Energy Consumption of Wireless IoT Nodes

Amen Hussain

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2017
Supervisor: Frank Alexander Krämer, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology
Energy Consumption of Wireless IoT Nodes

Amen Hussain

Submission date: June 2017
Responsible professor: Frank Alexander Kraemer, ITEM
Supervisor: Nattachart Tamkittikhun, ITEM

Norwegian University of Science and Technology
Department of Telematics
Abstract

The Internet of Things (IoT) is an emerging technology, encompassing a wide spectrum of applications related to industrial control, smart metering, home automation, agriculture, eHealth and so on. For these applications to run autonomously, the IoT devices are required to survive for months and years under strict energy constraints. When developing such applications, it is important for the application to know about its own energy consumption.

In this work, we propose and evaluate an energy consumption estimation approach for periodic sensing applications running on the IoT devices. Our approach is based on three phases. In the first phase, we identify the distinct activities such as sleep, transmit, sense and process in a sensing cycle. In the second phase, we measure the power consumption of these activities before the IoT device has been deployed in the network. The third phase takes place at run-time once the IoT device has been deployed, with the purpose of delivering the energy consumption of a sensing cycle. The energy consumption is calculated by using the activities’ power consumption and their durations obtained at run-time.

The proposed approach is simple and generic because it doesn’t involve any complex hardware for runtime power measurement. Moreover, this approach also incorporates the dynamic nature of sensing applications by run-time estimation of energy consumption. Our results show that the error of energy estimation for the chosen applications is between 0.04% and 2.928%.
Preface

This dissertation is part of the Telematics, Master’s Thesis taken in the 4th semester (spring 2017) of my 2-year MSc in Telematics-Communication Networks and Networked Services at the Norwegian University of Science and Technology (NTNU), awarding 30 ECTS credits.

This work is focused on the development of a precise and efficient energy consumption estimation model designed specifically for periodic sensing applications running on IoT sensing devices. It involves experimental work to understand the energy consumption by the sensing nodes. The development of energy consumption estimation model has given me an insight into various topics related to the power measurement process, LoRaWAN communication protocol and hardware and software platforms used by different IoT devices. In this work, I also developed a skill to design relevant experiments and analyze the gathered data efficiently. I would like to thank my supervisor, Nattachart Tamkittikhun and my responsible professor, Frank Alexander Kraemer for providing their insightful guidelines throughout this work.

I would also like to thank a good friend Knut Magnus for encouraging me to write the Python script for automatic power consumption measurement from the oscilloscope. This helped me in streamlining the power measurement process.

Amen Hussain
Contents

List of Figures xi
List of Tables xiii
List of Algorithms xvi

1 Introduction 1
  1.1 Problem Description 3
  1.2 Methodology 4
  1.3 Motivation 5
  1.4 Publication 5
  1.5 Report Structure 5

2 Background 7
  2.1 Energy Monitoring using Shunt Resistor 8
  2.2 Software based Energy Estimation 9
  2.3 Specialized Hardware based Approach 10
  2.4 CPU Instruction-set based Energy Estimation 10
  2.5 Energy Modeling using Parametric Approach 11
  2.6 Black-box 11

3 Power Measurement Setup 13
  3.1 Measurement Setup 14
  3.2 Measurement Procedure 15

4 Power and Energy Consumption Analysis 17
  4.1 LED 19
  4.2 Energy Modes 23
  4.3 Peer-to-Peer Radio Transmission 33
  4.4 Peer-to-Peer Radio Receive 39
  4.5 Cryptography 41
  4.6 Sensors 45
5 Energy Modeling 53
5.1 Energy Consumption Estimation Model 53
5.2 Run-time Logic for Energy Estimation 55

6 Energy-Aware Application: Increasing Sensing Phase 57
6.1 Particle Sensor 58
6.1.1 Energy Aware Application 58
6.1.2 Oscilloscope Measurement 61
6.1.3 Conclusion 61
6.2 Temperature Sensor 62
6.2.1 Energy Aware Application 63
6.2.2 Oscilloscope Measurement 65
6.2.3 Conclusion 65

7 Energy-Aware Application: Different Sensing Modes 67
7.1 Particle Sensor 67
7.1.1 Energy-Aware Application 68
7.1.2 Conclusion 71
7.2 Temperature Sensor 72
7.2.1 Energy-Aware Application 73
7.2.2 Conclusion 74

8 Energy-Aware Application: Waspmote 75
8.1 Application Logic 75
8.2 Design-time and Run-time Energy Estimation Logic 78
8.3 Energy Aware Application 80

9 Discussion and Conclusion 83

References 85

Appendices
A Picoscope Python Script 89
B mDot P2P Communication Header Files 95
B.1 dot_util.h 95
B.2 mDotEvent.h 96
B.3 RadioEvent.h 102
C mDot Particle Sensing Energy-Aware Application 105
D mDot Temperature Sensing Energy-Aware Application 113
E waspMote CO2 Sensing Energy-Aware Application 119
E.1 Header File .................................................. 119
E.2 Source File .................................................. 119
List of Figures

1.1 Sensing System Overview [THK17] .............................................. 2
3.1 Power Measurement Setup ...................................................... 14
3.2 mDot Current Consumption measurement using Python Script ....... 16
4.1 mDot Current Consumption when Powering a single Light Emitting Diode (LED) .......................................................... 21
4.2 mDot Current Consumption when Powering two LEDs ............... 21
4.3 mDot going into sleep mode ..................................................... 28
4.4 mDot not going into deep-sleep mode ....................................... 28
4.5 mDot Current Consumption during Sleep and Active States ......... 29
4.6 Noise Floor Waveform View ................................................... 30
4.7 Noise Floor Spectral View ...................................................... 30
4.8 mDot current consumption during deep sleep mode ................. 31
4.9 Issue in mDot Automatic Current Consumption in Deep Sleep Mode . 32
4.10 mDot Transmit Cycle .......................................................... 35
4.11 Manual: mDot_senddata Energy Consumption ....................... 36
4.12 Automatic: mDot_senddata() Energy Consumption ................. 37
4.13 Automatic: mDot_senddata() Power Consumption ................. 38
4.14 Current Consumption when receiving LoRA frame of size 240B ....... 40
4.15 Current Consumption when receiving LoRA frame of size 120B ....... 40
4.16 Current Consumption during Cryptographic Operation ............. 43
4.17 mDot Cryptographic Energy Consumption .......................... 44
4.18 mDot Cryptographic Power Consumption ........................... 45
4.19 mDot Interfacing with Particle Sensor Schematic Diagram ........ 47
4.20 mDot Temperature Sensor Current Waveform ....................... 51
4.21 mDot Particle Sensor Current Waveform ........................... 52
5.1 Power consumption of one cycle of a sensing application. Some phases are shortened on the x-axis, to save space. The labels reveal the actual duration [THK17]. .............................................. 53
5.2 An abstract model of the energy consumption, with focus on the different activity phases. Power and time axis are not to scale[THK17].


7.1 Energy Profiles of Particle Sensing Energy-aware Application with Different Sensing Modes.

7.2 Energy Profiles of Temperature Sensing Energy-aware Application with Different Sensing Modes.

8.1 Current Consumption of a CO2 Sensing Application using Moderate Sensing Mode.

8.2 CO2 Sensing: Energy Consumption Comparison under Different Sensing Modes.

9.1 Percentage Error in Estimated Energy under Different Sensing Modes Running on Different Sensing Devices.
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>LED: Energy Consumption</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Sleep Mode Energy Consumption (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Energy Consumption: A Comparison (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Power Consumption in Pre-, Post- and Main Transmission Phases (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Energy Consumption: Transmission (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Rx Energy Consumption: A Comparison (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Cryptography: Energy and Power Consumption (E-05 in this table is equivalent to ( \times 10^{-5} ) and so on.)</td>
<td>44</td>
</tr>
<tr>
<td>4.8</td>
<td>Sensor Energy Consumption: A Comparison</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Energy Consumption Sent by the Energy-Aware Particle Sensing Application</td>
<td>61</td>
</tr>
<tr>
<td>6.2</td>
<td>Energy Consumption Measurement using Oscilloscope for the Energy-Aware Particle Sensing Application</td>
<td>62</td>
</tr>
<tr>
<td>6.3</td>
<td>Energy Consumption Modelling Particle Sensing Application: A comparison</td>
<td>62</td>
</tr>
<tr>
<td>6.4</td>
<td>Temperature Sensor Energy and Power Consumption with Transmission</td>
<td>64</td>
</tr>
<tr>
<td>6.5</td>
<td>Energy Consumption Sent by the Energy-Aware Temperature Sensing Application</td>
<td>65</td>
</tr>
<tr>
<td>6.6</td>
<td>Energy Consumption Measurement using Oscilloscope for the Energy-Aware Temperature Sensing Application</td>
<td>66</td>
</tr>
<tr>
<td>6.7</td>
<td>Energy Consumption Modelling Temperature Sensing Application: A comparison</td>
<td>66</td>
</tr>
<tr>
<td>7.1</td>
<td>Sensor Modes Parameters in Particle Sensing Energy-Aware Applications</td>
<td>68</td>
</tr>
<tr>
<td>7.2</td>
<td>Power and Energy Consumption during Sleep Mode for Particle Sensor</td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>Particle Sensing: Energy Consumption Comparison under Different Sensing Modes</td>
<td>71</td>
</tr>
</tbody>
</table>
7.4 Particle Sensing: Energy Consumption Estimation for 10 Minutes under Different Sensing Modes .................................................. 72
7.5 Sensor Modes Parameters in Temperature Sensing Energy-Aware Applications ................................................................. 72
7.6 Temperature Sensing: Energy Consumption Comparison under Different Sensing Modes ...................................................... 74
7.7 Temperature Sensing: Energy Consumption Estimation for 10 Minutes under Different Sensing Modes ........................................ 74
8.1 Power and Energy Consumption of CO2 Sensing Application Running on WASPMOTE ................................................................. 79
8.2 Sensor Modes Parameters in CO2 Sensing Energy-Aware Applications .............................................................. 80
8.3 CO2 Sensing: Energy Consumption Comparison under Different Sensing Modes .......................................................... 82
List of Algorithms

4.1 C++ Program: Toggling LED ........................................ 19
4.2 C++ Program: Toggling two LEDs .............................. 20
4.3 C++ Program: MTS logger ....................................... 24
4.4 Python Program: Read Serial .................................. 25
4.5 C++ Program: Generate Event for Automatic Energy Measurement 26
4.6 C++ Program: Active Mode Current Consumption ........... 26
4.7 C++ Program: Entering Sleep Mode ............................ 27
4.8 C++ Program: mDot LoRA peer-to-peer ....................... 33
4.9 C++ Program: mDot LoRA peer-to-peer Automatic Measurement . 34
4.10 C++ Program: mDot Encryption ................................ 41
4.11 C++ Program: mDot Decryption ............................... 42
4.12 C++ Program: Generate input ................................ 42
4.13 C++ Program: Temperature Sensor DS18B20 ............... 46
4.14 C++ Program: Particle Sensor Configuring General-Purpose Input Output (GPIO) ........................................ 47
4.15 C++ Program: Sensing Value from Particle Sensor ........ 48
4.16 C++ Program: Main Function of Particle Sensor ............ 49
4.17 C++ Program: Get Reference Voltage in Particle Sensor ... 50
5.1 General Application with Energy Estimation [THK17] ... 56
6.1 C++ Program: Energy Aware Application ..................... 59
6.2 C++ Program: Temperature Sensing Modification in Energy Aware Application .......................................... 63
7.1 C++ Program: Energy Aware Application with Static Code Blocks . 69
8.1 waspmote CO2 Sensing Phase .................................. 76
8.2 waspmote Transmission Phase .................................. 77
8.3 waspmote Sleeping Phase ....................................... 78
8.4 waspmote Generate Event ..................................... 79
8.5 waspmote RTC to Measure Duration of each Phase .......... 80
A.1 Python Script: Defining package imports ..................... 89
A.2 Defining Energy Measurement Class ........................... 90
A.3 Energy Measurement Class: Calculating mean and standard error . 91
A.4 Energy Measurement Class: Writing to an excel file ....... 92
Chapter 1

Introduction

The wireless sensor nodes containing the capability to sense, process and send data over a communication network are the backbone of the IoT. The IoT technology enabled the smart objects to communicate with each other and this resulted in communication anytime, any media, anywhere, and anything [AIM10]. It is being forecasted that by 2020 there will be around 50 billion devices connected to IoT networks [Cis]. These devices will be used in the design of new engineering solutions to societal scale problems such as transportation, power metering, healthcare, agriculture, home automation etc. These devices are wireless and possess stringent size constraints. Since there is no cable attached to power these devices therefore, they run on limited battery powered resources. It has become a daunting challenge to power these devices so that they can run for several months or possibly years of unattended operation. To address this issue, researchers focused on three different areas:

1. the design of ultra-low hardware platforms
2. the development of intelligent system-level power management techniques
3. the use of environmental energy harvesting to make IoT devices self-powered [JLL+14]

In this thesis, we will work on the development of an energy estimation model inside an application running on the sensing node. This work comes under the areas of the development of intelligent system-level power management techniques.

The available energy budget limits the amount of processing possible at the sensing node. The available energy restricts how a sensor will operate. If the sensor consumes relatively more power then the application attempts to acquire less sensing data in order to conserve energy. Similarly, it also restricts how much data will be transmitted via the sensing node. Lastly, the available energy resource also influences the quality of sensing measurements. One can acquire more accurate sensing values
2 1. INTRODUCTION

if one can sense more often and this will increase the energy consumption of the sensing node. If the sensing node is aware of its energy resources than it can shift to an appropriate sensing mode. It can sense more often or rarely based on the available energy resource. The energy awareness by the application allows the efficient use of the sensing node. More importantly, it will also prevent the node to become unresponsive due to lack of available energy.

On the other hand, the sensing nodes are subject to heterogeneous and non-stationary environments [DRAP15]. When solar energy is used to power up the sensing nodes the available energy highly depends on the weather conditions. That makes the available energy resource variable with respect to the current weather conditions. Therefore, a sensing node should be able to change its sensing mode dynamically.

Fig. 1.1, explains the general context of these sense application. It illustrates the general architecture of an IoT periodic sensing application. This architecture can be realized in a number of IoT applications such as a city-wide sensing system for emissions of particles or CO$_2$ [ADK$^+$16], a climate monitoring system, or a tracking system for animals.

![Sensing System Overview](image)

**Figure 1.1:** Sensing System Overview [THK17]

In such applications, the cloud is responsible for data acquisition and management of the sensing nodes. One can also perform energy planning in the cloud if all the sensing nodes are sending their energy consumption along with the sensed data. The cloud can instruct the sensing device to change to an appropriate sensing mode based on the observed available energy resource.
In this work, we developed an approach for energy estimation of periodic sensing applications running on IoT sensing nodes. These applications consist of different activity phases in a sensing cycle and this cycle repeats itself periodically. The activity phases in such applications can be sleeping, sensing, transmitting, and processing. Our proposed energy estimation model consists of three processes; 1) activity phase identification, 2) design-time and 3) run-time. The activity phase identification can be done based on each operation such that sensing, transmission, processing etc. These phases can be clearly identified by the application developer. However, an activity phase can be further divided into three blocks i.e. pre-, post-, and main based on their power consumption. The process of further dividing the activity phase involves rigorous power consumption analysis of these activity phases. In the design-time process, we measured the power consumption of different activity phases in a laboratory setting. Whereas, in the run-time process, the application computes the duration of its activity phases and computes the energy.

Since the power consumption of different activity phases depends on the attached peripherals and they don’t change dynamically, therefore, we have decided to compute the power consumption of such applications at design-time. Similarly, to incorporate the dynamic nature of such applications we have introduced the run-time process to measure the duration of these activity phases. When shifting to different sensing modes, one can only modify the duration of the activity phases, therefore, which has been incorporated by the run-time process. Thus, the presented approach takes care of the dynamic nature of sensing nodes along with the influence on the power consumption when different peripherals devices are activated simultaneously. The presented approach is generic and applicable for all different IoT hardware platforms, since it doesn’t require any specialized hardware. Our evaluation results show that this approach of estimating energy consumption is quite accurate.

1.1 Problem Description

In this work, we will attempt to develop a model to predict the energy consumption of applications running on an embedded hardware platform. We will establish a model for specific IoT applications that consist of sensing cycles. In a sensing cycle, an application goes into an active state, reads a sensor value, processes the sensed data, transmits it to another node or gateway depending on the topology, and then goes again into the sleep state.

We will conduct various experiments on different hardware platforms to measure the power consumption under various states and activities. These activities can be processing data, transmitting packets, reading sensor values, actuating a device.
Similarly, the states can be sleep, deep sleep, active, initializing peripherals etc. The results of these experiments will be used as building blocks to establish the model to predict the energy consumption of an application.

To generalize this energy consumption prediction model for various hardware platforms and communication protocols, we need to parameterize the model. Because the current consumption, the processing speed, and the transition duration from one state to the other will be different for different hardware platforms. In addition, different communication protocols will have a different number of packets sent and received, as well as packet lengths. To develop such a generic model, we will try to establish some coarse questions to parameterize this model. These questions will be related to the power consumed by hardware platforms under different states and activities with different sensors and actuators attached.

In the beginning, we will use MultiConnect mDot as an experimental hardware platform and LoRaWAN as the underlying communication protocol to establish the energy consumption model for the IoT applications.

1.2 Methodology

To solve the above-mentioned problem statement, the methodology of applied research was used. In the first step, we gathered theoretical knowledge and understanding of the existing techniques in the literature. Relevant materials were gathered by different search engines. Our main search engines were; IEEE Xplore, ACM Digital Library, and Google.

Based on the literature, we adopted an energy consumption measurement technique that was independent of the underlying IoT hardware platform. Initially, our motive was to understand the energy profiles of sensing applications running on the IoT hardware platform. We finalized a target IoT hardware platform and developed some experiments to deepen our understanding about the energy consumption by such applications.

Once, we established a sound knowledge about the IoT hardware platform and the applications’ energy profiles, then we proposed a model for energy consumption estimation by the sensing application. To evaluate the developed energy estimation model, we developed different types of sensing applications with our proposed energy estimation model. We performed a comparative analysis between the actual energy consumption by the application with the estimated one proposed by developed model.
1.3 Motivation

This thesis is part of an on-going inter-disciplinary research project at the Faculty of Information Technology and Electrical Engineering in NTNU. The research project is called Autonomous Resource-Constrained Things. This main goal of this project is to obtain the optimum data quality from a sensor node, under a constrained energy budget. The energy budget of a sensor node can be determined by measuring battery voltages, energy consumption per sensing cycle, and harvested energy from the solar panel. These parameters along with the weather forecast, to predict the energy produced the solar panel, can be used in a machine learning algorithm to determine a sensor node's energy utilization. This work is vital in the area of measuring per cycle energy consumption of an application.

Author’s initial motivation was to gain a profound knowledge in the area of Internet of Things. The author wanted to experiment with different IoT hardware platforms and understand IoT operating systems i.e. mbed and contiki. However, after performing the literature review and some trivial experiments, the author began to realize the impact of the energy consumption estimation model on the performance of a sensing node. To develop an energy estimation model that will provide an application’s energy consumption estimates on a per sensing cycle basis became the prime motivation.

1.4 Publication

Parts of this thesis has been published in the International Conference on Mobile, Secure and Programmable Networking (MSPN’2017) with the title; Energy Consumption Estimation for Energy-Aware, Adaptive Sensing Applications.

1.5 Report Structure

This thesis work consists of some experimental work, the design of our proposed model, and evaluation of the proposed prototype for energy estimation. The report has been organized as follows: Chapter 2 enumerates and summarizes some of the existing techniques to estimate energy consumption of an application running on a

---

1. https://www.ntnu.edu/iik/aas
hardware platform. Chapter 3 describes the method selected to compute the power consumption of the application running on the sensing node. Chapter 4 describes the experimental work to understand the power and energy consumption by different activity phases. Chapter 5 presents the energy estimation model. In Chapter 6 and 7 we developed two different types of sensing applications with diverse energy profiles and presented our energy consumption estimation results. Chapter 8 presents how our estimation model performs when we have used waspmote as the IoT hardware platform and CO2 sensor. Chapter 9 presents the conclusion and how well our work corresponds to the problem description.
Chapter 2

Background

The energy consumption of an embedded device can be acquired using various approaches. However, we have broadly categorized these methods into the offline and online approaches, and software and hardware approaches. In the online approach, the energy consumption is estimated at the run-time, whereas in the offline approach the energy consumption is estimated in a laboratory setting before deploying the node in the network. Similarly, in the hardware-based approach a specialized hardware is being used to measure the energy consumption, however, in the software-based approach, the application software is being modified to estimate its energy consumption. Both the software and the hardware approaches can be used in the online and offline approaches to predict the cost of energy.

In recent years, a lot of research has been done in the energy profiling of mobile computing platforms. The efficient use of energy by these computing devices is a key factor for both device manufacturers and the application developers. Since these handheld devices also run on batteries and the battery capacity of these devices is strictly restricted to the constraints on device’s weight and size. Thus, it is critical for these devices to optimally manage their power consumption [DMMJ16].

In the following sections we will discuss some of the existing techniques used to measure power and energy consumption of mobile device, wireless sensor nodes and other embedded devices. Since all of these devices run under a restricted battery environment.
2. BACKGROUND

2.1 Energy Monitoring using Shunt Resistor

Carroll and Heiser [CH10], performed rigorous tests and then simulated several usage scenarios to present the significance of the power drawn by various components: CPU, memory, touchscreen, graphics hardware, audio, storage, and networking interfaces in a mobile device. For energy profiling, they measured the physical power consumption of each component. They inserted a sense resistor on the power supply rail for the relevant component and used ohm’s law (V=I * R) to measure the current consumption. Rice and Hay [RH10] conducted a study where they profiled energy consumption of the devices connected to 802.11 wireless networks. To measure power consumption, they used a similar approach as Carroll and Heiser. They inserted a shunt resistor in series with the power supply and the mobile phone to measure overall energy consumption by the mobile device. They used a high-precision 0.02Ω resistor in series with battery terminal and its connector on the phone. In both approaches, they used a shunt resistor in series with the component and power supply to measure the current consumption. These two approaches fall into the category of offline approach to estimate the energy consumption.

A. Milenkovic et al. [MMJ+05] suggested that run-time energy consumption measurement is critical for studies that are related to target power optimization. They proposed two approaches to measure the power consumption of a node in a wireless sensor network. In the first approach, they sampled power supply and the output voltage using a current probe. In the second approach, they sampled power supply and the voltage at the shunt resistor. S. Mijovic et al. [MCB15] presented a method to perform real-time measurement of the energy consumption by a Wireless Body Area Network (WBAN). They used a shunt resistor in series with the power supply to measure the overall current consumption by a wireless node. In this work, their emphasis was to design a MAC protocol that is both energy efficient and provides good latency.
2.2 Software based Energy Estimation

Pathaket et al. [PHZ11] proposed a system-call based power consumption modeling approach. They proposed that the power consumption is estimated based on the system calls sent to the Operating System (OS). When an application wants to access an I/O or memory driver this call goes through an OS and the proposed model can estimate how much energy will be consumed after the completion of this call. This model performs run-time power estimation and provides fine-grained as well as application level energy consumption. We have categorized these two approaches in the area of the software-based offline method to predict energy consumption.

Li and John [LJ03] developed an OS’s energy profile using a wide spectrum of applications. They proposed various models to effectively estimate run-time energy dissipation of an OS. Dzhagaryan et al. [DMMJ16] proposed an automated energy measurement method for applications running on Android mobile and bare-metal embedded computing devices. They exploited different hardware and software aspects of the application and several other approaches to perform run-time power measurement. For energy profiling, they used a battery simulator provided by the National Instruments. The battery simulator provides an unobtrusive, high-resolution and high-frequency sampling of the current drawn by the computing device.

F. Jalali et al. [JHA16], proposed flow-based and time-based models to measure energy consumption by a Nano Data Center (nDC). nDCs are highly distributed servers that work on the peer-peer basis to host and transmit content and applications. nDCs are becoming popular to be used as local servers for IoT services. In the flow-based model, the equipment’s power consumption is calculated with respect to overall data flows passing through the equipment. Similarly, in the time-based model, energy consumption is measured in terms of time that an equipment spends when performing a cloud service. Martinez et al. [MMVP15] presented a general methodology about how to model the energy consumption of a node in a wireless network at a pre-deployed or pre-production stage. In this approach, they took a system-level approach and considered all the energy intensive processes like networking, sensing, processing, and acquisition capabilities to estimate a node’s power consumption. Shnayder et al. [SHC04] presented a scalable simulation environment to estimate an accurate per-node power consumption in a wireless sensor network.
2.3 Specialized Hardware based Approach

Homb [Hom16] proposed a run-time approach to measure system-wide energy consumption using a specialized hardware. He performed real-time measurements of the current and voltage of the target device and stored the data on an SD card. This data was then manually transferred to a computer for further analysis. The only drawback of this approach is to process a bulk amount of data which adds to the overall energy consumption of the device.

Shin et al. [SSJ+02] also used the approach of using a specialized hardware to provide power consumption. They proposed an integrated hardware connected to the target embedded device and an associated software to deliver the data of power consumption. They have used ARM7TDMI as the integrated hardware to perform on-board energy measurements. Their model estimates an application’s energy consumption in a hybrid fashion. The energy consumption of the CPU core is directly measured by the integrated tool, however, to estimate the power consumption of the memory, they have proposed a memory-power model. In this model, they predict the memory’s power consumption by using collected memory traces of the system’s memory power consumption. In this method, they have employed all the four approaches; offline, online, software, and hardware, to estimate the power consumption.

2.4 CPU Instruction-set based Energy Estimation

Acevedo et al. [APJCA10] proposed the use of static code simulation and microprocessor’s power model, to predict the energy consumption of a program running on an embedded device. Their model considered three types of energy costs; the cost of execution of the instruction itself, the switching cost between instructions, and the cost of cache-miss and branch misprediction. This is a one-time process to estimate the power and energy of the applications running on a microprocessor. One must perform this process again if the Micro-Controller Unit (MCU) has been changed. In their work, they have shown that an error of energy estimation is 7% and 14.6% respectively. This approach is suitable for static code analysis. Similarly, this approach is also specific to estimate the cost of the processor itself and ignores the energy consumption by peripheral devices.

Bazzaz et al. [BSE13] suggested an instruction-level energy estimation model for embedded systems. They used an ARM7TDMI-based microcontroller to determine the model parameters. In their model, the total energy consisted on the energy
consumption of processor core, Flash memory, memory controller, and SRAM. Their experimental results were based on several embedded applications from MiBench benchmark suite [GRE+01] and showed the estimation error is less than 6%.

### 2.5 Energy Modeling using Parametric Approach

Laurent et al. [LJSM04] proposed a technique that used the functional and parametric modeling of the processor to estimate the runtime power consumption. In their approach, they extracted the important parameters of the target application and platform that significantly contributed to the power consumption. Then they developed some elementary assembly programs (called scenarios) and measured their power and energy consumption. Finally, these values were used to develop a regression model to estimate the power consumption for different application and hardware platforms. They have used the offline software-based approach to create an energy estimation model. In their approach, the average error in estimation varies upon different applications and processors.

Ktari and Abid [KA07], also used the parametric models based on the architecture and the algorithm to estimate energy consumption of Digital Signal Processing Applications. The estimation errors of their model are 7.1% and 7.6% for finite impulse response (FIR) application running on C6701 and C5501 processors respectively. Similarly, they observed estimation errors of 4% and 6.6% for the applications that computed Fast Fourier Transforms (FFT).

Bircher and John [BJ12] proposed that the processor’s performance counters can be used to measure CPU’s power consumption. They used the performance-related events within a microprocessor like cache misses and DMA transactions to determine power consumption in memory, disk, and other subsystems outside microprocessor. Based on performance matrix, they developed system-specific models to measure the power consumption of six subsystems: microprocessor, graphics processing unit (GPU), chipset, memory, I/O, and disk. Their results showed that these models produced an average error less than 9% per subsystem by considering different workloads.

### 2.6 Black-box

You et al. [eYHAC10] suggested an energy estimation technique that can model energy consumption under various combinations of devices’ states without the use
of additional hardware. They have used the nominal power consumption from the specifications of related devices to estimate the energy consumption. The technique of relying on the datasheet specification for energy estimation is called black-box energy estimation and it is suitable for the cases where you can’t connect any external hardware to measure the power consumption.
We have selected the shunt resistor approach as used by Carroll and Heiser [CH10] to measure power consumption of the sensing application. They measured the physical power consumption of each component by inserting a sense resistor on the power supply rail for each relevant component. Since the tester knows the voltage provided to each component therefore, they used ohm’s law \( V = I \cdot R \) to measure the current consumption. Once we know the current consumption the power can be computed using formula \( P = V \cdot I \).

In our approach, we have used the shunt resistor approach to measure the power consumption of each activity phase. Since the activity phases are defined by the application. Therefore, we will observe the overall power consumption by the application and then identify the distinct activity phases.

We selected the shunt-resistor approach for on-chip power measurement for two reasons. Firstly, we wanted to develop a platform and target independent current measurement setup. This method is considered best suited since it doesn’t restrict us to use a specific hardware platform. Any device that allows us to use external power supply to power up the target device can be used. Secondly, our main objective was to physically measure the power consumed by an IoT node and didn’t want any discrepancy in our measurements produced in a simulation environment.
3. POWER MEASUREMENT SETUP

3.1 Measurement Setup

There are three elements in our experimental setup:

1. the IoT device under test
2. high resolution digital oscilloscope
3. a stable power supply
4. current sense resistor

Following figure explains the design:

![Diagram of power measurement setup]

**Figure 3.1**: Power Measurement Setup

For the experimental purposes we have used mDot\(^1\) by Multiconnect as the IoT device. We will use a digital oscilloscope to measure the energy consumption by different network applications. A current sense resistor is connected in series with power supply and IoT device to measure overall current consumption by the device. It is a Through Hole Current Sense Resistor; PWR221T-30-10R0J. It has 10Ω resistance, the power rating is 30 W and resistance tolerance is ± 5% . The output from the oscilloscope is directly sent to the Desktop PC where all the readings from the oscilloscope will be logged. PicoScope R6.8.2 software is running on Desktop PC to capture data from the digital oscilloscope. The oscilloscope attached to Desktop

\(^1\)http://www.multitech.com/brands/multiconnect-mdot
PC is running a Windows-10 operating system. We used PC Oscilloscope PicoScope 6000 series model 6404D. Following is a list of specifications:

- 4 channels
- 500 MHz bandwidth
- 5 GS/s real-time sampling frequency
- 2 GS ultra-deep memory
- 170,000 waveforms per second
- Arbitrary waveform generator (AWG)
- USB 3.0 interface

3.2 Measurement Procedure

We adopted two methods to measure the current drawn by the sensing application running on IoT device. In the first approach, we are manually identifying the boundaries of the activity phase on the oscilloscope using the PicoScope-6 software. In this method, we are using the DC-Average function provided by the PicoScope-6 software to find the average current consumption in a given time range.

In the second approach, we wrote a Python script, that starts capturing data when a threshold has been exceeded. The Python script samples the data from the oscilloscope based on the set sampling frequency and the capture length. After the data has been logged by the oscilloscope, the Python script finds the ending boundary of the signal and calculates the mean and standard error of the captured waveform. To generate a fixed threshold value, we used the PA_2 pin on mDot to signal the oscilloscope to start capturing the data. In order to measure power consumption in distinct activity phase the application developer can send signal on PA_2 before entering and after exiting the activity phase. The python script uses these signal values and computes the mean power and energy consumption in between the signals. The complete Python script can be found in the Appendix. A. It takes following values as input:

1. the threshold value for signal channel
2. the capture length
3. number of samples to capture
This script generates an excel report where it outputs the measurement of current, power and energy consumption within a threshold signals. Following is the screenshot of its output:

![Figure 3.2: mDot Current Consumption measurement using Python Script](image)

**Figure 3.2:** mDot Current Consumption measurement using Python Script
Chapter 4

Power and Energy Consumption Analysis

To analyze the power and energy consumption by different activity phases in a sensing application, we will conduct various experiments in this chapter. These activity phases are related to the GPIO, different energy modes, radio transmission and reception, cryptographic operations, and sensor related operations. The experiments in this chapter are conducted on mDot IoT hardware platform provided by Multiconnect [Mul16]. The energy consumption in these experiments is calculated by using following formula:

$$ E = \mu(P) \times \Delta t $$

(4.1)

In the Eq. 4.1, $\mu(P)$ represents the mean power consumption and $\Delta t$ expresses the duration of an activity phase. To calculate $E(P)$ we have used following formula:

$$ P = \mu(I_{mDot}) \times V_{mDot} $$

(4.2)

In the following experiments, we have fixed the voltage across the mDot, and $\mu(I_{mDot})$ is the average current consumption. The current consumption is calculated across the 10 $\Omega$ sense resistor using an oscilloscope as explained in the Chapt. 3. In these experiments, the calculated energy is the average over 10 samples extracted by the oscilloscope.

The standard error in energy is calculated by multiplying standard deviation in $\Delta t$ and power. The standard error in power is equivalent to the standard error in current consumption measurement because we are assuming the voltage across mDot will remain constant. Following is the formula for compound variance:

$$ \sigma^2(\Delta t, I_{mDot}) = \sigma^2(\Delta t)\sigma^2(I_{mDot}) + (\mu(\Delta t))^2\sigma^2(I_{mDot}) + (\mu(I_{mDot}))^2\sigma^2(\Delta t) $$

(4.3)

Using the above formula, we calculated the variance in the energy consumption. The standard error is square root of variance; $\sigma(\Delta t, I_{mDot}) = \sqrt{\sigma^2(\Delta t, I_{mDot})}$. In the following experiments, we used the Eq. 4.3 to compute the standard error in the energy.
The experiments are organized as hypothesis, method and results sub-sections. They are organized as follows:

1. LED: In this section, we will observe how an LED connected to a GPIO influences the power consumption of mDot.

2. Energy Modes: In this section, we will discuss the impact on the power consumption of mDot by changing energy modes.

3. peer-to-peer Radio Transmission: Here we will understand the impact of mDot radio transmission on the power consumption and how the transmission packet length influences the energy consumption.

4. peer-to-peer Radio Reception: In this section, we will observe the length of receiving packet to the energy consumption of mDot.

5. Cryptography: The impact of cryptographic operations on the power consumption is observed in this section.

6. Sensor: In this section, we will observe the difference of energy consumption when different sensors are used. We will currently focus on particle and temperature sensors.
### 4.1 LED

**Hypothesis:** The mDot consumes more energy when it is used to power up an LED and the energy consumption of mDot increases linearly with respect to the number of LEDs attached.

**Method:** In this experiment, we will investigate the energy consumption of the LED when it is attached to the mDot. In IoT devices, an LED is powered up by attaching it to a GPIO pin of the embedded device. We used L05R3000F1 Red LED, whose maximum current rating is 20 mA and requires a voltage of 2.2 V across its terminals. The LED will be destroyed if more than 20 mA of current follows through it. To protect the LED, a current limiting resistor is attached in series with the LED and the GPIO pin. The formula to calculate the resistor value is as follows:

\[
R = \frac{(V_G - V_L)}{I_L}
\]  \hspace{1cm} (4.4)

The mDot is powered up by 5 V and the voltage observed across the GPIO pin is 2.7 V. The mDot has a microcontroller STM32F411RE which controls the GPIO of mDot. The datasheet of STM32F411RE confirms that a GPIO can safely draw 20 mA of current. Therefore, using Eq. 4.4 the minimum value of the resistor should be 25 Ω [STM16].

In this experiment, we used a resistor of 56.20 Ω and as per the formula, the current drawn by the LED during high state should be 8.89 mA. To compare the energy consumption when the LED is high and low, a C++ program for mDot was developed where the GPIO pin was configured in output mode and its state was toggled after every 200 ms. Following is the code snippet:

**Source code 4.1 C++ Program: Toggling LED**

```cpp
DigitalOut led1(PA_2);
int main(){
    while(true){
        led1 = !led1;
        Thread::wait(200);
    }
    return 0;
}
```
Similarly, to investigate the energy consumption when two LEDs light up, following code snippet was used.

### Source code 4.2 C++ Program: Toggling two LEDs

```c++
#include "mbed.h"
#include "mDot.h"

DigitalOut led1(PA_2);
DigitalOut led2(PA_0);

int main()
{
    while(true){
        led1 = !led1;
        led2 = !led2;
        Thread::wait(200);
    }
    return 0;
}
```

**Results:** The results gathered from this experiment shows that the current consumption is higher when the LED is powered ON. The current consumption by the mDot is observed to be 32,02 mA when the LED is in the *high state* and 19,88 mA when it is at a *low state*. Using the formula in the Eq. 4.2, the power consumption during high and low states is 0,160 W and 0,099 W respectively. Similarly, to compute energy consumption by the mDot Eq. 4.1 was used. The $\Delta t_{mDot}$ in the current scenario is the time mDot spent either in high or low state. Since in this experiment the LED remains the high and low for 200 ms, therefore, $\Delta t_{mDot}$ is 200 ms for both the cases. The mDot consumes 0,0199 J of energy when the LED is in low-state and 0,032 J of energy when LED is in the *high state*. These results indicate that the mDot consumes 0,0121 J of more energy when the LED is in the *high state*. 
Following figure shows the waveform of current consumption observed on the
digital oscilloscope.

![Waveform](image)

**Figure 4.1:** mDot Current Consumption when Powering a single LED

Two LEDs were attached to the mDot, in order to verify that the energy con-
sumption increases linearly. We obtained following figure from the oscilloscope when
two LEDs were connected.

![Waveform](image)

**Figure 4.2:** mDot Current Consumption when Powering two LEDs

The DC-average of the current consumption of mDot is 39,31 mA when both the
LEDs are at a high state and 14,66 mA when both are at a low state. Using the
Eq. 4.2, the power consumption is calculated to be 196,55 mW when LEDs are at the
high state. It is 73,3 mW when LEDs are in the low state. Similarly, using Eq. 4.1,
the mDot consumes 0,03931 J of energy when the LEDs are at the high state and
0,01466 J when the LEDs are at the low state.
Following table summarises the results obtained:

**Table 4.1: LED: Energy Consumption**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>1 LED</th>
<th>2 LEDs</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current_ON(mA)</td>
<td>32.02</td>
<td>39.31</td>
<td>7.29</td>
</tr>
<tr>
<td>Current_OFF(mA)</td>
<td>19.88</td>
<td>14.66</td>
<td>5.22</td>
</tr>
<tr>
<td>Difference</td>
<td>12.14</td>
<td>24.65</td>
<td>12.51</td>
</tr>
<tr>
<td>Power_ON(W)</td>
<td>0.160</td>
<td>0.196</td>
<td>0.36</td>
</tr>
<tr>
<td>Power_OFF(W)</td>
<td>0.099</td>
<td>0.0733</td>
<td>0.026</td>
</tr>
<tr>
<td>Difference</td>
<td>0.061</td>
<td>0.1227</td>
<td>0.0617</td>
</tr>
<tr>
<td>Energy_ON(J)</td>
<td>0.032</td>
<td>0.0392</td>
<td>0.007</td>
</tr>
<tr>
<td>Energy_OFF(J)</td>
<td>0.020</td>
<td>0.014</td>
<td>0.006</td>
</tr>
<tr>
<td>Difference</td>
<td>0.012</td>
<td>0.0252</td>
<td>0.0132</td>
</tr>
</tbody>
</table>

Tab. 4.1, shows that the current drawn by the LED is 12.14 mA that is 4 mA higher than what was calculated. However, when two LEDs were used the current drawn by the mDot doubles. Tab. 4.1 also confirms our hypothesis that the mDot consumes more energy when the LED is at the *high state*. Similarly, the table also proves our hypothesis that the energy consumption increases linearly by increasing the number of LEDs.
4.2 Energy Modes

**Hypothesis:** The current consumption during sleep and deep sleep modes is less than the active mode.

**Method:** To measure energy consumption during sleep mode we used the sleep Application Program Interface (API)s provided by the mDot library.\(^1\) We implemented a *for loop* equivalent of duration 292 ms to ensure that the mDot remains in the active mode. We used the *timer* provided by *mbed* to find exact time during the execution of the *for loop*. We have also used *MTSLog* to output the log messages from the program running on mDot using the serial port. This is used to confirm whether the mDot is entering different energy modes. Following code snippet shows how to configure MTS logger in mDot source code:

\(^1\)https://developer.mbed.org/teams/MultiTech/code/libmDot-dev-mbed5/
# Source code 4.3 C++ Program: MTS logger

```cpp
#include "mbed.h"
#include "mDot.h"
#include "MTSLog.h"

//initializing the USB to communicate with PC
Serial pc(USBTX, USBRX);
int main(){

    //setting baud rate for communication
    pc.baud(115200);

    // setting up the logging level
    mts::MTSLog::setLogLevel(mts::MTSLog::TRACE_LEVEL);

    Timer t;
    while(true){
        volatile int32_t j=0;

        t.start();
        for (; j<4; j++){
            for (volatile int32_t i=0; i<1000000; i++);
        }
        t.stop();
        logInfo("times in milliseconds: %d", t.read_ms());
        t.reset();
    }
    return 0;
}
```

To read the log messages sent over USB by mDot, a Python script has been used. This program reads the messages from mDot via a COM port and prints them on a terminal. Algo. 4.4 presents the Python script that receives mDot messages and outputs them to the terminal. This Python script takes the baud rate and port as input.
To automatically measure energy consumption during active state, we have used Python script developed in Chapt. 3. To signal the oscilloscope, we have used a GPIO pin. We generated a pulse of width 100 \( \mu \)s as an indicator to measure the current consumption from one point to the other as shown in Algo. 4.5. We measure the current consumption and duration from the oscilloscope and then use formulas to calculate the power and energy consumption.
Source code 4.5 C++ Program: Generate Event for Automatic Energy Measurement

```c++
void generate_event()
{
    DigitalOut signal(PA_2)
    signal = 1;
    wait_us(100);
    signal = 0;
}
```

The Algo. 4.3 has been modified such that it computes the current consumption during active mode.

Source code 4.6 C++ Program: Active Mode Current Consumption

```c++
int main(){
    while(true){
        volatile int32_t j=0;
        generate_event();
        for (; j<4; j++){
            for (volatile int32_t i=0; i<1000000; i++);
        }
        generate_event();
        //sleep for 1 second
        dot->sleep(1, mDot::RTC_ALARM, false);
    }
    return 0;
}
```

The Python script was run with following values to calculate power and energy consumption:

Listing 4.1: Python Script Run Command Active Mode

```
python energy.py -e Dot-Sleep_MTS_MDOT_F411RE_1s_Active
-t -0.300 -s 10 -F 100 -v 5.0 -c 500
```
The next step was to measure current consumption during sleep mode. The mDot sleep API is used to enter the sleep mode. The sleep API in mDot takes three parameters. The first parameter specifies the number of seconds mDot remains in sleep mode. The second parameter defines the wake-up event for the mDot. The wake-up event for the mDot can be triggered by the real-time counter or an external interrupt. When the wake-up event is set `mDot::RTC_ALARM`; it instructs the mDot to wake-up after the interval specified in the first parameter. The last parameter specifies whether or not the mDot will go to deep sleep. This API allows us to enter either **deep sleep** (standby) or **sleep** (stop) mode. In the present scenario, we have configured the mDot to enter the sleep mode for one second. Following is the code snippet to measure current consumption in the sleep mode:

Source code 4.7 C++ Program: Entering Sleep Mode

```cpp
#include "mbed.h"
#include "mDot.h"

int main(){
    mDot* dot = NULL;

    // reset to default configuration
    dot->resetConfig();

    while(true){
        volatile int32_t j=0;
        for (; j<4; j++){
            for (volatile int32_t i=0; i<1000000; i++);
        }

        // sleep mode energy mode
        generate_event();
        dot->sleep(1, mDot::RTC_ALARM, false)
        generate_event();
    }
    return 0;
}
```
To calculate power and energy consumption, we used Python script with the following input values:

**Listing 4.2:** Python Script Run Command Sleep Mode

```
python energy.py -e Dot-Sleep_MTS_MDOT_F411RE_1s_Sleep
-t -0.300 -s 10 -F 5 -v 5.0 -c 1500
```

We used MTSlog to display the log to confirm that mDot is going into sleep mode. Moreover, we checked the how mDot behaves when we set the `deep sleep` to `TRUE` by using the following API:

```
dot--sleep(1,mDot :: RTC_ALARM,true)
```

The following figures show the log output when `sleep` and `deep sleep` modes were activated respectively.

![Figure 4.3: mDot going into sleep mode](image1)

![Figure 4.4: mDot not going into deep-sleep mode](image2)

Fig. 4.3 indicates that the mDot goes into sleep mode after every 1s. However, Fig. 4.4 indicates that mDot does not support `deep sleep` mode. In this sub-section we have developed the source code as well as current measurement instructions for active, sleep and deep sleep modes and also confirmed that mDot doesn’t enter the deep sleep mode.

**Results:** The mDot is powered on with 5 V and oscilloscope is being used to measure the current consumption when mDot goes into different states. First, let’s see the screenshot from the oscilloscope when the mDot is in the active state for 292.5 ms and then goes to the sleep state for one second.


Fig. 4.5 shows that the sleep state can be further divided into three phases based on the current consumption waveform. Therefore, we have divided the sleep state into pre-, post- and main sleep phases. On a side note, we did some experiments where we varied the duration of sleep state and discovered that the duration of these phases remains constant.

Fig. 4.5 indicates that the mDot spends $\sim 161.91$ ms in post-sleep phase and draws $\sim 21.78$ mA of current. The execution of for loop takes $\sim 292.5$ ms and consumes a current of $\sim 37.68$ mA. Similarly, the pre-sleep phase takes $\sim 43.99$ ms and consumes $\sim 12.48$ mA of current. During the main sleep phase, the mDot consumes $\sim 240.57$ $\mu$A of current for the duration of $\sim 954.15$ ms.

These values are manually measured by the author using the digital oscilloscope. They are averaged over manually measured 10 samples from the oscilloscope using the DC-Average function. Since the measurement obtained from the Python script was an averaged value over the complete sleep state including the pre- and post-phase; therefore, we have used the manual method to separately measure the current consumption of theses pre- and post- phases.
Tab. 4.2 summarizes the calculated energy consumption by different phases in the sleep state.

**Table 4.2**: Sleep Mode Energy Consumption (E-05 in this table is equivalent to $\times10^{-5}$ and so on.)

<table>
<thead>
<tr>
<th>States</th>
<th>Duration(ms)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Sleep</td>
<td>0,044±7,62E-05</td>
<td>0,012±6,65E-05</td>
<td>0,062±6,65E-05</td>
<td>0,0027±3,07E-06</td>
</tr>
<tr>
<td>Sleep</td>
<td>0,954±1,04E-03</td>
<td>2,41E-04±3,42E-05</td>
<td>0,001±3,42E-05</td>
<td>0,0011±3,26E-05</td>
</tr>
<tr>
<td>Post-Sleep</td>
<td>0,162±2,41E-03</td>
<td>0,022±1,59E-04</td>
<td>0,109±1,59E-04</td>
<td>0,018±5,84E-05</td>
</tr>
</tbody>
</table>

According to the developer’s guideline for mDot [Mulb], the current consumption during sleep mode should be 50µA that is far less than the observed value. After some investigation, it was observed that a noise floor is present in our readings from the oscilloscope. We calculated that the oscilloscope measures a signal whose DC-average value is 216.7µA. The presence of this noise will not allow us to measure current consumption in the range of 50µA.

Fig. 4.6 and Fig. 4.7 represent the waveform and spectrum view of the observed noise floor. The spectral view shows that the noise is maximum at near the frequency of 0Hz which is DC signal.

**Figure 4.6**: Noise Floor Waveform View

**Figure 4.7**: Noise Floor Spectral View
Since the mDot is not entering the deep sleep mode, we expected to see no change in the power consumption. However, the oscilloscope waveform view showed that there are some variations in the current consumption of sleep and deep sleep modes. Fig. 4.8 is the captured image from oscilloscope which shows the current drawn by the mDot when deep sleep mode was set:

![Figure 4.8: mDot current consumption during deep sleep mode](image)

The waveforms of the post-sleep phase are identical in both sleep modes (sleep and deep sleep). Similarly, by comparing Fig. 4.5 and Fig. 4.8 an inconsistency during the execution of for loop can also be observed. In the sleep mode, the current consumption remains at 37.65 mA throughout the completion of for-loop. However, in the deep sleep mode, it goes to 37.75 mA for 70 ms and then drops to 28.22 mA for 222.2 ms. This variation in the deep sleep mode results in current consumption of 30.65 mA during the execution of for loop. Similarly, in the pre-sleep phase, we see two levels of current consumption; one at 24.48 mA and the other one at 12.55 mA. This resulted in an increased consumption current of value 20.25 mA and an increased duration of 129.7 ms in the deep sleep mode. Moreover, the current drawn during the main deep sleep phase is 238.5 µA and the duration is 951 ms.

We used the manual method to measure current consumption in the deep sleep mode because the PA_2 pin, to trigger the oscilloscope measurement, of mDot remains high in deep sleep mode as shown in Fig. 4.9. In Fig. 4.9, the Channel B represents the state of the PA_2 and the Channel A is the current consumption waveform during the deep sleep operation. This will lead to an erroneous calculation of current consumption in the deep sleep mode. Moreover, since the deep sleep mode
is again divided into three different phases; therefore, we used the manual method to separately measure the current consumption.

![Figure 4.9: Issue in mDot Automatic Current Consumption in Deep Sleep Mode](image)

The current consumption for active mode is measured using Algo. 4.6, where the mDot only executes a for loop. We have used the Python script to measure the current consumption during the active phase.

Tab. 4.3 presents a comparison of energy consumption under different modes of operation. In this table, we have averaged out the current consumption of complete state. The current consumption of pre- and post- phases has been added to the main phase to present an overall current consumption comparison among different modes of operation. We have used Python script to measure average current consumption during active and sleep modes. However, we couldn’t do the same for deep sleep due to the problem presented in Fig. 4.9. Hence, the results presented in the deep sleep mode are the DC-Average values of the complete state.

<table>
<thead>
<tr>
<th>States</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>0,292±3,17E-06</td>
<td>0,031±5,47E-05</td>
<td>0,154±5,47E-05</td>
<td>0,045±1,60E-05</td>
</tr>
<tr>
<td>Sleep</td>
<td>1,162±2,12E-03</td>
<td>3,89E-03±3,18E-05</td>
<td>0,019±3,18E-05</td>
<td>0,022±3,78E-05</td>
</tr>
<tr>
<td>Deep-Sleep</td>
<td>1,245±1,92E-03</td>
<td>5,13E-03±2,63E-05</td>
<td>0,026±2,63E-05</td>
<td>0,032±3,42E-05</td>
</tr>
</tbody>
</table>

Tab. 4.3 confirms that the mDot consumes more power in the active mode as compared to the two sleep modes. However, we observe more energy consumption
during *deep sleep* mode in contrast to *sleep* mode because our mDot version does not *deep sleep* mode.

### 4.3 Peer-to-Peer Radio Transmission

**Hypothesis:** The power consumed by the mDot increases linearly by increasing the number of bytes sent in a packet.

**Method:** To calculate power consumption during peer-to-peer communication, two mDots were configured to send and receive LoRaWAN frames after every one second. The following C++ code snippet explains the mDot transmit and receive process.

**Source code 4.8 C++ Program: mDot LoRA peer-to-peer**

```cpp
    dot = mDot::getInstance();
    dot->resetConfig();

    if (dot->getJoinMode() != mDot::PEER_TO_PIER) {
        dot->setJoinMode(mDot::PEER_TO_PIER);
    }
    frequency_band = dot->getFrequencyBand();
    tx_frequency = 869850000;
    tx_datarate = mDot::DR6;
    tx_power = 4;

    update_peer_to_peer_config_p2p(network_address, network_session_key,
                                    data_session_key, tx_frequency,
                                    tx_datarate, tx_power);

    while (true) {
        std::vector<uint8_t> tx_data;
        if (!dot->getNetworkJoinStatus()) {
            join_network_p2p();
        }
        for (uint32_t i=0; i<25; i++){
            tx_data.push_back((i%60)+33);
        }
        dot->send(tx_data);
        wait(1);
    }
```
This is an example program provided by MultiTech to transmit and receive data packets in peer to peer mode using LoRaWAN as the communication protocol [mula]. In the Algo. 4.8, the mDot first resets to the default configuration. Then the network join mode is set to peer-to-peer. In the next step, the network address, session keys, data rate, transmission frequency and power are configured. In the final step, mDot joins the network, creates a dummy data frame and transmits it.

According to the specifications of LoRaWAN, the maximum frame size using the configurations in this example is 243 bytes [SLE⁺15]. To support our hypothesis, we used 10 frames of increasing lengths and measured the current drawn during transmission. The mDot can transmit packets of different sizes by modifying the for loop iterator defined in the Algo. 4.8. We will increase the length of the frame each time by adding 24 more bytes and then use the oscilloscope to measure the current drawn and the duration of these transmissions.

For each set of data frames, we manually measured the current drawn by mDot using the oscilloscope. We took 10 samples for each data length and calculated mean value. We also did an automated analysis of the current consumption by the mDot using the Python script. In the automated analysis, we used a signal probe; PA₂, to instruct the oscilloscope when to start capturing the data from the measure probe. Algo. 4.9 presents the code snippet used for automatic energy consumption measurement.

**Source code 4.9 C++ Program: mDot LoRA peer-to-peer Automatic Measurement**

```cpp
while (true) {
    std::vector<uint8_t> tx_data;
    if (!dot->getNetworkJoinStatus()) {
        join_network_p2p();
    }
    for (uint32_t i=0; i<25; i++){
        tx_data.push_back((i%60)+33);
    }
    generate_event();
    dot->send(tx_data);
    generate_event();
    wait(1);
}
```

The `generate_event()` function generated a pulse of 100 µs. It is used to signal the oscilloscope to start capturing data. The execution of Algo. 4.9 gave us the
current consumption during the `send(tx_data)` function. In the result section, we will first present the results of the manual process and then the data gathered during automated process.

**Results:** During these experiments, we observed that the energy consumption of mDot increases linearly with respect to packet size. Similar to the previous section, we discovered that the frame transmission operation also consists of three phases. These three phases have been identified based on the current consumption observed in these phases. Fig. 4.11 presents the classification of these phases when the transmission frame length is 25 bytes.

![mDot Transmit Cycle](image)

**Figure 4.10: mDot Transmit Cycle**

During these experiments, we observed that the variations in the duration and current consumption during pre- and post- radio transmission phases are negligible when we increased the frame length. However, the duration of radio transmission phase increased linearly with the frame size. Fig. 4.11 displays the variation of energy consumption during these different phases with respect to frame length.
In Fig. 4.11, we observed that the current consumption during radio transmission phase is almost linear. Similarly, the variation in the pre-transmit and pos-transmit is insignificant. These small variations led to a conclusion that during these phases the mDot is performing constant operations that are not related to the packet length. Tab. 4.4 enlists the power consumption by these different phases under various frame lengths.

Table 4.4: Power Consumption in Pre-, Post- and Main Transmission Phases (E-05 in this table is equivalent to $\times 10^{-5}$ and so on.)

<table>
<thead>
<tr>
<th>PacketSize(B)</th>
<th>$P_{Pre-Transmit}$(W)</th>
<th>$P_{Transmit}$(W)</th>
<th>$P_{Post-Transmit}$(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0789±4.80E-05</td>
<td>0.3247±3.15E-04</td>
<td>0.1908±6.01E-04</td>
</tr>
<tr>
<td>25</td>
<td>0.0768±8.43E-05</td>
<td>0.3244±2.79E-04</td>
<td>0.1856±9.61E-04</td>
</tr>
<tr>
<td>49</td>
<td>0.0769±6.14E-05</td>
<td>0.3191±8.88E-04</td>
<td>0.0570±0.018</td>
</tr>
<tr>
<td>73</td>
<td>0.0777±1.78E-05</td>
<td>0.3165±1.69E-04</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>0.0777±4.74E-05</td>
<td>0.3290±3.97E-05</td>
<td>0.1849±9.42E-04</td>
</tr>
<tr>
<td>121</td>
<td>0.0776±3.14E-05</td>
<td>0.3265±9.45E-05</td>
<td>0.1888±9.92E-04</td>
</tr>
<tr>
<td>145</td>
<td>0.0785±6.81E-05</td>
<td>0.3218±1.12E-04</td>
<td>0</td>
</tr>
<tr>
<td>169</td>
<td>0.0780±9.51E-05</td>
<td>0.3248±5.30E-05</td>
<td>0.1513±4.29E-04</td>
</tr>
<tr>
<td>193</td>
<td>0.0779±8.14E-05</td>
<td>0.3246±2.53E-04</td>
<td>0.1719±1.33E-03</td>
</tr>
<tr>
<td>217</td>
<td>0.0783±9.14E-05</td>
<td>0.3256±6.46E-05</td>
<td>0.1836±8.69E-04</td>
</tr>
<tr>
<td>241</td>
<td>0.0800±6.44E-05</td>
<td>0.3263±6.21E-05</td>
<td>0.1885±4.80E-04</td>
</tr>
</tbody>
</table>
The author also performed an automated analysis of the current consumption on mDot. Following command was executed to capture current samples when packet size was 25 bytes:

Listing 4.3: Run Python Script Transmission

```
python energy.py -e Dot -P2P_MTS_MDOT_F4I1RE_1s_25B
    -t -0.300 -s 10 -F 100 -v 5.0 -c 280
```

In this case, the threshold voltage is -0.3. The number of samples to collect from the experiment is 10. The sampling frequency is 100 MHz. The voltage provided to mDot was 5.0 V and the capture length is 280 ms. The capture length and sampling frequency will be changed to gather data for different frame lengths. The Python script for collecting power and energy consumption measurements was executed for all the packet lengths. Fig. 4.12 presents the energy consumed by the mDot with respect to different packet lengths.

![Transmission Energy Consumption](image)

**Figure 4.12:** Automatic: mDot senddata() Energy Consumption
Fig. 4.12 indicates that the energy consumption follows almost linear pattern with respect to packet size. Similarly, we plotted the graph for power consumption with respect to packet size. We observed a slight increase in the power consumption with respect to packet size as shown in Fig. 4.13.

**Figure 4.13:** Automatic: mDot_senddata() Power Consumption

Tab. 4.5 summarizes the data gathered from the automated analysis. These results indicate that the energy consumption increases linearly and the relationship between frame length and overall transmission duration is linear. We also found during this experiment that the transmission phase can be divided into three phase; pre-, post-, and main transmission phases.
Table 4.5: Energy Consumption: Transmission (E-05 in this table is equivalent to \( \times 10^{-5} \) and so on.)

<table>
<thead>
<tr>
<th>PacketSize(B)</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.139±9.92E-08</td>
<td>0.031±7.98E-05</td>
<td>0.157±7.98E-05</td>
<td>0.022±1.11E-05</td>
</tr>
<tr>
<td>49</td>
<td>0.150±5.41E-04</td>
<td>0.037±5.09E-05</td>
<td>0.175±5.09E-05</td>
<td>0.026±2.16E-05</td>
</tr>
<tr>
<td>73</td>
<td>0.169±9.50E-05</td>
<td>0.038±1.20E-04</td>
<td>0.192±1.20E-04</td>
<td>0.032±2.03E-05</td>
</tr>
<tr>
<td>97</td>
<td>0.190±8.61E-04</td>
<td>0.040±5.87E-05</td>
<td>0.201±5.87E-05</td>
<td>0.038±3.64E-05</td>
</tr>
<tr>
<td>121</td>
<td>0.210±3.46E-07</td>
<td>0.042±5.41E-05</td>
<td>0.211±5.41E-05</td>
<td>0.044±1.14E-05</td>
</tr>
<tr>
<td>145</td>
<td>0.220±6.06E-04</td>
<td>0.044±4.59E-05</td>
<td>0.223±4.59E-05</td>
<td>0.049±2.88E-05</td>
</tr>
<tr>
<td>169</td>
<td>0.240±1.06E-07</td>
<td>0.046±5.19E-05</td>
<td>0.229±5.19E-05</td>
<td>0.055±1.25E-05</td>
</tr>
<tr>
<td>193</td>
<td>0.260±7.81E-04</td>
<td>0.047±6.38E-05</td>
<td>0.237±6.38E-05</td>
<td>0.062±4.06E-05</td>
</tr>
<tr>
<td>217</td>
<td>0.281±4.56E-08</td>
<td>0.048±6.19E-05</td>
<td>0.241±6.19E-05</td>
<td>0.068±1.74E-05</td>
</tr>
<tr>
<td>241</td>
<td>0.301±3.89E-07</td>
<td>0.049±4.39E-05</td>
<td>0.245±4.39E-05</td>
<td>0.074±1.32E-05</td>
</tr>
</tbody>
</table>

4.4 Peer-to-Peer Radio Receive

**Hypothesis:** The energy consumption of mDot does not increase by increasing the number of bytes received.

**Method:** To prove this hypothesis, we used the Algo. 4.8 and enabled the receive event.

\( \text{dot} \rightarrow \text{setEvents}(&\text{events}) \)

The Algo. 4.9 for peer-to-peer communication indicates that the mDot joins the network, then transmits the data and waits for 1 second. During this wait state the mDot is interrupted with a receive event and the function \( \text{PacketRx()} \) defined in \( \text{mDotEvent.h} \) file is called as given in Appendix. B.2. This function then switches the control to the user defined call-back function \( \text{MacEvent()} \) that is defined in the \( \text{radio\_event.h} \) file as given in Appendix. B.3.

To confirm that the mDot is correctly receiving the sent bytes, we modified the call-back function. In the call-back function when the mDot receives the desired number of bytes, a pulse of 10 ms is generated on \( \text{PA\_2\ GIPO\ pin} \). In the case of failure, no pulse is generated in the receive phase. The generation of the pulse will result in increased current consumption but we will ignore it during the measurement.

In this experiment, we manually measured the current consumption from the oscilloscope using the \( \text{DC\_Average} \) function. We failed to perform automatic analysis.
because mDot receive process is interrupt driven. This interrupt is defined inside the library code and for the automatic measurements, we can’t generate a pulse in the library code. Since we can’t mark the start of the receive event, therefore, we used the manual method to gather data.

We conducted two experiments to measure current consumption when the frame payload length was 120 and 240 bytes respectively. These frame lengths were chosen to prove the relationship between frame length and the energy consumption. If the frame length and energy consumption is related then we will observe different energy consumption values for these two cases.

**Results:** After conducting the measurements for two different frame lengths we received following waveforms on the oscilloscope:

![Figure 4.14: Current Consumption when receiving LoRA frame of size 240B](image)

![Figure 4.15: Current Consumption when receiving LoRA frame of size 120B](image)

Fig. 4.14 and Fig. 4.15 indicate that there is not much difference in the current consumption when the frame length is reduced to half. The red curve in the Fig. 4.14 and Fig. 4.15 confirm that the packets have been received correctly by the mDot. To compare their energy consumption, we also performed a manual data analysis where we took 10 sample of data during frame receive event.

**Table 4.6:** Rx Energy Consumption: A Comparison (E-05 in this table is equivalent to $\times 10^{-5}$ and so on.)

<table>
<thead>
<tr>
<th>Size(Bytes)</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>0.187±3.59E-04</td>
<td>0.02394±6.42E-05</td>
<td>0.120±6.42E-05</td>
<td>0.0225±1.49E-05</td>
</tr>
<tr>
<td>240</td>
<td>0.188±1.91E-04</td>
<td>0.02395±5.19E-05</td>
<td>0.120±5.14E-05</td>
<td>0.0226±1.05E-05</td>
</tr>
</tbody>
</table>

Tab. 4.6 proves that the frame size does not influence the energy consumption of mDot in the process of receiving. The difference in the average duration is around
4.5 Cryptography

Hypothesis: By increasing the number of bytes to encrypt, the energy consumption increases linearly.

Method: To measure energy consumption during encryption, we implemented 128-bit AES encryption and decryption using mbed. We used mbedtls library to implement AES encryption. Following is the code snippet to encrypt 16*n bytes of data.

Source code 4.10 C++ Program: mDot Encryption

```c
#include "mbedtls/aes.h"
 mbedtls_aes_context aes_enc;

 // static key
 unsigned char key[16] = "itzkbgulrcjmnv";
 key[15] = 'x';

 unsigned char iv[16] = {0xb2, 0x4b, 0xf2, 0xf7, 0x7a, 0xc5,
     0xec, 0x0c, 0x5e, 0x1f, 0x4d, 0xc1,
     0xae, 0x46, 0x5e, 0x75};
 mbedtls_aes_setkey_enc( &aes_enc, key, 16*8 );
 mbedtls_aes_crypt_cbc( &aes_enc, MBEDTLS_AES_ENCRYPT,
     strlen((const char*)input), iv, input, output );
```

According to the implementation of mbedtls, the input to the encryption function should be a multiple of 16. The function `mbedtls_aes_crypt_cbc()` is called for both encryption and decryption. The macro MBEDTLS_AES_ENCRYPT instructs the function to perform encryption. We also implemented the decryption function to check whether the encryption done in the previous call is correct or not. For the decryption process, we need to re-initialize the IV (initialization vector) and the macro

---

for the `mbedtls_aes_crypt_cbc()` is set to `MBEDTLS_AES_DECRYPT`. Following is the code snippet used for decryption.

**Source code 4.11 C++ Program: mDot Decryption**

```c++
#include "mbedtls/aes.h"

mbedtls_aes_context aes_dec;

// static key
unsigned char key[16] = "itzkbgulrcjmnv";
key[15] = 'x';

unsigned char iv[16] = {0xb2, 0x4b, 0xf2, 0xf7, 0x7a, 0xc5,
0xec, 0x0c, 0x5e, 0x1f, 0x4d, 0xc1,
0xae, 0x46, 0x5e, 0x75};
mbedtls_aes_setkey_dec( &aes_dec, key, 16*8 );
mbedtls_aes_crypt_cbc( &aes_dec, MBEDTLS_AES_DECRYPT,
strlen((const char*)output), iv,
output, input );
```

We generated a variable length input by using a for loop. Following is the code snippet to generate variable length input used during encryption.

**Source code 4.12 C++ Program: Generate input**

```c++
unsigned char input[300] = {0};
for (int i=0; i<ENCRYPTION_LENGTH; i++)
    input[i] = (i % 60) + 33;
```

We calculated energy consumption by varying input length for encryption. We gathered data for the current consumption during encryption and decryption for seven different input data lengths. The number of bytes used for encryption varied from 16 bytes to 112 bytes where there is a difference of 16 bytes between the two samples. The Python script was used to automatically gather data for current consumption. We did not perform any manual measurements for these cryptographic operations.

**Results:** We observed following waveform of current drawn by the mDot while performing encryption and decryption operations. Fig. 4.16 is the waveform of
cryptographic operations; encryption and decryption, when the input data length is 16 bytes.

![Figure 4.16: Current Consumption during Cryptographic Operation](image)

The Fig. 4.16 shows the current consumption during cryptographic operations, where the data size is 16 bytes. By looking at this Fig. 4.16, it is clear that the duration for encryption and decryption is in micro-seconds. Since the duration is so small, therefore, the energy consumption during encryption and decryption will be insignificant.

Tab. 4.7 summarizes the power and energy consumption of the mDot during cryptographic operations while varying the length of input data. The data presented in Tab. 4.7 is an average of 10 current consumption samples observed by the oscilloscope. We have summed the current consumption during encryption and decryption processes and presented the accumulated data in Tab. 4.7 as the current consumption during both cryptographic operations. The data presented in Tab. 4.7 for power and energy is also an accumulation over both operations.
Table 4.7: Cryptography: Energy and Power Consumption (E-05 in this table is equivalent to $\times 10^{-5}$ and so on.)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>7.45E-05±8.09E-07</td>
<td>0.025±3.10E-04</td>
<td>0.128±3.10E-04</td>
<td>9.58E-06±3.11E-08</td>
</tr>
<tr>
<td>32</td>
<td>9.65E-05±1.00E-06</td>
<td>0.025±2.59E-04</td>
<td>0.129±2.59E-04</td>
<td>1.25E-05±3.60E-08</td>
</tr>
<tr>
<td>48</td>
<td>1.18E-04±7.66E-07</td>
<td>0.026±3.18E-04</td>
<td>0.132±3.18E-04</td>
<td>1.55E-05±4.26E-08</td>
</tr>
<tr>
<td>64</td>
<td>1.39E-04±4.91E-07</td>
<td>0.026±2.97E-04</td>
<td>0.132±2.97E-04</td>
<td>1.84E-05±4.34E-08</td>
</tr>
<tr>
<td>80</td>
<td>1.61E-04±8.26E-07</td>
<td>0.026±2.72E-04</td>
<td>0.132±2.72E-04</td>
<td>2.14E-05±4.92E-08</td>
</tr>
<tr>
<td>96</td>
<td>1.83E-04±1.31E-06</td>
<td>0.026±3.10E-04</td>
<td>0.132±3.10E-04</td>
<td>2.43E-05±6.67E-08</td>
</tr>
<tr>
<td>112</td>
<td>2.05E-04±1.18E-06</td>
<td>0.026±2.89E-04</td>
<td>0.133±2.89E-04</td>
<td>2.74E-05±6.72E-08</td>
</tr>
</tbody>
</table>

Fig. 4.17 and Fig. 4.18 show the energy and power consumption respectively during cryptographic operations with respect to data length.

![Energy Consumption Vs. Data Length](image)

**Figure 4.17:** mDot Cryptographic Energy Consumption
4.6. SENSORS

4.6. Sensors

**Hypothesis:** The temperature sensor will consume less energy than the particle sensor.

**Method:** In this experiment, we have interfaced temperature sensor with the mDot. We used DS18B20 1-Wire temperature sensor [Dal]. We configured PA_5 GPIO pin of mDot to read temperature value from the sensor. We also inserted a 4.7 kΩ resistor between the VDD and DQ pin of the temperature sensor.
Following is the code snippet used to interface the temperature sensor with the mDot: The automatic analysis has been performed to capture the current

**Source code 4.13 C++ Program: Temperature Sensor DS18B20**

```cpp
#include "mbed.h"
#include "DS1820.h"
#include "mDot.h"

#define DATA_PIN PA_5
// Temperature sensor object
DS1820 probe(DATA_PIN);

int main(){
  float temperature = 0.0;
  probe.setResolution(9);
  while( 1 ) {
    generate_event(1);
    // Start temperature conversion, wait until ready
    probe.convertTemperature(true, DS1820::all_devices);
    temperature = probe.temperature();
    generate_event(1);
    // Wait for one second
    wait(2);
  }
  return 0;
}
```

consumption during temperature sensing by the mDot. Following is the argument to the Python script to capture data:

**Listing 4.4: Run Python Script Temperature Sensor**

```
python energy.py -e Dot-TTN_DS18B20_MTS_MDOT_F411RE_2s_RT
  -t 0.300 -s 10 -F 5 -v 5.0 -c 1000
```

Here the sampling frequency is reduced to 5 MHz and the capture duration to 1000 ms.

To measure energy consumption by the particle sensor, we used DN7C3CA006 particle sensor provided by SHARP corporation[SHA14]. The specification of particle sensor proposes that we should connect a 220 µF capacitor between GND and pin-6 of the particle sensor. The data sheet also suggests putting a resistor of 150 Ω
between $V_{cc}$ and the pin-6 [SHA14]. The particle sensor uses a fan to capture dust. We have used an NPN transistor to create a switch to control the fan. The mDot will turn on the fan of the particle sensor during particle sensing and then turn it off. The schematic diagram of particle sensor and the mDot is presented in Fig. 4.19.

![Schematic Diagram](image)

**Figure 4.19:** mDot Interfacing with Particle Sensor Schematic Diagram

A C++ program was written to interface the particle sensor with the mDot. Three GPIO pins of the mDot were configured to read analog voltage corresponding to the particles detected. PB0 of mDot is configured as an analog input. Another pin PA_0 has been configured to turn on and off the particle sensor’s LED. Similarly, to turn ON_OFF the fan, PA_3 of the mDot is configured as digital output. Following is the code snippet to configure the GPIO:

**Source code 4.14 C++ Program: Particle Sensor Configuring GPIO**

```cpp
#define FAN_PIN PA_3
#define \gls{led}_PIN PA_0
#define MEASUREV_PIN PB_0

DigitalOut fanCtrl(FAN_PIN);
DigitalOut ledpower(\gls{led}_PIN);
AnalogIn measure(MEASUREV_PIN);
```
Algo. 4.15 explains the process of sampling sensing data from the sensor.

**Source code 4.15 C++ Program: Sensing Value from Particle Sensor**

```c++
#define SAMPLING_TIME 280
#define DELTA_TIME 40
#define SLEEP_TIME 9680

float getDustVoltageSample(void)
{
    float dustVMeasured = 0.0;
    ledpower = 0; // turn the \gls{led} on

    // Wait samplingTime before reading V0output
    wait_us(SAMPLING_TIME);

    // read the dust value in (0-1.0) <-> (0V-3.3V)
    dustVMeasured = measure.read();
    // Wait deltaTime before shutting off \gls{led}
    wait_us(Delta_TIME);

    ledpower = 1; // turn the \gls{led} off
    wait_us(SLEEP_TIME); // No use in this example
    return dustVMeasured;
}
```

As specified in the datasheet that one should sample data from the particle sensor after every 10 ms [SHA14]. Therefore, we used SLEEP\_TIME as 9680 $\mu$s to complete 10 ms duration before the next sample. Algo.4.16 explains the main function of this sensor application.
Source code 4.16 C++ Program: Main Function of Particle Sensor

```cpp
#define FANFREESAMPLE 50
int main(void) {
    float vS = 0; // stores reference voltage
    float voMeasured = 0; // Stores measured output (0-1023) from sensor
    float calcVoltage = 0; // Calculate actual voltage output from sensor
    float dustDensity = 0;

    // To calculate dust density
    // find reference voltage without fan
    vS = getVsWithOutFAN()/FANFREESAMPLE;
    while(true) {
        generate_event();
        // Turn ON FAN to gather particle
        fanCtrl = 1;
        wait_ms(500);
        voMeasured = getDustVoltageSample();
        // Calculating output voltage from Raw analog signal to mV
        calcVoltage = voMeasured * (3.3) * 1000;

        // Calculating dust density
        dustDensity += (0.6 * (calcVoltage - vS));
        generate_event();
        // Turn off the FAN
        fanCtrl = 0;
        wait(2); // Wait for 2 second
    }
}
```

The fan is turned ON and there is a delay of 500 ms. By increasing this delay we will get a more precise value of dust density. For our experimental purposes, we have fixed it to 500 ms.
Following code snippet has been used, to compute reference voltage $V_s$:

```
float getVsWithOutFAN(void) {
    fanCtrl = 0; // Turn off the fan
    float vsMeasured_sum = 0;
    int fanFreeSampleCtr = 0;
    for (fanFreeSampleCtr = 0; fanFreeSampleCtr < ; fanFreeSampleCtr++)
        vsMeasured_sum += (getDustVoltageSample() * 3300);

    return vsMeasured_sum;
}
```

Python script was used to measure the current consumption during sensing phase. As can be seen in the code snippet 4.15, the function `generate_event()` is being called to capture the current consumption during one sensing phase. The Python script was called with the following input values:

```
Listing 4.5: Run Python Script Temperature Sensor
python energy.py -e Dot--ParticalSensor_MTS_MDOT_F411RE_500ms
-t -0.25 -s 10 -F 5 -v 5.0 -c 600
```

In this scenario, the sampling frequency is set to 5 MHz and the capture duration is 600 ms. Moreover, the threshold voltage is also changed to -0.25 V because the fan is consuming more current.

**Results:** In this section, we will first discuss the results gathered by the temperature sensor and then the particle sensor. The data gathered for the temperature sensor show that the sensor takes around 800 ms to return the temperature value to the mDot. Fig. 4.20 shows the current consumption waveform during temperature sensing.
Each horizontal box in this figure represents 200 ms. The energy measurement analysis of the temperature sensor shows that the sensor takes 763.53 ms to compute the temperature value. The average current consumption by the mDot is $17.26 \pm 5.97 \times 10^{-5}$ mA. The average power consumption is calculated to be $0.0863 \pm 5.97 \times 10^{-5}$ W and the average energy consumed is $0.066 \pm 5.90$ J.

In the case of the particle sensor, our experiments confirmed that the mDot is consuming around 158 mA of current. Fig. 4.21 presents the screenshot from the oscilloscope that explains the waveform of current observed.
The data analysis of the particle sensor shows that the sensor takes 510,01±4,16E-04 ms to send observed dust voltage where 500 ms is the fan running delay. Therefore, if we increase the fan on duration, the sensing phase duration will be increased. The average current consumption by the mDot is 98,59±3,07E-06 mA. The average power consumption is calculated to be 0,493±3,07E-06 W and the average energy consumed is 0,25±0,894 J. Tab. 4.8 shows that the particle sensor consumes more power and energy as compared to the temperature sensor. The temperature sensor consumes around 6 times less power than the particle sensor. This section proves that the power and energy consumption of an application depends on the type of sensor used in the application. Some sensor might consume less power than the others.

### Table 4.8: Sensor Energy Consumption: A Comparison

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Duration (ms)</th>
<th>Current (mA)</th>
<th>Power (W)</th>
<th>Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp</td>
<td>763,53±0,359</td>
<td>17,26±5,97E-05</td>
<td>0,086±5,97E-05</td>
<td>0,066±5,90</td>
</tr>
<tr>
<td>Particle</td>
<td>510,006±4,16E-04</td>
<td>98,59±3,07E-06</td>
<td>0,493±3,07E-06</td>
<td>0,25±0,894</td>
</tr>
<tr>
<td>Difference</td>
<td>-253,524</td>
<td>81,330</td>
<td>0,407</td>
<td>0,184</td>
</tr>
</tbody>
</table>

Figure 4.21: mDot Particle Sensor Current Waveform

The data analysis of the particle sensor shows that the sensor takes 510,01±4,16E-04 ms to send observed dust voltage where 500 ms is the fan running delay. Therefore, if we increase the fan on duration, the sensing phase duration will be increased. The average current consumption by the mDot is 98,59±3,07E-06 mA. The average power consumption is calculated to be 0,493±3,07E-06 W and the average energy consumed is 0,25±0,894 J. Tab. 4.8 shows that the particle sensor consumes more power and energy as compared to the temperature sensor. The temperature sensor consumes around 6 times less power than the particle sensor. This section proves that the power and energy consumption of an application depends on the type of sensor used in the application. Some sensor might consume less power than the others.
5.1 Energy Consumption Estimation Model

To establish an energy estimation model, one need to understand the energy consumption of different processes in a sensing application. We developed a temperature sensing application, that sampled the temperature using DS18b20 temperature sensor [Dal] interfaced with the mDot. The application sampled the temperature twice with a wait of ~0.1 seconds in between the two samples. Fig. 5.1 shows the current consumption by the mDot when the temperature sensing application was running.

![Power consumption of one cycle of a sensing application. Some phases are shortened on the x-axis, to save space. The labels reveal the actual duration [THK17].](image)

**Figure 5.1:** Power consumption of one cycle of a sensing application. Some phases are shortened on the x-axis, to save space. The labels reveal the actual duration [THK17].

The vertical axis represents the power consumption. With the help of experiments done in Chapt. 4, we could identify different phases in the application. Fig. 5.1 show that in the sleep phase, the mDot consumes ~1.2 mW. We have labeled the wake-up phase as post-sleep phase and it consumes ~110 mW and its duration is ~0.15 seconds. Similarly, the sensing phase consumes ~86 mW of power and takes
The pre-send phase consumes \( \sim 75 \) mW lasts for \( \sim 0.012 \) s. The duration of send phase is \( \sim 0.07 \) s and it consumes \( \sim 325 \) mW of power. The post-send phase has a power consumption of \( \sim 200 \) mW and the duration of \( \sim 0.024 \) s. After the post-send phase the mDot goes to the sleep phase and has a \( \sim 0.004 \) s long pre-sleep phase that consumes \( \sim 100 \) mW of power.

The power consumption waveform in the Fig. 5.1 allowed us to define some distinct phases in the sensing application. An abstraction of the actual power consumption waveform is presented in the Fig. 5.2.

\[
E = \sum_{n=1}^{N} P_n * \Delta t_n \tag{5.1}
\]

Where \( P_n \) and \( \Delta t_n \) represent the power consumption and the duration of the phase \( n \), and \( E \) is the total energy in one sensing cycle. \( N \) is the total number of
phases in a sensing application including the static phases.

In our approach, we calculate the values of $P_n$ and $\Delta t_n$ at different times. The power consumption of a phase and the duration of phase belong to two different properties of a sensing application. The duration of an operation is application dependent whereas the power consumption is the function of current drawn by the hardware itself. Since at the design-time of an application the application developer knows the hardware platform and the peripheral devices, therefore, we can estimate the power consumption of different phases at the design-time using the measurement setup specified in the Chapt. 3. However, to incorporate the run-time behavior of the application the duration is calculated by the application using timer functionality. The run-time calculation of the duration is also important because the planning algorithm implemented in the cloud can change the duration and the occurrence of different activity phases.

### 5.2 Run-time Logic for Energy Estimation

The $\Delta t_n$ is the execution time of a phase running in an application. It is computed the application itself at runtime. In the case of mDot hardware platform, an application can provide the duration of an activity phase using `timer.h` library. For other IoT hardware platforms one can find suitable APIs for duration calculation. `timer.h` library is provided by the `mbed-os`. Following APIs of `timer.h` has been used to get the duration[ARM]:

**Listing 5.1:** APIs of timer.h

```c
void start() \ start the timer
void stop() \ stop the timer
float read() \ get time passed in seconds
```

The Algo. 5.1 explains the energy calculation instructions implemented inside the application. It explains how to get timestamps for different activity phases and the implementation of Eq. 5.1.
In the next chapters we will develop and evaluate different types of energy-aware periodic sensing applications.
In this chapter, we will develop some applications for mDot IoT hardware platform where we will increase the occurrence of sensing phase as defined in Fig. 5.2 and observe how our estimation model performs under such kind of applications. We will implement the run-time logic as explained in the Algo. 5.1 in the application. We will use the `timer.h` APIs to compute duration. The power consumption of each phase is extracted from Chapt. 4. In these energy-aware applications, we will implement the energy model defined in Eq. 5.1. To prove the correctness of our energy estimation model we will compare our results with the actual energy consumption. We have used the shunt resistor approach as defined in the Chapt. 3 to measure actual current consumption. We have used the Python script to estimate the energy consumption of a sensing cycle by these applications. The data gathered was an average over 10 samples.

We have categorized these applications based on the sensor used. In one category of applications, we monitor and send the dust samples to a peer node. And in the other category, the applications monitor and send the temperature samples to the peer node. In this chapter, we will increase the number of sensing samples in one sense cycle and see how our estimation model performs.

The pseudo code of the application is as follows:
1. Run the particle or temperature sensor and get particle density or temperature value.
2. Process the sampled sensor data.
3. Repeat step 1 if more sensing samples are required.
4. Transmit the data to a gateway node.
5. Sleep for 2 seconds.
6. Repeat from step 1.
6.1 Particle Sensor

It has been decided to run the same application by varying the number of times we sample the particle sensor data. We will develop five different applications where the sensor data sample frequency will be one, two, five, ten and twenty. Based on the difference of sensor data sampling frequency we have decided to give following names to our applications:

- SenseApp_MTS_MDOT_F411RE_1SS
- SenseApp_MTS_MDOT_F411RE_2SS
- SenseApp_MTS_MDOT_F411RE_5SS
- SenseApp_MTS_MDOT_F411RE_10SS
- SenseApp_MTS_MDOT_F411RE_20SS

6.1.1 Energy Aware Application

In this approach, we built the energy consumption calculation model in our sensing application. We implemented the model presented in Eq. 5.1 in the application. The application transmits its energy consumption in one sense cycle along with the particle density observed by the sensor to a peer node.

To implement the model defined in Eq. 5.1, we have used `timer.h` to get the duration of operations from the application. The power consumption for each operation calculated in Chapter 4 has been hard-coded in the application. In these sample applications, we have enabled the receive operation and therefore, have added a fixed duration for receive phase. Since the receive operation is event based and the event service routine is defined inside the mDot library, therefore, we could not use the `timer.h` APIs. Moreover, the static blocks of pre- and post- phases has not been added separately in these applications. These static blocks have been included as part of the main activity phase. The value of $P_{SENSOR}$ is $\sim493$ mW, $P_{TX}$ has a value of $\sim156.9$ mW, $P_{SLEEP}$ is $\sim19.45$ mW and the value of $P_{RX}$ is $\sim120$ mW.

Moreover, in Chapt. 4, we have observed that the receive duration doesn’t change significantly by changing the number of bytes it receives. Therefore, in these applications we have fixed its duration to $\sim187$ms. Similarly, the sleep duration $SLEEP\_DUR$ is fixed to $\sim2$s. Moreover, the cost of processing has been ignored since the duration of processing phase is less than a $\mu$second.

Algo.6.1 gives you an overview of the energy-aware particle sensor application.
Source code 6.1 C++ Program: Energy Aware Application

```c++
float powerConsumption[3] = {P_SENSOR, P_TX, P_SLEEP, P_RX};
while(true){
    // Sensing Operation
    t_op.start();

    for (i=0; i<SENSING_SAMPLE; i++){
        pDensity = getParticleDensity(vS);
pDensity_sum  += pDensity;
pDensity_sum2 += pDensity * pDensity;
    stdErrorPDensity = ((pDensity_sum2 -
                        ((pDensity_sum * pDensity_sum)/(i+1)))/
                       (i+1));

    stdErrorPDensity = sqrt(stdErrorPDensity);
}  
pDensity = pDensity_sum/SENSING_SAMPLE;
t_op.stop();
duration[OP_1] = t_op.read();
t_op.reset();

// Filling mean particle density in the tx buffer
sprintf (buffer, "%f", pDensity);
for (i=0;buffer[i]!=0;i++)
    tx_data.push_back(buffer[i]);

// Filling standard error in the tx buffer
sprintf (buffer, "%f", stdErrorPDensity);
for (i=0;buffer[i]!=0;i++)
    tx_data.push_back(buffer[i]);

energy = 0.0;
for (int j=0; j<4; j++)
    energy += (duration[j] * powerConsumption[j]);

// Filling energy consumption to the tx buffer
sprintf (buffer, "%f", energy);
for (i=0;buffer[i]!=0;i++)
    tx_data.push_back(buffer[i]);

t_op.start();
send_data(tx_data);
t_op.stop();
duration[OP_2] = t_op.read_ms();
duration[OP_2] = duration[OP_2]/1000;
t_op.reset();

dot->sleep(SLEEP_DUR, mDot::RTC_ALARM, false);
duration[OP_3] = SLEEP_DUR;
// Taking care of rx during sleep
duration[OP_4] = DUR_RX/1000;
}
```
According to the Algo. 6.1, we are sending the energy consumption at the time of transmission. At this instant, the energy consumption of transmission itself is unknown along with the energy consumption of the sleep and receive operations. To compensate for these values, we are adding up the energy consumption of transmitting, receive and sleep operations of the previous cycle to the energy consumption of current cycle's sensing operation. This will allow us to transmit the energy consumption of all operations however, they will overlap between current and previous sensing cycles. Moreover, the first sample sent by the energy-aware application will be erroneous and it must be discarded.

Fig. 6.1 presents the energy profiles of these sensing applications.

**Figure 6.1:** Energy Profiles of Particle Sensing Energy-aware Application with Increasing Sensor Sampling Count

Since in these applications, the duration of one sensing sample is 10s and the power consumption is $\sim 493$ mW therefore, the sensing phase is contributing from (98%–99%) in the energy consumption in such applications.
Tab. 6.1 provides the energy consumption sent by the application itself.

**Table 6.1: Energy Consumption Sent by the Energy-Aware Particle Sensing Application**

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensing Sample Frequency</th>
<th>Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1SS</td>
<td>1</td>
<td>5,005±6.27×10⁻⁵</td>
</tr>
<tr>
<td>SenseApp..._2SS</td>
<td>2</td>
<td>9,933±0.0211</td>
</tr>
<tr>
<td>SenseApp..._5SS</td>
<td>5</td>
<td>24,737±0.0211</td>
</tr>
<tr>
<td>SenseApp..._10SS</td>
<td>10</td>
<td>49,412±0.0211</td>
</tr>
<tr>
<td>SenseApp..._20SS</td>
<td>20</td>
<td>98,762±0.0211</td>
</tr>
</tbody>
</table>

**6.1.2 Oscilloscope Measurement**

The energy-aware application developed in Sect. 8.3 is used for measuring actual energy consumption using the oscilloscope. We used the shunt resistor approach to measure current consumption by the mDot node. We used the Python script defined in Appendix. A to measure energy consumption. To use python script for automatic energy consumption measurement, we added `generate_event()` defined in the Algo. 4.5. This function was added at the start and end of `while(true)` loop, to get the boundaries of measurements. This script captured 10 samples from the oscilloscope and computed an average energy consumption for the specified application. Following is the command used to compute energy consumption when sensing sample count was 20:

**Listing 6.1: Python Script: Energy Aware Sense Application**

```python
python energy.py -e 
    Dot-EnergyAware_SenseApp_MTS_MDOT_F411RE_20SS -t -0.25 
    -s 10 -F 0.001 -v 5.0 -c 202500
```

To measure the current consumption for all other applications, the capture size was reduced accordingly. Tab. 6.2, gives the measured energy consumption using the oscilloscope.

**6.1.3 Conclusion**

The results reported in Tab. 6.1, and 6.2 show that the difference between energy consumption is insignificant for such kind of applications. Tab. 6.3 summarizes the observed differences.
### Table 6.2: Energy Consumption Measurement using Oscilloscope for the Energy-Aware Particle Sensing Application

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensing Sample Frequency</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1SS</td>
<td>1</td>
<td>5.0748±7.979E-04</td>
</tr>
<tr>
<td>SenseApp..._2SS</td>
<td>2</td>
<td>9.9881±5.845E-04</td>
</tr>
<tr>
<td>SenseApp..._5SS</td>
<td>5</td>
<td>24.727±1.689E-03</td>
</tr>
<tr>
<td>SenseApp..._10SS</td>
<td>10</td>
<td>49.305±1.500E-03</td>
</tr>
<tr>
<td>SenseApp..._20SS</td>
<td>20</td>
<td>98.408±3.346E-03</td>
</tr>
</tbody>
</table>

### Table 6.3: Energy Consumption Modelling Particle Sensing Application: A comparison

<table>
<thead>
<tr>
<th>App#</th>
<th>Estimated Energy(J)</th>
<th>Measure Energy(J)</th>
<th>Error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1SS</td>
<td>5,005±6.27012E-05</td>
<td>5.0748±7.979E-04</td>
<td>1.37%</td>
</tr>
<tr>
<td>SenseApp..._2SS</td>
<td>9,933±0.0211</td>
<td>9.9881±5.845E-04</td>
<td>0.55%</td>
</tr>
<tr>
<td>SenseApp..._5SS</td>
<td>24,737±0.0211</td>
<td>24.727±1.689E-03</td>
<td>0.04%</td>
</tr>
<tr>
<td>SenseApp..._10SS</td>
<td>49,412±0.0211</td>
<td>49.305±1.500E-03</td>
<td>0.22%</td>
</tr>
<tr>
<td>SenseApp..._20SS</td>
<td>98,762±0.0211</td>
<td>98.408±3.346E-03</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

Tab. 6.3, indicates that the percentage difference between the actual oscilloscope reading and the energy-aware application is in the range (0.04-1.37)%. One reason for such promising results can be that (98%-99%) of energy consumption in these applications is due to the sensing operation. We got good results because our model for energy consumption estimation for particle sensor is pretty efficient. From these results, we can’t generalize that our energy estimation model works efficiently because the energy profile in the current scenario is skewed towards the sensing phase.

### 6.2 Temperature Sensor

By following the section of particle sensing application we have developed five different applications for the temperature sensor. In these applications, we will vary the frequency of sensor sampling to be one, two, five, ten and twenty. Following names were given to such applications:

- SenseApp_MTS_MDOT_F411RE_1TS
- SenseApp_MTS_MDOT_F411RE_2TS
- SenseApp_MTS_MDOT_F411RE_5TS
- SenseApp_MTS_MDOT_F411RE_10TS
- SenseApp_MTS_MDOT_F411RE_20TS
6.2.1 Energy Aware Application

The modification in the application is same as explained in the code snippet. 6.1, however, to activate temperature sensor the for loop has been modified:

```
Source code 6.2 C++ Program: Temperature Sensing Modification in Energy Aware Application

for (i=0;i<SENSING_SAMPLE;i++)
{
  //Start temperature conversion, wait until ready
  probe.convertTemperature(true, DS1820::all_devices);
  temperature = probe.temperature();

  temperature_sum += temperature;
  temperature_sum2 += temperature * temperature;
  stdErrorTemperature = (temperature_sum2 -
                          ((temperature_sum * temperature_sum)/(i+1)))/(i+1);

  stdErrorTemperature = sqrt(stdErrorTemperature);
}
```

Moreover, in these applications, we have removed the receive phase from the application. Same as particle sensor, the static blocks of pre- and post- phases has not been added separately but they have been included as part of sleep and transmit phase.

The value of $P_{SENSOR}$ is $\sim$141.5 mW, $P_{TX}$ has a value of $\sim$156.9 mW, and $P_{SLEEP}$ is $\sim$19.45 mW. Moreover, the cost of processing has been ignored since the duration of processing phase is less than a $\mu$second.

The power consumption for temperature is $\sim$86.3 mW as measured and reported in tab. 4.8 in the Chapt. 4. However, we have used a different value in the current example. The reason for this is that in those measurements we computed the temperature sensor’s power consumption when no radio module was enabled. When we enable the radio module for transmission the power consumption of mDot increases and therefore, the power consumption of temperature sensor increased. To incorporate this change we again measured the power consumption by the temperature sensor using Python script and the setup defined in Chapt. 3. Tab. 6.4 shows the measured values:

The energy profiles of these applications are shown in Fig. 6.2. Since the temper-
Table 6.4: Temperature Sensor Energy and Power Consumption with Transmission

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp</td>
<td>0.763±2.34E-06</td>
<td>0.028±3.75E-05</td>
<td>0.141±3.75E-05</td>
<td>0.108±7.22E-08</td>
</tr>
</tbody>
</table>

The temperature sensor doesn’t consume as much power as the particle sensor, therefore, the energy distribution is not overly shadowed by the sensing phase. However, when we increase the sensing samples within a sense cycle we observe that the sensing phase starts to dominate the energy contribution.

Figure 6.2: Energy Profiles of Temperature Sensing Energy-aware Application with Increasing Sensor Sampling Count
Tab. 6.5 provides the energy consumption sent by the energy-aware temperature sensing application.

**Table 6.5: Energy Consumption Sent by the Energy-Aware Temperature Sensing Application**

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensing Sample Frequency</th>
<th>Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1TS</td>
<td>1</td>
<td>0.154±2.77E-17</td>
</tr>
<tr>
<td>SenseApp..._2TS</td>
<td>2</td>
<td>0.262±5.55E-17</td>
</tr>
<tr>
<td>SenseApp..._5TS</td>
<td>5</td>
<td>0.587±4.24E-04</td>
</tr>
<tr>
<td>SenseApp..._10TS</td>
<td>10</td>
<td>1.127±4.71E-04</td>
</tr>
<tr>
<td>SenseApp..._20TS</td>
<td>20</td>
<td>2.208±0</td>
</tr>
</tbody>
</table>

### 6.2.2 Oscilloscope Measurement

To measure the actual energy consumption we used the shunt resistor approach for measuring current consumption by the mDot node as defined in the Chapt. 3. We used the Python script defined in Appendix. A to measure energy consumption as explained in the Chapt. 3.

Following is the command used to compute energy consumption using Python script when sensing sample count was 20:

**Listing 6.2: Python Script: Energy Aware Sense Application**

```bash
python energy.py -e
    Dot-EnergyAware_TempSenseApp_MTS_MDOT_F411RE_1TS -t
    -0.25 -s 10 -F 1 -v 5.0 -c 3100
```

For all other applications, the capture size was reduced proportionally. Tab. 6.6, gives the measured energy consumption using the oscilloscope.

### 6.2.3 Conclusion

The results reported in the Tab. 6.5, and 6.6 show that the difference between energy consumption is insignificant. Tab. 6.7 summarizes the observed differences.

Tab. 6.7, indicates that the percentage difference between the actual oscilloscope reading and the energy-aware application is (0.16-1.40)%. The energy profile of temperature sensing application; *SenseApp MTS_MDOT_F411RE_1TS*, is a
Table 6.6: Energy Consumption Measurement using Oscilloscope for the Energy-Aware Temperature Sensing Application

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensing Sample Frequency</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1TS</td>
<td>1</td>
<td>0.155±6.10E-04</td>
</tr>
<tr>
<td>SenseApp..._2TS</td>
<td>2</td>
<td>0.259±4.96E-04</td>
</tr>
<tr>
<td>SenseApp..._5TS</td>
<td>5</td>
<td>0.588±1.10E-03</td>
</tr>
<tr>
<td>SenseApp..._10TS</td>
<td>10</td>
<td>1.111±1.12E-03</td>
</tr>
<tr>
<td>SenseApp..._20TS</td>
<td>20</td>
<td>2.197±2.09E-03</td>
</tr>
</tbody>
</table>

Table 6.7: Energy Consumption Modeling Temperature Sensing Application: A comparison

<table>
<thead>
<tr>
<th>App#</th>
<th>Estimated Energy(J)</th>
<th>Measure Energy(J)</th>
<th>Error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SenseApp..._1TS</td>
<td>0.154±2.77E-17</td>
<td>0.155±6.10E-04</td>
<td>0.24%</td>
</tr>
<tr>
<td>SenseApp..._2TS</td>
<td>0.262±5.55E-17</td>
<td>0.259±4.96E-04</td>
<td>1.49%</td>
</tr>
<tr>
<td>SenseApp..._5TS</td>
<td>0.587±4.24E-04</td>
<td>0.588±1.10E-03</td>
<td>0.16%</td>
</tr>
<tr>
<td>SenseApp..._10TS</td>
<td>1.127±4.71E-04</td>
<td>1.111±1.12E-03</td>
<td>1.40%</td>
</tr>
<tr>
<td>SenseApp..._20TS</td>
<td>2.208±0</td>
<td>2.197±2.09E-03</td>
<td>0.47%</td>
</tr>
</tbody>
</table>

bit balanced and we observed an error in estimation at around 0.24%. Similarly, the contribution of the sensing phase in the energy estimation of the applications SenseApp_MTS_MDOT_F411RE_2TS and SenseApp_MTS_MDOT_F411RE_5TS is 78% and 90% respectively but the error in estimation is still insignificant. These experimental results gave us more confidence in our energy consumption estimation approach. Since this approach is performing efficiently when other phases are also contributing in the energy estimation.

In next chapter, we will further design new particle and temperature sensing applications based on different sensing modes that can be set by the planning algorithm implemented in the cloud and observe how our model performs under such conditions.
As described in Chapt. 1, the energy-aware planning can be performed at the cloud for efficient use of sensor nodes. By considering this functionality, we have defined three sensing modes and developed energy-aware applications for these sensing modes. Following describes these three modes:

- Extravagant: The sensing node has an excessive amount of energy. The sensing node can send more data within a given time period.
- Moderate: The energy is available at a moderate level. Send sensor data at a normal pace.
- Thrifty: Energy resource is scarce. The sensing node should send sensor data once in a while.

These different sensing modes have been characterized based on the available energy at the sensing node. In order to operate in these different sensing modes, we have changed the duty cycle and the frequency of sampling sensor data. The cloud can choose any one of these modes based on the energy consumption data it receives. In this chapter, we will develop energy-aware applications based on the above-mentioned sensing modes. We will use temperature and particle sensors in these applications. We will observe their energy profiles and gather the estimated energy consumption. In the end, we will compare the estimated energy with the actual oscilloscope measurements.

### 7.1 Particle Sensor

In the scenario of Particle Sensing, we have only modified the duty cycle (sleep duration) to produce different sensing modes. Following table summarizes the
selection of different parameters in different sensing modes:

Table 7.1: Sensor Modes Parameters in Particle Sensing Energy-Aware Applications

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Duty Cycle(s)</th>
<th>Sensing Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Moderate</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>Thrifty</td>
<td>300</td>
<td>1</td>
</tr>
</tbody>
</table>

All the values presented in the Tab. 8.2 are with respect to one sensing cycle.

7.1.1 Energy-Aware Application

The technique of developing energy-aware applications is same as explained in the Chapt. 6, however in the current applications we have added the pre- and post-static blocks as separate blocks of energy. In the Chapt. 6, we averaged them over the main phase. There is also another addition in these applications that we are sending the energy consumption of each phase along with the accumulated energy. This has increased the transmission packet size along with the transmission phase energy consumption. In these example applications, we have removed the receive phase and have added an active phase. However, the contribution of active phase is again ignorable. Algo. 7.1 presents the modified code snippet of the particle sensing energy-aware applications.
Source code 7.1 C++ Program: Energy Aware Application with Static Code Blocks

```c
float powerConsumption[4] = {P_SENSOR, P_ACTIVE, P_TX, P_SLEEP};
float staticEnergy[4] = {0, 0, PRE_TX_ENERGY + POST_TX_ENERGY,
                        PRE_SLEEP_ENERGY + POST_SLEEP_ENERGY};

while(true){
    t_op.start();
    for (i=0; i<SENSING_SAMPLE; i++){
        pDensity = getParticleDensity(vS);
        ........ get standard error and mean particle density ........
    }
    pDensity = pDensity_sum/SENSING_SAMPLE;
    t_op.stop();
    duration[OP_1] = t_op.read();
    t_op.reset();
    t.start();
    sprintf (buffer, "%f", pDensity);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);
    sprintf (buffer, "%f", stdErrorPDensity);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);
    energy = 0.0;
    for (int i=0; i<4; i++){
        energy += ((duration[i] * powerConsumption[i]) +
                    staticEnergy[i]);
        sprintf (buffer, "OP%d:%f;",i+1 , ((duration[i]
                        * powerConsumption[i]) + staticEnergy[i]));
        for (int ind=0;buffer[ind]!=0;ind++)
            tx_data.push_back(buffer[ind]);
    }
    sprintf (buffer, "%f", energy);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);
    t.stop();
    duration[OP_2] = t_op.read_us();
    duration[OP_2] = duration[OP_2]/1000000;
    t_op.start();
    send_data(tx_data);
    t_op.stop();
    duration[OP_3] = t_op.read_ms();
    duration[OP_3] = duration[OP_3]/1000 - (PRE_TX_DUR + POST_TX_DUR);
    t_op.reset();
    dot->sleep(SLEEP_DUR, mDot::RTC_ALARM, false);
    duration[OP_4] = SLEEP_DUR - PRE_SLEEP_DUR;
}
```
The $P_{\text{SENSOR}}$ and $P_{\text{ACTIVE}}$ are $\sim 493$ mW and $\sim 153.6$ mW respectively, as calculated in the Chapt. 4. The transmission phase consumes $\sim 324.74$ mW as defined in the Chapt. 4 without averaging it with pre- and post- phases. The $P_{\text{SLEEP}}$ measured without the pre- and post- phases is $\sim 1.2$ mW. However, when the Particle Sensor is attached to the mDot the power consumption is observed to be increased. We measured the power and energy consumption during Sleep phase to incorporate the impact of Particle Sensor on the Sleep phase. We measured the power consumption during Sleep phase using Python script and the measurement setup described in the Chapt. 3. Following table summarizes the data gathered:

<table>
<thead>
<tr>
<th>State</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleep</td>
<td>$60.163\pm1.74E-03$</td>
<td>$9.15E-03\pm3.25E-05$</td>
<td>$0.046\pm3.25E-05$</td>
<td>$2.752\pm1.97E-03$</td>
</tr>
</tbody>
</table>

In the these sensing applications, we have set the sleep phase to $\sim 45.7$ mW. The $\text{PRE\_TX\_ENERGY}$, $\text{POST\_TX\_ENERGY}$, $\text{PRE\_SLEEP\_ENERGY}$, $\text{POST\_SLEEP\_ENERGY}$, $\text{PRE\_TX\_DUR}$, $\text{POST\_TX\_DUR}$, $\text{PRE\_SLEEP\_DUR}$, and $\text{POST\_SLEEP\_DUR}$ are set as explained in the Chapt. 4. While calculating the duration of sleep phase, we didn’t subtract the post-sleep duration from the sleep duration because it is not added to the main sleep phase. The complete source code for these applications can be found in Appendix. C. To change among different sensing modes, one needs to modify $\text{SLEEP\_DUR}$ and $\text{SENSING\_SAMPLE}$ in the source code defined in Appendix. C.

Fig. 7.1 show the energy distribution of different phases under different sensing modes. The data presented in the Fig. 7.1 is estimated over one sensing cycle.
7.1. PARTICLE SENSOR

7.1.2 Conclusion

In this sub-section, we will evaluate our energy estimation model for the particle sensing applications. Tab. 7.3 provides a comparison between the energy estimation by the application and the measured energy consumption using setup described in the Chapt. 4.

Table 7.3: Particle Sensing: Energy Consumption Comparison under Different Sensing Modes

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Estimated Energy(J)</th>
<th>Measured Energy(J)</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>5,420±1,33E-03</td>
<td>5,450±1,57E-03</td>
<td>0,582%</td>
</tr>
<tr>
<td>Moderate</td>
<td>7,727±8,93E-04</td>
<td>7,672±3,78E-03</td>
<td>0,706%</td>
</tr>
<tr>
<td>Thrifty</td>
<td>18,695±1,49E-04</td>
<td>18,681±0,0104</td>
<td>0,071%</td>
</tr>
</tbody>
</table>
The percentage difference for these applications is between 0.071% and 0.706%. This confirms that our estimation model suits well for applications with such a varied energy profiles as shown in Fig. 7.1. It is worth noting that the energy profiles of these applications consist mainly of sensing and sleeping phases. Although we achieved very good estimation results these results might vary if other activity phases will contribute.

Tab. 7.3 shows that the energy consumption in the Thrifty mode is greater than other modes. This is because the duration of one cycle of the Thrifty mode is 300 s whereas in the Extravagant mode the duration is 20 s. If we compare the energy consumption of all the three modes for 10 minutes it will be evident that the energy consumption during the thrifty mode will be much less than the other two modes. Following table presents the energy consumption estimation for 10 minutes:

**Table 7.4:** Particle Sensing: Energy Consumption Estimation for 10 Minutes under Different Sensing Modes

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>(\text{Duration}_{\text{one cycle}}) (s)</th>
<th># of Cycles per 600 s</th>
<th>Estimated Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>20</td>
<td>30</td>
<td>165,15</td>
</tr>
<tr>
<td>Moderate</td>
<td>70</td>
<td>9</td>
<td>73,404</td>
</tr>
<tr>
<td>Thrifty</td>
<td>310</td>
<td>2</td>
<td>41,752</td>
</tr>
</tbody>
</table>

**7.2 Temperature Sensor**

In the case of Temperature Sensing applications, we have modified both the duty cycle (sleep duration) and the number of sensing samples to produce different sensing modes. Following table summarizes the selection of different parameters in these sensing modes:

**Table 7.5:** Sensor Modes Parameters in Temperature Sensing Energy-Aware Applications

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Duty Cycle(s)</th>
<th>Sensing Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Moderate</td>
<td>60</td>
<td>5</td>
</tr>
<tr>
<td>Thrifty</td>
<td>160</td>
<td>1</td>
</tr>
</tbody>
</table>

All the values presented in the Tab. 7.5 are with respect to one sensing cycle.
7.2.1 Energy-Aware Application

These set of applications uses the same algorithm for energy estimation as specified in Algo. 7.1. The only difference is the values of $P\_\text{SENSOR}$ and $P\_\text{SLEEP}$. The $P\_\text{SENSOR}$ is $\sim 141.5$ mW and $P\_\text{SLEEP}$ is $\sim 1.2$ mW as measured in the Chapt. 4. Similar to the particle sensing applications we have added static blocks of energy as can be seen in Fig. 5.2. There is no receive phase in these applications. The packet size has also been increased similar to the above-mentioned particle sensing applications. We developed three different applications associated with each sensing mode and gathered their energy estimations. The complete source code for these applications can be found in Appendix. D. To change among different sensing modes, one needs to modify $\text{SLEEP\_DUR}$ and $\text{SENSING\_SAMPLE}$ in the source code defined in Appendix. D. Fig. 7.2 show the energy distribution of different phases under different sensing modes. The data presented in the Fig. 7.1 is estimated over one sensing cycle.

![Energy Profiles of Temperature Sensing Energy-aware Application with Different Sensing Modes](image)

**Figure 7.2:** Energy Profiles of Temperature Sensing Energy-aware Application with Different Sensing Modes
7.2.2 Conclusion

In this sub-section, we will evaluate our energy estimation model. Tab. 7.6 provides a comparison between the energy estimation by the application and the measured energy consumption using setup described in the Chapt. 4.

**Table 7.6:** Temperature Sensing: Energy Consumption Comparison under Different Sensing Modes

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Estimated Energy (J)</th>
<th>Measured Energy (J)</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>0.158±1.59E-04</td>
<td>0.157±1.32E-04</td>
<td>0.435%</td>
</tr>
<tr>
<td>Moderate</td>
<td>0.658±1.62E-04</td>
<td>0.666±7.64E-03</td>
<td>1.233%</td>
</tr>
<tr>
<td>Thrifty</td>
<td>0.349±1.49E-04</td>
<td>0.347±8.68E-03</td>
<td>0.588%</td>
</tr>
</tbody>
</table>

The results obtained in these applications indicate that our estimation model suites well for such kind of applications. Although the energy profiles of these applications were quite diverse our model fitted well. It is worth mentioning that in the temperature sensing applications, we observe that the transmission phase is also contributing along with the sensing and sleeping phases. And our estimation model still produced an insignificant error in estimation. These experiments increased our confidence on our energy estimation model.

Similarly, to prove that Thrifty sensing mode consumes much less energy over time we did similar kind of comparison as done in the previous section of particle sensing. Following table presents the energy consumption estimation for 10 minutes:

**Table 7.7:** Temperature Sensing: Energy Consumption Estimation for 10 Minutes under Different Sensing Modes

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>$Duration_{one_cycle}(s)$</th>
<th># of Cycles per 600 s</th>
<th>Estimated Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>2.022</td>
<td>300</td>
<td>47.4</td>
</tr>
<tr>
<td>Moderate</td>
<td>64.48</td>
<td>10</td>
<td>7.18</td>
</tr>
<tr>
<td>Thrifty</td>
<td>160.99</td>
<td>4</td>
<td>1.396</td>
</tr>
</tbody>
</table>
In this chapter, we will develop a sensing application for waspmote \(^1\) IoT hardware platform and compared how our estimation model performs under such type of applications. In these applications, we have used CO2 sensor to sense CO2 concentration in the air.

The pseudo code of the application is as follows:
1. Run the CO2 sensor and get the CO2 concentration.
2. Process the sampled sensor data.
3. Repeat step 1 if more sensing samples are required.
4. Transmit the data to a gateway node.
5. Sleep for 20 seconds.
6. Repeat from step 1.

We implemented the energy calculation model defined in Eq. 5.1 in these applications. To prove the accuracy of our energy estimation model using waspmote IoT hardware platform, we will compare the results from our energy estimation model to the actual energy consumption. We have used the shunt resistor approach as defined in the Chapt. 3 to measure actual energy consumption. We have used the Python script to estimate the energy consumption of a sensing cycle by the application. The data gathered is an average over 10 energy measurement samples.

8.1 Application Logic

We have used waspmote-pro-ide-v06.02 to write energy-aware sensing applications [was]. In this section, we will describe the source code of the CO2 sensing application. The complete source code can be found in the Appendix. E.

\(^1\)http://www.libelium.com/products/waspmote/
Following is the code snippet to get CO2 concentration:

**Source code 8.1 wasp mote CO2 Sensing Phase**

```c
Gas co2(SOCKET_A);

co2.ON();
PWR.deepSleep("00:00:00:10", RTC_OFFSET, RTC_ALM1_MODE1, ALL_ON);
co2Concentration = 0;
co2Concentration = co2.getConc();
co2.OFF();
```

Since the CO2 sensor for wasp mote requires some time to heat up before measuring the CO2 concentration. Therefore, for our experimental work, we have selected it to be 10s. One can increase this value to get better CO2 concentration samples. The first instruction declares the instance of CO2 sensor that is attached to the `SOCKET_A` of the wasp mote. The second instruction initializes the CO2 sensor. Then the wasp mote goes to the *deep sleep* state for 10 seconds while the sensors remain on because we have set the last argument of this function to `ALL_ON`. The function `co2.getConc()` returns the CO2 concentration and at the end, we turn off the CO2 sensor.
8.1. APPLICATION LOGIC

Algo. 8.2 presents the code for transmission.

**Source code 8.2 waspmote Transmission Phase**

```c
configureLoRaWAN();

frame.createFrame(BINARY);
frame.addSensor(SENSOR_GP_CO2, (double)co2Concentration);

errorLW = LoRaWAN.ON(SOCKET);
frequencyConfiguration();
errorLW = LoRaWAN.joinABP();

if(errorLW == 0)
{
    //Send unconfirmed packet
    errorLW = LoRaWAN.sendUnconfirmed(PORT, frame.buffer, frame.length);

    // Error messages:
    /*
    * '6' : Module hasn't joined a network
    * '5' : Sending error
    * '4' : Error with data length
    * '2' : Module didn't response
    * '1' : Module communication error
    */
    // Check status
    if( errorLW == 0 ) {
        USB.println(F("3. Send Unconfirmed packet OK"));
    } else {
        USB.print(F("3. Send Unconfirmed packet error = ");
        USB.println(errorLW, DEC);
    }
} else{
    USB.print(F("2. Join network error = ");
    USB.println(errorLW, DEC);
}
errorLW = LoRaWAN.OFF(SOCKET);
```
In the Algo. 8.2, the first instruction configures the LoRaWAN communication parameters. These parameters include device EUI, device address, network session key, application session key, application key, adaptive data rate and retransmission count for uplink confirmed packets. The details of these parameters can be found in the LoRaWAN manual; as given in the reference [SLE+15]. The next two instructions create a transmission frame and insert the value of CO2 concentration in the frame. Then the LoRAWAN is enabled and the frequency of operation is configured. The next instruction enables the waspmote to join in the Activation By Personalization (ABP) mode. In this mode, the device address and the security keys are hard-coded in the device and the device is activated using these hard-coded values. Finally, the LoRaWAN packet is sent in unconfirmed mode. After that, we have some instructions for debugging purposes and at the end, the LoRaWAN is turned off.

The third sleeping phase is quite simple. It is only one instruction where we shift the operating mode to the deep sleep and turn off all the peripheral devices. Following is the code snippet, where the waspmote goes to deep sleep for 20 seconds:

Source code 8.3 waspmote Sleeping Phase

```
PWR.deepSleep("00:00:00:20", RTC_OFFSET, RTC_ALM1_MODE1, ALL_ON);
```

8.2 Design-time and Run-time Energy Estimation Logic

Since it was a new hardware platform, therefore, we performed a design-time energy measurement process for each activity phase as defined in Fig. 5.2. For such type of applications, we only observed three activity phases; sense, transmit and sleep. We performed automatic power and energy measurement using the measurement setup defined in Chapt. 3. We configured a waspmote to signal the oscilloscope to start sampling the data from the measurement setup. We generated a 100 ms pulse at the start and end of each phase and computed the power and energy consumption during these phases. Following is code snippet is used to configure GPIO and generate pulse:
8.2. DESIGN-TIME AND RUN-TIME ENERGY ESTIMATION LOGIC

Source code 8.4 waspmote Generate Event

```c
pinMode(DIGITAL3, OUTPUT);
digitalWrite(DIGITAL3, 0);

void generate_event(){
    // setting event pin for oscilloscope measurements
    digitalWrite(DIGITAL3, 1);
    delay(100);
    digitalWrite(DIGITAL3, 0);
}
```

We have configured DIGITAL3 in waspmote as the event generator for the oscilloscope. The function `generate_event()` is called before the start and the end of each activity phase as defined in Algo. 8.1, 8.2 and 8.3. The following table summarizes the measure power and energy consumption is for each activity phase:

**Table 8.1: Power and Energy Consumption of CO2 Sensing Application Running on WASPMOTE**

<table>
<thead>
<tr>
<th>Activity Phase</th>
<th>Duration(s)</th>
<th>Current(A)</th>
<th>Power(W)</th>
<th>Energy(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense</td>
<td>11,07±7,48E-04</td>
<td>0,071±2,97E-05</td>
<td>0,357±2,97E-05</td>
<td>3,95±3,22E-04</td>
</tr>
<tr>
<td>Transmit</td>
<td>8,02±1,55E-03</td>
<td>0,057±5,61E-05</td>
<td>0,286±5,61E-05</td>
<td>2,30±4,58E-04</td>
</tr>
<tr>
<td>Sleep</td>
<td>19,71±3,24E-03</td>
<td>0,038±4,03E-05</td>
<td>0,189±4,03E-05</td>
<td>3,72±8,04E-04</td>
</tr>
</tbody>
</table>

For the run-time estimation, we implemented the run-time logic as explained in the Algo. 5.1 in the application. We used the Real-time Counter (RTC) APIs defined in Waspmote Pro API-v028 to compute the duration of each activity phase [lib]. For duration calculation, we used a `getEpochTime` function that returns the number of seconds passed since Thursday, 1 January 1970.
Following code snippet explains the use of $RTC$ timer to compute the duration of each phase: The Algo. 8.5, shows that one activates the RTC and measure the current

**Source code 8.5 waspmote RTC to Measure Duration of each Phase**

```
RTC.ON();
timestamp = RTC.getEpochTime();
// Write the code to enter an activity phase
// Write the code to exit an activity phase
Duration[ACTIVITY_PHASE] = RTC.getEpochTime() - timestamp;
RTC.off();
```

Epoch time. Then one writes the code to perform operations inside an activity phase i.e. sensing, sleeping, transmission, processing. After the activity phase ends one again gets the time stamp and computes the duration in seconds in the activity phase. The duration of each phase is stored in a globally defined variable $Duration[]$.

### 8.3 Energy Aware Application

We developed three CO2 periodic sensing applications based on different sensing modes: extravagant, moderate, and thrifty. These three applications have different duty cycles and CO2 sensing samples in an activity cycle. Following table summarizes the selection of different parameters in different sensing modes:

**Table 8.2: Sensor Modes Parameters in CO2 Sensing Energy-Aware Applications**

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Duty Cycle(s)</th>
<th>Sensing Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Moderate</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>Thrifty</td>
<td>120</td>
<td>1</td>
</tr>
</tbody>
</table>

We modified the $SLEEP\_DUR$ and $SENSING\_SAMPLE$ in the source defined in Appendix E, to generate these three applications.

Fig. 8.1, shows the oscilloscope dump of a sensing cycle of a CO2 energy-aware sensing application under *Moderate* sensing mode.

In these applications, we have categorized only three activity phases; sensing, transmission, and sleeping. The value of $P\_SENSOR$ is $\sim$362.65 mW, $P\_TX$ has a value of $\sim$293.05 mW, and the value of $P\_SLEEP$ is $\sim$193.6 mW.
The following figure provides the energy consumption distribution among the activity phases by the three CO2 sensing applications:

**Figure 8.1:** Current Consumption of a CO2 Sensing Application using Moderate Sensing Mode

**Figure 8.2:** CO2 Sensing: Energy Consumption Comparison under Different Sensing Modes
Tab. 8.3 provides a comparison between the energy estimation by the application and the measure energy consumption using setup described in Chapt. 4.

**Table 8.3:** CO2 Sensing: Energy Consumption Comparison under Different Sensing Modes

<table>
<thead>
<tr>
<th>Sensing Mode</th>
<th>Estimated Energy (J)</th>
<th>Measured Energy (J)</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extravagant</td>
<td>9,253±0</td>
<td>9,12±5,37E-04</td>
<td>1.47%</td>
</tr>
<tr>
<td>Moderate</td>
<td>10,288±0</td>
<td>9,995±7,76E-03</td>
<td>2.93%</td>
</tr>
<tr>
<td>Thrifty</td>
<td>29,088±0</td>
<td>29,761±4,34E-03</td>
<td>2.261%</td>
</tr>
</tbody>
</table>

These small differences presented in Tab. 8.3 in the energy estimation proves that our energy estimation model provides a good estimation even we have changed the hardware platform of sensing node. This shows that our energy consumption estimation model is generic and independent of the underlying hardware platform. Moreover, these results also prove that our energy estimation model fits well when under various energy distributions of activity phases. Moreover, these results also prove that our energy estimation model fits well when under various energy distributions of activity phases.
Chapter 9

Discussion and Conclusion

The results from the Chapt. 6, 7 and 8 explain that the developed energy consumption estimation model is suitable for various periodic sensing applications. The percentage differences between the estimated energy consumption and the measured values for all different sensing applications lie between 0.04% and 2.928%. That indicates that the presented approach performs well as compared to other approaches presented in the Chapt. 2.

The following figure presents the percentage error in the energy estimation under different sensing modes running on different sensing nodes:

**Figure 9.1:** Percentage Error in Estimated Energy under Different Sensing Modes Running on Different Sensing Devices
Fig. 9.1 shows that the percentage error in the energy estimation of Waspmote is greater than the other two in all the three sensing modes. The reason for this behavior is that in the Waspmote, we didn’t add pre- and post- activity phases separately. In the case of mDot, we ran various experiments as discussed in Chapt. 4 and concluded that the transmission and sleep operations consist of three activity phases, i.e. pre- post- and main phases. Since we didn’t run such experiments on Waspmote; therefore, we could not identify these static activity phases. The activity phase identification approach in Waspmote was simple and this resulted in higher estimation errors compared to its counterpart.

In the problem description, we proposed to establish a generic energy estimation model that will give precise energy estimations for a periodic sensing application using different communication protocols and IoT hardware platforms. Initially, we suggested that to generalize our energy estimation model we will use parametric approach. However, during our work, we found that there are two important parameters for energy estimation; one is the power consumption and the other is the duration. Similarly, we also figured out that the energy profile of an application consists of individual activity phases. Therefore, we withdrew the parametric approach and used the design-time and run-time based approach to compute the power and duration of each activity phase.

Our proposed model is generic and can be used for a variety of IoT hardware platforms that can be powered up by external power supply. This approach is applicable to all the IoT periodic sensing applications that have the APIs to compute the duration of operations. The proposed technique takes care of the influences on power consumption when different operations such as transmission, sensing, and processing co-exist in an application. It also incorporates the dynamic nature of the sensing application by including the run-time estimation phase.

The presented approach is computationally less expensive to implement inside an application and sends the energy estimation for each sensing cycle along with the meta-data to the backend cloud. The size of estimated energy is only 8 bytes that slightly adds to the energy consumption of the transmission which is negligible and reported to the backend in the next transmission cycle. The design-time power calculation saves the energy consumed by the co-processor or external hardware integrated to the IoT device to compute power consumption.
References


Source code A.1 Python Script: Defining package imports

```python
import math
import time
import inspect
import numpy as np
from picoscope import ps6000

from matplotlib.mlab import find
import pylab as pl
import xlwt
import argparse
```
Source code A.2 Defining Energy Measurement Class

class energyMeasure():

    def __init__(self):
        self.ps = ps6000.PS6000(connect=False)
        self.mCurr = []
        self.mDur = []
        self.sdCurr = []
        self.numberOfSamples=0
        self.captureLength = CLENGTH * 1E-3
        self.threshold = THRESHOLD
        self.samplingfreq = SAMPLINGFREQ
        self.capturesampleNo = self.captureLength * (self.samplingfreq * 1E6)

    def openScope(self):
        self.ps.open()

        self.ps.setChannel("A", coupling="DC", VRange=0.100,
                        probeAttenuation=1, BWLimited = 2)
        self.ps.setChannel("B", coupling="DC", VRange=1.0,
                        probeAttenuation=1, BWLimited = 2)
        self.ps.setChannel("C", enabled=False)
        self.ps.setChannel("D", enabled=False)
        res = self.ps.setSamplingFrequency(self.samplingfreq * 1E6,
                                            int(self.capturesampleNo))

        self.sampleRate = res[0]
        print("Sampling @ %f MHz, %d samples"%(res[0]/1E6, res[1]))

        #Use external trigger to mark when we sample
        self.ps.setSimpleTrigger(trigSrc="B", threshold_V=self.threshold,
                                  direction="Falling",
                                  timeout_ms=0)

    def closeScope(self):
        self.ps.close()

    def armMeasure(self):
        self.ps.runBlock()
def computeMeanAndStdevSubChannel(self, subChannelA, to):
    self.mCurr.append(np.mean(subChannelA))
    self.sdCurr.append(np.std(subChannelA))
    fs = self.sampleRate / 1000
    self.mDur.append(to / fs)
    print ("Number of samples: ", fs)
    print ("Duration: ", to / fs)
    print("MeanCurrent:", np.mean(subChannelA))
    print("StandardDeviation:", np.std(subChannelA))

def measure(self, filename):
    print("Waiting for trigger")
    while(self.ps.isReady() == False): time.sleep(0.01)
    print("Sampling Done")
    print("captureSampleNo: ", int(self.capturesampleNo))
    dataA = self.ps.getDataV("A", int(self.capturesampleNo))
    dataB = self.ps.getDataV("B", int(self.capturesampleNo))

    indices = []
    for index in range(len(dataB)):
        if dataB[index] >= THRESHOLD:
            break;

    print("Index: ", index)
    if self.numberOfSamples == 0:
        fig = pl.figure()
        pl.plot(dataB)
        pl.savefig("fig\dataB"+str(self.numberOfSamples)+filename+".png")
        fig.clf()
        pl.close()

        fig = pl.figure()
        pl.plot(dataA)
        pl.savefig("fig\dataA"+str(self.numberOfSamples)+filename+".png")
        fig.clf()
        pl.close()

    if (index+1) < len(dataB):
        self.numberOfSamples = self.numberOfSamples + 1
        self.computeMeanAndStdevSubChannel(dataA[1:index], index)
def output(self, filename, x, y, z):
    book = xlwt.Workbook()
    sh = book.add_sheet("Sheet 1")
    style = xlwt.XFStyle()
    # font
    font = xlwt.Font()
    font.bold = True
    style.font = font
    variables = [x, y, z, (self.sampleRate/1E6), self.captureLength * 1E3, self.capturesampleNo]
    desc = ['TextName', 'Voltage(V)', "Thresh_Vol", "Sampling Freq",
            "Capture Length(ms)", "Capture Samples/calc"]
    for n, (v_desc, v) in enumerate(zip(desc, variables)):
        sh.write(n, 0, v_desc, style=style)
        sh.write(n, 1, v)
        n += 2
    sh.write(n, 0, 'SampleDuration(ms)', style=style)
    sh.write(n, 1, 'Duration(ms)', style=style)
    sh.write(n, 2, 'MeanCurrent(A)', style=style)
    sh.write(n, 3, 'StandardError', style=style)
    index = 1
    for m, e1 in enumerate(self.mDur, n+1):
        sh.write(m, 0, 'Sample ' + str(index))
        sh.write(m, 1, e1)
        index += 1
    for m, e2 in enumerate(self.mCurr, n+1):
        sh.write(m, 2, e2)
    for m, e3 in enumerate(self.sdCurr, n+1):
        sh.write(m, 3, e3)
    m += 2
    sh.write(m, 0, "Average", style=style)
    sh.write(m, 1, np.mean(self.mDur))
    sh.write(m, 2, np.mean(self.mCurr))
    sh.write(m, 3, np.mean(self.sdCurr))
    power = np.mean(self.mCurr) * y
    m += 3
    sh.write(m, 0, "Power(Watt)", style=style)
    sh.write(m, 1, power)
    energy = (np.mean(self.mDur)/1000) * y * np.mean(self.mCurr)
    m += 1
    sh.write(m, 0, "Energy(J)", style=style)
    sh.write(m, 1, energy)
    book.save(filename)
if __name__ == "__main__":

    parser = argparse.ArgumentParser(description='Get statistics.

parser.add_argument('-e', dest='experimentName', type=str, 
    required=True, help='Name of the experiment')
parser.add_argument('-t', dest='threshold', type=float, 
    required=True, help='Current Threshold')
parser.add_argument('-s', dest='maxsamples', type=int, required=True, 
    help='Number of data samples for statistical analysis.')
parser.add_argument('-F', dest='samplingFreq', type=float, 
    required=True, help='Sampling frequency in MS/s.')
parser.add_argument('-v', dest='voltage', type=float, required=True, 
    help='Voltage to power up the board')
parser.add_argument('-c', dest='captureLen', type=float, required=True, 
    help='Capture duration in msec')

    args = parser.parse_args()

    THRESHOLD = args.threshold
    SAMPLINGFREQ = args.samplingFreq
    FILENAME = "\Data\Report_" + args.experimentName + ".xls"
    CLENGTH = args.captureLen

    em = energyMeasure()
    em.openScope()

    try:
        while em.numberOfSamples < 10:
            em.armMeasure()
            em.measure(args.experimentName)

            em.output(FILENAME,args.experimentName,args.voltage,THRESHOLD)
    except KeyboardInterrupt:
        pass

    em.closeScope()
mDot P2P Communication Header Files

B.1 dot_util.h

```c
#ifndef __DOT_UTIL_H__
#define __DOT_UTIL_H__

#include "mbed.h"
#include "mDot.h"
#include "MTSLog.h"
#include "MTSText.h"
#include "ISL29011.h"
#include "example_config.h"

extern mDot* dot;

void display_config();

void update_ota_config_name_phrase(std::string network_name,
                                    std::string network_passphrase,
                                    uint8_t frequency_sub_band,
                                    bool public_network,
                                    uint8_t ack);

void update_ota_config_id_key(uint8_t* network_id, uint8_t* network_key,
                               uint8_t frequency_sub_band,
                               bool public_network, uint8_t ack);

void update_manual_config(uint8_t* network_address, uint8_t* network_session_key,
                           uint8_t* data_session_key,
                           uint8_t frequency_sub_band,
                           bool public_network, uint8_t ack);

void update_peer_to_peer_config(uint8_t* network_address, uint8_t* network_session_key,
                                 uint8_t* data_session_key,
                                 uint32_t tx_frequency, uint8_t tx_datarate,
                                 uint8_t tx_power);

void update_network_link_check_config(uint8_t link_check_count,
                                       uint8_t link_check_threshold);

void join_network();
```
```cpp
void sleep_wake_rtc_only ( bool deepsleep ) ;
void sleep_wake_interrupt_only ( bool deepsleep ) ;
void sleep_wake_rtc_or_interrupt ( bool deepsleep ) ;
void sleep_save_io () ;
void sleep_configure_io () ;
void sleep_restore_io () ;
void send_data ( std :: vector < uint8_t > data ) ;
#endif

Listing B.1: dot_util.h (Used from [mula])

B.2 mDotEvent.h

```
typedef struct {
    LoRaMacEventInfoStatus Status;
    lorax::DownlinkControl Ctrl;
    bool TxAckReceived;
    uint8_t TxNbRetries;
    uint8_t TxDatarate;
    uint8_t RxPort;
    uint8_t *RxBuffer;
    uint8_t RxBufferSize;
    int16_t RxRssi;
    uint8_t RxSnr;
    uint16_t Energy;
    uint8_t DemodMargin;
    uint8_t NbGateways;
} LoRaMacEventInfo;

class mDotEvent: public lorax::MacEvents {
public:

    mDotEvent()
        :
            LinkCheckAnsReceived(false),
            DemodMargin(0),
            NbGateways(0),
            PacketReceived(false),
            RxPort(0),
            RxPayloadSize(0),
            PongReceived(false),
            PongRssi(0),
            PongSnr(0),
            AckReceived(false),
            TxNbRetries(0)
    {
        memset(&_flags, 0, sizeof(LoRaMacEventFlags));
        memset(&_info, 0, sizeof(LoRaMacEventInfo));
    }

    virtual ~mDotEvent() {} 

    virtual void MacEvent(LoRaMacEventFlags *flags,
        LoRaMacEventInfo *info) {
        if (mts::MTSLog::getLogLevel() ==
            mts::MTSLog::TRACE_LEVEL) {
            // Code for handling MacEvent...
std::string msg = "OK";
switch (info->Status) {
    case LORAMAC_EVENT_INFO_STATUS_ERROR:
        msg = "ERROR";
        break;
    case LORAMAC_EVENT_INFO_STATUS_TX_TIMEOUT:
        msg = "TX_TIMEOUT";
        break;
    case LORAMAC_EVENT_INFO_STATUS_RX_TIMEOUT:
        msg = "RX_TIMEOUT";
        break;
    case LORAMAC_EVENT_INFO_STATUS_RX_ERROR:
        msg = "RX_ERROR";
        break;
    case LORAMAC_EVENT_INFO_STATUS_JOIN_FAIL:
        msg = "JOIN_FAIL";
        break;
    case LORAMAC_EVENT_INFO_STATUS_DOWNLINK_FAIL:
        msg = "DOWNLINK_FAIL";
        break;
    case LORAMAC_EVENT_INFO_STATUS_ADDRESS_FAIL:
        msg = "ADDRESS_FAIL";
        break;
    case LORAMAC_EVENT_INFO_STATUS_MIC_FAIL:
        msg = "MIC_FAIL";
        break;
    default:
        break;
}
logTrace("Event: %s", msg.c_str());

logTrace("Flags Tx: %d Rx: %d RxData: %d RxSlot: %d LinkCheck: %d JoinAccept: %d",
    flags->Bits.Tx, flags->Bits.Rx, flags->Bits.RxData, flags->Bits.RxSlot, flags->Bits.LinkCheck, flags->Bits.JoinAccept);
logTrace("Info: Status: %d ACK: %d Retries: %d TxDR: %d RxPort: %d RxSize: %d RSSI: %d SNR: %d Energy: %d Margin: %d Gateways: %d",
    info->Status, info->TxAckReceived, info->TxNbRetries, info->TxDataRate, info->RxPort, info->RxBufferSize, info->RxRssi, info->RxSnr, info->Energy, info->DemodMargin, info->NbGateways);
}

virtual void TxDone(uint8_t dr) {
    RxPayloadSize = 0;
    LinkCheckAnsReceived = false;
    PacketReceived = false;
    AckReceived = false;
B.2. MDOTEVENT.H

PongReceived = false;
TxNbRetries = 0;

logDebug("mDotEvent − TxDone");
memset(&_flags, 0, sizeof(LoRaMacEventFlags));
memset(&_info, 0, sizeof(LoRaMacEventInfo));

_flags.Bits.Tx = 1;
_info.TxDatarate = dr;
_info.Status = LORAMAC_EVENT_INFO_STATUS_OK;
Notify();

void Notify() {
    MacEvent(&_flags, &_info);
}

virtual void TxTimeout(void) {
    logDebug("mDotEvent − TxTimeout");

    _flags.Bits.Tx = 1;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_TX_TIMEOUT;
    Notify();
}

virtual void JoinAccept(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr) {
    logDebug("mDotEvent − JoinAccept");

    _flags.Bits.Tx = 0;
    _flags.Bits.JoinAccept = 1;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_OK;
    Notify();
}

virtual void JoinFailed(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr) {
    logDebug("mDotEvent − JoinFailed");

    _flags.Bits.Tx = 0;
    _flags.Bits.JoinAccept = 1;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_JOIN_FAIL;
    Notify();
}

virtual void MissedAck(uint8_t retries) {
    logDebug("mDotEvent − MissedAck : retries %u", retries);
    TxNbRetries = retries;
    _info.TxNbRetries = retries;
}
```c
virtual void PacketRx(uint8_t port, uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr, lora::DownlinkControl ctrl, uint8_t slot, uint8_t retries = 0) {
    logDebug("mDotEvent - PacketRx");
    RxPort = port;
    PacketReceived = true;

    memcpy(RxPayload, payload, size);
    RxPayloadSize = size;

    if (ctrl.Bits.Ack) {
        AckReceived = true;
    }

    if (mts::MTSLog::getLogLevel() == mts::MTSLog::TRACE_LEVEL) {
        std::string packet = mts::Text::bin2hexString(RxPayload, size);
        logTrace("Payload: %s", packet.c_str());
    }

    _flags.Bits.Tx = 0;
    _flags.Bits.Rx = 1;
    _flags.Bits.RxData = size > 0;
    _flags.Bits.RxSlot = slot;
    _info.RxBuffer = payload;
    _info.RxBufferSize = size;
    _info.RxPort = port;
    _info.RxRssi = rssi;
    _info.RxSnr = snr;
    _info.TxAckReceived = AckReceived;
    _info.TxNbRetries = retries;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_OK;
    Notify();
}
```

```c
virtual void RxDone(uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr, lora::DownlinkControl ctrl, uint8_t slot) {
    logDebug("mDotEvent - RxDone");
}
```

```c
virtual void Pong(int16_t m_rssi, int8_t m_snr, int16_t s_rssi, int8_t s_snr) {
    logDebug("mDotEvent - Pong");
    PongReceived = true;
    PongRssi = s_rssi;
    PongSnr = s_snr;
}
```

```c
virtual void NetworkLinkCheck(int16_t m_rssi, int8_t m_snr, int16_t s_rssi, int8_t s_snr) {
    logDebug("mDotEvent - NetworkLinkCheck");
}
```
LinkCheckAnsReceived = true;
DemodMargin = s_snr;
NbGateways = s_gateways;

_flags.Bits.Tx = 0;
_flags.Bits.LinkCheck = 1;
_info.RxRssi = m_rssi;
_info.RxSnr = m_snr;
_info.DemodMargin = s_snr;
_info.NbGateways = s_gateways;
_info.Status = LORAMAC_EVENT_INFO_STATUS_OK;
Notify();
{
virtual void RxTimeout(uint8_t slot) {
    // logDebug("mDotEvent − RxTimeout");
    _flags.Bits.Tx = 0;
    _flags.Bits.RxSlot = slot;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_RX_TIMEOUT;
    Notify();
}

virtual void RxError(uint8_t slot) {
    logDebug("mDotEvent − RxError");
    memset(&_flags, 0, sizeof(LoRaMacEventFlags));
    memset(&_info, 0, sizeof(LoRaMacEventInfo));
    _flags.Bits.RxSlot = slot;
    _info.Status = LORAMAC_EVENT_INFO_STATUS_RX_ERROR;
    Notify();
}

virtual uint8_t MeasureBattery(void) {
    return 255;
}

bool LinkCheckAnsReceived;
uint8_t DemodMargin;
uint8_t NbGateways;

bool PacketReceived;
uint8_t RxPort;
uint8_t RxPayload[255];
uint8_t RxPayloadSize;

bool PongReceived;
int16_t PongRssi;
int16_t PongSnr;

bool AckReceived;
Listing B.2: mDotEvent.h (Used from [mula])

B.3 RadioEvent.h
```cpp
class RadioEvent : public mDotEvent
{
public:
  RadioEvent () {}

virtual ~RadioEvent () {}

/*! 
 * MAC layer event callback prototype.
 * 
 * \param [IN] flags Bit field indicating the MAC events occurred 
 * \param [IN] info  Details about MAC events occurred 
 */
virtual void MacEvent (LoRaMacEventFlags* flags, LoRaMacEventInfo* info) {
  if (mts::MTSLog::getLogLevel () == mts::MTSLog::TRACE_LEVEL) {
    std::string msg = "OK" ;
    switch (info->Status) {
    case LORAMAC_EVENT_INFO_STATUS_ERROR:
      msg = "ERROR" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_TX_TIMEOUT:
      msg = "TX_TIMEOUT" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_RX_TIMEOUT:
      msg = "RX_TIMEOUT" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_RX_ERROR:
      msg = "RX_ERROR" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_JOIN_FAIL:
      msg = "JOIN_FAIL" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_DOWNLINK_FAIL:
      msg = "DOWNLINK_FAIL" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_ADDRESS_FAIL:
      msg = "ADDRESS_FAIL" ;
      break ;
    case LORAMAC_EVENT_INFO_STATUS_MIC_FAIL:
      msg = "MIC_FAIL" ;
      break ;
    default :
      break ;
    }
    logTrace ("Event: %s", msg.c_str ());
  }
  logTrace ("Flags Tx: %d Rx: %d RxData: %dRxSlot: %d LinkCheck: %d JoinAccept: %d", 
```

flags->Bits.Tx, flags->Bits.Rx,
flags->Bits.RxData, flags->Bits.RxSlot, flags->Bits.LinkCheck,
flags->Bits.JoinAccept);
    logTrace("Info: Status: %d ACK: %d Retries: %d TxDR: %d
RxPort: %d RxSize: %d RSSI: %d SNR: %d Energy: %d Margin: %d
Gateways: %d",
    info->Status, info->TxAckReceived,
    info->TxNbRetries, info->TxDatarate, info->RxPort,
    info->RxBufferSize,
    info->RxRssi, info->RxSnr, info->Energy,
    info->DemodMargin, info->NbGateways);

    if (flags->Bits.Rx) {
        logDebug("Rx %d bytes", info->RxBufferSize);
        if (info->RxBufferSize > 0) {
            // print RX data as hexadecimal
            // printf("Rx data: %s\r\n", mts::Text::bin2hexString(info->RxBuffer,
            info->RxBufferSize).c_str());

            // print RX data as string
            std::string rx((const char*)info->RxBuffer,
                    info->RxBufferSize);
            printf("Rx data: %s\r\n", rx.c_str());
        }
    }
#endif

Listing B.3: RadioEvent.h (Used from [mula])
#include "dot_util.h"
#include "RadioEvent.h"

// Particle Sensor
#define FAN_PIN PA_3
#define LED_PIN PA_0
#define MEASURE_PIN PB_0
#define FANFREESAMPLE 50
#define SAMPLING_TIME 280
#define DELTA_TIME 40
#define SLEEP_TIME 9680
#define SENSING_SAMPLE 2
#define P_ACTIVE 0.1536
#define PSENSOR 0.493
#define P_TX 0.324745
#define P_SLEEP 0.0457
#define SLEEP_DUR 120
#define PRE_SLEEP_DUR 0.04399
#define SENSING_SAMPLE 1
#define PRE_SLEEP_ENERGY 0.0027
#define POST_SLEEP_ENERGY 0.0176
#define PRE_TX_ENERGY 0.00086779
#define POST_TX_ENERGY 0.00152
#define PRE_TX_DUR 0.011
#define POST_TX_DUR 0.00796
#define FAN_ON_DUR 10
#define OP_1 0
#define OP_2 1
#define OP_3 2
#define OP_4 3
// Partial Sensor
float getVsWithOutFAN(void);
float getPartialDensity(float);
float getDustVoltageSample(void);
void generate_event();

static uint8_t network_address[] = {0x01, 0x02, 0x03, 0x04};
static uint8_t network_session_key[] = {0x01, 0x02, 0x03, 0x04, 0x01,
                                       0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04};
static uint8_t data_session_key[] = {0x01, 0x02, 0x03, 0x04, 0x01,
                                      0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04};

mDot* dot = NULL;
Serial pc(USBTX, USBRX);
DigitalOut fanCtrl(FAN_PIN);
DigitalOut ledpower(LED_PIN);
AnalogIn measure(MEASUREV_PIN);

int main() {

  // Custom event handler for automatically displaying RX data
  RadioEvent events;
  uint32_t tx_frequency;
  uint8_t tx_datarate;
  uint8_t tx_power;
  uint8_t frequency_band;

  // Partial Sensor
  float vS = 0; // stores reference voltage
  pc.baud(115200);
  mts::MTSLog::setLogLevel(mts::MTSLog::TRACE_LEVEL);
  dot = mDot::getInstance();
  logInfo("mbed-os library version: %d",MBED_LIBRARY_VERSION);

  // start from a well-known state
  logInfo("defaulting Dot configuration");
  dot->resetConfig();

  // make sure library logging is turned on
  dot->setLogLevel(mts::MTSLog::INFO_LEVEL);

  // attach the custom events handler
  dot->setEvents(&events);

  // update configuration if necessary
  if (dot->getJoinMode() != mDot::PEER_TO_PEER) {

logInfo("changing network join mode to PEER_TO_PEER");
if (dot->setJoinMode(mDot::PEER_TO_PEER) != mDot::MDOT_OK) {
    logError("failed to set network join mode to PEER_TO_PEER");
}

frequency_band = dot->getFrequencyBand();
switch (frequency_band) {
    case mDot::FB_EU868:
        // 250kHz channels achieve higher throughput
        // DR6 : SF7 @ 250kHz
        // DR0 – DR5 (125kHz channels) available but much slower
        tx_frequency = 869850000;
        tx_datarate = mDot::DR6;
        // the 869850000 frequency is 100% duty cycle if the total
        // power is under 7 dBm – tx power 4 + antenna gain 3 = 7
        tx_power = 4;
        break;
    case mDot::FB_US915:
    case mDot::FB_AU915:
        default:
            // 500kHz channels achieve highest throughput
            // DR8 : SF12 @ 500kHz
            // DR9 : SF11 @ 500kHz
            // DR10 : SF10 @ 500kHz
            // DR11 : SF9 @ 500kHz
            // DR12 : SF8 @ 500kHz
            // DR13 : SF7 @ 500kHz
            // DR0 – DR3 (125kHz channels) available but much slower
            tx_frequency = 915500000;
            tx_datarate = mDot::DR13;
            // 915 bands have no duty cycle restrictions, set tx power
            // to max
            tx_power = 20;
            break;
}

// in PEER_TO_PEEM mode there is no join request/response
// transaction
// as long as both Dots are configured correctly, they should be
// able to communicate
update_peer_to_peer_config(network_address, network_session_key,
data_session_key, tx_frequency, tx_datarate, tx_power);

// save changes to configuration
logInfo("saving configuration");
if (!dot->saveConfig()) {
    logError("failed to save configuration");
}

// display configuration
display_config();
l logInfo("getting Vs");
136
vS = getVsWithOutFAN();
138
l logInfo("entering Loop");

float powerConsumption[4] = {P_SENSOR, P_ACTIVE, P_TX, P_SLEEP};
float duration[4] = {0.0};
float staticEnergy[4] = {0.0, PRE_TX_ENERGY+POST_TX_ENERGY,
PRE_SLEEP_ENERGY+POST_SLEEP_ENERGY};

Timer t_op;
float energy = 0.0;
char buffer[50]={0};

while (true) {
    // triggering oscilloscope to start sampling
    generate_event();

    float pDensity = 0.0;
    float pDensity_sum = 0.0;
    float pDensity_sum2 = 0.0;
    float stdErrorPDensity = 0.0;
    std::vector<uint8_t> tx_data;
    int i = 0;

    // Entering sensing phase
    t_op.start();
    for (i=0; i<SENSING_SAMPLE; i++) {
        // get the latest dust sample
        pDensity = getPartialDensity(vS);
        pDensity_sum += pDensity;
        pDensity_sum2 += pDensity * pDensity;
        stdErrorPDensity = (pDensity_sum2 - ((pDensity_sum *
        pDensity_sum)/(i+1)))/(i+1);
        stdErrorPDensity = sqrt(stdErrorPDensity);
    }

    pDensity = pDensity_sum/SENSING_SAMPLE;

    // Capturing sensing duration
    t_op.stop();
    duration[OP_1] = t_op.read();
    t_op.reset();

    // starting processing phase
    t_op.start();
    // join network if not joined
    if (!dot->getNetworkJoinStatus()) {
        join_network();
    }

    // adding average sensed data
// to the transmission frame
    sprintf(buffer, "%f", pDensity);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);

    // adding data separator
    // to the transmission frame
    tx_data.push_back(59);

    // adding std. error of sensed data
    // to the transmission frame
    sprintf(buffer, "%f", stdErrorPDensity);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);

    // adding data separator
    // to the transmission frame
    tx_data.push_back(59);

    energy = 0.0;
    for (int i=0; i<4; i++)
    {
        energy += ((duration[i] * powerConsumption[i]) + staticEnergy[i]);

        // adding each activity phase’s energy consumption
        // to the transmission frame
        sprintf(buffer, "OP%d:%f;", i+1, ((duration[i] * powerConsumption[i]) + staticEnergy[i]));
        for (int ind=0;buffer[ind]!=0;ind++)
            tx_data.push_back(buffer[ind]);
    }

    // adding overall energy consumption
    // to the transmission frame
    sprintf(buffer, "%f", energy);
    for (i=0;buffer[i]!=0;i++)
        tx_data.push_back(buffer[i]);

    // Capturing processing duration
    t_op.stop();
    duration[OP_2] = t_op.read_us();
    duration[OP_2] = duration[OP_2]/1000000;
    t_op.reset();

    // Sending data
    t_op.start();
    send_data(tx_data);
    t_op.stop();

    // Capturing transmission duration
    duration[OP_3] = t_op.read_ms();
duration[OP_3] = duration[OP_3]/1000 - (PRE_TX_DUR + POST_TX_DUR);
t_op.reset();

// Entering sleep mode for SLEEP_DUR seconds
dot->sleep(SLEEP_DUR, mDot::RTC_ALARM, false);
// Since the timer routine does not work in the sleep mode
// Therefore, I have fixed its duration.
// Moreover, I did not subtract the POST_SLEEP_DUR
// because after analysis I found that the POST_SLEEP_DUR
// is added to the actual sleep duration.
duration[OP_4] = SLEEP_DUR - PRE_SLEEP_DUR;

// Single the Oscilloscope to stop sampling
generate_event();

// Wait for a duration greater than the
// actual sensing cycle duration.
// This wait was added to get the precise measurement
// from the oscilloscope.
// You can comment this wait if you are not sampling
// automatically from the oscilloscope.
wait(SLEEP_DUR + (FAN_ON_DUR * SENSING_SAMPLE) + 5);
}
return 0;
}

float getDustVoltageSample(void)
{
    float dustVMeasured = 0.0;
    ledpower = 0; // turn the LED on
    wait_us(SAMPLING_TIME); //Wait samplingTime before reading V0output
    // read the dust value in (0−1.0)−−>(0V−3.3V)
    dustVMeasured = measure.read();
    wait_us(DELTA_TIME); //Wait deltaTime before shutting off LED
    ledpower = 1; // turn the LED off
    wait_us(SLEEP_TIME); // No use in this example
    return dustVMeasured;
}

float getVsWithOutFAN(void)
{
    fanCtrl = 0; // Turn off the fan
    float vsMeasured_sum = 0;
    int fanFreeSampleCtr = 0;
    for (fanFreeSampleCtr = 0; fanFreeSampleCtr < FANFREESAMPLE; fanFreeSampleCtr++)
float getPartialDensity(float vS) {
    float voMeasured = 0; // Stores measured output (0-1.0) from sensor
    float calcVoltage = 0; // Calculate actual voltage output from sensor
    float dustDensity = 0;

    // Turn on the FAN to start sampling
    fanCtrl = 1;
    wait(FAN_ON_DUR); // Minimum intermittent time is 10s
    voMeasured = getDustVoltageSample();

    // Calculating output voltage from Raw analog signal to mV
    calcVoltage = voMeasured * 3.3;

    // Calculating dust density
    dustDensity += (0.6 * (calcVoltage - vS));

    // Turn off the FAN
    fanCtrl = 0;
    return dustDensity;
}

void generate_event() {
    DigitalOut led1(PA_2);
    led1 = 1;
    wait_ms(100);
    led1 = 0;
}
# include "dot_util.h"
#include "RadioEvent.h"

#define P_ACTIVE 0.1536
#define P_SENSOR 0.1415
#define P_TX 0.324745
#define P_SLEEP 0.0012

#define SLEEP_DUR 2
#define PRE_SLEEP_DUR 0.04399
#define SENSING_SAMPLE 2
#define PRE_SLEEP_ENERGY 0.0027
#define POST_SLEEP_ENERGY 0.0176

#define PRE_TX_ENERGY 0.00086779
#define POST_TX_ENERGY 0.00152
#define PRE_TX_DUR 0.011
#define POST_TX_DUR 0.00796

#define OP_1 0
#define OP_2 1
#define OP_3 2
#define OP_4 3

// DS18B20 OneWire pin
// D13 on Dev Board, pin 18 on mDot.
// Compatible with Oxford Flood Network PCB temperature sensor.
#define DATA_PIN PA_5

// Temperature sensor object
DS1820 probe(DATA_PIN);

void generate_event();

static uint8_t network_address[] = { 0x01, 0x02, 0x03, 0x04 };  
static uint8_t network_session_key[] = { 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04 };
static uint8_t data_session_key[] = { 0x01, 0x02, 0x03, 0x04, 0x01, 
0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04, 0x01, 0x02, 0x03, 0x04 };

mDot* dot = NULL;
Serial pc(USBTX, USBRX);

int main() {

    // Custom event handler for automatically displaying RX data
    RadioEvent events;
    uint32_t tx_frequency;
    uint8_t tx_datarate;
    uint8_t tx_power;
    uint8_t frequency_band;

    pc.baud(115200);

    mts::MTSLog::setLogLevel(mts::MTSLog::TRACE_LEVEL);

dot = mDot::getInstance();

    logInfo("mbed-os library version: %d", MBED_LIBRARY_VERSION);

    // start from a well-known state
    logInfo("defaulting Dot configuration");
dot->resetConfig();

    // make sure library logging is turned on
    dot->setLogLevel(mts::MTSLog::INFO_LEVEL);

    // attach the custom events handler
    dot->setEvents(&events);

    // update configuration if necessary
    if (dot->getJoinMode() != mDot::PEER_TO_PEER) {
        logInfo("changing network join mode to PEER_TO_PEER");
        if (dot->setJoinMode(mDot::PEER_TO_PEER) != mDot::MDOT_OK) {
            logError("failed to set network join mode to PEER_TO_PEER");
        }
    }

    frequency_band = dot->getFrequencyBand();
    switch (frequency_band) {
        case mDot::FB_EU868:
            // 250kHz channels achieve higher throughput
            // DR6 : SF7 @ 250kHz
            // DR0 – DR5 (125kHz channels) available but much slower
            tx_frequency = 869850000;
            tx_datarate = mDot::DR6;
            // the 869850000 frequency is 100% duty cycle if the total
            // power is under 7 dBm – tx power 4 + antenna gain 3 = 7
            tx_power = 4;
            break;
        case mDot::FB_868:
            // 868MHz channels
            // DR7 : SF7 @ 868MHz
            // DR0 – DR6 (868MHz channels) available but much slower
            tx_frequency = 868000000;
            tx_datarate = mDot::DR7;
            // the 868000000 frequency is 100% duty cycle if the total
            // power is under 7 dBm – tx power 4 + antenna gain 3 = 7
            tx_power = 4;
            break;
    }
case mDot::FB_US915:
case mDot::FB_AU915:
default:
  // 500kHz channels achieve highest throughput
  // DR8  : SF12 @ 500kHz
  // DR9  : SF11 @ 500kHz
  // DR10 : SF10 @ 500kHz
  // DR11 : SF9  @ 500kHz
  // DR12 : SF8  @ 500kHz
  // DR13 : SF7  @ 500kHz
  // DR0−DR3 (125kHz channels) available but much slower
  tx_frequency = 915500000;
  tx_datarate = mDot::DR13;
  // 915 bands have no duty cycle restrictions, set tx power
to max
  tx_power = 20;
  break;
}
// in PEER_TO_PEER mode there is no join request/response
// transaction
// as long as both Dots are configured correctly, they should be
// able to communicate
update_peer_to_peer_config(network_address, network_session_key,
data_session_key, tx_frequency, tx_datarate, tx_power);

// save changes to configuration
logInfo("saving configuration");
if (!dot->saveConfig()) {
  logError("failed to save configuration");
}

// display configuration
display_config();

// Set the Temperature sensor resolution, 9 bits is enough and
// makes it faster to provide a reading.
probe.setResolution(9);

float powerConsumption[4] = {P_SENSOR, P_ACTIVE, P_TX, P_SLEEP};
float duration[4] = {0.0};
float staticEnergy[4] = {0.0, PRE_TX_ENERGY+POST_TX_ENERGY,
                        PRE_SLEEP_ENERGY+POST_SLEEP_ENERGY};

Timer t_op;
float energy = 0.0;
char buffer[50] = {0};
float dur;

while (true) {
  generate_event();
  volatile float temperature = 0.0;
  float temperature_sum = 0.0;
```c
float temperature_sum2 = 0.0;
float stdErrorTemperature = 0.0;
std::vector<uint8_t> tx_data;
int i=0;
float tempdur=0.0;

// Entering sensing phase
t_op.start();
for (i=0; i<SENSING_SAMPLE; i++){
    // Start temperature conversion, wait until ready
    probe.convertTemperature(true, DS1820::all_devices);
    temperature = probe.temperature();

    temperature_sum += temperature;
    temperature_sum2 += temperature * temperature;
    stdErrorTemperature = (temperature_sum2 -
                         ((temperature_sum * temperature_sum)/(i+1)))/(i+1);
    stdErrorTemperature = sqrt(stdErrorTemperature);
}

// Capturing sensing duration
//  t_op.stop();
//  duration[OP_1] = t_op.read();
//  t_op.reset();

// starting processing phase
//  t_op.start();
//  if (!dot->getNetworkJoinStatus()) {
//      join_network();
//  }

// adding average sensed data
// to the transmission frame
sprintf(buffer, "%f", temperature);
for (i=0; buffer[i]!=0; i++)
    tx_data.push_back(buffer[i]);

// adding data separator
// to the transmission frame
tx_data.push_back(59);

// adding std.error of sensed data
// to the transmission frame
sprintf(buffer, "%f", stdErrorTemperature);
for (i=0; buffer[i]!=0; i++)
    tx_data.push_back(buffer[i]);

// adding data separator
// to the transmission frame
```
tx_data.push_back(59);
energy = 0.0;
for (int i=0; i<4; i++)
{
    energy += (duration[i] * powerConsumption[i]) + staticEnergy[i]);

    // adding each activity phase’s energy consumption
    // to the transmission frame
    sprintf(buffer, "OP%d:%f;", i+1, (duration[i] *
    powerConsumption[i]) + staticEnergy[i]));
    for (int ind = 0; buffer[ind] != 0; ind++)
    tx_data.push_back(buffer[ind]);
}

// adding overall energy consumption
// to the transmission frame
sprintf(buffer, "%f", energy);
for (i=0; buffer[i] != 0; i++)
    tx_data.push_back(buffer[i]);

// Capturing processing duration
t_op.stop();
duration[OP_2] = t_op.read_us();
duration[OP_2] = duration[OP_2]/1000000;
t_op.reset();

// Sending data
t_op.start();
send_data(tx_data);
t_op.stop();

// Capturing transmission duration
duration[OP_3] = t_op.read_ms();
duration[OP_3] = duration[OP_3]/1000 - (PRE_TX_DUR + POST_TX_DUR);
t_op.reset();

// Entering sleep mode for SLEEP_DUR seconds
dot->sleep(SLEEP_DUR, mDot::RTC_ALARM, false);
// Since the timer routine does not work in the sleep mode
// Therefore, I have fixed its duration.
// Moreover, I did not subtract the POST_SLEEP_DUR
// because after analysis I found that the POST_SLEEP_DUR
// is added to the actual sleep duration.
duration[OP_4] = SLEEP_DUR - PRE_SLEEP_DUR;

// Single the Oscilloscope to stop sampling
generate_event();

// Wait for a duration greater than the
// actual sensing cycle duration.
// This wait was added to get the precise measurement
// from the oscilloscope.
// You can comment this wait if you are not sampling
// automatically from the oscilloscope.
wait (SLEEP_DUR + (FAN_ON_DUR * SENSING_SAMPLE) + 5);

return 0;

void generate_event()
{
    DigitalOut led1 (PA_2);
    led1 = 1;
    wait_ms (100);
    led1 = 0;
}
Appendix

waspMote CO2 Sensing
Energy-Aware Application

E.1 Header File

```c
#define DEVICE_ID "wmt-v12-v3-1"
#define DEVICE_EUI "0041549CB158AB46"
#define DEVICE_ADDR "260116E2"
#define NWK_SESSION_KEY "131CF400BBC2D7C0CB35E86CA9FA7022"
#define APP_SESSION_KEY "184F79B3706D6DA8F46AE4AFFFF721C4"
#define APP_KEY "184F79B3706D6DA8F46AE4AFFFF721C4"
#define TRANSMISSION_POWER 1 // 14 dBm

#define LW_CH 8
uint32_t lwFreqs[] = {868100000, 868300000, 868500000, 867100000, 867300000, 867500000, 867700000, 867900000};
```

configParams.h

E.2 Source File

```c
#include <WaspSensorGas_Pro.h>
#include <WaspFrame.h>
#include <WaspLoRaWAN.h>
#include "configParams.h"

#define VERSION 3
#define PORT 3 // Port to use in Back-End: from 1 to 223
#define SOCKET SOCKET0
#define MAX_SENSE_COUNT 1
```
```c
float itmTemperature, temperature; // Stores the temperature in celsius
float itmHumidity, humidity;   // Stores the relative humidity in %RH
float itmPressure, pressure;   // Stores the pressure in Pa
float itmCO2, itmNO2, co2Concentration, no2Concentration;

unsigned long timestamp;
int i;

Gas co2(SOCKET_A);
uint8_t errorLW;
volatile int senseCount;

float PowerConsumption[3] = {PC_SENSE, PC_TRANSMIT, PC_SLEEP};
unsigned long Duration[3] = {0, 0, 20};

void configureLoRaWAN();
void frequencyConfiguration();
uint8_t hexCharsToByte(char leftHexC, char rightHexC);

void setup()
{
    configureLoRaWAN();
    frame.setID(DEVICE_ID);

    // Making the digital output
    // for oscilloscope automatic
    // sampling trigger.
    pinMode(DIGITAL3, OUTPUT);
    digitalWrite(DIGITAL3, 0);
}

void loop()
{
    // start Oscilloscope Sampling
    generate_event();

    // Starting sensing process
    RTC.ON();
    timestamp = RTC.getEpochTime();

    co2.ON();
    delay(10000);
    co2Concentration = temperature = humidity = pressure = 0;
    senseCount = 0;

    while(senseCount < MAX_SENSE_COUNT)
```
{  
    delay(1000);
    //Sense the values and start it all over again if there is a value  
    // out of range.
    itmCO2 = co2.getConc();
    itmTemperature = co2.getTemp();
    itmHumidity = co2.getHumidity();
    itmPressure = co2.getPressure();
    co2Concentration += itmCO2;
    temperature += itmTemperature;
    humidity += itmHumidity;
    pressure += itmPressure;
    senseCount++;
    co2.OFF();
    // Storing the sensing duration
    Duration[0] = RTC.getEpochTime() - timestamp;
}
#if defined __DEBUG
    USB.println(F("Sense Duration Start:"));
    USB.println(timestamp);
    USB.println(F("Sense Duration End:"));
    USB.println(RTC.getEpochTime());
    USB.println(F("Sense Duration:"));
    USB.println(Duration[0]);
#endif __DEBUG
RTC.OFF();
    co2Concentration /= MAXSENSE_COUNT;
    temperature /= MAXSENSE_COUNT;
    humidity /= MAXSENSE_COUNT;
    pressure /= MAXSENSE_COUNT;
    // Create a new tx-frame
    frame.createFrame(BINARY);
    frame.addSensor(SENSOR_GP_CO2, (double)co2Concentration);
    // adds the energy consumption to tx-frame
    addEnergyConsumptiontoFrame();
#endif __DEBUG
    frame.showFrame();
#endif
// Entering transmission phase
RTC.ON();
timestamp = RTC.getEpochTime();
Switch on LoRaWAN
errorLW = LoRaWAN.ON(SOCKET);
frequencyConfiguration();
#ifdef _DEBUG
   // Check status
   if (errorLW == 0)
   {
      USB.println(F("1. Switch ON OK OK"));
   }
   else
   {
      USB.print(F("1. Switch ON error = ' "));
      USB.println(errorLW, DEC);
   }
#endif

errorLW = LoRaWAN.joinABP();

if (errorLW == 0)
{
   //Send unconfirmed packet
   errorLW = LoRaWAN.sendUnconfirmed(PORT, frame.buffer, frame.length);

   // Error messages:
   /*
   "6": Module hasn't joined a network
   "5": Sending error
   "4": Error with data length
   "2": Module didn't response
   "1": Module communication error
   */
   // Check status
   if (errorLW == 0)
   {
      #ifdef _DEBUG
      USB.println(F("3. Send Unconfirmed packet OK"));
      if (LoRaWAN._dataReceived)
      {
         USB.print(F(" There's data on port number ' "));
         USB.print(LoRaWAN._port,DEC);
         USB.print(F(".\r\n Data: ' "));
         for (i=0; i < 101; i++){
            USB.print(i);
            USB.print(\n );
            USB.print(\r
 );
            USB.print(LoRaWAN._data[i]);
            USB.print(\n );
         }
         USB.println("\n");
         testCount = (uint8_t) hexCharsToByte(LoRaWAN._data[0], LoRaWAN._data[1]); //Test only the first character
         USB.print("testCount : ");
         USB.println((int)testCount);
      }#endif
   }
E.2. SOURCE FILE

```c
else
{
  #ifdef _DEBUG
    USB.print(F("3. Send Unconfirmed packet error = "));
    USB.println(errorLW, DEC);
    #endif
  }
}
else
{
  #ifdef _DEBUG
    USB.print(F("2. Join network error = "));
    USB.println(errorLW, DEC);
    #endif
  }
}
#endif

errorLW = LoRaWAN.getRadioFreq();
if (errorLW == 0)
{
  USB.print(F("Operating radio frequency: "));
  USB.println(LoRaWAN._radioFreq);
}
else
  USB.print(F("Could not get the radio frequency."));
errorLW = LoRaWAN.getRadioFreqDeviation();
if (errorLW == 0)
{
  USB.print(F("Operating radio frequency deviation: "));
  USB.println(LoRaWAN._radioFreqDev);
}
else
  USB.print(F("Could not get the radio frequency deviation."));
errorLW = LoRaWAN.getRadioMode();
errorLW = LoRaWAN.getRadioPower();
if (errorLW == 0)
{
  USB.print(F("Operating radio power: "));
  USB.println(LoRaWAN._radioPower);
}
else
  USB.print(F("Could not get the radio power."));
if (errorLW == 0)
{
  USB.print(F("Operating radio mode: "));
  USB.println(LoRaWAN._radioMode);
}
else
  USB.print(F("Could not get the radio mode."));
```cpp
errorLW = LoRaWAN.getRadioBW();
if (errorLW == 0)
{
    USB.print(F("Operating radio bandwidth: "));
    USB.println(LoRaWAN._radioBW);
}
else
    USB.print(F("Could not get the radio bandwidth."));
errorLW = LoRaWAN.getRadioSF();
if (errorLW == 0)
{
    USB.print(F("Operating radio spreading factor: "));
    USB.println(LoRaWAN._radioSF);
}
else
    USB.print(F("Could not get the radio spreading factor."));
#endif

errorLW = LoRaWAN.OFF(SOCKET);
// storing transmission duration
Duration[1]= RTC.getEpochTime() - timestamp;
#ifdef _DEBUG
USB.print(F("Transmit Duration Start:"));
USB.println(timestamp);
USB.print(F("Transmit Duration End:"));
USB.println(RTC.getEpochTime());
USB.print(F("Transmit Duration:"));
USB.println(Duration[1]);
#endif _DEBUG
RTC.OFF();
#ifdef _DEBUG
// Check status
if (errorLW == 0)
{
    USB.println(F("4. Switch OFF OK"));
}
else
{
    USB.print(F("4. Switch OFF error = "));
    USB.println(errorLW, DEC);
}
#endif
// Entering sleep phase
PWRC.deepSleep("00:00:00:5", RTC_OFFSET, RTC_ALM1_MODE1, ALL_OFF);
generate_event();
// Adding a delay of 50s before
// starting the next cycle
delay(20000);
```
void generate_event()
{
    // setting event pin for oscilloscope measurements
    digitalWrite(DIGITAL3, 1);
    delay(100);
    digitalWrite(DIGITAL3, 0);
}

void addEnergyConsumptiontoFrame(void)
{
    char number[20];
    float energy = 0;
    for (int i = 0; i < 3; i++)
    {
        // adding the energy consumption of each activity phase
        Utils.float2String((PowerConsumption[i] * Duration[i]), number, 3);
        frame.addSensor(SENSOR_STR, number);
        #ifdef _DEBUG
        USB.println(number);
        Utils.float2String(PowerConsumption[i], number, 3);
        USB.println(number);
        Utils.float2String(Duration[i], number, 3);
        USB.println(number);
        #endif
        energy = energy + (PowerConsumption[i] * Duration[i]);
    }
    // adding accumulated energy consumption
    Utils.float2String(energy, number, 3);
    frame.addSensor(SENSOR_STR, number);
    #ifdef _DEBUG
    USB.println(number);
    #endif
}

uint8_t hexCharToInt(char c){
    switch (c){
        case '0': return 0;
        case '1': return 1;
        case '2': return 2;
        case '3': return 3;
        case '4': return 4;
        case '5': return 5;
        case '6': return 6;
        case '7': return 7;
        case '8': return 8;
        case '9': return 9;
        case 'A': case 'a': return 10;
        case 'B': case 'b': return 11;
        case 'C': case 'c': return 12;
        case 'D': case 'd': return 13;
    }
void frequencyConfiguration()
{
   //LoRaWAN must be turned on before this function is called.
   //This function depends on the constant 'LW_CH' and the array
   //lwFreqs' in configParams.h.
   int ch;
   for(ch=1; ch <= LW_CH; ch++)
   {
      errorLW = LoRaWAN.setChannelFreq(ch, lwFreqs[ch-1]);
      #ifdef _DEBUG
      if(errorLW == 0){
         //USB.print(F("The channel ");
         //USB.print(ch);
         //USB.print(F(" is set to ");
         //USB.print(lwFreqs[ch-1]/1000000.0);
         //USB.println(F(" MHz.");
      } else {
         //USB.print(F("Error when setting the frequency channel ");
         //USB.print(ch);
         //USB.print(F(". Error code: ");
         //USB.println(errorLW, DEC);
      }
      #endif
   }
}

void configureLoRaWAN()
{
   // 1. switch on
   uint8_t error = LoRaWAN.ON(SOCKET);
   #ifdef _DEBUG
   // Check status
   if ( error == 0 )
   {
      USB.println(F("1. Switch ON OK OK OK");
   }
   else
{  
    USB.print(F("1. Switch ON error = "));
    USB.println(error, DEC);
}
#endif

// ///////////////////////////////////////////
// 2. Reset to factory default values
// ///////////////////////////////////////////

error = LoRaWAN.factoryReset();

#if defined_DEBUG
    // Check status
    if ( error == 0 )
        {
            USB.println(F("2. Reset to factory default values OK"));
        }
    else
        {
            USB.print(F("2. Reset to factory error = "));
            USB.println(error, DEC);
        }
#endif

// ///////////////////////////////////////////
// 3. Set/Get Device EUI
// ///////////////////////////////////////////

// Set Device EUI
error = LoRaWAN.setDeviceEUI(DEVICE_EUI);

#if defined_DEBUG
    // Check status
    if ( error == 0 )
        {
            USB.println(F("3.1. Set Device EUI OK"));
        }
    else
        {
            USB.print(F("3.1. Set Device EUI error = "));
            USB.println(error, DEC);
        }
#endif

// Get Device EUI
error = LoRaWAN.getDeviceEUI();

#if defined_DEBUG
    // Check status
    if ( error == 0 )
        {

```
USB.printf(F("3.2. Get Device EUI OK.
"));
USB.printf(F("Device EUI: 
"));
USB.println(LoRaWAN._devEUI);
}
else
{
  USB.printf(F("3.2. Get Device EUI error = \
"));
  USB.println(error, DEC);
}
#endif

////////////////////////////////////////////////////////////////////////////////////
// 4. Set/Get Device Address
////////////////////////////////////////////////////////////////////////////////////

// Set Device Address
error = LoRaWAN.setDeviceAddr(DEVICE_ADDR);
#ifdef _DEBUG
  // Check status
  if( error == 0 )
  {
    USB.println(F("4.1. Set Device address OK"));
  }
else
  {
    USB.printf(F("4.1. Set Device address error = \
"));
    USB.println(error, DEC);
  }
#endif

// Get Device Address
error = LoRaWAN.getDeviceAddr();
#ifdef _DEBUG
  // Check status
  if( error == 0 )
  {
    USB.printf(F("4.2. Get Device address OK. \
"));
    USB.printf(F("Device address: \
"));
    USB.println(LoRaWAN._devAddr);
  }
else
  {
    USB.printf(F("4.2. Get Device address error = \
"));
    USB.println(error, DEC);
  }
#endif

////////////////////////////////////////////////////////////////////////////////////
// 5. Set Network Session Key
////////////////////////////////////////////////////////////////////////////////////
```cpp
error = LoRaWAN.setNwkSessionKey(NWK_SESSION_KEY);

#ifdef _DEBUG
    // Check status
    if ( error == 0 )
    {
        USB.println(F("5. Set Network Session Key OK"));
    }
    else
    {
        USB.print(F("5. Set Network Session Key error = '"));
        USB.println(error, DEC);
    }
#endif

// ////////////////////////////////////////////
// 6. Set Application Session Key
// ////////////////////////////////////////////
error = LoRaWAN.setAppSessionKey(APP_SESSION_KEY);

#ifdef _DEBUG
    // Check status
    if ( error == 0 )
    {
        USB.println(F("6. Set Application Session Key OK"));
    }
    else
    {
        USB.print(F("6. Set Application Session Key error = '"));
        USB.println(error, DEC);
    }
#endif

// ////////////////////////////////////////////
// 7. Set retransmissions for uplink confirmed packet
// ////////////////////////////////////////////
// set retries
error = LoRaWAN.setRetries(7);

#ifdef _DEBUG
    // Check status
    if ( error == 0 )
    {
        USB.println(F("7.1. Set Retransmissions for uplink confirmed packet OK"));
    }
    else
    {
```
USB.print(F("7.1. Set Retransmissions for uplink confirmed packet error = "));
USB.println(error, DEC);
}
#endif

// Get retries
error = LoRaWAN.getRetries();
#ifdef _DEBUG
// Check status
if ( error == 0 )
{
    USB.print(F("7.2. Get Retransmissions for uplink confirmed packet OK.");
    USB.print(F("TX retries: ");
    USB.println(LoRaWAN._retries, DEC);
}
else
{
    USB.print(F("7.2. Get Retransmissions for uplink confirmed packet error = ");
    USB.println(error, DEC);
}
#endif

////////////////////////////////////////////////////////////////////////////////
// 8. Set application key
////////////////////////////////////////////////////////////////////////////////
error = LoRaWAN.setAppKey(APP_KEY);
#ifdef _DEBUG
// Check status
if ( error == 0 )
{
    USB.println(F("8. Application key set OK"));
}
else
{
    USB.print(F("8. Application key set error = ");
    USB.println(error, DEC);
}
#endif

////////////////////////////////////////////////////////////////////////////////
// 13. Set Adaptive Data Rate (recommended)
////////////////////////////////////////////////////////////////////////////////
// set ADR
error = LoRaWAN.setADR("on");
574 #ifdef _DEBUG
575  // Check status
576  if( error == 0 )
577  {
578     USB.println(F("13.1. Set Adaptive data rate status to on OK"));
579  }
580  else
581  {
582     USB.println(F("13.1. Set Adaptive data rate status to on error = '"));
583     USB.println(error, DEC);
584  }
585 #endif
586
587  // Get ADR
588  error = LoRaWAN.getADR();
589
590 #ifdef _DEBUG
591  // Check status
592  if( error == 0 )
593  {
594     USB.println(F("13.2. Get Adaptive data rate status OK. "));
595     USB.println(F("Adaptive data rate status: '"));
596     if (LoRaWAN._adr == true)
597     {
598         USB.println("on");
599     }
600     else
601     {
602         USB.println("off");
603     }
604  }
605  else
606  {
607     USB.println(F("13.2. Get Adaptive data rate status error = '"));
608     USB.println(error, DEC);
609  }
610 #endif
611
612 //////////////////////////////////////////////////////////
613 // 14. Set Automatic Reply
614 //////////////////////////////////////////////////////////
615
616 // set AR
617 error = LoRaWAN.setAR("on");
618
619 #ifdef _DEBUG
620  // Check status
621  if( error == 0 )
622  {
623     USB.println(F("14.1. Set automatic reply status to on OK"));
624  }
625  else
{  
    USB.print(F("14.1. Set automatic reply status to on error = "));  
    USB.println(error, DEC);  
}
#endif

// Get AR  
error = LoRaWAN.getAR();

#ifdef _DEBUG
// Check status  
if( error == 0 )  
{
    USB.print(F("14.2. Get automatic reply status OK. "));  
    USB.print(F("Automatic reply status: "));  
    if (LoRaWAN._ar == true)  
    {  
        USB.println("on");  
    }  
    else  
    {  
        USB.println("off");  
    }  
}  
else  
{
    USB.print(F("14.2. Get automatic reply status error = "));  
    USB.println(error, DEC);  
}
#endif

///////////////////////////////////////////////
// 15. Save configuration
///////////////////////////////////////////////

error = LoRaWAN.saveConfig();

#ifdef _DEBUG
// Check status  
if( error == 0 )  
{
    USB.println(F("15. Save configuration OK"));  
}  
else  
{
    USB.print(F("15. Save configuration error = "));  
    USB.println(error, DEC);  
}
#endif

USB.println(F("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
Now the LoRaWAN module is ready for
joining networks and send messages.
Please check the next examples..."));
USB.println(F("--"));

error = LoRaWAN.OFF(SOCKET);

#ifdef _DEBUG
    // Check status
    if (error == 0)
    {
        USB.println(F("4. Switch OFF OK"));
    }
    else
    {
        USB.print(F("4. Switch OFF error = ");
        USB.println(error, DEC);
    }
#endif

CO2SensingEnergyAware.pde