RFID APPLICATION FOR BIOLOGICAL TELEMETRY

A Design Project Report
Presented to the Engineering Division of the Graduate School
Of Cornell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering

By
Diana M. Rodriguez Tobon
Project Advisor: Dr. Bruce Land
Degree Date: August, 2005

ABSTRACT

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: RFID Application for Biological Telemetry

Author: Diana M. Rodriguez Tobon

Abstract:

This project aimed to develop a wireless, inexpensive system to detect and monitor vital signs of animals. The system was based on Radio Frequency Identification (RFID) technology and consists of a passive RFID tag implanted underneath the skin of the animal and a base station connected to a computer. The passive micro-transponder tag collects power from the 125 kHz magnetic field generated by the base station, gathers information about the animal such as its temperature and heart rate and sends this information to the base station. The base station receives, decodes and makes the information available to the user for further analysis. Manchester code was used to send the information wirelessly. The system performed as desired only with a 10cm diameter antenna attached to the transponder.

Report Approved by
Project Advisor: _____Date:_____

EXECUTIVE SUMMARY

The purpose of this project was to design and implement a device to collect and monitor real-time biological information of a small animal. The project had two main requirements. The first one was to develop a system that would not alter the natural vital signs of the animals while collecting them. The second requirement asked for a low cost system. A RFID model was chosen to comply with these two requirements. The RFID systems consist of a tag implanted underneath the skin of the animal and a base station connected to a computer. A passive RFID tag, capable of generating power from a magnetic field, was chosen to eliminate the need for a periodic change of batteries inside the animal. A low power consumption microcontroller was added to the tag to collect and send the animal vital signs to the base station. The base station generates the 125 kHz magnetic field that powers the tag, receives the information and presents it to the user through a computer interface GUI.

The system performed as desired only when the tag was attached to a 10cm diameter antenna. When the antenna was reduce to a size that would fit inside the animal, the transponder failed to generate enough energy to power the microcontroller.

Alternatives to further explore this problem are discussed, including adding a battery to power the low power consumption microcontroller only. This battery would only be on when the base station is collecting information from the base station, and would be off when the base station is not close to animal. This would make the battery last a long time, preventing periodic changes.

TABLE OF CONTENTS

1. INTRODUCTION

Temperature, breathing rate, heart rate and blood pressure are all examples of vital signs. They are physical signals that assess the quality of the performance of a body. These signs change depending on the age of the body, its gender, its weight, how the body reacts to exercise, etc. (normal ranges for the average healthy adult are temperature: 97.8-99.1 F, breathing: 12-13 respirations per minute and heart rate: 60 -80 beats per minute at rest). However in animals limited information is drawn from a single measurement of any of these signs; only several measurements over an extended period of time give significant data.

For instance, when the temperature of an animal alternates from high to low values, what is known as a cycle, may represent the onset of pain or release of bacteria into the blood. These cycles are also important determining when to take some laboratory tests and when medications should be given. Detecting and monitoring vital signals in animals can provide useful information to assess the level of performance of their body, their behavior and health status. Therefore devices that can provide accurately vital signs in real-time are highly valued in veterinary clinics and animal researches.

This project aims to use new technologies to develop and implement an inexpensive device to detect and monitor animal vital signs.

2. DESIGN PROBLEM AND REQUIREMENTS

2.1. ALTERNATIVES IN THE MARKET RIGHT NOW

A company called Vetronics [1], part of Bioanalytical Systems, Inc was the first company to develop a device to monitor animal vital signs for a price that made it affordable for a wide range of users (~$5000.00). Currently Vertronics offers a wide variety of products to veterinarians and researchers. They include the ECG Analyzer to monitors the heart rate and heart rhythm and the VitalScan Monitor [2] (see figure1) to monitor temperature and blood pressure among other things. However while using these devices the animals have to be sedated or firmly hold down. To measure the temperature for example, a probe is inserted into the rectum of the animal.

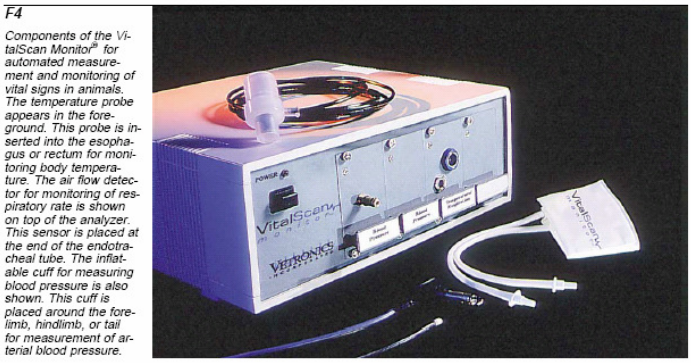

Figure1. VitalScan

Devices that collect vital signs when the animal is not directly connected to it are more practical for research. For example, a research where the main objective is to obtain real-time data from animals reacting to different environments would be highly limited if the animals can not move freely.

Presently, there is an alternative device to collect vital signs when the animal is not directly connected to it. Battery powered telemetry transmitters are devices that can be

implanted inside the animal and can collect and send the required vital signs wirelessly to a base station.

2.2. PROJECT REQUIREMENTS

To allow the monitored animal to move without restraints an important requirement for this project would be to make the device wireless. Devices that require cables from the animal to the base station are likely to affect movements of the animal during experiments. One of the major trade offs between a wireless device and a wired device is the amount of information per second that can be transmitted. The wireless device has a lower data transmission rate. An adequate rate is important specially to successfully transmit the heart rate of the animal. Designing the transmitting scheme of the wireless system this rate should be taken into account.

A wireless system requires two separate devices: a transmitter and a receiver. The transmitter is implanted inside the animal, collecting and sending data to the receiver. The receiver is placed close to a restricted area where the animal is located, receiving and decoding the data from the transmitter. The entire system has three major constrains. It should work within a specific area, the collected data should be accurate, and the overall cost including any future maintenance, should be low. Needless to say, the entire system must guarantee the safety of the research and the animals interacting with it.

In order to fulfill those requirements the design of the project has the following constrains. The transmitter to be implanted in the animal should be small enough to not harm the animal nor perturb the normal behavior and movements of it in any way. A second constrain is the cost of the transmitter. Implanting the device into the animal is a

delicate process that requires time and a lot of work. Given the device will not harm the animal in any way the idea is to implant the device into the animal once and leave it there for an extensive period of time. However, the goal is to monitor several animals at the same time, creating a need for a large number of transmitters. Hence the device to implant should be low-cost.

In addition, the transmitter should not require a periodic change of batteries. Changing the batteries would also involve removing the transmitter from the animal and as previously stated this complex procedure should not take place on a regular basis. This implies that the transponder should have a different way to obtain power. Designing the wireless system this must be taken into account. The receiver should generate a powerful magnetic field capable of powering the transmitter, as well as the transmitter should be sensitive enough to obtain power from the receiver's magnetic field.

One way the transmitter can accomplish this is through its antenna. However the size of the transmitter antenna limits its capacity to collect power from the reader's magnetic field. As the size of the antenna increases the transmitter's capability to collect power increase, however the bigger the antenna the more invasive the device is likely to be, creating a direct conflict with the size of the transmitter. This design problem requires finding common ground to comply with both requirements.

2.3. Design Specifications

2.3.1. Wireless Transmitter

- Collects animal vital signs
- Transmits animal vital signs in real time

- Fits beneath the skin of the animal
- Collects enough power from a magnetic field created by the base station to power itself
- Low power consumption
- Low cost

2.3.2. BASE STATION/READER

- Generates a powerful magnetic field capable of powering the transmitter
- Detects and decodes the information send by the transmitter
- Presents the information to the user

3. DESIGN AND IMPLEMENTATION

Implanted chip technology is currently applied to develop life-enhancing and personal safeguard systems that can be powered by miniaturized sources. An early attempt to develop an implanted medical/tracking system intended for people is known as Digital Angel. This device, developed by Applied Digital System ADS [3], was initially developed to be implanted beneath the skin to monitor vital signs such as heartbeats and blood pressure. In addition the device provided a global satellite position system for people that suffer from brain disorders such as Alzheimer's disease. However, the system was modified to be worn externally and combines two pieces, a watch and a small beeper that clips onto a waist band. This device currently sells for $400 and has a $30 per month service fee.

Figure2. Digital Angel.

Nevertheless, the implanted chip concept was not lost; soon after ADS developed the Verichip [4]. A Radio Frequency Identification Device RFID the size of a grain of rice, (see Figure3) that is implanted under the skin. This small device consists of a microchip attached to an antenna and it is also known as a RFID tag. The tag contains a unique serial number that can be scanned by a base station, also known as a reader. In addition, the tag gets its entire power from the field generated by the reader therefore it does not need a battery. Although, this device does not collect any information, it contains a

unique verification number that can be used in a range of applications. These applications include identifying personal information in a medical database and security access applications.

Figure3. Verichip[1]

3.1. WHY RFID?

RFID technology is a great choice to apply to this project. As the Verichip, it can be implanted underneath the skin. It can get power from the field generated by the reader and it is fairly inexpensive. Tag cost from 20U.S cents to $6 and low-frequency readers, ideal for this project, can be under $100. In addition, to the tag and the reader, a low power consumption microcontroller and some sensors are needed to detect vital signs from the animal. In fact, RFID tags with sensors to detect temperature, movement, and even radiation are already being implemented. [5]

3.2. BACKGROUND

RFID is a term given to identification systems that use radio waves to communicate. The system consists of a microchip attached to an antenna know as a transponder or tag, and a

[1] In October 13, 2004 the Food and Drug Administration FDA approved the Verichip for health care.

reader or base station also attached to an antenna. RFID tags are divided into active and passive tags, where the main difference between them is the way they communicate with the reader.

The communication starts when the reader creates a RF field around the tag. When the tag detects this field it sends a signal back to the reader. Active tags broadcast their signal to the reader using low-power radios. To that end active tags require a battery. In the other hand, passive tags use either inductive coupling or backscatter. These communication systems and the fact that passive tags have a very simple circuit allow them to acquire their power completely from the field. As a result, passive tags could have a battery as a backup but they do not require one.

As mentioned before, one of the requirements of the system is a transmitter that does not require a periodic change of batteries. Therefore, the system to implement should have a passive tag.

The downside of passive tags is their communication range. Because of their power source, active tags have a longer range than passive tags. Active tags typically have a range of 60 to 300 meters, while the range for passive tags goes from few inches to 30 feet. For passive tags this range is influenced by the power output of the reader, the method used to power the tag and the operating frequency of the system.

Passive tags can work at different frequencies. These frequencies are divided into Low-Frequency, High-Frequency and Ultra-High Frequency. LF is defined from 30 to 300 kHz, tags usually operate at 125 kHz and134kHz. HF is defined from 3 to 30MHz. UHF is defined from 300 to 3GHz. UHF read range, 10 feet or more, is longer than LF and HF

read range, up to 3 feet. This is due to the method use to transmit data at different frequencies.

Passive UHF tags adjust the amount of energy reflected to the reader generating a signal; this method is known as *Backscatter*. Passive LF and HF tags use *Inductive Coupling*. When the coil of the reader antenna and the coil of the tag antenna form a magnetic field, energy is transferred from the reader to the tag. The tag uses this energy to change the electrical load on the tag antenna, changes that the reader can sense and convert into a signal. However to form a magnetic field the coils have to be rather close limiting the read range [6].

On the other hand, radio waves behave differently at each of these frequencies making some frequencies appropriate for some applications and inappropriate for other. As the frequency increases radio waves start to lose their ability to penetrate some materials. LF and HF systems work better than UHF systems around metal and water. In particular UHF radio waves are absorbed by water. As a result, passive UHF can send information farther and faster but they are not appropriate for applications with materials that high water content, such as skin.

As previously mention in the project requirements, the tag should be fully functional underneath the skin of the animal. Therefore, the passive tag should be either LF or HF. Due to the availability of the tags, a Passive Low-Frequency was chosen to be implemented in the project. The operating frequency of the system is 125 kHz.

3.3. HARDWARE

3.3.1. MICRO-TRANSPONDER

The micro-transponder in charge of collecting the data and sending it to the reader includes three elements. The transponder interface for microcontroller Atmel U3280M, a low power microcontroller Atmel Tiny 13v, and the antenna: a parallel LC resonant circuit. Block diagram:

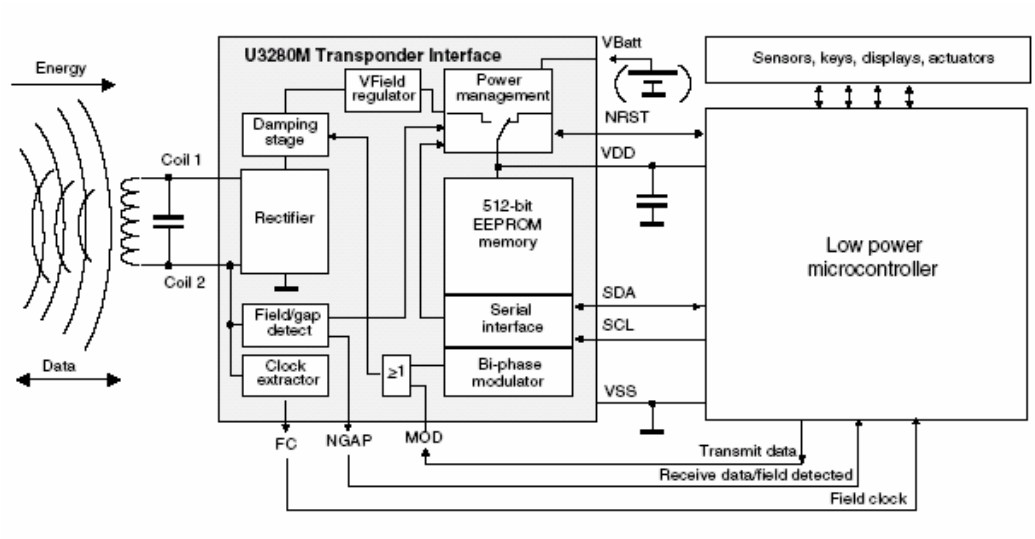

Figure4. Block diagram for Transponder and microcontroller. [7]

3.3.1.1. TRANSPONDER INTERFACE: ATMEL U3280M

The main reasons to utilize the U3280M in this project are: it can interact with a microcontroller, it is able to communicate wireless with a reader and it is capable of generating a DC power from the reader's electromagnetic field.

The U3280M is capable of wireless communicating with a reader. It contains a damping stage for transmitting and a gap-detection circuit for receiving data (see Firgure4). However, for this project the device is only going to be utilized to transmit data to the reader. To transmit the damping stage adjusts the coil voltage by varying its load

modulating the magnetic field as a result. The damping state can be controlled via the MOD pin or by the Two-Wire Serial Interface, also known as I2C Interface, which in addition provides Bi-phase or Manchester coding.

In an initial attempt to use the U3280M to transmit data to the receiver, the I2C was connected to an Atmel Mega32 microcontroller. The purpose of this trial was to determine how the I2C interacts with microcontrollers and having used the Mega32 previously made it a good choice before attempting to use the Tiny13. As its name indicates, the I2C contains two wires that connect the IC to the microcontroller (see Figure4); the SCL and the SDA lines. The SCL line is used to clock the data in and out of the device. The SDA line is a bi-directional line and it is used to transfer data in and out of the device.

Unfortunately this attempt was not successful. Although the U3280M I2C was designed to interact with other Atmel devices that have the same Serial Interface, during the modulation stage this device has a slightly different serial protocol. The regular I2C protocol includes an acknowledge cycle in the SDA line after 8 bits. However, when the Serial Interface is controlling the modulator stage the SCL and the SDA lines are used for continuous bit transfers and an acknowledge cycle after 8 bits must not be generated. Atmel response to this discrepancy was that the I2C interface is only recommended if an Atmel Marc4 microcontroller is used. In any other standard microcontroller, an acknowledge bit is always sent at the end of the transmission disturbing the modulation data stream. Atmel Marc4 microcontrollers are a great choice for wireless devices. However, they do not offer an Analog to Digital Converter (ADC), a part needed to collect the animal vital signs. Therefore, the MOD pin was chosen to control the damping

stage. The main disadvantage of using the MOD pin is that the Bi-phase/Manchester coding circuit offered by the U3280M can not be used. For this reason the data coding must be done by the microcontroller. A description of the data coding scheme and the algorithm to control the MOD pin can be found in the Tiny13's software part of the report.

As previously discussed the final wireless system should not require a battery inside the animal. The U3280M can be power by a battery but it can also generate a DC power from the reader's electromagnetic field. An internal rectifier stage (see Figure4) rectifies the AC from the LC-resonant circuit at the coil inputs and supplies the device and the Tiny13v with power. This rectified supply voltage from the coil is typically 2.9v. For normal operation it should be some where between 2.6v and 3.2v. In addition to having an ADC, the Tiny13v's operating voltage is between 1.8 and 5.5v making it a perfect candidate for the project.

To smooth and buffer the supply voltage a capacitor must be connected at the Voltage out pin (VDD). This buffer capacitor (CB) is also required to buffer the supply voltage during communication stages (damping and gaps). The size of the capacitor must guarantee that during modulation and gaps the ripple on the supply voltage is in the range between 100mv and 300mv. During communication stages the capacitor supplies voltage to the device, so its size depends on the length of these cycles. Starting from the value suggested in [7] an optimal value for CB was found. For a 350uA supply current, a 200mv voltage ripple and a no field supply length of 250us a value of 470nF was suggested, and for a no field supply length of 500us a value of 1000nF was suggested. A successful power supply was reached with a CB value of 1000nF.

In addition, the transponder offers a field clock (FC). This is a square wave from 0v to VDD at125kHz. The field clock is available to the microcontroller and in this project it is used to supply timer input to the Tiny13v to synchronize the algorithm to control the MOD pin.

See Appendix A for a complete diagram of the final micro-transponder circuit.

3.3.1.2. LOW POWER MICROCONTROLLER: ATMEL TINY13V

The main reasons to utilize the Tiny13v in this project are: it can process instructions efficiently at low power conditions, it offers an 8-bit Timer/Counter that can use an external clock source and it has an Analog to Digital Converter to gather the vital signs of the animal.

Atmel Tiny13v is a low power microcontroller based in the AVR RISC (Reduce Instruction Set Computing) architecture. This allows the microcontroller to approach a speed of 1 MIPS per MHz, allowing optimizing speed versus power consumption. In active mode, its lowest power consumption is 1.8v:240μA at 1MHz. It can go as fast as 4MHz at 1.8-5.5v, and 10MHz at 2.7-5.5v. For this project the main objective is to limit the power consumption of the micro-transponder to a minimum to increase its range of transmission. For that reason the processing speed of the Tiny13v was selected to be 1 MHz.

The Tiny13v provides an 8-bit Timer/Counter that can use an external clock source. The 8-bit Timer/Counter includes a programmable bi-directional counter unit that can be cleared, incremented or decremented at each timer clock (clkT0). The timer clock can be either an internal or external clock source. The transponder field clock (FC) is utilized as

the external clock source. This way the field clock can be captured. In addition, the Timer/Counter also contains an Output Compare Unit. An 8-bit comparator continuously compares the value of the counter with the value of an Output Compare Register. A match can generate a hardware interrupt. This synchronized hardware interrupt is then used to control the MOD pin. More details of the algorithm are discussed in the Tiny13v's software discussion.

An important feature of the Tiny13v for this project is its ADC. The ADC converts an analog input voltage to a 10-bit value through successive approximation. The minimum value represents GND and the maximum value represents either Vcc or an internal 1.1v reference voltage provided by the Tiny13v. In this case 10-bit precision is not required and only the 8 most significant bits of the ADC conversion result are used. Also, the internal reference voltage (Vref) was selected as the maximum value of the conversion.

The ADC circuit requires an input clock between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution is needed a faster input clock can be implemented to get higher sample rate. In this case only 8-bit resolution is needed so the input clock frequency can be a higher than 200 kHz. The clock frequency is controlled by the ADC prescaler that scales the clock of the microcontroller. The prescaler was set to 4 to obtain a 250 kHz input clock for the ADC. That is the processing speed of the microcontroller (1MHz) divided by 4.

The result of a single conversion is:

$$ADC = \frac{Vin \bullet 2^n}{Vref} \qquad (1)$$

Where Vin is analog voltage to convert, n is the number of bit resolution needed in this case 8 and Vref is the internal reference voltage 1.1v. This value is then coded and sent to

the reader using the modulation stage in the transponder controlled by the microcontroller. Once the reader receives the ADC value it converts it back to Vin using equation (1).

The ADC absolute accuracy described as the maximum deviation of an actual ADC value to the ideal ADC value due to offset error, gain error, differential error, and other errors described in [8] is the ideal value ± 0.5 LSB. Since only 8-bit precision is required the two LSB are not used and this error can be ignored.

To calculate the Hear Rate of the animal the voltage in the muscles is measured. This can be accomplished by using the Tiny13v's ADC directly. However, to calculate the temperature a temperature sensor is needed.

3.3.1.3. TEMPERATURE SENSOR LM61

A low operating voltage temperature sensor from National Semiconductor, it can operate at 2.7v. It can sense a -30°C to +100°C temperature range. Its output (Vo) is linearly proportional to +10mV/°C and has an offset of +600mV. That is:

$$Vo = \left(\frac{+10mV}{°C} + T°C \right) + 600mV \qquad (2)$$

Where Vo is the input of the ADC converter and T is the actual temperature of the animal in centigrades. Equation (2) is also used in the receiver to calculate the actual temperature. At 25°C the accuracy of the sensor is ±2.0 or ±3.0°C.

In addition, a low-pass filter was added to Vo before applying the signal to the ADC to avoid distortion from high frequency components. A simple RC low pass filter was build with a 1.5 kΩ resistor and a 0.1uF capacitor. The final circuit layout can be found in Appendix A.

3.3.1.4. TRANSPONDER ANTENNA

The antenna for the interface must be a coil. In addition, the coil must be connected with a capacitor in parallel to form a LC parallel resonant circuit, see Figure4. The resonance frequency of this LC circuit should be in the range from 100 kHz to 150 kHz. For this project the exact operating frequency of the system is 125 kHz. Hence the LC circuit should resonate at 125 kHz. The correct LC combination can be calculated using the following formula:

$$L = \frac{1}{C * (2 * \pi * fo)^2} \qquad (3)$$

Where fo is 125 kHz, the operating frequency.

When the micro-transponder is working without a battery, the operational working distance can be represented as the minimum coupling factor of the antenna for proper operation. The coupling factor depends on the power consumption of the micro-transponder and on the size of the antennas of the micro-transponder and the reader. If the power consumption is less or equal to a current of 150μA at the operating voltage, a minimum magnetic coupling factor below 0.5% is within reach. For applications with higher power consumption the coupling factor must be increase. The transponder has typical operating current during field supply of 40μA, the Tiny13v has a minimum operating current of 240μA and the temperature sensor has a minimum operating current of 10μA. Therefore, the coupling factor must be increase to provide enough power for the micro-transponder.

The reader's antenna should be a large size antenna to cover a significant area where the animal can move freely. The transponder's antenna design approach was to find the necessary inductance to obtain a high coupling factor between the antennas and then use

equation (3) to find the corresponding capacitor. First a rather large air-cored coil was built for the transponder, a coil of the same size of the reader's coil, to obtain a high coupling factor. Once the transponder was generating enough power to send data to the reader and the reader was receiving it correctly, the next step was to design a smaller coil with the same inductance capacity of the first one.

The inductance of the coil is controlled by four different factors and can be described by the following equation:

$$H = \frac{(4 * \pi * NoTurns\text{^}2 * coilArea * mu)}{(coilLength * 10{,}000{,}000)} \qquad (4)$$

Where NoTurns is the total number of turns wrap around the core, coilArea is the cross-sectional area of the coil, mu is the permeability of the core of the coil, and the coilLength is the final length of the coil.

Building a smaller coil requires to decrease its cross-sectional area, decreasing its inductance. To maintain the same inductance in the smaller coil either the total number of turns of the coil or the permeability of the core should be increased. Since the objective is to minimize the size of the antenna increasing the permeability of the coil is a better option than increasing the number of turns. Ferrite is a ceramic material that offers a great core choice. The permeability of ferrite ranges from 20 to more than 15,000 depending on the material it is made from and operating conditions such as temperature, field strength and frequency. Giving that the operating frequency is 125 kHz, the best choice for this application is to use a ferrite core made from manganese-zinc that offers a high permeability factor for the 1 kHz to 1 MHz frequency range. This material has a permeability of 800, while air has a permeability of 1.

An initial attempt to build a ferrite cored coil was made. Using equation (4) to find the exact number of turns for a given inductance, cross-sectional area, permeability and coil length was found, and then the insulated wire was manually wound into the ferrite core. Although, when attached to the transponder this coil proved to have almost enough inductance to power the micro-transponder, this significantly smaller coil (see Appendix A for picture) was still not small enough to fit inside an animal. Several attempts to decrease the size of this coil were made, but they all turn out to be unsuccessful.

After looking more closely at the different options for antennas in the RFID industry, Coilcraft, a company that manufactures coils, transformers and other magnetic products, turned out to produce the better coil for this application.

The Coilcraft 4308RV Series [10] of RFID Transponder coil for applications at 125 kHz is a very small coil that provides the needed inductance. In addition, its ceramic/ferrite laminate construction is more even than pure ferrite coils allowing longer read ranges.

3.3.2. READER

The reader in charge of generating a RF magnetic field and to receive and decode the data from the micro-transponder includes three devices: A read/write base station Atmel U2270B, a powerful microcontroller Atmel Mega32, and the receiver's antenna. Block diagram:

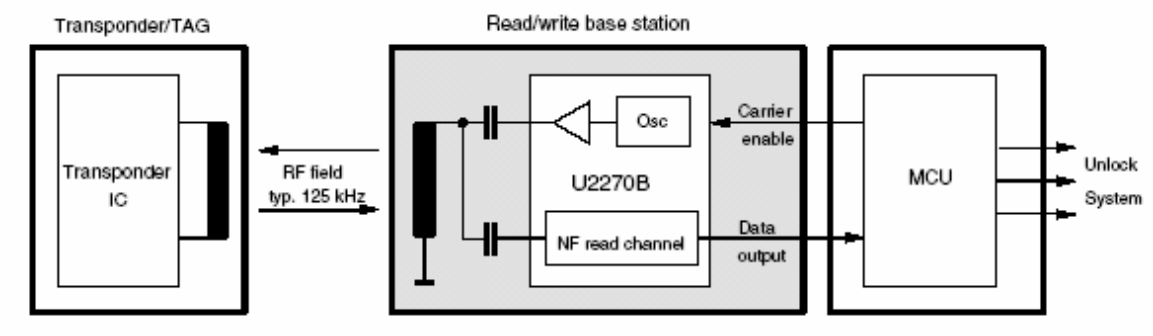

Figure5. Block diagram for the base station and microcontroller. [11]

3.3.2.1. READ/WRITE BASE STATION: ATMEL U2270B

The U2270B is an IC for read/write base stations in contact-less identification devices. For this application the base station is not going to write data to the transponder, it is only going to read from it. The main reasons to utilize the U2270B in this project are: it has the energy-transfer circuit to generate the magnetic field to supply power to the micro-transponder and it includes it includes all signal-processing circuits necessary to transform the small data signal from the transponder into a microcontroller-compatible signal.

The energy-transfer circuit consists of an internal power supply, an oscillator and a coil driver. The frequency of the oscillator is controlled by a current fed into the RF input (pin 15). The internal power supply ensures an independent frequency that can be selected by a fix resistor between the RF pin input and the VS pin, the internal power supply pin. For 125 kHz a resistor value of 110 kΩ is defined. The coil driver supplies the antenna coil with the appropriate energy to generate the magnetic field. It consists of two independent outputs Coil 1 (pin 9) and Coil 2 (pin 8). The coil driver can be operated in two different modes, common mode and differential mode. In common mode the Coil 1 and Coil 2 are

in phase to achieve a high-current output capability, while in differential mode Coil 1 and Coil 2 are in anti-phase to obtain a higher voltage. In differential mode the antenna coil impedance is higher resulting in better sensitivity. Hence the Differential was the mode implemented in the base station.

The signal-processing circuitry includes a low pass filter LPF, a differential amplifier and a Schmitt Trigger. The LPF is a fully-integrated 4^{th} order Butterworth low pass filter that removes the remaining carrier signal and high-frequency disturbances after demodulation. Its cut-off frequency depends on the operating oscillator frequency. In this case the oscillating frequency is 125 kHz and the cut off frequency is 6.9 kHz. The differential amplifier has a typical fixed gain of 30. It also has an HIPASS input (pin 16) that can be used to decouple the dc signal. For the 6.9 kHz cut off frequency a 220nF capacitor should be place between the HIPASS input and ground. Lastly, the Schmitt Trigger suppresses possible noise and makes the signal compatible with the microcontroller.

An amplifier circuit was added to increase the power of the magnetic field to provide more power to the transponder and to increase the range of the system.

See Appendix A for a complete diagram of the final base station circuit.

3.3.2.2. MICROCONTROLLER: ATMEL MEGA32

The Mega32 was chosen as the base station microcontroller because its speed and its numerous peripheral features that offer many tools to recover and successfully decode the information send by the transponder.

Atmel Mega32 is a high-performance, low power AVR 8-bit microcontroller that uses advance RISC architecture, same architecture mentioned in the Tiny13v description. It is operating voltage is 4.5 to 5.5v and its speed grades go from 0 to 16 MHz. For this application 5v was selected as its operating voltage and 16MHz its speed.

The most useful tools provided by the Mega32 to decode the data in the receiver are two Timer/ Counters and a programmable serial interface (USART). These counters work in the same way that the Tiny13v counter works. Here one of them uses an 8-bit register and the other a 16-bit register. Both timers are synchronized to detect the clock of the signal coming from the micro-transponder. Once the clock has been detected the next step is to sample the signal, decode it and output it through the interface.

The serial interface used to output the signal is the Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART). The USART is a very flexible communication device. For this application the data is transmitted to a computer through the USART via the RS232 communications standard. More details of how the Mega32 samples and decodes the data coming from the receiver along with the algorithm are discussed in the Mega32's software discussion.

3.3.2.3. BASE STATION ANTENNA

For the base station the antenna is also a coil, and as well as in the transponder's antenna the coil must be connected with a capacitor in parallel to form a LC parallel resonant circuit. The resonance frequency of this LC circuit should also be the operating frequency of the system 125 kHz.

As previously mentioned the reader's antenna should be a large size coil to cover a significant area where the animal can move freely. The designing approach for the base station antenna was to first design an antenna with a large coil and then increase or decreases its size to maximize the power generated by the transmitter.

The first antenna provided enough power to the transmitter when the transmitter had an antenna of the same size. However, when the coil of the transmitter was minimized the magnetic field provided by the first coil was not enough to power the micro-transmitter. To increase the effect of the magnetic field in the transponder's coil with less turns was build for the base station. Decreasing the number of turns of the coil increases the current in the coil providing a stronger magnetic field to the transponder.

All the coils designed for the base station were air-cored and consisted of hand wound insulated wire. See Appendix A for exact coil sizes, number of turns and pictures.

3.3.2.4. DEVELOPMENT BOARD: ATMEL STK500

The STK500 is a complete starter development system for AVR Flash Microcontrollers from Atmel Corporation. Both microcontrollers, the Tiny13v and the Mega32, were programmed and tested in the STK 500. In addition the STK 500 is compatible with CodeVisionAVR, software use to program the algorithms in the microcontrollers.

3.4. SOFTWARE

3.4.1. CODEVISIONAVR

CodeVisionAVR was used to edit, assemble, compile and program the microcontrollers. CodeVisionAVR is a C cross-compiler, Integrated Development Environment and Automatic Program Generator designed for the Atmel AVR family of microcontrollers.

3.4.2. MANCHESTER CODE

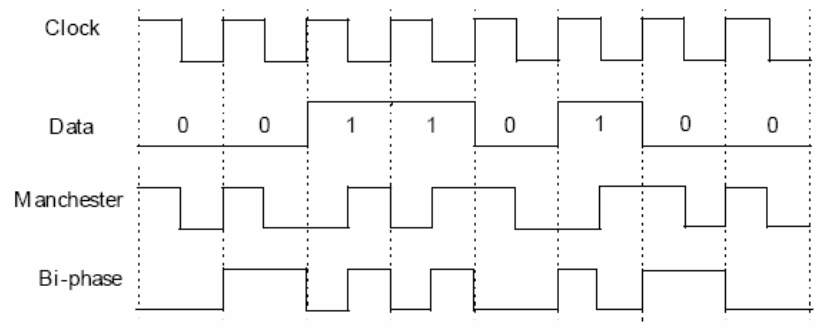

Figure6. Manchester and Bi-Phase Code [13]

Once the data is collected by the microcontroller, it is coded transmitted through the air, received by the reader and decoded. The Base Station is equipped to receive two different codes, Manchester Code and Bi-Phase Code. These two codes have the ability to transmit both the clock and the information at the same time. As shown in Figure6, both codes use two bits to represent one bit of data. The Manchester code represents a zero data bit as a one follow by a zero, and a one data bit as a zero followed by a one. The Bi-phase code represents a zero data bit as two identical bits, either two zeros or two ones. And a one data bit as two different bits, either a one follow by a zero or a zero follow by a one.

Both codes have advantages and disadvantages. Since Manchester code requires a transition either from high to low or from low to high for every data bit it makes the clock

recovery an easier process than the Bi-phase code. In the other hand, once the clock has been recovered, Bi-phase code makes a better distinction between logic zeros and ones making the final part of the decoding process more efficient and effective than Manchester code.

For this project the timing in the decoder code is affected by various elements such as modulation effects and channel noise. Therefore, Manchester code was chosen since it makes simpler the clock recovering process.

3.4.3. TRANSPONDER CODE

The transponder code was designed for the ATiny13v microcontroller. When the microcontroller is powered, it detects the field clock signal generated by the transponder interface. This 125 kHz square wave was used to synchronize microcontroller and code the data collected by the temperature sensor. The code detects the field clock, collects the data, codes it and controls the MOD pin in the transponder interface to send the data to the base station.

The data is sent from the transponder to the base station repeating the following sequence over and over again: three sync bytes follow by one hundred data bytes. The three sync bytes are one byte of logic zeros (0), and two bytes of logic ones (255). The sync bytes help the base station recover the clock. To increase the accuracy of the clock recovery process a data byte composed entirely of ones (255) is not allowed.

The program flow chart is shown in Figure7. When the field clock is in the rising edge an ISR starts. The sub-routine increases a counter, and checks if the program has finish sending a byte. Then the program codes and sends the data one bit at a time. Since

Manchester code requires two bits per data bit, sending one data bit requires accessing the interrupts twice. A byte has eight bits, therefore when the counter reaches 16, the entire byte has been sent and a new measurement can be collected.

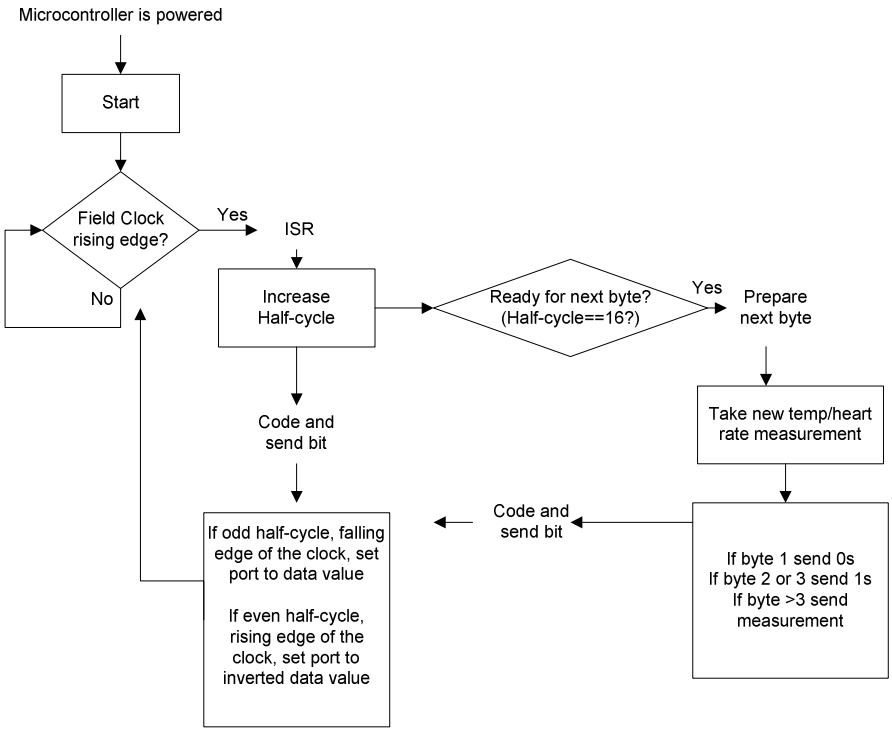

Figure7. Transponder code flow chart

3.4.4. BASE STATION CODE

The base station code was designed for the ATMega32 microcontroller. The microcontroller receives the data coming from the transponder through the Atmel U2270B. The output of the Atmel U2270B is a 2 kHz square wave from 5v to 0v. The code recovers the clock, finds the data, decodes it and displays it in Matlab through the USART interface and Matlab GUI.

The square wave is connected to the microcontroller external interrupt. At the edge of the wave an interrupt is triggered and a physical counter in the microcontroller is set to count

the time that goes by until the next edge. When the time between edges equals a full original clock cycle, about 513us, the clock is recovered. Once the clock has been recovered the code takes samples of the square wave and looks for the sequence described in the transponder code section; one byte of 0s followed by two bytes of 1s. Once this sequence is found the next one hundred bytes are considered data and sent to the interface.

This program and the Matlab GUI were developed by Dan Golden [12].

An attempt to modify the code to add a Digital Phase-Locked Loop PLL to increase the quality of the data was made. The goal was to use the DPLL to regenerate or recover the clock and lock to it. This modification would eliminate the need for the sync bytes every one hundred data sets. However, this effort was not successful due to the complexity of the code.

3.4.2. MATLAB GUI

A Matlab GUI was written to make the data collecting simpler for the user. The GUI was developed using Matlab's GUIDE feature and to run properly it requires Windows XP and works best with Matlab 7.0 or above. The GUI allows the user to interact with the device to collect data. The user can select between collecting a number of consecutive samples and letting the device collect as many samples as possible for a given period of time. In addition, the GUI displays the data points as they are acquired in real time, it allows the user to start and stop the acquisition process at any time, and the data points can be saved easily as data sequences in Matlab for further manipulation. A screen shot of the GUI can be found in Appendix A.

4. RESULTS

The system was first tested using antennas with the same diameter (Figure14 antenna as the Base Station antenna and Figure15 antenna as the transponder antenna, see Figures in Appendix A). This first test was successful. The transponder was able to power the microcontroller, collect and send information to the base station. Also, the base station successfully received, decoded and displayed the data in the GUI. Then the range of the system was determined by moving the antennas away from each other while monitoring the performance of the system. The system performed well until the antennas were about half foot away.

The next step was to determine the performance of the system using the small Ferrite Cored antenna (Figure17) in the transponder. This test was not as successful as the first one. This time the transponder could not power the microcontroller. In an attempt to increase the power of the base station's magnetic field, the antenna in the base station was replaced by an antenna with less turns and the same diameter, antenna shown in Figure15. Decreasing the number of turns in the coil increases the current going through the coil making the magnetic field more intense. This increased the voltage at the transponder from 1v to about 1.6v. However, this was still no enough since the Tiny13v microcontroller requires at least 1.8v to function. The transponder was then taken apart and the transponder interface responsible of generating the power was tested alone.

When tested independently the transponder interfaced performed better. A voltage of 2.8v was registered along with a 125 kHz field clock as shown in Figure8. But, when the interface was connected back to the microcontroller the voltage dropped to 1.6v. It was concluded that when using the small ferrite cored antenna the transponder interface does

not generate enough power for the entire micro-transponder. A way to deal with this problem would be to add a small battery to the microcontroller that would only be used when the transponder interface reaches a 2.8v. That way the battery will last a long time, preventing a periodic replacement.

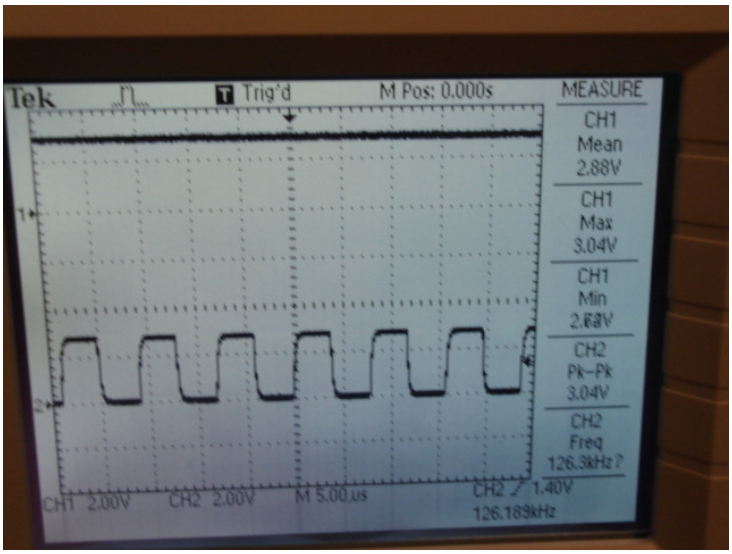

Figure8. Voltage and Field Clock at the Transponder Interface using the small Coilcraft's RFID Antenna

The final cost of the base station was $18.56 and the final cost of the transponder was $6.65, adding up to a grand total of $25.21. The final cost break downs for both devices can be found in Table1 and 2 in Appendix B.

The final size of the transponder is determined by the size of the transponder interface and the microcontroller. The transponder interface is a 5x5mm square with 1.7mm wide. The Tiny13 microcontroller can be found in two different packages. The smallest package is called S8S1 and it is a 4.95x 3.81mm rectangle with 0.25mm wide. The other two main components of the transponder, the antenna and the temperature sensor, do not affect the size of the device since they are significantly smaller than the interface and the microcontroller.

5. CONCLUSIONS

The objective of this project was to use RFID technology to develop a device to monitor animal vital signs. The device was required to be wireless, inexpensive, small enough to fit underneath the skin of the animal and the transponder should not require a periodic change of batteries. RFID technology provided a great alternative to monitor animal vital signs.

The first requirement was for a wireless device. Although the final transponder could not generate enough power to collect and send data to the base station, the results found in this project do encourage the further exploration of RFID technology in this area and other contact-less applications. The initial results proved that when the transponder generates enough power data is collected and successfully sent to the base station. Therefore, further work on this area should include looking for a more effective antenna for the transponder or ways to build a more powerful magnetic field at the base station.

Another approach would be to the addition of a battery to supply power to the microcontroller when the base station is collecting data from transponder. Since the battery would only be on when the base station is actually collecting data from the transponder it will not require periodic changes.

The second requirement was to design an inexpensive device. The total cost of the device was very low, $25.21, giving it a great advantage against other similar products such as the VitalScal from Vetronics that as previously discussed cost about $500.00. In addition, the cost of the transponder alone is $6.65, and since the base station can communicate with many transponders having one base station and many transponders can be a very inexpensive way to monitor several animal for a very low price.

The last requirement was to design the transponder small enough to fit underneath the skin of the animal. Once it has been soldered and package into a whole, from the dimensions of the main parts of the transponder it is fair to say that the final transponder should be very small, smaller than 1cm cube, and should fit underneath the skin of a small animal such a mouse.

6. ACKNOWLEDGMENTS

I would like to thank Bruce Land for all the support and advice throughout the year. Also I would like to acknowledge Dan Golden for all his hard work developing the base station code and the GUI.

7. REFERENCES

[1] Vetronics. <http://www.vetronics.com>

[2] "Monitoring Vital Signs in Clinical and Research Animals" by Janice M. Brigth. On VitalScan Monitor® by Vetronics, Inc. < http://www.currentseparations.com/issues/16-2/cs16-2b.pdf>

[3] Applied Digital Solution. <http://www.adsx.com>

[4] Verichip Official Websitehttp://www.4verichip.com/

[5] "RFID Sensors: From Battlefield Intelligence To Consumer Protection", RFID Journal. <www.rfidjournal.com>

([6] "$1 Wireless Interface" by Larry Martin, Circuit Cellar, Issue 163: 50,51, February 2004.)

[6] Getting Started, RFID Journal. <www.rfidjournal.com>

[7] Atmel U3280M Datasheet

http://www.atmel.com/dyn/resources/prod_documents/doc4688.pdf

[8] Atmel Tiny13v Datasheet

http://www.atmel.com/dyn/resources/prod_documents/doc2535.pdf

[9] Temperature Sensor LM61 Datasheet

http://www.national.com/ds/LM/LM61.pdf

[10] Coilcraft 4308RV Series RFID Transponder Coils.

http://www.coilcraft.com/ds/4308rv.pdf

[11] Atmel U2270B Datasheet

http://www.atmel.com/dyn/resources/prod_documents/doc4684.pdf

[12] Dan Golden's Project Website

http://instruct1.cit.cornell.edu//courses/eceprojectsland/STUDENTPROJ/2004to2005/dig4/dig4rfid.htm

[13] Atmel Electronic Immobilizers for The Automotive Industry

http://www.atmel.com/dyn/resources/prod_documents/doc4661.pdf

8. APPENDICES

8.1. APPENDIX A: SCHEMATICS AND PICTURES

8.1.1. BASE STATION

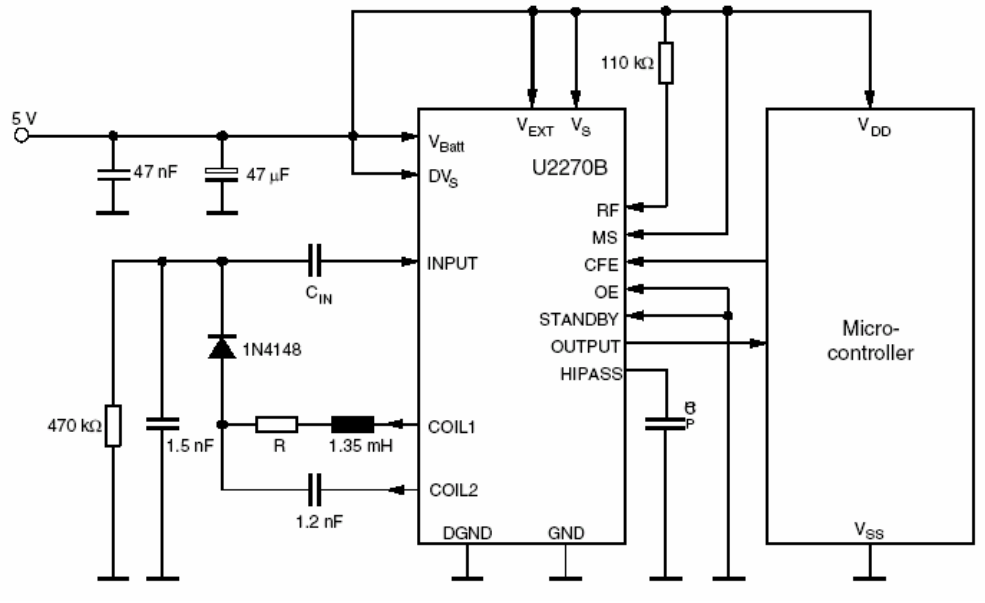

Figure9. Base Station Schematic

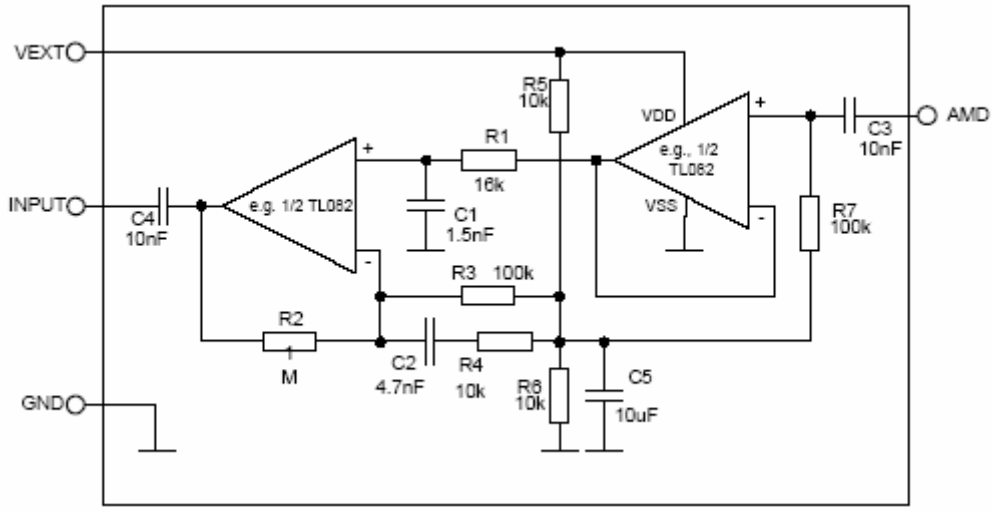

Figure10. Preamplifier added to the Base Station

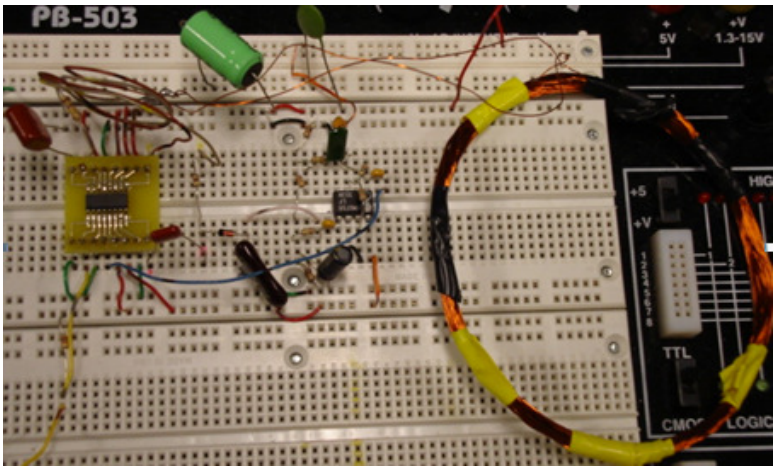

Figure11. Base Station

8.1.2. TRANSPONDER

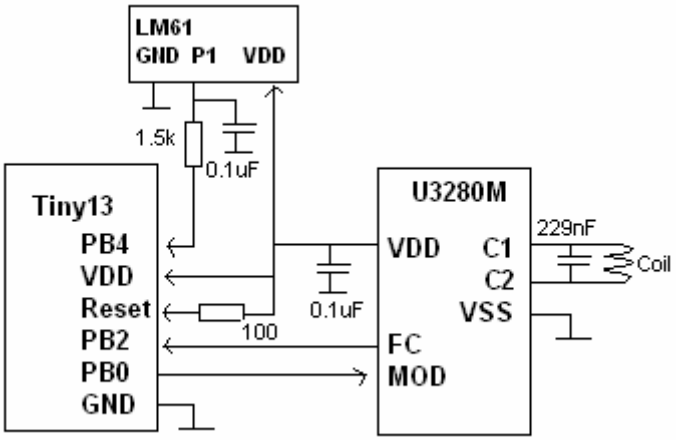

Figure12. Transponder Schematic

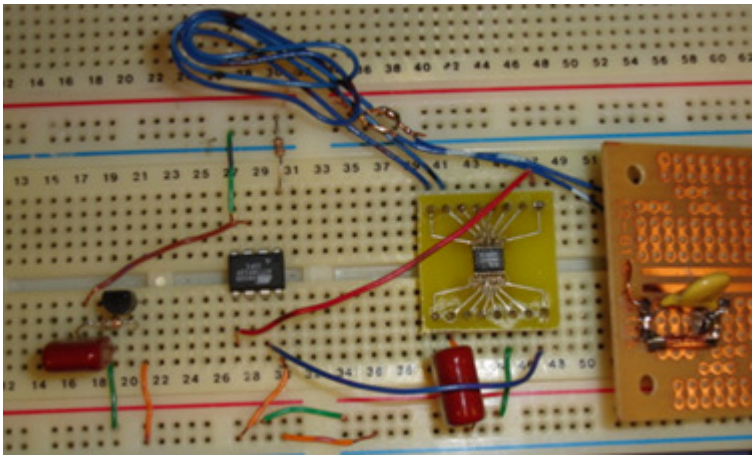

Figure13. Transponder

8.1.3. ANTENNAS

Figure14. Initial Base Station Coil
Resonates at ~125 kHz with a 1.1 nF capacitor
Diameter: 10cm or 3.9 inches
Wire Size: 0.333mm or 0.39 inches
No of Turns: 125
Estimated Inductance: 1.625 mH

Figure15. Initial Transponder Coil and Final Base Station Coil
Resonates at ~125 kHz with a 2.35 nF capacitor
Diameter: 10cm or 2.3 inches
Wire Size: 0.333mm or 0.39 inches
No of Turns: 67
Estimated Inductance: 689 uH

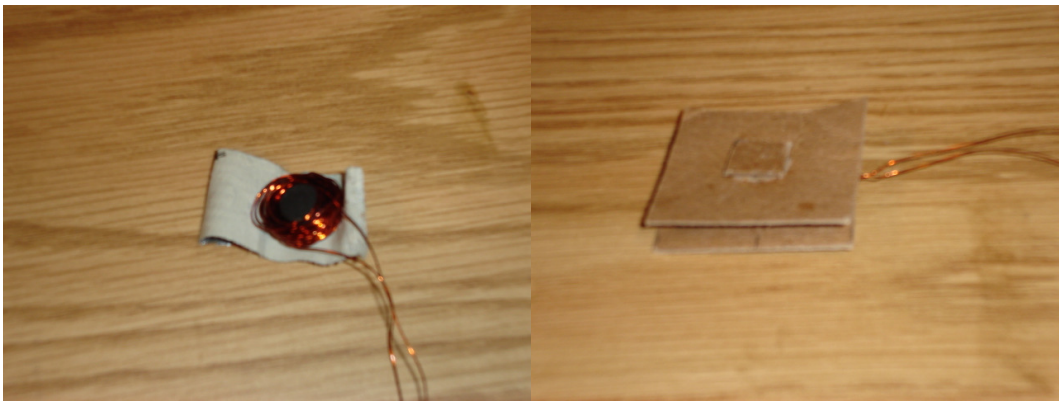

Figure16. Attempts to build a Ferrite Cored Coil for the Transponder

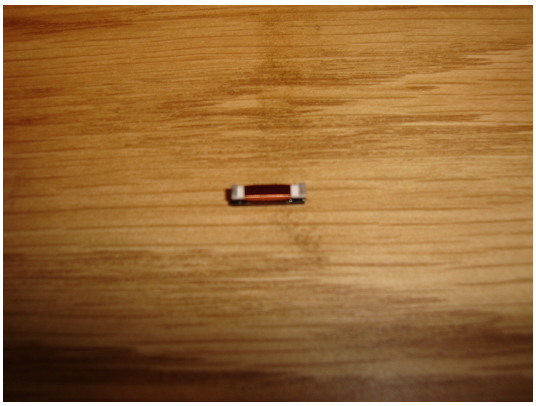

Figure17. Final Ferrite Cored Coil for the Transponder from Coilcraft [10] Resonates at ~125 kHz with a 294 nF capacitor

8.1.4. MATLAB GUI

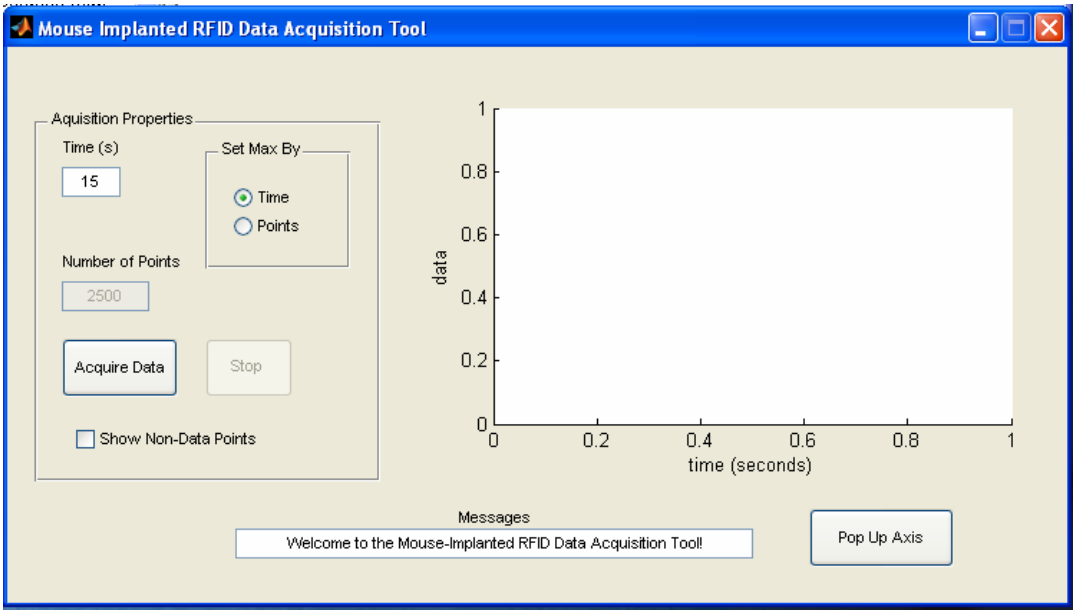

Figure18. GUI screen shot [12]

8.2. APPENDIX B: COST

Part Number	Description	Cost/Item	Quantity	Total Cost
U2270B	Base Station	$2.58	1	$2.58
ATMega32	Microcontroller	$8.00	1	$8.00
1N4148	Diode	$0.30	1	$0.30
LF353	Dual Operational Amp	$0.45	1	$0.45
P10980-ND	Capacitor (1.2nF)	$0.78	2	$1.56
495-1630-ND	Capacitor (1.5nF)	$0.50	2	$1.00
P4500A-ND	Capacitor (4.7nF)	$0.38	1	$0.38
399-1945-ND	Capacitor (47nF)	$0.48	2	$0.96
478-2265-ND	Capacitor (0.22uF)	$0.14	1	$0.14
495-1636-ND	Capacitor (10uF)	$0.65	3	$1.95
BC100YCT	Resistor (100Ω)	$0.19	1	$0.19
BC10.0YTR	Resistor (10KΩ)	$0.15	4	$0.60
100XBK-ND	Resistor (100kΩ)	$0.10	2	$0.20
110KXBK-ND	Resistor (110kΩ)	$0.10	1	$0.10
B0207C1M000F5T-ND	Resistor (1M)	$0.15	1	$0.15
			Total:	$18.56

Table1. Base Station Total Cost

Part Number	Description	Cost/Item	Quantity	Total Cost
U3280M	Transponder	$2.10	1	$2.10
ATTiny13v	Microcontroller	$1.40	1	$1.40
LM61	Temperature Sensor	$1.04	1	$1.04
4308RV	RFID Coil	$1.13	1	$1.13
B32922A2104M	Capacitor (0.1uF)	$0.29	2	$0.58
478-2038-ND	Capacitor (294nF)	$0.11	1	$0.11
BC100YCT	Resistor (100Ω)	$0.19	1	$0.19
1.5KETR-ND	Resistor (1.5kΩ)	$0.10	1	$0.10
			Total:	$6.65

Table2. Transponder Total Cost

8.3. APPENDIX C: SOURCE CODES

8.3.1. BASE STATION CODE

```
/*************************************************************
 *****          Mouse-Implanted RFID Project            *******
 *****               Base Station Code                  *******
 *****                                                  *******
 *****                 By Dan Golden                    *******
 *****                  09/11/2004                      *******
 *************************************************************/


/* This code is for the Atmel Mega32 Microcontroller */

/***********************************
 *****    Port Description    *******
 ***********************************

Inputs:
D[2]              Base Station Data Output

Outputs:
C[0:7]                  LEDs (for testing)


/***********************************
 *****    Encoding Notes      *******
 ***********************************

Manchester encoding is used to communicate from the transponder to the
base station. Traditionally, in manchester encoding, the data ALWAYS
contains an edge on the clock's falling edge; a logic 1 is a rising
edge, a logic 0 is a falling edge. However, this base station INVERTS
the received data; hence, logic 1 is a falling edge on the falling
clock edge, and logic 0 is a rising edge on the falling clock edge.

Ex.

            _   _   _   _   _   _   _   _
CLK          |_| |_| |_| |_| |_| |_| |_| |_|
            _   _     _   ___     ___   _
DATA Sent    |_| |___| |_|   |___|   |_| |_
              _   ___   _     ___     _   _
DATA Rec'd   _| |_|   |_| |___|   |___| |_|

           | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Obviously, the received data is what is parsed.

Clock Periods of Note:
Width of one half clock period:     256 us      =     64 CLK/64 cycles
Width of one full clock period:     513 us      =     128 CLK/64 cycles
Width of four clock periods (1/2 byte):2.05 ms  =     512 CLK/64 cycles
*/
```

```
/************************************
 *****            Includes     *******
 ************************************/

#include <Mega32.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/************************************
 *****        Definitions     *******
 ************************************/

#define dataInput PIND.2
#define debugOut PORTA.0
#define LEDs PORTC

/************************************
 *****     Global Variables   *******
 ************************************/

unsigned char count;          // General indexing variable

unsigned char bitStream[3];   // Used to parse input data
unsigned char bitNumber, byteNumber;
unsigned char lastEdge, thisEdge, CLKedge, getData;

unsigned char data;           // The data byte received at the Base
                              // Station from the transponder
unsigned char dataReady;      // True when the data is ready
unsigned char junkByte;       // True if the current byte is invalid
unsigned char junkCounter;   //#ofsuccessive junk bytes since last sync
char sync;                    // Sync level
                        // 0: not syncing (receiving data)
                        // -1: not synced (waiting for sync pulse)

/************************************
 *****   Function Prototypes  *******
 ************************************/

void initialize(void);

/************************************
 *****       Interrupts       *******
 ************************************/

/* This interrupt is used only for FINDING the clock from the initial
clock sync byte. Parsing DATA is done separately. */
interrupt [EXT_INT0] void data_edge(void)
{
      thisEdge = dataInput;

      // If the detected period is too low (T < 200 us), then label
      // this byte as junk.
      if(TCNT1 < 50)
            junkByte = 1;           // Label the byte as junk.
```

```
// 200 us < T < 384 us. This period occurs after a clock edge or
a same-bit
// boundary (e.g., 1-1 or 0-0).
else if (TCNT1 < 96)
{
      // If we're at a clock edge...
      if (CLKedge)
      {
            CLKedge = 0;
            getData = 1;
  }
      else
            CLKedge = 1;
}
// 384 us < T < 600 us.
// This period occurs after an opposing-bit boundary ONLY.
else if (TCNT1 < 150)
{
      CLKedge = 0;
      getData = 1;
}
// If the pulse period is too different from any legal values,
the byte is junk.
else
{
      junkByte = 1;
}

// Reset timer1
TCNT1 = 0;

// If we're on a rising clock edge, it's time to get the data
if (getData)
{
      getData = 0;

      // Shift bitstream down
      bitStream[0] >>= 1;
      bitStream[0] |= (bitStream[1] << 7);
      bitStream[1] >>= 1;
      bitStream[1] |= (bitStream[2] << 7);
      bitStream[2] >>= 1;

      // Insert the most recent bit into bitStream
      bitStream[2] |= ((!dataInput) << 7);

      bitNumber++;
}

// If we have sync, and we've just updated bitStream and
bitStream has
// all 8 bits of data, then read the data. If the data has been
// marked as junk for any reason, then set it to 0xff.
if ((~CLKedge) && (sync != -1) && (bitNumber == 8))
{
      // If the data is junk, make it 0xff.
      if (junkByte)
```

```
            {
                  data = 0xff;
                  junkCounter++;
            }
            else
            {
                  data = bitStream[2];
                  junkCounter = 0;
            }

            byteNumber++;

            // Force a sync every 100 data bytes, or if we get three
            junk
            // bytes in a row.
            if ((byteNumber == 100) || (junkCounter > 2))
            {
                  sync = -1;
                  TCNT0 = 0;
                  TIMSK |= 0b00000010;    // Send 0xff's while syncing
            }

            dataReady = 1;
            bitNumber = 0;
            junkByte = 0;
      }
      // If we're searching for the sync stream...
      else if ((~CLKedge) && (sync == -1))
      {
            // If we've found the sync, stop searching and get ready
            for data.
            if ((bitStream[0] == 0x00) && (bitStream[1] == 0xff) &&
                  (bitStream[2] == 0xff))
            {
                  sync = 0;
                  TIMSK &= 0b11111101;    // Stop sending 0xff's
                  bitNumber = 0;
                  byteNumber = 0;
            }
      }
}

interrupt [TIM0_COMP] void timer0_compare(void)
{
      data = 0xff;
      dataReady = 1;
}
```

```
/************************************
 *****          Main           *******
 ************************************/

void main(void)
{
      initialize();

      while(1)
      {
            if (dataReady)                // If we have received
data...
            {
                  LEDs = ~data;
                  dataReady = 0;          // Prepare for new data
                  printf("%c", data);     // send data to USART
            }


            if (TIFR & 0x04)        // If timer 1 has overflowed (T =
0.26 sec)...
            {
                  TIFR = TIFR | 0x04;     // Clear TOV1.
                  sync = -1;              // Start syncing.
                  TCNT0 = 0;
                  TIMSK |= 0b00000010;    // Send 0xff's while syncing.
            }

      }
}


/************************************
 *****     Initialization     *******
 ************************************/

void initialize(void)
{
      /**** Port initialization ****/
      DDRD = 0x00;      // D.2 is data input
      DDRA = 0x01;      // A.0 is debug output
      DDRC = 0xff;      // Port C is LED output
      LEDs = 0xff;      // All off initially

      /**** Interrupt initialization ****/
      MCUCR = 0b00000001;     // Any edge on INT0 triggers interrupt
      GICR = 0b01000000;      // Enable INT0

      /* Timer 1 is used for synchronizing with the transponder's data
      clock, which should run at around (125e3/64) = 1.95 KHz, which
      means that edges appear at 3.91 KHz (no opposing bit boundary) or
      1.95KHz (at opposing bit boundaries). */
      TCCR1B = 0b00000011;    // CLK/64 (250 KHz)

      /* Timer 0 is used to output 0xff data points at the same
      frequency as the usual data rate when the base is searching for
      the sync stream. */
```

```
        TCCR0 = 0b00001101;           // CTC on, CLK/1024 (15.6 KHz)
        OCR0 = 65;        // f = 240 Hz (almost 238.5 Hz, the data rate)
        TIMSK = 0b00000010;           // Enable OCIE0

        /**** USART initialization ****/
        UCSRB = 0b00011000 ; // TXC, RXC enabled
        UBRRL = 16 ; // 57.6 kbaud @ 16MHz

        /**** Variable initialization ****/
        lastEdge = 0;
        CLKedge = 0;
        bitNumber = 0;
        byteNumber = 0;
        junkByte = 0;
        for (count = 0; count < 3; count++)
              bitStream[count] = 0;

        data = 0;
        dataReady = 0;
        sync = -1;        // Initially, we don't have sync

        /**** Get this party started (enable interrupts) ****/
        #asm("sei");

}
```

8.3.2. TRANSPONDER CODE

```
//RFID Application for Biological Telemetry
//By Diana M. Rodriguez Tobon June, 2005
//The code detects the field clock and controls the MOD input in the
//transponder
// It also reads the temperature of the animal and codes it into
//manchester code
//protocol: byte(00000000) byte(11111111) byte(11111111) 100 byte(data)
/* This code is for the AtmelTiny13 Microcontroller */


/************************************
//Port Description

Connect XT1 on PORTE to PB3(XTAL1 on 2323) on PORTB on the STK500

Inputs:
B.2 = Field Clock Input (T0)
B.4 = ADC input ADC2

Outputs:
B.0 = Manchester data output
B.1 = 2 KHz Clock


/************************************
//Includes
#include <Tiny13.h>


/************************************
//Function Prototypes
void initialize(void);   //Initialization


/************************************
//Global Variables

unsigned char voltageD;  //digital voltage from ADC
unsigned char index;     // byte number from 0 to 103
unsigned char byte;      //byte to transmit
unsigned char halfcycle; //,number of half-cycles field clock has gone
                         //through transmission


//*********************************************************
//timer 0 compare Interrupt Sub Routine
//Enter here every field clock rising edge
interrupt [TIM0_COMPA] void timer0_com(void)
{
   // Three sync bytes (00-ff-ff), then 100 data bytes.

      PORTB.1 = ~PORTB.1; //field clock for debugging
```

```
    // Byte boundary.
    if (halfcycle == 16)    //if byte has been sent
    {                       //set next byte to transmit

      halfcycle = 0;
        voltageD=ADCH;
      if (voltageD == 255) voltageD = 0;  //to make sure 255 is
                                          //is not valid data

      //get new analog to digital value
      ADCSRA.6=1;

      if (index == 0)     byte = 0x00;    // set next byte according to
      else if (index < 3) byte = 0xff;    // protocol
            else byte = voltageD;

      bytenum++;
      if (index == 103) index = 0;

    }

    // On the falling edge of the clock, the output matches the data
    if (halfcycle & 1)
      PORTB.0 = (byte >> (halfcycle >> 1)) & 0x01;
    // On the rising edge of the clock, the output is the data inverted
    else
      PORTB.0 = (~(byte >> (halfcycle >> 1))) & 0x01;

    halfcycle++;
}


//*********************************************************
//Entry point and infinite loop
void main(void)
{
   initialize();

   while(1)
   {
   }//end while(1)
}//end main



//*********************************************************
//Initialize ports and variables
void initialize(void)
{
  //set up the ports
  DDRB.0=1;  // PORT.0 and PORT.1 are outputs
  DDRB.1=1;
  PORTB.0=0;
  PORTB.1=0;

  //Variables
  index=0;
```

```
  voltageD=0x00;
  byte=o;
  halfcycle=0;




  //set up timer 0
  OCR0A = 32;                 //set the compare to 32 periods
  TCCR0A=0b00000010;          //CTC mode
  TCCR0B=0b00000111;          //rising edge of external clock
  TIMSK0=0b00000100;          //enable compare match

  //set ADC registers in ADC noise canceler(Idle mode)
  ADMUX=0b01100010;           // Vref=1.1v(internalV), select ADCH
                              // select input channel PB4,
  ADCSRA=0b10000110;          // enable the ADC, single conversion mode
                              // clk prescaler to 64=150kHZ resolution
                              // no adc conversion complete interrup
                              // enable

   //turn on the interrupts
   #asm ("sei");

}//end initialize
```

8.3.2. GUI CODE

```
function varargout = serialcommgui(varargin)
% SERIALCOMMGUI M-file for serialcommgui.fig
%      SERIALCOMMGUI, by itself, creates a new SERIALCOMMGUI or raises the existing
%      singleton*.
%
%      H = SERIALCOMMGUI returns the handle to a new SERIALCOMMGUI or the handle to
%      the existing singleton*.
%
%      SERIALCOMMGUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in SERIALCOMMGUI.M with the given input arguments.
%
%      SERIALCOMMGUI('Property','Value',...) creates a new SERIALCOMMGUI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before serialcommgui_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to serialcommgui_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Copyright 2002-2003 The MathWorks, Inc.

% Edit the above text to modify the response to help serialcommgui

% Last Modified by GUIDE v2.5 04-Dec-2004 13:14:01

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @serialcommgui_OpeningFcn, ...
                   'gui_OutputFcn',  @serialcommgui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
```

```
if nargout
   [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
   gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before serialcommgui is made visible.
function serialcommgui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to serialcommgui (see VARARGIN)

% Choose default command line output for serialcommgui
handles.output = hObject;
handles.num_points = 2500;
handles.max_time = 15;
handles.use_time_not_points = 1;
handles.cancel_data_acq = 0;
handles.shownondata = 0;

% Update handles structure
guidata(hObject, handles);

set(handles.timeradio, 'Value', 1);
set(handles.numpointsbox, 'Enable', 'Off');

% Set up the axis
axes(handles.axes1);
xlabel('time (seconds)');
ylabel('data');


global cancel_data_acq;
cancel_data_acq = 0;
clear cancel_data_acq;

% UIWAIT makes serialcommgui wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = serialcommgui_OutputFcn(hObject, eventdata, handles)
```

```
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


function numpointsbox_Callback(hObject, eventdata, handles)
% hObject    handle to numpointsbox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of numpointsbox as text
%        str2double(get(hObject,'String')) returns contents of numpointsbox as a double

val = str2num(get(hObject, 'String'));
if (length(val) == 1) && (val > 0) && (val <= 1e6)
   handles.num_points = str2num(get(hObject, 'String'));
   guidata(hObject, handles);
else
   set(hObject, 'String', num2str(handles.num_points));
end

% --- Executes during object creation, after setting all properties.
function numpointsbox_CreateFcn(hObject, eventdata, handles)
% hObject    handle to numpointsbox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
   set(hObject,'BackgroundColor','white');
else
   set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on button press in getdatabutton.
function getdatabutton_Callback(hObject, eventdata, handles)
% hObject    handle to getdatabutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
set(handles.getdatabutton, 'Enable', 'off');    % Disable get data button.
set(handles.stopbutton, 'Enable', 'on');        % Enable stop button.
set(handles.timeradio, 'Enable', 'inactive');   % Disable time radio
set(handles.pointsradio, 'Enable', 'inactive'); % Disable time radio
if strcmp(get(handles.maxtimebox, 'Enable'), 'on')    % Disable time/points boxes
   set(handles.maxtimebox, 'Enable', 'inactive');
else
   set(handles.numpointsbox, 'Enable', 'inactive');
end
set(handles.popaxisbutton, 'Enable', 'off');    % Disable pop up axis button
set(handles.ndpointscheckbox, 'Enable', 'off'); % Disable non-data points checkbox


% Clear the axis
axes(handles.axes1);
cla;

global cancel_data_acq;

% Destroy any open serial port objects on COM1
s = instrfind('Port', 'COM1');
if length(s) ~= 0
   fclose(s);
   delete(s);
end

% A trial serial read to determine whether serial I/O is functioning.
% If not, 'fread(s)' will time out and data will have length 0.
s = serial('COM1', 'BaudRate', 57600, 'InputBufferSize', 1);
fopen(s);
data = fread(s);
fclose(s);
delete(s);
clear s;

if length(data) ~= 0    % If there was data in the buffer (reads are functioning)...

   % Create the serial port object. With a buffer size of 512, at a data rate
   % of 239 bps, it takes about 2 seconds to fill the buffer.
   s = serial('COM1', 'BaudRate', 57600, 'InputBufferSize', 512);
   fopen(s);

   data = [];

   badtime = []; % vector listing times where data was 255 (for debugging)
```

```
    set(handles.messagewindow, 'String', 'Getting data...');

    % Variable initialization.
    elaptime = 0;
    time = [];
    timelimit = handles.max_time;
    pointslimit = handles.num_points;
    percent_complete = 0;
    tic;
    while (handles.use_time_not_points && (toc < timelimit)) || ...
          (~handles.use_time_not_points && (length(data) < pointslimit)),

       % Pause to allow for interrupts, etc. We won't miss data doing
       % this, because the data buffer gets filled in the background.
       pause(0.5);

       if cancel_data_acq
          cancel_data_acq = 0;
          break;
       end

       new_data_index = length(data) + 1;

       data = [data fread(s)'];
       elaptime = toc;

       if handles.use_time_not_points
          if floor((elaptime / timelimit)*10) > percent_complete
             percent_complete = floor((elaptime / timelimit)*10);
             set(handles.messagewindow, 'String', ...
                [num2str(min(100, percent_complete*10)) '% complete - ' ...
                num2str(max(0,floor((timelimit - elaptime)*10))/10) ' seconds
remaining...']);
          end

       else
          if floor((length(data) / pointslimit)*10) > percent_complete
             percent_complete = floor((length(data) / pointslimit)*10);
             set(handles.messagewindow, 'String', ...
                [num2str(min(percent_complete*10)) '% complete - ' ...
                num2str(max(0,pointslimit - length(data))) ' data points remaining...']);
          end
       end

       if length(time) == 0
          newtime = linspace(0, elaptime, 513);
```

```
    else
        newtime = linspace(time(length(time)), elaptime, 513);
    end

    time = [time newtime(2:length(newtime))];

    % Clear all values of 255 from data
    count = new_data_index;

    % Leading 255's:
    if new_data_index == 1
        while data(1) == 255,
            badtime = [badtime time(1)];
            data = data(2:length(data));
            time = time(2:length(time));
        end
        count = count + 1;
    end
    % Middle 255's:
    while count < length(time),
        if data(count) == 255
            badtime = [badtime time(count)];
            data = [data(1:count-1) data(count+1:length(data))];
            time = [time(1:count-1) time(count+1:length(time))];
        else
            count = count + 1;
        end
    end
    % Trailing 255
    if data(length(data)) == 255
        badtime = [badtime time(length(time))];
        data = data(1:length(data)-1);
        time = time(1:length(time)-1);
    end

    % Plot the data on the GUI's axes.
    plot(time, data);


end


% If we have received data (i.e., if data acquisition hasn't been immediately
% cancelled)...
if length(time) > 1
    % If the data isn't ALL invalid...
```

```
if ~(all(data == 255))

   % Truncate the vectors to the appropriate time if time limit was used.
   if handles.use_time_not_points
      count = 1;
      while (time(count) <= timelimit) && (count < length(time)),
         count = count + 1;
      end
      data = data(1:count);
      time = time(1:count);

   else
      % Truncate the vectors to the appropriate number of points if
      % points limit was used.
      data = data(1:min(pointslimit, length(data)));
      time = time(1:min(pointslimit, length(time)));
   end

   % Make the bad data well-formed
   count = 1;
   while count < length(badtime),
      if badtime(count) > time(length(time))
         badtime = badtime(1:count - 1);
         break;
      else
         count = count + 1;
      end
   end

   baddata = [];
   btcount = 1;
   while badtime(btcount) <= time(1),
      baddata = [baddata data(1)];
      btcount = btcount + 1;
   end

   count = 1;
   while count <= length(time)
      if time(count) >= badtime(btcount)
         baddata = [baddata data(count - 1)];
         btcount = btcount + 1;
         if btcount > length(badtime)
            break
         end
      else
         count = count + 1;
```

```
            end
        end

        while btcount <= length(badtime)
            baddata(btcount) = data(length(data));
            btcount = btcount + 1;
        end

        clear btcount count;

        % Save time and data to the current workspace, so the user can mess
        % with them.
        assignin('base', 'baddata', baddata);
        assignin('base', 'badtime', badtime);
        assignin('base', 'mouse_data', data);
        assignin('base', 'mouse_time', time);

        set(handles.messagewindow, 'String', ...
            'Data vectors created in current workspace.');


        % Plot the data on the GUI's axes.
        plot(time, data); hold on;
        if handles.shownondata
            plot(badtime, baddata, 'r.');
        end

    else
        % If the data IS all invalid...
        set(handles.messagewindow, 'String', ...
            'No valid data received.');

    end
end

fclose(s);
delete(s);
clear s;

% If we didn't receive any data when clearing the input buffer and testing
% for I/O timout, then print an error.
else
    set(handles.messagewindow, 'String', 'Problem with serial I/O...');
end

set(handles.stopbutton, 'Enable', 'off');   % Disable stop button (reads are over).
```

```
pause(5);   % Wait for 5 seconds so the user can read the message.
set(handles.messagewindow, 'String', ...
   'Welcome to the Mouse-Implanted RFID Data Acquisition Tool!');
set(handles.getdatabutton, 'Enable', 'on');     % Re-enable get data button.
set(handles.timeradio, 'Enable', 'on');         % Re-enable time radio
set(handles.pointsradio, 'Enable', 'on');       % Re-enable time radio
if strcmp(get(handles.maxtimebox, 'Enable'), 'off')   % Re-enable time/points boxes
   set(handles.numpointsbox, 'Enable', 'on');
else
   set(handles.maxtimebox, 'Enable', 'on');
end
set(handles.popaxisbutton, 'Enable', 'on');     % Re-enable pop up axis button
set(handles.ndpointscheckbox, 'Enable', 'on'); % Re-enable non-data points checkbox



function maxtimebox_Callback(hObject, eventdata, handles)
% hObject    handle to maxtimebox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of maxtimebox as text
%        str2double(get(hObject,'String')) returns contents of maxtimebox as a double

val = str2num(get(hObject, 'String'));
if (length(val) == 1) & (val > 0) & (val <= 6000)
   handles.max_time = str2num(get(hObject, 'String'));
   guidata(hObject, handles);
else
   set(hObject, 'String', num2str(handles.max_time));
end

% --- Executes during object creation, after setting all properties.
function maxtimebox_CreateFcn(hObject, eventdata, handles)
% hObject    handle to maxtimebox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
   set(hObject,'BackgroundColor','white');
else
   set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

```
% --- Executes on button press in timeradio.
function timeradio_Callback(hObject, eventdata, handles)
% hObject    handle to timeradio (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of timeradio


% --- Executes on button press in pointsradio.
function pointsradio_Callback(hObject, eventdata, handles)
% hObject    handle to pointsradio (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of pointsradio


function messagewindow_Callback(hObject, eventdata, handles)
% hObject    handle to messagewindow (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of messagewindow as text
%        str2double(get(hObject,'String')) returns contents of messagewindow as a double


% --- Executes during object creation, after setting all properties.
function messagewindow_CreateFcn(hObject, eventdata, handles)
% hObject    handle to messagewindow (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
   set(hObject,'BackgroundColor','white');
else
   set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function radiopanel_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to radiopanel (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

if strcmp(get(hObject, 'Tag'), 'pointsradio')
   set(handles.numpointsbox, 'Enable', 'On');
   set(handles.maxtimebox, 'Enable', 'Off');
   handles.use_time_not_points = 0;
   guidata(hObject, handles);
else
   set(handles.numpointsbox, 'Enable', 'Off');
   set(handles.maxtimebox, 'Enable', 'On');
   handles.use_time_not_points = 1;
   guidata(hObject, handles);
end


function radiopanel_CreateFcn(hObject, eventdata, handles)
% hObject    handle to radiopanel (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on button press in stopbutton.
function stopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to stopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global cancel_data_acq;
cancel_data_acq = 1;
set(handles.messagewindow, 'String', 'Cancelling...');


% --- Executes on button press in popaxisbutton.
function popaxisbutton_Callback(hObject, eventdata, handles)
% hObject    handle to popaxisbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Pop up an external axis
figure;
time = evalin('base', 'mouse_time');
data = evalin('base', 'mouse_data');
badtime = evalin('base', 'badtime');
baddata = evalin('base', 'baddata');

plot(time, data);
xlabel('time (seconds)');
```

```
ylabel('data');
title('Mouse Data');
hold on;
if handles.shownondata
   plot(badtime, baddata, 'r.');
end



% --- Executes on button press in ndpointscheckbox.
function ndpointscheckbox_Callback(hObject, eventdata, handles)
% hObject    handle to ndpointscheckbox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of ndpointscheckbox

handles.shownondata = get(hObject, 'Value');
% Update handles structure
guidata(hObject, handles);
```