ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Embedded Software for Electrocardiograph Control in a Wireless Sensor Network

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Χαρίτος

Επιβλέπων:Γεώργιος Οικονομάκος

Επίκουρος Καθηγητής Ε.Μ.Π

ΑΘΗΝΑ, ΦΕΒΡΟΥΑΡΙΟΣ 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Embedded Software for Electrocardiograph Control in a Wireless Sensor Network

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Χαρίτος

Επιβλέπων:Γεώργιος Οικονομάκος

Επίκουρος Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τη 12η Φεβρουαρίου 2014.

....	....	....
Γεώργιος Οικονομάκος	Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή
Επ. Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π

ΑΘΗΝΑ, ΦΕΒΡΟΥΑΡΙΟΣ 2014

....

Χαρίτος Γεώργιος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Copyright © Γεώργιος Χ. Χαρίτος, 2014

Με επιφύλαξη παντός διακαιώματος.All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

ΠΕΡΙΛΗΨΗ

Η έξαρση και η συνεχής αύξηση των καρδιοπαθήσεων ανα τον κόσμο έχουν καταστήσει νέες μεθόδους παροχής ιατρικής περίθαλψης απαραίτητες. Η πρόσφατη τεχνολογική πρόοδος στους ασύρματους αισθητήρες και την κινητή τηλεπικοινωνία επιτρέπει νέα συστήματα περίθαλψης. Ειδικότερα, η διαθεσιμότητα μικρών, ελαφρών και εξαιρετικά χαμηλής ισχύος αισθητήρων δίνει νέες πιθανότητες για συνεχής παρακολούθηση ανθρώπινων βιοϊατρικών δεδομένων και επομένως επιτρέπει την πρόωρη ανίχνευση πιθανών παθήσεων. Επιπλέον , η ανάγκη για μείωση του μεγέθους των δεδομένων έχει οδηγήσει στη χρήση διαφόρων μεθόδων συμπίεσης. Ο σκοπός αυτής της πτυχιακής εργασίας είναι η δημιουργία ενός ενσωματωμένου λογισμικού για τον έλεγχο ηλεκτροκαρδιογραφήματος σε περιβάλλον ασύρματου δικτύου αισθητήρων. Ένσωματωμμένο λογισμικό είναι εκείνο το οποίο εγκαθίσταται μέσω ενός λειτουργικού συστήματος σε μια συσκευή καθορίζοντας ανάλογα την λειτουργία της. Ασύρματο Δίκτυο Αισθητήρων είναι ένα δίκτυο απο μικροσκοπικούς αισθητήρες(κόμβους του δικτύου) τοποθετημένους σε στρατηγικές θέσεις , σχεδιασμένο να μετραέι μεγέθη όπως θερμοκρασία,υγρασία, φωτεινότητα κτλ. Οι κόμβοι του δικτύου είναι μικροσκοπικές συσκευές οι οποίες στο εσωτερικό τους περιέχουν μικροεπεξεργαστή,πομποδέκτη,εξωτερική μνήμη,παροχή ισχύος , και ένα ή περισσότερα αισθητήρια όργανα. Ο σκοπός τους είναι να συλλέγουν πληροφορίες για το περιβάλλον τους , να επεξεργάζονται δεδομένα και να τα διανέμουν στο δίκτυο. Στην εργασία μας , το λειτουργικό σύστημα που θα χρησιμοποιήσουμε είναι το TinyOS από το Πανεπιστήμιο Berkeley της Καλιφόρνια. Το TinyOS χρησιμοποιεί τη γλώσσα nesC , μίε επέκταση της γλώσσας C . Επειδή δεν θα χρησιμοποιήσουμε hardware για , θα προσομοιώσουμε το λογισμικό με τη βοήθεια του προσομοιωτή MSPSim , προσομοιωτής του MSP430(μικροεπεξεργαστής για εφαρμογές εξαιρετικά χαμηλής ισχύος). Το λογισμικό μας θα εκτελεί τα εξής:

1)Εντοπίζει ένα ECG σήμα συλλέγοντας ένα αριθμό δειγμάτων

2)Επεξεργάζεται το σήμα με τεχνικές στο πεδίο της συχνότητας

3)Εκπέμπει το συμπιεσμένο σήμα με έναν RF πομποδέκτη , το CC2420 μικροτσίπ που περιλαμβάνει η αισθητήρια συσκευή

Τέλος , θα προσομοιώσουμε το πρόγραμμά μας με την πλατφόρμα shimmer. Στόχος μας είναι να αναλύσουμε και να αξιολογήσουμε διάφορους τρόπους συμπίεσης του ECG σήματος και να ανιχνεύσουμε εκείνες που επιτυγχάνουν μείωση της καταναλισκώμενης ισχύος. Θα καταλήξουμε σε αυτό το συμπέραμα μέσω μετρήσεων και συγκρίσεων απο τον MSPSim προσοποιωτή.

ABSTRACT

The widespread cardiovascular diseases around the world and their constant rise have rendered new ways of healthcare delivery imminent. Recent technological advances in wireless sensors and mobile communication enable new types of healthcare systems. Especially the availability of small, lightweight and ultra-low power sensors gives new possibilities for a continuously monitoring of human biomedical data and therefore allows an early detection of potential illness. Furthermore, the need for data size reduction has lead to the use of various compression techniques. The purpose of this bachelor thesis is to create an embedded software for the control of an electrocardiograph signal in a wireless sensor network. An embedded software is a software installed via an operating system on a hardware device thus designating the device's behavior accordingly. A wireless sensor network(WSN) is a network of sensor nodes (also known as motes) placed in strategic locations designated to measure temperature, light, humidity etc. Sensor nodes are tiny hardware devices that contain a microprocessor, a transceiver, an external memory , power supply and one or more sensors . Their purpose is to gather information about their environment, process data and send them throughout the network. In our project the operating system we will use to program the embedded software is the open source system TinyOS from Berkeley University of California. TinyOS uses the nesC language, an extension of C programming language. Since we won't be using any hardware for a wide variety of reasons we will simulate our software with the help of the mspsim simulator an MSP430(an ultra low-power microprocessor that most of the motes contain) simulator. Our software will perform the following:

1) Sense an ECG signal by getting a number of ECG samples.

2) Modify the signal with a Frequency-Domain Technique

3) Transceive the modified signal with an RF Transceiver the CC2420 chip that the mote contains.

Also we will simulate our software with the shimmer mote. Our goal is to analyze and evaluate different methods for compressing the ECG signal and detect the methods that succeed power consumption. We expect to reach this conclusion by gathering data from the MSPSim simulator and compare the results.

ACKNOWLEDGMENTS

First of all I would like to express my gratitude to Professor George Economakos for giving me the opportunity to write this bachelor thesis and of course for his support, patience and scientific supervision.

Furthermore, special thanks go to my colleagues and trusted partners Evangelos Simopoulos and George Teddes for their precious assistance and advice.

Finally I would like to thank my mother, father and whole family for their assistance, encouragement and loving care during my studies.

CONTENTS

Chapter 1 21
Introduction 21
1.1 Cardiovascular Diseases 21
1.2 The ECG signal 22
1.3 The Activity of the Heart 24
1.3.1 Electrocardiography 25
1.3.2 3-Lead Electrocardiography 26
1.3.3 Interpretation of an Electrocardiogram 27
1.3.4 ECG Hardware 28
1.4 ECG Data Compression 32
1.4.1 Neccesity for ECG Compression 32
1.4.2 ECG Compression Techniques 34
Chapter 2 35
Wireless Sensor Networks (WSN) 35
2.1 Introduction to WSN 35
2.2 Monitoring 35
2.3 Motes and Sensors 37
2.4 Wireless Sensor Networks 38
2.4.1 Energy Efficiency 38
2.4.2 Routing 40
2.4.3 Security 40
2.5 Operating Systems 40
Chapter 3 44
Motes 44
3.1 Intro to Motes 44
3.2 Common mote platforms 44

3.2.1 Physical Characteristics 47
3.2.2 Processor and Memory 47
3.2.3 Communications Capabilities 48
3.2.4 Sensor Support 49
3.2.5 Power Specifications 49
3.2.6 Price 50
3.3 The SHIMMER platform 50
3.3.1 Shimmer Key Principles 50
3.3.2 Shimmer Platform Design 51
3.3.3 Shimmer Key Features 52
3.3.4 Hardware Overview 54
3.3.5 Texas Instruments MSP430F1611 56
3.3.6 Texas Instruments CC2420 57
Chapter 4 60
TinyOS 60
4.1 Introduction 60
4.1.1 Networked, embedded sensors 60
4.2 TinyOS, what is it 62
4.2.1TinyOS, what it provides. 62
4.3 Example application 63
4.3.1Compiling and installing applications 64
Chapter 5 66
NesC 66
5.1 First Approach to nesC 66
5.2 Basic nesC Programming 68
5.3 Example Application:Blink 69
5.3.1 The Blink.nc Configuration 70
5.3.2 The Blink.nc Module 73

5.3.3 Compiling the Blink Application 75
5.3.4 Interfaces, Commands, and Events 76
5.4 Tasks 78
5.5 Radio communication 79
5.5.1 Basic Communications Interfaces 79
5.5.2 Active Message Interfaces 80
5.5.3 Components 80
5.5.4 Sending a Message over the Radio 81
5.5.5 Receiving a Message over the Radio 85
Chapter 6 89
The MSPSIM Simulator 89
6.1 The Simulator 89
6.1.1 Main Features 90
6.1.2 What is emulated of the MSP430 91
6.2 Sensor Board Simulation 91
Chapter 7 101
Design 101
7.1 ECG App 101
7.1.1 ECGC.nc 101
7.1.2 ECGAppC.nc 110
7.1.3 ECG.h 112
7.2 Discrete Fourier Transform(DFT) 114
7.2.1 Fast Fourier Transform(FFT) 115
7.2.2 FFT on Real-valued Data 115
7.2.3 C Code for RDFT 117
7.3 ECG-cordic 118
7.3.1 Cordic Algorithm 118
7.3.2 C code for sine cosine functions with Cordic algorithm 122

7.3.3 ECG-cordic app 123
7.4 ECG-lookup 124
7.4.1 Look-Up Table (LUT) 124
7.4.2 C code for Look-Up Table (LUT) 125
7.4.3 ECG-lookup app 126
7.5 ECG-lookup-interpolate 128
7.5.1 Linear Interpolation 128
7.5.2 C code for Linear Interpolation 129
7.5.3 ECG-lookup-interpolate app 129
Chapter 8 133
Simulation 133
8.1 Oscilloscope Simulation 133
8.2 Oscilloscope-cordic Simulation 134
8.3 Oscilloscope-lookup Simulation 135
8.4 Osciloscope lookup-interpolate Simulation 135
8.5 Power Consumption 136
8.6 Data Analysis 141
Chapter 9 146
Results , Discussion and Future Work. 146
9.1 Results-Discussion 146
9.2 Future Work 147
Chapter 10 150
Body-sensor Survey and Final Conclusions 150
10.1 Cardiac Surgery Department Survey 150
10.2 Cases of body-sensor appliance and final achievements 151
References 159

List of Figures

1.1 Graphic illustration of the factors for improved Healthcare Delivery…………………21

1.2 A typical representation of the ECG waves……………………………………………………23

1.3 Typical ECG waveform…………………………………………………………………………………23

1.4 Sectioned view of the heart……………………………………………………………………………24

1.5 Electrical conduction system of the heart……………………………………………………25

1.6 Standard 3-Lead ECG based on Einthoven's Triangle………………………………………26

1.7 Normal ECG waveform…………………………………………………………………………………27

1.8 A Holter device in ambulatory monitoring……………………………………………………29

1.9 Toumaz’s Sensium Life Pebble TZ203082………………………………………………………29

1.10 SHIMMER platform , a small wireless sensor platform that can record and transmit physiological and kinematic data in real-time…………………………………………30

1.11 IMEC’s wireless single-lead bipolar ECG patch…………………………………………30

2.1 Passive and Active monitoring………………………………………………………………………36

2.2 Volcano WSN…………………………………………………………………………………………………36

2.3 Mote picture…………………………………………………………………………………………………37

2.4 Mote architecture…………………………………………………………………………………………38

2.5 Heliomote………………………………………………………………………………………………………39

2.6 TinyOS Code snippet……………………………………………………………………………………41

2.7 Avrora………42

2.8 Cooja………42

3.1 TelosB/Tmote Sky…………………………………………………………………………………………45

3.2 Micaz………45

3.3 Shimmer………………………………………………………………………………………………………46

3.4 IRIS……………………………………………………………………………………………………46

3.5 Block diagram from memsic…………………………………………………………………………52

3.6 MSP430 Block diagram from moteiv………………………………………………………………55

3.7 CC2420 Simplified block diagram…………………………………………………………………57

3.8 CC2420 Simplified block diagram…………………………………………………………………58

4.1 A typical sensor network architecture……………………………………………………………61

4.2 Example application architecture…………………………………………………………………64

6.1a Sky window……………………………………………………………………………………96

6.1b USART 1Port Output window………………………………………………………………96

6.1c Duty Cycle Monitor window…………………………………………………………………97

6.1d Stack Monitor window…………………………………………………………………………97

6.1e MSPSimMonitor window………………………………………………………………………98

6.2 MSPSim terminal command line……………………………………………………………99

7.1 Rotation of a vector V by angle ϕ……………………………………………………………119

7.2 The yellow line is the LUT sin(x), and the blue one is LUT sin(x+1)…………………..128

8.1 Duty Cycle graph for the Oscilloscope app…………………………………………………..134

8.2 Duty Cycle graph for the Oscilloscope-cordic app………………………………………......134

8.3 Duty Cycle graph for the Oscilloscope-lookup app…………………………………………..135

8.4 Duty Cycle graph for the Oscilloscope-lookup-interpolate app………………………..135

8.5a: Power Score graph for 8-point DFT………………………………………………………140

8.5b: Power Score graph for 16-point DFT………………………………………………………141

9.1 Mobile Healthcare system context……………………………………………………………….148

List of Tables

1.1 Heart chambers viewed by the 3-lead ECG ……………………………………………………27

1.2 Segments and intervals of an ECG wave……………………………………………………28

3.1 Physical characteristics of motes………………………………………………………………47

3.2 Mote microprocessor specifications……………………………………………………………48

3.3a Mote Communication capabilities……………………………………………………………..48

3.3b Mote Communication capabilities……………………………………………………………..49

3.4 Mote prices……………………………………………………………………………………………50

3.5 Overview of Shimmer extension daugtherboards…………..……………………………52

3.6 CC2420 main features………………………………………………………………………………59

6.1 Function of Blink application………………………………………………………………………92

7.1 Cordic Uses………………………………………………………………………………………….121

8.1a 8-point Raw duty values (10 per second - command used = duty 10 "MSP430 Core.active") for 3 seconds (30 values) - peak-to-peak…………………………………………136

8.1b 16-point Raw duty values (10 per second - command used = duty 10 "MSP430 Core.active") for 3 seconds (30 values) - peak-to-peak…………………………………………137

8.1c 8-Point ROM(code) bytes,RAM(data) bytes, Average Duty Value and Maximum Duty Value table…………………………………………………………………………………………138

8.1d 16-Point ROM(code) bytes,RAM(data) bytes, Average Duty Value and Maximum Duty Value table…………………………………………………………………………………………..138

8.2a Power Score Table for 8-point DFT…………………………………………………………139

8.2b Power Score Table for 16-point DFT………………………………………………………..140

8.3 8-Point No Compression to Compression gain…………………………………………142

8.4 16-Point No Compression to Compression gain…………………………………………..143

8.5 16-Point to 8-Point Gain………………………………………………………………………………144

10.1 Patient Data………………………………………………………………………………………..151

Chapter 1

Introduction

1.1 Cardiovascular Diseases

Nowadays, our modern society is threatened by an incipient health care delivery crisis caused by the current demo-graphic and lifestyle trends. On the one hand, the world's population is fast aging resulting into an increased rate of cardiac disorders. On the other hand, our busy and often unhealthy lifestyles are gradually increasing the number of people unsuspectingly developing or living with chronic cardiovascular conditions for decades. Specifically, according to the World Health Organization, cardiovascular diseases are the number one cause of death worldwide, responsible for an estimated 17.1 million deaths in 2012 (i.e., 29% of all deaths worldwide) and economic fallout in billions of dollars. Their burden is only expected to rise due to the rapid aging of the world population and of course the increasing prevalence of unhealthy lifestyles. These increasingly existent cardiac diseases are requiring escalating levels of supervision and medical management, which are contributing to unmatchable health care costs .

Figure 1.1 *Graphic illustration of the factors for improved Healthcare Delivery*

Cardiovascular diseases require close and potentially continuous medical supervision and care. Soon they will require healthcare costs and medical management needs that are unbearable for traditional healthcare delivery systems.

Wireless body sensor network (WBSN) technologies promise to offer large-scale and cost-effective solutions to this problem. The use of wearable, miniaturized, and wireless sensors, able to continuously measure and wirelessly report cardiac signals, can definitely provide the ubiquitous, long-term and even real-time monitoring required by the patients, as well as its integration with the patient's medical record and its coordination with nursing/medical support .These solutions scope to outfitting patients with wearable, miniaturized and wireless sensors able to measure and wirelessly transmit and report cardiac signals to telehealth providers. They are ready to enable the required personalized, real-time and long-term ambulatory monitoring of chronic patients, its seamless integration with the patient's medical record and its coordination with nursing/medical support.

1.2 The ECG signal

An electrocardiogram (ECG or EKG) is a recording of the electrical activity of the heart over time produced by an electrocardiograph. The signal recorded, is graphically displayed in a two dimensional graph, where the height represents the measured electrical activity in millivolts and the width the interval of time in seconds. Electrical impulses in the heart originate in the sinoatrial node and travel through the heart muscle where they impart electrical initiation of systole or contraction of the heart. The electrical waves can be measured at selectively placed electrodes(electrical contacts) on the skin. Electrodes on different sides of the heart measure the activity of different parts of the heart muscle. An ECG displays the voltage between pairs of these electrodes, and the muscle activity that they measure, from different directions, also understood as vectors. . Figure 1.2 shows a typical ECG waveform. Among the relevant cardiac signals, the noninvasive electrocardiogram has long been used as a means to diagnose diseases reflected by disturbances of the heart's electrical activity. Beyond traditional electrocardiography, the automated processing and analysis of the ECG signal has been a popular subject of research and has witnessed substantial advances .In fact , a huge variety of algorithms have been suggested for the detection of the ECG characteristic waves, so-called ECG delineation, following a variety of approaches based on low-pass differentiation , the wavelet transform (WT) , dynamic time warping , artificial neural networks , hidden Markov models , or morphological transforms. The morphological and timing information of the detected waves, namely the QRS complex and P and T waves, can be used to diagnose many cardiac ailments .

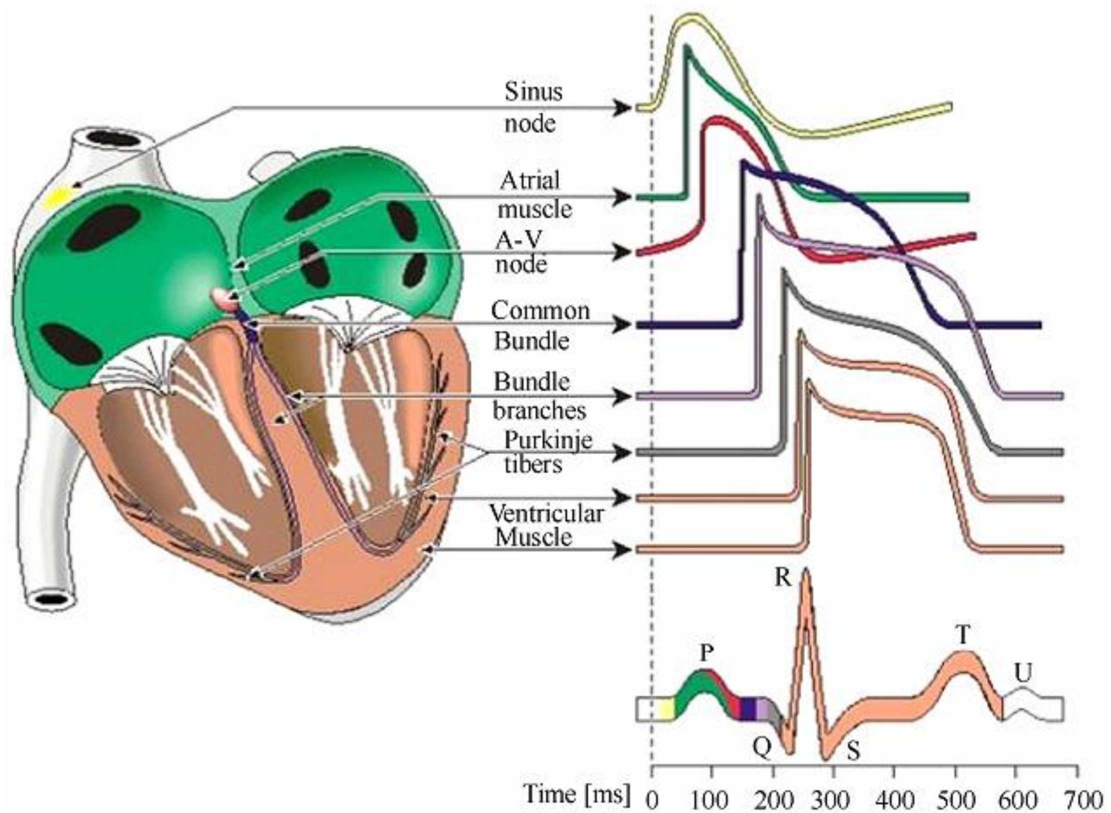

Figure 1.2 *A typical representation of the ECG waves*

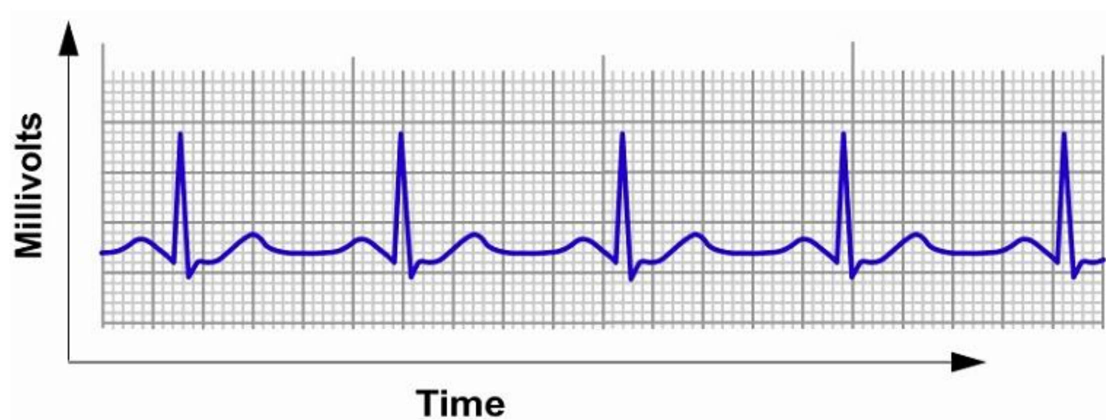

Figure 1.3 *Typical ECG waveform*

To understand how and what an ECG records, we must first understand and grasp how the heart itself works . For this reason , we describe the basic functionality of the heart, before describing the ECG waveform and its interpretation in detail.

1.3 The Activity of the Heart

The heart is nothing more than muscle with the sole purpose of pumping blood throughout our whole body. For this, the heart consists of various chambers: a right atrium, right ventricle, left atrium and left ventricle, as shown in Figure 1.4 . These two sides of the heart work together in perfect synchronicity to pump blood through our body system. The right side of the heart delivers deoxygenated blood from the body to the lungs, whereas the left side of the heart delivers oxygenated blood from the lungs to the body .

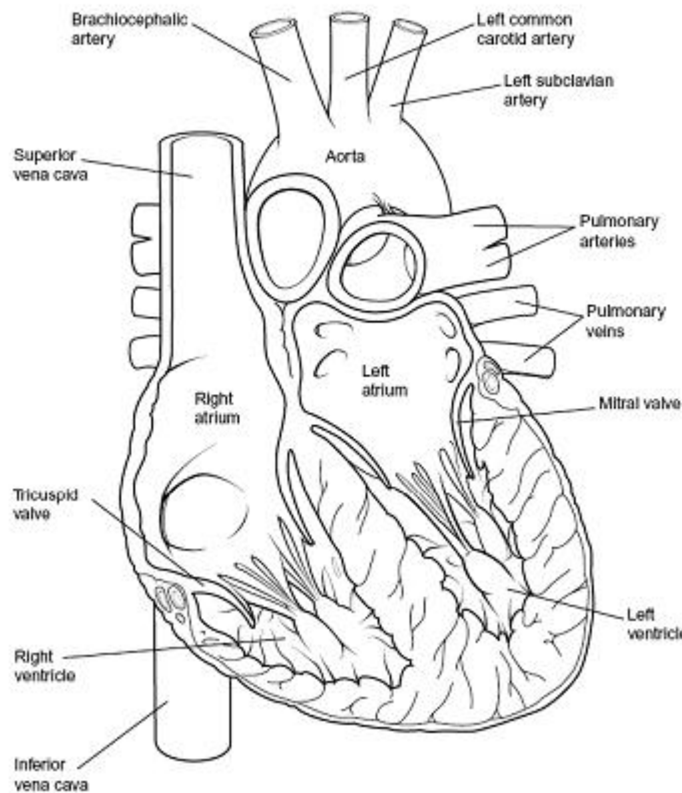

Figure 1.4 *Sectioned view of the heart*

A pumping cycle begins, when deoxygenated blood from the body returns to fill the right atrium of the heart. When the right atrium is full, it contracts and pushes the blood into the right ventricle. Now, when the right ventricle is filled, it contracts and pumps the blood further into the lungs. The oxygenated blood from the lungs is then returned to the left atrium of the heart. The left atrium contracts and pumps the blood into the left ventricle. This occurs simultaneously as a new contraction is taking place in the right atrium, filling the right ventricle with blood. Finally the left ventricle contracts and sends the blood to the rest of the body system. At the same time the right ventricle pumps blood into the lungs. After the contraction of the ventricles, a new cycle begins.

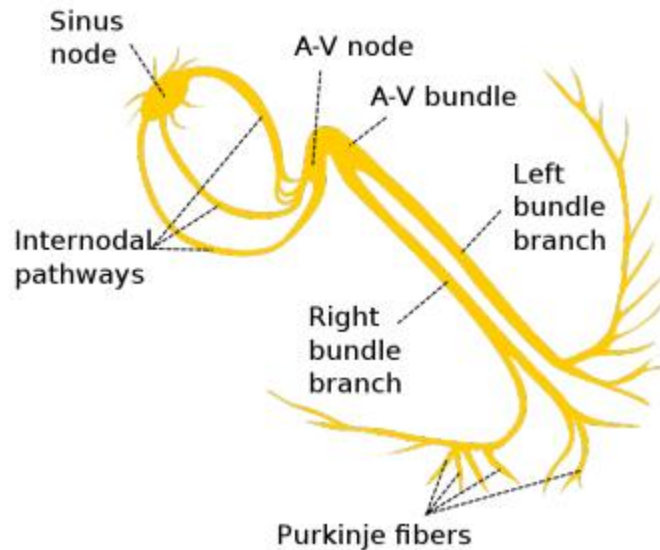

Figure 1.5 *Electrical conduction system of the heart*

1.3.1 *Electrocardiography*

In order the heart to contract , an electrical impulse is required. This electrical impulse is generated by the Sinoatrial Node (SA node) and propagates through the conduction system of the heart, shown in Figure 1.5 . First the electrical impulse spreads from the SA node throughout the muscle tissue of the right and left atrium and follows an internodal pathway directly to the atrioventricular node (AV node), where it is naturally delayed. This stimulates the artrial muscle cells, causing both atria to contract in unison and fill the ventricles with blood. Especially the delay, caused by the AV node is important, because it allows to fill the ventricle with blood by atrial contraction ,before the ventricles contract itself. After the AV node, the impulse continuous down to the Bundle of His, where the conduction system branches out to a right and left bundle branch and finally terminates in tiny fibres, known as Purkinje Fibres. This Purkinje Fibres conduct the electrical impulse throughout the ventricular, stimulating the muscle cells and causing a contraction of the right and left ventricles in unison. At the beginning of this cycle a resting heart is polarized, which means that the heart cells have a negative charge . As a stimulus occurs, the cells changes its charge to positive. This is called depolarization and causes the heart muscle fibres to shorten and consequently the heart muscle to contract. During this contraction the cells regains slowly a negative charge, as the electrical impulse is moving down along the conduction system, causing the heart muscle to relax. This process is known as repolarization. The potential change, which occurs during depolarization and repolarisation, is exactly what can be measured at the skin surface by electrodes . This recorded electrical activity can then be displayed in a two dimensional graph known as electrocardiogram.

1.3.2 *3-Lead Electrocardiography*

As mentioned, the ECG works by detecting the electrical changes on the skin, caused when the heart muscle depolarises. This is done by placing pairs of electrodes on either side of the heart. The output of a pair of electrodes is known as lead and is said to look at the heart from a specific perspective . These leads are also called bipolar leads, as they measure the voltage difference between two electrodes . Based on the number of leads recorded, several types of ECG's are differentiated. For example 3-lead ECG, 5-lead ECG and 12-lead ECG. These types of ECGs mainly di fferentiate from each other by the precision and accuracy of their recordings. A 12-lead ECG for example records more leads than a 3-lead ECG and therefore has a broader view on the heart. Consecutively the 3-lead ECG will be described in more detail, as it was used in this project. The 3-lead ECG is based on the most basic form of electrodes placement, known as Einthoven's triangle . Thereby the electrodes are placed as follow: one on the right arm (RA), one on the left arm (LA) and the third one representing the left leg (LL) is situated below the hearts apex. The electrodes then form the leads: LA + RA (Lead I), LA + LL (Lead II) and RA + LL (Lead III). Figure 1.6 illustrates the placement of the electrodes and the three leads they form . The leads are described by convention as follows :

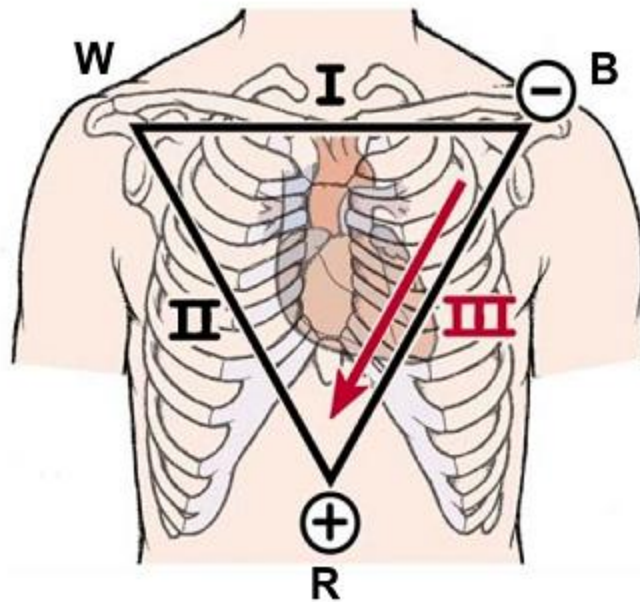

Figure 1.6 *Standard 3-Lead ECG based on Einthoven's Triangle*

- Lead 1 (LA-RA): measures the potential difference between the right arm electrode and the left arm electrode.

- Lead 2 (LA-LL): measures the potential difference between the right arm electrode and left leg electrode.
- Lead 3 (RA-LL): measures the potential difference between the left arm electrode and left leg electrode.

This 3-lead system provides three different views, able to monitor multiple regions of the heart and consequently yields to three different signals. Table 2.1 summarizes the chambers, viewed by the 3-lead ECG.

Lead	Views	Heart Chambers
Lead 1	Lateral	Left ventricle,left atrium
Lead 2	Inferior	Left and right ventricle
Lead 3	Inferior	Right and Left ventricle

Table 1.1 *Heart chambers viewed by the 3-lead ECG*

1.3.3 *Interpretation of an Electrocardiogram*

The recorded ECG signal shows a series of waves, that relate to the electrical impulses ,which occur during each beat of the heart. These waves are labeled with successive letters of the alphabet P, Q, R, S, T, and U, as shown in Figure 1.7. When it comes to the interpretation of an ECG signal, the attention turns to the segment and intervals, also illustrated in Figure 1.7 . The most important intervals and segments, are described and summarized in table 1.2 .

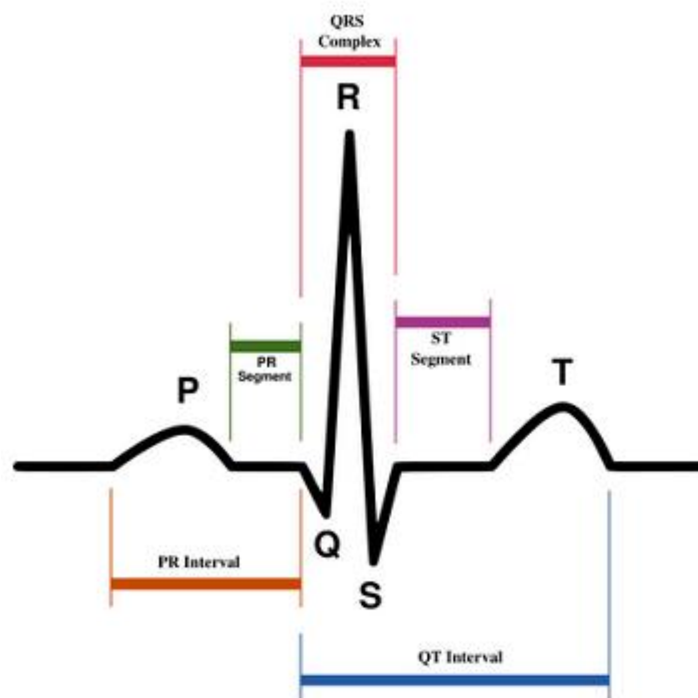

Figure 1.7 *Normal ECG waveform*

Feature	Description
RR Interval	Represents the interval between an R wave an the next R wave and is the inverse of the heart rate. A normal resting heart rate lies between 50 and 100 beats per minute.
P waves	The P wave represents the depolarization of the right and left atria.
PR segment	The PR segment coincides with the electrical conduction from the AV node through the Bundle of His, toward the Purkinje Fibers. Hence the PR segment corresponds to the time between the end of 28rterial depolarization, to the begin of ventricular depolarization.
PR interval	Represents the time measured between the beginning of the P wave to the beginning of the QRS complex. The PR interval reflects the time an electrical impulse takes to travel from the SA node through the atria and the AV node down to the Purkinje Fibres.
QRS Complex	The QRS complex represents the rapid depolarization of the two ventricles. Because the ventricles have a larger muscle mass than the atria, the QRS complex has a much larger amplitude than the P-wave.
ST Segment	The ST segment lies between the QRS complex and T wave and represents the period, when the ventricles are depolarized.
T wave	The T wave corresponds to the rapid ventricular repolarization.
QT interval	The QT interval represents the complete ventricular cycle, starting with the depolarization and ending with the repolarization.

Table 1.2:*Segments and intervals of an ECG wave*

Based on the rythm of the recorded ECG and the patterns of the segments and intervals in a ECG waveform, abnormalities can be detected and a diagnose can be given.

1.3.4 *ECG Hardware*

Traditionally, the automatic analysis of ECG signals, including delineation, was either taking place online on bulky, high-performance bedside cardiac monitors, or performed offline during a post-processing stage after ambulatory ECG recording using wearable, yet obtrusive, ECG data loggers . While the resting ECG monitoring is standard practice in hospitals, its ambulatory counterpart is still facing many technical challenges. For instance, the three-lead ECG is still nowadays recorded on a rather bulky commercial data-logging (Holter) device during one to five days of normal daily activities of a patient. These systems suffer from important limitations: limited autonomy, bulkiness, and no or limited wireless connectivity.

Figure 1.8 *A Holter device in ambulatory monitoring*

Recently, the realization of wireless-enabled low-power ECG monitors for ambulatory use has received significant industrial and academic interest . Effort has been dedicated to online automatic ECG analysis on miniature, wearable and wireless ECG monitors as an enabler of next-generation mobile cardiology systems .The most important highlights of these research and development efforts are:

1)Toumaz's Sensium Life Pebble TZ203082 , an ultra-small and ultra-low-power monitor for heart rate, physical activity, and skin temperature measurements with a reported autonomy of five days on a hearing aid battery.

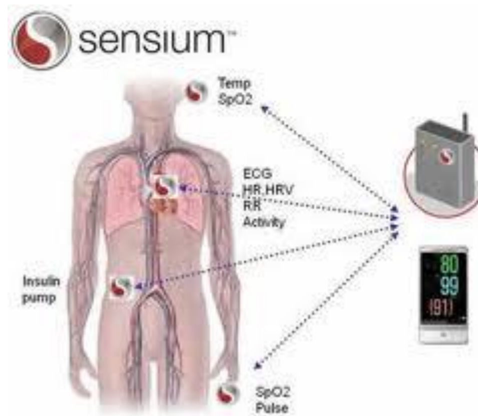

Figure 1.9 *Toumaz's Sensium Life Pebble TZ203082*

2) Intel's Shimmer , a small wireless wearable sensor platform able to record and wirelessly transmit three-lead ECG data as well as accelerometer, gyroscope, and galvanic skin response information.

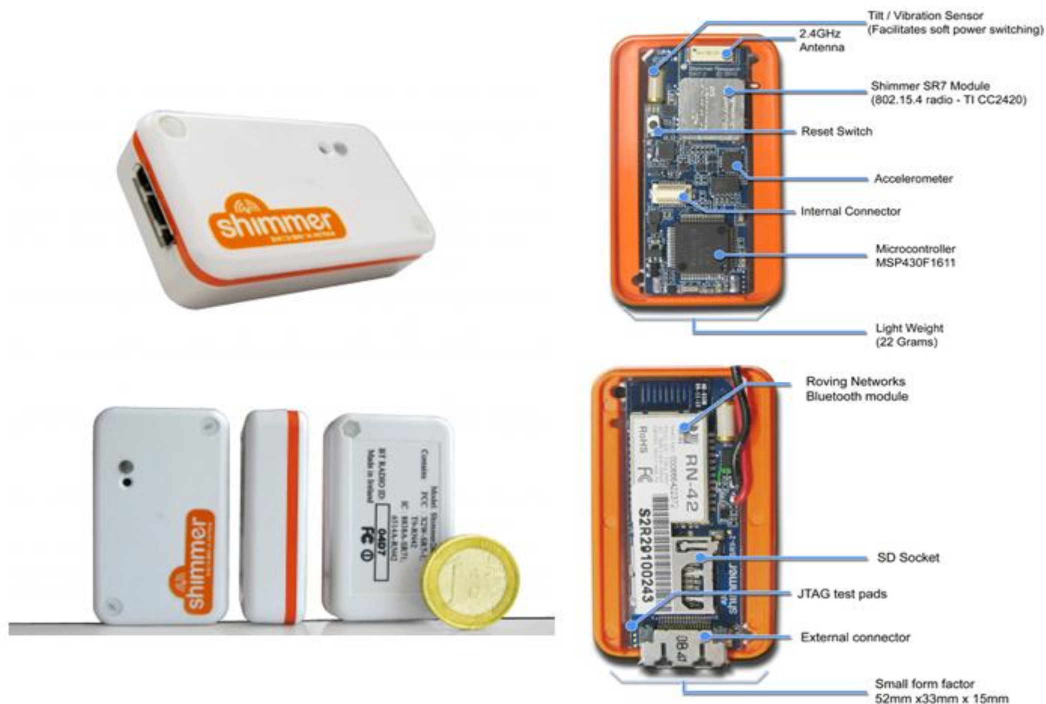

Figure 1.10 *SHIMMER platform , a small wireless sensor platform that can record and transmit physiological and kinematic data in real-time*

3) IMEC's wireless single-lead bipolar ECG patch for ambulatory monitoring claiming over ten days of monitoring on a 160mAh Li-ion battery (for undisclosed use conditions).

2. IMEC's wireless battery-operated electrocardiogram (ECG) patch monitor integrates electrodes, a biochip sensor, an MCU, and a radio in a package about the size of a very thin wristwatch. Algorithms running on the patch's processor monitor a patient's arrhythmias day and night.

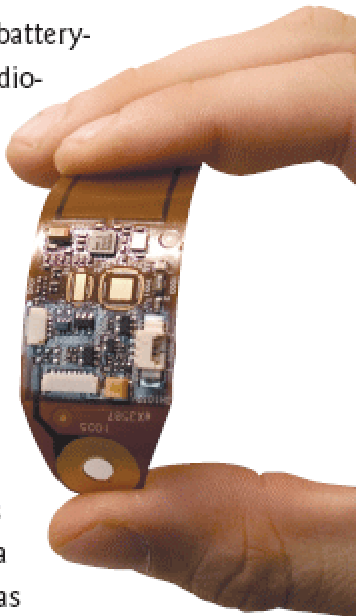

Figure 1.11 *IMEC's wireless single-lead bipolar ECG patch*

The clinical relevance of the first system is still being validated, as Toumaz aims to achieve more than the system's so far established accurate measurement of heart rate. The second system, which is based on commercial off-the-shelf components such as the TI MSP430 microcontroller and the CC2420 radio chip-set, operates on a Li-ion battery that provides about 1 Wh of energy. According to measurements, it is able to support a maximum of 6.5-day single-lead raw ECG sensing and storage on local memory. This autonomy figure is reduced by 25%, when the raw ECG data are wirelessly streamed using the ultra-low-power CC2420 in a perfect point-to-point link with no wireless protocol overhead. More importantly, this autonomy figure will undoubtedly dramatically decrease under realistic ambulatory monitoring. Finally, IMEC ultra-low-power wireless biopotential sensor node achieves its enhanced autonomy due to a proprietary customized ultra-low-power analog read-out ASIC [signal acquisition, amplification, and analog-to-digital conversion (ADC)], a proprietary ultra-low-power ultra-wideband wireless transceiver, and more importantly, dedicated signal processors to preprocess and compress the sensed data using state-of-the-art techniques, in order to reduce the airtime over power-hungry wireless links .Based on these premises, it is today acknowledged that the achievement of truly WBSN-enabled ambulatory monitoring systems requires more breakthroughs not only in terms of ultra-low-power read-out electronics and radios, but also and increasingly so, in terms of ultra-low-power dedicated digital processors and associated embedded feature extraction and data compression algorithms.

Wireless body sensor networks (WBSN) hold the promise to be a key enabling information and communications technology for next-generation patient-centric telecardiology or mobile cardiology solutions. Through enabling continuous remote cardiac monitoring, they have the potential to achieve improved personalization and quality of care, increased ability of prevention and early diagnosis, and enhanced patient autonomy, mobility, and safety. However, state-of-the-art WBSN-enabled ECG monitors still fall short of the required functionality, miniaturization, and energy efficiency. Among others, energy efficiency can be improved through embedded ECG compression, in order to reduce airtime over energy-hungry wireless links.

In our project we will use the shimmer platform . In fact we will run-simulate our program on shimmer platform . And that's because we require a sensor component that satisfies the requirements specified below:

- ✓ *Record ECG data* : The sensor must record raw ECG data in real-time.
- ✓ *Transmit ECG data* : The sensor transmits the recorded data wirelessly to the mobile application.
- ✓ *Remote control of sensor* : It should be possible to start and stop the recording of data, by sending corresponding commands wirelessly to the sensor.
- ✓ *Use of defined protocol* : A defined protocol should be used to send recorded data. Further the protocol should support the transmission of different types of biomedical data.

The shimmer mote meets all these requirements . In Chapter 3 we present the shimmer platform's main features in detail.

1.4 ECG Data Compression

1.4.1 *Neccesity for ECG Compression*

ECG compression is necessary for efficient storage and transmission of the digitized ECG signals. Any kind of ECG monitoring device generates a huge amount of data in the continuous long-term (24-48 hours) ambulatory monitoring tasks. In order to succeed good diagnostic quality, up to 12 different streams of data may be obtained from sensors placed on the patient's body. The sampling rates of ECG signals are from 125Hz to 500Hz, and each data sample may be digitized into 8 to 12 bits binary number. Even with one sensor at the lowest sampling rate of 125 Hz and 8-bit encoding, it generates data at a rate of 7.5KB per minute and 450KB per hour. For a sampling rate of 500Hz and 12-bit encoding recording, it generates data at a rate of 540KB per minute and 30MB per hour. The data rate from 12 different sensors totally will generate 12 times the amount of data and it is enormously big. Besides, recording for almost 24 hours may be of paramount importance for a patient with irregular heart rhythms . Monitor devices such as Holter must have a memory capacity of about 400-800 MB for a 12-lead recording, but such a big memory cost may render a solid-state commercial Holter device impossible. Thus, efficient ECG data compression to dramatically reduce the data storage capacity is a necessary solution. On the other hand, it makes possible to transmit ECG data over a telephone line from one cardiac doctor to another cardiac doctor to get opinions.

Coding is useful because it helps reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. On the downside , compressed data must be decompressed to be used and this extra processing may be detrimental to some applications. Basically, as we previewsly mentioned , a data coding algorithm seeks to minimize the number of code bits stored by reducing the redundancy present in the original signal. The design of data compression schemes therefore involves trade-offs among various factors including the degree of compression, the amount of distortion introduced (if using a lossy compression scheme) and the computational resources required to compress and uncompress the data. The most difficult part for any efficient ECG compression technique is to reduce the amount of data as much as possible while preserving the clinical significant signal for cardiac diagnosis, for analysis of ECG signal for various parameters such as heart rate, QRS-width, etc. Then the various parameters and the compressed signal can be transmitted with less channel capacity. Compression connotes the process of starting with a source of data in digital form (usually either a data stream or a stored file) and creating a representation that uses fewer bits than the original . An effective data compression scheme for ECG signal is required in many practical applications such as ECG data storage, ambulatory recording systems and ECG data transmission over telephone line or digital telecommunication network for telemedicine .

Data compression methods can be classified into two categories: 1) Lossless and 2) Lossy coding methods. Lossy compression is useful where a certain amount of error is acceptable for increased compression performance. Lossless or information preserving compression is used primarily in the storage of medical or legal records. In lossless data compression, the signal samples are considered to be realizations of a random variable or a random process and the entropy of the source signal determines the lowest compression ratio that can be achieved. In lossless coding the original signal can be perfectly reconstructed. For typical biomedical signals lossless (reversible) compression methods can only achieve Compression Ratios (CR) in the order of 2 to 1. On the other hand lossy (irreversible) techniques may produce CR results in the order of 10 to 1. In lossy methods, there is some kind of quantization of the input data which leads to higher CR results at the expense of reversibility. But this may be acceptable as long as no clinically significant degradation is introduced to the encoded signal. The CR levels of 2 to 1 are too low for most practical applications. Therefore, lossy coding methods which introduce small reconstruction errors are preferred in practice.

1.4.2 *ECG Compression Techniques*

Biomedical signals can be compressed in time domain, frequency domain, or time-frequency domain. ECG data compression algorithms have been mainly classified into three major categories : 1) Direct time-domain techniques, e.g., turning point (TP), amplitude-zone-time epoch coding (AZTEC) , coordinate reduction time encoding system (CORTES) and Fan algorithm. 2) Transformational approaches , e.g., discrete cosines trans-formation (DCT), fast fourier transform (FFT), discrete sine transform (DST), wavelet transform (WT) etc. 3) Parameter extraction techniques, e.g., Prediction and Vector Quantization (VQ) methods . The time domain techniques which are based on direct methods were the earlier approaches to biomedical signal compression. Transform Coding (TC) is the most important frequency-domain digital waveform compression method .When we compare these methods we find that direct data compression is a time domain compression algorithm which directly analyses samples where inter-beat and, intra-beat correlation is exploited. These algorithms suffer from sensitiveness to sampling rate, quantization levels and high frequency interference. It fails to achieve high data rate along with preservation of clinical information. In Transform based technique compressions are accomplished by applying an invertible orthogonal transform to the signal. Due to its decorrelation and energy compaction properties the transform based methods achieve better compression ratios. In transform coding, knowledge of the application is used to choose information to discard, thereby lowering its bandwidth .The remaining information can then be compressed via a variety of methods. When the output is decoded, the result may not be identical to the original input, but is expected to be close enough for the purpose of the application. In parameter extraction methods a set of model parameters/features are extracted from the original signal(model based) which involves methods like Linear term prediction (LTP) and analysis by synthesis.

Chapter 2

Wireless Sensor Networks (WSN)

2.1 Introduction to WSN

In times where technology evolves on a daily basis, new inexpensive solutions can be created. These new tools overcome the existing ones by requiring less work while achieving better results and offering more functionality. These new technologies also allow the conception of new tools to previously unsolvable problems.

Wireless Sensor Networks (WSN) are wireless networks formed by small low cost autonomous devices called motes, with the ability to sense the surrounding environment. An extension to WSN that adds the ability to act besides sensing over the environment is called Wireless Sensor and Actor Networks (WSAN). Both WSAN and WSN are possible solutions for several problems. Their main characteristics are easy deployment and low cost, while having the ability to sense and act without human intervention makes their usage highly attractive in many applications. They are being adopted in several fields of work. Some examples include: creating effective irrigation systems, fire alarms, structure health monitoring and medical or military applications.

2.2 Monitoring

Monitoring can be defined as the act of continuously observing something. It generally means to be aware of the state of a system. Environmental monitoring describes the processes and activities that need to take place to characterise and monitor the quality of the environment.

Figure 2.1: *Passive and Active monitoring*

When we refer to monitoring we can differentiate two types: active and passive. The difference between these two types is that while active monitoring necessarily involves human presence, being performed through field visits to the monitored environment, passive monitoring is done by autonomous systems not requiring human intervention. In this case, the monitoring system is placed in the environment, automatically acquiring data and either storing it locally for later retrieval or sending it to a remote system.

Several applications of WSNs in monitoring exist such as animal monitoring used by biologists to study animals in the wild, structure health monitoring used to ensure buildings or bridges condition, volcano monitoring used to study the seismic activity of volcanic areas and obviously forest monitoring mainly used for forest fire detection.

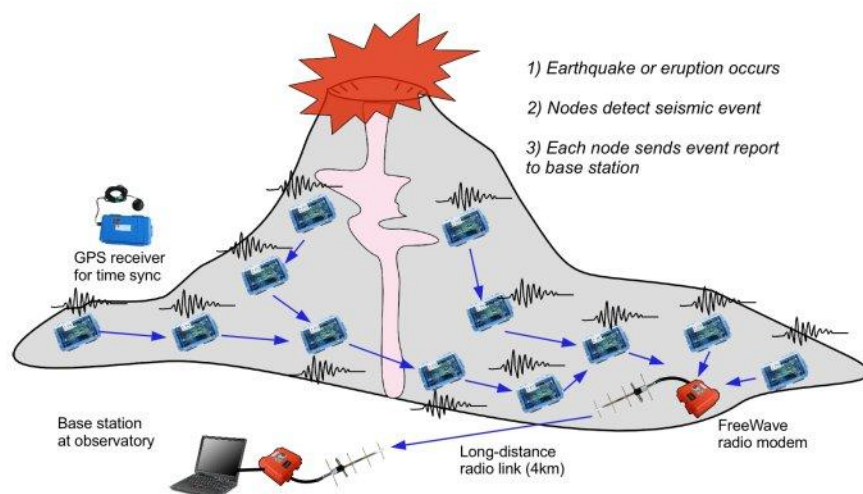

Figure 2.2: *Volcano WSN*

2.3 Motes and Sensors

We will cover the motes in more detail in Chapter 3 but an introduction here is necessary, as they are the basic building block of a WSN. A sensor node (also known as mote) may be described as a small low-cost device with the ability to perform some processing, gather sensory information and communicate with other connected nodes in the network. A mote is a node but a node is not always a mote. Its main components are a microcontroller, transceiver, external memory, power source and one or more sensors. A typical mote can be seen in the picture below.

Figure 2.3: *Mote picture*

The microcontroller and memory provide computational power and storage space respectively, while the power source – usually a battery – provides energy supply to the mote, making it autonomous. The mote captures data through the acquisition system composed of a set of sensors. These may be embedded directly in the mote or a separate sensor board connected to the mote via its I/O ports. Sensors of any type (e.g. temperature, humidity, light, acceleration etc.) can be connected depending on the type of data we intend to capture. Using a transceiver, the communication module allows data to be wirelessly transmitted and received between nodes. The typical architecture of a mote is depicted below. Again, more on this in Chapter 3.

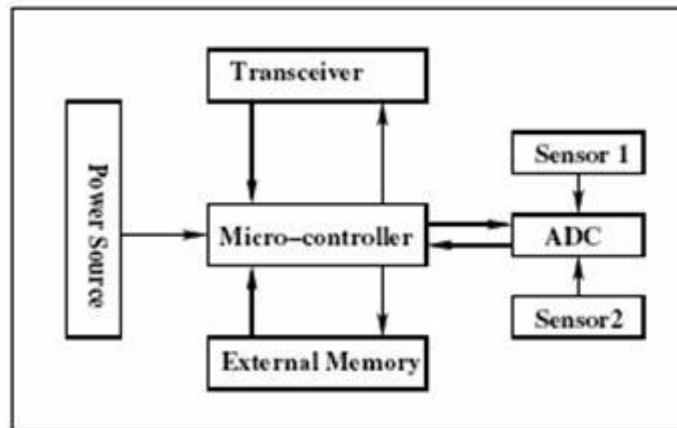

Figure 2.4: *Mote architecture*

2.4 Wireless Sensor Networks

Wireless sensor networks (WSN) are wireless networks formed by motes. The wireless and routing technologies in motes allow them to be deployed creating a WSN, where each node may capture environmental information and share it with all other motes. The system's cost can be highly reduced by avoiding cabling and instead use wireless technology. This also allows both a more flexible deployment and lower maintenance costs.

WSNs intend to provide a low cost solution to problems such as monitoring large areas, difficult access or hazardous environments. These networks can replace expensive active monitoring with cost effective passive monitoring. It is possible to set the motes to capture data for a certain period of time and transmit it to be stored in a central node called sink, where a person could be in order to access and monitor the captured information. The biggest challenges that WSNs designers are faced with nowadays are energy efficiency, routing and security. They are presented in more detail below.

2.4.1 *Energy Efficiency*

Energy management and consumption are critical challenges for WSNs as motes both require energy to operate each of their composing parts and being autonomous. The main objective of studies conducted in this field is to maximize the motes' lifetime. All motes' components require a certain amount of energy to operate even when it comes to small amounts. The connection of motes to a power

source such as a power socket, implies the use of cables, thus nullifying the benefits of wireless technology.

Most motes nowadays are battery powered, allowing them to be autonomous and wireless but also limiting their lifetime. What WSN designers can do to maximize a mote's lifetime is to minimize its hardware energy consumption. The power usage can be reduced by putting motes into sleep mode - a state where all mote's activity is stopped and all of its composing parts are switched off - or even by putting a single component to sleep when not in use (e.g. switch off the radio transceiver), thus reducing its duty cycle - the percentage of time during which a device is working.

Figure 2.5: *Heliomote*

Research is being done to find alternative or complementary power sources to batteries. Environmental energy harvesting methods are being studied as they allow the mote to collect energy from the environment. Two of the aforementioned methods include solar cells, that allow the conversion of sunlight to electricity through solar panels, and piezoelectric ceramic materials that convert environment vibrations to electricity. The use of energy harvesting techniques turns everlasting mote lifetime into a possibility. Some commercially available products already exist, such as the Heliomote.

2.4.2 *Routing*

Routing collected information between sensor nodes in WSNs presents several challenges. The different kinds of network topologies and their requirement for different routing protocols, the possibility that nodes are randomly deployed or large in quantity are some of the faced problems. Energy and computation constraints also impose new requirements to routing algorithms. A system failure or power shortage may turn off nodes, requiring new routes to be calculated so as to maintain network connectivity between the rest operating nodes.

Requirements such as low energy and memory consumption mean limited routing tables and new algorithms. Several routing protocols have been specifically designed for WSNs in order to appropriately fulfill these special needs. The existing routing protocols are categorized according to the network structure in which they operate and the protocol operation. Depending on the network structure they can be classified as flat, hierarchical or location-based routing. Depending on their operation they can be multipath-based, query-based, negotiation-based, QoS -based or coherent-based.

2.4.3 *Security*

The use of wireless technology in WSNs has numerous benefits but it also introduces several security threats that need to be considered. Motes' characteristics of limited computing power and low energy resources represent a challenge in producing an effective security solution.

Attacks against WSNs are divided into two types: attacks against the security mechanisms and against basic mechanisms. Some of the common WSN attacks are denial of service (DoS), attacks on information in transit, blackhole/sinkhole attacks, hello flood attacks or wormhole attacks. Most of those are caused when a malicious node sends false information to other nodes thus compromising the system. Detecting mechanisms to solve these attacks are still being developed.

2.5 Operating Systems

Due to specific requirements and constraints of sensor nodes and wireless sensor networks, operating systems have been created specifically targeting embedded

platforms, their needs and objectives. Reconfiguration, energy awareness and optimization, self-configuration, multi-hop communications, memory and computation power constraints, are some of the requirements these operating systems need to address.

Some of the most popular operating systems used, are Nano-RK developed at Carnegie Mellon University, SOS developed at University of California Los Angeles, MANTIS developed at the University of Colorado, BTNut developed at ETH Zurich, Contiki at Swedish Institute of Computer Science and, the most widely used, as well as the one that will be used in this thesis, TinyOS created at the University of California Berkeley.

```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
}
implementation {
  uint16_t counter = 0;

  event void Boot.booted() {
    call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
  }

  event void Timer0.fired() {
    counter++;
    call Leds.set(counter);
  }
}
```

Figure 2.6: *TinyOS Code snippet*

TinyOS is an open source operating system featuring a component-based architecture minimizing memory usage and providing an event-driven execution model allowing fine-grained power management and scheduling flexibility. Software programs developed in TinyOS are programmed using nesC, an extension to the C programming language. We will examine both TinyOS and nesC, in depth, in Chapters 4 and 5.

Simulators are software platforms specifically designed to simulate a WSN's or even a single mote's behavior. These platforms allow testing a developed program without having to install the software in the actual motes or (as in our case) without even having any physical sensor node. Simulators are immensely time-saving when we need to know the characteristics and operational parameters of a WSN involving hundreds or thousands of motes, prior to its installation.

```
   0      72595584  <==== 00.00.00.0F.A7.0F.41.88.22.22.00.FF.FF.01.00.3F.06.00.23.CE.BB  0.660 ms
   1      73169483  on  off on
   1      73169491  on  on  on
   1      73169500  off on  on
   1      73355979  ----> 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.62.C1  0.660 ms
   0      73356225  <==== 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.CE.B3  0.660 ms
   2      73356225  <==== 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.CE.B3  0.660 ms
==============================================================================
Simulated time: 73728000 cycles
Time for simulation: 4.349 seconds
Total throughput: 50.85859 mhz
Throughput per node: 16.952862 mhz
=={ Packet monitor results }==================================================
Node     sent (b/p)          recv (b/p)    corrupted (b)   lostinMiddle(p)
------------------------------------------------------------------------------
   0       756 / 36           1491 / 71           0        0
   1       756 / 36           1491 / 71           0        0
   2       735 / 35           1491 / 71           0        0
```

Figure 2.7:*Avrora*

Using a simulator, it is possible to monitor and analyze every single mote in a simulated network and its response during its life cycle. Energy consumption, packets received, sent or dropped and the mote's LEDs status are some of the variables usually observed. A large number of simulators exist, some of them are: TOSSIM the native simulator from TinyOS, Avrora developed at the University of California Los Angeles, Cooja originally created at the Swedish Institute of Computer Science as a Contiki simulator but now able to simulate nodes programmed in the TinyOS operating system as well, and MSPSim, a MSP430 simulator, also developed at SICS.

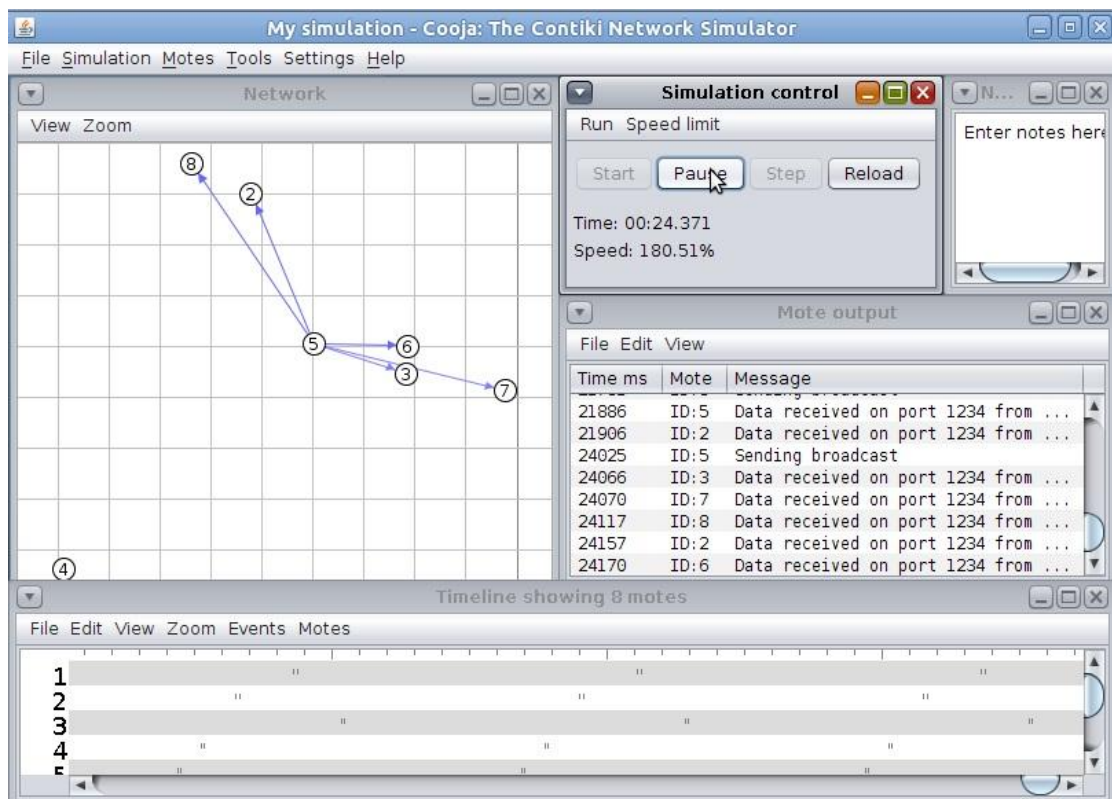

Figure 2.8: *Cooja*

Chapter 3

Motes

3.1 Intro to Motes

The term "mote" was coined by researchers in the Berkeley NEST (now WEBS and CENS projects) to refer to spatially distributed autonomous devices which use sensors to cooperatively monitor physical and/or environmental conditions (e.g. temperature, sound, pressure, vibration) at different locations. Practical WSN nodes, henceforth "motes", currently range in size from disc-shaped boards having diameters less than 1 cm to enclosed systems with typical dimensions less than 5 cm square.

Each mote is composed of a microcontroller, transceiver, memory, power source and one or more sensors, either embedded or external to the sensor board. The motes function within a WSN and typically fulfill one of two purposes: either data logging, processing (and/or transmitting) sensor information from the environment or acting as a gateway in the ad-hoc wireless network formed by all the motes to pass data back to a, usually but not necessarily unique, collection point.

In this chapter we present a brief review of several frequently used WSN motes, compared and contrasted under a number of different parameters.

3.2 Common mote platforms

TelosB/Tmote Sky: Wireless sensor modules developed from research carried out at University of California Berkeley and currently available in similar form factors from Crossbow and Advanticsys.

Figure 3.1 *TelosB/Tmote Sky*

MicaZ: Second and third generation wireless sensor networking mote family from Crossbow.

Figure 3.2 *Micaz*

SHIMMER: (Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability) is a wireless sensor platform designed to support wearable applications and is mainly used in the medical field.

Figure 3.3:*Shimmer*

IRIS: The latest wireless sensor network module from Crossbow. Incudes several improvements over the Mica2/MicaZ family of products. Improvements include increased transmission range.

Figure 3.4: *IRIS*

3.2.1 *Physical Characteristics*

The first parameter which may dictate mote selection for a given application is physical size. Table 3.1 provides an overall comparison of the physical dimensions of the motes in the previous section. This table also lists the motes' weight, which can be a decisive factor when choosing a certain WSN, especially in applications where the motes are components of a mobile unit or are integrated into wearable health monitoring solutions.

Mote Platform	WxLxH [cm]	Weight w/o batt [g]	Weight with batt [g]
TelosB/Tmote Sky	3.2 x 6.6 x 0.7	14.93	63.05
MicaZ	3.2 x 5.7 x 0.6	15.70	63.82
SHIMMER	2 x 4.4 x 1.3	4.87	10.36
IRIS	3.2 x 5.7 x 0.6	21.29	69.40

Table 3.1:*Physical characteristics of motes*

The SHIMMER platform's advantage is obvious. Its small dimensions and low weight make it much more suitable than the other in medical oriented applications. When a mote has to be part of a wearable application, its size and weight are of the utmost importance. Its low weight also minimizes the effect of the motes inertial mass when using the mote's embedded accelerometer. In our case, weight and size is going to be a deciding factor as the mote will be placed on the human body.

3.2.2 *Processor and Memory*

Table 3.2 reviews the microprocessor specifications (bus width and processor clock speed) for each of the respective motes examined. It also provides information on available on-board memory for each mote platform. There is a variety here in available memory sizes, possibly a reflection of their different application spaces.

Mote Platform	Microprocessor	Bus [bits]	Clock [MHz]	RAM [KB]	Flash [KB]	EEPROM
TelosB/ TmoteSky	Texas Instruments MSP430F1611	16	4	10	48	1M
MicaZ	Atmel Atmega	8	8	4	128	512K

	128L					
SHIMMER	Texas Instruments MSP430F1611	16	8	10	48	none
IRIS	Atmel Atmega 1281	8	8	8	640	4K

Table 3.2: *Mote microprocessor specifications*

In addition to these on-board memory capabilities, some sensor nodes also allow the option of saving data to additional external non-volatile memory.

3.2.3 *Communications Capabilities*

The TelosB/Tmote Sky, MicaZ and SHIMMER motes, employ the 802.15.4 compatible CC2420 radio chip from Texas Instruments, while the IRIS Mote uses (again a 802.15.4 compatible chip) Atmel's AT86RF230. These two radios are packet level radios, with a maximum packet length of 127 bytes. In addition to the CC2420, the SHIMMER mote also contains a second radio chip, a class 2 Bluetooth radio compatible with the Mitsumi WML-C46 series. Table 3.3a lists the operating specifications of the three radios and Table 3..3b gives the power consumption of each radio in sleep mode/switched off, idle/receive mode and when transmitting at a specified power level.

Radio Module	Frequency [MHz]	Modulation	Data Rate	Tx Power [dBm]	Rx Sensitivity [dBm]
TI CC2420	2400 - 2483.5	QQPSK	250 Kbps	-24 - 0	-95
Atmel AT86RF230	2405 - 2480	QQPSK	250 Kbps	-17 - 3	-101
Mitsumi WML-C46	2400 - 2483.5	GFSK	721 Kbps	-6 - 14	-82

Table 3.3a: *Mote Communication capabilities*

The CC2420 is a very popular chip for use on wireless sensor nodes, being used on three of the motes considered here. The CC2420 was the first 802.15.4 radio chip to be widely available in the market. 802.15.4 is very suitable for use in WSNs due to its very low power and flexibility. A feature of the CC2420 lacking on the other radios, is

its support for encryption using AES 128. This feature can greatly reduce the cost, both in terms of power and latency, of securing WSN communications.

Radio Module	Sleep [μA]	Idle/Rx [mA]	Tx [mA]
TI CC2420	0.02 - 426	18.8	17.4
Atmel AT86RF230	0.02	15.5	16.5
Mitsumi WML-C46	50 - 1400	40	60

Table 3.3b: *Mote Communication capabilities*

The WML-C46 is a class 2 Bluetooth radio, with a range of approximately 10 meters. WSNs were not considered as a target for Bluetooth when it was being designed and as a result it is not ideally suited for use with them, being overly complex for most applications. However, the presence of Bluetooth allows it to address a current problem faced by 802.15.4 devices, which is interoperability with existing devices. For many applications a Bluetooth enabled mobile phone or laptop can be a very convenient device to use for data aggregation or network querying.

3.2.4 *Sensor Support*

The TelosB/Tmote Sky offers a versatile set of onboard sensors, namely humidity, temperature and light sensors. In addition to the onboard sensors, the TelosB/Tmote Sky provides access to 6 ADC inputs, a UART and I2C bus and several general purpose ports. The MicaZ motes do not have onboard sensors. However, Crossbow offers an extensive set of sensor boards that connect directly to the MicaZ mote and are capable of measuring light, humidity, temperature, pressure etc. Additionally, actuators such as relays and buzzers can be attached too, in case of a WSAN. Intel's SHIMMER mote incorporates a 3 axis accelerometer and allows connection of other sensors through its expansion board. As in MicaZ, more types of sensors (most of them medically oriented) are available. The IRIS mote, in Crossbow tradition, does not offer any embedded sensor capabilities. However, it is equipped with a 51-pin expansion connector that existing MicaZ compatible, Crossbow sensor boards can be connected to.

3.2.5 *Power Specifications*

Both the TelosB and Tmote Sky boards are typically powered from an external battery pack containing two AA batteries. AA cells may be used in the operating range of 2.1 to 3.6V DC, however the voltage must be at least 2.7V when programming the microcontroller flash or external flash. MicaZ and IRIS motes are

also powered by a set of two AA batteries in an attached battery pack. The SHIMMER mote is powered by a rechargeable 450 mAh Li-Ion battery. The Shimmer design also includes a Texas Instruments BQ-24080 Smart Li Charger for battery management.

3.2.6 *Price*

Current (August 2013) pricing information for a single mote is shown in Table 3.4.

Mote Platform	Price
TelosB/Tmote Sky	77 €
MicaZ	77 €
SHIMMER	199 €
IRIS	87 €

Table 3.4: *Mote prices*

3.3 The SHIMMER platform

Since we will simulate our program on a shimmer platform it is essential that we present the platform's main features .

3.3.1 *Shimmer Key Principles*

The Shimmer platform was developed with the goal to allow biomedical researchers to focus on their research instead of the development of applications during the lifecycle of their research project . The key principles that underline the Shimmer wearable sensor platform, as stated on www.shimmer-research.com , are :

• *Flexible*

Shimmer is extremely flexible due to the fact that it can be programmed to meet exact data capture and transfer requirements. Moreover Shimmer can be adapted for different sensing purposes due to its modular expansions described later.

• *Highly Configurable*

Shimmer provides a suite of technologies to offer compatability with a wide variety of system technologies including other sensors. For example the platform offers Bluetooth or 802.15.4 radio for the communication with other devices and the

instant transmission of recorded biomedical signals. Furthermore a microSD card is included, allowing a local storage of recorded data.

• *Open Source*

All code and firmware is actively maintained and available online at Sourceforge .

• *Raw Data*

Shimmer provides raw data, giving the researchers and developers full control over the interpretation and analysis of the sensed data.

3.3.2 *Shimmer Platform Design*

The Shimmer platform consists of a baseboard which provides the sensor computational , data storage, communications and daughterboard connection capabilities. Figure 3.5 gives a brief overview of the sensor's technical specification. The core capabilities of the Shimmer baseboard can be extended via a wide range of daughterboards committed to provide wearable kinematic, biophysical, and ambient wireless sensing capabilities .For this a connector is provided, that allows the user to connect daughterboards to the baseboard. The extension capabilities make a flexible and thus valuable platform out of Shimmer. Table 3.5 gives an overview of the currently available extensions daugtherboards.

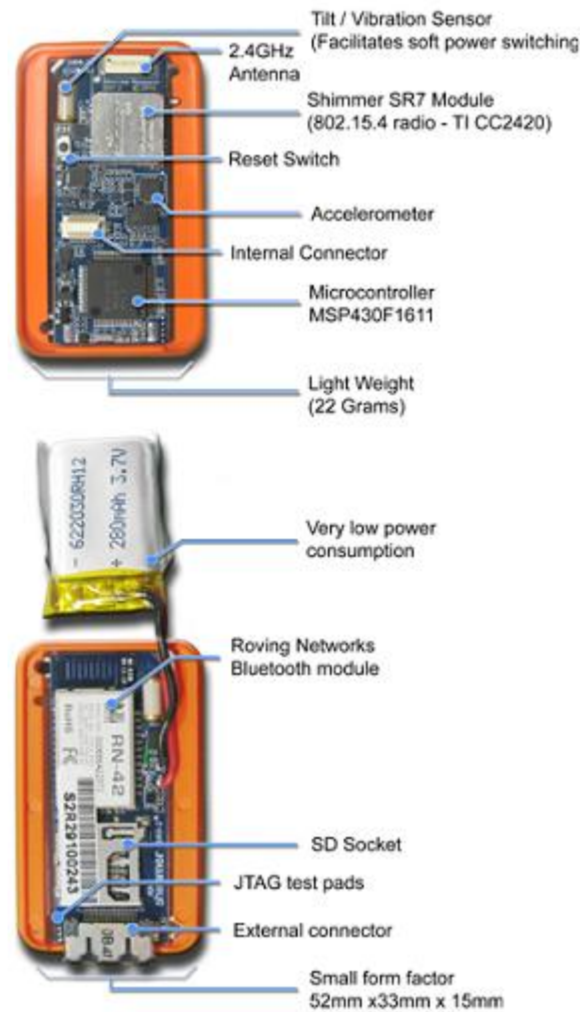

Figure 3.5 *Overview of shimmer's technical specifications*

Kinematic Sensing	Biophysical Sensing	Ambient Sensing
Gyroscope	Electrocardiograph(ECG)	Passive Infrared (PIR) Motion
Magnetometer	Electromyograph (EMG)	Temperature
Accelerometer	Galvanic Skin Response(GSR)	Light

Table 3.5 *Overview of Shimmer extension daugtherboards*

3.3.3 *Shimmer Key Features*

The main requirements a wearable sensor should comply in a healthcare environment are certainly, low power consumption, light weight, small form factor,

low power communication capabilities and modularity . These are exactly the features the Shimmer platform exhibits as consecutively described.

Low Power Consumption

The Shimmer baseboard is equipped with a MSP430 MCU and other hardware to minimize power consumption. Especially the MCU from Texas Instrument is known for its low power consumption during periods of inactivity and has a proven history for biomedical sensing platforms . Further the Shimmer firmware was optimized with focus on lowering the on-time of all hardware subsystems on the sensor hardware platform and thus extends operation life of the sensor.

Small Form Factor and Light Weight

With an enclosure size of 53mm x 32mm x 15mm and a weight of 22 gramms the Shimmer sensor is perfectly appropriate for sensing biomedical data as a wearable mobile sensor.

Communication Capabilities

One of the key features of Shimmer is certainly its ability to communicate wirelessly with other sensor and devices. For this the Shimmer platform provides two modules: the IEEE 802.15.4 radio module and the Bluetooth radio module. The Bluetooth module contains the full Version 2 Bluetooth Protocol Stack guaranteeing the compatibility with a wide range of other Bluetooth devices like mobile phones.

Firmware

The firmware of Shimmer is primarily developed by using TinyOS. It provides the low-level capabilities to control the sensor functions like: local processing of the sensed data, local storage of the data and communication of sensed data to a higher level application for advanced signal processing, display and data persistence . All Shimmer firmware is thereby available online at Sourceforge .The operating system TinyOS, is responsible for task scheduling, radio communication , time, I/O processing and has a very small footprint, which makes it suitable for sensor devices. TinyOS is completely written in nesC , an extension of the programming.

The SHIMMER platform consists of a base board, optional add on boards and a series of firmware versions to match the hardware configuration.

The main components on the SHIMMER base board are:

- Compact form factor, light & wearable (weight: 15 grams, volume: 53mm x 32mm x 15mm)
- Wireless communications via Bluetooth® and 802.15.4 (WML-C46A, CC2420)
- Offline data capture – micro SD card storage – 2 gigabytes
- SD data bypass 8MHz MSP430 CPU (10Kbyte RAM, 48Kbyte flash, 8 channels of 12 bit A/D)
- Open platform, driven by TinyOS
- Internal and external connectors for expansion
- Includes simple serial command interface for Bluetooth®
- Integrated TCP/IP stack for 802.15.4
- Integrated 3-axis MEMs accelerometer with selectable range
- Integrated tilt / vibration sensor
- Integrated Li-ion battery management
- Supported by BioMOBIUS™ graphical software platform
- Example LabView integration module

3.3.4 *Hardware Overview*

The image below illustrates a block diagram of the Shimmer baseboard interconnections between and integrated devices.

CPU

The core element of the platform is the low-power MSP430F1611 microprocessor which controls the operation of the device. Nearly every feature of the CPU is exercised in the Shimmer implementation! The CPU configures and controls various integrated peripherals through I/O pins, some of which are available on the internal/external-expansion connectors

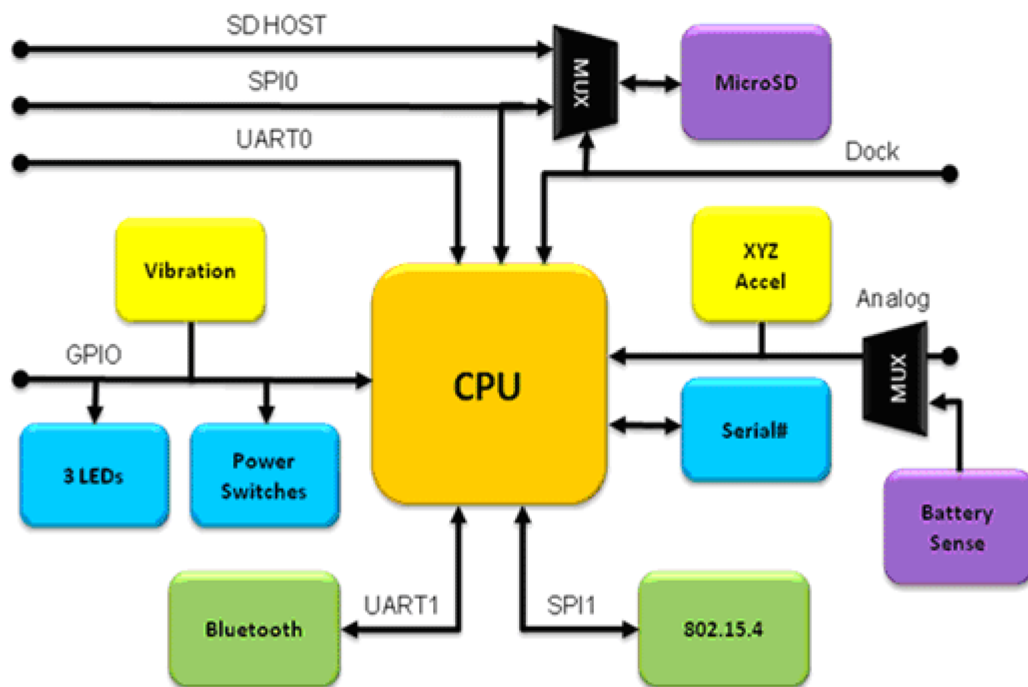

Figure 3.6 *Shimmer Baseboard Interconnections and Integrated Devices*

The CPU has an integrated 8-channel 12bit analog-to-digital converter (ADC) which is used to capture sensor data from the accelerometer, battery, or sensor expansions such as ECG, kinematics, GSR, and EMG. The external expansion allows communication to and from the baseboard using the docking station.

Data Transmission and Storage

For wireless data streaming the platform is equipped with both Bluetooth and 802.15.4 radio modules. The Shimmer board has a built in microSD Flash socket for additional storage. To improve usability, SHIMMER incorporates components to provide direct and immediate access to microSD flash memory using an external SD-flash card controller (SDHOST) for high-speed data transfer.

Functionality

A push-button power controller powers off the entire board after a held press of the reset button. Software controlled power switching is provided for both the Bluetooth radio module and microSD socket. Three light-emitting diodes (LED) are used to display application status.

Finally a triaxial MEMs accelerator and omni-directional tilt/vibration sensing round out the integrated peripheral suite.

3.3.5 *Texas Instruments MSP430F1611*

The low power operation of the Shimmer module is due to the ultra-low power Texas Instruments MSP430F1611 microcontroller featuring 10kB of RAM, 48kB of program flash memory and 128B of information storage. This 16-bit RISC processor features extremely low active and sleep current consumption that permits Shimmer to run for months on a single pair of AA batteries.

The MSP430 includes three clock sources:

- LFXT1CLK: Low frequency/high frequency oscillator that can be used either with low frequency 32768Hz = 32KHz watch crystals, or standard crystals or resonators in the 450KHz to 8MHz range.
- XT2CLK: Optional high frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 450KHz to 8MHz range.
- DCOCLK: Internal digitally controlled oscillator (DCO) with RC-type characteristics.

There are three clock signals available. The one that we are interested in is the Master Clock (MCLK). MCLK is software selectable and is derived from one of LFXT1CLK, XT2CLK or DCOCLK. It is used by the CPU and the system. By default it is sourced from DCOCLK and its default operating frequency in the case of a Shimmer mote is 8MHz.

Although it is software configurable, we will not modify the MCLK from its default value, because of the non CPU-intensive nature of our application. If one would like to do so and provided they are using TinyOS, the necessary component to look for would be MSP430ClockC and more specifically its MSP430ClockInit interface. Operating the mote at higher frequencies might be necessary for CPU-intensive programs, but doing so increases the mote's power consumption, while our main objective is to keep it as low as possible.

The DCO may be turned on from sleep mode in 6μs, however 292ns is typical at 20 ^{0}C. When the DCO is off, the MSP430 operates off an eternal 32768Hz watch crystal. In addition to the DCO, the MSP430 has 8 external ADC ports and 8 internal ADC ports. The internal ports may be used to read the internal thermistor or monitor the battery voltage. A variety of peripherals are available including SPI, UART, digital I/O ports, Watchdog timer and Timers with capture and compare functionality. The F1611 also includes a 2-port 12-bit DAC module, Supply Voltage Supervisor and 3-port DMA controller.

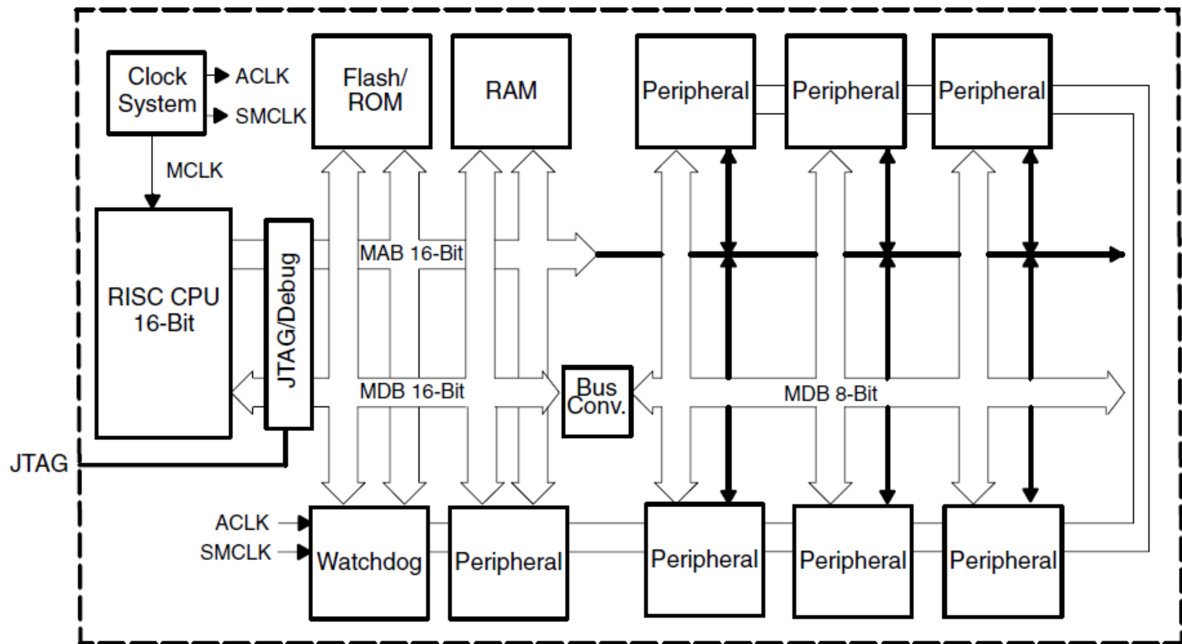

Figure 3.7 :*MSP430 Block diagram from moteiv*

The MSP430 has one active mode and five software selectable modes of operation. An interrupt event can wake up the device from any of the five low-power modes, service the request, and restore back to the low-power mode on return from the interrupt program. As developers, we will not have to worry about MCU management at all in most situations. TinyOS handles everything for us automatically. The low-power modes range from LPM0, which disables only the CPU and main system clock, to LPM4, which disables the CPU, all clocks and the oscillator, expecting to be woken by an external interrupt source. According to the TelosB/Tmote Sky datasheet, the MSP430F1611 draws 1.8 mA of current in Active mode and 5.1 µA in Sleep mode (both computed at 3.0V supply voltage).

3.3.6 *Texas Instruments CC2420*

The CC2420 is a true single-chip 2.4GHz IEEE 802.15.4 compliant RF transceiver designed for low-power and low-voltage wireless applications. CC2420 includes a digital direct sequence spread spectrum baseband modem providing an effective data rate of 250 Kbps. The CC2420 provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information. These features reduce the load on the host controller and allow CC2420 to interface low-cost microcontrollers. It is based on Chipcon’s SmartRF – 03 technology in 180nm CMOS.

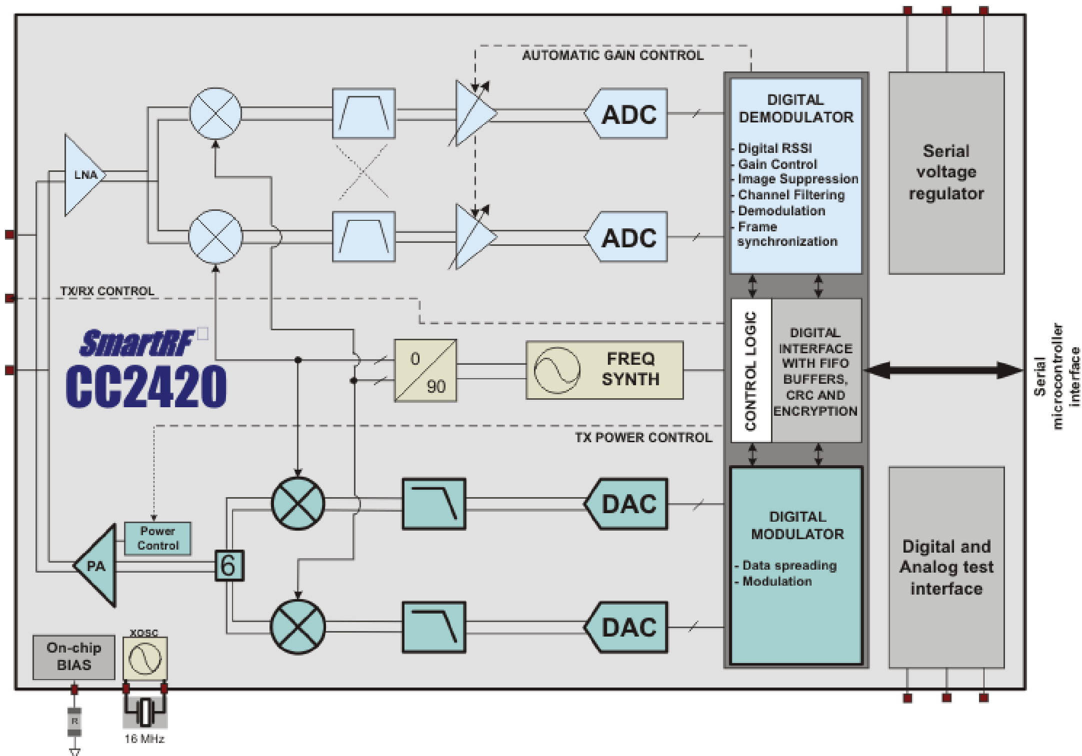

Figure 3.8: *CC2420 Simplified block diagram*

Its main features are summarized below

Frequency Band	2400 ~ 2483.5 MHz	IEEE 802.15.4 Compliant
Sensitivity	-90dBm(min), -95dBm typ	Receive Sensitivity
Transfer Rate	250 Kbps	IEEE 802.15.4
RF Power	-25 dBm ~ 0 dBm	Software Configurable
Range	~100m (outdoor), 20~30m (indoor)	Longer range possible with optional SMA antenna attached
Current Draw	RX: 18.8 mA, TX: 17.4 mA, Sleep: 1 µA	Lower RF Power Modes reduce consumption
RF Power supply	2.1V ~ 3.6V	CC2420 Input Power
Antenna	Dipole Antenna/ PCB Antenna	
Encryption	Hardware MAC encryption	

	AES-128
Buffer	128(RX) + 128(TX) data buffering

Table 3.6: *CC2420 main features*

The CC2420 is controlled by the TI MSP430 microcontroller through the SPI port and a series of digital I/O lines and interrupts. The radio may be put to sleep for low power duty cycled operation. The transceiver also has software configurable output power, which the transmission range is obviously dependent on.

Chapter 4

TinyOS

4.1 Introduction

TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, like the ones used in WSNs, ubiquitous computing, personal area networks, smart buildings, and smart meters. A worldwide community from academia and industry use, develop, and support the operating system as well as its associated tools . The main TinyOS website, www.tinyos.net , has instructions for downloading and installing the TinyOS programming environment. The website has a great deal of useful information such as common hardware platforms and how to install code on a node.

4.1.1 *Networked, embedded sensors*

TinyOS is designed to run on small, wireless sensors. Networks of these sensors have the potential to revolutionize a wide range of disciplines, fields, and technologies. Recent example uses of these devices include:

Golden Gate Bridge safety High-speed accelerometers collect synchronized data on the movement of and oscillations within the structure of San Francisco's Golden GateBridge. This data allows the maintainers of the bridge to easily observe the structural health of the bridge in response to events such as high winds or traffic, as well as quickly assess possible damage after an earthquake . Being wireless avoids the need for installing and maintaining miles of wires.

Volcanic monitoring Accelerometers and microphones observe seismic events on the Reventador and Tungurahua volcanoes in Ecuador. Nodes locally compare when they observe events to determine their location, and report aggregate data to a camp several kilometers away using a long-range wireless link. Small, wireless nodes allow geologists and geophysicists to install dense, remote scientific instruments , obtaining data that answers other questions about unapproachable environments.

Data center provisioning Data centers and enterprise computing systems require huge amounts of energy, to the point at which they are placed in regions that have low power costs. Approximately 50% of the energy in these systems goes into cooling, in part due to highly conservative cooling systems. By installing wireless sensors across machine racks, the data center can automatically sense what areas

need cooling and can adjust which computers do work and generate heat. Dynamically adapting these factors can greatly reduce power consumption, making the IT infrastructure more efficient and reducing environmental impact.

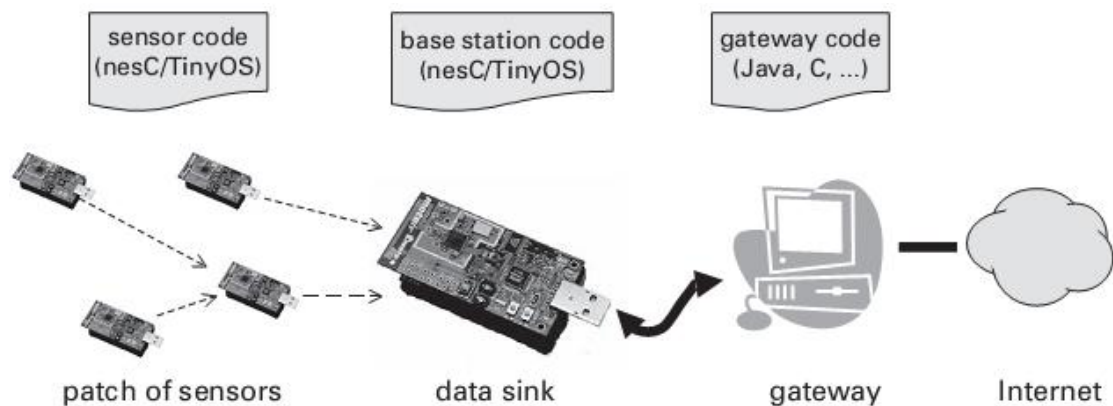

Figure 4.1 *A typical sensor network architecture. Patches of ultra-low power sensors, running nesC/TinyOS, communicate to gateway nodes through data sinks. These gateways connect to the larger Internet*.

Athough these three application domains are only a fraction of where networks of sensors are used, they show the key differences between these networks and most other computing systems. First, these "sensor networks" need to operate unattended for long periods of time. Second, they gather data from and respond to an unpredictable environment. Finally, for reasons of cost, deployment simplicity, and robustness, they are wireless. Together, these three issues – longevity, embedment, and wireless communication – cause sensor networks to use different approaches than traditional, wired, and human-centric or machine-centric systems. The sheer diversity of sensor network applications means that there are many network architectures, but a dominant portion of deployments tend to follow a common one, shown in Figure 4.1 of ultra-low power sensors self-organized to form an ad-hoc routing network to one or more data sink nodes. These sensor sinks are attached to gateways, which are typically a few orders of magnitude more powerful than the sensors: gateways run an embedded form of Linux, Windows, or other multitasking operating system. Gateways have an Internet connection, either through a cell phone network , long-distance wireless, or even just wired Ethernet .Energy concerns dominate sensor hardware and software design. These nodes need to be wireless, small, low-cost, and operate unattended for long periods. While it is often possible to provide large power resources, such as large solar panels, periodic battery replacement, or wall power, to small numbers of gateways, doing so to every one of hundreds of sensors is infeasible.

4.2 TinyOS, what is it.

As mentioned in this chapter before ,TinyOS is a lightweight operating system specifically designed for low-power wireless sensors. TinyOS differs from most other operating systems in that its design focuses on ultra low-power operation. Rather than a full-fledged processor, TinyOS is designed for the small, low-power microcontrollers motes have. Furthermore, TinyOS has very aggressive systems and mechanisms for saving power .TinyOS makes building sensor network applications easier. It provides a set of important services and abstractions, such as sensing, communication, storage, and timers. It defines a concurrent execution model, so developers can build applications out of reusable services and components without having to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, TinyOS's structure makes it reasonably easy to port to new platforms .TinyOS applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions. Later on we will go into the details on how nesC differs significantly from other C-like .

4.2.1*TinyOS, what it provides.*

At a high level, TinyOS provides three things to make writing systems and applications easier:

• a component model, which defines how you write small, reusable pieces of code and compose them into larger abstractions;

• a concurrent execution model, which defines how components interleave their computations as well as how interrupt and non-interrupt code interact;

• application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

The component model is grounded in nesC. It allows you to write pieces of reusable code which explicitly declare their dependencies. For example, a generic user button component that tells you when a button is pressed sits on top of an interrupt handler. The component model allows the button implementation to be independent of which interrupt that is – e.g. so it can be used on many different hardware platforms – without requiring complex callbacks or magic function naming conventions .The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM. First, every I/O call in TinyOS is split-phase: rather than block until completion, a request returns immediately and the caller gets a callback when the I/O completes. Since the stack

isn't tied up waiting for I/O calls to complete, TinyOS only needs one stack, and doesn't have threads .Instead, TinyOS introduces tasks, which are lightweight deferred procedure calls .Any component can post a task , which TinyOS will run at some later time. Because low-power devices must spend most of their time asleep, they have low CPU utilization and so in practice tasks tend to run very soon after they are posted (within a few milliseconds).Furthermore, because tasks can't preempt each other, task code doesn't need to worry about data races. Low-level interrupt code can have race conditions, of course: nesC detects possible data races at compile-time and warns you .Finally, TinyOS itself has a set of APIs for common functionality, such as sending packets, reading sensors, and responding to events .In addition to programming interfaces, TinyOS also provides a component structure and component libraries .TinyOS itself is continually evolving. Within the TinyOS community, "Working Groups" form to tackle engineering and design issues within the OS, improving existing services and adding new ones .The best way to stay up to date with TinyOS is to check its web page www.tinyos.net and participate in its mailing lists.

4.3 Example application

To better understand the unique challenges faced by sensor networks, we walk through a basic data-collection application. Nodes running this application periodically wake up ,sample some sensors, and send the data through an ad hoc collection tree to a data sink (as in Figure 4.1). As the network must last for a year, nodes spend 99% of their time in a deep sleep state .In terms of energy, the radio is by far the most expensive part of the node. Lasting a year requires telling the radio to be in a low power state. Low power radio implementation techniques are beyond the scope of this book, but the practical upshot is that packet transmissions have higher latency. Figure 4.2 shows the four TinyOS APIs the application uses: low power settings for the radio, a timer, sensors, and a data collection routing layer. When TinyOS tells the application that the node has booted, the application code configures the power settings on the radio and starts a periodic timer. Every few minutes, this timer fires and the application code samples its sensors. It puts these sensor values into a packet and calls the routing layer to send the packet to a data sink. In practice, applications tend to be more complex than this simple example. For example, they include additional services such as a management layer which allows an administrator to reconfigure parameters and inspect the state of the network, as well as over-the-air programming so the network can be reprogrammed without needing to collect all of the nodes. However, these four abstractions – power

control, timers, sensors, and data collection – encompass the entire datapath of the application.

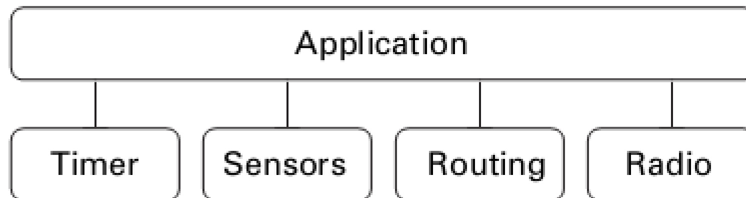

Figure 4.2: Example application architecture. Application code uses a timer to act periodically, sensors to collect data, and a routing layer to deliver data to a sink.

4.3.1*Compiling and installing applications*

You can download the latest TinyOS distribution, the nesC compiler, and other tools at www.tinyos.net . The TinyOS website has step-by-step tutorials to get you started. One part of TinyOS is an extensive build system for compiling applications. Generally, to compile a program for a sensor platform, one types make <platform>, e.g. make telosb. This compiles a binary. To install that binary on a node, you plug the node into your PC using a USB or serial connection, and type make <platform> install. The tutorials go into compilation and installation options in detail.

Chapter 5

NesC

5.1 First Approach to nesC

NesC (network embedded systems C), is a component-based, event driven programming language used to build applications for the TinyOS platform. NesC is built as an extension to the C programming language with components "wired" together to run applications on TinyOS. It offers a more ‘’holistic’’ approach to embedded systems while supporting the TinyOS’s design. It is a static language with all resources known at compile time and call-graph fully known at compile time. nesC’s contribution is to support the special needs of the WSN exposing a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design. Restrictions on the programming model al-low the nesC compiler to perform whole-program analyses, including data-race detection (which improves reliability) and aggressive function inlining (which reduces resource consumption)

There are a number of unique challenges that nesC must address:

Driven by interaction with environment: Unlike traditional computers, motes are used for data collection and control of the local environment, rather than general-purpose computation. This focus leads to two observations. First, motes are fundamentally event-driven, reacting to changes in the environment (message arrival ,sensor acquisition) rather than driven by interactive or batch processing. Second, event arrival and data processing are concurrent activities, demanding an approach to concurrency management that addresses potential bugs such as race conditions.

Limited resources: Motes have very limited physical resources ,due to the goals of small size, low cost, and low power consumption. We do not expect new technology to remove these limitations: the benefits of Moore’s Law will be applied to reduce size and cost, rather than increase capability

Reliability: Although we expect individual motes to fail due to hardware issues, we must enable very long-lived applications. For example, environmental monitoring applications must collect data without human interaction for months at a time. An important goal is to reduce run-time errors, since there is no real recovery mechanism in the field except for automatic reboot. Soft real-time requirements: Although there are some tasks that are time critical, such as radio management or sensor polling, we do not focus on hard real-time guarantees. Our experience so far

indicates that timing constraints are easily met by having complete control over the application and OS, and limiting utilization. One of the few timing-critical aspects in sensor networks is radio communication; however, given the fundamental unreliability of the radio link, it is not necessary to meet hard deadlines in this domain.

The basic concepts behind nesC are:

1. Separation of construction and composition: programs are built out of components, which are assembled ("wired") to form whole programs. Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.
2. Specification of component behaviour in terms of set of interfaces. Interfaces may be provided or used by components. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job.
3. Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event. Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.
4. Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages robust design, and allows for better static analysis of programs.
5. nesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

Program structure is the most essential and obvious difference between C and nesC .C programs are composed of variables, types, and functions defined in files that are compiled separately and then linked together. nesC programs are built out of components that are connected ("wired") by explicit program statements . the nesC compiler connects and compiles these components as a single unit .To illustrate and explain these differences in how programs are built, we compare and contrast C and nesC implementation of a very simple "hello world"-like mote application, Blink (boot and repeatedly blink a LED).

A few basic principles underlie nesC's design:

nesC is an extension of C: C produces efficient code for all the target microcontrollers that are likely to be used in sensor networks. C provides all the low-level features necessary for accessing hardware, and interaction with existing C code is simplified. Last but not least, many programmers are familiar with C.C does have significant disadvantages: it provides little help in writing safe code or in structuring applications. nesC addresses safety through reduced expressive power and structure through components. None of the new features in nesC are tied to C: the same ideas could be added to other imperative programming languages such as Modula-2 .

Whole-program analysis: nesC programs are subject to whole program analysis (for safety) and optimization (for performance).Therefore we do not consider separate compilation in nesC's design. The limited program size on motes makes this approach tractable.

nesC is a "static language": There is no dynamic memory allocation and the call-graph is fully known at compile-time. These restrictions make whole program analysis and optimization significantly simpler and more accurate. They sound more onerous than they are in practice: nesC's component model and parameterized interfaces eliminate many needs for dynamic memory allocation and dynamic dispatch. We have, so far, implemented one optimization and one analysis: a simple whole-program inliner and a data-race detector .nesC supports and reflects TinyOS's design: nesC is based on the concept of components, and directly supports TinyOS's event-based concurrency model. Additionally, nesC explicitly addresses the issue of concurrent access to shared data .In practice, nesC resolved many ambiguities in the TinyOS concepts of components and concurrency, and TinyOS evolved to the nesC versions as it was reimplemented.

5.2 Basic nesC Programming

The nesC language is primarily intended for embedded systems such as sensor networks. nesC has a C-like syntax, but supports the TinyOS concurrency model, as well as mechanisms for structuring, naming, and linking together software components into robust network embedded systems. The principal goal is to allow application designers to build components that can be easily composed into complete, concurrent systems, and yet perform extensive checking at compile time.

TinyOS defines a number of important concepts that are expressed in nesC. First, nesC applications are built out of **components** with well-defined, bidirectional **interfaces**. Second, nesC defines a concurrency model, based on **tasks** and **hardware event handlers**, and detects **data races** at compile time.

Components

Specification

A nesC application consists of one or more **components** linked together to form an executable. A component **provides** and **uses interfaces**. These interfaces are the only point of access to the component and are bidirectional. An interface declares a set of functions called **commands** that the interface provider must implement and another set of functions called **events** that the interface user must implement. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface.

Implementation

There are two types of components in nesC: **modules** and **configurations**. Modules provide application code, implementing one or more interface. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. This is called **wiring**. Every nesC application is described by a **top-level configuration** that *wires* together the components inside .nesC uses the filename extension ".nc" for all source files-interfaces, modules and configurations.

Concurrency Model

TinyOS executes only one program consisting of selected system components and custom components needed for a single application. There are two threads of execution: **tasks** and **hardware event handlers**. Tasks are functions whose execution is deferred. Once scheduled, they run to completion and do not preempt one another. Hardware event handlers are executed in response to a hardware interrupt and also runs to completion, but may preempt the execution of a task or other hardware event handler. Commands and events that are executed as part of a hardware event handler must be declared with the **async** keyword.

Because tasks and hardware event handlers may be preempted by other asynchronous code, nesC programs are susceptible to certain race conditions. Races are avoided either by accessing shared data exclusively within tasks, or by having all accesses within **atomic** statements. The nesC compiler reports potential **data races** to the programmer at compile-time. It is possible the compiler may report a false positive. In this case a variable can be declared with the **norace** keyword. The norace keyword should be used with extreme caution.

5.3 Example Application:Blink

The most concrete first-approach example application for nesC is the Blink application the simple test program "Blink" causes the red LED on the mote to turn on and off at 1Hz. Blink application is composed of two **components**: a **module**, called "BlinkM.nc", and a **configuration**, called "Blink.nc". All applications require a top-level configuration file, which is typically named after the application itself. In

this case Blink.nc is the configuration for the Blink application and the source file that the nesC compiler uses to generate an executable file. BlinkM.nc, on the other hand, actually provides **the implementation** of the Blink application. Blink.nc is used to wire the BlinkM.nc module to other components that the Blink application requires.

The reason for the distinction between modules and configurations is to allow a system designer to quickly "snap together" applications. For example, a designer could provide a configuration that simply wires together one or more modules, none of which he actually designed. Likewise, another developer can provide a new set of "library" modules that can be used in a range of applications.

Sometimes (as is the case with Blink and BlinkM) you will have a configuration and a module that go together. When this is the case, the convention used in the TinyOS source tree is that Foo.nc represents a configuration and FooM.nc represents the corresponding module. While you could name an application's implementation module and associated top-level configuration anything, to keep things simple it is better to adopt this convention in our code.

5.3.1 *The Blink.nc Configuration*

The nesC compiler, ncc, compiles a nesC application when given the file containing the top-level configuration. Typical TinyOS applications come with a standard Makefile that allows platform selection and invokes ncc with appropriate options on the application's top-level configuration.

Below we present the configuration for this application:

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

The first thing to notice is the key word `configuration`, which indicates that this is a configuration file. The first two lines,

```
configuration Blink {
}
```

simply state that this is a configuration called `Blink`. Within the empty braces here it is possible to specify `uses` and `provides` clauses, as with a module. This is important to keep in mind: a configuration can use and provide interfaces!

The actual configuration is implemented within the pair of curly bracket following key word `implementation` . The components line specifies the set of components that this configuration references, in this case `Main`, `BlinkM`, `SingleT-imer`, and `LedsC`. The remainder of the implementation consists of connecting interfaces used by components to interfaces provided by others.

`Main` is a component that is executed first in a TinyOS application. To be precise, the `Main.StdControl.init()` command is the first command executed in TinyOS followed by `Main.StdControl.start()`. Therefore, a TinyOS application must have Main component in its configuration. `StdControl` is a common interface used to initialize and start TinyOS components. Let us have a look at `tos/interfaces/StdControl.nc`:

```
StdControl.nc
Interface StdControl {
command result_t init();
command result_t start();
command result_t stop();
}
```

We see that `StdControl` defines three **commands**, `init()`,`start()`, and `stop()`. `init()` is called when a component is first initialized, and `start()` when it is started, that is, actually executed for the first time. `stop()` is called when the component is stopped, for example, in order to power off the device that it is controlling. `init()` can be called multiple times, but will never be called after either `start()` or `stop` are called. Specifically, the valid call patterns of `StdControl` are `init*(start | stop)*` . All three of these commands have "deep" semantics; calling `init()` on a component must make it call `init()` on all of its subcomponents. The following 2 lines in Blink configuration

```
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
```

wire the `StdControl` interface in `Main` to the `StdControl` interface in both `BlinkM` and `SingleTimer`. `SingleTimer.StdControl.init()` and

BlinkM.StdControl.init() will be called by Main.StdControl.init(). The same rule applies to the start() and stop() commands.

Concerning used interfaces, it is important to note that subcomponent initialization functions must be explicitly called by the using component. For example, the BlinkM module uses the interface Leds, so Leds.init() is called explicitly in BlinkM.init().

nesC uses arrows to determine relationships between interfaces. Think of the right arrow (->) as "binds to". The left side of the arrow binds an interface to an implementation on the right side. In other words, the component that uses an interface is on the left, and the component **provides** the interface is on the right.

The line

```
BlinkM.Timer -> SingleTimer.Timer;
```

is used to wire the Timer interface used by BlinkM to the Timer interface provided by SingleTimer. BlinkM.Timer on the left side of the arrow is referring to the interface called Timer (tos/interfaces/Timer.nc), while SingleTimer.Timer on the right side of the arrow is referring to the implementation of Timer (tos/lib/SingleTimer.nc). The arrow always binds interfaces (on the left) to implementations (on the right).

nesC supports multiple implementations of the same interface. The Timer interface is such an example. The SingleTimer component implements a single Timer interface while another component, TimerC, implements multiple timers using timer id as a parameter.

Wirings can also be implicit. For example,

```
BlinkM.Leds -> LedsC;
```

is really shorthand for

```
BlinkM.Leds -> LedsC.Leds;
```

If no interface name is given on the right side of the arrow, the nesC compiler by default tries to bind to the same interface as on the left side of the arrow.

5.3.2 *The Blink.nc Module*

Now let's look at the module BlinkM.nc:

```
BlinkM.nc
module BlinkM {
provides {

            interface StdControl;
}
uses {

            interface Timer;

            interface Leds;

      }

}
// Continued below...
```

The first part of the code states that this is a module called BlinkM and declares the interfaces it provides and uses. The BlinkM module provides the interface StdControl. This means that BlinkM implements the StdControl interface. As explained above, this is necessary to get the Blink component initialized and started. The BlinkM module also uses two interfaces: Leds and Timer. This means that BlinkM may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces.

The Leds interface defines several commands like redOn(), redOff(), and so forth, which turn the different LEDs (red, green, or yellow) on the mote on and off. Because BlinkM uses the Leds interface, it can invoke any of these commands. However Leds is just an interface: the implementation is specified in the Blink.nc configuration file.

Timer.nc is a little more interesting:

```
Timer.nc
interface Timer {

command result_t start(char type, uint32_t interval);
command result_t stop();
```

```
event result_t fired();
}
```

Here we see that `Timer` interface defines the `start()` and `stop()` commands, and the `fired()` event.

The `start()` command is used to specify the type of the timer and the interval at which the timer will expire. The unit of the interval argument is millisecond. The valid types are `TIMER_REPEAT` and `TIMER_ONE_SHOT`. A one-shot timer ends after the specified interval, while a repeat timer goes on and on until it is stopped by the `stop()` command.

How does an application know that its timer has expired? The answer is when it receives an event. The `Timer` interface provides an event:

```
  event result_t fired();
```

An **event** is a function that the implementation of an interface will signal when a certain event takes place. In this case, the `fired()` event is signaled when the specified interval has passed. This is an example of a **bidirectional** interface: an interface not only provides **commands** that can be called by users of the interface, but also signals **events** that call handlers in the user. Think of an event as a callback function that the implementation of an interface will invoke. A module that uses an interface must implement the events that this interface uses.

Let's look at the rest of `BlinkM.nc` to see how this all fits together:

```
BlinkM.nc, continued

implementation {

  command result_t StdControl.init() {

    call Leds.init();

    return SUCCESS;

  }

  command result_t StdControl.start() {

    return call Timer.start(TIMER_REPEAT, 1000) ;

  }

  command result_t StdControl.stop() {
```

```
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

As we see the `BlinkM` module implements the `StdControl.init()`, `StdControl.start()`, and `StdControl.stop()` commands, since it provides the `StdControl` interface. It also implements the `Timer.fired()` event, which is necessary since `BlinkM` must implement any event from an interface it uses.

The `init()` command in the implemented `StdControl` interface simply initializes the `Leds` subcomponent with the call to `Leds.init()`. The `start()` command invokes `Timer.start()` to create a repeat timer that expires every 1000 ms. `stop()` terminates the timer. Each time `Timer.fired()` event is triggered, the `Leds.redToggle()` toggles the red LED.

Also , a graphical representation of the component relationships within an application can be viewed. TinyOS source files include metadata within comment blocks that ncc, the nesC compiler, uses to automatically generate html-formatted documentation. To generate the documentation, we need to type `make <platform> docs` from the application directory.

5.3.3 *Compiling the Blink Application*

TinyOS supports multiple platforms. Each platform has its own directory in the `tos/platform` directory. In our example we will use the `telosb` platform. In the TinyOS source tree, compiling the Blink application for the `telosb` mote is as simple as typing

```
make telosb
```

in the apps/Blink directory. Of course this doesn't tell us anything about how the nesC compiler is invoked.

nesC itself is invoked using the ncc command which is based on gcc. For example, we can type

```
ncc -o main.exe -target=telosb Blink.nc
```

to compile the Blink application (from the Blink.nc top-level configuration) to main.exe, an executable file for the telosb mote. If we want to upload the code to the actual hardware mote, we use

```
avr-objcopy --output-target=srec main.exe main.srec
```

to produce `main.srec`, which essentially represents the binary `main.exe` file in a text format that can be used for programming the telosb mote. We then use another tool (such as uisp) to actually upload the code to the mote, depending on our environment. In general we will never need to invoke `ncc` or `avr-objcopy` by hand, the `Makefile` does all this for us, but it's nice to see that all we need to compile a nesC application is to run ncc on the top-level configuration file for our application. ncc takes care of locating and compiling all of the different components required by our application, linking them together, and ensuring that all of the component wiring matches up.We won't expand further in the hardware part of the mote since it is out of the scope of this project .Further information on how to install a nesC code such as `Blink` on a mote can be found on www.tinyos.net.

5.3.4 *Interfaces, Commands, and Events*

We learned that if a component uses an interface, it can call the interface's commands and must implement handlers for its events. We also saw that the `BlinkC` component uses the `Timer`, `Leds`, and `Boot` interfaces.Let's take a look at those interfaces:

```
interface Boot {
event void booted();
}

interface Leds {
/**
* Turn LED n on, off, or toggle its present state.
*/
async command void led0On();
async command void led0Off();
async command void led0Toggle();
async command void led1On();
async command void led1Off();
async command void led1Toggle();
async command void led2On();
async command void led2Off();
async command void led2Toggle();
/**
```

```
* Get/Set the current LED settings as a bitmask. Each bit
corresponds to
* whether an LED is on; bit 0 is LED 0, bit 1 is LED 1,
etc.
*/

async command uint8_t get();
async command void set(uint8_t val);
}

interface Timer
{
// basic interface

command void startPeriodic( uint32_t dt );
command void startOneShot( uint32_t dt );
command void stop();
event void fired();

// extended interface omitted (all commands)
}
```

Looking over the interfaces for `Boot`, `Leds`, and `Timer`, we can see that since `BlinkC` uses those interfaces it must implement handlers for the `Boot.booted()` event, and the `Timer.fired()` event.The `Leds` interface signature does not include any events, so `BlinkC` need not implement any in order to call the `Leds` commands. Here, again, is `BlinkC`'s implementation of `Boot.booted()`:

```
event void Boot.booted()
{
call Timer0.startPeriodic( 250 );
call Timer1.startPeriodic( 500 );
call Timer2.startPeriodic( 1000 );
}
```

`BlinkC` uses 3 instances of the `TimerMilliC` component, wired to the interfaces `Timer0`, `Timer1`, and `Timer2`. The `Boot.booted()` event handler starts each instance. The parameter `to startPeriodic()` specifies the period in milliseconds after which the timer will fire (it's millseconds because of the `<TMilli>`in the interface). Because the timer is started using the `startPeriodic()` command, the timer will be reset after firing such that the `fired()` event is triggered every n milliseconds. Invoking an interface command requires the call keyword, and invoking an interface event requires the signal keyword. `BlinkC` does not provide any interfaces, so its code does not have any signal statements.

Next, we present the implementation of the `Timer.fired()`:

```
event void Timer0.fired()
{
call Leds.led0Toggle();
}
event void Timer1.fired()
{
call Leds.led1Toggle();
}
event void Timer2.fired()
{
call Leds.led2Toggle();
}
```

Because it uses three instances of the `Timer` interface, `BlinkC` must implement three instances of `Timer.fired()` event. When implementing or invoking an interface function, the function name is always `interface.function`. As `BlinkC`'s three `Timer` instances are named `Timer0`, `Timer1`, and `Timer2`, it implements the three functions `Timer0.fired`, `Timer1.fired`, and `Timer2.fired`.

5.4 Tasks

All of the code we've looked at so far is synchronous. It runs in a single execution context and does not have any kind of pre-emption. That is, when synchronous (sync) code starts running, it does not relinquish the CPU to other sync code until it completes. This simple mechanism allows the TinyOS scheduler to minimize its RAM consumption and keeps sync code very simple. However, it means that if one piece of sync code runs for a long time, it prevents other sync code from running, which can adversely affect system responsiveness. For example, a long-running piece of code can increase the time it takes for a mote to respond to a packet. So far, all of the examples we've looked at have been direct function calls. System components, such as the boot sequence or timers, signal events to a component, which takes some action (perhaps calling a command) and returns. In most cases, this programming approach works well. Because sync code is non-preemptive ,however, this approach does not work well for large computations. A component needs to be able to split a large computation into smaller parts, which can be executed one at a time. Also, there are times when a component needs to do something, but it's fine to do it a little later. Giving TinyOS the ability to defer the computation until later can let it deal with everything else that's waiting first. Tasks enable components to perform general-purpose "background" processing in an application. A task is a function which a component tells TinyOS to run later, rather than now. A task is declared in the implementation module using the syntax

```
task void taskname() { ... }
```

where taskname() is whatever symbolic name we want to assign to the task. Tasks must return **void** and may not take any arguments. To dispatch a task for (later) execution, use the syntax

```
post taskname();
```

A component can post a task in a command, an event, or a task. Because they are the root of a call graph , tasks can safely both call commands and signal events. We will see later that, by convention, commands do not signal events to avoid creating recursive loops across component boundaries (e.g., if command X in component 1 signals event Y in component 2, which itself calls command X in component 1). These loops would be hard for the programmer to detect (as they depend on how the application is wired) and would lead to large stack usage. The post operation places the task on an internal task queue which is processed in FIFO order. When a task is executed, it runs to completion before the next task is run. Therefore, and as the above examples showed, a task should not run for long periods of time. Tasks do not preempt each other, but a task can be preempted by a hardware interrupts (which we haven't seen yet). In case we need to run a series of long operations, we should dispatch a separate task for each operation, rather than using one big task. The post operation returns an error_t, whose value is either SUCCESS or FAIL. A post fails if and only if the task is already pending to run (it has been posted successfully and has not been invoked yet).

5.5 Radio communication

TinyOS provides a number of interfaces to abstract the underlying communications services and a number of components that provide(implement) these interfaces. All of these interfaces and components use a common message buffer abstraction, called message_t, which is implemented as a nesC struct (similar to a C struct). message_t is an abstract data type, whose members are read and written using accessor and mutator functions.

```
typedef nx_struct message_t {
nx_uint8_t header[sizeof(message_header_t)];
nx_uint8_t data[TOSH_DATA_LENGTH];
nx_uint8_t footer[sizeof(message_footer_t)];
nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

5.5.1 *Basic Communications Interfaces*

There are a number of interfaces and components that use message_t as the underlying data structure.Let's take a look at some of the interfaces that are in the tos/interfaces directory to familiarize ourselves with the general functionality of the communications system:

• **Packet** - Provides the basic accessors for the message_t abstract data type. This interface provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.

• **Send** - Provides the basic address-free message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.

• **Receive** - Provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area.

5.5.2 *Active Message Interfaces*

Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. The term "AM type" refers to the field used for multiplexing. AM types are similar in function to the Ethernet frame type field, IP protocol field, and the UDP port in that all of them are used to multiplex access to a communication service. AM packets also includes a destination field, which stores an "AM address" to address packets to particular motes. Additional interfaces, also located in the tos/interfaces directory, were introduced to support the AM services:

• **AMPacket** - Similar to Packet, provides the basic AM accessors for the message_t abstract data type. This interface provides commands for getting a node's AM address, an AM packet's destination,and an AM packet's type. Commands are also provides for setting an AM packet's destination and type, and checking whether the destination is the local node.

• **AMSend** - Similar to Send, provides the basic Active Message sending interface. The key difference between AMSend and Send is that AMSend takes a destination AM address in its send command.

5.5.3 *Components*

A number of components implement the basic communications and active message interfaces. Let's take a look at some of the components in the /tos/system directory. We should be familiar with these components because our code needs to specify both the interfaces our application uses as well as the components which provide(implement) those interfaces:

• **`AMReceiverC`** - Provides the following interfaces: `Receive, Packet`, and `AMPacket`.

• **`AMSenderC`** - Provides `AMSend, Packet, AMPacket`, and `PacketAcknowledgements` as `Acks`.

• **`AMSnooperC`** - Provides `Receive, Packet`, and `AMPacket`.

• **`AMSnoopingReceiverC`** - Provides `Receive, Packet`, and `AMPacket`.

• **`ActiveMessageAddressC`** - Provides commands to get and set the node's active message address. This interface is not for general use and changing a node's active message address can break the network stack, so we best avoid using it unless we know what we are doing.

5.5.4 *Sending a Message over the Radio*

Our message will send both the node id and the counter value over the radio. Rather than directly writing and reading the payload area of the `message_t` with this data, we will use a structure to hold them and then use structure assignment to copy the data into the message payload area. Using a structure allows reading and writing the message payload more conveniently when our message has multiple fields or multi-byte fields (like `uint16_t` or `uint32_t`) because we can avoid reading and writing bytes from/to the payload using indices and then shifting and adding (e.g. `uint16_t x = data[0] << 8 + data[1]`). Even for a message with a single field, we should get used to using a structure because if we ever add more fields to our message or move any of the fields around, we will need to manually update all of the payload position indices if we read and write the payload at a byte level. Using structures is straightforward. The following defines a message structure with a `uint16_t node id` and a `uint16_t counter` in the payload:

```
typedef nx_struct BlinkToRadioMsg {

nx_uint16_t nodeid;
nx_uint16_t counter;

} BlinkToRadioMsg;
```

Given we are familiar with C structures, this syntax looks quite familar but the `nx_prefix` on the keywords `struct` and `uint16_t` stands out a little bit. The `nx_` prefix is specific to the nesC language and signifies that the `struct` and `uint16_t` are network types . Network types have the same representation on all platforms. The nesC compiler generates code that transparently reorders access to `nx_` data types and eliminates the need to manually address endianness and alignment (extra padding in structs present on some platforms) issues.

We will implement a new application, called BlinkToRadioC which will periodically broadcast a counter value.Now that we have defined a message type for our application, BlinkToRadioMsg, we will next see how to send the message over the radio.

Let's walk through the steps, one-by-one:

1. We will use the AMSend interface to send packets as well as the Packet and AMPacket interfaces to access the message_t abstract data type. We need to start the radio using the ActiveMessageC.SplitControl interface.

```
module BlinkToRadioC {
...

uses interface Packet;
uses interface AMPacket;
uses interface AMSend;
uses interface SplitControl as AMControl;
}
```

Note that SplitControl has been renamed to AMControl using the as keyword. nesC allows interfaces to be renamed in this way for several reasons. First, it often happens that two or more components that are needed in the same module provide the same interface. The as keyword allows one or more such names to be changed to distinct names so that they can each be addressed individually. Second, interfaces are sometimes renamed to something more meaningful. In our case, SplitControl is a general interface used for starting and stopping components, but the name AMControl is a mnemonic to remind us that the particular instance of SplitControl is used to control the ActiveMessageC component.

2. We need a message_t to hold our data for transmission. These declarations need to be added in the implementation block of BlinkToRadioC.nc:

```
implementation {

bool busy = FALSE;
message_t pkt;
...

}
```

Next, we need to handle the initialization of the radio. The radio needs to be started when the system is booted so we must call AMControl.start inside Boot.booted. The only complication is that in our current implementation, we start a timer inside Boot.booted and we are planning to use this timer to send messages over the radio but the radio can't be used until it has completed starting up. The radio signals that it has completed starting through the AMControl.startDone event. To ensure that we do not start using the radio

before it is ready, we need to postpone starting the timer until after the radio has completed starting. We can accomplish this by moving the call to start the timer, which is now inside `Boot.booted`, to `AMControl.startDone`, giving us a new `Boot.booted` with the following body:

```
event void Boot.booted() {
call AMControl.start();
}
```

We also need to implement the `AMControl.startDone` and `AMControl.stopDone` event handlers, which have the following bodies:

```
event void AMControl.startDone(error_t err) {

if (err == SUCCESS) {
call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
}
else {
call AMControl.start();
}
}
event void AMControl.stopDone(error_t err) {
}
```

If the radio is started successfully, `AMControl.startDone` will be called with the `error_t` parameter set to a value of SUCCESS. If the radio starts successfully, then it is appropriate to start the timer. If, however, the radio does not start successfully, then it obviously cannot be used so we try again to start it. This process continues until the radio starts, and ensures that the node software doesn't run until the key components have started successfully. If the radio doesn't start at all, a human operator might notice that the LEDs are not blinking as they are supposed to, and might try to debug the problem.

3. Since we want to transmit the node's id and counter value every time the timer fires, we need to add some code to the `Timer0.fired` event handler:

```
event void Timer0.fired() {
...
if (!busy) {
BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call
Packet.getPayload(&pkt,NULL));
btrpkt->nodeid = TOS_NODE_ID;
btrpkt->counter = counter;
if (call AMSend.send(AM_BROADCAST_ADDR, &pkt,
sizeof(BlinkToRadioMsg)) ==SUCCESS) {busy = TRUE;}
}
}
```

This code performs several operations. First, it ensures that a message transmission is not in progress by checking the busy flag. Then it gets the packet's payload portion and casts it to a pointer to the previously declared BlinkToRadioMsg external type. It can now use this pointer to initialise the packet's fields, and then send the packet by calling AMSend.send. The packet is sent to all nodes in radio range by specyfing AM_BROADCAST_ADDR as the destination address. Finally, the test against SUCCESS verifies that the AM layer accepted the message for transmission. If so, the busy flag is set to true. For the duration of the send attempt, the packet is owned by the radio, and user code must not access it.

4. Looking through the Packet, AMPacket, and AMSendinterfaces, we see that there is only one event we need to worry about, AMSend.sendDone:

```
/*** Signaled in response to an accepted send request.
msg is the message buffer sent, and error indicates
whether the send was successful.

@param msg the packet which was submitted as a send
request

 @param error SUCCESS if it was sent successfully, FAIL
if it was not, ECANCEL if it was cancelled @see send

@see cancel***/

event void sendDone(message_t* msg, error_t error);
```

This event is signaled after a message transmission attempt. In addition to signaling whether the message was transmitted successfully or not, the event also returns ownership of msg from AMSend back to the component that originally called the AMSend.send command. Therefore sendDone handler needs to clear the busy flag to indicate that the message buffer can be reused:

```
event void AMSend.sendDone(message_t* msg, error_t error)
{
if (&pkt == msg) {
busy = FALSE;
}
}
```

Note the check to ensure the message buffer that was signaled is the same as the local message buffer.This test is needed because if two components wire to the same AMSend, bothwill receive a sendDone event after either component issues a send command. Since a component writer has no way to enforce that her component will not be used in this manner, a defensive style of programming that verifies that the sent message is the same one that is being signaled is required.

5. The following lines can be added just below the existing components declarations in the implementation block of `BlinkToRadioAppC.nc`:

```
implementation {
...
components ActiveMessageC;
components new AMSenderC(AM_BLINKTORADIO);
...
}
```

These statements indicate that two components, `ActiveMessageC` and `AMSenderC`, will provide the needed interfaces. However, note the slight difference in their syntax. `ActiveMessageC` is a singleton component that is defined once for each type of hardware platform. `AMSenderC` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMSenderC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMSenderC`. We can extend the `enum` in the `BlinkToRadio.h` header file to incorporate the value of

```
AM_BLINKTORADIO:
enum {
AM_BLINKTORADIO = 6,
TIMER_PERIOD_MILLI = 250
};
```

6. The following lines will wire the used interfaces to the providing components. These lines should be added to the bottom of the implementation block of `BlinkToRadioAppC.nc`:

```
implementation {
...
App.Packet -> AMSenderC;
App.AMPacket -> AMSenderC;
App.AMSend -> AMSenderC;
App.AMControl -> ActiveMessageC;
```

5.5.5 *Receiving a Message over the Radio*

Now that we have an application that is transmitting messages, we can add some code to receive and process the messages. Below we add code that, upon receiving a message, sets the LEDs to the three least significant bits of the counter in the message.

1. We will use the `Receive` interface to receive packets.

```
module BlinkToRadioC {
...
```

```
uses interface Receive;
}
```

2. We need to implement the `Receive.receive` event handler:

```
event message_t* Receive.receive(message_t* msg, void*
payload, uint8_t len) {                          if (len
== sizeof(BlinkToRadioMsg)) {
BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
call Leds.set(btrpkt->counter);
}
return msg;
}
```

The `receive` event handler performs some simple operations. First, we need to ensure that the length of the message is what is expected. Then, the message payload is cast to a structure pointer of type `BlinkToRadioMsg*`and assigned to a local variable. Then, the counter value in the message is used to set the states of the three LEDs. Note that we can safely manipulate the `counter` variable outside of an atomic section. The reason is that receive event executes in task context rather than interrupt context (events that have the `async` keyword can execute in interrupt context). Since the TinyOS execution model allows only one task to execute at a time, if all accesses to a variable occur in task context, then no race conditions will occur for that variable. Since all accesses to `counter` occur in task context, no critical sections are needed when accessing it.

3. The following lines can be added just below the existing components declarations in the implementation block of `BlinkToRadioAppC.nc`:

```
implementation {
...
components new AMReceiverC(AM_BLINKTORADIO);
...
}
```

This statement means that a new instance of `AMReceiverC` will be created. `AMReceiver` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMReceiverC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMReceiverC` and is chosen to be the same as that used for the `AMSenderC` used earlier, which ensures that the same AM type is being used for both transmissions and receptions. `AM_BLINKTORADIO` is defined in the `BlinkToRadio.h` header file.

4. Update the wiring by insert the following line just before the closing brace of the implementation block in `BlinkToRadioAppC`:

```
implementation {
...
```

```
App.Receive -> AMReceiverC;
}
```

The Blink Application and its derivative applications we presented above were a small presentation of nesC's basic features . Most of those features are included in our project .Thus we got a better understanding of how nesC language actually works ,how it interacts with C language and in what way we will use TinyOS to better implement our application.

Chapter 6

The MSPSIM Simulator

Software development for wireless sensor networks is a challenging and time consuming task. The resource limited hardware with limited I/O and debugging abilities combined with the often cumbersome hardware debugging tools makes low-level debugging on the target hardware difficult. We present MSPSim , an extensible sensor board platform and MSP430 instruction level simulator that simulates sensor boards with peripherals for the purpose of reducing development and debugging time .The use of a simulator also enables testing without access to the target hardware and makes more advanced debugging and instrumenting possible.

6.1 The Simulator

Due to the distributed nature of sensor networks and resource-constraints of sensor nodes, code development for wireless sensor network is a challenging and time consuming task. Furthermore, the application development and debugging tools are still cumbersome. One of the most commonly used methods for debugging sensor nodes is using on-chip emulation via JTAG that makes it possible to single-step and debug a running application on the target hardware. This is useful for understanding execution patterns, stack usage, etc, but less useful for debugging communication, sensor drivers, etc. For the development of wireless sensor network applications, system simulators exist that simplify the development of algorithms and enable researcher to study the algorithm's behavior and interaction in a controlled environment .Cross-level simulation enables simultaneous simulation at different levels of the sensor network and hence supports simultaneous low-level debugging and application development . For cross-level simulation of our MSP430-based sensor node platforms we required an extensible instruction level simulation. Towards, this end, we use MSPsim. As Avrora , MSPsim is a sensor network simulator simulating nodes at the instruction-level, but for the MSP430. Unlike ATEMU that emulates the operations of individual nodes and simulates communication between them , MSPsim is designed for instruction-level simulation but can by design be incorporated in COOJA's cross-level simulation environment. Therefore , MSPSim is all about an extensible instruction level simulator for the MSP430 microcontroller

that is intended to be used as a component in a larger sensor net-work simulation system supporting cross-level simulation . For this reason MSPsim is designed to run multiple instances of the simulator in a single process unlike other simulators such as the GDB MSP430 simulator . MSPsim also contains a sensor board simulator that simulates hardware peripherals such as sensors, communication ports, LEDs, and sound devices such as a beeper. The design of MSPsim, together with its implementation in Java, makes it easy to adapt the simulator to new sensor boards.

The MSPsim is a Java-based instruction level simulator for the MSP430 microcontroller that simulates unmodified target platform firmware. Supports loading of IHEX and ELF firmware files, and has some tools for monitoring stack, setting breakpoints, and profiling .MSPsim is an instruction-level simulator which made it easy to achieve accurate timing simulation. Further, MSPsim can run unmodified target platform firmware. The simulator is easily extensible with peripheral devices making it possible to simulate various types of MSP430 based sensor nodes. In addition to simulate the MSP430 and sensor board hardware, MSPsim can show a graphical representation of the sensor board in an on-screen window. LEDs on the sensor board are displayed using the correct colors. The graphical output allows a system designed to visually verify that an application is correctly simulated by inspection of the LEDs.

6.1.1 *Main Features*

The main features of MSPSim are numbered below:

1) Instruction level emulation of MSP430 microprocessor
2) Supports loading of ELF and IHEX files
3) Easy to add external components that emulates external HW
4) Supports monitoring of registers, adding breakpoints, etc.
5) Built-in profiling of executed code
6) Statistics for various components modes (on/off, LPM modes, etc).
7) Emulates some external hardware such as TR1001 and CC2420.
8) Command Line Interface, CLI, for setting up breakpoints and output to files or windows.
9) GDB remote debugging support (initial)

6.1.2 *What is emulated of the MSP430*

a) CPU (instruction level simulation)
b) Timer A/B subsystem
c) USARTs
d) Digital I/O
e) Multiplication unit
f) Basic A/D subsystem (not complete)
g) Watchdog

6.2 Sensor Board Simulation

We will simulate a simple application on a telosb Sky mote with the mspsim .One of the design objectives of the MSPsim simulator is to simplify the adaptation to different types of sensor node platforms. To add support for a new sensor node platform only implementations of peripherals such as sensors, actuators such as beepers or LEDs, and radio and communication peripherals are needed. The implementation of those peripherals are typically relatively easy to make as many of them do not need to conform to strict timing requirements .We will simulate the fundamental application Blink , which we presented on an earlier chapter .As we recall , Blink application simply causes the red LED on the mote to turn on and off at 1Hz .Well , this is not exactly the case here because we use a different version of the Blink application from TinyOS – 2.1.2 to better fit the telosb mote and give us a better understanding of the simulation .More specifically , the Blink application we used causes the blue LED on the mote to turn on and off at 1Hz , the green Led on the mote to turn on and off at 2Hz and the red LED on the mote to turn on and off at 4Hz .In other words , it is a binary 3-bit counter from 0 to 7 decimal .A blinking LED account for binary ''1'' and o non-blinking LED accounts for binary ''0'' .It's function is portrayed on table 6.1 below.

BLUE LED	GREEN LED	RED LED	COUNTER(BIN)	COUNTER(DEC)
0	0	0	000	0
0	0	1	001	1
0	1	0	010	2
0	1	1	011	3
1	0	0	100	4
1	0	1	101	5
1	1	0	110	6
1	1	1	111	7

Table 6.1 *Function of Blink application*

First we show the `BlinkAppC` configuration

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;


  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

The application includes the components `MainC,BlinkC ,LedsC` and `TimerMilliC`.Then it creates three instances of the `TimerMilliC` `Timer0,Timer1` and `Timer2.`

```
components MainC, BlinkC, LedsC;
components new TimerMilliC() as Timer0;
components new TimerMilliC() as Timer1;
components new TimerMilliC() as Timer2;
```

Afterwards , it wires `BlinkC` to `MainC.Boot` setting `BlinkC` as the main component.

```
BlinkC -> MainC.Boot;
```

Next , it wires each timer of `BlinkC` with an instant of `TimerMilliC`

```
BlinkC.Timer0 -> Timer0;
BlinkC.Timer1 -> Timer1;
BlinkC.Timer2 -> Timer2;
```

Finally, it wires the `Leds` of `BlinkC` to the `LedsC` component

```
BlinkC.Leds -> LedsC;
```

Below , we present the module `BlinkC`.

```
#include "Timer.h"

module BlinkC()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    dbg("BlinkC", "Timer 0 fired @ %s.\n",
sim_time_string());
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
```

```
  {
    dbg("BlinkC", "Timer 1 fired @ %s \n",
sim_time_string());
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    dbg("BlinkC", "Timer 2 fired @ %s.\n",
sim_time_string());
    call Leds.led2Toggle();
  }
}
```

It uses interfaces `Timer` , `Leds` and `Boot`

```
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
```

Calls the `Timer` command 3 times for each timer when booted .Each timer is set with different fire limit.`Timer0` is fired with 250ms , `Timer1` with 500ms and `Timer2` with 1000ms

```
    event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

If a `Timerx` is fired , the respective x LED Blinks

```
  event void Timerx.fired()
  {
    dbg("BlinkC", "Timer x fired @ %s.\n",
sim_time_string());
    call Leds.ledxToggle();
  }
```

As we mentioned in the installation process of the mspsim , in order to simulate an application on a mote , first we need to compile the application on this mote with the msp430-gcc . After having install the msp430-gcc version 4.6.3 , we go to Blink application folder which includes the BlinkC configuration (`BlinkAppC.nc`) , the BlinkC module (`BlinkC.nc`) and the `Makefile`. Here we type the command:

make telosb

Then , a folder named build is created inside the Blink application folder . This folder contains folder named telosb .The latter includes an executable file named main.exe.

After we rename the file to main.elf (in order an executable file to be simulated with mspsim it must end with .elf) with the command:

mv main.exe main.elf

Finally while we are in Blink/build/telosb , we run the command:

mspsim main.elf

The mspsim simulator is activated and the simulation has started

The message :

Flash got reset! MSPSim>
Autoloading script: /home/homedirectory/mspsim/scripts/autorun.sc

MSPSim 0.97 starting firmware : main.elf

is displayed and the command line of mspsim is ready:

MSPSim>

Also , the following 5 Windows have opened:

- Sky

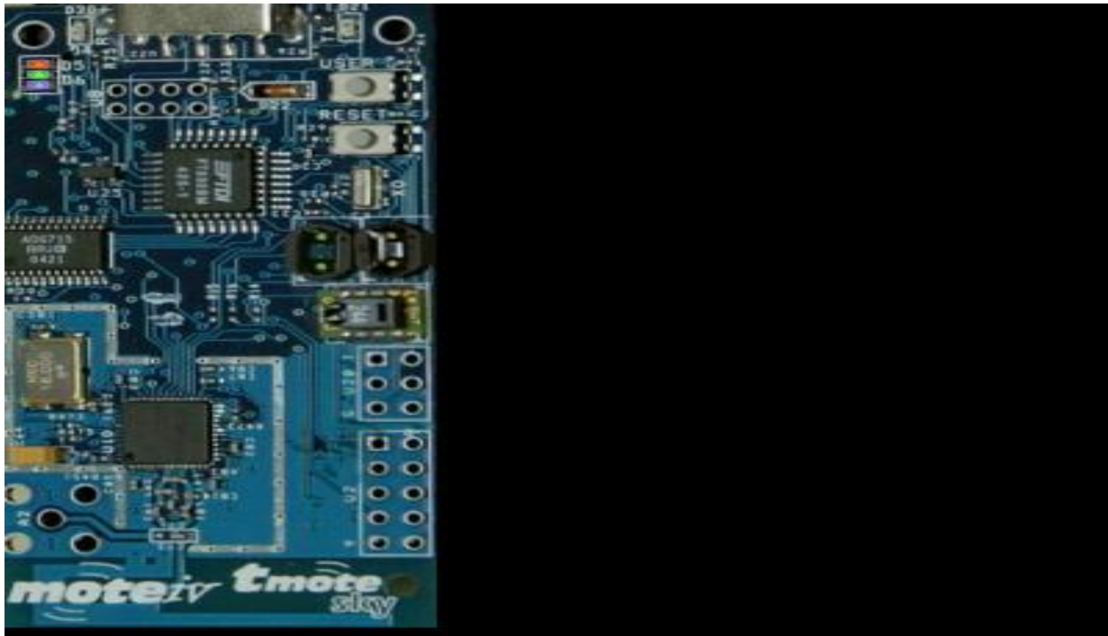

Figure 6.1a *Sky window*

The Sky window offers an optical-hardware representation of the mote .On the upper left side we can see the 3 Blinking Leds.

- USART1 Port Output

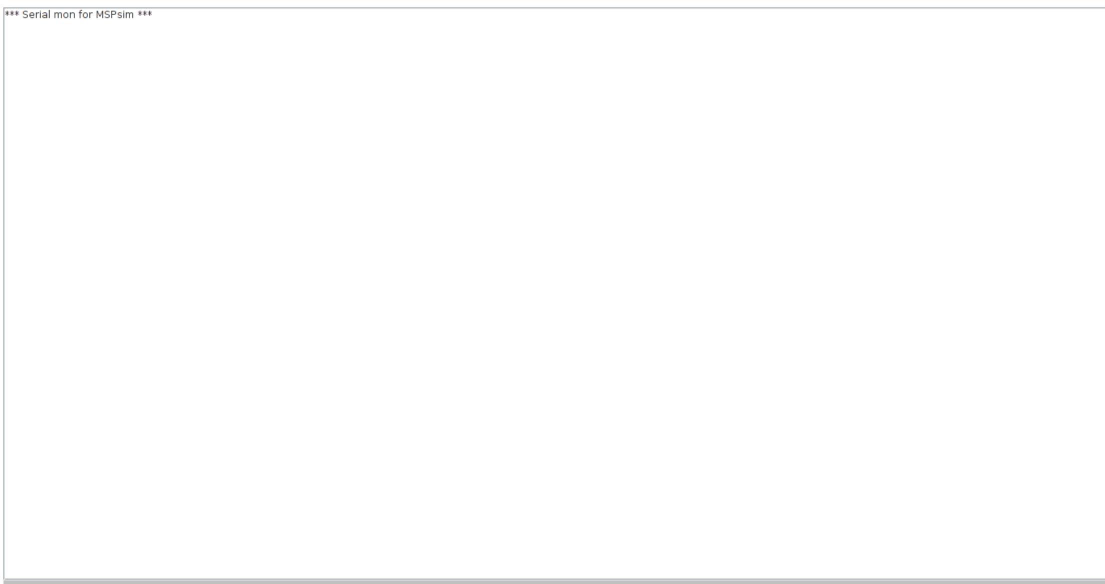

Figure 6.1b *USART 1Port Output window*

The USART1 Port Output concerns the usb port output that would function if we had an actual mote inserted on a usb port

- Duty Cycle Monitor

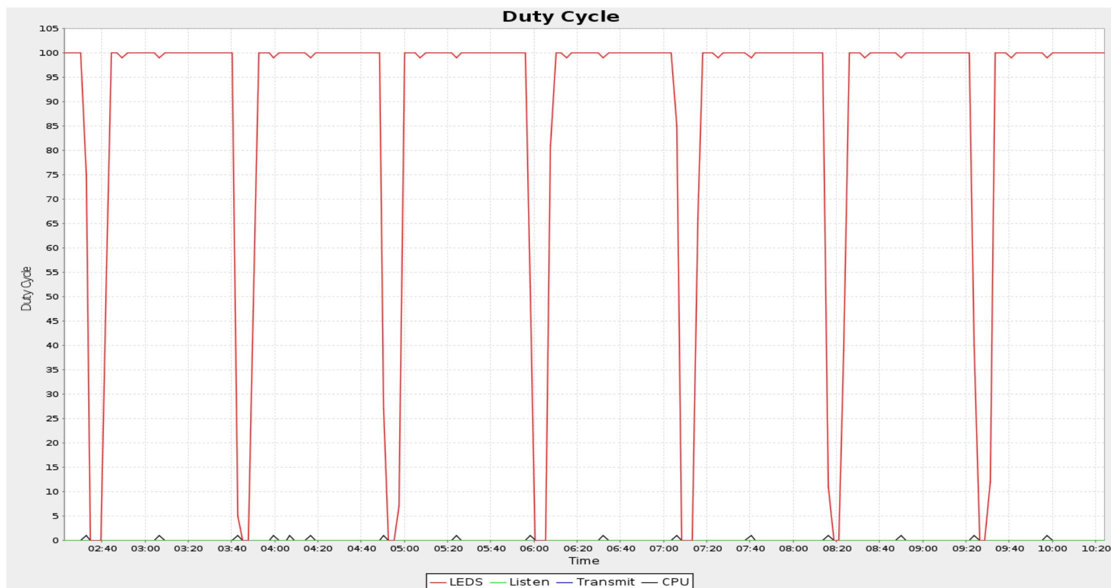

Figure 6.1c *Duty Cycle Monitor window*

The Duty Cycle Monitor shows the duty cycle of the LEDS(red color) , the duty cycle of the listening-sensing activity(green color) , the duty cycle of the Transmitting activity (Blue color) and the duty cycle of the CPU(blackcolor)

- Stack Monitor

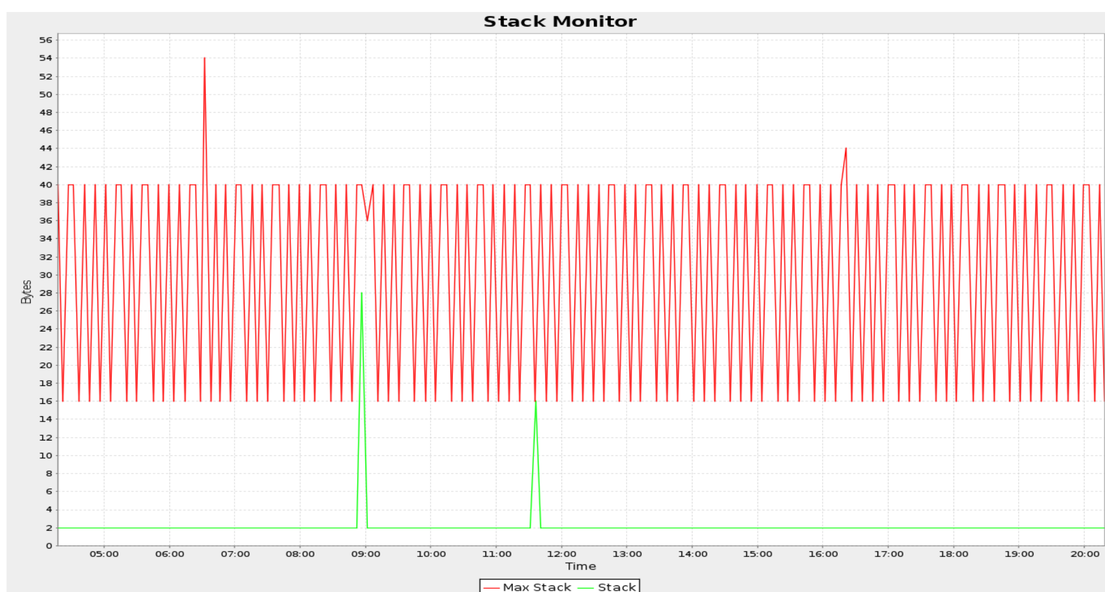

Figure 6.1d *Stack Monitor window*

The Stack monitor shows the Max Stack(red color) and the Stack (green color)

- MSPSim monitor

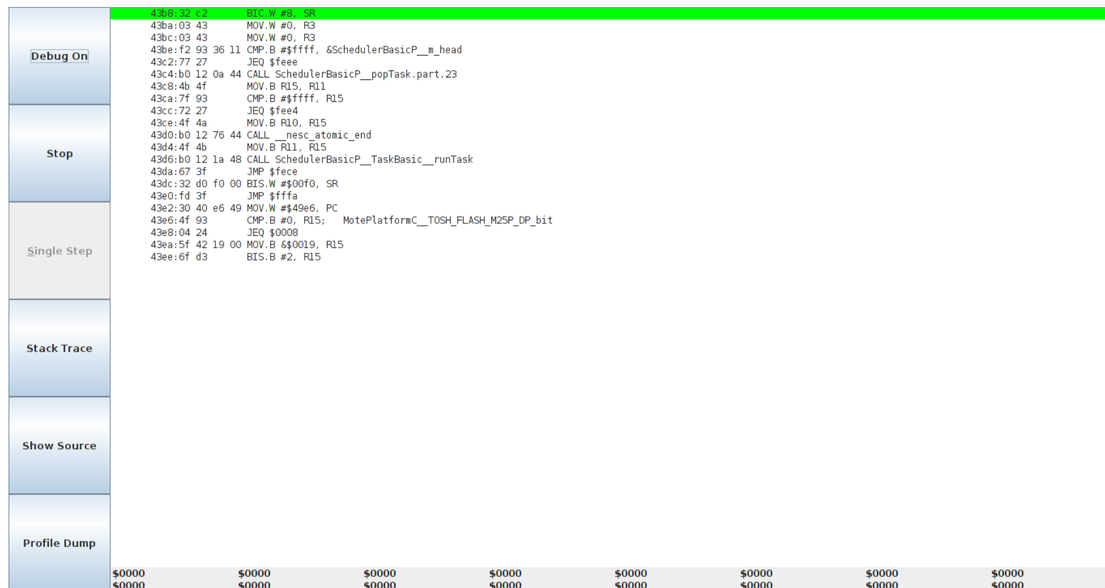

Figure 6.1e *MSPSimMonitor window*

The MSPSim monitor window shows the steps executed in assembly code. It offers step by step execution , debug monitor making it visible on the command line and start-stop actions.

In the command line , If we execute:

MSPSim>profile

The MSPSim command `profile` can be useful to see the number of cycles elapsed in each function of the program. For our Blink application after a certain period of time the output is shown in Figure 6.2

```
File Edit View Search Terminal Help
--------------------------------------------------
MSPSim>profile
************************ Profile Data ************************************
Function                                           Average    Calls  Tot.Cycles
SchedulerBasicP__TaskBasic__runTask                    524    25391    13308979
VirtualizeTimerC__0__fireTimers                        248    16157     4007911
TransformAlarmC__0__Counter__get                       107    34628     3724070
TransformAlarmC__0__set_alarm                          240     9234     2222522
SchedulerBasicP__TaskBasic__postTask                    79    25393     2006819
Msp430TimerP__0__Event__fired                           13   144527     1878851
__nesc_atomic_start                                     13   110805     1450222
Msp430TimerP__1__Event__fired                          109    10360     1131627
__nesc_atomic_end                                        8   110804      994903
SchedulerBasicP__popTask.part.23                        28    25391      710948
VirtualizeTimerC__0__Timer__startPeriodic              239        3         717
MotePlatformC__TOSH_FLASH_M25P_DP_bit                   26        8         213
RealMainP__Scheduler__runNextTask                       12        2          24
********** Profile IRQ *************************
Vector          Average    Calls  Tot.Cycles
00                    0        0          0
01                    0        0          0
02                    0        0          0
03                    0        0          0
04                    0        0          0
05                   58   144529    8382682
06                    0        0          0
07                    0        0          0
08                    0        0          0
09                    0        0          0
10                    0        0          0
11                    0        0          0
12                   66     1127      74382
13                  160     9233    1477280
14                    0        0          0
15                    0        0          0
gscharitos@gscharitos-Inspiron-N5110:~/examples/Blink/build/telosb$
```

Figure 6.2: MSPSim terminal command line

Also , MSPSim allows to watch the variable values evolution during the program execution with the command `symbol <application>` . It will be illustrated profoundly during our project.

Another command we will be using in our project is the duty command:

MSPSim>duty [freq] [chip]

Duty is useful for viewing each chips duty cycle at a frequency of our choice.

Chapter 7

Design

As we mentioned in previous chapters , the source application we will be using from the TinyOS libraries is the AccelECG application. Our new application will be named ECG application.In addition to the ECG app , we derived 3 more apps by modifying the ECG's code. The ECG-cordic , the ECG-lookup and the ECG-lookup-interpolate app.The following chapters describe these four applications in-detail.

7.1 ECG App

7.1.1 *ECGC.nc*

Below we present and analyse the code for the module ECGC.nc. The ECG is slightly different than the AccelECG . More specifically , we added removed the three accelerometer channels and replaced them with 3 ECG channels and also added 3 more ECG channels which adds up to 8 ECG channels.In other words the ECG app sends 8 ECG channels with each packet.

ECGC.nc

```
/*
 * Copyright (c) 2007, Intel Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or
without
 * modification, are permitted provided that the following
conditions are met:
 *
 * Redistributions of source code must retain the above
copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above
copyright notice,
 * this list of conditions and the following disclaimer in the
documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of the Intel Corporation nor the names of
its contributors
```

```
 * may be used to endorse or promote products derived from
this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Author: Adrian Burns
 *         November, 2007
 */


/**************************************************************
*********************

   This app uses Bluetooth to stream 8 ECG channels
   of data to a BioMOBIUS PC application.
   Tested on SHIMMER Base Board Rev 1.3, SHIMMER ECG board Rev
1.1.

   LOW_BATTERY_INDICATION if defined stops the app streaming
data just after the
   battery voltage drops below the regulator value of 3V.

   Default Sample Frequency: 100 hz

   Packet Format:
         BOF| Sensor ID | Data Type | Seq No. | TimeStamp |
Len | ECG | ECG  | ECG  | ECG   | ECG  | ECG  | ECG   | ECG  |
Dummy|  CRC | EOF
   Byte:  1 |    2      |     3     |    4    |     5-6   |  7
| 8-9 | 10-11| 12-13| 14-15 | 16-17| 18-19| 20-21 | 22-23| 24-
25| 26-27| 28


**************************************************************
********************/
```

```
/*
 * @author Adrian Burns
 * @date November, 2007
 *
 * @author Mike Healy
 * @date May 13, 2009 - ported to TinyOS 2.x
 *
 * @author Steve Ayer
 * @date   June, 2010
 * mods to use shimmerAnalogSetup interface and new GyroBoard
module
 */

#include "Timer.h"
#include "ECG.h"
#include "crc.h"
#include "RovingNetworks.h"

module ECGC {
   uses {
      interface Boot;
      interface FastClock;
      interface Init as FastClockInit;
      interface Init as BluetoothInit;
      interface Leds;
      interface Timer<TMilli> as SetupTimer;
      interface Timer<TMilli> as ActivityTimer;
      interface Timer<TMilli> as SampleTimer;
      interface LocalTime<T32khz>;
      interface StdControl as BTStdControl;
      interface Bluetooth;
      interface shimmerAnalogSetup;
      interface Msp430DmaChannel as DMA0;
   }
}

implementation {
   extern int sprintf(char *str, const char *format, ...)
__attribute__ ((C));

#ifdef LOW_BATTERY_INDICATION
   //#define DEBUG_LOW_BATTERY_INDICATION
   /* during testing of the the (AVcc-AVss)/2 value from the
ADC on various SHIMMERS, to get a reliable cut off point
      to recharge the battery it is important to find the
baseline (AVcc-AVss)/2 value coming from the ADC as it varies
      from SHIMMER to SHIMMER, however the range of
fluctuation is pretty constant and (AVcc-AVss)/2 provides an
accurate
      battery low indication that prevents getting any voltage
skewed data from the accelerometer or add-on board sensors */
#define TOTAL_BASELINE_BATT_VOLT_SAMPLES_TO_RECORD 1000
#define BATTERY_LOW_INDICATION_OFFSET 20 /* (AVcc - AVss)/2 =
Approx 3V-0V/2 = 1.5V, 12 bit ADC with 2.5V REF,
                                              4096/2500 =
1mV=1.6384 units */
```

```
   bool need_baseline_voltage, linkDisconnecting;
   uint16_t num_baseline_voltage_samples, baseline_voltage;
   uint32_t sum_batt_volt_samples;

#ifdef DEBUG_LOW_BATTERY_INDICATION
   #error "were going for debug mode yea?, comment me out
then"
   uint16_t debug_counter;
#endif /* DEBUG_LOW_BATTERY_INDICATION */

#endif /* LOW_BATTERY_INDICATION */

#define FIXED_PACKET_SIZE 28
#define FIXED_PAYLOAD_SIZE 12
   uint8_t tx_packet[(FIXED_PACKET_SIZE*2)+1]; /* (*2)twice
size because of byte stuffing */
                                               /* (+1)MSP430
CPU can only read/write 16-bit values at even addresses, */
                                               /* so use an
empty byte to even up the memory locations for 16-bit values
*/
   const uint8_t personality[17] = {
      0,1,2,3,4,5,0xFF,0xFF,
      SAMPLING_50HZ,SAMPLING_50HZ,SAMPLING_50HZ,SAMPLING_50HZ,

SAMPLING_50HZ,SAMPLING_50HZ,SAMPLING_0HZ_OFF,SAMPLING_0HZ_OFF,
FRAMING_EOF
   };

   norace uint8_t current_buffer = 0;
   uint16_t sbuf0[6], sbuf1[6], timestamp0, timestamp1;
   bool enable_sending, command_mode_complete,
activity_led_on;

/* default sample frequency every time the sensor boots up */
   uint16_t sample_freq = SAMPLING_200HZ;
   uint8_t NBR_ADC_CHANS;

   /* Internal function to calculate 16 bit CRC */
   uint16_t calc_crc(uint8_t *ptr, uint8_t count) {
      uint16_t crc;
      crc = 0;
      while (count-- > 0)
         crc = crcByte(crc, *ptr++);

      return crc;
   }

   void init() {
#ifdef USE_8MHZ_CRYSTAL
     call FastClockInit.init();
     call FastClock.setSMCLK(1);
#endif /* USE_8MHZ_CRYSTAL */

      call BluetoothInit.init();
```

```
      call shimmerAnalogSetup.addECGInputs();
      call shimmerAnalogSetup.finishADCSetup(sbuf0);

      NBR_ADC_CHANS = call
shimmerAnalogSetup.getNumberOfChannels();

      atomic {
         memset(tx_packet, 0, (FIXED_PACKET_SIZE*2));
         enable_sending = FALSE;
         command_mode_complete = FALSE;
         activity_led_on = FALSE;
      }

      call Bluetooth.disableRemoteConfig(TRUE);
   }

   event void Boot.booted() {
      init();
      call BTStdControl.start();
      /* so that the clinicians know the sensor is on */
      call Leds.led0On();
#ifdef LOW_BATTERY_INDICATION
      /* initialise baseline voltage measurement stuff */
      need_baseline_voltage = TRUE;
      num_baseline_voltage_samples = baseline_voltage =
sum_batt_volt_samples = 0;
      call Leds.led0On();
#ifdef DEBUG_LOW_BATTERY_INDICATION
      debug_counter = 0;
#endif /* DEBUG_LOW_BATTERY_INDICATION */
#endif /* LOW_BATTERY_INDICATION */
   }

#ifdef LOW_BATTERY_INDICATION
   task void sendBatteryLowIndication() {
      uint16_t crc;
      char batt_low_str[] = "BATTERY LOW!";

      /* stop all sensing - battery is below the threshold */
      call SetupTimer.stop();
      call ActivityTimer.stop();
      call shimmerAnalogSetup.stopConversion();
      call DMA0.stopTransfer();
      call Leds.led1Off();

      /* send the battery low indication packet to BioMOBIUS
*/
      tx_packet[1] = FRAMING_BOF;
      tx_packet[2] = SHIMMER_REV1;
      tx_packet[3] = STRING_DATA_TYPE;
      tx_packet[4]++; /* increment sequence number */
      atomic tx_packet[5] = timestamp0 & 0xff;
      atomic tx_packet[6] = (timestamp0 >> 8) & 0xff;
      tx_packet[7] = FIXED_PAYLOAD_SIZE;
      memcpy(&tx_packet[8], &batt_low_str[0], 12);
```

```
#ifdef DEBUG_LOW_BATTERY_INDICATION
      tx_packet[8] = (baseline_voltage) & 0xff;
      tx_packet[9] = ((baseline_voltage) >> 8) & 0xff;
#endif /* DEBUG_LOW_BATTERY_INDICATION */

      crc = calc_crc(&tx_packet[2], (FIXED_PACKET_SIZE-
FRAMING_SIZE));
      tx_packet[FIXED_PACKET_SIZE - 2] = crc & 0xff;
      tx_packet[FIXED_PACKET_SIZE - 1] = (crc >> 8) & 0xff;
      tx_packet[FIXED_PACKET_SIZE] = FRAMING_EOF;

      call Bluetooth.write(&tx_packet[1], FIXED_PACKET_SIZE);
      atomic enable_sending = FALSE;

      /* initialise baseline voltage measurement stuff */
      need_baseline_voltage = TRUE;
      num_baseline_voltage_samples = baseline_voltage =
sum_batt_volt_samples = 0;
      call Leds.led0On();
   }

   /* all samples are got so set the baseline voltage for this
SHIMMER hardware */
   void setBattVoltageBaseline() {
      baseline_voltage = (sum_batt_volt_samples /
TOTAL_BASELINE_BATT_VOLT_SAMPLES_TO_RECORD);
   }

   /* check voltage level and if it is low then stop sampling,
send message and disconnect */
   void checkBattVoltageLevel(uint16_t battery_voltage) {
#ifndef DEBUG_LOW_BATTERY_INDICATION
      if(battery_voltage < (baseline_voltage-
BATTERY_LOW_INDICATION_OFFSET)) {
#else
      if(debug_counter++ == 2500) {
#endif /* DEBUG_LOW_BATTERY_INDICATION */
         linkDisconnecting = TRUE;
      }
   }

   /* keep checking the voltage level of the battery until it
drops below the offset */
   void monitorBattery() {
      uint16_t battery_voltage;
      if(current_buffer == 1) {
         battery_voltage = sbuf0[5];
      }
      else {
         battery_voltage = sbuf1[5];
      }
      if(need_baseline_voltage) {
         num_baseline_voltage_samples++;
         if(num_baseline_voltage_samples <=
TOTAL_BASELINE_BATT_VOLT_SAMPLES_TO_RECORD) {
```

```
            /* add this sample to the total so that an average
baseline can be obtained */
            sum_batt_volt_samples += battery_voltage;
         }
         else {
            setBattVoltageBaseline();
            need_baseline_voltage = FALSE;
            call Leds.led0Off();
         }
      }
      else {
         checkBattVoltageLevel(battery_voltage);
      }
   }
#endif /* LOW_BATTERY_INDICATION */


   /* The MSP430 CPU is byte addressed and little endian */
   /* packets are sent little endian so the word 0xABCD will
be sent as bytes 0xCD 0xAB */
   void preparePacket() {
      uint16_t *p_packet, *p_ADCsamples, crc;

      tx_packet[1] = FRAMING_BOF;
      tx_packet[2] = SHIMMER_REV1;
      tx_packet[3] = PROPRIETARY_DATA_TYPE;
      tx_packet[4]++; /* increment sequence number */

      tx_packet[7] = FIXED_PAYLOAD_SIZE;

      p_packet = (uint16_t *)&tx_packet[8];

      if(current_buffer == 1) {
         p_ADCsamples = &sbuf0[0];
         tx_packet[5] = timestamp0 & 0xff;
         tx_packet[6] = (timestamp0 >> 8) & 0xff;
      }
      else {
         p_ADCsamples = &sbuf1[0];
         tx_packet[5] = timestamp1 & 0xff;
         tx_packet[6] = (timestamp1 >> 8) & 0xff;
      }
      /* copy all the data samples into the outgoing packet */
      *p_packet++ = *p_ADCsamples++; //tx_packet[8]
      *p_packet++ = *p_ADCsamples++; //tx_packet[10]
      *p_packet++ = *p_ADCsamples++; //tx_packet[12]
      *p_packet++ = *p_ADCsamples++; //tx_packet[14]
      *p_packet++ = *p_ADCsamples++; //tx_packet[16]
      *p_packet++ = *p_ADCsamples++; //tx_packet[18]
      *p_packet++ = *p_ADCsamples++; //tx_packet[20]
      *p_packet = *p_ADCsamples; //tx_packet[22]

      /* spare room in the packet so send the battery voltage
data */
      if(current_buffer == 1) {
         tx_packet[18] = (sbuf0[5]) & 0xff;
```

```
         tx_packet[19] = ((sbuf0[5]) >> 8) & 0xff;
      }
      else {
         tx_packet[18] = (sbuf1[5]) & 0xff;
         tx_packet[19] = ((sbuf1[5]) >> 8) & 0xff;
      }

      crc = calc_crc(&tx_packet[2], (FIXED_PACKET_SIZE-
FRAMING_SIZE));
      tx_packet[FIXED_PACKET_SIZE - 2] = crc & 0xff;
      tx_packet[FIXED_PACKET_SIZE - 1] = (crc >> 8) & 0xff;
      tx_packet[FIXED_PACKET_SIZE] = FRAMING_EOF;
   }

   task void sendSensorData() {
#ifdef LOW_BATTERY_INDICATION
      monitorBattery();
#endif /* LOW_BATTERY_INDICATION */

      atomic if(enable_sending) {
         preparePacket();

         /* send data over the air */
         call Bluetooth.write(&tx_packet[1],
FIXED_PACKET_SIZE);
         atomic enable_sending = FALSE;
      }
   }

   task void startSensing() {
      call ActivityTimer.startPeriodic(1000);

      call SampleTimer.startPeriodic(sample_freq);
   }

   task void sendPersonality() {
      atomic if(enable_sending) {
         /* send data over the air */
         call Bluetooth.write(&personality[0], 17);
         atomic enable_sending = FALSE;
      }
   }

   task void stopSensing() {
      call SetupTimer.stop();
      call SampleTimer.stop();
      call ActivityTimer.stop();
      call shimmerAnalogSetup.stopConversion();
      call DMA0.stopTransfer();
      call Leds.led1Off();
   }

   async event void Bluetooth.connectionMade(uint8_t status) {
      atomic enable_sending = TRUE;
      call Leds.led2On();
   }
```

```
   async event void Bluetooth.commandModeEnded() {
      atomic command_mode_complete = TRUE;
   }

   async event void Bluetooth.connectionClosed(uint8_t
reason){
      atomic enable_sending = FALSE;
      call Leds.led2Off();
      post stopSensing();
   }

   task void startConfigTimer() {
      call SetupTimer.startPeriodic(5000);
   }

   async event void Bluetooth.dataAvailable(uint8_t data){
      /* start capturing on ^G */
      if(7 == data) {
         atomic if(command_mode_complete) {
            post startSensing();
         }
         else {
            /* give config a chance, wait 5 secs */
            post startConfigTimer();
         }
      }
      else if (data == 1) {
         post sendPersonality();
      }

      /* stop capturing on spacebar */
      else if (data == 32) {
         post stopSensing();
      }
      else { /* were done */ }
   }

   event void Bluetooth.writeDone(){
      atomic enable_sending = TRUE;

#ifdef LOW_BATTERY_INDICATION
      if(linkDisconnecting) {
         linkDisconnecting = FALSE;
         /* signal battery low to master and let the master
disconnect the link */
         post sendBatteryLowIndication();
      }
#endif /* LOW_BATTERY_INDICATION */
   }

   event void SetupTimer.fired() {
      atomic if(command_mode_complete){
         call ActivityTimer.stop();
         post startSensing();
      }
```

```
   }

   event void ActivityTimer.fired() {
      atomic {
         /* toggle activity led every second */
         if(activity_led_on) {
            call Leds.led1On();
            activity_led_on = FALSE;
         }
         else {
            call Leds.led1Off();
            activity_led_on = TRUE;
         }
      }
   }

   event void SampleTimer.fired() {
      call shimmerAnalogSetup.triggerConversion();
   }

   async event void DMA0.transferDone(error_t success) {
      if(current_buffer == 0){
         call DMA0.repeatTransfer((void*)ADC12MEM0_,
(void*)sbuf1, NBR_ADC_CHANS);
         atomic timestamp1 = call LocalTime.get();
       current_buffer = 1;
      }
      else {
         call DMA0.repeatTransfer((void*)ADC12MEM0_,
(void*)sbuf0, NBR_ADC_CHANS);
         atomic timestamp0 = call LocalTime.get();
       current_buffer = 0;
      }
      post sendSensorData();
   }
}
```

7.1.2 *ECGAppC.nc*

The configuration ECGAppC is illustrated.

<u>ECGAppC.nc</u>

```
/*
 * Copyright (c) 2007, Intel Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or
without
```

```
 * modification, are permitted provided that the following
conditions are met:
 *
 * Redistributions of source code must retain the above
copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above
copyright notice,
 * this list of conditions and the following disclaimer in the
documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of the Intel Corporation nor the names of
its contributors
 * may be used to endorse or promote products derived from
this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Author: Adrian Burns
 *         November, 2007
 */

 /* This app uses Bluetooth to stream 3 Accelerometer channels
and 2 ECG channels
    of data to a BioMOBIUS PC application.
    Tested on SHIMMER Base Board Rev 1.3, SHIMMER ECG board
Rev 1.1. */

/*
 * @author Adrian Burns
 * @date November, 2007
 *
 * @author Mike Healy
```

```
 * @date May 13, 2009 - ported to TinyOS 2.x
 */


configuration ECGAppC {
}
implementation {
  components MainC, ECGC;
  ECGC -> MainC.Boot;

  components FastClockC;
  ECGC.FastClockInit -> FastClockC;
  ECGC.FastClock     -> FastClockC;

  components LedsC;
  ECGC.Leds -> LedsC;

  components new TimerMilliC() as SampleTimer;
  ECGC.SampleTimer -> SampleTimer;
  components new TimerMilliC() as SetupTimer;
  ECGC.SetupTimer    -> SetupTimer;
  components new TimerMilliC() as ActivityTimer;
  ECGC.ActivityTimer -> ActivityTimer;

  components Counter32khz32C as Counter;
  components new CounterToLocalTimeC(T32khz);
  CounterToLocalTimeC.Counter -> Counter;
  ECGC.LocalTime -> CounterToLocalTimeC;

  components RovingNetworksC;
  ECGC.BluetoothInit -> RovingNetworksC.Init;
  ECGC.BTStdControl -> RovingNetworksC.StdControl;
  ECGC.Bluetooth    -> RovingNetworksC;

  components shimmerAnalogSetupC, Msp430DmaC;
  MainC.SoftwareInit -> shimmerAnalogSetupC.Init;
  ECGC.shimmerAnalogSetup -> shimmerAnalogSetupC;
  ECGC.DMA0 -> Msp430DmaC.Channel0;

}
```

7.1.3 *ECG.h*

ECG.h

```
/*
 * Copyright (c) 2007, Intel Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or
without
```

```
 * modification, are permitted provided that the following
conditions are met:
 *
 * Redistributions of source code must retain the above
copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above
copyright notice,
 * this list of conditions and the following disclaimer in the
documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of the Intel Corporation nor the names of
its contributors
 * may be used to endorse or promote products derived from
this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Author: Adrian Burns
 *         November, 2007
 */
/*
 * @author Adrian Burns
 * @date November, 2007
 *
 * @author Mike Healy
 * @date May 13, 2009 - ported to TinyOS 2.x
 */

#ifndef ECG_H
#define ECG_H

enum {
```

```
  NUM_ACCEL_CHANS = 3
};

enum {
  NUM_GYRO_CHANS = 3
};

enum {
  SHIMMER_REV1 = 0
};

enum {
  PROPRIETARY_DATA_TYPE = 0xFF,
  STRING_DATA_TYPE = 0xFE
};

enum {
  SAMPLING_1000HZ = 1,
  SAMPLING_500HZ = 2,
  SAMPLING_250HZ = 4,
  SAMPLING_200HZ = 5,
  SAMPLING_166HZ = 6,
  SAMPLING_125HZ = 8,
  SAMPLING_100HZ = 10,
  SAMPLING_50HZ = 20,
  SAMPLING_10HZ = 100,
  SAMPLING_0HZ_OFF = 255
};

enum {
  FRAMING_SIZE      = 0x4,
  FRAMING_CE_COMP   = 0x20,
  FRAMING_CE_CE     = 0x5D,
  FRAMING_CE        = 0x7D,
  FRAMING_BOF       = 0xC0,
  FRAMING_EOF       = 0xC1,
  FRAMING_BOF_CE    = 0xE0,
  FRAMING_EOF_CE    = 0xE1,
};

#endif // ECG_H
```

7.2 Discrete Fourier Transform(DFT)

The applications we are about to describe are an extension of the ECG application. After sampling the default sensor , they compress the ECG signal samples with a Discrete Fourier Transform (DFT) and then broadcast the modified message over the radio every N (N=8 or N=16) readings.Before we proceed to their analysis first we need to address some aspects of the Fourier Transform.

7.2.1 *Fast Fourier Transform(FFT)*

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete fourier transform (DFT) and it's inverse. There are many distinct FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory. A DFT decomposes a sequence of values into components of different frequencies but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly. Computing a DFT of N points in the naive way, using the definition, takes O(N2) arithmetical operations , while an FFT can compute the same result in only O(Nlog N) operations. The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions—in practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to N/ log (N). This huge improvement made many DFT-based algorithms practical; FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers. The most well known FFT algorithms depend upon the factorization of N, but there are FFTs with O (Nlog N) complexity for all N, even for prime N. Many FFT algorithms only depend on the fact that is an Nth primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms. Fast Fourier Transform is a fundamental transform in digital signal processing with applications in frequency analysis, signal processing etc [7]. The periodicity and symmetry properties of DFT are useful for compression .The uth FFT coefficient of length N sequence {f(x)} is de-fined as in (1):

$$F(u) = \sum_{x=0}^{N-1} f(x) e^{-j2\pi ux/N} \qquad (1)$$

Where $u = 0,1,...., N-1.$

And its inverse transform is calculated from (2):

$$f(x) = \frac{1}{N} \sum_{u=0}^{N-1} f(x) e^{j2\pi ux/N} \qquad (2)$$

Where $x = 0,1,...., N-1.$

7.2.2 *FFT on Real-valued Data*

In our project the ECG data we collect are real values. An FFT algorithm on real values contains some interesting properties.

For real-valued input data $f_n \in \Re$ (i.e $f_n^* = \overline{fn} = f_n$)

$$F_k = \frac{1}{N}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n e^{-i2\pi nk/N} = \frac{1}{N}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n(\cos(\frac{2\pi nk}{N}) - i\sin(\frac{2\pi nk}{N}))$$

And so the real part is:

$$\mathrm{Re}\{F_k\} = \frac{1}{N}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n \cos(\frac{2\pi nk}{N})$$

and the imaginary part is:

$$\mathrm{Im}\{F_k\} = \frac{1}{N}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n \sin(\frac{2\pi nk}{N})$$

Also , only N independent real-valued coefficients are necessary since:

$$F_k^* = \frac{1}{N}\sum f_n^* \{\omega_N^{-nk}\}^* == \frac{1}{N}\sum f_n\, \omega_N^{-n(-k)} \quad = F_{-k}$$

So if N the number of samples , then DFT(0) and DFT(N/2) have only real part whilst DFT(1) to DFT(N/2-1) have both real and imaginary part but they are symmetric to samples DFT(N/2+1) to DFT(N-1).

And so we get that the DFT of N independent real-valued samples f_n is:

$\{f_{-N/2+1},\ldots,\quad f_0\quad \ldots,f_{N/2}\}$

DFT $\Downarrow$ $\Uparrow$ IDFT

$\{F_0,\mathrm{Re}\{F_1\},\mathrm{Im}\{F_1\},\quad \ldots.,\quad \mathrm{Re}\{F_{N/2-1}\},\mathrm{Im}\{F_{N/2-1}\},F_{N/2}\}$

In our project we will use both 8-point DFT (N=8) and 16-point DFT so the sensors will send:

8-point

$\{F_0,\mathrm{Re}\{F_1\},\mathrm{Im}\{F_1\},\mathrm{Re}\{F_2\},\mathrm{Im}\{F_2\},\mathrm{Re}\{F_3\},\mathrm{Im}\{F_3\},\mathrm{Re}\{F_4\},\mathrm{Im}\{F_4\},F_5\}$

16-point

$\{F_0,\mathrm{Re}\{F_1\},\mathrm{Im}\{F_1\},\mathrm{Re}\{F_2\},\mathrm{Im}\{F_2\},\mathrm{Re}\{F_3\},\mathrm{Im}\{F_3\},\mathrm{Re}\{F_4\},\mathrm{Im}\{F_4\}, \mathrm{Re}\{F_5\},\mathrm{Im}\{F_5\}$
$,\mathrm{Re}\{F_6\},\mathrm{Im}\{F_6\},\mathrm{Re}\{F_7\},\mathrm{Im}\{F_7\},F_8\}$

7.2.3 *C Code for RDFT*

The C code for RDFT (8-point) is presented below. It implements the algorithm described above.

```
/*
 * Real-input Discrete Fourier Transform (RDFT)
 * Using standard math.h functions
 * Author: George Economakos
 *
 */

#include "math.h"

/*
 * Computes the discrete Fourier transform (DFT) of the
given 8 element vector of real values(inreal[8]).*/

void rdft(uint16_t inreal[8]) {
      int k;
      int t;
      uint16_t outsymmetric[8];
      double sumreal;
      double sumimag;

      /* For k=0, sin(0)=0, cos(0)=1 so we have only real
output */
      sumreal = 0.0;
      for (t = 0; t < 8; t++)    /* For each input element
*/
        sumreal +=  (double)inreal[t];
      outsymmetric[0] = (uint16_t)sumreal;

      /* For outputs 1, 2 and 3 which are symmetric to 5,
6 and 7 and have both real and imaginary output, put one
next to the other in the output array */
      for (k = 1; k < 4; k++) {
            sumreal = 0.0;
            sumimag = 0.0;
            for (t = 0; t < 8; t++) {  /* For each input
element */
                  sumreal +=  (double)inreal[t]*cos(2*M_PI *
t * k / 8);
                  sumimag += -(double)inreal[t]*sin(2*M_PI *
t * k / 8);
            }
            outsymmetric[2*k-1] = (uint16_t)sumreal;
            outsymmetric[2*k] = (uint16_t)sumimag;
      }
```

```
      /* For k=4, sin(M_PI*t)=0, cos(M_PI*t)=+1 or -1  so
we have only real output */
      sumreal = 0;
      for (t = 0; t < 8; t++)    /* For each input element
*/
        sumreal +=  (double)inreal[t]*cos(M_PI * t);
      outsymmetric[7] = (uint16_t)sumreal;

      /* Copy output to input */
      for (t = 0; t < 8; t++)
        inreal[t] = outsymmetric[t];
}
```

The 16-point rdft is the exact same with the small difference that the input (inreal[16]) and output (outsymmetric[16]) vectors contain 16 elements and the loops are adjusted accordingly.

7.3 ECG-cordic

In the ECG-cordic app ,the DFT is implemented with the cordic algorithm.We used both an 8-point and 16-point DFT so as to compare the results.

7.3.1 *Cordic Algorithm*

CORDIC stands for COordinate Rotation DIgital Computer and it is a class of shift-adds algorithms for rotating vectors in a plane , which is usually used for the calculation of trigonometric functions , multiplication , division and conversion between binary and mixer radix number systems of DSP applications such as the Fourier Transform. All of the trigonometric functions can be computed or derived from functions using vector rotations, as will be discussed in the following sections. Vector rotation can also be used from polar to rectangular and rectangular to polar conversions, for vector magnitude, and as a building block in certain transforms such as DFT and DCT .The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shifts and adds. The algorithm credited to Jack E. Volder in the year 1959 is derived from the general rotation transform:

$x' = x \cos\phi - y \sin\phi$

$y' = y \cos\phi + x \sin\phi$

this rotates a vector in a Cartesian plane by the angle ϕ.

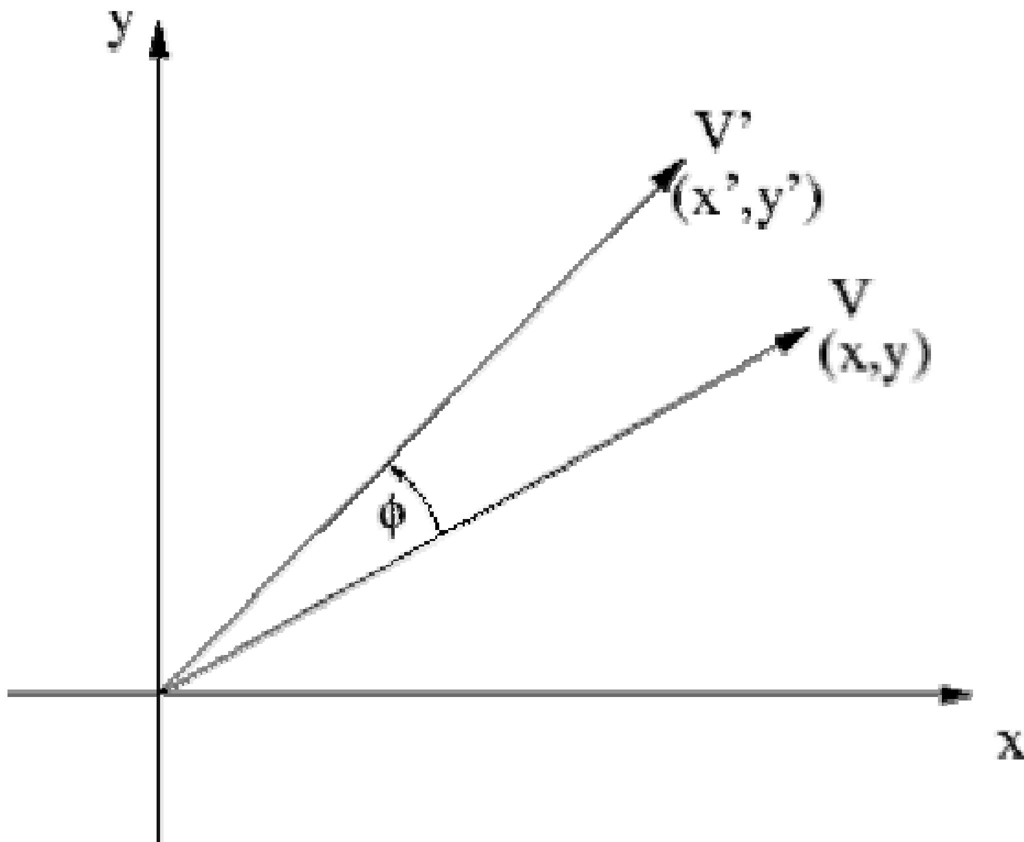

Figure 7.1: *Rotation of a vector V by angle φ*

These can be rearranged so that :

$$x' = \cos\phi[x - y\tan\phi]$$

$$y' = \cos\phi[y + x\tan\phi]$$

So far nothing is simplified. However if the rotation angles are restricted so that

$\tan\phi = \pm\ 2^{-i}$,the multiplication by the tangent term is reduced to simple shift operation. Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary operations. If the decision at each iteration i , is which direction to rotate rather than whether or not to rotate, then the cos(δi) term becomes a constant because cos(δi)= cos(-δi). The iterative rotation can now be expressed as:

$$x_{i+1}=K_i[x_i-y_i\times d_i\times 2^{-i}]$$

$$y_{i+1}=K_i[y_i+x_i\times d_i\times 2^{-i}]$$

where:

$$K_i=\cos(\tan^{-i}2^{-i})=\frac{1}{\sqrt{1+2^{-i}}}$$

$$d_i=\pm 1$$

Removing the scale constant from the iterative equations yields a shift add algorithm for vector rotation. The product of the Ki's can be applied elsewhere in the system or treated as a part of the system processing gain. The product approaches 0.6073 as the number of iteration goes to infinity. A good way to implement the $k=\prod_{i=0}^{n-1} K_i$ factor is to initialize the iterative rotation with a vector of length |k| which compensates the gain in the CORDIC algorithm.Therefore the rotation algorithm has a gain An of approximately 1.647.The exact gain depends on the no of iterations and obeys the relation:

$$A_n=\prod_n \sqrt{1+2^{-2i}}$$

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement system based on binary arctangents. Conversions between the angular systems and any other can be accomplished using a look up. A better conversion method uses an additional adder-subtract or that accumulate the elementary rotation angles at each iteration. The elementary angles can be expressed in any convenient angular unit. Those angular values are supplied by a small look up table or are hardwired depending on the application. The angle accumulator adds a third difference equation to the CORDIC algorithm:

$$z_{i+1}=z_i-d_i\times\tan^{-1}(2^{-i})$$

Obviously, in cases where the angle is useful in the arc tangent base, this extra element is not needed. The CORDIC rotator is normally operated in one of two models.The first called rotation by Volder rotates the input vector by a specified angle. The second mode called vectoring rotates the input vector to the x axis while recording the angle required to make that rotation.

In our case , given the special features of a real-valued DFT , we will use a cordic algorithm in 16 bit fixed point math to compute our DFT coefficients.

Rotation Mode

In rotation mode,the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of

the residual angle in the angle accumulator. The decision at each iteration is therefore based on the sign of the residual angle after each step. Naturally, if the input angle is already expressed in the binary arctangent base, the angle accumulator may be eliminated. For rotation mode, the CORDIC equations are:

$$x_{i+1}=x_i-y_i \times d_i \times 2^{-i}$$

$$y_{i+1}=y_i+x_i \times d_i \times 2^{-i}$$

$$z_{i+1}=z_i-d_i \times \tan^{-1}(2^{-i})$$

where $d_i=-1$ if $z_i<0$ and $d_i=1$ otherwise

which provides the following result

$$x_n=A_n[x_0 \cos z_0-y_0 \sin z_0]$$

$$y_n=A_n[y_0 \cos z_0+x_0 \sin z_0]$$

$$z_n=0$$

$$A_n=\prod_n \sqrt{1+2^{-2i}}$$

Below we present the table of the cordic uses for a variety of widely used functions:

OPERATION	MODE	INITIALIZE	DIRECTION
Sine, Cosine	Rotation	$x=1/A_n$, $y=0$, $z=\alpha$	Reduce z to Zero
Polar to Rect.	Rotation	$x=(1/A_n)X_{mag}$, $y=0$, $z=X_{phase}$	Reduce z to Zero
General Rotation	Rotation	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=\alpha$	Reduce z to Zero
Arctangent	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Vector Magnitude	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Rect. to Polar	Vector	$x=(1/A_n)x_0$, $y=(1/A_n)y_0$, $z=0$	Reduce y to Zero
Arcsine, Arccosine	Vector	$x=(1/A_n)$, $y=0$, $arg=\sin\alpha$ or $\cos\alpha$	Reduce y to Value in *arg* Register

Table 7.1 *Cordic Uses*

In our case we will use cordic to calculate sine and cosine functions , so we will initialize as follows:

$x=1/A_n=K_n=0.607$, $y=0$ and $z=a$(current angle for calculation)

7.3.2 *C code for sine cosine functions with Cordic algorithm*

The C code for sine and cosine calculations with the help of cordic algorithm is as follows:

```
//Cordic in 16 bit signed fixed point math
//Function is valid for arguments in range -pi/2 -- pi/2
//for values pi/2--pi: value = half_pi-(theta-half_pi)
and similarly for values -pi---pi/2
//
// 1.0 = 1073741824
// 1/k = 0.6072529350088812561694
// pi = 3.1415926536897932384626
//Constants
#define cordic_1K 0x26DD //* 1/An=0.607 in hexadecimal*//
#define half_pi 0x6487   //* pi/2 in hexadecimal*//
#define MUL 16384.000000
#define CORDIC_NTAB 16   //* 16-bit point math *//
Int32_t cordic_ctab [] = {0x3243, 0x1DAC, 0x0FAD, 0x07F5,
0x03FE, 0x01FF, 0x00FF, 0x007F, 0x003F, 0x001F, 0x000F,
0x0007, 0x0003, 0x0001, 0x0000, 0x0000, };//* tan-1(2-i)
for i=0 to 15*//

/*theta is the current angle for calculation, s is the
sine , c the cosine and n=16*/

void cordic(int32_t theta, int32_t *s, int32_t *c, int n)
{
  int32_t k, d, tx, ty, tz;
  int32_t x=cordic_1K,y=0,z=theta;
  n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;
  for (k=0; k<n; ++k) /*16 iterations*/
  {
    d = z>>15;
    //get sign. for other architectures, you might want
to use the more portable version
    //
    d = z>=0 ? 0 : -1;
    tx = x - (((y>>k) ^ d) - d); /*xi+1=xi-yidi2-i */
    ty = y + (((x>>k) ^ d) - d); /*yi+1=yi+xidi2-i */
    tz = z - ((cordic_ctab[k] ^ d) - d);
/*zi+1=zi-ditan-1(    2-i )*/
    x = tx; y = ty; z = tz;
  }
 *c = x; *s = y;
}
```

7.3.3 *ECG-cordic app*

The ECG-cordic app is an extended version of the ECG app. More specifically the ECG-cordicC.nc:

A) It contains the declaration of the cordic app:

```
void cordic(int32_t theta, int32_t *s, int32_t *c, int n)
```

B)along with its initial definitions mentioned in the code above:

```
#define cordic_1K 0x26DD //* 1/An=0.607 in hexadecimal*//
#define half_pi 0x6487   //* pi/2 in hexadecimal*//
#define MUL 16384.000000
#define CORDIC_NTAB 16   //* 16-bit point math *//
Int32_t cordic_ctab [] = {0x3243, 0x1DAC, 0x0FAD, 0x07F5,
0x03FE, 0x01FF, 0x00FF, 0x007F, 0x003F, 0x001F, 0x000F,
0x0007, 0x0003, 0x0001, 0x0000, 0x0000, };//* tan-1(2-i)
for i=0 to 15*//
```

C)the inclusion of the math.h library:

```
#include "math.h"
```

D) the declaration of the rdft function with the difference that the sine and cosine functions are calculated with the cordic algorithm. To avoid all kinds of confusion the modified rdft with cordic algorithm is presented below:

```
void rdft(uint16_t inreal[8]) {
     int k;
     int t;
     uint16_t outsymmetric[8];
     uint16_t sumreal;
     uint16_t sumimag;
     int32_t csin; /*parameters for the cordic function*/
     int32_t ccos;

     /* For k=0, sin(0)=0, cos(0)=1 so we have only real
output */
     sumreal = 0;
     for (t = 0; t < 8; t++)    /* For each input element
*/
       sumreal += inreal[t];
     outsymmetric[0] = sumreal;

     /* For outputs 1, 2 and 3 which are symmetric to 5,
6 and 7 and have both real and imaginary output, put one
next to the other in the output array */
```

```
    for (k = 1; k < 4; k++) {
         sumreal = 0;
         sumimag = 0;
         for (t = 0; t < 8; t++) {  /* For each input
element */
              cordic((int32_t)(2*M_PI * t * k / 8),
&csin, &ccos, 16);
              sumreal +=  inreal[t]*((uint16_t)ccos);
              sumimag += -inreal[t]*((uint16_t)csin);
         }
         outsymmetric[2*k-1] = sumreal;
         outsymmetric[2*k] = sumimag;
    }

    /* For k=4, sin(M_PI*t)=0, cos(M_PI*t)=+1 or -1  so
we have only real output */
    sumreal = 0;
    for (t = 0; t < 8; t++)   /* For each input element
*/
      cordic((int32_t)(M_PI * t), &csin, &ccos, 16);
      sumreal +=  inreal[t]*((uint16_t)ccos);
    outsymmetric[7] = sumreal;

    /* Copy output to input */
    for (t = 0; t < 8; t++)
      inreal[t] = outsymmetric[t];
}
```

E)The call of the rdft function in the body of the function preparePacket() before the data is copied into the outgoing packet

The ECG-cordicAppC.nc is the same with the ECGAppC.nc and so is the ECG-cordic.h file.The 16-point ECG-cordic is the same but uses the 16-point rdft.

7.4 ECG-lookup

The ECG-lookup app is another extension of the ECG app only that in this case the trigonometric functions are calculated via a look-up table(LUT).

7.4.1 *Look-Up Table (LUT)*

Look-Up Tables are a technique commonly used to accelerate numeric processing in applications with demanding timing requirements. They are often used in Image Processing and DSP (Digital Signal Processing) applications. The basic idea is to pre-compute the result of complex operations that are or can be expressed as a function of an integer value. The pre-computed results are typically stored in an array, which is used at run-time instead of performing the whole, time-consuming operation.

Simply put, a Look-Up-Table (LUT) is an array that holds a set of pre-computed results for a given operation. This array provides acces to the results in a way that is faster than computing each time the result of the given operation.

LUT's are typically used in real-time data acquisition and processing systems (often embedded systems), since these types of systems impose demanding and strict timing restriction. An important detail to consider is the fact that LUT's require a considerable amount of execution time to initialize the array (to pre-compute the results). In real-time systems, it is in general acceptable to have a delay during the initialization of the application (after all, the application will be presumably run right after boot, which takes a few seconds anyway).

Two important design considerations of a lookup table are its size and its accuracy. It is not possible to create a table for every possible input value u. It is also not possible to be perfectly accurate due to the quantization of sin(u) $$ or cos(u) $$ lookup table values.

The logic is that after we map our angles from a certain range, for example in our case to 16-bit unsigned integer here [0..16384] is mapped to [0..2pi], then we perform a logical and with 16383 to remap the values into this range when doing intermediate computations using larger integer types. Finally the sine and cosine functions are calculated with the new index on the lookup table. The algorithm in C code is presented below.

7.4.2 *C code for Look-Up Table (LUT)*

```
/*lookup tables are declared initialised*/

uint16_t lsintab[2048];

uint16_t lcostab[2048];

/*lookup tables are initialised*/


for (=0; i<SinTableSize; i++)

{

lsintab[i] = sin((i / 2048) * M_PI);

lcostab[i] = sin((i / 2048) * M_PI);

}
```

```
/* value is the current angle s for sin and c for cos*/

void lookup(int32_t value, uint16_t *s, uint16_t *c)
{
/*angle is mapped to [0..16383]*/
  uint16_t index = (value >> 2);

  /*logical and with 16383 for value remaping in case og
larger integer types*/
  *c = lcostab[index & 16383];
  *s = lsintab[index & 16383];
}
```

7.4.3 *ECG-lookup app*

The ECG-lookup app is an extended version of the ECG app. More specifically the ECG-lookupC.nc:

A) It contains the declaration of the cordic app:

```
void lookup(int32_t value, uint16_t *s, uint16_t *c)
```

B)along with its initial definitions mentioned in the code above:

```
#define half_pi 0x6487   //* pi/2 in hexadecimal*//
#define MUL 16384.000000

/*lookup tables are declared */

uint16_t lsintab[2048];

uint16_t lcostab[2048];
```

C)`/*lookup tables are initialised*/`

```
for (=0; i<SinTableSize; i++)

{

lsintab[i] = sin((i / 2048) * M_PI);

lcostab[i] = sin((i / 2048) * M_PI);

}
```

D)the inclusion of the math.h library:

```
#include "math.h"
```

E) the declaration of the rdft function with the difference that the sine and cosine functions are calculated with the lookup-tables. To avoid all kinds of confusion the modified rdft with lookup-tables is presented below:

```
void rdft(uint16_t inreal[8]) {
      int k;
      int t;
      uint16_t outsymmetric[8];
      uint16_t sumreal;
      uint16_t sumimag;
      int32_t lsin; /*parameters for the cordic function*/
      int32_t lcos;

      /* For k=0, sin(0)=0, cos(0)=1 so we have only real
output */
      sumreal = 0;
      for (t = 0; t < 8; t++)    /* For each input element
*/
        sumreal += inreal[t];
      outsymmetric[0] = sumreal;

      /* For outputs 1, 2 and 3 which are symmetric to 5,
6 and 7 and have both real and imaginary output, put one
next to the other in the output array */
      for (k = 1; k < 4; k++) {
            sumreal = 0;
            sumimag = 0;
            for (t = 0; t < 8; t++) {  /* For each input
element */
                  lookup((int32_t)(2*M_PI * t * k / 8),
&lsin, &lcos);
                  sumreal +=  inreal[t]*((uint16_t)lcos);
                  sumimag += -inreal[t]*((uint16_t)lsin);
            }
            outsymmetric[2*k-1] = sumreal;
            outsymmetric[2*k] = sumimag;
      }

      /* For k=4, sin(M_PI*t)=0, cos(M_PI*t)=+1 or -1  so
we have only real output */
      sumreal = 0;
      for (t = 0; t < 8; t++)    /* For each input element
*/
        lookup((int32_t)(M_PI * t), &lsin, &lcos);
        sumreal +=  inreal[t]*((uint16_t)ccos);
```

```
        outsymmetric[7] = sumreal;
             /* Copy output to input */
        for (t = 0; t < 8; t++)
          inreal[t] = outsymmetric[t];
}
```

F)The call of the rdft function in the body of the function preparePacket() before the data is copied into the outgoing packet
The ECG-lookupAppC.nc is the same with the ECGAppC.nc and so is the ECG-lookup.h file.The 16-point ECG-cordic is the same but uses the 16-point rdft.

The ECG-lookupAppC.nc is the same with the ECGAppC.nc and so is the ECG-lookup.h file.The 16-point ECG-lookup is the same but uses the 16-point rdft.

7.5 ECG-lookup-interpolate

The final app we created is the ECG-lookup-interpolate app which is quite similar to the previews app but includes linear interpolation between sin(x) and sin(x+1) and cos(x) and cos(x+1)

7.5.1 *Linear Interpolation*

The linear interpolation using LUTs follows the same pattern as described previewsly with the ECG-lookup app only that in this case instead we interpolate sin(x) with sin(x+1) as illustrated in the graph below.Thus we get a better approximation of the result.

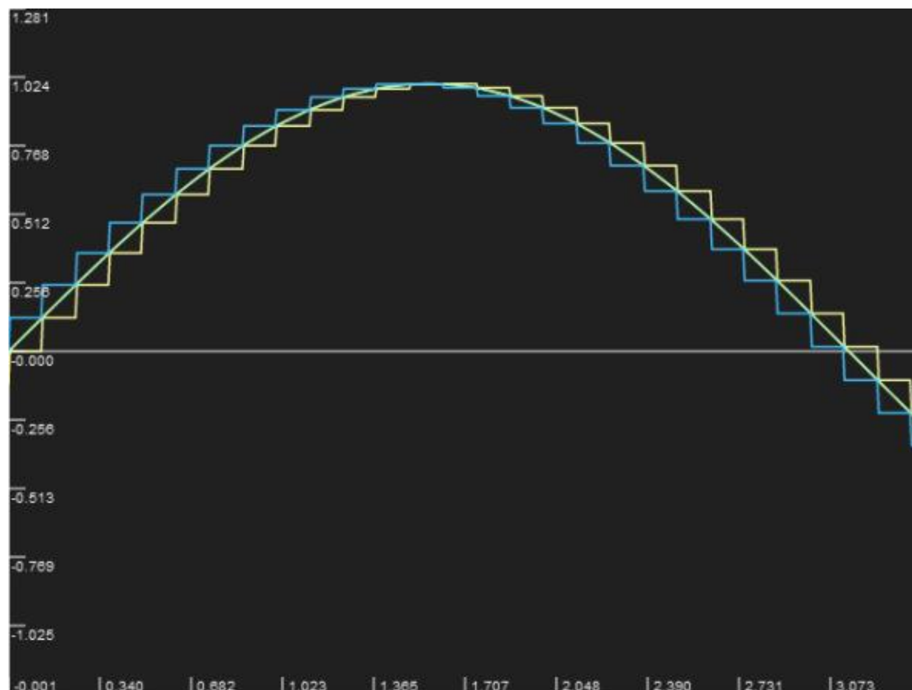

Figure 7.2 : *The yellow line is the LUT sin(x), and the blue one is LUT sin(x+1).*

7.5.2 *C code for Linear Interpolation*

```
#define half_pi 0x00006487
#define MUL 16384.000000
uint16_t lsintab[2048];
uint16_t lcostab[2048];

/*lookup tables are initialised*/

for (=0; i<SinTableSize; i++)

{

lsintab[i] = sin((i / 2048) * M_PI);

lcostab[i] = sin((i / 2048) * M_PI);

}


void lookup(int32_t value, uint16_t *s, uint16_t *c)
{
/*Angle is mapped tp [0..16383]*/
  uint16_t index = (value >> 2);
/*Lookup angle*/
  uint16_t left = lcostab[index & 16383];
/*Lookup angle+1*/
  uint16_t right = lcostab[(index+1) & 16383];
/*weight is computed*/
  uint16_t weight = (value & 3) / 4;
/*Linear interpolation for cosine*/
 *c = weight * (right - left) + left;
/*Lookup angle*/
  left = lsintab[index & 16383];
/*Lookup angle+1*/
 *right = lsintab[(index+1) & 16383];
/*Linear interpolation for sine*/
  *s = weight * (right - left) + left;
}
```

7.5.3 ***ECG-lookup-interpolate app***

The ECG-lookup- app is an extended version of the ECG app. More specifically the ECG-lookupC.nc:

A) It contains the declaration of the lookup-interpolate app:

```
void lookup-interpolate(int32_t value, uint16_t *s,
uint16_t *c)
```

B)along with its initial definitions mentioned in the code above:

```
#define half_pi 0x6487   //* pi/2 in hexadecimal*//
#define MUL 16384.000000

/*lookup tables are declared */

uint16_t lsintab[2048];

uint16_t lcostab[2048];
```

C)/*lookup tables are initialised*/

```
for (=0; i<SinTableSize; i++)

{

lsintab[i] = sin((i / 2048) * M_PI);

lcostab[i] = sin((i / 2048) * M_PI);

}
```

D)the inclusion of the math.h library:

```
#include "math.h"
```

E) the declaration of the rdft function with the difference that the sine and cosine functions are calculated with the lookup-interpolate tables. To avoid all kinds of confusion the modified rdft with lookup-interpolate tables is presented below:

```
void rdft(uint16_t inreal[8]) {
      int k;
      int t;
      uint16_t outsymmetric[8];
      uint16_t sumreal;
      uint16_t sumimag;
      int32_t lsin; /*parameters for the cordic function*/
      int32_t lcos;

      /* For k=0, sin(0)=0, cos(0)=1 so we have only real
output */
      sumreal = 0;
      for (t = 0; t < 8; t++)    /* For each input element
*/
```

```
        sumreal += inreal[t];
      outsymmetric[0] = sumreal;

      /* For outputs 1, 2 and 3 which are symmetric to 5,
6 and 7 and have both real and imaginary output, put one
next to the other in the output array */
      for (k = 1; k < 4; k++) {
           sumreal = 0;
           sumimag = 0;
           for (t = 0; t < 8; t++) {  /* For each input
element */
                lookup-interpolate((int32_t)(2*M_PI * t *
k / 8), &lsin, &lcos);
                sumreal +=  inreal[t]*((uint16_t)lcos);
                sumimag += -inreal[t]*((uint16_t)lsin);
           }
           outsymmetric[2*k-1] = sumreal;
           outsymmetric[2*k] = sumimag;
      }

      /* For k=4, sin(M_PI*t)=0, cos(M_PI*t)=+1 or -1  so
we have only real output */
      sumreal = 0;
      for (t = 0; t < 8; t++)   /* For each input element
*/
        lookup-interpolate((int32_t)(M_PI * t), &lsin,
&lcos);
        sumreal +=  inreal[t]*((uint16_t)ccos);
      outsymmetric[7] = sumreal;

      /* Copy output to input */
      for (t = 0; t < 8; t++)
        inreal[t] = outsymmetric[t];
}
```

F)The call of the rdft function in the body of the function preparePacket() before the data is copied into the outgoing packet.The ECG-lookup-interpolateAppC.nc is the same with the ECGAppC.nc and so is the ECG-lookupinterpolate.h file.The 16-point ECG-cordic is the same but uses the 16-point rdft.

The ECG-lookupinterpolateAppC.nc is the same with the ECGAppC.nc and so is the ECG-lookup-interpolate.h file.The 16-point ECG-lookup is the same but uses the 16-point rdft.

Chapter 8

Simulation

The shimmer platform is not compatible with the MSP430 and creates a lot of disfunctions with the AccelECG. For that reason we decided to simulate the Oscilloscope application located in the TinyOS libraries . Oscilloscope is a simple data-collection demo. It periodically samples the default sensor and broadcasts a message over the radio every 10 (in our case we switched to 8 and 16 for 8-point and 16-point DFT respectively) readings. These readings can be received by a BaseStation mote and displayed by the Java "Oscilloscope" application found in the java subdirectory. The sampling rate starts at 4Hz, but can be changed from the Java application. In addition to the Oscilloscope application , we derived 3 more applications by modifying the Oscilloscope's code. The Oscilloscope-cordic , the Oscilloscope-lookup2048 and the Oscilloscope-lookup2048-interpolate application. These four apps have the exact same add-ons and modifications with the respective ones we described in the previews chapters. They perform the same tasks and produce similar results , only that the Oscilloscope apps are easier to simulate with the telosb platform than the shimmer.For our simulation we will use the MSPSim.

8.1 Oscilloscope Simulation

First we simulate the Oscilloscope app.It sends ECG samples every 8 readings without doing any DFT.By executing the command:

```
build telosb
```

In the Oscilloscope app directory we compile our app on a telosb binary.For the 8 point,the ROM(code) bytes were computed 15860 and the RAM(data) bytes 492.Same results for the 16-point.In order to simulate our app with the MSPSim we execute in the build/telosb directory that was created the command:

```
mspsim main.exe
```

Below we present the **Duty Cycle** graph

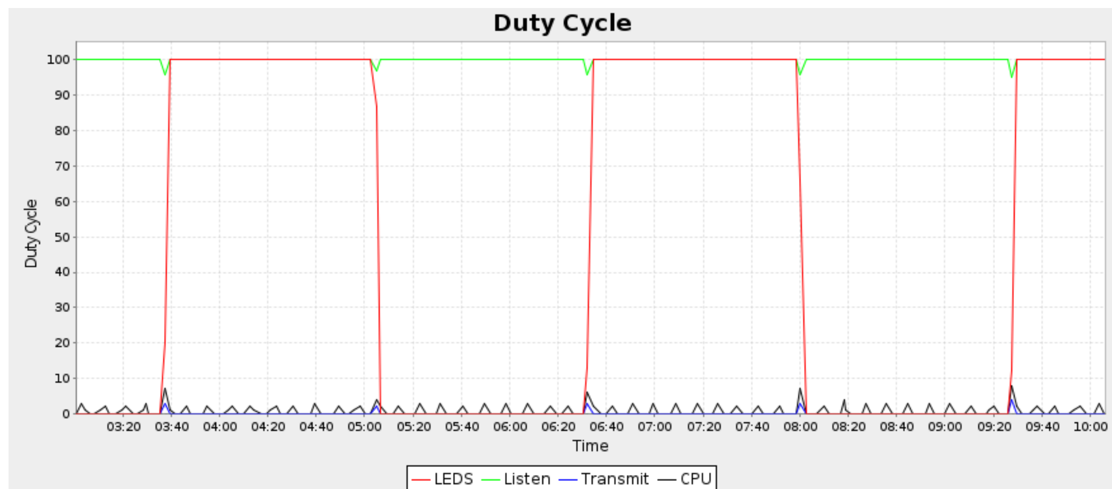

Figure 8.1 *Duty Cycle graph for the Oscilloscope app*

To get data for the CPU cycles we execute the command:

```
duty 10 "MSP430 Core.active"
```

which means we get 10 duty values/second. We let the command run for 3 seconds so we collect 30 duty values.

8.2 Oscilloscope-cordic Simulation

We compile and simulate Oscilloscope-cordic app on a telosb binary by following the same pattern as the Oscilloscope app mentioned above(same goes for the rest of the apps).For the 8-point ,the ROM(code) bytes were computed 18102 and the RAM(data) bytes 556 while for the 16-point , the ROM(code) bytes were computed 18110 and the RAM(data) bytes 556.

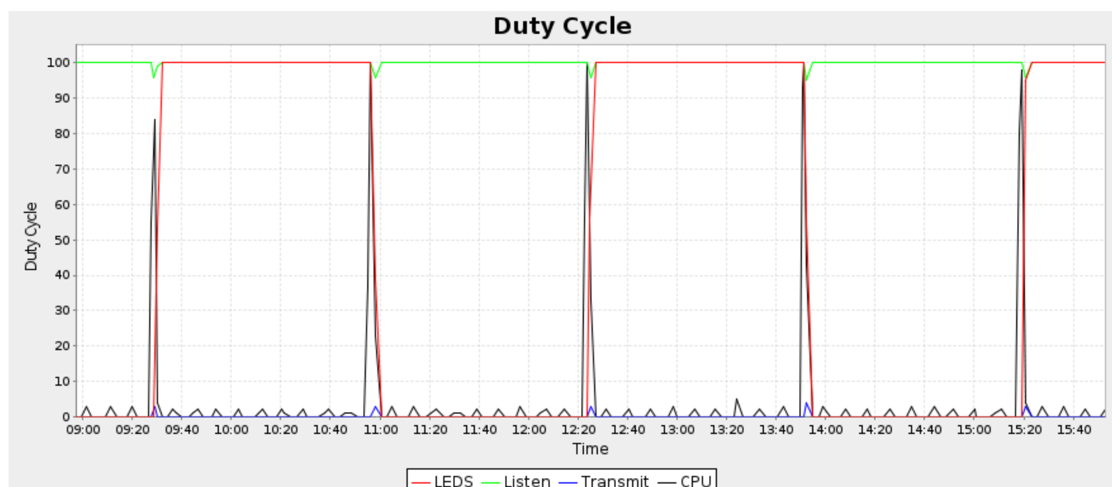

Figure 8.2 *Duty Cycle graph for the Oscilloscope-cordic app*

8.3 Oscilloscope-lookup Simulation

For the 8-point , the ROM(code) bytes were computed 15860 and the RAM(data) bytes 8684.Same figures for the 16-point.

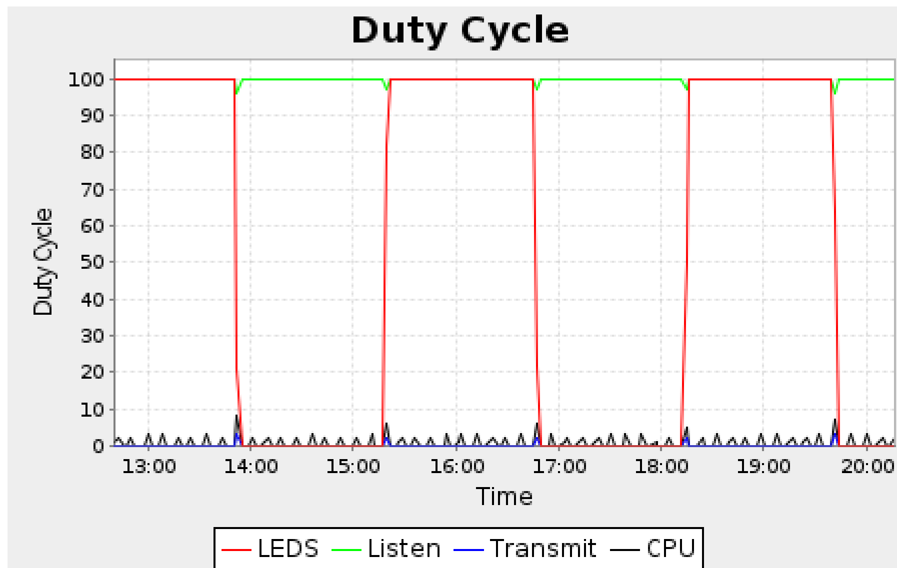

Figure 8.3 *Duty Cycle graph for the Oscilloscope-lookup app*

8.4 Osciloscope lookup-interpolate Simulation

For the 8-point , the ROM(code) bytes were computed 17806 and the RAM(data) bytes 8684 while for the 16-point the ROM(code) bytes were computed 17814 and the RAM(data) bytes 8684.

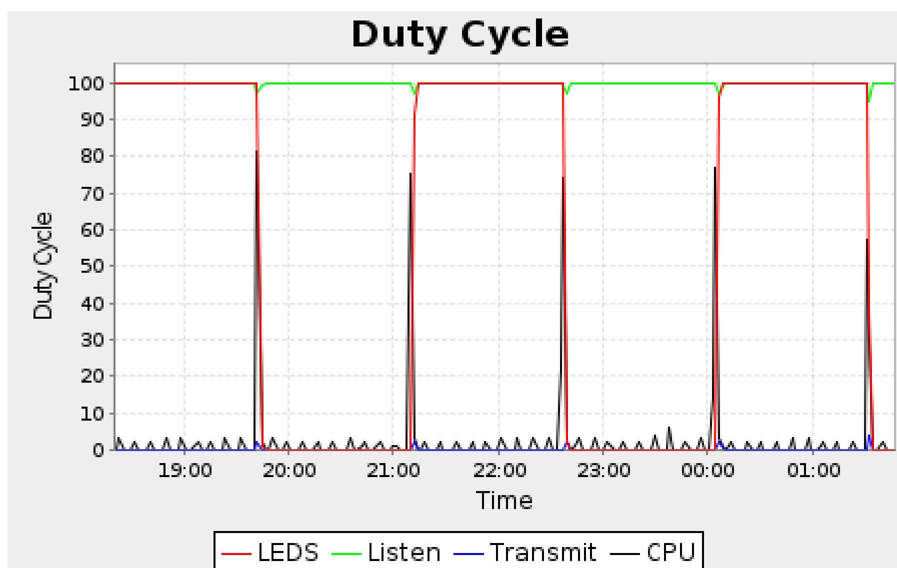

Figure 8.4 *Duty Cycle graph for the Oscilloscope-lookup-interpolate app*

8.5 Power Consumption

All the previews data we collected are illustrated in tables 8.1a to 8.1d

Oscilloscope	Oscilloscope-cordic	Oscilloscope-lookup	Oscilloscope-lookup-interpolate
0,21	**0,21**	**0,21**	**0,25**
2,13	**2,1**	**1,75**	**1,75**
0,21	**0,25**	**0,64**	**0,61**
1,7	**0,26**	**0,21**	**0,25**
0,64	**2,14**	**2,13**	**2,1**
0,21	**0,21**	**0,21**	**0,25**
2,18	**47,83**	**1,71**	**1,7**
0,21	**14,83**	**0,63**	**0,61**
1,7	**0,21**	**0,21**	**0,25**
0,64	**2,12**	**2,13**	**2,1**
0,21	**0,21**	**0,21**	**0,25**
2,13	**2,08**	**1,74**	**1,7**
0,21	**0,25**	**0,61**	**0,65**
1,74	**0,21**	**0,21**	**0,21**
0,61	**2,14**	**2,13**	**2,1**
0,21	**0,21**	**0,21**	**0,25**
2,13	**2,12**	**1,74**	**1,7**
0,21	**0,21**	**0,61**	**0,65**
1,74	**0,21**	**0,21**	**0,21**
0,61	**2,12**	**4,74**	**39,9**
0,21	**0,21**	**0,21**	**0,65**
2,13	**2,14**	**1,78**	**1,75**
0,21	**0,21**	**0,61**	**0,65**
4,34	**0,26**	**0,21**	**0,21**
0,61	**2,14**	**2,11**	**2,1**
0,21	**0,21**	**0,21**	**0,25**
2,18	**2,14**	**1,74**	**1,7**
0,21	**0,21**	**0,61**	**0,65**
1,74	**0,21**	**0,25**	**0,21**
0,59	**2,14**	**2,1**	**2,14**

Table 8.1a*: 8-point Raw duty values (10 per second - command used = duty 10 "MSP430 Core.active") for 3 seconds (30 values) - peak-to-peak.*

Oscilloscope	Oscilloscope-cordic	Oscilloscope-lookup	Oscilloscope-lookup-interpolate

2,11	0,21	0,25	0,78
2,12	0,78	2,1	1,57
2,13	1,55	0,25	0,21
2,14	0,21	0,21	2,1
2,15	2,18	2,18	0,25
2,16	0,21	0,21	0,78
2,17	0,76	2,1	1,57
2,18	1,57	0,25	0,21
2,19	0,21	0,21	2,15
2,2	2,14	2,13	0,25
2,21	0,21	0,21	14,64
2,22	14,62	2,1	100
2,23	100	0,25	52,13
2,24	100	0,21	2,12
2,25	41,88	2,13	0,21
2,26	0,21	0,21	0,78
2,27	0,81	4,7	1,57
2,28	1,54	0,25	0,21
2,29	0,21	0,21	2,12
2,3	2,14	2,13	0,21
2,31	0,21	0,21	0,76
2,32	0,81	2,13	1,57
2,33	1,54	0,21	0,21
2,34	0,21	0,21	2,14
2,35	2,18	2,18	0,21
2,36	0,21	0,21	0,78
2,37	0,81	2,13	1,57
2,38	1,54	0,21	0,21
2,39	0,21	0,21	2,18
2,4	2,14	2,13	0,21

Table 8.1b*: 16-point Raw duty values (10 per second - command used = duty 10 "MSP430 Core.active") for 3 seconds (30 values) - peak-to-peak.*

8-POINT	ROM (code) bytes	RAM (data) bytes	Average Duty Value	Maximum Duty Value
Oscilloscope	15860	492	1,07	4,3
Oscilloscope-cordic	18102	556	3	47,8
Oscilloscope-lookup	15860	8684	1,07	4,7
Oscilloscope-lookup-interpolate	17806	8684	2,26	40

Table 8.1c: *8-Point ROM(code) bytes,RAM(data) bytes, Average Duty Value and Maximum Duty Value table*

16-POINT	ROM (code) bytes	RAM (data) bytes	Average Duty Value	Maximum Duty Value
Oscilloscope	15860	492	2,26	2,4
Oscilloscope-cordic	18110	556	9,38	100
Oscilloscope-lookup	15860	8684	1,07	4,7
Oscilloscope-lookup-interpolate	17806	8684	6,46	100

Table 8.1d: *16-Point ROM(code) bytes,RAM(data) bytes, Average Duty Value and Maximum Duty Value table*

The Power Score(PS) we will calculate is the sum of the power consumption in the MSP430 and the Bluetooth(in our case CC2420) .For the MSP430 the Power Consumption is given in (1):

$$PC_{MSP430}=Y_1 * P_1 \quad (1)$$

Where Y_1=Maximum Duty Value and P_1=Average MSP430 Power Consumption

The Average MSP430 Power Consumption is calculated from the databook of the MSP430F1611 during program execution time , active mode at 8MHz:

I(AM)=5Ma , V_{CC}=3.6V

Average MSP430 Power Consumption=(5mA)x(3,6V)=18mW (2)

For the Bluetooth the PC is given in (3):

$$PC_{Bluetooth}=K*Y_2*P_2 \quad (3)$$

Where K=transmission reduction factor , Y_2=CC2420 duty cycle , P_3= CC2420 Power Consumption

As we concluded from the Paper " Efficient Algorithm for ECG Coding"[5] , The DFT provides Compression Ratio approximately 90%. This means that near 10% of the original data can be sent without any risk of critical data loss (since frequency-domain techniques are lossy and not lossless techniques). So in order to make sure we avoid losing critical data we can send half the DFT data . For that reason we introduce the transmission reduction factor K which reduces the CC2420 transmission according to the DFT data percentage we send.

The Average CC2420 Power Consumption is calculated from the databook of the CC2420 in transmission (TX) mode and it is:

I(TX)=18mA , V_{CC}=3.6V

Average CC2420 Power Consumption=(18mA)x(3.6V)=65mW (4)

Thus Total Power Score is :

$$PS= PC_{MSP430}+ PC_{Bluetooth} \quad (4)$$

$$(1),(2),(3),(4)\Rightarrow PS = 18mW*Y_1 + 65mW*Y_2*K \quad (5)$$

Depending on the percentage of DFT data we send , the transmission factor changes accordingly. For example if we use full transmission , the transmission factor is 100% , K=1.If we use half transmission , K=0.5 and for quarter transmission K=0.25. Of course when we do not use DFT then K=1.Finally the cc2420 duty cycle , Y_2=1 as we derived from the simulation.

All the results are places in tables 8.2a and 8.2b and Figures 8.5a , 8.5b

8-POINT	Power score with full transmission = P1*Y1+P2*Y2	Power score with half transmission = P1*Y1+K*P2*Y2	Power score with quarter transmission = P1*Y1+K*P2*Y2
Oscilloscope	**142,4**	**142,4**	**142,4**
Oscilloscope-cordic	**925,4**	**892,9**	**876,65**
Oscilloscope-lookup	**149,6**	**117,1**	**100,85**
Oscilloscope-lookup-interpolate	**785**	**752,5**	**736,25**

Table 8.2a:*Power Score Table for 8-point DFT*

16-POINT	Power score with full transmission = P1*Y1+P2*Y2	Power score with quarter transmission = P1*Y1+K*P2*Y2	Power score with quarter transmission = P1*Y1+K*P2*Y2
Oscilloscope	**108,2**	**108,2**	**108,2**
Oscilloscope-cordic	**1865**	**1832,5**	**1816,25**
Oscilloscope-lookup	**149,6**	**117,1**	**100,85**
Oscilloscope-lookup-interpolate	**1865**	**1832,5**	**1816,25**

Table 8.2b: *Power Score Table for 16-point DFT*

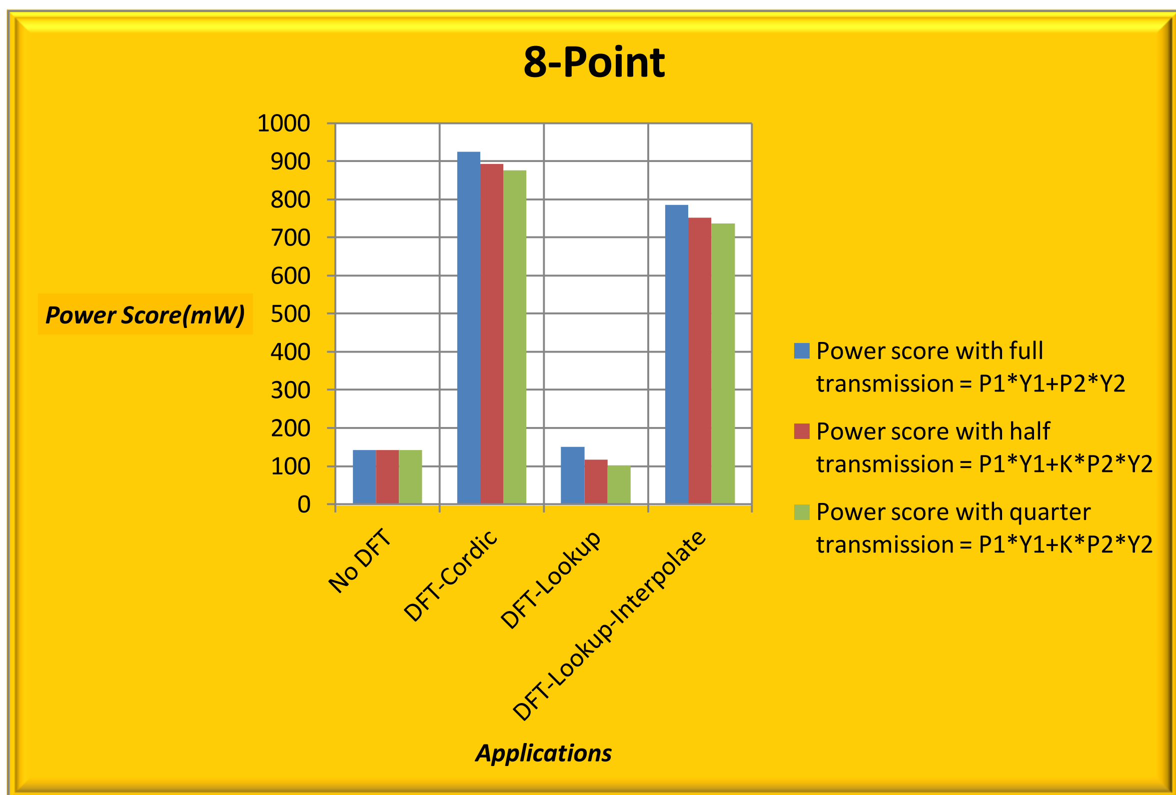

Figure 8.5a: *Power Score graph for 8-point DFT*

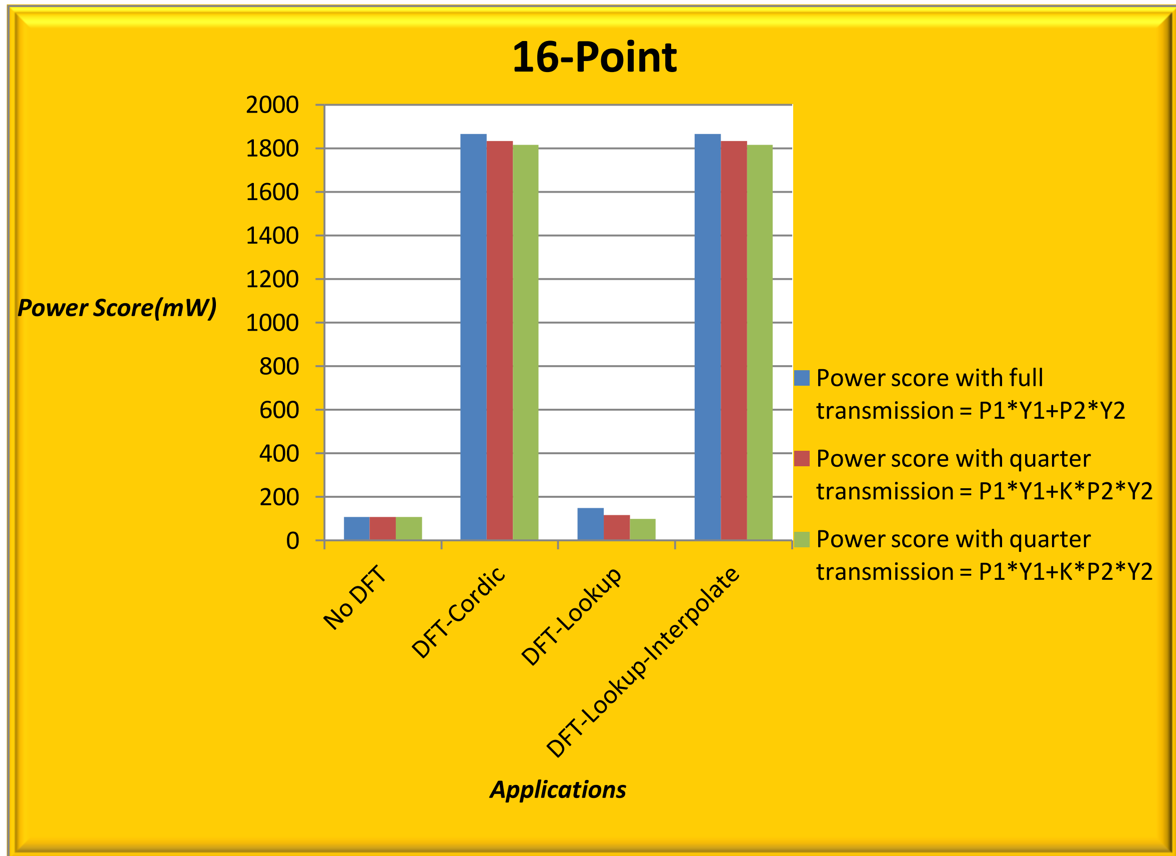

Figure 8.5b: *Power Score graph for 16-point DFT*

8.6 Data Analysis

From the Power Score graphs and tables it is obvious that 8-Point DFT is generally less power consuming than 16-point.More specifically:

No Compression to Compression Gain

8-Point:

a)The Osc to Osc-Cordic Gain is:

i)Full Transmission: (142.4-925.4)/142.4=-5.5=-550%

ii) Half Transmission: (142.4-892.9)/142.4=-5.27=-527%

iii) Quarter Transmission: (142.4-876.5)/142.2=-5.16=-516%

b)The Osc to Osc-Lookup Gain is:

i)Full Transmission: (142.4-149.6)/142.4=-0.05=-5%

ii) Half Transmission: (142.4-117.1)/142.4=0.18=18% (Positive Gain!)

iii)<u>Quarter Transmission:</u> (142.4-100.85)/142.2=0.29=29% (Positive Gain!)

c)The Osc to Osc-Lookup-Interpolate Gain is:

i)<u>Full Transmission:</u> (142.4-785)/142.4=-4.51=-451%

ii)<u>Half Transmission:</u> (142.4-752.5)/142.4=-4.28=-428%

iii)<u>Quarter Transmission:</u> (142.4-736.25)/142.2=-4.17=-417%

Results are shown in table 8.3

8-Point Gain	Full Transmission	Half Transmission	Quarter Transmission
Osc/Osc-Cordic	-550%	-5%	-451%
Osc /Osc-Lookup	-527%	18%	-428%
Osc / Osc -Lookup-Interpolate	-516%	29%	-417%

Table 8.3: *8-Point No Compression to Compression gain*

<u>16-Point:</u>

a)The Osc to Osc-Cordic Gain is:

i)<u>Full Transmission:</u> (108.2-1865)/108.2=-16.23=-1623%

ii)<u>Half Transmission:</u> (108.2-1832.5)/108.2=-15.94=-1594%

iii)<u>Quarter Transmission:</u> (108.2-1816.25)/108.2=-15.79=-1579%

b)The Osc to Osc-Lookup Gain is:

i)<u>Full Transmission:</u> (108.2-149.6)/108.2=-0.38= -38%

ii)<u>Half Transmission:</u> (108.2-117.1)/108.2=-0.08= -8%

iii)<u>Quarter Transmission:</u> (108.2-100.85)/108.2=0.07= 7% (Positive Gain!)

c)The Osc to Osc-Lookup-Interpolate Gain is:

i)<u>Full Transmission:</u> (108.2-1865)/108.2=-16.23=-1623%

ii)<u>Half Transmission:</u> (108.2-1832.5)/108.2=-15.94=-1594%

iii)<u>Quarter Transmission:</u> (108.2-1816.25)/108.2=-15.79=-1579%

Results are shown in table 8.4

16-Point Gain	Full Transmission	Half Transmission	Quarter Transmission
Osc/Osc-Cordic	-1623%	-38%	-1623%
Osc/Osc-Lookup	-1594%	-8%	-1594%
Osc/Osc-Lookup-Interpolate	-1579%	7%	-1589%

Table 8.4: *16-Point No Compression to Compression gain*

The data declare that in 3 cases (8-Point Osc-Lookup Half and Quarter Transmission and 16-point Osc-Lookup Quarter Transmission) we succeed Positive Gain. Due to the large amount of calculations made by Osc-Cordic and Osc-Lookup-Interpolate , their power score soars relatively to Osc and Osc-Lookup.

16-Point to 8-point Gain

a)The Osc app has no DFT so we expected the 16-point to be more power-consuming. The total gain is (108.2-142.4)/142.4= - 0.24= -24%.The gain is equal for all kinds of transmission since there is no compression.

b)The Osc-Cordic gain is :

i)Full Transmission: (1865-925.4)/1865=0.50=50%

ii) Half Transmission: (1832.5-892.9)/1832.5=0.51=51%

iii) Quarter Transmission: (1816.25-876.65)/1832.5=0.52=52%

c)The Osc-Lookup app gain is :

i)Full Transmission: (149.6-149.6)/149.6=0=0%

ii) Half Transmission: (117.1-117.1)/117.1=0=0%

iii) Quarter Transmission: (100.85-100.85)/100.85=0=0%

d)The Osc-Lookup-Interpolate gain is :

i)Full Transmission: (1865-785)/1865=0.58=58%

ii) Half Transmission: (1832.5-752.5)/1832.5=0.59=59%

iii) Quarter Transmission: (1816.25-736.25)/1816.25=0.59=59%

Results are shown in table 8.5

Application	Full Transmission	Half Transmission	Quarter Transmission
ECG	-24%	-24%	-24%
ECG-Cordic	50%	51%	52%
ECG-Lookup	0%	0%	0%
ECG-Lookup-Interpolate	58%	59%	59%

Table 8.5: *16-Point to 8-Point Gain*

Chapter 9

Results, Discussion and Future Work

9.1 Results-Discussion

In our project we dealt with ECG signal transmission and compression in a Wireless Sensor Network. We created software that transmits ECG signals with and without compression. Our goal was to evaluate different approaches to DFT compression and draw to conclusions on power consumption. With the assistance of the TinyOS open-source system we built 4 apps:

1) ECG App: Gathers ECG samples and sends them via Bluetooth

2) ECG-Cordic App: Gathers ECG samples , implements Discrete-Fourier-Transform on every 8 or 16 number of samples with the use of Cordic algorithm and sends the modified samples via Bluetooth.

3) ECG-Lookup App: Gathers ECG samples , implements Discrete-Fourier-Transform on every 8 or 16 number of samples with the use of Lookup-Tables(LUTs)and sends the modified samples via Bluetooth.

4) ECG-Lookup-Interpolate App: Gathers ECG samples , implements Discrete-Fourier-Transform on every 8 or 16 number of samples with the use of Lookup-Table Interpolation and sends the modified samples via Bluetooth.

These 4 apps were compiled on a Shimmer mote. In order to simulate properly their results we also created 4 new apps that execute the same task but with a much simpler code that produces more accurate results. We compiled and simulated these apps on a Telsob mote instead of a shimmer and collected data. From the data we collected we reached the conclusions above:

-The ECG-Cordic and ECG-Lookup-Interpolate apps are much more power consuming than the ECG-app and ECG-Lookup app both for 8-Point and 16-Point cases.

- Half-transmission equals to lower power consumption in both cases.

- 8-Point DFT is less power consuming than 16-Point .

- With the ECG- Lookup app we succeed less power consumption than the ECG app in half transmission in both 8-Point and 16-Point cases and in quarter transmission in 8-Point case. Thus the Lookup-Tables technique is the best suited for DFT compression.

9.2 Future Work

In our project we implemented a DFT Frequency Domain Technique to compress our ECG signal.As we mentioned in previews chapters , there is a variety of Frequency Domain Techniques. In the future we can compress the ECG signal with techniques such as:

1)The Discrete Cosine Transform(DCT) which was developed to approximate Karhunen-Loeve Transform (KLT) when there is high correlation among the input samples, as in the case of ECG signal

2)The Discrete Sine Transform(DST). Discrete sine transform (DST) is a Fourier-related transform similar to the discrete Fourier transform (DFT),but using a purely real matrix. It is equivalent to the imaginary parts of a DFT of roughly twice the length, operating on real data with odd symmetry (since the Fourier transform of a real and odd function is imaginary and odd), where in some variants the input and/or output data are shifted by half a sample.
3)Discret Cosine Transform-II (DCT-II). DCT-II can be viewed as special case of the discrete fourier transform (DFT) with real inputs of certain symmetry.

4)Discrete Wavelet Transform(DWT). The fundamental idea of wavelet transforms is that the transformation should allow only changes in time extension, but not shape. This is effected by choosing suitable basis functions that allow for this. Changes in the time extension are expected to be conform to the corresponding analysis frequency of the basis function. Based on the uncertainty principle of signal processing.

Apart from the Frequency Domain Techniques we can also use Direct Time-Domain techniques like:

1)Turning point (TP). The TP algorithm is used to save the important signal values by reducing its sampling rate as half. It takes three signal values to process at a time and the first signal value is saved in the compressed signal.Then , nnext two signal values are taken and from these two consecutive signal values , this algorithm retains either first or second value depending on the slope changes.

2)CORTES Algorithm. An enhanced method known as CORTES (Coordinate Reduction Time Encoding System) applies TP to some portions of the waveform and AZTEC to other portions and does not suffer from discontinuities.

3)Amplitude-Zone-Time Epoch Coding (AZTEC). The Amplitude Zone Time Epoch Coding (AZTEC) is one of the earliest ECG coding methods. It was developed by Cox as a preprocessing software for real-time monitoring of ECGs. It was observed to be useful for automatic analysis of ECGs such as QRS detection, but it is inadequate for visual presentation of the ECG signal as the reconstructed signal has a staircase appearance.

We could use all these techniques , compare the results and reach defining conclusions for Compression Ratios(CR) and Power Consumption.

Despite the compression , there is also the hardware part. The patients use the shimmer sensor(or any other) to record and send the compressed signal via Bluetooth o RF radio to a mobile phone(smartphone) and from the mobile phone to a central server. Through a web page , both patients and doctors can monitor the ECG signal thus giving the dealing with the circumstances acoordingly. Also the patient's exact position could be defined with an established GPS connection from the patient's mobile phone. This concept is illustrated graphically in figure 8.1

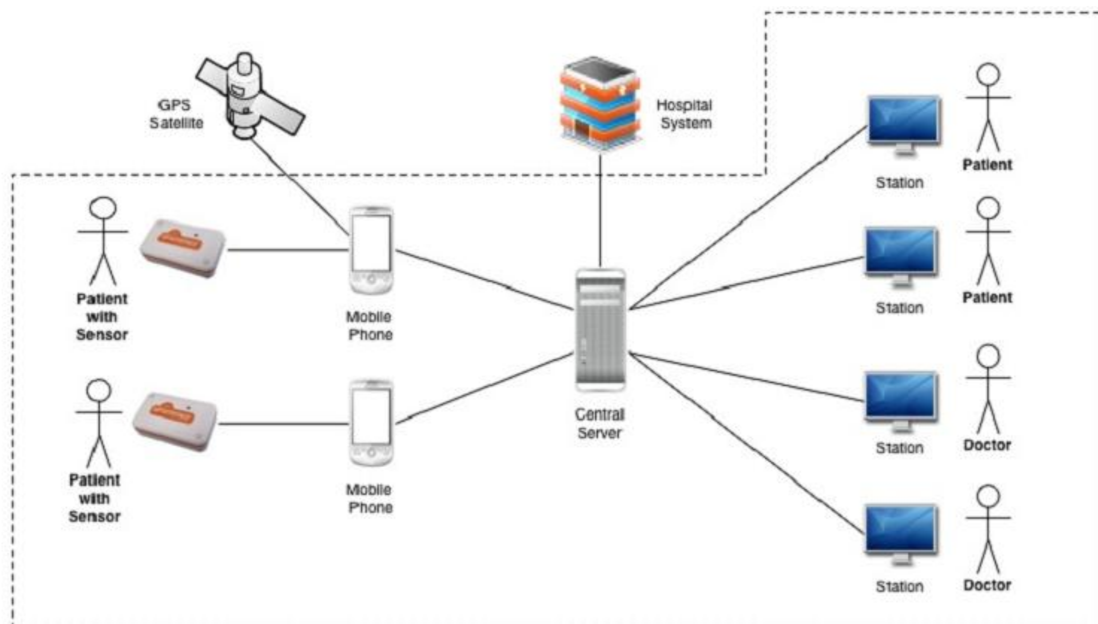

Figure 9.1 *Mobile Healthcare system context*

Chapter 10

Body-sensor Survey and Final Conclusions

10.1 Cardiac Surgery Department Survey

In order to obtain a brighter picture concerning the necessity and appliance of body sensors, we conducted a survey in a cardiac surgery department of a general hospital of the National Health System searching for possible usage of such sensors. This was a retrospective study, based on the registry data of the hospital. From the nature of the study, there was impossible any intervention in the management of the patients, and the registry was anonymous, so there was not any violation of privacy and we will not reveal any patient's consent or agreement.

During a one year period (January 1st 2012 – December 31st 2012), 626 patients were operated at the cardiac surgery department of the hospital, for a total number of 686 cardiac surgery procedures. Twenty four (24) patients (3.8%) were subjected to a second procedure and 4 patients (0.6%) to a third one.

After the procedure, the patients were transferred intubated in sedation to the Intensive Care Unit (ICU) where their vital signs (arterial pressure, central venous pressure, O_2 saturation, body temperature, urine output, ECG and mediastinal blood drain) were constantly monitored, necessary medications were granted and the protocol of postoperative recovery was followed. After stabilization of hemodynamic profile and body rewarming, sedation and mechanical ventilation were progressively withdrawn and the endotracheal tube was removed. Then, programs of physiotherapy and rehabilitation were begun. The exit from the ICU and the transfer to the general ward of the hospital was done if the patient was hemodynamically stable and without life-threatening complications. During the hospitalization in the ward, the patient was subjected to medication, vital signs recording (including arterial blood pressure, heart rate and rhythm and body temperature, usually every six or eight hours), intensive physiotherapy and rehabilitation. The presence of major complications (acute heart failure, life-threatening rhythm disturbances, pulmonary insufficiency, stroke, bleeding, and sepsis) demanded the retransferring of the patient to the ICU. The patient was discharged the hospital after full mobilization, no need for intravenous drug infusions and no serious complications.

1. From the 626 patients, 11 (1.8%) died on the operating table during the procedure and did not enter in the ICU.
2. From the 615 patients entered in the ICU, 38 (6.2%) succumbed during their state in the ICU after a mean hospitalization time of 9.1 days.
3. Mean time of the ICU state was 3.2 days (1-48 days) per patient.
4. From the 577 patients that were transferred from the ICU to the common ward of the hospital, 11 (1.9%) died after a mean hospitalization time of 4.1 days and 25 (4.3%) were retransferred in the ICU due to major postoperative complications. From these 25 patients, 6 were succumbed during their ICU state. The patient's common ward mean time state was 15 days (1-104 days).
5. Thus, from 626 patients who were undergoing a cardiac surgery procedure during the year 2012, 560 patients were discharged from the hospital to their home, a total percentage of 89.5%, while 66 patients (10.5%) died in the operating theater, the ICU, or the common ward.

The results are listed in the table below

Patient Category	Number	Category Percentage	Total Percentage
Total Number of Patients	626	100%	100%
Operating Table Losses	11	1,80%	1,80%
ICU Entrances	615	100%	98,24%
ICU Losses	38	6,17%	6,20%
Common Ward Entrances	577	100%	92,18%
Common Ward Losses	11	1,90%	1,76%
Readmission to ICU from Common Ward	23	3,99%	3,67%
Losses from Readmission to ICU from Common Ward	6	1,03%	0,96%

Table 10.1 : *Patient Data*

10.2 Cases of body-sensor appliance and final achievements

In the survey we are interested in searching for patients between those 577 that were hospitalized in the common ward after a cardiac surgery procedure during the

year 2012 and discharged later to their home and could possibly benefit from the use of body sensors. We selected 9 body sensors from www.cooking-hacks.com :

1. ***Pulse and oxygen in blood sensor (SPO_2)***

   Pulse oximetry, a noninvasive method of indicating the arterial oxygen saturation of functional hemoglobin. Oxygen saturation is defined as the measurement of the amount of oxygen dissolved in blood, based on the detection of Hemoglobin and Deoxyhemoglobin. Two different light wavelengths are used to measure the actual difference in the absorption spectra of HbO_2 and Hb. The bloodstream is affected by the concentration of HbO_2 and Hb, and their absorption coefficients are measured using two wavelengths 660 nm (red light spectra) and 940 nm (infrared light spectra). Deoxygenated and oxygenated hemoglobin absorb different wavelengths.

2. ***Airflow sensor (breathing)***

   The nasal airflow sensor is a device used to airflow rate to a patient in need of respiratory help or person. This device consists of a flexible thread which fits behind the ears, and a set of two prongs which are placed in the nostrils. Breathing is measured by these prongs.

3. ***Body temperature sensor***

   This sensor allows to measure body temperature. It is of great medical importance to measure body temperature. The reason is that a number of diseases are accompanied by characteristic changes in body temperature. Likewise, the course of certain diseases can be monitored by measuring body temperature, and the efficiency of a treatment initiated can be evaluated by the physician.

4. ***Electrocardiogram sensor (ECG)***

   The electrocardiogram (ECG or EKG) is a diagnostic tool that is routinely used to assess the electrical activity of the heart. The electrocardiogram (ECG) has grown to be one of the most commonly used medical tests in modern medicine. Its utility in the diagnosis of a myriad of cardiac pathologies ranging from myocardial ischemia and infarction to syncope and palpitations has been invaluable to clinicians for decades.

5. ***Glucometer sensor***

Glucometer is a medical device for determining the approximate concentration of glucose in the blood. A small drop of blood, obtained by pricking the skin with a lancet, is placed on a disposable test strip that the meter reads and uses to calculate the blood glucose level. The meter then displays the level in mg/dl or mmol/l.

6. ***Galvanic skin response sensor (GSR - sweating)***

Skin conductance, also known as galvanic skin response (GSR) is a method of measuring the electrical conductance of the skin, which varies with its moisture level. This is of interest because the sweat glands are controlled by the sympathetic nervous system, so moments of strong emotion, change the electrical resistance of the skin. Skin conductance is used as an indication of psychological or physiological arousal. The device measures the electrical conductance between 2 points, and is essentially a type of ohmmeter.

7. ***Blood pressure sensor (sphygmomanometer)***

Blood pressure is the pressure of the blood in the arteries as it is pumped around the body by the heart. When our heart beats, it contracts and pushes blood through the arteries to the rest of our body. This force creates pressure on the arteries. Blood pressure is recorded as two numbers—the systolic pressure (as the heart beats) over the diastolic pressure (as the heart relaxes between beats). High blood pressure (hypertension) can lead to serious problems like heart attack, stroke or kidney disease. High blood pressure usually does not have any symptoms, so we need to have our blood pressure checked regularly.

8. ***Patient position sensor (Accelerometer)***

The e-Health Body Position Sensor monitors five different patient positions (standing/sitting, supine, prone, left and right.) In many cases, it is necessary to monitor the body positions and movements made because of their relationships to particular diseases (i.e., sleep apnea and restless legs syndrome). Analyzing movements during sleep also helps in determining sleep quality and irregular sleeping patterns. The body position sensor could help also to detect fainting or falling of elderly people or persons with disabilities.

9. Electromyography Sensor (EMG)

An electromyogram (EMG) measures the electrical activity of muscles at rest and during contraction. EMG signals are used in many clinical and biomedical applications. EMG is used as a diagnostics tool for identifying neuromuscular diseases, assessing low-back pain, kinesiology, and disorders of motor control. EMG signals are also used as a control signal for prosthetic devices such as prosthetic hands, arms, and lower limbs. This sensor will measure the filtered and rectified electrical activity of a muscle, depending the amount of activity in the selected muscle.

From the 9 body sensors we have at our disposal , extremely useful and beneficial are the ones below:

1. ***Electrocardiogram sensor (ECG)***
   ECG sensor could be used in all of the 577 patients (100%)
2. ***Blood pressure sensor (sphygmomanometer)***
   Blood pressure would be useful for the 556 out of 577 patients (96%)
3. ***Pulse and oxygen in blood sensor (SPO_2)***
   SPO_2 sensor would be useful for the 521 out of 577 patients (90%)
4. ***Body temperature sensor***
   Body temperature sensor would be useful for the 327 out of 577 patients (57%)
5. ***Glucometer sensor***
   Glucometer sensor could be used for the 247 out of 577 patients (43%)

Judging from the data above gathered from the survey it becomes quite clear how important these sensors are what a huge breakthrough their use would mean for every kind of medical center in our country. The infinite number of complications could be minimized dramatically.

The most common complications in the patient's post - cardiac surgery period are:

1. ARRHYTHMIAS

Arrhythmias are the commonest postoperative complication. They are easily detected with the ECG sensor. Most of them are not life–threatening for the patient, but if they persist, they could reduce the cardiac output and may produce

intracardiac thrombus, with peripheral embolisms and catastrophic sequels. Thus, their diagnosis and monitoring of their evolution are of vital importance for the atient. On the contrary, some arrhythmias are deadly from the beginning so their prompt diagnosis and urgent treatment are life-saving for the patient.

2. HEART FAILURE

A serious number of patients are operated for heart failure. These patients postoperatively have the same signs and symptoms as before the operation, while the heart failure syndrome subsides progressively. These patients show low arterial pressure and arrhythmias that are detected from the respective sensors. These data are necessary for the defining or modification of the pharmaceutical care.

3. BLEEDING

Usually bleeding from the surgical sites appears in the ICU right after the operation and should be managed immediately, initially with conservative means and if it persists or occurred hemodynamic instability , with a new sternal opening and revision of the surgical sites. Later on the bleeding begins from another system, commonly the gastrointestinal. Tachycardia and low arterial pressure are the first signs which inform about and call for investigation for this dangerous complication.

4. PULMONARY PROBLEMS

Pulmonary disturbances are common for patients that are subjected to cardiac operations . A large percentage of the patients have problematic lungs usually from many years of smoking. Also the pulmonary function is aggravated postoperatively from the extra-corporeal circulation which is necessary for the cardiac operations , multiple transfusions and prolonged tracheal intubation. In the ICU , blood saturation in O_2 (SPO_2) is monitored constantly while blood gases are checked periodically.

5. RENAL PROBLEMS

Renal problems are common in the ICU and the Common Ward. They appear with low diuresis, increase potassium concentration in the blood and drop of pH. If not handled immediately with medical management or with mechanical systems of temporary blood dialysis , they endanger the patient's life. So, for the prevention or for the follow-up of the progress of this complication, combined monitoring with ECG, SPO_2 sensors and arterial pressure is extremely useful

6. NEUROLOGICAL PROBLEMS

Neurological disorders that appear in the ICU have multiple causes. The arterial network that feeds the brain often suffers , especially in patients with coronary artery disease or the elderly . These patients demand constant and combined monitoring with ECG , SPO_2 sensors , arterial pressure , temperature and glucose levels.

7. INFECTIONS

Infections are usually appearing after the patient's exit from the ICU. Inflammation of the mediastinal is a catastrophic complication and often dictates the reintroduction of the patient in the ICU for an extended period of time. The gravity of the patient's clinical picture calls for constant monitoring and reevaluation of their condition.

All the above complications are the main reasons for mortality in the perioperative period in the ICU and the Common Ward. In many cases , these complications can be prevented with constant monitoring. For example , arrhythmias , which as mentioned above are the most common complication , start with innocent forms and in the process they evolve with detrimental or fatal consequences. Thus , detection of arrhythmias and immediate treatment will reduce the perioperative morbidity and mortality. Patient's stay at the ICU and the common Ward is defined from the severity of the patient's clinical picture , the difficulty and the problems during the operation and the appearance of post-operative complications. Care for high – risk patients in order to achieve immediate diagnosis, and intensive follow-up for the progress of the above complications, is invaluable by monitoring with the aid of these sensors. The continuous patient's monitoring especially outside the ICU is multiple beneficial firstly for the patient's safety and secondly for the hospital for the following reasons :

- ***Saving Resources,*** *by reducing patient's length of stay in the ICU, in the Common Ward, and totally in the hospital.*
- ***Increasing hospital capacities,*** *by increasing the available beds in the ICU and in the hospital.*

- ***Improving patient's positive mood,*** *by reducing his hospital stay, especially and mainly by minimizing the length of stay in the difficult environment of the ICU.*

- ***Beginning quickly the rehabilitation programs in the family environment,*** *by preventing the appearance or reducing the severity of complications*

Finally, as far as the economic part is considered in the long run , it is certain that insurance companies will get largely involved with body-sensors and that poses another aspect of research to be done in this domain.

References

[1] Sacha Gilgen, " Mobile Healthcare on Android Devices" ,University of Zurich Department of Informatics (IFI)

[2] Hossein Mamaghanian, Student Member, IEEE, Nadia Khaled, Member, IEEE, David Atienza, Member, IEEE, and Pierre Vandergheynst, Senior Member, IEEE "Compressed Sensing for Real-Time Energy-Efficient ECG Compression on Wireless Body Sensor Nodes" IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING, VOL. 58, NO. 9, SEPTEMBER 2011*

[3] Francisco Rinc´ on, Joaquin Recas, Nadia Khaled, Member, IEEE, and David Atienza, Member, IEEE "Development and Evaluation of Multilead Wavelet Based ECG Delineation Algorithms for Embedded Wireless Sensor Nodes" , IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE, VOL. 15, NO. 6, NOVEMBER 2011

[4] PHILIP LEVIS an DAVID GAY "TinyOS Programming" , Cambridge University Press

[5] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Osterlind, Thiemo Voigt "Poster Abstract: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards" , Swedish Institute of Computer Science

[6] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt Swedish Institute of Computer Science , Robert Sauter, Pedro José Marrón University of Bonn and Fraunhofer IAIS "COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks

[7] Tatiparti Padma, M. Madhavi Latha, Abrar Ahmed "ECG compression and labview implementation" , GRIET, JNTU, Hyderabad,India, Member IETE; JNTU, Hyderabad, India, Member IEEE; GRIET, Hyderabad, India

[8] Tobias Neckel, Dirk Pfluger "Algorithms of Scientific Computing FFT on Real valued Data" , Technical University of Munchen

[9] Ms. Manjari Sharma, Dr. A. K. Wadhwani, "Efficient Algorithm for ECG Coding" , International Journal of Scientific & Engineering Research Volume 2, Issue 6, June-2011

[10] Wearable Sensor Technology|Shimmer|Wraerable Wireless Sensing Technology and Solutions , www.shimmersensing.com

[11] Toumaz Group , www.toumaz.com

[12] Interuniversity Microelectronics Centre ,IMEC , www.imec.be

[13] www.tinyOS.doc.net

[14] tinyos.stanford.edu

[15] sourceforge.net/projects/mspsim

[16] Vikas Kumar , Kulbir Singh "FPGA Implementation of DFT using Cordic algorithm" , Thapar University , Electronics and Comunication Engineering Department

[17] Sambit Kumar Dash Jasobanta Sahoo Sunita Patel , "Cordic Algorithm And it's Application in DSP" , Department of Electrical Engineering , National Institute of Technology , Rourkela