A NEW WIRELESS SENSOR NODE DESIGN FOR PROGRAM ISOLATION AND POWER FLEXIBILITY

Adam W. Skelton

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2009

APPROVED:

Shengli Fu, Major Professor
Miguel Acevedo, Committee Member
Yan Huang, Committee Member
Xinrong Li, Committee Member
Murali Varanasi, Chair of the Department of Electrical Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Skelton, Adam W. A New Wireless Sensor Node Design for Program Isolation and Power Flexibility. Master of Science (Electrical Engineering), December 2009, 66 pp., 6 tables, 13 illustrations, references, 98 titles.

Over-the-air programming systems for wireless sensor networks have drawbacks that stem from fundamental limitations in the hardware used in current sensor nodes. Also, advances in technology make it feasible to use capacitors as the sole energy storage mechanism for sensor nodes using energy harvesting, but most current designs require additional electronics. These two considerations led to the design of a new sensor node. A microcontroller was chosen that meets the Popek and Goldberg virtualization requirements.

The hardware design for this new sensor node is presented, as well as a preliminary operating system. The prototypes are tested, and demonstrated to be sustainable with a capacitor and solar panel. The issue of capacitor leakage is considered and measured.

Copyright 2009

by

Adam W Skelton

ACKNOWLEDGEMENTS

I would like to thank Dr. Shengli Fu for his guidance, Dr. Murali Varanasi for providing support and creating the new graduate program, and the entire CRI research group at UNT.

CONTENTS

ACKNOWLEDGEMENTS iii

CHAPTER 1. INTRODUCTION AND MOTIVATIONS 1

1.1. Overview of Wireless Sensor Networking 1

1.2. Power 5

1.2.1. Energy Harvesting 8

1.2.2. Energy Storage 8

1.3. Reprogramming and Code Security 11

1.3.1. Analysis from First Principles 13

1.3.2. Hardware Requirements 17

1.4. Radio Transceiver 18

1.5. Summary of the Design Goals 19

CHAPTER 2. HARDWARE DESIGN 21

2.1. Selection of Major Components 21

2.1.1. Microcontroller 21

2.1.2. Power Electronics 23

2.1.3. Radio Transceiver 24

2.2. Schematics 26

2.3. PCB Layout 28

CHAPTER 3. SOFTWARE DESIGN 33

3.1. Kernel Structure 34

3.2. Memory Protection and Regions 36

3.3. Over-the-Air Programming 37
3.4. Application Program 40
CHAPTER 4. SYSTEM VERIFICATION AND CONCLUSIONS 43
4.1. Hardware Testing 43
4.2. Power and Range Measurements 44
4.3. DC-DC Converter Efficiency and Sleep Power Consumption 45
4.4. Converting the Sensor Readings 46
4.4.1. Light Sensor 46
4.4.2. Temperature Sensor 47
4.5. System Demonstration 47
4.6. Capacitor Leakage Measurements and Energy Management 48
4.7. Code Security and Reprogramming 53
4.8. Conclusions 53
APPENDIX: SCHEMATICS 55
BIBLIOGRAPHY 60

CHAPTER 1

INTRODUCTION AND MOTIVATIONS

The purpose of this thesis is to present the hardware and software design of a new sensor node to be used in the construction of wireless sensor networks. The first chapter gives an overview of the topic of wireless sensor networks along with some of the pertinent research, and the motivations for designing a new platform, culminating in the design goals for this thesis. Chapter 2 covers the creation of the hardware, detailing all the design decisions and component selections, through the pcb layout, to the manufacture of prototypes. To meet the design goals, it was necessary to create at least a minimal operating system to be used independently of the user's application. Chapter 3 presents a preliminary version of this operating system, which facilitates the network reprogramming and code security features. The last chapter is concerned with showing that the design goals have been met. I give some performance measurements for the hardware, present sensor data that shows the working system, demonstrate the functionality of the reprogramming system, and offer some directions for future research with the system.

1.1. Overview of Wireless Sensor Networking

A sensor node, also commonly called a mote, is a small electronic device that collects some form of data of interest from the surrounding environment and either stores them for later retrieval or transmits them to another device for processing or storage. If the data link used for communication is a wireless channel, then the node is called a *wireless* sensor node. Typically, this also implies that there is no wire supplying power, and thus the node must contain its own power source as well. If there are multiple nodes in the same area which can intercommunicate, then they form a *wireless sensor network*.

Large-scale sensors networks have been used for many years, mainly by governments, ranging from the global scale, such as the Global Seismographic Network [45], the regional SNOTEL snow depth measurement network [46], and many more. The system of surveillance satellites orbiting the planet could also be considered a wireless sensor network of heterogeneous nodes. In these cases, the size of each node ranges from a few pounds to several tons. At the other end of the size scale, deeply embedded sensors have also seen widespread use, and are now ubiquitous in automobiles, home appliances, and similar applications. For most of these systems, there is only one sensor of a specific type that is part of an independent unit, without any communication with the external world.

However, advances in miniaturization, electronics, battery, and transceiver technology have made practical the use of networks of small, autonomous sensors to cover a local area and form an intercommunicating, usually multihop, network which forwards the data back to a central location, and this is what is typically meant by a wireless sensor network[47]. Wireless sensor networks are an active area of research and are currently being used in a variety of applications. The ultimate end goal of such research is the ability to permeate the physical world with a multitude of invisible, low maintenance sensors which will provide a rich set of data to be used for the monitoring or manipulation of the environment[9].

One particular domain in which wireless sensor networks are seeing increased use is in environmental and habitat monitoring. Wireless sensor networks may have the potential to revolutionize the environmental sciences in a similar way that satellite remote sensing did several decades ago[32]. Several example environmental applications to which wireless sensor networks are currently being applied can be found in [36], such as monitoring the nesting patterns of storm petrels, a type of bird [30], monitoring the movement patterns of zebras in Kenya using Global Positioning System (GPS) receivers [22], implementing virtual fences for herding cattle [4], monitoring several parameters of ocean water [48], and monitoring vineyards to improve plant care and harvesting [3].

The possible military applications of sensor networks are also tremendous, primarily in the area of battlefield monitoring, but the hardware is not yet to the point where many of them can be realized. But so far, wireless sensor networks have been used by the military for such things as tracking vehicles[49] and locating snipers [38]. In fact, one of the earliest sensor network initiatives was the Smart Dust project, located at UC Berkeley and funded by DARPA's MEMS program [23][17]. Their ambitious goal was to develop a fully functional node with a volume of only one cubic millimeter. Although this goal has still yet to be achieved, their efforts did spur the development of some larger sensor motes which have become somewhat of a standard, namely the Mica family of motes. The design presented here continues in their tradition of using off-the-shelf components to make custom boards [50]. Later on at Berkeley came the creation of the TinyOS operating system[15], which has grown tremendously and is now in widespread use in the wireless sensor network community.

There are several more domains that are rich with possible applications for wireless sensing. The medical industry is already using wireless sensors to monitor vital signs in hospitals [2], and one can envision sensors being used for constant monitoring of vitals and other health parameters in the elderly. But the uses need not be restricted to the elderly or sick. Once the sensors have matured enough, they could be used for constant monitoring of such parameters as heart rate, blood oxygen levels, and acceleration, providing valuable information to athletes or helping people get the recommended levels of cardiovascular activity. Wireless sensors can also be used for building and home automation, such as to increase energy efficiency by automatically turning off lights when a room is empty, or for real-time monitoring of power consumption.

The definition of a wireless sensor network given above covers a broad array of possible physical devices. For example, a network of three brick-sized sensors in a single room and a network of ten thousand dust-sized sensors covering a square mile would both constitute a wireless sensor network, although they will involve vastly different design requirements. For this reason, it is useful to further classify wireless sensor networks along several dimensions,

such as size, mobility, cost, energy requirements, and network topology in order to define a design space. One such scheme classifies sensor networks along fourteen dimensions[36]. According to their classification scheme, the design presented here would be an immobile, matchbox-sized, homogeneous, radio frequency, fully connected sensor network, with the other parameters, such as infrastructure, network topology, network size, and system lifetime being determined by the user. Another classification scheme characterizes the space and time usage of the system along the three dimensions of scale, variability, and autonomy[9].

Although the location of a particular application in the design space will determine their priority, the general design goals of wireless sensor nodes are typically small size, low power consumption, decent transmission range, accurate sensing, low maintenance, and reliability. Of course, there will be tradeoffs between these various goals, and not all of them will be achievable in a single design. The ZebraNet project mentioned earlier is an excellent example of some of the design issues that must be confronted when designing a wireless sensor network[22].

A block diagram of a typical sensor node is given in Figure 1.1. They usually consist of a microprocessor with at least a built-in analog to digital converter to sample the sensors. The microprocessor then interfaces with the data transmission system, which may also be on the same die as the microprocessor, although a separate module is usually used. Radio links are the most common, although optical links can be used as well[23]. The other necessary component is a power system.

There are already several existing hardware platforms for wireless sensor networking[14], and a summary of their features is given in Table 1.1. The most popular is the Mica family of motes [16], originally developed at the University of California, Berkeley, and now including the Mica, Mica-2, MicaZ, and Iris motes. This design is similar in several ways, and is intended to be an improved offering in this region of the design space, the "matchbox" sized sensor mote. Table 1.1 includes several columns that will be germane to the following

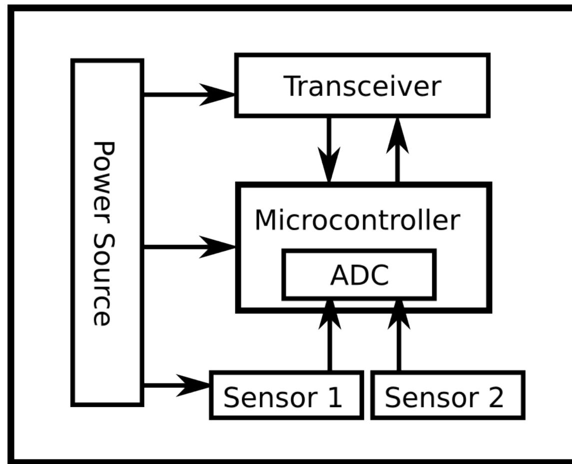


Figure 1.1. Sensor Node Block Diagram

discussions, in particular, the memory protection, operating modes, and boost converter fields. My design is included for comparison.

As you can see, there are already many offerings for the sensor network architect. In the following sections, I discuss my motivations for creating another design.

1.2. Power

Providing the energy needs of wireless sensors is a challenging design problem. To illustrate the severity of the problem, suppose you have a mote that draws 33mA when active (typical for the Mica series when the radio is active) powered by two alkaline AA batteries in series, each with a capacity of 1000mAh, for a power of about 100mW. The combined capacity is still 1000mAh, and dividing gives a lifetime of 33 hours, or about a day and a half. But the target lifetime for each mote, depending on the application, will typically be on the order of several months to years. The design problem is how to bridge this gap.

Just as in economics, there are two sides to consider, the supply and demand. The designer can attempt to make more energy available, or to reduce its consumption. We will consider the demand first.

For many sensing applications, the required sampling rate to capture all the activity is low. A temperature sensor, for example, probably only needs to be sampled every few

TABLE 1.1. Comparison of Sensor Mote Hardware

Name	Boost Converter	Microcontroller	Max Clock (MHz)	Word Size (bits)	Flash Size (KB)	RAM Size (KB)	Memory Protection	Execution Modes	USB	Radio Chip	802.15.4 Compliant
This Design	Yes	AT32UC3B	60	32	256	32	Yes	Yes	Yes	CC2520	Yes
Mica[16]	Yes	ATmega103L	4	8	640[1]	4	No	No	No[2]	TR1000	No
Mica2[51]	No	ATmega128L	8	8	640[1]	4	No	No	No[2]	CC1000	No
Mica2Dot[52]	No	ATmega128L	8	8	640[1]	4	No	No	No[2]	CC1000	No
MicaZ[53]	No	ATmega128L	8	8	640[1]	4	No	No	No[2]	CC2420	Yes
Iris[54]	No	ATmega1281	8	8	640[1]	8	No	No	No[2]	AT86RF230	Yes
Telos[55]	No	MSP430	8	16	1072[1]	10	No	No	Yes	CC2420	Yes
Firefly[31]	No	ATmega32L	8	8	32	2	No	No	No	CC2420	Yes
BTnode[56]	Yes	ATmega128L	8	8	128	244	No	No	No	CC1000	No
Eyes[42]	No	MSP430F149	5	16	60	2	No	No	No	TR1001	No
IMote2[57]	No	PXA271	416	32	32768	32768	Yes	Yes	Yes	CC2420	Yes
muPart[58]	No	12F675	5	8	1.4	1/16	No	No	No	12F675	No
Shimmer[59]	No	MSP430	8	16	48	10	No	No	No[2]	CC2420	Yes
Snow5[60]	No	MSP430	8	16	48	10	No	No	No[2]	CC1100	No
SunSpot[61]	No	ARM920T	180	32	4096	512	Yes	Yes	Yes	CC2420	Yes
TinyNode[62]	No	MSP430	8	16	560[1]	10	No	No	No	XE1205	No

[a] Includes internal program memory and an external flash

[b] Available with extra interface board

minutes. In these situations, the node can safely spend most of its time in a low-power sleep mode, only awakening at regular intervals to collect and transmit data. When the mote makes regular transitions between these two states, the average power consumption can be modelled by the following equation:

$$P_{avg} = P_{active} * d + P_{sleep} * (1 - d) \tag{1}$$

where d is the duty cycle, expressed as the percentage of the time in each cycle spent awake and active, P_{active} is the average power consumption when the mote is doing work, and P_{sleep} is the average power consumption when the mote is sleeping. For a typical sensor mote, P_{sleep} is less than 1mW, and may be just tens of μW. Thus the difference in power spans two to three orders of magnitude. If d is very low, then the average power will be dominated by P_{sleep}, and the system lifetime will go up accordingly. In fact, if d is sufficiently low, then P_{active} becomes largely irrelevant. To continue with our example, if the mote uses 0.5mW while sleeping and operates with a duty cycle of 1%, the system lifetime goes up to about 85 days.

Having each node spend most of its time asleep introduces other system design challenges, usually revolving around the problem of communication. No existing mote has the ability to be woken up from a deep sleep by radio activity, since sampling the channel requires significant energy. Thus, in order for the network to be operational, especially if it is a multihop network, the nodes must synchronize their activities. In fact, much work has been done on ways to synchronize the nodes, so that the time spent active is minimized[40][8][37].

The other side of the energy coin is the supply. There are two approaches that can be taken. The first is to frontload the mote at deployment with all the energy it will need for its intended lifetime. This is simple to do from an energy accounting standpoint. Simply calculate the total energy needed over the target lifetime. Then choose an energy storage technology, and find its energy density. Divide the two to get the required size of storage. There are several available energy storage options, but, for good reasons, primary

cell batteries are the method of choice in most electronics applications. A table of energy densities for a wide range of technologies, including most common battery chemistries and ultracapacitors, can be found here[63]. While batteries are far from having the highest energy density, the other options are typically too difficult, expensive, or dangerous to implement.

If this process will not yield a workable solution due to size or cost constraints, then the second approach must be taken, which is to use energy harvesting. If the system goal is perpetual operation, then there is no other choice.

1.2.1. Energy Harvesting

Energy harvesting is the extraction of ambient energy from the surrounding environment to power a device, which is also an active area of research[25][43]. If the amount of energy available in the environment is sufficient, then it may be more economical to harvest this energy than store all the future needs at the time of deployment.

The types of ambient energy available for harvesting are electromagnetic radiation (usually visible light from the sun, but not the only source), kinetic energy (vibration), and thermal gradients[34]. The application environment will determine which of these is the most feasible, but for most wireless sensor network designs, solar radiation is probably the most attractive. Solar harvesting has already been incorporated into several sensor network designs and has been proven to be effective for allowing perpetual operation[21]. For example, the Heliomote design[29] incorporates a connection for solar charging, a battery monitor for collecting harvesting information, a shunting circuit to prevent overcharging of the battery, and a DC-DC converter to provide a steady supply, but it is an expansion board, and adds significantly to the node size and cost. Thus another design goal of my system is that it should allow easy use of solar recharging, without requiring any expansion boards.

1.2.2. Energy Storage

The problem with energy harvesting is the same problem that plagues efforts to replace fossil fuels with solar and wind energy: availability. Because all forms of energy harvesting are unconstant and unpredictable, the device still needs an energy storage mechanism, whose

purpose will be to smooth out the variations in the power supply, effectively acting as a low-pass filter. But in this case, the role of the energy storage is fundamentally different than the scenario above. Instead of being dictated by the target lifetime of the device, the requirements for the energy storage will be determined by the characteristics of the ambient source, and how much smoothing it requires to keep the device alive. A device using solar power may only need to store enough energy to last a few days without sun. While there are several theoretical options (fuel cells come to mind), the most practical up until recently has always been to use rechargeable (also called secondary) batteries, which are based on a reversible electrochemical reaction.

1.2.2.1. Disadvantages of Batteries

Despite their popularity, batteries have several disadvantages which can hamper their use for long-term sensor deployments. The chief problem is the rate at which rechargeable batteries lose their capacity due to internal chemical and structural changes. The stated rate at which the batteries degrade varies, but a typical value for NiMH rechargeable batteries is that they are specified to maintain 80% of their capacity after 500 charge-discharge cycles[64]. This, however, is under test conditions, with a controlled temperature and charge-discharge profile. In practice, the rate at which rechargeable batteries lose their capacity is highly dependent on the temperatures and charge-discharge profiles to which the battery is subjected[64][65]. A low-power sensor node using a rechargeable battery to store solar energy may not have the liberty of controlling the charge profile, as this requires significant electronics. Also, fully draining a cell can cause polarity reversal which usually severely harms the cell's capacity. Exposure to high temperatures can cause venting from the cell, which is irreversible. A battery subjected to any of the above will fall short of the stated lifetime. In short, they are very sensitive to abuse. Also, depending on the conditions, the charging efficiency may be less than stellar[66]. Last, batteries raise the issue of toxic chemicals and disposal. For these reasons, more energy is being focused on new capacitor technologies as possible battery replacements.

1.2.2.2. Ultracapacitors

Traditionally, the energy density of capacitors has always been too low for them to be practical as a primary energy storage mechanism. Recently, the development of the electric double-layer capacitor[67] has added one to two orders of magnitude to the highest energy density availaable, allowing capacitors to close the gap somewhat between batteries. Thus, while still having an energy density much lower than the best batteries, it is now viable to use capacitors as the primary energy storage device when harvesting is available. Capacitors have several advantages over batteries. First, they are more resilient to temperature extremes and have longer lifetimes, being able to withstand hundreds of thousands or millions of charge-discharge cycles while retaining their capacity[68]. They are also indifferent to the way in which they are charged and discharged, provided the voltage does not reach high enough to cause breakdown. Fully discharging a capacitor will have no effect on its capacity. Also, there is a much simpler relationship between the voltage of a capacitor and its stored energy, as compared to batteries which have a complicated voltage-discharge curve[64]. This makes estimating the charge easier, without using a complicated Coulomb counter as is used for accurate battery measurements. The energy stored by a capacitor can be calculated from the voltage using the simple equation:

$$E_C = \frac{1}{2} C {V_C}^2 \tag{2}$$

Also, capacitors have excellent charge efficiency, since they work by merely storing free charges. They also have longer shelf lives and prevent fewer disposal issues.

There is another issue which both batteries and ultracapacitors face, but which can be more prominent in capacitors. They both self-discharge, gradually losing their energy even under no load. Unfortunately, the rate at which ultracapacitors leak is dependent on the voltage, and grows significantly as the capacitor approaches the rated voltage. Recently, some work has been done to add awareness of this phenomenon to wireless sensor networks[44]. Their idea is that under higher charge conditions, the node should actually use *more* energy,

increasing its total active time, since the energy was going to leak away regardless. In chapter 4 I will provide some leakage data which illustrates these concerns.

These consideration lead to my first set of design goals. To facilitate experimentation with capacitors as well as allowing batteries to be used, the mote should be able to run from either with no modifications or expansion boards required. Furthermore, it should operate throughout as much of the voltage range of the input as possible. While batteries should not be fully discharged, capacitors have useable energy all the way down to 0V. It should include connections for a solar panel and a charging circuit. There should also be a mechanism by which the microcontroller can shut down the solar charging, if it is necessary to monitor for overcharging. Also, the input and solar connections should be routed to the ADC, so that energy information is available at all times.

1.3. Reprogramming and Code Security

For a sensor network which may be spread over a wide area or placed in an inaccessible location, it is obviously a useful feature to be able to alter the program on each node without retrieving the device. Even if physical access to the network is available, it is usually quicker and more convenient to reprogram the nodes using the radio if possible, instead of physically connecting each device to a pc or programmer. In other words, the time taken to reprogram each node manually is $O(n)$, whereas using a network dissemination protocol requires $O(1)$ time for the user. Several protocols have been implemented to provide this network or "over-the-air" programming efficiently and robustly. The one most widely used at the moment is the Deluge protocol[18][13] based on the Trickle algorithm[28], but there are many others. There is XNP[69], a single-hop protocol in use since TinyOS 1.0, which has been extended to use the Rsync protocol for reduced data transmission[20]. Another incremental scheme was created using the EYES node[35]. MOAP (Multihop Over-the-Air Programming)[41] was the first protocol to use multi-hop routing to disseminate the programs, specifically targeted for the Mica2 motes. There are other systems as well[26][1][5].

Instead of pursuing that line of research further, I have concentrated on another problem somewhat indirectly related to the issue of over-the-air reprogramming. Namely, what if it is desirable to be able to execute *untrusted* or *arbitrary* code natively on the nodes, such as code submitted by a third party, and still guarantee that the reprogramming system will work, so that the node can be recovered. This question came to the fore as part of our efforts at UNT to provide a system whereby users can submit code through the web to be programmed onto a sensor network for experimentation[5].

To see where the difficulties arise, let us look at how the Deluge programming system works. The other systems are similar, their main differences being how they get the new program to the mote. Our concern here is what happens with the program before it is sent and after it is received by the target node. To begin with, the author of the new program must include and link in the Deluge subsystem, since TinyOS applications are monolithic. Then, when it is to be programmed onto the motes, the main application of each mote will yield control to the Deluge subsystem, which will then download and construct the binary according to its protocol, and store the binary in external flash memory (As far as I know, all the motes which work with the Deluge system are from the set above in Table 1.1 which include a flash module external to the microcontroller). Then Deluge will reset the system, and the bootloader will load the new program from the external flash into the microcontroller's internal program memory and execute it.

Here we see where using untrusted code can break the system. Assuming that the Deluge subsystem is included in the new program, all the application has to do is disable it and all reprogramming functionality will be cut off. This is verified in the Deluge manual[70].

This is of course a non issue if the program is from a trusted source. But if it's not, then the program can easily isolate itself. Also, Deluge requires a "Golden Image" for system recovery, which is supposed to be stored in an area of memory that the application cannot write[70]. This is, however, not the case for most of the hardware platforms for which Deluge is used. It should be noted that there have been attempts to secure the Deluge

system[7][19][24], but they are focused on the problem of making sure the corrent application makes it to each destination node and is unaffected by denial of service attacks or erroneous code injection attacks from third parties outside the network or any compromised nodes within the network, and do not speak to the issue of how an internal node may have been compromised in the first place, or how it can be recovered.

All the other over-the-air programming systems will suffer from the same issues, because this is fundamentally an architectural problem, and not a software design problem. The problem stems from the fact that with the microcontrollers typically used for these motes (see Figure 1.1), the user application is in almost complete control of the system. This will be discussed in more detail shortly.

One interesting approach to the code security and reprogramming problem is to use a small virtual machine, which acts as an interpreter for the application program, instead of letting it execute directly on the hardware. Such a virtual machine for sensor networks has already been created[27]. Maté is a secure system in the sense that if the virtual machine is designed correctly, the "machine" environment created for the user application will not be capable of disrupting the reprogramming system. But there are drawbacks to using an interpreter. The first, of course, is efficiency. Whenever the software must decode, check the validity, and execute each instruction instead of natively on the hardware, performance will suffer. The energy cost, is mitigated, however, by the fact that the code sizes are smaller, requiring less energy for reprogramming. Also, using a virtual machine introduces restrictions on the development system. The compiler, if a compiler is used, must target the virtual machine instead of the microcontroller itself, and the programmer is limited to whatever constructs are provided by the virtual machine, independent of the capabilities of the underlying processor.

1.3.1. Analysis from First Principles

To see what is required for a truly robust over-the-air programming system, let us do a theoretical analysis of the problem. What we want is a machine that is conceptually

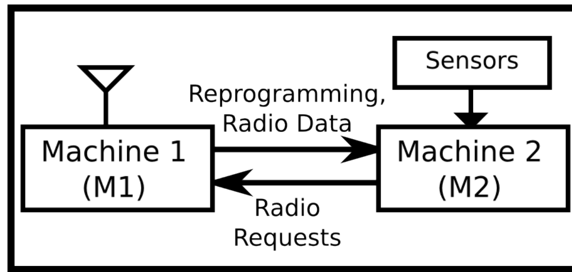


Figure 1.2. Desired Structure

composed of two parts, or sub-machines, as in Figure 1.2. Machine M1 is responsible for the reprogramming of the system, and machine M2 performs the desired functionality. Since the radio is necessary for the programming system, we may assign it to M1, and M2 may gain access to it by submitting requests to M1. M1 will respond to these requests with data, or it may reprogram machine M2 if a request to do so comes over the radio. Machine M2 is the main machine of the system, performing its desired functionality, such as reading the sensors or acting as a network hub or data aggregate. M2 only needs to communicate with M1 in order to access the radio. And what is most critical is that M2 not be able to disrupt the functionality of M1 and impair the reprogrammability of the system, even if it was written with malicious intent. Thus any inputs coming from M2 must be carefully checked for their effects.

Now this is a conceptual structure, and there are several ways that it could be incarnated. For example, two physically separate machines could be used, and the communication lines in Figure 1.2 could be physical wires or pcb traces connecting the two machines. In this case, M1 could consist of a dedicated radio transceiver chip and a microcontroller that drives whatever pins are necessary to program M2. And M2 could consist of another microcontroller that reads the sensors and processes the data, or whatever the intended funcationality of the device may be. This is, in fact, the way that the original Mica mote was implemented[16].

But what if it is desired that the entire system contain only one microcontroller, for space, power, and cost reasons. To simplify our inquiries further, suppose that the microcontroller is a general-purpose stored-program Von-Neumann architecture. We may also suppose that,

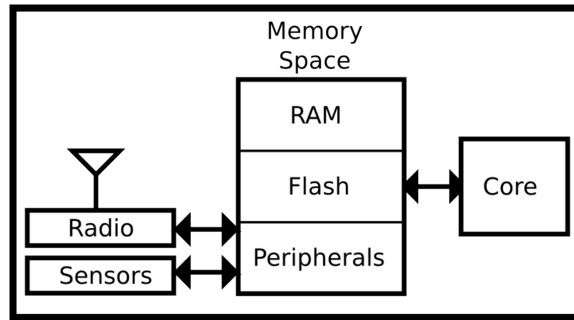


Figure 1.3. Single Processor Machine Diagram

since we are concerned with small embedded devices, that the memory space consists of a region for volatile storage (RAM), a region for non-volatile program and data storage (Flash), and another region which holds control registers for the peripheral systems, and the CPU executes the typical cycle of fetching, decoding, and executing instructions from the memory space. This situation is depicted in Figure 1.3.

What, then, are the necessary conditions for this machine to be functionally equivalent to Figure 1.2? The first thing we notice is that both M1 and M2 will require parts of all three areas in the memory space to perform their duties. Machine M1 will need some code stored in the Flash section to operate the radio and programming system, some RAM to store the state, and access to whatever peripheral modules communicate with the radio. Likewise, M2 will need all three areas, as it represents the main functionality of the system, and must communicate with the sensors and any other peripherals.

It should be noted that it is easy to construct a specific program for a given general-purpose machine that performs as in Figure 1.2, for a specific M2. Just write the portion pertaining to M2 so that it is "nice" and cooperates with M1. But we want a system that will work for *any* programming of M2, even one written with malicious intent.

It seems necessary, then, that the CPU must contain some mechanism by which the instructions belonging to M2 are not allowed to interfere with the memory areas belonging to M1, except to request services provided by M1. And M1 should not interfere with M2, except to provide requested data, or to reprogram it altogether if a request to do so comes

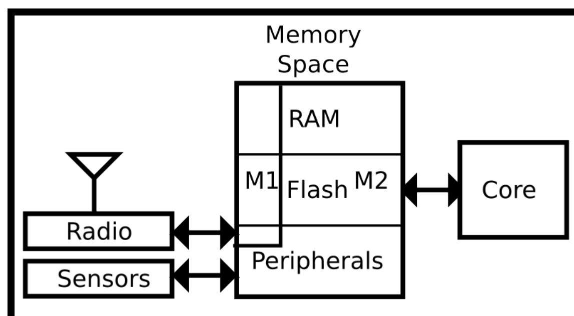


Figure 1.4. Memory Partitioning

over the radio, as shown inn Figure 1.4. Also, since it is presumed that the CPU can only execute a single stream of instructions, there must be a method of sharing the execution time between the two machines.

Now this state of affairs is not in any way new. Rather, what I have just described is an instance of the common partitioning of a computer system into an operating system and an application program. In this case, M1 is the operating system, because it is more fundamental, in the sense that it is responsible for recreating M2 if requested, and needs no services from M2 for its operation. And M2 is the application program, because it is untrusted and requires services from M1.

It is hard to say what is absolutely necessary to build such a system, but any system that fits the requirements put forth in [33] should be sufficient. This paper describes the conditions necessary for a machine to be "virtualizable", which means that it is capable of hosting an application program natively, where the application program is "jailed" because it can only use resources (eg. memory, peripherals, execution time) that have been allocated to it by a monitor program with higher privileges. According to this paper, an architecture is virtualizable if it is a "third generation architecture", and its instruction set meets some restrictions. A third generation architecture is defined by its having two modes of operation, supervisor and user, and a memory protection mechanism. In the paper, their memory system is relocation-based, but this is not necessary. This is an important point, for implementing a full virtual memory system is outside the energy budget for most microcontroller

applications. But all that is required is memory protection, which only necessitates determining the region in which each memory access falls. For the instruction set restrictions, their theorem states, put simply, that all the sensitive instruction should only be allowed to execute in supervisor mode.

1.3.2. Hardware Requirements

In conclusion, to support a robust over-the-air programming system with a single processor, it is sufficient to have execution modes, memory protection, and a well-designed instruction set.

Now the 8- and 16-bit microcontrollers used in current designs do usually support a rudimentary form of memory protection. These microprocessors typically have a dedicated portion of the address space, mapped to the flash memory, that is designated as the "bootloader". Any instructions executing from outside the bootloader that attempt to modify program memory are not allowed to do so. This does provide some protection for the flash memory, but not for the RAM or peripherals. Thus a rogue application may do whatever it pleases to the system, except for rewriting the program memory. This is clearly insufficient for providing a reprogramming system that requires the peripherals to work.

Examining other research shows that the lack of memory protection is a cause for many ills in embedded systems[6][10][39]. There have been attempts to circumvent the lack of these hardware features to provide code security. The t-kernel[12] uses load-time code modification to scan the program for potential problems. Deputy[6] is an attempt to provide memory safety by using a source-to-source compiler as part of the compilation pipeline. This would be inappropriate if the goal is to be able to run arbitrary code. All of these systems are rendered unnecessary with appropriate hardware support.

Now the reason most microcontrollers have no type of memory management unit (MMU) is that the extra cost in terms of die area and power consumption is prohibitive for low-power designs, and the functionality is not necessary for many applications. Microcontrollers simply

occupy another region of the design space compared to modern full-featured microprocessors. But, as mentioned above, only the memory protection subset of the traditional MMU functionality is necessary to meet our requirements. There is no need for any address translation or paging. But with advances in semiconductor manufacturing, the power costs of incorporating memory protection have been reduced, and many 32-bit microcontrollers now support this feature, as will be seen in the next chapter.

Designing the system with memory protection and an operating system/application boundary has other advantages as well. It makes debugging easier, because the CPU will catch many memory errors. Also, the application program and "operating system" no longer need to be bundled together. This reduces the energy cost of reprogramming the system, because the operating system portion does not need to be retransmitted. Another advantage of decoupling the two is that, since the application now runs inside a "sandbox", but still natively on the microcontroller, the application programmer may use any language he wishes, so long as a compiler is available that targets the microcontroller, and a system call library is available. Lastly, since the application is not "trusted", it is safer and easier to experiment with new ideas.

1.4. Radio Transceiver

Although the rate at which the electronics are capable of taking sensor readings is quite high, for most sensing applications a much lower sampling rate is necessary. For example, a temperature sensor connected to a 10-bit ADC and generating a sample every 10 seconds has a bit rate of only 1 bit/s. This means that the data rate required for the radio subsystem is modest. Rather, the radio system should be designed with energy efficiency and simplicity in mind. To this end, IEEE has developed the 802.15.4 specification[71], which offers physical (PHY) layer and media access control (MAC) layer standards for low power, low data rate wireless networks. The ZigBee®[72] suite of higher-level protocols is built on top of the 802.15.4 specification, so any device that implements 802.15.4 is also ZigBee compatible,

provided that the software implements the upper layers. About half of the existing motes are already 802.15.4 compliant, and I will continue in that vein.

1.5. Summary of the Design Goals

So far, no mention has been made of the sensors themselves. There are a large variety of possible sensors that could be used, and which ones are needed varies with the application. The approach taken with the Mica family is to use expansion boards that plug into a 51 pin connector and provide extra sensors. Since this is a prototype design meant to test other features, and the connector used by the Micas is rather large, I decided to forego using a connector and incorporate a light and temperature sensor directly onto the board, and to provide access to the other ADC pins through solder connections on the edge of the board. A future version could have a connector with daughterboards, or incorporate more sensors into the main board.

Cost is another primary objective. The SunSpot design provides most of the features mentioned here. However, they are prohibitively expensive for a large deployment. The Mica family combined with the Heliomote board[29] provides a platform for using capacitors and energy harvesting, but the total cost is several hundred dollars per node. My goal here is to provide a sensor node with a total cost of less than $40 per unit.

Since this is intended for the matchbox-sized design point, I chose a size constraint of $25cm^2$.

To summarize, these were the design goals for the new system:

(i) The microprocessor must meet the Popek and Goldberg virtualization requirements, to enable application code to be completely contained while running natively, facilitating a more robust over-the-air programming system and enhancing system stability.

(ii) The hardware should run stably and consistently over a wide range of input voltages, ideally from 0V to the system voltage, so that either batteries or capacitors in any configuration can be used.

(iii) There should be integrated charging circuitry, as well as a mechanism by which the microcontroller can disable further charging to prevent overcharging the storage.

(iv) The power consumption in a deep sleep should be less than 1mW.

(v) The radio transceiver should be IEEE 802.15.4 compliant, and have a line-of-sight transmission range of at least 100 meters.

(vi) The total size should not exceed $25cm^2$.

(vii) The total cost should not exceed $40.

The rest of this thesis documents my attempt to implement these goals.

CHAPTER 2

HARDWARE DESIGN

This chapter covers the process by which I designed and manufactured the hardware prototypes. Chapter four will cover system testing and verification. To design the hardware, first I selected the major components, then drew the system schematics, followed by the pcb layout. This I sent to a pcb manufacturer, along with the parts, who manufactured and assembled the boards. There were several design decisions to be made at each step, and I'll examine each of these in turn.

2.1. Selection of Major Components

2.1.1. Microcontroller

The markets and applications for microcontrollers are wide and diverse, and thus there are many selections from which to choose. It is impractical to investigate every microcontroller on the market, so, to narrow my search, I selected five of the most popular microcontroller companies and examined their offerings: Texas Instruments, STMicroelectronics, Microchip Technologies, Freescale Semiconductor, and Atmel.

Texas Instruments makes three families of microcontroller, the MSP430™, C2000™, and ARM®Cortex™-M3 based families. The MSP430 family, which, as can be seen by Table 1.1, is a popular choice. It is 16 bits wide, and has a small section of the flash that can be locked, but the rest is programmable by the application, and there is no RAM protection [73]. The C2000 is a 32-bit family, but it is not possible to program the flash from code running from the flash, and there is no protection for the RAM [74]. Since several companies make microcontrollers based on the ARM Cortex-M3 core, I will discuss it separately.

STMicroelectronics makes the STM8L, ST6, ST7, ST10, and STM32 families. The STM8L has a complicated flash protection system, but no RAM protection or operating

modes [75]. The ST6 cannot be reprogrammed by an onboard application, and offers no RAM protection or execution modes [76]. The ST7 family includes the "ICP" programming system, but it also cannot run from internal code [77]. In the ST10, the flash has three operating modes, User, Bootstrap (for programming), and Test, but mode transitions can only be made at system reset by controlling specific pins, and there is no RAM protection [78]. The STM32 family uses the ARM Cortex-M3 core.

Microchip Technology makes the famous PIC®line of microcontrollers. For all their 8- and 16-bit microcontrollers, they have no general memory protection and it seems that the chips can only be reprogrammed externally [79][80]. The 32-bit PIC32™family is more promising. It has two modes of operation, kernel and user, and has the "RTSP" system, which stands for "Run-time self programming". It also provided memory protection, both for the flash and for the RAM [81]. Thus it would be suitable for my design and will be considered further.

Freescale Semiconductors, formerly Motorola, makes a full complement of 8, 16, and 32 bit microcontrollers. The 8 and 16 bit lines lack the ability for self-programming and have no memory protection[82][83]. However, their 32 bit microcontrollers are based on the historic M68K architecture, which has full operating system support through execution modes, privilege levels, and virtual memory, and will be considered further[84].

Atmel makes an 8-bit line, the traditional AVR®, and a 32-bit line, called the AVR32. As seen from Table 1.1, the AVR family is also popular and is used in many of the sensor node designs. The AVR family has the ability to reprogram itself when executing from a dedicated bootloader section, and otherwise, no writes may be made to the Flash. But the RAM and peripherals are unprotected, and there are no execution modes[85]. The AVR32[86] is a relatively new design, launched in 2006. It is a RISC load/store architecture, with either a memory protection unit or a full memory management unit, and several execution levels, making it suitable for this design. For microcontroller designs, Atmel provides the UC3 implementation, which is what is considered here.

Therefore, my choice was between one of the many ARM Cortex-M3 based designs, the PIC32 family, the M68K family, or the AVR32 family. After examining their operating modes and instruction sets, it seems that all of them meet the virtualization requirements. First I ruled out the PIC32. Its design requires some memory address translation, and the memory protection areas are set and inflexible. I chose not to use one of the M68K families because it is based on a CISC background, whereas I wanted a simpler RISC load/store design. The Cortex-M3 and AVR32 UC3 architectures have many similarities. They are both Harvard load/store architectures with a three-stage pipeline and a 16x32-bit register file that includes the program counter, link register, and stack pointer. They both have variable-length instructions that are either 16- or 32-bits, for improved code density. And they both include a Memory Protection Unit (MPU) with eight configurable regions of protection, and a similar set of included peripherals. At this point, it's probably somewhat of a subjective decision, but I chose the AVR32 UC3 architecture for its simplicity, clean design, performance, and excellent documentation. To be more specific, I chose the AT32UC3B0256[87], which holds 256KB of onboard Flash, and 32KB of RAM. The integrated peripherals include all the microcontroller standards, including SPI, TWI, and USARTs. It also has a USB controller.

2.1.2. Power Electronics

The microcontroller and radio chip that I chose both require a stable voltage rail of 3.3V, and include their own regulator to provide 1.8V to their digital cores[87][91]. Since the input voltage is expected to vary over a wide range, it was clear that using a direct connection was out of the question, and some power electronics were needed. I decided to use a step-up or boost DC-DC converter, and chose the MAX1676 from Maxim[88]. It is a high-efficiency DC-DC converter with an onboard N-channel MOSFET synchronous rectifier with a stated resistance of 0.3Ω. The quiescent current is on the order of 20μA, and efficiences exceed 80% under most conditions. It only requires a handful of external components, the most significant being the inductor and output filter capacitor. The efficiency of a DC-DC converter is highly dependent on the losses in the energy storage element[88], which in this

case is an inductor. To maximize efficiency, I chose a very low loss inductor, with a DC resistance of only $28m\Omega$. In hindsight, this inductor was probably a little too large, as it dominates the size of the converter circuit, and is a significant part of the mass of the board. It might have been better to trade off some efficiency for a reduced size. Because a boost converter works by rapid switching, it introduces ripple into the output which must be minimized with a filter capacitor. The magnitude of the ripple is, in fact, not usually determined by the capacitance of the filter, but it's equivalent series resistance (ESR). I used a tantalum capacitor with an ESR of $25m\Omega$. While it is effective at reducing the noise on the power rail, it is also somewhat large, and a smaller capacitor might need to be used for a system with more aggressive size constraints.

Since a boost converter is needed to use a capacitor throughout its entire range while providing a constant output, I included the boost converter column in Table 1.1, to see which existing platforms could potentially run from a capacitor without modification. It is interesting to note that the original Mica mote did have such hardware, but it was removed from subsequent versions of the family, presumably for efficiency reasons. The only other node is the BTnode, which does not meet some of the other design requirements.

The stated minimum input voltage for this chip is 0.7V. However, under low load conditions, I have seen it continue working as low as 0.4V (see Figure 4.1). It is in fact impossible to reach the ideal of 0V, since, for any nonzero power load, the current required will approach infinity as the voltage approaches zero. For all the current battery chemistries, 0.5V represents a depletion of near 100%, and could actually be harmful for the battery. And for a capacitor with a maximum voltage of 2.3V, this would correspond to a depletion of 95%, which is acceptable.

2.1.3. Radio Transceiver

For the radio transceiver, I selected from among three IEEE 802.15.4 compliant 2.4 GHz RF transceivers: the Texas Instruments CC2420, found in many of the current mote designs,

Table 2.1. Comparison of Radio Transceivers

Name	CC2420[89]	AT86RF230[90]	CC2520[91]
Receive Power	18.8	15.5	18.5
Transmit Power (0dBm)	17.4		25.8
Transmit Power (1dBm)		14.5	
Transmit Power (5dBm)			33.6
Active Power, no RX	0.5	1.5	1.6
Sleep Power	20	20	30
Receive Sensitivity (dBm)	-95	-101	-98
Adjacent Channel Rejection (dB)	30	34	49
Alternate Channel Rejection (dB)	54	52	54
Interrupt Lines	4	1	5
Distinct Interrupt Events	4	6	23
Encryption Support	AES-128	No	AES-128
Separate TX and RX Buffers	Yes	No	Yes
Buffer Size	128 bytes	1 packet	128 bytes
Destination Address Filtering	Yes	Yes	Yes
Source Address Filtering	No	No	Yes
Packet Sniffing	No	No	Yes

the Atmel AT86RF230 found in the Iris mote, and the Texas Instruments CC2520, which is TI's successor to the CC2420. Table 2.1 compares their performance and features[89],[90],[91].

While the other two chips have a lower power consumption, I decided to use the CC2520 for its features, especially the interrupt system, hardware encryption, and address filtering. It also has a much tighter bandpass filter, as indicated by the channel rejection data, which means it should be more robust against coexisting systems and wideband interference.

2.2. Schematics

The full system schematics can be found in the appendix, to which the reader may refer.

For the solar charging system, I decided to use the traditional Schottky diode instead of a more complicated circuit. Other ideas were considered, including a power MOSFET to lower the voltage drop, or a Maximum Power Point Tracking (MPPT) system, but in the end I didn't want to use any electronics that required assistance from the microcontroller, since it would be spending most of its time in sleep anyway. And building another system autonomous from the main microcontroller would add to the size and cost. Furthermore, for many possible capacitor and solar panel configurations, charging efficiency is not of the utmost importance, since the panel may be able to fully charge the capacitor in a short time. For example, a 100F capacitor can be charged to 2.3V by a solar panel that provides 40mA in direct sunlight in less than 2 hours. More important, in these cases, is how long the system can last without direct sunlight.

For solar panel and battery or capacitor combinations in which it might be necessary to prevent overcharging, I added an NMOS transistor between the solar node and ground, with the gate driven by the microcontroller. If the software determines that the charging should be shut down, it can drive the gate high, shunting the solar panel to ground.

One interesting design issue for power supplies is the "bootstrapping" problem. The DC-DC converter has an input pin which determines whether or not it is active. But what is to drive this pin? If it is to be driven by some logic, then that logic must itself have a different power supply, otherwise you will end up with a system that either oscillates or cannot transition between on and off. The solution proposed in [44] was to use a separate circuit powered by another solar panel. I find this approach to be too cumbersome. My solution was to feed back the output of the converter to the shutdown pin. Thus, if it is on, it stays on, and if it off, it stays off. The microcontroller has no control at all. How, then, does it transition between states? Since the microcontroller needs power to maintain its state and operate the real time counter even in deep sleeps, I pursued an always on

approach. The converter stays active until there is no longer enough energy to maintain the output voltage, which ideally should never happen if the application can throttle its usage depending on the energy available. But if the converter is off, how will it restart itself when energy is available? For self-starting, all that is needed is a Schottky diode between the input and output, so that the output will follow the input as it rises. Once it reaches the threshold for the chip, it will activate and the output will rise to 3.3V, reverse biasing the diode. Since the activation voltage is higher than the minimum voltage needed to sustain the output, this acts as hysteresis, and should prevent any oscillations from occuring.

I did not include a coulomb counter to accumulate the charging and discharging currents because, while it would be useful to have this data, I did not want to further add to the system size and complexity, and most coulomb counters are designed for a specific battery chemistry. Furthermore, for capacitors, the energy flow can be calculated from the voltage readings, as I will show in Chapter 4.

The microcontroller interfaces to two connectors. The first is a 10-pin JTAG connector for programming and debugging. Since the microcontroller has an integrated USB interface, I included a USB connector. This means that each node also has the capability of interfacing directly to a PC and serving as a base station without any extra hardware, although this functionality has not been implemented in the software yet. For dedicated nodes, the USB connector could be removed to reduce the size and cost.

The microcontroller and radio transceiver communicate using the SPI interface, and some extra pins, since the CC2520 provides six reconfigurable general purpose IO (GPIO) pins. And of course the microcontroller drives the radio's enable and reset pins. The microcontroller's USARTs, SSC, and TWI modules are unused, thus their clocks are disabled by the software at system startup to reduce power consumption.

The microcontroller includes its own RC oscillator which can be used to drive the main clock and the integrated Real Time Counter (RTC). The radio transceiver requires a 32MHz crystal, which is then multiplied to 2.4GHz for the RF interface. It can also divide the

32MHz oscillator using a counter and output this on a pin for other components to use. By default, it outputs a 1MHz clock, which I have routed to the microcontroller so that it can be used as its clock source. By altering the configuration of the radio and microcontroller, a wide selection of clock frequencies is possible, ranging from the default 115kHz up to a maximum of 60MHz.

The microcontroller is also capable of driving a 32kHz crystal as another source for its RTC. This is more accurate than the onboard 115kHz RC oscillator, but I decided to forego it in favor of a one-crystal design. However, it could easily be incorporated for designs that require the extra accuracy for timing synchronization.

The two sensors, a phototransistor and a thermistor, are both implemented as voltage divider circuits, with the ADC sampling the interior node. The process for converting the sampled voltage into the underlying sense data is covered in the next chapter.

I included red, green, and blue LEDs for user feedback, which are connected to GPIO pins on the microcontroller through a series resistor. The series resistors were selected so that each LED consumes 5mA when active.

Texas Instruments supplies a reference design for the CC2520, along with a few antenna options[92]. I chose to use a pcb antenna, for reduced mechanical complexity and component count. For pcb antennas, TI provides a planar inverted-F antenna and a fully differential folded-dipole design. I decided to use the inverted-F PCB antenna, because, although the fully differential folded-dipole design requires no balun and is thus easier and cheaper to implement, the dimensions were too large for my size requirements.

2.3. PCB Layout

The PCB layout process has three major steps: deciding on a specific part for each element in the schematics, importing their footprints from a library or creating them from scratch, and doing the place and route. Each of these took a significant amount of the time for this project. Before selecting parts, I decided to use surface mount technology exclusively, and also to make the entire board Reduction of Hazard Substances (ROHS)[93]

compliant, which means that the entire board is free of lead, mercury, cadmium, and other toxic chemicals.

The footprints for many of the components, especially the bypass capacitors, were standard, but for the rest, I had to create the footprints "by hand" using the dimensions given in the datasheets. While this was a tedious process, it taught me several lessons about proper pcb design. Before doing the place and route, I decided on dimensions of 1.2"x2.8". This fits a standard 2 AA battery holder, with the PCB antenna extended slightly off one end. I also decided to place all the components on one side and to avoid using interior layers if possible, for reduced costs. To reduce noise, I made most of the bottom layer a ground plane, and left most of the signal routing on the component side, only using vias to the back side when necessary. The microcontroller and radio chip are connected directly to the ground plane through vias under the chips.

The most difficult part of the layout was copying over the balun and antenna from the reference design, since it uses pcb tracings as inductors, which must be exact, and I was unable to use the original source files. One mistake I made was not being sure of the pcb design rules for minimum copper width, drill hole size, and soldermask spacing before beginning. I made the design with one set of rules, then when I chose a manufacturer, they had another set of requirements, which required me to rework some of the board. It would have been better to decide on a manufacturer first, find their design rules, and then begin. Also, I should have included a power connection for the expansion sensors. As it stands, there are only the ADC connections and one ground connection. Figure 2.1 gives the final pcb layout, with all the major components labeled. The cost for the prototypes, and the estimated cost for 100 units is given in Table 2.2.

After building these prototypes, I see several ways in which the size could be reduced. I ended up with some empty space around the radio transceiver. Also, a lot of the space was used for signal routing, which could be reduced by using an internal signal layer, although this would increase manufacturing costs. Also, I could have used a smaller JTAG connector.

Figure 2.1. PCB Layout

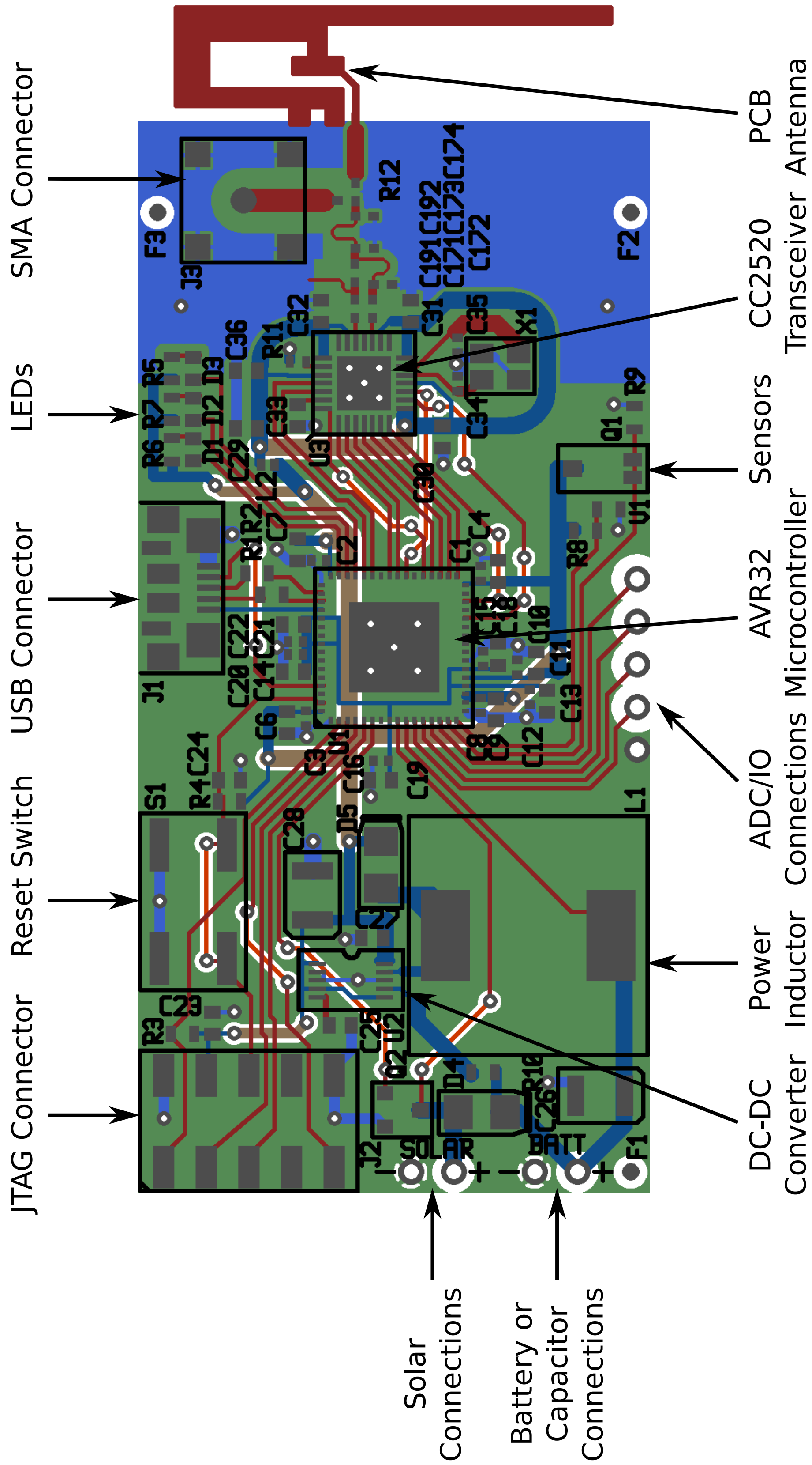


Figure 2.2. Prototype

Table 2.2. Hardware Costs

	For 3 Units	For 100 Units (Estimated)
Parts	$38.25	$20.00
Boards	$30.00	$5.00
Assembly	$250	$15.00
Total	$318.25	$40.00

I used this one because it matches the default connector for the programmer hardware, but I could have built another one to match whatever I decided to use on the board. The inductor and USB connector have already been mentioned. The next chapter will cover the software I created to run the system, and the last chapter will cover testing of the boards and design verification.

CHAPTER 3

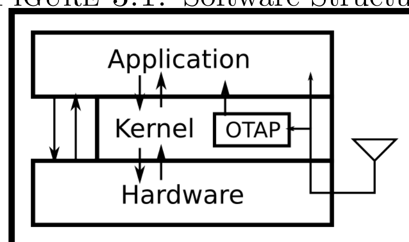
SOFTWARE DESIGN

To test and verify the functionality of the hardware, implement a basic over-the-air programming system, and provide a basis for further work, I have written a minimal operating system kernel. Although the hardware may be programmed with a single monolithic application that runs in a privileged mode, as with the existing hardware platforms, for a robust reprogramming system it is essential to use the kernel/user boundary. This kernel is a work in progress, and should be considered alpha level.

My vision of the software structure is given in Figure 3.1. There is to be a small kernel of trusted and tested code which controls access to the radio transceiver and the microcontroller configuration. This is analogous to machine M1 from Figure 1.2. It is not necessary for the kernel to protect all parts of the hardware, since, for example, the LEDs and ADC are not critical to the reprrogramming mission. But the user may wish to include this functionality in the kernel. The idea is that the kernel should be small and easy to understand so that it can be modified by the architect of the sensor network. Then the application program is to implement the main functionality of the system, and run in the unprivileged mode.

In short, this is a microkernel design, and is not intended to provide most of the traditional operating system services. There is no multitasking or any process management, or than setting up the memory protection, as it is assumed that there is only one application running.

FIGURE 3.1. Software Structure



The application is responsible for switching between different tasks if required. There is no file system as well, and nothing resembling file services.

How much of the networking stack to include in the kernel is a difficult design problem. For simplicity and reliability, it is best to include as little as possible and let the application implement the rest. Unfortunately, some upper layer services might be necessary for the reprogramming system, depending on the chosen algorithm. The initial version of my over-the-air programming system, which will be introduced shortly, only requires the physical layer services. Therefore my kernel only implements the physical layer, although a MAC layer implementation could be added.

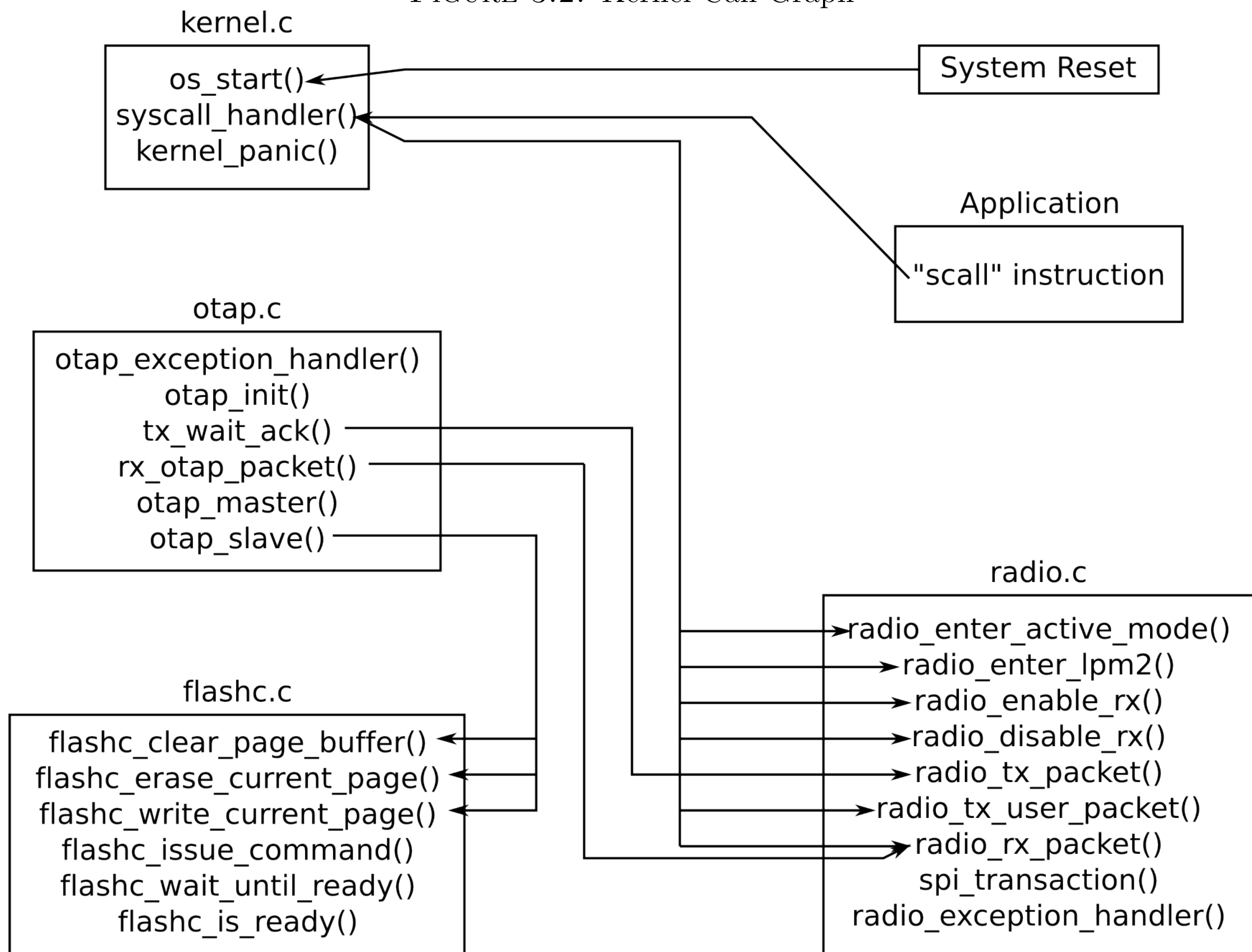
Currently, Atmel only supplies a version of the GNU toolchain[94] for the AVR®32, so the language choices are assembly, C, and C++. I chose to use C. Since nesC translates to C first[11] before it is compiled, it should not be too difficult to build a nesC compiler for this architecture, but I have not pursued this further. This could be a direction for future work. It should be emphasized that there is no requirement for the kernel and application to be written in the same language. The kernel could be written in C, and the applications written using nesC, if a compiler is made available.

The source code requires some files from Atmel which supply defines for the hardware and data types, and a few drivers from the AVR32 UC3 Software Framework[95], but is otherwise standalone. The source is written with formatted comments which are meant to be extracted using the Doxygen[97] automatic documentation generation system.

3.1. Kernel Structure

The kernel is broken down into three main components, each in its own file, with some subsidiary files for drivers and the C runtime. At system reset, execution begins in the C runtime setup (crt0.S), which merely copies any initialized data to RAM and sets some of the registers, then transfers control to the initialization function in the main kernel file, kernel.c. Its tasks are to bring all the pins of the microcontroller into their default configuration, disable the clocks for all unused modules, then configure the memory protection, which is

Figure 3.2. Kernel Call Graph



covered in the next section. Then it initializes the interrupt system, and jumps to the user application. This file also includes the system call handler, which is called whenever the application executes an scall instruction. It performs the requested task, and returns to the user. The system calls are covered in another section. A call graph of the major functions in the kernel is given in Figure 3.2.

The radio subsystem is in the file radio.c. It is a driver for the TI CC2520 radio transceiver. Not all of the functionality of the CC2520 is implemented, but the most significant parts are. It includes functions for bringing up the radio into active mode, bringing

it down into sleep, activating the receiver, and receiving and transmitting packets. The automatic encryption functions are not yet implemented. This file only implements the IEEE 802.15.4 physical layer.

A significant design decision was how to handle received packets and power for the radio. This is a complicated issue that ties together with the over-the-air programming system and energy management. As it stands, energy management is the sole responsibility of the application program. While it is certainly possible to use the real-time clock to impose quotas on the application execution time, I have not implemented this yet. This is a potential weakness for the reprogramming system, because the application could potentially deplete the node, rendering it unable to reprogram. Also, since the application is responsible for energy management, it is allowed to shut down the radio via a system call, rendering the system unable to enter reprogramming mode if a request should arrive. This will be discussed in the reprogramming section. Currently, my procedure for handling received packets is as follows: When a packet arrives, the interrupt handler merely sets a flag indicating that data is waiting. The user application should notice this changle whenever it resumes execution, and then execute a system call to retrieve the data. This system call will then pull the data from the radio chip and analyze it to determine if it is a reprogramming packet. If not, it passes it on to the application.

3.2. Memory Protection and Regions

Table 3.1 shows the configuration of the memory protection system. It defines what areas of memory may be accessed, and the allowed permissions for both privileged and unprivileged modes. This means that the kernel itself can be restricted, allowing easier detection of kernel bugs. Any memory location not included in one of these regions is not allowed to be accessed. The hardware supports up to 8 regions, of which I am using 6. Each region is subdivided into 16 subregions. For each region, there are two sets of permissions. Each of the 16 subregions is then set to use one of these two possibilites. For more details on these memory areas, see the AT32UC3B data sheet[87]. The regions of the most interest

are the RAM, the Flash, and the Peripheral Bus A. Everything else is allocated solely to the kernel. The kernel is allocated 8kB of RAM with the other 24kB given to the application. The application *is* allowed read access to the kernel RAM, to avoid the need to copy radio traffic into user space, but this could be changed for added security. The kernel uses the lower 16kB of the Flash, with the rest available to the application. The application is not given write permission to its Flash area, but must go through the kernel. For the peripherals, the application is given direct access to the pulse width modulation (PWM) Timer/Counter (TC) and ADC modules, since these are incapable of disrupting the rest of the system. The General Purpose I/O (GPIO module is protected, meaning the application cannot retask any of the microcontroller's pins.

The memory protection works as intended, and has already helped me isolate several bugs in my code. When an illegal access occurs, an interrupt is thrown and the address of the offending instruction is placed on the system stack. This allows the error to be quickly located.

3.3. Over-the-Air Programming

The (preliminary) version of the over-the-air programming system is in the file otap.c. The central idea is that the programming happens by doing a direct transfer of a region of the Flash memory from a master device to a slave device, without any semantics applied to it. Typically this would be the entire application program. But this is a very flexible approach, so the transfer could just consist of a small bug fix or an extension to the current program. It is up to the system initiating the reprogram to ensure that the transfer goes into the correct memory location, and that the resulting memory represents a meaningful program.

As it stands, the reprogramming process is to be entirely controlled by a base station, or a master node. The individual nodes perform no automatic checking for updates, or communication with their neighbors to compare program versions. This is also not a flooding

Table 3.1. Memory Protection Regions

Region	Description	Base Address	Size	Subregion Size	Subregions	Privileged Permissions	Unprivileged Permissions
0	RAM	0x00000000	32kB	2kB			
					0-11	RW	RW
					12-15	RW	R
1	CPU Local Bus	0x40000000	4kB	256B	0-15	RW	None
2	Flash	0x80000000	256kB	16kB			
					0	RX	None
					1-15	RW	RX
3	USB Configuration	0xd0000000	64kB	4kB	0-15	RW	None
4	Peripheral Bus B	0xfffe0000	64kB	4kB	0-15	RW	None
5	Peripheral Bus A	0xffff0000	16kB	1kB			
					0-11,13	RW	None
					12,14,15	RW	RW

system. Each program transfer is between a master and slave. Afterwards, the slave may become a master to transfer to another node, but only until its transfer is complete.

A significant problem is how to handle a multihop system. With this implementation, there must be a single hop between master and slave. Thus, the program must be propagated along each step until the desired node is reached. There is still a multihop issue to be addressed. For how is the base station to command a distant node to reprogram one of its neighbors? To handle this, I have a dedicated packet type that wraps one of the other OTAP packets, which are enumerated below, and includes all the routing information it needs to get to its destination.

The reprogramming system contains five distinct packet types. There is an alert packet, which just informs the surrounding nodes or base station that the OTAP module has control of this node. There is an "initiate reprogramming as master" packet, which commands a node to initiate a reprogramming on one of its neighbors. There is an "initiate reprogramming as slave" packet, which is sent by the master to its target after it receives the above packet. There are data packets, which are sent in sequence once the reprogramming is initiated. And, last, there is the multihop wrapper packet, mentioned above. If a node receives one of these packets, it strips the first bytes of data off of it, which tells it where to send the packet next. Thus, the base station must know the topology of the network, and is responsible for routing the master initiation packet to its destination. This relieves the nodes from having to do any routing for the reprogramming system. This way, it is not necessary to have any of the upper layers of the network stack in the kernel. In fact, all that is needed is the physical layer. OTAP packets are distinguished by their PAN identifier fields. I use a dedicated PAN ID, which application programs may not use.

The reprogramming system can be triggered in one of two ways. The normal mode of entry is for the system to take control whenever a packet is received that is from the dedicated reprogramming PAN ID. Whenever the application program executes the system call to retrieve a packet, the kernel parses the packet and determines if it is a reprogramming packet.

However, as mentioned above, this requires the cooperation of the application program, which is what this system was designed to prevent. Also, even in the normal course of events, the radio will not be receiving most of the time anyway, as this is much too energy intensive. Therefore another entry mechanism must be available.

In other words, for most of its life the node will be isolated from the network. How, then, to guarantee that it can be reprogrammed? There seems to be no other option than to wake the node on a regular basis, enable the radio, and listen for activity. This is the approach I've taken. At set intervals, the kernel will take control, transmit one of the alert packets mentioned above, and activate the receiver for a certain amount of time. This provides a guaranteed window for the node to be reprogrammed. The base station, upon receiving this alert packet, can then send an initiation packet back to the node. The only remaining issue is how to guarantee that the alert packet successfully arrives at the base station, since the kernel only includes the physical layer. Currently, I have no solution to this problem.

Once a reprogramming packet has been received and the reprogramming module in otap.c has taken control, it parses the packet and performs the necessary actions. The following flowchart shows the general algorithm.

3.4. Application Program

Building an application program is straightforward. Three things are necessary: a compiler that targets the AVR32 architecture, a system call library, and a linker script that will link the program for the correct memory areas. As shown in Table 3.1, the application program is to reside in the Flash memory beginning at location 0x80004000 and can use the rest of the Flash. It is allowed to use the first 24kB of RAM. Besides the areas of Peripheral Bus A mentioned above, it may access no other memory locations. This constitues its "jail." When the kernel transitions to application mode, it does not provide any type of extra environment for the application, it merely jumps to memory location 0x80004000. Thus, the application must copy any initialized data to RAM before using it.

FIGURE 3.3. Over-the-Air Programming Flowchart

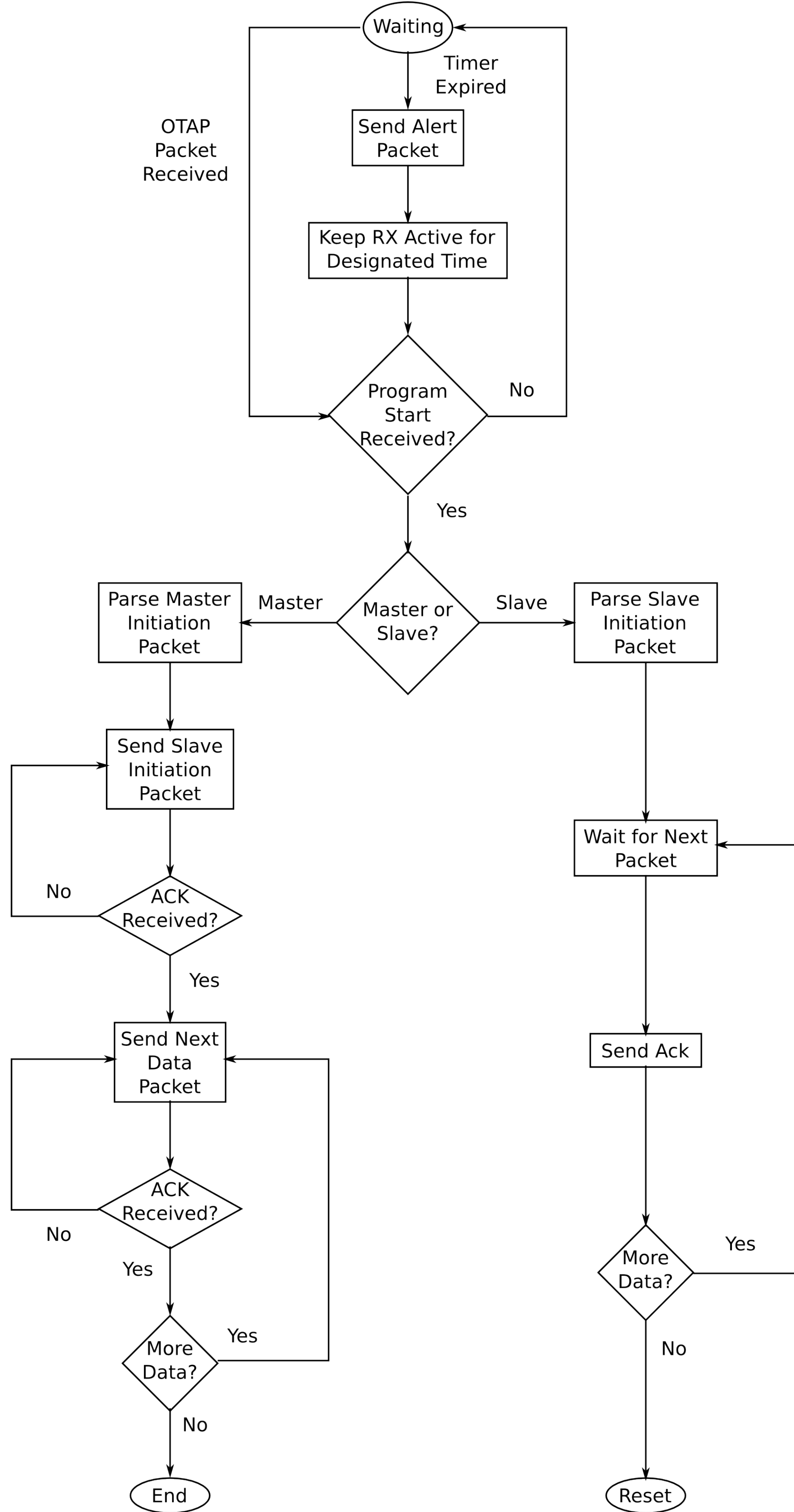


Table 3.2. System Calls

Name	Number	Parameter	Function
RED_LED_ON	101	None	Turns red LED on
RED_LED_OFF	102	None	Turns red LED off
GREEN_LED_ON	103	None	Turns green LED on
GREEN_LED_OFF	104	None	Turns green LED off
BLUE_LED_ON	105	None	Turns blue LED on
BLUE_LED_OFF	106	None	Turns blue LED off
RADIO_ACTIVE	107	None	Powers up radio
RADIO_SLEEP	108	None	Powers down radio
ENABLE_RX	109	None	Activates radio receiver
DISABLE_RX	110	None	Disactivates radio receiver
TX_PACKET	111	Packet	Transmits a packet
RX_PACKET	112	Buffer	Receives a packet
SLEEP	113	Sleep level	Sends microcontroller to sleep

Regardless of the language used, the linker must place the resulting code in the correct locations in the executable file. I have written a linker script which will do this for any linker which understands the common linker command language.

The system calls may be invoked manually using the *scall* instruction, or by linking to a system call library. I have provided such a library for the C language. To invoke a system call, the system call number must be placed in register R12, with any parameters placed in R11, R10, on down. Any return values are placed in R12. Table 3.2 gives the current list of system calls with their number, parameters, and return value.

CHAPTER 4

SYSTEM VERIFICATION AND CONCLUSIONS

In this chapter we will at last analyze the performance of the finished system and try to show that it meets the design goals that were set forth in Chapter 1. First I'll cover electrical testing of the boards and present some measurements for power consumption and transmission range. Then I'll show sensor readings for two experimental setups collected over a period of two weeks. These will show that the sensors work as intended, and also demonstrate the solar harvesting system in action. Then we'll turn to the issue of capacitor leakage and energy management. Finally, I'll wrap up by suggesting some areas for further work.

4.1. Hardware Testing

The bare circuit boards were electrically tested by the manufacturer prior to assembly, so I could be reasonably sure that the traces and vias were functional. Furthermore, since this design has no internal layers, all traces can be inspected visually. The assemblers also performed x-ray inspection of all the solder joints for the small components, and, most especially, the leadless IC's (the microcontroller and radio). But in the event that there was a trace or solder failure, my plan was to use the solder joints and vias as test access points. Almost every net in the circuit can be reached this way.

Seeing no need to manually test each net first, I began with a functional test of the system. First, I applied power and verified that the DC-DC converter generates a steady 3.3V supply. Then I plugged the JTAG programmer into the JTAG port and attempted to communicate with the microcontroller. The microcontroller responded with its manufacturing data. Once communication was established, I wrote a small program to test the LEDs, and then began writing the radio driver. After several days I had a working program that allowed me to

Table 4.1. Power Consumption in Various States

Microcontroller	Radio	Vin (V)	I (mA)	P (mW)
Deep Sleep	Deep Sleep	3.4	0.133	0.45
Idle	Listening	3.4	26.5	90.1
Active	TX/RX	3.4	40	136

verify that the radio chip and antenna work by pinging packets back and forth between two nodes. At this point I was confident that the prototypes were defect free. The only component that has not been tested is the USB connector, since I have not yet written any software to use that interface. So far, I have not detected any hardware errors or glitches in the system.

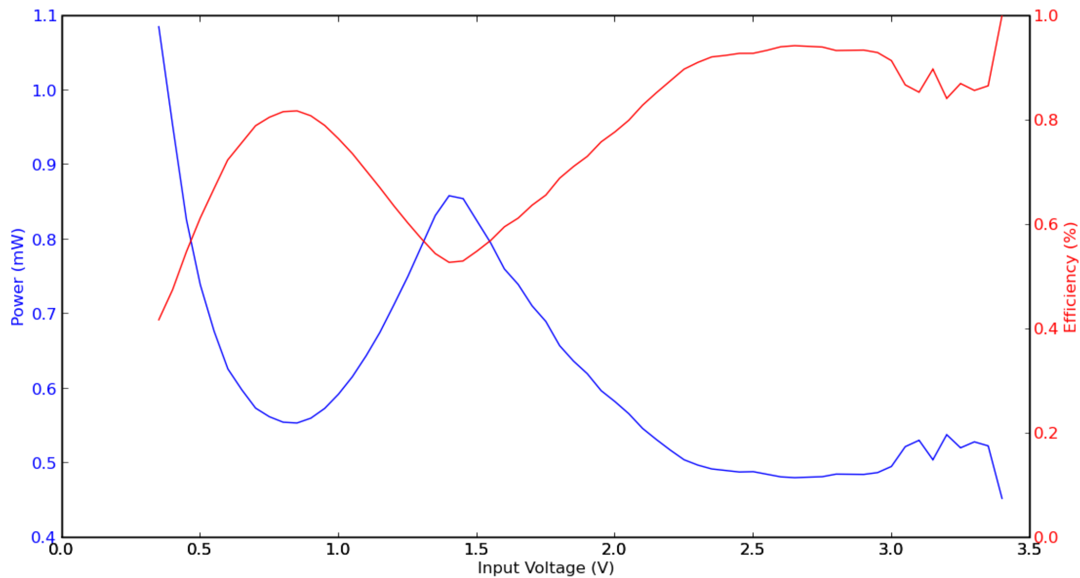
4.2. Power and Range Measurements

To measure the power consumption, I applied a 3.4V regulated input to the system, in series with a multimeter. This effectively eliminates the DC-DC converter, since the output voltage through the Schottky diode will stay above its target of 3.3V, at least for light loads. In the next section I will examine the efficiency of the DC-DC converter. Table 4.1 gives the measured power consumption for several common system states.

The power consumption while sleeping is very sensitive to the configuration of the pins in the microcontroller. In fact, during optimization, changing the configuraion of just one pin lowered the power consumption by a factor of two. Each pin can either be driven by the microcontroller, driven by another device, or left floating, and each pin has an integrated pull-up resistor which may be enabled or disabled. I believe that I've found the configuration that results in the minimum current draw. This is found in the source code for kernel.c.

Next I performed a radio range measurement. The data sheet for the CC2520[91] claims a 400m line-of-sight range. This of course will depend on the gain of the antenna, and the data sheet does not state which antenna was used for the testing. My testing indicates a line-of-sight range of roughly 100m.

Figure 4.1. Power Consumption in Sleep Mode and Efficiency of DC-DC Converter



4.3. DC-DC Converter Efficiency and Sleep Power Consumption

A concern when using power electronics for low-power systems is their efficiency. This becomes especially important for the light loads which will occur in deep sleeps, as the power consumption while sleeping is a primary determinant of system lifetime. The DC-DC converter used in this design is highly efficient, but its efficiency does decline as the load drops[88]. This is inevitable, due to leakage and the deliberate losses in the damping circuitry used to prevent ringing. The data sheet shows the efficiency as a function of load, but not as a function of the input voltage. To quantify this and verify the manufacturer's efficiency data, I programmed one of the nodes to enter and stay in a deep sleep. In this state, only the DC-DC converter and real-time clock in the microcontroller are operational. I then measured the current draw for input voltages ranging from 3.4V all the way down to as low as the system would stay alive. Figure 4.1 shows the power consumption and efficiency versus input voltage.

From these charts we see two things. First, the power consumption is below 1mW for the entire input range, except at the very end before it drops out. For about half of the

input range, the power consumption is below 0.6mW. Thus that design goal has been met. Second, there is a "hump" in the curve, where the DC-DC converter drops significantly in efficiency, corresponding to an input voltage of roughly 1.3-1.7V. This is possibly due to the converter transitioning between two different modes of operation.

4.4. Converting the Sensor Readings

Before presenting the sensor readings, I will discuss how the raw data for the sensors can be converted back into the originating stimulus. To begin with, the ADC takes an input voltage and converts it into a 10-bit value, with the maximum value defined as the system voltage (3.3V in this case). Therefore, to convert the binary value back into voltage, you use the following formula:

$$V = x * 3.3/1023 \tag{3}$$

This will suffice for the battery and solar measurements, since we are interested in the voltage. But for the light and temperature sensors, another step is necessary. Both sensors are implemented as voltage dividers, with the ADC sampling the interior node.

4.4.1. Light Sensor

The light sensor uses a phototransistor which acts as a current source proportional to the amount of incident light (illuminance). The photo current versus illuminance curve is highly linear, with the following relation[96]:

$$E = 2 * 10^6 * I \tag{4}$$

The current can be found using Ohm's law across the series resistor.

$$I = \frac{V_{ADC}}{R_9} \tag{5}$$

Therefore the total conversion can be written as:

$$E = \frac{2 * 10^6 * V_{ADC}}{R_9} \tag{6}$$

$$R_9 = 10k\Omega \tag{7}$$

4.4.2. Temperature Sensor

The temperatue sensor is a thermistor, whose resistance varies with temperature following an exponential relationship given by the B-parameter equation[98]:

$$R = R_0 * \exp B[1/T - 1/T_0] \tag{8}$$

where R_0 and T_0 are the resistance and temperature at a known reference. The resistance of the thermistor must be calculated using the voltage obtained from the ADC. Analyzing the voltage divider gives:

$$R = \frac{R_8 * V_t}{V_{cc} - V_t} \tag{9}$$

Rearranging and combining these equations gives the temperature as a function of the measured voltage and several parameters[98], as follows:

$$T = \left[1/T_0 + 1/B * ln[\frac{R_8 * V_t}{R_0 * (V_{cc} - V_t)}]\right]^{-1} \tag{10}$$

$$T_0 = 25\text{deg}C = 298.15K \tag{11}$$

$$R_0 = 100k\Omega \tag{12}$$

$$B = 4300 \tag{13}$$

$$R_8 = 100k\Omega \tag{14}$$

$$V_{cc} = 3.3V \tag{15}$$

These conversions can either be included in the microcontroller, or performed by another machine once the data has escaped the sensor network.

4.5. System Demonstration

To test the sensors and to evaluate the performance of the solar harvesting system using capacitors, I made two experimental setups and deployed them outdoors for a period of two weeks. Both motes were set to wake every ten seconds, sample the sensors, store the data in the Flash, and then resume a deep sleep.

The first uses a 220F ultracapacitor for storage, with a rated voltage of 2.3V. I paired this with a 4-cell solar panel with an area roughly $9in^2$, and deployed the setup in a small clear plastic container under the rear window of my car. Figure 4.2 shows the results. I have converted the raw temperature data into temperature using the conversion given above. From this we can see that the system is capable of perpetual operation. At the beginning the voltage drops sharply, but this is due to capacitor leakage, which will be discussed shortly. After that, it reaches an equilibrium level of roughly 1.6V. In the middle is a sequence of rainy days, which were under heavy cloud cover. Yet the capacitor was able to charge every day.

The second setup uses two AA alkaline batteries for power, with no energy harvesting, and a soil moisture sensor connected through the expansion connections on the side of the board. I placed the node and batteries in a slighly translucent plastic container and planted it in a potted plant outside my apartment, under heavy shade. Figure 4.3 shows the results. Here we see that the light sensor displays a more useful range. In the other setup, it is usually either saturated or off. I am not sure how to interpret this soil moisture data, and whether it indicates if the sensor is working correctly.

4.6. Capacitor Leakage Measurements and Energy Management

Previous research[44] indicates that capacitor leakage can be an important factor in the energy equation, and should not be ignored. Their measurements show that the leakage power grows in a steep, possibly exponential curve, as the capacitor voltage approaches its rated voltage. To measure this effect, I charged an ultracapacitor to its rated voltage of 2.3V and used it to power one of the sensor nodes running the same program used to collect the above data. That is, it wakes every ten seconds to sample the sensors and then returns to sleep. But in this case, I used no solar panel and left it indoors, so that readings could be taken while the capacitor discharged. Now under the assumption that the power used by the mote is constant (which is not quite true, as we've seen above, due to the DC-DC converter efficiency), we can calculate the total power dissipation based on the

FIGURE 4.2. Experimental Setup #1

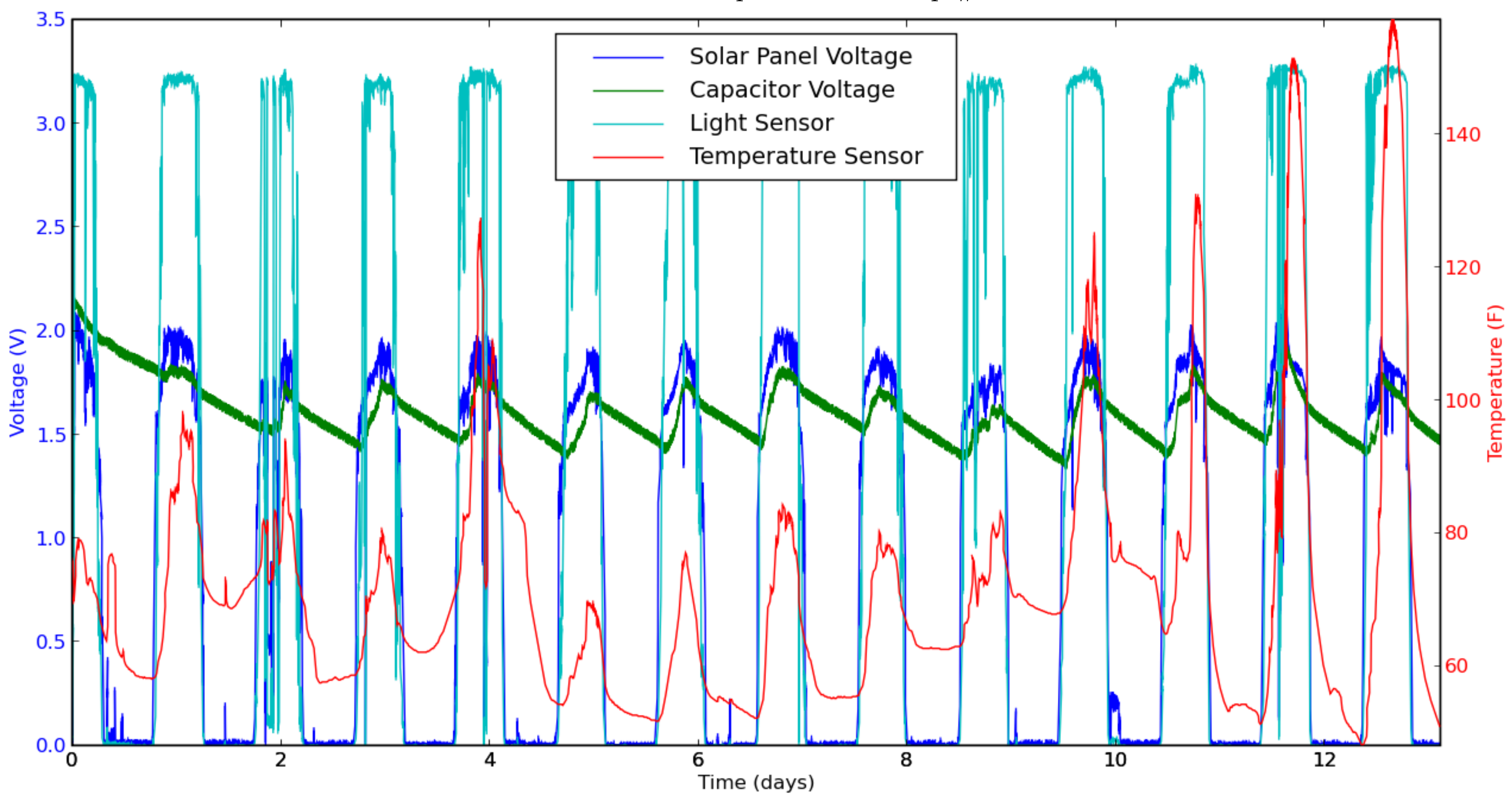
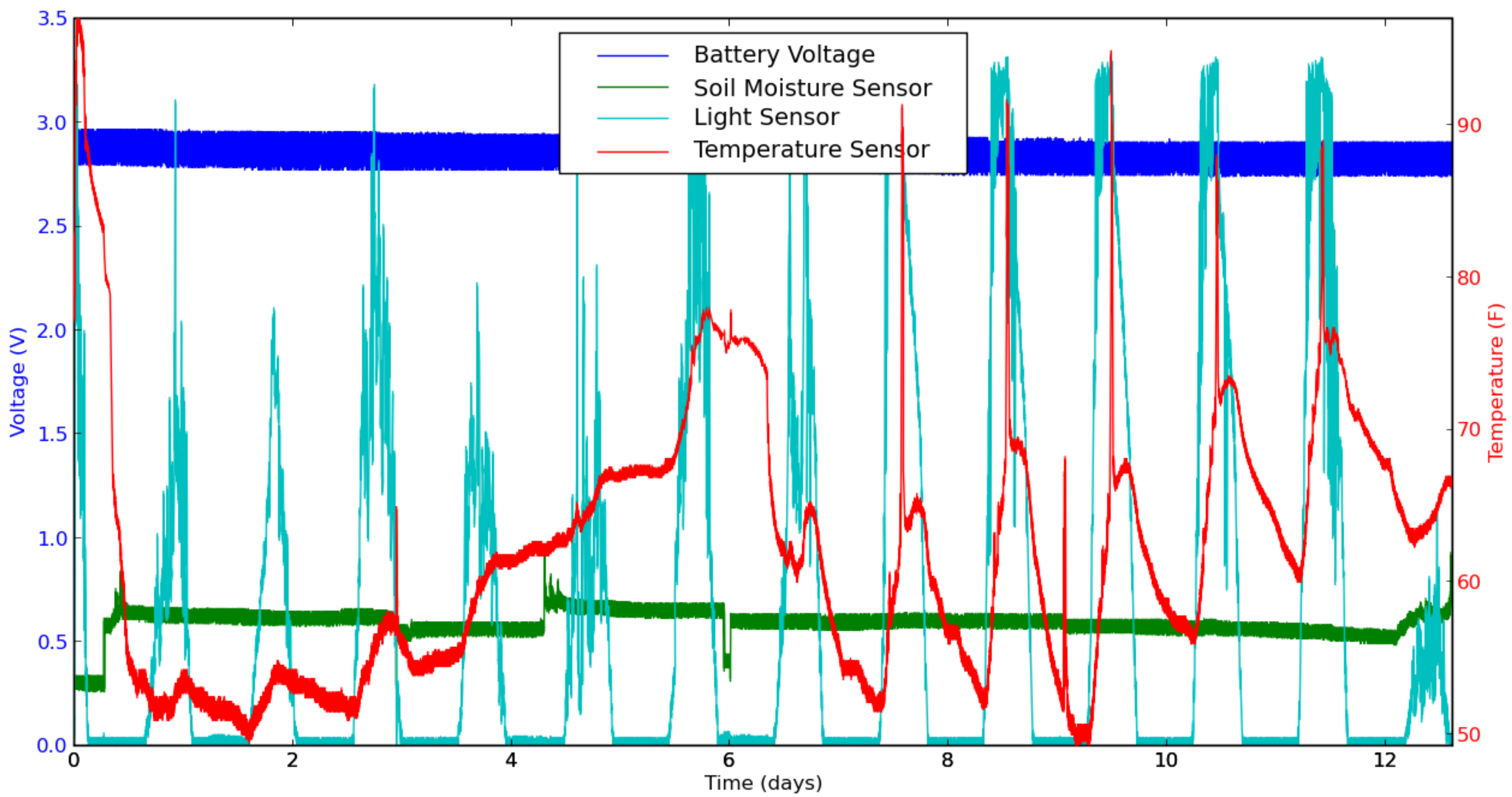


FIGURE 4.3. Experimental Setup #2



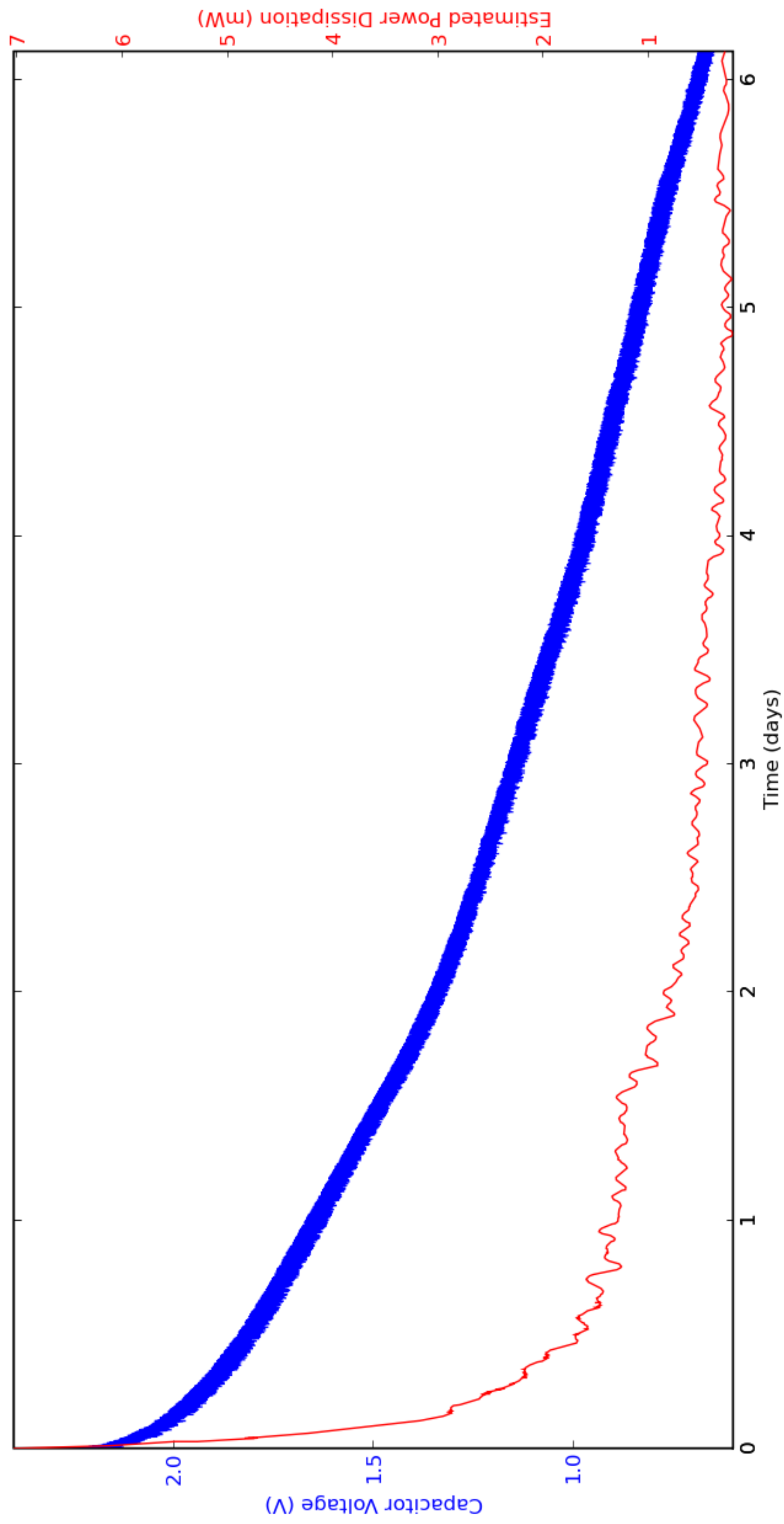
capacitor voltage measurements, and any extra loss must be due to leakage. In other words, $P_{total} = P_{mote} + P_{leakage}$. This also serves as a case study in estimating energy and power from the capacitor voltage readings.

The capacitor voltage is calculated directly from the sensor readings. Then this can be converted to energy using $E = \frac{1}{2}CV^2$. Then, theoretically, the power can be derived by differentiating the energy. However, here a problem arises. Although the voltage trend line is unmistakeable, it is actually a rather noisy signal. Some of this is due no doubt to inaccuracies in the ADC and quantization noise. But I suspect a bigger source is due to the switching nature of the DC-DC converter. It works by building up a large current in the inductor which can reach up to 1A, then transferring this energy to the load. In between cycles, it may be drawing no current from the capacitor at all. Due to the internal resistance of the capacitor, this large current variation will cause variations in the voltage seen at the capacitor. Also, the line from the capacitor to the ADC is routed near the inductor, which may be causing additional crosstalk noise.

Furthermore, differentiation increases the higher frequency components, which are predominantly noise in this signal. So, to recover the underlying power signal, I had to apply an aggressive low-pass filter to the voltage signal. I tried to keep the filter length minimal, so that it could be easily implemented in the microprocessor, but I was not able to achieve acceptable results with a filter of length less than 512. The filter used here is a 1024 tap FIR filter with a normalized cutoff frequency of 0.002 rad/sample. The results are given in Figure 4.4.

Despite the difficulty in recovering the signal, it shows what I expected. The capacitor does indeed exhibit a large leakage power for high voltages which dwarfs the power consumed by the device. The low-pass filter obscures the peak, which is roughly 10mW at 2.3V. This matches the curves given in [44]. As the voltage decreases, so does the leakage, but it still dominates the device power until the voltage is below 1V. The “hump” from Figure 4.1 is also visible as the capacitor transitions through 1.5V.

FIGURE 4.4. Capacitor Voltage and Power for Leakage Estimation



This shows that not only can the leakage not be neglected, but it causes diminishing returns for any attempts to reduce the power consumption of the device in sleep mode, as it is not the dominant factor. The approach taken in [44] is to use a leakage model to adjust the duty cycle, and actually *increase* the power consumption of the device, since the energy was going to be used anyway. Then, when the voltage is lower, the duty cycle can be reduced to conserve energy and wait for another recharging cycle.

4.7. Code Security and Reprogramming

Verifying the robustness of the over-the-air programming system is still an area that needs much work. So far I have demonstrated that the application jail works as intended by loading random code into the application space to see if it could compromise the system. It quickly traps to the kernel, which can choose not to return control. I have also verified the transfer of a program from one node to another, but that has been the extent of my testing. To facilitate further testing requires implementing some base station software using the USB interface, which could be an area for future work. Furthermore, it was not my intention to replace the existing protocols and algorithms, but to provide a new hardware platform designed to facilitate guaranteed recovery to which these protocols can be ported.

4.8. Conclusions

Lack of power source flexibility and fundamental hardware limitations have necessitated the creation of a new wireless sensor node which can meet the requirements that are being placed on sensor networks to be reprogrammable, stable, and sustainable. In Chapter 1 I laid out a list of design goals for this new hardware, to which we may now return for comparison:

(i) The microcontroller, through its memory protection and operating modes, does meet the requirements for virtualization set forth in [33].

(ii) The hardware starts up with an input voltage of roughly 0.7V, and stays operational under light loads down to 0.4V. The maximum input voltage is roughly 3.5V.

(iii) There is an integrated diode for solar charging, and a shunt transistor to prevent overcharging.

(iv) The power consumption in a deep sleep with the real-time clock running is under 1mW throughout the input voltage range.

(v) The radio interface is IEEE 802.15.4 compliant, with a line-of-sight transmission range of at least 100 meters.

(vi) The size is $21.7cm^2$.

(vii) The prototype costs were significantly over $40, but with a large batch it should be possible to meet this goal.

While these goals have (mostly) been met, there is still much room for improvement. I would like to conclude by offering some suggestions for what else could be done, and some possible directions for further research.

(i) A USB driver needs to be written into the kernel, and a base station program written that provides two-way communication with a pc through the USB interface.

(ii) Although it is 802.15.4 compliant, interoperability with other 802.15.4 motes should be demonstrated.

(iii) The layout could be reworked to shrink the size significantly.

(iv) A table of energy costs for certain standard operations could be calculated and compared against other platforms.

(v) Other sensor options should be provided, either natively on the board or through an expansion connector.

(vi) A leakage aware control system as in [44] could be implemented to allow dynamic scaling of the duty cycle.

(vii) It might be advantageous to build most or all of the 802.15.4 MAC layer into the kernel.

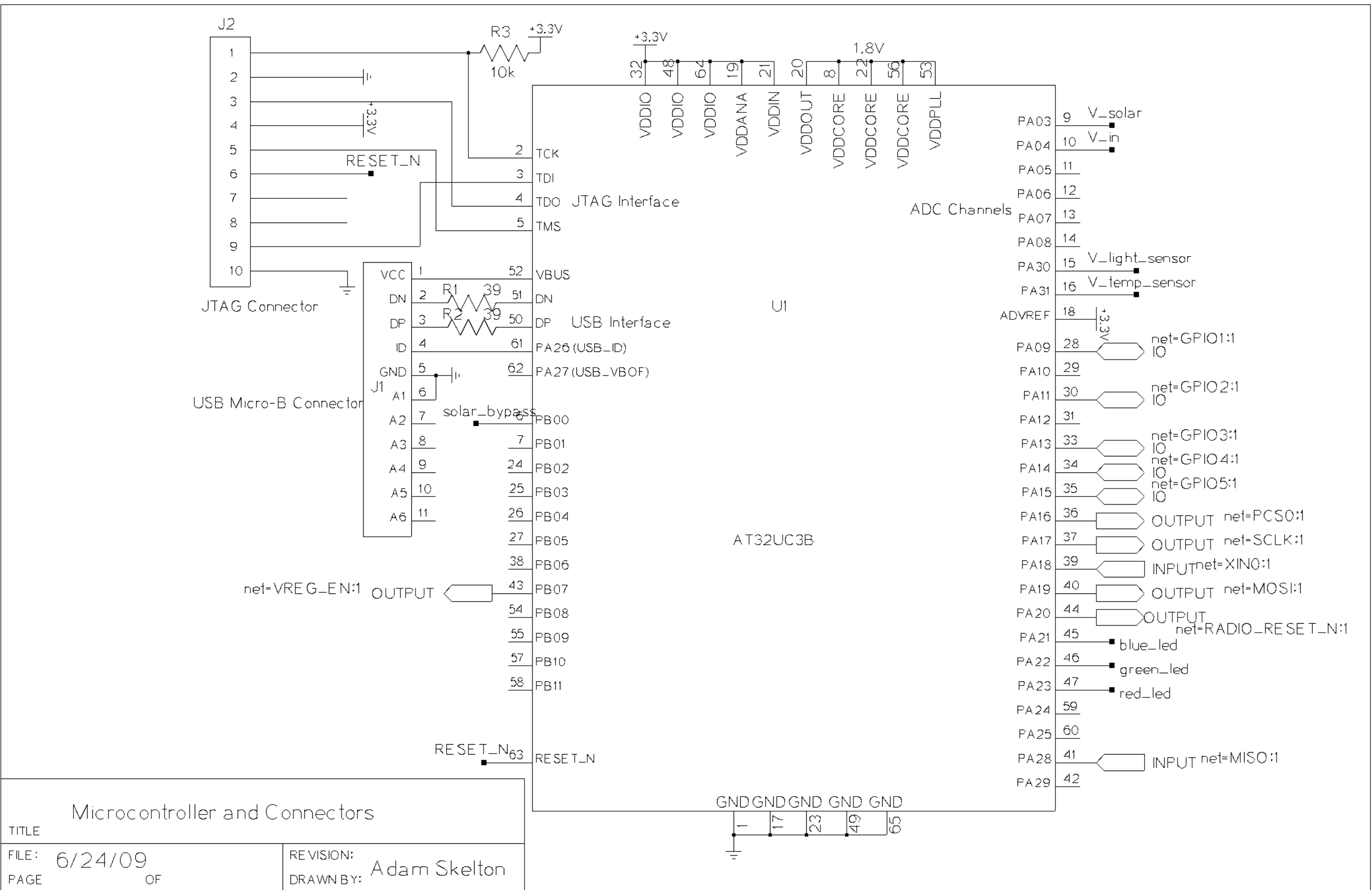
(viii) Different antenna options could be considered for improved range.

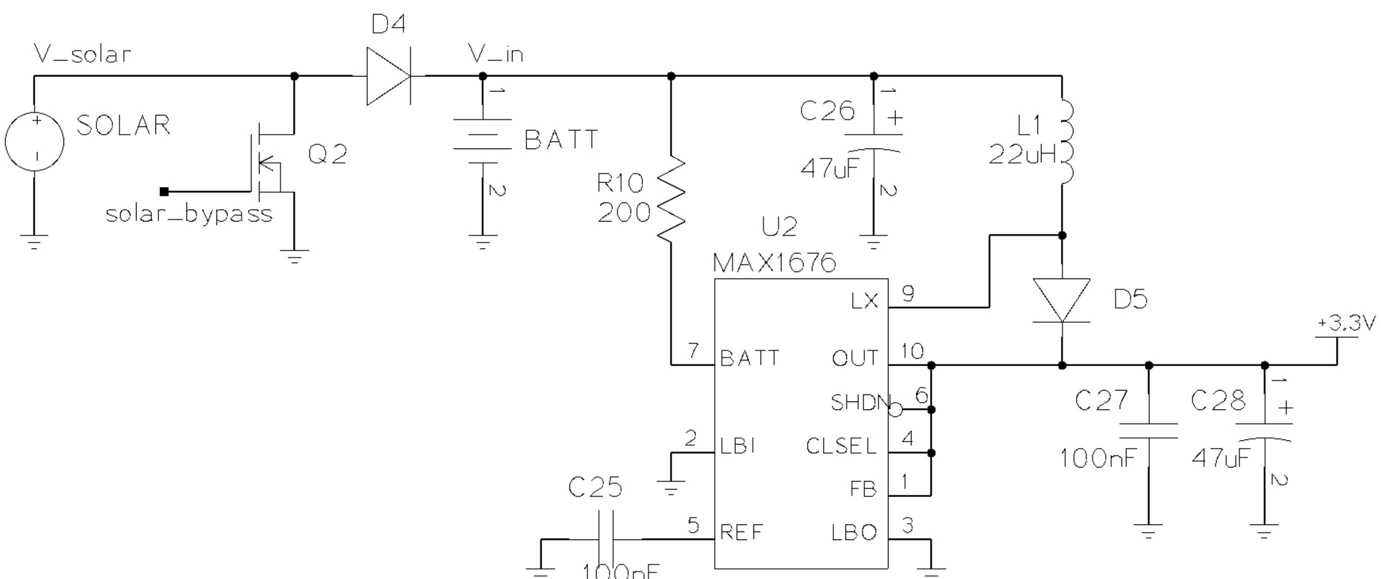
(ix) One of the existing multihop over-the-air programming systems should be ported to this platform.

(x) The feasibility of porting nesC and tinyOS to this platform should be investigated.

APPENDIX

SCHEMATICS


J2
JTAG Connector
R3 10k
+3.3V
RESET_N
JTAG Interface
TCK
TDI
TDO
TMS
VCC
DN
DP
ID
GND
A1
A2
A3
A4
A5
A6
J1
USB Micro-B Connector
R1 39
R2 39
VBUS
USB Interface
PA26(USB_ID)
PA27(USB_VBOF)
solar_bypass
PB00
PB01
PB02
PB03
PB04
PB05
PB06
PB07
PB08
PB09
PB10
PB11
net=VREG_EN:1 OUTPUT
1.8V
VDDIO
VDDANA
VDDIN
VDDOUT
VDDCORE
VDDPLL
U1
AT32UC3B
ADC Channels
PA03 V_solar
PA04 V_in
PA05
PA06
PA07
PA08
PA30 V_light_sensor
PA31 V_temp_sensor
ADVREF
PA09 net=GPIO1:1 IO
PA10
PA11 net=GPIO2:1 IO
PA12
PA13 net=GPIO3:1 IO
PA14 net=GPIO4:1 IO
PA15 net=GPIO5:1 IO
PA16 OUTPUT net=PCS0:1
PA17 OUTPUT net=SCLK:1
PA18 INPUT net=XIN0:1
PA19 OUTPUT net=MOSI:1
PA20 OUTPUT net=RADIO_RESET_N:1
PA21 blue_led
PA22 green_led
PA23 red_led
PA24
PA25
PA28 INPUT net=MISO:1
PA29
GND GND GND GND GND
Microcontroller and Connectors
TITLE
FILE: 6/24/09
PAGE OF
REVISION:
DRAWN BY: Adam Skelton



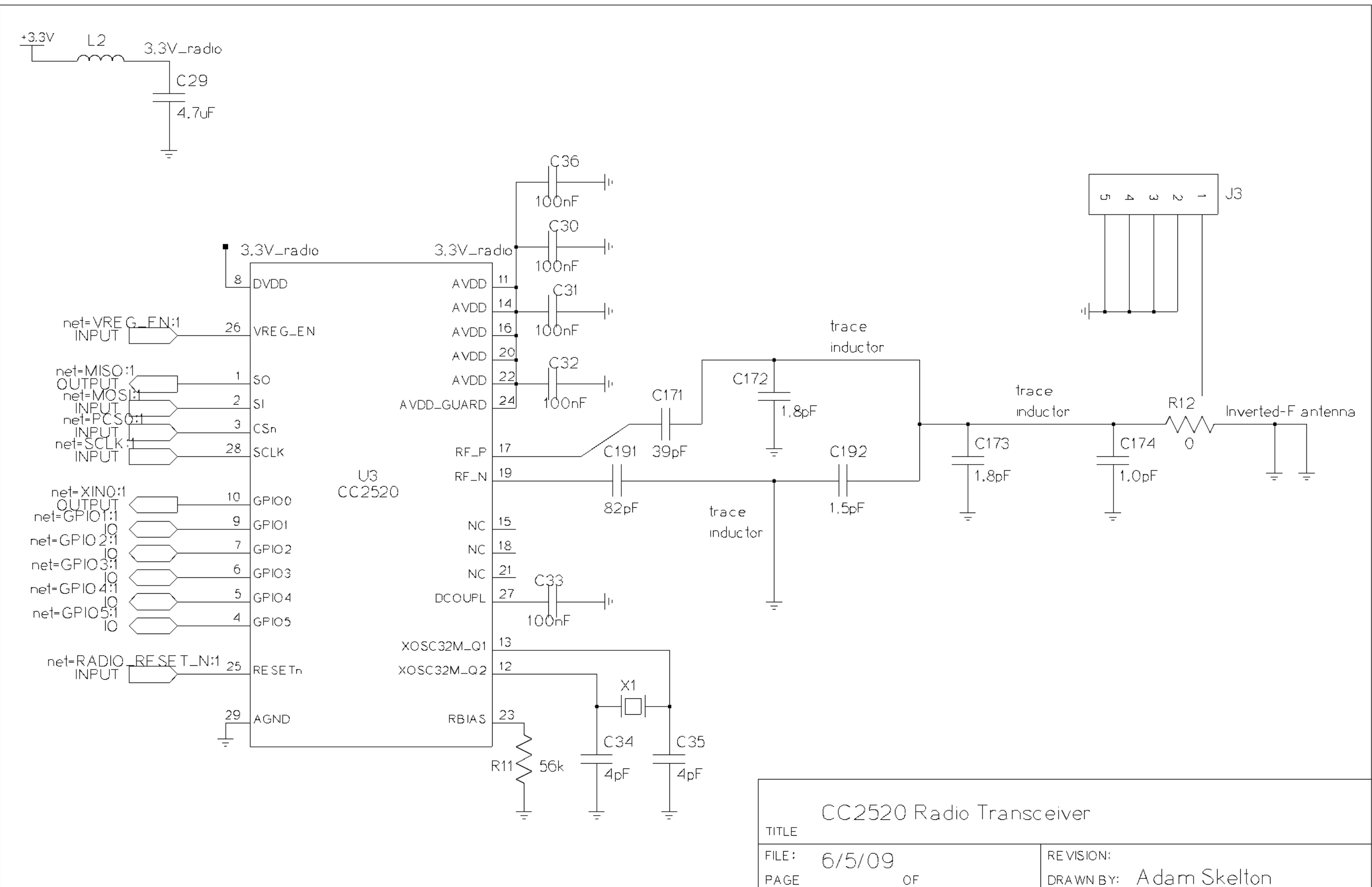
TITLE Power Electronics

FILE: 6/4/09

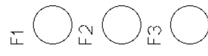
PAGE OF

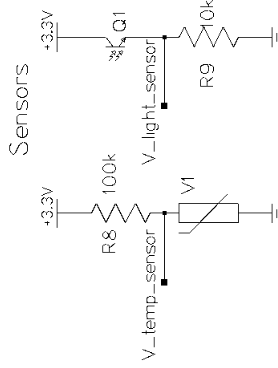
REVISION:

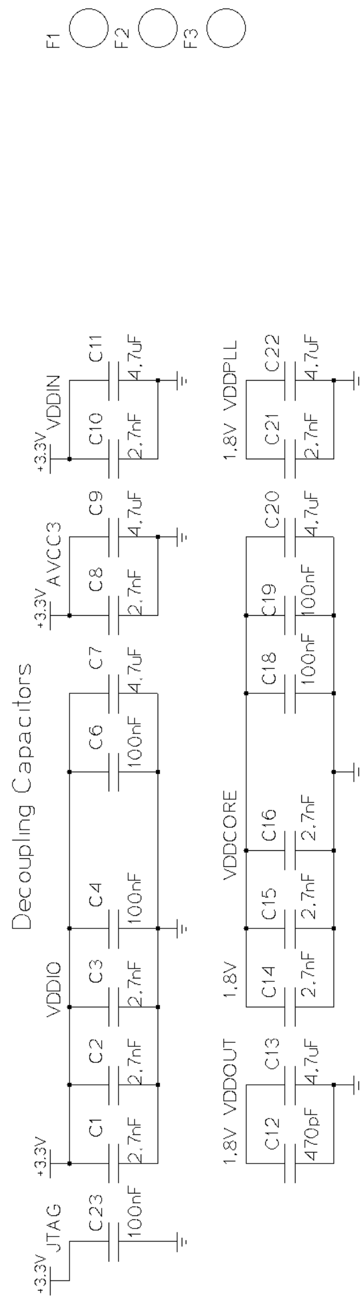
DRAWN BY: Adam Skelton

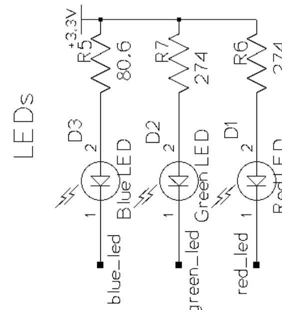


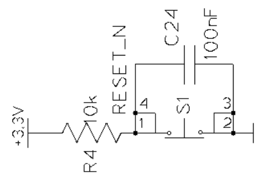
+3.3V
L2
3.3V_radio
C29
4.7uF
C36
100nF
C30
100nF
C31
100nF
C32
100nF
J3
U3
CC2520
DVDD
VREG_EN
SO
SI
CSn
SCLK
GPIO0
GPIO1
GPIO2
GPIO3
GPIO4
GPIO5
RESETn
AGND
AVDD
AVDD_GUARD
RF_P
RF_N
NC
DCOUPL
XOSC32M_Q1
XOSC32M_Q2
RBIAS
net=VREG_EN:1
INPUT
net=MISO:1
OUTPUT
net=MOSI:1
INPUT
net=PCS0:1
INPUT
net=SCLK:1
INPUT
net=XIN0:1
OUTPUT
net=GPIO1:1
IO
net=GPIO2:1
IO
net=GPIO3:1
IO
net=GPIO4:1
IO
net=GPIO5:1
IO
net=RADIO_RESET_N:1
INPUT
C171
39pF
C172
1.8pF
C191
82pF
C192
1.5pF
trace
inductor
C173
1.8pF
C174
1.0pF
R12
0
Inverted-F antenna
C33
100nF
X1
C34
4pF
C35
4pF
R11
56k
TITLE
CC2520 Radio Transceiver
FILE:
6/5/09
PAGE
OF
REVISION:
DRAWN BY:
Adam Skelton











Sensors
+3.3V
R8
100k
V1
V_temp_sensor
Q1
R9
10k
V_light_sensor

TITLE Sensors, LEDs, and Decoupling Capacitors
FILE: 6/5/09
PAGE OF
REVISION:
DRAWN BY: Adam Skelton

BIBLIOGRAPHY

[1] Mahesh (Umamaheswaran) Arumugam, *Infuse: a tdma based reprogramming service for sensor networks*, SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems (New York, NY, USA), ACM, 2004, pp. 281–282.

[2] H. Baldus, K. Klabunde, and G. Müsch, *Reliable set-up of medical body-sensor networks*, Wireless Sensor Networks, Springer Berlin / Heidelberg, 2004, pp. 353–363.

[3] J. Burrell, T. Brooke, and R. Beckwith, *Vineyard computing: sensor networks in agricultural production*, Pervasive Computing, IEEE 3 (2004), no. 1, 38–45.

[4] Zack Butler, Peter Corke, Ron Peterson, and Daniela Rus, *From robots to animals: Virtual fences for controlling cattle*, Int. J. Rob. Res. 25 (2006), no. 5-6, 485–508.

[5] Shu Chen, Yan Huang, and Chengyang Zhang, *Toward a real and remote wireless sensor network testbed*, WASA '08: Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 385–396.

[6] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr, *Efficient memory safety for tinyos.*, SenSys (Sanjay Jha, ed.), ACM, 2007, pp. 205–218.

[7] P.K. Dutta, J.W. Hui, D.C. Chu, and D.E. Culler, *Securing the deluge network programming system*, Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on, 0-0 2006, pp. 326–333.

[8] Jeremy Eric Elson, *Time synchronization in wireless sensor networks*, Ph.D. thesis, University of California, Los Angeles, 2003.

[9] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, *Connecting the physical world with pervasive networks*, Pervasive Computing, IEEE 1 (2002), no. 1, 59–69.

[10] Aurélien Francillon and Claude Castelluccia, *Code injection attacks on harvard-architecture devices*, CCS '08: Proceedings of the 15th ACM conference on Computer and communications security (New York, NY, USA), ACM, 2008, pp. 15–26.

[11] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, *The nesc language: A holistic approach to networked embedded systems*, PLDI '03: Proceedings of the ACM

SIGPLAN 2003 conference on Programming language design and implementation (New York, NY, USA), ACM, 2003, pp. 1–11.

[12] Lin Gu and John A. Stankovic, *t-kernel: providing reliable os support to wireless sensor networks*, SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems (New York, NY, USA), ACM, 2006, pp. 1–14.

[13] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg, *Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes*, IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks (Washington, DC, USA), IEEE Computer Society, 2008, pp. 457–466.

[14] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy, *The platforms enabling wireless sensor networks*, Commun. ACM 47 (2004), no. 6, 41–46.

[15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, *System architecture directions for networked sensors*, SIGPLAN Not. 35 (2000), no. 11, 93–104.

[16] Jason L. Hill and David E. Culler, *Mica: A wireless platform for deeply embedded networks*, IEEE Micro 22 (2002), no. 6, 12–24.

[17] V. S. Hsu, J. M. Kahn, and K. S. J. Pister, *Wireless communications for smart dust*, Electronics Research Laboratory Technical Memorandum M98/2, vol. 98, 1998.

[18] Jonathan W. Hui and David Culler, *The dynamic behavior of a data dissemination protocol for network programming at scale*, SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems (New York, NY, USA), ACM, 2004, pp. 81–94.

[19] Sangwon Hyun, Peng Ning, An Liu, and Wenliang Du, *Seluge: Secure and dos-resistant code dissemination in wireless sensor networks*, Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on, April 2008, pp. 445–456.

[20] Jaein Jeong and D. Culler, *Incremental network programming for wireless sensors*, Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on, Oct. 2004, pp. 25–33.

[21] X. Jiang, J. Polastre, and D. Culler, *Perpetual environmentally powered sensor networks*, Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, April 2005, pp. 463–468.

[22] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein, *Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet*, SIGARCH Comput. Archit. News 30 (2002), no. 5, 96–107.

[23] J. M. Kahn, R. H. Katz, and K. S. J. Pister, *Next century challenges: mobile networking for "smart dust"*, MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking (New York, NY, USA), ACM, 1999, pp. 271–278.

[24] Michael G. Kallitsis and Dr.Peng Ning, *Vulnerabilities of the deluge data dissemination protocol*, http://www4.ncsu.edu/~mgkallit/files/kallitsis06Vulnerabilities.pdf.

[25] Aman Kansal and Mani B. Srivastava, *An environmental energy harvesting framework for sensor networks*, ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design (New York, NY, USA), ACM, 2003, pp. 481–486.

[26] S.S. Kulkarni and Limin Wang, *Mnp: Multihop network reprogramming service for sensor networks*, Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on, June 2005, pp. 7–16.

[27] Philip Levis and David E. Culler, *Maté: a tiny virtual machine for sensor networks*, ASPLOS, 2002, pp. 85–95.

[28] Philip Levis, Neil Patel, David Culler, and Scott Shenker, *Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks*, NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (Berkeley, CA, USA), USENIX Association, 2004, pp. 2–2.

[29] Kris Lin, Jennifer Yu, Jason Hsu, Sadaf Zahedi, David Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani Srivastava, *Heliomote: enabling long-lived sensor networks through solar energy harvesting*, SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems (New York, NY, USA), ACM, 2005, pp. 309–309.

[30] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson, *Wireless sensor networks for habitat monitoring*, WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (New York, NY, USA), ACM, 2002, pp. 88–97.

[31] Rahul Mangharam, Anthony Rowe, Raj Rajkumar, and Ryohei Suzuki, *Voice over sensor networks*, Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International, Dec. 2006, pp. 291–302.

[32] K. Martinez and J. K. Hart, *Environmental sensor networks: A revolution in earth system science?*, Earth-Science Reviews 78 (2006), no. 3-4, 177–191.

[33] Gerald J. Popek and Robert P. Goldberg, *Formal requirements for virtualizable third generation architectures*, Commun. ACM 17 (1974), no. 7, 412–421.

[34] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava, *Design considerations for solar energy harvesting wireless embedded systems*, IPSN '05: Proceedings of the 4th

international symposium on Information processing in sensor networks (Piscataway, NJ, USA), IEEE Press, 2005, p. 64.

[35] Niels Reijers and Koen Langendoen, *Efficient code distribution in wireless sensor networks*, WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications (New York, NY, USA), ACM, 2003, pp. 60–67.

[36] K. Romer and F. Mattern, *The design space of wireless sensor networks*, Wireless Communications, IEEE 11 (2004), no. 6, 54–61.

[37] M.L. Sichitiu and C. Veerarittiphan, *Simple, accurate time synchronization for wireless sensor networks*, Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE, vol. 2, March 2003, pp. 1266–1273 vol.2.

[38] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton, *Sensor network-based countersniper system*, SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems (New York, NY, USA), ACM, 2004, pp. 1–12.

[39] Matthew Simpson, Bhuvan Middha, and Rajeev Barua, *Segment protection for embedded systems using run-time checks*, CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (New York, NY, USA), ACM, 2005, pp. 66–77.

[40] F. Sivrikaya and B. Yener, *Time synchronization in sensor networks: a survey*, Network, IEEE 18 (2004), no. 4, 45–50.

[41] Deborah Estrin Thanos Stathopoulos, John Heidemann, *A remote code update mechanism for wireless sensor networks*, Tech. report, November 26 2003.

[42] Lodewijk F.W. Hoesel van, Stefan Dulman, Paul J.M. Havinga, and Harry J. Kip, *Design of a low-power testbed for wireless sensor networks and verification*, 2003.

[43] C.M. Vigorito, D. Ganesan, and A.G. Barto, *Adaptive control of duty cycling in energy-harvesting wireless sensor networks*, Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on, June 2007, pp. 21–30.

[44] Ting Zhu, Ziguo Zhong, Yu Gu, Tian He, and Zhi-Li Zhang, *Leakage-aware energy synchronization for wireless sensor networks*, MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services (New York, NY, USA), ACM, 2009, pp. 319–332.

[45] *Global seismographic network*, `http://www.iris.edu/hq/programs/gsn`.

[46] *Snotel: Snowpack telemetry network*, `http://www.wcc.nrcs.usda.gov/snow/`.

[47] *Wikipedia: Wireless sensor networks*, `http://en.wikipedia.org/wiki/Wireless_sensor_network`.

[48] *Argo - global ocean sensor network*, http://www.argo.ucsd.edu/.

[49] *29 palms: Tracking vehicles with a uav-delivered sensor network*, http://www.argo.ucsd.edu/.

[50] *Smart dust: Autonomous sensing and communication in a cubic millimeter*, http://robotics.eecs.berkeley.edu/~pister/SmartDust/.

[51] *Mica2 datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

[52] *Mica2dot datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2DOT_Datasheet.pdf.

[53] *Micaz datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[54] *Iris datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/IRIS_Datasheet.pdf.

[55] *Telos datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[56] *Btnode rev3 hardware reference*, http://www.btnode.ethz.ch/Documentation/BTnodeRev3HardwareReference.

[57] *Imote2 datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf.

[58] *mupart014x ilmt preliminary datasheet*, http://particle.teco.edu/upart/datasheets/upart014xilmt.pdf.

[59] *Shimmer hardware guide*, http://www.snm.ethz.ch/pub/uploads/Projects/SHIMMER_HWGuide_REV1P3.pdf.

[60] *Snow 5: A modular platform for sophisticated real-time wireless sensor networking*, http://www.snm.ethz.ch/pub/uploads/Projects/SNoW5%20documentation.pdf.

[61] *Sun small programmable object technology (sun spot) theory of operation*, http://sunspotworld.com/docs/Red/SunSPOT-TheoryOfOperation.pdf.

[62] *Tinynode 584 embedded wireless network node*, http://www.tinynode.com/uploads/media/SH-TN584-103.pdf.

[63] *Wikipedia: Energy densities table*, http://en.wikipedia.org/wiki/Energy_density#True_energy_densities.

[64] *Ni-mh rechargeable batteries*, http://www.duracell.com/OEM/Pdf/others/TECHBULL.pdf.

[65] *Nickel-metal hydride application manual*, http://data.energizer.com/PDFs/nickelmetalhydride_appman.pdf.

[66] *The effect of phev and hev duty cycles on battery and battery pack performance*, http://www.pluginhighway.ca/PHEV2007/proceedings/PluginHwy_PHEV2007_PaperReviewed_Valoen.pdf.

[67] *Electric double-layer capacitor*, http://en.wikipedia.org/wiki/Supercapacitor.

[68] *Nesscap ultracapacitor products*, http://www.nesscap.com/data_nesscap/spec_sheets/Spec%2003.pdf.

[69] *Mote in network programming user reference*, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[70] *Deluge 2.0 - tinyos network programming*, http://www.cs.berkeley.edu/~jwhui/deluge/deluge-manual.pdf.

[71] *Ieee standard 802.15.4-2006*, http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf.

[72] *Zigbee alliance*, http://www.zigbee.org.

[73] *Msp430x4xx family user's guide*, http://focus.ti.com/lit/ug/slau056i/slau056i.pdf.

[74] *Tms320 family data manual*, http://focus.ti.com/lit/ds/symlink/tms320c2801.pdf.

[75] *Stm8l programming manual*, http://www.st.com/stonline/products/literature/pm/15433.pdf.

[76] *St62 in-circuit programming*, http://www.st.com/stonline/books/pdf/docs/2495.pdf.

[77] *On-board programming methods for xflash and hdflash st7 mcus*, http://www.st.com/stonline/products/literature/an/9056.pdf.

[78] *How to program/reprogram the st10f269zx flash memory*, http://www.st.com/stonline/books/pdf/docs/14336.pdf.

[79] *Pic18f1220/1320 data sheet*, http://ww1.microchip.com/downloads/en/DeviceDoc/39605F.pdf.

[80] *dspic33f/pic24h flash programming specification*, http://ww1.microchip.com/downloads/en/DeviceDoc/70152G.pdf.

[81] *Pic32mx family reference manual*, http://ww1.microchip.com/downloads/en/DeviceDoc/61132B.pdf.

[82] *Hcs08 family reference manual*, http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf.

[83] *Mc68hc812a4 data sheet*, http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC68C812A4.pdf.

[84] *M68000 8-/16-/32-bit microprocessors users manual*, http://www.freescale.com/files/32bit/doc/ref_manual/MC68000UM.pdf.

[85] *Atmega128 data sheet*, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.

[86] *Avr32 architecture document*, http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf.

[87] *At32uc3b complete datasheet*, http://www.atmel.com/dyn/resources/prod_documents/doc32059.pdf.

[88] *Max1674/max1675/max1676 full data sheet*, http://datasheets.maxim-ic.com/en/ds/MAX1674-MAX1676.pdf.

[89] *Cc2420 2.4 ghz ieee 802.15.4 / zigbee-ready rf transceiver*, http://focus.ti.com/lit/ds/symlink/cc2420.pdf.

[90] *At86rf230 low power 2.4 ghz transceiver for zigbee, ieee 802.15.4, 6lowpan, rf4ce and ism applications*, http://www.atmel.com/dyn/resources/prod_documents/doc5131.pdf.

[91] *Cc2520 datasheet 2.4 ghz ieee 802.15.4/zigbee rf transceiver*, http://focus.ti.com/lit/ds/symlink/cc2520.pdf.

[92] *Application note an058: Antenna selection guide*, http://www.ti.com/litv/pdf/swra161a.

[93] *Rohs compliance in the eu*, http://www.rohs.eu/english/index.html.

[94] *Gcc, the gnu compiler collection*, http://gcc.gnu.org/.

[95] *Avr32 uc3b software framework 1.4.0*, http://www.atmel.com/dyn/resources/prod_documents/AVR32-SoftwareFramework-AT32UC3B-1.4.0.zip.

[96] *Temt6000x01 ambient light sensor*, http://www.vishay.com/doc?81579.

[97] *Doxygen: Source code documentation generator tool*, http://www.doxygen.org/.

[98] *Ntc thermistors data sheet*, http://www.murata.com/catalog/r44e11.pdf.