AN OSGI BASED FRAMEWORK FOR AN INTELLIGENT SENSOR NETWORK

By

JAMES A. RUSSO

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2004

To my wonderful wife, Julie

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Abdelsalam (Sumi) Helal, for his enthusiasm, encouragement, guidance and support. I also thank Daniel Nieten for the work he did on the initial software framework. I would also like to deeply thank Steven Vanderploeg, who put forth considerable effort to help design, lay out and assemble the hardware.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN OSGI BASED FRAMEWORK FOR AN INTELLIGENT SENSOR NETWORK

By

James A. Russo

December 2004

Chair: Abdelsalam (Sumi) Helal
Major Department: Computer and Information Science and Engineering.

Sensor networks are currently being used in a wide range of sensing and actuation applications. A sensor network consists of small embedded devices which are used to sense and interact with their environment for long periods of time with minimal user interaction. The network typically reports its findings back to a sensor gateway where the data can be stored and analyzed. Such a system could be used to measure environmental conditions in a greenhouse, or the mechanical vibrations taking place in a large suspension bridge.

This thesis describes the software and hardware architecture of an intelligent sensor network. Both the physical sensor nodes and extensible software architecture are developed. The hardware uses a stack concept allowing the individual layers to be replaced allowing the node to be customized to the specific application. A cooperative operating system is also designed for the sensor node, allowing multiple processes to make use of the CPU in a consistent manner.

The software framework is based on the well defined OSGi service framework. The sensor nodes store their own driver which is automatically downloaded into the framework when the sensor node is first discovered. This surrogate concept allows great flexibility in the configuration of the network, removing the need to update any software on the sensor gateway itself.

CHAPTER 1
INTRODUCTION

In recent years, computers have become more and more a part of our everyday life. Pervasive computing is about tightly integrating computers with their human operators, whereby the operators do not even need to know about the computers existence, the computer seamlessly and unobtrusively can help the user with various types of tasks. Computers should become a tool, and therefore not be the focus of the attention when performing a task. The concept of ubiquitous computing describes a time where computers will be everywhere and the PCs of the modern day will no longer have a purpose. This will be because computers will be everywhere, in the walls, in clothing, within a simple piece of paper on a desk [1].

In order to some day achieve the vision of pervasive and ubiquitous computing there needs to be the ability for computers to sense and interact with their surroundings. Sensor integration can become cumbersome due to the various types of sensors involved and the various software and hardware interfaces involved In recent years there has been enormous progress made in the design of low-cost, low-power microcontrollers which now enable the design of communicating, autonomous devices previously not possible. One such device is the wireless sensor network, which consists of numerous low-cost, autonomous nodes which provide sensor readings and possibly actuate devices. The applications of such devices are wide ranging and everything from home automation to military applications are currently being researched.

What is presented in this thesis is the design for a wireless sensor network which seamlessly integrates with the Open Services Gateway Initiative (OSGi) framework, allowing other OSGi applications to benefit from the information the sensor network has to offer. OSGi is a service framework providing an environment where applications can make use of transient services. A transient service is a service provided by an application which may appear after the application wishing to use it is started.

Motivation

The motivation for this project was to design both hardware and software for a flexible, easy to use wireless sensor network platform. This project was designed for the Smart House environment housed in the Pervasive Computing laboratory, but its use is not restricted to this application. The concept is very flexible, lending itself to use in many different types of applications. In the Smart House, there are many different applications which make use of physical information provided by sensors. This sensor network framework is designed to provide a uniform and consistent way to implement and deploy various sensors within the home. Various sensor boards would be designed and plugged into the processor and networking board to allow a flexible and fully customizable solution.

Examples

One example of a wireless sensor network would be the environmental monitoring of a home. Temperature and Humidity sensors would be located throughout the house and this information would be logged for historical reasons as well as possibly providing alerts or activating some type of environment control system. In a typical non-wireless sensor network scenario one would hard-wire the sensors to each room in the home, connect these sensors to a computer somehow, and then have to program which “port” on the computer correlates to which sensor located in the home. Further, one would need to specify which port contains which type of sensor, either humidity or temperature. This quickly becomes an unwieldy and complex system.

Using the technology discussed in this paper, one would use a wireless sensor node for each room, connect both a humidity sensor and temperature sensor to each node, and program the firmware for each node. In addition to the firmware, the driver which decodes the data as temperature and humidity readings would be contained on the node. One would then install the sensor gateway software and connect the base sensor node to a PC, and finally turn on the nodes. There would be no configuration of the PC beyond installing the software and connecting it to the base sensor node. The drivers will be automatically downloaded from each detected sensor node and installed into the PC.

The system is scalable to hundreds of sensor nodes by use of an 8-bit node identifier, which is easily expandable to multi-byte allowing even mode addressable nodes. With an 8-bit node identifier you can have up to 254 nodes, with one node as the sensor gateway and one address used as a broadcast address (255). The only scalability issue imagined would be that of access to the communication medium. However, even that is easily overcome with the replaceable nature of the communication module. An Ethernet module could be developed to allow for near limitless nodes using lightweight UDP messages for communication.

The system is open and upgradeable due to its reliance on the OSGi framework for the sensor gateway. The bundles created for the sensor network are easily portable to any existing OSGi framework and can be incorporated into existing OSGi deployments. Since the system is written in Java, the underlying machine architecture is not relevant. The system will run on any hardware which implements the Java virtual machine. The ability for the sensor nodes to store their driver information either on chip or accessible via a URL allows the system to be configured and easily upgraded.

The organization of this thesis is as follows. In Chapter 2, we discuss the background material for wireless sensor network and discuss other research currently being performed. In Chapter 3 we discuss the problem and the approach taken in finding a solution. In Chapter 4 we discuss the hardware used for creation of the sensor node, In Chapter 5 we discuss the software architecture used. In Chapter 6 we discuss information on design considerations when developing a sensor node using the platform. Finally, Chapter 7 provides conclusion and directions in future work.

CHAPTER 2
BACKGROUND AND RELATED RESEARCH

In this chapter we present some of the background on wireless sensor networks, and define some terms which will be used throughout this thesis.

Definitions

A sensor node is an autonomous embedded device which has the ability to collaborate with other devices and ultimately provides services to an application. Those services could be in the form of data readings, allowing the actuation of some device, or both. A sensor node is typically battery powered, but this is not a strict requirement. The node normally communicates with the other sensor nodes, and with a sensor network gateway by means of a wireless transceiver, but wireless is also not a strict requirement. Other communication mechanisms are available including Ethernet, and simple serial bus based communication including RS232 and RS485. Basically a sensor node combines sensing, processing and communication into a tiny embedded device.

A sensor network is comprised of two or more sensor nodes working together to gather information and provide one or more services to an application. The sensor network is simply a logical grouping of sensor nodes. A single sensor node can comprise a sensor network, but in a typical deployment there are multiple nodes. Data and communication between the sensor network gateway and the sensor nodes is transmitted along the sensor network. The sensor network is not a physical medium like RF, or a wire, but rather a logical instrument.

A sensor network gateway, could simply be a specially designated node, but typically is a sensor node connected to a higher class computer like a PDA or a standard desktop PC. Depending on the communication mechanism being used, a sensor node may be acting as the communication device for the sensor network gateway. This might be done because the embedded processor on the sensor node is needed to interact with the communication hardware. This would not be the case if the communication channel was something on which the computer could already communicate. When a sensor node is used in this function, it does not do any processing on the incoming or outgoing data, the packets are blindly passed to the host computer for processing. An example where the sensor node would be used on a PC gateway is in RF communication, and one in which it would not be use would be sending UDP packets over Ethernet.

In some sensor networks the sensor network gateway branches the sensor network and some larger network such as the Internet. This enables applications to directly communicate with the sensor nodes and can allow more complex applications to be developed. A sensor network gateway proves to be a critical link between the sensor network and the traditional networking infrastructures including Ethernet, the 802.11 communication standards, and other wide area networks [2].

Related Research

Currently there is a great deal of research being performed on wireless sensor network topics. University of California at Berkeley has been performing research on various aspects of sensor networks, including the development of a series of sensor nodes called motes. Crossbow is a company which now commercially distributes the Motes designed at UC Berkeley [2]. Intel research has also been involved with the Berkeley research and the development of the Imote 1.0, a Bluetooth enabled wireless sensor node [3].

Also designed at UC Berkeley is TinyOS [4], a component based operating system designed for wireless embedded sensor networks. The TinyOS platform allows rapid innovation and development while minimizing code size as is required on the small sensor node devices. With TinyOS it is possible to take different components and 'wire' them together at a software architecture level to build your complete sensor network platform. Various pre-built components are available and provide message transmission, routing, sensor reading, among other services. The TinyOS operating system uses a C like derivative language called nesC.

With the emergence of the Motes hardware, the TinyOS and a stable platform on which to develop, research has turned to sensor node applications.

Alan Mainwaring et al. [5] describe the use of wireless sensor network in monitoring the habitat of nesting birds. A wireless sensor network is well suited for this type of application as it is capable of monitoring the burrows without disturbing the animal. The sensor node is able to collect localized measurement over long periods of time. This type of research and data collection was not previously possible.

Other research has been taking place on the topic of query processing, or obtaining data from the sensor network. A sensor network has the potential to generate large amounts of continuously flowing data. This data represents the evolving status of systems and must be easily queried. Several query processing systems have been developed including Directed Diffusion [6], TinyDB [7], and Cougar [8]. Basically, query processing systems are languages and methodologies which enable the end application to query and obtain information from the sensor network in a form which is most easily used by that application. In TinyDB, users specify SQL-like queries which reflect their data processing and acquisition needs. The queries specify the readings needed, the subset of nodes from which the readings are to be obtained from along with simple transformations over the data. The queries are designed in a PC and injected into the network. The query must then be processed within the sensor network in an energy efficient manner and the results of that query returned to the originating location. Directed Diffusion does not describe a query language and instead focuses on the query routing mechanisms and flexible in-network processing. The in-network diffusion allows the sensor network application developers to design their own in-network processing primitives, which is more flexible compared to TinyDB where the in-network processing is limited to the pre-defined functions.

Having the potential of thousands of wireless sensor nodes spread across the areas and interacting and sensing people, privacy and security has become another important area of research. Purposed applications of wireless sensor networks include many applications where security of the network will be essential. The security techniques applied to normal networks cannot typically be applied to sensor networks due to their unique requirements to be economically viable. Research in this field includes design of key-management schemes [9], and the design of secure routing algorithms [10]. Other security aspects such as the protection from eavesdropping, tampering, traffic analysis and denial of servers still require much research.

With events and data being generated by a wireless sensor network, time synchronization becomes and important issue as data generated typically needs a global timestamp. Having a globally synchronized clock can also provide more effective power management and media access for communication [11].

Several groups [11,12] have developed time synchronization schemes which aim to synchronize the time on single and multi-hop wireless sensor networks.

Berkeley Motes

The University of California at Berkeley has been the leader in the research and design of wireless sensor networks. They have designed and implemented a series of wireless sensor nodes called Motes [4]. The WeC mote [13,14] was the first to be designed using an AT90LS8535 microprocessor. It operated at 4 MHz, contained 8KB of program memory and 512B of RAM. The Rene [14] was the next node to be designed, changing the form factor from a circular dot to a rectangle about the size of 2 AA batteries. The Rene node incorporated a higher-end processor, the ATMega163. Like the Rene it operated at 4 MHz, but contained 2 times the amount of memory with 16KB of program memory and 1KB of RAM. The MICA and the MICA-2 nodes [14] contained the more powerful ATMEGA128 processor with 128KB of program memory and 4KB of RAM. The MICA and MICA-2 enabled the developed of TinyOS, an event and component based operating system for the sensor nodes. The MICA-2 also incorporated a new RF radio, the ChipCon CC1000 [15] which removed the need for the ATMEL processor to handle many of the low-level radio functions. The MICA-2 was also later put into a “dot” form factor and called the Mica2Dot.

CHAPTER 3
THE PROBLEM AND OUR APPROACH

In this chapter we discuss the problems with existing wireless sensor network systems and our approach solving these problems.

The problem addressed by this thesis is to create a wireless sensor network system where the services provided by the sensor nodes can easily be used by various pervasive computing applications. Currently, there is a lack of integration between wireless sensor networks and applications. Many products exist to obtain and log readings from the sensor networks, but few allow applications to easily integrate and benefit from the services which the wireless sensor nodes offer [16]. By creating a sensor network framework and middleware, we can solve this problem by providing a consistent programming model which will be easy for programmers and system developers of pervasive environments to utilize. The need exists to externalize and abstract the unfamiliar concept of sensors and the sensor network to the pervasive application and bring the data and services provided by the sensor network within easy reach.

The programming model of a sensor network framework should be easy to deploy and make use of existing technology where possible. When a new sensor is added, no configuration should be required on the sensor network gateway. Applications should be able to immediately and automatically discover and make use of the new sensor nodes. Services provided by a sensor node could be simple data such as a sensor reading, actuation such as the operating of a device, or both. We surmise that the value or benefit of a wireless sensor network is increased when the number of applications benefiting from its services increases. Our solution is to create a sensor network framework which allows applications to easily make use of the services which a wireless sensor network provides, therefore increasing the sensor network value.

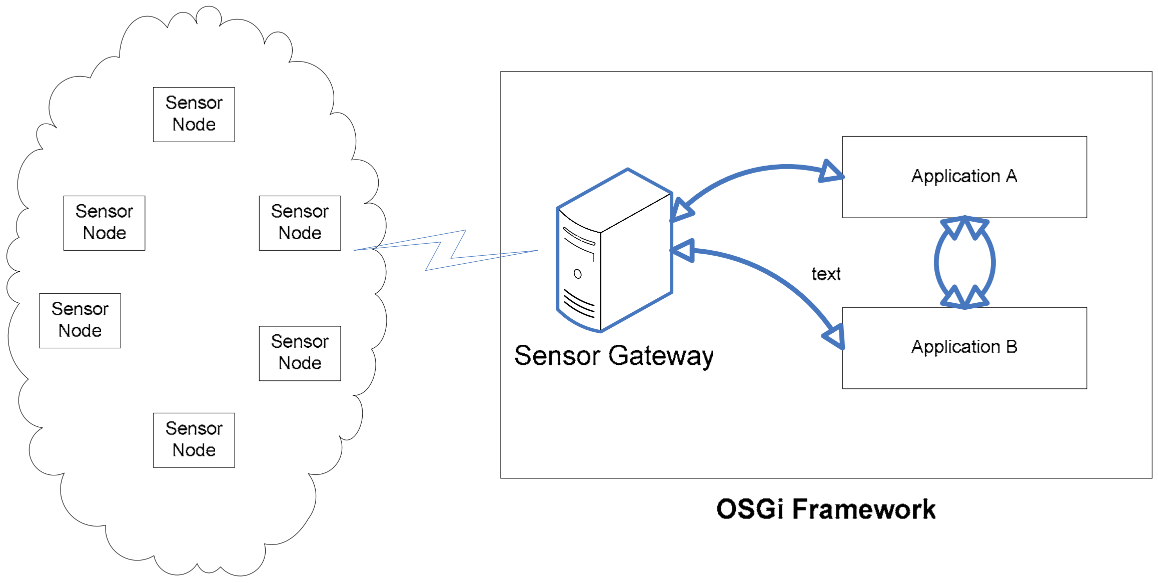


Figure 3-1. Diagram of our sensor network architecture

Our approach designs and implements a wireless sensor framework which is both easy to deploy and also easy for applications to use. We designed not only the software framework, but also the sensor platform hardware and by working on both we were able to tailor each and come up with a system which solves the problem of application and sensor network integration. On the hardware side, we use standard commercial off-the-shelf (COTS) components to ensure that the design is easily adoptable and financially feasible. On the software side, we have created a sensor gateway framework based on proven existing technologies and standards such as OSGi and Java.

OSGi is a standardized component oriented computing environment for networked services [17]. The OSGi framework allows applications to be developed and the life cycle of these applications managed from anywhere on the network. The software components installed into the OSGi framework can be libraries or applications which automatically discover and make use of other available services. This dynamic component model is a great fit for a wireless sensor network, especially in a pervasive computing environment. As new sensor nodes are brought online, a pervasive computing application running within the OSGi framework can automatically discover and make use of their services. By allowing the applications to dynamically discover new sensors, they can better understand their environment and the context in which they are operating. The OSGi framework is written in Java, which makes Java the software of choice for our sensor network architecture. The entire sensor network framework is written in Java, with the exception of the sensor node operating system and firmware which is written in C.

We designed a surrogate architecture where the sensor node driver on the sensor gateway which is used to interact with the sensor node can be stored on the sensor node itself. Standard low-level functionality exists on the sensor node and the sensor gateway software to enable the gateway to automatically discover, download and install the required software to communicate with the sensor node. This sensor node driver contains an interface which can be used by other applications to interact with the sensor node. The software contains the instructions which make sense of the sensor data and provide it to other applications which are running within the OSGi framework. If the OSGi bundle containing the sensor node is too large to be stored on the sensor node itself, it is possible to just store a URL to where the gateway software can be downloaded. The referral URL can use any protocol which the sensor gateway server has access to, such as http and ftp. Size might not be the only reason why the referral concept would be used. By having the software on a web server, upgrading the sensors is as easy as replacing the software and sending a reset command. All the required downloading and installation of the gateway software for the individual nodes is performed by the sensor gateway bundles installed in the framework.

CHAPTER 4
SENSOR NODE HARDWARE

In this chapter we discuss the hardware architecture used for the sensor network platform, specially the sensor nodes. We first discuss the overall design goals, and then go through each module in the wireless sensor node.

The design goals for the sensor network hardware were to create a flexible hardware platform capable of handling a wide array of sensor network applications. The board was to be small, run on a wide range of power sources, and have onboard EEPROM storage for the actual configuration data. While some work was done on power efficiency and size, this was not an overwhelming design goal as other groups are leading the way in this research and their solutions could easily be implemented on the platform which we are developing [4, 18].

The sensor node hardware design is flexible in that it makes use of stackable modules, where each module provides certain functionality to the node. A typical sensor node deployment might contain 3 modules: a processor module, a communication module and finally a sensor module. The processor module would contain the actual microcontroller, a power supply and possibly external EEPROM memory. The power supply is not required to be part of the processor module. If a different power supply was to be developed, it could easily just become another module within the module stack. The communication module could contain an RF radio or some other communication device such as a Bluetooth radio, Ethernet or even infrared if the application environment could support it. Finally, the sensor module would contain the actual sensors and interface circuitry allowing the interface of the sensor hardware to the microcontroller. The sensor module would be changed for each sensor node application, where the other layers are independent of the actual sensor being used.

Figure 4-1 shows a block diagram of the typical node configuration. The arrows between the nodes are a standard bus-type connection existing on all the boards. This bus allows all the various ports and power connections to connect from board-to-board.

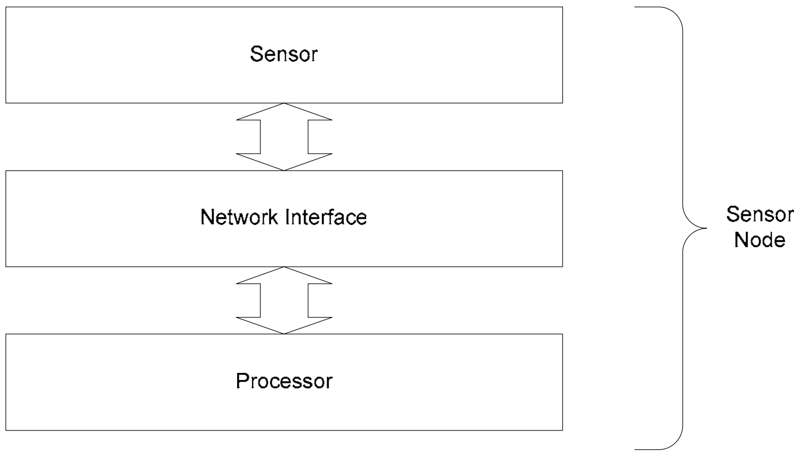


Figure 4-1. Block diagram of the typical hardware node configuration

Processor Module

The process module used contains an ATMEGA128L processor from Atmel technologies [19]. This processor was chosen due to its rich feature set and wide range of open source development tools, such as the open source C compiler GCC [20,21]. By making use of these tools, applications can be developed in C using the same tool chain used to develop standard UNIX applications. The processor contains 128KB of self-programmable flash memory for the application, 4KB of onboard EEPROM storage and an 8 channel 10bit A/D converter. The 10bit A/D converter is especially important as many simple sensors output values as analog voltage directly related to the physical phenomena being sensed. This can simplify the sensor circuitry and remove the need for an external analog-to-digital converter and voltage reference. The CPU has a maximum clock speed of 8 MHz, providing the ability to process 8 million instructions per second. [18] Also contained on the processor board is an RS232 line level converter which produces the standard 12 volt RS232 levels from the processors 3.3 volt serial outputs on the CPU. This chip is needed so that the processor can communicate with a standard PC serial port. An additional 1Mbit EEPROM is also included on the processor allowing applications to take long-term readings which will persist even with the loss of power. The final component on the processor board is the power supply chip. The supply power for all the boards in the stack is provided by the power supply on the processor module. The MAX710 from Maxim technologies was used allowing the use of a wide-range of source voltages. The MAX710 also has a unique feature of being able to make use of a battery over its full lifespan, regulating the voltage for new batteries and using a charge pump for older near-depleted batteries. The MAX710 is configured to produce the 3.3V required for the processor and other components to operate. Figure 4-2 shows a picture of the processor module.

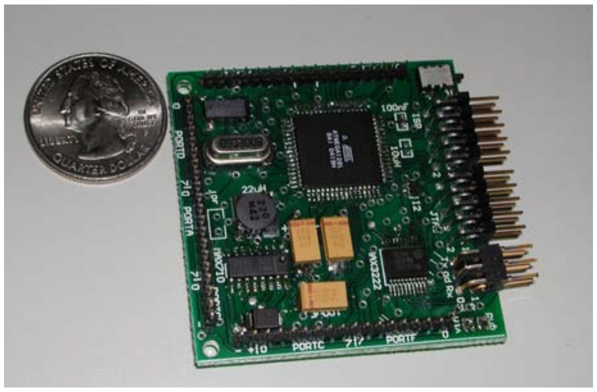


Figure 4-2. Image of the sensor processor module

The Atmel processor used on the processor board contains 4 KB of persistent EEPROM storage. In addition to this on-processor storage, the processor board contains a 1 Megabit serially addressable EEPROM. This additional EEPROM can be used to store the downloadable bundle which is used by the server framework to interact with this sensor node. By using this external EERPOM chip, the 4Kb of onboard EEPROM can be reserved for storing application specific data such as sensor readings or configuration data.

Communication Module

The communication module allows the transmission and reception of packets with the outside world. Since this module is just one of many possible modules which makeup a sensor node, the actual communication mechanism can be changed. This allows various communication modules depending on the need of the sensor network. Any device which is capable or receiving and transmitting data could act as a communication module. The only changes required would be to the communication driver in the sensor node operating system. A standard packet format is used for all communication, so minimal software changes would be required to introduce a new communication mechanism.

One of the communication boards currently developed is an RF communication board, using the TR1000 transceiver from RF Monolithics [22]. This small (7mm x 10mm) device contains all the RF related components required to allow half-duplex short-range wireless communications. The device operates in the 916.5 MHz ISM band, and is therefore free from any type of license or air time fee. The microcontroller operates the radio by using two control lines to place the radio into transmit/receive mode, and then two additional lines for data transmit and receive. The radio is powered by 3.3V and requires just a few external components including an antenna. The radio has various modes of operation one of which is a low-current sleep mode. This low current sleep mode reduces the power consumption to .7uA. The power consumption is 12mAwhile transmitting, and 1.8mA which receiving [22].

Other communication modules which are under development include an Ethernet module which would be capable of transmitting the sensor network packets via UDP over a standard Ethernet network, or even wireless 802.11 using an 802.11 Ethernet bridge device.

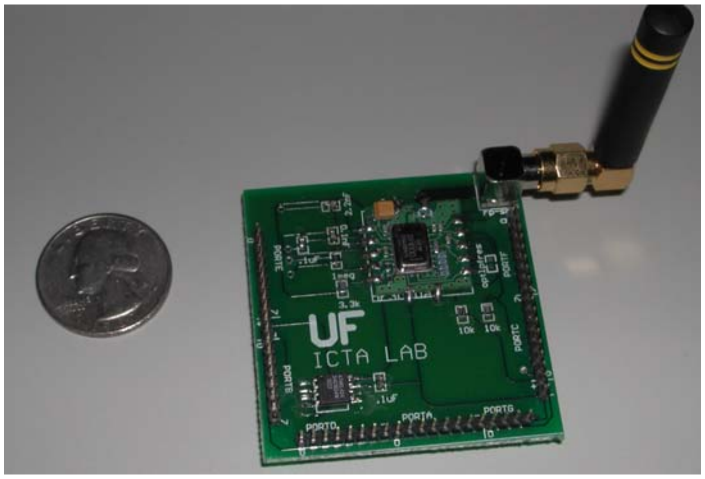


Figure 4-3. A picture of the radio communication module currently developed

Sensor Module

The sensor module contains the physical sensor and any other circuitry required to interface the sensor with the processor board. Different sensor boards would be created depending on the specific application and a single sensor node could possibly have multiple sensor boards all working together. In addition to the specific sensor boards, prototyping boards have been created allowing the quick implementation of the various sensors before manufacturing of a final sensor module. Communication with the processor module is done via the pins where the sensor module plugs into the module stack. The majority of the pins on the Mega128L processor are contained within those pin headers. The communication mechanisms would include SPI, serial, and also analog voltages via the onboard 8 channel 10-bit ADC converter.

The sensor board could be as simple as providing a physical connection and isolation to the digital input and outputs on the processor (to monitor switches, etc), or as complex as the Sensor Board having its own processor which communicates with the node processor via some inter-chip communication mechanism such as I2C. Figure 4-4Figure shows a sensor board which is designed to allow quick prototyping. The board contains a large prototyping area so that sensor components can be connected to the processor using either point-to-point solder or wire wrapping techniques. Once a sensor design is tested, a schematic would be designed and sent to a PC board house for a final board to be created.

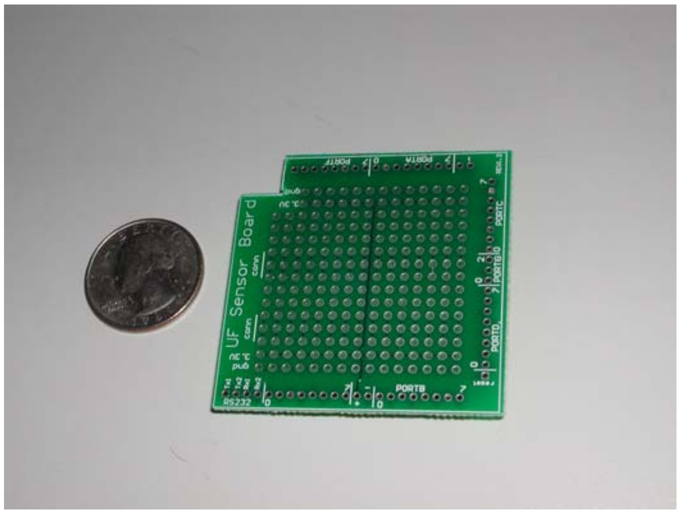


Figure 4-4. A picture of a prototype sensor module

Debug and Programming Module

While not required for operation of a sensor node, a debug and programming module has been designed allowing he module to easily be connected to a host computer for debugging, or special applications. The module brings the two serial ports to standard

DB9 connectors. This enables the connection of the module to a standard PC when programming and building the software which would run on the sensor nodes. Using a serial port when debugging the firmware can be very helpful.

In addition to the DB9 connectors, the debug and programming module also includes four push button switches and eight LED lights. A standard power connector is also provided for easy connection to a standard wall style power supply transformer. Headers for the JTAG port, used for programming and real-time debugging, are also accessible on the module. Figure 4-5 shows a picture of the debug and programming module.

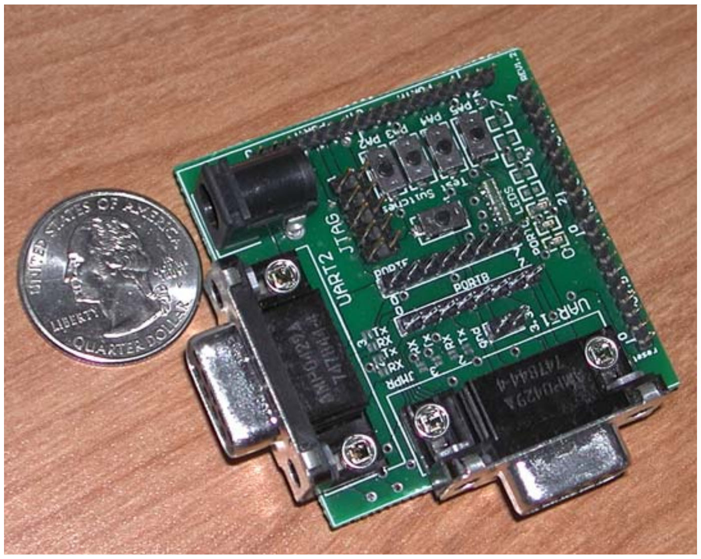


Figure 4-5. A picture of debug and programming module

CHAPTER 5
SENSOR PLATFORM SOFTWARE ARCHITECTURE

Design Goals

The design goals for the sensor platform software architecture are

- Ease of Deployment
- Scalability
- Understandable Programming Model
- Configurability and Extensibility

General Operation

A sensor network consists of a sensor gateway and a number of sensor nodes. Sensor nodes can communicate with each other, but typically the data contained and measured from within a sensor network propagates to the sensor gateway where it is recorded or used by other applications. In our system, the sensor gateway does not need to be preconfigured for a specific type of sensor node. The sensor node themselves contain the driver and associated software used to decode sensor data and interact with the individual sensor node.

A generic packet format is used allowing 16 bytes of application specific payload data. The payload area is application specific and the content of the payload is up to the designer of the sensor node software. For example, a temperature sensor might have byte 1 of the payload be the temperature in Celsius, whereas a light sensor might use a 32 bit value for the intensity of light and place that value in payload locations 0 through 3. It is up to the sensor process on the sensor node and the driver which is downloaded and installed into the sensor gateway to speak the same language and be written to agree on the positions and meaning of any data contained within the payload. Figure 5-1 below shows the format of the packet data used for communication within the sensor network.

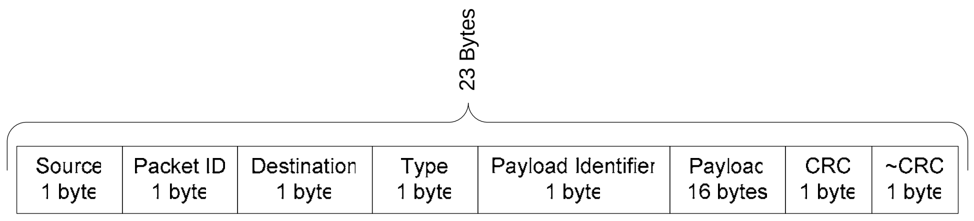


Figure 5-1. The format of the packet used within the sensor network

The source field of the packet is the node identifier which is sending the packet. The packet identifier field is an increasing number which is generated for each new packet generated on a specific source; it is used to uniquely identifier packets within a short period of time. The destination field is used to identify the destination node of the packet, typically the base station is node 0, where a broadcast address for the network is node identifier 255. The type field indicates the various packet type, all the possible types and their descriptions are listed in Table 5-1. The payload identifier is used to specify the payload type, and its value is dependent on the packet type. One example use of the payload identifier is during the EEPROM download, where this location is used to specify the block being sent during an EEPROM_BLOCK_RESPONSE. Without this field the full 16 bytes of payload would not have been able to be used during the EEPROM download process. The payload is 16 bytes of application data which is possibly used depending on the packet type. Finally, two CRC bytes are included to be able to quickly validate a message. The first CRC byte is simply an XOR of all the values in the packet, while the 2nd CRC byte is just the complement of the first.

Table 5-1. Table showing all the packet types and their descriptions

Packet Type	***Description***
ALIVE	Packet set by sensor nodes when they first come online.
DEAD	Packet sent by sensor node when it is going offline.
PING	Packet used to determine if a sensor node is alive. Responds with PONG.
PONG	Packet used in response to PING
ACK	Packet used for general acknowledgement.
NAK	Packet used for general negative acknowledgement.
EEPROM_READ_BLOCK_REQUEST	Packet used to requests a block from the EEPROM.
EEPROM_READ_BLOCK_RESPONSE	Packet used in response to the read block request containing the EEPROM data.
EEPROM_DETAILS_REQUEST	Packet used to request the EEPROM details from the EEPROM.
EXECUTE_SENSOR_ROUTINE	Packet used to request that the sensor routine on the sensor node be executed.
SENSOR_DATA	Packet from the sensor node with the sensor data.
RESET	Packet to the sensor node requesting that it reboot itself.

Figure 5-2 shows some pseudo code which is used for the checksum calculations. By using the two checksum bytes which are complements of another, a quick determination can be done on the validity of the packet by just comparing the two checksum bytes and if invalid the CPU cycles needed to calculate the packets actual checksum value can be saved. The driver software is stored within the EEPROM on the sensor node itself. The EEPROM on the Atmel processor is 4096 bytes in size and split into 16 byte blocks. The first block of the EEPROM is reserved for the operating system use and contains the actual node identifier for the node at location 0. The second block within the EEPROM contains information on the driver software which is located in the remaining blocks.

```
for (i=0..20)
        packet[CRC1] ^= packet[i];

packet[CKSUM2] = ~packet[CKSUM1]
```

Figure 5-2. Pseudo code for the checksum generation

Contained in this EEPROM information block include the number of data blocks which make up the driver software as well as the type of data. There are two possible types of data which can be stored in the EEPROM. The first type is an actual OSGi bundle, while the second type is simply a string containing the URL to where the bundle can be located. Figure 5-3 shows the EEPROM storage configuration.

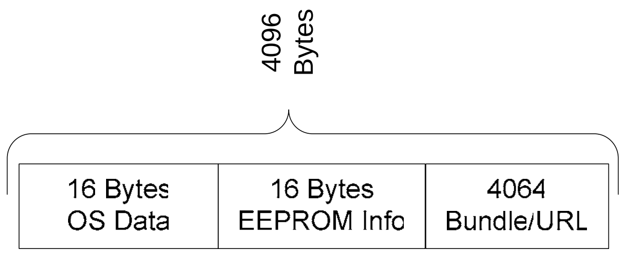


Figure 5-3. EEPROM storage configuration.

The driver software is downloaded during the initial negotiation when a sensor is first brought online. Figure 5-4 shows the protocol flow of messages between both the sensor gateway and the sensor node. You can see that the first message will be an ALIVE message when the sensor node first comes online. The ALIVE message will be repeated until finally acknowledged from the sensor gateway. Upon receiving the ALIVE message, the sensor gateway will ask the sensor node for details regarding the EEPROM image. These details will include the type of the EEPROM contents, either a referral or the actual data and the size of the data. Once this data is obtained, the actual EEPROM is downloaded from the sensor node using a sequence of EEPROM_BLOCK_REQUEST messages, to which the sensor node responds back with EEPROM_BLOCK_RESPONSE messages containing the actual data. If the EEPROM details revealed that the data is actually a referral, the resource referenced by that URL is downloaded and installed into the OSGi framework. If the data contains the actual driver that data stream is treated as an OSGi bundle and installed into the framework.

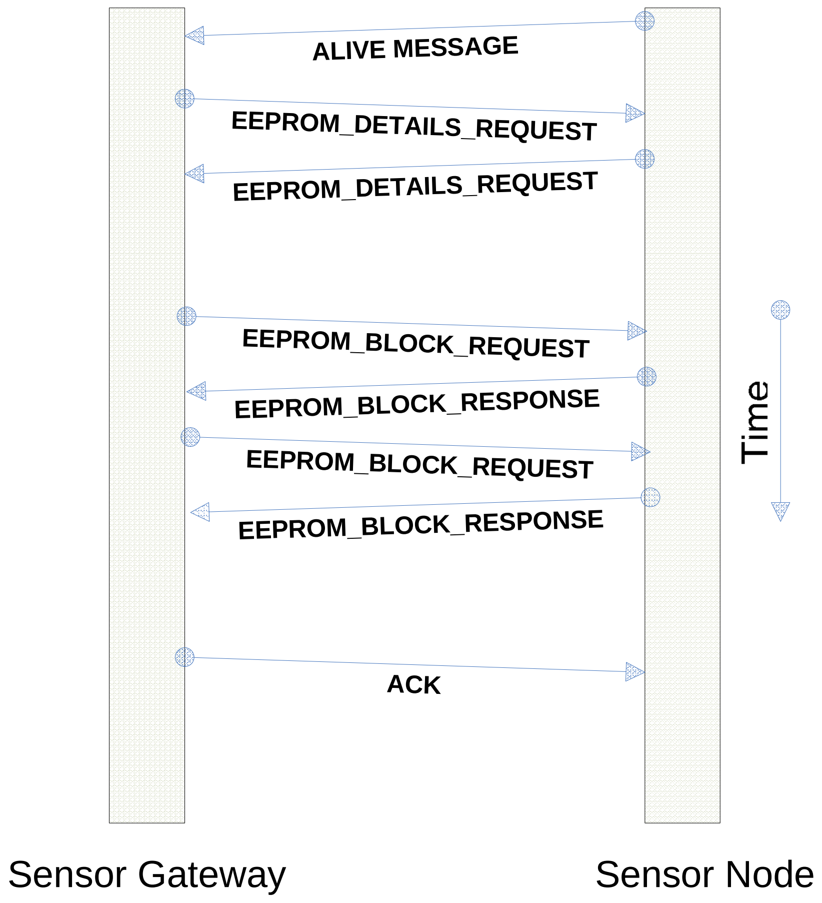


Figure 5-4. The flow of messages during the startup of a sensor on the network

OSGi Architecture and Background

OSGi is the service orientated framework on which the sensor gateway is built. The Open Services Gateway Initiative was founded in March 1999 and its purpose was to create open specifications for the network delivery of managed services to local networks and devices. It aims to enable an entirely new category of smart devices due to its flexible and managed deployment of services. In OSGi, all applications are made up of services delivered in convenient containers called bundles. The framework allows the installation and configuration of these bundles after the framework is deployed and fully operational. The programming model is such that the arrival and departure of services within the framework is handled in a consistent manner allowing services to depend on other services which are not yet available or gracefully handle the departure of reconfiguration of service [16].

The OSGi specification is complete and discusses the features and functionality of OSGi in great detail. The specification provides support for native binary compatibility, remote management and configuration, device driver support, logging, security and many others. In our sensor network architecture we primarily focus on the core OSGi functionality relating to the installation of bundles and the creation of services included those bundles. Figure 5-5 shows the components of an OSGi bundle. A bundle is simply a Java archive file (jar) containing interfaces, implementations for those interfaces and a special Activator class. The manifest file contained within the jar file includes special OSGi specific headers which control how the bundle will be used within the framework. Data contained in the manifest file include which packages are exported and imported from the bundle, the bundle name, version and vendor and finally the full name of the Activator class which will be called when this bundle is started. A bundle does not need to contain an activator and this is commonly done when a bundle just needs to export a number of Java packages which will be later imported into other bundles within the framework. This type of bundle is referred to as a library bundle [16]. Figure 5-6 shows an example manifest file for an OSGi bundle. The Activator class within an OSGi bundle contains two methods that are called when the bundle is started or stopped within the framework. These hooks give the application programmer the ability to create objects which will represent services and register them with the framework. Typically there is a known interface for an object and then an implementation for that interface.

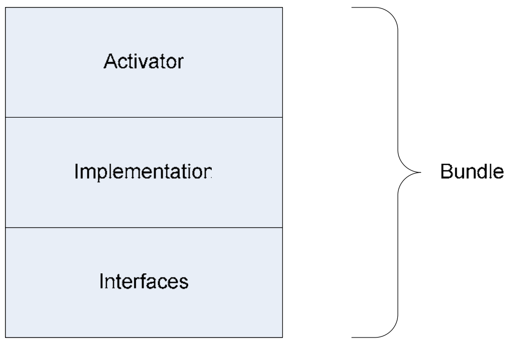


Figure 5-5. The contents of an OSGi bundle

```
Exported-Package: com.domain.package
Imported-Packet: com.domain.otherpackage
Bundle-Activator: com.domain.package.impl.Activator
Bundle-Version: 1.0
Bundle-Description: This is the bundle which
provides some functionality.
Bundle-Vendor: Some Company
Bundle-UpdateLocation: http://www.somelocation.com/
```

Figure 5-6. An example OSGi bundle manifest file

All services are registered within the framework using a string to identify the service. The standard convention is to use the name of the interface for the service name. Also associated with services are a set of key/value properties. Since there can be multiple services with the same name (interface), you can use the properties associated with that service to further refine a search for a specific service. An example of this could be an application which provides a spell check service. The service defines an interface with one function indicating whether the word is spelling correctly. Multiple implementations could exist for different spoken languages and a language property could be used to location the specific service object which matches your needs. The filter language when making a properties based query is the same as is used with the LDAP database system and quite powerful. Figure 5-7 show an example of how this spelling service might be obtained using the OSGi getService() call.

```
…

spellchecker =  bc.getService(“com.domain.spellcheck”,
          ”(Language=English)”);

spellchecker.check(“tomoto”);

…
```

Figure 5-7. Java code showing how to obtain a service with OSGi

Sensor Gateway Software

SensorNetworkDevice Bundle

The SensorNetworkDevice bundle is the lowest level bundle in the sensor network framework. It simply provides two methods, one for transmitting a packet on the network and one for receiving a packet from the network. It simply exists to abstract the real communication mechanism from the underlying communication hardware. For example, if using a Radio Module, the data would need to be transmitted via the RF and be first sent to a sensor node which is setup to be used as a base station. If the nodes are connected via a standard Ethernet network standard Java communication primitives can be used. The manifest of this bundle can contain a header which defines the implementation which will be used for the SensorNetwork service. Currently by default, the implementation is the NetworkPacketReader which simply uses UDP broadcast packets for the communication. This is the default as it is the communication mechanism which is used by the sensor node emulator. Any class which implements the PacketReader interface is able to be used as the underlying implementation for the SensorNetworkDevice OSGi service. Figure 5-8 shows the how the various layers used in the SensorNetwork bundle relates to one another.

SensorNetworkManager Bundle

The SensorNetworkManager bundle is the heart of the system and is the center of all communication within the sensor network. It provides methods for sending messages to any node on the network, as well as automatically discovering new nodes by the reception and handling of the ALIVE packets. When the SensorNetworkManager starts, it attempts to locate and obtain a service reference to a SensorNetwork service. The SensorNetwork service, as mentioned above, provides the primitive transmit and receive packet methods which are needed by the SensorNetworkManager. The SensorNetworkManager has a receiver thread which is always reading any available packets from the underlying SensorNetwork service and then processing those packets. The processing of those packets may consist of the creation and destruction of SensorNode objects or the dispatching received packets. Contained in the SensorNetworkManager bundle is a configurable keep-alive mechanism which can automatically send out PING packets to the sensors on the network and await their PONG reply. This can be used in situations where the nodes may no longer be responding and are to be removed from the system. A sensor node is also removed whenever a DEAD packet is received from that sensor node, but there are times when the sensor node is not able to send the DEAD packet when going offline. This can happen for instance when the battery fails or the node is somehow physically damaged. It is conceivable that a node could be programmed to monitor its own battery voltage and send the DEAD packet and shut itself down when the voltage falls below a safe operational level.

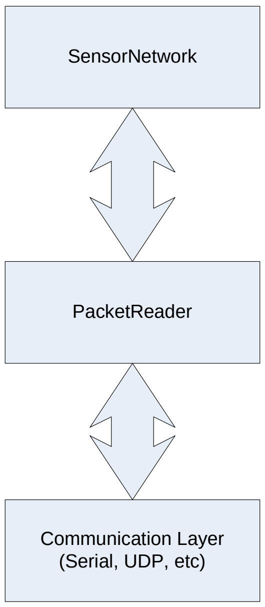


Figure 5-8. Figure showing the stack used in the SensorNetwork OSGi bundle

Within the OSGi specification there are provisions for Managed Services. Managed services are services which obtain configuration information from the framework itself. Configurations are stored within a special Configuration Manager bundle, and when a bundle is started, it can contact the configuration manager and obtain its configuration options. The SensorNetworkManager uses this configuration system for the configuration of the keep alive thread. The configuration is stored under the *edu.ufl.icta.sensors.drivers.sensornetworkmanager* configuration manager PID and can contain any of the fields found in Table 5-2.

SensorNode Service

The sensor node service is created in response to the reception of an ALIVE packet on the sensor network. When the sensor driver bundle is downloaded and installed, it does a query for the SensorNode service with the specified NodeId attribute. Once this service is obtained, it registers itself with the SensorNode to receive any packets which are sent from that node to the sensor gateway.

Table 5-2. Configuration variables for use in the sensor network manager

Field Name	***Type***	***Description***
keepAliveActive	Boolean	True if the keep alive thread is to be active.
keepAliveInterval	Integer	The number of milliseconds between subsequent runs of the keep alive thread.
keepAliveTimeout	Integer	The number of millisecond in which a node must respond to the PONG before it is considered dead.

The system can support more then one Sensor Driver attached to a SensorNode if the needed. The SensorNode provides roughly the same methods as the SensorNetworkManager service, but messages will be sent to the node identifier associated with the specific SensorNode object on which the method is invoked.

SensorNode Driver Bundle

The SensorNode driver bundle is the bundle which is installed onto the sensor node, or placed somewhere to be downloaded when that location is downloaded from the from a location specified by the sensor node. This is a standard OSGi bundle however, to save space the bundle is treated a little differently and does not contain an activator class within the bundle itself. A generic SensorNodeActivator class exists within the SensorNetworkManager bundle to activate the SensorNode driver bundles. Certain information needs to be contained within the SensorNode driver bundle, so that the activator knows what class to construct and register with the framework. This information is contained as headers within the manifest file of the sensor node driver bundle.

Three manifest headers are required, and they are described in Table 5-3. Any headers which start with the prefix ‘Sensor-‘, including the three required headers, will be included as properties on the registered service. This enables the developed to specify application specific properties for the service being registered within the framework. The ‘Sensor-‘ prefix will not be removed when added as a property. The sensor node driver implementation contains the information on how to decode the packets and typically implements the *PacketListener* interface allowing the object to be called whenever any packets are received for that sensor node. When the implementation is constructed the activator looks for a constructor accepting a BundleContext and SensorNode object and uses that constructor if found. This provides the implementation with access to the both the sensor node and the OSGi framework. Having access to the sensor node allows the implementation to send packets to the sensor node at any time. Having access to the bundle context gives the implementation the ability to obtain other services which are running in the framework including access to other sensors.

Table 5-3. The required headers in the sensor node driver bundle

Header	***Description***
Sensor-NodeId	This is the node identifier which the driver will be attached to.
Sensor-Interface	Fully qualified package name which will be used for the Interface to the sensor. This will be the name of the service exported.
Sensor-Implementation	Fully qualified packet name which will be used for the implementation of the sensor.

The SensorNode driver should implement the PacketListener interface along with the sensor interface which it is actually implementing. By the SensorNode driver implementing the PacketListener interface, it the packetRecieved method will automatically be called whenever a packet is received from that sensor node. This method can return true if it has handled the method, or false indicating that the packet should be dispatched to other PacketListener for that sensor node.

Sensor Node Firmware

The sensor node firmware is written in C for the Atmel Mega128L using GCC [20, 21] and the Linux operating system as a development environment. Programming of the micro controller is accomplished using an AVRISP in-circuit programming which connects to the computer via an available serial port. The AVRISP module plugs into the serial (or USB) port of a computer and enables the downloading of the code and data onto the processor. Real time debugging support is provided via the JTAG port, the AVR JTAG-ICE module and GDB [21]. The Real time debugging support allows the debugging of an application while it is running on the embedded processor in the application circuit. This enables the debugging of the various operating system components in real time on the sensor node hardware. In addition to programming the actual software which is running on the processor, the AVRISP programmer is also used to program the EEPROM which contains the Driver or URL reference to where the driver can be located and other runtime information such as the node identifier. A simple cooperative operating system was written for the Mega128L. The operating system makes it easier to write and maintain the complex firmware required on the sensor node operation by allowing separate cooperating processes to be developed. Each process can setup timers and send messages allowing a form of inter-process communication. The operating system abstracts the communication with the hardware and provides a standard set of API calls which the different processes running within the node can use.

Contained within the operating system, there exists the concept of a process. A process is simply a function and associated data structure used to store the functions current state. This process concept differs slightly from processes used within typical PC operating systems, but the end result is the similar, multiple paths of execution. Most conventional operating systems for PCs are preemptive, meaning that a running process can be forced to give up the CPU. The operating system used on the sensor nodes is cooperative, meaning that each process must voluntarily release the CPU to allow other processes to execute.

Whenever that process is called, the associated process data structure and message structure is passed to the process function. Contained within the message structure is a signal, which indicates the action which the process is to perform. This signal can be specified during a startTimer() or injectMessage() call. The startTimer() call specifies that a message is to be sent to a process after a certain period of time. The injectMessage() call specifies that a message is to be sent right away. The process has full control over how those signals are defined and used. When the operating system is first started, all process functions are called with a *NULL* value for the message argument. This gives each process the ability to configure itself, and possibly call startTime() to register timers so that it is then called periodically.

The process function would typically handle the incoming message, which could be generated by a timer expiring or other processes sending messages. After processing the message the process function updates the state variable within the process data structure placing that process in a new state and then returns as soon as possible. Since the operating system is cooperative in nature, there is no context switch and each process must return as soon as possible in order to allow the execution of other processes. For example, a process which may be waiting on some external device may mark its state as waiting, register a timer to be called again in a few milliseconds and finally return. This allows other processes which are waiting delivery of messages to be executed in the meantime. A cooperative operating system has many advantages in a small embedded environment since there is no context switch time, and also no storage space is used to store the CPU registers and the stack for each process. There is a form of context switching, but this context switching is done at a function level, where the individual contexts are simply program subroutines or functions.

The operating system which runs on the sensor nodes is not solely restricted for sensor network applications and could easily be adapted to other embedded applications. With this understanding, we now discus the various processes which exist within the operating system for the specific sensor node application. Within the sensor node there exists three processes, the Core Process which handles and processes incoming packets, the Communication Process which receives and validates incoming messages, and the Sensor Process which handles all interaction with the sensor specific hardware. Figure 5-9 shows a diagram with the individual components.

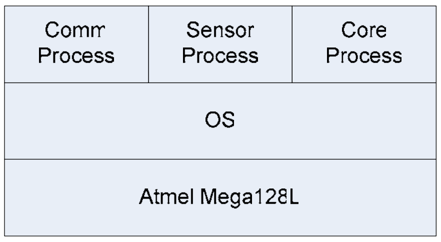


Figure 5-9. A diagram of the sensor node components

Core Process

The core process is where all of the various packet types are processed on the sensor node. It also handles the broadcasting of ALIVE packets when the sensor node first comes online, and continues broadcasting them until the first ACK packet is received. The core process receives messages from the Communication process whenever a new packet has been received and validated. A lookup table maps the packet type to a function which will be used to process that packet. For example, the PING packet type maps to a function which simply sends a PONG packet. When an EXECUTE_SENSOR_ROUTINE packet is received, a message is created pointing the incoming packet and sent to the Sensor Process. The Sensor process is a pluggable module containing the code to interact with the actual sensor hardware.

Communication Process

The purpose of this process is to inspect the incoming packets, and if a valid packet is available, a message will be sent to the CoreProcess (indicated by the CORE_PID) specifying that that a packet is available at the operating system and it should obtain and process the packet. This process will be called periodically due to the use of the startTimer() call at the end of the function. By starting a timer, the operating system will call this process back when that timer expires in the specified number of CPU ticks.

```
void communicationProcess(PROCESS *p, IPC_MSG *m) {

    if (m && packetProcess())
       jrosRaiseEventOnce(CORE_PID,CORE_SIG_PACKET);

    jrosStartTimer(0,COMM_TICKS);
}
```

Figure 5-10. Code for the communication process

The message sent to the process when this timer expires will contain the signal passed to the start timer function, in this case 0. The actual time the process is called will not be sooner then COMM_TICKS, but could be later if another function has held on to the CPU. This is the tradeoff with a cooperating operating system, if a process does not cooperate and release the CPU in a timely manner it can affect the other messages which are in the queue ready to be dispatched to other processes.

Sensor Process

The Sensor process is a replaceable process designed to be modified as different sensor hardware is used. Since the sensor process is just another process is can interact with the operating system just like other modules. It can inject messages for the core process to simulate the reception of packets, or it can register timers so that it is called periodically to make sensor readings and send SENSOR_DATA packets to send the results back to the sensor gateway. There is however, one reserved signal for the sensor process (SENSOR_SIG_EXEC) which is sent when an EXECUTE_SENSOR packet is received and handled by the Core process.

There is a direct relationship to the data generated by the Sensor Process and the data decoded by the driver which is installed into the sensor gateway. The two components are designed together where both can send and receive messages to the other. Messages for the sensor node specific to the application originate from the sensor node Driver, installed in the sensor gateway within the OSGi framework. These messages would be received at the Sensor Process, and some action performed. Messages which originate from the sensor node (SENSOR_DATA) contain the payload which will only make sense to the sensor node Driver. Figure 5-11 shows a diagram of this interaction.

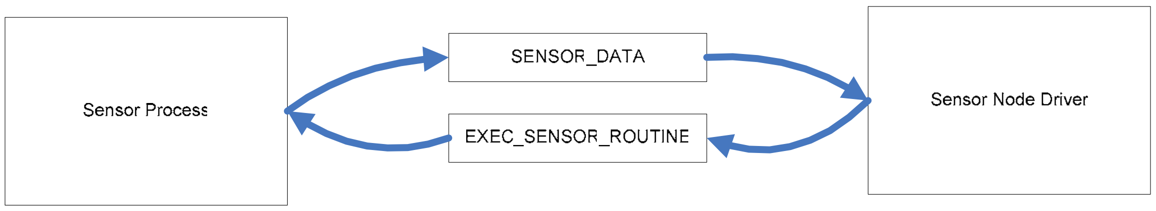


Figure 5-11. A figure showing the process/driver interaction

Sensor Driver

While not actually part of the firmware, contained on the sensor node and stored within the EEPROM is the sensor node driver, or a reference (URL) to where the driver can be downloaded. Allowing either a URL or the actual bundle contents to be stored on the sensor node provides flexibility in the situation where the downloaded bundle is too large to efficiently be stored and transmitted from the sensor node itself. This also provides a level of flexibility in case the bundle needs to be changed dynamically. The driver is actually an OSGi bundle which when installed into OSGi attaches to the SensorNode service created when a node is newly discovered by the sensor network Manager and provides the sensor specific functionality. When obtained and installed by the sensor network manager, the driver registers itself with the SensorNode service indicating that it would like to be called when a packet is received from that sensor node. The sensor driver also has the ability to send packets to the sensor node. Since the sensor driver is just another OSGi service, it can implement some Java Interface, allowing any other service within the OSGi framework to make use of the functionality which the sensor provides.

The purpose of the sensor driver is to implement any server side functionality provided by the sensor. This includes the implementation of any methods which would generate packets destined to the individual sensor node as well as the decoding of any packets containing data sent from the sensor node. The creation of the EEPROM image from the original OSGi bundle (jar format) is performed by a custom *ant* task which is included as part of the sensor network build system. Figure 5-12 depicts a driver download, when the sensor node contains a URL to where the driver can be located. In step 1, the URL is obtained from the sensor during the sensor node alive negotiation. Step

2 has the sensor network manager bundle, installed in the OSGi framework, downloading the contents of the URL and finally in Step 3 the downloaded driver is installed.

Sensor Components and Relationships

It is important to understand where each of the software components originates, and which need to be created when deploying a sensor node.

If deploying a new sensor node using an existing communication medium, a sensor process and a corresponding OSGI bundle (sensor driver) need to be created. The sensor process will run on the sensor node and receive any EXEC_SENSOR_ROUTINE packets. Upon reception of these packets, it will communicate with the sensors and then reply back with the sensor result or possibly perform some actuation. The OSGi bundle must be created and interacts with the sensor process. When the OSGi bundle sends a EXEC_SENSOR_ROUTINE, that packet, including the payload, is delivered to the sensor process on the node. It is up to the sensor process to determine how to decode and react to the payload contained in the EXEC_SENSOR_ROUTINE message. This provides great flexibility in both sensing and actuator applications. Dynamically configurable sensors could be developed with the payload of the EXEC_SENSOR_ROUTINE contains both commands and configuration instructions.

If a different communication mechanism were desired, the communication module within the operating system would need to be updated to properly send and receive packets using that new communication mechanism. All the other components (including any previously developed sensor processes) would not need to be modified due to the abstraction layer present in the operating system.

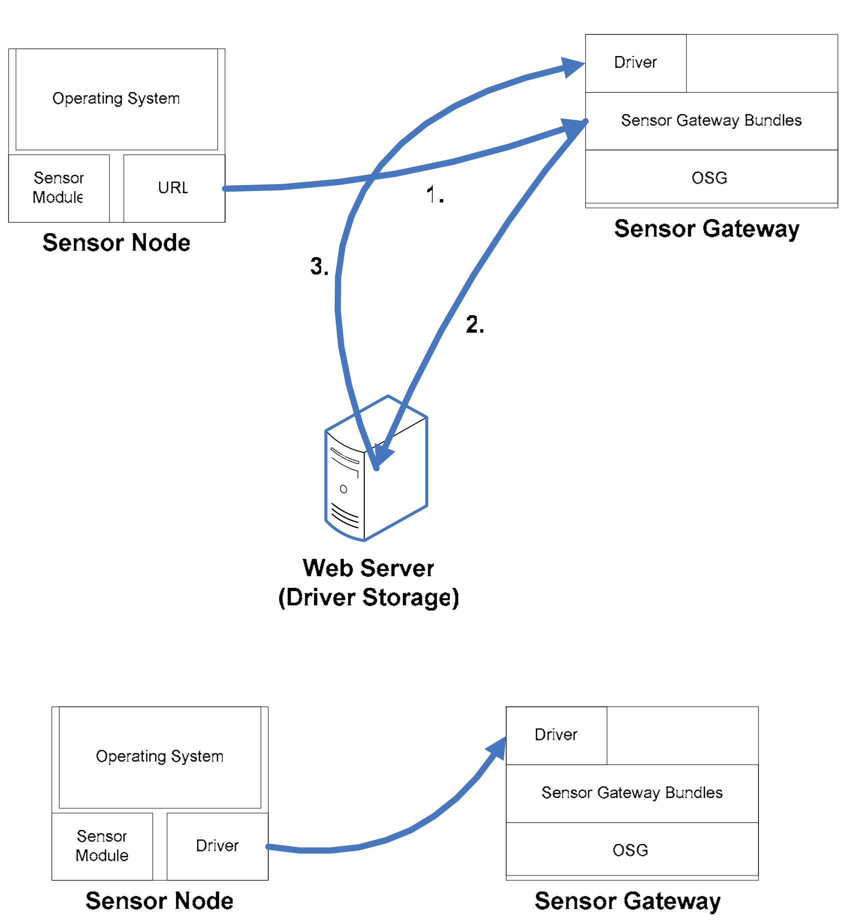


Figure 5-12. Diagram showing the driver download when the driver is stored on a web server

A new communication module would be developed for the operating system which implements initialize, send and receive functions. These functions would be hooked into the operating system at runtime, providing the operating system with the methods to be called when sending or receiving packets. The initialization function is required to setup any of the IO ports which are required to communicate with the new hardware. On the sensor gateway side, a new implementation for the PacketReader class would need to be developed. This class is analogous to the communication module in the firmware and is what the sensor gateway uses for sending and receiving packets. Figure 5-13 shows the relationships and data flow between the various components of the system.

Sensor Node Emulator

To aid in the development and debugging of sensor nodes, an emulator has been designed which replicates all the functionality of a sensor node by using a standard computer network and the transmission of UDP packets.

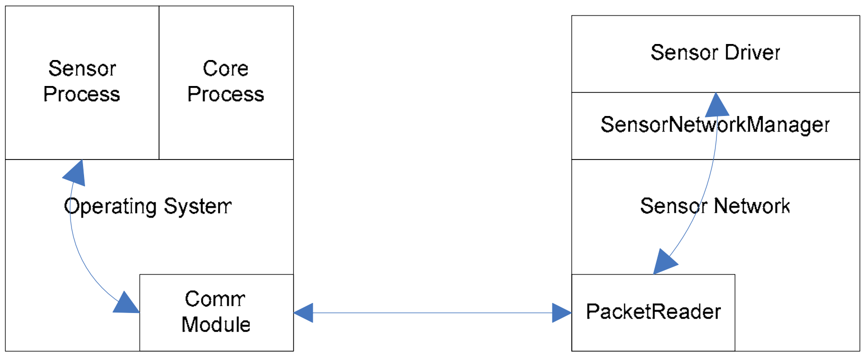


Figure 5-13. Diagram showing the relationship between sensor components

The emulator is a Java Swing application which interacts with the framework in exactly the same way as a sensor node would. Loaded into the emulator is the actual EEPROM image which would be loaded into the real sensor node and this EEPROM image contains the node information along with sensor node driver or a reference to the driver in the form of a URL.

The emulator uses a plug-in architecture to allow the creation of a user interface which would be used to simulate the manipulation of the sensor. The GUI code and sensor driver code are contained within a class which implements the emulator plugin interface. This interface provides a method which is called whenever an EXECUTE_SENSOR_ROUTINE message is received. This allows the plug-in to read the value from the GUI and generate a SENSOR_DATA packet which will be sent to the sensor gateway as a response. The components of the plug-in are shown in figure 5-14.

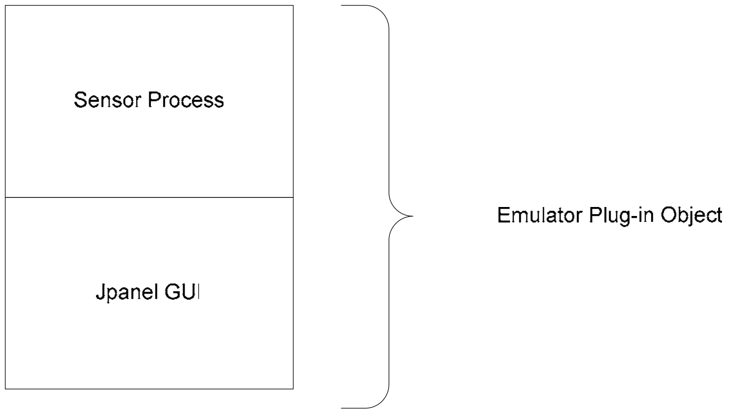
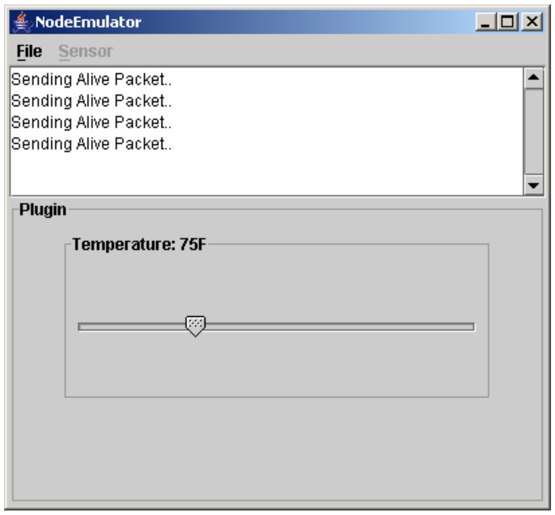


Figure 5-14. The components of the emulator plug-in object

A screen shot of the emulator is shown below in Figure 5-15. This emulator depicts a temperature sensor and contains a slider used to manipulate the temperature reading which will be reported back to the sensor gateway. Also provided in the plug-in is the ability to specify a menu which will show up in the emulator. This can be used for complex plug-ins requiring more user interaction.



```
Sending Alive Packet..
Sending Alive Packet..
Sending Alive Packet..
Sending Alive Packet..
```

Figure 5-15. A screen shot on the emulator with the temperature sensor plug-in

CHAPTER 6
DESIGN CONSIDERATIONS

In this chapter we will be designing a simple sensor node which will be measuring the light level. We will assume that there is a communication mechanism already in place between the sensor node and the sensor gateway, so we will not consider this in our design. We will discuss all the various steps involved including the design of the packet format, the design of the sensor process and finally the design of the sensor driver OSGi bundle.

Sensor Packet Payload

The first step in the design of a sensor node is to specify the format of the payload used for communication both to and from the sensor node. Our sensor will be a very simple sensor which reports light level measured via a light sensitive resistor connected to one of the ADC ports of the microcontroller. The value of the light will be an integer from 0 to 255, where the higher levels indicate darkness. Since we a dealing with one way communication of just one byte of data, the payload design is very simple. The payload of the EXECUTE_SENSOR_ROUTINE will not be used, as the sensor will not receive any data from the sensor gateway. The payload of the SENSOR_DATA packet will contain a single byte in position 0 containing the value of the light reading. Figure 6-1 shows the SENSOR_DATA packet.

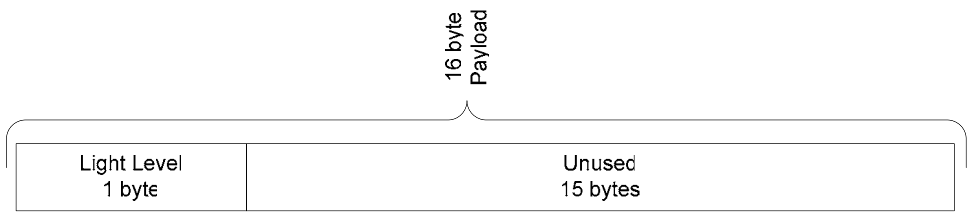


Figure 6-1. Figure showing the SENSOR_DATA packet to be used in the light level sensor

Sensor Process

The next task would be the creation of the Sensor Process. This is a piece of C code which is linked with the operating system to create the firmware which is then installed on the sensor node. The sensor process will be called when an EXECUTE_SENSOR_ROUTINE packet is received, and will in response generate a SENSOR_DATA packet response. Our sensor process will also setup a timer causing the light reading to be broadcast at a set interval. This broadcasting will be in addition to the on-demand reporting created by the reception of the EXECUTE_SENSOR_ROUTINE packet.

```
void sensorProcess(PROCESS *p, IPC_MSG *m) {
     static char buffer[16];
        if (m == NULL)  { a2dInit(); return; }

        if (m->signal == SENSOR_SIG_EXEC) {
                jrosClearTimer(2);
                return;
        }

        buffer[0] = a2dConvert8bit(1);
        packetSend(0,PACKET_SENSOR_DATA,buffer);
        jrosStartTimer(2,notifyTime);

        if (m->signal == SENSOR_SIG_EXEC)
                jrosFree(m->data.packet);
}
```

Figure 6-2. The sensorProcess function

Figure 6-2 shows the code for the sensor process. The first part of the function performs the initialization of the analog to digital conversion, this code will only be called when the node first starts up and initialized all the processes by passing a *NULL* message. The second block of code will reset the periodic timer if the incoming message is requesting a manual sensor reading. This reset of the timer prevents the sensor reading from broadcasting two sensor readings in a short period of time. The final piece of code actually calls the analog to digital conversion, placing the result in the first byte of the payload. A packet is then generated and sent to node 0 (the gateway node) containing the light value reading in the first byte of the payload. The final component performs a memory free operation on the message data packet if the function was called with a SENSOR_SIG_EXEC message. When this message is received the MSG structure contains the actual packet which was received (EXECUTE_SENSOR_ROUTINE) and could be used to control certain aspects of the sensor process. One function of this could be to turn on and off automatic reporting, or perhaps set the automatic reporting interval.

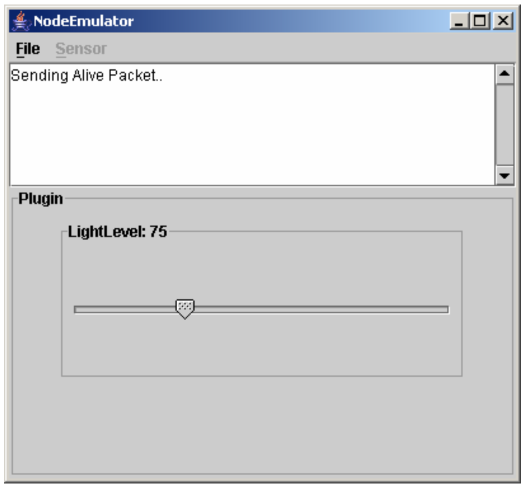
Sensor Driver

The next component which must be created is the Sensor Driver. The Sensor Driver requires two components, an implementation and an interface. The interface is a standard Java interface and defines the methods which are available for others to call. Careful thought must be put into the interface, since once written and distributed; it is very difficult to change it with breaking the existing functionality. Our interface will be called *LightLevelSensor* and define a single method *getLightLevel* returning a byte. The name of the interface is the name by which other services will be able to obtain and use the data provided by the sensor.

With the interface written an implementation which provides the real code for the methods specified for the interface must be written. This class will be passed the SensorNode object, so that it could communicate with the SensorNode and it will contain a method which will be called whenever a packet is received from the sensor node. If the packet is of type SENSOR_DATA, the data will be decoded and a value will be assigned to a local class variable. The implementation also shows a more advanced feature that if the data is too old (30 seconds), a call to the sensor node will be performed and that method will wait until the new packet is delivered before returning, up to a maximum of 5 seconds. The full source code for the example light sensor is shown in Appendix E.

Emulator Plug-in

Typically, the sensor node would be tested on the emulator before deployed on the actual sensor node. In order to use the emulator, you must create the equivalent Sensor Process for the emulator. This will read some pseudo light sensor and generate a SENSOR_DATA packet with the correct value. Typically, the pseudo light sensor is the value from some GUI component within the emulator plugin. In order to create the emulator plugin a class implementing the Emulator interface is created and a JPanel is created and returned by the getDisplayPanel() method. For the lightlevel sensor, the panel is simply a panel with a slidebar ranging from 0 to 255. In addition to returning the JPanel, the plugin has a executeSensorRoutine() method which needs to return the payload which will be sent back to the sensor gateway when a EXECUTE_SENSOR_ROUTINE packet is received. The emulator framework itself has the ability to automatically send SENSOR_DATA packets at specified intervals. Figure 6-4 shows a screen shot of the completed emulator.



Sending Alive Packet..

Figure 6-4. Screen shot of the light level emulator

Build System

Included within the sensor network architecture is an *ant* based build system, making it easy to compile the Sensor Drivers and emulators used for testing the system. Once the interface, implementation and plug-in are developed an *ant* build script is modified to include the correct package names, the node identifier, and other configuration options. Once this information is placed in the build script, pre-determined *ant tasks* can be run creating the different files needed to emulate the sensor and also place the EEPROM image on the hardware sensor node. Once all the various bundles are built the *api* bundle can be distributed to other applications which might be using your sensor node service and the *all* bundle placed on a web server and referenced by the sensor node, or if small enough an EEPROM image file can be created and downloaded right into the sensor node itself.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Achievements and Contribution

This thesis is an attempt to create a sensor network architecture which is functional, well designed and expandable. It uses well known proven technology such as OSGi and Java. By making use of commercial off-the-shelf (COTS) components, the hardware sensor nodes can be manufactured at a reasonable price. The hardware uses a stack based architecture allowing the individual layers to be redesigned and replaced without affecting any of the other components. This layered architecture allows customization of the sensor nodes for a specific application. The software framework can automatically download the required software used to interact with that node from the node itself. This surrogate concept allows the nodes to be easily upgraded without requiring software changes to the sensor gateway. The software can not only reside on the node itself, but also referenced by a URL which the sensor gateway accesses. This referral URL allows the sensor node driver to easily be updated. A cooperative operating system was designed for the Atmel microprocessor which is simple, easy to understand and yet very functional.

Future Work

Hopefully, this thesis just describes the beginning of this sensor framework. Using the concepts developed thus far, it is hoped that this system will find its way into many other applications and be improved along the way. The next step in the framework is to develop applications and determine where the framework fails to meet the expectations of the application.

Different communication and sensor boards can be developed to support many different communication and sensing requirements. Communication boards envisioned include Bluetooth, Ethernet, Cellular (CDPD) and likely many others. Various sensor boards could include support for the common 1-wire sensors from Dallas Semiconductors and an infinite number of sensors to be defined by the specific application at hand.

More work can be done with the OSGi component to implement caching of the Sensor Driver, so that shutting down and restarting of the sensor node does not cause the bundle to be re-downloaded. Cryptographically signing the Sensor Drivers would also be needed to enhance security.

APPENDIX A
BILL OF MATERIALS FOR PROCESSOR BOARD

Table A-1. Bill of materials for the processor board

Qty	Digikey Part #	Description	Cost
1	445-1025-1-ND	INDUCTOR MULTILAYER 10UH 1608	0.44
1	ATMEGA128L-8AI-ND	IC AVR MCU 128K 8MHZ LV 64-TQFP	15.05
1	296-1112-1-ND	IC SINGLE INVERTER GATE SOT23-5	0.4
1	MBRS130T3OSCT-ND	DIODE SCHOTTKY 30V 1A SMB	0.4
1	399-1788-1-ND	CAPACITOR TANT 4.7UF 35V 10% SMD	2.18
1	XC754CT-ND	CRYSTAL 8.000MHZ 16PF SMD	1.35
2	PCC220ACVCT-ND	CAP CERAMIC 22PF 50V 0603 SMD	0.044
2	PCC180ACVCT-ND	CAP CERAMIC 18PF 50V 0603 SMD	0.044
1	XC488CT-ND	CRYSTAL 32.768KHZ 12.5PF SMD	1.08
6	399-1101-1-ND	CAP .10UF 25V CERAMIC Y5V 0603	0.061
1	296-13086-1-ND	IC DRVR/RCVR MULTCH RS232 20SOIC	2.13
2	P10.0KHCT-ND	RES 10.0K OHM 1/16W 1% 0603 SMD	0.09
1	MAX710ESE-ND	IC DC/DC CONV STEP-UP/DWN 16SOIC	7.61
2	311-1069-1-ND	CAP CERAMIC 100PF 50V NP0 0603	0.062
1	S2105-20-ND	CONN HEADER 2MM SNGL STR 20POS	0.8
1	S2105-18-ND	CONN HEADER 2MM SNGL STR 18POS	0.76
1	S2105-22-ND	CONN HEADER 2MM SNGL STR 22POS	0.84
1	A26592-ND	CONN HEADER RT/A 6POS .100 30AU	1.86
1	A26597-ND	CONN HEADER RT/A .100 10POS 15AU	2.55
1	401-1094-1-ND	SWITCH TACT SILVER PLATE SMT	0.46
1	308-1228-1-ND	POWER INDUCTOR 22UH 1.20A SMD	1.1
2	399-1777-1-ND	CAPACITOR TANT 100UF 10V 10% SMD	2.6
1	AT24C1024W10SI2.7-ND	IC SEEPROM 1M 2.7V 8SOIC	6.3

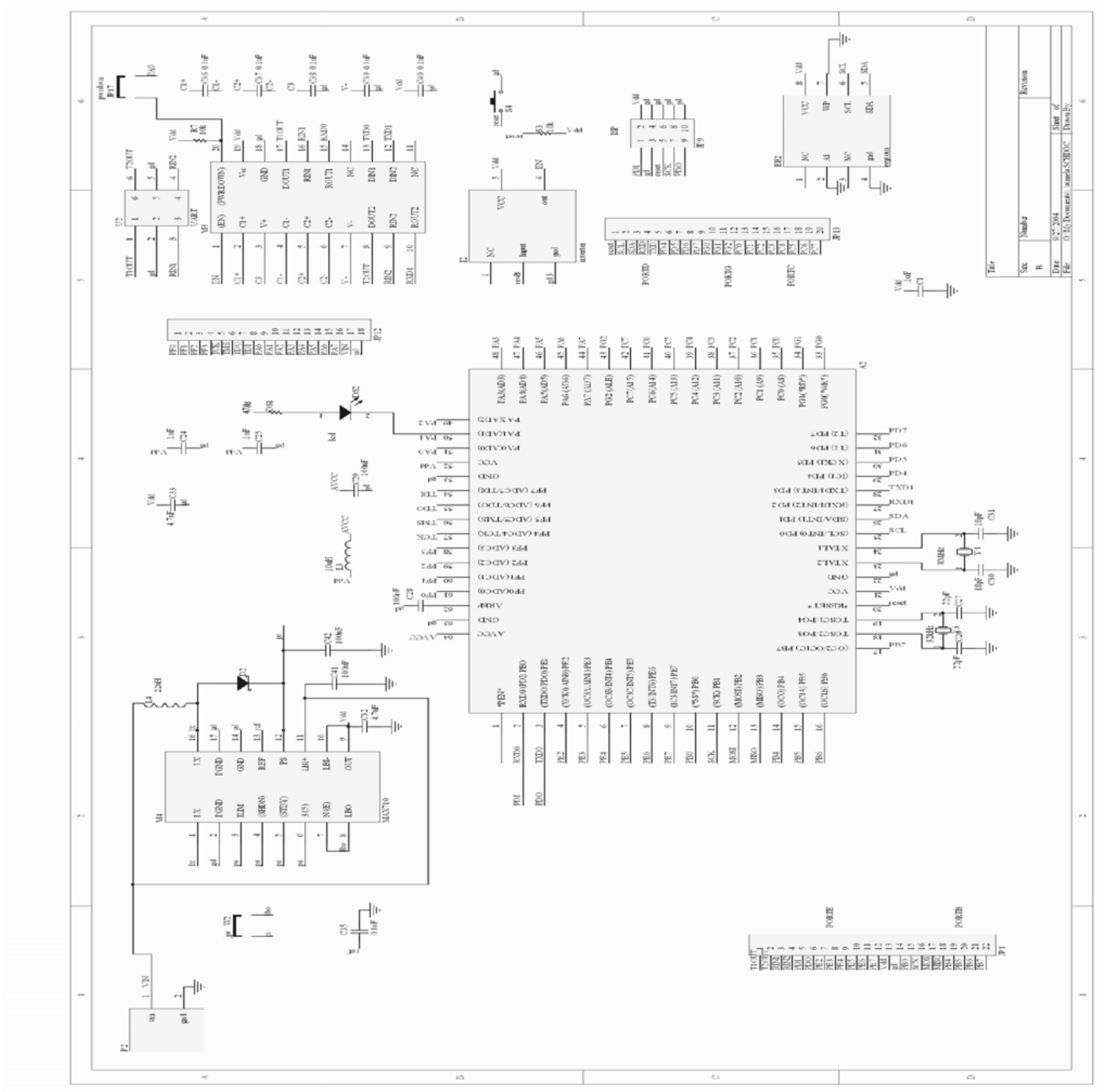
APPENDIX B
BILL OF MATERIALS FOR RADIO COMMUNICATION BOARD

Table B-1. Bill of materials for the radio communication board

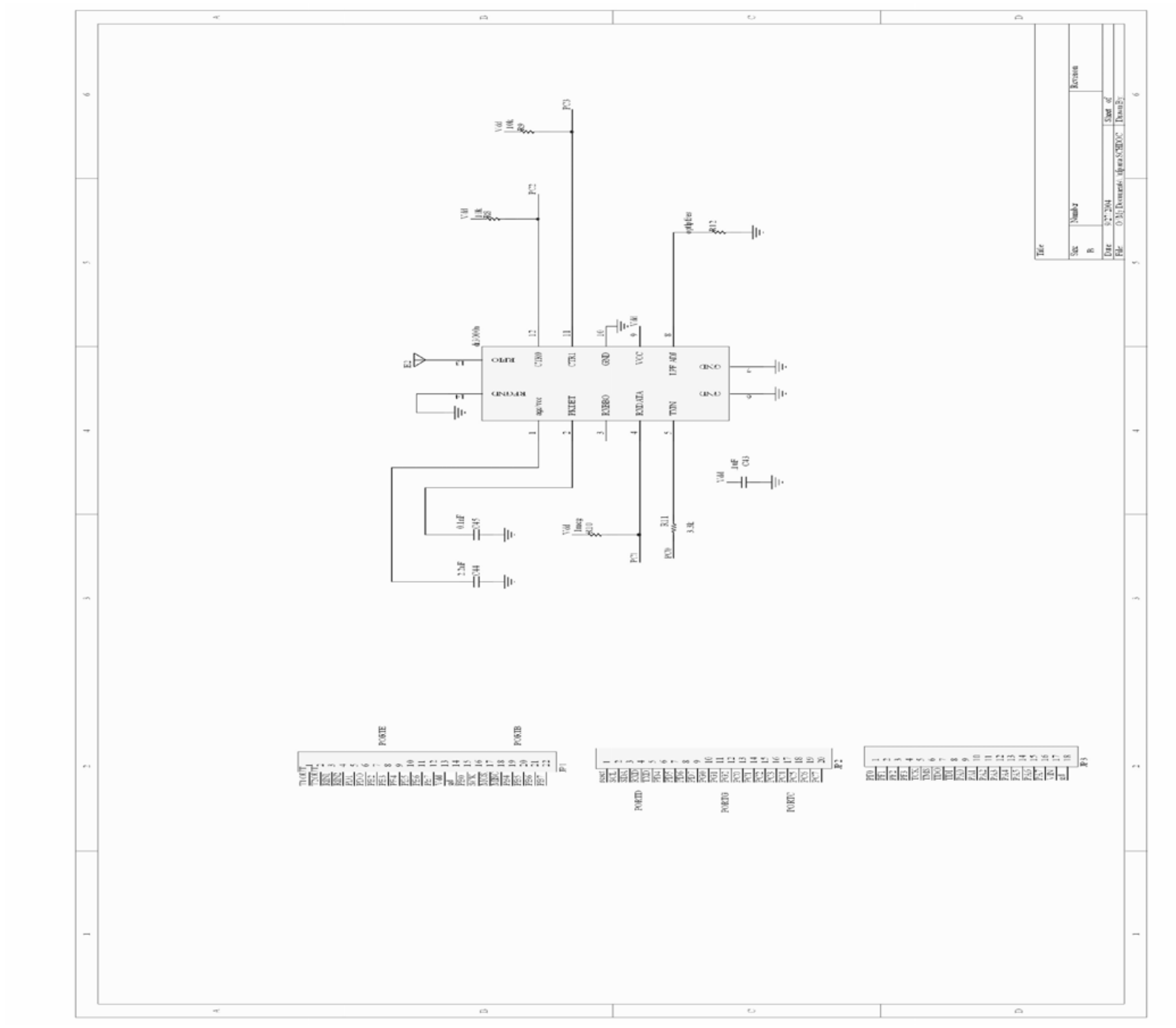
Qty	Digikey Part #	Description	Cost
1	ANT-916-JJB-RA-ND	ANTENNA MINI 7MM 916MHZ RT ANGLE	1.78
1	N/A	RFM's DR3000-1 wireless transceiver	30.00
1	S2105-20-ND	CONN HEADER 2MM SNGL STR 20POS	0.8
1	S2105-18-ND	CONN HEADER 2MM SNGL STR 18POS	0.76
1	S2105-22-ND	CONN HEADER 2MM SNGL STR 22POS	0.84
1	S2102-18-ND	CONN HEADER 2MM SINGLE SMD 18POS	2.03
1	S2102-20-ND	CONN HEADER 2MM SINGLE SMD 20POS	2.24
1	S2102-22-ND	CONN HEADER 2MM SINGLE SMD 22POS	2.49
1	ANT-916-CW-RCR-ND	ANTENNA 916MHZ RA 1/4 WHIP RPSMA	6.39
2	Stock	10k ohm resistors 0603	
1	Stock	1 meg ohm resistors 0603	
1	Stock	2.2nF Capacitor 0603	
1	Stock	0.1nF Capacitor 0603	
1	Stock	0.1uF Capacitor 0603	

APPENDIX C
CIRCUIT DIAGRAMS

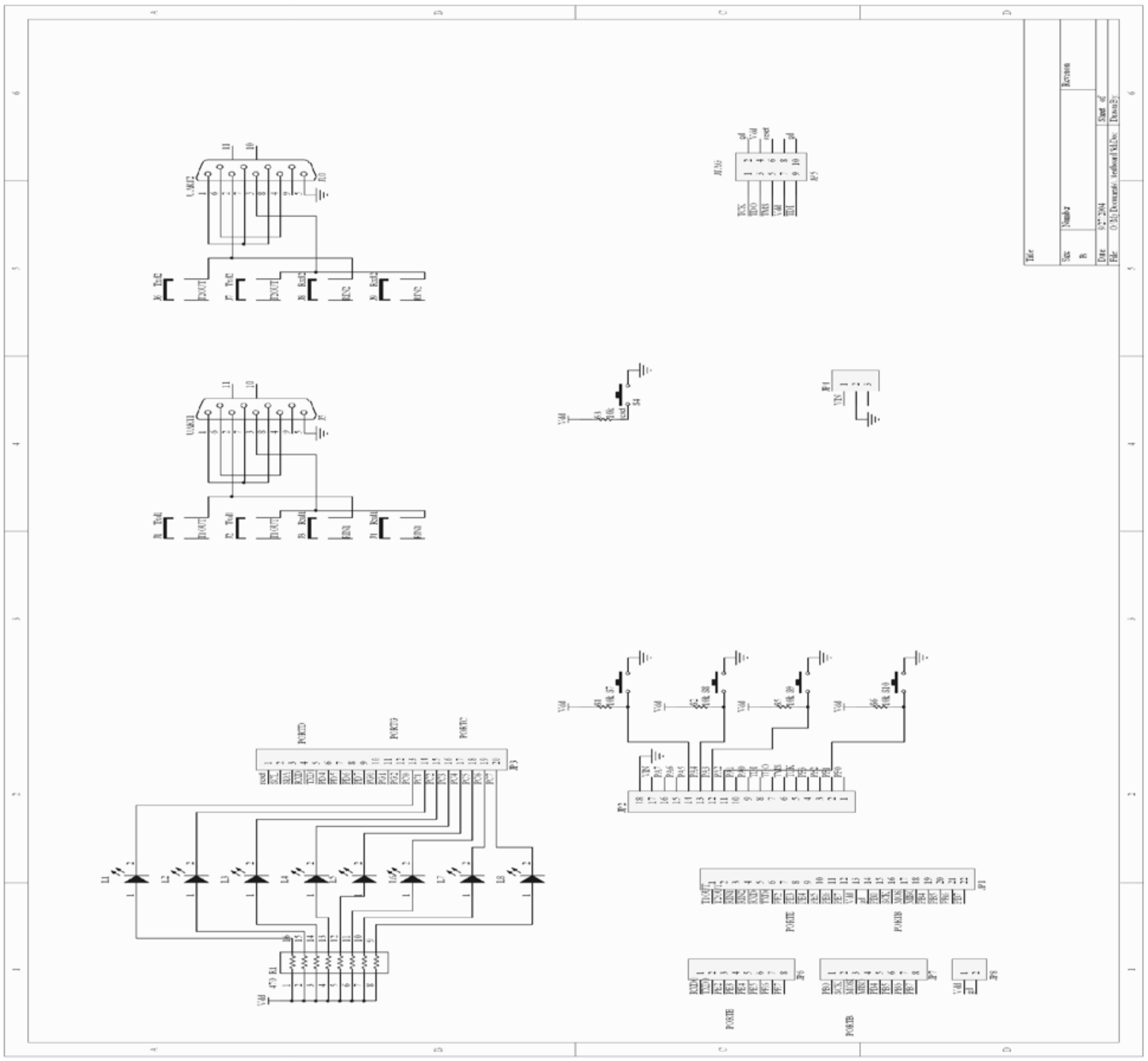
Processor Board Circuit Diagram



Radio Communication Board Circuit Diagram



Debug/Programming Board Circuit Diagram



APPENDIX D
EXAMPLE SOURCE CODE

This appendix provides the complete source code for the example application which was developed in chapter 6. The interface file, LightLevelSensor.java, the implementation file LightLevelSensorImpl.java, and the emulator plug-in are all provided.

LightLevelSensor.java

```
/*
 * Created on May 25, 2004
 */
package edu.ufl.icta.smarthome.lightlevel;

/**
 * Interface used for a Light Level Sensor. Provides methods
for obtaining the
 * current light level.
 *
 * @author jr
 * @version $Id$
 */
public interface LightLevelSensor {
    /**
     * Obtain the current temperature reading from the
Sensor. The value is
     * returned in Celsius.
     *
     * @return the temperature of the sensor in C.
     */
    byte getLightLevel();
}
```

LightLevelSensorImpl.java

```
/*
 * Created on Jun 3, 2004
 */
package edu.ufl.icta.smarthome.lightlevel.impl;
```

```
import org.osgi.framework.BundleContext;

import edu.ufl.icta.sensors.common.packets.Packet;
import edu.ufl.icta.sensors.node.SensorNode;
import
edu.ufl.icta.sensors.sensornetworkmanager.PacketListener;
import edu.ufl.icta.sensors.util.Logger;
import edu.ufl.icta.smarthome.lightlevel.LightLevelSensor;

public class LightLevelSensorImpl implements
LightLevelSensor, PacketListener {
    protected final int DATA_TIMEOUT = 30 * 1000;
    protected byte currentLightLevel;
    protected long data_time;
    protected SensorNode node;
    protected Logger log;
    protected BundleContext context;

    public LightLevelSensorImpl(BundleContext context,
SensorNode node) {
        this.context = context;
        log = new Logger(context);
        this.node = node;
    }

    public synchronized byte getLightLevel() {
        if (System.currentTimeMillis() - data_time >
DATA_TIMEOUT) {
            node.executeSensorRoutine();
            try {
                wait(5000);
            } catch (InterruptedException e) {

            }
        }
        return currentLightLevel;
    }

    public void processPayload(byte b[]) {
        data_time = System.currentTimeMillis();
        currentLightLevel = b[0];
    }

    public synchronized boolean dispatchPacket(Packet
packet) {
        if (packet.getType() == Packet.SENSOR_DATA) {
            notifyAll();
            processPayload(packet.getPayload());
            return true;
        }
        return false;
    }
}
```

LightLevelPlugin.java

```
package edu.ufl.icta.sensors.emulator.plugins;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JMenu;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.border.TitledBorder;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import
edu.ufl.icta.osgi.sensors.emulator.NodeEmulatorPlugin;
/**
 * This code was generated using CloudGarden's Jigloo
SWT/Swing GUI Builder,
 * which is free for non-commercial use. If Jigloo is being
used commercially
 * (ie, by a for-profit company or business) then you should
purchase a license -
 * please visit www.cloudgarden.com for details.
 */
public class LightLevelPlugin extends javax.swing.JPanel
          implements
               NodeEmulatorPlugin {
     private JSlider lightLevelSlider;

     /**
      * Constructor for the Plugin. Initializes the plugin
for use in the
      * emulator.
      *
      */
     public LightLevelPlugin() {
          initGUI();
     }
     /**
     * Initializes the GUI.
     * Auto-generated code - any changes you make will
disappear.
     */
     public void initGUI(){
          try {
               preInitGUI();

               lightLevelSlider = new JSlider();

               GridLayout thisLayout = new GridLayout(1,1);
               this.setLayout(thisLayout);
               thisLayout.setRows(1);
```

```
			thisLayout.setHgap(0);
			thisLayout.setVgap(0);
			thisLayout.setColumns(1);
			this.setPreferredSize(new
java.awt.Dimension(320,125));

			lightLevelSlider.setMaximum(255);
			lightLevelSlider.setPreferredSize(new
java.awt.Dimension(320,125));
			lightLevelSlider.setBorder(new
TitledBorder(null, "LightLevel: 0", TitledBorder.LEADING,
TitledBorder.TOP, new java.awt.Font("Dialog",1,12), new
java.awt.Color(0,0,0)));
			this.add(lightLevelSlider);
			lightLevelSlider.addChangeListener( new
ChangeListener() {
				public void stateChanged(ChangeEvent
evt) {
					lightLevelSliderStateChanged(evt);
				}
			});

			postInitGUI();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
	/**
	 * Method called before GUI is initialized.
	 *
	 */
	private void preInitGUI() {
		/* Do nothing */
	}
	/**
	 * Method called after the GUI is initalized. Used to
setup a default value
	 * for the temperatureSlider.
	 *
	 */
	public void postInitGUI() {
		lightLevelSlider.setValue(75);
	}
	/**
	 * Method called when the temperatuteSlider is
adjusted.
	 *
	 * @param evt
	 *            The event for the change.
	 */
	protected void lightLevelSliderStateChanged(ChangeEvent
evt) {
		TitledBorder t = (TitledBorder)
lightLevelSlider.getBorder();
```

```
          lightLevelSlider.setBorder(BorderFactory
                    .createTitledBorder("LightLevel: "
                              + ((JSlider)
evt.getSource()).getValue()));
     }
     /*
      * (non-Javadoc)
      *
      * @see
support.plugins.NodeEmulatorPlugin#getDisplayPanel()
      */
     public JPanel getDisplayPanel() {
          return this;
     }
     /*
      * (non-Javadoc)
      *
      * @see support.plugins.NodeEmulatorPlugin#getMenu()
      */
     public JMenu getMenu() {
          return null;
     }
     /*
      * (non-Javadoc)
      *
      * @see    */
     public byte[] executeSensorRoutine(byte[] arg0) {
          byte b[] = new byte[16];
          b[0] = (byte) lightLevelSlider.getValue();
          b[0] = (byte) b[0];
          return b;
     }
}
```

LIST OF REFERENCES

1. Weiser, M., "Ubiquitous Computing" [online] Available: http://www.ubiq.com/hypertext/weiser/UbiCompHotTopics.html [Accessed 29 September 2004]

2. Hill J., Horton M., Kling R., Krishnamurthy L., "The Platform Enabling Wireless Sensor Networks" Communications of the ACM, Volume 47, Issue 6, 41-46.

3. King, R. "Intel Research Mote" [online] Available: http://webs.cs.berkeley.edu/retreat-1-03/slides/imote-nest-q103-03-dist.pdf [Accessed 16 September 2004]

4. Hill J., Szewczyk R., Woo A., Hollar S., Culler D., Pister K., "System Architecture Directions for Networked Sensors" [online] Available: http://webs.cs.berkeley.edu/tos/papers/tos.pdf 2000. [Accessed 16 September 2004]

5. Mainwaring A., Polastre J., Szewczyk R., Culler D., Anderson J., "Wireless Sensor Networks for Habitat Monitoring" In Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications. ACM Press, New York, 2002, 88-97

6. "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks" [online] Available: http://www.isi.edu/scadds/projects/diffusion.html. [Accessed 16 September 2004]

7. "TinyDB: A Declarative Database for Sensor Networks" [online] Available: http://telegraph.cs.berkeley.edu/tinydb. [Accessed 16 September 2004]

8. "Cougar: The Network is The Database" [online] Available: http://www.cs.cornell.edu/database/cougar/ [Accessed 16 September 2004]

9. Eschenauer, L. and Gligor, V. "A Key-Management Scheme for Distributed Sensor Networks" In Proceedings of the 9th ACM Conference on Computer and Communication Security. ACM Press, New York, 2002, 41-47.

10. Karlof, C. and Wagner, D. "Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures" [online] Available: http://webs.cs.berkeley.edu/papers/sensor-route-security.pdf [Accessed 22 November 2004]

11. Ping, S. “Delay Measurement Time Synchronization for Wireless Sensor Networks” [online] Available: http://www.intel-research.net/Publications/Berkeley/081120031327_137.pdf [Accessed 16 September 2004]

12. Elson Jeremy, Lewis Girod , Deborah Estrin, “Fine-grained network time synchronization using reference broadcasts”, ACM SIGOPS Operating Systems Review, Volume 36, Issue SI, 147-163.

13. Hollar, S. “COTS Dust” Master’s Thesis, August 2000.

14. “The Scientist and Engineer’s Guide to TinyOS Programming” [online] Availble: http://ttdp.org/tpg/html/book/book1.htm [Accessed 16 September 16, 2004]

15. “Chipcon CC1000 Transciever Chip” [online] Available: http://www.chipcon.com/files/CC1000_Brochure_3_0_.pdf [Accessed 16 September 2004]

16. Crossbox Sensicast Software http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/SMS-DEVE-100_012604.pdf [Accessed 16 September 2004]

17. “OSGi Alliance Website” [online] Available: www.osgi.org [Accessed 16 September 2004]

18. Hohlt, B., Doherty L., Brewer, E. “Flexible Power Scheduling for Sensor Networks” [online] Available: http://webs.cs.berkeley.edu/tos/papers/ipsn_hohlt.pdf [Accessed 16 September 2004]

19. Atmel Corporation, “Atmel ATMega128L Complete Datasheet” [online]. Available from: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf. [Accessed 16 September 2004]

20. “GCC Compiler Project Website” [online] Available: http://gcc.gnu.org/ [Accessed 16 September 2004]

21. “AVR LIBC Probject Website” [online] Available: http://savannah.nongnu.org/download/avr-libc/ [Accessed 16 September 2004]

22. RF Monolithics Corporation, “TR1000 Transceiver Datasheet” [online] Available: http://www.rfm.com/products/data/tr1000.pdf. [Accessed 16 September 2004]

BIOGRAPHICAL SKETCH

James Russo was born in New York and raised in South Florida. He obtained his bachelor’s degree in computer engineering from Florida Atlantic University in August 2002. He then joined the University of Florida and the Pervasive Computing Lab in August 2003 to peruse his master’s degree. He is married to Julie, a pediatric dentist, and upon completion of his degree they both are planning to move to the Orlando, Florida, area where Julie will open up her own dental practice. James has plans to continue working as a Systems Engineer for NTT/Verio, the company where he has been employed since 1996.