Design & Implementation of a Wireless Sensor Prototyping Kit

by

Jamison Roger Hope

S.B., 2004

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Ruaidhri M. O'Connor
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

BARKER

Design & Implementation of a Wireless Sensor Prototyping Kit

by

Jamison Roger Hope

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In recent years, wireless sensor networks (WSN) has become an active area of research among computer scientists. In this work, JONA, a prototyping kit for wireles sensors, will be described. The intention of this kit is to open WSN research to interested parties outside of the electrical engineering and computer science communities, who may wish to use wireless sensor networks in their own work. The kit's hardware and software are based upon de facto standards for academic research (Crossbow and TinyOS), with an emphasis on low cost and ease of development. This research has the dual goals of describing a classroom kit and developing a self-contained document providing background material suitable for an introductory project-based class on WSN.

Thesis Supervisor: Ruaidhri M. O'Connor
Title: Assistant Professor

Acknowledgments

First of all, I would like to thank Prof. Ruaidhri M. O'Connor for agreeing to supervise this thesis and for allowing me this opportunity to learn about wireless sensor networks. Your guidance and patience have been indispensable throughout this endeavor.

I would also like to thank Dr. Nathaniel Osgood, who has consistently made the dreary, windowless basement of Building 1 a fun place to work. Your enthusiasm has been an inspiration, and your assistance has proven invaluable time and again.

I must also express my appreciation of the MIT Technology and Development Program and its partnership with the Malaysia University of Science and Technology, which provided me with financial support for this school year, releasing my parents from a great burden they carried faithfully for my four years as an undergraduate.

My mom, Karen Hope. You were my first teacher, and I can always count on you for wise counsel. You've always encouraged me to do my best while maintaining a balance in my life. Most importantly, you taught me to put my trust in God. Happy birthday!

My dad, Bill Hope. You supported my scholastic education, but you also taught me and showed me lots of stuff that you don't learn about in school, especially about fish and wildlife. Thanks for showing me why Florida's such a great and unique place to be. You also taught me a lot about buildings and engines and all sorts of physical systems like that. Any knack I now have for engineering has its roots in that.

My cousin, Isaac Benjamin. You remind me that the world isn't always as complicated as the grown-ups make it seem, and you show me what it means to have faith like a child. Always remember that Jesus loves you, even when you make mistakes. And if the Creator of the Universe knows you and loves you, then who cares what anybody else thinks?

The rest of my family. You've always done nothing but offer me continuous moral and financial support. Any accomplishments I've realized have only been possible because of you.

My friends, some of whom I've known just in these past five years, others more than half my life. Thank you all for always encouraging me and making sure that I remember that life "es muy divertirse." I truly do get by with a little help from my friends.

Saving the best for last, my Lord and Savior, Jesus Christ, without whom I would not be here. What a relief it is to know that I don't have to worry about anything, that, no matter what, God loves me and will take care of me! Here are some most-excellent Bible verses:

> For the wages of sin is death; but the gift of God is eternal life through

Jesus Christ our Lord. *Romans 6:23, KJV*

Trust in the LORD with all thine heart; and lean not unto thine own understanding. In all thy ways acknowledge him, and he shall direct thy paths. *Proverbs 3:5-6, KJV*

Every good and perfect gift is from above, coming down from the Father of the heavenly lights, who does not change like shifting shadows. *James 1:17, NIV*

This is the confidence we have in approaching God: that if we ask anything according to his will, he hears us. And if we know that he hears us—whatever we ask—we know that we have what we asked of him. *1 John 5:14-15, NIV*

Do not be anxious about anything, but in everything, by prayer and petition, with thanksgiving, present your requests to God. *Philippians 4:6, NIV*

Commit to the LORD whatever you do, and your plans will succeed. *Proverbs 16:3, NIV*

Cast your cares on the LORD and he will sustain you; he will never let the righteous fall. *Psalm 55:22, NIV*

"Therefore I tell you, do not worry about your life, what you will eat or drink; or about your body, what you will wear. Is not life more important than food, and the body more important than clothes? Look at the birds of the air; they do not sow or reap or store away in barns, and yet your heavenly Father feeds them. Are you not much more valuable than they? Who of you by worrying can add a single hour to his life?

"And why do you worry about clothes? See how the lilies of the field grow. They do not labor or spin. Yet I tell you that not even Solomon in all his splendor was dressed like one of these. If that is how God clothes the grass of the field, which is here today and tomorrow is thrown into the fire, will he not much more clothe you, O you of little faith? So do not worry, saying, 'What shall we eat?' or 'What shall we drink?' or 'What shall we wear?' For the pagans run after all these things, and your heavenly Father knows that you need them. But seek first his kingdom and his righteousness, and all these things will be given to you as well. Therefore, do not worry about tomorrow, for tomorrow will worry about itself. Each day has enough trouble of its own." *Matthew 6:25-34, NIV*

How awesome is that? A Heavenly mandate to relax. Dude, you rock, God.

And now, I guess it's time for me to demonstrate that, as Dilbert says, "there's nothing wrong with my verbal skills; it only seems that way because my math skills are so high."

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Wireless Sensor Networks

In recent years, wireless sensor networks (WSN) has become an active area of research. Indeed, wireless sensor networks have been promised to "change the way we live our everyday lives" [24]. Some of the applications to which networks have already been deployed include monitoring animal habitats [32], providing inventory control [38], and monitoring environmental conditions in buildings [34].

A wireless sensor network consists of spatially distributed clusters of autonomous, smart devices which collectively measure, process, and communicate sensor data through self-configuring "ad hoc" networks. Each device (or "mote"[1]) contains a radio transceiver, a processing unit (microcontroller) and an array of sensors. Motes are typically powered by batteries, but power harvesting techniques are sometimes used as well. Depending on the intended application, WSN nodes may be set up to sense environmental factors such as light, heat, humidity, even seismic activity, or to detect artificial signals such as radio[2] which they can use to triangulate the locations of objects emitting the signals [18].

[1]Use of the term "mote" reflects the somewhat ambitious goal of having wireless sensor nodes the size of microscopic particles, so-called "smart dust" [39].

[2]In this case, a node may be equipped with something like an RFID tag reader, or it might use its primary transceiver.

1.2 Current Research and WSN Implementations

The apparent applicability of WSN to a diverse set of monitoring needs has led to widespread interest in the broader engineering community. Research groups at a number of universities, including MIT, have been investigating WSN-related issues, and there are already commercial implementations available from vendors including MIT startups Ember Corp. [17] and Millennial Net, Inc. [33], and UC Berkeley startups Dust Networks^{TM} [16] and Crossbow Technology, Inc. [13].

1.2.1 Berkeley, TinyOS and nesC

At the University of California, Berkeley, WSN researchers have produced several generations of node hardware based on commercial, off-the-shelf components, that have been manufactured by Crossbow and others. Berkeley research has also focused on software issues, and they have developed a minimal open source[3] operating system designed specifically for WSN nodes and other embedded devices called TinyOS [23].

Following the initial release of TinyOS, Berkeley researchers went on to develop a new programming language called nesC [19] to reinforce the programming model found in TinyOS. Since their introduction, nesC and TinyOS have acquired a large following [47] and have become the de facto standards for WSN software in academic research.

Crossbow motes such as MICA—which was, in fact, first developed as a Berkeley project [22] and then made commercially available by Crossbow—running TinyOS have become a de facto standard for academic research into WSN, due to the open source nature of the software and the freely available hardware schematics [47]. Additionally, Berkeley researchers have partnered with Intel to create other hardware platforms for TinyOS [26].

[3]See Section 2.4.1 on page 61.

1.2.2 Habitat Monitoring on Great Duck Island

MICA motes were used—by a Berkeley group in collaboration with the College of the Atlantic—for a habitat monitoring project on Great Duck Island in Maine [32].

Seabirds, specifically Leach's Storm Petrels, use this island for nesting, and biologists wanted to study them. However, human incursion into their habitat can have catastrophic impacts; so, the researchers decided to use a wireless sensor network to study the birds remotely. Motes were placed around the island—including within nest burrows—while the birds were away, to prepare for the upcoming mating season. They were fitted with an array of sensors to measure, among other things, temperature, light levels, humidity, and infrared radiation (to detect the body heat of the petrels when they were in their nests). The motes formed a network and sent their sensor readings to a computer connected to the Internet. Researchers could then receive the data back at Berkeley, safely out of the birds' way. During a four-month deployment, 1.2 million readings were logged [40].

The experiment was by no means flawless; some motes suffered corrosion in their battery terminals, and others failed for various reasons. However, the GDI project served as a valuable proof-of-concept, that wireless sensor networks are a viable alternative to traditional habitat monitoring techniques, with many advantages over the latter, including reduced cost and reduced intrusiveness into sensitive areas.

1.2.3 Seismic Monitoring

Other, even earlier experiments involved using motes—in this case Rene, MICA's predecessor—to monitor seismic activity. Two controlled experiments were performed [20], one in Japan and the other on the Berkeley campus. In each case, motes were deployed with two-axis[4] accelerometers to measure accelerations due to simulated seismic activity.

In Japan, explosive charges were detonated under the ground to induce soil liquefaction, a phenomenon which occurs during earthquakes. Sensors then measured the

[4]That is, they can sense acceleration both side-to-side and forward-and-backward.

resultant accelerations at ground level in different points.

In the other experiment, motes were placed in a grid formation on a wall and elsewhere in a full-scale three storey wood-frame building on a shake table. The building was then shaken to simulate an earthquake and accelerometer data was collected, to see which parts of the building had suffered the most stress. The WSN network successfully pointed to a particular area of a side wall—verified by a more expensive and cumbersome traditional sensor apparatus—which did not show any obvious visible damage.

There were problems encountered during both of these experiments, particularly with regards to radio interference, but they demonstrated that WSN technology can become a useful tool to measure the structural health of buildings and other man-made structures. For example, a bridge fitted with a WSN network may perform a self-diagnostic following a tremor and determine whether it is still safe for people to cross [20].

1.2.4 The MIT μAMPS Project

Some of the more prevalent theoretical WSN research issues are reducing power expenditure to increase battery life; miniaturizing components; and developing software which will allow individual nodes to form an ad hoc network—one which has not been carefully designed ahead of time, but rather emerges spontaneously as nodes discover and begin to communicate with their neighbors—and to work together to solve problems.

It is self-evident that devices which run on electricity consume more power when they are on than when they are off. So, one of the simplest ways to conserve energy in an electronic system is to turn components off when they are not needed. This technique, called *duty cycling*, is often employed in laptop computers, which will turn their CPUs—and other components—off every time they are not needed. Duty cycling can also be employed in WSN motes, having them shut down sensors and the radio, and put the processor in a "sleep" mode[5] whenever they have nothing to do.

[5]Microcontrollers which have this type of low-power mode typically have the ability to wake

Here at MIT in the Microsystems Technology Laboratories, the μAMPS (micro-Adaptive Multi-domain Power aware Sensors) Project [10] is focused on extending node battery life while hardware components are turned on by making components which are "power aware" [35]. After producing an initial node called μAMPS-1 based on commercial, off-the-shelf components [11], the Project turned to designing custom parts, such as a power aware radio transceiver [29].

Some of the recommendations produced by the μAMPS Project to increase power awareness include dynamic voltage scaling, energy-aware computing, and multi-hop routing [11].

Dynamic Voltage Scaling

Dynamic voltage scaling [21] is a scheme for making computing devices power aware by adjusting power consumption based on changes in the computation load. The speed of a processor depends on a circuit element called a crystal oscillator which produces a periodic waveform (a "clock signal"). Each computation performed by the processor takes an integral number of oscillator periods. It is often possible to make a processor faster by supplying it with a faster oscillator, but this increase does not come free; for the chip to continue to function properly, its voltage supply must be increased (which will cause it to consume more energy). The converse is also true: by reducing clock speed, the chip can operate at a lower supply voltage[6].

Therefore, when power consumption is an issue, it is best not to make the computer faster than it needs to be. Dynamic voltage scaling takes advantage of this idea by examining the current processing load and setting the supply voltage and associated oscillator frequency as low as they can be while still getting the job done on time.

themselves back up at predetermined times or in response to some external stimulus. A built-in timer keeps ticking away while the rest of the microcontroller naps.

[6]However, there is still an inherent minimum voltage below which the processor will not operate, regardless of how slow its crystal oscillator is.

Energy-Aware Computing

Energy-aware computing is a related concept, but instead of adjusting processor voltage based on load, it switches processing elements based on the size of the data to be processed.

For example, suppose the computer needs to add $3 + 2 = 5$. In binary, this equation is $11 + 10 = 101$. Each of the inputs is two bits long, and the output is three bits long[7]. In a typical PC CPU, each arithmetic operation is performed on 32-bit inputs. In order to compute $3 + 2 = 5$, they perform the addition:

$$\begin{array}{r} 00000000000000000000000000000011 \\ +\ 00000000000000000000000000000010 \\ \hline 00000000000000000000000000000101 \end{array}$$

Not only do they compute the values of the three bits needed to hold the result of 5, they also must compute the values of the other twenty-nine bits. This needless computation wastes power. In a processor employing energy-aware computing, rather than having a single 32-bit adder component, it might have several: 5-, 8-, 10-, 16-, and 32-bit versions, for instance. Then when the computer saw that it had to add $3 + 2 = 5$, it could choose to perform the computation using the smallest adder that could adequately hold the sum, in this case the 5-bit version[8]. Then, it only must add $00011 + 00010 = 00101$, avoiding twenty-nine unnecessary 1-bit additions and their associated waste of power[9].

Multi-Hop Routing

One final suggestion in [11] is to use multi-hop routing for communications. Suppose a node in a wireless sensor network needs to send a message to another node on the other end of the network. It can either try to send the message directly to the

[7]See Appendix B on page 151 for an explanation of binary and bits.

[8]Of course, it would have to make this determination based on seeing two 2-bit inputs, not on seeing the 3-bit output (which it has yet to compute); it can do this because the sum of any two n-bit numbers is at most $(n + 1)$ bits long.

[9]The act of inspecting data input sizes will be a new source of power loss—nothing comes free; it is assumed that the average amount of power required to examine the data will not exceed the average amount of power saved by using appropriately-sized components.

other node, or it can send it to a closer node which will then forward the message on toward its destination. Each transmission of the message is called a "hop" because it hops from node to node. The choice, then, is between *single-hop* and *multi-hop* communication.

It may seem simpler to have every node be able to communicate directly with every other node; this avoids the problem of having intermediate nodes which must figure out how to route messages. However, broadcasting a radio signal is one of the most energy-intensive operations a node can perform, and the energy required to transmit a message a distance r grows as r^2 or faster [29]. Suppose there are four nodes of a network arranged as shown in Figure 1-1, and Node A has a message to send to Node D. Let r be the distance between each node.

If Node A is to send its message directly to Node D, it must have a transmission radius of $3r$. The energy requirement is then proportional to $(3r)^2 = 9r^2$. However, if Node A sends its message to Node B, Node B sends it to Node C, and Node C delivers it to Node D, then each node only needs a transmission radius of r. The energy requirement for the three transmissions is, then, proportional to $r^2 + r^2 + r^2 = 3r^2$, which is a substantial savings over the original $9r^2$.

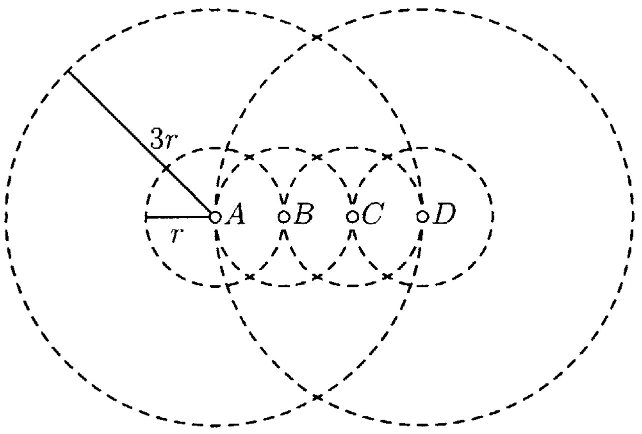

Figure 1-1: Transmission radii in a wireless sensor network.

1.2.5 Other Related Research

Other theoretical research into specific issues related to wireless networking—and to the *ad hoc* networking found in WSN networks in particular—has been going on for years.

Piconet [7] is an embedded wireless communication network which was introduced in 1997, three years before the debut of TinyOS. It is a radio protocol designed to enable ad hoc communication between various nearby personal electronics (much like today's Bluetooth). Piconet is meant to be an add-on feature for these electronics, and it could in fact be used in conjunction with TinyOS, which is designed to work with a variety of interchangeable radio modules, including Bluetooth [8] and ZigBee [50][10].

Regardless of the communication protocol chosen, a wireless network must be able to send data where it needs to go, when it needs to go. This will generally entail routing a message from node to node until it reaches its destination. If all the nodes are stationary, then they may be able to develop routing tables which might say, for example, that "Node y lies between me (Node x) and Node z. So, if I have a message for z, I should send it to y and let y worry about routing it onward." Of course, these tables will change over time; if y's battery dies, then x will have to find a new route to z.

If nodes are mobile, then routing becomes more tricky. A node cannot simply send its message to a neighbor, expecting the destination node to be somewhere on the other side, because that neighbor or the destination (or both) might have moved. A node could simply transmit its message once to one of its neighbors and hope that it eventually reaches its destination, but this could lead to a very large delay as the message makes its way around the network. It may be a better idea to send out several copies of the message along different paths, so that one of the copies will reach its destination quicker than the solitary message would have. Flooding the network with duplicate messages will, on the other hand, increase congestion and reduce the network's overall data rate. It has been suggested that, for general mobile wireless

[10]Crossbow offers the MICAz mote which uses the ZigBee protocol for communication; the Intel®Mote uses Bluetooth. Both run TinyOS as their operating system.

networks, the ratio of average delay to average data rate is proportional to the number of nodes in the network, or higher [37]. In other words, any scheme to reduce delay by a certain factor will reduce the network's data throughput capacity by at least that same factor.

In a WSN network, though, battery life may be more of an issue than delay. With a network of hundreds or thousands of nodes spread throughout an animal habitat, changing batteries is not an option, so it is important for the radio communication to consume as little power as possible—even if this makes it take longer for messages to get where they are going.

As noted previously, the μAMPS group has been looking at ways to make sensor nodes more energy-efficient and power aware. Section 1.2.4 described energy-aware computing in which the internals of a processor are dynamically reconfigured to conserve energy. Well, peripheral components like radios can also incorporate energy awareness, but not quite in the same way. For instance, a mote could have its radio adjust its transmission power—remember from Figure 1-1 that more power means a larger transmission radius—so that its signals reach their destinations but do not needlessly carry further. Essentially, the mote would not "shout" if a "whisper" would suffice.

An energy-aware processor would be able to adjust its own parameters automatically; its energy awareness would be built into the hardware, without needing any special functionality to be programmed into it. On the other hand, the processor, which is the brain of the mote, must always remain in control of its peripheral devices, so it would not be good for them to go changing their parameters on their own without the processor knowing about it. Instead, the microcontroller should control the power-saving features of the peripherals; for it to do so, this control must be programmed in the software. To this end, some of the μAMPS effort has been on designing protocols, algorithms, and applications that are aware of details of the hardware[11] and can tweak parameters to minimize energy usage [44].

[11]Note that this is in direct contrast to the programming language abstract layer which allows for portability across hardware platforms; here, the software must be tailor-made for a specific set of hardware components.

Meanwhile, researchers at MIT's Laboratory for Information and Decision Systems have also studied the issue of energy efficiency in wireless networks, specifically the issue of energy efficient communication. In [28], they present "cooperative routing" which saves energy through a combination of energy-efficient route selection—i.e. choosing the right hops to minimize energy expenditure—and energy-efficient transmission—making each hop as efficient as possible. When a node has a message to transmit and it sends it to some intermediate node as part of a multi-hop path, there will often be other intermediate nodes which also overhear the message. With cooperative routing, some subset of the nodes which heard the transmission (possibly all of them) will all rebroadcast the message at the same time. Their simultaneous transmissions will reinforce each other and the message will carry further than it would have if only one node had transmitted it. Equivalently, they can transmit at lower power than normal and have the message travel as far as it would have if one transmitted it at full power. The energy savings of cooperative routing comes from this second formulation. Although an individual node may consume more energy than it would have without cooperative routing[12], the total amount of energy used throughout the network in the delivery of the message will be reduced. Depending on the network's *topology*—how the nodes are physically arranged—cooperative routing can provide energy savings of more than 50% [28].

1.3 How to Read This Document

This document introduces a new wireless sensor prototyping platform called the JONA. This prototyping kit is intended to facilitate the exploration of wireless sensor networks. Most of the proposed uses of WSN technology are solutions to problems faced by biologists, civil engineers, factory workers, construction workers, and many other diverse groups. One thing (perhaps the only thing) these groups have in common is that their primary disciplines are neither electrical engineering nor computer

[12]Consider a node which does not lie along the original multi-hop path but is within hearing distance. With traditional routing, this node will not transmit at all, so it will use no energy; with cooperative routing, it will transmit, so its power consumption will increase.

science. So, it is not reasonable to assume that all individuals who might be interested in WSN, and this prototyping kit, are already well-versed in EECS theory; someone "who wants to use wireless sensor node technology should not have to earn a Computer Science Ph.D. in order to do so" [20]. This document, intended to serve as a self-contained User's Guide for the kit, therefore provides a survey of the basics of electronics, microcontrollers, and TinyOS programming prior to a discussion of the kit itself.

Chapter 2 offers something of a crash course in electrical engineering and computer science. It is intended to supply the reader with enough of a background to make sense of the subsequent chapters.

Chapter 3 explains in greater detail the need which the JONA is intended to meet. It goes on to describe hardware and software requirements which must be satisfied, along with other design goals.

Chapter 4 presents the implementation of the JONA Prototyping Kit. Along with an examination of its constituent parts, there is a tutorial on how to program JONA nodes.

Chapter 5 briefly describes a number of sample applications which have been developed for JONA nodes, showing how they may be used in an actual deployment of a wireless sensor network.

Chapter 6 concludes with an analysis of JONA strengths and weaknesses, and suggests future directions for the prototyping kit.

Chapter 2

An Overview of Relevant Theory

This work is aimed at providing material suited to the development of a class on WSN applications. In particular, the JONA kit was developed for a Civil Engineering course on WSN being taught simultaneously at MIT—as 1.961—and at Malaysia University of Science and Technology (MUST)—as CEM508. The text chosen for this class was *Practical Electronics for Inventors* by Paul Scherz [43]. This chapter provides a brief overview[1] of some main ideas in electrical engineering and computer science theory found in that book. A cursory understanding of these concepts is assumed in later chapters. This chapter also introduces some parts which can be used to achieve specific functionalities in a prototype, and equipment which will come in handy to test and monitor the workings of the electronic devices. Each section is fairly self-contained—though later topics do build on concepts introduced in earlier ones—and the reader may feel free to skip over familiar material. Sections herein can then be referred back to while reading the rest of the document without having to reread the entire chapter.

[1]The material in this chapter is really a set of "snapshots" of the relevant sections of the text, which the student should read.

2.1 Elementary Circuit Theory

2.1.1 Current

Current, expressed in *amperes* (A), is a measure[2] of the flow of electricity. More precisely, it represents the rate of flow of electric charges (i.e. electrons) past the point where the measurement is being taken. In equations, current is usually represented by the letter I. If a current is constant over time, then it is indicated by the uppercase I; one that varies over time is signified by the lowercase i[3]. If the current from point A to point B in a circuit is measured to be -10mA, it is equivalent to say that the current from point B to point A is 10mA; by convention, this current would be said to flow in the direction in which it carries a positive value, i.e. from B to A, and in a circuit diagram it would be indicated by an arrow from B to A.

This convention that current flows in the positive direction stems from the erroneous belief once held that in an electric current, positively-charged particles were flowing around the circuit. Later, the particles flowing in current—electrons—were discovered, and were found to be negatively charged. This meant that the particles actually flowed in the opposite direction[4]. However, when an electron moves, it leaves behind a "hole"[5]; when another electron comes behind the first, it might fill in the hole left by the first. In doing so, it will leave its own hole further back. So, as negatively-charged electrons move in one direction, positively-charged holes "move" in the opposite direction, i.e. in the direction of current flow.

[2]Current is measured using a device called an *ammeter*. This is typically one of the "meters" in a *multimeter*. See Section 2.3.1 on page 59.

[3]"i" is really shorthand notation for the symbol "$i(t)$", which makes the current's nature as a function of time explicit.

[4]By definition, a current of 1 ampere is a flow of 1 Coulomb—a unit of positive charge—per second. This rate of positive-charge influx could either be the result of the arrival of mobile positive charges, or it could be the result of the departure of mobile negative charges. The latter was found to be the truth.

[5]In a neutral atom, there are just enough electrons to counter the positive charge carried by the protons in the atom's nucleus. When an electron leaves, the atom will have a charge imbalance and it will have a positive net charge. So, this hole can be thought of as a virtual particle carrying a positive charge.

Kirchhoff's Current Law The Conservation of Mass law of physics states that matter is neither created nor destroyed. It follows from this statement that all the electrons which flow into a given point in a circuit must flow back out of it. In other words, the sum of all the currents flowing into a point in a circuit must equal the sum of all the currents flowing out of that point,

$$\sum i_{in} = \sum i_{out}, \tag{2.1}$$

and this is true for every point in the circuit. This is known as *Kirchhoff's Current Law* (KCL). It is often more convenient—since current flow directions may not be known *a priori*—to consider all currents to be flowing inward, and to state KCL as

$$\sum i_{in} = 0. \tag{2.2}$$

Currents which really flow outward will carry a negative sign ($i_{j_{out}} = -i_{j_{in}}$), so this is equivalent to subtracting $\sum i_{out}$ from both sides of Equation 2.1.

KCL can often be used to calculate unknown current values in a circuit. For instance, consider the left side of Figure 2-1. All the current which flows down through the top circuit element must go on to flow down through the other three, so $i_1 = i_2 + i_3 + i_4$. As long as three of these can be found through some other means, this equation will provide the value of the fourth.

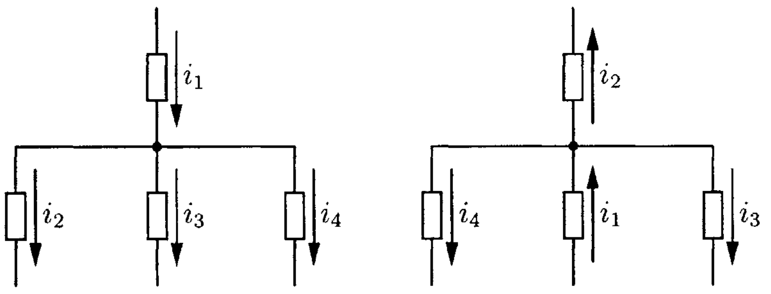

Figure 2-1: A simple application of KCL.

Now notice that the right side of Figure 2-1 shows the same connections between the circuit elements (although they are rearranged spatially). The relationship between the currents is the same as before, but it is not as immediately recognizable as such. In a real circuit, it may be difficult to tell at a glance whether currents will be flowing up or down or right or left, so it is often simpler to let all currents flow inward, as in Figure 2-2. Once the relationship between the currents has been established (in this case, $i_a+i_b+i_c+i_d = 0$), other information may help to determine which currents are positive and which are negative.

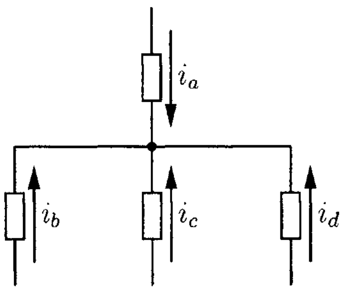

Figure 2-2: Another simple application of KCL.

2.1.2 Voltage

Voltage, expressed in *volts* (V), is a measure[6] of the difference in electrical potential (the "voltage drop") between two points in space. A place of higher potential is more positively charged than a place of lower potential. Like charges repel and opposites attract, so moving a positively charged particle from a lower potential to a higher potential requires the application of energy to overcome the particle's natural tendencies. This is analogous to the energy required to lift something; in each case, the object is being supplied with more potential energy. Voltage measures potential, though, not potential energy. Just as lifting a more massive object would require more energy to be applied, so would moving a particle carrying a greater charge; to determine the potential difference between two points, the measured energy expenditure must be normalized. The voltage, then, is the energy per unit charge. Voltage is represented in equations by the letter V. As with current, uppercase V indicates a constant voltage, and lowercase v or $v(t)$ one which varies over time.

Voltage is a relative measurement; in order to determine the voltage at some

[6]Voltage is measured using a *voltmeter*, also a standard component of a multimeter.

particular point of interest, it must be compared to that of some other point[7]. In a circuit diagram, a voltage between two points is indicated by a "+" at one point and a "−" at the other. The *voltage drop* is the amount by which the voltage drops in moving from the positive end (+) to the negative end (−). Also, voltage is additive. Consider Figure 2-3. If the voltage drop across device A is measured to be V_A, and the drop across B V_B, then the total potential difference between the "+" terminal (end) of A and the "−" terminal of B is $(V_A + V_B)$. Finally, note that if the drop from the "+" terminal to the "−" terminal of A is V_A, then the drop measured from the "−" terminal to the "+" terminal of A will be $-V_A$: voltage has a direction associated with it.

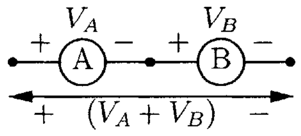

Figure 2-3: Two Voltage Sources in Series

Digital electronics typically have two important voltage levels or "rails": power and ground. Ground, the common reference point, is usually (though there are exceptions) the lowest potential level found in the circuit, with a potential[8] of 0V. The power rail is (usually) the highest potential found in the circuit, and is the wire which delivers the electricity to drive each component. It is often referred to[9] as "V_{CC}" or "V_{DD}" and is often 5V, though increasingly devices are being designed to run at 3V (or lower).

Kirchhoff's Voltage Law Voltages is a vector quantity (it has both magnitude and direction), so the sum of all the voltage drops around any closed loop in a circuit

[7]In practice, a single point is chosen to serve as the reference to all other points in a circuit. In wired equipment (i.e. equipment powered by an electrical outlet rather than by batteries), this reference point is often connected by a wire to the Earth itself. The equipment is then said to be "grounded" and the reference point is referred to as "ground". In fact, the reference plane is often referred to as "ground" even when the circuit is not truly grounded.

[8]Recall that this voltage is the potential difference between this point and the reference—this same point.

[9]These subscripts refer to details of a device's internal construction; their precise meaning is beyond the scope of this material.

must be zero,

$$\sum_{k \in loop} v_k = 0. \tag{2.3}$$

This property is known as *Kirchhoff's Voltage Law* (KVL), and it follows from Conservation of Energy[10].

KVL can be used to calculate unknown voltages. Consider Figure 2-4. There are two loops, one involving the voltage source, v_s, and v_1, v_2, and v_3; and the other involving v_2, v_3 and v_4.

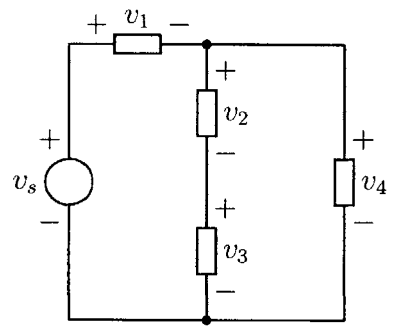

Figure 2-4: A simple KVL example.

When a circuit element labelled v is crossed from the negative end to the positive end[11], the voltage will increase by v; when it is crossed from positive to negative, it will decrease by v. So, going clockwise around the left loop, KVL says that $v_s - v_1 - v_2 - v_3 = 0$. Meanwhile, the right loop (also clockwise) yields the equation $v_3 + v_2 - v_4 = 0$. If any three of these voltages are determined through measurement or other calculation, then these two KVL equations can be used to find the remaining two.

[10]Imagine moving an electron around the loop. As it goes from point to point, energy is either added to it—as it moves from a place of higher potential to a place of lower potential (energy must be added to overcome the repellent force between the negatively-charged electron and the (relatively) negatively-charged point of lower potential)—or taken away from it—as it moves from a lower voltage to a higher, more positive voltage (to which the electron is naturally attracted); when the electron gets back to its starting place, it must have the same amount of energy that it started out with, so all its changes in voltage must have cancelled out. If this were not the case, then sending the electron around the loop repeatedly would either create more and more energy in the Universe, or it would drain more and more energy out of the Universe. Either would violate Conservation of Energy, which states that energy is neither created nor destroyed.

[11]Note that these designations of a "positive end" and a "negative end" are chosen arbitrarily. If v happens to be negative, then this "increase by v" will actually be a reduction, and vice versa.

2.1.3 Power

Power, expressed in *Watts* (W), is a measure of the rate of energy usage. Because voltage is energy per unit charge, and current is charge per unit time, power in electrical systems can be calculated using the equation,

$$P = IV, \tag{2.4}$$

where P denotes power, and I and V are as described above.

For example, suppose there is a two-terminal device connected to a 9V battery. The voltage drop across the two terminals is measured to be 9V, and the current through the device is measured to be 1mA. Then the device is consuming 9mW of power[12].

2.1.4 Resistance and Resistors

A *resistor* is a circuit element which is composed of a material which impedes or "resists" the flow of electricity through it. *Resistance*[13] R is measured in units called *Ohms* (Ω).

The relation between current, voltage and resistance is described by *Ohm's Law*:

$$I = V/R \tag{2.5}$$

(see Figure 2-5). This equation can be combined with the power equation to express power consumption in terms of resistance and either current or voltage: $P = IV = I^2R = V^2/R$.

When two resistors with resistance values R_1 and R_2 are placed in series (end-to-end), the combination is equivalent to a single resistor with value $R_{equiv} = R_1 + R_2$ (see

[12]The prefix "m" applied to a unit symbol denotes "milli-", i.e. one thousandth of the base unit. See Appendix A on page 149 for a table of other common prefixes.

[13]The resistance of a resistor is usually considered to be an intrinsic, unchanging property, so the uppercase R is used as its symbol in equations. There are related devices such as thermistors whose resistance changes as a function of some environmental condition—in this case, temperature. For these, lowercase r might be used for consistency to indicate that the associated resistance is not a constant, but usually the uppercase R is used even in these cases.

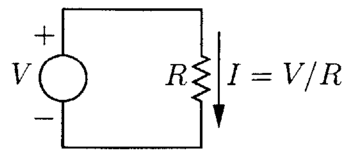

Figure 2-5: Ohm's Law

Figure 2-6). If they are placed in parallel (side-to-side), the combination is equivalent to a single resistor with value $R_{equiv} = 1/(1/R_1 + 1/R_2) = R_1R_2/(R_1 + R_2)$ (see Figure 2-7). These results follow from Ohm's Law and Kirchhoff's Laws, and they readily generalize to larger, more complex resistor networks.

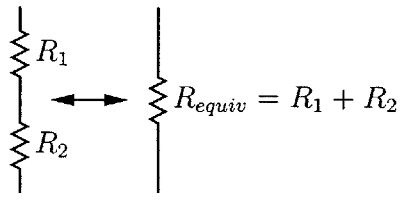

Figure 2-6: Two Resistors in Series

Let V_1 and V_2 represent, respectively, the voltages across resistors R_1 and R_2 in Figure 2-6. By KVL, the total voltage across both resistors, V_{equiv}, must be equal to the sum of V_1 and V_2:

$$V_{equiv} = V_1 + V_2. \tag{2.6}$$

By Ohm's Law, the currents through the two resistors are given by

$$I_1 = V_1/R_1 \tag{2.7}$$

and

$$I_2 = V_2/R_2. \tag{2.8}$$

By KCL, the current through R_1 must equal the current through R_2,

$$I = I_1 = I_2. \tag{2.9}$$

Now consider a single equivalent resistor, R_{equiv}. The current through this resistor,

I_{equiv}, will equal the current flowing through R_1 and R_2:

$$I_{equiv} = I, \tag{2.10}$$

and will, by Ohm's Law, be given by the equation

$$I_{equiv} = V_{equiv}/R_{equiv}. \tag{2.11}$$

Combining Equation 2.6 with Equations 2.7, 2.8, and 2.11 yields

$$I_{equiv}R_{equiv} = I_1R_1 + I_2R_2. \tag{2.12}$$

According to Equations 2.9 and 2.10 all the currents are equal, so each term in Equation 2.12 can be divided by I to give the equivalence relationship for two resistors in series,

$$R_{equiv} = R_1 + R_2. \tag{2.13}$$

If n resistors are placed in series, the equivalent resistance is $R_{equiv} = \sum_{i=1}^{n} R_i$.

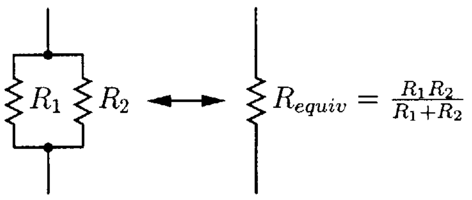

Figure 2-7: Two Resistors in Parallel

Now, let V_1 and V_2 be the voltages (measured from top to bottom) across the two resistors R_1 and R_2 in Figure 2-7. According to KVL, the sum of the voltages around the closed loop formed by R_1 and R_2 must be 0. Following the loop in the clockwise direction, this says that $-V_1 + V_2 = 0$, or

$$V_{equiv} = V_1 = V_2. \tag{2.14}$$

According to Ohm's Law, the currents through the two resistors are given by the two equations

$$I_1 = V_1/R_1 \tag{2.15}$$

$$I_2 = V_2/R_2. \tag{2.16}$$

By KCL, the current I_{equiv} flowing down into the pair of resistors must be equal to the sum of the currents flowing through them,

$$I_{equiv} = I_1 + I_2. \tag{2.17}$$

Now, an equivalent resistor R_{equiv} will have to satisfy Ohm's Law, so

$$I_{equiv} = V_{equiv}/R_{equiv}. \tag{2.18}$$

Replacing the terms in Equation 2.17 with the results of Equations 2.18, 2.15, 2.16, and 2.14 gives $V/R_{equiv} = V/R_1 + V/R_2$, or

$$1/R_{equiv} = 1/R_1 + 1/R_2. \tag{2.19}$$

Solving Equation 2.19 for R_{equiv} yields the equivalence relationship for two resistors in parallel,

$$\begin{aligned} R_{equiv} &= \frac{1}{1/R_1 + 1/R_2} \\ &= \frac{R_1 R_2}{R_1 + R_2} \end{aligned} \tag{2.20}$$

If n resistors are placed in parallel, the equivalent resistance is $R_{equiv} = 1/(\sum_{i=1}^{n} 1/R_i)$.

Voltage Divider

One extremely common application of resistors is as a *voltage divider*. This is a circuit which uses two resistors in series to scale down an input voltage by some constant factor. Consider Figure 2-8. According to KCL, all the current which flows through the resistor marked R_1 must[14] flow through R_2 to ground (indicated by the inverted triangular symbol at the bottom). As v_{in} varies over time, this current, i, will follow it, by Ohm's Law and Equation 2.13, according to the equality

$$i = \frac{v_{in}}{R_1 + R_2}. \tag{2.21}$$

Now consider i as it flows through the second resistor. According to Ohm's Law,

$$i = v_{out}/R_2. \tag{2.22}$$

Equations 2.21 and 2.22 can be combined to relate v_{out} to v_{in}:

$$\frac{v_{out}}{R_2} = \frac{v_{in}}{R_1 + R_2}. \tag{2.23}$$

Finally, solving for v_{out},

$$v_{out} = v_{in}\frac{R_2}{R_1 + R_2}. \tag{2.24}$$

For example, if R_1 is 120kΩ and R_2 is 180kΩ, then Equation 2.24 simplifies to $v_{out} = 3v_{in}/5$. Notice that the scaling factor is determined by the ratio of resistances, not the actual values themselves. For instance, if R_1 were 2Ω and R_2 were 3Ω, the output would still be scaled by a factor of 3/5; the current flowing through the resistors, on the other hand, would be several orders of magnitude larger, and the voltage divider would consume much more power[15].

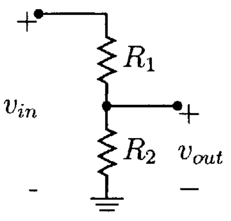

Figure 2-8: Voltage Divider

[14]This assumes that no current escapes through the terminal marked v_{out}. In general, v_{out} will be connected to other circuitry with some effective resistance of R_{load}. As long as $R_{load} \gg R_2$, the current which escapes into the load will be negligible.

[15]Recall that power is given by the equation $P = V^2/R$. With the voltage fixed, reducing R increases P.

2.1.5 Capacitance and Capacitors

A *capacitor* is a circuit element which stores charge. How much charge a given capacitor can store is indicated by its *capacitance*, measured in *Farads* (F). In equations, capacitance is denoted by C. The relation between voltage and current for a capacitor is given by

$$i_C = C dv_C/dt, \tag{2.25}$$

where $i_C = i_C(t)$ is the (changing) current through the capacitor, dv_C/dt is the rate of change of the voltage ($v_C(t)$) across the capacitor, and C is the capacitance.

Like resistors, capacitors can be placed in series and in parallel; however, the equations for determining the resultant capacitances are the reverse of their resistor analogs. If two capacitors with values C_1 and C_2 are placed in series, the equivalent capacitance is given by $C_{equiv} = C_1C_2/(C_1 + C_2)$ (see Figure 2-9). If they are placed in parallel, the equivalent capacitance is $C_{equiv} = C_1 + C_2$ (see Figure 2-10). As was the case for resistors, these results can easily be obtained from Kirchhoff's Laws and the current-voltage relation for capacitors, equation 2.25.

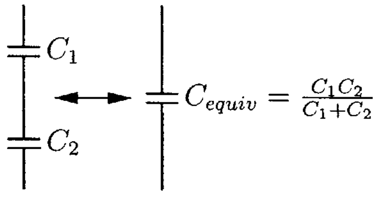

Figure 2-9: Two Capacitors in Series

Let i_C be the current flowing through the capacitors labelled C_1 and C_2 in Figure 2-9. By KCL, this current is equal to the current flowing through each capacitor,

$$i_C = i_{C_1} = i_{C_2}. \tag{2.26}$$

Meanwhile, KVL states that the total voltage across the two capacitors, v_C, is equal to the sum of the voltages across them:

$$v_C = v_{C_1} + v_{C_2}. \tag{2.27}$$

Taking the first derivative of v_C as defined in Equation 2.27 gives

$$dv_C/dt = dv_{C_1}/dt + dv_{C_2}/dt. \tag{2.28}$$

Now, using Equations 2.25 and 2.26 to write the currents through C_1, C_2, and the equivalent combined capacitor C_{equiv} yields

$$i_C = C_1 dv_{C_1}/dt \tag{2.29}$$
$$i_C = C_2 dv_{C_2}/dt \tag{2.30}$$
$$i_C = C_{equiv} dv_C/dt. \tag{2.31}$$

Solving Equations 2.29, 2.30 and 2.31 for their respective voltage terms and substituting them into Equation 2.28 produces

$$i_C/C_{equiv} = i_C/C_1 + i_C/C_2, \tag{2.32}$$

which simplifies to

$$1/C_{equiv} = 1/C_1 + 1/C_2. \tag{2.33}$$

Noting the similarity to Equation 2.19, this can be rewritten, in a form mirroring Equation 2.20, as

$$C_{equiv} = \frac{C_1 C_2}{C_1 + C_2}. \tag{2.34}$$

It follows immediately that if n capacitors are placed in series, the equivalent capacitance will be given by $C_{equiv} = 1/(\sum_{i=1}^{n} 1/C_i)$.

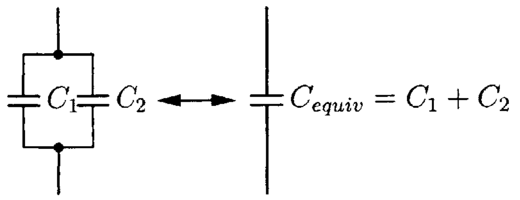

Figure 2-10: Two Capacitors in Parallel

Now consider Figure 2-10. By KVL, the voltages across all three capacitors shown must be the same,

$$v_{C_1} = v_{C_2} = v_{C_{equiv}}. \tag{2.35}$$

However, the currents flowing through each of the three capacitors will be different:

$$i_{C_1} = C_1 dv_{C_1}/dt \tag{2.36}$$

$$i_{C_2} = C_2 dv_{C_2}/dt \tag{2.37}$$

$$i_{C_{equiv}} = C_{equiv} dv_{C_{equiv}}/dt. \tag{2.38}$$

According to KCL, the total current through the two capacitors—which will be the current through the single equivalent capacitor—is equal to the sum of the individual currents,

$$i_{C_{equiv}} = i_{C_1} + i_{C_2}. \tag{2.39}$$

Substituting in the terms from Equations 2.36, 2.37, and 2.38, this becomes

$$C_{equiv} dv_{C_{equiv}}/dt = C_1 dv_{C_1}/dt + C_2 dv_{C_2}/dt. \tag{2.40}$$

Due to the equalities presented in Equation 2.35, both sides of Equation 2.40 can be divided by $dv_{C_{equiv}}/dt$ to obtain

$$C_{equiv} = C_1 + C_2. \tag{2.41}$$

It then generalizes that if n capacitors are placed in parallel, the equivalent capacitance will be $C_{equiv} = \sum_{i=1}^{n} C_i$.

2.1.6 Inductance and Inductors

An *inductor* is a circuit element which stores current. *Inductance* is measured in *Henries* (H), and is represented in equations by L. The relation between voltage and current for an inductor is given by

$$v_L = L di_L/dt, \tag{2.42}$$

where v_L is the voltage across the inductor as a function of time, di_L/dt is the rate of change of the current ($i_L(t)$) through the inductor, and L is the inductance.

Equation 2.42 and KVL and KCL can be used to determine the combinatorial rules for inductors. They happen to be the same as those for resistors: if inductors with values L_1 and L_2 are placed in series, the result is $L_{equiv} = L_1 + L_2$ (see Figure 2-11); if they are placed in parallel, it is $L_{equiv} = L_1L_2/(L_1 + L_2)$ (see Figure 2-12).

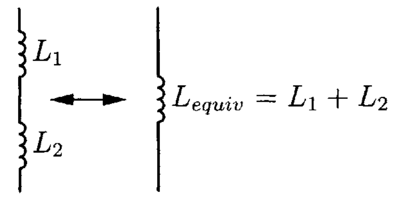

Figure 2-11: Two Inductors in Series

Consider Figure 2-11. As was the case previously with circuit elements in series, the overall current is equal to the individual currents,

$$i_{L_{equiv}} = i_{L_1} = i_{L_2}, \tag{2.43}$$

and the overall voltage is equal to the sum of the individual voltages,

$$v_{L_{equiv}} = v_{L_1} + v_{L_2}. \tag{2.44}$$

Each of these voltages can be found using Equation 2.42:

$$v_{L_{equiv}} = L_{equiv} di_{L_{equiv}}/dt \tag{2.45}$$

$$v_{L_1} = L_1 di_{L_1}/dt \tag{2.46}$$

$$v_{L_2} = L_2 di_{L_2}/dt \tag{2.47}$$

Substituting these terms into Equation 2.44, and noting the equalities of Equation 2.43, the equivalent inductance is found to be

$$L_{equiv} = L_1 + L_2. \tag{2.48}$$

For n inductors in series, the overall inductance will be $L_{equiv} = \sum_{i=1}^{n} L_i$.

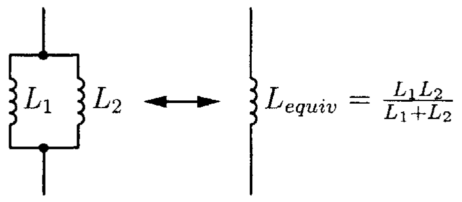

Figure 2-12: Two Inductors in Parallel

For inductors in parallel as in Figure 2-12, the voltages will all be the same,

$$v_{L_{equiv}} = v_{L_1} = v_{L_2}, \tag{2.49}$$

and the total current will be the sum of the individual currents,

$$i_{L_{equiv}} = i_{L_1} + i_{L_2}. \tag{2.50}$$

Taking the derivative of both sides of Equation 2.50,

$$di_{L_{equiv}}/dt = di_{L_1}/dt + di_{L_2}/dt. \tag{2.51}$$

Equation 2.42 can be written for each of the three inductors as

$$di_{L_{equiv}}/dt = v_{L_{equiv}}/L_{equiv} \tag{2.52}$$
$$di_{L_1}/dt = v_{L_1}/L_1 \tag{2.53}$$
$$di_{L_2}/dt = v_{L_2}/L_2 \tag{2.54}$$

Substituting each of these into Equation 2.51,

$$v_{L_{equiv}}/L_{equiv} = v_{L_1}/L_1 + v_{L_2}/L_2, \tag{2.55}$$

and since all the voltages are equal as shown in Equation 2.49, this simplifies to

$$1/L_{equiv} = 1/L_1 + 1/L_2. \tag{2.56}$$

This familiar form can be rewritten to solve for L_{equiv} as:

$$L_{equiv} = \frac{L_1 L_2}{L_1 + L_2}. \tag{2.57}$$

Additionally, if n inductors are placed in parallel, the equivalent inductance will be given by $L_{equiv} = 1/(\sum_{i=1}^{n} 1/L_i)$.

2.1.7 Diodes

A diode is a two-terminal circuit element[16] which only allows current to flow in one direction—from the anode to the cathode—and only if the voltage applied exceeds a minimum threshold, generally of about 0.6V for diodes made of silicon (see Figure 2-13). As long as the applied voltage exceeds the threshold, the diode will pass a virtually unlimited amount of current[17], so a resistor is typically placed in series with the diode to limit the current as per Ohm's Law.

A light-emitting diode, or LED, is a diode which emits photons of light when current flows through it. An LED, like other diodes, will begin to pass current at 0.6V, but at this voltage the amount of light emitted will be imperceptible. Usually about 1.6V is a good target voltage for a brightly shining LED.

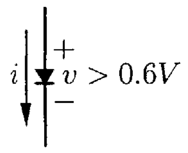

Figure 2-13: Diode

[16]Specifically, a diode is a *pn-junction*, i.e. the point of contact between a p-type semiconductor and an n-type semiconductor. The p-type or positive end is called an "anode", and the n-type or negative end is called a "cathode"; hence, a "diode" is a device with two (di-) "-odes". For more information on semiconductors, see Appendix D on page 157.

[17]The actual current-voltage relation for diodes is beyond the scope of this document. Suffice it to say that a higher voltage will be accompanied by a higher current.

2.1.8 Transistors

A transistor is a three-terminal semiconductor-based device which can be used as an amplifier or as a switch. It may be helpful to think of a transistor like a kind of electronic faucet. Rather than controlling the flow of water, a transistor controls the flow of electric current. In switch applications, the faucet is merely considered to be open or closed (on or off); in amplifier applications, what matters is *how open* the faucet is. This amount can be controlled to achieve a particular target flow. There are a number of types of transistors. In some, the control mechanism depends on an applied voltage; in others, on an applied current. This section will consider a particular class called the bipolar junction transistor, or BJT, which is current-controlled.

A bipolar junction transistor, as the name suggests, consists of two junctions, each of which is a pn-junction like the one in a diode. In an npn BJT, a p-type region is sandwiched between two n-type regions; in a pnp BJT, an n-type region is sandwiched between two p-type regions (see Figure 2-14).

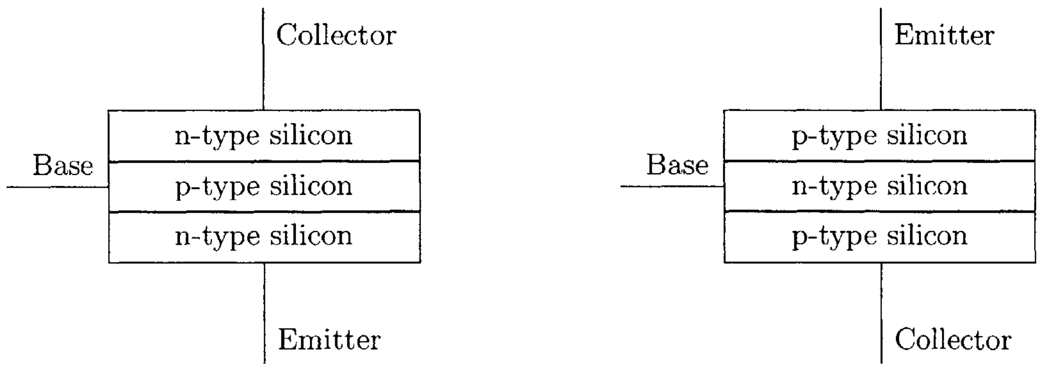

Figure 2-14: Bipolar Junction Transistors: npn (left) and pnp.

Each of the three terminals of a BJT (Collector, Base, and Emitter[18]) has a current associated with it (i_C, i_B, and i_E). For npn transistors, i_C and i_B flow into

[18]The designations "collector" and "emitter" refer to the movement of charge carriers through the silicon: in an npn BJT, the emitter emits electrons which are then collected by the collector; in a pnp BJT, the emitter emits holes which are then collected by the collector (In both cases, a small percentage of the charge carriers are actually collected at the base terminal, as indicated by Equation 2.59.).

the transistor, and i_E flows out of it; for pnp transistors, i_E flows into the transistor, and i_C and i_B flow out of it (see Figure 2-15). The collector current, i_C, is proportional to the base current, i_B, according to the equation[19]

$$i_C = i_B\beta, \tag{2.58}$$

where β is a large constant[20] (on the order of 100). Finally, the emitter current, i_E, is equal to the sum of the other two currents:

$$i_E = i_C + i_B. \tag{2.59}$$

This equality is necessary to satisfy KCL.

As was the case for a diode, a BJT—whose functionality is also based on the characteristics of the pn-junction—has certain minimum voltage requirements necessary for current to flow. For an npn transistor, v_{BE}—the voltage drop from the base to the emitter—must be at least about 0.6V, and $v_{CB} \geq 0$; for a pnp transistor, v_{EB}—the drop from the emitter to the base—must be at least 0.6V, and $v_{BC} \geq 0$. In other words, in Figure 2-15, the potential at each terminal must decrease going down (recall that current flows from higher potential to lower potential).

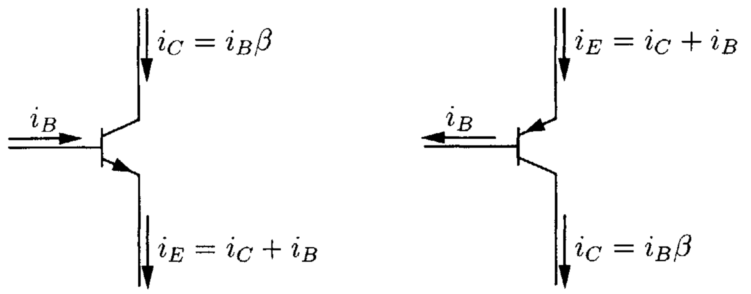

Figure 2-15: Circuit diagram BJTs—npn (left) and pnp—with currents labelled.

[19]Equation 2.58 will hold for small values of i_B. As i_B grows, i_C will be bounded from above by the amount of current which the power source can supply. Additionally, it will be limited by the device's ability to dissipate heat. In high-current applications transistors are often fitted with a *heat sink*, a piece of metal with a high surface area which helps to draw off heat and keep the transistor cool.

[20]The exact value of β varies from transistor to transistor, so a particular value should not be assumed.

In analog circuits, BJTs are useful as amplifiers ($i_E = i_B(\beta+1)$). Transistor radios made use of this property to boost the signal received by the antenna before sending it to the speaker. In digital circuits, they tend to be used as switches. When $i_B = 0$, the transistor is off and no current flows through it; when $i_B > 0$, the transistor is on and current flows.

2.1.9 Operational Amplifiers

An *operational amplifier* (op amp) is a differential amplifier with a very large voltage gain. It has two inputs, v_+ and v_-, and one output, v_{out}. The input-output relationship is described by

$$v_{out} = A(v_+ - v_-) \tag{2.60}$$

where A is the gain factor. A is large (on the order of 10^6), and is usually assumed (safely) to be infinity in calculations[21]. Feedback between the output and one of the inputs—i.e. external connections between the terminals through wire, resistors, capacitors, or other circuit elements—is generally used to provide some predictable (albeit smaller) gain, and to establish some desired input-output relationship.

For example, suppose an op amp is configured as in Figure 2-16 (the op amp's power rail connections are generally not shown). This circuit has a voltage divider between the op amp's output pin and ground, with the divider's output connected to the "−" input pin. According to Equation 2.24, the voltage at the "−" input, v_-, is $3v_{out}/5$. Combining this with Equation 2.60 yields $v_{out} = A(v_{in} - 3v_{out}/5)$. Solving for v_{out}, $v_{out} = Av_{in}/(1 + 3A/5)$. Because $A \gg 1$, the denominator can be approximated by $3A/5$, and the A cancels with the one in the numerator. The input-output relationship for the idealized device then becomes

$$v_{out} = 5v_{in}/3 \tag{2.61}$$

so if a 3V signal is applied to v_{in}, v_{out} will be 5V. It is important to note that an op

[21]As is the case with the β value of a BJT, A for a particular op amp is guaranteed to be large, but it varies from one chip to the next.

amp will only produce voltages up to the difference of its power rails; if the op amp is powered by 5V and v_{in} is 4.5V, v_{out} will be 5V, not 7.5V.

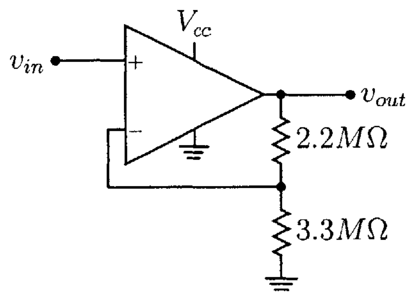

Figure 2-16: An op amp configured to be a voltage multiplier.

2.2 Microcontrollers and Their Parts

The circuit elements introduced thus far are all useful for constructing analog circuitry to interact with the real world. For instance, a number of sensors function as variable resistors, with their resistances dependent upon environmental factors such as light (in the case of a photoresistor) or heat (in the case of a thermistor). These can be used as R_1 in a voltage divider with a constant v_{in} to create a circuit which produces an output voltage that depends on the relevant environmental condition. In other cases, a sensor may produce an analog voltage which is simply too small to deal with directly, so it must be amplified by an op amp before being measured. Now that these circuit elements have been introduced, the "brain" of a WSN mote will be discussed.

As with many other embedded devices—electronic systems tailored to particular applications—WSN motes are typically controlled by a microprocessor (CPU) which executes a program stored onboard. The microprocessor is often found as part of a *microcontroller.* In addition to the processor, a microcontroller contains memory and other peripheral components all integrated into a single chip. By contrast, these components are often separated in a desktop PC so that they can be used interchangeably with a variety of different microprocessors; in an embedded device, they are merged and the combined chip is typically optimized for low power consumption or some other desired trait. In this section, the architecture of a typical microcontroller is presented. Specific examples are drawn from the Atmel ATmega128 (see Figure 2-17), which is the microcontroller at the heart of the JONA system.

2.2.1 CPU

Each microcontroller is built around a *central processing unit* (CPU; see Figure 2-18) which executes the individual instructions making up the program which controls the actions of the embedded system. The set of instructions which a particular CPU is capable of executing is determined by its *Instruction Set Architecture* (ISA)[22]. There are a number of different instruction sets used by a number of different microprocessor families; fortunately, it is generally possible to write programs in a "portable" language like C [27] which can then be translated by a compiler into machine instructions appropriate to various CPUs[23].

Data Register File

Registers are storage containers which can hold chunks of information. In microcontrollers, registers typically hold one byte (eight bits) each, but sixteen-bit registers are also common. More powerful microprocessors often have registers of thirty-two or sixty-four bits, or more. The *data register file* is a set of registers which hold data which is being processed. The ATmega128's data file contains thirty-two 8-bit general purpose registers.

Instruction Register

The *instruction register*, as its name implies, is used to store instructions. Whereas the ATmega128 has a number of registers in its data register file, it only has a single register to hold instructions. It uses this register to hold the next instruction which it must perform. While a particular instruction is being executed, the subsequent

[22]The ATmega128 uses an ISA called "AVR" [6] which is also used by a number of other microcontrollers made by Atmel.

[23]This is fortunate in that if it is later decided to switch to a different microcontroller, the software already written need not be rewritten using the new instruction set, but only recompiled for the new architecture.

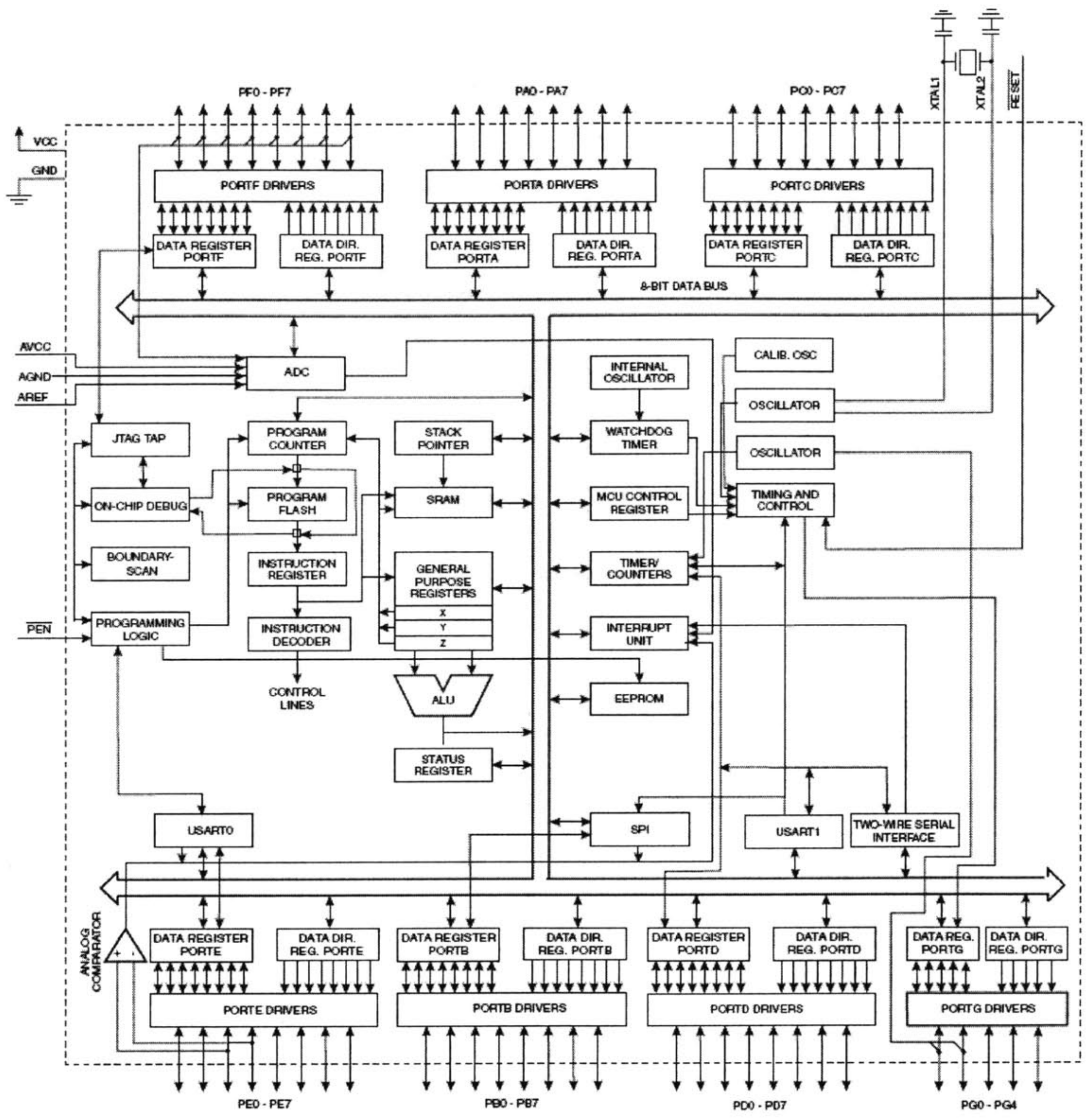

Figure 2-17: Block diagram of the ATmega128 [5].

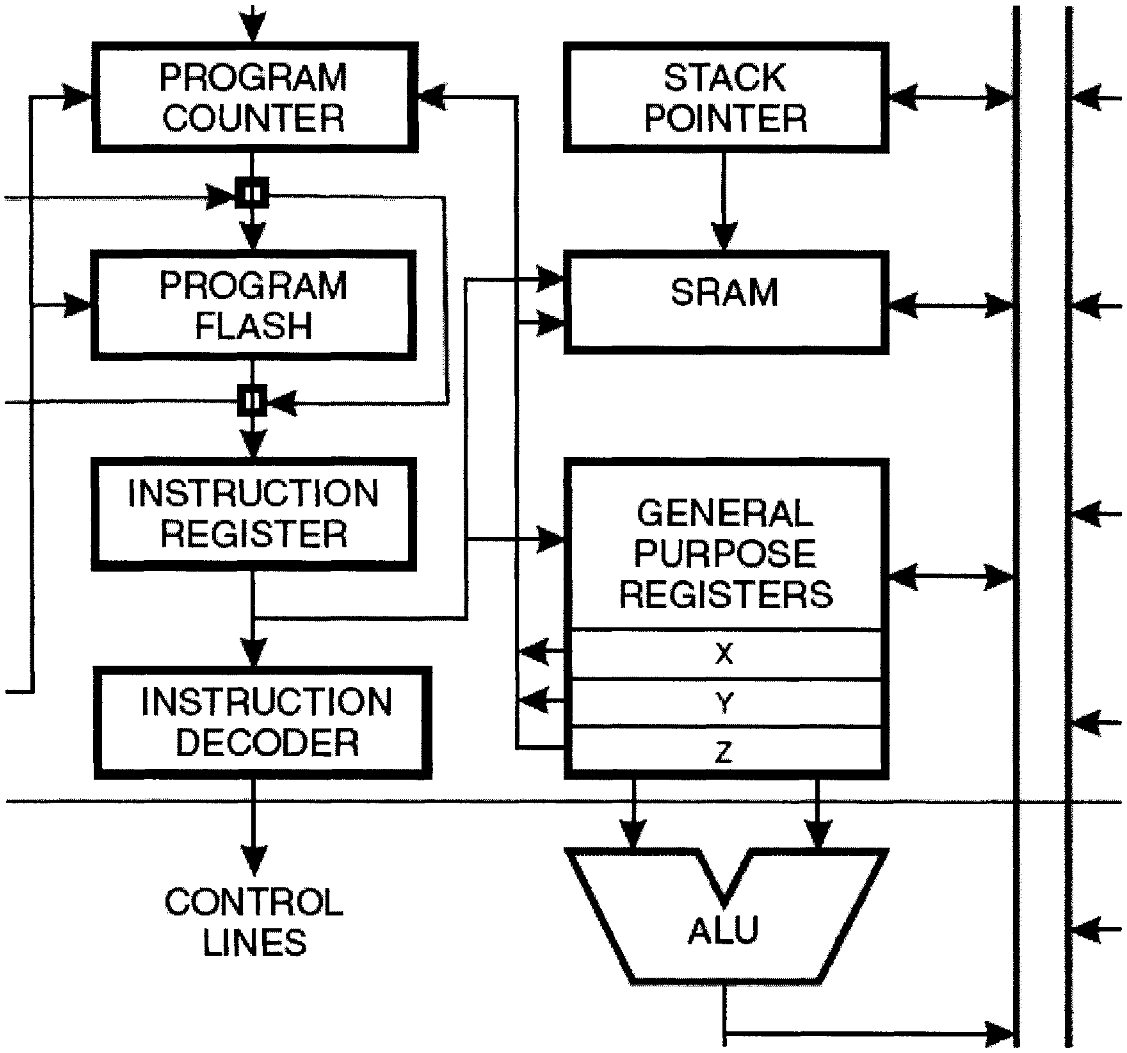

Figure 2-18: Closeup of the ATmega128 CPU, showing the ALU and registers [5].

instruction to be executed will be read into the register. This is known as *prefetching* because the instruction is "fetched" from program memory one cycle ahead of when it is to be executed.

Arithmetic Logic Unit (ALU)

The *arithmetic logic unit* (ALU) is the component of the CPU which performs arithmetic (and other) operations on data. It is capable of adding, subtracting, multiplying, and dividing, as well as determining whether one number equals or is greater than or less than another number.

Program Execution

To execute a particular instruction, the CPU performs the following sequence of steps:

- First, the CPU reads the prefetched instruction from the instruction register and decodes it.
- If it is a memory access instruction, then the CPU copies the contents of the indicated SRAM memory address into the indicated data register (for a read) or copies the data register's contents into the memory location (for a write).
- Otherwise, the instruction will typically tell the CPU to perform some operation on two inputs and produce an output. So, the CPU configures the ALU to perform the correct operation, feeds it the appropriate inputs from the data register file, and then directs the output to the correct data register.
- Meanwhile, the CPU will prefetch the next program instruction from the program flash memory into the instruction register to prepare for the next loop.

2.2.2 I/O Ports and Parallel Communications

To communicate with the outside world, a microcontroller has metal *pins* protruding from its packaging. Some are used to connect it to the power and ground rails and a crystal oscillator , while others are used to connect it to other circuit elements to provide input and output (I/O) capabilities. The microcontroller can send bits by setting voltage levels on pins to ground (for 0) or V_{CC} (for 1); it can receive bits by having other devices set these voltage levels, and then testing the pins to see what they were set to.

Some or all of the I/O pins may be split into groups called *ports* (indicated by the groupings along the top and bottom of Figure 2-17) which can send and receive multiple bits of data in *parallel*. Each bit is sent on one of the pins. Typically, a port has eight pins, so that it can send or receive one byte at a time. It is not necessary to use all eight pins, though; each pin in a port can be individually read from or written to.

Internally, each port has associated with it registers for input and output. When the CPU writes a byte to the output register, the register sets the voltage levels on each of the corresponding pins to the appropriate rail voltage to output the byte to anything which might be connected to the port. When the CPU reads from the input register, it translates the voltage levels on the pins—which are presumed to be set correctly by some other device[24]—into a bit/byte of data.

Some microcontrollers, including the ATmega128, have a third register associated with each port, called a Data Direction Register (DDR). This register is used to set which pins are inputs which pins are outputs. Before writing to a pin, its corresponding DDR bit must be set to 1 to put it in output mode; before reading from the pin, its DDR bit must be set to 0 for input mode. If the DDR is not properly set, then the value read from or written to the port may not be what is seen on the pins. For example, if `DDRA` (the DDR for Port A) is holding the value 0xF0—-i.e. 0b11110000—-and the value 0xCC is written to Port A, then the output could be 0xCC, but it may be anything in 0xC0–0xCF. The upper four bits (sometimes called the high *nibble*) are setting their pins properly because they are in output mode, but the lower four are in input mode, so their values are either being set externally or they are floating.

Using an I/O port for communication is fast; in a single clock cycle, 8 bits of data

[24]If an input pin is not connected to anything, it is said to be *floating*, and its voltage is not guaranteed to be at a valid logic level; it may read as a 0 or as a 1.

can be transmitted or received. However, this parallel communication is expensive because it takes up a lot of space and wire. Two devices communicating in parallel need to have at least 8 wires connecting them[25].

2.2.3 Serial Communications Options

It is often the case that the speed of parallel communication is not deemed to be worth the cost of space and materials required. For instance, if the microcontroller needs to send a 32-bit number, it would have to devote four ports to the task in order to send it all at once. It would often be more reasonable to send it in small pieces, one after the other, using fewer wires (albeit requiring more time) to do so. This is known (especially when one bit is sent at a time) as *serial communication.* There are three widespread protocols for serial communication. These are I2C, SPI, and RS-232. Most microcontrollers implement these protocols, though not all feature both I2C and SPI; all three are supported by the ATmega128.

Inter Integrated Circuit

Inter Integrated Circuit (I2C) is a serial communications protocol designed for sending data between devices which are typically located on the same circuit board, all being powered by the same V_{CC} and sharing a common ground rail. It is a *multimaster bus,* which means that more than two devices may be connected to the same I2C wires, and that any of the attached devices may initiate a transmission (i.e. be master).

The I2C bus consists of two wires, SDA and SCL, which are both *bidirectional*—the same wire is used for sending and receiving. SDA is the serial data line, and SCL is the serial clock line. As a transmitting device sends bits on SDA, it sends pulses on SCL so that the listeners know when to read each bit. Because the bus may have

[25]There will usually be a few other wires in addition to the eight data wires acting as control lines: for example, there might be one to signal that a device is ready to transmit, another to signal that it is ready to receive, etc. There may also be address lines to redirect data amongst several chips connected to a single port, or amongst several registers within a single chip (This is how data is transferred in a desktop computer between a CPU and one of the millions of storage locations of a memory chip.). These control/address lines will typically be connected to I/O pins of another port of the microcontroller.

several devices attached to it, the transmitter must be able to indicate which device is the intended recipient of its data. So, each device on the I2C bus has a 7-bit address, and the transmitter sends the address on SDA before sending any data. Each other device on the bus listens to the address to determine whether it should listen to the rest of the transmission. This is essentially the same mechanism used in telephony; to call a particular phone number, special sounds corresponding to each digit are sent on the voice line, and then when someone answers the phone, the caller proceeds to send data (i.e. talk) on that same voice wire.

Serial Peripheral Interface

Serial Peripheral Interface (SPI) is another protocol used between devices on a circuit board. Unlike I2C, SPI is not multimaster; one device is the master, and the others are slaves or peripherals. The master is the only device which may initiate communication.

SPI uses four wires: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK (Serial Clock), and $\overline{\text{CS}}$ (Chip Select). Each of these is *unidirectional.* The master first selects a peripheral device by setting its $\overline{\text{CS}}$ line *low*[26]. It then sends eight pulses on the SCK line. During each pulse, the master sends one bit on MOSI and the slave sends one bit on MISO; after the eight pulses, the two devices have each sent the other one byte.

This *full-duplex* communication, compared to I2C's *half-duplex* communication (in which data is only transmitted in one direction at a time), along with SPI not needing to transmit any addresses, lets SPI achieve significantly higher data transmission rates. The tradeoff is that the protocol requires more wires than I2C (but still far less than would be needed for parallel communication) and that each peripheral may only communicate with the master device (whereas with I2C any device on the bus may send data directly to any other device). For more detailed discussions of I2C, SPI, and how the two compare, refer to [9].

[26]The bar over the letters in "$\overline{\text{CS}}$" indicates that it utilizes *negative logic*, i.e. a low voltage (logical 0) enables the device while a high voltage (logical 1) disables it.

RS-232, the UART, and the Serial Port

The third serial communications protocol, *RS-232*, was designed for communication between a DTE (Data Terminal Equipment) and a DCE (Data Communication Equipment). Unlike the communicating devices in I2C and SPI, the DTE and DCE are intended to be physically separated and connected by a *serial cable* which plugs into each device's *serial port*. The DTE and DCE have often taken the forms of a computer and a modem, respectively, but the protocol can be used to connect many types of equipment. Also in contrast to the other two protocols, RS-232 is not a bus; it only supports the presence of one DTE and one DCE. To simplify RS-232 communication, the serial port of a device is typically controlled by a component called the UART (Universal Asynchronous Receiver-Transmitter). This converts between parallel and serial so that other parts of the device (such as a CPU) may deal with the data as, for instance, a byte in parallel, rather than having to handle each bit individually.

RS-232, like SPI, is full-duplex. It has a wire for sending bits and another wire for receiving bits (simultaneously). Its other wiring differs somewhat from what is found in I2C or SPI, mostly because it is intended to connect two pieces of equipment, rather than two chips in one piece of equipment. For example, there is a ground wire, to ensure that voltages sent from the two ends share a common reference point[27]. There are also lines to signal that a device has data it wishes to transmit, and to signal that a device is prepared to receive data. One wire RS-232 does *not* have is a clock signal. Instead, both devices are configured ahead of time for a particular transmission rate (usually denoted the "baud rate")—typically several thousand bits per second—and then during actual communication the receiver is responsible for reading the incoming bits at the appropriate rate.

Usually, the DTE is a computer and the DCE is some peripheral component, like a modem. However, it is possible to connect a DTE to another DTE (or a DCE to another DCE) using a *null modem adapter* which must be inserted between the serial

[27]I2C and SPI do not explicitly have ground wires as part of their protocols, because it is assumed that the communicating devices already share common power rails.

cable and the serial port (on one end or the other). This adapter rearranges the wires to make sure that they all end up going to the right places. Another option is to use a *crossover cable*, which is a special serial cable that makes these rearrangements internally.

2.2.4 The Analog to Digital Converter (ADC)

Sensor devices often function by translating some physical property into an analog voltage. For instance, an accelerometer or tilt-meter might produce an output of 2.5V when it is sitting flat; its output may then continuously move lower toward 2.0V as it is tilted one way, and higher toward 3.0V as it is tilted the other way. Or, a microphone may produce a voltage signal whose amplitude is proportional to that of the incoming sound wave. For a digital device to make sense of this data, it must convert the analog signal into a digital one. This is the job of the *Analog to Digital Converter* (ADC).

An n-bit ADC converts any voltage in its input range—typically between 0V and some configurable upper limit AV_{CC}—into a digital number between 0 and $2^n - 1$. As there are an infinite number of voltages in the range [0, AV_{CC}] but only 2^n possible outputs, there is some loss (quantization error) associated with the conversion. Voltages very near each other will likely map to the same digital value; other programs or devices which read this value will then be unable to distinguish between the various inputs which could have produced the value.

An n-bit ADC with range [0, $AV_{CC}/2$] will have twice the resolution of an n-bit ADC with range [0, AV_{CC}]. So, the resolution of an ADC reading can be improved by reducing the input range. However, there may be cases in which reducing the input range is not an option. The input signal may be expected to vary from one end of the range to the other, perhaps in some extreme situation which must be detected. For example, an accelerometer may typically produce outputs between 2.3 and 2.7 Volts, but swing all the way from 0 to 5 during an earthquake. In this case, increasing resolution by decreasing range is not an option; if a higher resolution is required, a higher-bit, higher-cost (and slower) ADC is the only option.

2.3 Common Laboratory Equipment

This section introduces a couple of useful tools which can help with debugging or otherwise monitoring an electronics project such as a WSN prototype.

2.3.1 Multimeter

A *multimeter*, as its name suggests, is a device which can measure multiple electrical properties. In particular, multimeters are able to measure voltage (as a voltmeter), current (as an ammeter) and resistance (as an ohmmeter). Additionally, many have the ability to test capacitance and inductance, as well as several other parameters.

Multimeters have two *probes*—one red for positive, the other black for negative—which are used to take measurements, and three (or more) sockets into which they may be plugged. The black probe plugs into the ground socket (often labelled "COM" for Common). The red probe plugs into one of the other sockets; which one depends on what is being measured. Usually, one of the remaining sockets is used for measuring current, and the other is used to measure voltage (and most other quantities). At the end of each probe is a rigid metal tip or (removable) clip.

To measure current, the multimeter must be inserted in *series* with the circuit so that all current is forced to flow through it. For the reading to be meaningful, the effect of the addition of the multimeter into the circuit must be minimized. In particular, this means that the multimeter must have a very low resistance. The ammeter socket (typically labelled with a unit of current like "A" or "mA") is, therefore, set up to allow substantial currents with negligible internal resistance. Current flowing from the red probe to the black probe will be measured as being positive.

To measure voltage, the multimeter must be placed across a circuit, in *parallel* with it. The voltmeter socket (indicated by a "V") is configured to have very high resistance between itself and COM, and it is typically unable to support very large currents.

Another common feature, which can be quite useful, is a setting to test *continuity*. In this mode, the multimeter will beep if there is no resistance between the two probe tips (i.e. if the two points being tested are connected only by metal/wire). This can be used to detect short circuits (if it beeps when it should not) or breaks in the circuit.

2.3.2 Oscilloscope

An *oscilloscope* is an instrument which displays a graph of one or more potential differences as functions of time. The display repeatedly updates from left to right, and so oscilloscopes are particularly useful for observing periodic signals. There is a probe for each input channel. At the end of the probe there is an alligator clip for the ground wire and a main clip to connect to the point of interest[28].

There are knobs to adjust the scales and offsets of both the vertical axis (voltage) and the horizontal axis (time) to zoom in on a particular segment of a waveform. Each of the input channels shares a common time axis, but their voltage scales and offsets can be set independently.

To make it easier to find a particular feature in a signal, the oscilloscope can be triggered from of one of its channels. This synchronizes the displayed waveform so that it is phase locked with this signal. When the trigger function is on, the scope will wait for the trigger event—typically, the waveform crossing a specified threshold from below (in a rising-edge trigger) or above (in a falling-edge trigger)—to occur before redrawing the display. The trigger event will then be centered on the display (or offset as desired).

2.4 Programming Languages & Source Code

The lowest-level programming language is called assembly. Each assembly instruction directly corresponds to an instruction which the microprocessor can execute; assembly instructions are, in fact, just brief textual descriptions of the underlying machine

[28]The probe's ground wire is directly connected to that of the electrical outlet through the oscilloscope. So, if the device being observed is also plugged into an outlet, then it is important that the alligator clip only be connected to the circuit's ground; otherwise the clip will introduce a short circuit.

instructions. As mentioned in Section 2.2.1, a CPU will implement one of many different Instruction Sets. Assembly is not a single language but rather a collection of dialects, one for each architecture.

At a higher level are many other programming languages like C [27], C++ [45], and Java [3]. A program written in one of these is more portable—it is not tied to a particular ISA—because other people have written compilers which will translate it into instructions appropriate to many different processors. Furthermore, they allow the programmer to use constructs like *variables* and complex data structures rather than being limited to working physical memory addresses and register contents. In short, they provide an *abstraction layer* between the program and the hardware on which it will run.

With languages like C, it is often possible to make use of the abstractions when they are convenient and yet still be able to directly access the hardware if need be. This limits the portability of the program—the parts which directly access hardware must be changed when the hardware changes—but it can make it much more powerful than it would be if the abstraction barrier were impenetrable.

2.4.1 Open Source Software

When a program is to be distributed to others, the programmer may choose to make available the source code or just a pre-compiled executable file. Companies typically release only the executable, so that they can maintain control over the program. However, some companies and organizations advocate releasing software as *open source*, whereby the source code is also released. Users of the program are then allowed[29] to modify it if they find a bug or want to add a new feature. Academic software, such as TinyOS, is released as open source because the authors did not intend it to be a source of revenue but rather as a platform for academic research.

[29]Cf. Appendix K.

Chapter 3

Motivation and Design Goals

3.1 Classroom Needs

With all the research going on, and the commercial options available[1], it may seem strange that another set of WSN hardware has been developed for academic research. The kit described herein is not intended to compete with these other options; rather, it is intended to complement them by opening up the research to the computer science laity: to students who have little or no formal education in electronics; students of civil engineering, environmental engineering, biology, and other fields.

It is toward applications in fields such as these that WSN promises so much, and so this kit has been designed to provide these students with an introduction to WSN technology and to demonstrate what this technology can do for them. This way, as EECS researchers at MIT's MTL and LIDS laboratories and elsewhere continue to make technological breakthroughs, their future customers can at least know enough of the basics to understand how those breakthroughs might be significant[2].

[1]For instance, Crossbow offers a MICA2 Classroom Kit [14].

[2]Of course, it is entirely possible that students learning about WSN with the JONA kit could have breakthroughs of their own!

3.1.1 1.961/CEM508

The JONA kit is predated by the class for which it was developed. Earlier incarnations of the course at MIT had used commercially available motes, but students encountered multiple frustrations—in both hardware and software—which seemed to limit these motes' usefulness.

The problems were typically caused or exacerbated by design choices which the mote creators had made—these motes were typically designed for commercial deployment, not for prototyping and educational lab work. So, it was decided that a new kit should be developed which would address these issues. The following two sections will detail the main issues that were faced when using the commercial motes and the design requirements which they implied for the JONA prototyping kit.

3.1.2 Software Requirements

One requirement of the JONA software was that it should have some basis in pre-existing WSN software. The JONA kit was being developed to introduce students to WSN technology, but it was not expected to be the final commercial solution that they would later use in their jobs; so, what they learned using JONA had to be relevant to other systems out there.

Additionally, the software should provide complete access to the hardware on which it was running. One of the motes that was used previously had pre-installed software which could not be circumvented. Among other things, this software was responsible for controlling the built-in ADC. When taking measurements, this software would take a fixed number of samplings over the course of a second, and then it would compute the average and provide the user with this single number. The user was given no provision for accessing the data used to compute the mean, or for changing the sampling rate.

In some cases, this software limitation is not a major concern. For instance, if the motes were used to record temperature, the built-in filtering would not matter, because temperature will not vary significantly over the course of a second. However, civil engineers might be interested in things like vibrations due to seismic activity. So, students in the class connected an accelerometer to the ADC, to see how the mote would respond to a simulated earthquake. The software controlling the ADC prevented them from seeing that the mote had been shaken at all[3].

The chief software requirement, therefore, was that the user should be able to control the hardware directly to obtain the data as he or she saw fit. If the situation called for onboard statistical analysis, then the mote could be programmed to perform it; otherwise, it would be able to transmit the raw data back to the user.

Berkeley's TinyOS seemed to satisfy these software requirements well. The class had encountered it through motes from Crossbow. However, the Crossbow hardware was less flexible and somewhat fragile in early commercial offerings. More recent versions have addressed these issues, but it was still decided that a new hardware kit should be developed.

3.1.3 Hardware Requirements

In the setting of a class examining wireless sensor networking, the WSN nodes must, above all, facilitate the investigation process. Commercial mote hardware, however, is designed for deployment; when a set of set of nodes is ready to be deployed, the nodes have already gone through the prototyping phase and have been specialized and miniaturized. So, these two design goals are not merely divergent; in fact, they are antithetical. Hardware designs appropriate for final versions can hinder the study of alternative options.

Certainly, the software of a production model can be modified and experimented on, but by this point the choices which went into the design of the hardware have been set in substrate, so to speak. The Crossbow motes, for example, were designed to be low-power devices which are as compact as possible. They run on a pair of AA

[3]Suppose the data is plotted on a graph so that 0 indicates no acceleration, and positive and negative numbers indicate acceleration as the building sways to the right and to the left. Then suppose that the data for a particular second is −300, −200, 0, 200, 300, 200, 0, −200. This clearly indicates some sort of back-and-forth movement, but the average value over the whole set of data points is 0!

batteries, and the circuit board—tightly packed with components—is only slightly larger in area than the two batteries side-by-side.

The circuit board, which sits atop the battery holder, has parts on both sides of it; the parts on the bottom side are inaccessible. This is no problem for a mote which is being deployed, because no one will be planning to access those parts. In a classroom investigation, however, all the components should be exposed so that they can be examined with the oscilloscope, or even be replaced by other parts. Additionally, in a choice between accessibility and size, accessibility should win. A prototyping kit does not need to be as small as physically possible—especially when this miniaturization makes investigation more difficult.

The Crossbow motes have an expansion plug to connect secondary boards containing sensors, and there are several sensorboard configurations available on their website. It is also possible to design and have made custom boards. However, designing and having manufactured a new circuit board is by no means a fast process. In a class spanning a single semester, time is of the essence, so laboratory equipment must enable new prototyping ideas to be implemented and tested quickly. Also, the expansion plug only exposes a subset of the microcontroller's ports and pins. The rest remain untouchable (except, perhaps, by some very careful soldering). It would be better if all of the microcontroller were easily accessible without having to resort to skillful modifications.

Another important feature a classroom mote must possess is good input/output. The Crossbow motes have three LEDs which can be used, for example, as status indicators or to provide a visual representation of data which has been collected or received. This was considered to be a step in the right direction, and so it was decided that JONA should also have LEDs—it should just have more of them. In final deployments they should not be used, since they consume a significant amount of power; but during the prototyping stage, they can provide valuable insight into what the mote is doing.

Even more importantly, it should be simple and convenient to connect JONA motes to computers, both for reprogramming and for relaying data once deployed.

This was generally not found in commercial motes, which often require other hardware in order to interface with a PC. In contrast, it was decided that each JONA mote should have its own serial port for easy RS-232 access.

Finally, the JONA hardware should satisfy the above requirements and cost less than the commercial options. The disadvantages of the commercial motes will seem far less significant if the alternative is more expensive.

3.2 Design Goals

In summary, the JONA kit had the following design goals:

- Be a low-cost prototyping kit.
- Be relevant to other work, in terms of both hardware and software, so that someone learning about WSN via JONA could go on to apply that knowledge to other WSN products.
- Have software and hardware which facilitates rather than hinders the exploration of alternative configurations. Software should allow direct, easy access to the hardware (programmatically speaking), and hardware should be somewhat oversized and allow direct, easy access to all microcontroller pins and peripheral components (physically speaking).
- Be programmable without requiring additional hardware (other than a connecting cable).
- Make prodigious—but not gratuitous—use of that most-important invention of the solid state era, the blinky light.

Chapter 4

Implementation of the JONA Prototyping Kit

The first decision made regarding the implementation of the JONA kit was that it should run TinyOS for its software. TinyOS had shown great promise with the Crossbow MICA motes, and it was decided that adapting the operating system to the new platform—which was allowed by its open source license agreement—would be the ideal way to equip JONA with a fully capable software environment without having to build one from scratch. This decision prompted an additional hardware requirement: the JONA hardware should readily support TinyOS, with only a minimal set of changes or extensions to the OS needed.

To this end, it was decided that the JONA kit should be built around an Atmel AVR microcontroller. MICA uses an Atmel chip, and so it had already been well established that TinyOS would work with this hardware. Similarly, the radio transceiver would be[1] the TR1000 by RF Monolithics [42]—the radio found in the MICA.

[1]The Chipcon CC1000, found in the MICA2, was also considered, but the TR1000 proved to have a simpler wiring solution. Both transceivers run at 3V, so incoming and outgoing signals must be translated between 3 and 5V levels; all TR1000 lines are unidirectional, meaning that the conversions could be performed by resistor dividers and op amps. The CC1000, on the other hand, has bidirectional lines, which would necessitate solutions less elementary in nature.

4.1 The JONA Hardware Platform

4.1.1 The PROBOmega128 Prototyping Board

After searching for and comparing a number of different Atmel prototyping board options, the PROBOmega128 board (see Figure 4-1) developed by Dr. Erik Lins [30] was chosen to be the basis for the JONA hardware. It provides convenient access to all pins of the microcontroller, it has a built-in serial port to provide RS-232 access, it is vendor supported, and it is cost effective.

At the heart of the PROBOmega128 is an Atmel ATmega128 AVR microcontroller (located on the underside of the board; see Figure 4-2), the 5V version of the low-power (3.3V) ATmega128L found in Crossbow's MICA2 motes. It runs at a speed of 14.7456 MHz[2]. The board can be powered by a voltage between 7 and 25 Volts applied to a barrel connector on the side (from a standard AC adapter, for instance). A voltage regulator then takes this input voltage and supplies the rest of the board with a steady 5V. In Figure 4-1, the barrel connector is on the far side of the board, with the voltage regulator to its right. The serial port is on its left edge.

Nearly all of the ATmega128's sixty-four pins—all, in fact, but five, four of which are connected to two crystal oscillators—are directly connected to header pins—large pins spaced one tenth of an inch apart which allow for quick connections to other components via wirewrapping or ribbon cables—grouped by logical function[3]. Counter-clockwise from the bottom in Figure 4-1 (see also Figure 4-3), these header pin groupings are `CON_ADC`, `CON_BUS`, `CON_PIO`, `CON_ISP`, and the two-pin `J1` jumper connector in the middle, which can be used to reset the microcontroller.

`CON_ADC` provides access to the ATmega128's built-in 10-bit analog-to-digital converter. The ADC has eight input channels, provided as the alternate functionality of the eight pins of I/O Port F. The ATmega128 can, therefore, be wired to eight separate analog voltage sources at the same time. `CON_BUS` contains data, address and control lines which can be used to connect external memory or other such de-

[2]The MICA2's ATmega128L, being lower-powered, only runs at a clock speed of 8 MHz.

[3]These connections can be seen in the schematic in Figure E-1 in Appendix E as the parallel lines running outward from the square-shaped microcontroller.

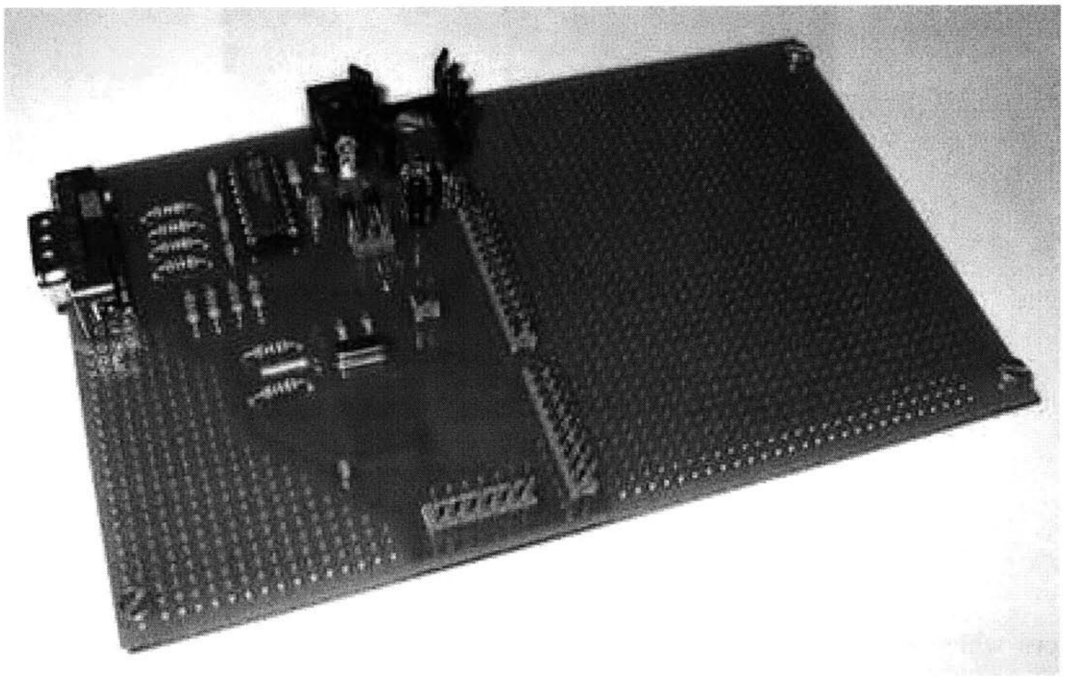
Figure 4-1: PROBOmega128 (Top side) [30].

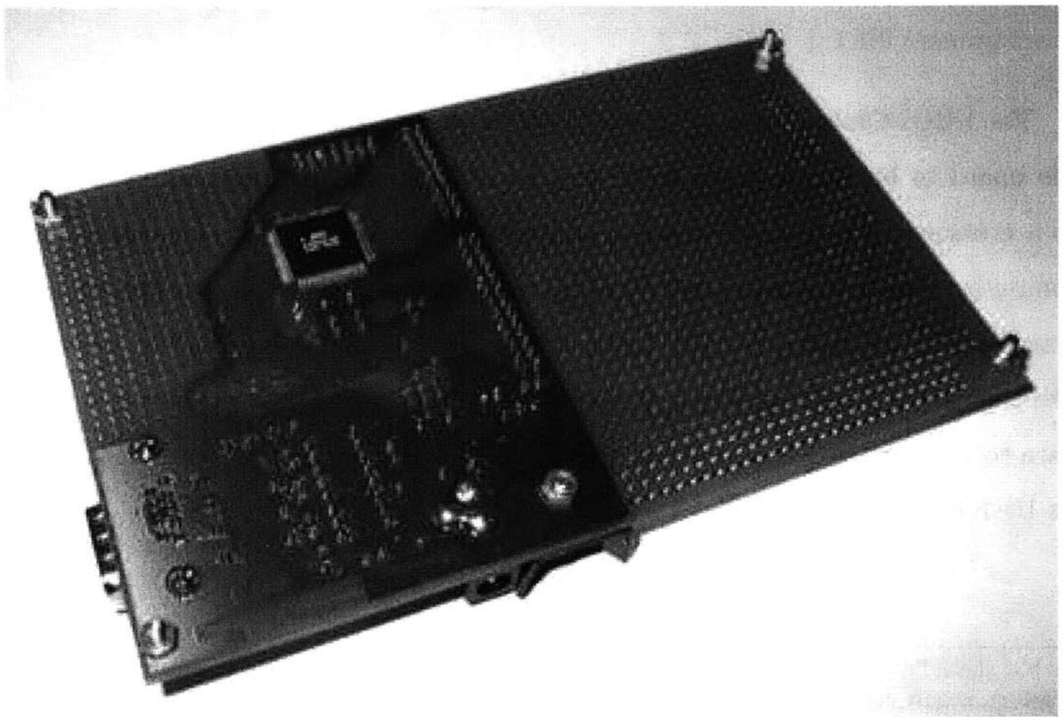
Figure 4-2: PROBOmega128 (Bottom side) [30].

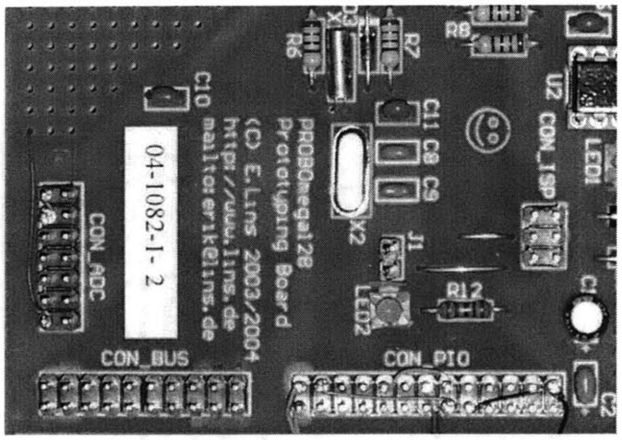

Figure 4-3: Header pin groupings.

vices which make use of an address and parallel communication. These lines can also be used as generic I/O, as Ports A, C, and G[4]. CON_PIO has I/O Ports B, D, and E. CON_ISP contains six pins—including the SPI interface's MISO, MOSI and SCK lines—used to reprogram the microcontroller by a device called an In-System Programmer (ISP).

The PROBOmega128 also has a built-in serial port (see Figure 4-4) which enables the board to be programmed even without an ISP. The board is wired as a DTE, so a crossover cable or null modem adapter must be used when connecting it to a computer[5]. Once a JONA mote has been programmed, its serial port can be used to transfer information between it and a computer[6]. A mote connected to a computer like this is often used as a *base station* in a WSN network. Other motes send their data to the base station via radio, and the base station then sends the data through its UART to a computer.

[4]Of the ATmega128's seven I/O ports, Ports A–F have eight pins each, numbered 0–7. Port G, however, is only composed of five pins, two of which are connected to a crystal oscillator which drives an internal timer. CON_BUS has header pins connected to the other three pins of Port G, namely PG0, PG1 and PG2.

[5]See Section 2.2.3 on page 57.

[6]See Section 4.2.3 on page 82.

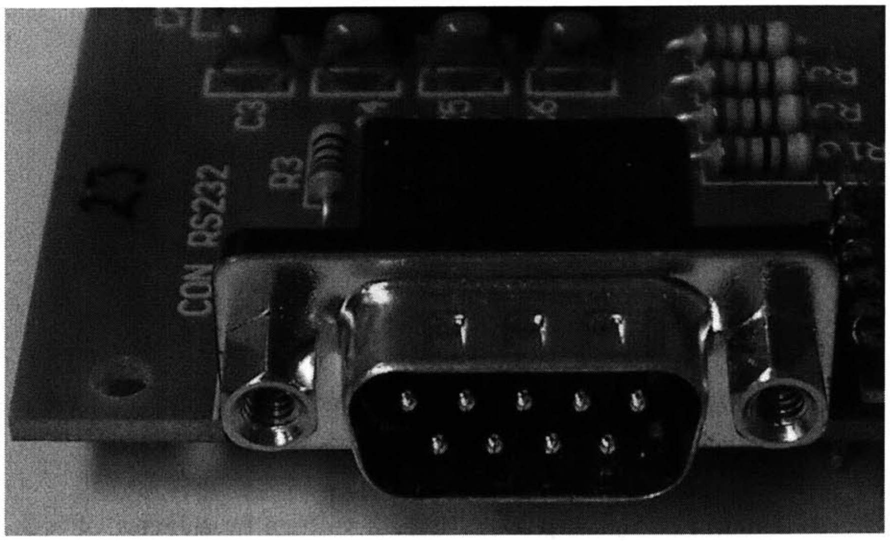

Figure 4-4: The serial port, labelled "CON_RS232".

4.1.2 The DR3000-1 Transceiver Module

For radio communications, the JONA kit is equipped with the DR3000-1 916.50 MHz transceiver module (see Figure 4-5) from RF Monolithics [41]. This is a prototyping module which features RFM's TR1000 transceiver integrated with the necessary resistors, capacitors, and inductors on a small printed circuit board (PCB). A piece of wire about four inches long serves as an antenna. The rest of the module is wired to the PROBOmega128 as shown in the schematic in Appendix F on page 163 (see Figures 4-6, 4-7, and 4-8). There are two control lines, CNTRL0 and CNTRL1, and a transmit line, TXMOD, from the microcontroller to the transceiver, and a receive line, RXDATA, from the transceiver to the microcontroller.

Because the TR1000 is designed to be a low-power device, the DR3000-1 requires a 3V V_{CC}, rather than the 5V used by the rest of the JONA (it will not operate at 5V). To accommodate this requirement, the radio module has its own voltage regulator to obtain 3V power from the board's 5V power line. Additionally, the control signals from the microcontroller to the radio chip must go through a voltage divider[7] to reduce their voltages by a factor of (approximately) 3/5, and the RXDATA output

[7]See Section 2.1.4 on page 38.

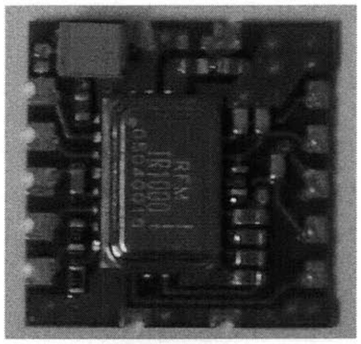
Figure 4-5: The DR3000-1 transceiver module.

signal must go through an op amp[8] to boost its voltage by a factor of 5/3.

The TXMOD does not need to have its voltage scaled down because the TR1000 looks at the *current* coming in on that pin, rather than the voltage. It uses this current measurement to set its transmission power. Putting a resistor in series on the TXMOD line will reduce the transmission power, which will reduce the mote's transmission radius. By using a *potentiometer*—an adjustable resistor—the transmission radius can be tuned to be as small as possible—thereby saving energy—while remaining large enough to reach the next mote. MICA uses a software-controlled digital potentiometer to do just that. For the JONA kit, however, it was decided to keep the wiring simpler; a reduced or adjustable transmission radius is not necessary to learn the basics of RF communication in wireless sensor networks. At the same time, interested students can certainly add resistors or potentiometers to evaluate their effects.

Caveat Lector

Notice from Table F.1 on page 164 that the send line, TXMOD, is connected to SPI pin MISO (Master In Slave Out) and that the receive line, RXDATA, is connected to SPI pin MOSI (Master Out Slave In). Additionally, the microcontroller's SPI chip select pin, $\overline{\text{SS}}$, is pulled low. These three observations would imply that the microcontroller is acting as the slave with the DR3000-1 as master.

[8]See Section 2.1.9 on page 48.

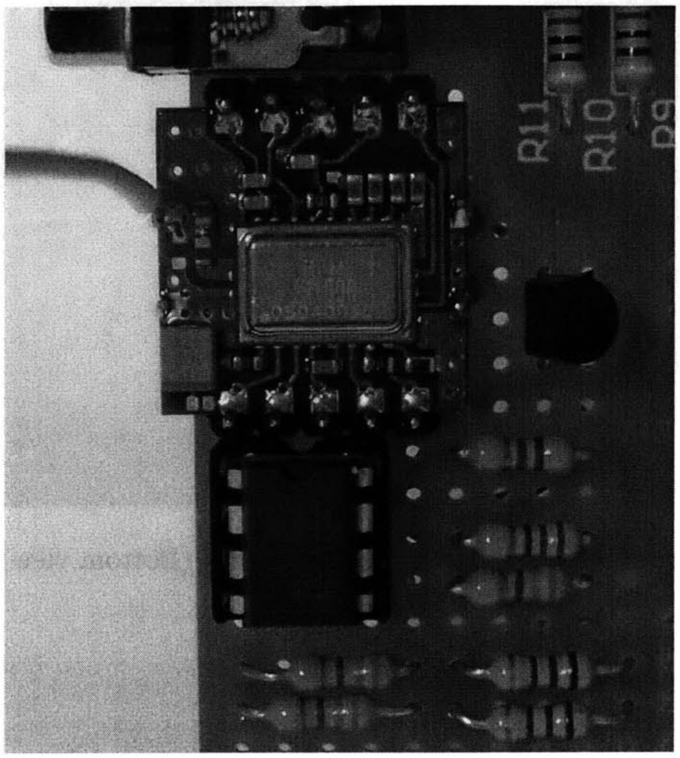

Figure 4-6: The DR3000-1 mounted (Top view).

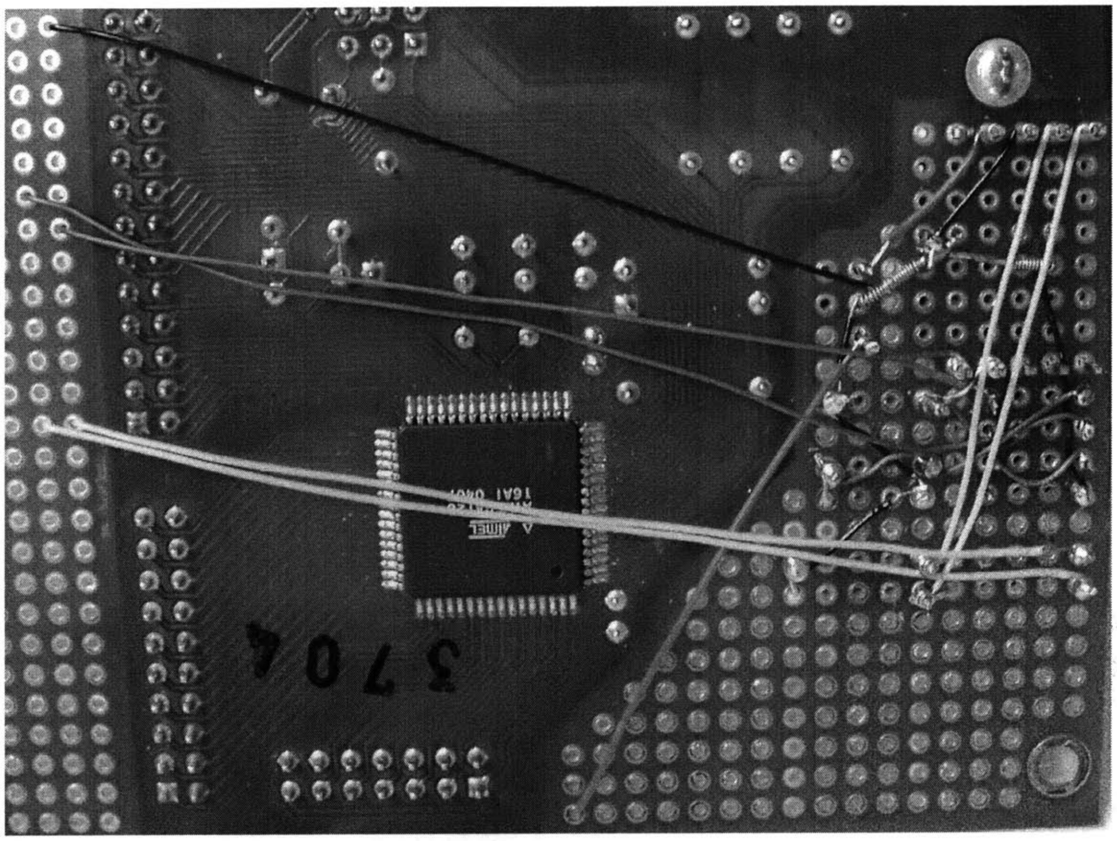

Figure 4-7: The DR3000-1 mounted (Bottom view).

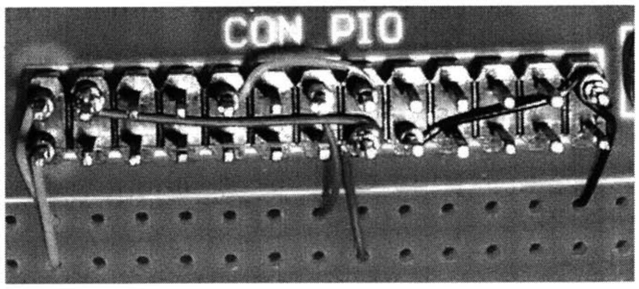

Figure 4-8: The DR3000-1 CON_PIO connections.

Table 4.1: LedsArray pin connections.

LedsArray Bit	9	8	7	6	5	4	3	2	1	0
Physical Pin	PC1	PC0	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

However, the DR3000-1 does not implement the SPI protocol; it simply sends or receives bits, based on the settings of its two control lines. So, the microcontroller must also take on part of the master functionality. In particular, recall from Section 2.2.3 that the master is responsible for sending pulses on the Serial Clock line. To this end, SCK is wired to OC2, which is an output pin from internal Timer/Counter2. The microcontroller uses Timer/Counter2 to send pulses on OC2, which the SPI unit detects and interprets as SCK pulses. This bit of hardware trickery—and the software which makes it work—were borrowed from the MICA platform.

4.1.3 The LedsArray

The last of the major hardware components is the LedsArray, a set of ten LEDs connected to ten of the pins of `CON_BUS`: the eight of Port A and pins PC0 and PC1 of Port C (see Figure 4-10). The component is arranged logically as shown in Table 4.1. The most significant bit of the 10-bit number (the 512's place) is connected to PC1, and the least significant bit (the 1's place) is connected to PA0. Each LED is wired as shown in Figure 4-9. When the pin outputs a logical 0 (i.e. 0V), there is a voltage drop across the LED and the resistor and current flows, lighting up the LED; when the pin outputs a logical 1 (5V), there is no voltage drop and the LED is off.

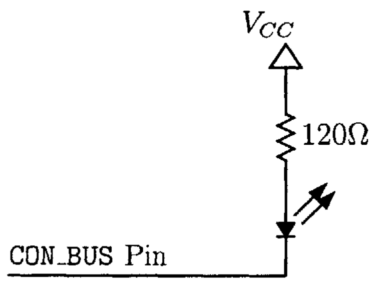

Figure 4-9: One LED of the LedsArray.

The LedsArray component can be used to provide visual feedback. In particular, it was designed to be able to display all 10 bits of an ADC conversion[9]—each lit LED indicating a "1", the rest signifying "0". It is not limited to displaying ADC results, though. The LedsArray hardware is controlled by a LedsArray software module which can be used in conjunction with any other software that wants to make use of it.

As previously mentioned, it would not be prudent to be running ten LEDs in an actual WSN deployment; this would decimate the mote's battery life. However, during the planning and testing stages (i.e. while *prototyping*) the LedsArray can be an invaluable component of the JONA Prototyping Kit. See Chapter 5 for several examples in which the LedsArray is used to verify the proper operation of other components.

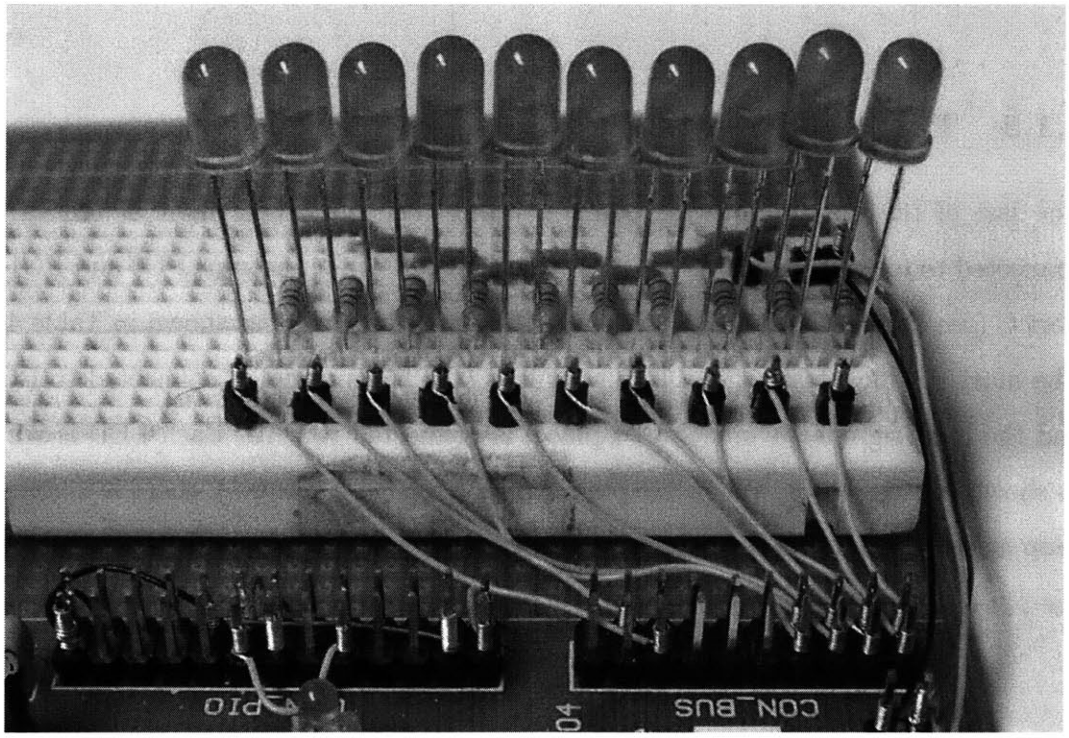

Figure 4-10: The LedsArray (in Green).

[9]Compare this to the LEDs of a MICA mote, which can only display three bits at a time.

4.2 The JONA Software Platform

The JONA kit runs TinyOS [23]—written in nesC [19]—for its software. To write, compile, and install TinyOS applications, the TinyOS software distribution must be installed on a desktop computer. It requires a UNIX-like command line environment such as that found in Linux or Mac OS X. For Windows, the TinyOS installer includes a Linux-like environment called Cygwin [15]. The TinyOS software distribution is arranged as shown in Table 4.2. All JONA-specific extensions to the base distribution are placed in its platform directory.

4.2.1 TinyOS Design

TinyOS is a modular, event-driven system, designed to be power-efficient, minimal in size, and easily adaptable to various hardware platforms. The paradigm used is one of layers built on top of one another, in which communication is achieved through a combination of *commands* and *events*. Lower-level components may trigger events in the components above them, and higher-level components may issue commands to the components below them. At the lowest level, events are triggered by hardware interrupts—signals generated in the microcontroller in response to events such as the reception of a new byte from the UART or an internal timer going off.

Events are asynchronous, in that the issuer does not wait for a response before going on to something else; furthermore, they are intended to be fairly small blocks of code which execute quickly. Commands must also execute quickly, as they may be called by an event handler which must wait for their return values before it completes[10]. Processor-intensive (i.e. time-consuming) jobs are to be done by *tasks*. TinyOS provides a small first-in-first-out (FIFO) queue to store tasks waiting to be performed. Each task runs to completion before the next begins, so they are atomic (indivisible) with respect to one another, though a task may be interrupted by an event (hence the desire for events to be processed as quickly as possible).

TinyOS, with its emphasis on size and speed, does not permit common operat-

[10]In order to prevent infinite loops, commands are not allowed to trigger events.

Table 4.2: The TinyOS Directory

apps/ TinyOS applications, each in its own subdirectory.

doc/ TinyOS documentation.

tools/ Various supporting programs to be run on the computer, such as a Java oscilloscope to graph readings from a mote's ADC.

tos/ TinyOS operating system components used by the applications in **apps/**.

- **interfaces/** Interface files (like C header files) listing the commands and events associated with particular *interfaces*. Components providing an interface must provide implementations of each of the listed functions. For example, any component providing the `StdControl` interface (defined in *StdControl.nc*) must implement `init()`, `start()`, and `stop()` functions.
- **lib/** Library functions such as Counters and Multihop Routing grouped into subdirectories.
- **platform/** Each hardware platform has its own subdirectory here. Files specific to particular hardware go there, and can override like-named files anywhere else in this source tree.
- **sensorboards/** Crossbow sensors are grouped on expansion cards called *sensorboards*. Each has its own subdirectory here with a header file defining constants particular to that sensorboard, along with implementation files for each of the hardware components found on that board. A stub directory called "**none/**" containing a blank header file is to be used when no sensorboard is present.
- **system/** General system files used (or overridden) by all hardware platforms.
- **types/** Definitions of TinyOS data structures such as `TOS_Msg`, representing a radio packet (in *AM.h*).

ing system features such as dynamic memory allocation[11] and multithreading[12]. By allowing only static memory allocation, the compiler is able to determine the (constant) amount of memory the application will require throughout its lifetime, which lets it ensure that a wireless sensor node will never run out of storage space. If the program fits onto the mote during installation, it will fit throughout its entire execution. Static allocation and lack of multithreading also contribute to faster operation, because memory addresses can be determined by the compiler (rather than by the program as it runs) and the mote does not have to spend the time to switch from one thread to the next.

4.2.2 nesC Design

TinyOS was initially written in C, but it was soon rewritten in nesC, which provides support for the TinyOS abstraction model in the syntax of the language itself. For instance, nesC provides keywords to indicate whether a function is a command, an event, or a task. At the same time, nesC is an extension of C, so traditional C functions can be called alongside nesC functions, and nesC programs can take advantage of C's low-level hardware access. On top of the C foundation nesC adds component-based design structure.

This structure takes the form of three types of files (all of which carry the *.nc* filename extension): interfaces, configurations, and modules. An interface file provides the description of an interface which modules may *provide* or *use.* This interface defines a contract of how its providers and users will interact, by declaring functions called commands and events. If a module provides an interface, it must implement each of the commands listed in the interface file; if it uses the interface, it must implement each of the events listed in the file. Configuration files "wire together" modules by connecting the interface used by one to the interface provided by another. Once

[11]Dynamic memory allocation refers to the ability of a program to request control of more and more memory as it runs (and to release control of memory it no longer needs).

[12]Multithreading refers to the ability of a computer (or embedded device) to execute more than one program or "thread" at the same time, by running one for a while, then switching to the next, and so on.

this connection is made, the interface user can issue commands to the provider, and the provider can trigger events in the user.

4.2.3 Hello, World!

In order to demonstrate nesC syntax and TinyOS's event-driven programming model, a small example program—which, by tradition, displays "Hello, World!"—will now be investigated. The full text of the program is provided in Appendix G on page 165. This program uses the `ByteComm` interface for communication over the UART[13]. When running this program, the mote should be connected via serial cable to a computer, and the computer should be running a terminal program (such as Windows's *Hyper-Terminal*) to view the incoming message.

The Configuration File

```
configuration HelloWorld {
}
```

The HelloWorld configuration file, *HelloWorld.nc*[14], begins with this declaration stating that what follows is indeed a configuration by the name of HelloWorld. If the configuration used or provided any interfaces, they would be listed between the curly braces.

```
implementation {
  components Main, UART, HelloWorldM;
```

The next line indicates that what follows—everything between the { and the matching }—is the implementation of the HelloWorld configuration. The first line

[13]`ByteComm` can also be used to send and receive bytes over the radio. This means that by changing the wiring in a configuration file, a program that used the UART can be modified to use the radio without having to go through the entire implementation module changing UART commands to radio commands.

[14]The TinyOS naming convention is that an application's name is the name of its top-level configuration file. Furthermore, the name of a module should end in an uppercase M [48]. So, HelloWorld's configuration is found in *HelloWorld.nc*, and its module, HelloWorldM, is found in *HelloWorldM.nc*.

of the implementation lists the components which will be wired together to create HelloWorld; in this case `Main`, `UART`, and `HelloWorldM`. All applications use the `Main` component, which is in charge of initializing, starting, and stopping the other components at runtime. `UART` is a (software) component which provides access to the (hardware) UART. `HelloWorldM` is HelloWorld's module, which contains the actual code that will send the message out over the serial port.

```
  Main.StdControl -> HelloWorldM;
  Main.StdControl -> UART;
```

These two lines wire the `StdControl` interface used by `Main` to the implementations provided by `HelloWorldM` and `UART`. Notice that an interface used by one component can be wired to multiple implementations.

```
  HelloWorldM.ByteComm -> UART;
}
```

The next line wires the `ByteComm` interface used by `HelloWorldM` to the implementation provided by `UART`. Finally, the } marks the end of the implementation, and the end of the file.

The Module File

```
module HelloWorldM {
  provides {
    interface StdControl;
  }
  uses {
    interface ByteComm;
  }
}
```

The HelloWorld module file, *HelloWorldM.nc*, begins much like the configuration file, declaring itself as the `HelloWorldM` module. In this case, however, the file provides and uses interfaces, and so they are declared between the curly braces. Recall that, in the configuration file, the `StdControl` interface provided here is wired to

Main, and the ByteComm interface used here is wired to the implementation provided by UART.

```
implementation {
  uint8_t index, *string = (uint8_t*)"Hello, World!\r\n";
```

The implementation section begins with a list of C-style variables which the module will be using. In this case, index will be an 8-bit unsigned integer (i.e. a byte holding a value between 0 and 255), and string will be a *pointer* to an 8-bit unsigned integer[15]. This pointer is then set to point to the beginning of the hello world string, which is cast to be a pointer to a uint8_t[16].

The last two characters of the hello world string are the special symbols[17] \r and \n. These represent a carriage return and a line feed (newline), and will be transmitted as the two bytes 0x0D and 0x0A. When the computer's terminal program receives this sequence, it will move the cursor to the beginning of the next line, just as if someone had hit the return (enter) key on the keyboard.

```
  command result_t StdControl.init() {
    atomic {
      index = 0;
    }
    return SUCCESS;
  }
```

The first function in the HelloWorldM module is the init() function of the StdControl interface. It is a nesC command (which will be issued by Main when

[15]That is, string will not contain the integer itself; it will contain the integer's memory address. See Figure 4-11.

[16]A sequence of characters enclosed in double quotes is represented internally as an array of chars, which are 8-bit numbers intended to represent regular characters—letters, numerals, punctuation in some encoding scheme, usually one known as *ASCII* (see Appendix C on page 153). This array will occupy sequential locations in memory, and it will be *null-terminated*—it will end with a byte holding the value 0x00. Because a char is stored as a byte, it can safely be treated as a uint8_t, which is also a byte. When treated as a pointer, the array's value is its address in memory—the location of its first element. So, the variable string will contain the address of the letter "H". See [27] for more details.

[17]In C, the backslash ("\") provides a mechanism for representing hard-to-type or invisible characters, using *escape sequences* such as \r and \n. Others include "\"" to produce a double quote, "\t" to produce a horizontal tab, and "\\" to produce the backslash itself. When used in a textual context, the *number* 0x00 is often represented as the escape sequence "\0" to distinguish it from the *numeral* "0". Refer to Section 2.3 of [27] for a complete list of C's escape sequences.

the mote is first turned on), and it will return a value of the type result_t, indicating success or failure. StdControl.init() does not require any parameters, but if it did they would be listed inside the parentheses. The init() function sets the globally-accessible variable index to hold the number 0. Because other functions can access index, this assignment is enclosed in an atomic block, which will ensure that nothing else tries to read or write index at the same time. After setting the variable, StdControl.init() finishes by returning the result_t indicating successful completion of the command.

```
command result_t StdControl.start() {
  uint8_t i;
  atomic {
    i = index;
    ++index;
  }
  return call ByteComm.txByte(string[i]);
}

command result_t StdControl.stop() {
  return SUCCESS;
}
```

HelloWorldM then implements the other two functions of the StdControl interface, start() and stop(). There is nothing for the HelloWorldM module to do when it is commanded to stop, so it just returns SUCCESS. StdControl.start(), however, is responsible for initiating the machinations which will cause the hello world message to be transmitted. It has its own private uint8_t called i; inside an atomic block, it reads the value stored in index—namely the number 0, from StdControl.init()—and writes it into i. It then increments by 1 the value stored in index (to the number 1).

Finally, rather than returning a static value like SUCCESS, it lets its return value be whatever the result is of calling the txByte() command of the ByteComm interface. This command has a single parameter, a uint8_t holding the byte to be transmitted. StdControl.start() supplies it with this parameter by obtaining the i^{th} element of the string array[18]. So, UART—which is wired to the ByteComm interface used by

[18]Arrays in C-based languages are *zero-based*: the first element is located at index 0, the second

HelloWorldM—is commanded to transmit "H" over the serial port.

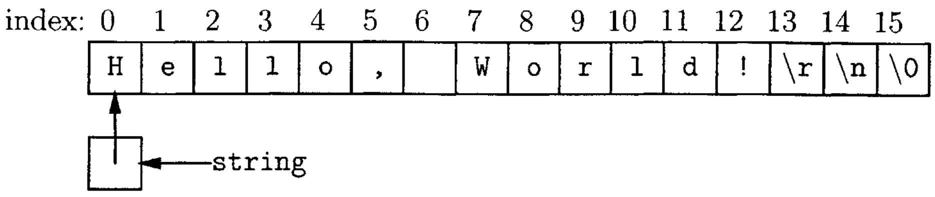
index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
H e l l o , W o r l d ! \r \n \0
string

Figure 4-11: The HelloWorld string array.

```
async event result_t ByteComm.rxByteReady(uint8_t dat, bool err,
                                          uint16_t str) {
  return SUCCESS;
}

async event result_t ByteComm.txDone() {
  return SUCCESS;
}
```

The rest of the module implements handlers for asynchronous[19] events which can be triggered by ByteComm. The first of these events is rxByteReady(), which is triggered when a byte is received at the UART from the serial port (i.e. a byte sent from the computer). This program will ignore any received bytes, so the event handler simply returns SUCCESS. The next, txDone(), is triggered as soon as a transmission has been completed (such as when the "H" has been sent). This is nice to know, but there is still nothing to do, so the handler returns SUCCESS.

```
async event result_t ByteComm.txByteReady(bool success) {
  uint8_t c, i;
  atomic {
    i = index;
  }
  c = string[i];
```

The txByteReady() event signals that UART, having already sent a byte and signalled txDone(), is finished cleaning up after itself and is ready to send another byte.

at index 1, and so on. See Figure 4-11.

[19]The events are labelled *asynchronous* because they are not part of the regularly-scheduled (synchronous) task queue. Events can interrupt each other, so the compiler uses the async keyword to check to make sure that one event does not attempt to write to a global variable while another is reading it (hence the atomic blocks).

The boolean parameter—which will be either true or false—indicates whether the previous transmission was successful. A more defensively-written, robust program could use this to retransmit a byte in the case of a failure, but in the interests of simplicity HelloWorld ignores it and assumes that all transmissions are successful.

As in `StdControl.start()`, this event handler begins by reading the value of `index` and storing it in a local variable named `i`[20]. It then reads the byte stored at index `i` of the `string` array and stores it in `uint8_t c`[21]

```
if(c) {
  atomic {
    ++index;
  }
  return call ByteComm.txByte(c);
}
```

The next statement, `if(c)`, checks whether c is "true." In C (and, therefore, in nesC), the number 0 is considered to be false, and any nonzero value is considered to be true. So, c will evaluate as being true all the way to the end of the array, and then on the next attempt after the \n it will be false[22].

If c is true, then the instructions between the curly braces will be executed. First, `index` will be incremented, so that the next time it will point to the next element of the hello world string. Then the handler will return, commanding `ByteComm` to transmit c as its final act.

```
      else {
      return SUCCESS;
    }
  }
}
```

If c did not evaluate to true, then `string[i]` must have been the null character marking the array's end. In that case, the if statement's `else` clause will be executed

[20]Note that this is not the same `i` as the one found in `StdControl.start()`; they have the same name, but they are distinct.

[21]Even though `string` is a global variable, this read does not have to occur within an `atomic` block because the `string` array is immutable and all accesses are reads; no function attempts to write a new value to it.

[22]Recall that character strings in C are null-terminated.

instead. Having reached the end of the string, there is nothing left to do, and so the event handler returns `SUCCESS`. The next three } symbols mark the ends of the else clause, the `ByteComm.txByteReady()` event handler, and the `HelloWorldM` module's implementation.

Notice how TinyOS's event-driven model comes into play. The first byte is transmitted as a result of a command issued by `Main`; the rest are sent in response to events received from `UART`.

Installation

TinyOS uses the *make* utility to compile and install applications. So, each application has in its directory a file called *Makefile* which tells *make* what to do. Sometimes a *Makefile* must specify where *make* should look for components, but HelloWorld is a simple program, and its *Makefile* is correspondingly simple.

```
COMPONENT=HelloWorld
include ../Makerules
```

The first line sets the `COMPONENT` variable to be the name of the application, "`HelloWorld`". The *Makefile* then instructs *make* to include the contents of the file *Makerules*, located up one level in the **apps/** directory. This file contains the details of how *make* should go about making `COMPONENT`[23].

Table 4.3 illustrates the various commands which can be issued to *make* which are supported by *Makerules*. To compile HelloWorld for the JONA platform, the command `make jona` should be executed from within the HelloWorld application's directory. To subsequently install it onto a mote, the command to be used is `make reinstall jona`. To do both steps at once, the command is `make install jona`.

Before a JONA mote can be reprogrammed, it must be restarted in programming mode. When a PROBOmega128 is turned on, it checks whether pin PF7 is being pulled low; if so, it enters programming mode, otherwise it executes whatever program

[23]See Appendix H on page 169 for the extensions made to the *Makerules* file which provide support for the JONA platform.

Table 4.3: Usage of *make*.

Command	Result
make <platform>	Compiles the application for the indicated platform.
make all	Compiles the application for all available platforms.
make clean	Removes files created during compilation (such as the executables), leaving only the source code.
make install[.n] <platform>	Compiles the application for the indicated platform, and installs it onto the connected mote. If a number n is provided, the mote is assigned that address.
make reinstall[.n] <platform>	Installs the already-compiled application onto the connected mote, assigning it address n if provided.
make docs <platform>	Generate documentation files for the indicated platform, placing them in the **docs/** directory.

is already installed. So, to reprogram the mote, pin PF7 should be pulled low by putting a jumper cap[24] across `CON_ADC` pins 11 and 12 (see Figures 4-13 and 4-14). If programming mode is successfully entered, the PROBOmega128's LED will blink thrice and then stay off. The general procedure for programming a JONA mote, then, is as follows:

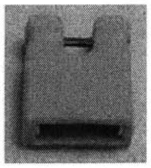

Figure 4-12: A jumper cap.

1. Connect the mote to a computer via a serial cable, while the mote is off[25].

[24] A *jumper cap* is a small clip which fits across two pins and provides an electrical (i.e. metal) connection between them. See Figure 4-12.

[25] RS-232 uses relatively high voltages (tens of volts), so connecting the cable while the mote is on

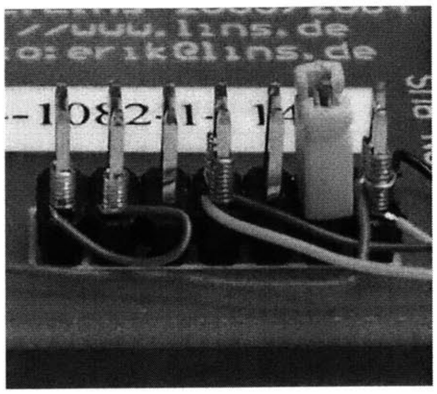

Figure 4-13: CON_ADC with a jumper cap in place.

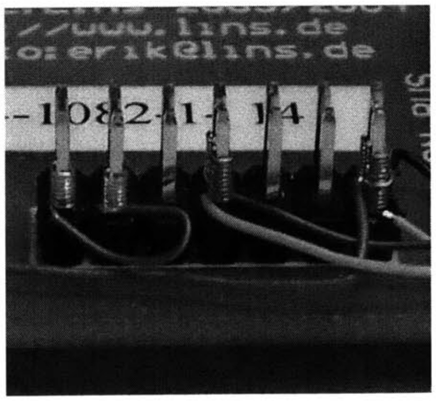

Figure 4-14: CON_ADC with the jumper cap removed.

2. Place a jumper cap across `CON_ADC` pins 11 and 12 to ground PF7.

   These two steps can be performed in either order. Additionally, although connecting the serial cable with the power on is dangerous, connecting the jumper cap with the power on is not.

3. Turn on the mote (i.e. plug it in).

4. After the LED's three blinks, issue the installation command—`make install jona` or `make reinstall jona`—on the computer.

Execution

After the HelloWorld application has been installed, it is ready to be executed. First, a terminal program like Windows's *HyperTerminal* should be launched on the computer and set up to communicate with the serial port with a baud rate of 9600, and the jumper cap should be removed from `CON_ADC`. Then the mote should be restarted. At this point, `Main` will call the `StdControl init()` and `start()` commands, which will trigger the mote to transmit "`Hello, World!`" to the computer.

4.2.4 The JONA TinyOS platform directory

This section will describe the files found in the **jona/** platform directory which enable TinyOS to work with the JONA hardware. Although TinyOS is intended to be portable, many of the general files in **tos/system/** assume particulars about MICA hardware, such as the presence of a potentiometer to adjust RF signal strength. Other platforms, such as MICA2, which do not have this potentiometer must either provide their own implementations of all of these base files or provide their own place-holder potentiometer module which does nothing. Some of the following files are used to override MICA assumptions in the TinyOS system, but others provide all-new functionality to support features of the JONA hardware. This section will briefly indicate each file's function, pointing out significant features; see Appendix I for complete file listings.

may cause voltage spikes which can damage the mote or the computer.

General Files

The first two files, *.platform* and *hardware.h*, are found in all platform directories.

.platform The *.platform* file is used by *make* to set compiler options and to determine where to look for source code files it needs to include.

```
@opts = ("-gcc=avr-gcc",
         "-mmcu=atmega128",
         "-fnesc-target=avr",
         "-fnesc-no-debug");

push @opts, "-mingw-gcc" if $cygwin;

@commonplatforms = ("mica128", "mica", "avrmote");
```

The "`@opts`" list says that the mote's microcontroller is an ATmega128, which uses the AVR ISA, and that the appropriate C compiler is *avr-gcc*[26]. The "`@common-platforms`" list tells *make* that if it needs a file, and it does not find it in this platform directory, it is allowed to look in the **mica128/** platform directory. If the file is not found there, then it may try the **mica/** and **avrmote/** directories. After searching in all these places, it then moves on to the general source tree, in **tos/interfaces/**, **tos/system/**, and **tos/types/**[27]. This is how a platform is able to override the TinyOS defaults; if *make* finds a required file in the platform directory, then it stops looking and will never see the original version.

hardware.h While the *.platform* file contains directives for *make*, the *hardware.h* file contains instructions and definitions for the compiler itself. For instance, consider the following four lines:

```
TOSH_ASSIGN_PIN(RFM_RXD,  B, 2);
TOSH_ASSIGN_PIN(RFM_TXD,  B, 3);
```

[26]*GCC*, the *GNU Compiler Collection*, is a family of open source compilers produced by the Free Software Foundation. The nesC compiler, *ncc*, translates the nesC source into a C program, which is then compiled by *avr-gcc* into AVR machine code which will execute on the ATmega128.

[27]If required files are in **tos/lib/**, the directory must be explicitly included in the application's *Makefile*.

```
TOSH_ASSIGN_PIN(RFM_CTL0, D, 7);
TOSH_ASSIGN_PIN(RFM_CTL1, D, 6);
```

These assign[28] the *symbolic names* `RFM_RXD`, `RFM_TXD`, `RFM_CTL0` and `RFM_CTL1` to the *physical pins* PB2, PB3, PD7, and PD6. Henceforth, other components can refer to the abstract ideas of the radio's receive, transmit, and control lines, without having to be aware of where these lines are actually connected.

`TOSH_ASSIGN_PIN(name, port, bit)` causes the creation of five functions:

- `TOSH_SET_##name##_PIN()`
- `TOSH_CLR_##name##_PIN()`
- `TOSH_READ_##name##_PIN()`
- `TOSH_MAKE_##name##_OUTPUT()`
- `TOSH_MAKE_##name##_INPUT()`

These will, respectively, *set* the pin's output to 1, *clear* it to 0, *read* the pin's value, and set the port's DDR (see Section 2.2.2) to make the pin an *output* or an *input*. The sequence "`##`" in the function names indicates concatenation. This lets `TOSH_ASSIGN_PIN()` create functions whose names depend on the `name` argument passed to it. For example, `TOSH_ASSIGN_PIN(RFM_RXD, B, 2)` will create:

- `TOSH_SET_RFM_RXD_PIN()`
- `TOSH_CLR_RFM_RXD_PIN()`
- `TOSH_READ_RFM_RXD_PIN()`
- `TOSH_MAKE_RFM_RXD_OUTPUT()`
- `TOSH_MAKE_RFM_RXD_INPUT()`

[28]Note that these are the connections indicated by Table F.1 in Appendix F.

Sensor Files

The following six files enable the JONA to have sensors without having a sensorboard. Each of the files is based on its counterpart in the **micasb/** sensorboard directory. The files included are for an accelerometer and for a photosensor; they can be used as templates to create the files necessary for other analog sensor types.

accel.h & photo.h These two files, based on *sensorboard.h*, tell the compiler which ADC pins the accelerometer and photosensor are connected to. In particular, *accel.h* states that the accelerometer's x-axis sensor is connected to ADC channel 3 and that its y-axis sensor is connected to channel 4, while *photo.h* says that the photosensor is connected to channel 1.

Accel.nc & Photo.nc *Accel.nc* and *Photo.nc* are configuration files for the accelerometer and photosensor. They are unchanged from their **micasb/** versions, except that they read from *accel.h* and *photo.h* instead of *sensorboard.h.*

AccelM.nc & PhotoM.nc *AccelM.nc* and *PhotoM.nc* are the implementation files for the accelerometer and photosensor. They too are based upon the **micasb/** counterparts, but in addition to referencing *accel.h* and *photo.h* they remove references to "control pins." In the micasb sensorboard, each sensor can be individually turned on or off via a control line, because a switch has been placed between each sensor and the power rail. However, for simplicity, each sensor in the JONA is directly connected to V_{CC}.

Files Overriding MICA

This group of files overrides the counterparts in the **mica/** platform directory, or in **tos/system/**.

HPLPotC.nc & HPLSlavePinC.nc As mentioned previously, the MICA uses a software-controlled digital potentiometer to adjust the broadcasting power of its

TR1000 radio. The JONA uses this same radio chip, and so it uses many of the same files as the MICA. However, the JONA has no such potentiometer. So, this *HPLPotC.nc*, which is a module file for the "hardware presentation layer" potentiometer component, returns `SUCCESS` for each of the `HPLPot` interface's commands without actually doing anything. This approach is also taken by the MICA2 platform, which has its own "dummy" *HPLPotC.nc*. Likewise, *HPLSlavePinC.nc* returns `SUCCESS` for the `HPLSlavePin` interface's commands without doing anything. The original SlavePin was a feature of the MICA, which had an external memory chip (its microcontroller, the ATmega103, did not have as much storage space as the newer ATmega128 has). The SlavePin was a control line associated with this external chip. As the JONA has no such chip, it has no need of a SlavePin.

ChannelMonC.nc, RadioTimingC.nc, & SpiByteFifoC.nc The three files *ChannelMonC.nc*, *RadioTimingC.nc*, and *SpiByteFifoC.nc* control aspects of the JONA's radio. The JONA kit runs at 14.7456 MHz, nearly four times faster than the 4-MHz MICA. This means that if the JONA were to use the MICA radio files, the radio would have a transmission bit rate four times that of the MICA radio. However, it turns out that this bit rate exceeds the radio's capability. So, these three files have been modified from their MICA counterparts to scale down radio-related timings by a factor of 8.

HPLUARTM.nc The *HPLUARTM.nc* file is the hardware presentation layer for the UART. It is responsible for setting up and mediating access to the UART hardware, using the `HPLUART` interface. The ATmega128 actually has two UARTs called[29] USART0 and USART1. The serial port of the PROBOmega128 is connected to USART0, and so it is this UART which *HPLUARTM.nc* controls. As such, it shares many similarities with the MICA2's *HPLUART0M.nc*.

HPLUARTM.nc sets up the UART to have a baud rate of 9600.

[29]They are called *USART* because they are capable of operating in a synchronous mode, in addition to the regular asynchronous operation.

ADCC.nc & HPLADCM.nc *ADCC.nc* and *HPLADCM.nc* mediate access to the analog-to-digital converter. `ADCC` is a software abstraction layer that sits atop `HPLADCM`, which directly accesses the ADC hardware. These two files override the default *ADCC.nc* and *HPLADCC.nc* files in **tos/system/**. Those defaults are based upon the specifications of the ADC found in MICA's ATmega103 microcontroller; these are borrowed from MICA2, which features the low-power version of JONA's own ATmega128 microcontroller.

ADCC.nc is, in fact, identical to the original in the **mica2/** platform directory. The file *HPLADCM.nc* is functionally equivalent to the MICA2 version, but a short list of cosmetic changes have been made. In particular, the MICA2 version uses a deprecated AVR function[30] called "outp". This version makes use of the replacement function, "outb".

LED Files

The next six files have to do with the PROBOmega128's single built-in LED.

Led.nc *Led.nc* is an interface file declaring the commands appropriate for the solitary LED. It is based upon TinyOS's *Leds.nc* interface file, which declares functions for the MICA's red, yellow and green LEDs. These are `init()`, `On()`, `Off()`, `Toggle()`, `get()`, and `set()`. The functions which are capitalized had three counterparts in *Leds.nc* (e.g. `redOn()`, `yellowOn()`, and `greenOn()`); when the interface was copied and modified into `Led`, the color prefixes were simply dropped, and the capitalization remained.

LedC.nc *LedC.nc* provides the implementation of the Led module. To turn on the LED, it calls `TOSH_CLR_LED_PIN()`; to turn it back off, it calls `TOSH_SET_LED_PIN()`. This may seem backwards, but recall that the LEDs are wired between a pin of the microcontroller and V_{CC}, not ground. To turn the LED on, the pin voltage must be brought low to create a voltage difference across the diode.

[30]These "functions" are actually macros defined in *<avr/sfr_defs.h>*.

IntToLed.nc *IntToLed.nc* provides the configuration of an `IntToLed` component which provides the `IntOutput` interface. It is based on the 3-LED *IntToLeds.nc* configuration file.

IntToLedM.nc *IntToLedM.nc* provides the implementation of the `IntToLed` component. The `IntOutput.output()` command takes a 16-bit number as a parameter. `IntToLedM` will turn on the LED if the number is odd, otherwise it will turn the LED off. It then sends the `IntOutput.outputComplete()` signal to indicate that it is finished.

LedsC.nc & LedsM.nc A new single-LED component had been made, but many TinyOS system files expected to be able to use the `LedsC` component. *LedsC.nc* and *LedsM.nc* are configuration and module files for a replacement `LedsC` component. It simply maps the three `Leds` LEDs to the one `Led` LED.

LedsArray Files

The last four files have to do with the physical 10-LED LedsArray component. As the LedsArray is an entirely new component and not merely the JONA equivalent of one already found in MICA and the others, they will be examined more closely than the other platform directory files. These files provide a case study into how to create TinyOS components and their interfaces.

LedsArray.nc *LedsArray.nc* defines the `LedsArray` interface:

```
interface LedsArray {
  async command result_t init();
  async command result_t allOn();
  async command result_t allOff();
  async command result_t allToggle();
  async command uint16_t get();
  async command result_t setv(uint16_t value);
  async command result_t inc();
  async command result_t dec();
  async command result_t setav(uint16_t value);
  async command result_t clrav(uint16_t value);
}
```

LedsArray.init() is the initializer, which sets up the LedsArray pins to be output. When initialization is complete, all LEDs will be off. LedsArray.allOn() makes sure that all ten LEDs are turned on, regardless of their previous states. Likewise, LedsArray.allOff() turns all ten off. LedsArray.allToggle() turns off any LEDs which are on, and turns on the ones which are off. Each of these returns SUCCESS upon completion.

LedsArray.get() returns a uint16_t representing the state of the LedsArray. LedsArray.setv(uint16_t value) sets the LedsArray to display value. In each of these cases, only the low ten bits of the sixteen-bit uint16_t are used.

LedsArray.inc() increments the displayed value by one, and LedsArray.dec() decrements it by one. These are equivalent to calling LedsArray.setv() with arguments of, respectively, LedsArray.get()+1 and LedsArray.get()-1.

The last two functions, setav() and clrav(), each take in a uint16_t holding a 10-bit number. For each of the ten bits, if the bit is 0, no action is taken, but if it is 1, then the corresponding LED is turned on—for setav()—or off—for clrav(). This allows for individual LEDs to be turned on or off, without inadvertently affecting the states of the others.

LedsArrayC.nc The *LedsArrayC.nc* file provides the implementation module of the LedsArrayC component.

```
module LedsArrayC {
  provides interface LedsArray;
}
implementation
{
  uint16_t ledsOn;
```

First of all, LedsArrayC is declared to provide the LedsArray interface. The module then declares a global variable, ledsOn, which will be a uint16_t. ledsOn will let LedsArrayC keep track of which LEDs are on.

```
  void output(uint16_t num);
```

The output() function is a "helper" which the various LedsArray commands will use to set the bits of PORTC and PORTA to the appropriate values. output() takes as a parameter a uint16_t which will be interpreted as a set of bits to write to the pins. If an LED is supposed to be on, the corresponding bit in num should be a 0; otherwise, it should be a 1. In either case, the high six bits of num should all be 0 to ensure correct operation. This statement declares to the compiler that a function called output(uint16_t) exists, so that when other functions refer to it, the compiler will know what they are talking about. output() is not defined, however, until the end of the file.

```
async command result_t LedsArray.init() {
  atomic {
    ledsOn = 0;

    DDRC |= 0x3;
    DDRA = 0xFF;
    output(0x3FF);
  }
  return SUCCESS;
}
```

The body of the init() function is contained within an atomic block, so that nothing else can interfere with its assignments. First, the ledsOn variable is set to be 0, which will represent all LEDs being off. Then, it proceeds to turn off all the LEDs.

Recall that each Port has a data direction register associated with it, which establishes which pins of the Port will be input or output, with those bits set to 1 indicating output. The value 0xFF is written to DDRA. 0xFF is 0b11111111 in binary[31], so this sets all eight pins of Port A to be output.

The statement DDRC |= 0x3 is shorthand for DDRC = (DDRC | 0x3). DDRC, the DDR for Port C, will be assigned the value of the expression (DDRC | 0x3). The vertical bar, "|", is the bitwise *or* operator. It will construct its output as follows: for each bit position, if the corresponding bit in DDRC is 1, or the corresponding bit in 0x3 is 1, then that bit of the output will be 1. It will only be 0 if both input bits

[31]See Appendix B on page 151.

Table 4.4: The bitwise or (|) operation.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

are 0. For example, suppose DDRC is initially set to 0xA0. In that case, DDRC | 0x3 will be 0xA3. If DDRC is already set to 0x3, then DDRC | 0x3 will still be 0x3. So, regardless of DDRC's previous value, this line will ensure that its lowest two bits, PC0 and PC1, are set to be output.

Having set the DDRs to the appropriate values, output() is called with an argument of 0x3FF: ten 1's. All of the LedsArray pins will output a high voltage, there will be no drop across the diodes, and all the LEDs will be off.

```
async command result_t LedsArray.allOn() {
  atomic {
    output(0);
    ledsOn = 0x3FF;
  }
  return SUCCESS;
}
```

The LedsArray.allOn() function sets the ten LedsArray pins to 0, so that the LEDs will turn on. Finally, the value 0x3FF is stored in ledsOn to record the fact that all LEDs have been turned on.

```
async command result_t LedsArray.allOff() {
  atomic {
    ledsOn = 0;
    output(0x3FF);
  }
  return SUCCESS;
}
```

LedsArray.allOff(), like init(), calls output() with an argument of 0x3FF to turn off all LEDs, and stores 0 in ledsOn.

Table 4.5: The one's complement (~) operation.

Input	Output
0	1
1	0

Table 4.6: The bitwise and (&) operation.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

```
async command result_t LedsArray.allToggle() {
  atomic {
    output(ledsOn);
    ledsOn = (~ledsOn & 0x3FF);
  }
  return SUCCESS;
}
```

It may be noticed that at any given time, the values of the ten bits of `ledsOn` and those of the ten bits written to the pins are opposites. When an LED is lit, its pin will hold 0, but `ledsOn` will hold a 1. `allToggle()` takes advantage of this fact and simply calls `output()` with `ledsOn` as its argument. This will invert the values on each of the pins, and toggle the states of the ten LEDs.

To invert the bits of `ledsOn` itself, the *one's complement* operator is used. This unary operator changes every 0 to a 1 and every 1 to a 0. Once that is done, though, the top six bits (which were 0's) are now 1's. To clear those bits without affecting the rest, the bitwise *and* operator is used. This is often used to "mask off" certain bits. Each bit of output will be 0 unless both input bits are 1. In this case, `ledsOn` is anded with 0b0000001111111111, so the high six bits will be 0, and the low ten will be whatever they are in `~ledsOn`. This result is finally written back into `ledsOn`.

```
async command uint16_t LedsArray.get() {
  uint16_t rval;
  atomic {
    rval = ledsOn;
  }
  return rval;
}
```

LedsArray.get() makes a local copy of the global ledsOn variable and returns it. In order to return a value stored in a variable, a function must access that variable's memory location—it must read the variable. By atomically copying ledsOn before returning its value, get() ensures that nothing else will attempt to write a new value to ledsOn while it is reading it. In general, this is the appropriate method to pass a global variable in a return statement.

It would not work to put return ledsOn; within the atomic block, because the function would exit at the return statement, and the program would never see the atomic block's closing brace. It would return to whatever had called get() and still think that it was in the atomic block. This is not what the caller would be expecting, and could lead to trouble. So, atomic blocks must always end *before* any return statement is reached.

```
async command result_t LedsArray.setv(uint16_t ledsNum) {
  atomic {
    ledsOn = (ledsNum & 0x3FF);
    output(~ledsOn & 0x3FF);
  }
  return SUCCESS;
}
```

LedsArray.setv() starts off by writing the value of its parameter, ledsNum, into ledsOn—after clearing the upper six bits, as always. It then sends ledsOn's one's complement to output(). In each of the remaining functions, that same argument is passed to output(), but in each case ledsOn is modified in a unique way beforehand.

```
async command result_t LedsArray.inc() {
  atomic {
    ledsOn = (ledsOn+1) & 0x3FF;
    output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }

  async command result_t LedsArray.dec() {
    atomic {
      ledsOn = (ledsOn-1) & 0x3FF;
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }
```

In the inc() and dec() functions, ledsOn is increased or decreased by one, and the low ten bits of the result is stored back in ledsOn.

```
  async command result_t LedsArray.setav(uint16_t turnOn) {
    atomic {
      ledsOn |= (turnOn & 0x3FF);
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }
```

For setav() to turn on some LEDs without affecting the others, the turnOn variable is combined with ledsOn using the bitwise or operator.

```
  async command result_t LedsArray.clrav(uint16_t turnOff) {
    atomic {
      ledsOn &= (0x3FF & ~turnOff);
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }
```

The bitwise or operation is useful for setting bits; the output is 1 unless both inputs are 0. Likewise, the bitwise and operation is useful for clearing bits; the output is 0 unless both inputs are 1. LedsArray.clrav() uses it for just that reason. First, the argument must be inverted, because the bits of turnOff which are 1 are those which denote LEDs to turn off; bits of 0 indicate that no action should be taken regarding their LEDs. By inverting this, LEDs to be turned off will be denoted by a 0. Then when this number is anded with ledsOn, those bits will be cleared.

```
  void output(uint16_t num) {
    PORTC = (PORTC & 0xFC) | (num >> 8);
    PORTA = num & 0xFF;
  }
}
```

Lastly, the `output(num)` function is responsible for writing `num` to the appropriate Ports. To do this, it will write values into the special variables `PORTC` and `PORTA`. These are defined[32] to point to the *output* registers for Ports C and A, just as `DDRC` and `DDRA` point to those ports' *data direction* registers. (The third set of Port registers, the *input* registers, are named `PINC`, `PINA`, and so on for the rest of the Ports.)

The upper byte of `num` will consist of six 0's and the two bits which must go to PC1 and PC0, while the lower byte will all go to Port A. To extract the upper byte, `num` is bit shifted to the right eight places. This is equivalent to dividing `num` by 2^8 (and discarding the remainder). For instance, (0x2C5 >> 8) will yield 0x2.

That procedure obtains the proper values for PC1 and PC0; the other six Port C bits should remain as they were. To achieve this, the upper six bits are extracted from the old value of `PORTC` by anding it with 0xFC (0b11111100). These two disjoint parts[33] are then ored together to produce the new value for `PORTC`.

Finally, the low byte is taken from `num` by anding it with 0xFF and it is then written into `PORTA`.

IntToLedsArray.nc The `IntToLedsArray` (like `IntToLed`) is a modification of `IntToLeds` which uses `LedsArrayC` rather than `LedsC`. The "`IntTo...`" components provide the interface `IntOutput`, defined in **tos/interfaces/***IntOutput.nc*:

```
interface IntOutput {
  command result_t output(uint16_t value);
  event result_t outputComplete(result_t success);
}
```

[32]These are defined in the <*avr/iom128.h*> header file, found in **/usr/local/avr/include/**.

[33]Note: If the upper 6 bits of (`num` >> 8) are not 0, then these two parts will *not* be disjoint, and `num` could affect Port C pins other than PC1 and PC0. This is why those six bits must be cleared before calling `output()`.

The user will command `IntToLedsArray` to output an integer, and, having done so, `IntToLedsArray` will respond by signalling the `outputComplete()` event. `IntToLedsArray` is defined in *IntToLedsArray.nc* as follows:

```
configuration IntToLedsArray
{
  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  components IntToLedsArrayM, LedsArrayC;

  IntOutput = IntToLedsArrayM.IntOutput;
  StdControl = IntToLedsArrayM.StdControl;
  IntToLedsArrayM.LedsArray -> LedsArrayC.LedsArray;
}
```

`IntToLedsArray` claims to provide the interfaces `IntOutput` and `StdControl`, and yet *IntToLedsArray.nc* contains implementations of neither of them. Instead, it *delegates* them to `IntToLedsArrayM` (which also provides these interfaces). Whenever some other module wired to `IntToLedsArray` calls `IntOutput.output()`, the call will actually go to `IntToLedsArrayM` instead.

The configuration then wires the `LedsArray` interface used by `IntToLedsArrayM` to the implementation provided by `LedsArrayC`.

IntToLedsArrayM.nc The last file in the **jona/** directory, *IntToLedsArrayM.nc*, contains the `IntToLedsArrayM` module which the `IntToLedsArray` needs.

```
module IntToLedsArrayM {
  uses interface LedsArray;

  provides interface IntOutput;
  provides interface StdControl;
}
```

As all modules do, `IntToLedsArrayM` starts out by declaring which interfaces it provides and uses. Wirings to these interfaces were made already in *IntToLedsArray.nc*.

```
implementation
{
  command result_t StdControl.init()
  {
    call LedsArray.init();
    call LedsArray.allOff();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }
```

First, IntToLedsArrayM implements the three commands of the StdControl interface. To initialize itself, IntToLedsArrayM makes sure that LedsArray is ready. It has nothing to do for start() or stop(), because its only action is in response to an IntOutput.output() command; it uses no timers or any other components which might have to be explicitly started or stopped.

```
  task void outputDone()
  {
    signal IntOutput.outputComplete(SUCCESS);
  }

  command result_t IntOutput.output(uint16_t value)
  {
    call LedsArray.setv(value);

    post outputDone();

    return SUCCESS;
  }
}
```

When IntToLedsArrayM receives the IntOutput.output(value) command, it tells LedsArray to display value. It then posts the outputDone() task, which in turn will signal[34] the IntOutput.outputComplete() event.

[34]Recall that commands may not directly trigger events. Otherwise, some function in a module may issue a command, which triggers an event, which causes the command to be issued again, and so on, with the original function call never terminating.

Chapter 5

Deployment

Once the JONA Prototyping Kit had been constructed, it needed to be tested to see how it would perform as a WSN mote. Additionally, it had been noted that TinyOS documentation can leave something to be desired, and may not suffice for someone with little or no programming background. To address these hardware and software issues, a number of sample applications were written which would test JONA components and provide examples of how the JONA kit could be used, and how it could use the facilities provided by the base TinyOS distribution.

This chapter reviews several of these applications—`TestLedsArray`, `TestUart-SendReceive`, `OscilloscopeJonaRF`, and `HumiditySense`—pointing out the JONA features which they are intended to demonstrate. Complete listings of each of the applications discussed below are available in Appendix J on page 217.

5.1 LedsArray

Before the LedsArray component could be used to verify proper operation of other modules, its own hardware and software had to be verified. To do so, a program called `TestLedsArray` was written which sets a repeating timer, and then changes the LedsArray display each time the Timer fires.

`TestLedsArray` was modified to test the different `LedsArray` commands (`inc()`, `dec()`, `setv()` and so forth). One particular incarnation, used to test `setv()` and get(), will be examined here. This version of the application was also known as "KnightRider" due to the appearance of the resultant animation[1].

The *Makefile* for KnightRider is as simple as that of HelloWorld:

```
COMPONENT=KnightRider
include ../Makerules
```

The configuration file, however, introduces something new:

```
configuration KnightRider {
}
implementation {
  components Main, KnightRiderM, TimerC, LedsArrayC;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> KnightRiderM.StdControl;

  KnightRiderM.Timer -> TimerC.Timer[unique("Timer")];
  KnightRiderM.LedsArray -> LedsArrayC;
}
```

Notice the wiring for the interfaces used by KnightRiderM—Timer and LedsArray. The latter is wired to a component in the **jona/** platform directory, and is as one would expect. However, the Timer connection is a bit strange. It turns out that TimerC does not provide a single Timer interface; it provides a whole array of them. This lets an application have and use several timers at once. Consequently, each Timer interface used must be wired to a particular element of TimerC's Timer[] array.

Furthermore, each Timer interface used in the application should be wired to a *unique* element of the Timer[] array; otherwise different timers will interfere with each other, because they are using the same underlying Timer. This may seem like a simple problem to solve; merely use a different index number each time something is wired to TimerC; wire the first to Timer[0], then Timer[1], and so on. However, other system components could also be using timers, and may have already

[1]Knight Rider © MCA Universal City Studios.

claimed Timer[1]. Clearly, hard-coding a particular number for the Timer[] index is dangerous at best.

To avoid this difficulty, TinyOS provides the unique() function. unique() is designed so that each time it is called with a particular argument—in this case, "Timer"—it will return a new number—for that argument. So, two calls to unique("Timer") are guaranteed to yield distinct results, but one call to unique("Timer") and another to unique("T") *could* return the same number. To keep things simple, TinyOS convention says that the argument to unique() should be the name of the interface for which it is being called (in double quote marks). So, the KnightRider configuration file wires KnightRiderM's Timer to TimerC.Timer[unique("Timer")], and it can know for certain that it will have its own Timer, courtesy of TimerC.

KnightRiderM.nc starts out, as always, with a listing of the interfaces it uses and provides.

```
module KnightRiderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface LedsArray;
  }
}
```

There are two ways to list interfaces provided or used by a module. Each interface listed can be individually preceded by provides or uses, as they are in *IntToLedsArrayM.nc* (see Section 4.2.4 on page 105); or, the two groups can be contained in blocks delimited by curly braces, as they are here (and as they were in HelloWorldM; cf. Section 4.2.3 on page 83). The nesC compiler makes no differentiation between the two styles, so it is largely a matter of personal preference. When a large number of interfaces are being used or provided, encasing them in blocks like this can make for a more readable source file.

```
implementation {
```

```
uint8_t dir = 0;

command result_t StdControl.init() {
  call LedsArray.init();
  return call LedsArray.setv(1);
}

command result_t StdControl.start() {
  // Start a repeating timer that fires every 100ms
  return call Timer.start(TIMER_REPEAT, 100);
}

command result_t StdControl.stop() {
  return call Timer.stop();
}
```

The implementation of `KnightRiderM` begins with the definition of a `uint8_t` variable called `dir` which will keep track of the direction of motion of the lit LED. It then implements the three `StdControl` functions. `StdControl.init()` initializes the `LedsArray` and sets its initial value to 1, which will turn on the LED at the low end of the array.

`StdControl.start()` sets up a `Timer` that will repeatedly fire at 100 millisecond intervals. Notice the line beginning with "//". Two consecutive slashes indicates to the compiler that the rest of that line contains a *comment*[2], and should be skipped over. Comments are useful for marking sections which need further attention later, or for providing a description in words of what nearby code is supposed to be doing. This latter purpose is highly valuable, because it makes it much easier for someone else to read and understand one's code; unfortunately, the TinyOS distribution files tend to be somewhat sparse in their commentary. Examine the code listings in the appendices for more extensive use of comments.

To start the repeating timer, `Timer.start()` is called with the two arguments `TIMER_REPEAT` and `100`. The latter is the timer interval in so-called *binary milliseconds*, units of time equal to 1/1024 of a second. The first argument indicates the type of timer this shall be, either `TIMER_ONE_SHOT` or `TIMER_REPEAT`.

`StdControl.stop()` calls `Timer.stop()` to stop the repeating timer. It is impor-

[2]It is also possible to make comments which span multiple lines by putting `/*` at the start and `*/` at the end.

tant to note that care must be taken when using a command as a return value. In this case, `Timer.stop()` is certain always to provide a value of `SUCCESS`. However, `Timer.stop()` will return `FAIL` if no timer is running (such as if a `TIMER_ONE_SHOT` `Timer` had already fired before `stop()` were called). If this failure is tolerable and not indicative of a problem, then `StdControl.stop()` (or whatever the relevant function happens to be) should have a body like:

```
    call Timer.stop();
    return SUCCESS;
```

This would isolate `Timer`'s failure and prevent anything else from seeing it. In fact, even this implementation of `StdControl.stop()` would not see it, because `Timer.stop()`'s return value is discarded. As an alternative, the return value could be stored in a local variable of type `result_t`, or it could be used as the test of an if statement (`if( call Timer.stop() ) {...} else {...}`).

```
  event result_t Timer.fired() {
    uint16_t cnt = call LedsArray.get();

    if( !dir && (cnt < 0x200) ) {
      cnt <<= 1;
    } else if( dir && (cnt > 0x1) ) {
      cnt >>= 1;
    } else {
      dir ^= 0x1;
    }
    return call LedsArray.setv(cnt);
  }
}
```

The `Timer.fired()` event handler is where the real action takes place. Each time the timer fires, this function recalls the old value being displayed by the LedsArray, updates it appropriately, and then gives the LedsArray the new value to display.

The first time the timer fires, `cnt` will hold a value of 0x1 and `dir` will be 0. So, the first `if` statement will evaluate to true: This test "`!dir`" means "NOT `dir`"; it will be TRUE when `dir` is zero (i.e. FALSE), and FALSE when `dir` is nonzero (i.e. TRUE). The second test checks whether `cnt` is less than 0x200. These results are

Table 5.1: The logical NOT (!) operator.

Input	Output
FALSE	TRUE
TRUE	FALSE

Table 5.2: The logical AND (&&) operator.

Input 1	Input 2	Output
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

combined by the logical AND operator, which evaluates to TRUE when both its left and right sides are TRUE. There is also a logical OR operator, which evaluates to TRUE when either operand is TRUE.

So, the first `if` statement's test will pass and its body will be executed. `cnt` will be left-shifted by one bit position. This is the same as multiplying `cnt` by a factor of 2. So, the LedsArray will display the sequence of numbers shown in Table 5.4.

At this point, the first test will fail, because (0x200 $<$ 0x200) is FALSE. Now consider when `dir` has been changed. Then the second `if` test will be true, because `dir` will be nonzero, and 0x200 is greater than 0x1. So then `cnt` will be right-shifted by one bit position (divided by 2), and the LedsArray will display the reverse of the previous sequence, shown in Table 5.5.

Table 5.3: The logical OR (||) operator.

Input 1	Input 2	Output
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Table 5.4: LedsArray display sequence while `dir` is FALSE.

Hex	Binary
0x001	0b0000000001
0x002	0b0000000010
0x004	0b0000000100
0x008	0b0000001000
0x010	0b0000010000
0x020	0b0000100000
0x040	0b0001000000
0x080	0b0010000000
0x100	0b0100000000
0x200	0b1000000000

Table 5.5: LedsArray display sequence while `dir` is TRUE.

Hex	Binary
0x200	0b1000000000
0x100	0b0100000000
0x080	0b0010000000
0x040	0b0001000000
0x020	0b0000100000
0x010	0b0000010000
0x008	0b0000001000
0x004	0b0000000100
0x002	0b0000000010
0x001	0b0000000001

Table 5.6: The bitwise exclusive-or (^) operation.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Then the second test will fail (0x1 is not greater than 0x1), and it will be time to reverse directions again. To perform this reversal, the statement

```
dir ^= 0x1;
```

is executed. This exhibits the bitwise *exclusive-or* (xor) operator. For each bit position, the output will be 1 when *exactly* one of its inputs is 1. If they are both 1, or if they are both 0, then the output will be 0. So, when `dir` is 0, then (`dir ^ 0x1`) will be 0x1, and `dir` will become nonzero. When `dir` is 0x1, then (`dir ^ 0x1`) will be 0x0, and `dir` will again be zero.

Notice that in `KnightRiderM`, `dir` is a global variable and yet it is read and written without using `atomic` blocks. This is acceptable because `Timer.fired()` is the only function to access the variable. The nesC compiler realizes this and does not issue a warning.

Similar programs were written to test the other `LedsArray` commands as implemented by `LedsArrayC`. For instance, to test `inc()`, the `Timer.fired()` handler was implemented as follows:

```
event result_t Timer.fired() {
  return call LedsArray.inc();
}
```

Following these tests, the LedsArray was certified as being fully operational, and dependable for testing other components and applications.

5.2 UART: Bidirectional Communication

The HelloWorld application demonstrated that the JONA UART was capable of sending data to a computer. It remained to be seen, however, whether it was also capable of receiving. To remedy this, a program called `TestUartSendReceive` was written which would receive bytes from the UART, modify them in some recognizable way, and then send them back.

In particular, `TestUartSendReceive` adds one to each received byte. When used in conjunction with a terminal program like Windows's *HyperTerminal*, this has the effect that whenever a letter is typed, the next letter alphabetically[3] is displayed. For instance, typing "Hello" will produce the output "Ifmmp".

```
configuration TestUartSendReceive {
}
implementation {
  components Main, HPLUARTC, TestUartSendReceiveM, LedC;

  Main.StdControl -> TestUartSendReceiveM;

  TestUartSendReceiveM.HPLUART -> HPLUARTC;
  TestUartSendReceiveM.Led -> LedC;
}
```

The configuration file for `TestUartSendReceive` wires its module to `HPLUARTC` through the `HPLUART` interface. Recall that `HelloWorld` made use of the more abstract `ByteComm` interface, which can be used with the UART or the radio. Here, where it is specifically the UART which is being tested, that abstraction has been removed and the UART's hardware presentation layer (HPL) is used directly. With the HPL verified, other applications can safely use `ByteComm`—the preferred UART interface—knowing that the underlying hardware is properly configured.

JONA inherits the `HPLUARTC` component from the **avrmote/** platform directory. The contents of *HPLUARTC.nc* are as follows:

[3]Actually, what is displayed is the next symbol in the ASCII encoding scheme. See Appendix C on page 153.

```
configuration HPLUARTC {
  provides interface HPLUART as UART;
}
implementation
{
  components HPLUARTM;

  UART = HPLUARTM;
}
```

Notice the `provides` line. It says that it provides the interface `HPLUART` *as UART*. Interfaces provided or used can be arbitrarily renamed using the `as` keyword. This can enable a component to use or provide an interface multiple times, by assigning each instance a unique name. `HPLUARTC` defers the implementation of the interface to `HPLUARTM`, and *HPLUARTM.nc*—which is one of the files found in the **jona/** platform directory—uses[4] this alias throughout its implementation. Finally, note that this is the `HPLUART` interface and *not* the `UART` component.

So, `HPLUARTC`'s listings imply that `TestUartSendReceive` will actually be using the **jona/** directory's *HPLUARTM.nc* to communicate through the UART. This is precisely as it should be, since it is this file which needs testing.

```
module TestUartSendReceiveM {
  provides {
    interface StdControl;
  }
  uses {
    interface Led;
    interface HPLUART;
  }
}
```

TestUartSendReceiveM.nc begins by declaring the interfaces which the configuration file wired for it. Note that `TestUartSendReceiveM` does not rename the `HPLUART` interface. This demonstrates that the provider and user of an interface do not have to call it the same thing.

```
implementation {
```

[4]cf. the *HPLUARTM.nc* listings in Section I.11 on page 197.

```
  command result_t StdControl.init() {
    call HPLUART.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }

  async event result_t HPLUART.get(uint8_t data) {
    uint8_t response = data+1;
    call Led.Toggle();
    call HPLUART.put(response);
    return SUCCESS;
  }

  async event result_t HPLUART.putDone() {
    return SUCCESS;
  }
}
```

`TestUartSendReceiveM` uses `StdControl.init()` to initialize the UART. It then sits idle until the serial port receives a byte. This reception will trigger the `HPLUART.get()` event, in response to which the module will spring into action. It will prepare its response, toggle the onboard LED, and transmit its reply. It will then go back to being idle until another byte is received.

This test application, as simple as it was, was sufficient to demonstrate that the JONA has a working UART. It would not be difficult to expand `TestUartSendReceive` into a program providing more meaningful interaction between the mote and the computer. It could test the received byte and take an action which depended on its value. For example, the following will capitalize all lowercase letters, and leave all other characters untouched.

```
  async event result_t HPLUART.get(uint8_t data) {
    uint8_t response = data;
    if( 'a' <= response && response <= 'z' ) {
      response += 'A' - 'a';
    }
    call Led.Toggle();
    call HPLUART.put(response);
    return SUCCESS;
  }
```

First, the function checks[5] whether `response` is between `'a'` and `'z'`. If so, then it adds to `response` the (constant) offset between the lowercase and uppercase letters, (`'A'-'a'`). For this to work, it is necessary that the lowercase letters be represented by consecutive values, increasing from `'a'` to `'z'`, and that the same is true of the uppercase letters. Note that this is indeed the case for ASCII (cf. Tables C.3 and C.4 in Appendix C). In particular, A–Z is in 65–90 and a–z is in 97–122. To convert to uppercase, -32 must be added to each lowercase letter.

```
  async event result_t HPLUART.get(uint8_t data) {
    if( data == 'Y' || data == 'y' ) {
      call Led.On();
    } else if( data == 'N' || data == 'n' ) {
      call Led.Off();
    }
    return call HPLUART.put(data);
  }
```

Consider this version of `HPLUART.get()`. Whenever the received byte is the letter *y* (either case), it turns on the LED, and whenever it is *n*, it turns it off. All other letters and symbols are ignored. Now the UART is not merely swapping one character for another; it is providing a way to issue commands to a mote from a computer. With appropriate modifications, this mote could forward the command over its radio, and the command could then be disseminated throughout an entire network. Of course, this assumes that the radio is capable of sending information.

5.3 ADC & RFM

The test applications discussed so far have failed to address the two most important parts of a wireless sensor node: the transceiver and the sensor. To verify the func-

[5]The sequence `<=` represents *less than or equal to* (i.e. $\leq$). Other numerical comparisons which can be used as TRUE/FALSE tests in programs are *less than* (<), *greater than* (>), *greater than or equal to* (>=), *equal to* (==), and *not equal to* (`!=`). See [27].

tionality of the radio and the ADC, the application `OscilloscopeRF` was adapted to the JONA platform, as a new program called `OscilloscopeJonaRF`.

The `Oscilloscope` programs are a set of TinyOS applications which take sensor readings and then send them in groups called *packets* to the computer where they are displayed by a virtual oscilloscope in the form of a Java program. `Oscilloscope` is intended to be used when the mote is directly connected to the PC, as it sends its sensor readings out through the UART. `OscilloscopeRF`, on the other hand, sends its readings over the radio. Another mote, running the `TOSBase` program, acts as a base station, receiving the radio packets and forwarding them to the computer through its serial port.

`OscilloscopeRF` will compile, install, and run on the JONA platform as is, but it takes advantage of the separate LEDs of the `Leds` component; as these all map to the same physical LED on a JONA, the program was modified to use the LedsArray instead. Additionally, the sensor was changed from the photosensor to the accelerometer[6]. This latter change was motivated by the idea that the accelerometer would provide a more appreciable demonstration of the relationship between the factor being observed and the resultant ADC reading; seeing the graph change in response to the accelerometer being shaken is more readily understandable (and, frankly, more interesting) than seeing it be affected by changes in lighting.

As mentioned above, this scenario involves two motes and a PC. The software running on each of these three devices is discussed, respectively, in Sections 5.3.1, 5.3.2, and 5.3.3.

[6]In particular, the accelerometer used was the ADXL311EB evaluation board from Analog Devices [2]. This printed circuit board features their ADXL311 accelerometer chip [1] and accompanying circuitry, with a 5-pin header providing access to all power and signal lines (see Figure 5-1).

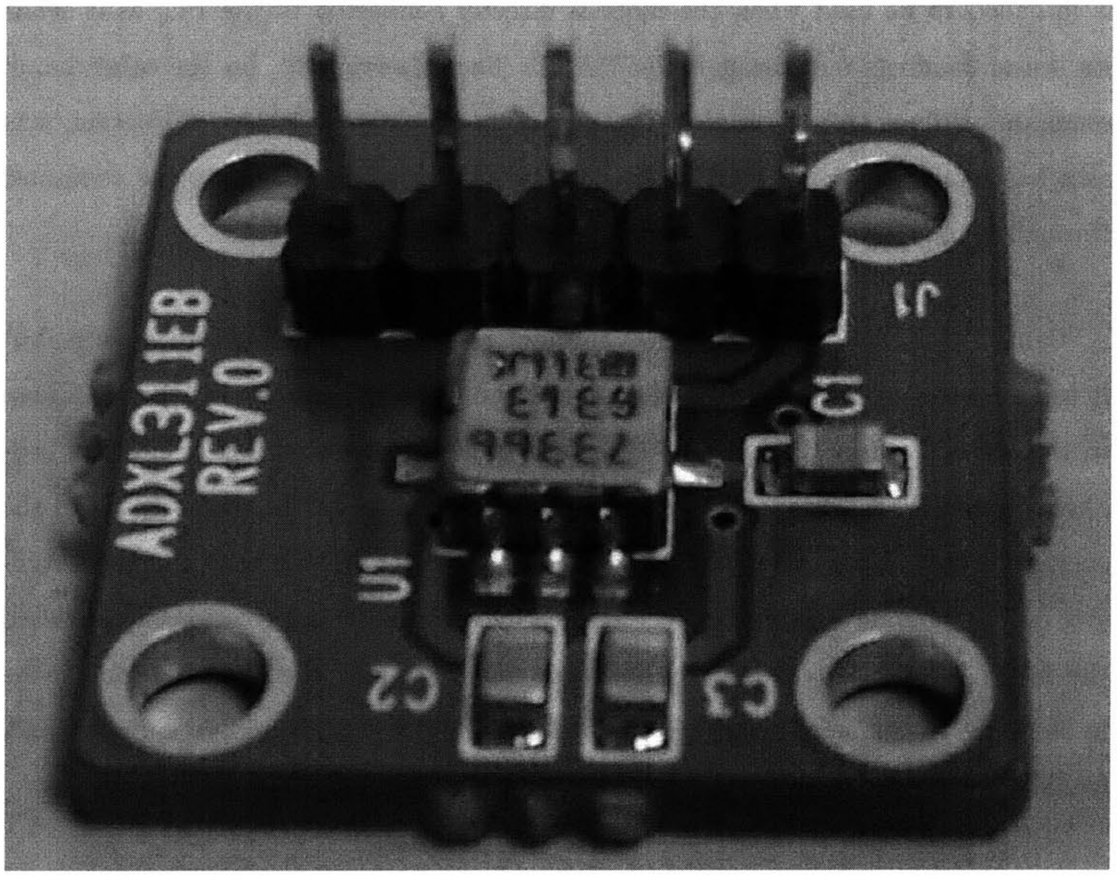

Figure 5-1: ADXL311EB accelerometer evaluation board [2].

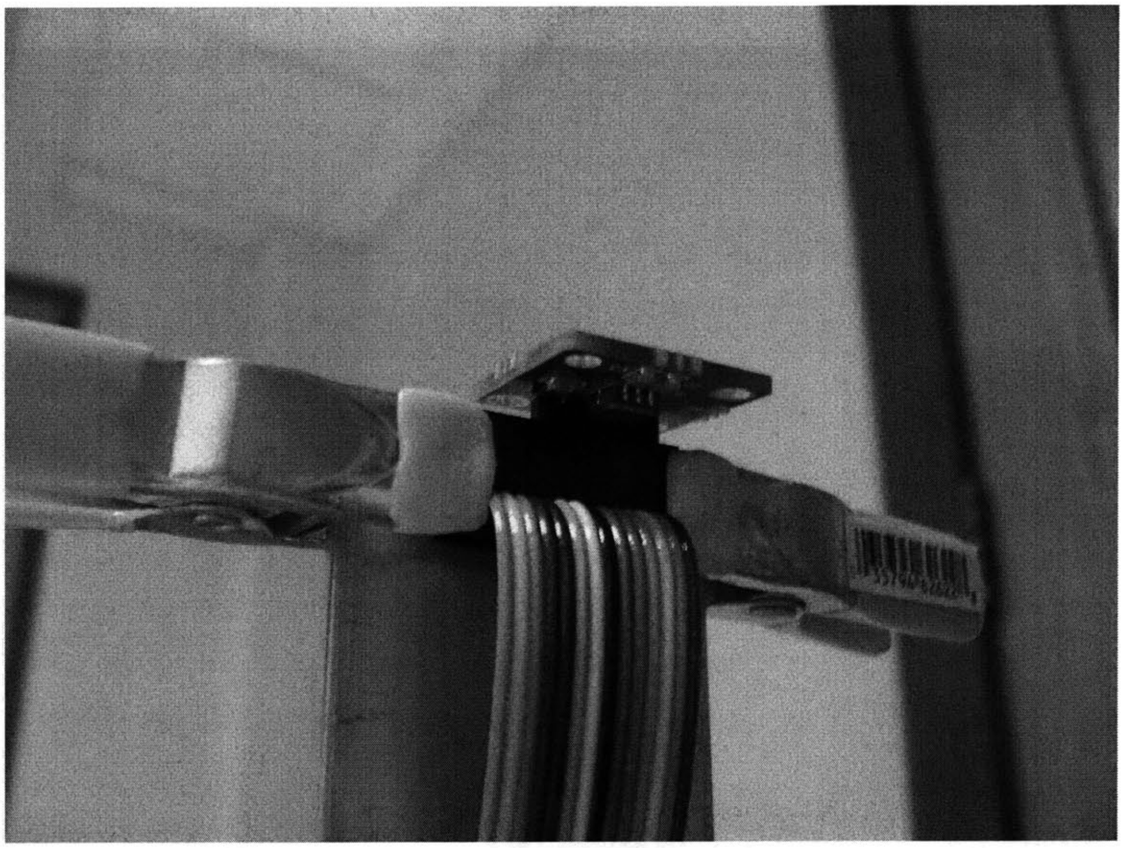

Figure 5-2: ADXL311EB mounted atop the cantilever.

5.3.1 Collecting & Transmitting Readings with Oscilloscope-JonaRF

The mote with the accelerometer[7] runs `OscilloscopeJonaRF`. This program is found in its entirety in Appendix J beginning on page J.3.

Perhaps the first noticeable difference of this program with respect to the others discussed so far is that its *Makefile* has an extra line:

```
COMPONENT=OscilloscopeJonaRF
PFLAGS=-I../Oscilloscope
include ../Makerules
```

The `PFLAGS` variable holds command line arguments to pass to the nesC compiler.

[7]The accelerometer was mounted to the top of a flexible metal cantilever which could swing back and forth. See Figures 5-2 and 5-3.

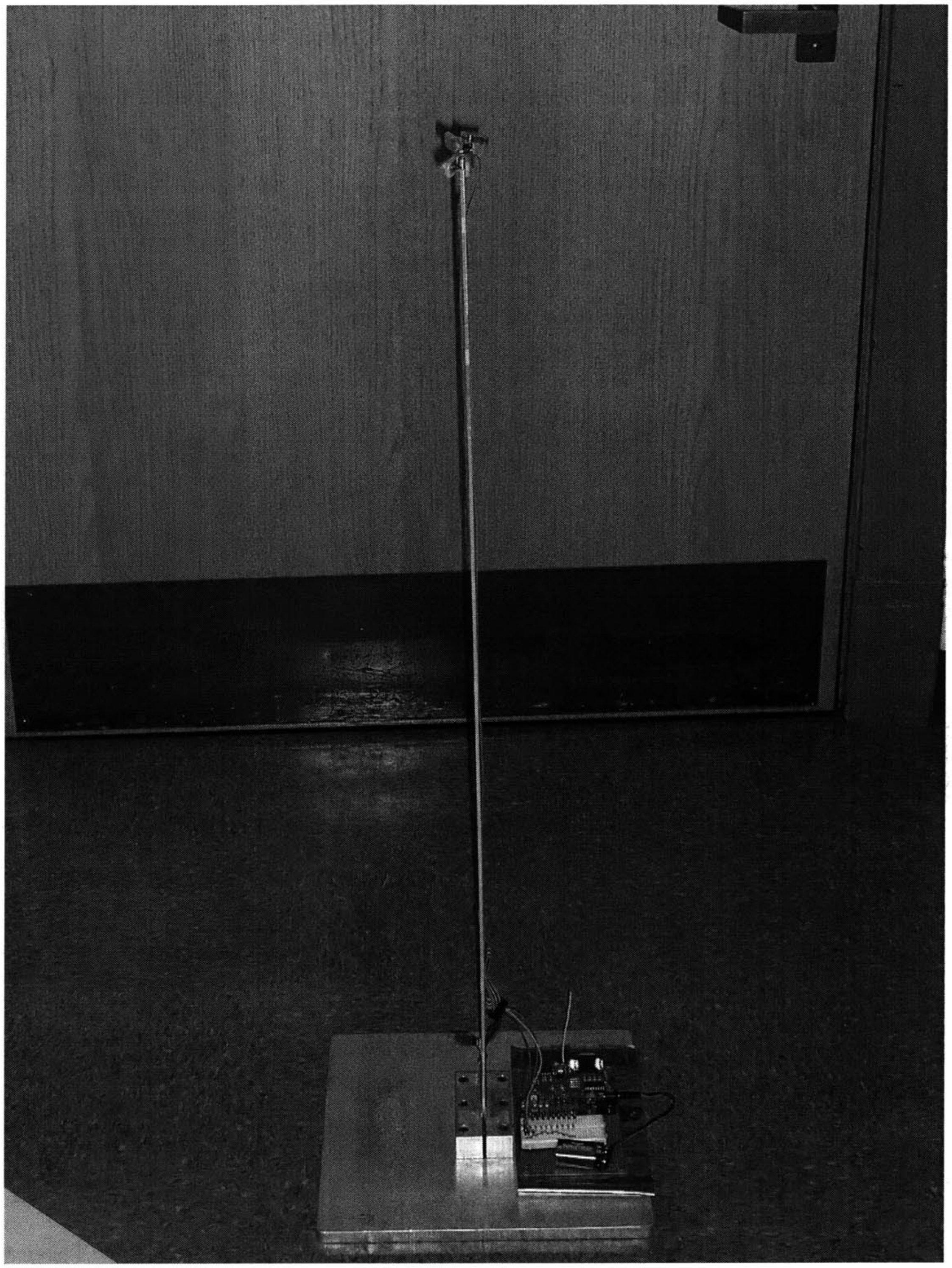

Figure 5-3: The cantilever with the JONA mote at the bottom.

Makerules will add others, but here it is set to contain the argument `-I../Oscilloscope`. The `-I` flag is an *include directive*, telling the compiler to look in the named directory[8] when searching for included header and source files during compilation. So, this line of *Makefile* says to the compiler that it should be sure to check **../Oscilloscope/** (i.e. **apps/Oscilloscope/**) for header files.

This is, of course, the directory containing the original `Oscilloscope` application. It is included here because `OscilloscopeJonaRF` needs to borrow one of the files found there: *OscopeMsg.h*. This header file provides a description of the packets which the `Oscilloscope` programs use to package up and transmit their readings. `OscilloscopeJonaRF` adopts this file from the **../Oscilloscope/** directory, rather than having to duplicate it in its own.

Having told the compiler where to look for it, `OscilloscopeJonaRF` actually uses *OscopeMsg.h* in the following line, found at the top of both *OscilloscopeJonaRF.nc* and *OscilloscopeJonaRFM.nc*:

```
includes OscopeMsg;
```

When the compiler encounters that line, it will look for and read *OscopeMsg.h* before continuing. So, the rest of the file may safely refer to the contents of *OscopeMsg.h*, knowing that the compiler will understand those references.

OscopeMsg.h contains the following definitions and declarations:

```
enum {
  BUFFER_SIZE = 10
};

struct OscopeMsg
{
  uint16_t sourceMoteID;
  uint16_t lastSampleNumber;
  uint16_t channel;
  uint16_t data[BUFFER_SIZE];
};

struct OscopeResetMsg
```

[8]In addition to, not instead of, the locations normally checked.

```
{
  /* Empty payload! */
};

enum {
  AM_OSCOPEMSG = 10,
  AM_OSCOPERESETMSG = 32
};
```

In addition to defining the constants BUFFER_SIZE, AM_OSCOPEMSG and AM_OSCOPE-RESETMSG, it declares data structures called OscopeMsg and OscopeResetMsg. O-scopeMsg represents the packets which the Oscilloscope program will use to send its sensor readings. In addition to the ten-element data array which will hold the readings themselves, each OscopeMsg will keep track of which mote it is from and which ADC channel the readings are from[9]. It will also include a sequence counter, so that the PC can keep track of the samples in order and detect when a message is lost. OscopeResetMsg is a message which can be sent back from the PC which will instruct the mote to reset this sequence number back to 0. There is no information to transmit for this message, so its data structure is empty.

The configuration file of OscilloscopeJonaRF does not need to know about the particulars of how these data structures are defined, but it does make use of the AM_OSCOPEMSG and AM_OSCOPERESETMSG constants:

```
configuration OscilloscopeJonaRF { }
implementation
{
  components Main, OscilloscopeJonaRFM, TimerC, LedC, Accel,
             GenericComm as Comm, LedsArrayC;

  Main.StdControl -> OscilloscopeJonaRFM;
  Main.StdControl -> TimerC;

  OscilloscopeJonaRFM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeJonaRFM.Led -> LedC;
  OscilloscopeJonaRFM.SensorControl -> Accel;
  OscilloscopeJonaRFM.ADC -> Accel.AccelX;
  OscilloscopeJonaRFM.CommControl -> Comm;
  OscilloscopeJonaRFM.ResetCounterMsg -> Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeJonaRFM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
```

[9]At least, the channel field *could* be used for this. The Oscilloscope programs actually just set its value to 1.

```
  OscilloscopeJonaRFM.LedsArray -> LedsArrayC;
}
```

Notice that, like `TimerC`, `GenericComm` provides arrays of interfaces. However, in this case it is not the goal to have a unique instance of the interface, but rather the particular one corresponding to a specific number. An application could be written which would send and receive a number of different types of messages; the `ReceiveMsg` and `SendMsg` indices specify which type of message each interface should handle. In this case, the `ResetCounterMsg` interface of `OscilloscopeJonaRFM` is assigned to take care of `AM_OSCOPERESETMSG` messages, while its `DataMsg` interface deals in `AM_OSCOPEMSG` packets.

The rest of the wiring should be familiar. ADC readings will be taken from the channel assigned to the x-axis of the (2D) accelerometer, `LedC` and `LedsArrayC` will be used to provide the `Led` and `LedsArray` interfaces, and `OscilloscopeJonaRFM` will control its own `Timer`.

```
module OscilloscopeJonaRFM
{
  provides interface StdControl;
  uses {
    interface Timer;
    interface Led;
    interface StdControl as SensorControl;
    interface ADC;
    interface StdControl as CommControl;
    interface SendMsg as DataMsg;
    interface ReceiveMsg as ResetCounterMsg;
    interface LedsArray;
  }
}
```

The `OscilloscopeJonaRFM` module starts out listing its interfaces. Notice that two of the interfaces used are `StdControl`; they have been renamed so that it can unambiguously use them both. The rest of this discussion will focus on a few excerpts from *OscilloscopeJonaRFM.nc* which are significant to understanding *what* the program does. The parts omitted here (which can be found in Appendix J) show *how* it does what it does, but they do not provide any particularly useful insight.

The timer is set (in the `StdControl.start()` function) to repeatedly fire every 125 milliseconds. Each time it fires, it commands the ADC to take a reading:

```
event result_t Timer.fired() {
  return call ADC.getData();
}
```

When the ADC is finished with its conversion, it signals the `ADC.dataReady()` event. The handler for this event puts the new data in the next available slot in the `OscopeMsg`'s `data` array, and posts a task called `dataTask()` which will transmit the packet if the array is full. It then executes this sequence (`data` is the new reading, and `ledsArray` is a `uint16_t`):

```
    if( data < 349 ) {
      ledsArray = 0x001;
    } else if( data < 390 ) {
      ledsArray = 0x003;
    } else if( data < 431 ) {
      ledsArray = 0x007;
    } else if( data < 472 ) {
      ledsArray = 0x00F;
    } else if( data < 513 ) {
      ledsArray = 0x01F;
    } else if( data < 553 ) {
      ledsArray = 0x03F;
    } else if( data < 594 ) {
      ledsArray = 0x07F;
    } else if( data < 635 ) {
      ledsArray = 0x0FF;
    } else if( data < 676 ) {
      ledsArray = 0x1FF;
    } else {
      ledsArray = 0x3FF;
    }
    call LedsArray.setv( ledsArray );
```

The range of values which `data` can take on is divided[10] into blocks, and each block turns on one more LED than the last. This lets the LedsArray act like a "volume indicator" for the ADC. If `data` is small, then only a few LEDs will be illuminated; if `data` is larger, then correspondingly more LEDs will be turned on.

Finally, `dataTask()` sends the message by calling `DataMsg.send()`.

[10]The cutoffs between blocks may seem arbitrary. They were based on a "typical" range of values observed as the accelerometer was being moderately shaken back and forth on the cantilever.

5.3.2 Receiving Readings with TOSBase

The message is then received by another mote running `TOSBase`. This application works "out of the box" with no modifications for JONA, and so it will not be examined in detail. All `TOSBase` does is to forward packets between the radio and the UART. It can be used with any message type, not just `OscopeMsg`, and it provides a bidirectional link (messages received by the radio are sent over the UART, and messages received at the UART are sent over the radio). So, after passing through `TOSBase`, the `OscopeMsg` finds itself at the serial port of the PC.

5.3.3 Displaying Readings with Java

The PC must be running two programs in order to display the sensor readings on the oscilloscope. The *SerialForwarder* program is invoked by the command `java net.tinyos.sf.SerialForwarder -comm serial@COM1:9600`, which tells it to connect to the serial port with a baud rate of 9600 (the rate for which JONA is configured[11]). It then links the serial port to a TCP/IP port, similar to how `TOSBase` linked the radio and UART. Other programs, running on this same PC or on another machine linked by the Internet, can then communicate with the WSN network through this port.

One such program is the Java *Oscilloscope* program, invoked by the command `java net.tinyos.oscope.oscilloscope`. This will connect to *SerialForwarder* and graph the incoming sensor readings (see Figure 5-4). For more information on these Java utilities, refer to TinyOS Tutorial Lesson 6 [49], and the Java source code, found in **tools/java/**.

5.4 Constructing an ADC Input

The accelerometer used in the previous section was a device which produced as its output a voltage between 0 and V_{CC} which represented its current state of acceler-

[11]cf. the description of *HPLUARTM.nc* on page 95.

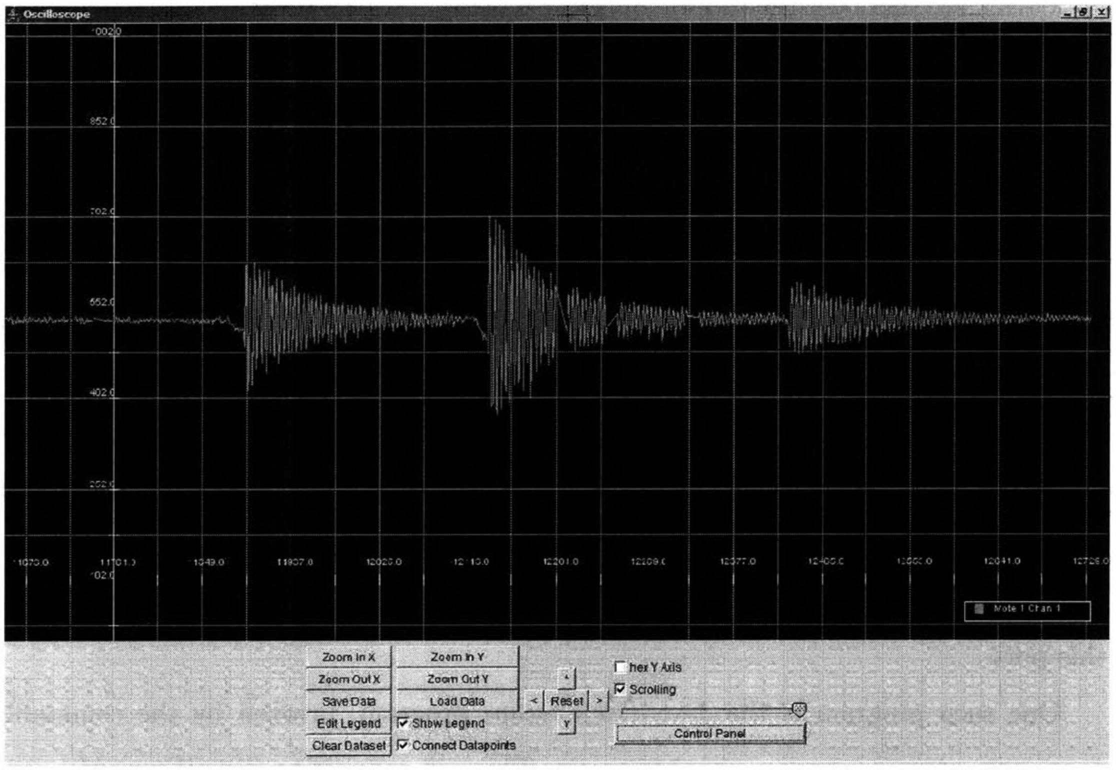

Figure 5-4: Screenshot of the Java Oscilloscope, showing accelerometer sensor readings.

ation. In general, however, a sensor's response to its stimulus could be manifested by a change in resistance or capacitance, or any other sort of electrical property. It may then be necessary to construct a circuit which will translate this property into an analog voltage which can be measured by the ADC.

For instance, a thermistor is a device whose resistance depends on its temperature. As previously mentioned, a sensible way to use a thermistor with an ADC would be to use it as one of the resistors in a voltage divider. Recall from Section 2.1.4 that the relationship between the input and output voltages and the resistances of the two resistors is given by $v_{out} = v_{in}R_2/(R_1 + R_2)$ (Equation 2.24). In that discussion, v_{out} was taken to be a function of v_{in}, with constant R_1 and R_2; however, the equation can be equally understood to be a function of R_1 with a constant R_2 and a constant V_{in} (say, V_{CC}).

Note that when $R_1 = R_2$, $v_{out} = V_{in}/2$. This implies that for v_{out} to be roughly centered between 0 and V_{in}, R_2 should be chosen to be roughly the average value that R_1 is expected to take on. Consider Table 5.7. These values were taken from an ordinary thermistor from RadioShack. By inspection, a decent value for R_1 would be 10k.

So, the thermistor is placed as R_1 in a voltage divider configuration (see Figure 2-8) with a 10k resistor, using V_{CC} as the input and connecting the output to a channel of the ADC. The next task is to determine how the voltage will change as a function of the temperature. One option would be to take Table 5.7, determine what v_{out} will be for each temperature, and write the table into the program. Then when a voltage is converted, look up in the table to see what the nearest temperature is (e.g. The voltage is measured to be 2.48V. The nearest value will be 2.5V which occurs when the temperature is 25°; so, for the voltage to be 2.48V, the temperature must be slightly less than 25°.). This will work, but it will be slow, because it will take time to compare the measurement to each table value to find the nearest one (even if it performs a binary search on the table).

It would be much better to determine explicitly the relationship between the temperature and the voltage. That way, when a voltage is measured, the temperature

Table 5.7: Thermistor Characteristics

Temperature (°C)	Resistance (kΩ)	Temperature (°C)	Resistance (kΩ)
−50	320.2	35	6.941
-45	247.5	40	5.826
-40	188.4	45	4.912
-35	144	50	4.161
-30	111.3	55	3.537
-25	86.39	60	3.021
-20	67.74	65	2.589
-15	53.39	70	2.229
-10	42.45	75	1.924
-5	33.89	80	1.669
0	27.28	85	1.451
5	22.05	90	1.366
10	17.96	95	1.108
15	14.68	100	0.9375
20	12.09	105	0.8575
25	10	110	0.7579
30	8.313		

can be calculated quickly. Unfortunately, the relationship (shown in Figure 5-5) is quite nonlinear.

However, the range of temperatures indicated in Table 5.7 and Figure 5-5 is probably larger than the sensor is likely to experience. Consider the section of the graph between -15 and 40°C (5–104°F), shown in Figure 5-6. On this range of temperatures, the relation is almost linear. If it is known that the temperature being sensed will be limited to this range (e.g. if the sensor is measuring room temperature), then approximating it with a line or piecewise-linear function will suffice for most applications.

5.5 Sensing Without the ADC

Sometimes, converting a sensor's output into a steady voltage for the ADC can be a nontrivial task. For example, the Humirel HS1101 humidity sensor [25] is a variable capacitor. Recall from Section 2.1.5 that the current-voltage relationship for a capacitor is $i_C = C dv_C/dt$ (Equation 2.25). If the voltage across the capacitor is constant,

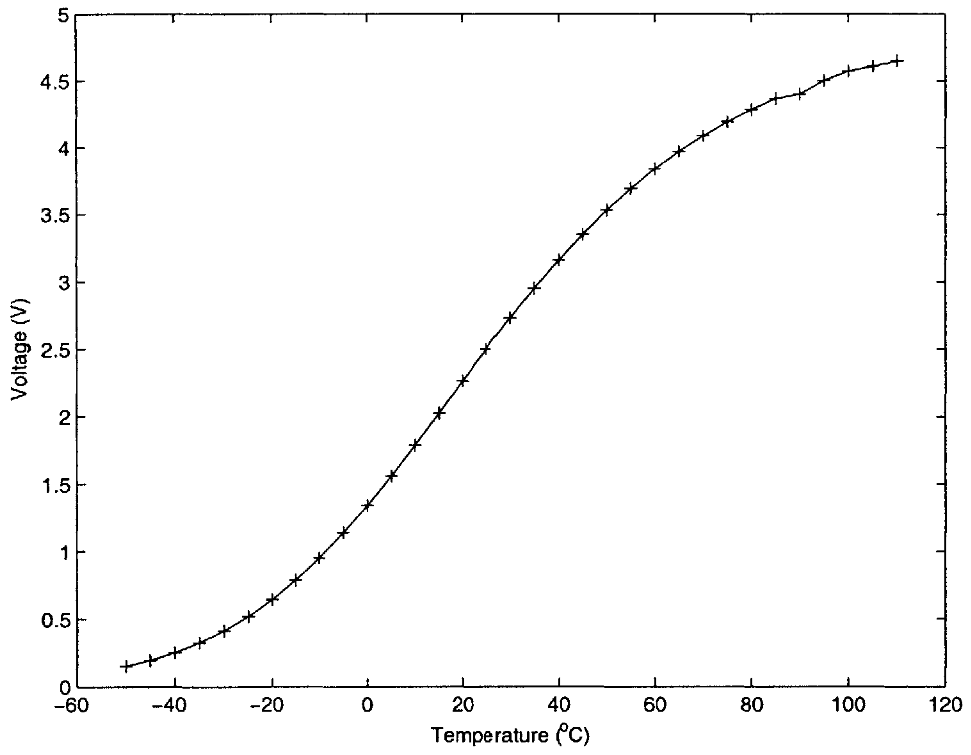

Figure 5-5: Voltage vs. Temperature

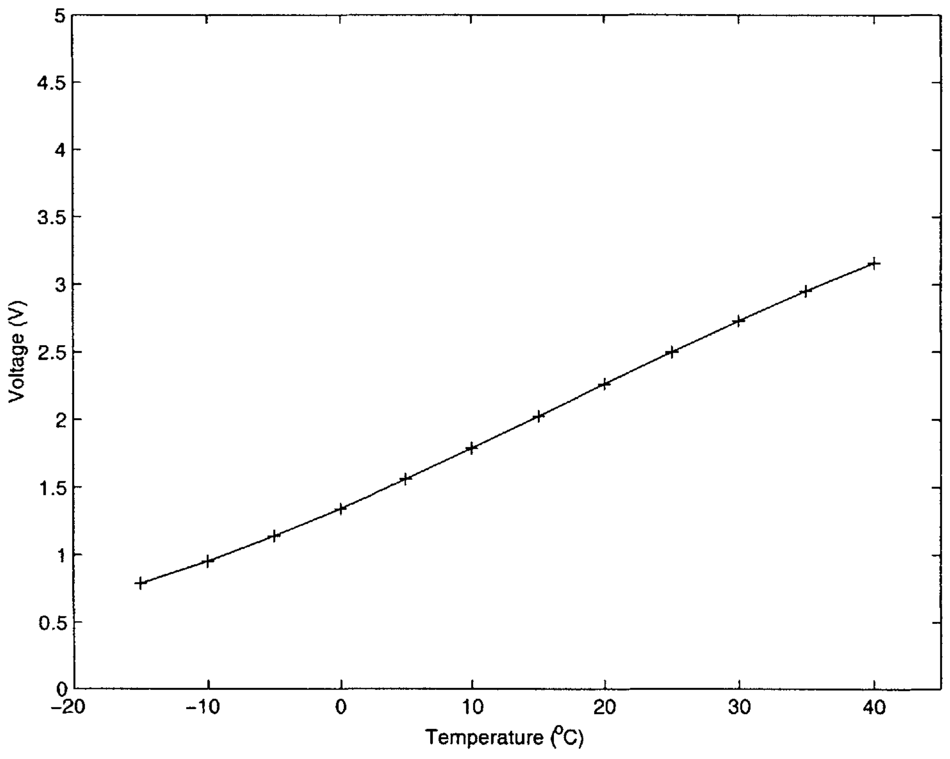

Figure 5-6: Voltage vs. Temperature: -15–40°

then the current through the capacitor will be 0; but this is true regardless of the value of C, so applying or reading a constant voltage across the capacitor will provide no information regarding the value of its capacitance. On the other hand, if a sinusoidal voltage is applied to the capacitor, then the current will also be sinusoidal, with its amplitude proportional to the capacitance. The problem has now become that of converting an oscillating current into a steady analog voltage (as well as generating the original sinusoidal voltage signal). This is still far from a simple task.

The HS1101 data sheet [25] proposes an alternative course of action: page 3 provides a schematic (see Figure 5-7) for a circuit, built around the TLC555 timer from Texas Instruments [46], which will generate a 0–V_{CC} square wave[12] whose frequency is a function of the capacitance. This square wave signal can be connected to an input pin, and then the microcontroller can count the length of an oscillation to determine the HS1101's capacitance. `HumiditySense` is an application which does just that.

`HumiditySense` takes a reading from the humidity sensor circuit (see Figure 5-8), converts it into an integer representing the relative humidity in hundredths of a percent, and then sends this number to the LedsArray and the radio using the `IntOutput` interface provided by `IntToLedsArray` and `IntToRfm`. The latter is found in the **lib/Counters/** directory, for which the *Makefile* adds an include directive:

```
COMPONENT=HumiditySense
PFLAGS=-I%T/lib/Counters
include ../Makerules
```

The nesC compiler will replace `%T` by the TinyOS system directory, so that the line is interpreted as something like "`-I/opt/tinyos-1.x/tos/lib/Counters`" [36]. Using the `%T` abbreviation enables the *Makefile* to refer to the **tos/** directory without having to know or worry about where it actually is. Note that the location of *Makerules* must be specified directly, because this line is read and interpreted by *make* and not *ncc*.

[12]A square wave is a periodic signal which alternates between a steady low value and a steady high value.

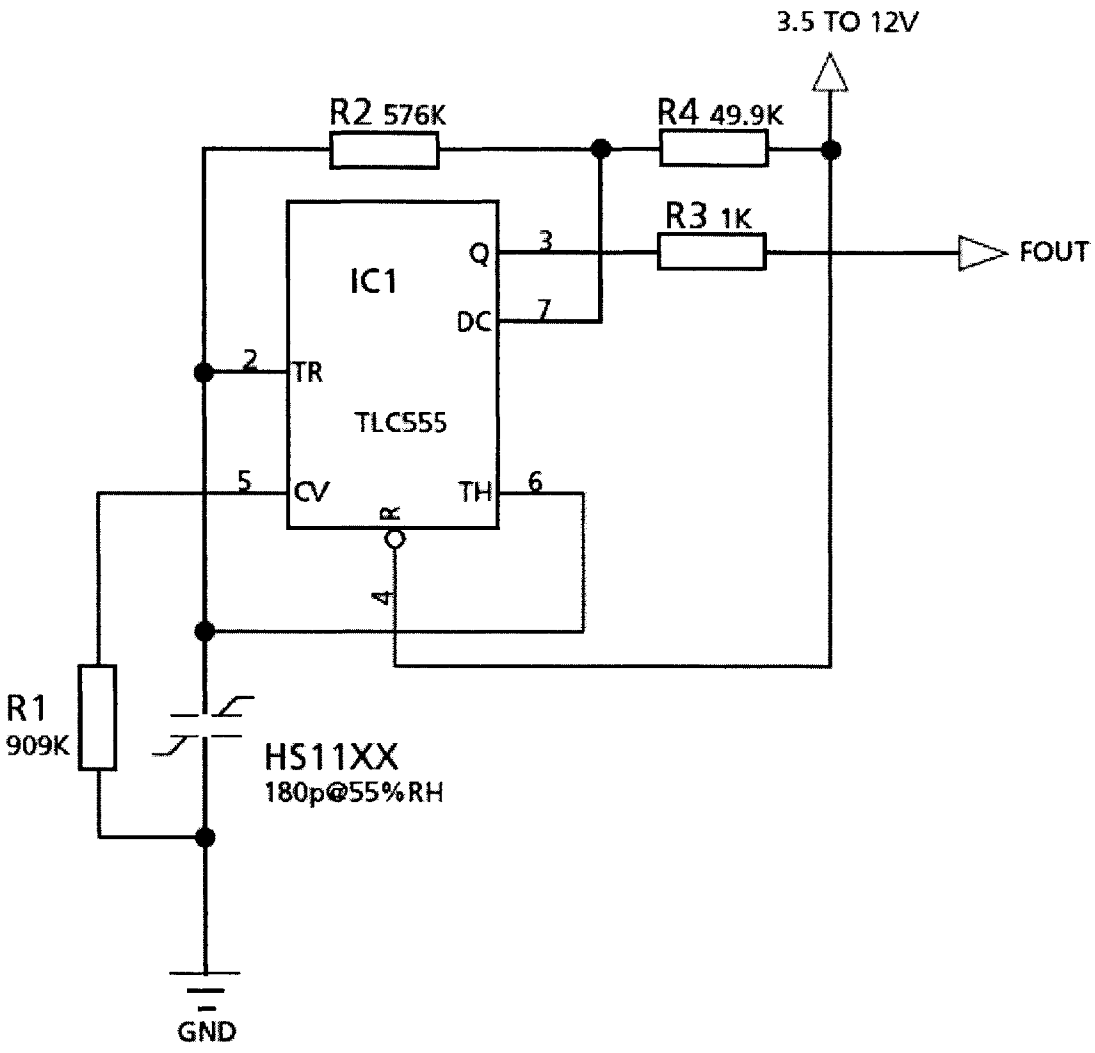

Figure 5-7: HS1101 Frequency Output Circuit [25]

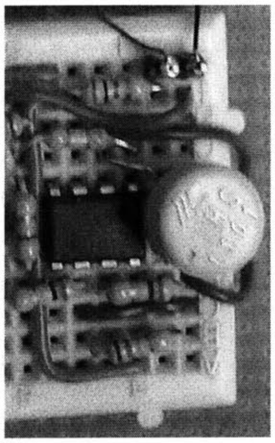

Figure 5-8: HS1101 humidity sensor with circuitry.

```
configuration HumiditySense {
}
implementation {
  components Main, HumiditySenseM, TimerC, IntToLedsArray, LedC, IntToRfm;

  Main.StdControl -> HumiditySenseM;
  Main.StdControl -> TimerC;
  Main.StdControl -> IntToLedsArray;
  Main.StdControl -> IntToRfm;

  HumiditySenseM.Timer -> TimerC.Timer[unique("Timer")];
  HumiditySenseM.Led -> LedC;
  HumiditySenseM.IntOutput -> IntToRfm;
  HumiditySenseM.IntOutput -> IntToLedsArray;
}
```

HumiditySense.nc wires the interfaces used by `HumiditySenseM` to the components which provide them. Notice that the *single* `IntOutput` interface used by `HumiditySenseM` is wired to *both* `IntToRfm` and `IntToLedsArray`. `HumiditySense` can be changed to use just one or the other—or some other component entirely (which provides the `IntOutput` interface)—just by changing the appropriate lines in the configuration. *HumiditySenseM.nc* can remain untouched.

```
module HumiditySenseM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Led;
    interface IntOutput;
  }
}

implementation {

  command result_t StdControl.init() {
    DDRC &= 0xFB;
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 500);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }
```

The `HumiditySenseM` module starts by using `StdControl.init()` to set pin PC2 (the pin to which the humidity sensor circuit is connected) to be an input. It then sets up a timer which repeats every 500 milliseconds. The actual sensing and processing occur in the `Timer.fired()` event handler:

```
event result_t Timer.fired() {
  uint16_t count;
  call Led.Toggle();
  loop_until_bit_is_clear(PINC, 2);
```

The capacitance (in pF) at the current relative humidity, $C@\%RH$, will be calculated by using the formula $t_{high} = C@\%RH \times (R2 + R4) \times \ln 2$, found in the HS1101 data sheet [25]. $R2$ and $R4$ are constant resistor values, so the only unknown is t_{high}, the length of time the square wave spends at its high value in each period. In order to accurately determine t_{high}, it must first find a *rising edge*, when the signal transitions from low to high. So, it makes sure that it is starting during a low phase by waiting until the input pin is low before continuing.

```
count = 0;
loop_until_bit_is_set(PINC, 2);
do { ++count; } while( bit_is_set(PINC, 2) );
```

While the square wave is low, `Timer.fired()` initializes `count` to be 0, and then it waits for the rising edge. It then increments `count` repeatedly, as many times as it can, while the square wave is high.

Now, in order to convert the final value of `count` into a time, it must be determined how long each iteration of the `while` loop takes. The JONA crystal oscillator is known to have a frequency of 14.7456MHz [30]; taking the reciprocal, each clock cycle lasts 0.067817μs. So, the problem is now reduced to determining the number of clock cycles per iteration. This, of course, depends on what AVR assembly instructions are executed each time through the loop. The ATmega128 manual [5] lists the number of clock cycles which each instruction takes to execute, so once this sequence is determined, calculating the loop's realtime length will simply be a matter of arithmetic.

Fortunately, there is a simple way to see what the assembly version of an application looks like. *Makerules* includes the following block of code:

```
ifndef OPTFLAGS
OPTFLAGS := -Os
endif
```

`OPTFLAGS` is a list of arguments to be passed to *avr-gcc*. The manual[13] for *avr-gcc* says that "`-Os`" optimizes the machine code for size, to make the program as small as possible (at least, as small as it can figure out how to make it). There is another flag mentioned in the manual which will prove useful for the current endeavor: `-S`. This argument will instruct *avr-gcc* to stop once it has produced the assembly code, rather than going on to turn the assembly code into an executable binary.

So, this implies that setting `OPTFLAGS` to `-S` in the *Makefile* and then running *make* will produce the assembly code as output, and then this file can be examined to see what instructions are going to be called for the loop. However, notice that *Makerules* will only set `OPTFLAGS` if it has not yet been defined. It is important that

[13]The *avr-gcc* manual can be accessed by executing the command `man avr-gcc` at a command prompt.

avr-gcc still see the `-Os` flag, so that the assembly output is properly optimized. So, the *Makefile* should be altered[14] to be:

```
COMPONENT=HumiditySense
PFLAGS=-I%T/lib/Counters
OPTFLAGS=-Os -S
include ../Makerules
```

Now, the command `make jona` should be executed as usual. At the last step, there will be an error message saying that *avr-objcopy* could not recognize the file format of **build/jona/***main.exe*. This is because *main.exe* is now a text file containing assembly, rather than the executable that *avr-objcopy* was expecting. According to this file, which can be examined with any text editor, the loop

```
do { ++count; } while( bit_is_set(PINC, 2) );
```

translates into the following assembly instructions (the label `.L1617` and the register `r28` may differ):

```
.L1617:
  adiw r28,1
  sbic 51-0x20,2
  rjmp .L1617
```

The register `r28` holds the `count` variable; `adiw r28,1` says to add 1 to the contents of the `r28`. `sbic 51-0x20,2` says that if bit 2 of address `51-0x20`—the address of PINC (cf. [5])—is cleared, then the program should skip over the next two bytes. Those two bytes contain the instruction `rjmp .L1617`, which says to jump back to the label `.L1617` and continue execution from there.

So, each iteration of the loop requires the execution of these three instructions. According to the ATmega128 manual, `adiw` takes 2 clock cycles, `sbic` either 1, 2, or 3, and `rjmp` 2. So altogether, the loop requires either 5, 6, or 7 cycles; but which?

[14]Of course, when it is time to compile and install the program, the `OPTFLAGS` line should be taken back out.

Well, according to the AVR documentation, when its test evaluates to false—as it will each time through the loop but the last—then `sbic` takes 1 cycle [6]. Each iteration, then, takes 5 cycles, or $5 \times 0.067817 = 0.339\mu$s. The value of `count` can now be translated into a capacitance:

```
/***************************************************************
 * Capacitance in pF = (count incs)*(0.339us/inc)/((621k)*ln2)
 *                   = 0.78756 * count
 *
 *   (0.78756 * count) will be a number between 100 and 300
 *   so mult. by 78.756 will be between 10,000 and 30,000
 *   which is still 16 bits (this may reduce roundoff error
 *   since there's no floating point unit)
 **************************************************************/
count *= 78.756; // now count represents cap in 10fF units
```

At this point, the `count` variable represents a capacitance, in units[15] of femtofarads. It must still be translated into a humidity measurement. For this task, the HS1101 data sheet again proves handy. Page 2 provides a graph of capacitance vs. relative humidity (see Figure 5-9).

This graph is nonlinear[16], but like that of the thermistor it can be approximated by a piecewise linear function. `HumiditySenseM` does just that, converting `count` into a number in units of hundredths of a percent of relative humidity:

```
/***************************************************************
 * Convert capacitance into %RH:
 * Uses a piecewise-linear approximation to the equation in the
 * HS1101 datasheet.
 *
 * count < 18500: %RH*100 = 3*count - 48900
 * count > 18500: %RH*100 = 2*count - 30400
 **************************************************************/
if( count < 18500 ) {
  count *= 3;
  count -= 48900;
} else {
  count *= 2;
  count -= 30400;
```

[15]See Table A.1 in Appendix A on page 150.

[16]The curve is actually a cubic function. The data sheet provides the equation $C(pf) = C@55\% \times (1.2510^{-7}RH^3 - 1.3610^{-5}RH^2 + 2.1910^{-3}RH + 9.010^{-1})$.

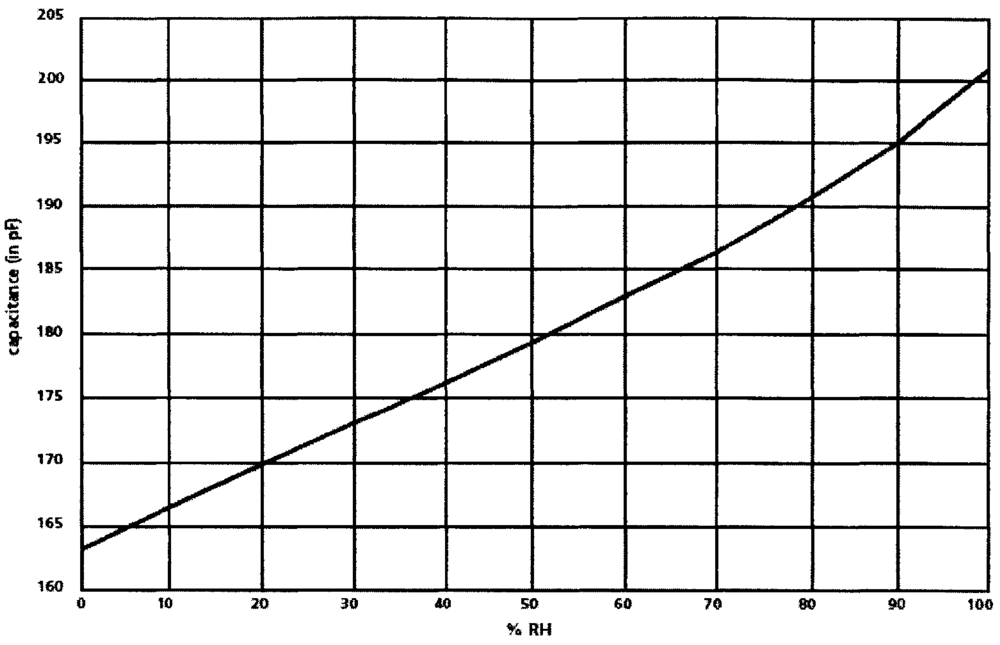

Figure 5-9: Typical response curve of HS 1100/HS 1101 in humidity [25]

```
    }
    return call IntOutput.output(count);
  }

  event result_t IntOutput.outputComplete(result_t success) {
    return SUCCESS;
  }
}
```

As seen in *HumiditySense.nc*, that call to `IntOutput.output()` will be heard by both `IntToLedsArray`—which will display the low ten bits of `count` on the LEDs—and `IntToRfm`—which will put `count` in a radio packet and transmit it. A nearby mote running a program like `RfmToLeds` (one of the sample applications which comes with TinyOS) can then listen to the incoming radio packet and display `count` on its own LEDs.

If a mote is running `TOSBase`, then the packet will be sent onward over its UART to a PC. `HumiditySense` packets cannot be used with *Oscilloscope* as is, because it expects to receive packets containing `OscopeMsg` messages. However, *SerialForwarder* could be used to redirect the humidity readings to some other application which understands the messages produced by `IntToRfm`. Alternatively, the PC could run the Java program *Listen* (and not *SerialForwarder*), which will print out the received packets byte by byte to the terminal. *Listen* is invoked with a command like `MOTECOM=serial@COM1:9600 java net.tinyos.tools.Listen`. Refer to TinyOS Tutorial Lesson 6 [49] for more information.

Finally, it should be noted that `HumiditySense` is, by far, the least portable program discussed in this document. Not only does it depend on JONA's hardware configuration, it also depends on JONA's clock speed. If the clock were replaced by one which was faster or slower, all of the calculations for capacitance and humidity based on the number of loop iterations would have to be redone. Additionally, as with any sensor, if accuracy is of any concern, it is important to test and calibrate the sensor, to see how it really behaves with a given known humidity. The "typical responses" indicated on data sheets sometimes have very little to do with reality. That said, `HumiditySense` succeeds in taking and transmitting humidity measurements.

5.6 Alternative Power Sources

The PROBOmega128 draws its power from a standard AC adapter's barrel connector, but this does not imply that it must be powered from a wall outlet. This section briefly mentions a few alternatives.

5.6.1 The 9V Battery Adapter

One simple option which has been done here in the lab is to equip a JONA board with a 9V battery adapter (see Figure 5-10). It will plug into the barrel connector on the side, and is indistinguishable to the rest of the system from a regular AC adapter.

The downside of this approach is that the battery will die (and if the LedsArray component is used, it will die very quickly). However, for quick testing of how intermote communication works when they are spread out down the hall, the battery adapter works quite nicely.

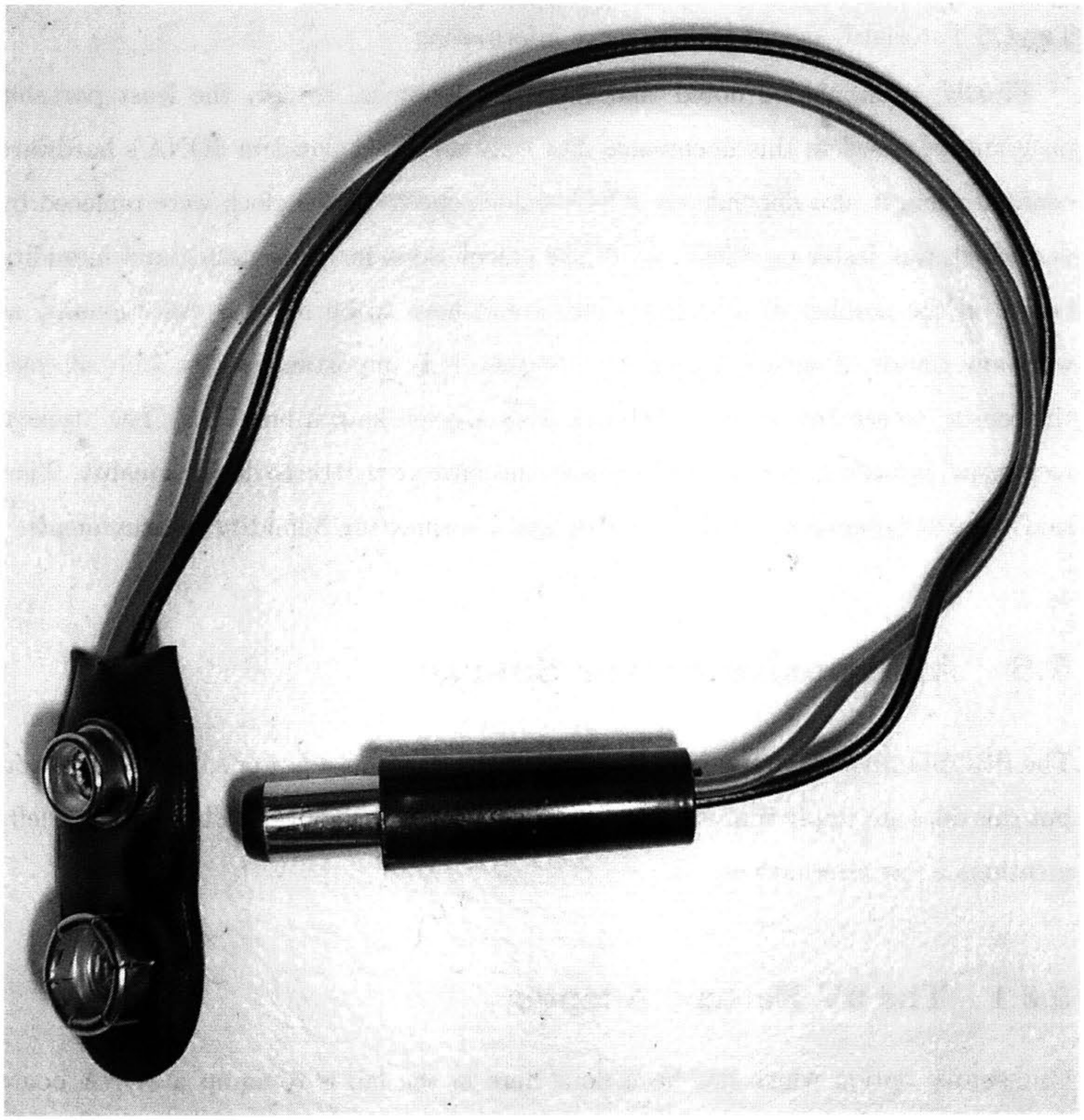

Figure 5-10: A 9V battery adapter for the JONA barrel connector.

5.6.2 Solar/RF Power Harvesting

There is other research being done, both at MIT and at MUST, to power JONA motes by other means. MUST researchers have connected motes to large solar panels and associated circuitry which collect solar energy during the day and provide power to the motes through the night.

Meanwhile, here at MIT, Drs. Nathaniel Osgood and Ruaidhri O'Connor have used smaller solar panels to collect power from indoor fluorescent lighting, storing the energy in capacitors in parallel. Once the voltage across the capacitors reach a certain level, a voltage trigger turns on the mote and powers it long enough to take a sensor reading and transmit it before the capacitors discharge. It then repeats the process. They are also looking into harnessing the energy of RF radiation, from cellular telephones, Wi-Fi, and local radio stations, for example.

Chapter 6

Conclusion

The JONA mote as presented in Chapter 4 is a functional WSN prototyping kit. As demonstrated in Chapter 5, JONA motes can be used for wireless communication, for taking sensor readings—with the ADC or by other means—and for relaying information between a WSN network and a PC.

They provide ample workspace and oversized connectors to facilitate quick hardware prototyping, and their built-in serial port allows for quick software prototyping. The JONA software platform is based on TinyOS, an open source operating system with an active development community and widespread adoption in the academic community. However, there is still room for improvement in a future JONA2.

6.1 Hardware

As the radio transceiver and other accessories are designed to (or at least can) run at 3V, the PROBOmega128 board should be modified to have a 3V V_{CC}. This would necessitate a reduction in the clock speed, but it would greatly simplify the wiring between the microcontroller (which would then be the low-power ATmega128L) and the radio module. It would also permit a 3V battery supply (e.g. a pair of AAs), rather than the current 9V solution, which has an intrinsic inefficiency associated with the conversion from 9V to 5V.

Furthermore, the radio module could be built into the main PCB, rather than being a secondary part soldered/wire-wrapped in place. Sensors, the LedsArray, and the like should continue to be separate add-on components, because each application will differ in which it may need, but all WSN systems require wireless communications.

It may also be advantageous to replace the RFM TR1000 with the Chipcon CC1000 [12]. The TR1000 has performed well, but anecdotal evidence suggests that the CC1000 has better noise immunity and other laudable characteristics (e.g. [20]). The DR3000-1 modules have been observed to be somewhat sensitive to RF interference from cellular telephones. (It should be noted that the TR1000 was initially chosen over the CC1000 because it was the simpler of the two to adapt for the 5V PROBOmega128 board.)

Going with the Chipcon radio would also permit a more "pure" usage of the ATmega128's I/O facilities than the somewhat abusive use of the SPI interface which employing the RFM chip entailed[1]; the CC1000 is a full-fledged I2C device. This would have the added benefit of freeing the SPI interface for other devices (see below).

6.2 Software

The JONA software platform as described in Section 4.2 is complete for the JONA hardware at present; most TinyOS applications should compile with no changes (unless they are specifically designed for other hardware). Furthermore, it is hoped that the software examples presented in Chapters 4 and 5 will be more demonstrative to CS neophytes than perhaps some of the TinyOS Tutorial examples tend to be. This notwithstanding, there remain two TinyOS technologies (in particular) of which it would be nice to see good, clearly explained examples: ad hoc multi-hop routing and in-network programming.

TinyOS provides an automated ad hoc multi-hop routing mechanism, found in the **Broadcast/**, **Queue/** and **Route/** directories of **tos/lib/** (see **doc/***ad-hoc.pdf*). This lets individual applications focus on data collection and processing, without having to worry about how to route messages back to the base station. TinyOS

[1]cf. Section 4.1.2 on page 74.

includes a sample application called `Surge` which makes use of this to collect photosensor measurements and send them back to the PC. An associated Java program will then produce a network graph showing the paths from various nodes back to the base station (see **doc/multihop/***multihop_routing.html*).

`Surge`—with slight modification to remove dependency on the micasb sensorboard—compiles and runs on JONA motes, but it fails to point out certain intricacies involved with adopting the multi-hop functionality. For instance, multi-hop routing adds an overhead to each radio packet of 7 bytes. This means that some messages, such as `OscopeMsg`, which worked with the regular radio stack will now be too long and require modification. Ideally, a sample application would take the user through all the steps of converting an application which uses the radio directly into one which uses the multi-hop stack, including making necessary changes to the message data structure. It would also be good to see the Java *Oscilloscope* program adapted to display the data from the new application.

In-network programming (INP; see **doc/***Xnp.pdf*) is an add-on which lets motes receive and install new programming while deployed (i.e. without being physically connected to a PC). The program is transmitted via radio, and motes listen for and copy it into memory. The JONA hardware does not support INP, because an external Flash memory is required to store the program as it is being downloaded. An appropriate addition to JONA would be the Atmel AT45DB041B Flash memory chip [4], which is similar to the chip used in MICA2 (which does support INP). The AT45DB041B is an SPI device, so moving the radio to the I2C interface would simplify matters considerably (see above). With INP supported in hardware, a good sample application would show how to convert a non-INP program and would offer any caveats against areas that might cause trouble when creating new INP applications.

6.3 Final Thoughts

The objective of this endeavor was to develop a hands-on educational tool for WSN. Even without the enhancements suggested in the previous two sections, the JONA

Prototyping Kit meets this goal. It provides a good, undaunting starting point for novice users, while offering a level of flexibility which permits and encourages advanced experimentation.

The development of JONA has proven to be a valuable experience, presenting many useful insights into issues of accessibility and usability of this type of kit, and it provides a solid basis for further work into the design and implementation of JONA Version 2.

Appendix A

Common Prefixes

In most scientific disciplines, prefixes and units used are those of the International System of Units (SI[1]). Each prefix corresponds to a multiple of the base unit, in powers of 10, where the base unit is the SI standard for measuring some natural physical quantity, such as the meter for length or the ampere for electrical current[2]. However, computer science also uses many of the same prefix symbols[3] and pronunciations as binary prefixes (powers of 2) to denote quantities of information, usually with a base of bits (b) or bytes (B). Table A.1 offers some common prefixes[4] along with their SI and binary meanings. Note that there are no fractional binary prefixes, and that in each case, the binary value is slightly larger than the SI value.

[1]The abbreviation "SI" is taken from the system's French name, *Système International d'Unités.*

[2]The official SI base unit of mass is the kilogram, not the gram; but when other SI prefixes are attached to units of mass, they are applied to the gram (e.g. milligram (mg) rather than microkilogram (μkg)).

[3]In the case of kilo, the SI symbol for 10^3 is always lowercase "k", but sometimes the uppercase "K" is used to denote 2^{10}.

[4]The table's set of prefixes is far from exhaustive; there are others to express values both larger and smaller than the extremes shown here, as well as a few rarely-used intermediate values, like "deka" for $10^1 = 10$.

Table A.1: Common SI and Information Theory Prefixes

Prefix	Symbol	SI Value	Binary Value
tera	T	$10^{12} = 1\ 000\ 000\ 000\ 000$	$2^{40} = 1\ 099\ 511\ 627\ 776$
giga	G	$10^{9} = 1\ 000\ 000\ 000$	$2^{30} = 1\ 073\ 741\ 824$
mega	M	$10^{6} = 1\ 000\ 000$	$2^{20} = 1\ 048\ 576$
kilo	k	$10^{3} = 1\ 000$	$2^{10} = 1\ 024$
centi	c	$10^{-2} = 0.01$	
milli	m	$10^{-3} = 0.001$	
micro	μ	$10^{-6} = 0.000\ 001$	
nano	n	$10^{-9} = 0.000\ 000\ 001$	
pico	p	$10^{-12} = 0.000\ 000\ 000\ 001$	
femto	f	$10^{-15} = 0.000\ 000\ 000\ 000\ 001$	

Appendix B

Bits, Binary and the Digital Abstraction

Humans usually deal with numbers using base ten (decimal) arithmetic. Numbers have a one's place, a ten's place, a hundred's place, and so on. In other words, each digit represents a place value of the form 10^i, for $i = \{0, 1, \ldots\}$. For example, the number 167 means $1 \times 10^2 + 6 \times 10^1 + 7 \times 10^0$. Each digit can take on one of ten values, namely 0–9.

Binary—base two—numbers have place values of the form 2^i. There is a one's place, a two's place, a four's place, etc. Each binary digit, or *bit*, can take on one of two values: 0 or 1. The binary number 10100111 means $1\times2^7+1\times2^5+1\times2^2+1\times2^1+1\times2^0$, or 167 decimal.

Eight bits make up a unit called a *byte* (B). Binary numbers in computers are usually stored in one or more bytes. A single byte can store one of 256 values[1], 0–255. For brevity, numbers which are stored internally as binary are often displayed as *hexadecimal*, base sixteen. Every four bits can be represented by a single hexadecimal digit, which may take on values between 0 and 15. For 10–15, the letters A–F are used. So, the decimal number 167, which is 10100111 in binary, is A7 in hexadecimal: $10 \times 16^1 + 7 \times 16^0$ (see Table B.1). In order to avoid ambiguity regarding a number's

[1]This range assumes that the numbers are unsigned. If a number is allowed to take on both positive and negative values, then a byte might store numbers between -128 and $+127$ (inclusive).

intended base, binary numbers are often prefixed by "0b" and hexadecimal numbers are often prefixed by "0x". If a number carries no prefix, it is generally assumed to be decimal. For example, "10" means *ten*, "0b10" means *two*, and "0x10" means *sixteen.*

Table B.1: Binary and Hexadecimal Representations for 0–15.

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0b0000	0x0	8	0b1000	0x8
1	0b0001	0x1	9	0b1001	0x9
2	0b0010	0x2	10	0b1010	0xA
3	0b0011	0x3	11	0b1011	0xB
4	0b0100	0x4	12	0b1100	0xC
5	0b0101	0x5	13	0b1101	0xD
6	0b0110	0x6	14	0b1110	0xE
7	0b0111	0x7	15	0b1111	0xF

Binary is a convenient representation to use for numbers in electronics because it only requires two states to be defined: one to represent a logical "0" and one to represent a logical "1". A natural choice is to use voltage levels: low voltage for one, and high voltage for the other. This often takes the form of ground for 0, V_{CC} for 1. Generally, there are (disjoint) ranges of voltages which represent each, so that if an input is slightly above ground (or slightly below V_{CC}) it still maps correctly to 0 (or 1). These two ranges are referred to as *valid logic levels.* In between them there is a "forbidden zone" of voltages which are not guaranteed to read as 0 or as 1; some circuitry may respond to them as if they were low (0), while other circuitry may respond as if they were high (1).

Of course, analog voltages could also represent different numbers: 0V for 0, 0.5V for 0.5, 3V for 3, and so on. The great advantage of using the *digital abstraction* is noise immunity. If there is noise on the wire and 0V becomes 0.4V, the data is still read as "0"; it is not corrupted (whereas in the analog method, any change in the voltage signifies a change in the data).

Appendix C

The ASCII Encoding Scheme

Tables C.1–C.4 below lists the characters and actions defined by the American Standard Code for Information Interchange (ASCII). A byte can hold any of the values between 0 and 255, but ASCII only defines 0–127; values outside of this range have been used in different ways by different computers to represent accented characters and other symbols. The first 32 ASCII codes do not represent printable characters but rather actions to be taken by a device (such as a printer or teletype); these include the codes indicated by escape sequences like `\r` (0xD) and `\n` (0xA) in C.

Table C.1: ASCII Codes 0x00–0x1F

Dec	Hex	Chr	Dec	Hex	Chr
0	0x0	NUL (null)	16	0x10	DLE (data link escape)
1	0x1	SOH (start of heading)	17	0x11	DC1 (device control 1)
2	0x2	STX (start of text)	18	0x12	DC2 (device control 2)
3	0x3	ETX (end of text)	19	0x13	DC3 (device control 3)
4	0x4	EOT (end of trans.)	20	0x14	DC4 (device control 4)
5	0x5	ENQ (enquiry)	21	0x15	NAK (neg. acknowledge)
6	0x6	ACK (acknowledge)	22	0x16	SYN (synchronous idle)
7	0x7	BEL (bell)	23	0x17	ETB (end trans. block)
8	0x8	BS (backspace)	24	0x18	CAN (cancel)
9	0x9	TAB (horizontal tab)	25	0x19	EM (end of medium)
10	0xA	LF (line feed, newline)	26	0x1A	SUB (substitute)
11	0xB	VT (vertical tab)	27	0x1B	ESC (escape)
12	0xC	FF (form feed)	28	0x1C	FS (file separator)
13	0xD	CR (carriage return)	29	0x1D	GS (group separator)
14	0xE	SO (shift out)	30	0x1E	RS (record separator)
15	0xF	SI (shift in)	31	0x1F	US (unit separator)

Table C.2: ASCII Codes 0x20–0x3F

Dec	Hex	Chr	Dec	Hex	Chr
32	0x20	Space	48	0x30	0
33	0x21	!	49	0x31	1
34	0x22	”	50	0x32	2
35	0x23	#	51	0x33	3
36	0x24	$	52	0x34	4
37	0x25	%	53	0x35	5
38	0x26	&	54	0x36	6
39	0x27	’	55	0x37	7
40	0x28	(	56	0x38	8
41	0x29	)	57	0x39	9
42	0x2A	*	58	0x3A	:
43	0x2B	+	59	0x3B	;
44	0x2C	,	60	0x3C	<
45	0x2D	-	61	0x3D	=
46	0x2E	.	62	0x3E	>
47	0x2F	/	63	0x3F	?

Table C.3: ASCII Codes 0x40–0x5F

Dec	Hex	Chr	Dec	Hex	Chr
64	0x40	@	80	0x50	P
65	0x41	A	81	0x51	Q
66	0x42	B	82	0x52	R
67	0x43	C	83	0x53	S
68	0x44	C	84	0x54	T
69	0x45	E	85	0x55	U
70	0x46	F	86	0x56	V
71	0x47	G	87	0x57	W
72	0x48	H	88	0x58	X
73	0x49	I	89	0x59	Y
74	0x4A	J	90	0x5A	Z
75	0x4B	K	91	0x5B	[
76	0x4C	L	92	0x5C	\
77	0x4D	M	93	0x5D	]
78	0x4E	N	94	0x5E	ˆ
79	0x4F	O	95	0x5F	_

Table C.4: ASCII Codes 0x60–0x7F

Dec	Hex	Chr	Dec	Hex	Chr
96	0x60	‘	112	0x70	p
97	0x61	a	113	0x71	q
98	0x62	b	114	0x72	r
99	0x63	c	115	0x73	s
100	0x64	d	116	0x74	t
101	0x65	e	117	0x75	u
102	0x66	f	118	0x76	v
103	0x67	g	119	0x77	w
104	0x68	h	120	0x78	x
105	0x69	i	121	0x79	y
106	0x6A	j	122	0x7A	z
107	0x6B	k	123	0x7B	{
108	0x6C	l	124	0x7C	|
109	0x6D	m	125	0x7D	}
110	0x6E	n	126	0x7E	˜
111	0x6F	o	127	0x7F	DEL

Appendix D

Semiconductors

Materials which conduct electricity well—like copper—are called *conductors*; those which do not—like glass—are called *insulators*. Conductors and insulators lie at opposite ends of a spectrum of *conductivity*. The middle of the spectrum is the domain of *semiconductors*—like silicon.

A solitary, electrically-neutral atom of silicon has four electrons in its outermost layer[1]. In a crystalline lattice, it shares each electron with a neighboring atom[2], so that each atom in the lattice has a total of eight electrons, filling the layer (see Figure D-1).

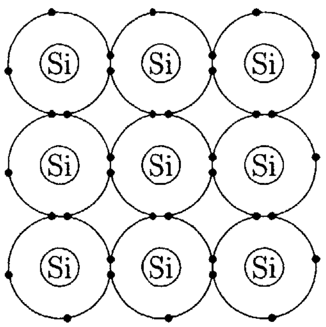

Figure D-1: Silicon Lattice

Electricity does not flow well in pure silicon, because all the electrons are busy keeping the silicon atoms bonded to each other. This can be remedied by introducing

[1]These electrons are called *valence electrons.*

[2]Two atoms each contribute one electron to a *covalent bond* which binds them together.

impurities into the silicon crystal through a process called *doping*. In doped silicon, some of the silicon atoms with their four valence electrons are replaced by atoms of other elements which have a different number—generally three or five—of valence electrons. In either case, extra charge carriers are added, in the form of holes or electrons.

If silicon is doped with phosphorus, which has five valence electrons, then it is called *n-type*[3] doped silicon. Phosphorus has one more electron than silicon but it also has one more proton, so the n-type region is still electrically neutral. The phosphorus atom, like the silicon atom it replaced, uses four valence electrons to bond with the four atoms around it; unlike the silicon atom, it still has a free electron after forming these four bonds (see Figure D-2). This free electron is called a *carrier electron*, and it really is free: it can move throughout the lattice, carrying its charge. These carrier electrons facilitate the flow of electricity through the silicon.

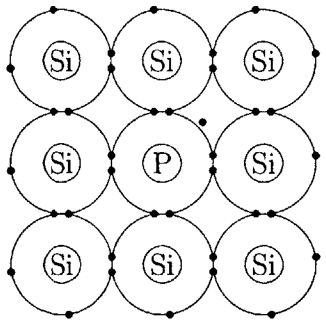

Figure D-2: n-type Doped Silicon Lattice with Negative Charge Carrier (Electron)

If silicon is doped with boron, which has only three valence electrons, then it is called *p-type*[4] doped silicon. The p-type region is electrically neutral because boron, with fewer electrons, has correspondingly fewer protons. However, when the boron atom tries to bond with the four silicon atoms around it, it comes up one electron short; rather than having a full valence shell with eight electrons, it only has seven (its three plus one from each of its four neighbors; see Figure D-3). In other words,

[3]It is called "n-type" because the added charge carriers are electrons, which are *negatively* charged.

[4]It is called "p-type" because the added charge carriers are holes, which are *positively* charged.

replacing silicon atoms with boron atoms introduces holes, which also facilitate the flow of electricity[5].

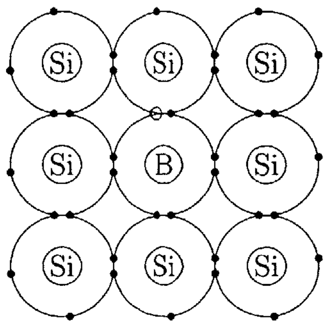

Figure D-3: p-type Doped Silicon Lattice with Positive Charge Carrier (Hole)

[5]See Section 2.1.1 on page 30.

Appendix E

PROBOmega128 Schematic

Figure E-1 below shows the schematic for the PROBOmega128, © Dr. Erik Lins [31].

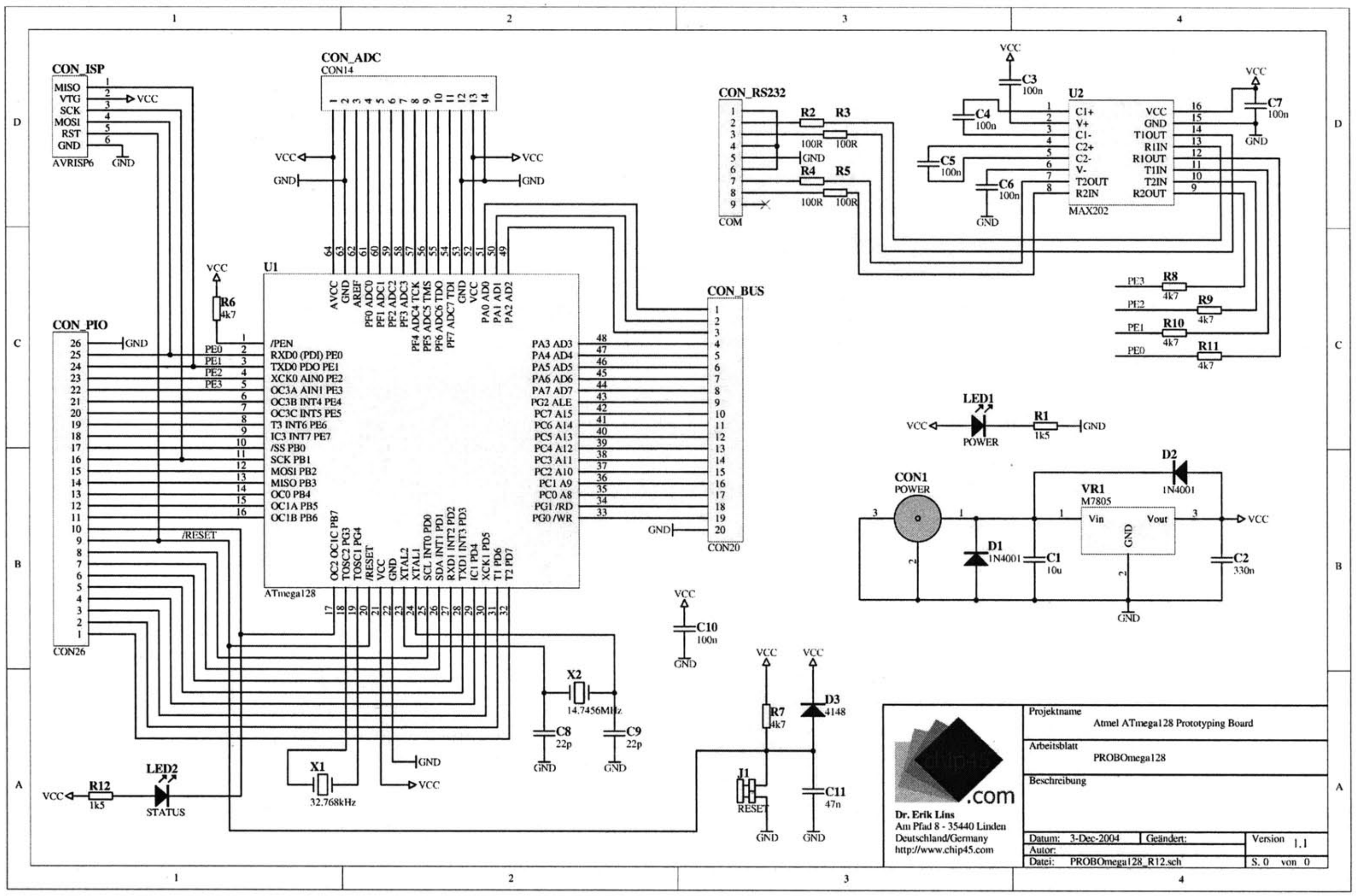

Figure E-1: PROBOmega128 schematic [31].

Appendix F

JONA Radio Module Schematic

Figure F-1 below illustrates the wiring of the JONA radio module. The power supply connected to Pin 9 (VCC) is provided by a 3V voltage regulator (not shown) which converts the PROBOmega128's 5V supply into the 3 volts needed by the DR3000-1. "(NC)" indicates that no connection is made to that pin; the other connections are as listed in Table F.1.

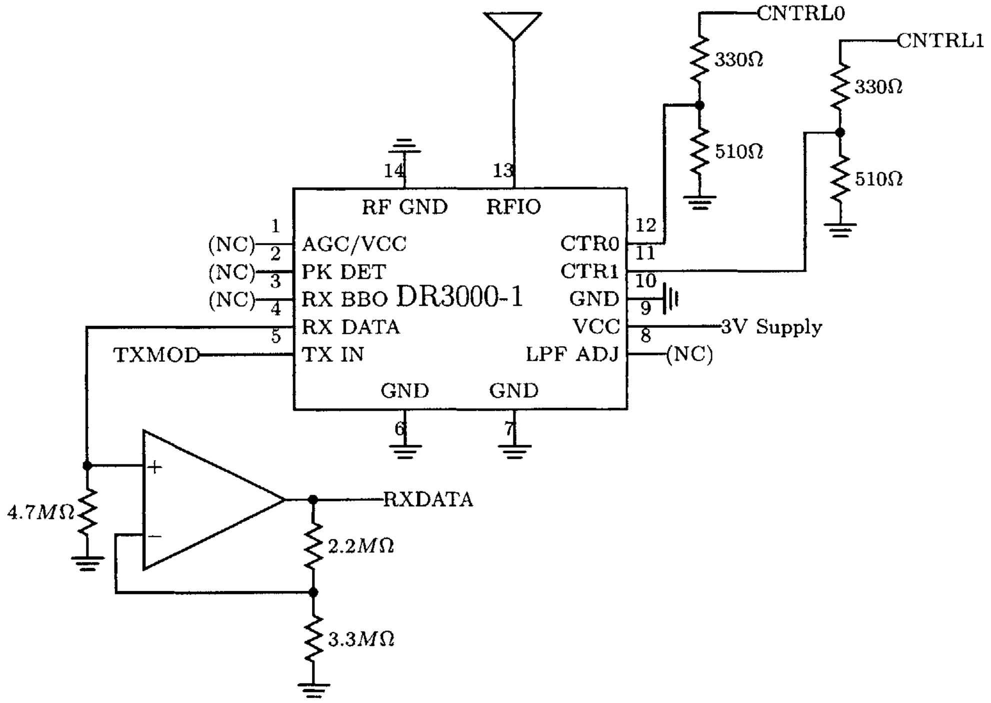

Figure F-1: DR3000-1 wiring schematic.

Table F.1: DR3000-1↔PROBOmega128 Connections in the JONA.

Figure F-1 Label	`CON_PIO` Pin 1	`CON_PIO` Pin 2
	PB0 ($\overline{SS}$)	GROUND
	PB1 (SCK)	PB7 (OC2)
RXMOD	PB2 (MOSI)	PD4 (IC1)
TXMOD	PB3 (MISO)	
CNTRL1	PD6 (T1)	
CNTRL0	PD7 (T2)	

Appendix G

HelloWorld: A nesC/TinyOS Application Example

Below are the contents of *HelloWorld.nc* and *HelloWorldM.nc*, the configuration and module files for a Hello World application which demonstrates usage of the `ByteComm` interface for communication over the UART and exemplifies TinyOS's event-driven model. Also included is the application's *Makefile*, used to compile and install the program.

G.1 HelloWorld.nc

```
configuration HelloWorld {
}
implementation {
  components Main, UART, HelloWorldM;
  Main.StdControl -> HelloWorldM;
  Main.StdControl -> UART;
  HelloWorldM.ByteComm -> UART;
}
```

G.2 HelloWorldM.nc

```
module HelloWorldM {
  provides {
    interface StdControl;
  }
  uses {
    interface ByteComm;
  }
}
implementation {
  uint8_t index, *string = (uint8_t*)"Hello, World!\r\n";

  command result_t StdControl.init() {
    atomic {
      index = 0;
    }
    return SUCCESS;
  }

  command result_t StdControl.start() {
    uint8_t i;
    atomic {
      i = index;
      ++index;
    }
    return call ByteComm.txByte(string[i]);
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }

  async event result_t ByteComm.rxByteReady(uint8_t dat, bool err, uint16_t str) {
    return SUCCESS;
  }

  async event result_t ByteComm.txDone() {
    return SUCCESS;
  }

  async event result_t ByteComm.txByteReady(bool success) {
    uint8_t c, i;
    atomic {
      i = index;
    }
    c = string[i];
    if(c) {
      atomic {
        ++index;
      }
      return call ByteComm.txByte(c);
    } else {
      return SUCCESS;
    }
  }
}
```

G.3 Makefile

```
COMPONENT=HelloWorld

include ../Makerules
```

Appendix H

Additions Made to the Makerules File to Support JONA

This appendix discusses the additions made to the *Makerules* file (in the **apps/** directory) which were necessary to extend the base TinyOS distribution to support the new JONA platform. Section H.1 enumerates these additions, and Section H.2 provides a full listing of the resultant file.

H.1 The Additions

1. When compiling or installing a TinyOS application, the name of the target platform is passed as an argument to *make*, as in `make install mica2`. Each of the valid platforms is stored in a list in the *Makerules* file. In the original *Makerules*, this list is created by the line:

   ```
   PLATFORMS = mica mica2 mica2dot pc
   ```

   The JONA platform has now been included in the list:

   ```
   PLATFORMS = mica mica2 mica2dot jona pc
   ```

2. The TinyOS development environment provides several options as to how to install a program onto a mote. These are listed in the following comment[1] in *Makerules*:

```
##########################################################################
# Programming Boards : command line input
# Command line input for programmers:
#   none          : default to parallel programming board
#   MIB510=<dev> : use mib510 serial port programming board at port <dev>
#   EPRB=<host>  : use eprb at hostname <host>
#   AVRISP=<dev> : use AVRISP serial programmer at port <dev>
##########################################################################
```

The file proceeds to check the command line input to see which option (if any) was chosen. For instance, to install a program onto a MICA2 mote using an AVRISP programmer[2], the command issued might look like `make install mica2 AVRISP=/dev/ttyS0`. *Makerules* then sets the PROGRAMMER and PROGRAMMER_FLAGS variables appropriately for the choice indicated. For instance, it uses the following lines[3] if the AVRISP option was selected:

```
ifdef AVRISP
  PROGRAMMER := STK
  PROGRAMMER_FLAGS=-dprog=stk500 -dserial=$(AVRISP) $(PROGRAMMER_PART) \
                   $(PROGRAMMER_EXTRA_FLAGS_AVRISP)
endif
```

If no programming board was specified, *Makerules* assumes[4] as a default a Crossbow programmer which is connected to the computer's parallel port:

[1]In *make* syntax, a comment starts with the character "#" and continues to the end of the line. This is analogous to C's "//".

[2]That is, an in system programmer (ISP) for AVR microcontrollers.

[3]The backslash at the end of the PROGRAMMER_FLAGS line indicates that what follows is actually a continuation of the same line; the backslash itself and the subsequent newline should be ignored. In the file, this facilitates the creation of arbitrarily long statements without the file's individual lines becoming unwieldily lengthy; here, it enables a representation of the file's contents which does not violate the document's margins.

[4]The following two lines are found immediately before the comment listing the other options; *Makerules* initially assumes the default and then changes the variables if and when it sees that an alternative was selected.

```
PROGRAMMER_FLAGS=-dprog=dapa $(PROGRAMMER_PART) $(PROGRAMMER_EXTRA_FLAGS)
PROGRAMMER := DAPA
```

Now, each of these options represents a physical programmer board, connected to the computer, which motes are plugged into for programming. However, the JONA hardware has its own serial port and does not need a separate board; it connects directly to the computer. So, it would not make sense to specify a new programmer for JONA as a command line option. At the same time, the default choice, which uses the parallel port, is also inappropriate.

To overcome this minor paradox, the JONA itself is viewed as a programmer board and made a new option. It is not necessary to create a new command line option, because `jona` already appears as an argument to *make* to specify the platform. So, the following lines have been added to this section of *Makerules*, which treat JONA as a programmer and yet check for it not as a specified *programmer* but as a specified *platform.*

```
### If the serial port has not been defined, then set it to the
### default.
ifeq (x$(SERIAL_PORT),x)
  SERIAL_PORT=/dev/ttyS0 -dspeed=115200
endif
### If the platform is jona, then use STK on the serial port.
### (The jona does not use a separate programmer device, so there
### will not be a programmer listed on the command line.)
ifeq ($(PLATFORM), jona)
  PROGRAMMER := STK
  PROGRAMMER_FLAGS=-dprog=stk500 -dserial=$(SERIAL_PORT) $(PROGRAMMER_PART)
endif
```

The JONA mote is presumed to be connected to the serial port accessible as device `ttyS0`; if it is actually connected to a different serial port, then it can be specified on the command line (as in `make install jona SERIAL_PORT=-/dev/ttyS1 -dspeed=115200`). The programmer indicated, *STK*, is the firmware on the JONA mote[5] when enables it to write the new program into its

[5] *STK500* is also responsible for blinking the mote's LED when it is put into programming mode.

memory. The PROGRAMMER_PART variable is defined later on in *Makerules*, and will be discussed in turn.

3. The Crossbow motes use sensors which are packaged onto cards called *sensorboards*. Recall from Table 4.2 that the TinyOS distribution includes a **sensorboards/** directory to contain files related to each card. At the very least, each card's subdirectory contains a file called *sensorboard.h* which declares what sensors are available on that card. The **sensorboards/** directory contains three subdirectories: **basicsb/**, **micasb/**, and **none/**. The first two correspond to actual sensorboards, while **none/** is a placeholder which is intended to be used when no sensorboard is actually present.

   A particular TinyOS application can specify with which sensorboard it is to be used by including a line like "SENSORBOARD = micasb" in its *Makefile*. If it does not, *Makerules* sets default values for each platform by executing the following block of code:

```
#Sensor Board Defaults
ifeq ($(SENSORBOARD),)
  ifeq ($(PLATFORM),mica)
    SENSORBOARD = micasb
  endif
  ifeq ($(PLATFORM),mica2)
    SENSORBOARD = micasb
  endif
  ifeq ($(PLATFORM),mica128)
    SENSORBOARD = micasb
  endif
  ifeq ($(PLATFORM),rene2)
    SENSORBOARD = basicsb
  endif
  ifeq ($(PLATFORM),pc)
    SENSORBOARD = micasb
  endif
  ifeq ($(PLATFORM),mica2dot)
    SENSORBOARD = basicsb
  endif
endif
```

   The outermost ifeq tests whether the variable SENSORBOARD is equal to nothing (i.e. has not yet been assigned). If that is the case, then it compares the value of the PLATFORM variable—set by the command line argument to *make*—to the following platforms[6]: mica; mica2; mica128; rene2; pc; mica2dot. It checks these one at a time, and upon finding a match it sets SENSORBOARD to the appropriate value.

The JONA platform does not use bundles of sensors on sensorboard cards; rather, sensors are wired directly to the PROBOmega128 as needed. Nevertheless, TinyOS dictates that any platform using an ADC must define a sensorboard. The ADC interface file expects to be able to read from a *sensorboard.h* file to learn the values of certain constants associated with each sensor (such as to which ADC channel each sensor is wired). However, the JONA sensors are independent, and they each have their own files to define these constants.

So, the JONA platform uses the dummy **none/** sensorboard directory, because its *sensorboard.h* is an empty placeholder. It can safely be included without the worry that it might override a sensor's definitions. To achieve this, another inner ifeq was added:

```
ifeq ($(SENSORBOARD),)
  ...
  ifeq ($(PLATFORM),jona)
    SENSORBOARD = none
  endif
endif
```

4. Having specified the programmer hardware and the sensor hardware, *Makerules* must now specify the hardware of the mote itself, in particular its microcontroller. This is the meaning of the PROGRAMMER_PART variable mentioned above. As it did for defining the SENSORBOARD variable, *Makerules* uses a series of ifeq statements to determine which platform has been selected:

[6]Note that *rene2* and *mica128* are not valid platforms, according to the PLATFORMS list. These are older Berkeley/Crossbow mote platforms which were replaced, respectively, by the mica and mica2 families. This is one of a number of examples of outdated legacy code which can be found in the TinyOS source. The mica128 also remains as a platform directory. Its contents are used by several other platforms, including JONA (see Section 4.2.4 on page 92).

```
ifeq ($(PLATFORM), mica)
PROGRAMMER_PART=-dpart=ATmega103 --wr_fuse_e=fd
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) -dpart=ATmega103
endif
endif
ifeq ($(PLATFORM), mica128)
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) -dpart=ATmega128
endif
endif
ifeq ($(PLATFORM), mica2)
BOOTLOADER=$(XNP_DIR)/inpispm2.srec
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) -dpart=ATmega128
endif
endif
ifeq ($(PLATFORM), mica2dot)
BOOTLOADER =$(XNP_DIR)/inpispm2d.srec
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) -dpart=ATmega128
endif
endif
```

To support JONA, the following snippet has been added to this set:

```
ifeq ($(PLATFORM), jona)
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
endif
```

The `PROGRAMMER_PART` variable declares the microcontroller to be an ATmega128. It then instructs the programmer to write the value `0xFF` into the ATmega128's Extended Fuse. This fuse—which is a register that can only be written to by the programmer (in particular, the microcontroller cannot change its value during runtime)—ensures that the ATmega128 is *not* in ATmega103 compatibility mode[7]. This mode is designed to provide backwards compatibility for

[7] Cf. Table 117 on page 289 of [5].

systems designed around the older microcontroller[8] (for which the ATmega128 was the direct replacement), at the cost of disabling the ATmega128's new features. As JONA was not originally designed around the ATmega103, there is no reason to use this compatibility mode, which would only serve to limit the microcontroller's capabilities.

5. The last change made to *Makerules* does not, in fact, have anything to do with the JONA platform. In the rule for compiling the executable, `$(MAIN_EXE)`, "-v" has been added to the compiler's list of command line arguments. This tells *ncc* to be *verbose*, to write out to the terminal a description of each step of the compilation as it is performed. This output is often helpful to track down program bugs, and it is also conducive to the understanding of what exactly the nesC compiler does to turn a TinyOS application's source code into an executable program.

```
$(MAIN_EXE): $(BUILDDIR) FORCE
     @echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
     $(NCC) -o $(MAIN_EXE) -v $(OPTFLAGS) $(PFLAGS) $(CFLAGS) \
              $(COMPONENT).nc $(LIBS) $(LDFLAGS)
```

H.2 Makerules

```
#-*-Makefile-*-
# Base Makefile for nesC apps.
#
# Created: 6/2002,  Philip Levis <pal@cs.berkeley.edu>
#
# Updated: 6/18/2002 Rob von Behren <jrvb@cs.berkeley.edu>
#          Multi-platform support
#
# Updated: 6/20/2002 David Gay <dgay@intel-research.net>
#          Compile via gcc, make tos.th system-wide, not app-wide
#          (still need to ponder group selection)
#
# Updated: 6/27/2003 Jaein Jeong <jaein@cs.berkeley.edu>
```

[8]Such as MICA; notice that the PROGRAMMER_PART definition for MICA says to write 0xFD into the fuse, which makes newer MICA motes, which actually have an ATmega128, still act as though they contained an ATmega103.

```
#          In-network programming support for mica2, mica2dot platforms
#
###############################################################################

# this needs to be -dlpt=3 on thinkpads
# PROGRAMMER_EXTRA_FLAGS :=
# We don't actually set it here, so you can either set the
# PROGRAMMER_EXTRA_FLAGS environment variable (recommended) or
# define it in ../Makelocal

-include $(shell ncc -print-tosdir)/../apps/Makelocal

# User configuration:
# Specify user values in Makelocal to override the defaults here

ifndef DEFAULT_LOCAL_GROUP
DEFAULT_LOCAL_GROUP := 0x7d
endif

ifndef OPTFLAGS
OPTFLAGS := -Os
endif

ifndef NESC_FLAGS
NESC_FLAGS := -Wnesc-all
endif

# configure the base for the app dirs.  This is used to generate more
# useful package names in the documentation.
ifeq ($(BASEDIR)_x, _x)
BASEDIR := $(shell pwd | sed 's@\(.*\)/apps.*$$@\1@' )
endif

# The output directory for generated documentation
ifeq ($(DOCDIR)_x, _x)
DOCDIR := $(BASEDIR)/doc/nesdoc
endif

#####################################################
#
#####################################################

ifeq ($(PLATFORMS)_x, _x)
PLATFORMS = mica mica2 mica2dot jona pc
endif

OBJCOPY        = avr-objcopy
SET_ID         = set-mote-id
PROGRAMER        = uisp

ifdef MSG_SIZE
PFLAGS := -DTOSH_DATA_LENGTH=$(MSG_SIZE) $(PFLAGS)
endif
```

```
#ifdef APP_DIR
#PFLAGS := -I$(APP_DIR) $(PFLAGS)
#endif

# This is for network reprogramming
# If XNP is defined, add the network reprogramming related files
# to the search path and generate a timestamp to make each build unique.
XNP_DIR := ../../tos/lib/Xnp
ifdef XNP
PFLAGS := -I$(XNP_DIR) $(shell $(XNP_DIR)/ident.pl .ident_install_id $(COMPONENT)) \
          $(PFLAGS)
endif

PFLAGS := $(PFLAGS) -Wall -Wshadow \
          -DDEF_TOS_AM_GROUP=$(DEFAULT_LOCAL_GROUP) $(NESC_FLAGS)

ifndef TINYSEC
TINYSEC        := false # default: disable tinysec
endif
# The tinysec keyfile to use and the default key name (this re matches the
# first key. you can explicitly list keys by: make mica KEYNAME=mykeyname
KEYFILE        := $(HOME)/.tinyos_keyfile
KEYNAME         := '\w+'

ifeq ($(TINYSEC),true)
TINYSEC_KEY    := $(shell mote-key -kf $(KEYFILE) -kn $(KEYNAME))
ifeq ($(TINYSEC_KEY),)
$(error tinysec key has not been properly set. It is needed for tinysec. \
       Check to make sure that the script exists)
endif
PFLAGS   := $(PFLAGS) -DTINYSEC_KEY="$(TINYSEC_KEY)" -DTINYSEC_KEYSIZE=8
endif

NCC            = ncc
LIBS          = -lm

#######################################################################
# Choose platform options, based on MAKECMDGOALS
#######################################################################


# be quieter....
#ifeq ($(VERBOSE_MAKE)_x, _x)
#MAKEFLAGS += -s
#endif
#export VERBOSE_MAKE

define USAGE


Usage:  make <platform>
        make all
        make clean
        make install[.n] <platform>
        make reinstall[.n] <platform> # no rebuild of target
        make docs <platform>

        Valid platforms are: $(PLATFORMS)


endef


PLATAUX=$(PLATFORMS) all
PLATFORM := $(filter $(PLATAUX), $(MAKECMDGOALS))
PFLAGS := -target=$(PLATFORM) $(PFLAGS)
MAKECMDGOALS := $(filter-out $(PLATAUX), $(MAKECMDGOALS))


#############################################################################
# Programming Boards : flags
#############################################################################
PROGRAMMER_FLAGS=-dprog=dapa $(PROGRAMMER_PART) \
                 $(PROGRAMMER_EXTRA_FLAGS)
PROGRAMMER := DAPA

#############################################################################
# Programming Boards : command line input
# Command line input for programmers:
#   none          : default to parallel programming board
#   MIB510=<dev>  : use mib510 serial port programming board at port <dev>
#   EPRB=<host>   : use eprb at hostname <host>
#   AVRISP=<dev>  : use AVRISP serial programmer at port <dev>
#############################################################################

### If MIB510 then
MIB5100 := $(subst MIB510=,,$(filter MIB510=%,$(MAKECMDGOALS)))
ifneq ($(MIB510_),)
  MIB510 := $(MIB5100)
endif
MAKECMDGOALS := $(filter-out MIB510=%,$(MAKECMDGOALS))

### If STK is a set environment variable or if STK=xxx appears on the command
### line, then take it to be a network address and program assuming an stk500
### module.
EPRB0 := $(subst EPRB=,,$(filter EPRB=%,$(MAKECMDGOALS)))
ifneq ($(EPRB_),)
  EPRB := $(EPRB0)
endif
MAKECMDGOALS := $(filter-out EPRB=%,$(MAKECMDGOALS))

ifneq (x$(MIB510),x)
   PROGRAMMER := STK
   PROGRAMMER_FLAGS=-dprog=mib510 -dserial=$(MIB510) $(PROGRAMMER_PART) \
                    $(PROGRAMMER_EXTRA_FLAGS_MIB)
endif
ifneq (x$(EPRB),x)
  PROGRAMMER := STK
  PROGRAMMER_FLAGS=--dprog=stk500 --dhost=$(EPRB) $(PROGRAMMER_PART) \
                  $(PROGRAMMER_EXTRA_FLAGS_STK)
endif
ifdef AVRISP
  PROGRAMMER := STK
  PROGRAMMER_FLAGS=--dprog=stk500 --dserial=$(AVRISP) $(PROGRAMMER_PART) \
                  $(PROGRAMMER_EXTRA_FLAGS_AVRISP)
endif

### If the serial port has not been defined, then set it to the default.
ifeq (x$(SERIAL_PORT),x)
  SERIAL_PORT=/dev/ttyS0 --dspeed=115200
endif
### If the platform is jona, then use STK on the serial port.
### (The jona does not use a separate programmer device, so there
### will not be a programmer listed on the command line.)
ifeq ($(PLATFORM), jona)
  PROGRAMMER := STK
  PROGRAMMER_FLAGS=--dprog=stk500 --dserial=$(SERIAL_PORT) \
                  $(PROGRAMMER_PART)
endif

#Sensor Board Defaults
ifeq ($(SENSORBOARD),)
        ifeq ($(PLATFORM),mica)
                SENSORBOARD = micasb
        endif
        ifeq ($(PLATFORM),mica2)
                SENSORBOARD = micasb
        endif
        ifeq ($(PLATFORM),mica128)
                SENSORBOARD = micasb
        endif
        ifeq ($(PLATFORM),jona)
                SENSORBOARD = none
        endif
        ifeq ($(PLATFORM),rene2)
                SENSORBOARD = basicsb
        endif
        ifeq ($(PLATFORM),pc)
                SENSORBOARD = micasb
        endif
        ifeq ($(PLATFORM),mica2dot)
                SENSORBOARD = basicsb
        endif
endif

BUILDDIR = build/$(PLATFORM)
MAIN_EXE = $(BUILDDIR)/main.exe
MAIN_SREC = $(BUILDDIR)/main.srec

ifeq ($(PLATFORM), pc)
OPTFLAGS := -g -O0
PFLAGS := -pthread $(PFLAGS) -fnesc-nido-tosnodes=1000 \
        -fnesc-cfile=$(BUILDDIR)/app.c
MAIN_TARGET = $(MAIN_EXE)
else
PFLAGS := $(PFLAGS) -finline-limit=100000 -fnesc-cfile=$(BUILDDIR)/app.c
MAIN_TARGET = $(MAIN_SREC)
endif

PFLAGS := -board=$(SENSORBOARD) $(PFLAGS)

# added options to support network reprogramming. This sets the correct bootloader
# for mica2 and mica2dot platforms. And this also sets the programmer flag for
# native ATmega128.
ifeq ($(PLATFORM), mica)
PROGRAMMER_PART=-dpart=ATmega103 --wr_fuse_e=fd
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) \
        -dpart=ATmega103
endif
endif
ifeq ($(PLATFORM), mica128)
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) \
        -dpart=ATmega128
endif
endif
ifeq ($(PLATFORM), jona)
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
endif
ifeq ($(PLATFORM), mica2)
BOOTLOADER=$(XNP_DIR)/inpispm2.srec
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) \
        -dpart=ATmega128
endif
endif
ifeq ($(PLATFORM), mica2dot)
BOOTLOADER =$(XNP_DIR)/inpispm2d.srec
PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
ifdef AVRISP
  PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) \
        -dpart=ATmega128
endif
endif

######################################################################
# Rules for documentation generation
######################################################################
```

```
# add documentation flags to ncc, if requested
DOCS := $(filter docs, $(MAKECMDGOALS))
MAKECMDGOALS := $(filter-out docs, $(MAKECMDGOALS))
ifeq ($(DOCS)_x, docs_x)
build: FORCE
	@echo "    Making documentation for $(COMPONENT) on $(PLATFORM)"
	nesdoc $(DOCDIR)/$(PLATFORM) -fnesc-is-app $(PFLAGS) $(CFLAGS) \
		$(COMPONENT).nc
endif

# dummy rule for 'docs' target - so make won't complain about it
docs:
	@true



#########################################################################
# Rules for debugging
#########################################################################

# add documentation flags to ncc, if requested
DBG := $(filter debug, $(MAKECMDGOALS))
MAKECMDGOALS := $(filter-out debug, $(MAKECMDGOALS))
ifeq ($(DBG)_x, debug_x)
OPTFLAGS := -O1 -g -fnesc-no-inline
endif

# dummy rule for 'debug' target - so make won't complain about it
debug:
	@true


# For those who like debugging optimised code, there's debugopt
DBGOPT := $(filter debugopt, $(MAKECMDGOALS))
MAKECMDGOALS := $(filter-out debugopt, $(MAKECMDGOALS))
ifeq ($(DBGOPT)_x, debugopt_x)
OPTFLAGS := $(OPTFLAGS) -g
endif

# dummy rule for 'debug' target - so make won't complain about it
debugopt:
	@true



#########################################################################
# top-level rules.  switch based on MAKECMDGOALS
#########################################################################

#
# rules for make clean
#
ifeq ($(MAKECMDGOALS)_x, clean_x)
```

```
PLATFORM=

$(PLATAUX):
        @echo ""

else

ifeq ($(PLATFORM)_x,_x)
$(error $(PLATAUX) $(MAKECMDGOALS) $(USAGE))
endif

MAKECMDGOALS := $(patsubst install.%,install,$(MAKECMDGOALS))
MAKECMDGOALS := $(patsubst reinstall.%,reinstall,$(MAKECMDGOALS))

#
# rules for make install <platform>
#
ifeq ($(MAKECMDGOALS)_x, install_x)

$(PLATAUX):
        @true

else
ifeq ($(MAKECMDGOALS)_x, reinstall_x)

$(PLATAUX):
        @true

else
ifeq ($(MAKECMDGOALS)_x, inp_x)

$(PLATAUX):
        @true

else
all:
        for platform in $(PLATFORMS); do \
                $(MAKE) $$platform $(DOCS) || exit 1; \
        done

$(PLATFORMS): build

endif
endif
endif
endif

######################################################################
######################################################################
##                                                                 ##
##                       Begin main rules                          ##
##                                                                 ##
######################################################################
######################################################################
```

```
ifneq ($(DOCS)_x, docs_x)
build: $(MAIN_TARGET)
endif

install: $(MAIN_SREC) FORCE
        @$(MAKE) $(PLATFORM) re$@ PROGRAMMER="$(PROGRAMMER)" \
            PROGRAMMER_FLAGS="$(PROGRAMMER_FLAGS)"

install.%: $(MAIN_SREC) FORCE
        $(MAKE) $(PLATFORM) re$@ PROGRAMMER="$(PROGRAMMER)" \
            PROGRAMMER_FLAGS="$(PROGRAMMER_FLAGS)"

ifeq ($(PROGRAMMER),DAPA)  ### program via parallel port

reinstall: FORCE
        @echo "    installing $(PLATFORM) binary"
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --erase
        sleep 1
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --upload if=$(MAIN_SREC)
        sleep 1
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --verify if=$(MAIN_SREC)

reinstall.%: FORCE
        @echo "    installing $(PLATFORM) binary"
        $(SET_ID) $(MAIN_SREC) $(MAIN_SREC).out \
            `echo $@ |perl -pe 's/^reinstall.//; $$_=hex if /^0x/i;'`
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --erase
        sleep 1
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --upload if=$(MAIN_SREC).out
        sleep 1
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --verify if=$(MAIN_SREC).out

else  ### Otherwise, program via the stk500 where STK specifies a network address

reinstall: FORCE
        @echo "    installing $(PLATFORM) binary"
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --erase --upload  if=$(MAIN_SREC)

reinstall.%: FORCE
        @echo "    installing $(PLATFORM) binary"
        $(SET_ID) $(MAIN_SREC) $(MAIN_SREC).$*.out \
            `echo $@ |perl -pe 's/^reinstall.//; $$_=hex if /^0x/i;'`
        $(PROGRAMER) $(PROGRAMMER_FLAGS) --erase --upload  \
            if=$(MAIN_SREC).$*.out

endif  ### Done programming


$(MAIN_EXE): $(BUILDDIR) FORCE
        @echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
        $(NCC) -o $(MAIN_EXE) -v $(OPTFLAGS) $(PFLAGS) $(CFLAGS) \
            $(COMPONENT).nc $(LIBS) $(LDFLAGS)
        @echo "    compiled $(COMPONENT) to $@"
```

```
	@objdump -h $(MAIN_EXE) | perl -ne '$$b{$$1}=hex $$2 if \
	    /^\s*\d+\s*\.(text|data|bss)\s+(\S+)/; END { printf("%16d bytes in \
	      ROM\n%16d bytes in RAM\n",$$b{text}+$$b{data},$$b{bss}); }'

$(MAIN_SREC): $(MAIN_EXE)
	$(OBJCOPY) --output-target=srec $(MAIN_EXE) $(MAIN_SREC)

$(BUILDDIR):
	mkdir -p $(BUILDDIR)

clean: FORCE
	rm -rf $(BUILDDIR)
	rm -f core.*
	rm -f *~

# uploading boot loader for network reprogramming. Do this after loading app srec file.
# using either 'make install' or 'make reinstall'
inp: FORCE
	$(PROGRAMER) $(PROGRAMMER_FLAGS_INP) --upload if=$(BOOTLOADER)

FORCE:

.phony: FORCE
```

Appendix I

JONA Platform Directory Files

Below are the contents of the files in the JONA platform directory, in alphabetical order. See Section 4.2.4 for descriptions of these files organized by function.

I.1 .platform

```
@opts = ("-gcc=avr-gcc",
        "-mmcu=atmega128",
        "-fnesc-target=avr",
        "-fnesc-no-debug");

push @opts, "-mingw-gcc" if $cygwin;

@commonplatforms = ("mica128", "mica", "avrmote");
```

I.2 accel.h

```
/*
 * Authors:            Alec Woo, David Gay, Philip Levis
 * Date last modified:  6/25/02
 */

/**
 * @author Alec Woo
 * @author David Gay
 * @author PhilipLevis
 */

/**
 * @author Jamison Hope
 */
/**
 * All irrelevant portions of sensorboard.h have been commented out.
 * They have been been left in the file (rather than being deleted
 * entirely) so that its heritage is more obvious.
 */
enum {
  // TOSH_ACTUAL_PHOTO_PORT = 1,
  // TOSH_ACTUAL_TEMP_PORT = 1,
  // TOSH_ACTUAL_MIC_PORT = 2,
  TOSH_ACTUAL_ACCEL_X_PORT = 3,
  TOSH_ACTUAL_ACCEL_Y_PORT = 4,
  // TOSH_ACTUAL_MAG_X_PORT = 6,
  // TOSH_ACTUAL_MAG_Y_PORT = 5
};

enum {
  // TOS_ADC_PHOTO_PORT = 1,
  // TOS_ADC_TEMP_PORT = 2,
  // TOS_ADC_MIC_PORT = 3,
  TOS_ADC_ACCEL_X_PORT = 4,
  TOS_ADC_ACCEL_Y_PORT = 5,
  // TOS_ADC_MAG_X_PORT = 6,
  // TOS_ADC_VOLTAGE_PORT = 7,  defined this in hardware.h
  // TOS_ADC_MAG_Y_PORT = 8,
};

// enum {
//   TOS_MAG_POT_ADDR = 0,
//   TOS_MIC_POT_ADDR = 1
// };

// TOSH_ALIAS_PIN(PHOTO_CTL, INT1);
// TOSH_ALIAS_PIN(TEMP_CTL, INT2);
// TOSH_ALIAS_PIN(TONE_DECODE_SIGNAL, INT3);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MIC_CTL, PW3);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(SOUNDER_CTL, PW2);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(ACCEL_CTL, PW4);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MAG_CTL, PW5);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MIC_MUX_SEL, PW6);
```

I.3 Accel.nc

```
/**
 * This Accel.nc is based upon micasb's Accel.nc, with the "includes"
 * line changed to use accel.h rather than sensorboard.h.
 *
 * @author Jamison Hope
 */
includes accel;
configuration Accel
```

```
{
  provides interface ADC as AccelX;
  provides interface ADC as AccelY;
  provides interface StdControl;
}
implementation
{
  components AccelM, ADCC;

  StdControl = AccelM;
  AccelX = ADCC.ADC[TOS_ADC_ACCEL_X_PORT];
  AccelY = ADCC.ADC[TOS_ADC_ACCEL_Y_PORT];
  AccelM.ADCControl -> ADCC;
}
```

I.4 AccelM.nc

```
/*
 * Authors:    Alec Woo, Su Ping
 */
/**
 * @author Alec Woo
 * @author Su Ping
 */

/**
 * This AccelM.nc is based upon micasb's AccelM.nc, with the "includes"
 * line changed to use accel.h rather than sensorboard.h. It also
 * removes irrelevant "CTL" pin references.
 *
 *@author Jamison Hope
 */
includes accel;
module AccelM {
  provides interface StdControl;
  uses {
    interface ADCControl;
  }
}
implementation {

  command result_t StdControl.init() {
    call ADCControl.bindPort(TOS_ADC_ACCEL_X_PORT,
                             TOSH_ACTUAL_ACCEL_X_PORT);
    call ADCControl.bindPort(TOS_ADC_ACCEL_Y_PORT,
                             TOSH_ACTUAL_ACCEL_Y_PORT);
    dbg(DBG_BOOT, "ACCEL initialized.\n");
    return call ADCControl.init();
  }
  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
      return SUCCESS;
  }
}
```

I.5 ADCC.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified:  6/25/02
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

configuration ADCC
{
  provides {
    interface ADC[uint8_t port];
    interface ADCControl;
  }
}
implementation
{
  components ADCM, HPLADCM;

  ADC = ADCM;
  ADCControl = ADCM;
  ADCM.HPLADC -> HPLADCM;
}
```

I.6 ChannelMonC.nc

```
/*
 * Modified for JONA by Nathaniel Osgood and Jamison Hope.
 */
module ChannelMonC {
  provides interface ChannelMon;
  uses {
    interface Random;
  }
}
```

```
implementation {
  enum {
    IDLE_STATE,
    START_SYMBOL_SEARCH,
    PACKET_START,
    DISABLED_STATE
  };

  enum {
    SAMPLE_RATE = 100/2*4
  };

  unsigned short CM_search[2];
  char CM_state;
  unsigned char CM_lastBit;
  unsigned char CM_startSymBits;
  short CM_waiting;

  async command result_t ChannelMon.init() {
    atomic {
      CM_waiting = -1;
    }
    return call ChannelMon.startSymbolSearch();
  }

  async command result_t ChannelMon.startSymbolSearch() {
    atomic {
      //Reset to idle state.
      CM_state = IDLE_STATE;
      //set the RFM pins.
      TOSH_SET_RFM_CTL0_PIN();
      TOSH_SET_RFM_CTL1_PIN();
      TOSH_CLR_RFM_TXD_PIN();
      cbi(TIMSK, OCIE2); // clear interrupts
      cbi(TIMSK, TOIE2); // clear interrupts
      cbi(TIMSK, OCIE2); // clear interrupts
      outb(TCCR2, 0x0A); // scale the counter: modified from mica's 0x09 for no
                         // scaling to 0x0a for scale/8
      outb(OCR2, SAMPLE_RATE); // set upper byte of comp reg.
      sbi(TIMSK, OCIE2); // enable timer1 interrupt
      outb(TCNT2, 0x00); // clear current counter value
      sbi(DDRB, 6);
    }
    return SUCCESS;
  }

  TOSH_SIGNAL(SIG_OUTPUT_COMPARE2) {
    uint8_t bit = TOSH_READ_RFM_RXD_PIN();
    atomic { // Unnecessary, but nesC doesn't understand SIGNAL
      //fire the bit arrived event and send up the value.
      if (CM_state == IDLE_STATE) {
        CM_search[0] <<= 1;
        CM_search[0] = CM_search[0] | (bit & 0x1);
        if(CM_waiting != -1){
```

```
        CM_waiting --;
        if(CM_waiting == 1){
          if ((CM_search[0] & 0xfff) == 0) {
            CM_waiting = -1;
            signal ChannelMon.idleDetect();
          }else{
            CM_waiting = (call Random.rand() & 0x1f) + 30;
          }
        }
      }
      if ((CM_search[0] & 0x777) == 0x707){
        CM_state = START_SYMBOL_SEARCH;
        CM_search[0] = CM_search[1] = 0;
        CM_startSymBits = 30;
      }
    }else if(CM_state == START_SYMBOL_SEARCH){
      unsigned int current = CM_search[CM_lastBit];
      CM_startSymBits--;
      if (CM_startSymBits == 0){
        CM_state = IDLE_STATE;
      }
      if (CM_state != IDLE_STATE) {
        current <<= 1;
        current &=  0x1ff;  // start symbol is 9 bits
        if(bit) current |=  0x1;  // start symbol is 9 bits
        if (current == 0x135) {
          cbi(TIMSK, OCIE2);
          CM_state = IDLE_STATE;
          signal ChannelMon.startSymDetect();
        }
        if (CM_state != IDLE_STATE) {
          CM_search[CM_lastBit] = current;
          CM_lastBit ^= 1;
        }
      }
    }
  }
  return;
}

async command result_t ChannelMon.stopMonitorChannel() {
  //disable timer
  atomic {
    cbi(TIMSK, OCIE2);
    CM_state = DISABLED_STATE;
  }
  return SUCCESS;
}

async command result_t ChannelMon.macDelay() {
  atomic {
    CM_search[0] = 0xff;
    if(CM_waiting == -1) {
      CM_waiting = (call Random.rand() & 0x2f) + 80;
      }
    }

    return SUCCESS;
  }
}
```

I.7 hardware.h

```
/* Authors:            Jason Hill, Philip Levis, Nelson Lee, David Gay,
 *                     Nathaniel Osgood, Jamison Hope
 */
/**
 * @author Jason Hill
 * @author Philip Levis
 * @author Nelson Lee
 * @author David Gay
 * @author Nathaniel Osgood
 * @author Jamison Hope
 */

/**
 * The JONA hardware.h header file. It is based upon the header files of
 * mica, mica128, and mica2.
 */
#ifndef TOSH_HARDWARE_H
#define TOSH_HARDWARE_H

#define TOSH_NEW_AVRLIBC // jona requires avrlibc v. 20021209 or greater
#include <avrhardware.h>

// avrlibc may define ADC as a 16-bit register read.  This collides
// with the nesc ADC interface name
uint16_t inline getADC() {
  return inw(ADC);
}
#undef ADC

TOSH_ASSIGN_PIN(DEBUG, A, 0);
TOSH_ASSIGN_PIN(LED, B, 7);

/* HPLPowerManagementM.nc, which is inherited from MICA, uses
 * ATmega103 names for the registers. In particular, it calls
 * the UART control register "UCR". We must map this name
 * to the appropriate control register in the ATmega128, namely
 * USART0's control and status register B: UCSR0B.*/
#define UCR UCSR0B

TOSH_ASSIGN_PIN(INT1, D, 1);
TOSH_ASSIGN_PIN(INT2, D, 2);
TOSH_ASSIGN_PIN(INT3, D, 3);

TOSH_ASSIGN_PIN(RFM_RXD,  B, 2);
TOSH_ASSIGN_PIN(RFM_TXD,  B, 3);
TOSH_ASSIGN_PIN(RFM_CTL0, D, 7);
TOSH_ASSIGN_PIN(RFM_CTL1, D, 6);

// These are for SOFTWARE based I2C on pins A4 and A5 only
// Hardware based I2C uses different pins. (D0, D1)
TOSH_ASSIGN_PIN(I2C_BUS1_SCL, A, 4);
TOSH_ASSIGN_PIN(I2C_BUS1_SDA, A, 5);

TOSH_ASSIGN_PIN(UART_RXD0, E, 0);
TOSH_ASSIGN_PIN(UART_TXD0, E, 1);
TOSH_ASSIGN_PIN(UART_RXD1, D, 2);
TOSH_ASSIGN_PIN(UART_TXD1, D, 3);

void TOSH_SET_PIN_DIRECTIONS(void)
{
  outb(DDRA, 0x00);
  outb(DDRB, 0x00);
  outb(DDRD, 0x00);
  outb(DDRE, 0x02);
  outb(PORTE, 0x02);
  TOSH_MAKE_LED_OUTPUT();

  TOSH_MAKE_RFM_CTL0_OUTPUT();
  TOSH_MAKE_RFM_CTL1_OUTPUT();
  TOSH_MAKE_RFM_TXD_OUTPUT();

  TOSH_SET_LED_PIN();
}

// Sensor-independent ADC constants

// The number of virtual ADC channels.
enum {
  TOSH_ADC_PORTMAPSIZE = 12
};

enum
{
  TOSH_ACTUAL_VOLTAGE_PORT = 7,
  TOSH_ACTUAL_BANDGAP_PORT = 30, // Per Table 98 of the ATmega128 manual
  TOSH_ACTUAL_GND_PORT = 31, // (and mica2's hardware.h)
};
enum
{
  TOS_ADC_VOLTAGE_PORT = 7,
  TOS_ADC_BANDGAP_PORT = 10, // Borrowed from mica2's hardware.h
  TOS_ADC_GND_PORT = 11, //
};

#endif //TOSH_HARDWARE_H
```

I.8 HPLADCM.nc

```
/*
 * Authors:         Jason Hill, David Gay, Philip Levis
 * Version:         $Id: HPLADCM.nc,v 1.6.4.6 2003/08/26 09:08:16 cssharp Exp $
 */

// The hardware presentation layer. See hpl.h for the C side.
// Note: there's a separate C side (hpl.h) to get access to the avr macros

// The model is that HPL is stateless. If the desired interface is as stateless
// it can be implemented here (Clock, FlashBitSPI). Otherwise you should
// create a separate component

/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * This is the mica2 HPLADCM, modified for use with the jona hardware.
 * This modification mainly entails replacing the deprecated ``outp''
 * with its replacement, ``outb'' (see <avr/sfr_defs.h>).
 *
 * @author Jamison Hope
 */
module HPLADCM {
  provides {
    interface StdControl;
    interface HPLADC as ADC;
  }
}
implementation
{
  /* The port mapping table */
  bool init_portmap_done;
  uint8_t TOSH_adc_portmap[TOSH_ADC_PORTMAPSIZE];

  void init_portmap() {
    /* The default ADC port mapping */
    atomic {
      if( init_portmap_done == FALSE ) {
        int i;
        for (i = 0; i < TOSH_ADC_PORTMAPSIZE; i++)
          TOSH_adc_portmap[i] = i;

        // Setup fixed bindings associated with ATmega128 ADC
        TOSH_adc_portmap[TOS_ADC_BANDGAP_PORT]
                        = TOSH_ACTUAL_BANDGAP_PORT;
        TOSH_adc_portmap[TOS_ADC_GND_PORT] = TOSH_ACTUAL_GND_PORT;
        init_portmap_done = TRUE;
      }
  }
}

command result_t StdControl.init() {
  call ADC.init();
}

command result_t StdControl.start() {
}

command result_t StdControl.stop() {
  cbi(ADCSR,ADEN);
}

async command result_t ADC.init() {
  init_portmap();

  // Enable ADC Interupts,
  // Set Prescaler division factor to 64
  atomic {
    outb(ADCSR, ((1 << ADIE) | (6 << ADPS0)));

    outb(ADMUX,0);
  }
  return SUCCESS;
}

async command result_t ADC.setSamplingRate(uint8_t rate) {
  uint8_t current_val = inb(ADCSR);
  current_val = (current_val & 0xF8) | (rate & 0x07);
  outb(ADCSR, current_val);
  return SUCCESS;
}

async command result_t ADC.bindPort(uint8_t port, uint8_t adcPort) {
  if (port < TOSH_ADC_PORTMAPSIZE &&
      port != TOS_ADC_BANDGAP_PORT &&
      port != TOS_ADC_GND_PORT) {
    init_portmap();
    atomic TOSH_adc_portmap[port] = adcPort;
    return SUCCESS;
  }
  else
    return FAIL;
}

async command result_t ADC.samplePort(uint8_t port) {
  atomic {
    outb(ADMUX, (TOSH_adc_portmap[port] & 0x1F));
  }
  sbi(ADCSR, ADEN);
  sbi(ADCSR, ADSC);

  return SUCCESS;
  }

  async command result_t ADC.sampleAgain() {
    sbi(ADCSR, ADSC);
    return SUCCESS;
  }

  async command result_t ADC.sampleStop() {
    // SIG_ADC does the stop
    return SUCCESS;
  }

  default async event result_t ADC.dataReady(uint16_t done) { return SUCCESS; }

  TOSH_SIGNAL(SIG_ADC) {
    uint16_t data = inw(ADCL);
    data &= 0x3ff;
    sbi(ADCSR, ADIF);
    cbi(ADCSR, ADEN);
    __nesc_enable_interrupt();
    signal ADC.dataReady(data);
  }
}
```

I.9 HPLPotC.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified: 6/25/02
 */

// The hardware presentation layer. See hpl.h for the C side.
// Note: there's a separate C side (hpl.h) to get access to the avr macros

// The model is that HPL is stateless. If the desired interface is as stateless
// it can be implemented here (Clock, FlashBitSPI). Otherwise you should
// create a separate component
// Pot is not used in mica2dot

/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * The JONA has no potentiometer, so do nothing.
 *
 * @author Jamison Hope
 */
module HPLPotC {
  provides interface HPLPot as Pot;
}
implementation
{
  command result_t Pot.decrease() {
    return SUCCESS;
  }

  command result_t Pot.increase() {
    return SUCCESS;
  }

  command result_t Pot.finalise() {
    return SUCCESS;
  }
}
```

I.10 HPLSlavePinC.nc

```
/*
 * Authors:             Jason Hill, David Gay, Philip Levis
 * Date last modified:  6/25/02
 */

// Low-level slave pin control

/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * The JONA has no slave pin, so do nothing.
 *
 * @author Jamison Hope
 */
module HPLSlavePinC {
  provides interface HPLSlavePin as SlavePin;
}
implementation
{
  async command result_t SlavePin.high() {
    return SUCCESS;
  }

  async command result_t SlavePin.low() {
    return SUCCESS;
  }
}
```

I.11 HPLUARTM.nc

```
module HPLUARTM {
  provides interface HPLUART as UART;
}
implementation {

  async command result_t UART.init() {

    // Set up UART for 9600 baud rate:
    outb(UBRR0H, 0); // per ATmega128 manual table 74: 95 = (osc/(16*baud))-1
    outb(UBRR0L,95); // baud = 9600, osc = 14.7456E6

    // Enable reciever and transmitter and their interrupts
    outb(UCSR0B, ((1<<RXCIE) | (1<<TXCIE) | (1<<RXEN) | (1<<TXEN)));

    // Set frame format: 8 data-bits, 1 stop-bit
    outb(UCSR0C, ((1<<UCSZ1) | (1<<UCSZ0)));

    return SUCCESS;
  }

  async command result_t UART.stop() {

    // Clear out USART0's three status registers.
    outb(UCSR0A, 0x00);
    outb(UCSR0B, 0x00);
    outb(UCSR0C, 0x00);

    return SUCCESS;
  }

  async default event result_t UART.get(uint8_t data) { return SUCCESS; }
  TOSH_SIGNAL(SIG_UART0_RECV) {
    if (inb(UCSR0A) & (1 << RXC))
      signal UART.get(inb(UDR0));
  }

  async default event result_t UART.putDone() { return SUCCESS; }
  TOSH_INTERRUPT(SIG_UART0_TRANS) {
    signal UART.putDone();
  }

  async command result_t UART.put(uint8_t data) {
    sbi(UCSR0A, TXC);
    outb(UDR0, data);
    return SUCCESS;
  }
}
```

I.12 IntToLed.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified: 6/25/02
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/*
 * Author:             Jamison Hope
 * Date last modified: 5/10/05
 */
/**
 * @author Jamison Hope
 */
/**
 * IntToLedsM modified to use Led
 */
configuration IntToLed
{
  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  components IntToLedM, LedC;

  IntOutput = IntToLedM.IntOutput;
  StdControl = IntToLedM.StdControl;
  IntToLedM.Led -> LedC.Led;
}
```

I.13 IntToLedM.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis, Nelson Lee
 * Date last modified: 6/25/02
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 * @author Nelson Lee
 */

/*
```

```
 * Author:              Jamison Hope
 * Date last modified: 5/10/05
 */
/**
 * @author Jamison Hope
 */
/**
 * IntToLedsM modified to use Led
 */
module IntToLedM {
  uses interface Led;

  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  command result_t StdControl.init()
  {
    call Led.init();
    call Led.Off();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }

  task void outputDone()
  {
    signal IntOutput.outputComplete(SUCCESS);
  }

  command result_t IntOutput.output(uint16_t value)
  {
    if (value & 1) call Led.On();
    else call Led.Off();

    post outputDone();
    return SUCCESS;
  }
}
```

I.14 IntToLedsArray.nc

```
/*
 * Authors:             Jason Hill, David Gay, Philip Levis
 * Date last modified:  6/25/02
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/*
 * Author:              Jamison Hope
 * Date last modified: 11/4/04
 */
/**
 * @author Jamison Hope
 */

/**
 * IntToLeds modified to use LedsArray instead.
 */
configuration IntToLedsArray
{
  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  components IntToLedsArrayM, LedsArrayC;

  IntOutput = IntToLedsArrayM.IntOutput;
  StdControl = IntToLedsArrayM.StdControl;
  IntToLedsArrayM.LedsArray -> LedsArrayC.LedsArray;
}
```

I.15 IntToLedsArrayM.nc

```
/*
 * Authors:             Jason Hill, David Gay, Philip Levis, Nelson Lee
 * Date last modified:  6/25/02
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 * @author Nelson Lee
 */

/*
 * Author:              Jamison Hope
 * Date last modified: 5/10/05
 */
/**
```

```
 * @author Jamison Hope
 */

/**
 * IntToLedsM modified to use LedsArray
 */
module IntToLedsArrayM {
  uses interface LedsArray;

  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  command result_t StdControl.init()
  {
    call LedsArray.init();
    call LedsArray.allOff();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }

  task void outputDone()
  {
    signal IntOutput.outputComplete(SUCCESS);
  }

  command result_t IntOutput.output(uint16_t value)
  {
    call LedsArray.setv(value);

    post outputDone();
    return SUCCESS;
  }
}
```

I.16 Led.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified: 6/1/03
 */
/**
 * Abstraction of the LEDs.
 *
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * Based upon TinyOS's Leds, this interface provides access to the LED
 * on the PROBOmega128.
 *
 * @author Nathaniel Osgood
 * @author Jamison Hope
 */
interface Led {

  /**
   * Initialize the LED; among other things, initialization turns
   * it off.
   *
   * @return SUCCESS always.
   */
  async command result_t init();

  /**
   * Turn the LED on.
   *
   * @return SUCCESS always.
   */
  async command result_t On();

  /**
   * Turn the LED off.
   *
   * @return SUCCESS always.
   */
  async command result_t Off();

  /**
   * Toggle the LED. If it was on, turn it off. If it was off,
   * turn it on.
   *
   * @return SUCCESS always.
   */
  async command result_t Toggle();


  /**
   * Get current Led information
   *
   * @return A uint8_t typed value representing Led status
   */
   async command uint8_t get();

  /**
  * Set Led to a specified value
  *
  * @param value ranging from 0 to 1 inclusive; for higher
  *        values, an even number will turn off the LED, and
  *        an odd number will turn it on.
  *
  * @return SUCCESS Always
  */
  async command result_t set(uint8_t value);
}
```

I.17 LedC.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified: 6/2/03
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * Based upon TinyOS's LedsC, this component provides access to the LED
 * on the PROBOmega128.
 *
 * @author Nathaniel Osgood
 * @author Jamison Hope
 */
module LedC {
  provides interface Led;
}
implementation
{
  uint8_t fOn;

  async command result_t Led.init() {
    atomic {
      fOn = 0;
      TOSH_SET_LED_PIN();
    }
    return SUCCESS;
  }

  async command result_t Led.On() {
    atomic {
      TOSH_CLR_LED_PIN();
      fOn = 1;
    }
    return SUCCESS;
  }

  async command result_t Led.Off() {
     atomic {
       TOSH_SET_LED_PIN();
       fOn = 0;
     }
     return SUCCESS;
  }

  async command result_t Led.Toggle() {
    result_t rval;
    atomic {
      if (fOn)
        rval = call Led.Off();
      else
        rval = call Led.On();
    }
    return rval;
  }

  async command uint8_t Led.get() {
    uint8_t rval;
    atomic {
      rval = fOn;
    }
    return rval;
  }

  async command result_t Led.set(uint8_t p_fOn) {
    result_t rval;
    atomic {
      if (p_fOn % 2) // if p_fOn is even,
        rval = call Led.Off();
      else // p_fOn is odd
        rval = call Led.On();
    }
    return rval;
  }
}
```

I.18 LedsArray.nc

```
/**
 * Interface file for an array of 10 LEDs intended to be connected
 * to PC1, PC0, PA7..PA0 (i.e. PC1 is the MSB and PA0 is the LSB).
 *
 * @author Jamison Hope
 */
interface LedsArray {
```

```
/**
 * Sets the DDRs for the ten pins to be output, and initializes
 * the LedsArray to display 0x000.
 */
async command result_t init();

/**
 * Turns on all LEDs, regardless of previous state.
 */
async command result_t allOn();

/**
 * Turns off all LEDs, regardless of previous state.
 */
async command result_t allOff();

/**
 * Toggles all LEDs. Those which were on will now be off, and
 * vice versa.
 */
async command result_t allToggle();

/**
 * Returns a 16-bit unsigned integer holding a value in the range
 * [0,0x3FF] representing the current displayed value.
 */
async command uint16_t get();

/**
 * Displays value, turning on and off the LEDs as appropriate.
 */
async command result_t setv(uint16_t value);

/**
 * Increments the displayed value. Same as setv( get() + 1 ).
 */
async command result_t inc();

/**
 * Decrements the displayed value. Same as setv( get() - 1 ).
 */
async command result_t dec();

/**
 * Turns the corresponding LEDs on without affecting the states of
 * the others.
 */
async command result_t setav(uint16_t value);

/**
 * Clears the corresponding LEDs without affecting the states of
 * the others.
 */
async command result_t clrav(uint16_t value);
```

```
}
```

I.19 LedsArrayC.nc

```
/**
 * Implementation file for an array of 10 LEDs intended to be connected
 * to PC1, PC0, PA7..PA0 (i.e. PC1 is the MSB and PA0 is the LSB).
 *
 * @author Jamison Hope
 */
module LedsArrayC {
  provides interface LedsArray;
}
implementation
{
  uint16_t ledsOn;

  // Defined later, does actual setting of pins.
  void output(uint16_t num);

  /**
   * Sets PC1..0 and PA7..0 to be output, leaves LEDs initially off.
   */
  async command result_t LedsArray.init() {
    atomic {
      ledsOn = 0;

      DDRC |= 0x3;
      DDRA = 0xFF;
      output(0x3FF);
    }
    return SUCCESS;
  }

  /**
   * Turns all LEDs on.
   */
  async command result_t LedsArray.allOn() {
    atomic {
      output(0);
      ledsOn = 0x3FF;
    }
    return SUCCESS;
  }

  /**
   * Turns all LEDs off.
   */
  async command result_t LedsArray.allOff() {
    atomic {
      ledsOn = 0;
    output(0x3FF);
  }
  return SUCCESS;
}

/**
 * Toggles all LEDs.
 */
async command result_t LedsArray.allToggle() {
  atomic {
    output(ledsOn);
    ledsOn = (~ledsOn & 0x3FF);
  }
  return SUCCESS;
}

/**
 * Return the displayed value.
 */
async command uint16_t LedsArray.get() {
  uint16_t rval;
  atomic {
    rval = ledsOn;
  }
  return rval;
}

/**
 * Set the displayed value.
 */
async command result_t LedsArray.setv(uint16_t ledsNum) {
  atomic {
    ledsOn = (ledsNum & 0x3FF);
    output(~ledsOn & 0x3FF);
  }
  return SUCCESS;
}

/**
 * Increment the displayed value.
 */
async command result_t LedsArray.inc() {
  atomic {
    ledsOn = (ledsOn+1) & 0x3FF;
    output(~ledsOn & 0x3FF);
  }
  return SUCCESS;
}

/**
 * Decrement the displayed value.
 */
async command result_t LedsArray.dec() {
  atomic {
      ledsOn = (ledsOn-1) & 0x3FF;
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }

  /**
   * Turn on additional LEDs indicated by parameter.
   */
  async command result_t LedsArray.setav(uint16_t turnOn) {
    atomic {
      ledsOn |= (turnOn & 0x3FF);
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }

  /**
   * Turn off additional LEDs indicated by parameter.
   */
  async command result_t LedsArray.clrav(uint16_t turnOff) {
    atomic {
      ledsOn &= (0x3FF & ~turnOff);
      output(~ledsOn & 0x3FF);
    }
    return SUCCESS;
  }

  /**
   * Set the ten LedsArray pins to the value indicated by
   * the parameter, which should be in [0,0x3FF]. Numbers
   * outside of this range may affect non-LedsArray pins
   * of Port C.
   */
  void output(uint16_t num) {
    PORTC = (PORTC & 0xFC) | (num >> 8);
    PORTA = num & 0xFF;
  }
}
```

I.20 LedsC.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified:  6/2/03
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */
```

```
/**
 * Overrides the TinyOS LedsC component to map all Leds commands
 * to the JONA Led. Uses module in file LedsM.nc.
 *
 * @author Jamison Hope
 */
configuration LedsC {
  provides interface Leds;

}
implementation {
  components LedC, LedsM;

  Leds = LedsM.Leds;
  LedsM.Led -> LedC;
}
```

I.21 LedsM.nc

```
/*
 * Authors:            Jason Hill, David Gay, Philip Levis
 * Date last modified: 6/2/03
 */
/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */

/**
 * The TinyOS LedsC component modified to map all Leds commands
 * to the JONA Led. Wirings are made in JONA's LedsC.nc.
 *
 * @author Jamison Hope
 */
module LedsM {
  provides interface Leds;
  uses interface Led;
}
implementation
{
  async command result_t Leds.init() {
    return call Led.init();
  }

  async command result_t Leds.redOn() {
    return call Led.On();
  }

  async command result_t Leds.redOff() {
     return call Led.Off();
  }

  async command result_t Leds.redToggle() {
    return call Led.Toggle();
  }

  async command result_t Leds.greenOn() {
    return call Led.On();
  }

  async command result_t Leds.greenOff() {
    return call Led.Off();
  }

  async command result_t Leds.greenToggle() {
    return call Led.Toggle();
  }

  async command result_t Leds.yellowOn() {
    return call Led.On();
  }

  async command result_t Leds.yellowOff() {
    return call Led.Off();
  }

  async command result_t Leds.yellowToggle() {
    return call Led.Toggle();
  }

  async command uint8_t Leds.get() {
    return call Led.get();
  }

  async command result_t Leds.set(uint8_t ledsNum) {
    return call Led.set(ledsNum);
  }
}
```

I.22 photo.h

```
/*
 * Authors:            Alec Woo, David Gay, Philip Levis
 * Date last modified:  6/25/02
 */
/**
 * @author Alec Woo
 * @author David Gay
 * @author Philip Levis
 */
```

```
/**
 * @author Jamison Hope
 */
/**
 * All irrelevant portions of sensorboard.h have been commented out.
 * They have been been left in the file (rather than being deleted
 * entirely) so that its heritage is more obvious.
 */
enum {
  TOSH_ACTUAL_PHOTO_PORT = 1,
  // TOSH_ACTUAL_TEMP_PORT = 1,
  // TOSH_ACTUAL_MIC_PORT = 2,
  // TOSH_ACTUAL_ACCEL_X_PORT = 3,
  // TOSH_ACTUAL_ACCEL_Y_PORT = 4,
  // TOSH_ACTUAL_MAG_X_PORT = 6,
  // TOSH_ACTUAL_MAG_Y_PORT = 5
};

enum {
  TOS_ADC_PHOTO_PORT = 1,
  // TOS_ADC_TEMP_PORT = 2,
  // TOS_ADC_MIC_PORT = 3,
  // TOS_ADC_ACCEL_X_PORT = 4,
  // TOS_ADC_ACCEL_Y_PORT = 5,
  // TOS_ADC_MAG_X_PORT = 6,
  // TOS_ADC_VOLTAGE_PORT = 7, defined this in hardware.h
  // TOS_ADC_MAG_Y_PORT = 8,
};

// enum {
// TOS_MAG_POT_ADDR = 0,
// TOS_MIC_POT_ADDR = 1
// };

// TOSH_ALIAS_PIN(PHOTO_CTL, INT1);
// TOSH_ALIAS_PIN(TEMP_CTL, INT2);
// TOSH_ALIAS_PIN(TONE_DECODE_SIGNAL, INT3);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MIC_CTL, PW3);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(SOUNDER_CTL, PW2);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(ACCEL_CTL, PW4);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MAG_CTL, PW5);
// TOSH_ALIAS_OUTPUT_ONLY_PIN(MIC_MUX_SEL, PW6);
```

I.23 Photo.nc

```
/**
 * This Photo.nc is based upon micasb's Accel.nc, with the "includes"
 * line changed to use photo.h rather than sensorboard.h. It has also
 * been modified to be appropriate to Photo rather than Accel.
 *
 * @author Jamison Hope
 */
includes photo;
configuration Photo
{
  provides interface ADC as PhotoADC;
  provides interface StdControl;
}
implementation
{
  components PhotoM, ADCC;

  StdControl = PhotoM;
  PhotoADC = ADCC.ADC[TOS_ADC_PHOTO_PORT];
  PhotoM.ADCControl -> ADCC;
}
```

I.24 PhotoM.nc

```
/*
 * Authors:    Alec Woo, Su Ping
 */
/**
 * @author Alec Woo
 * @author Su Ping
 */

/**
 * This Photo.nc is based upon micasb's Accel.nc, with the "includes"
 * line changed to use photo.h rather than sensorboard.h. It has also
 * been modified to be appropriate to Photo rather than Accel, and it
 * removes irrelevant "CTL" pin references.
 *
 * @author Jamison Hope
 */
includes photo;
module PhotoM {
  provides interface StdControl;
  uses {
    interface ADCControl;
  }
}
implementation {

  command result_t StdControl.init() {
    call ADCControl.bindPort(TOS_ADC_PHOTO_PORT,
                             TOSH_ACTUAL_PHOTO_PORT);
    dbg(DBG_BOOT, "PHOTO initialized.\n");
    return call ADCControl.init();
  }
  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
      return SUCCESS;
  }
}
```

I.25 RadioTimingC.nc

```
/*
 * Modified for JONA by Nathaniel Osgood and Jamison Hope.
 */
module RadioTimingC {
  provides interface RadioTiming;
}
implementation {

  async command uint16_t RadioTiming.getTiming() {
    //enable input capture.
    cbi(DDRB, 4);
    while(TOSH_READ_RFM_RXD_PIN()) { }
    outb(TCCR1B, 0x42);  // Changed from mica's 0x41 so it would get clk/8 from
                         // prescaler (per modification to ChannelMonC and
                         // SpiByteFifoC, as necessitated by the higher speed
                         // of the PROBOmega oscillator and limits of the RFM
                         // chip)
    //clear capture flag
    outb(TIFR, 0x1<<ICF1);
    //wait for the capture.
    while((inb(TIFR) & (0x1 << ICF1)) == 0) { }
    sbi(PORTB, 6);
    cbi(PORTB, 6);
    return __inw_atomic(ICR1L);
  }

  async command uint16_t RadioTiming.currentTime() {
    return __inw_atomic(TCNT1L);
  }
}
```

I.26 SpiByteFifoC.nc

```
/*
 * Modified for JONA by Nathaniel Osgood and Jamison Hope.
 */
module SpiByteFifoC
{
  provides interface SpiByteFifo;
  uses interface SlavePin;  // Not really, but other TinyOS files think so;
                            // we either have to change them all, or fake
                            // it here.
}
implementation
{
  uint8_t nextByte;
  uint8_t state;

  enum {
    IDLE,
    FULL,
    OPEN,
    READING
  };

  enum {
    BIT_RATE = 20 * 4 / 2 * 5/4
  };


  TOSH_SIGNAL(SIG_SPI) {
    uint8_t temp = inb(SPDR);
    // Assume state == FULL (we've missed a deadline and are dead if it
    // isn't...)
    outb(SPDR, nextByte);
    state = OPEN;
    signal SpiByteFifo.dataReady(temp);
  }

  async command result_t SpiByteFifo.send(uint8_t data) {
    result_t rval = FAIL;
    atomic {
      if(state == OPEN){
        nextByte = data;
        state = FULL;
        rval = SUCCESS;
      }
      else if(state == IDLE){
        state = OPEN;
        signal SpiByteFifo.dataReady(0);
        cbi(PORTB, 7);
        sbi(DDRB, 7);
        outb(SPCR, 0xc0);
        outb(SPDR, data);
        //set the radio to TX.
        TOSH_CLR_RFM_CTL0_PIN();
        TOSH_SET_RFM_CTL1_PIN();
        //start the timer.
        cbi(TIMSK, TOIE2);
        cbi(TIMSK, OCIE2);
        outb(TCNT2, 0);
        outb(OCR2, BIT_RATE);
        outb(TCCR2, 0x1a);  // Changed scale from mica's no scaling (0x19) to
```

```
                              // clk/8 (0x1a)
    rval = SUCCESS;
   }
  }
  return rval;
}

async command result_t SpiByteFifo.idle() {
  atomic {
    outb(SPCR,  0x00);
    outb(SPDR,  0x00);
    outb(TCCR2, 0x00);
    nextByte = 0;
    TOSH_MAKE_RFM_TXD_OUTPUT();
    TOSH_CLR_RFM_TXD_PIN();
    TOSH_CLR_RFM_CTL0_PIN();
    TOSH_CLR_RFM_CTL1_PIN();
    state = IDLE;
    nextByte = 0;
  }
  return SUCCESS;
}

async command result_t SpiByteFifo.startReadBytes(uint16_t timing) {
  uint8_t oldState;
  // This state transition is sufficient because no other
  // function can execute when in the READING state. That is,
  // except txMode() and idle(), but they only modify the RFM control
  // pins, which this function doesn't deal with. - pal
  atomic {
    oldState = state;
    if (state == IDLE) {
      state = READING;
    }
  }
  if(oldState == IDLE){
    outb(SPCR, 0x00);
    cbi(PORTB, 7);
    sbi(DDRB, 7);
    outb(TCCR2, 0x0);
    outb(TCNT2, 0x1);
    outb(OCR2, BIT_RATE);
    //don't change the radio state.
    timing += (400-19);
    if(timing > 0xfff0) timing = 0xfff0;
    //set the phase of the clock line
    outb(TCCR2, 0x1a); // Changed scale from mica's no scaling (0x19)
                       // to clk/8 (0x1a)
    outb(TCNT2, BIT_RATE - 20);
    while(inb(PINB) & 0x80){;}
    while(__inw(TCNT1L) < timing){outb(TCNT2, 0x0);}
    outb(SPCR, 0xc0);
    outb(SPDR, 0x00);
    sbi(PORTB, 6);
    cbi(PORTB, 6);
    return SUCCESS;
  }
  return FAIL;
}

async command result_t SpiByteFifo.txMode() {
  atomic {
    TOSH_CLR_RFM_CTL0_PIN();
    TOSH_SET_RFM_CTL1_PIN();
  }
  return SUCCESS;
}

async command result_t SpiByteFifo.rxMode() {
  atomic {
    TOSH_CLR_RFM_TXD_PIN();
    TOSH_MAKE_RFM_TXD_INPUT();
    TOSH_SET_RFM_CTL0_PIN();
    TOSH_SET_RFM_CTL1_PIN();
  }
  return SUCCESS;
}

async command result_t SpiByteFifo.phaseShift() {
  unsigned char f;
  atomic {
    f = inb(TCNT2);
    if(f > 20) f -= 20;
    outb(TCNT2, f);
  }
  return SUCCESS;
}

/**
 * Even though JONA has no chance of signalling this event,
 * we must provide an implementation since we officially
 * use the SlavePin interface.
 */
event result_t SlavePin.notifyHigh() {
  return SUCCESS;
}
}
```

Appendix J

Sample Applications

Below are the complete code listings for the sample applications discussed in Chapter 5. In order, they are:

1. TestLedsArray (KnightRider)
2. TestUartSendReceive
3. OscilloscopeJonaRF
4. HumiditySense

J.1 KnightRider

J.1.1 Makefile

```
COMPONENT=KnightRider
include ../Makerules
```

J.1.2 KnightRider.nc

```
// "I made this." -- Jamison Hope

/**
 * Configuration for TestLedsArray application.
 **/
configuration KnightRider {
}
implementation {
  components Main, KnightRiderM, TimerC, LedsArrayC;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> KnightRiderM.StdControl;

  KnightRiderM.Timer -> TimerC.Timer[unique("Timer")];
  KnightRiderM.LedsArray -> LedsArrayC;
}
```

J.1.3 KnightRiderM.nc

```
// "I made this." -- Jamison Hope

/**
 * Implementation for TestLedsArray application.
 **/
module KnightRiderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface LedsArray;
  }
}
implementation {
  uint8_t dir = 0;

  /**
   * Initialize TestLedsArrayM. In this version,
   * set up LedsArray and then output an initial
   * value of 0x001.
   **/
  command result_t StdControl.init() {
    call LedsArray.init();
    return call LedsArray.setv(1);
  }

  /**
   * Start the repeating timer.
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 100ms
    return call Timer.start(TIMER_REPEAT, 100);
  }

  /**
   * Stop the timer.
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  /**
   * Recall previous value, update it, and
   * display the result. When either end
   * is reached, reverse the direction.
   **/
  event result_t Timer.fired() {
    uint16_t cnt = call LedsArray.get();

    if( !dir && (cnt < 0x200) ) {
      cnt <<= 1;
    } else if( dir && (cnt > 0x1) ) {
      cnt >>= 1;
    } else {
      dir ^= 0x1;
    }
    return call LedsArray.setv(cnt);
  }
}
```

J.2 TestUartSendReceive

J.2.1 Makefile

```
COMPONENT=TestUartSendReceive
include ../Makerules
```

J.2.2 TestUartSendReceive.nc

```
/**
 * Based in part upon TestUartSimple.
 **/
configuration TestUartSendReceive {
}
implementation {
  components Main, HPLUARTC, TestUartSendReceiveM, LedC;

  Main.StdControl -> TestUartSendReceiveM;

  TestUartSendReceiveM.HPLUART -> HPLUARTC;
  TestUartSendReceiveM.Led -> LedC;
}
```

J.2.3 TestUartSendReceiveM.nc

```
// Note: Communication is at 9600 baud.
module TestUartSendReceiveM {
  provides {
    interface StdControl;
  }
  uses {
    interface Led;
    interface HPLUART;
  }
}
implementation {

  command result_t StdControl.init() {
    call HPLUART.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return SUCCESS;
  }

  async event result_t HPLUART.get(uint8_t data) {
    uint8_t response = data+1;
    call Led.Toggle();
    call HPLUART.put(response);
    return SUCCESS;
  }

  async event result_t HPLUART.putDone() {
    return SUCCESS;
  }
}
```

J.3 OscilloscopeJonaRF

J.3.1 Makefile

```
COMPONENT=OscilloscopeJonaRF
PFLAGS=-I../Oscilloscope
include ../Makerules
```

J.3.2 OscilloscopeJonaRF.nc

```
includes OscopeMsg;

/**
 * This configuration describes the Oscilloscope application,
 * a simple TinyOS app that periodically takes sensor readings
 * and sends a group of readings over the radio. The default
 * sensor used is the Photo component. This application uses
 * the AM_OSCOPEMSG AM handler.
 */
configuration OscilloscopeJonaRF { }
implementation
{
  components Main, OscilloscopeJonaRFM, TimerC, LedC, Accel,
           GenericComm as Comm, LedsArrayC;

  Main.StdControl -> OscilloscopeJonaRFM;
  Main.StdControl -> TimerC;

  OscilloscopeJonaRFM.Timer -> TimerC.Timer[unique("Timer")];
  OscilloscopeJonaRFM.Led -> LedC;
  OscilloscopeJonaRFM.SensorControl -> Accel;
  OscilloscopeJonaRFM.ADC -> Accel.AccelX;
  OscilloscopeJonaRFM.CommControl -> Comm;
  OscilloscopeJonaRFM.ResetCounterMsg -> Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeJonaRFM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
  OscilloscopeJonaRFM.LedsArray -> LedsArrayC;
}
```

J.3.3 OscilloscopeJonaRFM.nc

```
/*
 * Authors:   Jason Hill
 * History:   created 10/5/2001
 *
 */

/**
 * @author Jason Hill
 */

/**
 * @author Jamison Hope
 */

includes OscopeMsg;

/**
 * This module implements the OscilloscopeJonaRFM component, which
 * periodically takes sensor readings and sends a group of readings
 * over the UART. BUFFER_SIZE defines the number of readings sent
 * in a single packet. The Yellow LED is toggled whenever a new
 * packet is sent, and the red LED is turned on when the sensor
 * reading is above some constant value.
 */
module OscilloscopeJonaRFM
{
  provides interface StdControl;
  uses {
    interface Timer;
    interface Led;
    interface StdControl as SensorControl;
    interface ADC;
    interface StdControl as CommControl;
    interface SendMsg as DataMsg;
    interface ReceiveMsg as ResetCounterMsg;
    interface LedsArray;
  }
}
implementation
{
  uint8_t packetReadingNumber;
  uint16_t readingNumber;
  TOS_Msg msg[2];
  uint8_t currentMsg;

  /**
   * Used to initialize this component.
   */
  command result_t StdControl.init() {
    call Led.init();
    call Led.Off();
    call LedsArray.init();

    //turn on the sensors so that they can be read.
    call SensorControl.init();

    call CommControl.init();

    atomic {
      currentMsg = 0;
      packetReadingNumber = 0;
      readingNumber = 0;
    }

    dbg(DBG_BOOT, "OSCOPE initialized\n");
    return SUCCESS;
  }

  /**
   * Starts the SensorControl and CommControl components.
   * @return Always returns SUCCESS.
   */
  command result_t StdControl.start() {
    call SensorControl.start();
  call Timer.start(TIMER_REPEAT, 125);
  call CommControl.start();
  return SUCCESS;
}

/**
 * Stops the SensorControl and CommControl components.
 * @return Always returns SUCCESS.
 */
command result_t StdControl.stop() {
  call SensorControl.stop();
  call Timer.stop();
  call CommControl.stop();
  return SUCCESS;
}

task void dataTask() {
  struct OscopeMsg *pack;
  atomic {
    pack = (struct OscopeMsg *)msg[currentMsg].data;
    packetReadingNumber = 0;
    pack->lastSampleNumber = readingNumber;
  }

  pack->channel = 1;
  pack->sourceMoteID = TOS_LOCAL_ADDRESS;

  /* Try to send the packet. Note that this will return
   * failure immediately if the packet could not be queued for
   * transmission.
   */
  if (call DataMsg.send(TOS_BCAST_ADDR, sizeof(struct OscopeMsg),
                        &msg[currentMsg]))
    {
      atomic {
        currentMsg ^= 0x1;
      }
      call Led.Toggle();
    }
}

/**
 * Signalled when data is ready from the ADC. Stuffs the sensor
 * reading into the current packet, and sends off the packet when
 * BUFFER_SIZE readings have been taken.
 * @return Always returns SUCCESS.
 */
async event result_t ADC.dataReady(uint16_t data) {
  uint16_t ledsArray = 0;
  struct OscopeMsg *pack;
  atomic {
    pack = (struct OscopeMsg *)msg[currentMsg].data;
    pack->data[packetReadingNumber++] = data;
    readingNumber++;
```

```
      dbg(DBG_USR1, "data_event\n");
      if (packetReadingNumber == BUFFER_SIZE) {
        post dataTask();
      }
    }

    // Update the LedsArray display
    if( data < 349 ) {
      ledsArray = 0x001;
    } else if( data < 390 ) {
      ledsArray = 0x003;
    } else if( data < 431 ) {
      ledsArray = 0x007;
    } else if( data < 472 ) {
      ledsArray = 0x00F;
    } else if( data < 513 ) {
      ledsArray = 0x01F;
    } else if( data < 553 ) {
      ledsArray = 0x03F;
    } else if( data < 594 ) {
      ledsArray = 0x07F;
    } else if( data < 635 ) {
      ledsArray = 0x0FF;
    } else if( data < 676 ) {
      ledsArray = 0x1FF;
    } else {
      ledsArray = 0x3FF;
    }
    call LedsArray.setv( ledsArray );

    return SUCCESS;
  }

  /**
   * Signalled when the previous packet has been sent.
   * @return Always returns SUCCESS.
   */
  event result_t DataMsg.sendDone(TOS_MsgPtr sent, result_t success) {
    return SUCCESS;
  }

  /**
   * Signalled when the clock ticks.
   * @return The result of calling ADC.getData().
   */
  event result_t Timer.fired() {
    return call ADC.getData();
  }

  /**
   * Signalled when the reset message counter AM is received.
   * @return The free TOS_MsgPtr.
   */
  event TOS_MsgPtr ResetCounterMsg.receive(TOS_MsgPtr m) {
    atomic {
      readingNumber = 0;
    }
    return m;
  }
}
```

J.3.4 OscopeMsg.h

```
/*
 * Authors:             Nelson Lee
 * Date last modified:  6/27/02
 *
 */

/* Message types used by Oscope. */

/**
 * @author Nelson Lee
 */

enum {
  BUFFER_SIZE = 10
};

struct OscopeMsg
{
    uint16_t sourceMoteID;
    uint16_t lastSampleNumber;
    uint16_t channel;
    uint16_t data[BUFFER_SIZE];
};

struct OscopeResetMsg
{
    /* Empty payload! */
};

enum {
  AM_OSCOPEMSG = 10,
  AM_OSCOPERESETMSG = 32
};
```

J.4 HumiditySense

J.4.1 Makefile

```
COMPONENT=HumiditySense
```

```
PFLAGS=-I%T/lib/Counters
include ../Makerules
```

J.4.2 HumiditySense.nc

```
/* "I made this." - Jamison Hope */

/**
 * Configuration file for HumiditySense, an application
 * which senses relative humidity from a Humirel HS1101
 * whose supporting circuitry is connected to PC2.
 **/
configuration HumiditySense {
}
implementation {
  components Main, HumiditySenseM, TimerC, IntToLedsArray, LedC, IntToRfm;

  Main.StdControl -> HumiditySenseM;
  Main.StdControl -> TimerC;
  Main.StdControl -> IntToLedsArray;
  Main.StdControl -> IntToRfm;

  HumiditySenseM.Timer -> TimerC.Timer[unique("Timer")];
  HumiditySenseM.Led -> LedC;
  HumiditySenseM.IntOutput -> IntToRfm;
  HumiditySenseM.IntOutput -> IntToLedsArray;
}
```

J.4.3 HumiditySenseM.nc

```
/* "I made this." - Jamison Hope */

/**
 * Module file for HumiditySense.
 **/
module HumiditySenseM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Led;
    interface IntOutput;
  }
}

implementation {

  command result_t StdControl.init() {
    DDRC &= 0xFB;  /* set PORTC2 to be input */
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 500);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired() {
    uint16_t count;
    call Led.Toggle();
    loop_until_bit_is_clear(PINC, 2);
    count = 0;
    loop_until_bit_is_set(PINC, 2);
    do { ++count; } while( bit_is_set(PINC, 2) );

    /**************************************************************
     * Capacitance in pF = (count incs)*(0.339us/inc)/((621kOhm)*ln2)
     *                   = 0.78756 * count
     *
     *     (0.78756 * count) will be a number between 100 and 300
     *     so mult. by 78.756 will be between 10,000 and 30,000
     *     which is still 16 bits (this may reduce roundoff error
     *     since there's no floating point unit)
     **************************************************************/
    count *= 78.756; // now count represents cap in 10fF units

    /**************************************************************
     * Convert capacitance into %RH:
     * Uses a piecewise-linear approximation to the equation in the
     * HS1101 datasheet.
     *
     *  count < 18500: %RH*100 = 3*count - 48900
     *  count > 18500: %RH*100 = 2*count - 30400
     **************************************************************/
    if( count < 18500 ) {
        count *= 3;
        count -= 48900;
    } else {
        count *= 2;
        count -= 30400;
    }
    return call IntOutput.output(count);
  }

  event result_t IntOutput.outputComplete(result_t success) {
    return SUCCESS;
  }
}
```

Appendix K

Software Licenses

The following licenses apply to all software files listed in the preceding Appendices.

K.1 Berkeley License

K.2 Intel License

K.3 MIT License

Bibliography

[1] Analog Devices, Inc. ADXL311 ultracompact $\pm 2g$ dual-axis accelerometer data sheet. `http://www.analog.com/UploadedFiles/Data_Sheets/243920868ADXL311_B.pdf`. 119

[2] Analog Devices, Inc. ADXL311EB dual axis accelerometer evaluation board data sheet. `http://www.analog.com/UploadedFiles/Evaluation_Boards/Tools/304023032930274734167868ADXL311EB_a.pdf`. 14, 119, 120

[3] Ken Arnold, James Gosling, and David Holmes. *The Java^{TM}Programming Language, Third Edition.* Addison-Wesley, 2000. 61

[4] Atmel Corporation. AT45DB041B data sheet. `http://atmel.com/dyn/resources/prod_documents/doc3443.pdf`. 147

[5] Atmel Corporation. ATmega128 data sheet. `http://atmel.com/dyn/resources/prod_documents/doc2467.pdf`. 13, 51, 52, 137, 138, 174

[6] Atmel Corporation. AVR Instruction Set. `http://atmel.com/dyn/resources/prod_documents/doc0856.pdf`. 50, 139

[7] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leask. Piconet: Embedded mobile networking. *IEEE Personal Communications*, pages 8–15, October 1997. 24

[8] Bluetooth®. `http://www.bluetooth.com/`. 24

[9] John Catsoulis. *Designing Embedded Hardware.* O'Reilly & Associates, 2002. 56

[10] Anantha Chandrakasan, Fred S. Lee, et al. MIT μAMPS Project. `http://www-mtl.mit.edu/research/icsystems/uamps/`. 21

[11] Anantha Chandrakasan, Rex Min, Manish Bhardwaj, Seong-Hwan Cho, and Alice Wang. Power aware wireless microsensor systems. In *Keynote Paper ESSCIRC*, Florence, Italy, September 2002. 21, 22

[12] Chipcon AS. CC1000 data sheet. `http://www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf`. 146

[13] Crossbow Technology, Inc. `http://www.xbow.com/`. 18

[14] Crossbow Technology, Inc. MOTE-KIT4x00 MICA2 Classroom Kit. `http://www.xbow.com/Products/productsdetails.aspx?sid=93`. 63

[15] Cygwin. `http://www.cygwin.com/`. 79

[16] Dust Networks^{TM}. `http://www.dust-inc.com/`. 18

[17] Ember Corporation. `http://www.ember.com/`. 18

[18] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the Fifth Annual ACM International Conference on Mobile Computing and Networking*, 1999. 17

[19] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2003. 18, 79

[20] Steven D. Glaser. Some real-world applications of wireless wensor nodes. In *Proceedings, SPIE Symposium on Smart Structures & Materials/ NDE 2004*, San Diego, CA, March 14–18 2004. 19, 20, 27, 146

[21] Vadim Gutnik and Anantha P. Chandrakasan. Embedded power supply for low-power DSP. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 5, number 4, pages 425–435, December 1997. 21

[22] Jason Hill and David Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002. 18

[23] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, November 2000. 18, 79

[24] Gregory T. Huang. Casting the wireless sensor net. *Technology Review*, July/August 2003. 17

[25] Humirel, Inc. HS1100 / HS1101 relative humidity sensor data sheet. `http://www.humirel.com/product/fichier/HS1101-HS1100.pdf`. 14, 130, 133, 134, 136, 140

[26] Intel Corporation. Research - Research Areas - Sensor Nets / RFID. `http://www.intel.com/research/exploratory/wireless_sensors.htm`. 18

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall P T R, 1988. 50, 61, 84, 118

[28] Amir Ehsan Khandani, Jinane Abounadi, Eytan Modiano, and Lizhong Zheng. Cooperative routing in wireless networks. In *Allerton Conference on Communications, Control and Computing*, October 2003. 26

[29] Fred S. Lee. Analysis, design, and prototyping of a narrow-band radio for application in wireless sensor networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, May 2002. 21, 23

[30] Dr. Erik Lins. PROBOmega128 prototyping board. `http://lins.de/index.pl?page=PROBOmega128&lang=en`. 13, 70, 71, 137

[31] Dr. Erik Lins. PROBOmega128 prototyping board schematic. `http://www.chip45.com/en/downloads/probomega128v1.2bschematic.pdf`. 14, 161, 162

[32] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications*, 2002. 17, 19

[33] Millennial Net, Inc. `http://www.millennial.net/`. 18

[34] Millennial Net, Inc. Millennial Net opens new windows of opportunity for JELD-WEN. `http://www.millennialnet.com/pr/pressroom_fullstory.cfm?storyID=55`. 17

[35] Rex Min, Manish Bhardwaj, Seong-Hwan Cho, Amit Sinha, Eugene Shih, Alice Wang, and Anantha Chandrakasan. An architecture for a power-aware distributed microsensor node. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2002. 21

[36] ncc Manual Page. `http://www.tinyos.net/tinyos-1.x/doc/nesc/ncc.html`. 133

[37] Michael J. Neely and Eytan Modiano. Capacity and delay tradeoffs for ad-hoc mobile networks. IEEE Transactions on Information Theory, to appear, 2005. 25

[38] Kris Pister, Barbara Hohlt, Jaein Jeong, Lance Doherty, and J.P. Vainio. Ivy: A sensor network infrastructure for the University of California, Berkeley College of Engineering. `http://www-bsac.eecs.berkeley.edu/projects/ivy/`. 17

[39] Kris Pister, Joe Kahn, Bernhard Boser, and Steve Morris. SMART DUST: Autonomous sensing and communication in a cubic millimeter. `http://www-bsac.eecs.berkeley.edu/~pister/SmartDust/`. 17

[40] Joseph Robert Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California, Berkeley, 2003. 19

[41] RF Monolithics, Inc. DR3000-1 916.50 MHz Transceiver Module data sheet. http://www.rfm.com/products/data/dr3000-1.pdf. 73

[42] RF Monolithics, Inc. TR1000 916.50 MHz Hybrid Transceiver data sheet. http://www.rfm.com/products/data/tr1000.pdf. 69

[43] Paul Scherz. *Practical Electronics for Inventors.* McGraw-Hill, 2000. 29

[44] Eugene Shih, Seong-Hwan Cho, Nathan Ickes, Rex Min, Amit Sinha, Alice Wang, and Anantha Chandrakasan. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks. In *Proceedings of the Seventh Annual ACM International Conference on Mobile Computing and Networking*, pages 272–286, July 2001. 25

[45] Bjarne Stroustrup. *The C++ Programming Language, Special Edition.* Addison-Wesley, 2000. 61

[46] Texas Instruments, Inc. TLC555 LinCMOS^{TM} timer data sheet. http://focus.ti.com/lit/ds/symlink/tlc555.pdf. 133

[47] TinyOS Community Forum. http://www.tinyos.net/. 18

[48] TinyOS Naming Conventions. http://www.tinyos.net/tinyos-1.x/doc/tutorial/naming.html. 82

[49] TinyOS Tutorial. http://www.tinyos.net/tinyos-1.x/doc/tutorial/. 127, 141

[50] ZigBee^{TM}Alliance. http://www.zigbee.org/. 24

Index