Power-efficient and Reliable MAC for Routing in Wireless Sensor Networks

MSc DCNDS - Group C - Project Report

Ioannis Daskalopoulos Hamadoun Diall

Kishore Raja

. . .

Supervisors: Stephen Hailes (UCL) and George Roussos (Birkbeck College)

. . .

University College London

Department of Computer Science

MSc Data Communications, Networks and Distributed Systems

. . .

Gower Street, London WC1E 6BT, UK

September 2005

Acknowledgments

We would like to express our sincere thanks and gratitude to Dr. Stephen Hailes (Directory of Studies, UCL), our primary supervisor for providing a chance to work on Wireless Sensor networks. His guidance and assistance throughout the project enabled us to achieve our Objectives. We would also like to express our gratitude to Dr. George Roussos (Birkbeck College), our secondary supervisor, for his invaluable guidance and technical support on TinyOS and Wireless Sensor networks in general. Being fairly new to TinyOS environment, assistance of George helped us in Porting phase and later phases as well.

Also special thanks to Tom Torfs (from IMEC) for his constant interaction and feedback to our queries on understanding the hardware sensor platform.

A note of thanks to Andre Barroso (from University College Cork, Ireland). The code he wrote for D-systems platform (consisting of Nordic nRF2401 Radio Transceiver) has been ported to IMEC Sensor platform.

We would also like to thank Dr.Niki Trigoni (from Birkbeck College), our stakeholder for stress testing of our MAC algorithm with her Multi-query optimisation routing protocol.

We would like to thank several Research fellows at UCL including Adam Greenhalgh for his feedback on making the MAC protocol energy efficient, Vladimir Dyo for his feedback on Sensor network applications and usage, Andrew Cox for some interesting exchange of technical information about TinyOS as well as helping us review the User's Guide.

A note of thanks to Application engineers from Nordic semiconductors, which enabled in better undertanding the funcationality of nRF2401 Radio Transceiver.

A special thanks to the people from TinyOS Mailing list, for providing a feedback to numerous queries on TinyOS and Wireless Sensor networks.

Thanks to Dennis Timm (Head of Technical Support Group) for providing an oscilloscope and a laboratory power supply unit. And lastly thanks to John Andrews (Head of External Network Services) for providing us the USB cables and other misc. hardware components.

Contents

List of Figures

List of Tables

Chapter 1

Project Overview

1.1 Motivation

Recent technological advances are enabling the continuous miniaturisation of components necessary to develop tiny and inexpensive sensor devices capable of performing computations and communicating wirelessly over short distances. These devices, loaded with appropriate software, can be used to build wireless sensor networks that provide up-to-date, accurate measurement data from their target environments. In themselves, wireless sensor networks present an impressive array of possible applications, right now and in the near future, as we will see shortly.

Looking even further, as *ubiquitous computing* edges closer from science fiction and research articles to reality, wireless sensor networks are slated to play a fundamental role in the fulfilment of this vision. Ubiquitous computing, "the calm technology that recedes into the background of our lives" (as originally coined by Marc Weiser [WEI1991]), will largely depend on implicit inputs regarding the user's context and environment which can be determined, to a large extent, with the help of imperceptible sensor networks, embedded throughout and even worn by the users themselves. Wireless sensor networks possess, thus, a tremendous potential to grow into something as large and important as the Internet currently is:

> Just as the Internet allows access to digital information anywhere, sensor networks will provide vast arrays of real-time, remote interaction with the physical world. [PIN2004]

Wireless sensor networks form a relatively new and very active area of scientific investigation of electronic engineering and computer science, that keeps attracting increasing amounts of attention from research communities worldwide. Nonetheless, numerous questions still remain open in different aspects of both, hardware and software. Focussing on networking software in particular, issues exist in all layers of the communication stack, from medium sharing methods, through data routing and network architecture, to abstractions for application development.

Wireless sensor networks pose, indeed, unique challenges: small size and low cost requirements impose severe resource constraints on sensor nodes, spanning processing

power, storage space, network bandwidth and, above all, energy. The significance of these constraints lies in the fact that most of the pre-existing paradigms developed for resource-rich and well-connected systems simply cannot be applied. Currently, energy awareness and efficiency is the single most important metric in the design of sensor nodes or when devising algorithms for data collection, processing and communication. This is unlikely to change in the foreseeable future:

> While there is the Moore's Law that predicts doubling the complexity of microelectronic chips every 18 months, and Gilder's Law, which theorizes a similar exponential growth in communication bandwidth, there is no equivalent forecast for battery technology. [VIE2003]

In this project we initially intended to look at multi-hop network routing issues within wireless sensor networks, taking energy consumption considerations into account. We also had access to a modular prototype sensor platform [TOR2004] developed by IMEC -- a world-class, high-tech microelectronics research centre in Belgium[1]. As their system lacked a proper operating system or general-enough abstractions for a networking stack, we were required to build or port some of this basic software.

Our motivation to undertake this project stemmed chiefly from the team members' interest in small, embedded systems. The fact that we would have access to a wireless sensor hardware platform to actually implement on (and not restrict ourselves to simulation alone) promised a great challenge and played a decisive role in the selection of this particular project.

1.2 Report Structure

The remainder of this chapter provides background information about wireless sensor networks, surveys some of the relevant work in this area intersecting with our project, discusses our objectives and approach, and closes with a summary of our main achievements. The rest of the report is layed out as described below:

- Chapter 2 provides a description of the hardware and software components involved in the project.
- Chapter 3 describes the process of porting TinyOS onto the IMEC prototype sensor modules.
- Chapter 4 delves into the details of the design and implementation work performed throughout the project.
- Chapter 5 provides an insight to the testing and simulation procedures carried out during the project.

[1]IMEC's mission: To perform R&D, ahead of industrial needs by 3 to 10 years, in microelectronics, nanotechnology, design methods and technologies for ICT systems. Website: http://www.imec.be

- Chapter 6 presents an evaluation and critical analysis of the project outcomes.
- Chapter 7 provides an overview of the project management aspects, including techniques employed.
- Chapter 8 wraps up the report with ideas for future work and concluding remarks.

1.3 Background

This section covers background material on the topic of wireless sensor networks necessary to help the reader make a sense out of the remaining chapters as well as understand the reasoning behind some of the solutions adopted throughout the project.

1.3.1 Wireless Sensor Networks (WSN)

Traditionally, sensor applications have consisted of a relatively small number of passive sensing devices directly linked to a central processing facility. Their scale and range of deployability scenarios are strongly limited by the wires necessary to connect individual sensors, or small clusters of thereof, to the processor.

As mentioned earlier, developments in short-range radio communication and microelectronics technology brought ample space for innovation in sensor networking applications. Thus, wireless sensor networks open the possibility of ubiquitous, untethered monitoring of (and, using actuators, even interaction with) our surrounding environment at resolutions previously unattainable and in ways yet unimaginable.

A **wireless sensor network** can be defined as an infrastructure-less, self-organizing network composed of numerous, tiny, low-cost sensor nodes. The **sensor nodes** are self-contained devices consisting of four main components:

1. *sensing devices*, e.g. temperature, humidity, light, sound, vibration, atmospheric pressure, motion, acceleration, chemical/biological sensors, etc;
2. a *wireless communication unit*, most commonly using low-power radio frequency, but can also use infrared or other optical media;
3. an *on-board microcontroller* with a limited amount of computing and storage resources[2];
4. a *power source*, generally a small battery, possibly rechargeable from the energy available in the environment (e.g. solar, thermal, kinetic, etc).

Depending on the target application, sensor nodes may have additional features such as *actuators* to interact with the target environment, *positioning system* to enable the node to determine its location (e.g. GPS), and/or *mobilizers* to allow the node to move. Sensor networks with such characteristics enter into the realm of robotics, and are usually

referred to in related literature as *actuated wireless sensor networks* or *wireless sensor and actor networks* [AKY2004].

1.3.2 Applications

Although commercially wireless sensor networks still do not bear a significant relevance, a number of potential applications scenarios are envisaged by researchers in different sectors. The table below presents a few examples, some of which have already been prototyped in the context of experimental projects [MAI2002], [DOO2005], or even custom-built systems deployed in production settings [KRI2004].

Table 1.1: Applications of wireless sensor networks.

Area	**Examples**
Biologic research	Non-invasive habitat monitoring and study of wildlife species/populations.
Environmental monitoring	Pollution (air, water, soil); early alarms for forest fires, volcanoes, earthquakes, and other natural hazards.
Disaster response	Emergency support for disaster recovery scenarios to identify risks and hazards, locate people/survivors.
Healthcare	Staff and patient tracking; on-the-body vital signs monitoring; drug administration control.
Agriculture	Livestock and crops monitoring; microclimate management for dairy production or vines; soil fertility analysis.
Industrial sector	Manufacture process automation, monitoring & control; equipment failure prediction; production quality assurance.
Retail sector	Stock management, product tracking, quality monitoring.
Architecture	Office smart spaces; home automation; intrusion detection.
Transportation	Monitoring of internal systems in cars, ships and aircrafts.
Local authorities	Road traffic reporting, analysis and coordination.
Military	Surveillance; enemy identification; target acquisition and tracking; logistic operations support.

1.3.3 Resource Constraints

Sensor nodes possess very limited resources in terms of network connectivity, computing power, data storage and, above all, energy. In order to unobtrusively pervade their target environments in large numbers, sensor devices need to be inexpensive and as

[2]Currently, devices exist on the market that combine a microcontroller unit and radio communication in a single chip, such as the Nordic nRF24E1 or ChipCon CC2430.

small as possible (for a given application) -- current developer platforms are not larger than a matchbox and research efforts are underway to achieve "autonomous sensing and communication in a cubic millimeter" [PIS1999]. Physical size as well as low unitary cost requirements are key factors for the stringent resource constraints imposed on sensor nodes.

Programming models, testing and debugging techniques must undergo profound changes as these devices feature no keypads or screens (at best simple LEDs), so most input/output will be achieved through wireless communication. Low power radio systems are very prone to interference of all kinds, exhibiting transient lack of connectivity (due to bit errors which result in packet loss) and asymmetric links [WOO2003]. Moreover network bandwidth is relatively narrow, with raw bit rates in the range of a few Kbps up to around 1 Mbps, yielding even lower net data rates.

Comparatively with many other types of embedded systems, sensor nodes are also strapped for processing power and memory. A diverse set of micro-processors can be used depending on the application, with a diverse range of performances: 4- to 32-bit CPUs, one to several dozens of MIPS [VIE2003]. Likewise, storage can vary from less than one kilobyte to a few hundreds or even enter the megabyte range, possibly divided in two sections: (a) flash/ROM to store and execute program code, and normal (b) RAM used as work memory to store and manipulate data. Again, programming paradigms and algorithms must be thoroughly redesigned in order to fit the computation and space restrictions of wireless sensor networks.

Usually, sensor nodes have a *finite power supply* and, once deployed, it is often impractical to replace or recharge the batteries due to their small size, hazardous location or sheer network scale. Hence it is absolutely critical to optimise power usage in all aspects of individual sensor node operation as well as in interactions with others, in order to maximise the system's lifetime and, thereby, its cost-effectiveness.

Energy scavenging or *harvesting* technology is getting small enough [KAN2003] to be used in sensor nodes to recharge the battery from energy available in the surrounding environment -- e.g. sunlight, thermal gradients, vibrations, etc. Nevertheless, even when employing energy scavengers it is crucial to remain energy-efficient, as the amount of electrical power actually drawn from the environment may not be abundant or continuous (for instance when using solar cells, night operation relies on energy accumulated during daytime).

1.3.4 Architecture

The diagram presented below depicts a typical wireless sensor network architecture. Sensors nodes, running a small software image that controls their behaviour, are deployed in the target area, referred to as the *sensor field*. Once active, they form an *ad hoc* wireless network according to algorithms and protocol specifications coded into their operating software.

A *base station*, also called *sink*, is a node interfacing the sensor field and connected to external networks (via the Internet or other uplink), in effect acting as a gateway between the wireless sensor network and the outside world -- where users can apply

more computing resources to control the sensors and deal with their data. As the figure also illustrates, sensor nodes can resort to multi-hop routing to communicate with the base station(s) or other distant parts of the network (not necessarily to save energy [MIN2003]).

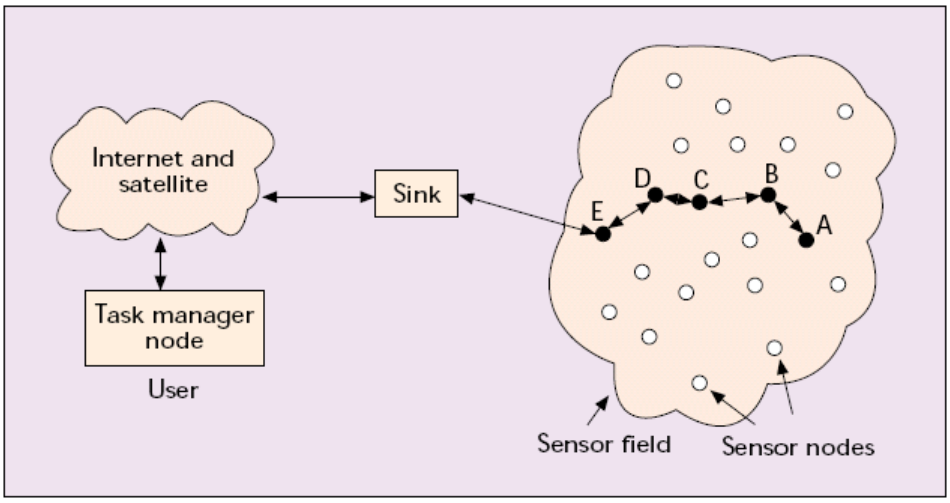

Figure 1.1: A typical wireless sensor network architecture (taken from [AKY2002]).

This architectural setup enables the development of powerful and sophisticated applications to bridge the gap between the users' need for fairly high-level interfaces and the complexities of low-level programming necessary to ensure efficient operation under the severe resource constraints in the sensor nodes themselves. Hence, these applications empower users to interact with the sensor network in meaningful ways to extract the information required, while retaining an overall energy-aware performance.

For instance, in many cases wireless sensor network applications can be viewed as a database system [GOV2002], or as a spreadsheet [HOR2005], that "stores" data about the target environment (or whatever is relevant to the context in which their are applied), allowing users to submit queries and obtain results. Thus, using database terminology, the sensor field can be thought of as a dynamic or "live" *table*, where each sensor node is a *row* (or *record*) and each type of sensor that the node bears is a *columm* (or *attribute*). A number of query processing systems have been proposed to use SQL-like abstractions to describe the required data [GEH2004], and work progresses towards exploiting more resourceful systems to offload unnecessary burden from the sensor nodes themselves (e.g. multi-query optimization to reduce communication [TRI2005]).

1.3.5 Challenges

There are many considerations and factors to take into account in all phases of design, implementation and deployment of wireless sensor networks.

- Sensor field:
  - dimensions and shape of the target area;
  - physical terrain characteristics (mountains, rocks, grass, water, etc.);
  - hazards to the sensor nodes' integrity (storms, fire, battlefield, etc.);
  - radio environment (effects on signal propagation, interference sources).
- Sensor nodes:
  - features (type of micro-controller, radio, sensors, actuators);
  - location info (none, manually programmed, automatic systems);
  - mobility (stationary or mobile);
  - power source (battery type and capacity, energy scavenger unit);
  - heterogeneity (different types of nodes w.r.t. the above aspects).
  - total/maximum number of nodes;
  - deployment layout (hierarchical, grid-alignment, random, etc);
  - node proximity (communication range, signal strength, link quality);
  - network density (average number of neighbours, active/sleeping);
  - network dinamicity (battery depletion, node malfunction, additional node deployments, varying radio conditions).
- Base stations:
  - total/maximum number of sinks;
  - communication between sinks;
  - type of uplinks to external networks;
  - location of sink(s) relatively to sensor nodes;
  - sink mobility (stationary or mobile).

Sensor networks are, indeed, a new family of wireless *ad hoc* networks. However, they differ in a number of key areas from the traditional ones, and to a significant degree the work focussing on MANETs is generally is not directly applicable. Some of the most important and unique challenges presented by wireless sensor networks are highlighted below [AKY2002], [WAN2002].

Size, scale and resource constraints: The sensor devices are markedly smaller and deployed in numbers potentially several orders of magnitude higher, hence most protocols and algorithms need to be thouroughly revised and/or re-designed from scratch. Moreover, the harsher resource limitations of sensor networks imply that solutions tend to be very application-specific (throughput and latency requirements, sampling rates, etc), therefore no single protocol can address all scenarios optimally.

Network lifetime and duty cycles: The success of wireless sensor networks is intimately related to their cost-effectiveness, a fact that entails the necessity of unattended operation during the application's lifetime -- at least days or weeks (for a disaster response scenarios or short-term wildlife studies) up to several months, or even years (factory automation, precision agriculture, on-going environmental monitoring). This usually translates into minimal active duty cycles or, in other words, that sensor nodes spend most of their time in very low-power, inactive or sleep modes. A duty cycle as low as 1% (active), or even less, may be required to deploy long-lived, battery-powered applications [POL2004].

Radio energy consumption: In low-power radio systems (used in wireless sensor nodes) communication, as opposed to computation, vastly dominate the energy consumption budget, consequently minimizing the operation of the transceiver is key. In principle, algorithms and protocols that trade off communication for computation will achieve greater energy savings.

Network density: Sensor nodes are usually densely deployed, thus there is a high degree of data redundancy in the network. This characteristic lends itself to application of in-network processing and data aggregation techniques in order to reduce network traffic and therefore conserve energy resources.

Fault tolerance: A single sensor node is not "trustworthy" as they are prone to failure (lack of power, radio interference, physical damage, security breach), hence fault tolerance is core requirement for any algorithm or protocol to ensure continuous system operation.

Network dynamicity: Furthermore, sensor networks may need to be deployed randomly (droped from an airplane, delivered in an artillery shell or missiles) and operated in any kind of environment, including very extreme ones (near active volcanoes, battlefields), so dynamic self-configuration and organisation is required. Sensor node and/or base station mobility further complicates matters.

Communications paradigm: Sensor network applications have a different, more specialized purpose than traditional *ad hoc* networks -- basically collect and forward data, react to changes in the environment, respond to commands... Data flows and traffic patterns are usually asymmetric, contrasting with point-to-point communication in MANETs, and nodes do not have global addresses and sensor queries are often data-centric (based on attribute or location), rather than node-centric.

1.4 Objectives and Resources

This project started out with a fairly wide scope and ambitious goals. Based on the information we had available at the time for the initial project definition in May 2005, we set preliminary high-level objectives and drafted an approach.

However, as the project unfolded we ran into a myriad of issues (detailed in later chapters), including a quite steep learning curve for the operating system chosen. These issues hampered our progress somewhat and we had to redefine the goals/scope a few times, which ultimately shifted the project's weight from an initially more research-oriented emphasis to more of an engineering perspective.

1.4.1 Initial Objectives and Approach

Below are listed our initial objectives for this project:

- *Primary outcome*: development and testing of a power-aware, multi-hop routing protocol for wireless sensor networks.
- *Supplementary outcomes*:
  - Analysis of security issues in the multi-hop routing protocol (if time permits).
  - Establishment of a programmer-friendly operating environment for the IMEC sensor hardware platform, by porting an existing OS for sensors.
  - Production of the necessary documentation throughout the process to facilitate further work by following researchers.

Our basic strategy to carry out the project was outlined as follows:

1. Survey existing OS solutions for wireless sensor networks (TinyOS and Contiki were considered) and try to port one to the IMEC's prototype platform; if porting an OS turned out unfeasible, specify a set of features with a clear interface and build simple network stack using of the existing code from IMEC.
2. Carefully analyse current research in this area and evaluate existing *ad hoc* multi-hop routing algorithms.
3. Select a limited application scenario for the routing functionality (at this point we were still unsure if TCP/IP support would be considered).
4. Either adopt and apply an existing multi-hop routing solution or design a novel protocol.
5. Validate the solution/algorithm through simulations, implement in software for the actual hardware prototype and conduct empirical experimentations.

1.4.2 Adjusted Goals and Scope

One of the major hurdles faced during the TinyOS porting efforts turned out to be some limitations of the IMEC platform w.r.t. the radio modes supported (more on that later). This hindered the usage of MAC protocols already implemented for TinyOS which, in turn, dwindled the time we had available to concentrate on multi-hop routing algorithms. In light of the above, the initial objectives were scaled-down as follows:

- Provide a usable TinyOS port to the IMEC sensor module platform, featuring support for radio communication and sensing devices.
- Develop a MAC protocol with reliability and energy efficiency features built-in, suited to work on the IMEC platform accounting for some specific design limitations of its Nordic nRF2401 radio.
- Experiment with multi-hop routing protocols included in TinyOS on top of the developed MAC, both on the actual hardware and in simulation.
- Produce a proper release (including appropriate user-level and technical documentation) of the TinyOS port and all software developed to IMEC, as a project stakeholder.

Some of the key assumptions regarding the scope for our adjusted project objectives are given below:

- Software to run on IMEC sensor , as well as TinyOS' simulator (TOSSIM).
- Network is expected to be dynamic (nodes can be added, removed, etc).
- Network is composed of homogenous, stationary sensor nodes.
- Tree-based, multi-hop routing to a single, stationary sink.

1.4.3 IMEC Project Kit

The project was initially supplied by IMEC with the following resources:

- **Hardware**:
  - 3 sensor modules (featuring an MSP430 micro-controller, nRF2401 radio, plus sensors for temperature, relative humidity and light);
  - 3 rechargeable batteries (2.4V, 15mAh) with matching connectors;
  - 1 USB programming board to interface the modules (BSL and UART);
  - 1 USB stick radio interface (with nRF2401 radio);
  - 1 JTAG programming connector for USB stick (parallel port).

- **Software**:
  - Sample set of applications demonstrating a centralized network for sensor data collection and logging on a PC using the USB stick as base station (star-topology, TDMA-based network with fixed 8-bit addresses, no routing algorithms);
  - Radio interface code used in the above applications released under the BSD license.
  - Windows-based utility (bslprog.exe) to load software onto the sensor modules over the USB programming board (using MSP430 BSL).
- **Documentation**:
  - Basic 'Getting Started' guide for the above hardware and software (`GETTINGSTARTED.txt` file);
  - Hardware schematic diagrams (classified as **confidential**);
  - Diagram depicting the correct assembly of the sensor prototype's modules.

Throughout the project we sought to obtain additional hardware on an as-needed basis (e.g. power supplies, oscilloscopes, components for electronic circuitry, etc). All this is discussed and explained in greater detail in the next chapter.

1.5 Existing Work

As mentioned earlier, wireless sensor networks currently are still a new and very hot research topic, with a wide range of issues to being addressed in all of its facets: systems support for embedded devices, communication methods and protocols, data aggregation and processing, application architectures, single node and overall network energy efficiency, among others.

We aimed to port an operating system onto the IMEC sensor module prototypes in order to provide an environment featuring a set of useful hardware abstractions that lend more flexibility to the development process. This necessarily forced us to look into systems like TinyOS [HIL2000] and Contiki [DUN2004]. SOS [HAN2005], another operating system for wireless sensor network derived from TinyOS, was first released after we had analysed possible solutions for our project.

Testifying the vast amount of current research activity going on in this field, a significant number of energy efficient algorithms/protocols have been devised specifically for wireless sensor networks, both for medium access control (or MAC) [DEM2005] and data routing [AKK2003], and even integrated link-layer/routing approaches [CUI2005].

Per our initial goals we started looking at several alternative routing algorithms, that can be essentially classified according to the taxonomy below [AKK2003]:

- Data-centric protocols.
- Hierarchical protocols.
- Location-based protocols.
- QoS-aware protocols.

However, as the goals were adjusted to focus more on link-layer issues, we analysed different MAC protocols, which can be subdivided essentially into two classes (as happens in traditional networks):

- Contention-based protocols (e.g. ALOHA, CSMA).
- Schedule-based protocols (e.g. TDMA).

Although TDMA-like protocols tend to be more power-efficient due to the lack of collisions and synchronized transmitters/receivers, they can be trickier to implement in a dynamic, multi-hop scenario. Torfs et al. [TOR2004], from IMEC, propose such an approach for multi-hop MAC, based on TDMA.

As we intended to give higher priority to adaptability to varying radio conditions in a dynamic network setting, we chose a contention-based approach. Initially, we looked at protocols already implemented in TinyOS (either in the core distribution or third-party contributed software), namely B-MAC, S-MAC and T-MAC.

B-MAC recurringly turns off the radio tens for several of milliseconds (tens or hundreds); to send a packet, a node transmits a preamble long enough to ensure that the destination will be "online" to receive it [PHC2004]. This imposes a considerable energy penalty on the sender for each packet, which is aggravated with network density impacts. S-MAC, on the other hand, attempts conserve energy by synchronizing sleep/wake schedules among groups of nodes in a given area; bordering nodes act as gateways between two clusters with different schedules [YEW2004]. In a way, S-MAC can be thought of as a contention-based protocol with time synchronization, particularly suitable to regular patterns of somewhat high-volume of traffic. T-MAC, derived from S-MAC, tries to improve performance under a variable, bursty traffic load [DAM2003].

All of these protocols rely on some sort of carrier sense mechanism (or otherwise low-level access to the underlying radio stream) that is not possible in nRF2401's ShockBurst mode (more on this later), which we are forced to use on our prototype platform due to the lack of a high-enough clock frequency (which, in turn, would increase the energy consumption). In the end, due to time constraints, this left us with no alternative other than implementing a simple, ALOHA-based algorithm.

During our survey on operating systems and related hardware we found a sensor platform -- D-Systems [BAR2004], developed at Cork University, Ireland -- that features the same radio as the one on IMEC's prototype and with TinyOS support. We used their software release as a base for developing support for our platform, as detailed further in the report.

1.6 Achievements and Contributions

Due to unforseen issues and time constraints we were unable to explore our ambitious, initial objectives for this project to their fullest extent, but we successfully met most of the refined goals to a satisfactory degree:

- First TinyOS 1.x port to a platform combining an MSP430 micro-controller and an nRF2401 radio chip[3].
- TinyOS support of all relevant subsystems of IMEC's prototype -- most of MSP430 features and peripherals, nRF2401 radio chip in ShockBurst mode, and all sensing devices.
- Development of a simple, yet reliable and energy efficient, MAC protocol with ALOHA-style operation, tailored for the specific characteristics of the radio unit present on IMEC's prototype platform.
- Empirical evaluation suggests a high packet delivery ratio (above 95%) with relatively low radio duty cycles (25% active), particularly for applications generating regular traffic patterns.
- The above results translate into significant energy savings, if we consider that the radio subsystem accounts for an overwhelming slice of the sensor node's power budget (˜23 mA with radio turned on vs. almost nil when off).
- MAC algorithm implementation extended to support simulation under TOSSIM.
- Multi-hop routing protocols bundled with TinyOS work on top of the developed radio stack, which was tested empirically (to the extent possible with only three nodes!) and in simulation.
- Software prepared for release, including user-level and technical documentation.

1.7 Chapter Summary

This chapter presented an overview of the project as whole. We described our motivations and goals, covered preliminary background information about wireless sensor networks and related work, and highlighted our main achievements.

The next chapter provides a description of the hardware and software components involved in the project.

[3]It should be noted, however, that the MSP430 micro-controller enjoyed a quite fully-featured support under TinyOS, which helped our efforts a great deal, and we also used D-systems' nRF2401 radio stack as a reference for our own MAC implementation.

Chapter 2

Hardware and Software Platforms

This chapter aims to introduce the hardware components that comprise the IMEC Sensor Cubes and provide a short introduction to their capabilities and limitations. The overall architecture of the device is considered and selected key sub-components are presented, including the Microcontroller Unit, the Radio Chip, the Temperature-Humidity Sensors and the Photosensor. In addition to the above, some auxiliary hardware components used throughout the project are presented. The later sections of the chapter describe the initial analysis performed to choose an Operating system. Also the TinyOS operating system is shortly introduced together with TOSSIM, its Simulator.

2.1 Architecture and Hardware Components

2.1.1 Overall Cube Architecture

A pioneering for sensor modules, layered architecture is the distinctive characteristic of the IMEC Sensor Cube. Its modular design incorporates self-contained building blocks of hardware components, that can be easily plugged together to form sensors with different capabilities every time. This makes the sensors very versatile and suitable for a large range of applications, since their functionality can be tailored to the application needs.

The main hardware components of an IMEC Cube include the Microcontroller, the Radio Communications module, the Power Module and the Sensor Module. The prototypes used in this project featured functional blocks that were implemented as 14mm x 14mm printed circuit boards and were plugged together to make up a four-layer stack. The top layer of the stack is the Radio Layer, mostly occupied by a Nordic nRF2401 2.4GHz wireless transceiver chip, together with an integrated antenna. One layer below resides the Texas Instruments MSP430 microcontroller which is the "heart" of the sensor module as it is responsible for data processing and control. In the same layer, a 32.768kHz crystal provides a local time reference and a clock source to the system. The microcontroller and radio layers together form the core of the sensor and they are designed to work closely together, a fact that justifies their physical proximity - direct connection.

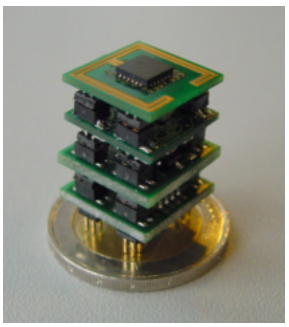

Figure 2.1: The IMEC Sensor Cube. Layers from top: Radio Module, Microcontroller, Power Module, Sensor Module. Courtesy of IMEC.

Under the Microcontroller layer, two additional layers are providing the Power Management (layer 3) and Sensing (layer 4) features. The Power Management layer is designed in a way that it can accept power from an energy scavenger(e.g. a solar cell) so as to sustain the battery life. Normal power supply like batteries is also connected to this layer. The available sensing equipment is comprised of a Sensirion SHT15 Temprature/Humidity sensor and a Light-Dependent Resistor (for measuring illumination). These sensors produce accurate measurements while consuming very little power when in use or standby. Finally the batteries provided with the prototype sensors are Varta 2-cell NiMH batteries with a voltage rating of 2.4V.

In the following subsections the different layers of the IMEC Sensor Cube are considered and some key features of the hardware are presented. By understanding the underlying platform and its capabilities, someone could understand better the limitations, ideas and solutions presented in later chapters.

2.1.2 Texas Instruments MSP430 Microcontroller

The Microcontroller Unit (MCU) that the IMEC Electronic Engineers selected for their Sensor Cube is the ultra low power MSP430F149 from Texas Instruments. MSP430 is a widely used and well understood MCU platform, targeted to a wide range of applications from metering and portable instrumentation to consumer electronics. Its low power consumption characteristics emerging from its intelligent architecture and five low power modes, together with the very small wake-up time from power saving (6us), fully justify its employment instead of other MCUs in the IMEC Sensor Cubes.

In addition to the above, very attractive, characteristics for sensor nodes, the MCU offers a powerful, modern, 16-bit RISC CPU with 16-bit registers. Depending on the available power, the CPU can function at speeds up to 4MHz. Regarding the memory of the system, a total of 2KB of RAM is available, backed by (a large for the microcosm of sensors) 60KB of Flash/ROM. Both memories can be used for code and data. In the case of Flash/ROM, word or byte tables can be stored and used directly with no need

for them to be copied to RAM. Some very useful features of the MSP430 that are worth noting include the constant generators for code efficiency, the two 16-bit timers, a 12-bit Analog-to-Digital converter (for converting analog input from e.g. sensors to equivalent digital representations), two Universal Serial Synchronous/Asynchronous Communication Interfaces (USART) and finally a total of 48 I/O pins for connecting peripheral components.

Digital I/O of the MSP430 MCU

The MSP430 MCU used in the IMEC platform has a total of 48 I/O pins, grouped in a total of 6 I/O ports (P1 - P6) of 8 pins each. Throughout this document individual pins will be identified by their port number, followed by their "in-port" number (for example, the first pin of the first port P1 would be P1.0 and the last P1.7). Users can configure individual pins to have input or output direction and can also read and write to them at will. Two out of the six ports of the MCU (ports P1 and P2) offer interrupt capability for each and every pin involved (i.e. from pin P1.0 to P2.7) and can be configured to interrupt the CPU on a rising or falling edge of an incoming signal.

The configuration of the digital I/O is done by the user and typically a C header file contains the desired pin settings. By setting values of dedicated "I/O setup registers", the user can select the initial value of the pins (0 or 1) and the direction of the pins (INPUT or OUTPUT) depending on the way that they will be used. In addition to the above, the functionality of individual pins can be selected to be simple I/O or to provide peripheral functionality of the MCU like SPI explained below. The correct configuration of the I/O pins is vital for the successful communication of data to and from the MCU and for its operation in general.

Universal Synchronous/Asynchronous Receive/Transmit

The Universal Synchronous/Asynchronous Receive/Transmit (USART), is an interface to a single hardware module which supports two serial modes of communicating data to and from peripheral devices connected to the MCU. The modes available that are of interest to us are the Universal Asynchronous Receive/Transmit (UART) and the Synchronous Peripheral Interface (SPI). The MCU used in the IMEC Sensor Cube provides two USART modules with the same functionality: USART0 and USART1.

The synchronous (SPI) mode allows the MCU to connect to a peripheral device (e.g. Radio Chip) as either a master or a slave, depending on which device controls the communication. When using this mode, three pins of the MCU are of interest: Slave In Master Out (SIMO), Slave Out Master In (SOMI) and USART Clock (UCLK). Consider a set up where the MCU is the master and a Radio Chip is the slave. The SIMO pin is used when the master sends data to the slave (MCU -> Radio) and the SOMI pin is used when the slave has data to send to the master (Radio -> MCU). The UCLK is used as a clock source that synchronizes and controls the data rate of the transfers.

The communications through SPI are Byte oriented and utilise independent transmit and receive shift registers and buffers. In the set up described above, when the MCU

wishes to write a byte to the Radio, it does so by writing it bit by bit on its transmit shift register. While this occurs at the rising edges of the UCLK, the receive shift register of the MCU is filled bit by bit with data from the transmit register of the slave-Radio concurrently (but at the falling edges of the clock). It is therefore a requirement to write something to a slave in order to get the data it is willing to transmit to the master.

An alternative way of communicating data via a connection is to do so without using a hardware controller like SPI. Such methods are known as bit banging techniques and mainly involve simulation of a protocol in software. The MSP430 allows the use of such techniques and, therefore, its general purpose pins can be used for the emulation of serial communication interfaces.

Clock System and Timers

The clock system of the MSP430 uses a low frequency clock input driven from a common 32kHz watch crystal. This power efficient clock source is used to provide three basic timer modules, namely, the Watchdog Timer, Timer_A and Timer_B. The watchdog timer module has the sole purpose of initiating a system restart after a serious software error. In the case that the watchdog function is not needed, the timer can be used to "fire" (generate interrupts) at predetermined intervals. Both Timer_A and Timer_B are similar 16-bit timers/counters with the capabilities of a) capturing the time of an event occurrence (signal or interrupt) and of b) generating an interrupt at predetermined intervals. Both timers are also capable of generating interrupts on counter overflow. The timers can operate in various modes as they can be configured to count upwards (0-65535) downwards (65535 - 0) Up-Down (0-65535-0) and Continuously. Their clock source that ultimately controls their tick granularity and therefore their precision, can be selected and configured at will.

An interesting feature available by both Timer_A and Timer_B is the ability to use them in Capture Mode. The Capture Mode is used when there is a need to record the time that an event has occurred. In order to do so, the capture inputs of the timer are associated with the particular event of interest (an MCU pin or an internal signal). The user can register interest on rising (0 to 1), falling (1 to 0) or both transitions of the bespoken pin or signal value. Upon capture, the time will be stored in a buffer and an interrupt will be raised by the timer. The capture feature of timers is very useful in situations where an interrupt must be raised when a pin value changes, but the pin happens to be in a port that without interrupt capability. With the above state of affairs, a neat way of "extracting" an interrupt is by associating a capture timer with the pin of interest. Now, upon transition, the timer which is monitoring the pin, will capture the time of the transition occurrence, but at the same time will raise the timer interrupt providing thus the required CPU interruption.

ADC12 Analog-to-Digital Converter

The ADC12 is an accurate 12-bit Analog-to-Digital converter and is onboard the MSP430 MCU. It is a high performance module capable of converting analog signals (e.g. from

sensing hardware) to corresponding 12-bit digital representations. By employing a dedicated buffer, the ADC12 is capable of converting and storing samples without any CPU intervention and at the very high rate of 200 thousand samples/sec. For ease of use, conversions can be triggered by software or any of Timer_A or Timer_B. The signal to be digitized is fed to the ADC module from one of its eight input channels that are easily configurable. It is also worth noting that "when the ADC12 is not actively converting, the core is automatically disabled and automatically re-enabled when needed" [MSP430U], a fact that underlines again that the MSP430 was designed with low power consumption as a primary requirement.

2.1.3 Nordic nRF2401 Radio Transceiver

The Radio Transceiver chip used in the IMEC platform is the nRF2401 from Nordic Semiconductors. The chip provides all the hardware necessary for transmitting and receiving at the 2.4 - 2.5GHz ISM band in a tiny package and is characterized by its low power consumption, built-in power-saving modes and relatively high bitrates for transmission/reception (250kbps and 1Mbps). Control and configuration of the Radio is achieved by loading to it a 15-byte configuration word (or parts of it depending on the changes required). The most notable feature of the Nordic Radio chip is its ability to transmit and receive data in two different modes: The ShockBurst Mode and the Direct Mode. Important features common to both modes as well as the two modes themselves are presented below, in an attempt to describe their features and explore advantages and possible drawbacks related to their use.

The nRF2401 can be configured to a great extent to fit a wide range of possible applications and requirements. The very useful DuoCeiver feature allows simultaneous reception of two different signals provided that the latter are 8MHz apart. This means that even though a single antenna is used, the Radio can receive simultaneously from two potential transmitters. This results to the notion of the two Channels of the Nordic Radio. Each one of the channels while in ShockBurst mode can have its own address, that can range from 8 to 40 bits. In Direct Mode, the addressing scheme is up to the programmer since it is performed by software and not the chip itself. Regarding the data length of the packets of the two channels, this again can be different, channel-specific and set depending on need. From the transmitter's point of view, data can be send using any of the frequencies starting from 2.4GHz, and going up to 2.524GHz. An important note within this context would be the fact that configuration information and therefore functionality of the chip, cannot be changed while the latter is transmitting or receiving data. This restriction does not allow for example dynamic change of the data length of any channel in a fast and an efficient way.

The following are the modes of operation that a Nordic nRF2401 may be and their related current consumption:

- Transmit Mode: 13mA (Average @ 0dBm output Power)
- Receive Mode: 23mA (Average for both channels ON @ 0dBm output Power)

- Configuration Mode : 12uA (Average)
- Stand-By Mode : 12uA (Average)
- Power Down Mode @ 400nA (Average)

Note: The above values are taken from [NRF2401].

The ShockBurst and Direct Modes of Operation

The main idea behind the ShockBurst mode is to achieve a reduction in the power consumed while transmitting, by employing a FIFO structure available onboard the Radio chip. When in this mode, the data to be transmitted is clocked in the FIFO structure, typically in a low data rate, to be accumulated there. The transmission of data occurs then in bursts and in a significantly higher data rate, resulting in dramatically smaller transmission times and therefore lower power consumption. As stated in the datasheet of the chip: "by allowing the digital part of the application to run at low speed while maximizing the data rate on the RF link, the nRF ShockBurst mode reduces the average current consumption in applications considerably" [NRF2401].

To justify even further this approach, consider an MCU that is only capable of providing data to the Radio chip at low data rates (e.g. 10kbps), and wishes to transmit a total of 100 bits. If the Radio was to transmit immediately and as the data arrived from the MCU, the transmission time would be 10ms, forcing the Radio to stay in transmit mode for that time. More often than not, though, single-chip radio transceivers can have transmission speeds of 1Mbps (which is the case for the nRF2401 used in the IMEC platform). Considering all the above it is possible to see the dramatic reduction in transmission time that ShockBurst mode can achieve as the data would be now transmitted in one burst and at 1Mbps, resulting in a transmission time of 1ms. The reduction in power consumption comes from the fact that the Radio stays in the costly transmit mode for less time as it doesn't idle-wait for data to arrive at slow rates restricted by the MCU. In addition to the reduced energy consumption, shorter transmission times also help to reduce the risk of on air collisions.

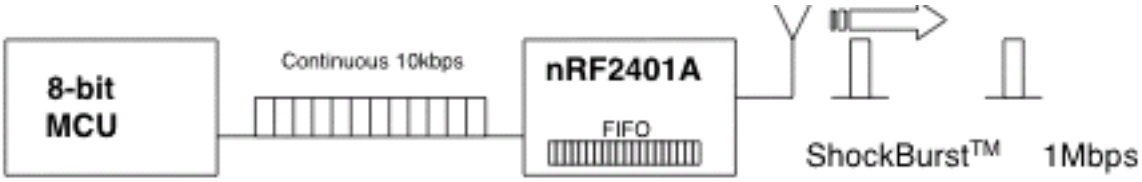

Figure 2.2: The ShockBurst Transmission Principle (taken from [NRF2401])

In addition to the reduced power consumption resulting from the ShockBurst principle of operation, there are some more features available that are of interest:

- Automatic address check upon reception of data: The MCU will be notified only if the data was addressed to that specific node.

- Automatic 8 or 16-bit CRC calculation and addition to the data to be transmitted by dedicated hardware: The MCU does not need to calculate CRC for packets.
- Automatic CRC check upon data reception by dedicated hardware: The MCU will not be notified of reception of a corrupted packet.
- Removal of Address, CRC fields upon reception: Only the payload will be passed to the MCU, reducing unnecessary communication.

The ShockBurst approach to the radio communications is very attractive, as it reduces overheads at the MCU when communicating data and also saves energy. There are though some limitations which result from its design that might make it inappropriate for use in certain situations:

- There is no obvious way of implementing Broadcasts or a Promiscuous mode since every receiver has its own address and the chip discards everything but packets addressed to it.
- There is a limit of 256 bits maximum ShockBurst frame length (including Address and CRC bits) which may require fragmentation and reassembly to be performed in upper layers of the protocol stack.
- There is absolutely no way of determining whether a packet was received but corrupted (CRC check failed) or was not received at all.
- There is no way of performing Carrier Sense since the chip will not provide any information about ongoing transmissions unless what is on air is addressed to it.

A solution that appears to be straightforward in situations that the ShockBurst mode is inadequate, is to use the Direct mode. When the Nordic Radio is operated in the Direct mode, it behaves like a common transceiver, There is a fundamental requirement in order to operate in this mode and as described in the device datasheet: "Data must be at 1Mbps +/-200ppm, or 250kbps +/-200ppm at low data rate setting, for the receiver to detect the signals" [NRF2401]. This requirement dictates the employment of a clock source fast and accurate enough to enable the clocking of the data from the MCU into the Radio chip at those specified rates. In the case that transmission at 1Mbps is needed, then a 16MHz crystal is required.It should be made clear that unless the data is clocked into the chip at the correct rate, they will never reach their destination, since no receiver will pick-up the signal.

Direct mode does not provide any of the power-saving that ShockBurst mode does, nor does it perform the utility functions described in the bullet points above. Therefore, it may be the case that the Address and CRC checking overhead has to go to the MCU; however, Direct mode might be preferable simply because the implementation of Broadcasts, Promiscuous Mode and Carrier Sense is feasible and more straightforward. It is very important to note that the IMEC Sensor Cubes, unfortunately, **do not have** the ability to operate the Radio in Direct mode due to the lack of a clock source that is fast enough.

Another limitation of the IMEC platform results from the lack of complete physical connections to the Radio Chip. Although the latter allows the communication of data regarding its operation performance via specific connection pins, those pins in the case of the IMEC Cubes are not physically wired to the MCU. Therefore precious data that could be used while developing a power-awareness scheme to provide feedback of the radio operation, like e.g. received signal power (for link quality estimation) **cannot be obtained by any means**. Finally, it was an IMEC design decision to wire-bond the SIMO and SOMI (see section 2.1.2) pins of the MCU outside the unit and prior to their connection to the data pin of the Radio Chip. This, in fact, complicated the data communications to and from the Radio as SPI could be used for sending data to the radio but not for clocking out data from the radio. In order to overcome this, the technique of bit-banging was employed (see section 2.1.2). It should be noted that the limitations described above were the sources of many difficulties (discussed in later chapters) that the Group had to overcome in the course of this project.

2.1.4 Humidity, Temperature and Light Sensing Hardware

For humidity and temperature measuring purposes, the Cubes are equipped with the Sensirion SHT 15 [SHT15XX] multi sensor module. The SHT15 can be configured to measure either humidity or temperature by setting a "measurement mode" register accordingly. When a measurement has to be taken, the MCU initiates it by issuing a command to the sensor module. After the measurement has been performed, the SHT15 interrupts the MCU and delivers calibrated, digital data thanks to the onboard to the sensor module, Analog-to-Digital converter. The data is delivered to the MCU via a "serial interface circuit on the same chip"[SHT15XX].

Also in the bottom layer of the Cube resides the Light-Dependent Resistor (LDR), an active electronic component used for capturing changes in illumination. This component is connected to one of the eight ports of ADC12 of the MSP430. The basic idea behind this circuit setup is to measure changes in the voltage across the LDR. Ohms law states that changes in the resistance of a circuit will cause changes in the voltage across that varying resistor due to a different voltage drop, provided that the current that flows through the circuit stays constant. In the LDR case, the resistance varies as the illumination varies, causing thus variations in the voltage measured by the ADC12. The latter deals with converting the analog voltage measurements to 12-bit digital representations for further use.

2.1.5 Auxiliary Hardware

Throughout the project, the Group has utilised several pieces of hardware and equipment for various purposes. For the purpose of downloading code to the target Cubes, the USB interface & programming board was used that was provided by IMEC. The programming board that can be seen in the picture below, was also a means of receiving information from the sensors for debugging and monitoring purposes. In addition to the

USB programming board, a programming connector for parallel port connection (JTAG) was provided which was used to some extent.

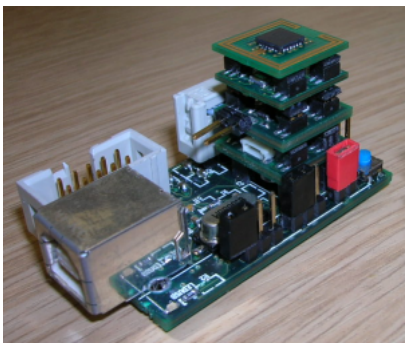

Figure 2.3: The IMEC USB Programming Board. Here with a Cube plugged.

Besides the Stack version of the IMEC Sensors, a similar in terms of radio functionality module was provided that was in the form of a USB stick. That particular version does not include any sensing hardware on board. The USB module was not used extensively in the project since it was found that it featured a 4MHz crystal instead of the 32.7KHz that the Cubes utilise. TinyOS unfortunately assumes a 32.7KHz crystal as its clock source, a fact that automatically introduces the need of complex changes to the OS when it comes to port to a platform with a different clock source. Other equipment that was involved in the project include:

- Hewlett Packard 54540C Digital Storage Oscilloscope
- Farnell LT30-2 Laboratory Power Supply Unit
- Mastech M-830B Digital Multimeter

2.1.6 The Telos and Mica Platforms

The IMEC Cube is an amazing example of modern Electronic Engineering and it represents the trends in sensor networks hardware. There exist though a number of similar platforms which are in general bigger in size and provide similar functionality. The most important are the Telos and Mica motes. Mica was the platform that TinyOS was originally built around and as expected, it has significantly influenced the OS in its initial releases (TinyOS was initially running only on Micas). The Telos platform which was developed at the University of California, Berkeley, brought up the requirement of porting TinyOS to other platforms that employed different MCUs and Radio chips. Along the process, the libraries available and the hardware interfacing of TinyOS was significantly enriched, to include components that are not Mica specific. Unfortunately, though, some libraries and applications for TinyOS are still closely "attached" to the Mica platform, making their use difficult on other. An attempt of making TinyOS more platform independent is the ongoing development of its second version (TinyOS 2.0).

2.2 Analysis of Operating Systems

This section provides a detailed description of the initial analysis. One of the key tasks in any project is to perform the initial analysis. The analysis can be on any of the topics including, choosing a right protocol/algorithm, selecting a right simulator for large scale simulation, or determining the right hardware components while building a prototype model. The analysis in our project is twofold. The first phase involves in selecting a right operating system for the wireless sensor Cube with the specified hardware modules as described in Chapter 3. The second phase is to port the Operating system onto the sensor Cube. This involves in executing the platform independent kernel on the Microcontroller, and driving individual hardware components (Radio transceiver and Sensor unit).

Designing an Operating system for sensor nodes has a unique challenge in making the system lightweight and allowing abstractions that provide a rich enough execution environment while staying within the limitations of the constrained devices. We explored the following options to decide on a right Operating environment:

2.2.1 Building from Low-level libraries

The IMEC Hardware Sensor Cubes were supplied with some basic low-level C libraries. Using these libraries, it was possible to establish a point-to-point communication between a Sensor node and the base station (a small USB stick connected to the USB port of the Desktop Computer). An application which uses a Centralized TDMA scheme was built on top of these libraries. As mentioned earlier, these libraries are only useful for basic operations and lacked the following:

- A proper Operating system kernel for scheduling tasks, performing memory management and other vital aspects.
- It was extremely difficult to write new applications, as the application programmer had no base kernel architecture on top of which s/he can write their applications.
- From a Routing Protocol perspective, a centralized MAC was not a right fit. Considering the density and dynamicity of the Sensor network, it will be very difficult to maintain time synchronization between all the nodes in the network.
- To perform exhaustive testing, large scale simulation has to be done and we lacked a Simulator. Of course there are lot of Network simulators available [NS2] [BRE1992], but none of these are suitable for specific features of the sensor networks. Also the Simulators for Wireless sensor networks are application specific and should allow simulation of certain physical characteristics like noise, variation, uncertainty to execution, possibly providing a lossy model etc.
- We also explored the option of adapting one of the existing simulators to meet our specific needs. This needs a substantial learning curve in understanding the

existing features of the simulator and additional time to be spent in adding new features. This could be a project by itself.

2.2.2 Contiki OS

Contiki [DUN2004] is an Operating System developed for resource constrained Wireless devices. It provides dynamic loading and unloading of individual programs and services. The kernel is event-driven, but the system supports preemptive multi-threading that can be applied on a per-process basis. Preemptive multi-threading is implemented as a library that is linked only with programs that explicitly require multi-threading. Contiki has been implemented in C language and has been ported to a number of microcontroller architectures including the Texas Instrument MSP430 and Atmel AVR. The single unifying characteristic of today's platforms is the CPU architecture which uses a memory model without segmentation or memory protection mechanisms. Program code is stored in a reprogrammable ROM and data in RAM. Contiki has been designed so that the only abstraction provided by the base system is CPU multiplexing and support for loadable programs and services. Other abstractions can be implemented as libraries or services and provide mechanisms for dynamic service management.

Module	Code size (AVR)	Code size (MSP430)
Kernel	1044	810
Service Layer	128	110
Program Loader	-	658
Multi-threading	678	582
Timer Library	90	60
Replicator Stub	182	98
Replicator	1752	1558
Total	3874	3876

Figure 2.4: Size of compiled code in bytes (taken from [DUN2004])

2.2.3 TinyOS

TinyOS [HIL2000], designed at UC Berkeley, is an open-source Operating system for Wireless embedded sensor networks. It features a component based architecture which enables rapid innovation and implementation while minimizing code as required by the severe memory constraints inherent in sensor networks. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools, all of which can be used as-is or be further refined for a custom application. It also uses an Event-driven execution model that enables fine grained power management, yet allows the scheduling facility made necessary by the unpredictable nature of wireless communication and physical world interfaces. TinyOS has been ported to numerous platforms and sensor boards. It has also been used by a wide community in simulation

to develop various algorithms and protocols. Various Research groups and companies use TinyOS and are actively contributing code back to the community.

Component Name	Code size (bytes)	Data size (bytes)
Multihop Router	88	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
Packet	334	40
Radio_byte	810	8
RFM	310	1
Photo	84	1
Temperature	64	1
UART	196	1
UART_packet	314	40
I2C_bus	198	8
Processor_Init	172	30
TinyOS Scheduler	178	16
C runtime	82	0
Total	3450	226

Figure 2.5: Code and data size breakdown of an example system. Only processor init, TinyOS scheduler and C runtime are required for every application, other components are included as needed. (taken from [HIL2000])

2.2.4 Sensor Operating System (SOS)

SOS [HAN2005] is an Operating system for mote-class wireless networks developed by Networks and Embedded systems Lab at UCLA. SOS uses a common kernel that implements, messaging, dynamic memory, module loading and unloading and other services. It uses dynamically loaded software modules to create a system supporting dynamic addition, modification and removal of network devices. Modules send messages and communicate with the kernel via a System jump table but can also register function entry points for other modules to call. SOS has no memory protection but the system nevertheless protects against common bugs. It uses dynamic memory both in memory and application modules easing programming complexity and increasing temporal memory reuse. Priority scheduling is used to move processing out of interrupt context and provide improved performance for time-critical tasks.

Note: SOS is a young project developed recently (in mid 2005) and hence was not available for us when we were exploring the various Operating systems to be used.

Platform	ROM (bytes)	RAM (bytes)
SOS Core	20464	1163
Dynamic Memory Pool	-	1536
TinyOS with Deluge	21132	597
Bombilla Virtual Machine	39746	3196

Figure 2.6: Memory footprint for base operating system with ability to distribute and update node programs compiled for Mica2 motes. (taken from [HAN2005])

2.2.5 Comparison between TinyOS and Contiki OS

Feature	Contiki	TinyOS
Run-time Dynamic Reprogramming	**Possible**. Contiki system is divided into two parts: Core and Loaded programs. Core consists of Kernel and a set of base services. Loaded programs can be loading and unloading individually, at run-time.	**Not Possible**. All Components of TinyOS (including Kernel, libraries and application programs) are compiled and built as a static image. However, run-time reprogramming can be achieved using Mate OS (a virtual machine for TinyOS devices)
Code size	3876 bytes	3450 bytes (including all components)
Event Kernel Size	810 bytes (includes different services provided)	178 bytes
Event Queue Scheduler	FIFO and poll handlers with priorities	Just FIFO

2.2.6 Other Embedded Operating systems

Name	Preemption	Protection	ROM size	Configurable	Targets
pOSEK	Tasks	No	2K	Static	Microcontrollers
pSOSystem	POSIX	Optional		Dynamic	PII ->ARM Thumb
VxWorks	POSIX	Yes	286K	Dynamic	Pentium -> Strong ARM
QNX Neutrino	POSIX	Yes	>100K	Dynamic	PII -> NEC chips
QNX Realtime	POSIX	Yes	100K	Dynamic	PII -> 386's
OS-9	Process	Yes		Dynamic	Pentium -> SH4
Chorus OS	POSIX	Optional	10K	Dynamic	Pentium -> Strong ARM
Ariel	Tasks	No	19K	Static	SH2, ARM Thumb
CREEM	data-flow	No	560	Static	ATMEL 8051

Figure 2.7: A comparison of selected architecture features of several embedded Operating systems (taken from [HIL2000])

2.2.7 Rationale for choosing TinyOS

After spending sufficient amount of time exploring the various Operating systems and their features, we elected to use TinyOS for the following reasons:

- Among embedded operating systems for Sensor networks, TinyOS is the oldest and most matured.
- TinyOS has already been ported to several sensor platforms, including the one consisting of MSP430 microcontroller, which is our main interest
- A Hardware abstraction layer for MSP430 has already been written and is available in TinyOS distribution code.
- The build tool chain is well organized and available with very minimal modifications for a new platform
- Consists of a variety of applications and routing protocols
- Also comes with a simulator, which uses a probabilistic bit error model, which can capture network behaviour at high fidelity while scaling to thousands of nodes.
- TinyOS is currently used by a wide range of active researchers, communities, universities and other development organizations. An active and well organized mailing list exists which is a valuable resource

2.3 The TinyOS Operating System and Development Tools

2.3.1 TinyOS and the NesC Programming Language

TinyOS is an open source operating system developed at University of California, Berkeley and is targeted to embedded systems and platforms with memory constraints. Originally developed around the MICA platform it is one of the most popular OS solution for wireless sensors, with successful ports to a number of platforms like MICAZ, TelosA and TelosB. Despite what its name suggests, TinyOS is not an operating system within the precise meaning of the term. It would be more appropriate to classify TinyOs as a programming framework for developing embedded systems, which, by providing a well defined set of components, allows the building of an application specific OS that can have different degrees of functionality depending on needs.

TinyOS inherits the programming model of NesC, an extension of the C programming language. In NesC, Components are the basic building blocks of an application and contain code which provides the functionality described in corresponding Interfaces. Creating an application in TinyOS mainly involves the coding of the modules that will contain the application code and creation of a Configuration that dictates which components are used by the application and how they will interact, This approach actually goes

down to the Operating System itself and, depending on the application it is compiled with, it only uses the required modules every time. A good description of the TinyOS as a whole is "a set of reusable system components along with a task scheduler"[LVS2004].

Besides the differences in the code organization and some syntax differentiation from C, NesC also imposes some limitations in an attempt to enable code to be more robust and efficcient. The main differences are:

- NesC does not allow the use of function pointers, thus making the call graph of a program known at compile time. This allows optimization by reducing message passthrough between components.
- Dynamic memory allocation is not supported and the components of a program can only statically declare, preventing thus memory fragmentation and errors due to allocation failures at runtime.

There are two types of components in the NesC programming language: Namely, Modules and Configurations. The former include code and provide and use bidirectional Interfaces. The latter are a description of how the various modules comprising the working code of an application will interact, by defining which interfaces are used and by which modules. Whenever a module is required to provide multiple instances, a Parameterized Interface is used that allows the creation of many instances of that particular Interface, each one differentiated by a small integer identifier.

Communication and collaboration between the components of an application is achieved by the means of Commands and Events. In order for a module to be usable it has to provide a set of commands and events, specified in its corresponding interface, The commands provided by a component are there to be used by other modules for requesting a service, whereas the events signaled by the module itself represent a way of informing others of a service routine completion or a hardware interrupt. "From a traditional OS perspective, commands are analogous to downcalls and events to upcalls" [LVS2004]. Ongoing lengthy but non time-critical computations in the execution model of TinyOs are represented by Tasks. Tasks can be posted to the scheduler in a non-blocking fashion, making the code that performs the post very responsive. The FIFO scheduler of TinyOS then runs the tasks at a later time. The fact that tasks run to completion but can be preempted by events and interrupt handlers, effectively provides to the developer a concurrency model of finer granularity compared to e.g. threads that run indefinitely.

Regarding abstractions concerned with communications within TinyOS, the Active Messages represent the basis for them. The Active Message interface specifies small (36-byte) packets with handler IDs and from a user's point of view it is "an unreliable, single-hop datagram protocol, and provides a unified communication interface to both the radio and the built-in serial port (for wired nodes such as basestations)"[LVS2004].

2.3.2 The TOSSIM Simulator and TinyViz

TOSSIM is a discrete event simulator for TinyOS applications. Its simple, yet powerful simulation engine is capable of simulating application scenarios involving thousands of

sensors and at network bit granularity. A great advantage of TOSSIM is the fact that TinyOS applications literally run unchanged within its framework. This is achieved by replacing the lower level modules of TinyOS that communicate directly with hardware, with modules that emulate their functionality on the PC. An equally interesting and helpful feature is the ability of the simulator to connect with GUI frontends, providing thus an easy way of monitoring and controlling a simulation.

The modular architecture of TinyOS "abstracts each hardware resource as a component"[LVS2003] and it is this fact that enables TOSSIM to simulate TinyOS applications without any changes to the application code. The developers of TOSSIM have created a framework that maintains the interfaces of TinyOS subcomponents and seamlessly integrates to it emulated components that run on the PC. This was achieved simply by replacing components of the OS that sit on top of actual hardware with appropriate emulators. The emulated raw hardware in TOSSIM includes the following modules:

- Analog-to-Digital Converter (ADC)
- Clock
- EEPROM
- Boot sequence component
- Radio Stack components
- Sensing modules

For the purpose of simulation of wireless communications between nodes, TOSSIM employs a simple model in which the network is a directed graph. Within the network graph, individual sensors are represented by vertices, each one with a bit error probability associated with it. State information is also maintained for every node concerning its particular interests for information "on air". The above simple model allows the reproduction of both perfect and non perfect transmission conditions by varying the bit error rates associated with nodes. Furthermore, the hidden terminal problem can also be reproduced within the simulator by adding or removing connections directed to, or originating from, specific nodes within the graph. Finally, due to the bit-level granularity of the communications, the functionality to capture problems inherent to packet transmission like, for example, corruption of data, is also available.

Taking into account the functionality that the above provide, developers can vary the accuracy and complexity of the underlying radio behaviour in simulations at will. It is worth noting that models of the Radio component of TOSSIM are themselves self-contained and separate to the simulation engine. This fact allows the following extremely desirable functionality:

- Link probabilities, controlling bit error rates, can be user-controlled and changed at runtime.

- Transmission (represented by special events) propagate via simulated input channels associated with each and every note participating in the simulation.
- Built in radio model for single-cell, error-free transmissions mainly for testing protocols in single hop situations only.
- Built-in radio model for multihop, error-free transmissions.

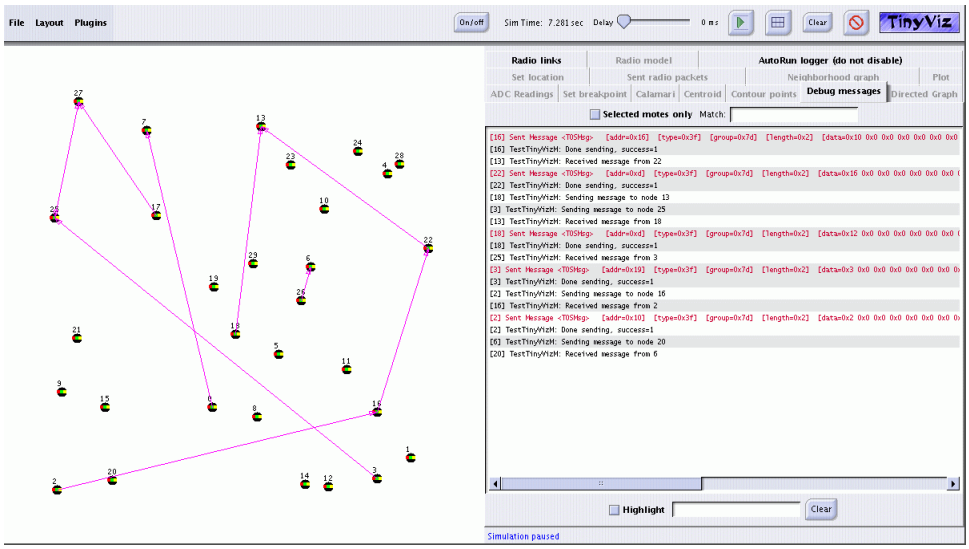

Figure 2.8: The TinyViz visualisation tool in action.

Although the above functionality enables the developer to create more realistic and "tailor-made" simulations, an equally important feature of TOSSIM is its ability to provide "communication services for interacting with external applications" [LVS2003]. Connection to such services is achieved over TCP sockets that TOSSIM opens and on which it then awaits for connections. Once connected, an external component can monitor or influence the parameters of a simulation by querying and/or setting them. Information made available from TOSSIM includes user-added debug messages on the source code, packets sent/received and readings from sensors. Commands issued to TOSSIM from external components can change bit error probabilities for specific links, activate/deactivate specific nodes, set sensor measurements values and inject packets. TinyViz, which can be seen in the screenshot above, is the TOSSIM visualisation tool and is a good example of what communication services can provide.

2.4 Chapter Summary

In this chapter hardware components that comprise the IMEC Sensor Cubes have been presented and their capabilities and limitations were also mentioned. By selecting and presenting some key sub-components of the sensor (Microcontroller Unit, Radio Chip,

Temperature-Humidity Sensors and Photosensor), the necessary basis to understand the following chapter related to the porting of TinyOS has been established. Through the analysis of various operating systems for sensor networks, the selection of TinyOS has been also justified. The chapter concluded with a short account on the chosen operating system and its simulator, TOSSIM. The following chapter aims to provide an insight to the process involved while porting TinyOS.

Chapter 3

Porting TinyOS onto IMEC sensor cubes

TinyOS uses an event driven approach, meaning every processing step is triggered by some kind of event. Every triggered "*task*" is enqueued into a worker queue and processed by the main loop until none is left. After that, TinyOS waits for the next event to occur. To simplify the entire building process, the whole operating system and the applications are built together in one step. The nesC compiler combines all components of the application and operating system and builds a C file which is passed to a C compiler. Porting of TinyOS is lot different (compared to porting a traditional operating system) and simpler. The following sub-sections of the chapter describes the various steps and processes involved in porting:

3.1 Bootstrapping and system initialization

The TinyOS distribution consists of a "**tos**" directory, which contains the whole operating system and all platform definitions. Some standard applications which serve as examples on how to use different parts of the OS are available in a directory "**apps**". The tos directory itself consists of:

- (system), the platform independent operating system part
- (interfaces), common interface definitions
- (lib), a library with commonly used functions
- (platform), platform dependent hardware definitions and access driver functions
- (sensorboard), definitions for different sensor boards that can be used in combination with the motes

The Bootup process can be best explained by an example application. The following application (Blink), is analogous to "Hello World" programs in many systems. It sets up

a hardware timer to Toggle a LED on and off. Apart from the hardware clock source, "Blink" does not use any peripherals of the system. The application demonstrates the following:

- The TinyOS system kernel can successfully be executed on MSP430 Microcontroller
- The build tool chain is in place

Blink Application:

```
configuration Blink {
}

implementation {
  components Main,BlinkM,SingleTimer,LedsC;

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

The bootup process of TinyOS starts in tos/platform/msp430/Main.nc:

```
module MainM
{
   uses command result_t hardwareInit();
   uses interface StdControl;
}
implementation
{
   int main() __attribute__ ((C, spontaneous))
   {
      call hardwareInit();
      TOSH_sched_init();
      call StdControl.init();
      call StdControl.start();
      __nesc_enable_interrupt();

      for(;;) { TOSH_run_task(); }
   }
}
```

The first call in the main function is "hardwareinit()", which is connected to HPLInit via the Main.nc component:

```
configuration Main
{
   uses interface StdControl;
}
implementation
{
   components MainM, HPLInitC;

   StdControl = MainM.StdControl;
   MainM.hardwareInit -> HPLInitC;
}
```

HPLInitM calls TOSH_SET_PIN_DIRECTIONS() from hardware.h, which in turn calls macros to set the direction registers of the microcontroller:

```
module HPLInitM
{
   provides command result_t init();
   uses interface StdControl as MSP430ClockControl;
}
implementation
{
   command result_t init()
   {
       TOSH_SET_PIN_DIRECTIONS();
       call MSP430ClockControl.init();
       call MSP430ClockControl.start();
       return SUCCESS;
   }
}
```

```
TOSH_ASSIGN_PIN(RED_LED, 1, 0);
TOSH_ASSIGN_PIN(GREEN_LED, 1, 2);
TOSH_ASSIGN_PIN(YELLOW_LED, 1, 3);

void TOSH_SET_PIN_DIRECTIONS(void)
{
    //LEDS
    TOSH_SET_RED_LED_PIN();
    TOSH_SET_GREEN_LED_PIN();
    TOSH_SET_YELLOW_LED_PIN();
    TOSH_MAKE_RED_LED_OUTPUT();
    TOSH_MAKE_GREEN_LED_OUTPUT();
    TOSH_MAKE_YELLOW_LED_OUTPUT();
```

```
        . . .
        . . .
    }
```

After setting the pin directions, hardware clock source is initialized and lastly StdControl.init() and start() are called. These two commands dispatch to all connected (or wired) modules. In the Blink application, SingleTimer.StdControl and BlinkM.StdControl are the connected modules. After enabling interrupts, TinyOS system kernel enters an infinite loop calling TOSH_run_task() repeatedly. This function processes all pending tasks in the queue until none is left and then enters sleep mode until an Interrupt request (IRQ) occurs.

3.2 Platform Definition

There exists a sub-directory for each new platform under the platform directory. This sub-directory should contain the following two files namely, "*.platform*" and "*hardware.h*". hardware.h is used to assign functions to the pins of the microcontroller. The macros TOSH_ASSIGN_PIN and TOSH_ASSIGN_OUTPUT_PIN which are defined in msp430hardware.h are used for this purpose. These two macros define functions to set, clear, read, toggle, to make input/output port etc. These functions are called through TOSH_SET_PIN_DIRECTIONS(). This function, apart from setting the pin directions also does some preliminary initialization of the hardware before the first TinyOS component is used. .platform is described in the next sub-section.

The platform directory also contains components to access hardware features and other I/O subsystems specific to the hardware platform, like ADC, MSP430Timer, etc. All these are placed in a separate directory "*msp430*" under the platform directory. The components in this directory mask any components in system and lib directories.

3.3 Setting up the Build tool chain

The TinyOS compilation environment has three parts:

- a "make system"
- a nesC compiler driver (ncc)
- the nesC-to-C compiler

The "make system" is like an IDE (minus the text editor). It provides simple ways to compile, install and otherwise manage TinyOS programs for different platforms, using a set of standard options. To do this, it invokes 'ncc' (to compile) and various mote-programming tools ('uisp', 'msp430-bsl', etc). "ncc" is the driver for the nesC compiler. It is (purposefully) designed to work as an extension to "gcc" (the driver for all the GNU compilers). In particular, ncc accepts all gcc options and will also happily compile C files,

assembly code, etc. It does this by simply invoking gcc with a few extra options to make gcc recognise .nc files. "nesc1" is the actual nesC compiler. It reads in nesC components and C source files, and outputs a C file (usually found in build/<platform>/app.c, though this is just the result of the -fnesc-cfile=build/<platform>/app.c option added by the "make system". This compiler currently has a limitation that it does not have most of "C99" features and ignores #pragma's.

The first step is to make "ncc" recognise the new platform; "ncc" requires each platform directory to contain a .platform file (written in perl) which specifies compilation options for that platform. For "ncc" to accept -target=<newplatform>, we must either place the <newplatform> directory in tos/platform, or specify -I<path to newplatform> as an option to "ncc".

Typical options specified in a .platform are as follows:

```
@opts = ("-gcc=msp430-gcc",
         "-mmcu=msp430x149",
         "-fnesc-target=msp430",
         "-fnesc-no-debug");
```

The next step is to get "nesc1" understand the new platform. The "env" target documented in tinyos-1.x/doc/nesc/envtarget.html explains how to configure nesC compiler for the new platform.

The last step is "Extending the make system":

- The "**make**" system directory contains the following set of files: - *.target* files (Valid make target) - *.extra* files (dummy target for defining extra make variables) - *.rules* files (part of "msp" subdirectory)
- In tools/make directory the new platform name (imec) has to be added to all.target file
- Also a new file with name imec.target has to be created in tools/make directory.
- A README file in tools/make directory describes the procedure in detail

3.4 Establishing Radio Communication with Nordic nRF2401 Transceiver

After porting the hardware independent system kernel of TinyOS, the next step is to write code to drive the Radio transceiver through MCU. This is achieved in the following described steps

3.4.1 Performing hardware pin settings between MCU and Radio Chip

As explained in Section 4.3.2, the hardware.h file (in the platform specific directory) is used to assign names to the individual pins of the MCU.

3.4.2 Writing hardware layers as per the TEPs (TinyOS Extension Proposal)

In order to simplify management, reading and tracking development, TinyOS defines a standard called as TEP (TinyOS Extension Proposal). TEPs [TEP] are documents that describe the proposals. Two TEP's are defined to write the radio code: Radio Physical Layer and Radio Link layer

These provide the hardware abstraction architecture for Radio Components used in TinyOS. The hardware abstraction of a radio component can be divided into three separate layers. Following the tradition of three layer Hardware Abstraction Architecture (HAA), these layers are labelled as Hardware Physical Layer (HPL), Hardware Adaptation Layer (HAL) and Hardware Interface Layer (HIL)

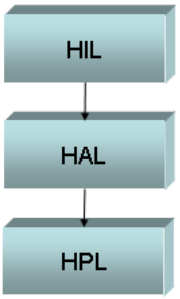

Figure 3.1: Hardware Abstraction Architecture

The three layers are divided in such a way that the HPL and HIL are radio dependent, while the HAL both radio and MCU dependent. Thus the HPL and HIL are platform independent and do not rely on a specific MCU for their implementation.

- Hardware Physical Layer:

  The Radio HPLlayer is the bottom most layer in the stack and is highly radio dependent. Radio configuration, setting pin directions, selecting radio modes (like Shockburst, direct, powerdown, standby) are some of the functions that are implemented in this layer.

- Hardware Adaptation Layer:

  This layer is responsible for connecting the radio to the microcontroller used on a given platform. Normally this is the only layer which will be PLATFORM dependent. Since this layer has intimate knowledge of what process is being used to communicate with the radio, it has a lot of flexibility in how it chooses to implement things. The actual implementation and definition of the interfaces provided by this layer is completely arbitrary and must only be made to match those expected by the HIL and HPL layers that it connects to.

- Hardware Interface Layer:

  This layer is used to provide a platform independent interface to the radio hardware. This interface does not exist as of now and is expected to evolve in the future (may be in TinyOS 2.0).

DSYS25 [DSYS] is a sensor platform comprised of an Atmel AVR ATMEGA 128 microcontroller and a Nordic nRF2401 Transceiver. The Hardware Abstraction layers from this platform has been taken as reference and the code has been ported to IMEC platform comprising of MSP430 Microcontroller. Most of the Radio Transceiver code has been retained and the MCU specific code has been replaced with that of MSP430 MCU. The code is organized as follows:

- TinyOS distribution consists of a directory (msp430) which has the Hardware physical layer interfaces for MSP430 MCU and its peripherals (including UART, SPI, etc.). The following HPL files are used namely, HPLSpiM, HPLUARTC (including HPLUART01M, HPLUART1M) and HPLPowerManagement. Also the modules, MSP430Clock (interface for hardware clock source), MSP430TimerM and MSP430Interrupt are used.

- HPL and HAL layers are implemented in the following files, HPLnRF2401, HPLnRF2401Rx and nRF2401Control. All three files are radio dependent and MCU dependent. Strictly speaking, the HPLnRF2401 must not be MCU dependent. It provides the following interfaces:

```
command result_t init();                          //Initialize nRF2401 pins
async command result_t beginConfigMode();         //Enter configuration mode
async command result_t endConfigMode();           //Exit from config mode
async command result_t setTXShockBurstMode();     //Set to Transmit mode
async command result_t setRXShockBurstMode();     //Set to Receive mode
async command result_t SBurstSend();              //Send ShockBurst frame
async command result_t setStandByMode();          //Set nRF2401 to standby
async command result_t setPowerDownMode();        //Set nRF2401 to powerdown
async command result_t write_byte(uint8_t data);
async command result_t write_bit(uint8_t data);
```

- HPLnRF2401Rx provides an interface to read a byte from the Radio chip and an event which would be triggered as soon as a byte is read:

```
async command uint8_t read_byte();    // Read a byte from the radio
async event result_t UveGotPckt();    // Signal reception of a packet
```

- Lastly nRF2401Control uses the commands provided by the HPL layer to provide following logical abstraction commands to the user:

```
// Select Channel 1 or 2
async command result_t SelectChannel(uint8_t channel, uint8_t mode);

async command result_t TxMode();
async command result_t RxMode();
```

Apart from this, it also implements the events to handle packet reception.

3.4.3 Verifying Radio communication using an example application

After successfully constructing the HPL and HAL layers, the actual Radio communication was verified by using two applications (CntToLedsAndRfm and RfmToLeds). CntToLedsAndRfm is an application that maintains a counter on a timer and displays the lowest three bits of the counter value on its LEDS (unfortunately IMEC has only one LED, hence only the lowest significant bit of the counter will be displayed). It also sends out each counter value over the Radio.

RfmToLeds listens for messages over the radio and as soon as it receives a message, it sets the MCU's LED with the least significant bit of the received counter value. By using two Sensor nodes, (one with CntToLedsAndRfm and the other with RfmToLeds) Radio communication has been tested.

3.4.4 Porting TinyOS Sensor code

As described earlier, the IMEC sensor cubes feature three types of sensing devices. Incorporating support for these sensors on the IMEC platform consisted in the following actions:

- Sensirion SHT15 Humidity and Temperature sensor:

  It was found that the Telos platform features an SHT11 sensor whose one-wire protocol is the same as on the SHT15. Although this sensor does its own analog-to-digital conversions, the components to support it implement the standard *ADC* and *ADCError* interfaces. This design was deemed quite solid as it provides a consistent external view of the sensor to other TinyOS components, according to the specifications in TEP 101 - Analog-to-Digital Converters.

  To support our platform, we added the appropriate pin names and settings in `hardware.h`, then simply copied the relevant files (`Humidity*.nc` from `tos/platform/telos/`) and modified the interrupt pin to P1.1 to match (in `HumidityProtocolC.nc`).

- Light-dependent resistor (LDR):

  Given that the ADC12 is very well supported in TinyOS' implementation of the MSP430 platform and designed according the hardware abstractions specified in TEP 101 - Analog-to-Digital Converters, it was pretty straightforward to enable the light sensor on the IMEC prototype. Using the existing code base as a reference we created the files `LightSensor*` to read the LDR output on ADC12's input channel 0 (pin P6.0, configured as peripheral function).

- MSP430 internal temperature sensor:

  This internal sensor is connected to input channel 10 on the ADC12 (refer to previous point) and is directly supported in the MSP430 platform (see files `InternalTemp*` in `tos/platform/msp430/`).

3.5 Chapter Summary

The chapter described the procedure for Porting TinyOS onto IMEC platform, including Bootstrapping and system initialization, Platform Definition and Setting up the Build tool chain. It then described the various steps and processes used in porting Radio communication code with verification using an example applcation. And lastly a short description of porting TinyOS Sensor code has been provided. The next chapter describes the Design and Implementation of a Reliable and Energy-efficient MAC.

Chapter 4

Design and Implementation

Once we have a proper operating environment, the next vital task is to design a working Energy-efficient protocol stack. The term "*Power-aware*" or "*Energy-efficient*" is often used in Wireless Sensor network research, as the common objective is to maximize the network lifetime. Since sensor nodes are assumed to be rendered useless when they run out of battery power, the individual layers of a protocol stack must be **Energy-efficient** by reducing any potential energy wastes. Several papers [DEM2005] [JON2001] [BAN2003] [HEI2000] have been proposed for building energy efficiency. Another challenge in Wireless sensor Networks is the severe resource constraints that will reduce the scope of various design choices. The capabilities of sensor devices are very different from traditional nodes in a computer network. The devices have a very limited amount of storage, processing power and most importantly, energy resources.

The Design of the project is further divided into two phases. The first phase involves in designing a suitable "*Energy-efficient*" MAC layer as per the laid out Requirements in Chapter 2. The next phase is to select an appropriate Routing protocol to fit on top of the MAC layer. Due to time constraints we were unable to design a Routing Protocol and had to concentrate more on MAC and other layers of Protocol stack. The chapter is divided into the following sections:

- Design of Energy-efficient MAC layer
- Routing Protocol (Surge)

4.1 Design of Energy-efficient MAC

With Wireless sensor networks, not only do we have the contention for the media, we must also try and ensure that the receiver and transmitter can actually communicate. Thus the need for a MAC arises. Several MAC methods can be used as mentioned below:

MAC	Description
ALOHA	Originally designed to send packet data over radio networks. It has a simple procedure wherein you transmit the data and if a collision occurs, wait for random time and re-transmit.
CSMA	Sense the channel; transmit data only if the channel is free. Collisions can be detected at the receiver.
TDMA	In Time Division Multiple access (TDMA), the channel is split into time-frames which is further subdivided into time-slots. Each transmitter will be allocated a time-slot during which it can transmit, using the whole of the channel.
FDMA	Frequency Division Multiple Access (FDMA) scheme splits the RF spectrum into fixed number of channels. One of them will be maintained as a signalling channel. A source requests for a channel when it has data to transmit
CDMA	Code Division Multiple Access (TDMA) scheme splits the RF spectrum into fixed channels as in FDMA. The use of a channel is based on a pre-assigned hopping sequence, which controls the transmitter by asking it to transmit part of the message on a particular channel, and then hop to another channel.
DFWMAC	Distributed Foundation Wireless MAC (DFWMAC) is a four-way handshake mechanism also known as RTS-CTS scheme, which ensures confirmed delivery of data frame

Figure 4.1: Various MAC schemes

4.1.1 Design constraints

As explained in Chapter 2.1.3 the following were some of the constraints that we had to work with, while designing the MAC layer.

- Nordic Radio transceiver cannot operate in Direct Mode

  Due to lack of high speed clock source, we cannot use the Transceiver's Direct mode, where we would have had better control of the Radio.

- ShockBurst FrameSize (256 bits)

  The Control header and the payload cannot exceed 256 bits (32 bytes) which is the maximum frame size in Shockburst mode

- Configuration word usage

  The Radio's Configuration word cannot be altered while it is transmitting/receiving data. In the initial design phase, we were thinking of sending/receiving link level acknowledgements on the same channel as the data channel. Remember that the Radio can be operated in two separate channels (which are separated by a frequency of 8Mhz). The approximate time taken to transmit a full Shockburst frame (of size 32 bytes) at a speed of 250kbps is 1ms. Using the Data width field (in the configuration word) we can alter the size of Shockburst frame, so that lesser width can be used for ACK packets. If ACK is sent in the same channel, then the Data width of configuration word cannot be altered approximately for 1ms (assuming full Shockburst size data is sent). Because of this limitation, we chose to use Channel 2 for sending/receiving ACK packets.

4.1.2 Design considerations

Before choosing a right MAC algorithm, we explored the following design options with respect to Wireless sensor networks.

- Carrier sensing

  Carrier sense involves in the radio continuously listening to the channel to determine if it is unused by any other station. The amount of Energy spent for reception is generally higher than that for transmission, because of the reason that the cost of signal processing to decode the radio signal is more complex than encoding the signal for transmission. Typically `P(rx) > P(tx) > P(idle)`, where P indicates Power.

  With carrier sense, collisions can be avoided, however it will be less energy-efficient. In order to build energy efficiency in MAC layer we decided not to use Carrier sense, rather to use a simple Aloha based protocol. A Duty cycle periodically switches the radio between stand-by mode and Receive mode. The amount of time a radio stays in either mode can be pre-configured in the Duty-cycle

- Link level acknowledgements and retransmissions

  Since carrier sensing is not used, there could be collisions in the network. Also, with the use of Duty cycle a Receiver node maybe in stand-by mode when the transmitter is sending the data and hence will not be able to receive the data. To overcome both these problems, we use Link level acknowledgements and retransmissions. As soon as the transmitter sends the data, it waits for an acknowledgement for a duration of `ACK_WAITING` time (which can be pre-configured). If no acknowledgement arrives for the above mentioned time frame, then the transmitter retransmits the packet. Remember that the transmitter cannot distinguish a packet which was either lost due to collision or because the receiver was in stand-by mode. The transmitter retransmits until it gets an acknowledgement or reaches the Maximum Retransmission count (which can be pre-configured).

- Fragmentation

  In ShockBurst mode, the Radio can transmit no more than 32 bytes of data (including ShockBurst Address, CRC and payload from MCU). The payload from MCU is an Active Message packet [BUO2003] (constructed by the TinyOS AM layer), containing control header fields and actual data (payload) information sent from application.

  An obvious necessity of fragmentation clearly comes in picture, because of the limited Shockburst frame size. 3 bytes are used for Shockburst address, 1 byte for CRC, 8 bytes for AM control header, which leaves a payload size of `32 - (3+1+8) = 20 bytes`. Considering the amount of data that is usually transmitted in sensor networks, we felt fragmentation is not needed. The application normally sends a few bytes of data (as sensor readings) and some amount of control information as

part of the Routing protocol. In this scenario, fragmentation adds an additional few bytes of fragment header which can be avoided.

- RTS/CTS

  RTS/CTS (Request to send/Clear to send) is a mechanism used in Wireless systems to avoid the of Hidden terminal and Exposed terminal problems. In the case of densely populated wireless networks, a node might not hear the transmission of a neighbouring node (hidden terminal) and hence transmits, resulting in a collision. Also a node A might overhear transmission of a neighbour node B (exposed terminal) to a node C, and C may not be in range of A. Because A overhears B's transmission, it might backoff even though it is transmitting to some other node (say D). To avoid both of these, RTS/CTS mechanism is used to establish a session before doing the transmission.

  RTS/CTS is best suited for unicast transmissions and not for broadcast communication. A Routing protocol usually performs a broadcast, majority of the times (for Route discovery and Route maintenance). Unicast communication is used only for forwarding the data to a specific node in the Route tree. Hence we decided not to implement RTS/CTS mechanism as this feature might not add a significant value to the MAC layer. The data forwarding (which uses a unicast transmission) is protected by link-level acknowledgements and retransmission, should there be a problem of Hidden terminal.

- Dual channel usage

  As mentioned above in Section 2.1.3, the Nordic Transceiver can be operated with two channels (separated by a frequency of 8MHz). Because of the limitations of configuration word usage (as mentioned in Design constraints), we decided to use Channel 1 to send data (with full Shockburst frame size of 32 bytes) and Channel 2 for sending/receiving ACK packets (with a Shockburst frame size of 13 bytes). A reduced frame size for ACK packets will utilize less power for transmitting/receiving data.

- Promiscuous mode

  Two addresses are used for data packets. The first is the ShockBurst address (with which all the nodes in the network will be configured). Any data packet is addressed to the shockBurst broadcast address (0xFFFF). Apart from this address, a Destination address field is used in Active Message header. Each node will have a ShockBurst address (set to 0xFFFF) and `TOS_LOCAL_ADDRESS`. The Destination address in the AM structure, should match the `TOS_LOCAL_ADDRESS` of the Receiver node.

  With the above design, each node will be operated in Promiscuous mode, which is an important requirement of the Routing protocol.

4.1.3 Mac Design with Idle-ARQ table management

After a thorough analysis of the various design options, we elected to use an approach based on table management with Idle-ARQ Sequence number implementation. The MAC table is designed as follows:

> The MAC layer can have no more than one outstanding packet at any given point of time. A packet is said to be outstanding, when it has been transmitted and not yet acknowledged and the maximum retransmission count has not yet been reached. Allowing multiple outstanding packets might result in a collision (with ALOHA based approach) and will also increase the complexity of sequence number management.
>
> As soon as the MAC layer accepts a packet for transmission, it locks itself for any further transmissions, until the current packet is successfully transmitted. Before performing the actual transmission of the packet, the following checks will be performed by the MacTable as described in the below flowchart.

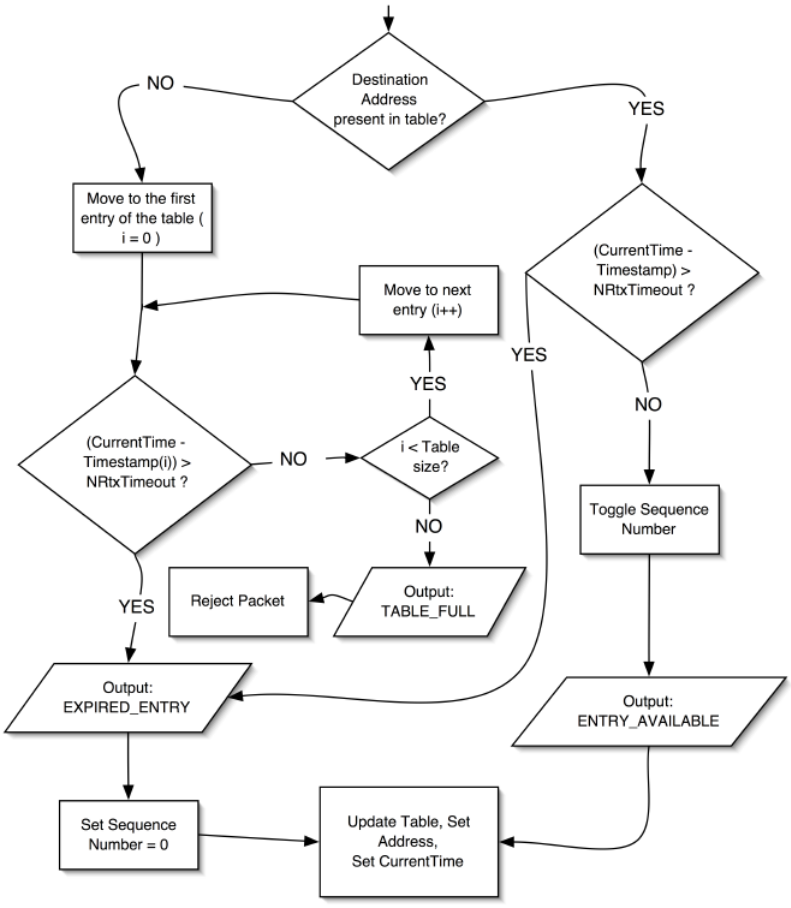

Figure 4.2: Flow chart describing the logic of the transmitter's MacTable.

A similar table is maintained at the Receiver and its operation is explained as below

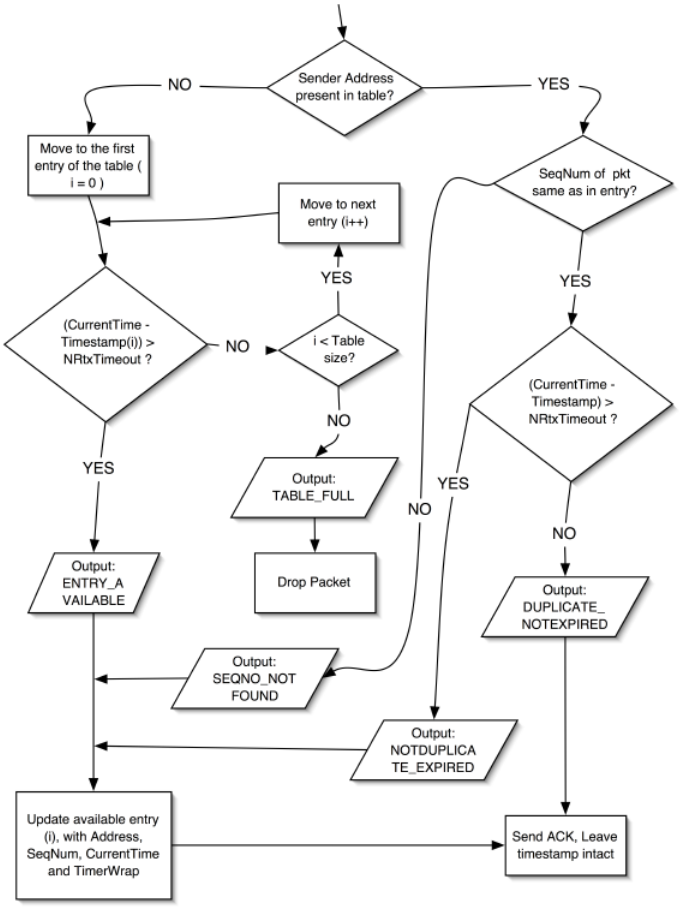

Figure 4.3: Flow chart describing the logic of the receiver's MacTable.

StateChart diagrams

The MAC layer maintains the following Radio states:

STATE	Description
DISABLED_STATE	The Radio is able to neither send nor receive data
TX_STATE	when the Radio layer is ready to send the data to the Transceiver for transmission.
RX_STATE	When the Radio layer is ready to receiver either ACK or Data packets
IDLE_STATE	When the Radio is placed in RX_STATE, with the DutyCycle switching between Standby and Listen modes.

STATE	Description
WAITING_ACK_STATE	When the Radio has performed a transmission and has switched to RX_STATE, waiting for an ACK packet.

Events that trigger a state change

EVENT	Description
init	Performs Component initialization.
start	Starts the component
stop	Stops the Component
send	Triggered when an application has data to send.
ACK_Received	Triggered when the Radio layer receives an ACK packet
Data Sent Over	As soon as data has been sent over the air.
Radio	
AckTimer fired	When the ACK_WAITING time is completed.

The following statechart diagrams displays the various states and the events that cause state transitions.

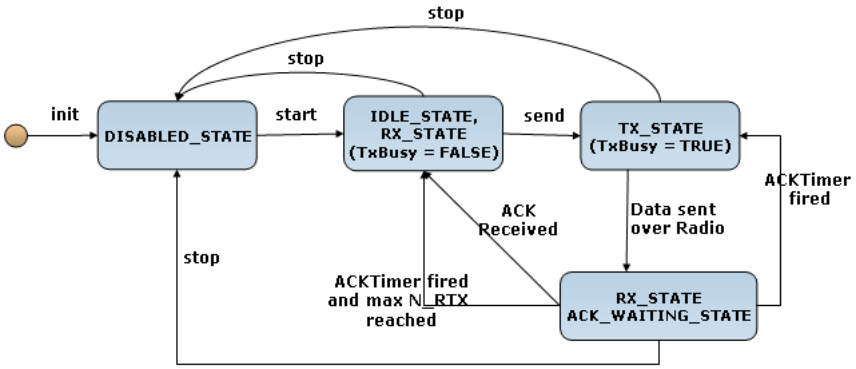

Figure 4.4: Radio State Diagram

The MAC layer also maintains the following BackOff states, related to BackOffTimer that is started to perform Random backoff (classic ALOHA protocol style):

STATE	Description
BO_OFF	The Radio layer is not in Backoff state.
BO_SENSE	In Backoff state (for a Random time)
BO_BUSYCHANNEL	Data received from a node, during Backoff State, hence the Backoff has to be restarted.
BO_DELAY	In Backoff state after restarting the Backoff Timer (again for a Random time)

Events that trigger a state change

EVENT	Description
init	Performs Component initialization.
start	Starts the component
stop	Stops the Component
send	Triggered when an application has data to send.
Data Received	Triggered when the Radio layer receives data
BackOffTimer fired	When the Random BackOff delay is completed.

The following Statechart diagrams displays the various states and the events that cause state transitions.

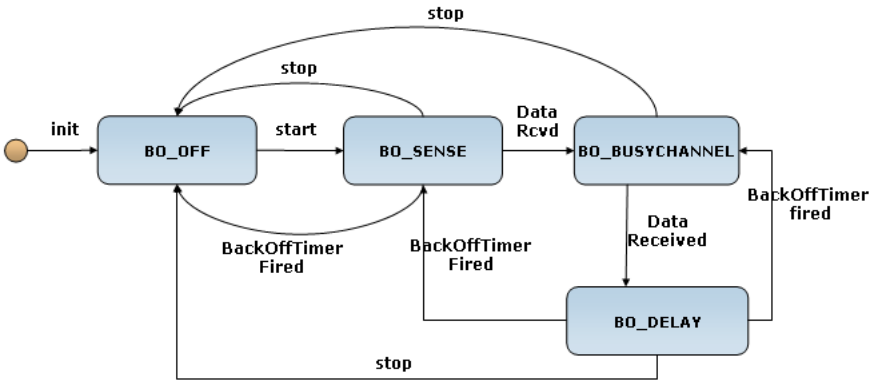

Figure 4.5: BackOff State Diagram

A DutyCycle logic is used to continuously switch the Radio between Standby mode and Receive mode. DC_LISTEN and DC_SLEEP are the states maintained and the below statechart diagram displays the events that trigger a state change.

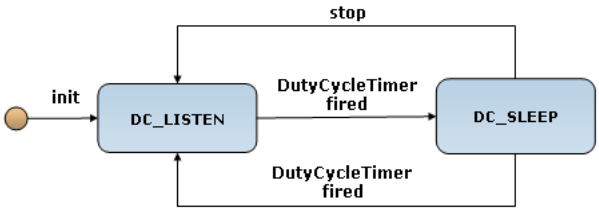

Figure 4.6: DutyCycle State Diagram

Component Interactions

The MacTable logic is implemented in a module (MacTableM.nc), which contains both the features of Transmitter and Receiver. It uses an instance of MSP430Timer (hardware TimerB of MSP430 MCU, which has a clock source of 32KHz) to read the current time and store the value in the table. The Radio layer wires to MacTableM, infact it wires to two instances: one for Transmitter and another for Receiver.

The MAC layer (in IMEC platform) consists of a top level configuration file (nRF2401RadioC) which is used to do the necessary wiring of the components to `system` modules.

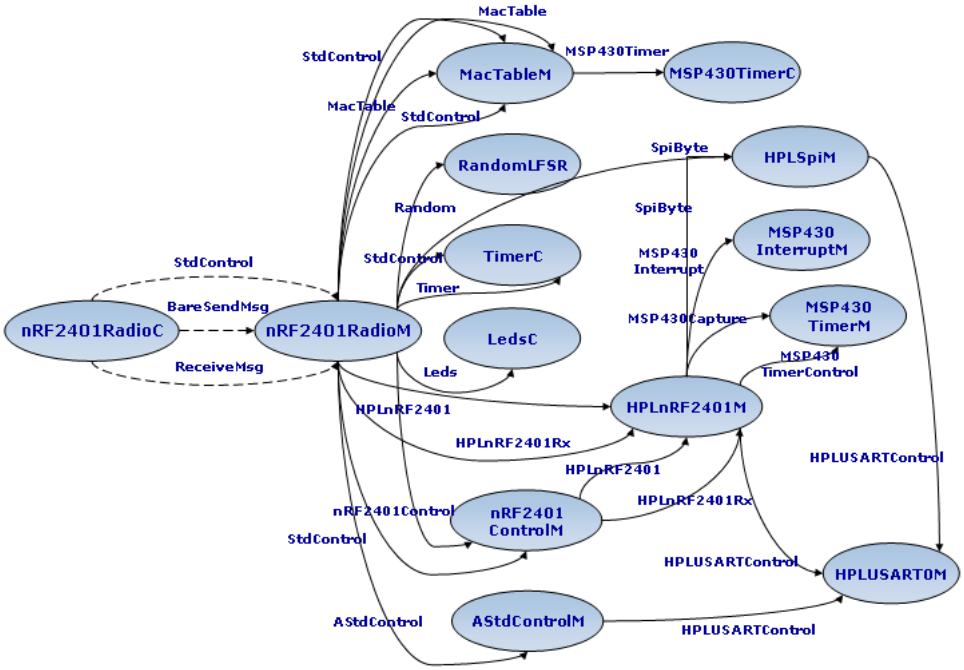

Figure 4.7: Wiring diagram of MAC layer and other Hardware components

The `system` layer contain modules which provide interfaces to send data over any Radio (for any given platform). The below diagram displays the various components present:

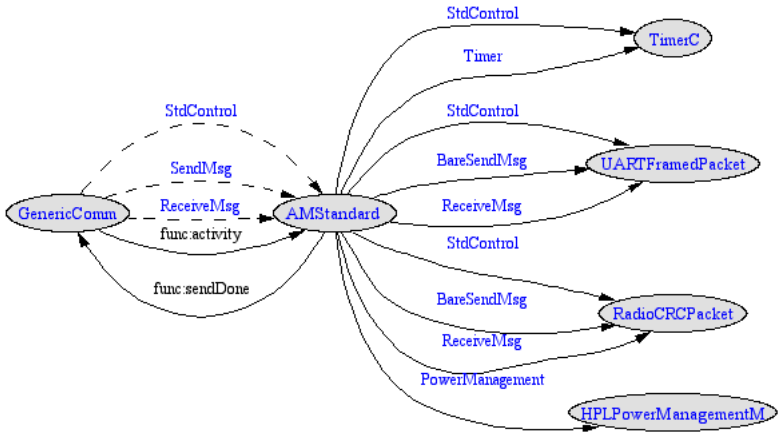

Figure 4.8: Wiring diagram of System components

Interfaces in RadioCRCPacket have to be wired to IMEC platform specific interfaces.

Figure 4.9: Wiring System interfaces to Platform specific implementations

Building Energy efficiency

To summarize, the following implementation specific features have been used to make the MAC Energy efficient.

- No fragmentation, reducing the overhead of fragment header and thereby saving power for transmitting additional unnecessary fragment header bytes.
- No carrier sense and hence saving Power by avoiding the Radio to listen to the channel, until it is free.
- Lower data width for ACK's

4.2 Routing Protocol

As mentioned in the first chapter, time constraints hindered us from devoting more time to the developement of multi-hop routing algorithms. With the MAC protocol in place, we considered it worthwhile to experiment with routing components already present in TinyOS, e.g. *MultiHopRouter*, as another opportunity to test our work.

Surge is an application that demonstrates *ad hoc* multi-hop routing of sensor readings[4]. It is designed to be used in conjunction with a Java GUI that displays the network topology based on the packets received from the sensor network. Each *Surge* node takes sensor readings and forwards them, using in the *MultiHopRouter* component, to the base station, which is always the mote with *node id* 0 (zero). The motes can also respond to broadcast commands (*Bcast* component) sent from the Java control panel through the base station, for example to change sampling rate, make a node go to sleep or wakeup.

Initially, *Surge* could not be compiled to an MSP430 platform due to the lack of the `qsort()` function in mspgcc's standard C library (`libc.a`) used by the *MultiHopRouter* component (`tos/lib/Route`). This issue was addressed providing a quick sort implementation locally.

The following two figures depict an overview of the sequence of actions in the *Surge* application -- from sampling a sensor, sending the packet through the network stack, until it is actually transmitted over the air by the radio chip.

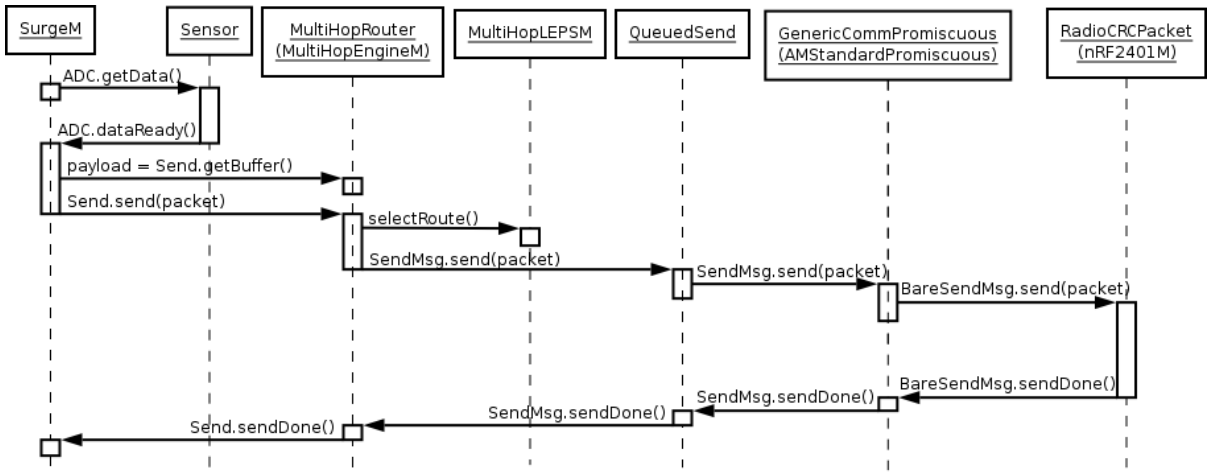

Figure 4.10: Sequence diagram of the *Surge* application sampling a sensor and using *MultiHopRouter* to send the packet (until reaching the nRF2401 radio hardware-specific components).

[4]Just a note of interest: *Surge* was actually the application used in the first large-scale demo of self-organzing wireless sensor network at UC Berkeley (http://today.cs.berkeley.edu/800demo).

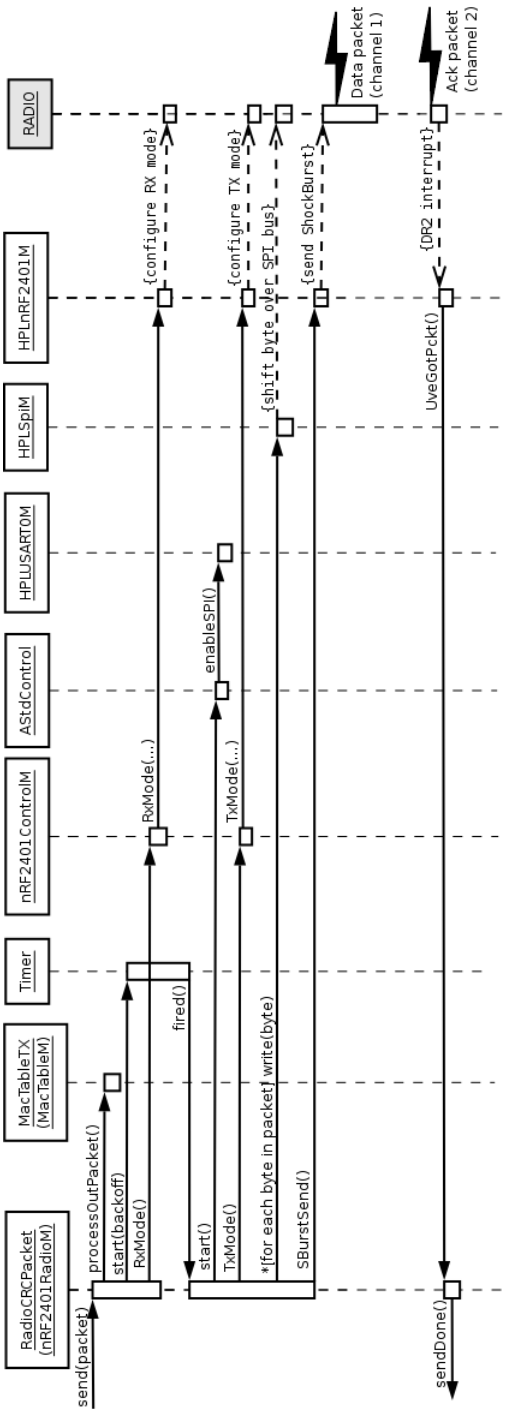

Figure 4.11: Sequence diagram of the nRF2401 radio stack while transmitting a packet. The logic of the MAC protocol is also shown.

4.3 Chapter Summary

The chapter described the Design of MAC layer with specific description on Design constraints and considerations. It presented the necessary State chart and component diagrams, and concluded with a short description on the Routing Protocol. The next chapter provides a detailed description of Testing and Simulation.

Chapter 5

Testing and Simulation

This chapter aims to provide an insight to the Testing and Simulation procedures followed in the course of completion of the project. Initially, validation techniques followed for low level modules that interface hardware (Radio chip, Sensing hardware) are presented, together with their respective results. The chapter continues to provide information regarding the testing of communication in higher levels, acknowledgements and duty cycle operation. Finally, extensive information is provided regarding test cases exercised while the MAC Algorithm was tested. Despite the significant effort put in the testing of the software and encouraging results, that indicate expected behaviour, the Group cannot claim that software artefacts produced in the course of this project are completely defect-free.

5.1 Radio Hardware Interface

Interfacing the Radio chip hardware was an important part of the Implementation phase of the project. During the adaptation of the D-Systems code for interfacing the Radio, numerous changes were performed. The fine-tuning of the code was necessary and mostly was related to the adjustment of time related constraints (correction of existing delays and introduction of new ones) and pin settings refinement, necessary for the correct operation of the hardware. Since radio communication failure would result in a set of serious problems, it was deemed essential to ensure its correct operation.

5.1.1 Radio Transmission

The first step in the process involved the testing of transmission. It was very important to verify that the Cube was actually transmitting information (modulating the carrier) before attempting to receive data. For this purpose we employed a digital storage oscilloscope able of picking up signals at a user-defined frequency. The testing setup involved connecting a piece of wire to one of the scope inputs which functioned as a simple "home made" receiver antenna. The monitoring frequency band of the oscilloscope was set to be from 2.4GHz to 2.5GHz and the Cubes were programmed to continuously transmit

packets of data at 250ms intervals in ShockBurst mode. After successful reception of the signal at the Oscilloscope screen, the original test program was modified so as to explore different situations. The transmitting frequency was varied and transmissions were also performed in both channels of the Nordic Radio, in an attempt to ensure that switching between transmission frequencies and channels behaved as expected. In addition to the above, configuration words were loaded onto the radio that forced the transmission of various packet formats by changing the address, payload, CRC both in terms of contents and size (width). The signal was present at the Oscilloscope screen throughout the attempted transmissions and that result was satisfactory enough to move on.

5.1.2 Radio Reception

The next logical step was to assess the functionality of the Reception part of the Cube. The testing setup involved a Cube programmed to transmit a fixed, user defined packet to a specific address (FFFF), including a predefined Hex value in the payload section. Initially the transmissions were taking place in Channel 1 of the Radio chip. The transmitter was also set to include a 16-bit CRC (calculated and appended by the Radio chip). At the receiving Cube, the Radio was configured to be in ShockBurst mode and have the address of FFFF. Also the configuration word loaded, forced the chip to expect packets with the same payload width as the ones transmitted, and enabled the 16-bit CRC check. Whereas the TX Cube was operated by using a battery, the RX Cube was plugged into the USB programming board and was powered by it. Most importantly, the USB connection also allowed the communication of data from the Sensor to the PC via the UART interface. The code of the RX included debug statements that sent the packet contents over to the PC for each and every received packet. By monitoring the COM port, we were able to capture the data sent to the PC which reflected the received data.

Various transmission combinations were attempted, including:

- Transmission to Channel 1: Data were sent to the first channel frequency (@2.400GHz) and with address set to FFFF.
- Transmission to Channel 2: Data were sent to the second channel frequency (@2.408GHz) and with address set to FFFF. (@2.408GHz)
- Different addresses for Channels 1 and 2: The two channels were set to have distinct addresses (FFFF, FFFA) and the TX generated packets destined accordingly.
- Simultaneous reception from channels 1 and 2: A Cube was transmitting to Channel 1 and a second TX Cube was set to transmit to Channel 2, both with the addresses associated to the RX Channels.
- Different packet formats in two channels.

Blinking of the receiver's led was used to indicate incoming data that were also printed on the console. It is worth noting that the use of a transmitting Cube in order to test the receiver, further exercised the code related to transmission.

5.2 Sensing Hardware Interface

For the purpose of interfacing the underlying sensing hardware provided by the IMEC platform, related libraries available by TinyOS were utilised. Similarities in the ways of driving the hardware assumed by TinyOS and the hardware provided by IMEC led to the conclusion that the latter could be interfaced with minor changes to the provided code. Being a vital part of the sensor module, its sensing capabilities had to be verified and proven to work. In addition to that, it had to be ensured that the manipulation of the available TinyOS libraries during the porting would not introduce defects.

The first sensing module that was subject to testing was the Sensirion temperature/humidity sensor. A simple program was written, the sole purpose of which was to set the sensor to Temperature mode and ask for readings in 1 sec intervals. Again by keeping the Cube on the USB programming board, the collection of the data on the PC was possible by using the UART interface. Data were written to the COM port whenever the sensor interrupted the MCU to return the measurements. The readings that appear below were checked with a room thermometer and found to be consistent.

```
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 27 (C)
Temperature: 28 (C)
```

The simple program used to obtain the temperature measurements was slightly modified in order to obtain humidity data. The Sensirion was set to humidity mode and measurements were requested in the same fashion (every 1 sec). Similarly the data were written to the COM port every time the Sensor issued an interrupt to notify for "fresh" measurements. The output below is what was captured at the COM port of the PC. It should be noted that the Humidity readings were close to an indoor Hygrometer.

```
Humidity: 45 (%)
Humidity: 45 (%)
Humidity: 75 (%)
Humidity: 75 (%)
Humidity: 75 (%)
Humidity: 75 (%)
Humidity: 46 (%)
Humidity: 49 (%)
```

Although the humidity/temperature readings obtained from the Sensirion module were calibrated and there was a clear way of scaling them was described in the Data Sheet, this was not the case for the Light-Dependent Resistor. A different strategy was

followed to verify the correctness of the readings. First of all, readings were obtained from the ADC12 module of the MCU every 1 second. As described above the USB programming board was again the way of obtaining the data. In an attempt to check the measurements captured at the COM port of the PC, the sensor was placed in a small box and the lid was closed gradually. The values obtained are listed below and gradually reflect the transition from a bright environment to complete darkness. It should be noted that appropriate scaling should be performed to these values, if they are to be used within an application.

```
Light-dependent resistor: 0x01e5
Light-dependent resistor: 0x0103
Light-dependent resistor: 0x00f7
Light-dependent resistor: 0x006e
Light-dependent resistor: 0x0023
Light-dependent resistor: 0x0009
Light-dependent resistor: 0x0003
Light-dependent resistor: 0x0001
Light-dependent resistor: 0x0000
```

5.3 Duty Cycle, Radio Communications and Acknowledgements

For the purpose of testing the integration of Acknowledgements to the radio communications, a simple model of a single transmitter and receiver was used. At the Transmitting Cube, the purpose of the experiments was to verify the following:

- Correct pause of the duty cycle and transmission of packet.
- Immediate switch to "Receive" mode for ACK waiting.
- Retransmission of unacknowledged packets.

The associated test cases and their respective results appear below:

ID:Title	DCT1 : Transmission with duty cycle.
Description	1. Start a duty cycle of 50ms sleep - 50ms awake. 2. Transmit a total of three packets @10ms, @70ms and @110ms with destination address set to FFFF. 3. Repeat with other duty cycle combinations and transmission times.
Results	Expected: All three packets should be received at a receiver with its address set to FFFF.

ID:Title	ACK1 : Receive mode switching - ACK waiting.
Description	1. Set TX address of Channel 2 to be FFFF and Channel2 frequency @2.410GHz. Begin in TX mode. 2. Make another node (Helper) to constantly transmit packets at Channel2 frequency and addressed to FFFF. 3. Make the TX node to transmit a packet to any address and immediately switch to Channel 2 and Receive mode. 4. Print received packet and its respective reception time on the console. 5. Verify that the TX receives from Channel 2 for the total of the ACK waiting time.
Results	Expected: The TX node receives packets from Helper node with timestamps from time t (first packet) to time t+ACKWaitingTime.

ID:Title	RTX1 : Packet Retransmission.
Description	1. Configure the TX to retransmit unacknowledged packets for n times, with waiting time between retransmissions set to 1 sec. 2. Set the led to blink prior packet transmission. 3. Transmit a single packet to address FFFA (no receiver). 4. Repeat with various numbers of retransmissions and waiting times in between.
Results	Expected: The led blinks the same number of times as the number of retransmissions.

At the receiving end, the experiments aimed to stress the following functionality:

- Pause of the duty cycle and reception of packet.
- Preparation of ACK packet and switching to "Transmit" mode.
- Transmission of ACK and resuming of duty cycle.

The associated test cases and their respective results appear below:

ID:Title	DCRX1 : Duty Cycle RX.
Description	1. Configure the TX to transmit a total of 5 packets at t = {1,3,5,7,9} sec and with destination address FFFF. 2. Configure the RX to have a duty cycle of 2s sleep period and 2s awake period. Set the RX address to be FFFF - begin asleep. 3. Make the RX to blink the led upon packet reception. 4. Power up both the RX and TX at the same time. 5. Perform tests for both Channels and with more combinations.

Results	Expected: Only packets transmitted at t = 3 and t = 5 should be received, giving 2 blinks at the Receiver.

ID:Title	ACKTX1 : ACK Preparation and TX
Description	1. Enable full functionality of the ACK scheme on both TX andRX. 2. Configure the TX to send a total of 5 packets every 1 sec. Enable waiting for ACK after transmission. Blink the led after receiving a packet from Channel2 (ACK packet). 3. Configure the RX to ACK every received packet.
Results	Expected: The TX should blink a total of 5 times indicating successful reception of all the 5 ACKs.

5.4 MAC Algorithm

The MAC algorithm undoubtedly represents one of the most important outcomes of this project. The effort that was put in by all Group members during the design phase of the MAC, resulted in an implementation that performed well from its early versions.The basic module of the algorithm is the MAC table which as described in earlier chapters, performs all the necessary management required to provide the notion of communication sessions within an Idle-RQ communications protocol frame. Before attempting to implement the MAC, we aimed to first prove the concept behind our design. This was achieved by rapidly developing a model that simulated the behaviour of the Receiver's MAC table in Java.

The Receiver maintained a 5 element data type that was implemented as a Java Monitor and represented the MAC Table. The Monitor approach was used to ensure consistency of data within the table and to simulate the fact that a single packet can arrive successfully at any time. In addition to that, the Monitor ensured the blocking of "Transmitter" threads from actually "sending" their data to the table holder (RX). It must be made clear though that no actual transmission of data was taking place and the only purpose of the simulation was to prove that if cleaning of the table entries occurs fast enough, a large number of Transmitter threads can be served, i.e. not block when placing data in the table. It was proven that a larger number of TX threads (50) could all eventually be served by a table with capacity of 5 entries, provided that entries were cleared often enough. That proof of concept was very encouraging since it has shown that our approach would be capable of receiving from multiple transmitters, by maintaining a data structure that is realistic for the resources of the Cube.

Due to the profound catastrophic impacts that a flawed MAC would have on the radio communications of a Cube, it was seen as the only option to exhaustively test the developed algorithm. The testing that was carried out was split in two phases: On-Target Testing and Simulation Testing. The former mainly involved path-testing of

the algorithm while it was running on the motes, whereas the latter involved testing in situations that could not be achieved with the hardware provided (e.g. 200-node networks).

5.4.1 On-Target Testing

The MAC algorithm was tested on the Cubes in order to verify its functionality and correct behaviour. The testing was done in two discrete steps, at the TX and at the RX. At the transmitter the tests aimed to check the MAC table processing function that is responsible of maintaining the sequence numbers related to ongoing communication sessions with peers. The test cases exercised appear below:

ID:Title	TXNAE1 : Destination ADDR not present, expired entry exists.
Description	1. Manually set all entries in the table to have their ADDR set to FFFF and timestamp 10. Set NRtxTimeout to be 10. Set currentTime to always return 30 to make all entries expired. 2. Program a Cube to be a Receiver with address FFFA that only blinks its led upon reception of a packet. 3. Invoke macTableProcessing for destination address FFFA. 4. Print the return value of macTableProcessing and the table contents on the console. 5. Set timestamps accordingly so only one entry is expired. Repeat with the expired entry in various positions of the table.
Results	Expected: The macTableProcessing function must return EXPIRED_ENTRY. The table contents must remain unchanged except the first expired entry. That entry should hold address FFFA, SeqNum = 0, and timestamp = 30. The Receiver should blink its led.

ID:Title	TXNANE1 : Destination ADDR not present, all entries not expired.
Description	1. Manually set all entries in the table to have their ADDR set to FFFA and timestamp 10. Set NRtxTimeout to be 10. Set currentTime to always return 10 so none of the entries expires. 2. Program a Cube to be a Receiver with address FFFF that only blinks its led upon reception of a packet. 3. Invoke macTableProcessing for destination address FFFF. 4. Print the return value of macTableProcessing and the table contents on the console.
Results	Expected: The macTableProcessing function must return TABLE_FULL. All table entries must remain unchanged. The Receiver should not blink its led.

ID:Title	TXAAEE1 : Entry for destination ADDR exists but has expired.
Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to allways return 20 (all entries expired). 2. Program a Cube to be a Receiver with address FFFF that only blinks its led upon reception of a packet. 3. Invoke macTableProcessing for destination address FFFF. 4. Print the return value of macTableProcessing and the table contents on the console.
Results	Expected: Function must return EXPIRED_ENTRY and all entries must remain intact except the one related to the destination address. That entry should have sequence number set to 0 and its timestamp updated to 20. The Receiver should blink its led.

ID:Title	TXAANE1 : Entry for destination ADDR exists and has not expired.
Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to always return 5 (no entries expired). 2. Program a Cube to be a Receiver with address FFFF that only blinks its led upon reception of a packet. 3. Invoke macTableProcessing for destination address FFFF. 4. Print the return value of macTableProcessing and the table contents on the console.
Results	Expected: The macTableProcessing function must return ENTRY_AVAILABLE. All entries within the table must remain intact except the one holding the destination address. That entry must have sequence number set to 0 and timestamp set to 5.

After performing minor corrections to the TX part of the MAC algorithm, it was decided that the test results were satisfactory enough to move on to the RX testing. The aim of the tests was mainly to ensure that correct identification of incoming packets occurs at various situations. The related test cases appear below:

ID:Title	RXNANE1 : Source ADDR not present, all entries not expired.

Description	1. Manually set all entries in the table to have their ADDR set to FFFF and timestamp 10. Set NRtxTimeout to be 10. Set currentTime to always return 10 so none of the entries expires. 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return TABLE_FULL. All table entries must remain unchanged. Packet should be dropped and not acknowledged (RX should not blink).

ID:Title	RXNAEE1 : Source ADDR not present, all or some entries expired.
Description	1. Manually set all entries in the RX table to have ADDR = FFFF. Set all or some of the timestamps to be 0. Set NRtxTimeout to 5. Set currentTime to always return 10 so entries with timestamp less than 10 will appear as expired. 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return EXPIRED_ENTRY. All table entries must remain unchanged except the first expired entry that must now hold address FFFA, SeqNum = 1 and timestamp = 10. The packet should be acknowledged (RX led should blink). Actual: Pass.

ID:Title	RXAASN1 : Entry for source ADDR exists, sequence number in packet different to the expected one.

Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to allways return 5 (no entries expired). 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB with SeqNum = 0. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return SEQNO_NOT_FOUND. All table entries must remain unchanged except the entry related to address FFFA that should have SeqNum = 1 and timestamp = 5. The packet should be acknowledged as it is a duplicate (RX led should blink).

ID:Title	RXAESN1 : Entry for source ADDR exists, sequence number in packet different to the expected one.
Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to allways return 5 (no entries expired). 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB with SeqNum = 0. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return SEQNO_NOT_FOUND. All table entries must remain unchanged except the entry related to address FFFA that should have SeqNum = 1 and timestamp = 5. The packet should be acknowledged (RX led should blink).

ID:Title	RXAESN2 : Entry for source ADDR exists and has not expired. Duplicate packet arrives (same SeqNum)

Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to allways return 5 (no entries expired). 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB with SeqNum = 1. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return DUPLICATE_NOTEXPIRED. All table entries must remain unchanged. The packet should be acknowledged as it is a duplicate (RX led should blink).

ID:Title	RXAESN3 : Entry for source ADDR exists but has expired. A packet with SeqNum equal to the one in the table entry arrives.
Description	1. Fill the table with entries for addresses FFFF, FFFA, FFFB, FFFC, FFFD. Set the timestamps for all entries to be 0 and their SeqNum to be 1. Set the NRtxTimeout to be 10 and currentTime to allways return 20 (all entries expired). 2. Program a Cube to be a Transmitter with address FFFA that transmits a single packet to FFFB with SeqNum = 1. Configure the RX to have FFFB as its own address and to blink its led when transmitting an ACK. 3. Power up the RX and then the TX. Print the return value of macTableProcessing of the RX as well as its Table contents on the console.
Results	Expected: The macTableProcessing function must return NOTDUPLICATE_EXPIRED. All table entries must remain unchanged except the entry related to address FFFA that should have SeqNum = 1 and timestamp = 20. The packet should be acknowledged (RX led should blink).

5.4.2 Simulation Testing

In order to explore the behaviour of the developed algorithm in a large scale network setup, it was decided to incorporate its functionality in the simulator of TinyOS, TOSSIM. This did not only allow the testing of the MAC with a large number of nodes, but also provided a version of the simulator that is suitable for the simulation and analy-

sis of applications designed for the IMEC platform. After careful analysis of the various modules involved in the radio subsystem of the simulator, we have identified as the best place to introduce our MAC scheme to be directly below RadioCRCPacket and above the MicaHighSpeedRadioM. Despite the necessary wirings between existing and newly introduced modules, it was also necessary to implement the, expected by the former, interfaces.

The application used to stress the MAC table was TestTinyViz, an application that randomly initiates packet transmissions from nodes to their neighbours. The application was configured to simulate networks of around 20 nodes and by including output statements in various places within the code, we managed to monitor the behaviour of the MAC Table. Most important of all, the simulations ran proved once again that it is possible to communicate with more than 5 peers by maintaining a table with only 5 positions, provided that the latter allows reuse of entries that are obsolete. In the console output appended below, nodes can be seen that are sending packets and receive acknowledgements and nodes that discard entries no longer needed so as to facilitate communication with others.

```
...
Node 9:  In MacProtocolM RECEIVE :: Recvd from Node : [7]
Node 5: In SendtoSPI() Sending from [5] to [1].
Node 5: TIME Before : [67543062]
Node 4:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 7:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 3:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 6:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 0:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 1:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 1:  RX Entry Expired : Node [1] Rcvd a pkt from node [5]
Node 1:  Signalled packet receive
Node 1:  Sending ACK frame:
Node 2:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 9:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 8:  In MacProtocolM RECEIVE :: Recvd from Node : [5]
Node 9:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 4:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 5:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 5:  ACK Received  ::: N_RTX Value : [1]
Node 3:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 6:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 8:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 2:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 7:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 0:  In MacProtocolM RECEIVE :: Recvd from Node : [1]
Node 0:   Timeout before get-
```

```
ting ACK:  N_RTX Value:  ((((1))))
Node 0:   N_RTX Value:  [[[[1]]]
Node 0: In SendtoSPI() Sending from [0] to [4].
Node 0: TIME Before : [70721418]
...
```

5.5 Chapter Summary

Within this chapter, after presenting the testing and validation techniques used for low level modules that interface hardware (Radio chip, Sensing hardware), the test cases used for examining the functionality of communication in higher levels, acknowledgements and duty cycle operation have been presented. Extensive information was also provided regarding test cases exercised while the MAC Algorithm was tested, and also, regarding the use of TOSSIM for testing purposes. The following chapter seeks to evaluate the project outcomes and draw conclusions regarding their usability.

Chapter 6

Evaluation and Critical Analysis

In order to measure the value and usability of the project outcomes and the work performed, a series of trials have been performed. In this chapter we seek to evaluate the behaviour of the communications in a controlled environment so as to draw conclusions regarding the effects that the Duty Cycle, Retransmissions and MAC Algorithm introduce to the scenario. The metrics that were considered in the process are mainly packet delivery ratio and percentage of sleep time for the nodes. The evaluation procedures that will shortly be presented aimed to provide an understanding and draw conclusions on how different duty cycle lengths and ratios, acknowledgements and retransmissions can favour packet delivery ratio. Also an important discovery is the fact that all the above can be fine tuned in order to serve specific application needs more efficiently.

6.1 Experiments Set Up and Equipment

The Experiments were conducted in the 8th floor corridor of the New Engineering Building of UCL, Malet Place. Since this building is Wireless LAN enabled, there was a significant amount of interference present, as both the Cubes and 802.11 utilise the 2.4GHz band. More precisely, at the corridor that the experiments took place, two wireless networks were present with signal power that varied from 1/5 to 3/5 bars indicated in a laptop's wireless card driver. It was chosen not to include any physical obstacles in between the communicating nodes, as the latter were found to communicate with great difficulties even when a single wall was in between.

Throughout the experiments two Cubes were used that were powered from constant voltage sources in order to eliminate any effects that inadequate power supply would introduce. The first sensor was powered by a laboratory power supply, set to output 2.7 volts, whereas, the second was powered directly from the USB programming board that also provided 2.7 volts. The sensors were placed in a "line of sight" fashion and at distances of 1, 6 and 12 meters that were accurately measured with a measurement tape.

Since the aim was to explore the impact of ACKs, retransmissions, physical proximity and duty cycle lengths and ratios to packet delivery ratio, an application was developed

that enabled us to vary these factors at will. The application that provided the testbed for the evaluation is comprised by two distinct parts: One for the Transmitter and one for the Receiver.

The part of the application destined for the transmitting node was coded to transmit a total of 100 packets and can be configured to enable acknowledgements and retransmissions. The ACK-RTX scheme, when enabled, transmits a packet and waits for a total of 4ms for ACK reception. If an ACK is not received within that window of opportunity, the transmitter proceeds to retransmit the packet. In the case that the number of 8 retransmissions has been reached, the transmitter moves on to transmit the next packet. With the above state of affairs, a TX could potentially transmit 800 packets if an ACK is never sent for any packet. When the acknowledgements are disabled at the transmitting end, packets are sent one after the other and the number of transmitted packets is always 100. An important aspect of the application that initiates packet transmissions, is the fact that packets are "fired" every 175ms, which represents a firing rate relatively high for sensor network applications (a typical value would be 1pkt/sec). It is important to note that the measurements obtained, and consequently, the conclusions inferred from them are **strictly related** to this model application firing rate of 1pkt/175ms or otherwise 5.71pkts/sec. As it will be explained in the rest of this chapter, the above restriction is due to the fact that some duty cycles are found to match better certain packet generation rates and vice versa.

As far as the receiving part of the application is concerned, that was again implemented with ease of functionality change and minimal interference to its operation in mind, It has been identified that the overhead of writing data to the UART for them to be recorded at the laptop used to collect them, was significant and could interfere with the operation of the protocol. Therefore, it was decided to minimise communications to and from the laptop by maintaining the received packets count in a dedicated variable at the RX Cube's memory, the value of which was not transmitted until the end of the end of the 100 packets transmission cycle. Since the number of packets received was not always 100, a timer was used to indicate a timeout that initiated the transmission of the value to the PC. The employment of the timer is regarded as an additional but necessary overhead that does interfere with the execution of the protocol but is considered insignificant enough to be accepted as a solution.

The experiments conducted, involved both the transmission and reception parts of the developed application and a variety of combinations of the monitored and controlled factors-variables. This enabled collection of data used for the purposes of reasoning and drawing of conclusions. The next sections describe the experiments performed in detail and the corresponding conclusions drawn.

6.2 Power Consumption Measurements

The figures below illustrate power consumption data captured with a digital storage oscilloscope while connected to the receiver and transmitter nodes, respectively.

The first graph plots current (in Amps) against time (in seconds) in the receiver node

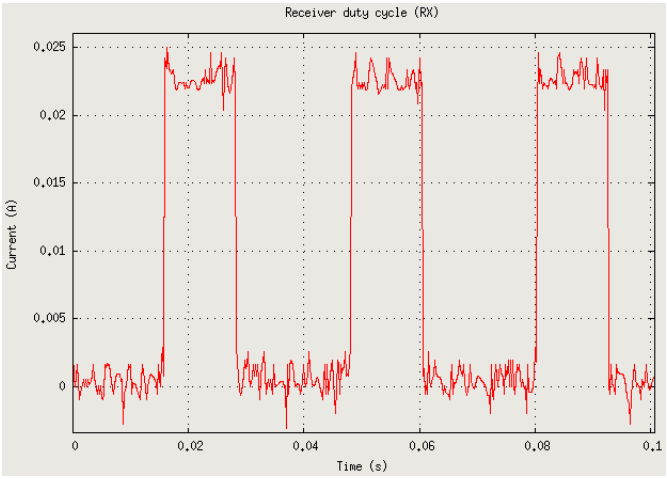

Figure 6.1: Power consumption in receiver's duty cycle: 12ms active, 20ms sleep.

whilst in normal duty cycle. It allows us to contemplate the striking differences in energy consumption of radio enabled vs. disabled: an average of 23 mA with the radio chip active and close to 0 when the radio is off. It should also be noted that the transition time between radio on/off is in the order of 200 us.

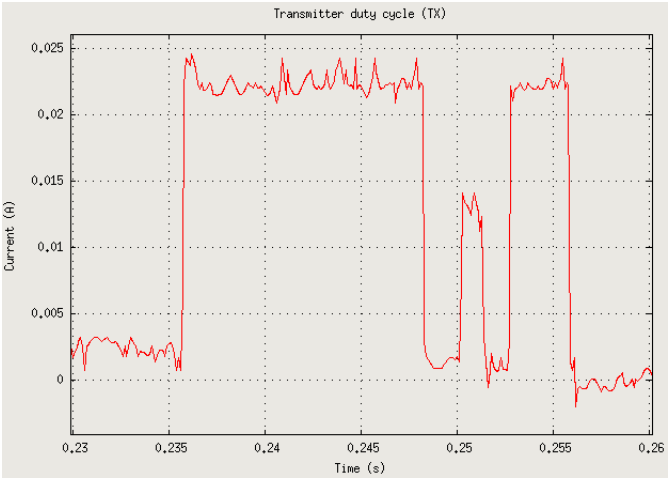

Figure 6.2: Power consumption in transmitter's duty cycle.

Zooming in on the transmitter node's operation, we observe the difference between the level of energy spent while listening (in receive mode) vs. sending data (in transmit mode). Three main features (spikes in current) are displayed:

- the first, a 12 ms period relates to energy spent with radio in receive mode during the node's normal duty cycle;

- the second, 1 ms spike refers to the node switching to transmit mode in order to send a packet;
- finally, the third spike pertains to the switchover to receive mode and wait for an ACK packet (which was never received in this case).

6.3 Experiments Analysis and Description

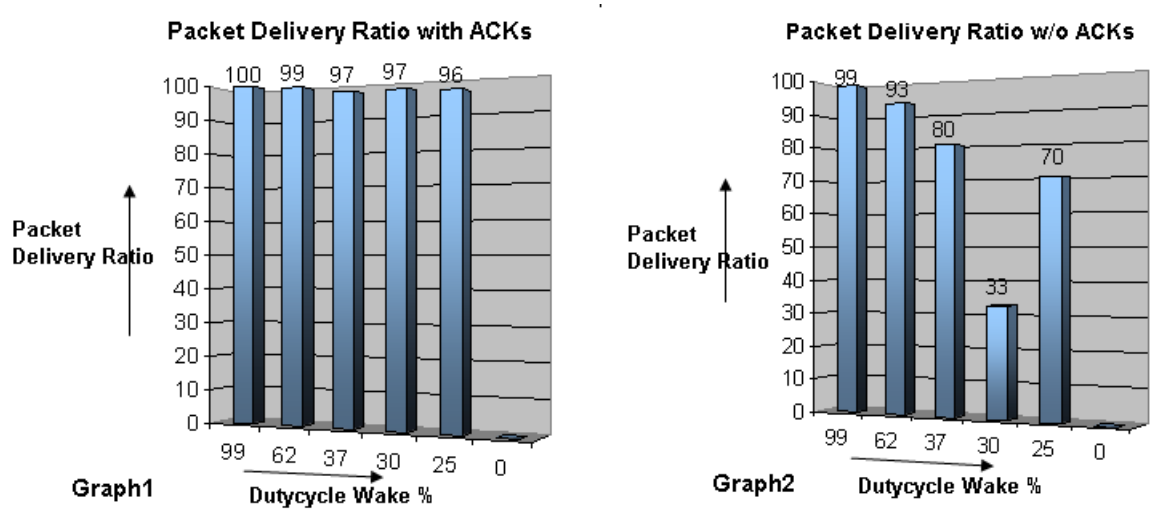

Figure 6.3: Packet Delivery Ratio with and without ACKs

With the above mentioned setup, two sets of experiments were conducted. In the first set of experiments, 100 Shockburst packets were transmitted, with different Dutycycle periods at the receiver. For each dutycycle value, the experiement was repeated three times and the average Packet Delivery Ratio was recorded. The transmitter and receiver were placed approximately 6 meters apart, and acknowledgements (and retransmissions) were enabled at both transmitter and receiver nodes. Receiver was configured with RfmToLeds application. Packet Delivery Ratio for 5 different values of dutycycle (at receiver) have been recorded and plotted in Graph1. It is clearly visible that inspite of very low wake time in dutycycle (25% wake time, 75% sleep time), Packet Delivery Ratio is 96%. So the MAC is reliable, inspite of saving 75% of energy (radio power). Remember that the MAC does not do any carrier sensing. The main reason for having a high Packet Delivery Ratio (inspite of low wake time) is because of acknowledgements. It can be argued that acknowledgements and retransmissions involve some amount of radio power consumption, but this is very less compared to amount of power saved while sleeping (for 75% of the time). Also note that acknowledgement packets are very small in length compared to the normal data packets.

A second set of experiments were conducted, with exactly same setup as the first set of experiments, except that acknowledgements and retransmissions were disabled.

The experiment was repeated for the same values of Dutycycles at the receiver. It can be observed from the graph that, with a 62% dutycycle wake time, the Packet Delivery Ratio seems to be reasonable (93% in this case), but if we try to sleep more (decrease the wake time), the Packet Delivery Ratio seems to deteriorate. This is because when the transmitter sends data over the radio to the receiver, and if the receiver is sleeping, the packet will be lost. In the first set of experiments, transmitter retransmits the packets with a delay of 4ms between each retransmission and in the meantime, if the receiver wakes up it picks up the data. So the bottom-line is, to achieve a Packet Delivery Ratio of atleast 90%, the receiver should have a dutycycle of no less than 60% wake time (40% sleep time). This is definitely not power-efficient compared to the case where acknowledgements are enabled. Therefore use of acknowledgements with a right dutycyle logic can make the MAC reliable and power-efficient.

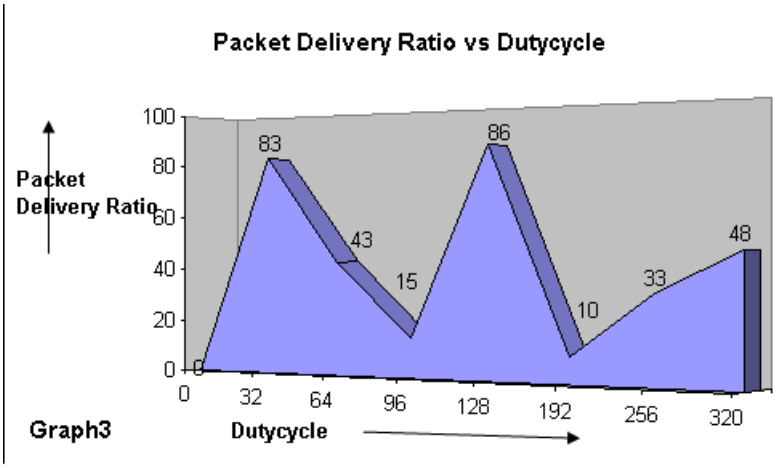

Figure 6.4: Packet Delivery Ratio with different duty cycle values (having the same (sleep time)/(wake time) ratio)

A third set of experiments were conducted again to observe the packet Delivery Ratio. The experiments are conducted by choosing different dutycycle values at the receiver such that all the values have a ratio of 37.5% wake time and 62.5% sleep time. The recorded values are plotted in Graph 3. The values shown in x-axis are the full dutycycle values (including both sleep and wake times). An interesting observation is inferred from the graph which reveals a varying Packet Delivery Ratio as the dutycycle value increases. This is again very specific to the above mentioned setup and the result values will vary with a different setup. Even though the ratio of wake time and sleep time is same for all the selected dutycycle values, only a certain set of dutycycle values will produce a high Packet Delivery ratio.

6.4 Critical Assessment and Conclusion

As mentioned in the Design and Implementation chapter, there are different MACís available for Wireless sensor networks. Each one of these is better suited according to the application practical scenario and needs. In our setup we assume a simple tree based routing (Surge application), wherein nodes in the network forward data to a root node (normally connected to a PC). With the above scenario in mind we had to choose a right MAC and tailor it by making it both reliable and energy efficient as much as possible. Features including (fragmentation and carrier sensing) have been avoided saving some amount of power.

To conclude, a simple ALOHA based MAC protocol (with a proper dutycycle management at the receiver and with acknowledgements and retransmission enabled), can be made both reliable and Energy efficient for routing in Wireless sensor networks.

Chapter 7

Project Management

In this chapter we highlight some of the Project management techniques used. Right from the planning phase through to the deployment phase, project management techniques and activities were used. In the planning phase the project was broken down into multiple phases, where each phase spanned multiple iterations, with each iteration going through the entire Project life cycle (popularly known as Agile Development)

The first section describes about the team composition, highlighting some of the major tasks and responsibilities of team members. In the next section we describe the Process approach and technique. This is followed by a chart that displays the overall schedule of the project and the milestones achieved. The further sections in the chapter focus on tools, risk management and team communication.

7.1 Team Composition

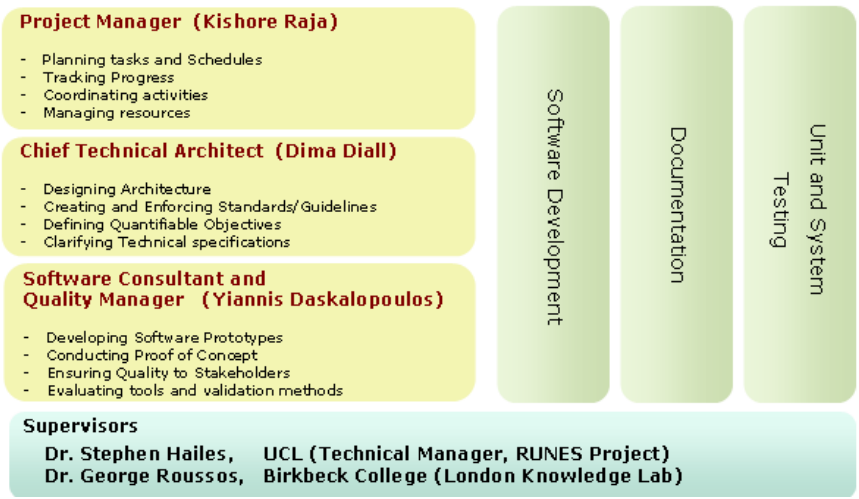

Figure 7.1: Team Composition

The above figure highlights the team composition with roles and responsibilities. A flat team structure was maintained throughout the project and all members were involved in project management activities. Some of the specific tasks are described as follows:

- **Kishore Raja** had the role of Project Manager and was involved in kick-start of the project (initiating the Porting), planning tasks and activities, assigning work to team members and tracking the overall progress.
- **Dima Diall** had the role of Technical Architect and was involved in setting the Content management software, installing Version Control system (Subversion) and documentation proposal.
- **Ioannis Daskalopoulos** had the role of Software Consultant and was involved in Risk Management, ensuring quality of the project. He was also involved in setting up an appropriate environment for maintenance of hardware sensor cubes and configuring Oscilloscope for capturing radio transmission.

Initially a team of 4 people was formed. Before starting the project we lost a team member, as he discontinued the course.

Apart from the specific tasks, all team members were collectively involved in:

- Porting of TinyOS, Design, Implementation and testing.
- Writing various Project documents
- Unit Testing, Integration and overall System Testing

7.2 Approach and Technique

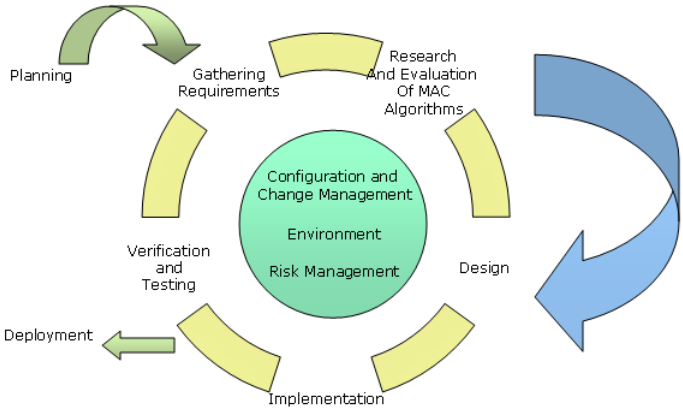

Figure 7.2: Project process overview

As per the guidelines, laid out in the initial Project presentation, Agile Process development with Extreme programming principles and practices were followed. The rationale for choosing Agile development is described below:

- Research oriented project

  During the initial stages, the goals of the project were not clear and hence were able to do a better estimate as the project progressed. Right from the initial porting phase to the design of MAC, we had to address many uncertainties and hurdles and hence an Agile process was well suited.

- Unclear and changing requirements

  This again holds good for each phase, as the requirements had to be refined.

- Iterative and Incremental Development

- Small team

After the initial Porting phase a 3-week iteration was started with a plan to add new functionality and maybe fix bugs in the existing functionality, during each iteration. However the iterations could not be strictly maintained, even though the time spent for some activities (Radio communication, design of MAC layer) lasted for more than 3-weeks. Due to some uncertainties that were encountered in the project, the focus was diverted towards completing the task, rather than in tracking iterations.

Pair programming was used for all phases in the project. Most of the code that is included in the final release is created by two people working together at a single computer. Since there were odd number of people in the project, one person had to effectively switch between teams to create pairs.

7.3 Schedule and Milestones

7.3.1 Gantt chart

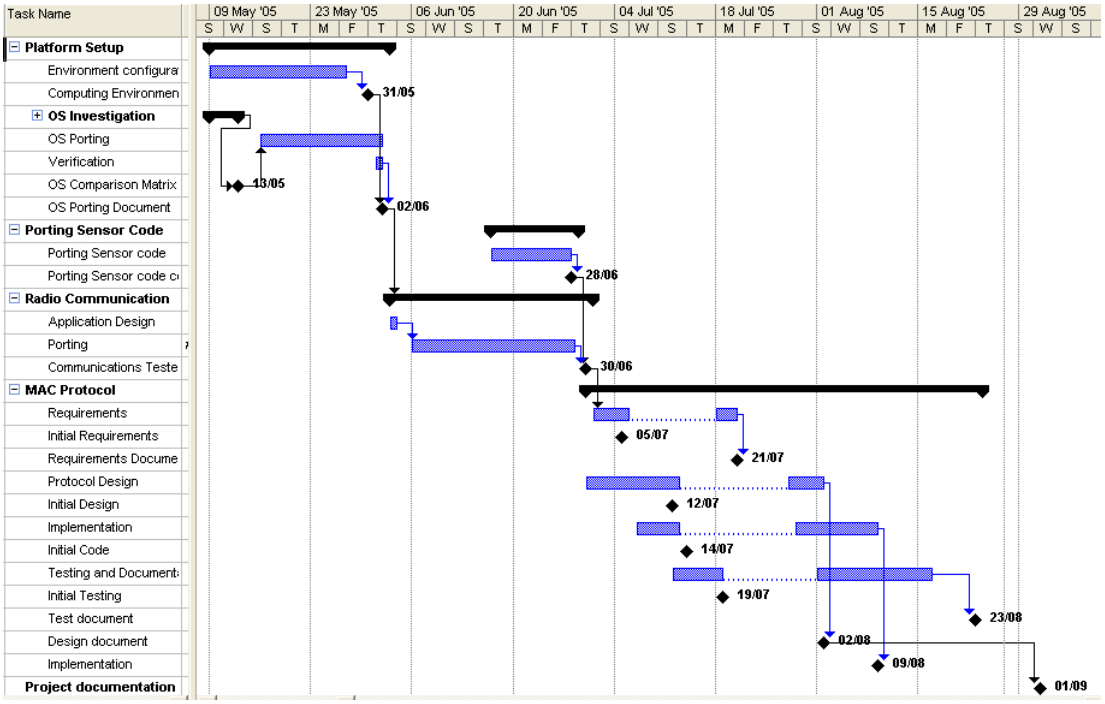

Figure 7.3: Project schedule

The project is mainly divided into four phases:

- OS Investigation and Porting
- Establishing Radio Communication
- Porting Sensor code
- Design and implementation of MAC and Routing Protocol

The Gantt chart displays the overall schedule of the project and highlights the milestones achieved (more of which in the next section). The estimated schedule (produced during the planning phase) has been modified to display the actual schedule of the project. Most of the phases remain unchanged, except that the time for establishing Radio communication took much longer than estimated. The Porting phase had two parallel activities (porting Radio code and porting Sensor code). After the porting phase, Agile process with 3-week iterations were started (as estimated), though the duration of iteration could not strictly be maintained.

7.3.2 Milestones

No.	Task Name	Completion Date	
1.	OS Comparison Matrix	13/05/05	MAY
2.	Environment configured	31/05/05	
3.	Porting TinyOS (Blink application)	02/06/05	JUNE
4.	Porting Sensor code	28/06/05	
5.	Establishing Radio Communication	30/06/05	
6.	Initial Requirements	05/07/06	JULY
7.	Initial Design	12/07/05	
8.	Initial Code	14/07/05	
9.	Initial Testing	19/07/05	
10.	Final Requirements document	21/07/05	
11.	Final Design document	02/08/05	AUGUST
12.	Final code	09/08/05	
13.	Final Test Document	23/08/05	
14.	Project related documents (Group report, Individual report, User guide, Wiring diagrams, Executive summary)	01/09/05	

Figure 7.4: Milestones

The table above displays the various milestones in the project. The first five milestones (in light grey background) are part of the initial Porting process including establishing Radio communication. Agile process was not used for these milestones. For the later set of milestones (No.6 till No.14) agile programming practice with 3-week iterations were used.

In May 2005, a Presentation of the project plan was given, which highlighted both Technical and Project management policies. The project was then kick-started with the analysis phase, which mostly involved in reading various Operating systems, and research papers related to Wireless Sensor Networks. After choosing a right Operating system, porting onto IMEC sensors was initiated. In the meantime, Environment configuration was done in parallel to setup the various tools (like Content management system, Version control system etc.)

The first version of TinyOS (system independent kernel) was successfully ported onto the IMEC platform in first week of June 2005. This also involved setting up the compilation environment for the IMEC platform. The next vital task was to establish Radio communication, which was a major bottleneck in the project. This was completed on 30th June 2005. In the meantime porting of MSP430 sensor code onto IMEC platform was carried on in parallel and was completed on 28th June. Both the above tasks were divided among team members.

July 2005 mainly involved brainstorming the requirements and design of MAC layer. 3-week iterations were started and, in each iteration, all phases of project life cycle were carried on. During each iteration, the requirements were refined and hence the design and implementation. Both requirements and design specific features were documented in the project website and the contents were consolidated into a formal document during the final phase of the project.

In August 2005 the MAC layer was ported to perform large scale simulation with TOSSIM. Also a routing protocol (from TinyOS distrution) was run on IMEC sensor hardware. Most of the Requirements and Design related issues were frozen in the first week. Write-up of Project submission documents (including Group report, Individual report, User guide, wiring diagrams and executive summary) was started during the second week.

7.4 Support Tools

- Web Content Management system (Trac)

  In order to manage the content (documents, meeting minutes, tasks, etc..) a Content Management Software (Trac) was set up on internal Unix machines. Team members initially investigated on various Content Management software and Dima came up with a proposal of using Trac. Investigation, installation and setup was done during the Initial Platform setup phase. Apart from the usual User space, an additional project space (of around 300MB) was used for this purpose.

  Trac is a Web Content management system with built-in wiki support enabling features like documenting, collaborative editing, feedback and discussion notes. Wiki uses a simple syntax based Mark-up language rather than HTML, which makes editing easier and encourages team members to annotate and contribute to text content. All project related documents including Requirements, Design, Code, test cases and meeting minutes were maintained in Trac.

- Software Configuration Management (Subversion)

  Software Configuration Management is a set of activities that are designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions, etc.. Among these categories, Version Control system is important as it controls the file changes and remembers what they changed and when. Advanced branching and private branches facilitate concurrent development of versions.

  Subversion is a free open source Version control system, managing files and directories over time. A tree of files is placed in a Central repository (much

like an ordinary server, except that it remembers every change made to files and directories). Subversion allows access to its repository across networks, which enabled team members to manage code and content outside university networks.

7.5 Risk Management

A set of Risks have been identified during the initial phase of the project. These risks were documented in a table and tracked as the project progressed.

Name	Probability	Impact	Strategy/Action	Owner	Exposure
Hardware Failure	Moderate	Catastrophic	Use grounding bands when interfering and handle with care. In case of failure, notify supervisors and IMEC for a possible replacement.	Yiannis	
Limited Resources	Moderate	Catastrophic	Predict and assess the size of the system regularly. Remove unnecessary features of the OS – Protocol Stack. Try to drive hardware directly where possible. Code efficiently and use optimization wherever possible.	Kishore	
Power Consumption	Moderate	Catastrophic	Monitor power consumption throughout the development process and make extensive use of power save modes and on-chip utilities available.	Yiannis	
Porting Issues	High	Serious	Study Device Data sheets and join relative mailing lists in early phases of the project. Seek help on newsgroups and from the Supervisors. Alternatively we could build the Protocol on top of existing libraries	Dima	
Staff Unavailability	Low	Serious	Make sure all team members are aware of each others work by overlapping their activities. Ultimately other members should be able to continue the work of an absent one. Plan the schedule with the supervisors	Kishore	
Bugs in Libraries	Moderate	Tolerable	Gain an understanding of the libraries available and become familiar with the code in the beginning of the project. Modify the libraries to correct defects as they are observed.	Dima	
Changing Requirements	Moderate	Tolerable	Use information hiding in the design if possible. Discuss regularly the requirements with the supervisors.	Yiannis	
Time Management Issues	Moderate	Tolerable	Investigate the possibility of reusing components instead of developing them. Reallocate the available resources (Developers). Give importance to high priority objectives.	Kishore	
Development Environment	Low	Tolerable	Seek material in related Newsgroups and ensure all team members are familiar with the environment from the early stages. Ask supervisors for any non-straightforward limitations of the platform.	Dima	

Figure 7.5: Risks

Risks are usually listed in the order of impact. During weekly meeting, the Risks were reviewed and the status of each risk was updated (indicating the progress, if any). Also action points were assigned to monitor/overcome the risk. Each Risk was also associated with a owner who was primarily responsible for tracking it.

Although tracking and updating of Risks was actively performed during the initial phase of the project, less importance was given at later stages. Along with other Project management activities, Risks were not regularly tracked, as we were completely masked by Technical activities. With a small team size, it became difficult to perform too many project management activities and the focus was more to achieve the project goals. However most of the High impact risks were actively tracked and preventive measures were taken to avoid them. One example was the risk on Hardware Failure. Since we

had only three sensor cubes, it was very important for us to keep the cubes in a healthy working condition. The pin connectors for batteries used for charging these cubes were very sensitive and hence we had to build additional extending wires and connectors for flexible operation.

7.6 Team Communication

The team had an effective communication plan and followed it well throughout the project. Various meetings were planned during the project planning phase:

- Internal Meetings
- Supervisor meetings
- Ad-hoc meetings

Regular weekly meetings were conducted (usually on Monday or Tuesday) and the minutes were recorded and stored in project website. Meeting with Supervisors were also conducted regularly. Ad-hoc meetings were conducted as and when needed. During the design phase of the project, several adhoc meetings were held and hence the regular weekly meetings were skipped. Also by the end of the project (during August), meeting minutes were not recorded (though more ad-hoc meetings were held)

7.6.1 Example meeting minutes

Date	23/05/2005 (Monday)
Time	10:30
Location	8.06
Contributors	Dima, Kishore, Yiannis
Next meeting	TBD

Discussion

- The group mainly discussed the boot-up process of TinyOS and the structure and contents of several files regarding the pinout settings in the TELOS platform (.platform, hardware.h).
- Different approaches from IMEC and TinyOS for setting the pinouts have been identified.
- It was agreed that the actual hardware connections have to be examined in the schematics as well as the way that IMEC sets the pinouts.
- It has been agreed between the members, that each one should read one paper

regarding application scenarios of sensor networks, per week. Papers have also been assigned.

- Finally the risks have been re-assessed: no change in status.

Action Points

- **All**:
  - Continue working on Porting TinyOS.
  - Background reading of papers (on free time).
- **Kishore**:
  - Continue working on boot strapping and various hardware initialisation.
  - Understand the existing make system, to create one for IMEC platform.
  - Read the paper on "VigilNet".
- **Dima**:
  - Examine the Ports available on the MSP430 and how they connect to various components (ADC, Sensor modules, etc).
  - Read the "Survey of Energy Aware Routing Protocols on Sensor Networks".
- **Ioannis**:
  - Examine the Nordic RF datasheet thoroughly and look at existing files to discover which ports of the MSP430 are used with the RF module.
  - Examine schematics of IMEC to document actual hardware interconnections.
  - Read the paper on "SMECN".

7.7 Chapter Summary

The chapter started with a description of the team structure followed by a description of the various Project management activities employed including Approach and technique, schedule and milestones and support tools. It then described the Risk management technique, followed by a section on communication paradigm used. The next chapter draws the final conclusions followed by future work.

Chapter 8

Conclusions and Future Work

This chapter aims to provide an overview of the design and implementation process, as well as to briefly summarise the project's achievements. Possible ideas for improvements and future work are also presented.

8.1 Problems and Solutions

We all started out as absolute novices with respect to wireless sensor networks and everything was new: the hardware components, the software paradigms, etc. As the project unfolded we ran into many deadends, at every turn we discovered new or more efficient ways of doing things, forcing us to backtrack and repeat the design-implement-test cycle multiple times. The learning process was costly (but gratifying) and hampered our progress somewhat and we also went through a number of goals and scope redefinitions.

The biggest obstacle we faced in this project probably is the fact that the only things *tiny* about the motes and TinyOS are the hardware resources and part of operating system's name, respectively. The hardware is small and simple, but in the beginning it seemed amazingly complex.

Regarding TinyOS, as shown in the table below, its code base boasts more than 160,000 physical lines of code in different programming languages (nesC, C, C++, Java, plus a host of scripting ones); third-party software contribute to a threefold increase in the "pile of code" and add a few more languages to the "melting pot".

Table 8.1: TinyOS code size in CVS snapshot 1.1.13 (data obtained using David A. Wheeler's 'SLOCCount').

TinyOS source code size	**Physical lines of code**	
Programming languages	**total**	**percent**
Java	82,710	49.80%
nesC	55,593	33.47%
C	15,412	9.28%
C++	5,666	3.41%
Others (Python, Perl, Shell)	6,710	4.04%
Total	**166,091**	**100.00%**
Contributed code in CVS (Aug 2005)	528,925	318.46%

TinyOS was initially developed around the UC Berkeley motes -- the Mica family with Atmel AVR micro-controllers -- and this heritage looms at the very core of version 1.x code base as well as throughout the documentation. TinyOS, in every of its aspects (systems design, programming models, networking, application classes, etc) was and still is fertile ground for many PhD degrees in the last few years. To a certain degree, this equates to research-quality software with:

- experimental interfaces with unfinished implementations;
- deprecated interfaces, commands and other types of cruft;
- a number of misleading comments, some plain wrong;
- inexistent or incomplete documentation, or out of sync with code;
- reams of variations to interfaces, modules (...) in contributed code[5];

Much of this is bound to change in version TinyOS 2.0, designed from the ground up with portability in mind (micro-controllers, radio chips, etc). The features and infrastructure of the future incarnation of the operating system is being steered by a formal Working Group[6], composed of veteran core developers, through the so-called TinyOS Extension Protocols. This group is also establishing the policy to be used by the community for introducing new functionality to the system.

A sample of the challenges we came across and managed to overcome during the project include the following (among others):

nRF2401's ShockBurst mode: In this mode the radio provides a simple packet interface to the application programmer; the chip generates/detects the preamble, automatically matches the destination address as well as CRC computation and checking. This also offloads most of the time-critical processing requirements from

[5]Our project's code, too, will eventually add to this.
[6]TinyOS 2.0 Working Group: http://www.tinyos.net/scoop/special/working_group_tinyos_2-0

the MCU, as the data may be clocked into the radio chip at a bit rate much slower than the one used for actual on-air transmission.

The particular issue turned out as a hindrance regarding MAC protocol options we had available, namely the possibility of adopting CSMA-based, some of which already had implementations under TinyOS (e.g. B-MAC, S-MAC, or T-MAC). Thus, we adopted an approach based on an ALOHA-type algorithm.

Interrupt for packet on radio channel 2: These particular IMEC prototypes were designed to operate the nRF2401 radio in ShockBurst mode and only in channel 1 -- the DR2 interrupt from the radio chip is connected to pin P4.4 on the MSP430, which is not interrupt-capable. This proved to be a limitation for us, as we intended to use both channels.

As a lucky coincidence that pin is also used by Timer_B, so it was possible to workaround this problem by using the timer register TB4 in capture mode and, hence, trigger an interrupt on the arrival of packets from channel 2.

MSP430 word-alignment issues: As a 16-bit MCU, the MSP430 is optimized to access word-sized data at even memory addresses; the result of accessing 16-bit quantities at odd memory addresses is unspecified [MSP430U]. In the initial design phases we overlooked this important detail and had a packet format for the MAC protocol a 7-byte header.

As mspgcc, under normal circumstances, allocates data structures at even addresses this meant that the payload would start at an odd address. If a layer above the MAC happens to expect word-sized data at the start of its buffer (which is the MAC's payload field) problems will occur. We adopted the simplest solution possible: increase the header size to 8 bytes, an even number. This is consistent with other MSP430 platforms under TinyOS and ensure compatibility with existing applications.

Non-preemptive interrupts in TOSSIM: Being a discrete event simulator, interrupts cannot preempt other running code as would happen on real hardware. The initial implementation of the ACK mechanism in the radio stack relied on a hardware interrupt being triggered. When porting the MAC protocol to TOSSIM we ran into this problem and had to redesign the code in order for it to run properly in simulation.

Bug in C compiler (mspgcc): While experimenting with multi-hop routing protocols in TinyOS, using our customised *Surge* application, we apparently hit a nasty bug in mspgcc... Its nature is very, very weird and left us perplexed: the C compiler crashed with a segmentation fault depending on the length of the payload field of our platform's packet structure (`TOS_Msg`).

For this reason, we had to keep the payload's size smaller than what it could be to able to compile *Surge*. Anyway, under normal conditions a compiler should not

segfault irrespective of its input. We filed a bug report to the appropriate mailing list, but no response emerged still.

8.2 Project Achievements

Due to several issues, as summarised above, we were forced to down-scale some of the original project's objectives. Still, we managed to achieve most of the refined goals to a satisfactory degree of success:

- First TinyOS 1.x port to a platform combining an MSP430 micro-controller and an nRF2401 radio chip.
- TinyOS support of all relevant subsystems of IMEC's prototype -- most of MSP430 features and peripherals, nRF2401 radio chip in ShockBurst mode, and all sensing devices.
- Development of a simple, yet reliable and energy efficient, MAC protocol with ALOHA-style operation, tailored for the specific characteristics of the radio unit present on IMEC's prototype platform.
- Empirical evaluation suggests a high packet delivery ratio (above 95%) with relatively low radio duty cycles (25% active), particularly for applications generating regular traffic patterns.
- The above results translate into significant energy savings, if we consider that the radio subsystem accounts for an overwhelming slice of the sensor node's power budget (˜23 mA with radio turned on vs. almost nil when off).
- MAC algorithm implementation extended to support simulation under TOSSIM.
- MAC algorithm implementation extended to support simulation under TOSSIM.
- Multi-hop routing protocols bundled with TinyOS work on top of the MAC developed, which was tested empirically (to the extent possible with only three nodes!) and in simulation.
- Software prepared for release, including user-level and technical documentation.

It must be noted, however, that the MSP430 micro-controller enjoys a quite fully-featured support under TinyOS, which helped our efforts a great deal, and we also used D-systems' nRF2401 radio stack as a reference for our own MAC implementation.

8.3 Future Work

As in all aspects of life, *time* is a precious and scarce resource... Certain issues contributed to further affect our ambitious plans. Hence we are aware of a number of limitations in our work, some of which could be addressed as proposed below:

Multi-hop routing protocol: Finally, regarding one of the initial goals of the project, we barely scratched the surface of energy-efficient, multi-hop routing algorithms. This would be a very interesting area to explore further...

Low power listening (LPL) mode: We have seen that B-MAC [PHC2004] uses low radio duty cycles to achieve energy savings while in recieve mode, but when a node needs to send data it transmits a long-enough packet preamble to ensure that it falls in the destination's wake period and hence stay active until the actual packet data is received.

As it was shown in the evaluation of the MAC, the rate at which packets are generated from an application is strongly correlated with the duty cycle of the MAC layer. If they tend to coincide in time, even without acks the packet delivery ratio appears to be high enough.

We would like to explore extending this *low power listening* scheme to our radio stack, taking into account the limitations of nRF2401's ShockBurst mode: a node can only send and receive simple packets (e.g. cannot generate long preambles to catch the receiver's attention). However, a simple mathematical model could be derived to fine-tune the MAC's retransmissions mechanism in a way that would ensure delivery in an active window of the duty cycle (for instance spreading retransmissions in an exponential manner).

Naturally, as with B-MAC, this taxes the sender in terms of energy consumption. However, the tradeoff seems favourable considering that usually in wireless sensor networks a node spends much more time listening than sending, as well as the receive mode consumes generally more power transmit node in low-power radio systems (as our energy analysis also confirms).

Link quality estimation: As described earlier, the ShockBurst mode of nRF2401 only gives access to a high-level packet interface, the software running on the MCU is only informed when a packet is successfully received by the radio chip (address match and CRC pass).

This scenario does not help a routing protocol estimate link quality to potential neighbours by monitoring bit error or packet loss rates. Nonetheless, a minimal (and unperfect) estimation mechanism could be based around acknowledgemnts in the MAC layer.

Further evaluation of MAC: The MAC algorithm evaluation was rather simple, involving one transmitter and one receiver, with or without acks/retransmissions... Interesting topics to explore include the complexities in the interaction between

factors such as application packet generation patterns, MAC-layer duty cycles, parameters of the retransmissions mechanism, network density, distance, transmit power (-25 to 0 dBm), bit rate (250 Kbps or 1 Mbps), or even compare with other platforms (such as Telos or D-systems).

Also of interest would be to see how lowering power supply voltage (within a controlled range) affects the reliability of communications.

Support for IMEC's USB stick radio interface: The USB stick -- designed to be used as base station -- is similar in architecture to the sensor cubes, the exceptions are the lack of sensing devices and a 4MHz crystal as clock source. The clock subsystem in TinyOS assumes for platforms with 32KHz crystals, hence quite extensive changes are required. However, a preliminary level of support was provided (make system and pin directions) and the modifications to the clock subsystem identified in a `README` file (under `tos/platform/imecusb/`).

8.4 Concluding Remarks

In conclusion, we would like to mention that we started this project knowing nearly nothing about wireless sensor networks, much less the low-level details of the hardware components involved or the software paradigms developed to interface them efficiently; we emerged, in the end, having accomplished some important objectives and done a handfull of interesting things in the process. It is also expected that the deliverables of this project will be (hopefully) of interest to the wider community...

We trekked a long, bumpy path, full of obstacles, but looking back it was well worth it, as all group members have acquired a profound and extensive knowledge about wireless sensor networks!

Bibliography

[AKK2003] K. Akkaya and M. Younis, "A Survey on routing protocols for wireless sensor networks", Ad Hoc Networks (in press), Nov. 2003.

[AKY2002] I. Akyldiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey", IEEE Communications Magazine, Vol. 40, No. 8, pp. 102-116, August 2002.

[AKY2004] I. Akyildiz, I. Kasimoglu, Wireless sensor and actor networks: research challenges, Ad Hoc Networks 2 (in press), pp. 351-367, 2004.

[BAN2003] S. Bandyopadhyay, E. Coyle, "An energy efficient hierarchical clustering algorithm for Wireless Sensor Networks", In Proceedings of IEEE INFOCOM, 2003

[BAR2004] A. Barroso, et al. "Demo Abstract: The DSYS25 Sensor Platform", Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SENSYS2004), Baltimore, USA, November 2004. (Demo).

[BAR2005] A. Barroso, U. Roedig, and C. Sreenan, "u-MAC: An Energy-Efficient Medium Access Control for Wireless Sensor Networks", Proceedings of the 2nd IEEE European Workshop on Wireless Sensor Networks (EWSN2005), Istanbul, Turkey, IEEE Computer Society Press, February 2005.

[BEC2004] K. Beck, "Extreme Programming Explained: Embrace Change", Addison Wesley, 2nd edition, 2004.

[BRE1992] E. Brewer, C. Dellarocas, A. Collbrook, W. Weihl, "PROTEUS: A High-Performance Prallel-Architecture Simulator, Measurement and Modeling of Computer System", Technical Report MIT/LCS/TR-516, MIT, sep 1991

[BUO2003] P. Buonadonna, J. Hill, and D. Culler, "Active Message Communication for Tiny Networked Sensors", In INFOCOM, 2001.

[CUI2005] S. Cui, R. Madan, A. Goldsmith, and S. Lall, "Joint Routing, MAC, and Link Layer Optimization in Sensor Networks with Energy Constraints", submitted to ICC'05, Korea, May 2005.

[DAM2003] T. van Dam and K. Langendoen, "An adaptive energyefficient mac protocol for wireless sensor networks", In The ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, CA, November 2003.

[DEM2005] I. Demirkol, C. Ersoy, and F. Alagoz, "MAC Protocols for Wireless Sensor Networks: a Survey", 2005.

[DOO2005] D. Doolin and N. Sitar, "Wireless sensors for wildfire monitoring", Proceedings of SPIE Symposium on Smart Structures & Materials, San Diego, California, March 2005.

[DSYS] DSYS25 Sensor Platform, Available at <http://www.cs.ucc.ie/misl/dsystems/HTML/dsys25.php>, Date Accessed: 27/05/2005

[DUN2004] A. Dunkels, et al, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”, In First IEEE Workshop on Embedded Networked Sensors, 2004.

[GEH2004] J. Gehrke, and S. Madden, “Query processing in sensor networks”, IEEE Pervasive Compute. 3, pp. 46-55, Jan. 2004

[GOV2002] R. Govindan, et al, “The sensor network as a database”, Technical Report 02-771, Computer Science Department, University of Southern California, Sept. 2002.

[HAN2005] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, “A dynamic operating system for sensor networks”, Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005

[HEI2000] W. Heinzelman, A. Chandrakasan, “Energy-Efficient Communication protocol for Wireless Microsensor Networks”, IEEE Proceedings of the Hawaii International Conference on System Sciences, January 2000, pp. 1-10

[HIL2000] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System Architecture directions for networked sensors”, In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 93-104, Cambridge, MA, USA, Nov. 2000. ACM

[HOR2005] J. Horey, et al, “The Design of a Spreadsheet Programming Interface for Sensor Networks”, Department of Computer Science, UNM Conference (2005)

[JON2001] C. Jones, K. Sivalingam, P. Agarwal, J. Chen, “A survey of Energy Efficient Network Protocols for Wireless Networks”, Wireless Networks, 7(4):343-358, 2001.

[KAN2003] A. Kansal, and M. Srivastava, “An environmental energy harvesting framework for sensor networks”, Proceedings of the 2003 international symposium on low power electronics and design, August 2003.

[KRI2004] L. Krishnamurty, et al., “Wireless Sensor Networks in Intel Fabrication Plants (poster)”, Research at Intel Day, May 2004.

[LVS2003] P. Levis, et al, “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications.” In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)

[LVS2004] P. Levis, et al, “TinyOS: An Operating System for Sensor Networks”, to appear in Ambient Intelligence, J. Rabaey ed. Available online at <http://leitl.org/docs/intel/IR-TR-2004-60-ai-bookchapter-tinyos.pdf>, Date Accessed: 20/06/05.

[LEV2004] P. Levis, et al, “The Emergence of Networking Abstractions and Techniques in TinyOS”, In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation, 2004.

[MAI2002] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. “Wireless sensor networks for habitat monitoring”. In ACM Workshop on Sensor Networks and Applications, 2002.

[MIN2003] R. Min and A. Chandrakasan, “Top Five Myths about the Energy Consumption of Wireless Communication”, Mobile Computing and Communications Review, 7/1 p. 65-67, 2003.

[MSP430U] Texas Instruments, “ MSP430x1xx Family User’s Guide”, Available online at <http://ti.com/msp430>. Date Accessed: 20/06/05.

[NRF2401] Nordic Semiconductor, “nRF2401A Single Chip 2.4 GHz Radio Transceiver”, Available online at <http://www.sparkfun.com/datasheets/IC/ nRF2401A.pdf>. Date Accessed: 20/06/05.

[NS2] The Network Simulator, Available online at <http://www.isi.edu/nsnam/ns/>, Date Accessed: 06/05/2005.

[PIN2004] J. Pinto, "Intelligent Sensor Networks", AutomationWorld, pp. 62, May 2004.

[PIS1999] K. Pister, J. Kahn, and B. Boser, "Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes", Highlight Article in Electronics Research Laboratory Research Summary, 1999.

[PHC2004] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks", The 2nd International Conference on Embedded Networked Sensor Systems, Baltimore, MD, November 2004.

[POL2004] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, "The mote revolution: Low power wireless sensor network devices", In Hot Chips 16, 2004.

[SHT15XX] "SHT1x/SHT7x Humidity & Temperature Sensor", Sensirion. Available online at <http://www.sensirion.com/images/getFile?id=25>. Date Accessed: 20/06/05.

[TEP] TinyOS TEPs, Available online at <http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/beta/teps/>, Date Accessed: 14/05/2005.

[TOR2004] T. Torfs, C. Van Hoof, S. Sanders, C. Winters, and S. Brebels, "Wireless network of autonomous environmental sensors", IEEE Sensors Conference, 2004.

[TRI2005] A. Trigoni, Y. Yao, A. Demers, J. Gehrke and R. Rajaraman, "Multi-Query Optimization for Sensor Networks". International Conference on Distributed Processing on Sensor Systems (DCOSS), 2005.

[VIE2003] M. Vieira, et al., "Survey on wireless sensor network devices", IEEE Emerging Technologies and Factory Automation, vol. 1, p. 537-544, 2004.

[WAN2002] L. Wang, "Survey on Sensor Networks", IEEE Communication Magazine, pp. 230-255, 2002.

[WEI1991] M. Weiser, "The computer for the 21st century", Scientific American, 265(3):94--104, September 1991.

[WOO2003] A. Woo, T. Tony, and D. Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks", ACM SenSys, Los Angelos, 2003.

[YEW2004] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated, adaptive sleeping for wireless sensor networks", ACM/IEEE Transactions on Networking, vol. 12, no. 3, pp. 493 - 506, June 2004.