Dog Smart Vest Microprocessing

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Bruce Andrew Beitman, B.S.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2012

Master's Examination Committee:

Joanne DeGroat, Advisor

Xiaorui Wang

Fusun Ozguner
Abstract

This abstract describes the design concept for a wireless detection dog system from the microprocessor perspective. The concept in general was suggested through the air force and this document will flesh out the issues overcome. The end goal was to allow the dog to detect while not leashed to a trainer; using wireless technology that does not use cell phone infrastructure to send live audio and video data between the dog and its trainer.

The first major issue confronted was sending live data wirelessly. This problem continued to cause issues and was handled different ways through the process of designing the system. Other addressed concerns include: low power and low weight components, real time data handling from the wireless devices, and control of multiple devices. This document focuses primarily on the microprocessor and its code, but the project is described in its entirety to put context to the problems and solutions accomplished by the microprocessor.
Dedication

This document is dedicated to my wife Rachel Beitman
Acknowledgments

I would like to acknowledge a few people who helped tremendously in designing and completing this project with me: Professor DeGroat for her many suggestions, ideas, and overall helpfulness; Justin Shen for his insights and help; Zheng Ma and Peng Peng for their perspectives and help.
Vita

June 2006 ................................................. Worthington Christian High School

2010 ......................................................... B.S. Electrical & Computer Engineering,

Ohio State University

2010 to present ......................................... Graduate Research Associate, Department

of Electrical & Computer Engineering, The

Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering
# Table of Contents

Abstract ................................................................................................................................. ii

Dedication ............................................................................................................................... iii

Acknowledgments .................................................................................................................. ivv

Vita ....................................................................................................................................... v

List of Tables .......................................................................................................................... ix

List of Figures .......................................................................................................................... x

Chapter 1: Introduction ......................................................................................................... 1

1.1 Problem Statement .......................................................................................................... 1

1.2 System Requirements ..................................................................................................... 1

Chapter 2: System Design .................................................................................................... 4

2.1 First Prototype ............................................................................................................... 4

2.2 Microprocessor Requirements ....................................................................................... 4

2.3 Second Prototype .......................................................................................................... 5

2.3 Third Prototype ............................................................................................................. 5

Chapter 3: Choosing the Microprocessors ......................................................................... 7

3.1 Processor Requirements ............................................................................................... 7
List of Tables

Table 1. Component Cost ........................................................................................................6
List of Figures

Figure 1. Initial design blue print ..........................................................................................6
Figure 2. Completely Broken Down System ........................................................................16
Figure 3. Buffer Flow Diagram ..............................................................................................28
Figure 4. GPS Accuracy .........................................................................................................32
Figure 5. Dropped Packets in varying conditions .................................................................33
Figure 6. Transmission distance in varying conditions .........................................................
Chapter 1: Introduction

The system to be developed is for a multi-purpose dog vest that would allow the trainer to give long distance commands and receive information about the dog's position. This would allow missions that would otherwise require a trainer to be with the dog. Some of these advantages include: tracking multiple dogs with a single trainer, allowing the dog to roam without the hindrance of the trainer, allowing the dog to move in areas that might be difficult for the trainer.

1.1 Problem Statement

The system was specified to include audio relay from the trainer to the dog; from the dog to the trainer would include video, GPS, and current battery life. The wireless data was to be encrypted and verified for errors. The system had to withstand exposure to outdoor elements and be relatively robust. The OSU team was responsible for building the proof of design with recommendations for the field-able prototype build. The first two prototypes discussed in this paper are the proof of design concepts while the third prototype described is the actual prototype as it was field tested.

1.2 System Requirements
The requirements for the system were: weight under 2 lbs, at least 5 mile transmission range, at least 2 hours of full time use, and cost under $1000 per unit in production (note the costs in the expense report are for evaluation purposes and would drastically decrease in production). Most of these requirements were easy to achieve when disregarding one or more of the other requirements. The weight was mostly affected by the transceiver and the battery; the heavier and more power the transceiver carried the further it could send data and the heavier the battery the longer it would last. The only requirement that was independent was the price; this was rather easy to stay under since it is a production value (see Table 1). Also, after the air force confirmed that military radios would be used, the radio was eliminated from the cost of a unit which was by far the most expensive. This shows the tight requirement between transmission distance, weight, and power. The microprocessor needed to be as low power as possible so that the power sent to the transceiver could be maximized, and also maximize the time the system had power. Utilizing any power down capabilities was very important since the microprocessor had to minimize power at all times. This is described more in Chapter 3.

The initial design also included a relay that was attached to some high altitude device such as a weather balloon. The relay would have a separate frequency so as to not interfere with short distance transmissions. This would require a protocol for the handler to choose whether to use the relay or not and the dog microprocessor to choose which frequency the dog transceiver would listen for. The protocol would send some dummy signal to the closer option, i.e. not the relay, and expect a return command. If there was
no reply then the further option was checked, i.e. transmission to the relay. The
transmission relay would attach a tag to the signal and verify that it was relaying the data
to the corresponding transceiver. This would allow the user to know where the signal
would be lost if a response was not heard. This system was never fully developed as the
air force denied the suggestion for a relay, which created problems with the transmission
distance requirement. This is addressed more in Chapter 4.
Chapter 2: System Design

As the system was being developed, its design changed as the team became more familiar with the issues and solutions that existed. The system needed to be broken down into smaller components, so the end goal was split into smaller chunks. Ignoring video and data encryption was the first simplification.

2.1 First prototype

The first design was focused on the audio and picture data. From this perspective a blueprint was created of this first prototype (see Figure 1). Specific simplifications are described in Chapter 5. This design included 3 microcontrollers, one for the trainer, one for the dog, and one for the computer. The order of data transfer would be trainer, dog, and then computer. In this way the trainer would send audio data to the dog, the dog microcontroller would output this audio and send picture and GPS data to the computer microcontroller, which would prepare the data for receiving to a computer that the trainer would probably have.

2.2 Microcontroller Requirements

The design requirements for each of the microprocessors are fairly straightforward from Figure 1. The trainer microprocessor needed an analog data input for audio data, along with an analog to digital (A/D) converter to allow transmission of
the audio data. The dog microprocessor needed a digital to analog (D/A) converter to allow interpretation of the digital audio stream, at least 3 communication port capabilities to communicate with a camera, a global positioning system (GPS), and a radio. The computer microprocessor needed only some memory and processing capabilities to take the wireless data and interpret it for the computer. Another factor was the processing and communication speed needed to be fast enough for live audio to be sent and heard reasonably. This is described in more detail in Chapter 4. The video component was ignored since the speed requirement was going to be a major issue to reach with the other requirements.

2.3 Second Prototype

This initial system design was changed after the camera peripheral had software that easily read the picture stream. This eliminated the need for data preparation before transmission to the computer. This second prototype design approach came from the idea that the computer did not know that its data was being sent and handled by multiple devices connected serially. But due to the handshaking protocol verification and replication of the data was capable without any knowledge to the computer. This is described in more detail in Chapter 7.

2.4 Third Prototype

The final system design was developed with another project team, RNET, who continued this work. The trainer board was replaced with an Android application that did
all the data handling required to send audio data to the dog and also handled the reception data from the dog. This application is described in more detail in Chapter 9.

### Table 1 Component Cost

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Parts</th>
<th>Shipping</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller boards &amp; JTAG</td>
<td>3 &amp; 1</td>
<td>-</td>
<td>-</td>
<td>$584.00</td>
</tr>
<tr>
<td>Transcievers</td>
<td>2</td>
<td>$650.00</td>
<td>$50.00</td>
<td>$728.00</td>
</tr>
<tr>
<td>Camera &amp; GPS</td>
<td>1 &amp; 1</td>
<td>-</td>
<td>-</td>
<td>$139.90</td>
</tr>
<tr>
<td>USB/TTL Converters</td>
<td>2</td>
<td>$24.00</td>
<td>$18.76</td>
<td>$42.76</td>
</tr>
<tr>
<td>RS232/TTL Converters</td>
<td>2</td>
<td>$25.96</td>
<td>$18.76</td>
<td>$44.72</td>
</tr>
</tbody>
</table>

Grand Total $1,539.38

Figure 1. Initial design blue print
Chapter 3: Choosing the Microprocessor

3.1 Processor Requirements

For the proof of design the majority of the processing would certainly be on the dog. Implementing more user functionality on the trainer side is out of the scope of a proof of design. Therefore the functionality of the dog processor will be the major bulk of the project. The dog processor includes real time processing of incoming audio data, D/A conversion of the audio data, filtering picture commands from audio data, routing picture commands, routing picture data, and routing GPS data. Most of this is data handling and routing, but the processor does need to perform D/A conversion in real time and handle picture commands simultaneously.

A number of factors needed to be considered when choosing the microprocessor. Reasonable speed was required as real time deadlines needed to be met, but also low weight and low power capabilities were very important. In addition, fabricating a board was outside the scope of a proof of design project, so an evaluation board that allowed quick interfacing and possibly some example code would fit the proof of design concept. While ignoring the size and inefficiency of the evaluation board, special attention was given to how this would be ported to the next design and what was easily fixed. While the proof of design itself suffered no power or weight restrictions, these factors were considered and optimized when possible during the design phase.
3.2 Choosing the Processor

For stability and strong support along with the low weight and power requirements the ARM processor seemed a very appropriate choice. ARM has been the leading designer in low power processors for a while. Knowing that power is a major issue, and power and performance are indirectly related, the choosing process was mostly finding the processor that minimally accomplished the requirements. Looking at ARM's options for architectures the Cortex series, being the newest in production, fit best as it meets the performance needed and has very low power consumption. In addition, it has numerous power-down options. [3]

The Cortex series is broken down into three categories: Ax, Rx, and Mx. As the project does not require high end processing technology, the M series was an obvious fit. But narrowing within the M series took some research. The Cortex-M1 is built with FPGA’s in mind and is not what the project would need. The Cortex-M0, M3, and M4 have many similarities, the main difference is the instruction set and targeted applications. The Cortex-M0 has a very low gate count and limited instruction set. It is the smallest processor available from ARM, and thus extremely low functional. This might be a problem with the D/A conversion. Conversion will be performed simultaneous to picture data routing and GPS gathering; with so many real time interrupts the Cortex-M0 seems too low functional. The Cortex-M4 is a very powerful processor, compared to the M0. This processor includes a floating point unit, and this is certainly beyond the requirements of the project. The Cortex-M3 is very comparable to the M0, but it has
considerably more instructions and larger memory addressing. This processor is designed with fast interrupt in mind and is implemented in many real time systems. While the Cortex-M0 could accomplish the system requirements it would likely need some small additional units to aid in handling the amount of live data. Therefore, the Cortex-M3 was strongly suggested and accepted by the air force. [3]

3.3 Choosing the Evaluation Board

As discussed earlier fabricating a board is out of scope for the proof of design, so evaluation boards were researched for implementation. A highly supported board from a branch of ARM called Keil was found that implemented many of the Cortex series processors. These boards had their own Integrated Development Environment (IDE) including simulators for some of the processor capabilities and a couple example projects showing the simulations and processing capabilities. The IDE and JTAG equipment also included a debugging interface that allowed for stepping through the code. This proved extremely helpful when trying to accomplish many tasks with the microprocessor. [2]

The Cortex series has much functionality and Keil used this by making many types of evaluation boards utilizing the different capabilities of the Cortex series. As discussed earlier the Cortex-M3 was chosen as most accurate to the project. Many Keil Cortex-M3 boards were evaluated for implementation in the project. [2]

Here is an overview of the evaluation boards that were considered:
The MCB1343 had a clock of 72 MHz, 8K RAM, 32K FLASH, Analog input, 1 Serial port. [2]

The MCBSTM32EXL had a clock of 72MHz, 96K RAM, 1M FLASH, Analog input, Analog output (Speaker), 1 Serial port, and LCD screen. [2]

The MCB1760 had a clock of 100 MHz, 64K RAM, 512K FLASH, Analog input, Analog output (Speaker), 2 Serial port, and LCD screen. [2]

The MCBSTM32F200 had a clock of 120MHz, 128K RAM, 1M FLASH, Analog input, Analog output (Speaker), 3 Serial port, LCD screen, and a digital camera. [2]

The MCB1343 was the cheapest at $120. While it had the minimal peripherals, there would be extra work to get a speaker connected to the analog out. The LCD screen, while certainly outside the requirements of the proof of concept, was considered since it might be helpful with serial communication testing and debugging. The clock speed seemed adequate but was on the low end for a Cortex-M3. The clock speed was relatively important since live audio data was being processed. But beyond this the processor would be handling 3 data inputs and one of these would have two data types and one of the data types would need to be relatively fast since it’s live audio. The requirements for the live audio are described in more detail in Chapter 4. Also, the RAM and FLASH would need to be reasonable for buffering any audio data that might need to be handled at a slightly later time. This is described more in Chapter 8. At the time of choosing the processor the team was not aware how the system would be implemented but wanted to just keep options open while developing the system.
The MCBSTM32EXL was $280 and had a speaker built into the analog out and an LCD screen, but a single serial port might be difficult to debug with since the project required a lot of data routing. This processor also had the slower clock speed, but a much more reasonable RAM and FLASH size which would allow for significant audio buffering.

The MCBSTM32F200 was $350 and had 3 Serial ports and a camera. While a camera was a requirement, infra red was also needed. The clock speed was high which is advantageous to handling the audio data and might cut down on any need on buffering. The RAM and FLASH were large as well, but the small advantages for the steep price difference didn’t seem to be worth it.

The MCB1760 was chosen for $250. Two serial ports seemed adequate; it included the speaker and LCD, and did not include the camera that would need replaced. The clock was neither the slowest nor the fastest, and the RAM and FLASH fell in the middle of the pack as well. The team did not want the processor to be too slow, but the higher the clock and bigger the memory the more power the chip would draw. While it is just a proof of concept, how the proof of concept performs with its design will yield a more educated suggestion for components to use in the prototype.
Chapter 4: Audio Handling and Sampling Rates

The audio code for the microprocessor is described in more detail in Chapter 5. To understand the quality of audio to aim for, a description of sampling frequency is needed. The typical standard for human hearing capabilities has been accepted as 20 Hz to 20 KHz. Using the Nyquist Theorem to accomplish standard human hearing would require a sampling rate of 40 KHz. But this is beyond the requirement for simple understanding human speech in audio. Standard telephone lines use a sampling rate of 8 KHz, but this causes some sounds to not be reproduced properly. [1]

In addition to the sampling rate is the quality of the A/D conversion, and D/A (assume that whatever is done to the A/D is also done to the D/A). The more bits used for conversion allows each splice of audio data to be represented by more values allowing a better reproduction of the original sound. Thus the greater number of bits used in each conversion will produce a more accurate result, but this will fill the bandwidth quicker. The sampling rate can be modeled by this simple equation, where \( f \) is the sampling rate, \( b \) is the speed that the data can be sent (baud rate), and \( b_{\text{res}} \) is the resolution of bits used in the A/D conversion.

The distance requirement becomes very restricting at this point. The frequency and the data rate are very closely related since data is understood in the change in
frequency, and frequency and the distance of transmission is inversely related. This caused the most difficult problem in the system as a whole. Large bandwidth was to be sent over a large distance indiscreetly with low power and small antennas.

Within the commercial radio market nothing exceeded 115,000 bits per second or baud that was close to the other requirements. This was discussed with the air force and was decided that it was adequate for the proof of design concept but that in field tests, military grade radios would probably replace the commercial radios. Similar protocol to military radios was used so that a military radio would be a drop in replacement in the system. Due to the limitation of 115K baud, after testing it was found that 8 bits provided adequate reproduction of the analog signal. This yielded slightly over 14 KHz sampling rate which gives a slightly better reproduction than standard radio or telephone systems. To put this into more perspective, standard audio CD sampling rate uses 44 KHz, but also remember that standard telephone sampling rate is 8 KHz [2]. To create a better sampling rate and quality of sound, multiple approaches were researched and explained to the air force; these are described in detail in Chapter 7.
Chapter 5: System Fragmentation

The work of the engineer often prescribes breaking a problem into smaller problems. This was done by characterizing what could be accomplished and successfully tested with a smaller component that would also help the project as a whole. This was constantly being applied throughout the entire project; some detail will be given to the larger simplifications. When merging multiple steps together, the option of going back a step to retest and see what was going on was invaluable to getting multiple protocols and components to communicate smoothly. Some of the specifics of problems that arose during these phases are explained in Chapter 7. In this section the details of what stages the system was broken into are described.

Getting rid of the relay was simple; since it will be invisible to all the controllers anyways and is a drop in component. Audio was focused first since it would be easier to implement and test. The audio transmission was broken down by getting audio data to be converted from analog to digital and back again using the same processor. This was done with both the trainer and the dog board. This simple set up of the system was called the completely broken down system because this is where testing started and at any point the team could test each component with the lowest possible component interaction (see Figure 2). When testing with many components that yielded confusing results, the completely broken down system was a quick way to find the problematic component. The next test was to send data from the GPS to a computer via the dog microprocessor, and
from the camera to a computer via the microprocessor. This seems trivial, to treat the microprocessor as a wire in a sense, but timing protocols were tricky and this becomes increasingly important when filtering data types. The next test was sending data from computer to computer using the radios using simple character data that required no timing to see that data was sent and received properly. Once this was seen to be accomplished accurately audio data was sent instead of simple data. These tests were set aside and the camera was tested via the microprocessor and the radios. Up to this point all the microprocessor was never handling more than a single data type at one time. The multiple data types are GPS data, audio data, camera command data, and camera return data. The GPS data and camera return data come to the dog microprocessor in different ports and are much easier to distinguish, but audio data and camera command data come in the same port to the dog microprocessor. The way the dog microprocessor handles this is explained in detail in Chapter 7. The audio and camera were tested together with the system wired, then wirelessly through the radios. Then the GPS was tested alone wired and wirelessly. Then finally, the GPS was tested with the audio and camera wired and wirelessly. At each of these steps many problems appeared and were handled before proceeding to the next step.
Figure 2 Completely Broken Down System
Chapter 6: Microprocessor Architecture and Code Structure

As explained earlier, two microprocessors were needed in the final prototype. These were the trainer and dog processors. The trainer microprocessor was less complex and ultimately not going to be used in the prototype, see Chapter 9. The trainer processor was an ARM7, including a 15 MHz clock; whereas the dog microprocessor, a Cortex-M3, had many more components with a 25 MHz clock. Working with the sample code that came from Keil helped understand many of the technical details of the processor.

6.1 Common Important Registers

The ARM7 and Corext-M3 had many similarities. Both processors had PINCON registers that determined what a general input output pin’s (GPIO) function was. The Cortex-M3 had a PINMODE register that would add a pull up resistor to the contact on that pin. Both processors had a PCONP register that determined what peripherals of the chip were powered on [4]. This was very helpful in showing the power restriction capabilities of the processor, as many of the peripherals were not used. All of these registers were represented in C as structs that pointed to possible options for the processor, such as input, output, or simple binary.

6.2 A/D Conversion
The structures of the A/D and D/A converters were very similar on both the ARM7 and Corex-M3. They both had multiple channels for holding analog data for conversion in addition to a global register that was used as default for ease of instructions when using only a single stream of audio data. They both had interrupt enable registers for converting data based on an external event; although they were not needed as the audio conversion did not hinge on anything but the clock. The Cortex-M3 was capable of 10 bits of resolution where the ARM7 had only 8 bits, but this was very adequate for the proof of design. Both processors included a clock divider that would alter the global processor clock for the converter, and a start register that would determine the start of conversion based on the edge of the clock.

6.3 Serial Communication

Both processors had one full modem uart controller and one non-full modem uart controller. The Cortex-M3 had two additional non-full modem uart controllers which also had low communication protocol (TTL). There was an optional peripheral clock register that would override the global clock for the uart controller. The Baud rate was determined by additional registers: FDR (DivAddVal and MulVal), DLL, DLM, and PCLK, the global clock, where the baud rate equals:

\[
BAUD = \frac{PCLK}{16 \times (256 \times DLM + DLL) \times (1 + \text{DivAddVal/MulVal})}
\]
This algorithm obviously cannot be exact but there is always error when dealing with baud and data transfer. No issues were found working at multiple baud rates up to 115,200 bits per second which is what the final prototype used. For this baud calculation the following registers were used DLL = 0x09, DLM = 0x00, FDR = 0x21 (This means DivAddVal is 1 and MulVal is 2), which calculates to 115740. In order to time multiple devices, a check serial port function was created (See Appendix) for each uart peripheral which in turn would create an interrupt and process the serial data. This was a critical step in handling multiple sets of incoming data. In addition to this was of course reading and writing to the serial port. These reading and writing were either blocking or non blocking; blocking meaning they would wait for data or data to be cleared to send the next character. While blocking read and write was useful in testing, this would force missed data on other uart ports since data was being received and transmitted along multiple ports. Also, a change baud rate procedure became important when supporting all the functionality to the camera. This is described in more detail in Chapter 7.
Chapter 7: Issues and Solutions

7.1 Independent A/D Conversion

Each step of the design process was briefly outlined in Chapter 4. Now some description will be given to some of the problems that needed to be compensated in these steps. A/D conversion was the first test. A microphone was hooked up to each board individually and the output of the conversion to the D/A converter was set to the speaker. This step got a lot of the A/D and D/A conversion code working correctly and familiarized the team with timing issues. The timing issues were the most complicated part of this design. Using the debugger and simulator the registers could be watched receiving the correct values and processing data, but only static was received at the speaker. This was finally resolved when it was discovered that the system frequency was set to 10 ms, which is extremely low for audio processing. When this was resolved the audio worked correctly.

7.2 Serial Communication

The next step was to get serial data from the computer to a processor. A simple port monitoring program called X-CTU was used to communicate from the computer to the processor. The Keil boards used RS232 protocol for 2 (all the ports for the trainer processor) ports. Using the block reading this was relatively straightforward. A lot of time was spent understanding the RS232 protocol. Since both the Keil board and the
computer had female ports initially a male to male cable was used, which worked with the computer but not when used from processor to processor. Eventually it was discovered that the computer would compensate for cross wiring and male to male cables were not meant to be used from a single component to another single component. In addition, the Keil board had one full modem RS232 port and a simple RS232 port. The simple RS232 port had a standard serial port, but if any data was sent besides TX, RX, or GND the entire board would die. By the second prototype RS232 cables specific for each port of both processors were soldered. During the first prototype many tests were being interchanged but since a final design was not yet able to be understood making specific cables seemed unreasonable. To compensate minimal lines were wired to a breadboard and routed to wherever testing was needed. These simple tests were typically sending random characters at 9600 baud.

This next step was the first combination process. Sending audio converted from one processor to the other processor to be reconverted; this step was comparatively very straightforward to the problems that arose in the first two steps and was accomplished quickly.

7.3 Incorporating the Camera

With the audio working correctly focus moved to the camera. The camera had many features helpful to the system design. One of these was the light detection and automatic switch to IR (infrared). This was a requirement from the air force and cut the work needed to be done considerably since the camera took care of the IR itself. The
camera also came with computer software that processed and sent data on the fly. Detaching the camera from this program seemed out of the scope of the proof of design. Instead of working with the dog processor and testing as had been done previously, it was much easier to work the other way. The camera was attached directly to the computer and slowly tested the camera data through each component. This allowed the software to be kept to verify the validity of the data, and no need to translate or port data from one device to another. The first step was routing data from the computer through the trainer processor. At this point the team realized there was no need for the additional computer processor since the picture data would be processed by the camera software and the GPS data could be embedded in this message. This design is referred to as the second prototype.

Receiving data and sending it character by character was not capable on the trainer processor. Routing camera commands straight to the transceiver avoiding the trainer processor was considered, but this would cause data type confusion. This was solved by reading the characters from the computer until a command was recognized. Since the trainer was only receiving data from the computer regarding camera commands, these could be embedded into the processor and forwarded after receiving enough data to determine what command was being sent. The dog processor, on the other hand, had no problem receiving and sending characters within a single baud cycle. This completed the camera communicating via the peripheral devices. The camera talked to the dog processor, which talked to the transceiver, which sent packets to the other transceiver, which sent the data directly to the computer. But this created the two most
difficult problems to solve in the system. Mentioned earlier is the problem of data type confusion, determining whether data received at the dog processor end was audio or camera command data. Also the camera has only volatile memory, thus when the system is shut down any settings set by commands will be lost. This included the baud rating, upon start up the camera is set to 38400 baud, but the rest of the system is configured with non-volatile memory to 115K baud.

7.4 Distinguishing Data Types

To fix the data fusion problem a flag was created. When a certain character was flagged as being received by the dog processor it knew that the next so many characters were going to be a picture command. The exact number of characters could vary, so all incoming data was buffered to be parsed. As soon as a command was recognized a trigger was set to be run in the next pass. This allowed for immediate switching to audio interpretation and the picture data to be handled when time allowed. This system would not work if the flag was sent and audio data was intermingled within the picture command. This could easily happen if the user was talking and sending a picture command at the same time. This needed to be guaranteed from the trainer processor, thus the need for the picture commands to be routed through the trainer processor. When a picture command starting to be sent is recognized by being fed serial data, the trainer processor ignores all audio data until the picture command is finished being sent. In addition a filter to the A/D converter was added that would make sure that the flag was never sent during audio transmission. This should have very minimal consequences on
the audio quality; since 8 bits were used to send audio data this limited 1 in 256 possible outcomes for a $1/(8000^{th} - 14000^{th})$ of a second.

7.5 Resolving Baud Rates

To fix the change in baud rate, a routine was created on the dog board that could change the baud rate. The routine would lower the dog processor baud rate, tell the camera to alter its baud rate to 115K baud, and then the dog processor would go back to the 115K baud rate itself. Forcing this to occur after start up was considered, but some timer would need to be created due to different start up times and this might cause a problem if the user sends data immediately. This was left as a requirement to the user to send the change baud rate before using the camera. This seemed appropriate again for the proof of design, but would need altering during the prototype phase. When developing the prototype this was not an issue since there was much more control on the trainer’s side with an Android OS.

7.6 Incorporating the GPS

Finally, the GPS was incorporated into the system. The GPS used the simple uart port on the dog processor (not a serial port). The GPS warm up time after shut down was over 20 seconds, therefore it could not be shut down or the picture and GPS data would not be accurate to each other. The GPS had a simple configuration of constantly sending the coordinates and a few other parameters such as time and precision of the data. Some of the precision data was used to verify that a string obtained was reasonably accurate for
sending back to the trainer. The two parameters used to determine this was the HDOP (horizontal dilution of precision) and the number of satellites [5]. For a prototype the acceptable HDOP value could be altered, but for the proof of design anything at or below a 2 was accepted if the number of satellites were at or above 3 (see Appendix Data Filter). If data was unacceptable then another string was pulled from the GPS up to a limit of 100. Mention of sending a GPS error signal if bad data was received for more than some set amount of time was recommended to building the prototype. The development beyond the described system would require user input which was not considered in the proof of design concept. But user input would allow for the amount of time to check for GPS and the minimum required HDOP and satellite readings to be set by the user. The GPS data was read when a picture command was received and once the GPS data was acceptable it was attached to the beginning of the picture data as it was sent back.
Chapter 8: Bandwidth Solutions

Since the final prototype had a great restriction with the distance of transmission, a lot of time was spent finding ways to increase the data sent with lower frequencies. As discussed in Chapter 1 the easy fixes to this problem is increasing the power and weight to the transceiver. Unfortunately both these push the weight and duration of system power restrictions considerably. Also important to note, is the weight and power requirements are much less strict on the trainer’s equipment. The trainer equipment was suggested to be replaced with a larger and more power intensive transmission system. Another key factor is height of transmitters, but as discussed previously the air force was very strict on how discrete the system was and the available heights were not helpful to the distance requirement. Another important factor is frequency [7]. The team worked on developing a system that would receive data at one speed and output it at another. This would obviously only work for short periods of time depending on how big the processor memory was, but the transmissions were supposed to be relatively brief. This idea showed promise for two techniques: store and forward and data buffering.

The time delay for store and forward would be considerable considering the lower frequency, completion of transmission, storing to non-volatile memory, and reading from memory all had to occur before anything was heard. This idea seemed most appealing since the data could be transmitted at a very low frequency and captured, thus
propagating the signal much further. This approach was denied by the air force due to the delay between transmission and actually hearing the transmission.

Data buffering was an idea for lower frequencies that were still close to MHz range. The code at the trainer processor would slowly fill a buffer while data was being sent at a slower rate from the buffer simultaneously. The code at the dog board would wait until its buffer was full, and then start pulling data out at the faster baud rate (See Figure 3). The frequency, and sampling audio rate were the two factors that could scale; the buffer used all the local memory in the processor. This would be better implemented if the RAM was larger. The code was developed but was also eventually denied by the air force; the air force later verified that the radios were going to be replaced with military grade radios that will perform better than what was available commercially anyways.
Figure 3 Buffer Flow Diagram
Chapter 9: Prototype Continued Work

While the proof of design worked, it was certainly bulky, extremely minimal from a user perspective, and did not meet some of the problem statement. To build the fieldable prototype the air force sent the proof of design along with the notes and specifications for a prototype to RNET, a company with experience manufacturing parts. The proof of design and specifications were explained to RNET. RNET developed an Android application that would replace the trainer board and an Android development system that took the place of the dog processor, although the protocol for communications and peripherals were maintained. Many approaches and code examples were explained and implemented to help port the design to the new system.

A system the team built for them was the API (application programming interface) to communicate from their system to the transceivers. This allowed for the Android application to easily swap settings between different transceivers, which is what the air force said would likely occur at the field tests.

Another system the team helped build for them was the camera detachment. Since all code needed to be an Android application and the proof of design used the supplied camera program, a new program that interpreted camera commands and data needed to be built. The specifications that came with the camera were very short and uninformative, so the team wrote specifications for binary commands, timing, and voltage requirements for the camera to be used in the application.
Chapter 10: Results

During the development process numerous tests were run to verify the validity of the system. The proof of concept, or second prototype as described earlier, was developed over numerous steps at which the system was tested. These steps were described in detail in Chapter 5, the fragmentation of the system. While many tests were run during each of these steps the tests and results described in this section only show the significant data as it pertains to the final proof of concept design and the suggested steps in the design of the prototype.

The two major components of the proof of concept system that would depend on certain conditions were the transceivers and the GPS. This document focused primarily on the processing component of this project, and while these two components were not the main point of this thesis, their results show the accuracy of the system as a whole.

The GPS was tested in multiple locations and during different weather conditions to determine its accuracy. The locations would heavily impact the time in which data was received and accuracy of the data was when multiple high altitude objects (buildings) were nearby. The conditions that heavily impacted the time data was received and the accuracy of the data was cloudy and/or stormy weather. The fields mentioned were tested many times and in multiple locations on multiple days, these readings were averaged and extreme outliers were removed (for example never connecting, which occurred during
heavy cloud and many buildings but would eventually receive data when moving to more open areas.)

Figure 4 GPS Accuracy

The transceivers were tested in three environments: altitude changing (hilly), altitude unchanging (flat), city (buildings nearby), and forested (many trees, but otherwise flat). The forested area was densely populated at some points along approximately half the distance and brush populated the other half. The transceivers were specified to work up to sixty miles line of sight; while we were unable to test this we did verify altitude unchanging up to 2 miles. The other two conditions would yield drastically different results depending on how much change was involved in the line of sight. Also, the transceivers used CRC retransmit on error; therefore accuracy is determined by the
number of dropped packets. Packets would begin to be dropped much more frequently immediately after line of sight was cut off, but packets could still be reliably sent while close to line of sight. The graph represents the distance that data could be reliably sent and audio and picture data was received without significant notice from dropped packets. Within this reliable distance based on the environment the percentage of lost packets is shown. Also note that the forested area and altitude unchanging were limited due to amount of nearby flat locations, note the low percentage of packets dropped. These two tests would have been continued but due to limitations of nearby environments with these conditions, the team was unable to produce results that pushed the transceivers to their limit in these conditions.

![Transmission Distance (miles)](image)

Figure 5 Transmission Distance in varying conditions
Figure 6 Dropped Packets in varying conditions
Chapter 11: Conclusion

At the beginning of this paper the wireless dog system was described and problem statements overviewed. Some of the major hurdles included long range transmission and live data feed. The goal was to produce a proof of design that could be followed up with a field-able prototype. Throughout the project, there were specifications that did not need to be followed through since the project was a proof of design; but recommendations were noted since additions and changes would be needed when building the prototype.

The final proof of design concept, described as the second prototype, accomplished the majority of the problem statement with a bare bone system. The two requirements not met were power sustainability and transmission distance. Both were researched and suggested to the prototype build and thus overlooked as expected in a proof of design build. Much of the work building the proof of design was combining multiple protocols and components with different timing problems. The ARM processor has a lot of capabilities and Keil’s evaluation board allowed easy access to many of those capabilities.
List of References


Appendix

Processor Code:

```c
/* Define clocks */
#define XTAL (12000000UL)    // Oscillator frequency
#define OSC_CLK (XTAL)       // Main oscillator frequency
#define RTC_CLK (32000UL)     // RTC oscillator frequency
#define IRC_OSC (4000000UL)   // Internal RC oscillator frequency

/ * Clock Variable definitions */
#define SystemFrequency IRC_OSC; // System Clock Frequency (Core lock)

/**
 * Initialize the system
 *
 * @param  none
 * @return none
 *
 * @brief Setup the microcontroller system.
 * Initialize the System and update the SystemFrequency
 */

void SystemInit (void)
{
  #if (CLOCK_SETUP)
    // Clock Setup
    LPC_SC->SCS = SCS_Val;
    if (SCS_Val & (1 << 5)) { // If Main Oscillator is enabled
      while ((LPC_SC->SCS & (1<<6)) == 0); // Wait for Oscillator to be ready
    }

    LPC_SC->CLKCFG = CCLKCFG_Val;  // Setup Clock Divider
    LPC_SC->PCLKSEL0 = PCLKSEL0_Val;  // Peripheral Clock Selection
    LPC_SC->PCLKSEL1 = PCLKSEL1_Val;
    LPC_SC->CLKSRCSEL = CLKSRCSEL_Val; // Select Clock Source for PLL0
  #if (PLL0_SETUP)
    LPC_SC->PLL0CFG = PLL0CFG_Val;
    LPC_SC->PLL0CON = 0x01; // PLL0 Enable
    LPC_SC->PLL0FEED = 0xA0;
    LPC_SC->PLL0FEED = 0x55;
    while (!(LPC_SC->PLL0STAT & (1<<26))); // Wait for PLOCK0
    LPC_SC->PLL0CON = 0x03; // PLL0 Enable & Connect
    LPC_SC->PLL0FEED = 0xA0;
  #endif
  
  SystemFrequency = IRC_OSC; // System Clock Frequency (Core lock)
}
```

```c
void SystemInit (void)
{
  #if (CLOCK_SETUP)
    // Clock Setup
    LPC_SC->SCS = SCS_Val;
    if (SCS_Val & (1 << 5)) { // If Main Oscillator is enabled
      while ((LPC_SC->SCS & (1<<6)) == 0); // Wait for Oscillator to be ready
    }

    LPC_SC->CLKCFG = CCLKCFG_Val;  // Setup Clock Divider
    LPC_SC->PCLKSEL0 = PCLKSEL0_Val;  // Peripheral Clock Selection
    LPC_SC->PCLKSEL1 = PCLKSEL1_Val;
    LPC_SC->CLKSRCSEL = CLKSRCSEL_Val; // Select Clock Source for PLL0
  #if (PLL0_SETUP)
    LPC_SC->PLL0CFG = PLL0CFG_Val;
    LPC_SC->PLL0CON = 0x01; // PLL0 Enable
    LPC_SC->PLL0FEED = 0xA0;
    LPC_SC->PLL0FEED = 0x55;
    while (!(LPC_SC->PLL0STAT & (1<<26))); // Wait for PLOCK0
    LPC_SC->PLL0CON = 0x03; // PLL0 Enable & Connect
    LPC_SC->PLL0FEED = 0xA0;
  #endif
  
  SystemFrequency = IRC_OSC; // System Clock Frequency (Core lock)
}
```
LPC_SC->PLL0FEED = 0x55;
#endif

#if (PLL1_SETUP)
LPC_SC->PLL1CFG = PLL1CFG_Val;
LPC_SC->PLL1CON = 0x01;            // PLL1 Enable
LPC_SC->PLL1FEED = 0xAA;
LPC_SC->PLL1FEED = 0x55;
while (!(LPC_SC->PLL1STAT & (1<<10))); // Wait for PLOCK1
LPC_SC->PLL1CON = 0x03;            // PLL1 Enable & Connect
LPC_SC->PLL1FEED = 0xAA;
LPC_SC->PLL1FEED = 0x55;
#else
LPC_SC->USBCCLKCFG = USBCCLKCFG_Val; // Setup USB Clock Divider
#endif
LPC_SC->PCONP = PCONP_Val;            // Power Control for Peripherals
LPC_SC->CLKOUTCFG = CLKOUTCFG_Val;   // Clock Output Configuration
#endif

/*********************************************************
 * Data Filter
 * @param none
 * @return none
 * @brief Filters incoming data from transceiver, searching for the
 * flag 0x56. Then filters data until command is recognized, and
 * routes to appropriate output.
 * 
 * *******************************************************/

void DataHandling(void)
{
if (checkreceiveport(1)) {  
    receive = SER_getChar (1);  // Get Char from serial line 1

    // Filter Input, determine if Picture command
    // If Picture command determine what type of command

    if (receive == 0x56) {
        receive = SER_getChar (1);
        if (receive == 0x00) {
            receive = SER_getChar (1);
            if (receive == 0x26) {
                PicReset = 1;
                receive = SER_getChar (1);
            }
            else if (receive == 0x24) {

37
PicBaud = 1;
receive = SER_getChar (1);
}
else if (receive == 0x36) {
receive = SER_getChar (1);
if (receive == 0x01) {
PicStop = 1;
}
if (receive == 0x00) {
PicTake = 1;
}
}
}
else if (receive == 0x34) {
PicSize = 1;
receive = SER_getChar (1);
}
else if (receive == 0x31) {
receive = SER_getChar (1);
if (receive == 0x05) {
receive = SER_getChar (1);
if (receive == 0x01) {
PicCompR = 1;
receive = SER_getChar (1);
CompR = SER_getChar (1);
}
if (receive == 0x04) {
PicImSize = 1;
receive = SER_getChar (1);
ImSize = SER_getChar (1);
}

// If data comes in port 0 send it to 1 for transmission
if (checkreceiveport(0)) {
    receive = SER_getChar(0); // Get Char from serial line 0
    SER_putChar(1,receive); // Write Char to serial line 1
}

// Data passing for all camera commands
if (PicReset == 1) {
    // Initialize
    SER_putChar(0,0x56);
    SER_putChar(0,0x00);
    SER_putChar(0,0x26);
    PicReset = 0;
}
else if (PicTake == 1) {
    // Get GPS data first
GetGPS(); // Call the function for reading GPS string

GPScount = 0;

// Continue calling function until good data obtained
while(((Lat31 > 0x35) || ((Lat29 < 0x32) && (Lat28 == 0x30))) && (GPScount < 100)){
    GetGPS();
    GPScount++;
}

// Read the data out of the array
while(GPScount != 74){
    charit = GPS[GPScount];
    SER_putChar(1, charit);
    GPScount++;
}

// Take picture
SER_putChar(0, 0x56);
SER_putChar(0, 0x00);
SER_putChar(0, 0x36);
SER_putChar(0, 0x01);
SER_putChar(0, 0x00);
PicTake = 0;
}

if (PicSize == 1) {
    // Read size
    SER_putChar(0, 0x56);
    SER_putChar(0, 0x00);
    SER_putChar(0, 0x34);
    SER_putChar(0, 0x01);
    SER_putChar(0, 0x00);
    PicSize = 0;
}

if (PicRead == 1) {
    // Read picture
    SER_putChar(0, 0x56);
    SER_putChar(0, 0x00);
    SER_putChar(0, 0x32);
    SER_putChar(0, 0x0C);
    SER_putChar(0, 0x00);
    SER_putChar(0, 0x00);
    SER_putChar(0, 0x00);
    SER_putChar(0, PicSize1);
    SER_putChar(0, PicSize2);
    SER_putChar(0, 0x00);
    SER_putChar(0, 0x0A);
PicRead = 0;
}

if (PicStop == 1) {
    SER_putchar(0x56);
    SER_putchar(0x00);
    SER_putchar(0x36);
    SER_putchar(0x01);
    SER_putchar(0x03);
    PicStop = 0;
}

if (PicCompR == 1) {
    SER_putchar(0x56);
    SER_putchar(0x00);
    SER_putchar(0x31);
    SER_putchar(0x05);
    SER_putchar(0x01);
    SER_putchar(0x01);
    SER_putchar(0x12);
    SER_putchar(0x04);
    SER_putchar(0xCompR);
    PicCompR = 0;
}

if (PicImSize == 1) {
    SER_putchar(0x56);
    SER_putchar(0x00);
    SER_putchar(0x31);
    SER_putchar(0x05);
    SER_putchar(0x04);
    SER_putchar(0x01);
    SER_putchar(0x00);
    SER_putchar(0x19);
    SER_putchar(0xImSize);
    PicImSize = 0;
}

if (PicBaud == 1) {
    // Change Baud to 384
    ChangeBaud(3);

    // Change Baud to 115
    SER_putchar(0x56);
    SER_putchar(0x00);
    SER_putchar(0x24);
    SER_putchar(0x03);
    SER_putchar(0x01);
    SER_putchar(0x0D);
    SER_putchar(0xA6);
    my_delay(100);
    ChangeBaud(1);
    PicBaud = 0;
}
A/D Conversion and Transmission

@param  none
@return none

@brief Performs A/D Conversion with an option of scaling data with a volume. Data is then scanned if it is the picture command flag, if not then the data is sent to the serial port.

if ((IOPIN0 & 0x00004000) == 0) {
    AD0CR |= 0x01000000;  // Start A/D Conversion
    while ((val & 0x80000000) == 0) {  // Wait for end of A/D Conversion
        val = AD0DR0;  // Read A/D Data Register
    }
    AD0CR &= ~0x01000000;  // Stop A/D Conversion
    Volume = .75;  // Volume Level (0-255)

    val = (val >> 6) & 0x03FF;  // Extract AIN0 Value
    data = val;

    if(!(val == 0x56))  // Filter for Flag
        temp = sendchar (data,1);  // Send data
}

ADC Initializer

@param  none
@return none

@brief Initializes the pins and registers for the ADC

void ADC_init (void) {
    LPC_PINCON->PINSEL1 &= ~(3<<14);
    LPC_PINCON->PINSEL1 |= (1<<14);  // Enable pin 25

    LPC_SC->PCONP |= (1<<12);  // Enable power to ADC block
    LPC_ADC->ADCR = (1<< 0) | (1<< 8) | (1<<21);  // select AD0.2 pin ADC clock is 25MHz/5 enable ADC

LPC_ADC->ADINTEN = (1<< 8);     // global enable interrupt
LPC_PINCON->PINSEL1 &= ~(3<<20); // Enable the DAC
LPC_PINCON->PINSEL1 |= (1<<21);
NVIC_EnableIRQ(ADC_IRQn);     // enable ADC Interrupt
}

/*---------------------------------------------------------------
  *  Baud Rate Changer
  *  *
  *  * @param  Baud : determines what Baud rate for the processor (1,2,3,4)
  *  * @return none
  *  *
  *  * @brief  Changes the Baud rate of the processor to one of the four
  *  * options based on the input to the function
  *  *----------------------------------------------------------------
  *---------------------------------------------------------------*/

void ChangeBaud(int Baud) {
  LPC_UART_TypeDef *pUart;
  pUart = (LPC_UART_TypeDef *)LPC_UART0;
  pUart->LCR = 0x83;       // 8 bits, no Parity, 1 Stop bit
  if (Baud == 1) {
    pUart->DLL = 0x09;
    pUart->FDR = 0x21; // 115200 Baud Rate @ 25.0 MHZ PCLK
  }
  if (Baud == 2) {
    pUart->DLL = 0x12;
    pUart->FDR = 0x21; // 57600 Baud Rate @ 25.0 MHZ PCLK
  }
  if (Baud == 3) {
    pUart->FDR = 0xA4; // FR 1,507, DIVADDVAL = 1,      
    pUart->DLL = 0x1D; // MULVAL = 10 38400 Baud Rate
                          // 38400 Baud Rate @ 25.0MHz PCLK
  }
  if (Baud == 4) {
    pUart->DLL = 0x7A; // 9600 Baud Rate @ 25.0MHz PCLK
    pUart->FDR = 0x31; // FR 1,507, DIVADDVAL = 1,
                        // MULVAL = 2 9600 Baud Rate
  }
  pUart->DLM = 0;       // High divisor latch = 0
  pUart->LCR = 0x03;    // DLAB = 0
}

/*---------------------------------------------------------------
  *  Write character to Serial Port
  *  *
  *  * @param  uart : determines which uart controller you write to
  *---------------------------------------------------------------*/
* (1,2,3,4)
  c : the character to be written
* @return c : the character to be written
* @brief Writes the input character to the input serial port.
*---------------------------------------------------------------------*/

int SER_putChar (int uart, int c) {
  LPC_UART_TypeDef *pUart;

  if (uart == 0) {
    pUart = (LPC_UART_TypeDef *)LPC_UART0;
  }
  if (uart == 1) {
    pUart = (LPC_UART_TypeDef *)LPC_UART1;
  }
  if (uart == 2) {
    pUart = (LPC_UART_TypeDef *)LPC_UART2;
  }
  if (uart == 3) {
    pUart = (LPC_UART_TypeDef *)LPC_UART3;
  }

  while (!(pUart->LSR & 0x20));
  return (pUart->THR = c);
}

// * Blocking Read
* @param uart : determines which uart controller you read (0,1,2,3)
* @return RBR : the character to be read
* *
* @brief Read RBR character from uart Serial Port and block port
*---------------------------------------------------------------------*/

int SER_getChar (int uart) {
  LPC_UART_TypeDef *pUart;

  if (uart == 0) {
    pUart = (LPC_UART_TypeDef *)LPC_UART0;
  }
  if (uart == 1) {
    pUart = (LPC_UART_TypeDef *)LPC_UART1;
  }
  if (uart == 2) {
    pUart = (LPC_UART_TypeDef *)LPC_UART2;
  }
  if (uart == 3) {
    pUart = (LPC_UART_TypeDef *)LPC_UART3;
  }

  while (!(pUart->LSR & 0x01));
return (pUart->RBR);
}

/* Non-Blocking Read
 * @param uart : determines which uart controller you read (0,1,2,3)
 * @return RBR : the character to be read
 * *
 * @brief Read RBR character from uart Serial Port and block port
 */

int SER_getChar_nb (int uart) {
LPC_UART_TypeDef *pUart;

if (uart == 0) {
    pUart = (LPC_UART_TypeDef *)LPC_UART0;
} if (uart == 1) {
    pUart = (LPC_UART_TypeDef *)LPC_UART1;
} if (uart == 2) {
    pUart = (LPC_UART_TypeDef *)LPC_UART2;
} if (uart == 3) {
    pUart = (LPC_UART_TypeDef *)LPC_UART3;
}

if (pUart->LSR & 0x01)
    return (pUart->RBR);
else
    return 0;
}

/* Serial Port Checker
 * @param uart : determines which uart controller you read (0,1,2,3)
 * @return none
 * *
 * @brief Check if there is a character on the Serial Port
 */

int checkreceiveport(int uart) {
LPC_UART_TypeDef *pUart;

// Switch based on uart input to set uart to correct port
if (uart == 0) {
    pUart = (LPC_UART_TypeDef *)LPC_UART0;
}
if (uart == 1) {
    pUart = (LPC_UART_TypeDef *)LPC_UART1;
}
if (uart == 2) {
    pUart = (LPC_UART_TypeDef *)LPC_UART2;
}
if (uart == 3) {
    pUart = (LPC_UART_TypeDef *)LPC_UART3;
}

// If anything is on port return 1, otherwise return 0
if (pUart->LSR & 0x01) {
    return 1;
} else {
    return 0;
}

/**************************************************************************/
* UART Initializer
* @param   uart : determines which uart controller you read (0,1,2,3)
* @return  none
* *
* @brief    Initializes pins and registers for UART port
*-----------------------------------------------------------------------*/

void SER_init (int uart) {

    LPC_UART_TypeDef *pUart;

    if (uart == 0) {                               /* UART0 */
        LPC_PINCON->PINSEL0 |= (1 << 4);  // Pin P0.2 used as TXD0 (Com0)
        LPC_PINCON->PINSEL0 |= (1 << 6);  // Pin P0.3 used as RXD0 (Com0)

        pUart = (LPC_UART_TypeDef *)LPC_UART0;
    }

    if (uart == 1) {                               /* UART1 */
        LPC_PINCON->PINSEL4 |= (2 << 0);  // Pin P2.0 used as TXD1 (Com1)
        LPC_PINCON->PINSEL4 |= (2 << 2);  // Pin P2.1 used as RXD1 (Com1)

        pUart = (LPC_UART_TypeDef *)LPC_UART1;
    }

    if (uart == 2) {                               /* UART2 */
        LPC_PINCON->PINSEL0 |= (1 << 20); // Pin P1.2 used as TXD0 (Com2)
        LPC_PINCON->PINSEL0 |= (1 << 22); // Pin P1.3 used as RXD0 (Com2)

        pUart = (LPC_UART_TypeDef *)LPC_UART2;
    }

    if (uart == 3) {                               /* UART3 */
        LPC_PINCON->PINSEL0 |= (1 << 1);  // Pin P0.0 used as TXD0 (Com3)

        pUart = (LPC_UART_TypeDef *)LPC_UART3;
    }
LPC_PINCON->PINSEL0 |= (1 << 3); // Pin P0.1 used as TXD0 (Com3)

pUart = (LPC_UART_TypeDef *)LPC_UART3;
}

pUart->LCR = 0x83; // 8 bits, no Parity, 1 Stop bit
pUart->DLL = 0x09;
pUart->FDR = 0x21; // 115200 Baud Rate @ 25.0 MHZ PCLK

//pUart->DLL = 0x12;
//pUart->FDR = 0x21; // 57600 Baud Rate @ 25.0 MHZ PCLK

//pUart->DLL = 0x7A; // 9600 Baud Rate @ 25.0MHZ PCLK
//pUart->FDR = 0x31; // FR 1,507, DIVADDVAL = 1,
// MULVAL = 2

//pUart->FDR = 0xA4; // 38400 Baud Rate @ 25.0MHZ PCLK
//pUart->DLL = 0x1D; // 38400 Baud Rate @ 25.0MHZ PCLK

pUart->DLM = 0; // High divisor latch = 0
pUart->LCR = 0x03; // DLAB = 0

void GetGPS(void){

charit = 0;

// Loop through characters until the $ is reached denoting
// the beginning of the GPS string
while (receive != 0x24) {
    receive = SER_getChar (2);
    receive = SER_getChar (2);
}

// Loop through GPS String, 74 characters long
while (charit <= 74) {
receive = SER_getChar (2); // Grab character from port
GPS[charit] = receive;  // Put character in array
receive++;             // Iterate pointer into array

/************************************************************************
* Reverse Buffering Code
*
* @param none
* @return none
*
* @brief Buffers input and output to allow reading and writing of
* different speeds
************************************************************************/

// If data is received then prepare to output
if (pUart->LSR & 0x01) {
while (1) {
    // If Buffer is not full fill it with data first
    if (FillBuff == 0) {
        while (Index < 0x00001FF0) {
            if (pUart->LSR & 0x01) {
                slice = SER_getChar (1);
                SoundBuffer[Index] = slice;
                Index++;
            }
        }
        FillBuff = 1;
    }
    // If serial data is ready to come in grab it
    if (pUart->LSR & 0x01) {
        if (Index < 0x00001FFC) {
            slice = SER_getChar (1);
            SoundBuffer[Index] = slice;
            Index++;
        } else {
            Index = 0;
            slice = SER_getChar (1);
            SoundBuffer[Index] = slice;
            Index++;
            Loop++;
        }
    }
}
// While SIndex != Index output sound data from buffer
if (SIndex != Index) {
    if (Index < 0x00001FFC) {
        pslice = SoundBuffer[SIndex];
        SIndex++;
        // Set Speaker Output
        DACR = (pslice + 0x8000) & 0xFFFF;
        my_delay(1);
    }
    else {
        SIndex = 0;
        pslice = SoundBuffer[SIndex];
        SIndex++;
        // Set Speaker Output
        DACR = (pslice + 0x8000) & 0xFFFF;
        my_delay(1);
    }
}

// If SIndex == Index reset buffer and indexes
if (SIndex == Index) {
    while (reset < 0x00001FFC) {
        SoundBuffer[reset] = 0;
        reset++;
    }
    SIndex = 0;
    Index = 0;
}