionFired(unsigned long time) {
      if ((Pool::v_3 == 1)) {
        return true;
      } return false;
    }
};
//Smorithm
/*---- Temperature -------*/
class st_4_1 : public State
{
  public:
    st_4_1()
    {
      SetTransitionsCount(1);
    }
    void StateEnter()
    {
      memcpy(&Pool::v_1, packet.data + 1, sizeof(Pool::v_1) + 1);
      Serial.print("High temperature is reported ");
      Serial.print(Pool::v_1.dfi_0);
      Serial.print(" degree Celsius from device ");
      Serial.print(Pool::v_1.dfi_2);
      Serial.print(" at time ");
      Serial.print(Pool::v_1.dfi_1);
      Serial.print(" ,event number: ");
      Serial.println(Pool::v_1.dfi_3);
      time = millis();
    }

    void StateUpdate()
    {

    }

    void StateExit()
    {
      Pool::v_3 = 0;
    }
};

//Smorithm

class tr_9_4_1 : public Transition {
  public: bool IsTransitionFired(unsigned long time) {
      if (true) {
        return true;
      } return false;
    }
};
//Smorithm
/*---- Humidity -------*/
class st_5_1 : public State
{
  public:
    st_5_1()
    {
      SetTransitionsCount(1);
    }
    void StateEnter()
    {
      memcpy(&Pool::v_2, packet.data + 1, sizeof(Pool::v_2) + 1);
```

```
      Serial.print("Humidity ");
      Serial.print(Pool::v_2.dfi_0);
      Serial.print("% from device ");
      Serial.print(Pool::v_2.dfi_2);
      Serial.print(" at time: ");
      Serial.println(Pool::v_2.dfi_1);
      time = millis();
    }

    void StateUpdate()
    {

    }

    void StateExit()
    {
      Pool::v_3 = 0;
    }
};

//Smorithm

class tr_7_5_1 : public Transition {
  public: bool IsTransitionFired(unsigned long time) {
      if (true) {
        return true;
      } return false;
    }
};



StateMachineHandler1::StateMachineHandler1()
{
  //States count
  State** st = new State*[3];

  //States initialization

  st[0] = new st_2_1();
  st[0]->SetTransitionsCount(2);
  Transition** trans_2 = new Transition*[2];

  trans_2[0] = new tr_6_2_1();
  trans_2[0]->SetTargetState(2);
  st[0]->SetTransitions(trans_2);

  trans_2[1] = new tr_10_2_1();
  trans_2[1]->SetTargetState(1);
  st[0]->SetTransitions(trans_2);

  st[1] = new st_4_1();
  st[1]->SetTransitionsCount(1);
  Transition** trans_4 = new Transition*[1];

  trans_4[0] = new tr_9_4_1();
  trans_4[0]->SetTargetState(0);
  st[1]->SetTransitions(trans_4);

  st[2] = new st_5_1();
  st[2]->SetTransitionsCount(1);
```

```
  Transition** trans_5 = new Transition*[1];

  trans_5[0] = new tr_7_5_1();
  trans_5[0]->SetTargetState(0);
  st[2]->SetTransitions(trans_5);


  //Start state
  stMachine.StartState = 0;
  stMachine.CurrentState = 0;

  stMachine.SetStates(st);
}

void StateMachineHandler1::Setup()
{
  Serial.begin(9600);
  Pool::InitRadio(CFREQ_868, PA_LongDistance, syncWord);
  stMachine.InitStateEnter();
}
```

```
//Smorithm
#ifndef STATEMACHINEHANDLER1_H
#define STATEMACHINEHANDLER1_H

#include "StateMachine.h"
#include "Pin.h"
#include "Pool.h"

//States
class st_2_1;
class st_4_1;
class st_5_1;


//Transitions
class tr_6_2_1;
class tr_10_2_1;
class tr_9_4_1;
class tr_7_5_1;


class StateMachineHandler1
{
public:
        StateMachineHandler1();
        StateMachine stMachine;
        void Setup();
};

#endif
```

BIOGRAPHY

Sajjad NEMATZADEHMIANDOAB is a Master student in computer science and engineering at Istanbul Sabahattin Zaim University (IZU). He is researching on Ad-hoc networks and machine learning and he is head of the graduate project team in WSN and ML laboratories. He was a member of "Smart Agriculture in IZU Campus" BAP project that their team could be finished it successfully. His current research interests include wireless sensor networks, machine learning, game engines and software development systems. He has one accepted paper under publish in SCI ındex journal based on his thesis topic.