NON-BLOCKING HARDWARE CODING

FOR EMBEDDED SYSTEMS

by

Derek Caleb Klein

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Engineering

Boise State University

May 2011

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Derek Klein

Thesis Title: Non-Blocking Hardware Coding for Embedded Systems

Date of Final Oral Examination: 16 March 2011

The following individuals read and discussed the thesis submitted by student Derek Klein, and they also evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination, and that the thesis was satisfactory for a master's degree and ready for any final modifications that they explicitly required.

Sin Ming Loo, Ph.D.	Chair, Supervisory Committee
Thad B. Welch, Ph.D.	Member, Supervisory Committee
Hao Chen, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Sin Ming Loo, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Sin Ming Loo, for his guidance and patient support throughout this process. His contribution has been essential to my academic pursuits.

Additionally, I would like to thank those who were involved in this research: Josh Kiepert, Jim Hall, Michael Pook, Kelsey Drake, and Ross Butler. Their significant contributions made this research possible. I would like especially thank Josh Kiepert, Jim Hall, and Michael Pook for their participation in the implementation of several of the non-blocking techniques discussed in this thesis. Michael Pook deserves further thanks for his assistance in the final editing of this thesis.

Finally, I would like to thank my family. Their love and support has made my success in academics and life in general possible. I would like to specifically acknowledge my brother, Daren Klein, who inspired me to go to college with his dedication to overcoming dyslexia while learning to read. My mother deserves special thanks for putting up with me as my teacher and mother growing up. A person could not ask for a better mother. Thanks, mom.

This work is funded by FAA Cooperative Agreement No. 04-C-ACE-BSU and 07-C-RITE-BSU[1].

[1] Although the FAA has sponsored this project, it neither endorses nor rejects the findings of this research. The presentation of this information is in the interest of invoking technical community comment on the results and conclusions of the research.

ABSTRACT

NON-BLOCKING HARDWARE CODING

FOR EMBEDDED SYSTEMS

Derek Klein

Master of Science in Computer Engineering

Embedded Systems can be found in devices that people use every day. In the pursuit of faster and smarter devices, more powerful processing units are needed in these embedded systems. A key component of powerful processing units is the supporting software. While the raw processing power of microcontroller has been continually advancing, the improvements in the supporting software for medium scale embedded systems have been lacking. This thesis focuses on improving the software on medium scale systems by discussing the practical application of non-blocking coding techniques. The basic concept of how non-blocking code improves the performance of a system is relatively easy to understand. However, non-blocking code is considerably more challenging to implement in practice. This thesis shows that, by utilizing some commonly known coding techniques and practices together in a systematic manner, it is possible to obtain practical non-blocking software on medium scale embedded systems. It was found that under certain conditions more than 20% of the total processor time can be saved by converting a blocking I^2C driver to non-blocking. The freed processing time improved the performance of the network tasks by increasing the throughput from 68% to 100%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
DC	Direct Current
FAA	Federal Aviation Administration
FAT	File Allocation Table
FIFO	First In First Out
GPIO	General Purpose Input/Output
I/O	Input/Output
I^2C	Inter-Integrated Circuit
ISR	Interrupt Service Routine
OS	Operating System
PMON	Personal Monitor
SD	Secure Digital
SDIO	Secure Digital Input Output
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TWI	Two Wire Interface
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Buss
UART	Universal Asynchronous Receiver/Transmitter

CHAPTER 1: INTRODUCTION

1.1 Embedded Systems

An embedded system is an electronic system that is part of a larger system. They can be found in devices that most people use on a daily basis. For example, embedded systems can be found in everything from communication systems (e.g., cellular phones, radios, etc.) to household appliances (e.g., dishwashers, refrigerators, etc.). Consumers continue to desire faster and smarter features on our devices and appliances that require more powerful processing units. A key component of a powerful processing unit is its supporting software. This thesis will discuss some practical concepts that can greatly improve the supporting software of embedded systems.

As the demands of embedded systems continue to grow, there is a need to produce systems that are more flexible, responsive, robust, and cost effective. For example, the original home thermostats were electromechanical systems that used bimetal strips to simply open and close a circuit based on the ambient temperature. The user interface was simple. Due to the demand for a better user interface and power management, today's thermostats have considerably more functionality. The ability to schedule different temperature settings for different times of the day is common for thermostats currently on the market. Some of the nicer thermostats even have touch screen interfaces. These additional features require a more advanced embedded system with supporting software.

Embedded hardware is continually adapting to meet the increasing requirements of more demanding applications. One area where hardware technology has made notable increases is the processing units on embedded systems. Depending on the type of embedded system, the processing unit is typically a microprocessor or one of many different types of microcontrollers. The main difference between microprocessors and microcontrollers is that microprocessors are only composed of a processing unit. Whereas, microcontrollers include timers, memory, and specialized I/O hardware along with the processor in a single package. The exact quantity, variety, and type of specialized hardware on an embedded system are dependent on the application and embedded system complexity. For all types of embedded systems, the hardware capabilities such as available memory, processing speed, power efficiency, interface flexibility, and cost effectiveness are continually advancing.

Advances in the supporting software are needed to take full advantage of the improvements in hardware. Hardware and software need to work together to create the powerful processing units in embedded systems today. Without software, the hardware will be useless. Without hardware, the software will have no place to run. Even with the advancements continually being made, embedded systems hardware still has limited resources. Two of the most common limitations are memory and processor capacity. In order to make up for these limitations, some embedded hardware (particularly microcontrollers) has specialized devices that are designed to optimize specific tasks. Embedded software is responsible for managing hardware resources, processing the data, and controls how a system operates. In order to take advantage of advances in hardware, the software needs to make effective use of the available hardware capabilities. What is meant by “effective use” is that the software does not add unnecessary obstacles to the performance of the hardware. For example, blocking code (a piece of code that prevents other processes from running while it waits for information to complete execution) in a hardware driver might prevent another specialized piece of hardware from being fully utilized. Development of effective non-blocking software is highly dependent on the hardware platform. The focus of this thesis is to demonstrate a systematic way of applying non-blocking coding practices to embedded systems in such a way as to take full advantage of advances in embedded hardware.

1.2 Embedded System Categories

There is a wide variety of embedded systems, so it is difficult to have a set of coding techniques apply to every type of system. The coding techniques and practices discussed in this thesis primarily apply to a limited category of embedded systems. So it is important to clarify what type of embedded system will be the focus of this thesis. There are multiple ways to categorize embedded systems. Embedded systems could be categorized based on the type of application, cost of the system, system functionality, complexity, or hardware capabilities. The capability of the hardware is one particularly relevant aspect for embedded software. For the purpose of this thesis, embedded systems will be organized into three categories: small scale, medium scale, and large scale, based on the hardware capabilities of the system. A description of these categories is provided in this section and summed up in Table 1.1.

Table 1.1: Embedded System Catagorization

Category	**Category Description**
Small Scale	Capable of supporting 1 to 2 applications. Typically on smaller 8-bit microcontrollers that do not have enough memory to support an operating system.
Medium Scale	Capable of supporting multiple applications. Typically on 32-bit and smaller microcontrollers that do not have enough memory to support an embedded operating system.
Large scale	Capable of supporting multiple applications. Typically on large microcontrollers or microprocessors that have more than enough memory to support an embedded operating system.

For this thesis, small scale systems will be defined as being capable of supporting only one or two applications. These systems typically have smaller 8-bit microcontrollers with very limited memory. An example of this type of system would be a garage door opener. A garage door opener has one application that monitors a button and sends a signal to the garage door controls when the button is pushed. The application is composed of two tasks: the first is monitoring the button and the second is communicating wirelessly with the garage door controls.

Medium scale systems are capable of supporting multiple applications running at the same time. For this thesis, medium scale systems will be defined as systems that have enough hardware capabilities to support multiple applications but do not have the hardware capability to practically support an operating system at the same time. A good example of this would be the controller in a refrigerator. The control system on an advanced refrigerator is responsible for making ice and dispensing water in addition to controlling the temperature of two separate compartments. Each of these responsibilities is composed of multiple tasks themselves. Since the fridge's responsibilities do not change over time, a processing unit with expansion capabilities or an operating system is not practical.

Large scale systems are similar to general computing systems (e.g., desktop computers) in that they are capable of supporting a wide variety of applications. They typically have an abundance of memory to support their various functions. Large scale systems typically have enough memory and processing speed to support a compact operating system. A smart phone is a good example of a large scale embedded system.

1.3 Software for Sensor Networks

A sensor network is a good example of a medium scale embedded system. Like the name suggests, the sensor network's main purpose is to collect data from a network of sensors. A network is composed of multiple sensor nodes. The number and variety of sensors on each node is dependent on the sensing application. The data collected from each sensor node is typically transmitted across a wireless network to a central location for processing. So, with each sensor node, there are multiple applications that often need to operate at the same time. The data collection itself is often broken up into multiple tasks. The drivers for the interfacing hardware, the initial processing of the data, and scheduling the intervals for collecting data are just a few of the more common tasks that are directly used to collect the sensor data. In addition to collecting sensor data, transmitting the data across the network can involve multiple tasks. First, there is the driver for interfacing with the network radio. Second, if the system needs to be able to send the data to different locations on the network, a network manager/server has to be maintained by a task. Occasionally, the data is stored locally in addition to being transmitted. Sensor networks are typically designed to continuously monitor the subject of interest. Consequently, all of sensor node's tasks need to continuously run alongside each other. If one of the tasks is prevented from performing its duties, the entire system could be negatively affected. One solution to such a problem is to upgrade the system scale and include an operating system. However, since the general idea is to have a sensor network composed of a large number of nodes, it is desirable to keep each node as cost effective as possible. Thus, large scale embedded systems are impracticable for the typical sensor network.

1.4 Thesis Purpose

This thesis demonstrates coding techniques for non-blocking software on medium scale embedded systems. The scientific community has focused on establishing coding techniques and practices for large scale embedded systems. However, there is still a need for medium scale embedded systems in research and industry. Industry typically relies on the experience of designers to produce a quality non-blocking code. However, differences in the abilities of designers can lead to a variety of approaches that results in software that is difficult to maintain and port from one platform to another. A formulated systematic approach can help establish more consistent coding practices that are easier to port across platforms. By examining how the different aspects of non-blocking coding techniques work together, a systematic approach of applying these techniques to medium scale systems can be established.

1.5 Overview

The following chapters will discuss many of the aspects of non-blocking coding practices as they apply to medium scale embedded systems. This discussion will focus on sensor networks in particular. Chapter 2 will cover the background of embedded system software and how it relates to this thesis. Chapter 3 will provide a description of the sensor node hardware used for this research. Chapter 4 will discuss several coding techniques that are used in developing non-blocking code. Chapter 5 provides analysis for two non-blocking coding examples. Finally, the conclusions and possible future work are presented in Chapter 6.

CHAPTER 2: RESEARCH BACKGROUND

2.1 Limited Related Research

There is a significant amount of research focused on improving software in embedded systems. Unfortunately, most of the research is dedicated to large scale embedded systems. Over the years, there have been a lot of operating systems based software research developed on general computing systems (e.g., desktop computers). Presently, the hardware on large scale embedded systems has advanced to the point where an operating system is a practical option. Embedded software research is primarily focused on applying the operating system principle developed on general computing systems to large scale embedded systems. Consequently, there is a lot of potential with research focused on large scale embedded systems. Unfortunately, this research based on operating systems does not really apply to the medium scale embedded systems that are the focus of this thesis. Medium scale systems typically do not have the memory resources to support an embedded operating system. In addition to this, medium scale systems tend to be very specialized. This specialization makes applying generalized operating systems a challenging option. So developing a generic operating system that covers medium scale embedded systems is not practical.

To date, general coding guidelines for coding practices in medium scale embedded systems have not been established in the industry. There is very little published research that is applicable to medium scale systems. The only published research found that relates to medium scale systems have to do with schedulers [1]. Although schedulers are a crucial component of embedded systems with or without an operating system, they are not the only important aspect to which coding guidelines could be applied. With the wide variety of specialized medium scale embedded systems, it is also difficult to establish coding techniques that apply directly to every application. Consequently, coding techniques for these types of systems have been gained primarily through personal experience and instruction. Some of these techniques are discussed in Section 2.2. With such a limited amount of research for medium scale systems, there is still a need to develop effective coding practices that are not based on an operating system. This thesis attempts to demonstrate techniques and methods that will produce effective non-blocking software that can be applied to these types of systems.

2.2 Common Coding Practice

Since there is limited published research on coding practices in embedded systems, most coding practices are based on experience. This experience typically consists of problems encountered and instruction from experienced colleagues. Embedded system designers often have different experience based on the variety of hardware platforms on which they have worked. Consequently, coding practices can vary quite a bit among designers. In spite of this, certain common practices have been developed. Most of the practices stem from the need to use a common framework for development and the clear benefits of each technique/practice. These common practices cover a wide range of applications and systems. The more notable practices are related to interfacing with hardware, and the main system loop.

2.2.1 Hardware Interface

The main purpose for most embedded systems is interfacing with hardware whether it is a sensor, control device (e.g., DC motor, Solenoid etc.), or communication (e.g., UART, wireless transceiver, etc.). There are many different approaches taken to interfacing with hardware. Three of the more notable approaches are polling, waiting, and interrupting.

Polling can be used by itself or in concert with the other two approaches. As a result, polling is a little bit harder to distinguish from the others. The basic idea of polling is that the condition of some hardware is periodically checked. When polling is used by itself, a task/application that is accessing the hardware will periodically check the status registers associated with the hardware. If the status registers indicate that the hardware is ready, the task will perform the related operations and return control back to the scheduler. If the status registers indicate that the hardware is not ready, then the task will return the control back to the scheduler without performing the related action. The problem with polling by itself is that there is high probability of missing a hardware action while another task is being handled.

Another common method is the wait-poll approach. When a hardware interface function is called by an application, the function waits until the hardware action is complete. For example, suppose an application needs to access the value of an analog-to-digital converter (ADC). The process flow of the ADC reading function using this method is shown in Figure 1.1. At the beginning of the function, the hardware to starts an ADC read. The function then waits for the ADC hardware to finish. This usually involves polling a flag in a hardware register. After the ADC finishes, the function returns the result from the ADC register. The advantage of this method of interfacing with the hardware is that it is very simple to implement. Unfortunately, this blocks other potentially useful work from being completed. Processor cycles are wasted during each wait.

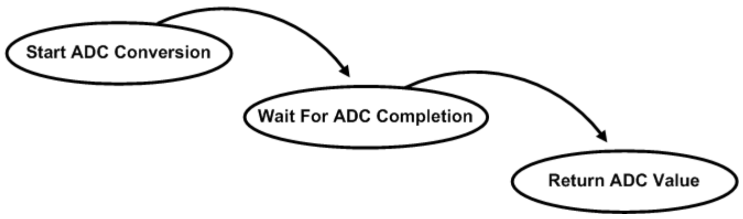

Figure 2.1: ADC Reading Function

Another approach to interfacing with hardware is through the use of interrupts. Most of the hardware interfaces on microcontrollers have interrupts that are triggered when either a monitored device changes state or a particular hardware task completes. When an interrupt occurs, the microcontroller will pause the current process and execute an interrupt service routine (ISR). The ISR will perform the operations for the hardware that triggered the interrupt. It is important to keep an ISR short, because it will prevent other lower level interrupts from occurring. So, it is generally not a good idea to call functions from inside an ISR (especially I/O functions such as "printf"). Sometimes, in order to keep the entire ISR short, only the absolutely necessary operations are performed inside the ISR. The rest of the operation is performed by a task that polls a flag set by the

ISR. Interrupts are important to non-blocking systems as they give an effective alternative to the wait-poll method. Unfortunately, interfacing with an interrupt is not always trivial. When using an interrupt, it is necessary to figure out how much of a given process must occur in the interrupt. Typically, for an interrupt to be used effectively, it has to control a state machine. Sometimes this is done indirectly through the use of flags. Other times, the interrupt service routine has to modify the state machine directly. One must carefully balance the time critical nature of a given process against the need to keep ISR's as short as possible.

2.2.2 Main Function

One of the most common practices in medium scale embedded systems is related to the centralized control of the main function. The main by default is the first function executed upon startup. One thing that medium embedded systems have in common is that there are a variety of settings that have to be initialized at start up. Additionally, there are typically a few tasks that continually run. Consequently, a typical main will have an initialization routine followed by a while loop like the example in Figure 2.2. Notice that the initialization routines are executed first and are responsible for setting all of the hardware registers to the appropriate values for the applications involved. The actual organization of the initialization sequence will depend on the hardware platform and the designer’s preferences. The next piece of code to be executed is the main loop. The main while loop is the core of a typical embedded system. It is responsible for continually updating the tasks for the different applications running on the system. While there are different scheduling systems, some of which are discussed in Section 2.4, this type is the most common. This scheduling routine is commonly referred to as the super loop (an infinite loop). There are a couple of reasons this method is commonly used. The first is that it is simple. Each task will eventually get its turn to run as long as none of the tasks block indefinitely. The other reason is that embedded systems operate continually. The main while loop will continue to execute the application tasks indefinitely as long as there is not a catastrophic failure.

```
main()
{
         Init_sensor1();  // setup i/o register and calibration values
         Init_uart();   // setup baud rate and port
        While()
        {
                Interface_sensor1();   //get data from sensor1
                 Interface_uart();   // output collected data to uart and
                                    //  send status message
        }
 }
```

Figure 2.2: Common Main Function in Embedded Systems

The potential problem with this method is that, if one task blocks for too long, then the other processes will be losing processing time. So, it is important to minimize the execution length of each task. Most designers are aware of this potential problem. Unfortunately, the effective ways to implement non-blocking techniques covered in this thesis are not universally understood. Consequently, blocking techniques are still commonly used in these types of embedded systems. For example, it is still common to have I/O calls to use the wait-poll method described in Section 2.2. This is typically used when accessing hardware that has a more defined communication length (e.g., an I^2C read or write). Many of the problems from blocking code will not create a problem until the hardware platform is pushed to its limits, such as when the processing speed is barely fast enough to execute all of the tasks in time. Another situation were a system is pushed to its limits is when the particular piece of code is continually needing to be processed. On the other hand, systems that are not pushed to their limits do not always suffer the same detrimental effects from blocking software. Consequently, the individual designers still frequently use blocking calls in certain situations based on their experience.

2.3 Operating Systems

Operating systems are composed of multiple aspects and components that make them hard to define. One way of viewing an operating system is as a resource allocator [2]. In other words, an operating system is designed to manage the data, processes, and hardware. The purpose of an operating system is to create an easy-to-use interface that effectively uses the hardware. Operating systems have been around for some time in general computing systems. Consequently, most of the aspects of operating systems have been well established. The operating system program itself is referred to as a kernel. A kernel is a standalone process that directly controls all of the aspects of the operating systems. A few of the more notable aspects that the kernel is responsible for managing are dynamic memory, process scheduling, and I/O subsystem. The fact that operating systems have been well defined and provide easy portability makes them an attractive option for embedded systems today.

2.3.1 Dynamic Memory

Memory usage is a problem faced by all computing systems. In any system, there is a limited amount of memory available. Processes can require various amounts of memory at different times during their given tasks. An operating system minimizes the amount of memory consumed by dynamically allocating memory. By dynamically allocating memory when it is needed, memory can be shared and re-use by multiple processes. So, when one process is finished using a piece of memory, it becomes available for another process.

2.3.2 Process Scheduler

The primary function of an operating system is scheduling. An operating system has multiple schedulers each with a unique purpose. However, the most commonly referred to scheduler (referred to hereafter as the process scheduler) is responsible for allocating processing time for the different processes. A process, in its essence, is simply a task that is being executed. Furthermore, the execution of a process must progress in a sequential fashion [2]. The operating system keeps track of a process through a process control block. The process control block holds all the information pertaining to the process such as process state, memory information, program counter, and all other relevant information.

The typical embedded computing system can only run one process at a time. So the different processes have to take turns using the processor. When processes are switched, the state of the current process is stored to its process block, and the saved state of the next process is loaded. This is known as context switching. Depending on the system, a considerable amount of processing time can be spent on context switching. Sometimes, a process is divided up into multiple smaller light-weight processes known as threads. Threads take turns using the processor just like processes. However, multiple threads are part of the same process, so they can be switched out with less overhead since they share much of the same information. All of the switching between processes and threads is controlled by the process scheduler.

There are a multitude of scheduling algorithms to accomplish context switching, some of which will be discussed in greater detail in Section 2.4. All of them have the basic idea of minimizing wasted clock cycles while still remaining fair to all of the processes and threads. So, when a process or thread is idle, it is switched out for another in the scheduler’s ready queue. Furthermore, when a process is waiting for I/O, it is rotated out of the scheduler’s ready queue into a waiting queue.

One advantage of operating systems is that a blocking code does not actually block. This accomplished by rotating idle processes out of the ready queue into a waiting queue [2]. Thus, designers are able to write simpler codes without the concern that one process will block another on the system. This is another reason why the wait-poll I/O access technique described in Section 2.2.1 is a common practice in embedded applications without operating systems.

2.3.3 I/O Subsystem

The I/O subsystem is the part of the kernel that is responsible for managing the hardware. The purpose of the I/O subsystem is instrumental in achieving the goals of the operating system as a whole. First, the I/O subsystem provides a standardized interface for the applications. The standardized interface significantly increases the portability of applications between hardware systems as well as the extendibility of the hardware interfaces. Consequently, additional hardware can be easily integrated into the system by creating device drivers that follow a standard interface. “The purpose of the device driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel” [2]. The second goal of the I/O subsystem is to optimize access to the hardware. What is meant by "optimizing" is to reduce that amount of resources spent on accessing the I/O hardware, which is accomplished through scheduling and buffering. The exact approach and method for scheduling and buffering are different for each operating system. However, the main concepts remain the same.

With multiple processes running on a system, there are occasions where two or more processes will need to access a particular piece of hardware at the same time. Since both systems cannot have access to the hardware at the same time, the I/O scheduler is responsible for scheduling their access to the hardware. I/O system calls that are not immediately processed are placed on a queue until the I/O is free. The general concept for the I/O scheduler is to organize the system calls in a way that reduces the overall wait time while still treating each process fairly [2].

Buffering is another way in which the effectiveness of interfacing with I/O hardware is improved. A couple of the problems that buffering helps resolve in embedded systems are the different data transfer rates and sizes [2]. The different rates can occur between an application running on the main processor and an I/O port or between two different I/O ports. With the different speeds, the faster device will have to wait for the slower device. By storing the data temporarily in a buffer, the faster device is free to perform other tasks and then transmit the data in bursts. Buffering is also used to adapt devices that have different data-transfer sizes. Data is often transmitted in segments called packets. When communicating across a medium that only supports smaller packet sizes, the larger packets have to be broken down and then recombined on the other end. Buffers help facilitate the breaking down and recombination of the packet by providing a place to store the smaller packets until all the components of the larger packet have been transmitted.

2.4 Scheduler

As mentioned in Section 2.2.2, there are a variety of scheduling approaches. This section will discuss how each of these approaches manages access to the processor. A process scheduler is necessary for any system with more than one process operating at the same time. This is true for embedded systems with or without an operating system. Because process schedulers play such a significant role in multitasking systems, it is possible to find published research covering a wide variety of scheduling methods. Some research that is of particular interest to this thesis is found in [1]. This research covers scheduling methods that can be directly applied to embedded devices without an operating system. The three scheduling methods discussed are Superloop, time triggered, and cooperative [1].

2.4.1 Superloop

The Superloop is one of the most simple and commonly used schedulers in embedded systems. In fact, this is the same scheduling method described in Section 2.2.2. A Superloop, like its name suggests, is simply an infinite loop through all of the tasks of the system in the order specified by the designer. Scalability is easy since a task is simply added to the while loop in the necessary order. The primary drawback is responsiveness and reliability. Embedded systems often have time-critical components. The Superloop does not have the ability of to accurately schedule a period for each task.

2.4.2 Time Triggered

The Time Triggered scheduler uses a timer interrupt to determine when each task is called. Since a timer is used for each task, this is not a very scalable scheduler system. As a result, Time Triggered schedulers are not commonly used in embedded systems.

2.4.3 Cooperative

A Cooperative scheduler is essentially a combination of the two previously discussed schedulers. One timer is set to interrupt at a regular interval, which will be the minimum time resolution for the different tasks. Each task is then assigned a period that is a multiple of the minimum resolution of the interrupt interval. A function is then constantly called to update the interrupt count for each task and run tasks that have reached their interrupt period. This results in a scheduler that has the scalability of the Superloop with the timing reliability of the Time Triggered scheduler. This is a commonly used scheduler for sensor systems. However, this type of scheduler is not without its limitations. It is still important that the task calls in a cooperative scheduler are short. If one task blocks longer than one timer interrupt period, a time-critical task might be missed.

2.5 Systems without Operating System

While operating systems' overhead make them impractical for a significant portion of embedded systems, there are concepts that can still be applied to embedded systems without a full operating system. A couple of concepts that are particularly useful are buffering and storing process state information. Although the implementation is different (and can be difficult) without an operating system, buffering I/O data is still just as relevant. An embedded system still has to deal with different transfer rates and data packet sizes with or without an operating system. A key aspect of an operating system that prevents a process from blocking is its ability to store the process state into a process control block. Likewise, any non-blocking system has to have the ability to store the state of a process so another process can have its turn on the processor. Without an operating system, the responsibility of saving its state falls to each individual process.

By applying these concepts to some of the existing coding practices and scheduling techniques, an embedded system can still make effective use of the hardware without an operating system. However, the implementation of these concepts without the use of an operating system can be quite a challenge. It is the goal of this thesis to demonstrate a generic and systematic way to accomplish this task and thereby reduce the development time required to design a system.

CHAPTER 3: HARDWARE PLATFORM

3.1 Sensor Modules

The platform used to implement the non-blocking coding techniques discussed in this thesis is the Fusion board shown in Figure 3.1. The Fusion board was developed by the Hartman Systems Integration Laboratory at Boise State University to replace the old PMON board [3, 4]. Each Fusion board can be used as a single sensor collecting device, or it can be combined with multiple Fusion sensor nodes to form a sensor network. The Fusion board has been designed to increase performance while maintaining the flexibility of the older PMON modules [3]. The fusion module takes advantage of the improved hardware capabilities developed over the past few years.

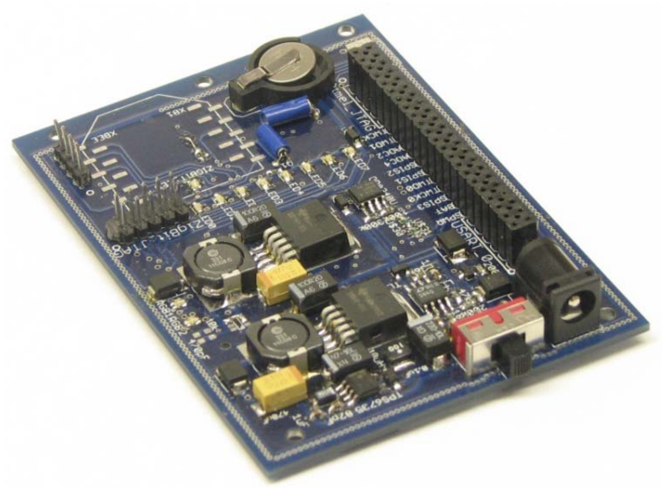

Figure 3.1: Sensor Module Board

3.2 Processor

Many sensor networks used in embedded applications in particular require a flexible microcontroller with plenty of memory. The ATMEL microcontroller AT32UC3A3256S, a 32-bit microcontroller, was chosen for the sensor modules because of its ability to meet these requirements [5]. With 256 kbytes flash and 64 kbytes high-speed SRAM, this microcontroller has plenty of memory to work with. Additionally, the 96 DMIPS and 66MHz processing unit found in this microcontroller is significantly faster than the PMON's 8-bit microcontroller PIC18F8722 [6].

This microcontroller also has a wide variety of hardware communication modules that make it ideal for a flexible system. It has one multimedia secure digital card communication port for storing data to removable media. Also, a high-speed universal serial bus (USB) module is available for data transmission with a computer. An eight channel 10-bit analog to digital converter is available for interfacing with analog sensors. Additionally, there are two SPI modules with the capability of being set as a master or a slave. Each SPI module has 4 chip select signals allowing for communication with 8 devices across the two ports. There are also two TWI modules that are capable of communicating using the I^2C protocol. Finally, there are also four USART modules that are available on the microcontroller.

3.3 Breakout Board Sensor Interface

The greatest asset of the PMON modules is their flexible sensor interface [3]. In order to maintain this flexibility, the Fusion sensor modules are designed to have a sensor breakout board attached to the main board through a header. This allows different communication protocols to be used for different applications without redesigning the entire system. Each application has a unique breakout board that is designed to communicate to the microcontroller through the header. A variety of communication protocols are routed to the header to maximize options for different sensor applications.

One of the most common sensor communication protocols is I^2C. The lines from both of the I^2C capable modules are routed to the header. By having accesses to both I^2C capable modules, two sensors with the same address can be used.

Another common communication protocol used for sensors is UART. For the Fusion modules, UART is the primary method for transmitting data directly to a monitoring computer. Unfortunately, only one device can interface with each UART module at a time. Consequently, it does not take very many sensors to use up the available UART modules. Therefore, the lines from three of the microcontroller’s four available UART modules are routed to the breakout board header. However, UART0 is primarily used for sending data to the computer (as a terminal or debugging port). This leaves only two UART lines dedicated to sensor use.

SPI communication protocol is not as common for sensors as UART and I^2C, but there are occasions where it is the only option. There are also occasions when the higher speed of SPI is very useful. One of the SPI modules is dedicated to communicating with the ZigBit module as described in Section 3.5. So the lines from only one of the SPI modules are routed to the breakout header.

Sensors are not always set up to communicate digitally. In fact, a fair portion of sensors have analog output signals. Some analog sensors are self-contained with a simple voltage output that can be read directly by an ADC. Other analog sensors require additional circuitry to produce a meaningful voltage output. In either case, an ADC is required in order for the signal to be processed and recorded. Therefore, seven of the eight channels of the ADC module on the microcontroller are routed to the breakout board header. Three of the seven share the I/O lines with the SPI module. So, there are only four dedicated ADC channels on the header. If more ADC channels are needed, external ADC's with I^2C communication protocols can be added to the breakout board.

3.4 Power

There are a variety of voltages required to operate different sensors. The digital sensors generally require a set voltage. This voltage is usually 3.3V, but occasionally it is 5V. Analog sensors usually have a range of voltages at which they can operate. So, one can often use 3.3V or 5V depending on what is available. But some analog sensors operate with greater accuracy at 5V than 3.3V. Also, it is common for analog sensors using a 5V source to require a -5V as well. There are sensors that require other voltages. But 3.3V, 5V, and -5V are the most common for sensors that are suited for battery operated systems where low power is necessary.   Therefore, the Fusion sensor modules are designed with power regulation for these common voltage levels. The positive 3.3V and 5V are obtained by using simple switcher regulators. An inverter is used to get the -5V from the +5V.

3.5 Network Communications

One of the critical components of a sensor network is the communication between sensor modules. The wireless communication between sensor modules is handled by Atmel’s ZigBit wireless module. The ZigBit module operates at 2.4 GHz with a dual chip antenna. The ZigBit is compatible with the IEE 802.15.4 ZigBee wireless network protocol stack [7]. The ZigBee protocol supports a mesh network that supports self-healing. The ZigBit hardware has several different communication protocols available to interface with the microcontroller. The options available are USART, I^2C, and SPI. The SPI module was utilized for the purpose of speed. SPI on the ZigBit can transmit as fast as 500 kbps, whereas I^2C can only transmit at 250 kbps, and UART can only transmit at 115.2 kbps. The SPI is transmitting slower than normal because it is actually synthesizing SPI communication over the USART hardware. Also, the synthesized SPI can only operate as a Master device. Therefore, the microcontroller operates as the SPI slave device. Even with these drawbacks, the SPI is the optimal choice because it is faster and does not have to share I^2C lines with other devices.

3.6 Data Storage

It is often necessary to store data either temporarily or long term on the sensor modules. This usually requires more memory than is available on the microcontroller. A Secure Digital (SD) memory card is used for this purpose because of its flexibility. The ability to load files directly from a computer makes the SD card ideally suited to store the calibration and various sensor settings. The SD memory size can be chosen based on the needs of the application, although it is difficult to find a sensor application that requires more memory than the smallest SD card on the market today. The SD memory card slot interfaces with the microcontroller using four bit SD protocol. The microcontroller is able to support SD V2.0 and SDIO V1.1 specifications.

CHAPTER 4: NON-BLOCKING CODING PRACTICES

Non-blocking code, like its name suggests, is code that does not block other processes from accessing the processor. In an embedded system, there are multiple tasks that need to be processed at the same time. However, the typical embedded processor can execute only one task at a time. Anytime one task is being processed, the other tasks are unable to run. In other words, the other tasks are being blocked. So technically, any time any task is being executed, it is blocking other tasks from being executed. By this definition, any system that has more than one task is technically non-blocking. Clearly, non-blocking code needs to be more clearly defined. A more precise definition of the practical differences between blocking and non-blocking code will be provided in the following sections. This chapter will also discuss several coding techniques that are used in developing non-blocking code.

4.1 Blocking Code

For the purpose of this thesis, code will be considered blocking when one of the two following conditions is met. It will be considered blocking code whenever the process being executed is waiting for some external condition. For example, a loop that is simply waiting for an I/O response before executing the rest of the process (e.g., the wait-pole approach discussed in Section 2.2). This is not the same as when an array of characters is processed by a processing loop. In other words, blocking code occurs when processor cycles are spent on something other than useful computation. This results in inefficient use of processing resources. The second condition is when one task keeps a time-critical task from being processed. For example, one task performs a processor intensive calculation on some data that could be done at a later time. This results in another task that handles incoming data to miss an incoming packet. This type of blocking code is difficult to detect. The effect of this blocking condition can be fatal to the system when critical information is either corrupted or lost entirely. These two blocking conditions are not mutually exclusive. The first blocking condition can often cause the second condition to occur.

Interfacing with external hardware is not the only situation where blocking code can become an issue. Avoiding blocking code is also a concern when communicating from one application to another. Communication between applications can be divided into two main categories: synchronous and asynchronous. Synchronous communication has the advantage that the data transmission is guaranteed. However, synchronous communication is essentially blocking code since communication only occurs when both applications are ready. For example, suppose that a sensor application requires data from another sensor in order to complete its calculations. The first sensor application will have to wait for the results from the second sensor application before it can finish its own calculations. Consequently, the data throughput can be significantly limited. Asynchronous communication between applications has the advantage of higher throughput since data is sent without regard to the receiving application’s status. However, this requires data buffering, which runs the risk of lost data when the receiving application does not retrieve the data in a timely manner. Asynchronous communication is typically the better option due to the high throughput, especially when all of the applications are updated frequently enough to minimize the buffering issues.

While blocking code has a detrimental effect on processing efficiency and can, under certain conditions, cause serious failures. There are some benefits to using blocking code. First and foremost, it is easier to write. The very logical flow of code testifies to this. Consider the basic function call in C shown in Figure 4.1. A function is given some input parameters. The function then performs an operation and returns a value. Function calls are sequential in nature. One function finishes its operation before the next function is started. Blocking code is sequential as well. One task must finish before the next one gets to start. The sequential nature of blocking code makes it easy to follow and develop. There are occasions where a task is critical enough to make blocking a necessity.

```
<Return Type> <Function Name> (<Input Parameters>)
```

Figure 4.1: Basic Function Call Format

4.2 Non-Blocking Code

Non-blocking code is essentially code that does not block in the manner described in Section 4.1. However, according to Silberschatz and Galvin, this definition of non-blocking code can be further divided into categories. The first is when a function call returns immediately with whatever results are available. The second type is referred to as asynchronous. For this type, the function call returns immediately, but a callback function is registered for when the data is ready [2]. For the purpose of this thesis, non-blocking code will be defined as the absence of blocking code defined in the previous section. This definition encompasses both types described by Silberschatz and Galvin.

As with anything else, there are tradeoffs for implementing non-blocking code. The primary tradeoff with non-blocking code is its complexity. While it offers a more effective use of processing resources, it does add a significant amount of complexity. In order to better understand this complexity, several techniques used to achieve non-blocking code need to be considered. The first is buffing data. For a task to be non-blocking, it cannot pause while waiting for data to arrive. Therefore, it is necessary to safely store data until the task is given an opportunity to process the data. The type of buffer that is considered in this thesis is the circular buffer (see Section 4.2.1). The second technique used in the implementation of non-blocking code is a state machine. It is often necessary to leave a task or function before it is completed to keep from blocking another task. So, it is important to keep track of what point in the process the task left. A state machine accomplishes this by having multiple states that represent different points in the process. The effects and operation of a state machine will be discussed in further detail in Section 4.2.2. The final technique is the callback function. It is sometimes necessary to let a task know when an action it requested is completed. A callback function is an effective way to let the task know without having the task continue to poll the hardware driver. In Section 4.2.3, the details of callback functions will be discussed in further detail.

4.2.1 Circular Buffer

Most of the buffering needs in embedded systems usually involve transferring data. When transferring data, it is typically a requirement that the data maintain its order. The best way to keep it in order is to use a first in first out (FIFO) buffer. A circular buffer is a FIFO buffer that is well suited for embedded systems. In embedded systems, memory and processing speed remain two of the most significant limiting factors. Buffers tend to contribute significantly to memory usage. In larger computing systems, the memory for buffers is allocated dynamically by a memory manager. However, memory managers tend to add significant overhead that makes them impractical for small to medium scale embedded systems. Consequently, buffer memory on embedded systems is usually permanently allocated. So, it is important for buffers to optimize their use of the available memory. Circular buffers are one of the simplest buffering methods used to optimize memory. The basic circular buffer shown in Figure 4.1 is composed of a couple of pointers and an array of some data type. The two pointers reference the head and tail of the data. The head points to the location in the array where the next incoming data is placed. So, as data is received, it is placed at this location and the head is moved to the next location in the array. The tail points to the location in the array containing the next data that will be removed from the buffer. After the data is removed, the tail will be moved to the next location in the array. When either the head or tail reach the end of the array, they wrap around to the beginning of the memory array. Consequently, this wrap around results in a buffer that is circular in nature shown in Figure 4.1. Since the buffer is circular, it continually reuses the memory available without having to be reset. Each read and write only involves one memory access, the increment of a pointer, and a few safety checks. So, the processing overhead is relatively low.

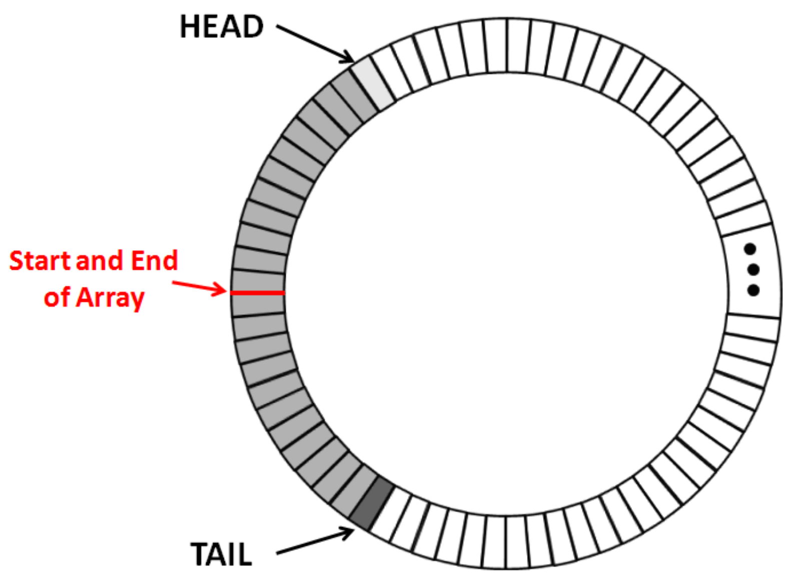

Figure 4.1: Circular Buffer

While circular buffers are one of simplest techniques used in non-blocking coding practices, it does have a few potential complications. First, there is the risk of a race condition occurring. Consider an example where the buffer is empty so that the head and tail point to the same location on the array. The head is incremented first and the process is interrupted before the data is loaded. Thus, the pointers indicate that the buffer has data when it is actually still empty. If the interrupting routine reads from the tail location, it will get invalid data. This situation is easily prevented by having the head increment only after the data has been placed in the buffer. Another type of race condition has the potential to occur when the head or tail is accessed from more than one location. For example, consider the situation where data is read from the tail, and the process is interrupted before the tail is incremented. If the interrupting routine attempts to read the memory from the tail, this location will be read twice, and the next location will be skipped. This problem cannot be fixed by reordering the memory access and pointer progression. The only solution is to ensure that the head and tail each be accessed from a single location.

Another complication is that, since the size of the buffer is constant, circular buffers have the potential to overflow. One approach to solve this problem is to make the buffer ridiculously big. This ensures that the buffer never has the chance of overflowing. While this method may be necessary for critical applications, it would negate the point of using a circular buffer (to make optimal use of memory). In either case, there is always a chance that some condition will occur that fills the buffer. Therefore, it is important to minimize the consequences of buffer overflow. The solution could be different depending on the application. One application might require a flag to be set to stop incoming data. Another application might be able to simply throw incoming data away when the buffer is full. In either case, it is necessary to accurately determine when the buffer is full. This would seem simple except for the fact that the head and tail will point to the same location on the array when the buffer is full or empty. The key is to not let the buffer completely fill up. However, due to fact that the buffer wraps around, it is difficult to determine the amount left in the buffer using math. One of the simplest solutions is to simply have a variable keep track of how much data is in the buffer. The variable needs to be updated after the data has been place in the buffer for the same reasons as those discussed for incrementing the head pointer in the same fashion.

While the circular buffers are conceptually simple and efficient, there are a few subtle problems that make them detail intensive to implement. However, circular buffers are possibly one of the most effective techniques widely used in implementing non-blocking code. Circular buffers can be made even more effective by eliminating some of the details of implementation. This can be accomplished by making a generic circular buffer that can be reused in multiple applications. This way, the detail intensive testing part of the implantation only has to be performed once. An example of this can be seen in Figure 4.3. The circular buffer is composed of a structure that contains information about the buffer and a memory array that stores the data. The structure stores a pointer to the start of the data array. When the tail and head reach the end, it is important to know where to start again. Pointers to the head and tail are also located in the structure. It does not matter if the head and tail are literally pointers or just offsets from the start of the array, as long as they accurately reference the locations of the head and tail. The circular buffer for this research actually used offsets to simplify the math. The used space and size variables indicate the amount of data in the buffer and the size of the buffer respectively. By implementing a generic circular buffer, the main problem left is to prevent accessing the head or tail from more than one spot. For most embedded applications, this is simple. If the circular buffer is used for a hardware driver, then one end is accessed by the hardware. This is commonly accomplished in an interrupt routine. The other end of the circular buffer, in this case, is accessed by the application responsible for using the given piece of hardware. Solving this problem can get more difficult when circular buffers are used in high level applications. This is particularly true in embedded systems that are flexible and have a wide variety of applications, as is the case for sensor networks. A problem occurs when multiple applications need the data from one piece of hardware. In these situations, it may be necessary to use a data manager.

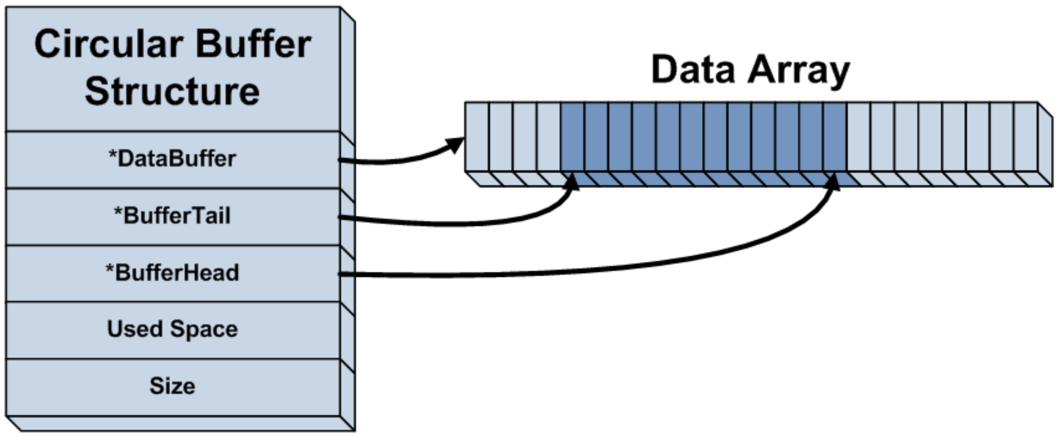

Figure 4.3: Generic Circular Buffer

<u>4.2.2 State Machine</u>

The finite state machine is essential to non-blocking coding practices. Embedded systems are generally used in applications that require microcontrollers to interface with a considerable amount of hardware. Hardware interfaces usually operate at a different speed to that of the main processor. Consequently, the processor is required to wait when communicating with hardware interfaces. The waiting can be implemented with blocking code or by using a state machine to keep track of the hardware's progress. It is also not uncommon for embedded systems to have more than one application that is constantly running. In order for these applications to take turns using the processor, it is necessary to keep track of the state of each process. Although it is possible under certain conditions to have a circular buffer operate in an efficient manor without the help of a state machine, it would be highly unlikely to have an entire embedded system running off of non-blocking code without the use of a state machine.

There are two types of state machines: synchronous and asynchronous. The essential difference between these is that a synchronous state machine will only change its state in sync with a particular event. In fact, another name for a synchronous state machine is an event-driven state machine. In hardware, all the changes in a synchronous state machine occur at a clock edge [8]. In other words, the clock functions as the event that drives the state machine. The nature of software is such that the changes to the state machine will occur in some type of function call. So the function calls work as the event that drives software state machines. For the purpose of this thesis, state machines will be referring to the event-driven type. Event-driven state machines can be further categorized by the type of event that drives them. State machines used for hardware drivers are usually interrupt driven. Most hardware interfaces have interrupts associated with them. The interrupts trigger when certain aspects of the hardware interface, defined by the user, change. These changes are usually designed to coincide with specific states in the driver's state machine. Therefore, it usually makes sense to have the state machine be driven by the interrupt of the associated hardware. On the other hand, application level state machines are typically driven by a scheduler.

Typically, in embedded systems, the state machines default to an idle state. Upon receiving a start request from an application or hardware, the driver transitions to the starting state of the desired action. On simple state machines, there may only be one possible action sequence that continues to propagate through a set order of states like the one shown in Figure 4.4. Each state is used for keeping track of the time or number of events that have occurred. Unfortunately, an embedded system is rarely that simple. Typically, the state machine will have multiple action sequences with their different starting points. Each action sequence will usually have more factors determining the next state other than the event driving the state machine. Sometimes, an error will occur in hardware, and the hardware driver state machine needs to be able to account for the error condition. It is evident that state machines can be considerably complex. The considerable differences between each application prevent the easy reuse of a state machine template. The complexity of state machines is probably one of the most significant drawbacks to implementing non-blocking code.

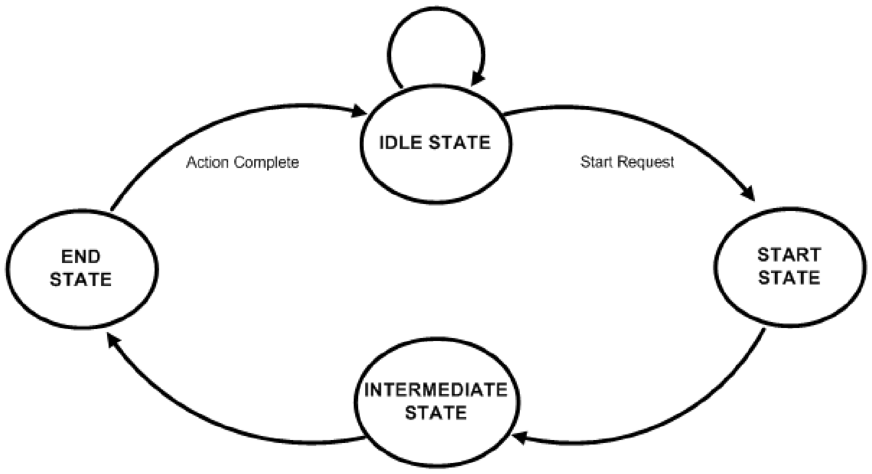

Figure 4.4: State Machine

The complexity is not the only issue facing the use of state machines. Synchronous state machines are critically dependent on the timing of the driving event. In hardware driven state machine, the driving event is a clock edge, which is typically reliable. Whereas in software, the driving event is either the task function being called or an interrupt occurring. So, an important state transition will be missed if a condition for transitioning the state machine occurs momentarily, and the driving event does not occur during this time. For example, consider a situation where incoming data from a sensor for an application is received into the applications buffer, but the applications task responsible to process the data is not called before the data is overwritten. This will result in the overwritten data never getting processed. It is not always necessary for driving events to occur at a precise time interval, but it is necessary for them to occur often enough to not miss any changes of conditions. The timing for applications and other task driven state machines is usually handled by a scheduler. With a scheduler, it is important for each task to follow the good citizen approach and release as soon as possible. Otherwise, another task might miss a critical event.

Another problem is that each state machine requires permanent memory allocation to keep track of the state and related variables. Good coding practices dictate that one should avoid using global variables as much as possible for two reasons. First a global variable permanently consumes a piece of memory which is a valuable resource. Secondly, global variables decrease the readability of the code [9].

4.2.3 Callback Function

While callback functions are not essential to non-blocking coding practices, they provide several beneficial services. Callback functions facilitate more efficient use of memory and processing resources. One benefit to callback functions is that wasted processing cycles devoted to calls used to monitor the condition of another application are reduced. Furthermore, memory used for flags that indicate when action is completed is freed up as a result of the use of callback functions.

Callback functions are simply functions that are used to call the user back when a particular action is complete. Consider a typical example represented in Figure 4.5. In this case, an application task makes a function call to the hardware driver to perform an action. What the application and action requested is not important. What is important is that the action involves waiting for the hardware to respond. So, the hardware driver starts the action in motion on the hardware, and then immediately returns control back to the application task. The application then completes whatever else it needs to and returns control to the scheduler. This is often simply involves the application saving its state. The scheduler continues to provide the driver task its share of processing time to complete the action. After the action is complete, the driver task will call the application callback function to indicate that it is complete. The callback function can update the state machine or a flag to let the application know that the action is complete. However, even more importantly, if the application has any time-critical actions based on the ending of the driver's action, they can be in the callback function instead of waiting for the applications task to be executed again.

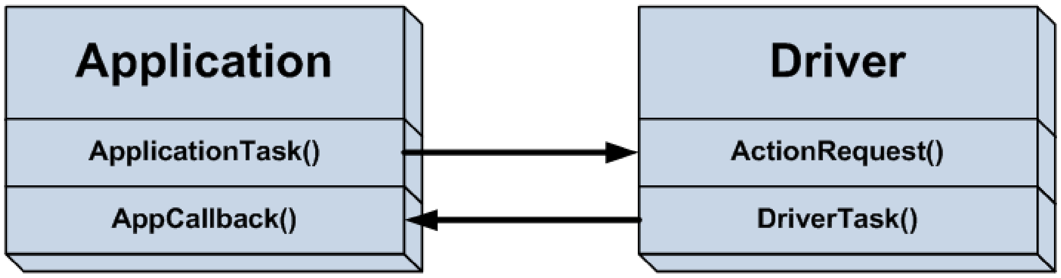

Figure 4.5: Callback Function

The same thing can be accomplished with the use of global flags or a polling function called from the application. With non-blocking code, there is already more than enough permanent variables adding to the clutter and consuming memory. So, eliminating the need for global flags can be a very useful accomplishment in non-blocking code. Having a polling function inside an application task means that it will be called every time. This might not seem like much of a cost in processing time, but, if the application is not doing anything other than checking to see if the driver is finished, it is doubling the processing cost of the application task at this time. Additional cluttering of the software aside, this processing time can add up quickly. Since implementation complexity is one of the largest limitations of non-blocking code, anything that can simplify the code is a good thing.

4.3 Example Code

It is sometimes easier to see the difference of blocking and non-blocking coding techniques by comparing an example of each style. The example code shown for both cases is responsible for transmitting data using specialized hardware on the microcontroller. The general method would be same for any of the common communication protocols (UART, SPI, I^2C, etc). These examples demonstrate how blocking techniques are simpler to implement, and how the blocking techniques can be optimized.

4.3.1 Blocking Code Example

A blocking example of transmitting data using specialized hardware designed to implement a generic digital communication protocol is shown in Figure 4.6. With blocking code, all of the process occurs in the function call itself. This means that each function has to wait for each byte to finish before loading the next one. The waiting is typically done by polling the hardware in a while loop as shown in Figure 4.6. This method has the advantage that the requested process is complete when the function returns. The completed process, upon returning control, makes it easier to write sequential executing code for the calling application.

```
void blocking_transmit(char *data, int length)
{
        unsigned int i;

        for (i = 0; i < size; i++)
        {
                Example_Write_Register = data++;   // put character in hardware
                                                   // write register.

                while(example_Hardware_is_Busy); // poles the hardware waiting
                                                 // for the transmission of the
                                                 // previous character to finish.
        }
        return;
}
```

Figure 4.6: Blocking Transmit Function

4.3.2 Non-Blocking Example

The non-blocking transmit is composed of multiple components. This is the main reason why non-blocking code is so complicated. The first component of the transmit request is shown in Figure 4.7. The transmit request loads the relevant information into buffers. The relevant information is composed of the data that needs to be transmitted and a callback function that is used to indicate when the data transmission is complete. The transmit request immediately returns after loading the buffers and makes sure that the related interrupt is turned on. By returning as quickly as possible, it prevents other tasks from being unnecessarily blocked.

```
void nonblocking_transmit_request(char *data, int length, void *CallbackFunction)
{
        unsigned int i;
        Circular_Buffer_Add(Callback_Buffer,CallbackFunction);

        for (i = 0; i < length; i++)
        {
                Circular_Buffer_Add(Data_Buffer,data++);
        }
        Start_hardware_interupts(); // sometime interrupts need to be restarted
        Return;
}
```

Figure 4.7: Non-Blocking Transmit Request

The second and most important component of a non-blocking data transmission is the interrupt service routine shown in Figure 4.8. The interrupt service routine is responsible for loading the next byte of data into the hardware register after previous byte is finished transmitting. In order to get the highest performance out of the hardware, the bytes need to be loaded into the register as soon as the previous one is finished sending. The interrupt is designed to interrupt the other tasks to execute the service routine. So, the loading of the register occurs as quickly as possible. If this were handled in the task, it could occasionally take some time before each byte is loaded. Some data transmissions are time sensitive, especially when receiving data. Since service routine interrupts other process, it is important that they are kept as short as possible. So, it is not a good idea to call functions from inside an ISR.

```
__attribute__((__interrupt__))
void interrupt_service_routine(void)
{
        if  (write_register_empty)
        {
                Write_register = Circular_Buffer_Get(Data_Buffer);
        }
}
```

Figure 4.8: Non-Blocking Transmit Interrupt Service Routine

The final component is the task shown in Figure 4.9. The task is used to execute processes that are not time critical and do not fit inside an ISR. The callback function is called from the driver's task function. The task function monitors the status of the packet being transferred as a whole and then calls the callback function associated with that packet after it has completely transmitted. The callback function is typically not time critical but can be computationally intensive. Consequently, it is better to have the task handle the callback functions.

```
void Nonblocking_transmit_task(void)
{
        if  (packet_is_completely_sent)
        {
                CallbackFunction = Circular_Buffer_Get(Callback_Buffer);
                (*CallbackFunction)();
        }
        return;
}
```

Figure 4.9: Non-Blocking Transmit Task

The fact that non-blocking is composed of multiple parts inherently makes it more complicated. However, by establishing generic structures (i.e., circular buffers) that can be used over again in a similar manner, the development time can be reduced. For example, implementing the non-blocking I^2C driver using these techniques took far less time than implementing the non-blocking ZigBit Driver did without these techniques. It is fair to say that developing the ZigBit driver had plenty of problems not related to these techniques. However, problems with a non-generic circular buffer increased the time it took to solve these problems. Having established non-blocking techniques decreased development time by reducing repetition and conflict between developers.

4.4 Appropriate Uses of Blocking Code

While this paper is focused on the benefits and implementation of non-blocking coding practices, it is necessary to recognize times where blocking code is a better option. This section will attempt to better define the times when using blocking code is necessary.

The most common reason for implementing non-blocking code is development time. There are, in fact, times where it is necessary to sacrifice performance for the sake of time. For example, during the development stage, in order to test hardware or other aspects of the system, it may be necessary to temporarily implement some blocking code. Also, there are situations were speed of operation and full functionality are not as important as getting the system running as soon as possible. However, this reasoning is probably used too often.

Another reason for using blocking code could be dictated by the simplicity of a system. Blocking code might not have detrimental effects on extremely simple systems where there is only one application. When there are not multiple applications competing for processing time, then the one application will not be able to block other applications.

Initialization routines are another area where it is acceptable to have blocking calls. It is usually not critical how long the initialization sequence takes to execute. What is important is that all of the initialization sequences are executed in order. In microcontrollers, the initialization of one component often requires the initialization of another component first. Consequently, it is usually necessary to have each initialization call block until it is complete.

Finally, if an embedded system is complex and large enough, the advantages of the OS system outweigh the drawbacks of the OS overhead. An operating system behaves as non-blocking while the coding style is essentially the same as blocking. This is only an option if the hardware is able to support the OS.

CHAPTER 5: NON-BLOCKING ANALISIS

Due the complex nature of embedded systems, it is difficult to measure and show the practical effectiveness of non-blocking coding practices. It is more meaningful to examine and analyze the individual parts of the system separately. A hardware driver is one part of the overall system where the differences are easier to identify and understand. Sections 5.1 and 5.2 will examine two hardware drivers to show the effects of non-blocking code techniques. The first driver is an I^2C hardware driver that is used to interface with some sensors. The second is a UART hardware driver responsible for streaming data to a computer. The Fusion platform described in Chapter 3 is used to implement the non-blocking code for both examples. Therefore, it would be useful at this time to provide brief descriptions of the software framework for the hardware drivers. There are a couple of key aspects to the Fusion sensor node framework that are pertinent to the driver analysis. These aspects of the framework are scheduling techniques, network communication layers, and the software framework. The scheduling software is responsible for calling the applications that use the hardware drivers. This system actually uses two of the scheduling techniques described in Section 2.4. The code as a whole is scheduled by a Superloop. That is, the tasks for all the applications and drivers are called in order from a while loop. This allows each state to continually update their present status. This is one of the more common scheduling methods used in embedded systems for blocking and non-blocking techniques. However, in either case, it places the burden of each task to complete its task in the minimum time possible (be a “good citizen”). In addition to the top level Superloop, a cooperative scheduler is used to control time sensitive actions like sensor reading. This scheduling technique will be used for both examples.

In addition to determining the effects of non-blocking code, it will also be useful to examine the effects of blocking code on a system. Thus, the network communication will be used to examine the effects that blocking code has on other processes. Many embedded systems currently use some type of wireless communication. The Fusion sensor nodes are setup to communicate using Atmel’s ZigBit radios as mentioned in Section 3.5. The ZigBit radios are responsible for handling the ZigBee wireless mesh networking. A basic diagram of the code layers for the Fusion network can be seen in Figure 5.1. The microcontroller is setup to communicate with the Zigbit Driver itself and utilizes two circular buffers. The Network layer on top of this is responsible for breaking up large data packets into small enough packets to be sent across the network. The Fusion Server layer sets up ports between different Fusion modules. The data manager layer is not actually part of the network. The data manager is responsible for controlling were the data from the applications is sent. Consequently, it interfaces the network with the application layer. For the fusion sensor nodes, the application layer consist of the sensor drivers and any other task that collects data. The Network layers are all implemented using non-blocking techniques. The network communication will be the process used to examine the effects that blocking I^2C and UART drivers have on other processes.

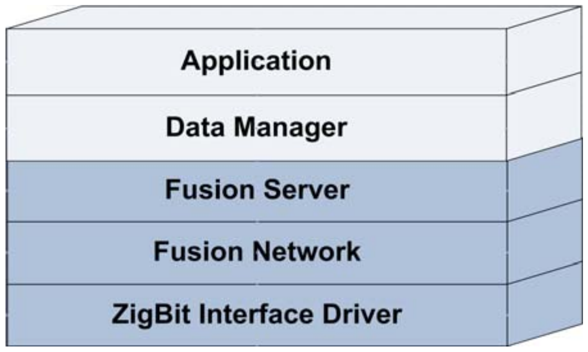

Figure 5.1: Fusion Network Software Interface

The entire Fusion framework, which includes the network and schedulers, is built on top of the software development framework that was provided with the Atmel microcontroller. This development framework includes drivers and service routine for interfacing with the hardware [10]. Unfortunately, there are large parts of the supplied framework that are implemented using blocking techniques. In these situations, it is often necessary to replace the existing blocking framework using non-blocking techniques. For example, the non-blocking UART driver bypassed the supplied UART driver that was supplied with the microcontroller. On the other hand, the blocking example of UART driver was built on top of the supplied UART driver.

5.1 I^{2}C Hardware Driver Analysis

The I^{2}C communication on the Fusion sensor nodes is primarily used to communicate with sensors. For this example, it will be communicating with three different sensors: a magnetometer, an accelerometer, and a gyroscope. These sensors were chosen due to the fact that they are commonly used in applications requiring high sample rates. These sensors are designed to sample at tens of milliseconds. Some I^{2}C sensors, designed for slower applications, do not have the ability to sample at this rate. In this example, three measurements each are read from the magnetometer and accelerometer while four measurements are read from the gyroscope for a total of ten measurements. However, since it is possible to retrieve all of the measurements from one sensor with one I^2C reading, it will only require three I^2C communication packets to retrieve all ten measurements.

I^2C is a fairly complicated communication protocol. Each I^2C communication is composed of several parts. Every communication starts with a device address so that the devices (sensors in this case) can tell to which one the communication is directed. The last bit in the device address byte is used to indicate a read or write. In addition to sending a device address, some sensors have multiple registers and require a register address to indicate which one to access. In addition to the complicated formats of I^2C, start and stop bits are also required at the beginning and end of the communication along with acknowledgement signals after each byte. Fortunately, the Fusion microcontroller has specialized I^2C hardware that is capable of handling the signaling details. Hardware registers still need to be set for the I^2C hardware to know what to communicate. Consequently, each communication needs to have all the relevant information associated with it. Furthermore, multiple I^2C devices will share the same buffer and the information will be different for each device.

5.1.1 I^2C Blocking Driver Description

The blocking I^2C driver is fairly simple. It does not use a state machine or circular buffer. It waits for each stage (device address, the register address, and each data byte) to finish before moving on to the next stage. Only after the entire I^{2}C communication is finished and an acknowledgement is received from the device for the last stage does the function return control to the calling application.

5.1.2 I^{2}C Non-Blocking Driver Description

Since I^{2}C is a relatively complicated communication protocol, it requires all of the non-blocking techniques discussed in Chapter 4. It utilizes a couple of generic circular buffer, a state machine, and callback functions.

A circular buffer of pointers is used for both the read and the write communications in the I^{2}C driver. Pointers are used to minimize memory usage and complexity. The pointers in the buffer point to I^{2}C packet structures as shown in Figure 5.2. Each packet contains the relevant information for an I^{2}C communication. The packet structure contains pointers to both a device structure, data buffer, and a callback function. The device structure contains the information relevant to that device, such as its address and the module to which the device is connected. So, multiple communication packets can be created for one device and pointed to its device structure. The data buffer that the structure points to shown in Figure 5.2 is used for the purpose of storing the actual data. The device structure also contains communication details such as data length, register address, register address length, and a read/write indicator. The device driver application is responsible for instantiating and maintaining the device structures and buffers for each I^{2}C communication packet. So, memory is only set aside for the applications requiring

I^{2}C communication. This minimizes the amount memory that is required for the circular buffers. However, this implementation does have the drawback that the users of the I^{2}C module have the responsibility of managing the structures themselves.

Implementing a circular buffer has added a considerable amount of complexity. However, using a generic buffer has reduced the complexity via the use of pointer to structures. The generic circular buffer discussed here is the same format as the one used in the UART example. The pointer setup on the circular buffer has the additional benefit of reducing the overhead on read and write request calls. When the read or write function is called, the only input parameter needed is a pointer to the structure created in the device driver. The read/write function simply places the pointer on the circular buffer and then returns.

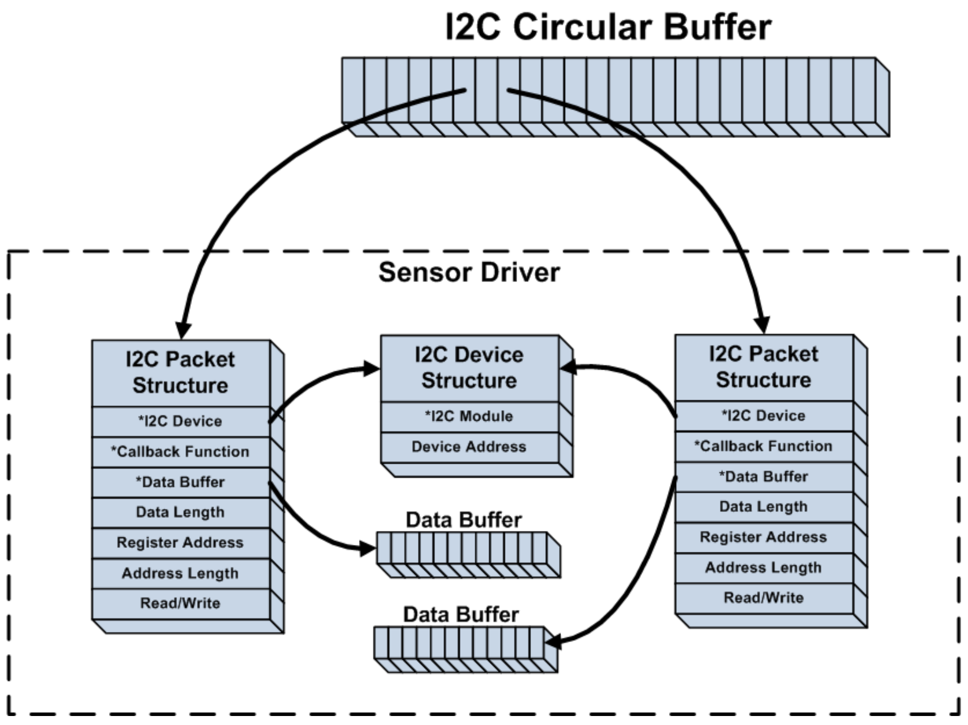

Figure 5.2: Fusion I^{2}C Driver

The callback function is used to further simplify the operation of letting each device driver know when its packet has been sent or received. With the variety of devices and sensor packets, it would take a complex system of flags to indicate to each device driver which packets had been processed. By having a pointer to the callback function in the I^{2}C packet structure, the device driver has complete control over the callback response. The callback function is typically used to simply set a flag for the device driver once a communication has finished.

The state machine on the I^{2}C goes through the same process as the blocking communication. The main difference is that, wherever the blocking driver would wait, the state machine has a state representing that position. This state is maintained without keeping control of the processor until the hardware receives the required information. The state machine is driven by the hardware interrupt. The hardware interrupt is set to trigger when changes relevant to the state machine occur. At each interrupt, a series of conditions are checked to make sure that there are not any errors, and that the system is ready to move to the next state.

5.1.3 I^{2}C Waveform Analysis

For this example, the same three sensors previously mentioned are read using both blocking and non-blocking I^{2}C function calls. Each of these sensors is read every 5ms. In order to visually examine the effects of blocking code, a couple of general purpose input output (GPIO) pins were used to indicate the start and end of function calls related to I^2C communication. These GPIO pins were viewed on a digital logic analyzer along with the associated I^2C signals. Snapshots of the blocking and non-blocking results from digital analyzer are shown in Figures 5.3 and 5.4, respectively. The signal used to represent the time spent executing the I^2C functions is D11. It is set high at the beginning and set low at the end of every I^2C function call. So, the amount of time spent in each function is indicated by the signal remaining high. In a similar fashion, the D10 signal represents the time spent executing the Sensor Task. The D10 signal is set high when entering the Sensor Task function and is set low when upon completion. The sensor task's application is responsible for the devices that use the I^2C communication. Signals D1 and D0 represent the I^2C data and clock lines, respectively.

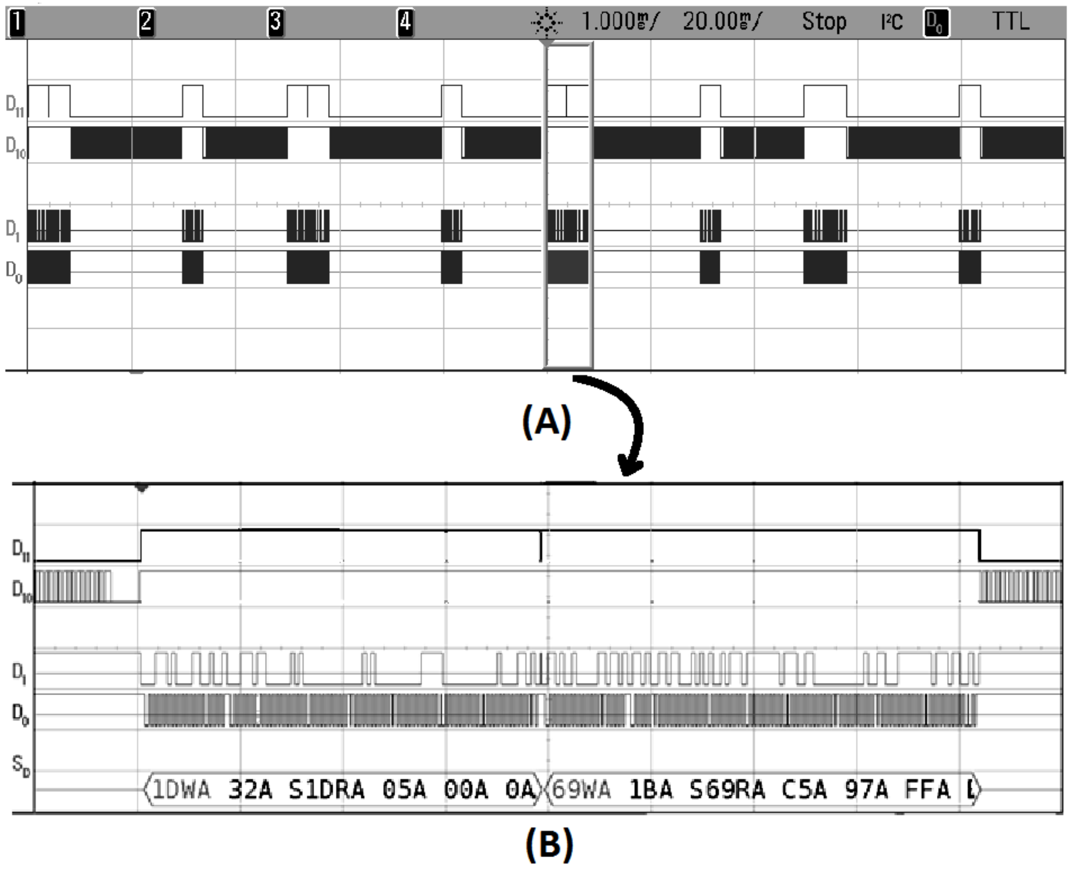

Figure 5.3: Blocking I^{2}C Waveform (a) 20ms Resolution (b) 1ms Resolution

With the Superloop scheduling method, the processor continually loops through all of the tasks. When none of the tasks block, the main scheduling loop will be executed in a very short time. This results in signal D10, which represents the sensor task execution time, toggling so quickly that it appears as solid section in Figure 5.3(a). There is a resolution of 20 ms per division for Figure 5.3(a). When I^{2}C communication occurs, signal D10 is held high indicating that the sensor task is blocking the main scheduling loop. The I^{2}C communication is indicated by the solid portions on the D1 and D0 signals in Figure 5.3(a). The reason the scheduling loop is blocked during I^{2}C communication is due to the fact that the I^{2}C read function indicated by D11 block the sensor task, which in turn blocks the scheduler. In this example, the accelerometer and gyroscope are typically read from inside the same sensor task call. This is why both D11 and D10 are held high during the blocking I^{2}C communication.

In the wider section of I^{2}C activity, the D11 signal has two sections indicating that there are two I^{2}C communications that take place in one sensor task call. These are the I^{2}C communications for reading the accelerometer and gyroscope sensors. The narrower I^{2}C activity sections indicate when the magnetometer is read. It can be seen from Figure 5.3(a) that all of three sensors are read once every 50ms in this example. Figure 5.3(b) displays a zoomed-in view of accelerometer and gyroscope sensor readings at a resolution of 1ms per division. The decoded I^{2}C message is displayed in the lower part of Figure 5.3(b). The I^{2}C address of the accelerometer is 0x1D, and the gyroscope is 0x69. The relatively slow nature of I^{2}C communication results in the sensor reads taking considerably longer than the typical scheduler cycle. Figure 5.3(b) shows that the scheduler typically cycles well over ten times per millisecond. Whereas, the I^{2}C sensor read can easily exceed 4ms. Figure 5.3(b) shows that it typically takes longer than 8ms to complete the sensor task when the two sensors are read. Figure 5.3(a) show a magnetometer takes about half as much time, which is 4ms. So, there is a total of 20ms activity on the I2C communication lines every 50ms. During this time, the main scheduling loop is kept from cycling. This means that approximately 24% of processing time is wasted on simply polling I^{2}C hardware. Furthermore, Superloop schedulers work on the basis that all of the tasks are cycled though at least once before any state machine needs to change. So, if one of the tasks block for much more than a millisecond, there is a good chance another task will miss an important state update. For some systems (e.g., this sensor system), this could only result in losing a couple of data samples. However, for some systems (e.g., a critical control system), a missed state change could result in catastrophic failure. So, it is important to consider the possible failures before risking the use of blocking code.

In contrast, Figure 5.4 shows that, with the non-blocking example, the sensor task does not block the main scheduler from cycling through the tasks. For ease of comparison, Figure 5.4 displays the non-blocking example similarly to the blocking example in Figure 5.3. All of the signals are represented by the same names (D0, D1, etc.) that were used for the blocking example in Figure 5.3. The only difference for the example shown in Figure 5.4 is the use of a non-blocking I^2C driver to interface with the sensor. The non-blocking driver uses an I^2C task to execute callback functions. Thus, signal D11 is held high whenever the I^2C task function is called in addition to the I^2C read and writes. Since the non-blocking I^2C driver releases control back to the sensor task when read requests are called, the scheduler continues to cycle while the sensors are being read. Figure 5.4(a), which is also has a resolution of 20ms per division, shows both the I^2C tasks and the sensor tasks are cycling at the same time as the microcontroller is communicating with the sensors. There is the occasional blip where a task takes a little longer than usual. This blip is usually caused by the sensor task processing the data directly after the I^2C read is completed. Figure 5.4(b), which has a resolution 1ms per division, shows an example of how the processing of the data usually takes about 0.5ms.

So, even though the total I2C communication activity is still over 12ms every 50ms, there is only 2ms of this time where the Sensor Task blocks the main scheduling loop from cycling at its normal pace. This means that about only 4% of the processing time is spent waiting on the sensor task with a non-blocking I^{2}C driver. This means that about 20% of the processer availability is freed in this example. If the system never uses more than 80% of the processing time, then wasting 20% may not be a significant problem. This is particularly true if the system's only task is reading the sensors. However, there are occasions where the system using less than 50% might still have a time-critical occasion that cannot afford to be blocked for more than 1ms at a time. In this situation, the wasted time is irrelevant since the I^{2}C reads block for more than 4ms at a time.

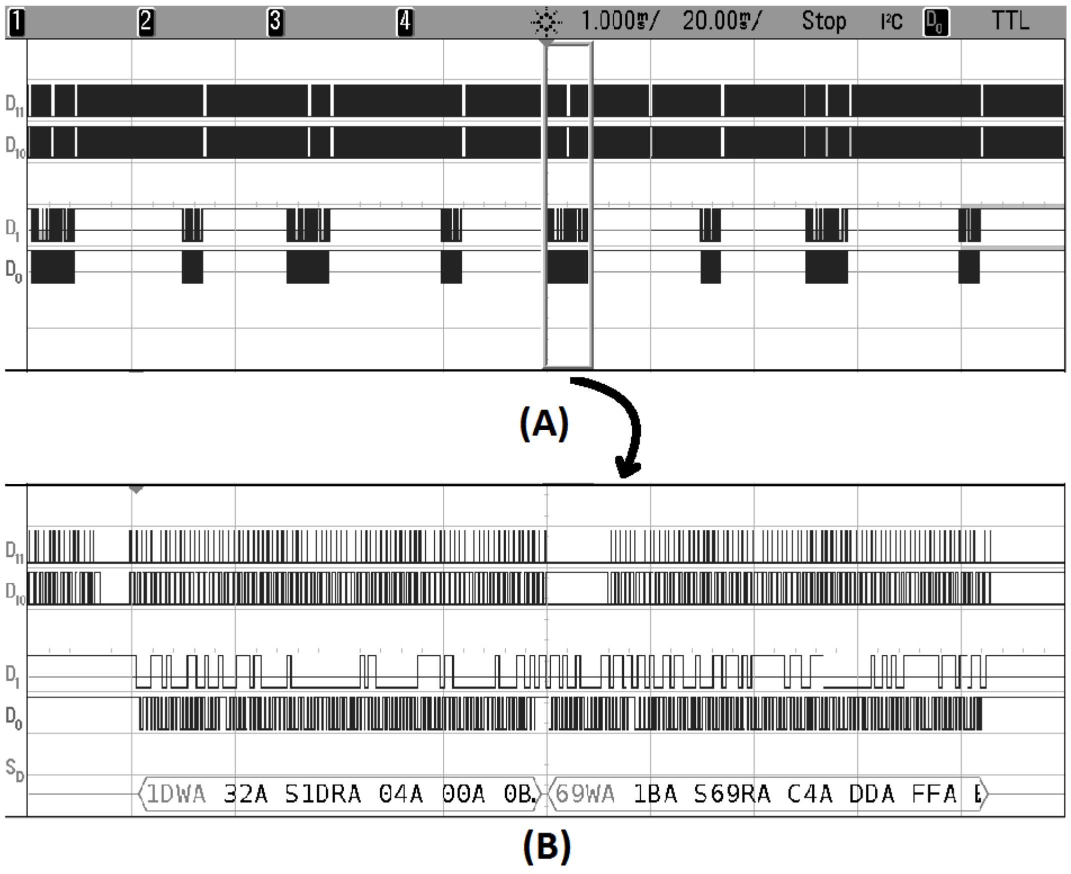
1.000ms/ 20.00ms/ Stop I²C TTL
D11
D10
D1
D0
SD
1DWA 32A 51DRA 04A 00A 0B
69WA 1BA 569RA C4A DDA FFA
(A)
(B)

Figure 5.4: Non-Blocking I^{2}C Waveform (a) 20ms Resolution (b) 1ms Resolution

While a sample period of 50ms may be very short for some sensors, it is actually relatively slow for inertial navigation applications where accelerometers, gyroscopes, and magnetometers are commonly used. This is a practical example of how blocking code can have a detrimental effect on a system. Clearly, the exact effects of blocking code are different in each application. As a result of many of the factors involved, it is difficult to determine analytically how blocking code will affect other tasks.

5.1.4 I^2C Throughput Analysis

One of the biggest problems with blocking calls is how they affect the performance of other tasks. For this example, the effects of blocking calls on the network communication tasks are examined. This section will examine how blocking I^2C calls will affect other completely separate tasks. The other tasks for this example will be the network. In order to examine the effects on the network, measurements from analog sensors are sent across the network. Analog sensors were chosen for the simple fact that they are not directly affected by the I^2C driver. The two analog sensors were read every 120ms and sent through the data manager directly to UART0 and across the ZigBee network to the Fusion coordinator module. This works out to be about 16.7 measurements per second. A monitoring computer receives the measurements from UART0 and the Fusion coordinator. The number of measurements are compared to see how many measurements are dropped by the network communication. This provides a practical way to measure the effects of blocking and non-blocking I^2C drivers on the network by comparing the amount of lost measurements for both.

All of the tests in this example use the same analog sensor measurement and network setup. Each test involves several I^2C sensors reading at different rates with blocking and non-blocking drivers. In order to keep the I^2C sensor measurements from directly affecting the throughput of the network, the measurements were not sent to the network or the UART. As in Section 5.1.3, the three I^2C sensors used are an accelerometer, gyroscope, and a magnetometer. Five tests with different sampling periods were performed using both blocking and non-blocking I^2C drivers. The different sampling periods demonstrate how the effects change when the system is pushed to its limits. The length for each test was approximately 5 minutes.

The results of using the non-blocking I^2C driver are shown in Table 5.1. The results from the blocking I^2C driver are shown in Table 5.1. The imprecise time it takes for the network to establish a connection resulted in slight variation in the total number of measurements for each test. Only measurements after a network connection was made were counted in this test since they would be the only measurements for which transmitting across the network was attempted. As shown in Table 5.1, none of the analog measurements were dropped by the network task when the non-blocking I^2C reads were used. On the other hand, 1,611 of the 4,995 analog measurements were dropped by the network tasks for the test utilizing blocking function calls.

Table 5.1: Non-Blocking I^2C Driver Effect on Network Tasks

I^2C Non-Blocking Code Sample Period	50 ms	40 ms	30 ms	20 ms	10 ms
Analog Sensor Measurements Collected	4994	4983	5015	4989	4992
Analog Sensor Measurements Dropped by Network	0	0	0	0	0

Table 5.1: Blocking I^2C Driver Effect on Network Tasks

I^2C Blocking Code Sample Period	50 ms	40 ms	30 ms	20 ms	10 ms
Analog Sensor Measurements Collected	4991	4995	5019	4995	4995
Analog Sensor Measurements Dropped by Network	0	0	17	18	1611

Another way to look at these test results is displayed in Figure 5.5. The throughput represents the percentage of measurements that make it through the network. Since none of the measurements were dropped when the non-blocking I^2C driver was being used, the throughput was 100% for all non-blocking I^2C tests (shown in Figure 5.5). The frequency of I^2C function calls indicates how often the I^2C sensors were being read. So, for the test where all three sensors were being read every 10ms, there were 300 function calls every second.

Figure 5.5 shows that increasing the frequency of blocking I^2C calls also increases the adverse effects on the network. The network throughput dramatically drops off to 68% when the when the I^2C blocking functions are called 300 times a second. However, these drastic effects of the blocking I^2C calls only occur when the system is taxed to its limits. Even under the best conditions, the network cannot handle much more than the 16.7 analog measurements per second without dropping measurements. At lower sample rates, the effects of blocking code are not noticeable. In fact, the network’s throughput is still over 99% at a 150Hz sample rate. This demonstrates that there are occasions where blocking code does not have the detrimental effects on other tasks if they are not very long and the maximum performance of the system is not needed.

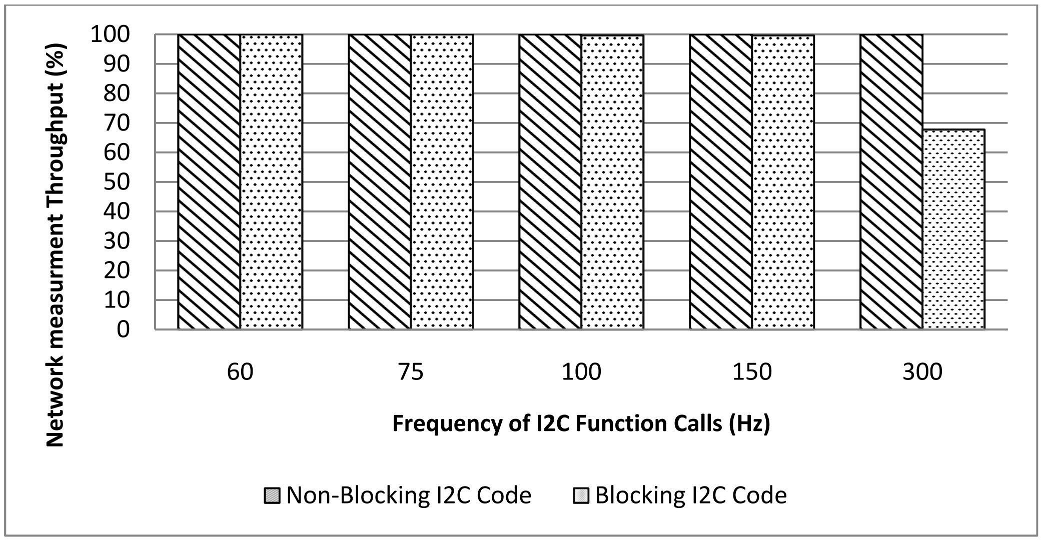

Figure 5.5: I2C Effect on Network Throughput

5.2 UART Driver Analysis

The UART protocol is used to communicate with both sensors and the monitoring computer. In addition to status information of the network, it is often necessary to send all of the data from the sensors to the monitoring computer. Since the fusion modules are designed to be capable of supporting a considerable number of sensors, the quantity of data sent to the monitoring computer can be quite large. Consequently, the UART interface needs to be capable of handling a high data rate.

UART communication is one of the simplest forms of serial communication. One aspect in particular that makes UART simple is the fact that it is limited to interfacing between only two devices (point-to-point communication). UART protocol has two communication lines in addition to a ground. As indicated by the name, one of the lines is for transmitting and the other is for receiving. Data is sent from one device’s transmit line to the other device’s receive line. Data is typically transmitted one byte at a time. The microcontroller on the fusion module’s UART hardware has received and transmit data registers that hold one byte of data. Since data is being sent to one location one byte at a time, the communication procedure simply consists of loading the transmit register one byte at a time and reading the receive register one byte at a time. The difficulty lies with the fact that, if another byte of data is sent before the receive register is read, the previous byte of data will be lost. Also, the UART module has to finish sending before the next byte can be loaded into the transmit register. This can present difficulty since most of the data messages sent are multiple bytes long.

5.2.1 UART Blocking Driver

The blocking UART driver is shown in Figure 5.6(a). The write function of the blocking driver is directly responsible for loading all of the byte from one message into the transmit register. So, the write function loads one byte into the transmit register and then sits there and poles the hardware until the byte is finished sending. Upon the completion of sending one byte, another is loaded into the transmit register. The write function continues this process until the entire message is sent, at which point the function releases control of the processor. The read function will retrieve a specified number of bytes from the receive transmitter as they come in. The read function continues to poll the receive register until the entire message is received. Since the time for the message to arrive is indeterminate, it is necessary to have a timeout counter that stops the waiting if it takes too long. However, it should be noted that the receive function is not used in the following tests.

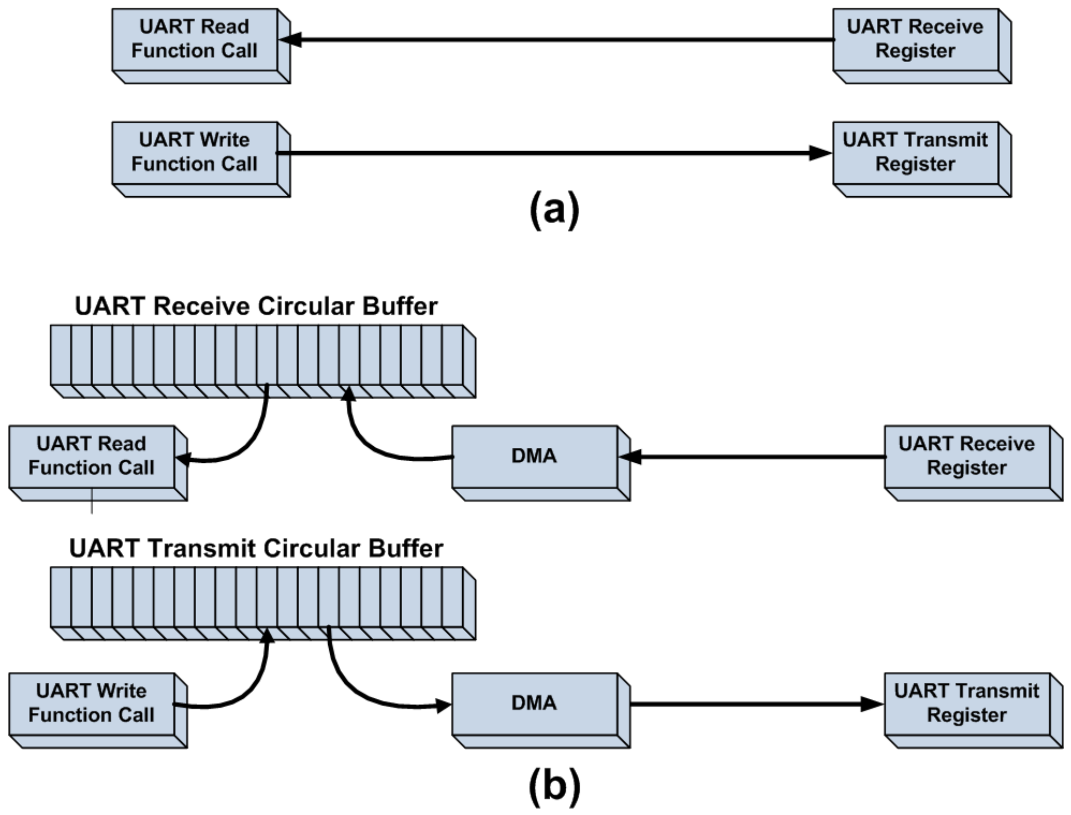

Figure 5.6: Fusion UART Driver (a) Blocking Driver (b) Non-Blocking Driver

5.2.2 UART Non-Blocking Driver

The UART driver is an example of a simpler non-blocking system. The only one of the non-blocking components discussed in Chapter 4 that was needed for this driver is a circular buffer. The main reasons for this are the simplicity of UART communications. UART communication simply consists of transmitting a series of bytes on one line, and receiving a series of bytes on another line. A couple of circular buffers are used: one for the receive side and the other for the transmit side (as shown in Figure 5.6(b)). The write function simply loads the message into the transmit buffer and returns. A direct memory access (DMA) module on the fusion board automatically loads the next byte into transmit register from the circular buffer after each byte is sent. The DMA further reduces the load on the processor since transferring of data occurs in hardware without the need to interrupt the processor at every byte. However, the DMA also adds to the complexity of the system since a modified circular buffer is needed to interface with the DMA. The receive set up is essentially the reverse of the write setup, with the DMA loading the circular buffer and the receive function retrieving data from the circular buffer. The primary difference is that, while the write function placed the entire message on the buffer, the read will only retrieve whatever data is available until it reaches the requested message length. Since each byte of a message is sent in the same manner, there is no need for a state machine. Consequently, it is also unnecessary to have a UART task to update a state machine. In addition, since the applications do not typically need to know exactly when a message is sent, using callback functions is not really necessary.

5.2.3 UART Waveform Analysis

For this example, the effects of a blocking UART driver were tested on UART1 of the Fusion modules. UART1 was chosen since UART0 is already being used to communicate directly with the monitoring computer. The rest of the framework on the Fusion module continued to run, including the non-blocking driver on UART0. For this example, a 100 byte long message was transmitted every 5ms on UART1. Similarly to the way the I^2C waveform analysis was performed, GPIO signals were triggered to indicate when a UART function started and ended. The digital analyzer was used to capture the results, which are displayed in Figures 5.7 and 5.8. Signal D13 is held high during any function call related to UART1. Signal D10 is used to represent the sensor task function just as it was in the I^{2}C example. Signals D2 and D3 are the UART's receive and transmit lines, respectively.

The effects of using blocking UART write function calls are displayed in Figure 5.7(a) at a resolution of 20ms per division. The thicker portions of D3 show exactly when the data is being transmitted. During this time, D13 is held high, indicating that the UART write function is maintaining control of the processor. The other tasks on the Superloop are blocked from being executed while the UART is sending the message. This is demonstrated by signal D10 being held low when the message is transmitted and continually cycles the rest of the time. Figure 5.7(b) displays a zoomed in view of the message at a resolution of 1ms per division. The 100 byte message blocks for well over 8ms as seen in Figure 5.7(b). Since this message is sent every 50ms, about 16% of the processing time is spent on the UART write function.

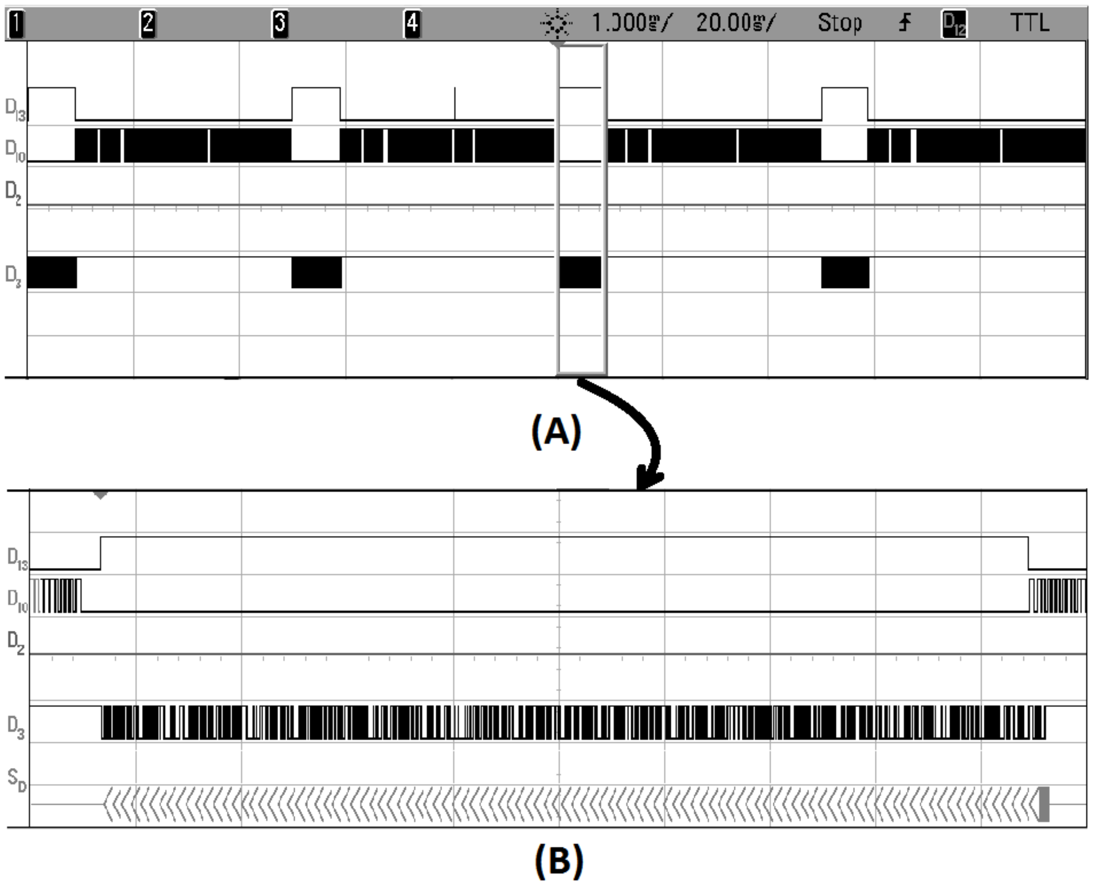

Figure 5.7: Blocking UART Waveform

Figure 5.8(a) shows the non-blocking results at a resolution of 20ms per division. Signal D13 shows that UART functions are called briefly at the beginning and end of the message transmission. The blip at the beginning of the transmission is the write function. Figure 5.8(b) has a resolution of 1ms per division, and shows that the write function takes less than 0.2ms to load the data into the circular buffer. The blip at the end of the message is the DMA interrupt routine, indicating the data transfer requested is complete. This interrupt is used for updating the reference pointers on the circular buffer, and requires even less time than the write function. The total processing time used for one message is less than 0.25ms. So, less than 0.5% of the processing time is dedicated to the UART when the driver is non-blocking. For this example, 15.5% percent of processor cycles are saved by using a non-blocking driver.

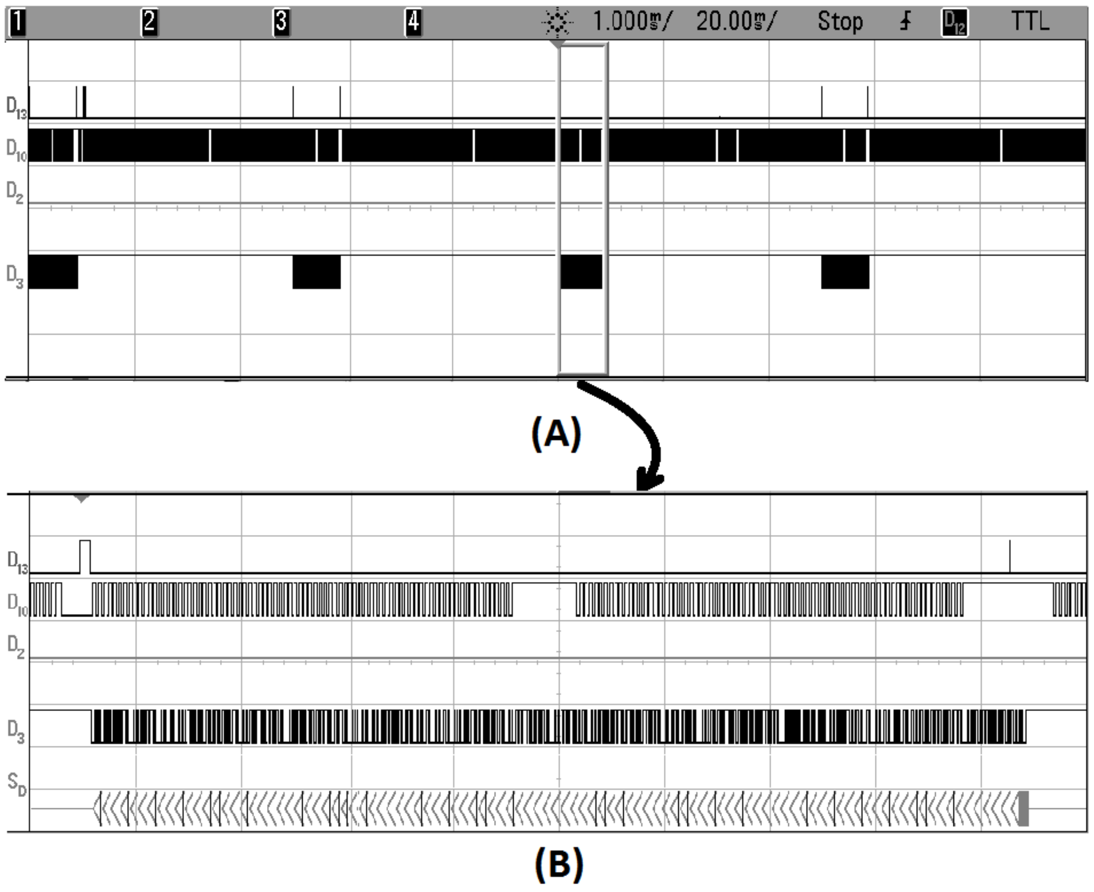

Figure 5.8: Non-Blocking UART Waveform

<u>5.2.4 UART Blocking Driver Effects on Network Throughput</u>

Similarly to the I^{2}C example, the network communication tasks are used to examine the effects of the blocking UART driver on other system tasks. About 16.7 analog measurements per second are sent through the data manager directly to the monitoring computer through UART0 and across the ZigBee network to the Fusion coordinator module. The number of measurements dropped by the network communication is used to determine the effects of blocking and non-blocking UART drivers. In order to keep the blocking driver from having any direct effect on the collected analog measurements, the UART0 driver was kept non-blocking for all of the tests. Only the UART1 driver was tested with blocking and non-blocking code. The same 100 byte long message used in Section 4.2.3 was sent directly to UART1 at different intervals. The different message intervals demonstrate how the effects change when the system is pushed to its limits. Just like the I^2C tests, the length for each of these tests was about 5 minutes.

The results of the using the non-blocking and blocking UART drivers are shown in Tables 5.3 and 5.4, respectively. The imprecise time it takes for the network to establish a connection resulted in slight variation in the total number of measurements for each test. Like the I^2C example, the non-blocking UART driver does not cause the network task to lose any of the transmitted measurements as shown in Table 5.1. Unlike the I^2C example, the blocking effects are still noticeable at the lower communication rates of the UART. It is easier to see this aspect in Figure 5.9, which graphically displays the network throughput as a percentage of analog measurements that make it to the coordinator. While higher frequency of blocking function calls generally resulted in the lower network throughput (85% at 100Hz), this trend was not consistent for all frequencies. The network only dropped 6 measurements with a blocking UART driver sending measurements at intervals of 30ms (see Table 5.4). It is possible that at 30ms the timing is such that network events do not occur at the same time as the blocking UART communication. However, the sensor scheduler resolution of 10ms prevented this explanation from being examined further. Consequently, the exact level of detrimental effects from blocking code is unpredictable. The only certainty is that, if the blocking function call is long enough or occurs often enough, they will have detrimental effects on other system tasks.

Table 5.3: Non-Blocking UART Driver Effect on Network Tasks

UART Non-Blocking Call Interval	50 ms	40 ms	30 ms	20 ms	10 ms
Analog Sensor Measurements Collected	4985	4984	5006	5000	5008
Analog Sensor Measurements Dropped by Network	0	0	0	0	0

Table 5.4: Blocking UART Driver Effect on Network Tasks

UART Blocking Call Interval	50 ms	40 ms	30 ms	20 ms	10 ms
Analog Sensor Measurements Collected	4981	5000	4987	4993	4984
Analog Sensor Measurements Dropped by Network	198	316	6	443	745

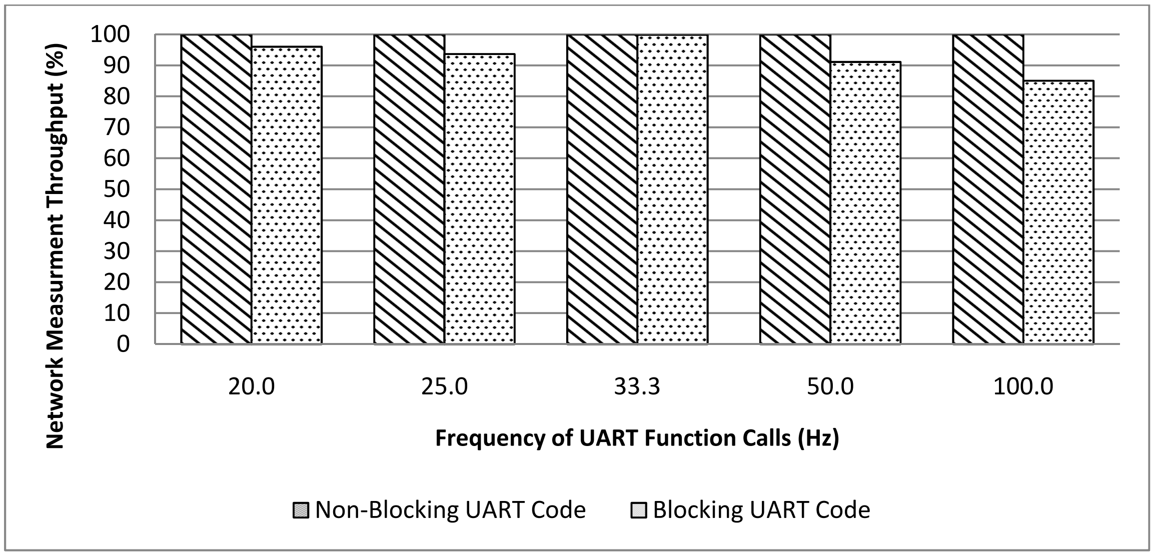

Figure 5.9: UART Effect on Network Throughput

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Summary and Conclusions

Software is a critical component of most embedded systems. Several coding techniques were discussed in this thesis that can greatly improve the effective use of embedded system hardware. Medium scale embedded system microcontrollers have specialized hardware that is designed to interface with different communication protocols. With supporting software that does not block, it is possible to fully utilize the specialized hardware that will, in turn, improve the performance of the embedded system. This thesis has demonstrated that blocking code in one area can have severely detrimental effects on other parts of a system. The degree to which blocking code affects the rest of the system is dependent on multiple factors of which the duration and frequency of the blocking calls play a significant part. Due to the unreliability of blocking code, it is highly recommended that the non-blocking techniques be used on critical systems (e.g., vehicle controls, weapons systems, etc.).

Developing non-blocking code can be quite a time-consuming challenge. By using the techniques in a systematic manner as described this thesis, the development time can be reduced. However, the wide variety of embedded systems makes it difficult to apply the same generic non-blocking coding techniques to all embedded systems. Consequently, it is not possible to reduce development time for non-blocking code to a level directly compatible with blocking code. So, blocking code is still worth using on occasions were reliability and performance are not as important as a quick development time.

An operating system accomplishes many of the same goals as non-blocking coding practices without significantly increasing development time. However, operating systems typically require more overhead in the form of memory and processor speed. So, in cases where the system has enough memory and processing speed to support an operating system, it is typically the better choice.

6.2 Future Work

Many non-blocking coding techniques relevant to medium scale embedded systems were covered in this thesis. However, there were other aspects that stood out as having the potential to contribute to this systematic approach to non-blocking systems. These areas are linked lists and secure digital (SD) card interface.

6.2.1 Linked List

Linked lists are standard data structures that can be used for buffering or long term storage. The primary advantage of linked lists is that data can be removed from any part the list. Whereas, circular buffers data can only remove data from one part of the buffer. The simplicity of circular buffers works very well for data that needs to be transferred in the order it is loaded into the buffer. However, there are situations when it is desirable to have the ability to prioritize data buffers so that more time-critical communication is processed first. This is particularly applicable to I^2C where multiple sensors are attached to the same module and use the same buffer. Some I^2C sensors have higher time constraints than other sensors. Therefore, there are occasions where it is beneficial to place sensors with higher time constraints at the front of the buffer. A linked list with each link having a priority would effectively achieve this purpose. The I^2C driver could load the links with the highest priority. There a few potential issue with priority management to consider. However, it would be worth examining if a generic linked list would be a practical non-blocking solution for this type of situations. It may be that a generic linked list solution is not possible without incurring too much overhead for medium scale embedded systems.

6.2.2 SD Card Interface

SD card are commonly used on embedded systems to store data. The Fusion node has an SD card to store sensor measurements and calibration data in separate files. A file management system is used to make it significantly easier to read and write data to the SD card. A file management system is included in the provided framework for the microcontroller on the Fusion sensor nodes [10]. This file management system supports the file allocation table (FAT) files system that most SD cards use [11]. The FAT file system is a common protocol that is supported by almost all computers, allowing files generated on an embedded system like the Fusion nodes to be read on SD reader on a computer. This is one of the reasons why SD are cards commonly used in embedded systems. However, the file management system included in with the framework uses blocking techniques. It would be worth investigating to see how difficult it would be to replace the blocking code with non-blocking code using the techniques described in this thesis. Like other embedded systems, the fusion node's file management system is composed of multiple levels [11]. The multiple layers add to the complication of implementing non-blocking techniques. Implementing non-blocking techniques requires all of the interfacing layers to conform to the same non-blocking practices. Therefore, it may be necessary to completely redo the file management, which could take a considerable amount of development time. It is possible that implementing non-blocking coding techniques could prove to be impracticable on complex file management systems.

REFERENCES

[1] Madhusudan, R., Keerthi, T., & Nitin, N. "Comparison of Process Scheduling Methodologies for Embedded Systems," Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on , vol.1, no.1, pp.387-391, 16-18 Dec. 2009

[2] Silberschatz, A. and Galvin, P. *Operating System Concepts*, 5th. Menlo Park California: Addison-Wesley 1998. Print

[3] Owen, M. "Portable Wireless Multipurpose Sensor System for Environmental Monitoring," M.S. thesis, Boise State University, Boise, ID, 2007.

[4] Loo, S., Owen, M., & Kiepert, J. "Modular, Portable, Reconfigurable, and Wireless Sensing System," Journal of ASTM International, Vol. 5, No. 4, May 2008.

[5] AT32UC3A3/A4 Series Preliminary." Atmel Corporation. March 2010. Web, 28 Feb 2011.

[6] "PIC18F8722 Family Datasheet" Microchip Technology Inc. Feb 2008. Web 28 Feb 2011

[7] "ZigBit 2.4 GHz Wireless Modules" Atmel Corporation. 2011, Web, 28 Feb 2011.

[8] Brown, Stephen, and Zvonko Vranesic. Fundamentals of Digital Logic with VHDL Design. 2nd. NY: McGraw-Hil, 2005. Print

[9] Pook, M., Loo, S. M., Planting, A., Kiepert, J., & Klein, D. (2010). "Coding Practices for Embedded Systems," *2010 ASEE Annual Conference* 10.1 (2010). Print.

[10] "AVR UC3 Software Framework".Atmel Corporation. May 2007. Web, 28 Feb 2011.

[11] "AVR114: Using the ATMEL File System management for AT32UC3x, At90USBx, and ATmega32U4" Atmel Corporation. Sept. 2008. Web, 28 Feb 2011.