MORGAN & CLAYPOOL PUBLISHERS

Embedded Systems Design with the Amtel AVR Microcontroller

Steven F. Barrett

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Mitchell A. Thornton, *Series Editor*

Embedded Systems Design with the Atmel AVR Microcontroller Part I

Synthesis Lectures on Digital Circuits and Systems

Pragmatic Power
William J. Eccles
2008

Multiple Valued Logic: Concepts and Representations
D. Michael Miller, Mitchell A. Thornton
2007

Finite State Machine Datapath Design, Optimization, and Implementation
Justin Davis, Robert Reese
2007

Atmel AVR Microcontroller Primer: Programming and Interfacing
Steven F. Barrett, Daniel J. Pack
2007

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

Pragmatic Circuits: DC and Time Domain
William J. Eccles
2006

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

Introduction to Logic Synthesis using Verilog HDL
Robert B.Reese, Mitchell A.Thornton
2006

Microcontrollers Fundamentals for Engineers and Scientists
Steven F. Barrett, Daniel J. Pack
2006

Embedded Systems Design with the Atmel AVR Microcontroller – Part I

Steven F. Barrett

www.morganclaypool.com

ISBN: 9781608451272 paperback
ISBN: 9781608451289 ebook

DOI 10.2200/S00138ED1V01Y200910DCS024

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Lecture #24
Series ISSN
Synthesis Lectures on Digital Circuits and Systems
Print 1932-3166 Electronic 1932-3174

Embedded Systems Design with the Atmel AVR Microcontroller Part I

Steven F. Barrett
University of Wyoming

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #24

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This textbook provides practicing scientists and engineers an advanced treatment of the Atmel AVR microcontroller. This book is intended as a follow on to a previously published book, titled "Atmel AVR Microcontroller Primer: Programming and Interfacing." Some of the content from this earlier text is retained for completeness. This book will emphasize advanced programming and interfacing skills. We focus on system level design consisting of several interacting microcontroller subsystems. The first chapter discusses the system design process. Our approach is to provide the skills to quickly get up to speed to operate the internationally popular Atmel AVR microcontroller line by developing systems level design skills. We use the Atmel ATmega164 as a representative sample of the AVR line. The knowledge you gain on this microcontroller can be easily translated to every other microcontroller in the AVR line. In succeeding chapters, we cover the main subsystems aboard the microcontroller, providing a short theory section followed by a description of the related microcontroller subsystem with accompanying software for the subsystem. We then provide advanced examples exercising some of the features discussed. In all examples, we use the C programming language. The code provided can be readily adapted to the wide variety of compilers available for the Atmel AVR microcontroller line. We also include a chapter describing how to interface the microcontroller to a wide variety of input and output devices. The book concludes with several detailed system level design examples employing the Atmel AVR microcontroller.

KEYWORDS

Atmel microcontroller, Atmel AVR, ATmega164, microcontroller interfacing, embedded systems design

Contents

Acknowledgments

I would like to dedicate this book to my close friend and writing partner Dr. Daniel Pack, Ph.D., P.E. Daniel elected to "sit this one out" because of a thriving research program in unmanned aerial vehicles (UAVs). Daniel took a very active role in editing the final manuscript of this text. Also, much of the writing is his from earlier Morgan & Claypool projects. In 2000, Daniel suggested that we might write a book together on microcontrollers. I had always wanted to write a book but I thought that's what other people did. With Daniel's encouragement we wrote that first book (and five more since then). Daniel is a good father, good son, good husband, brilliant engineer, a work ethic second to none, and a good friend. To you, good friend, I dedicate this book. I know that we will do many more together.

Steven F. Barrett
October 2009

Preface

In 2006 Morgan & Claypool Publishers (M&C) released the textbook, titled "Microcontrollers Fundamentals for Engineers and Scientists." The purpose of the textbook was to provide practicing scientists and engineers with a tutorial on the fundamental concepts and the use of microcontrollers. The textbook presented the fundamental concepts common to all microcontrollers. This book was followed in 2008 with "Atmel AVR Microcontroller Primer: Programming and Interfacing." The goal for writing this follow-on book was to provide details on a specific microcontroller family – the Atmel AVR Microcontroller. This book is the third in the series. In it the emphasis is on system level design and advanced microcontroller interfacing and programming concepts. Detailed examples are provided throughout the text.

APPROACH OF THE BOOK

We assume the reader is already familiar with the Atmel AVR microcontroller line. If this is not the case, we highly recommend a first read of "Atmel AVR Microcontroller Primer: Programming and Interfacing." Although some of the content from this earlier volume is retained in this current book for completeness, the reader will be much better served with a prior solid background in the Atmel AVR microcontroller family.

Chapter 1 contains an overview of embedded systems level design. Chapter 2 presents a brief review of the Atmel AVR subsystem capabilities and features. Chapters 3 through 7 provide the reader with a detailed treatment of the subsystems aboard the AVR microcontroller. Chapter 8 ties together the entire book with several examples of system level design.

Steven F. Barrett
October 2009

CHAPTER 1

Embedded Systems Design

Objectives: After reading this chapter, the reader should be able to

- Define an embedded system.
- List all aspects related to the design of an embedded system.
- Provide a step-by-step approach to embedded system design.
- Discuss design tools and practices related to embedded systems design.
- Apply embedded system design practices in the design of a microcontroller system employing several interacting subsystems.

In this first, chapter we begin with a definition of just what is an embedded system. We then explore the process of how to successfully (and with low stress) develop an embedded system prototype that meets established requirements. We conclude the chapter with an extended example. The example illustrates the embedded system design process in the development and prototype of instrumentation for a kinesiology laboratory. We will revisit this example throughout the book.

1.1 WHAT IS AN EMBEDDED SYSTEM?

An embedded system contains a microcontroller to accomplish its job of processing system inputs and generating system outputs. The link between system inputs and outputs is provided by a coded algorithm stored within the processor's resident memory. What makes embedded systems design so interesting and challenging is the design must also take into account the proper electrical interface for the input and output devices, limited on-chip resources, human interface concepts, the operating environment of the system, cost analysis, related standards, and manufacturing aspects (Anderson, 2008).

1.2 EMBEDDED SYSTEM DESIGN PROCESS

In this section, we provide a step-by-step approach to develop the first prototype of an embedded system that will meet established requirements. There are many formal design processes that we could study. We will concentrate on the steps that are common to most. We purposefully avoid formal terminology of a specific approach and instead concentrate on the activities that are accomplished as a system prototype is developed. The design process we describe is illustrated in Figure 1.1 using a Unified Modeling Language (UML) activity diagram. We discuss the proper construction process of activity diagrams later in the chapter.

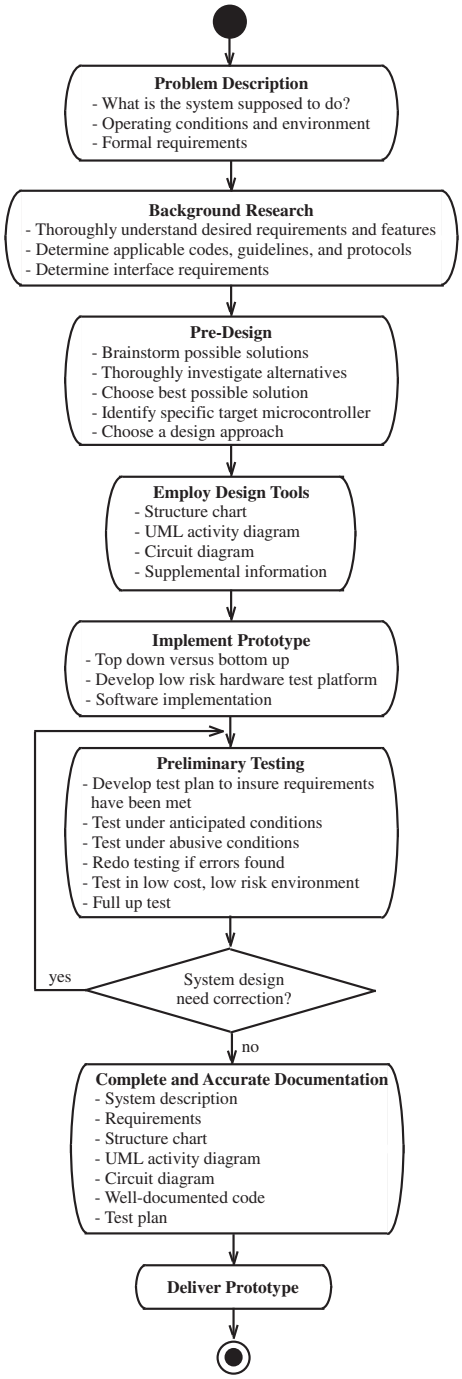


Figure 1.1: Embedded system design process.

1.2.1 PROBLEM DESCRIPTION

The purpose of the problem description step is to determine the goal of the system. To achieve this step, you must thoroughly investigate what the system is supposed to do. Questions to raise and answer during this step include but are not limited to:

- What is the system supposed to do?
- Where will it be operating and under what conditions?
- Are there any restrictions placed on the system design?

To answer these questions, the embedded designer interacts with the client to ensure clear agreement on what is to be done. The final result of this step is a detailed listing of system requirements and related specifications. The establishment of clear, definable requirements may necessitate considerable interaction between the design engineer and the client. It is essential that both parties agree on system requirements before proceeding further in the design process.

1.2.2 BACKGROUND RESEARCH

Once a detailed list of requirements has been established, the next step is to perform background research related to the design. In this step, the designer will ensure they understand all requirements and features desired by the client. This will again involve interaction between the designer and the client. The designer will also investigate applicable codes, guidelines, protocols, and standards related to the project. This is also a good time to start thinking about the interface between different portions of the project, particularly, the input and output devices peripherally connected to the microcontroller. The ultimate objective of this step is to have a thorough understanding of the project requirements and related project aspects.

1.2.3 PRE-DESIGN

The goal of the pre-design step is to convert a thorough understanding of the project into possible design alternatives. Brainstorming is an effective tool in this step. Here, a list of alternatives is developed. Since an embedded system typically involves both hardware and/or software, the designer can investigate whether requirements could be met with a hardware only solution or some combination of hardware and software. Generally speaking, a hardware only solution executes faster; however, the design is fixed once fielded. On the other hand, software provides flexibility and a slower execution speed. Most embedded design solutions will use a combination of both to capitalize on the inherent advantages of each.

In this step, the designer must also investigate potential digital design technology solutions. There are a wide variety of alternatives including:

- programmable gate arrays,
- microprocessors,

- digital signal processors (DSP),
- microcontrollers, and
- mixed mode processing combining available technologies.

A description of available technologies and their inherent features is provided in "Microcontrollers Fundamentals for Engineers and Scientists (Barrett, 2006)."

Once alternative design solutions have been investigated, it is now time to choose a single approach to develop into a full design. To make the final choice, the tradeoffs between competing designs must be evaluated. It is important to involve the client in this step.

Once a design alternative has been selected, the general partition between hardware and software can be determined. It is also an appropriate time to select a specific hardware device to implement the prototype design. If a microcontroller technology has been chosen, it is now time to select a specific controller. This is accomplished by answering the following questions:

- What microcontroller subsystems or features i.e., ADC, PWM, timer, etc.) are required by the design?
- How many input and output pins are required by the design?
- What is the expected maximum operating speed of the microcontroller expected to be?
- Is the specific microcontroller and a full complement of related documentation and support tools readily available? Employing the latest, new processor without complete documentation, support, and development tools leads to stress and frustration.
- As the design progresses, is it easy to migrate to a pin-for-pin compatible microcontroller with larger memory resources? Atmel makes this easy to do. For example, the ATmega164 (16K) is also available in a 32K variant (ATmega324) and a 64K variant (Atmega644).
- Is the specific microcontroller going to be available in sufficient quantities for a production run? You do not want to choose a specific controller that is too old or too new unless sufficient production quantities have been verified as being available.
- Are all required peripheral hardware readily available with a full complement of supporting documentation? Again, employing hardware without proper documentation leads to stress and frustration.

Choosing a specific microcontroller may seem a bit overwhelming. Atmel eases the stress by providing a complete family of microcontrollers. Also, Atmel's STK500 programming board (to be discussed in the next chapter) provides low cost programming support for virtually all controllers in the ATmega product line. For example, if the ATmega164 was chosen as the potential design solution, it is easy to migrate to a pin-for-pin compatible microcontroller with additional memory resources (ATmega324, ATmega644).

1.2.4 DESIGN

With a clear view of system requirements and features, a general partition determined between hardware and software, and a specific microcontroller chosen, it is now time to tackle the actual design. It is important to follow a systematic and disciplined approach to design. This will allow for low stress development of a documented design solution that meets requirements. In the design step, several tools are employed to ease the design process. They include the following:

- Employing a top-down design, bottom up implementation approach,
- Using a structure chart to assist in partitioning the system,
- Using a Unified Modeling Language (UML) activity diagram to work out program flow, and
- Developing a detailed circuit diagram of the entire system.

Let's take a closer look at each of these. The information provided here is an abbreviated version of the one provided in "Microcontrollers Fundamentals for Engineers and Scientists." The interested reader is referred there for additional details and an indepth example (Barrett, 2006).

Top down design, bottom up implementation. An effective tool to start partitioning the design is based on the techniques of top-down design, bottom-up implementation. In this approach, you start with the overall system and begin to partition it into subsystems. At this point of the design, you are not concerned with how the design will be accomplished but how the different pieces of the project will fit together. A handy tool to use at this design stage is the structure chart. The structure chart shows the hierarchy of how system hardware and software components will interact and interface with one another. You should continue partitioning system activity until each subsystem in the structure chart has a single definable function (Page-Jones, 1988). A sample structure chart is provided in Section 1.3.

UML Activity Diagram. Once the system has been partitioned into pieces, the next step in the design process is to start working out the details of the operation of each subsystem we previously identified. Rather than beginning to code each subsystem as a function, we will work out the information and control flow of each subsystem using another design tool: the Unified Modeling Language (UML) activity diagram. The activity diagram is simply a UML compliant flow chart. UML is a standardized method of documenting systems. The activity diagram is one of the many tools available from UML to document system design and operation. The basic symbols used in a UML activity diagram for a microcontroller based system are provided in Figure 1.2 (Fowler, 2000).

To develop the UML activity diagram for the system, we can use a top-down, bottom-up, or a hybrid approach. In the top-down approach we begin by modeling the overall flow of the algorithm from a high level. If we choose to use the bottom-up approach, we would begin at the bottom of the structure chart and choose a subsystem for flow modeling. The specific course of action chosen depends on project specifics. Often, a combination of both techniques, a hybrid approach, is used. You should work out all algorithm details at the UML activity diagram level prior to coding any software. If you, first, can not explain the system operation at this higher level first, you have no

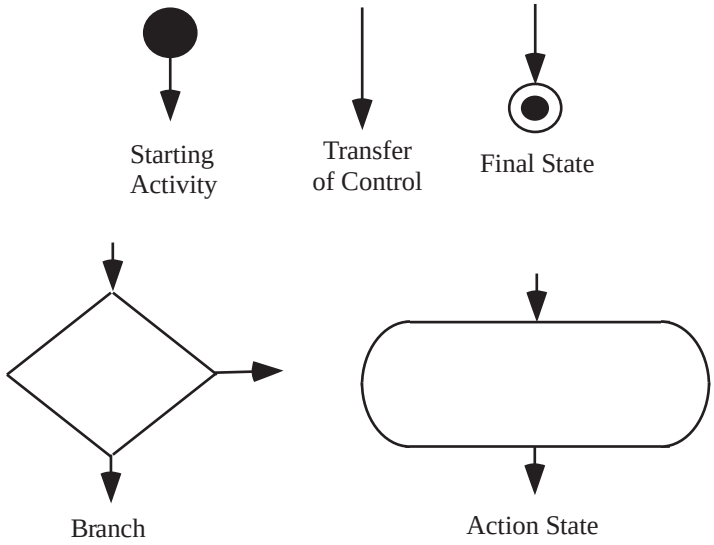


Figure 1.2: UML activity diagram symbols. Adapted from (Fowler, 2000).

business being in the detail of developing the code. Therefore, the UML activity diagram should be of sufficient detail so you can code the algorithm directly from it (Page-Jones, 1988).

In the design step, a detailed circuit diagram of the entire system is developed. It will serve as a roadmap to implement the system. It is also a good idea, at this point, to investigate available design information relative to the project. This would include hardware design examples, software code examples, and application notes available from manufacturers.

At the completion of this step, the prototype design is ready for implementation and testing.

1.2.5 IMPLEMENT PROTOTYPE

To successfully implement a prototype, an incremental approach should be followed. Again, the top-down design, bottom-up implementation provides a solid guide for system implementation. In an embedded system design involving both hardware and software, the hardware system including the microcontroller should be assembled first, so the required signals of the software can interact. As the hardware prototype is assembled on a prototype board, each component is tested for proper operation as it is brought online. This allows the designer to pinpoint malfunctions as they occur. A low cost, low risk hardware prototype approach is discussed in more detail in the next chapter.

Once the hardware prototype is assembled, coding may commence. As before, software should be brought online incrementally. You may use a top down, bottom up, or hybrid approach, depending on the nature of the software. Because of bringing the software online in slow, stated increments, issues can be identified and corrected early on.

1.2.6 PRELIMINARY TESTING

To test the system, a detailed test plan must be developed. Tests should be developed to verify that the system meets all of its requirements and also intended system performance in an operational environment. The test plan should also include scenarios in which the system is used in an unintended manner. As before, a top-down, bottom-up, or hybrid approach can be used to test the system.

Once the test plan is completed, actual testing may commence. The results of each test should be carefully documented. As you go through the test plan, you will probably uncover a number of run time errors in your algorithm. After you correct a run time error, the entire test plan must be performed again. This ensures that the new fix does not have an unintended affect on another part of the system. Also, as you process through the test plan, you will probably think of other tests that were not included in the original test document. These tests should be added to the test plan. As you go through testing, realize your final system is only as good as the test plan that supports it!

Once testing is completed, you might try another level of testing where you intentionally try to "jam up" the system. In another words, try to get your system to fail by trying combinations of inputs that were not part of the original design. A robust system should continue to operate correctly in this type of an abusive environment. It is imperative that you design robustness into your system. When testing on a low cost simulator is complete, the entire test plan should be performed again with the actual system hardware. Once this is completed, you should have a system that meets its requirements!

1.2.7 COMPLETE AND ACCURATE DOCUMENTATION

With testing complete, the system design should be thoroughly documented. Much of the documentation will have already been accomplished during system development. Documentation will include the system description, system requirements, the structure chart, the UML activity diagrams documenting program flow, the test plan, results of the test plan, system schematics, and properly documented code. To properly document code, you should carefully comment all functions, describing their operation, inputs, and outputs. Also, comments should be included within the body of the function, describing key portions of the code. Enough detail should be provided such that code operation is obvious. It is also extremely helpful to provide variables and functions within your code names that describe their intended use.

You might think that a comprehensive system documentation is not worth the time or effort to complete it. Complete documentation pays rich dividends when it is time to modify, repair, or update an existing system. Also, well-documented code may often be reused in other projects: a method for efficient and timely development of new systems.

1.3 EXAMPLE: KINESIOLOGY AND HEALTH LABORATORY INSTRUMENTATION

To illustrate the design process, we provide an in depth example. The Kinesiology and Health (KNH) Department required assistance in developing a piece of laboratory research equipment. Members

of the research team met with the design engineer to describe the system and determine specific requirements.

Problem description and background research. The system concept is illustrated in Figure 1.3. The KNH researchers needed a display panel containing two columns of large (10 mm diameter) red LEDs. The LEDs needed to be viewable at a distance of approximately 5 meters. The right column of LEDs would indicate the desired level of exertion for the subject under test. This LED array would be driven by an external signal generator using a low frequency ramp signal. The left column of LEDs would indicate actual subject exertion level. This array would be driven by a powered string potentiometer. As its name implies, a string potentiometer is equipped with a string pull. The resistance provided by the potentiometer is linearly related to the string displacement. Once powered, the string pot provides an output voltage proportional to the string pull length. The end of the string would be connected to a displacing arm on a piece of exercise equipment (e.g., a leg lift apparatus). After the requirements were determined, the characteristics of the available signal generator and the string pot were reviewed in detail.

Pre-design. With requirements clearly understood, the next step was to brainstorm possible solutions. Two possible alternatives clearly became evident: a complete hardware implementation or a microcontroller-based solution. Whichever approach was chosen, it was clear that the following features were required:

- The ability to independently drive a total of 28 large (10 mm diameter) LEDs.
- An interface circuit to drive each LED.
- Two analog-to-digital channels to convert the respective string potentiometer and signal generator inputs into digital signals.
- Input buffering circuitry to isolate and protect the panel from the string pot and signal generator and to guard against accidental setting overload.
- An algorithm to link the analog input signals to the appropriate LED activation signals.

A complete hardware implementation was chosen for the project. Due to the high number of output pins required to drive the LEDs and the minimal algorithm required to link the analog input to the LED outputs, it was determined that a microcontroller was not the best choice for this project. (Do not despair; we will implement this project using a microcontroller later in the book!)

Design. With a design alternative chosen, the next step was to develop a detailed design. The overall project was partitioned using the structure chart illustrated in Figure 1.4.

From the structure chart, a circuit diagram was developed for each subsystem within the circuit. To develop the analog-to-digital converter with a 14-bit output, a small test circuit using only 4 bits was first developed to test the concept. The small scale test circuit is shown in Figure 1.5. The full up, two array, 14-bit output driving a total of 28 large LEDs circuit diagram is provided in Figure 1.6. The LED circuit is described in greater detail in Chapter 7.

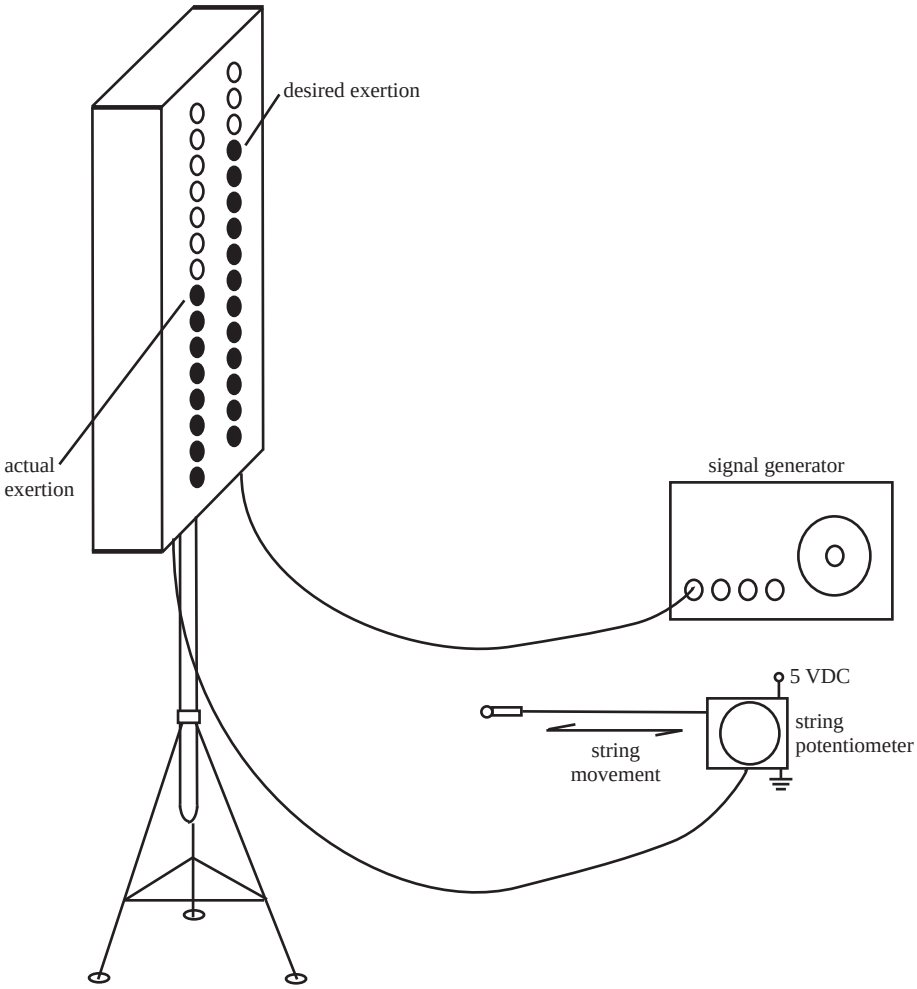


Figure 1.3: KNH project overview. A display panel contains two arrays of 14 large (10 mm diameter) red LEDs each. The right array driven by a signal generator indicates the desired assertion level. The left array is driven by a string potentiometer transducer. This LED array indicates the actual exertion level as determined by the string length of the potentiometer.

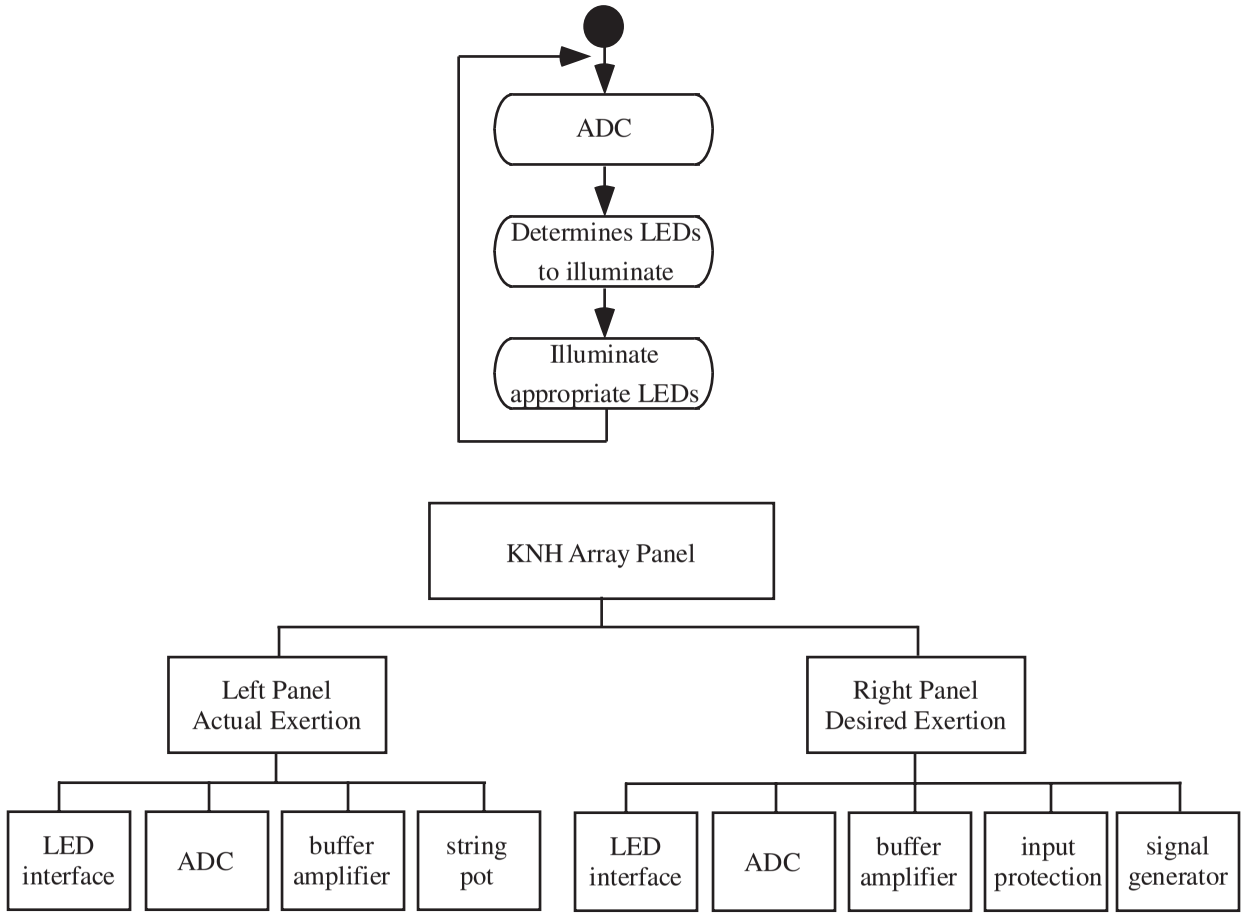


Figure 1.4: UML activity diagram and structure chart for KNH arrays.

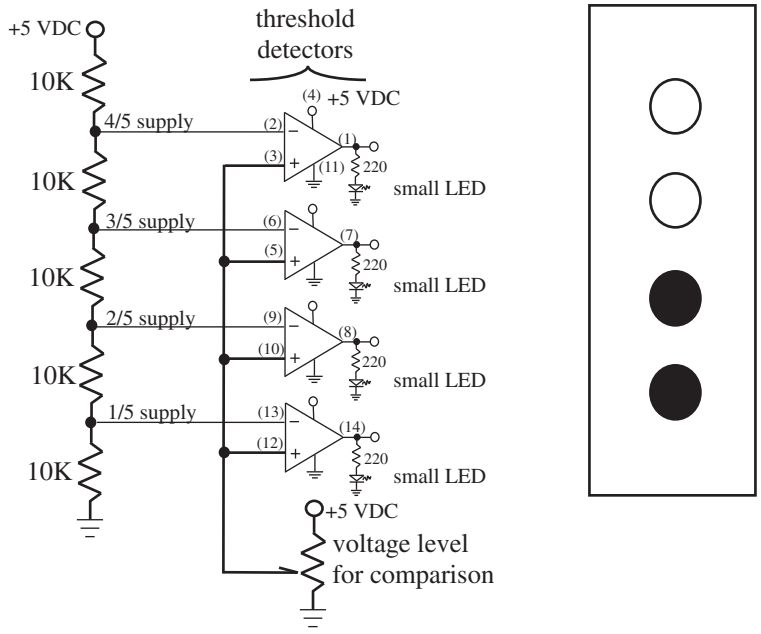


Figure 1.5: Small scale test circuit for the KNH arrays. The input voltage is compared against a bank of threshold detectors, each set to a different fraction of the supply voltage. As the input voltage is increased, more LEDs will illuminate. This is commonly referred to as a rain gage indicator. The threshold detector configuration will be discussed, in detail, later in the book.

Implement Prototype and Testing. The circuit was then implemented on a powered, prototype board to ensure correct operation. The circuit was slowly brought online, a subsystem at a time, starting from the lowest level of the structure chart. Each subsystem was implemented and tested as it was brought online.

Once the array was completely prototyped and tested, the design was converted over to a printed circuit board (PCB) layout. The PCB was assembled and tested to ensure proper operation. The PCB circuit was tested against the prototyped circuit.

With the circuit correctly operating, it was installed in an aluminum chassis and delivered to the Kinesiology and Health Department. The research team then tested the array panel under actual laboratory conditions. As a result of the testing, several circuit modifications were required to allow a "programmable" rest and exercise period for the desired exertion array. As shown in Figure 1.7, any rest period and exercise period may be selected by careful choice of the signal parameters applied to the desired exertion array. In the example shown, a rest period of 2s and an exercise period of 1s was desired. The signal generator was set for a ramp signal with the parameters shown. The input protection circuitry limits the excursion of the input signal to approximately 0 to 4 volts.

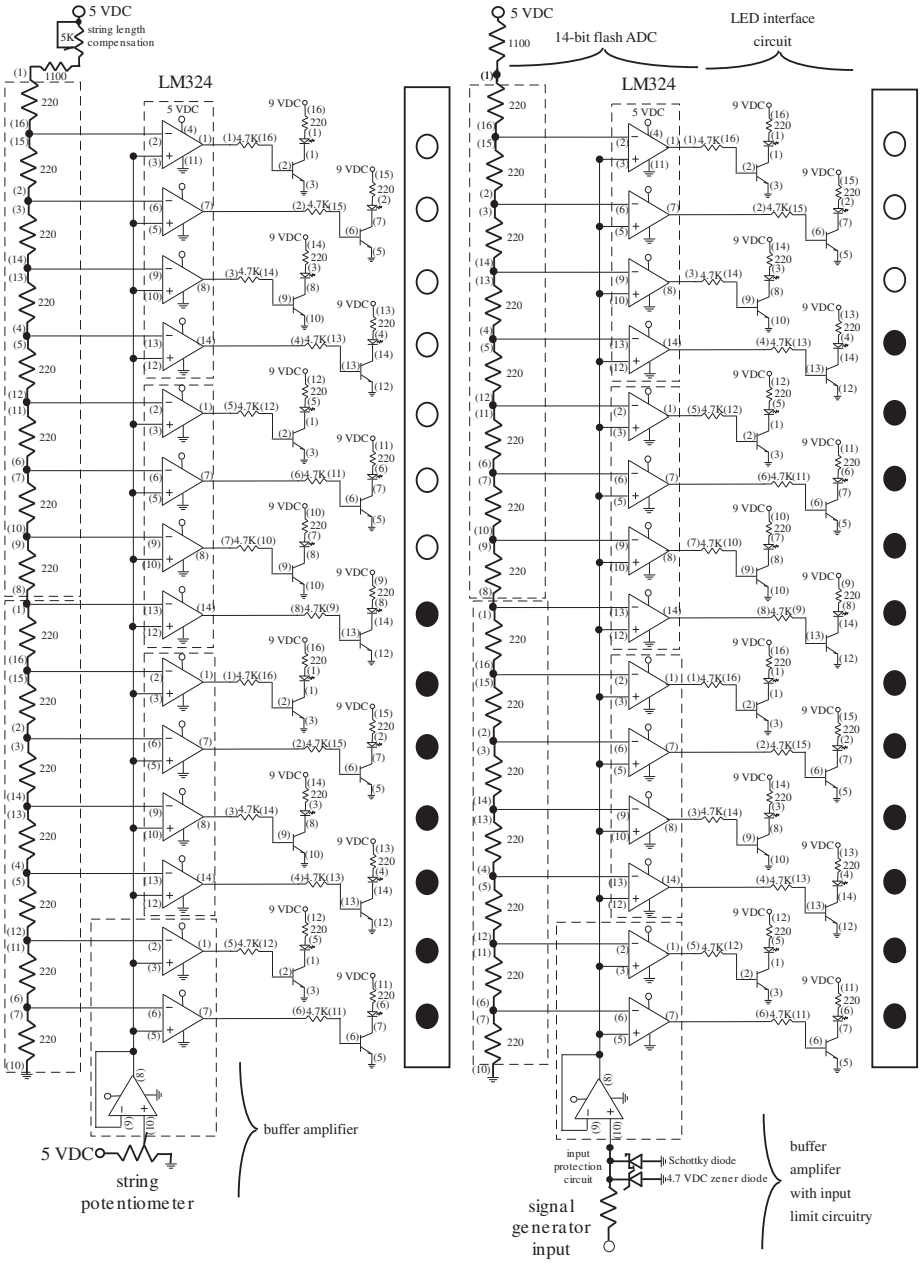


Figure 1.6: Circuit diagram for the KNH arrays. (left) Actual exertion array and (right) desired exertion array.

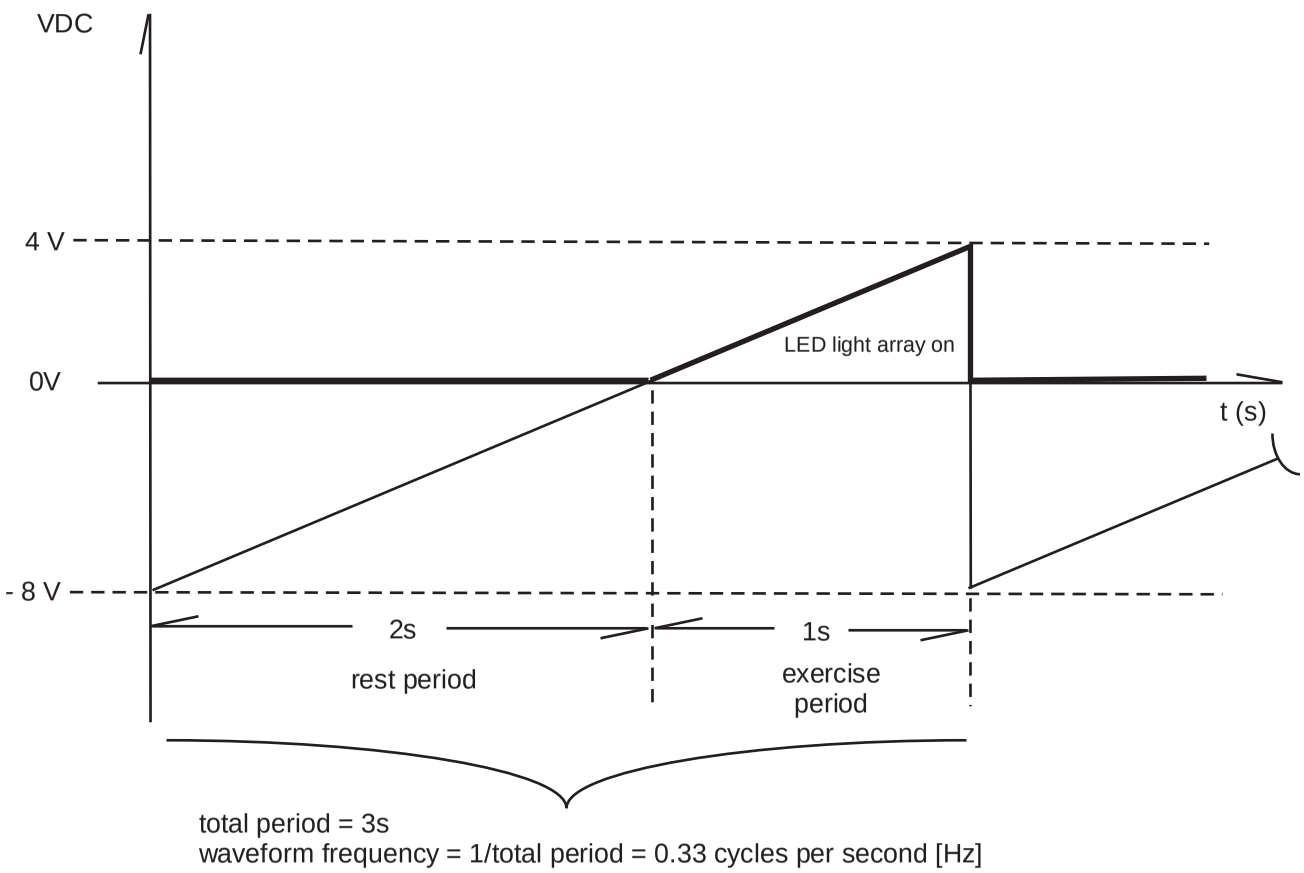


Figure 1.7: KNH array settings. LEDs will illuminate for input voltages between 0 and 4 VDC. More LEDs, incrementally, illuminate as input voltage increases in response to signal generator input signal. The input to the array is protected by a 4.7 VDC zener diode and a Schottky diode to limit voltage input to levels between approximately 0 and 4.7 VDC.

1.4 SUMMARY

In this chapter, we discussed the design process, related tools, and applied the process to a real world design. As previously mentioned, this design example will be periodically revisited throughout the text. It is essential to follow a systematic, disciplined approach to embedded systems design to successfully develop a prototype that meets established requirements.

1.5 CHAPTER PROBLEMS

1.1. What is an embedded system?

1.2. What aspects must be considered in the design of an embedded system?

1.3. What is the purpose of the structure chart, UML activity diagram, and circuit diagram?

1.4. Why is a system design only as good as the test plan that supports it?

1.5. During the testing process, when an error is found and corrected, what should now be accomplished?

1.6. Discuss the top-down design, bottom-up implementation concept.

1.7. Describe the value of accurate documentation.

1.8. What is required to fully document an embedded systems design?

1.9. Calculate the signal parameters required to drive the KNH array with a 1s rest time and a 2s exercise time.

1.10. What is the purpose of the input protection circuitry for the desired exertion array in the KNH panel?

REFERENCES

M. Anderson, Help Wanted: Embedded Engineers Why the United States is losing its edge in embedded systems, *IEEE-USA Today's Engineer*, Feb 2008.

S. Barrett and D. Pack "Microcontroller Fundamentals for Engineers and Scientists," Morgan & Claypool, 2006.

M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language," Addison Wesley, 2000.

M. Page Jones, "The Practical Guide to Structured Systems Design," Yourdon Press, 1988.

CHAPTER 2

Atmel AVR Architecture Overview

Objectives: After reading this chapter, the reader should be able to

- Provide an overview of the RISC architecture of the ATmega164.
- Describe the different ATmega164 memory components and their applications.
- Explain the ATmega164 internal subsystems and their applications.
- Highlight the operating parameters of the ATmega164.
- Summarize the special ATmega164 features.

2.1 ATMEGA164 ARCHITECTURE OVERVIEW

In this section, we describe the overall architecture of the Atmel AVR ATmega164. We begin with an introduction to the concept of the Reduced Instruction Set Computer (RISC) and briefly describe the Atmel Assembly Language Instruction Set. A brief introduction of the assembly language is warranted since we will be programming mainly in C throughout the book. [1] We then provide a detailed description of the ATmega164 hardware architecture.

2.1.1 REDUCED INSTRUCTION SET COMPUTER—RISC

A microcontroller is an entire computer system contained within a single integrated circuit or chip. Microcontroller operation is controlled by a user-written program interacting with the fixed hardware architecture resident within the microcontroller. A specific microcontroller architecture can be categorized as either accumulator-based, register-based, stack-based, or a pipeline architecture.

The Atmel ATmega164 is a register-based architecture. In this type of architecture, all operands of an operation are stored in registers collocated with the central processing unit (CPU). This means that before an operation is performed, the computer loads all necessary data for the operation to its CPU. The result of the operation is also stored in a register. During program execution, the CPU interacts with the register set and minimizes slower memory accesses. Memory accesses are typically handled as background operations.

Coupled with the register-based architecture is an instruction set based on the RISC concept. A RISC processor is equipped with a complement of very simple and efficient basic operations. More

[1] We want to emphasize the importance of understanding the assembly language programming language; however, especially for implementing applications that are time critical.

complex instructions are built up from these very basic operations. This allows for efficient program operation. The Atmel ATmega164 has 131 RISC type instructions. Most can be executed in a single clock cycle. The main objective of the RISC processor is to simplify hardware and instructions for the purpose of increasing the overall performance of the computer. The ATmega164 is also equipped with additional hardware to allow for the multiplication operation in two clock cycles. In many other microcontroller architectures, multiplication typically requires many more clock cycles. For additional information on the RISC architecture, the interested reader is referred to Hennessy and Patterson.

The Atmel ATmega164 is equipped with 32 general purpose 8-bit registers that are tightly coupled to the processor's arithmetic logic unit (ALU) within the CPU. The processor is designed following the Harvard Architecture format, as it is equipped with separate, dedicated memories and buses for program and data information. The register-based Harvard Architecture, coupled with the RISC-based instruction set, allows for fast and efficient program execution and allows the processor to complete an assembly language instruction every clock cycle. Atmel indicates the ATmega164 can execute 20 million instructions per second (MIPS) when operating at the maximum clock speed of 20 MHz (Atmel).

2.1.2 ASSEMBLY LANGUAGE INSTRUCTION SET

An instruction set is a group of instructions a machine "understands" to execute. A large number of instructions provide flexibility but requires more complex hardware. Thus, an instruction set is unique for a given hardware and can not be used with another hardware configuration. Atmel has equipped the ATmega164 with 131 different instructions. Most of these instructions are executed in a single clock cycle.

For the most efficient and fast execution of a given microcontroller, assembly language should be employed. Assembly language is written to efficiently interact with a specific microcontroller's resident hardware. To effectively use the assembly language, the programmer must be thoroughly familiar with the low-level architecture details of the controller. Furthermore, the learning curve for a given assembly language is quite steep, and lessons learned do not always transfer to another microcontroller. For these reasons, we will program the Atmel ATmega164 using the C language throughout the text. The C programming language allows for direct control of microcontroller hardware at the register level while being portable to other microcontrollers in the AVR line. When a C program is compiled during the software development process, the program is first converted to assembly language and then to the machine code for the specific microcontroller.

We must emphasize that programming in C is not better than assembly language or vice versa. Both approaches have their inherent advantages and disadvantages. We have chosen to use C in this book for, the reasons previously discussed. Throughout this book, we will use the ImageCraft ICC AVR compiler for our coding examples (ImageCraft). The specific features discussed are also available on other Atmel AVR C compilers.

2.1.3 C OPERATOR SIZE

When declaring a variable in C, the number of bits used to store the operator is also indirectly specified. In Figure 2.1, we provide a list of common C variable sizes used with the ImageCraft ICC AVR compiler. The size of other variables such as pointers, shorts, longs, etc., are contained in the compiler documentation (ImageCraft).

Type	Size	Range
unsigned char	1	0..255
signed char	1	-128..127
unsigned int	2	0..65535
signed int	2	-32768..32767
float	4	+/-1.175e-38.. +/-3.40e+38
double	4	+/-1.175e-38.. +/-3.40e+38

Figure 2.1: C variable sizes used with the ImageCraft ICC AVR compiler.

When programming microcontrollers, it is important to know the number of bits used to store the variable and also where the variable will be assigned. For example, assigning the contents of an unsigned char variable, which is stored in 8-bits, to an 8-bit output port will have a predictable result. However, assigning an unsigned int variable, which is stored in 16-bits, to an 8-bit output port does not provide predictable results. It is wise to insure your assignment statements are balanced for accurate and predictable results. Recall the modifier "unsigned" indicates all bits will be used to specify the magnitude of the argument. Signed variables will use the left most bit to indicate the polarity (±) of the argument.

2.1.4 BIT TWIDDLING

It is not uncommon, in embedded system design projects, to have every pin on a microcontroller employed. Furthermore, it is not uncommon to have multiple inputs and outputs assigned to the same port but on different port input/output pins. Some compilers support specific pin reference.

Another technique that is not compiler specific is bit twiddling. Figure 2.2 provides bit twiddling examples on how individual bits may be manipulated without affecting other bits. The information provided here was extracted from the ImageCraft ICC AVR compiler documentation (ImageCraft).

Syntax	Description	Example
a | b	bitwise or	PORTA |= 0x80; // turn on bit 7 (msb)
a & b	bitwise and	if ((PINA & 0x81) == 0) // check bit 7 and bit 0
a ^ b	bitwise exclusive or	PORTA ^= 0x80; // flip bit 7
~a	bitwise complement	PORTA &= ~0x80; // turn off bit 7

Figure 2.2: Bit twiddling (ImageCraft).

2.1.5 ATMEGA164 ARCHITECTURE OVERVIEW

We have chosen the ATmega164 as a representative of the Atmel AVR line of microcontrollers. Lessons learned with the ATmega164 may be easily adapted to all other processors in the AVR line. A block diagram of the Atmel ATmega164's architecture is shown in Figure 2.3.

As can be seen from the figure, the ATmega164 has external connections for power supplies (VCC, GND), external time base input pins to drive its clocks (XTAL1 and XTAL2), processor reset (active low RESET), and four 8-bit ports (PA0-PA7, PB0-PB7, PC0-PC7, and PD0-PD7), which are use to interact with the external world. As we shall soon see, these ports, may be used as general purpose digital input/output ports, or they may be used for the alternate functions. The ports are interconnected with the ATmega164's CPU and internal subsystems via internal buses. The ATmega164 also contains a timer subsystem, an analog-to-digital converter, an analog comparator, an interrupt subsystem, a timing subsystem, memory components, and extensive serial communication features (Atmel).

In the next several subsections, we briefly describe each of these internal subsystems shown in the figure. Detailed descriptions of major subsystem operation and programming are provided later in this book. We can not cover all features of the microcontroller due to limited space.

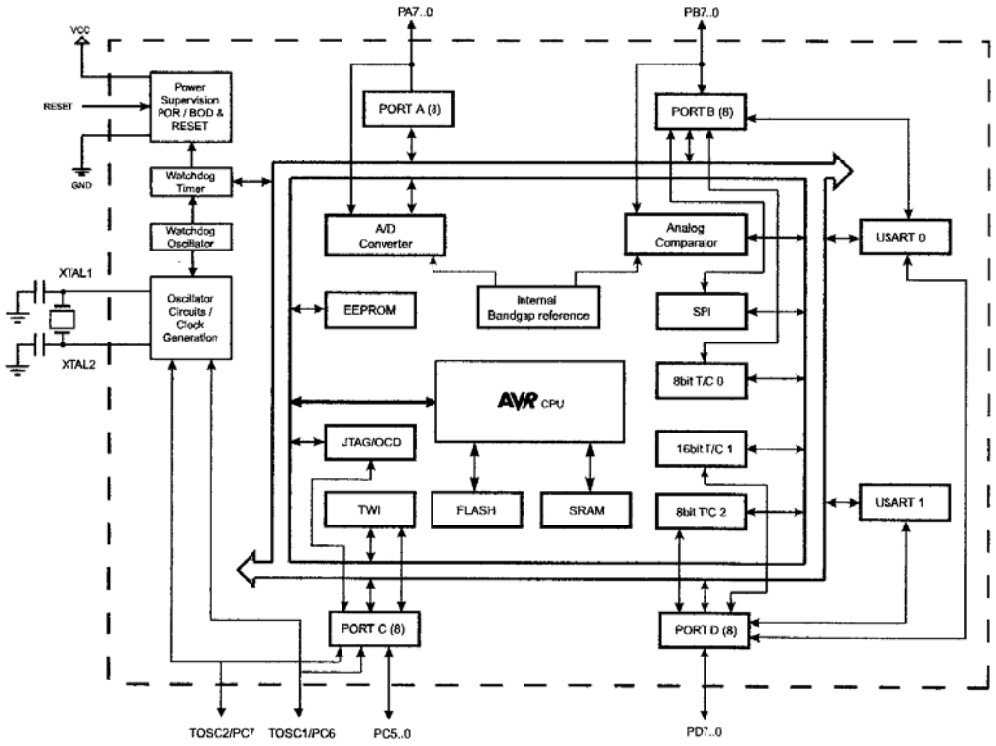


Figure 2.3: Atmel AVR ATmega164 Block Diagram. (Figure used with permission of Atmel, Incorporated.)

2.2 NONVOLATILE AND DATA MEMORIES

The ATmega164 is equipped with three main memory sections: flash electrically erasable programmable read only memory (EEPROM), byte-addressable EEPROM for data storage, and static random access memory (SRAM). We discuss each memory component in turn, next.

2.2.1 IN-SYSTEM PROGRAMMABLE FLASH EEPROM

Flash EEPROM is nonvolatile: memory contents are retained when microcontroller power is lost. Flash EEPROM is used to store programs. It is called bulk programmable since it can be erased and programmed as a single unit. Also, should a program require a large table of constants, it may be included as a global variable within a program and programmed into flash EEPROM with the rest of the program. The ATmega164 is equipped with 16K bytes of onboard reprogrammable flash memory. This memory component is organized into 8K locations with 16 bits at each location.

The flash EEPROM is in-system programmable. In-system programmability means the microcontroller can be programmed while resident within a circuit. It does not have to be removed from the circuit, as was the case for older EEPROM technologies, for programming. Instead, a host personal computer (PC) connected via a cable to a microcontroller downloads the program directly to the microcontroller. Alternately, the microcontroller can be programmed outside of its resident circuit using a flash programmer board. We will use this technique throughout the book since many of our examples will use the microcontroller pins required for ISP programming. Specifically, we will use the Atmel STK500 AVR Flash MCU Starter Kit for programming the ATmega164. This inexpensive development board (less than $100) is readily available from a number of suppliers.

The flash EEPROM is used to bulk store application programs for execution. Another type of onboard EEPROM is the byte-addressable type. It is used to store variables needed by the system even in the event of a power failure. We discuss this type of memory next.

2.2.2 BYTE-ADDRESSABLE EEPROM

Byte-addressable memory is used to permanently store and recall variables during program execution. It, too, is nonvolatile. It is especially useful for logging system malfunctions and fault data during program execution. It is also useful for storing data that must be retained during a power failure but might need to be changed periodically. Examples where this type of memory is used are found in applications to store system parameters, electronic lock combinations, and automatic garage door electronic unlock sequences. The ATmega164 is equipped with 512 bytes of EEPROM.

2.2.3 ACCESSING BYTE-ADDRESSABLE EEPROM EXAMPLE

In this section, we describe how to read from and write to the byte-addressable EEPROM aboard the ATmega164. As previously mentioned, the features of the ImageCraft ICC AVR compiler will be used to access the memory. Other compilers have similar features.

To configure a program to access byte-addressable EEPROM memory, the following steps must be accomplished:

1. The compiler specific support function definitions for accessing EEPROM must be included within the main C program. Using the ImageCraft ICC AVR compiler, this is accomplished by including the header file **eeprom.h** as shown below:

```
//EEPROM support functions
#include<eeprom.h>
```

2. The variables to be placed in EEPROM must be declared as global variables. For the ImageCraft ICC AVR compiler, this is accomplished using the #pragma construct illustrated below. Other variables to be stored in EEPROM would be included between the two #pragma statements. Also, a normal global variable is declared to correspond to the EEPROM based variable to facilitate reading from and writing to the variable from the EEPROM location. In the fol-

lowing example, the variable residing in EEPROM is called "test_data_EEPROM" while the corresponding variable used within the program is called "test_data."

```
#pragma data: eeprom
unsigned int test_data_EEPROM = 0x1234;
#pragma data:data

//global variables
unsigned int test_data;
```

3. The EEPROM variable may then be written to, or read from, a byte-addressable EEPROM memory location using the following ICC AVR compiler specific function calls.

```
EEPROM_READ((int) &test_data_EEPROM, test_data);

EEPROM_WRITE((int)&test_data_EEPROM, test_data);
```

When a program containing byte-addressable EEPROM variables is compiled, a $< *.eep >$ file is generated in addition to the $< *.hex >$ file. The $< *.eep >$ file is used to, initially, configure the EEPROM when the ATmega164 is programmed using the STK500. The STK500 may also be used to read the contents of the EEPROM.

2.2.4 STATIC RANDOM ACCESS MEMORY (SRAM)

Static RAM memory is volatile. That is, if the microcontroller loses power, the contents of SRAM memory are lost. It can be written to, and read from, during program execution. The ATmega164 is equipped with 1K (actually 1120) bytes of SRAM. A small portion (96 locations) of the SRAM is set aside for the general purpose registers used by the CPU and also for the input/output and peripheral subsystems aboard the microcontroller. A complete ATmega164 register listing and accompanying header file are provided in Appendices A and B, respectively. During program execution, RAM is used to store global variables, support dynamic memory allocation of variables, and to provide a location for the stack.

2.2.5 PROGRAMMABLE LOCK BITS

To provide users with memory security from tampering, the ATmega164 is equipped with six memory lock bits. These lock bits are programmed using the Atmel STK500 programming board. The lock bits may be configured for the following options:

- No memory lock features enabled,
- No further programming of memory is allowed using parallel or serial programming techniques, or

- No further programming or verification of memory is allowed using parallel or serial programming techniques.

The machine code contents of Flash or byte-addressable EEPROM memory may be read using an STK500 and the AVR Studio software. If one tries to read the contents of Flash memory with the lock bits set for the highest level of security, default memory values are displayed.

2.3 PORT SYSTEM

The Atmel ATmega164 has four, 8-bit general purpose, digital input/output (I/O) ports designated PORTA, PORTB, PORTC, and PORTD. All of these ports also have alternate functions, which will be described later. In this section, we concentrate on the basic digital I/O port features.

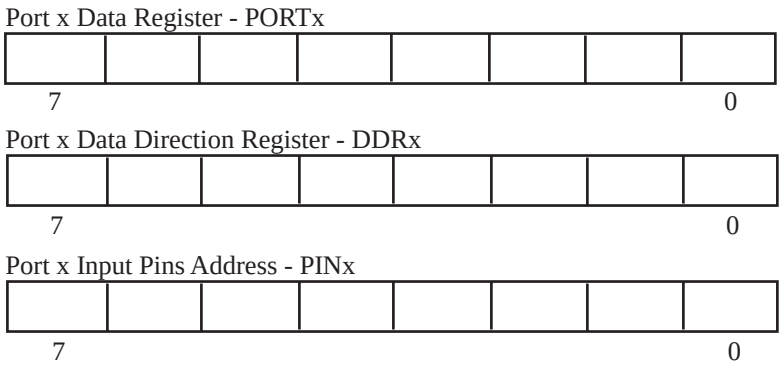
As shown in Figure 2.4, each port has three registers associated with it.

- Data Register PORTx —- used to write output data to the port.
- Data Direction Register DDRx —- used to set a specific port pin to either output (1) or input (0).
- Input Pin Address PINx —- used to read input data from the port.

Figure 2.4(b) describes the settings required to configure a specific port pin to either input or output. If selected for input, the pin may be selected for either an input pin or to operate in the high impedance (Hi-Z) mode. In Hi-Z mode, the input appears as high impedance to a particular pin. If selected for output, the pin may be configured further for either logic low or logic high.

Port pins are usually configured at the beginning of a program for either input or output, and their initial values are then set. Usually, all eight pins for a given port are configured simultaneously. A code example is provided below to show how ports are configured. Note that since we are using the C programming language with a compiler include file, the register contents are simply referred to by name. Note that the data direction register (DDRx) is first used to set the pins as either input or output, and then the data register (PORTx) is used to set the initial value of the output port pins. Any pins not used in a specific application should be configured as output pins since an unused pin configured as an input can be a source for noise to enter the processor.

```
//*********************************************************************
//initialize_ports: initial configuration for I/O ports
//*********************************************************************

void initialize_ports(void)
{
DDRA=0xfc;  //set PORTA[7:2] as output, PORTA[1:0] as input (1111_1100)
PORTA=0x03; //initialize PORTA[7:2] low
```



a) port associated registers

DDxn	PORTxn	I/O	Comment	Pullup
0	0	input	Tri-state (Hi-Z)	No
0	1	input	source current if externally pulled low	Yes
1	0	output	Output Low (Sink)	No
1	1	output	Output High (Source)	No

x: port designator (A, B, C, D)
n: pin designator (0 - 7)

b) port pin configuration

Figure 2.4: ATmega164 port configuration registers.

```
DDRB=0xa0;  //PORTB[7:4] as output, set PORTB[3:0] as input
PORTB=0x00; //disable PORTB pull-up resistors

DDRC=0xff;  //set PORTC as output
PORTC=0x00; //initialize low

DDRD=0xff;  //set PORTD as output
PORTD=0x00; //initialize low
}
```

To read the value from a port pin configured as input, the following code could be used. Note the variable used to read the value from the input pins is declared as an unsigned char since both the port and this variable type are eight bits wide.

```
unsigned char   new_PORTB;            //new values of PORTB

:
:
:

new_PORTB = PINB;                     //read PORTB
```

2.4 PERIPHERAL FEATURES—INTERNAL SUBSYSTEMS

In this section, we provide a brief overview of the peripheral features of the ATmega164. It should be emphasized that these features are the internal subsystems contained within the confines of the microcontroller chip. These built-in features allow complex and sophisticated tasks to be accomplished by the microcontroller.

2.4.1 TIME BASE

The microcontroller is a complex synchronous state machine. It responds to program steps in a sequential manner as dictated by a user-written program. The microcontroller sequences through a predictable fetch-decode-execute sequence. Each program instruction issues a series of signals to control the microcontroller hardware to accomplish instruction related operations.

The speed at which a microcontroller sequences through these actions is controlled by a precise time base called the clock. The clock source is routed throughout the microcontroller to provide a time base for all peripheral subsystems. The ATmega164 may be clocked internally using a user-selectable resistor capacitor (RC) time base, or it may be clocked externally. The RC internal time base is selected using programmable fuse bits. We will discuss how to do this in the application section of this chapter. You may choose an internal fixed clock operating frequency of 1, 2, 4 or 8 MHz.

To provide for a wider range of frequency selections, an external time source may be used. The external time sources, in order of increasing accuracy and stability, are an external RC network, a ceramic resonator, and a crystal oscillator. The system designer chooses the time base frequency and clock source device appropriate for the application at hand. As previously mentioned, the maximum operating frequency of the ATmega164P with a 5 VDC supply voltage is 20 MHz.

2.4.2 TIMING SUBSYSTEM

The ATmega164 is equipped with a complement of timers which allows a user to generate a precision output signal, measure the characteristics (period, duty cycle, frequency) of an incoming digital

signal, generate precision delays, implement a real time clock, or count external events. Specifically, the ATmega164 is equipped with two 8-bit timer/counters and one 16-bit counter. We discuss the operation, programming, and application of the timing system in Chapter 6.

2.4.3 PULSE WIDTH MODULATION CHANNELS

A pulse width modulated or PWM signal is characterized by a fixed frequency and a varying duty cycle. A duty cycle is defined as the percentage of time a periodic signal is logic high over the signal period. It may be formally expressed as:

$$duty\ cycle[\%] = (on\ time/period) \times (100\%)$$

The ATmega164 is equipped with three separate timers to support six channels of pulse width modulation (PWM) operation. The PWM channels coupled with the flexibility of dividing the time base down to different PWM subsystem clock source frequencies allow a user to configure the microcontroller to generate a wide variety of PWM signals: from relatively high frequency low duty cycle signals to relatively low frequency high duty cycle signals.

PWM signals are used in a wide variety of applications including controlling the position of a servo motor, DC motor speed control, or generating an analog DC output signal. We discuss the operation, programming, and application of the PWM system in Chapter 6.

2.4.4 SERIAL COMMUNICATIONS

The ATmega164 is equipped with a host of different serial communication subsystems including the Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART), the serial peripheral interface (SPI), and the Two-wire Serial Interface. What all of these systems have in common is the serial transmission of data. In a serial communications transmission scheme, data are sent a single bit at a time from transmitter to receiver.

2.4.4.1 Serial USART

The serial USART is used for full duplex (two way) communication between a receiver and a transmitter. This is accomplished by equipping the ATmega164 with independent hardware for the transmitter and receiver pair. The USART is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and the receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence.

The ATmega164 USART is quite flexible. It has the capability to be set to a variety of data transmission rates known as the Baud (bits per second) rate. The USART may also be set for data bit widths of 5 to 9 bits with one or two stop bits. Furthermore, the ATmega164 is equipped with a hardware generated parity bit (even or odd) and parity check hardware at the receiver. A single parity bit allows for the detection of a single bit error within a byte of data. The USART may also be configured to operate in a synchronous mode. We discuss the operation, programming, and application of the USART in Chapter 3.

2.4.4.2 Serial Peripheral Interface—SPI

The ATmega164 Serial Peripheral Interface (SPI) can also be used for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and the receiver must share a common clock source. This requires an additional clock line between the transmitter and the receiver but allows for higher data transmission rates as compared to the USART.

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit half residing in the transmitter and the other 8-bit half residing in the receiver. The transmitter is designated as the master since it is providing the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. We discuss the operation, programming, and application of the SPI in Chapter 3.

2.4.4.3 Two-wire Serial Interface—TWI

The TWI subsystem allows the system designer to network a number of related devices (microcontrollers, transducers, displays, memory storage, etc.) together into a system using a two wire interconnecting scheme. The TWI allows a maximum of 128 devices to be connected together. Each device has its own unique address and may both transmit and receive over the two wire bus at frequencies up to 400 kHz. This allows the device to freely exchange information with other devices in the network within a small area (Atmel).

2.4.5 ANALOG TO DIGITAL CONVERTER—ADC

To interact with external analog signals, the ATmega164 is equipped with an eight channel analog to digital converter (ADC) subsystem. The ADC converts an analog signal from the outside world into a binary representation suitable for use by the microcontroller. The ATmega164 ADC has 10 bit resolution, which means that an analog voltage between 0 and 5 will be encoded into one of 1024 binary representations between $(000)_{16}$ and $(3FF)_{16}$. This provides the ATmega164 with a voltage resolution of approximately 4.88 mV. We discuss the operation, programming, and application of the ADC in Chapter 4 (Atmel).

2.4.6 ANALOG COMPARATOR

The ATmega164 is equipped with an analog comparator. An analog comparator may be used as a threshold detector or used to clean up a corrupted digital signal. The analog comparator has two external inputs designated AIN0 and AIN1. When the voltage at AIN0 is greater than the voltage at AIN1, the output of the comparator, AC0, is set to logic one (Atmel).

2.4.7 INTERRUPTS

The normal execution of a program is performed by following a designated sequence of instruction execution. However, sometimes this normal sequence of instructions must be interrupted to respond to higher priority events inside and outside the microcontroller. When these higher priority events occur, the microcontroller must temporarily suspend normal operation and execute event specific

actions called an interrupt service routine. Once the higher priority event has been serviced, the microcontroller returns and continues to operate normally.

Interrupts may also be employed to continuously monitor the status of an ongoing event while the microcontroller is processing other activities. For example, a microcontroller controlling access for an electronic door must monitor input commands from a user and generate the appropriate PWM signal to open and close the door. Once the door is in motion, the controller must monitor door motor operation for obstructions, malfunctions, and other safety related parameters. This may be accomplished using interrupts. In this example, the microcontroller is responding to user input status in the foreground while monitoring safety related status in the background using interrupts as illustrated in Figure 2.5.

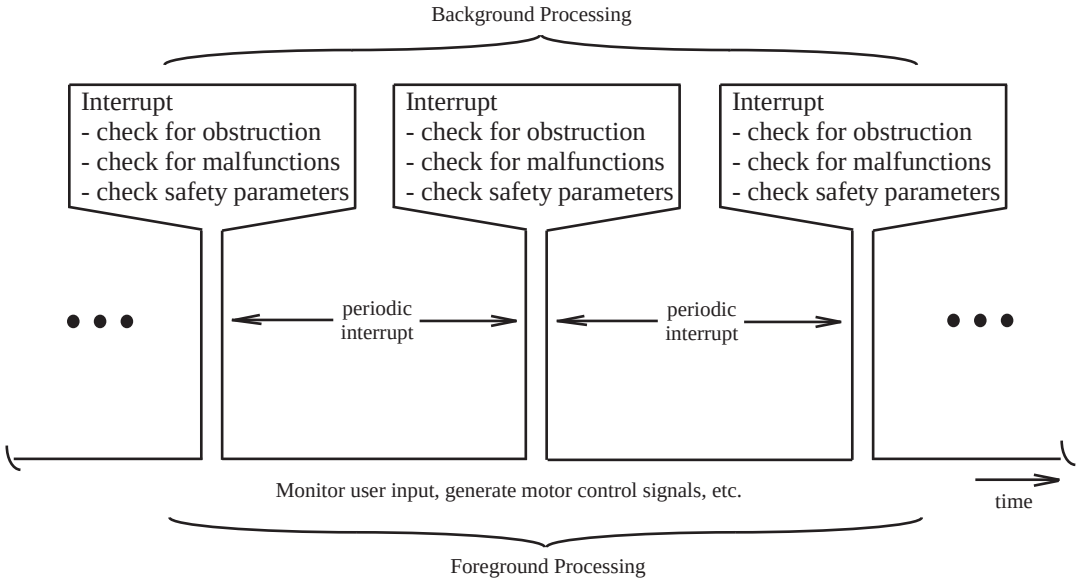


Figure 2.5: Interrupt used for background processing. The microcontroller responds to user input status in the foreground while monitoring safety related status in the background using interrupts.

The ATmega164 is equipped with mechanisms to detect a complement of 31 interrupt sources. Three of the interrupts are provided for external interrupt sources while the remaining 29 interrupts support the efficient operation of peripheral subsystems aboard the microcontroller (Atmel). We discuss the operation, programming, and application of the interrupt system in Chapter 5.

2.5 PHYSICAL AND OPERATING PARAMETERS

In this section, we provide data on the physical layout and operating parameters of the ATmega164 microcontroller. As a system designer, it is important to know the various physical and operating parameter options available to select the best option for a given application.

2.5.1 PACKAGING

The ATmega164 comes in a variety of different packaging styles: a 40-pin plastic dual in line package (PDIP), a 44-lead thin quad flat pack (TQFP) package, a 44-pad quad flat non-lead/micro lead frame (QFN/MLF) package, and a 44 pin DRQFN package. The pinout diagram for the PDIP and the TQFP/QFN/MLF packaging options are shown in Figure 2.6 (Atmel).

2.5.2 POWER CONSUMPTION

The ATmega164 operates at supply voltages from 1.8 to 5.5 VDC. In the application examples that follow, we will use a standard laboratory 5 VDC power supply but also discuss a method of providing a 5 VDC supply using an off-the-shelf 9 VDC battery. The current draw for the microcontroller is quite low. For example, when the ATmega164 is actively operating at 1 MHz from a 1.8 VDC power source, the current draw is 0.4 mA. When placed in the power-down mode, the microcontroller will draw less than 0.1 microamp of current from the voltage source.

2.5.3 SPEED GRADES

The ATmega164 operates with a clock speed from 0 to 20 MHz. As previously discussed, the operating speed of the microcontroller is set by the time base chosen for the processor. Since faster microcontroller operation is not always the best option, the system designer must determine the optimal speed of microcontroller operation for a given application. The microcontroller's power consumption is directly related to operating speed. That is, the faster the operating speed of the microcontroller the higher its power consumption. This becomes especially critical in portable, battery-operated embedded systems applications.

That completes our brief introduction to the features of the ATmega164. In the next section, we discuss how to choose a specific microcontroller for a given application.

2.6 CHOOSING A MICROCONTROLLER

To choose a specific microcontroller for a given application, the following items must be considered:

- The microcontroller must have the specific subsystems required by the application. For example, does the application require analog-to-digital conversion, serial communications, etc.?
- How many digital input/output pins are required by the controller?
- How much Flash EEPROM memory is required by the application?

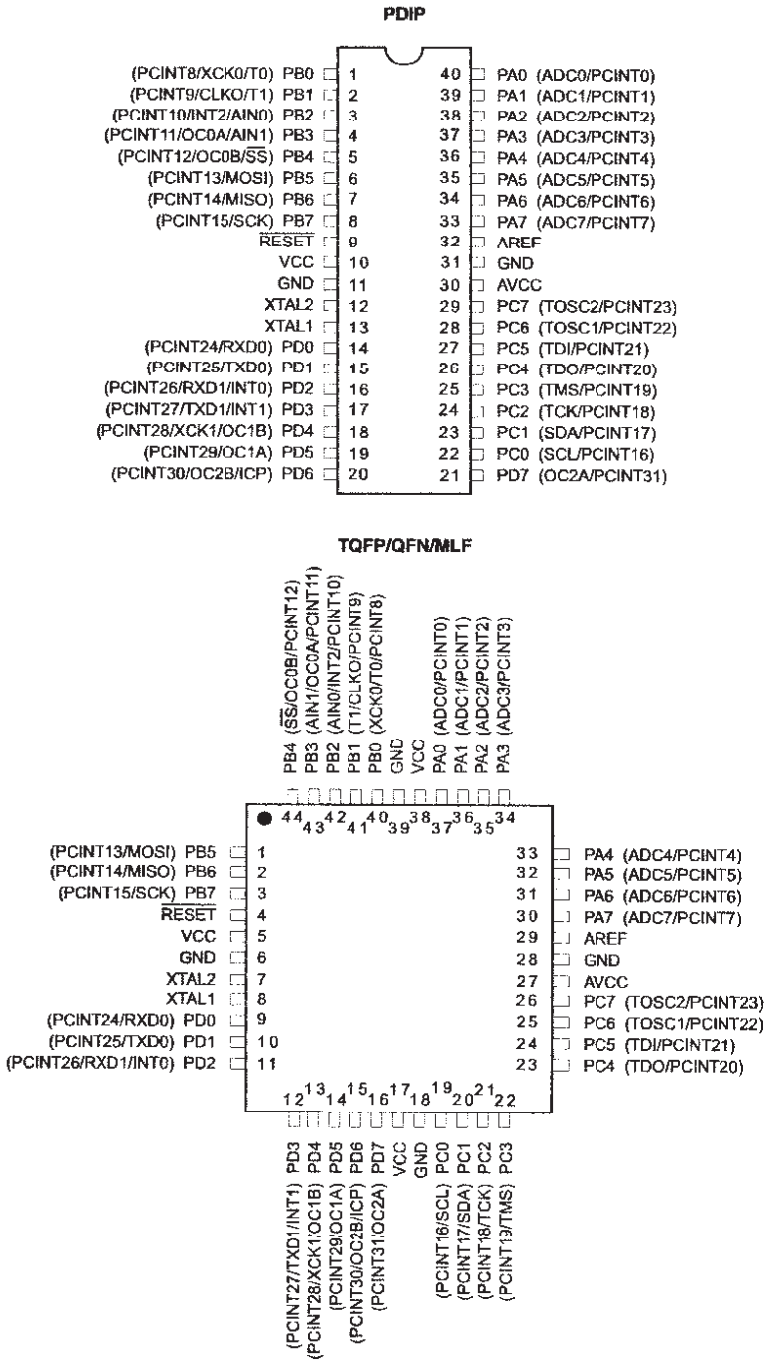


Figure 2.6: Atmel AVR ATmega164 Pinout Diagram: a) 40-pin PDIP, b) TQFP/QFN/MFL (Figure used with permission of Atmel, Incorporated.)

To answer these questions, it is helpful to construct a diagram illustrating all peripheral devices and their connection to the microcontroller. This will help to determine the required microcontroller subsystems and the number of digital input/output pins.

The memory question is more difficult to answer. How can you determine the size of Flash EEPROM required for a specific application before you write the code for the application? In this case, you can choose a family of pin-for-pin compatible microcontrollers such as the ATmega164/324/644. These microcontrollers have similar features and have memory assets of 16K, 32K and 64K, respectively.

An application can be developed starting with the 16K microcontroller. As the application evolves, one can transition to the 32K and 64K as required. When transitioning to another pin-for-pin compatible microcontroller, it is essential to use the specific header file for the microcontroller and also ensure that the interrupt numbers have not changed.

2.7 APPLICATION: ATMEGA164 TESTBENCH

The purpose of the Testbench is to illustrate the operation of selected ATmega164 subsystems working with various input and output devices. Most importantly, the Testbench will serve as a template to develop your own applications.

In this section, we provide the hardware configuration of a barebones Testbench and a basic software framework to get the system up and operating. We will connect eight debounced tact switches to PORTB and an eight channel tri-state light emitting diode (LED) array to PORTA. The software will check for a status change on PORTB. When the user depresses one of the tact switches, the ATmega164 will detect the status change, and the corresponding LED on PORTA will transition from red to green.

2.7.1 HARDWARE CONFIGURATION

Provided in Figure 2.7 is the basic hardware configuration for the Testbench. We will discuss in detail the operation of the input and output devices in Chapter 7.

PORTA is configured with eight tact (momentary) switches with accompanying debouncing hardware. We discuss the debounce circuit in detail in Chapter 7. PORTB is equipped with an eight channel tri-state LED indicator. For a given port pin, the green LED will illuminate for a logic high, the red LED for a logic low, and no LEDs for a tri-state high-impedance state. We discuss this circuit in detail in Chapter 7.

Aside from the input hardware on PORTB and the output display hardware on PORTA of the controller, there are power (pins 10, 30, 32) and ground (pins 11, 31) connections. A standard 5 VDC power supply may be used for the power connections. For portable applications, a 9 VDC battery equipped with a 5 VDC regulator (LM340-05 or uA7805) may be used as a power source. Pins 9 through 11 have a resistor (1M), two capacitors (1.0 uF), and a tact switch configured to provide a reset switch for the microcontroller. We use a ZTT 10 MHz ceramic resonator as the time base for the Testbench. It is connected to pins 12 (XTAL2) and 13 (XTAL1) of the ATmega164.

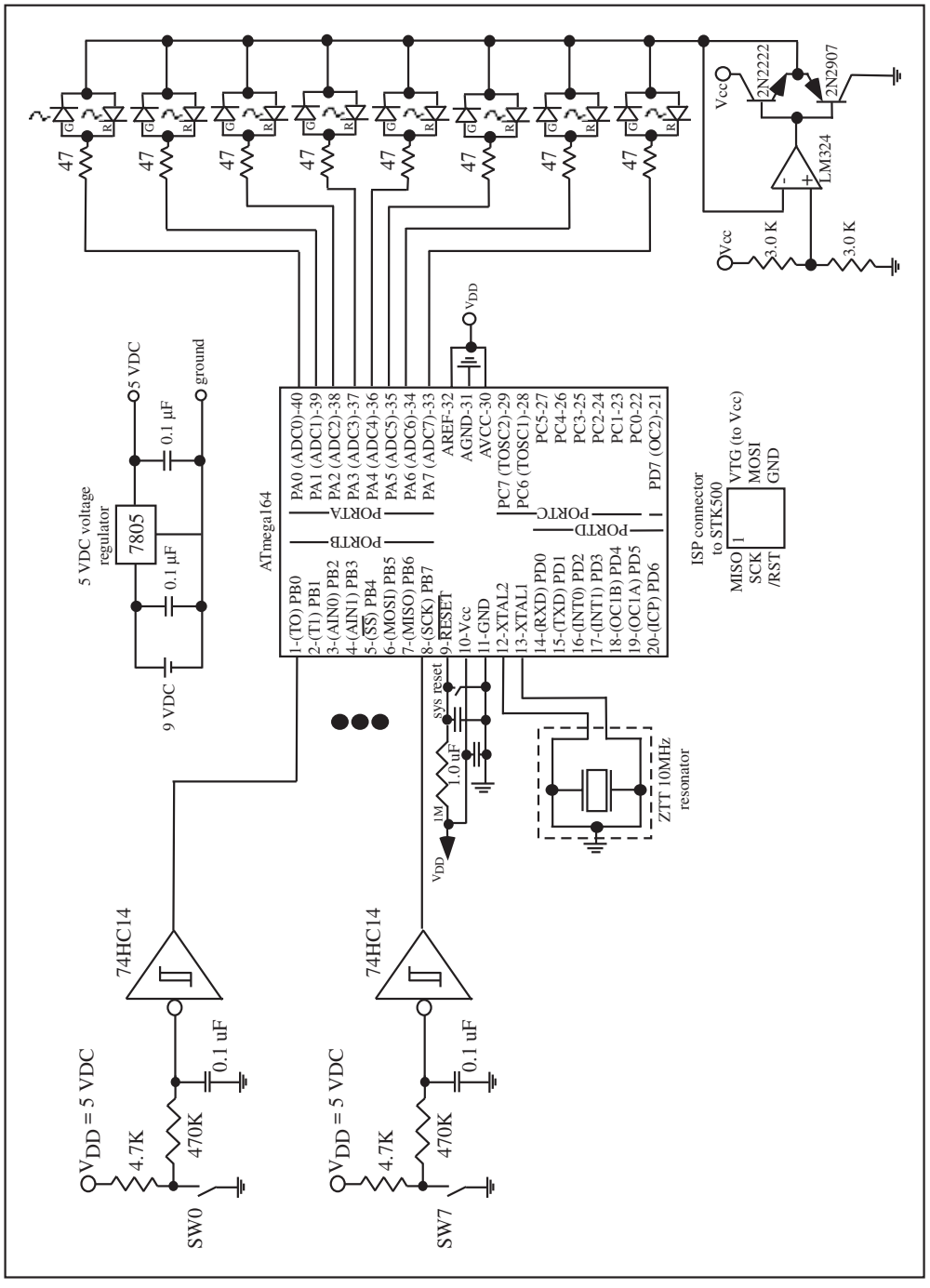


Figure 2.7: ATmega164 Testbench Hardware.

2.7.2 SOFTWARE CONFIGURATION

The Testbench software is provided below. The program contains the following sections:

- Comments.
- Include files —- We have included the ImageCraft ICC AVR include file for the ATmega164 (iom164v.h). This file provides the software link between the names of the ATmega164 hardware registers and the actual hardware locations. When a register is used by name in the program, reference is made to the contents of that register.
- Function prototypes.
- Global variables.
- Main program—We begin the main program by calling the function to initialize the ports and then enter a continuous loop. Within the loop body the ATmega164 monitors for a status change on PORTB. When the user depresses one of the tact switches connected to PORTB, a change of status is detected and the appropriate LED is illuminated on PORTA.
- Function definition.

```
//*************************************************************************
//file name: testbench.c
//function: provides test bench for ATMEL AVR ATmega164 controller
//target controller: ATMEL ATmega164
//
//ATMEL AVR ATmega164 Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin 1 PB0 to active high RC debounced switch
//Pin 2 PB1 to active high RC debounced switch
//Pin 3 PB2 to active high RC debounced switch
//Pin 4 PB3 to active high RC debounced switch
//Pin 5 PB4 to active high RC debounced switch
//Pin 6 PB5 to active high RC debounced switch
//Pin 7 PB6 to active high RC debounced switch
//Pin 8 PB7 to active high RC debounced switch
//Pin 9 Reset
//Pin 10 VDD
//Pin 11 Gnd
//Pin 12 Resonator
//Pin 13 Resonator
//Pin 14 PD0 to tristate LED indicator
//Pin 15 PD1 to tristate LED indicator
```

```
//Pin 16 PD2 to tristate LED indicator
//Pin 17 PD3 to tristate LED indicator
//Pin 18 PD4 to tristate LED indicator
//Pin 19 PD5 to tristate LED indicator
//Pin 20 PD6 to tristate LED indicator
//Pin 21 PD7 to tristate LED indicator
//Pin 22 PC0
//Pin 23 PC1
//Pin 24 PC2
//Pin 25 PC3
//Pin 26 PC4
//Pin 27 PC5
//Pin 28 PC6
//Pin 29 PC7
//Pin 30 AVcc to VDD
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//Pin 33 PA7
//Pin 34 PA6
//Pin 35 PA5
//Pin 36 PA4
//Pin 37 PA3
//Pin 38 PA2
//Pin 39 PA1
//Pin 40 PA0
//***********************************************************************

//include files*****************************************************
#include<iom164v.h>                    //ImageCraft ICC AVR include
                                       //file for ATmega164

//function prototypes***********************************************
void initialize_ports(void);           //initialize ports


//main program******************************************************

//global variables
unsigned char   old_PORTB = 0x00;      //present value of PORTB
```

```
unsigned char   new_PORTB;              //new values of PORTB

void main(void)
{
initialize_ports();                     //initialize ports

while(1){//main loop
  new_PORTB = PINB;                     //read PORTB
  if(new_PORTB != old_PORTB){           //process change in PORTB input
    switch(new_PORTB){                  //PORTB asserted high
           case 0x01:                   //PB0 (0000_0001)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x01;                //turn on PD0 LED (0000_0001)
             break;

           case 0x02:                   //PB1 (0000_0010)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x02;                //turn on PD1 LED (0000_0010)
             break;

           case 0x04:                   //PB2 (0000_0100)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x04;                //turn on PD2 LED (0000_0100)
             break;

           case 0x08:                   //PB3 (0000_1000)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x08;                //turn on PD3 LED (0000_1000)
             break;

           case 0x10:                   //PB4 (0001_0000)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x10;                //turn on PD4 LED (0001_0000)
             break;

           case 0x20:                   //PB5 (0010_0000)
             PORTD=0x00;                //turn off all LEDs PORTD
             PORTD=0x20;                //turn on PD5 LED (0010_0000)
             break;
```

```
            case 0x40:                  //PB6 (0100_0000)
              PORTD=0x00;               //turn off all LEDs PORTD
              PORTD=0x40;               //turn on PD6 LED (0100_0000)
              break;

            case 0x80:                  //PB7 (1000_0000)
              PORTD=0x00;               //turn off all LEDs PORTD
              PORTD=0x80;               //turn on PD7 LED (1000_0000)
              break;

            default:;                   //all other cases
            }                           //end switch(new_PORTB)
          }                             //end if new_PORTB
      old_PORTB=new_PORTB;              //update PORTB
  }                                     //end while(1)
}                                       //end main

//***********************************************************************
//function definitions
//***********************************************************************

//***********************************************************************
//initialize_ports: initial configuration for I/O ports
//***********************************************************************

void initialize_ports(void)
{
DDRA=0xff;                              //set PORTA[7:0] as output
PORTA=0x00;                             //initialize PORTA[7:0] low

DDRB=0x00;                              //PORTB[7:0] as input
PORTB=0x00;                             //disable PORTB pull-up resistors

DDRC=0xff;                              //set PORTC as output
PORTC=0x00;                             //initialize low

DDRD=0xff;                              //set PORTD as output
PORTD=0x00;                             //initialize low
```

```
}
//*********************************************************************
```

2.8 PROGRAMMING THE ATMEGA164

Programming the ATmega164 requires several hardware and software tools. We briefly mention required components here. Please refer to the manufacturer's documentation for additional details at www.atmel.com.

Software Tools: Throughout the text, we use the ImageCraft ICC AVR compiler. This is a broadly used, user-friendly compiler. There are other excellent compilers available. A listing is provided in Appendix C. The compiler is used to translate the source file (testbench.c) into machine language for loading into the ATmega164. We use Atmel's AVR Studio to load the machine code into the ATmega164.

Hardware Tools: We use Atmel's STK500 AVR Flash MCU Starter Kit (STK500) for programming the ATmega164. The STK500 provides the interface hardware between the host PC and the ATmega164 for machine code loading. The STK500 is equipped with a complement of DIP sockets, which allows for programming all of the microcontrollers in the Atmel AVR line. The STK500 also allows for In-System Programming (ISP) (Atmel).

2.8.1 PROGRAMMING PROCEDURE

In this section, we provide a step-by-step procedure to program the ATmega164 dual inline package (PDIP) using the STK500 AVR Flash MCU Starter Kit. Please refer to Figure 2.8. It should be noted that the STK500 will be used to program the microcontroller, which will then be placed in the Testbench circuit.

1. Load AVR Studio (free download from www.atmel.com).
2. Ensure that the STK500 is powered down.
3. Connect the STK500 as shown in Figure 2.8.
4. Insert the ATmega164 into the red 40 pin socket. Note the location of pin1 in Figure 2.8.
5. Power up the STK500.
6. Start up AVR Studio on your PC.
7. Pop up window "Welcome to AVR Studio" should appear. Close this window by clicking on the "Cancel button."
8. Click on the "AVR icon." It looks like the silhouette of an integrated circuit. It is on the second line of the toolbar about half way across the screen.

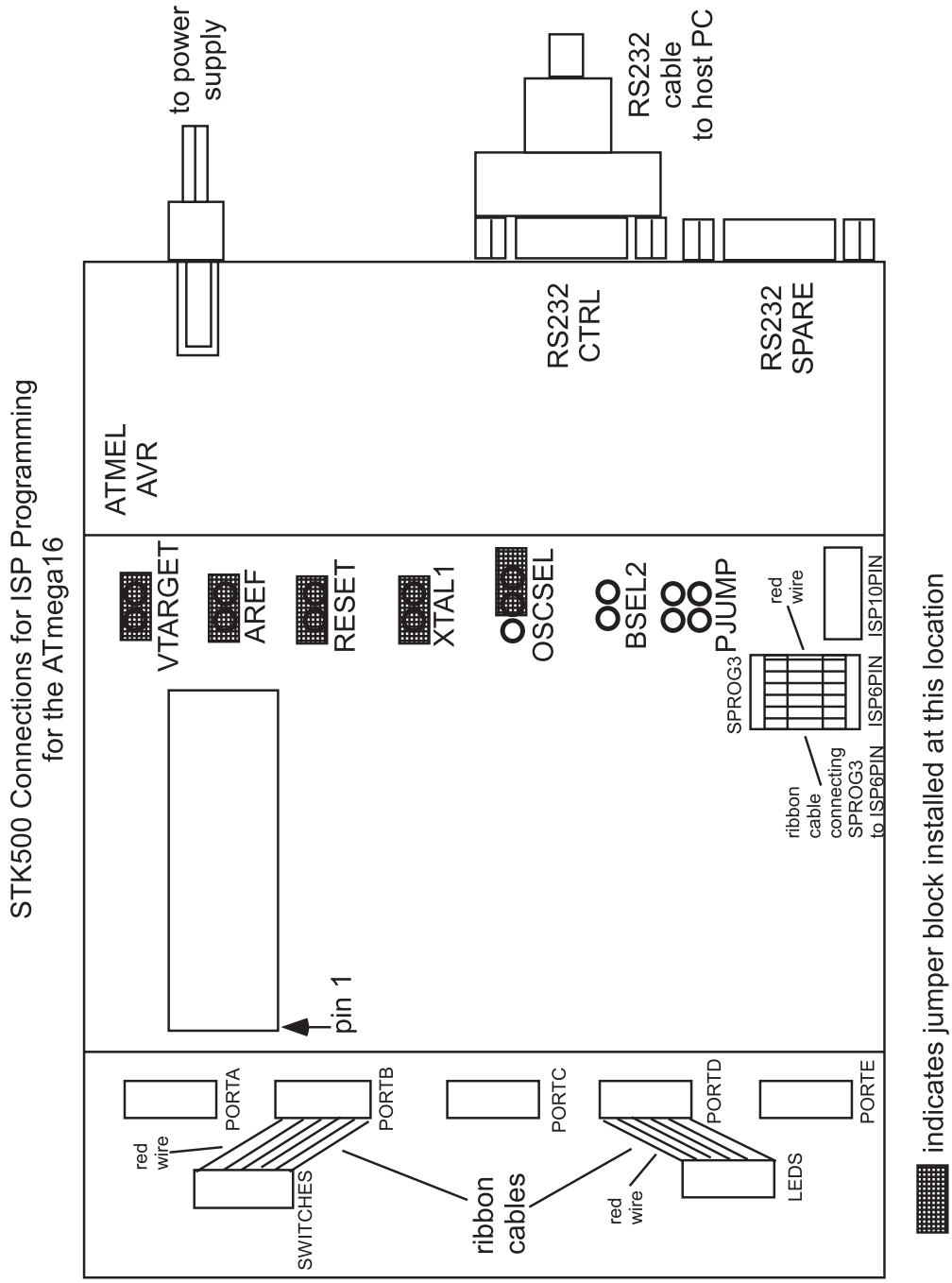


Figure 2.8: Programming the ATmega164 with the STK500.

9. This should bring up a STK500 pop up window with six tabs (Program, Fuses, Lockbits, Advanced, Board, Auto). At the bottom of the Program tab window verify that the STK500 was autodetected. Troubleshoot as necessary to ensure STK500 was autodetected by AVR Studio.

10. Set all tab settings:

    - Program:
      - Select device: ATmega164
      - Programming mode: ISP, Erase Device Before Programming, Verify Device After Programming
      - Flash: Input HEX file, Browse and find machine code file: <yourfilename.hex>
      - EEPROM: Input HEX file, Browse and find machine code file: <yourfilename.EEP>
    - Fuses: Set the following fuses
      - Boot flash section size = 128
      - Brown out detection at Vcc = 4.0V
      - External Crystal/Resonator High Frequency; Start-up time 16K CK + 64ms
    - Lock bits:
      - Mode 1
      - Application Protection Mode 1
      - Boot loader Protection Mode 1
      - Advanced: N/A
      - Board: N/A
    - Auto:
      - Erase Device
      - Program Flash
      - Verify Flash
      - Program Fuses
      - Verify Fuses
      - Read Fuses

11. Programming step:

    - Program Tab: click program
    - Fuse Tab: click program (don't forget this step; otherwise processor runs very slow!)

12. Power down the STK500. Remove the programmed chip from the STK500 board and place it in the Testbench circuit.

2.9 IN-SYSTEM PROGRAMMING (ISP)

ISP programming allows programming of the microcontroller without it being removed from the circuit. To program using ISP procedures, the ISP6PIN connector on the STK500 is connected to the MISO, SCK, RST, and MOSI pins on the target microcontroller. The VTG pin on the ISP6PIN is connected to Vcc and the GND pin to the circuit ground. The ISP6PIN is illustrated in Figure 2.8.

2.10 SOFTWARE PORTABILITY

The software techniques discussed in the textbook are based on the ATmega164; however, the developed software may be easily ported for use with other Atmel AVR microcontrollers. To ease the transition to another microcontroller, it is suggested using a direct bit assignment technique. Instead of setting an entire register content at once, selected bits may be set. The individual bit definitions for the ATmega164 are provided in Appendix B.

For example, to set the UCSRB register, the following individual bit assignments may be used:

```
UCSRB = (1<<TXEN)|(1<<RXEN);          //Enable transmit and receive
```

as opposed to:

```
UCSRB = 0x08;                         //Enable transmit and receive
```

When transporting code, the header file for the specific microcontroller must be used. Also, the interrupt vector numbers may require change.

2.11 SUMMARY

In this chapter, we provided a brief overview of the ATmega164 microcontroller, a representative sample of the AVR microcontrollers. Information presented in this chapter can be readily applied to other microcontrollers in the AVR line. We then provided the Testbench hardware and software that we use throughout the text to illustrate peripheral subsystem operation aboard the ATmega164. In rest of the book, we provide additional details on selected ATmega164 subsystems.

2.12 CHAPTER PROBLEMS

2.1. What is a RISC processor?

2.2. How does the ATmega164 interact with the external world?

2.3. What are the different methods of applying a clock source to the ATmega164? List the inherent advantages of each type.

2.4. Describe the three different types of memory components aboard the ATmega164. What is each used for?

2.5. Describe the three registers associated with each input/output port.

2.6. What is the advantage of activating pull-up resistors with the digital input pins?

2.7. With a specific port selected, can some port pins be declared as output pins while others as input pins? If so, describe how this may be accomplished.

2.8. Describe the serial communication features aboard the ATmega164. Provide a suggested application for each.

2.9. What is the maximum possible bit rate with the USART subsystem? With the SPI?

2.10. What is the purpose of the ADC system? Describe an application for this system.

2.11. What is the purpose of the interrupt system? Describe an application for this system.

2.12. What is the purpose of the PWM system? Describe an application for this system.

2.13. What is the best clock speed at which to operate the ATmega164 for a specific application?

2.14. Sketch a flow chart or UML activity diagram for the testbench.c program

2.15. What is the purpose of the analog comparator subsystem? Describe an application for this system.

2.16. You have run out of Flash EEPROM memory when developing an application employing the ATmega164. What will you do to transition to another microcontroller? Provide a step-by-step description of the transition process.

2.17. What is the advantage of ISP programming over other microcontroller programming techniques?

2.18. A constant is declared as unsigned int = 177. Sketch the bit contents of this variable.

2.19. A constant is declared as int = 177. Sketch the bit contents of this variable.

2.20. A constant is declared as unsigned char = 177. Sketch the bit contents of this variable.

2.21. A constant is declared as char = 177. Sketch the bit contents of this variable.

2.22. Use bit twiddling techniques to set the ATmega164 PORTD[5:2] to logic one while not changing the other bits.

2.23. Use bit twiddling techniques to set the ATmega164 PORTB[7:5] to logic zero while not changing the other bits.

REFERENCES

S. Barrett and D. Pack, Microcontroller Fundamentals for Engineers and Scientists, Morgan & Claypool Publishers, 2006.

J. Hennessy and D. Patterson, Computer Architecture A Quantitative Approach, 3 ed, Morgan Kaufman Publishers, 2003.

Atmel 8-bit AVR Microcontroller with 16/32/64K Bytes In-System Programmable Flash, ATmega164P/V, ATmega324P/V, 644P/V data sheet: 8011I-AVR-05/08, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.

ImageCraft Embedded Systems C Development Tools, www.imagecraft.com, 706 Colorado Ave. #10-88, Palo Alto, CA 94303.

CHAPTER 3

Serial Communication Subsystem

Objectives: After reading this chapter, the reader should be able to

- Describe the differences between serial and parallel communication methods.
- Describe the operation of the Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART).
- Program the USART for basic transmission and reception.
- Design a full duplex USART-based microcontroller communication system.
- Design a USART-based system employing a RF transceiver.
- Describe the operation of the Serial Peripheral Interface (SPI).
- Design a SPI-based system to extend the features of the Atmel AVR microcontroller.
- Describe the purpose and function of the Two Wire Interface (TWI).
- Describe the function and purpose of the Controller Area Network (CAN).
- Describe the purpose and function of the Zigbee IEEE 802.15.4 interface.

3.1 SERIAL COMMUNICATIONS

Microcontrollers must often exchange data with other microcontrollers or peripheral devices. Data may be exchanged by using parallel or serial techniques. With parallel techniques, an entire byte of data is typically sent simultaneously from the transmitting device to the receiver device. While this is efficient from a time point of view, it requires eight separate lines for the data transfer.

In serial transmission, a byte of data is sent a single bit at a time. Once eight bits have been received at the receiver, the data byte is reconstructed. While this is inefficient from a time point of view, it only requires a line (or two) to transmit the data. Serial transmission helps preserve the use of precious microcontroller input/output pins.

The ATmega164 is equipped with a host of different serial communication subsystems including two channels of serial USART, a serial peripheral interface or SPI channel, and the Two-wire

Serial Interface (TWI). What these systems have in common is the serial transmission of data. Specially configured microcontrollers can be readily purchased equipped with Controller Area Network (CAN) and Zigbee communication features.

In the ensuing discussion, we assume the reader is familiar with serial communication terminology. If this is not the case, we recommend a review of selected sections of "Atmel AVR Microcontroller Primer: Programming and Interfacing." For completeness, we include an ASCII table in Figure 3.1.

3.1.1 ASCII

The American Standard Code for Information Interchange or ASCII is a standardized, seven bit method of encoding alphanumeric data. It has been in use for many decades, so some of the characters and actions listed in the ASCII table are not in common use today. However, ASCII is still the most common method of encoding alphanumeric data. For example, the capital letter "G" is encoded in ASCII as 0x47. The "0x" symbol indicates the hexadecimal number representation. Unicode is the international counterpart of ASCII. It provides standardized 16-bit encoding format for the written languages of the world. ASCII is a subset of Unicode. The interested reader is referred to the Unicode home page website at `www.unicode.org` for additional information on this standardized encoding format.

		Most significant digit							
		0x0_	0x1_	0x2_	0x3_	0x4_	0x5_	0x6_	0x7_
Least significant digit	0x_0	NUL	DLE	SP	0	@	P	`	p
	0x_1	SOH	DC1	!	1	A	Q	a	q
	0x_2	STX	DC2	"	2	B	R	b	r
	0x_3	ETX	DC3	#	3	C	S	c	s
	0x_4	EOT	DC4	$	4	D	T	d	t
	0x_5	ENQ	NAK	%	5	E	U	e	u
	0x_6	ACK	SYN	&	6	F	V	f	v
	0x_7	BEL	ETB	'	7	G	W	g	w
	0x_8	BS	CAN	(	8	H	X	h	x
	0x_9	HT	EM	)	9	I	Y	i	y
	0x_A	LF	SUB	*	:	J	Z	j	z
	0x_B	VT	ESC	+	;	K	[	k	{
	0x_C	FF	FS	'	<	L	\	l	|
	0x_D	CR	GS	-	=	M	]	m	}
	0x_E	SO	RS	.	>	N	^	n	~
	0x_F	SI	US	/	?	O	_	o	DEL

Figure 3.1: ASCII Code. The ASCII code is used to encode alphanumeric characters. The "0x" indicates hexadecimal notation in the C programming language.

3.2 SERIAL USART

The serial USART is used for full duplex (two way) communication between a receiver and a transmitter. This is accomplished by equipping the ATmega164 with independent hardware for the transmitter and the receiver. The USART is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and the receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence. The Atmel USART also has synchronous features. When linking components via a serial interface, it is important to ensure the communication parameters between the transmitter and receiver are matched. Both transmitter and receiver should be set for the same Baud rate, same number of start and stop bits, parity, polarity, voltage level, etc.

The ATmega164 USART is quite flexible. It has the capability to be set to a variety of data transmission or Baud (bits per second) rates. The USART may also be set for data bit widths of 5 to 9 bits with one or two stop bits. Furthermore, the ATmega164 is equipped with a hardware generated parity bit (even or odd) and parity check hardware at the receiver. A single parity bit allows for the detection of a single bit error within a byte of data. The USART may also be configured to operate in a synchronous mode. We now discuss the operation, programming, and application of the USART (Atmel).

3.2.1 SYSTEM OVERVIEW

The block diagram for the USART is provided in Figure 3.2. The block diagram may appear a bit overwhelming but realize there are four basic pieces to the diagram: the clock generator, the transmission hardware, the receiver hardware, and three control registers (UCSRA, UCSBR, and UCSRC). We discuss each in turn.

3.2.1.1 USART Clock Generator

The USART Clock Generator provides the clock source for the USART system and sets the Baud rate for the USART. The Baud Rate is derived from the overall microcontroller clock source. The overall system clock is divided by the USART Baud rate Registers UBRR[H:L] and several additional dividers to set the Baud rate. For the asynchronous normal mode (U2X bit = 0), the Baud Rate is determined using the following expression (Atmel):

$$Baud\ rate\ =\ (system\ clock\ frequency)/(16(UBRR\ +\ 1))$$

where UBRR is the contents of the UBRRH and UBRRL registers (0 to 4095). Solving for UBRR yields:

$$UBRR\ =\ ((system\ clock\ generator)/(16\ \times\ Baud\ rate))\ -\ 1$$

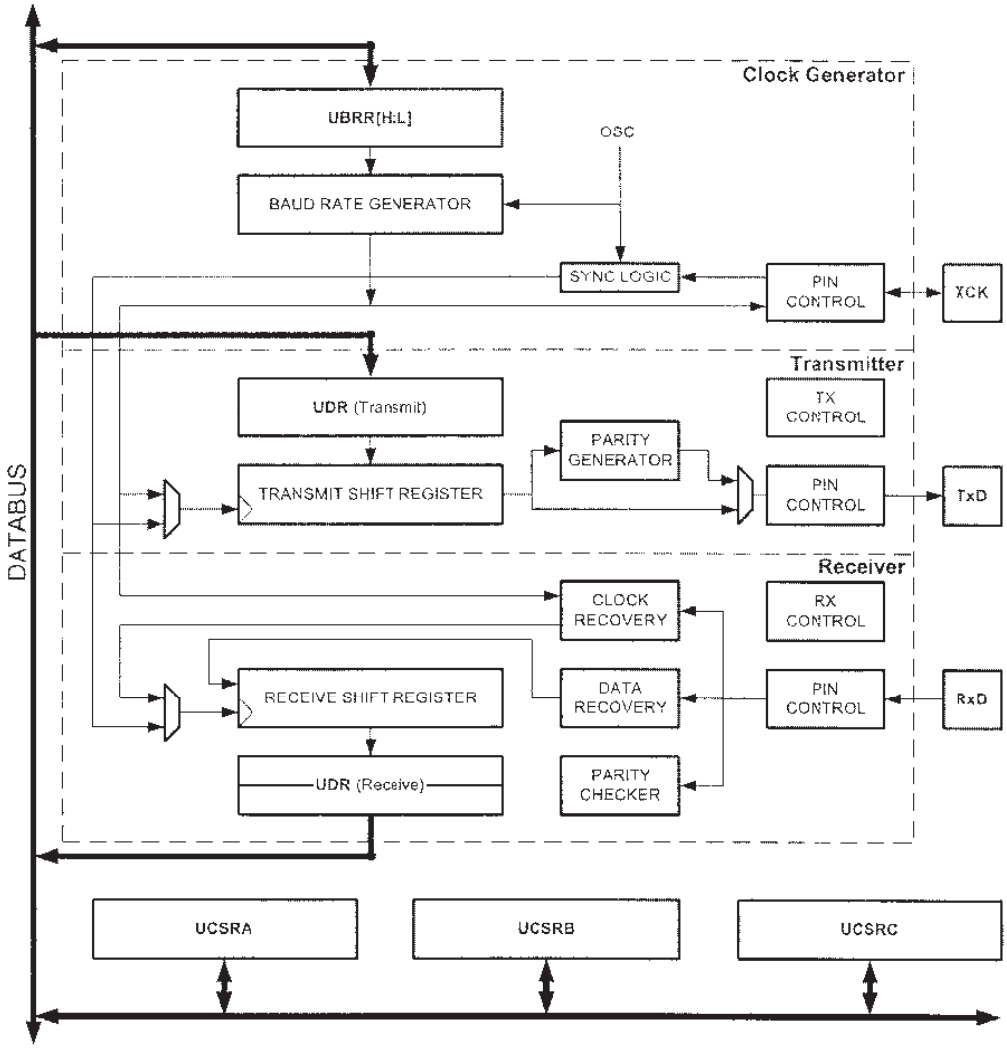


Figure 3.2: Atmel AVR ATmega164 USART block diagram. (Figure used with permission of Atmel, Incorporated.)

3.2.1.2 USART Transmitter

The USART transmitter consists of a Transmit Shift Register. The data to be transmitted is loaded into the Transmit Shift Register via the USART I/O Data Register (UDR). The start and stop framing bits are automatically appended to the data within the Transmit Shift Register. The parity is automatically calculated and appended to the Transmit Shift Register. Data is then shifted out

of the Transmit Shift Register via the TxD pin a single bit at a time at the established Baud rate. The USART transmitter is equipped with two status flags: the UDRE and the TXC flags. The USART Data Register Empty (UDRE) flag sets when the transmit buffer is empty indicating it is ready to receive new data. This bit should be written to a zero when writing the USART Control and Status Register A (UCSRA). The UDRE bit is cleared by writing to the USART I/O Data Register (UDR). The Transmit Complete (TXC) Flag bit is set to logic one when the entire frame in the Transmit Shift Register has been shifted out, and there are no new data currently present in the transmit buffer. The TXC bit may be reset by writing a logic one to it (Atmel).

3.2.1.3 USART Receiver

The USART Receiver is virtually identical to the USART Transmitter except for the direction of the data flow is reversed. Data is received a single bit at a time via the RxD pin at the established Baud Rate. The USART Receiver is equipped with the Receive Complete (RXC) Flag. The RXC flag is logic one when unread data exists in the receive buffer (Atmel).

3.2.1.4 USART Registers

In this section, we discuss the register settings for controlling the USART system. We have already discussed the function of the USART I/O Data Register (UDR) and the USART Baud Rate Registers (UBRRH and UBRRL). **Note:** The USART Control and Status Register C (UCSRC) and the USART Baud Rate Register High (UBRRH) are assigned to the same I/O location in the memory map. The URSEL bit (bit 7 of both registers) determine which register is being accessed. The URSEL bit must be one when writing to the UCSRC register and zero when writing to the UBRRH register (Atmel).

USART Control and Status Register A (UCSRA) The UCSRA register contains the RXC, TXC, and the UDRE bits. The function of these bits have already been discussed.

USART Control and Status Register B (UCSRB) The UCSRB register contains the Receiver Enable (RXEN) bit and the Transmitter Enable (TXEN) bit. These bits are the "on/off" switch for the receiver and transmitter, respectively. The UCSRB register also contains the UCSZ2 bit. The UCSZ2 bit in the UCSRB register and the UCSZ[1:0] bits contained in the UCSRC register together set the data character size.

USART Control and Status Register C (UCSRC) The UCSRC register allows the user to customize the data features to the application at hand. It should be emphasized that both the transmitter and receiver be configured with the same data features for proper data transmission. The UCSRC contains the following bits:

- USART Mode Select (UMSEL) - 0: asynchronous operation, 1: synchronous operation
- USART Parity Mode (UPM[1:0]) - 00: no parity, 10: even parity, 11: odd parity

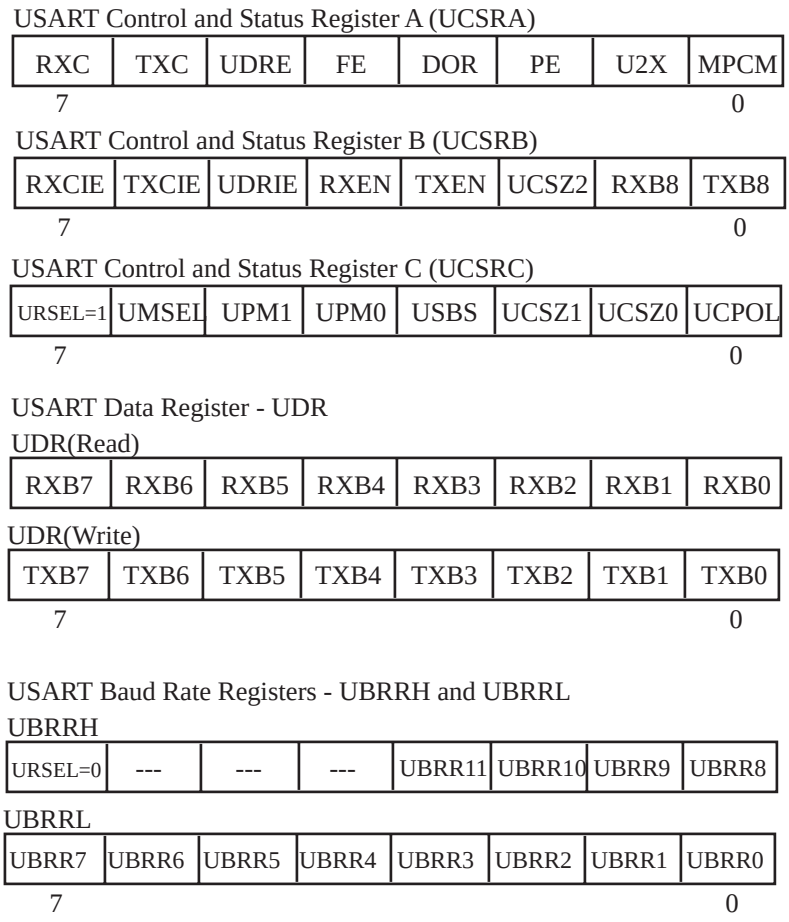


Figure 3.3: USART Registers.

- USART Stop Bit Select (USBS) - 0: 1 stop bit, 1: 2 stop bits
- USART Character Size (data width) (UCSZ[2:0]) - 000: 5-bit, 001: 6-bit, 010: 7-bit, 011: 8-bit, 111: 9-bit

3.2.2 SYSTEM OPERATION AND PROGRAMMING

The basic activities of the USART system consist of initialization, transmission, and reception. These activities are summarized in Figure 3.4. Both the transmitter and receiver must be initialized with the same communication parameters for proper data transmission. The transmission and reception

activities are similar except for the direction of data flow. In transmission, we monitor for the UDRE flag to set, indicating the data register is empty. We then load the data for transmission into the UDR register. For reception, we monitor for the RXC bit to set, indicating there is unread data in the UDR register. We then retrieve the data from the UDR register.

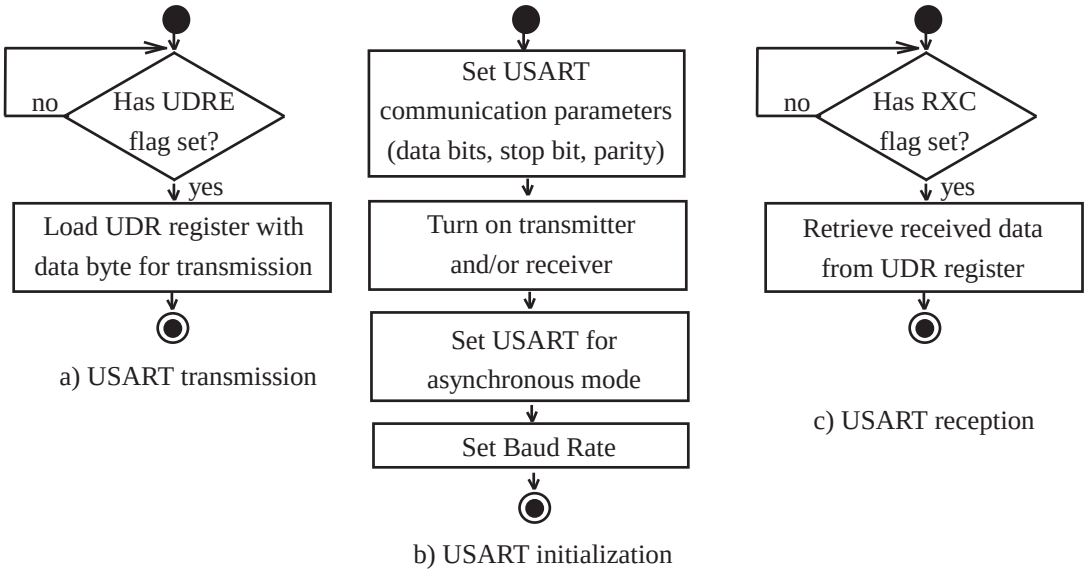


Figure 3.4: USART Activities.

To program the USART, we implement the flow diagrams provided in Figure 3.4. In the sample code provided, we assume the ATmega164 is operating at 10 MHdz, and we desire a Baud Rate of 9600, asynchronous operation, no parity, one stop bit, and eight data bits. Provided in Figure 3.5 is a sample hardware configuration to test basic USART operation. In this example, $(AC)_{16}$ is repeatedly sent. In the American Standard Code for Information Interchange or ASCII code, this would be the equivalent of sending a single quotation mark with even parity.

To achieve 9600 Baud with an operating frequency of 10 MHz, it requires that we set the UBRR registers to 64 which is 0x40.

```
//**************************************************************************
//USART_init: initialize the USART system
//**************************************************************************

void USART_init(void)
{
UCSRA = 0x00;                  //control
```

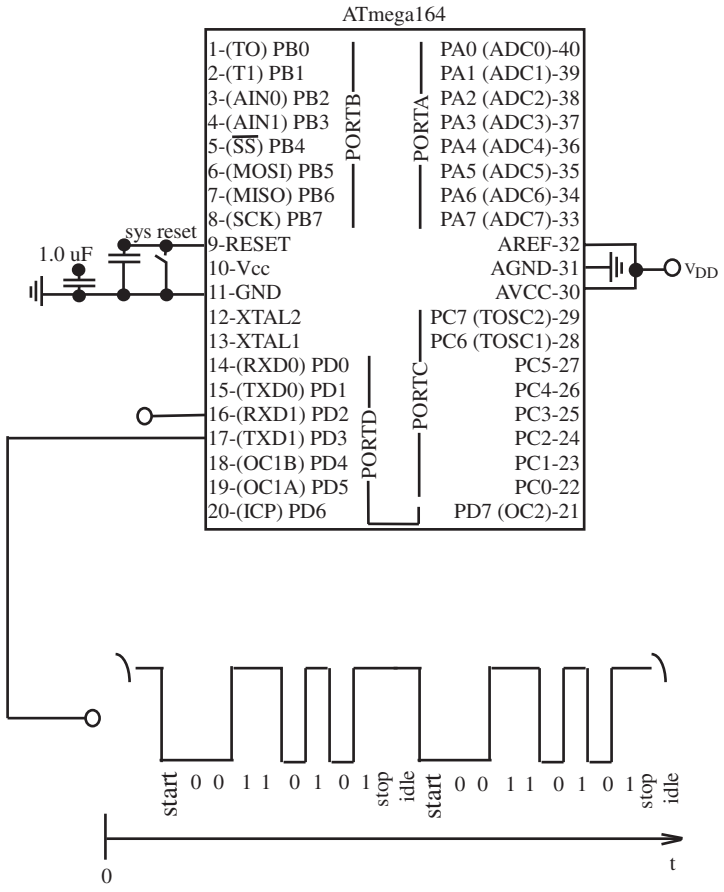


Figure 3.5: USART example. The pattern $(AC)_{16}$ is repeatedly transmitted via the USART.

```
register initialization
UCSRB = 0x08;                  //enable transmitter
UCSRC = 0x86;                  //async, no parity, 1 stop bit, 8 data bits
                               //Baud Rate initialization
UBRRH = 0x00;
UBRRL = 0x40;
}

//**************************************************************************
//USART_transmit: transmit single byte of data
//**************************************************************************
```

```
void USART_transmit(unsigned char data)
{
while((UCSRA & 0x20)==0x00)  //wait for UDRE flag
  {
  ;
  }
UDR = data;                  //load data to UDR for transmission
}

//**************************************************************************
//USART_receive: receive single byte of data
//**************************************************************************

unsigned char USART_receive(void)
{
while((UCSRA & 0x80)==0x00)  //wait for RXC flag
  {
  ;
  }
data = UDR;                  //retrieve data from UDR
return data;
}

//**************************************************************************
```

3.2.3 FULL DUPLEX USART-BASED MICROCONTROLLER LINK

The serial USART communication system allows two microcontrollers to be connected together in a full-duplex (two-way) system. This is an effective way to extend the capability of a single microcontroller should you run out of input/output pins in a specific application. Figure 3.6 provides a basic full-duplex link between two different microcontrollers. We have purposely used two different microcontrollers in this example. One microcontroller (ATmega164) is using a 10 MHz external resonator as a time base while the other is being clocked from the internal RC oscillator (ATmega168) set for 8 MHz. For proper operation, the two microcontrollers do not have to have the same operating frequency, but they must have the same USART communication parameters. In particular, they should have the same:

- Baud rate [bits per second]
- Voltage level on the USART bit stream

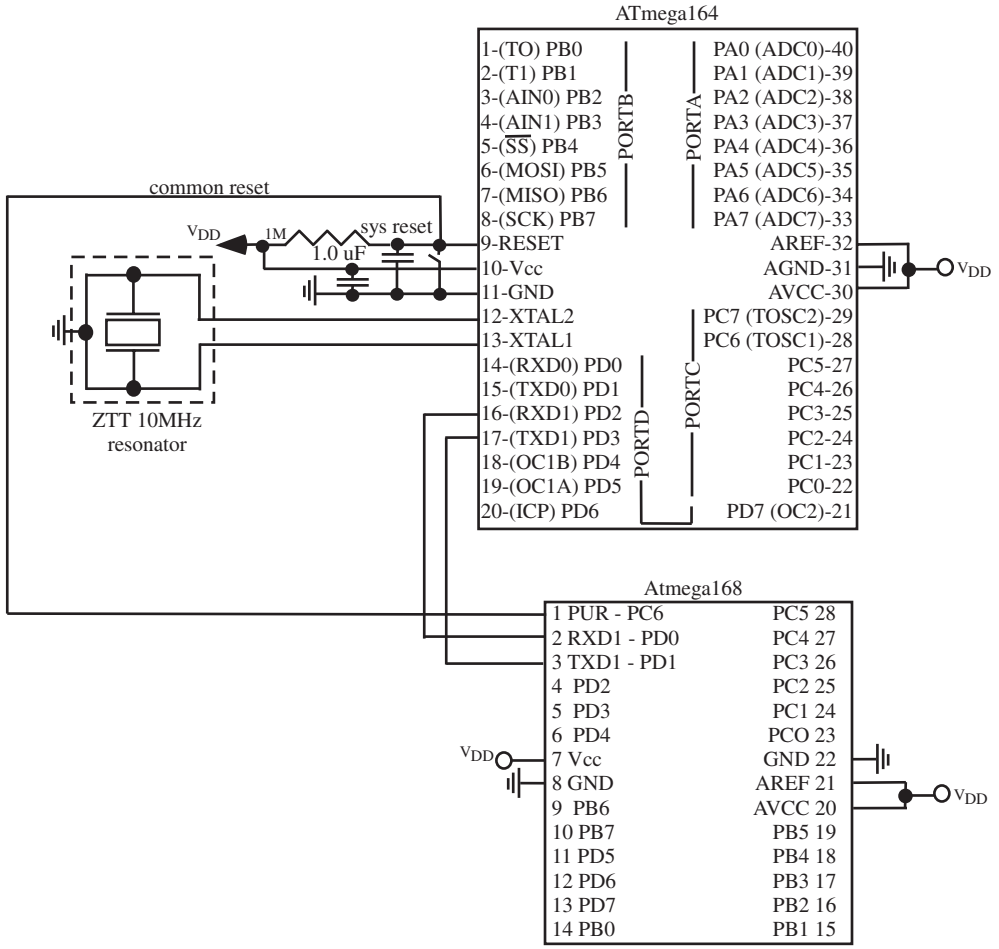


Figure 3.6: Full duplex USART communication link.

- Signal polarity
- Number of stop bits
- Number of pits per transmission
- Parity (even, odd, or none)

All of these parameters are set within the USART initialize function. As the software is developed for the system, you must also determine a communication protocol such that both systems are not transmitting at the same time.

In the following code example, the ATmega164 is acting as the supervisor microcontroller. It issues commands to the ATmega168. The ATmega168 processes the commands via a switch statement and also responds back to the ATmega164 with requested status and an inverted version of the command code sent. This allows the ATmega164 to receive confirmation that the ATmega168 has received the sent command.

A snapshot of the ATmega164 code is provided below. The ATmega164 is equipped with two different USART channels. In this specific example, we use USART channel 1. Note how the commands are transmitted to the ATmega168. The ATmega164 transmits the command to the ATmeag168 and then continuously polls its USART receiver to ensure the inverted version of the command word is received from the ATmega168. This provides some level of system integrity within the USART two way link. Should the ATmega164 not receive a response in an allotted amount of time or receive an incorrect response code, the command may be sent again. This protocol insures some level of communication link integrity.

```
//***************************************************************************
//target controller: ATMEL ATmega164
//***************************************************************************
//
//***************************************************************************
//***************************************************************************
//void InitUSART(void)
//***************************************************************************

void InitUSART_ch1(void)
{
//USART Channel 1 initialization
//System operating frequency: 10 MHz
// Comm Parameters: 8 bit Data, 1 stop, No Parity
// USART Receiver Off
// USART Transmitter On
// USART Mode: Asynchronous
// USART Baud Rate: 9600

UCSR1A=0x00;
UCSR1B=0x18;                  //RX on, TX on
UCSR1C=0x06;                  //1 stop bit, No parity
UBRR1H=0x00;
UBRR1L=0x40;
}
```

```
//*****************************************************************************
//*****************************************************************************
//void USART_TX_ch1_ch1(unsigned char data):
//USART Channel 1
//*****************************************************************************
void USART_TX_ch1(unsigned char data)
{
// Set USART Data Register

usart_delay = 0;
                              //data register empty?
while((!(UCSR1A & 0x20))&&(usart_delay < 2));
         UDR1= data;          // sets the value in ADCH to the
                              // value in the USART Data Register
}

//*****************************************************************************
//unsigned char USART_RX_ch1(void)
//USART Channel 1
//*****************************************************************************

unsigned char USART_RX_ch1(void)
{
unsigned char rx_data;

usart_delay = 0;
                              // checks to see if receive is complete
while((!(UCSR1A & 0x80))&&(usart_delay < 538));
rx_data=UDR1;                 // returns data
return rx_data;
}

//Command 01*****************************************************************
//*****************************************************************************
//void command_01_164(void);
//*****************************************************************************

void command_01_164(void)
{
```

```
unsigned char rx_data;

rx_data = 0x00;              //reset rx_data

while(rx_data != 0xfe)
  {
  USART_TX_ch1(0x01);
  rx_data = USART_RX_ch1();
  }
}

//Command 02**********************************************************************
//*********************************************************************************
//void command_02_164(void);
//*********************************************************************************

void command_02_164(void)
{
unsigned char rx_data;

rx_data = 0x00;              //reset rx_data

while(rx_data != 0xfd)
  {
  USART_TX_ch1(0x02);
  rx_data = USART_RX_ch1();
  }
}

//Command 03**********************************************************************
//*********************************************************************************
//void command_03_164(void);
//*********************************************************************************

void command_03_164(void)
{
unsigned char rx_data;

rx_data = 0x00;              //reset rx_data
```

```
while(rx_data != 0xfc)
  {
  USART_TX_ch1(0x03);
  rx_data = USART_RX_ch1();
  }
}
//***************************************************************************
```

Provided below is the code snapshot for the ATmega168. In this example, it is serving as the auxiliary microcontroller. The ATmeag168 is placed in a continuous loop waiting for instructions from the ATmega164. When an instruction is received via the USART, it will process the received command through a switch statement to determine the appropriate action required by the command. Once the action is complete, it transmits the inverted version of the command back to the ATmega164.

```
//***************************************************************************
//target controller: ATMEL ATmega168
//***************************************************************************

//include files*************************************************************
//ATMEL register definitions for ATmega168
#include<iom168pv.h>         //contains register definitions
#include<macros.h>           //stack overflow definitions

//function prototypes*******************************************************
void process_new_USART_value(unsigned char);
void InitUSART(void);
void USART_TX(unsigned char data);
unsigned char USART_RX(void);

//***************************************************************************
//main program**************************************************************
//The main program checks PORTB for user input activity.  If new activity
//is found, the program responds.
//***************************************************************************

//global variables
unsigned int    input_delay;
unsigned char   prev_USART_RX_value;
unsigned char   new_USART_RX_value;
```

```
void main(void)
{
InitUSART();

while(1)
  {
   _StackCheck();                          //check for stack overflow
   new_USART_RX_value = USART_RX();
   process_new_USART_value(new_USART_RX_value);
  }
}//end main


//*************************************************************************
//process_new_USART_value:
//*************************************************************************

void process_new_USART_value(unsigned char newest_USART_RX_value)
{
switch(newest_USART_RX_value)
  {
  //process change in USART RX value
  case 0x01:  actions_for_command_01();
              USART_TX(0xfe);
              break;

  case 0x02:  actions_for_command_02();
              USART_TX(0xfd);
              break;

  case 0x03:  actions_for_command_03();
              USART_TX(0xfc);
              break;

               :
        :
        :
```

```
  default:    USART_TX(0xaa);
  }
}
```

3.2.4 USART-BASED RADIO FREQUENCY MICROCONTROLLER LINK

To extend the range of a USART-based communication link between two microcontrollers, a radio frequency (RF) link may be used. "Plug and play" RF transmitter and receiver modules are available from a wide-variety of manufacturers. One such manufacturer, LINX Technologies, provides transmitter and receiver modules that operate at 315, 418, or 433 MHz. The modules employ a Carrier-Preset Carrier Absent (CPCA) amplitude modulation technique for communication. Basically, when a logic one is transmitted, the high-frequency carrier is on, and for a logic zero, the carrier is off. This modulation technique is illustrated in Figure 3.7. LINX indicates that serial transmission rates may be as high as 5,000 bits per second at a range of 3,000 feet (LINX).

3.2.5 USART-TO-PC

Figure 3.8 illustrates a communication link between the ATmega164 and a personal computer (PC) via a serial link. In this communication link, the microcontroller transmits via the USART. The USART signal voltage levels are at V_{DD} for a logic high and 0 volts for a logic low. These signal levels must be translated to RS-232 compatible levels for the PC.

A PC uses the RS-232 standard for serial transmission. With the RS-232 standard (EIA-232), a logic one is represented with a -12 VDC level while a logic zero is represented by a +12 VDC level. Chips are commonly available (e.g., MAX232) that convert the 5 and 0 V output levels from a transmitter to RS-232 compatible levels and convert back to 5V and 0 V levels at the receiver. [1] The RS-232 standard also specifies other features for this communication protocol.

As in previous USART examples, the communication parameters of both systems must be configured to the same settings. This example provides a handy method to print status statements from the microcontroller during code development. ASCII variables sent from the microcontroller may be displayed on the PC screen or stored to a file via the Hyper Terminal application resident within most Windows based operating systems.

3.2.6 USART SERIAL LIQUID CRYSTAL DISPLAY

The USART may also be used to equip a microcontroller application with a Liquid Crystal Display (LCD). LCDs are available using either a parallel or serial data connection path. The serial connection only requires a single connection line to the microcontroller via the USART. The parallel connection type LCD requires an entire 8-bit port and several additional port pins to communicate with the microcontroller. The serial type LCDs are especially useful when a limited number of microcontroller pins are available. However, the serial configured LCD costs approximately five times more than a

[1] It is interesting to note the MAX232 generates ±12 VDC signal levels from a single 5 VDC supply!

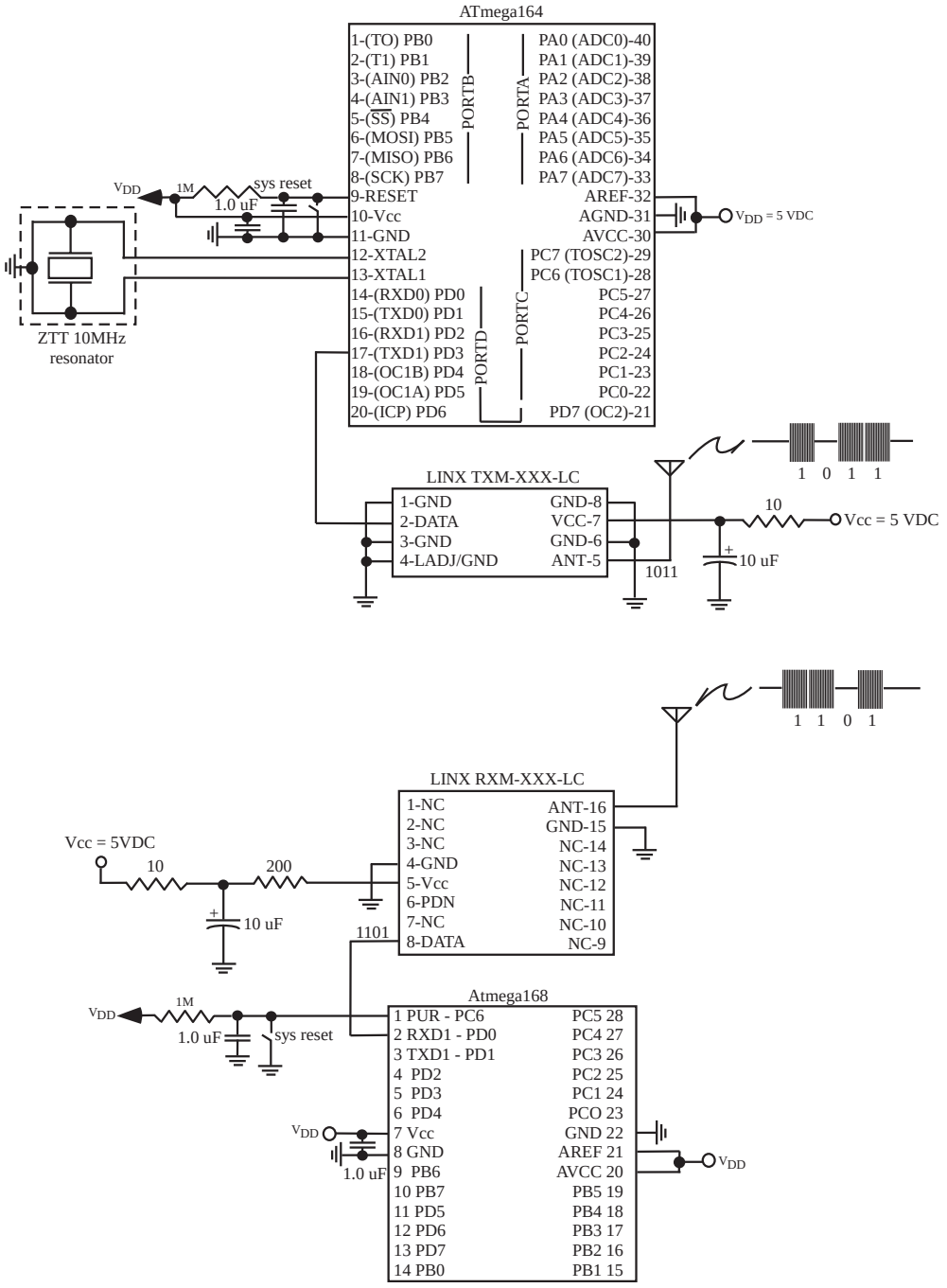


Figure 3.7: USART radio frequency link. The "XXX" specifies the operating frequency of the transmitter and receiver modules.

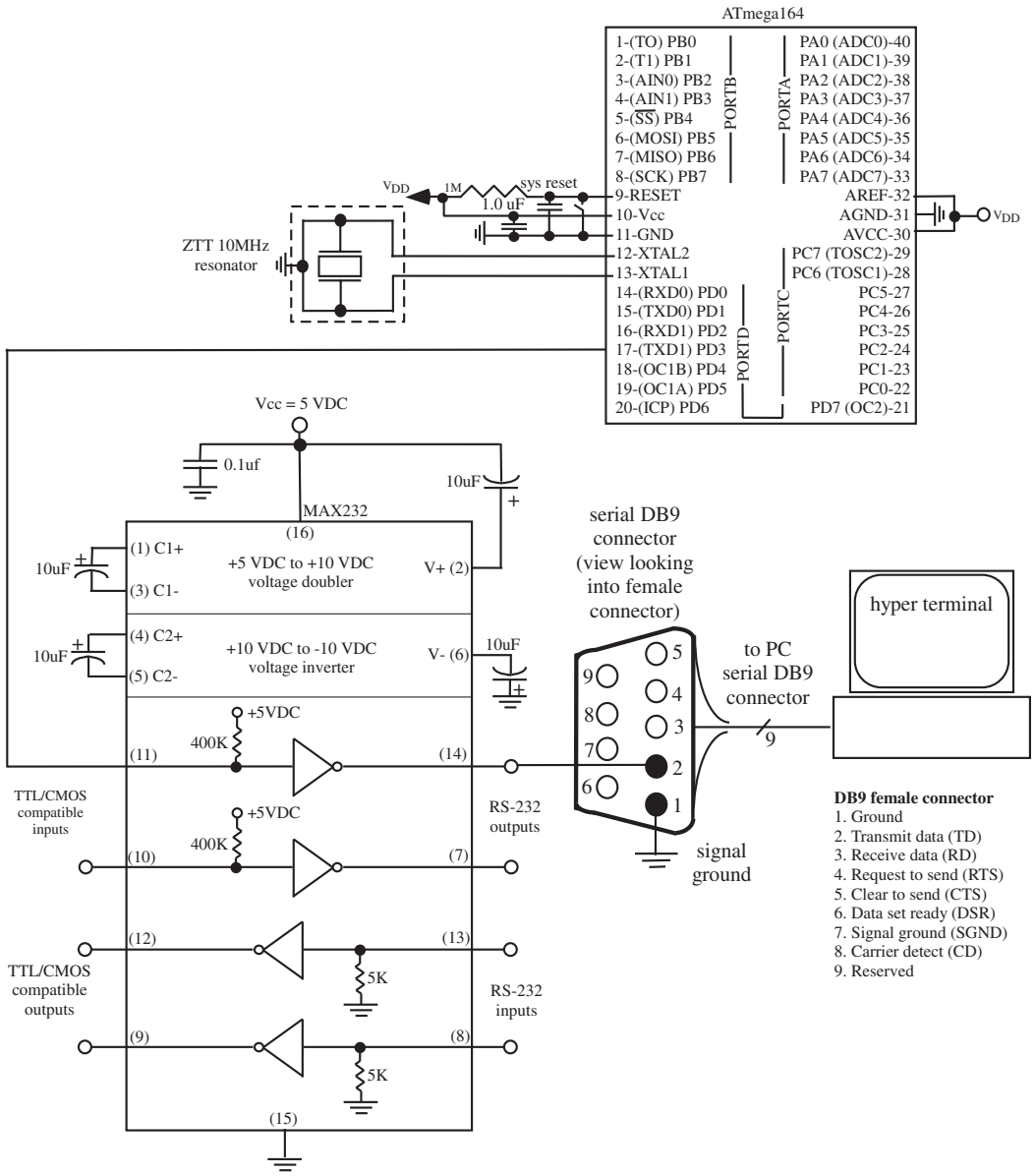


Figure 3.8: USART-to-PC communication link. This provides a handy method to print status statements from the microcontroller during code development.

parallel configured LCD with the same number of display lines and characters. We discuss the LCD interface in more detail in Chapter 7.

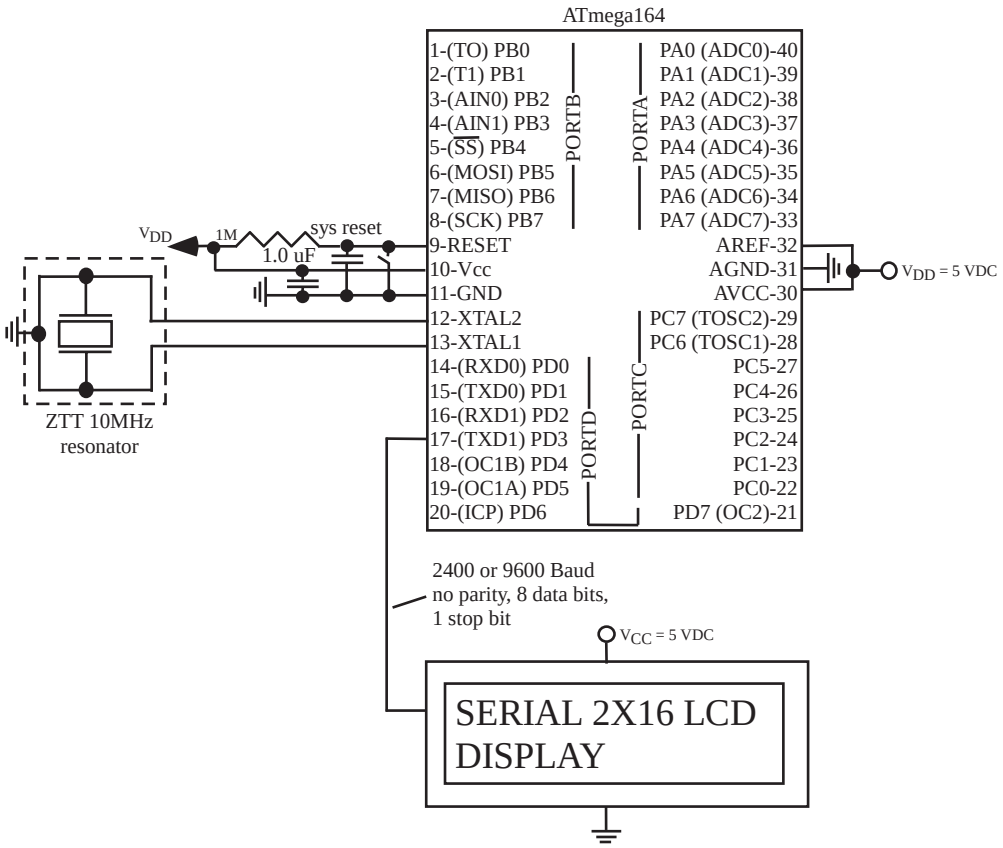


Figure 3.9: Serial LCD connection via the USART.

3.2.7 SERIAL PERIPHERAL INTERFACE—SPI

The ATmega164 Serial Peripheral Interface or SPI also provides for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and receiver share a common clock source (SCK). This requires an additional clock line between the transmitter and receiver but allows for higher data transmission rates as compared to the USART. The SPI system allows for fast and efficient data exchange between microcontrollers or peripheral devices. There are many SPI compatible external systems available to extend the features of the microcontroller. For example, a liquid crystal display or a digital-to-analog converter could be added to the microcontroller using the SPI system.

3.2.7.1 SPI Operation

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit half residing in the transmitter and the other 8-bit half residing in the receiver as shown in Figure 3.10. The transmitter is designated the master since it is providing the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. A slave is chosen for reception by taking its Slave Select ($\overline{SS}$) line low. When the $\overline{SS}$ line is taken low, the slave's shifting capability is enabled.

SPI transmission is initiated by loading a data byte into the master-configured SPI Data Register (SPDR). At that time, the SPI clock generator provides clock pulses to the master and also to the slave via the SCK pin. A single bit is shifted out of the master designated shift register on the Master Out Slave In (MOSI) microcontroller pin on every SCK pulse. The data is received at the MOSI pin of the slave designated device. At the same time, a single bit is shifted out of the Master In Slave Out (MISO) pin of the slave device and into the MISO pin of the master device. After eight master SCK clock pulses, a byte of data has been exchanged between the master and slave designated SPI devices. Completion of data transmission in the master and data reception in the slave is signaled by the SPI Interrupt Flag (SPIF) in both devices. The SPIF flag is located in the SPI Status Register (SPSR) of each device. At that time, another data byte may be transmitted (Atmel).

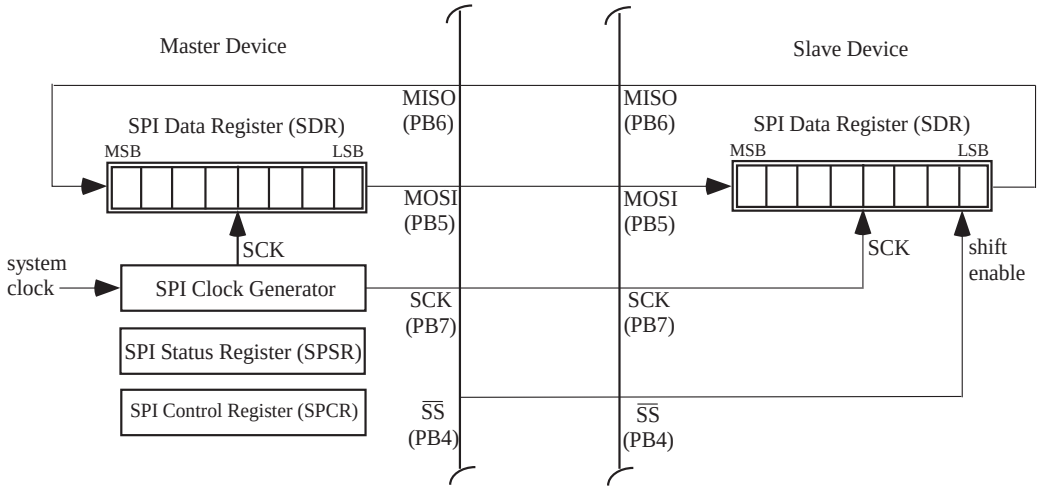


Figure 3.10: SPI Overview.

3.2.7.2 Registers

The registers for the SPI system are shown in Figure 3.11. We will discuss each one in turn.

SPI Control Register (SPCR) The SPI Control Register (SPCR) contains the "on/off" switch for the SPI system. It also provides the flexibility for the SPI to be connected to a wide variety of devices

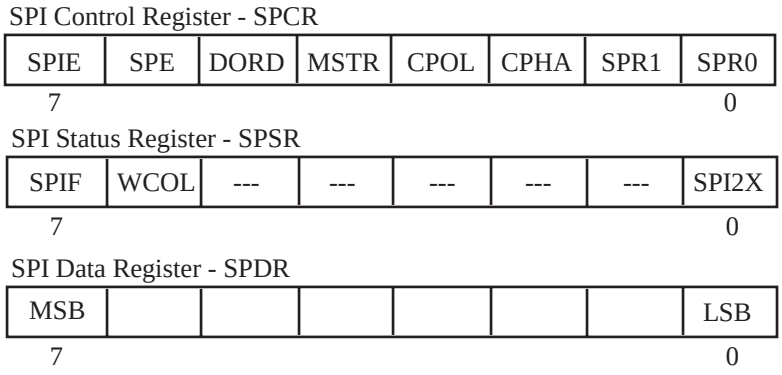


Figure 3.11: SPI Registers.

with different data formats. It is important that both the SPI master and slave devices be configured for compatible data formats for proper data transmission. The SPCR contains the following bits:

- SPI Enable (SPE) is the "on/off" switch for the SPI system. A logic one turns the system on and logic zero turns it off.
- Data Order (DORD) allows the direction of shift from master to slave to be controlled. When the DORD bit is set to one, the least significant bit (LSB) of the SPI Data Register (SPDR) is transmitted first. When the DORD bit is set to zero, the Most Significant Bit (MSB) of the SPDR is transmitted first.
- The Master/Slave Select (MSTR) bit determines if the SPI system will serve as a master (logic one) or slave (logic zero).
- The Clock Polarity (CPOL) bit determines the idle condition of the SCK pin. When CPOL is one, SCK will idle logic high; whereas, when CPOL is zero, SCK will idle logic zero.
- The Clock Phase (CPHA) determines if the data bit will be sampled on the leading (0) or trailing (1) edge of the SCK.
- The SPI SCK is derived from the microcontroller's system clock source. The system clock is divided down to form the SPI SCK. The SPI Clock Rate Select bits SPR[1:0], and the Double SPI Speed Bit (SPI2X) are used to set the division factor. The following divisions may be selected using SPI2X, SPR1, SPR0:
  - 000: SCK = system clock/4

- 001: SCK = system clock/16
- 010: SCK = system clock/64
- 011: SCK = system clock/1284
- 100: SCK = system clock/2
- 101: SCK = system clock/8
- 110: SCK = system clock/32
- 111: SCK = system clock/64

SPI Status Register (SPSR) The SPSR contains the SPI Interrupt Flag (SPIF). The flag sets when eight data bits have been transferred from the master to the slave. The SPIF bit is cleared by first reading the SPSR after the SPIF flag has been set and then reading the SPI Data Register (SPDR). The SPSR also contains the SPI2X bit used to set the SCK frequency.

SPI Data Register (SPDR) As previously mentioned, writing a data byte to the SPDR initiates SPI transmission.

3.2.7.3 Programming

To program the SPI system, the system must first be initialized with the desired data format. Data transmission may then commence. Functions for initialization, transmission and reception are provided below. In this specific example, we divide the clock oscillator frequency by 128 to set the SCK clock frequency.

```
//*****************************************************************************
//spi_init: initialize spi system
//*****************************************************************************

void spi_init(unsigned char control)
{
DDRB = 0xA0;            //Set SCK (PB7),
MOSI (PB5) for output, others to input
                        //Configure SPI Control Register (SPCR)
SPCR = 0x53;
//SPIE:0,SPE:1,DORD:0,MSTR:1,CPOL:0,CPHA:0,SPR:1,SPR0:1
}

//*****************************************************************************
//spi_write: Used by SPI master to transmit a data byte
```

```
//*********************************************************************

void spi_write(unsigned char byte)
{
SPDR = byte;
while (!(SPSR & 0x80));
}

//*********************************************************************
//spi_read: Used by SPI slave to receive data byte
//*********************************************************************

unsigned char spi_read(void)
{
while (!(SPSR & 0x80));

return SPDR;
}

//*********************************************************************
```

3.2.8 EXTENDING THE ATMEL AVR FEATURES VIA THE SPI

The SPI system may be used to connect additional subsystems to the Atmel microcontroller. For example, the SPI may be used to:

- Connect additional memory components to the microcontroller.
- Connect mass storage data loggers such as an SD card.
- Provide a real time clock.
- Connect transducers to the microcontroller.
- Equip the microcontroller with additional ports.
- Equip the microcontroller with multiple digital-to-analog converter (DAC) channels.
- Equip the microcontroller with a higher resolution analog-to-digital converter (ADC).

Example: KNH Array In Figure 3.12, we have implemented the KNH arrays described in Chapter 1 using the Atmel ATmega164. The string potentiometer and the signal generator are connected to two different analog-to-digital converter channels on the microcontroller (ADC0 and

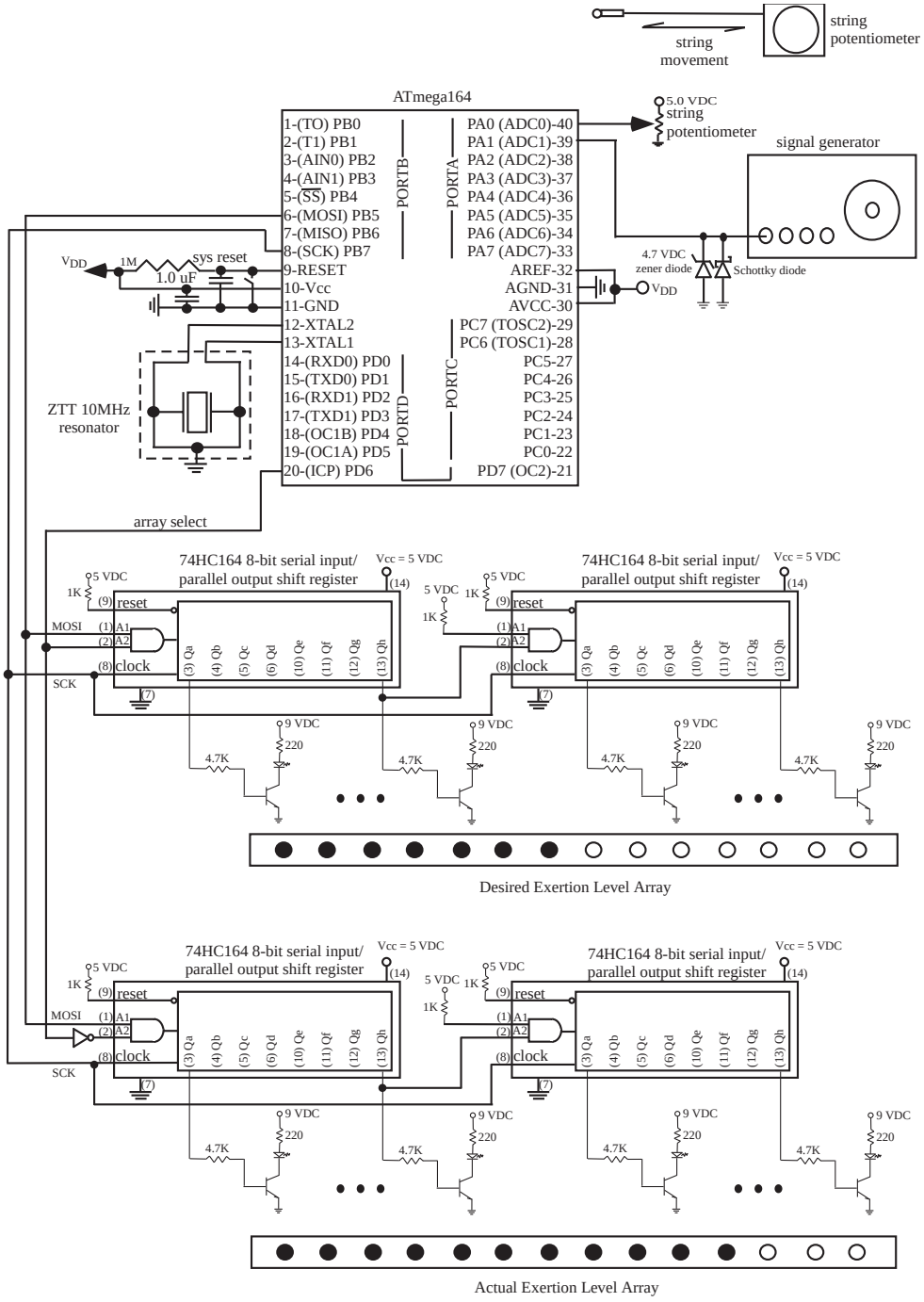


Figure 3.12: KNH arrays implemented with SPI system.

ADC1). [2] Each array is connected to the Master Out Slave In (MOSI) pin on the microcontroller. The bit stream to illuminate specific LEDs consistent with the voltage levels sensed is sent out via the MOSI. Port D, pin 6 (PD6) is used to select which array the bit stream will be sent to. When PD6 is logic high, the desired exertion level array receives the bit stream. When logic 0, the actual exertion level array receives the bit stream. The bit stream is shifted into one of two 16-bit shift registers constructed of two 74HC164 8-bit serial input/parallel output shift registers. The SCK signal from the microcontroller serves as the shift clock source for the 16-bit shift registers. The individual bit outputs on the shift registers ($Q_a - Q_h$) are connected to individual interface circuits to drive the large red light emitting diodes. The interface circuit will be discussed in detail later in the book. The external hardware required to implement the shift registers may appear excessive. Realize that these may be implemented using discrete chips, a programmable hardware device such as a Complex Programmable Logic Device (CPLD) or a Field Programmable Gate Array (FPGA).

Example - Fish tank temperature monitor. In Figure 3.13, we have configured an ATmega164 to monitor the water temperature in a series of fish tanks. The temperature transducer from each tank is connected to separate analog-to-digital converter channels (ADC0 to ADC3). An algorithm converts the sensed analog voltage into an 8-bit data stream that indicates the relative temperature of the tank water. The 8-bit data stream is sent out over the MOSI line. Port D pins 5 and 6 (PD5, PD6) are used to select which of the 8-bit shift registers (74HC164) will receive the serial bit stream via a 74HC139 1-to-4 decoder. The SCK signal from the microcontroller serves as the shift clock source for the 8-bit shift registers. The individual bit outputs on the shift registers ($Q_a - Q_h$) are connected to individual interface circuits to drive the light emitting diodes. The interface circuit will be discussed in detail later in the book. As in the previous example, the external hardware required to implement the shift registers and routing hardware may appear excessive. As before, they may be implemented using discrete logic, a CPLD or a FPGA.

3.3 NETWORKED MICROCONTROLLERS

There are a number of methods of networking microcontrollers. These include the Two-wire Serial Interface (TWI), the Controller Area Network (CAN) protocol, and the Zigbee wireless interface. Unfortunately, these network protocols are quite involved. We do not have the space to adequately cover each here. Only a brief overview will be provided of each system. The interested reader is referred to the Atmel website for detailed application notes on each system (Atmel).

3.3.1 TWO-WIRE SERIAL INTERFACE

The TWI subsystem allows the system designer to network a number of related devices (microcontrollers, transducers, displays, memory storage, etc.) together into a system using a two wire interconnecting scheme over a relatively small area. The TWI allows a maximum of 128 devices to be connected together. Each device has its own unique address and may both transmit and receive

[2] We again employ the Zener and Schottky diode combination to safely limit the input to the ADC.

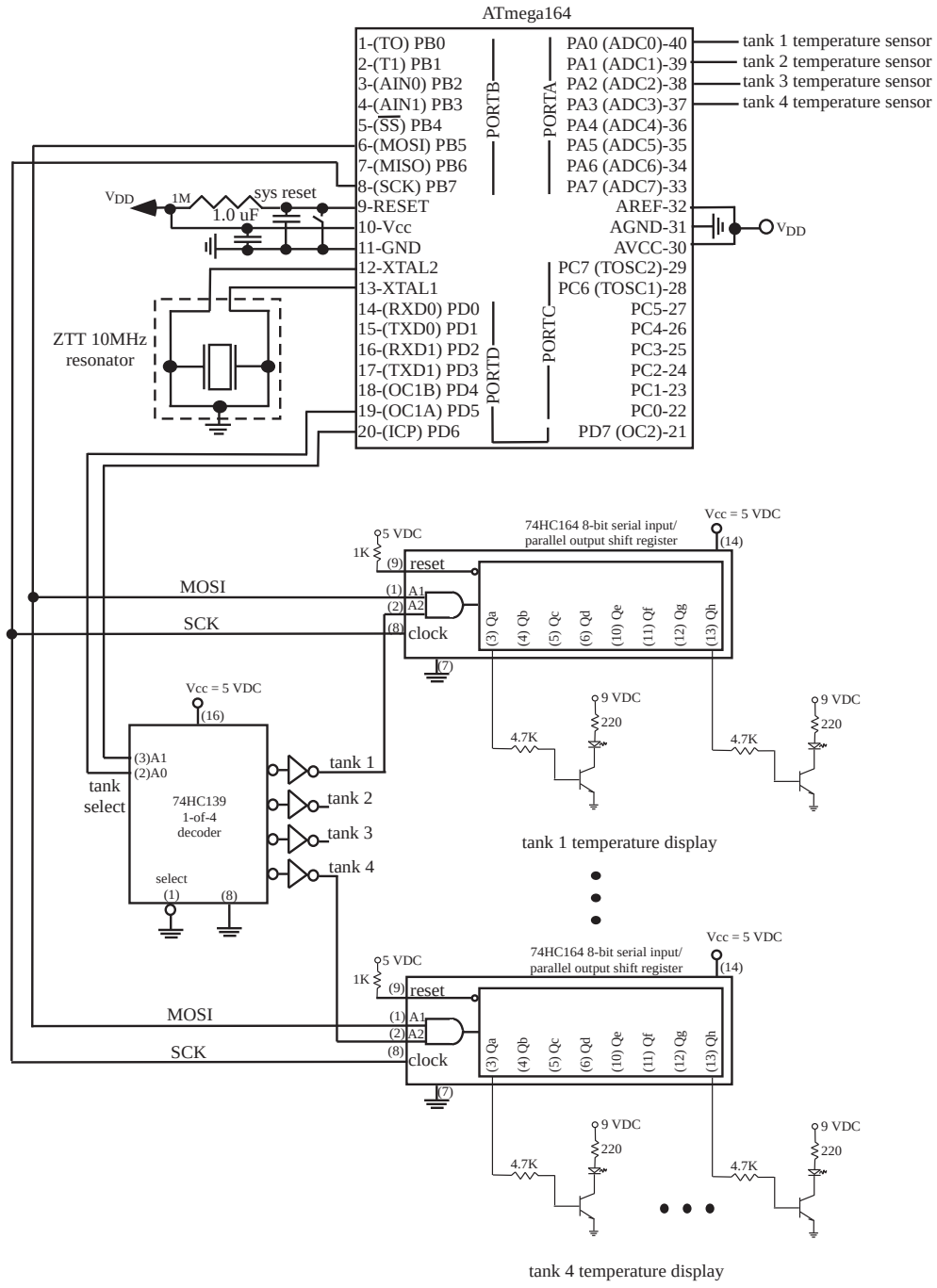


Figure 3.13: Aquarium tanks instrumented via SPI system.

over the two wire bus at frequencies up to 400 kHz. This allows the device to freely exchange information with other devices in the network within a small area. The TWI is fully compatible with the Philips Inter-IC (I2C) bus. The Atmel website has several application notes and sample software to allow one to establish a TWI interface.

3.3.2 CONTROLLER AREA NETWORK (CAN)

The CAN protocol allows multiple CAN configured devices to exchange data over a CAN configured network. It may be used over a larger area than the TWI system. The CAN protocol does not use origination or destination addresses in its messages. Instead, destination identifiers embedded in the message may be used to address a specific CAN node or a collective group of nodes. Filters set within each node determine which messages the node will respond to. The CAN protocol has achieved wide acceptance within the automotive industry as a way of linking and sharing data among related subsystems within an automobile (S. Barrett, 2006). The Atmel website has several application notes and sample software to allow one to establish a CAN interface.

3.3.3 ZIGBEE WIRELESS IEEE 802.15.4 INTERFACE

The Zigbee protocol allows microcontrollers to be interconnected together via a low data rate, 2.4 GHz radio frequency link into a wireless personal area network (WPAN). Devices within the Zigbee WPAN can either be configured as a full function device (FFD) or a restricted function device (RFD). An FFD configured device serves as the network coordinator. The coordinator establishes the network and will then allow other RFD configured devices to join the network. The coordinator also routes messages between the RFDs within the network. With the Zigbee protocol various network topologies may be formed (F. Eady, 2007). Atmel provides a Zigbee protocol "starter kit." The interested reader is referred to the Atmel website (`www.atmel.com`).

3.4 SUMMARY

In this chapter, we have discussed the differences between parallel and serial communications and key serial communication related terminology. We then, in turn, discussed the operation of USART, SPI and networked serial communication systems. We also provided basic code examples to communicate with the USART and SPI systems and to extend the features of the microcontroller.

3.5 CHAPTER PROBLEMS

3.1. Summarize the differences between parallel and serial conversion.

3.2. Summarize the differences between the USART, SPI, and TWI methods of serial communication.

3.3. Draw a block diagram of the USART system, label all key registers, and all keys USART flags.

3.4. Draw a block diagram of the SPI system, label all key registers, and all keys USART flags.

3.5. If an ATmega164 microcontroller is operating at 12 MHz, what is the maximum transmission rate for the USART and the SPI?

3.6. What is the ASCII encoded value for "Claypool?"

3.7. Draw the schematic of a system consisting of two ATmega164 that will exchange data via the SPI system. The system should include RS-232 level shifting.

3.8. Write the code to implement the system described in the question above.

3.9. Add USART and SPI features to the Testbench.

3.10. The global positioning system (GPS) consists of a constellation of 24 (or more) satellites to precisely determine your location. The GPS system transmit data at 4800 Baud, 8 data bits, no parity, and one stop bit. Write a function to read GPS data using the USART system.

3.11. Write a function that transmits a data byte over the SPI system to a specified water tank temperature display. The data byte and tank number is passed to the function as arguments. Reference Figure 3.13.

3.12. Write a function that transmits two data bytes over the SPI system to a specified KNH array. The data bytes and array destination is passed to the function arguments. Reference Figure 3.12.

3.13. What is the primary differences between the TWI, CAN, and Zigbee protocols?

3.14. Research and pick a peripheral device to interface to the ATmega164 via the SPI system. Describe the interface in detail.

REFERENCES

Atmel 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash, ATmega164, ATmega164L, data sheet: 2466L-AVR-06/05, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.

S. Barrett and D. Pack, *Microcontrollers Fundamentals for Engineers and Scientists,* Morgan and Claypool Publishers, 2006.

LINX Technologies, Inc., 159 Ort Lane, Merlin, OR 97532, www.linxtechnologies.com.

S. Barrett and D. Pack, *Embedded Systems Design and Applications with the 68HC12 and HCS12,* Pearson Education, 2005.

F. Eady, *Hands-On Zigbee Implementing 802.15.4 with Microcontrollers,* Newnes (Elsevier Imprint), 2007.

CHAPTER 4

Analog to Digital Conversion (ADC)

Objectives: After reading this chapter, the reader should be able to

- Illustrate the analog-to-digital conversion process.
- Assess the quality of analog-to-digital conversion using the metrics of sampling rate, quantization levels, number of bits used for encoding and dynamic range.
- Design signal conditioning circuits to interface sensors to analog-to-digital converters.
- Implement signal conditioning circuits with operational amplifiers.
- Describe the key registers used during an ATmega164 ADC.
- Describe the steps to perform an ADC with the ATmega164.
- Program the ATmega164 to perform an ADC.
- Design systems using the interacting features of the serial communication interface and the ADC system.
- Describe the operation of a digital-to-analog converter (DAC).
- Extend the features of the Atmel AVR to include a DAC using a parallel port and also the SPI system.

A microcontroller is used to process information from the natural world, decide on a course of action based on the information collected, and then issue control signals to implement the decision. Since the information from the natural world, is analog or continuous in nature, and the microcontroller is a digital or discrete based processor, a method to convert an analog signal to a digital form is required. An ADC system performs this task while a digital to analog converter (DAC) performs the conversion in the opposite direction. We will discuss both types of converters in this chapter. Most microcontrollers are equipped with an ADC subsystem; whereas, DACs must be added as an external peripheral device to the controller.

In the ensuing discussion, we assume the reader is familiar with the basic ADC process and terminology. If this is not the case, we encourage the reader to review these concepts in "Microcontroller Fundamentals for Engineers and Scientists." In the first section, we discuss the conversion

process itself, followed by a presentation of the successive-approximation hardware implementation of the process. We then review the basic features of the ATmega164 ADC system followed by a system description and a discussion of key ADC registers. We conclude our discussion of the analog-to-digital converter with several illustrative code examples. We conclude the chapter with a discussion of the DAC process and interface a multi-channel DAC to the ATmega164. Throughout the chapter, we provide detailed examples.

4.1 SAMPLING, QUANTIZATION AND ENCODING

In this subsection, we provide an abbreviated discussion of the ADC process. This discussion was condensed from "Atmel AVR Microcontroller Primer Programming and Interfacing." The interested reader is referred to this text for additional details and examples (S. Barrett, 2008). We present three important processes associated with the ADC: sampling, quantization, and encoding.

Sampling. We first start with the subject of sampling. Sampling is the process of taking 'snap shots' of a signal over time. Naturally, when we sample a signal, we want to sample it in an optimal fashion such that we can capture the essence of the signal while minimizing the use of resources. In essence, we want to minimize the number of samples while retaining the capability to faithfully reconstruct the original signal from the samples. Intuitively, the rate of change in a signal determines the number of samples required to faithfully reconstruct the signal, provided that all adjacent samples are captured with the same sample timing intervals.

Harry Nyquist from Bell Laboratory studied the sampling process and derived a criterion that determines the minimum sampling rate for any continuous analog signals. His, now famous, minimum sampling rate is known as the Nyquist sampling rate, which states that one must sample a signal at least twice as fast as the highest frequency content of the signal of interest. For example, if we are dealing with the human voice signal that contains frequency components that span from about 20 Hz to 4 kHz, the Nyquist sample theorem tells us that we must sample the signal at least at 8 kHz, 8000 'snap shots' every second. Engineers who work for telephone companies must deal with such issues. For further study on the Nyquist sampling rate, refer to Pack and Barrett listed in the References section. Sampling is important since when we want to represent an analog signal in a digital system, such as a computer, we must use the appropriate sampling rate to capture the analog signal for a faithful representation in digital systems.

When a signal is sampled a low pass anti-aliasing filter must be employed to insure the Nyquist sampling rate is not violated. In the example above, a low pass filter with a cutoff frequency of 4 KHz would be used before the sampling circuitry for this purpose.

Quantization. Now that we understand the sampling process, let's move on to the second process of the analog-to-digital conversion, quantization. Each digital system has a number of bits it uses as the basic unit to represent data. A bit is the most basic unit where single binary information, one or zero, is represented. A nibble is made up of four bits put together. A byte is eight bits.

We have tacitly avoided the discussion of the form of captured signal samples. When a signal is sampled, digital systems need some means to represent the captured samples. The quantization

of a sampled signal is how the signal is represented as one of the quantization levels. Suppose you have a single bit to represent an incoming signal. You only have two different numbers, 0 and 1. You may say that you can distinguish only low from high. Suppose you have two bits. You can represent four different levels, 00, 01, 10, and 11. What if you have three bits? You now can represent eight different levels: 000, 001, 010, 011, 100, 101, 110, and 111. Think of it as follows. When you had two bits, you were able to represent four different levels. If we add one more bit, that bit can be one or zero, making the total possibilities eight. Similar discussion can lead us to conclude that given n bits, we have 2^n unique numbers or levels one can represent.

Figure 4.1 shows how n bits are used to quantize a range of values. In many digital systems, the incoming signals are voltage signals. The voltage signals are first obtained from physical signals with the help of transducers, such as microphones, angle sensors, and infrared sensors. The voltage signals are then conditioned to map their range with the input range of a digital system, typically 0 to 5 volts. In Figure 4.1, n bits allow you to divide the input signal range of a digital system into 2^n different quantization levels. As can be seen from the figure, the more quantization levels means the better mapping of an incoming signal to its true value. If we only had a single bit, we can only represent level 0 and level 1. Any analog signal value in between the range had to be mapped either as level 0 or level 1, not many choices. Now imagine what happens as we increase the number of bits available for the quantization levels. What happens when the available number of bits is 8? How many different quantization levels are available now? Yes, 256. How about 10, 12, or 14? Notice also that as the number of bits used for the quantization levels increases for a given input range the 'distance' between two adjacent levels decreases accordingly.

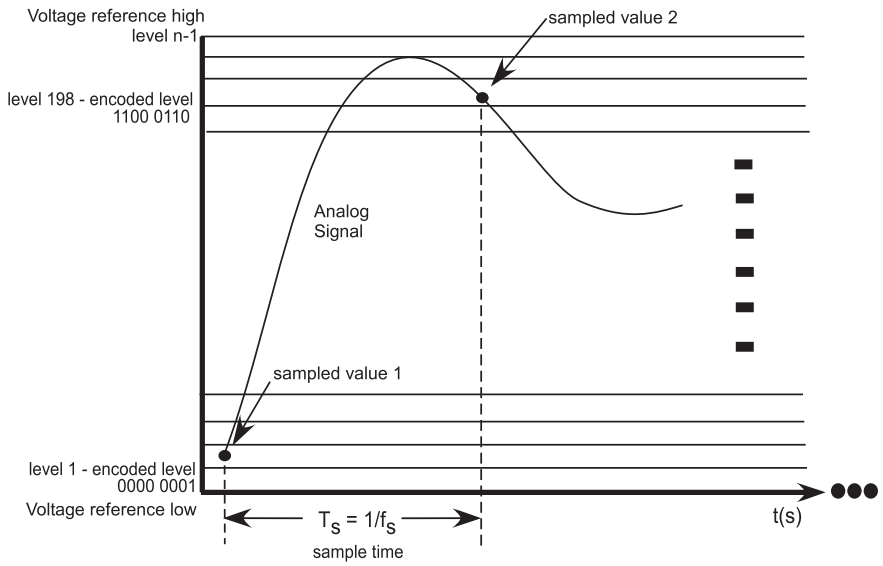


Figure 4.1: Sampling, quantization, and encoding.

Finally, the encoding process involves converting a quantized signal into a digital binary number. Suppose again we are using eight bits to quantize a sampled analog signal. The quantization levels are determined by the eight bits and each sampled signal is quantized as one of 256 quantization levels. Consider the two sampled signals shown in Figure 4.1. The first sample is mapped to quantization level 2 and the second one is mapped to quantization level 198. Note the amount of quantization error introduced for both samples. The quantization error is inversely proportional to the number of bits used to quantize the signal.

Encoding. Once a sampled signal is quantized, the encoding process involves representing the quantization level with the available bits. Thus, for the first sample, the encoded sampled value is 0000_0001, while the encoded sampled value for the second sample is 1100_0110. As a result of the encoding process, sampled analog signals are now represented as a set of binary numbers. Thus, the encoding is the last necessary step to represent a sampled analog signal into its corresponding digital form, shown in Figure 4.1.

4.1.1 RESOLUTION AND DATA RATE

Resolution. Resolution is a measure used to quantize an analog signal. In fact, resolution is nothing more than the voltage 'distance' between two adjacent quantization levels we discussed earlier. Suppose again we have a range of 5 volts and one bit to represent an analog signal. The resolution in this case is 2.5 volts, a very poor resolution. You can imagine how your TV screen will look if you only had only two levels to represent each pixel, black and white. The maximum error, called the resolution error, is 2.5 volts for the current case, 50 % of the total range of the input signal. Suppose you now have four bits to represent quantization levels. The resolution now becomes 1.25 volts or 25% of the input range. Suppose you have 20 bits for quantization levels. The resolution now becomes 4.77×10^{-6} volts, 9.54×10^{-5}% of the total range. The discussion we presented simply illustrates that as we increase the available number of quantization levels within a fixed voltage range, the distance between adjacent levels decreases, reducing the quantization error of a sampled signal. As the number grows, the error decreases, making the representation of a sampled analog signal more accurate in the corresponding digital form. The number of bits used for the quantization is directly proportional to the resolution of a system. You now should understand the technical background when you watch high definition television broadcasting. In general, resolution may be defined as:

$$resolution \;=\; (voltage\ span)/2^b \;=\; (V_{ref\ high} - V_{ref\ low})/2^b$$

for the ATmega164, the resolution is:

$$resolution \;=\; (5 \;-\; 0)/2^{10} \;=\; 4.88\, mV$$

Data rate. The definition of the data rate is the amount of data generated by a system per some time unit. Typically, the number of bits or the number of bytes per second is used as the data rate of a system. We just saw that the more bits we use for the quantization levels, the more accurate we can represent a sampled analog signal. Why not use the maximum number of bits current technologies can

offer for all digital systems, when we convert analog signals to digital counterparts? It has to do with the cost involved. In particular, suppose you are working for a telephone company and your switching system must accommodate 100,000 customers. For each individual phone conversation, suppose the company uses an 8 KHz sampling rate (f_s) and you are using 10 bits for the quantization levels for each sampled signal.[1] This means the voice conversation will be sampled every 125 microseconds (T_s) due to the reciprocal relationship between (f_s) and (T_s). If all customers are making out of town calls, what is the number of bits your switching system must process to accommodate all calls? The answer will be 100,000 $\times$ 8000 $\times$ 10 or eight billion bits per every second! You will need some major computing power to meet the requirement. For such reasons, when designers make decisions on the number of bits used for the quantization levels and the sampling rate, they must consider the computational burden the selection will produce on the computational capabilities of a digital system versus the required system resolution.

Dynamic range. You will also encounter the term "dynamic range" when you consider finding appropriate analog-to-digital converters. The dynamic range is a measure used to describe the signal to noise ratio. The unit used for the measurement is Decibel (dB), which is the strength of a signal with respect to a reference signal. The greater the dB number, the stronger the signal is compared to a noise signal. The definition of the dynamic range is 20 $log\ 2^b$ where b is the number of bits used to convert analog signals to digital signals. Typically, you will find 8 to 12 bits used in commercial analog-to-digital converters, translating the dynamic range from 20 $log\ 2^8$ dB to 20 $log\ 2^{12}$ dB.

4.2 ANALOG-TO-DIGITAL CONVERSION (ADC) PROCESS

The goal of the ADC process is to accurately represent analog signals as digital signals. Toward this end, three signal processing procedures, sampling, quantization, and encoding, described in the previous section must be combined together. Before the ADC process takes place, we first need to convert a physical signal into an electrical signal with the help of a transducer. A transducer is an electrical and/or mechanical system that converts physical signals into electrical signals or electrical signals to physical signals. Depending on the purpose, we categorize a transducer as an input transducer or an output transducer. If the conversion is from physical to electrical, we call it an input transducer. The mouse, the keyboard, and the microphone for your personal computer all fall under this category. A camera, an infrared sensor, and a temperature sensor are also input transducers. The output transducer converts electrical signals to physical signals. The computer screen and the printer for your computer are output transducers. Speakers and electrical motors are also output transducers. Therefore, transducers play the central part for digital systems to operate in our physical world by transforming physical signals to and from electrical signals. It is important to carefully design the interface between transducers and the microcontroller to insure proper operation. A poorly designed interface could result in improper embedded system operation or failure. Interface techniques will be discussed in detail in Chapter 7.

[1] For the sake of our discussion, we ignore other overheads involved in processing a phone call such as multiplexing, de-multiplexing, and serial-to-parallel conversion.

4.2.1 TRANSDUCER INTERFACE DESIGN (TID) CIRCUIT

In addition to transducers, we also need a signal conditioning circuitry before we apply the ADC. The signal conditioning circuitry is called the transducer interface. The objective of the transducer interface circuit is to scale and shift the electrical signal range to map the output of the input transducer to the input range of the analog-to-digital converter, which is typically 0 to 5 VDC. Figure 4.2 shows the transducer interface circuit using an input transducer.

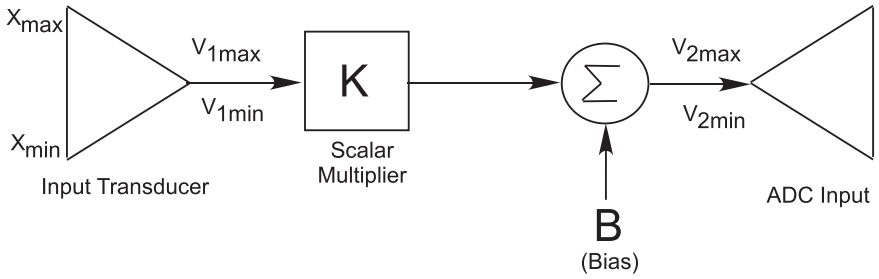


Figure 4.2: A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.

The output of the input transducer is first scaled by constant K. In the figure, we use a microphone as the input transducer whose output ranges from -5 VDC to +5 VDC. The input to the analog-to-digital converter ranges from 0 VDC to 5 VDC. The box with constant K maps the output range of the input transducer to the input range of the converter. Naturally, we need to multiply all input signals by 1/2 to accommodate the mapping. Once the range has been mapped, the signal now needs to be shifted. Note that the scale factor maps the output range of the input transducer as -2.5 VDC to +2.5 VDC instead of 0 VDC to 5 VDC. The second portion of the circuit shifts the range by 2.5 VDC, thereby completing the correct mapping. Actual implementation of the TID circuit components is accomplished using operational amplifiers.

In general, the scaling and bias process may be described by two equations:

$$V_{2max} = (V_{1max} \times K) + B$$

$$V_{2min} = (V_{1min} \times K) + B$$

The variable V_{1max} represents the maximum output voltage from the input transducer. This voltage occurs when the maximum physical variable (X_{max}) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to provide the voltage V_{2max} to the input of the ADC converter (Electrical).

Similarly, The variable V_{1min} represents the minimum output voltage from the input transducer. This voltage occurs when the minimum physical variable (X_{min}) is presented to the input

transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to produce voltage V_{2min} to the input of the ADC converter.

Usually the values of V_{1max} and V_{1min} are provided with the documentation for the transducer. Also, the values of V_{2max} and V_{2min} are known. They are the high and low reference voltages for the ADC system (usually 5 VDC and 0 VDC for a microcontroller). We thus have two equations and two unknowns to solve for K and B. The circuits to scale by K and add the offset B are usually implemented with operational amplifiers.

Example: A photodiode is a semiconductor device that provides an output current corresponding to the light impinging on its active surface. The photodiode is used with a transimpedance amplifier to convert the output current to an output voltage. A photodiode/transimpedance amplifier provides an output voltage of 0 volts for maximum rated light intensity and −2.50 VDC output voltage for the minimum rated light intensity. Calculate the required values of K and B for this light transducer so it may be interfaced to a microcontroller's ADC system.

$$V_{2max} = (V_{1max} \times K) + B$$

$$V_{2min} = (V_{1min} \times K) + B$$

$$5.0\ V = (0\ V \times K) + B$$

$$0\ V = (-2.50\ V \times K) + B$$

The values of K and B may then be determined to be 2 and 5 VDC, respectively.

4.2.2 OPERATIONAL AMPLIFIERS

In the previous section, we discussed the transducer interface design (TID) process. Going through this design process yields a required value of gain (K) and DC bias (B). Operational amplifiers (op amps) are typically used to implement a TID interface. In this section, we briefly introduce operational amplifiers including ideal op amp characteristics, classic op amp circuit configurations, and an example to illustrate how to implement a TID with op amps. Op amps are also used in a wide variety of other applications including analog computing, analog filter design, and a myriad of other applications. We do not have the space to investigate all of these related applications. The interested reader is referred to the References section at the end of the chapter for pointers to some excellent texts on this topic.

4.2.2.1 The ideal operational amplifier

A generic ideal operational amplifier is illustrated in Figure 4.3. An ideal operational does not exist in the real world. However, it is a good first approximation for use in developing op amp application circuits.

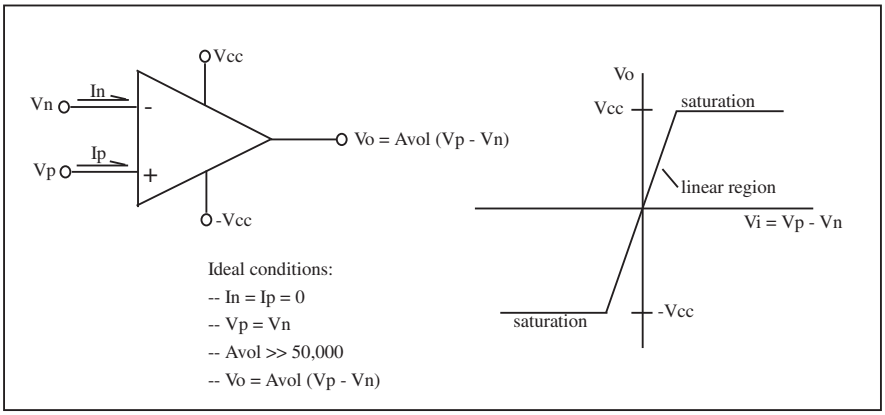


Figure 4.3: Ideal operational amplifier characteristics.

The op amp is an active device (requires power supplies) equipped with two inputs, a single output, and several voltage source inputs. The two inputs are labeled Vp, or the non-inverting input, and Vn, the inverting input. The output of the op amp is determined by taking the difference between Vp and Vn and multiplying the difference by the open loop gain (Avol) of the op amp, which is typically a large value much greater than 50,000. Due to the large value of A_{vol}, it does not take much of a difference between Vp and Vn before the op amp will saturate. When an op amp saturates it does not damage the op amp but the output is limited to $\pm V_{cc}$. This will clip the output, and hence distort the signal, at levels slightly less than $\pm V_{cc}$. Op amps are typically used in a closed loop, negative feedback configuration. A sample of classic operational amplifier configurations with negative feedback are provided in Figure 4.4(L. Faulkenberry, 1982).

It should be emphasized that the equations provided with each operational amplifier circuit are only valid if the circuit configurations are identical to those shown. Even a slight variation in the circuit configuration may have a dramatic effect on circuit operation. It is important to analyze each operational amplifier circuit using the following steps:

- Write the node equation at Vn for the circuit.
- Apply ideal op amp characteristics to the node equation.
- Solve the node equation for Vo.

As an example, we provide the analysis of the non-inverting amplifier circuit in Figure 4.5. This same analysis technique may be applied to all of the circuits in Figure 4.4 to arrive at the equations for Vout provided.

Example: In the previous section, it was determined that the values of K and B were 2 and 5 VDC, respectively. The two-stage op amp circuitry provided in Figure 4.6 implements these

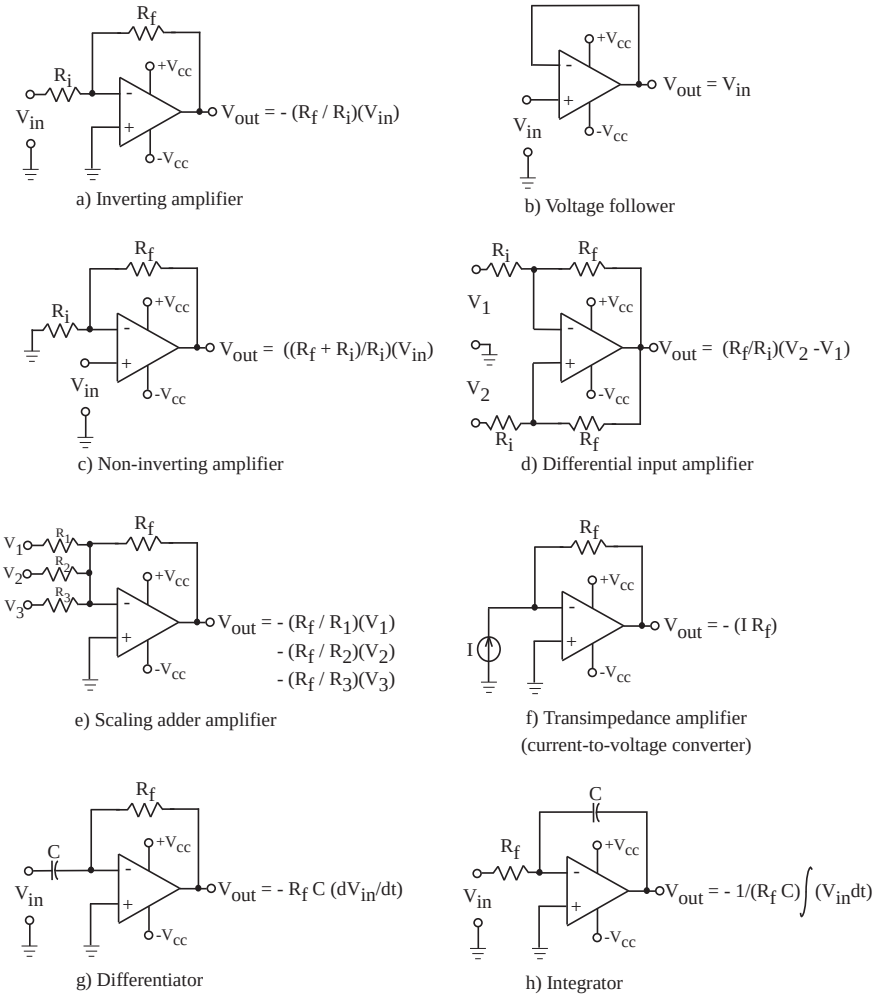
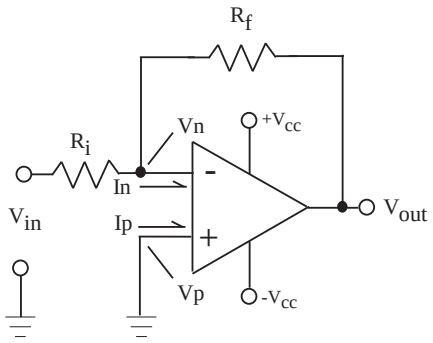


Figure 4.4: Classic operational amplifier configurations. Adapted from (L. Faulkenberry, 1982).

values of K and B. The first stage provides an amplification of -2 due to the use of the non-inverting amplifier configuration. In the next second stage, an adding amplifier is used to add the output of the first stage with a bias of −5 VDC. Since this stage also introduces a minus sign to the result, the overall result of a gain of 2 and a bias of +5 VDC is achieved.



Node equation at Vn:

$(Vn - Vin)/ Ri + (Vn - Vout)/Rf + In = 0$

Apply ideal conditions:

$In = Ip = 0$

$Vn = Vp = 0$ (since Vp is grounded)

Solve node equation for Vout:

$V_{out} = - (R_f / R_i)(V_{in})$

Figure 4.5: Operational amplifier analysis for the non-inverting amplifier. Adapted from (L. Faulkenberry, 1982).

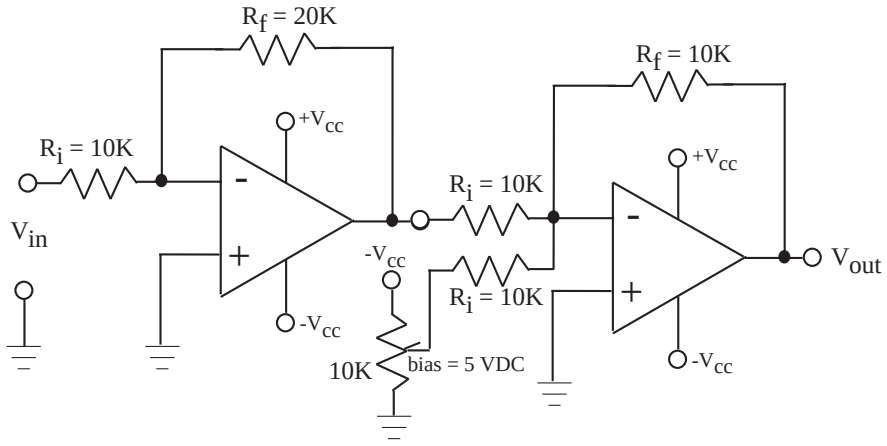


Figure 4.6: Operational amplifier implementation of the transducer interface design (TID) example circuit.

4.3 ADC CONVERSION TECHNOLOGIES

The ATmega164 uses a successive-approximation converter technique to convert an analog sample into a 10-bit digital representation. In this section, we will discuss this type of conversion process. For a review of other converter techniques the interested reader is referred to "Atmel AVR Microcontroller Primer: Programming and Interfacing." In certain applications, you are required to use converter technologies external to the microcontroller.

4.3.1 SUCCESSIVE-APPROXIMATION

The ATmega164 microcontroller is equipped with a successive-approximation ADC converter. The successive-approximation technique uses a digital-to-analog converter, a controller, and a comparator to perform the ADC process. Starting from the most significant bit down to the least significant bit, the controller turns on each bit at a time and generates an analog signal, with the help of the digital-to-analog converter, to be compared with the original input analog signal. Based on the result of the comparison, the controller changes or leaves the current bit and turns on the next most significant bit. The process continues until decisions are made for all available bits. One can consider the process similar to a game children play often. One child thinks of a number in the range of zero to ten and asks another child to guess the number within n turns. The first child will tell the second child whether a guessed number is higher or lower than the answer at the end of each turn. The optimal strategy in such a situation is to guess the middle number in the range, say five. If the answer is higher than five, the second guess should be eight. If the answer is lower than five, the second guess should be three. The strategy is to narrow down to the answer by partitioning the available range into two equal parts at every turn. The successive-approximation method works similarly in that the most significant bit is used to partition the original input range of an analog-to-digital converter into two halves, the second most significant bit divides the remaining half into two quarters of the input range, and so forth. Figure 4.7 shows the architecture of this type of converter. The advantage of this technique is that the conversion time is uniform for any input, but the disadvantage of the technology is the use of complex hardware for implementation.

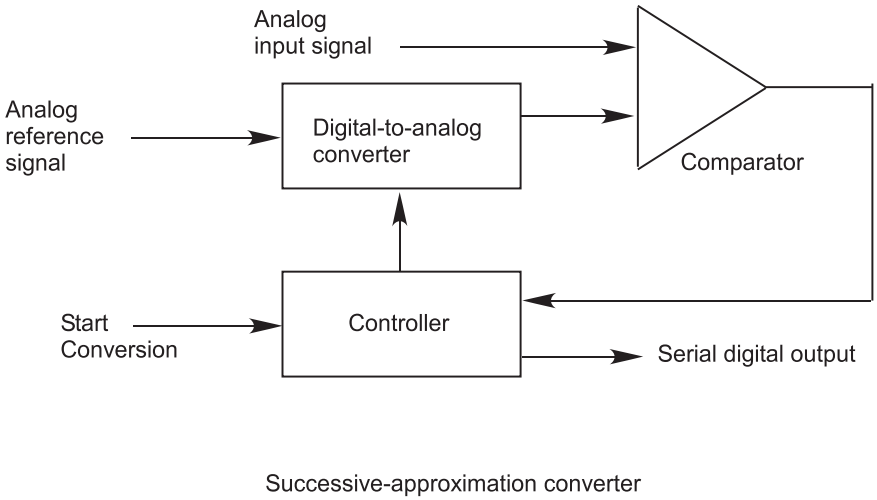


Figure 4.7: Successive-approximation ADC.

4.4 THE ATMEL ATMEGA164 ADC SYSTEM

The Atmel ATmega164 microcontroller is equipped with a flexible and powerful ADC system. It has the following features (Atmel):

- 10-bit resolution
- ±2 least significant bit (LSB) absolute accuracy
- 13 ADC clock cycle conversion time
- 8 multiplexed single ended input channels
- Selectable right or left result justification
- 0 to Vcc ADC input voltage range

Let's discuss each feature in turn. The first feature of discussion is "10-bit resolution." Resolution is defined as:

$$Resolution = (V_{RH} - V_{RL})/2^b$$

V_{RH} and V_{RL} are the ADC high and low reference voltages. Whereas, "b" is the number of bits available for conversion. For the ATmega164 with reference voltages of 5 VDC, 0 VDC, and 10-bits available for conversion, resolution is 4.88 mV. Absolute accuracy specified as ±2 LSB is then ±9.76 mV at this resolution (Atmel).

It requires 13 analog-to-digital clock cycles to perform an ADC conversion. The ADC system may be run at a slower clock frequency than the main microcontroller clock source. The main microcontroller clock is divided down using the ADC Prescaler Select (ADPS[2:0]) bits in the ADC Control and Status Register A (ADCSRA).

The ADC is equipped with a single successive-approximation converter. Only a single ADC channel may be converted at a given time. The input of the ADC is equipped with an eight input analog multiplexer. The analog input for conversion is selected using the MUX[4:0] bits in the ADC Multiplexer Selection Register (ADMUX).

The 10-bit result from the conversion process is placed in the ADC Data Registers, ADCH and ADCL. These two registers provide 16 bits for the 10-bit result. The result may be left justified by setting the ADLAR (ADC Left Adjust Result) bit of the ADMUX register. Right justification is provided by clearing this bit.

The analog input voltage for conversion must be between 0 and Vcc volts. If this is not the case, external circuitry must be used to insure the analog input voltage is within these prescribed bounds as discussed earlier in the chapter.

4.4.1 BLOCK DIAGRAM

The block diagram for the ATmega164 ADC conversion system is provided in Figure 4.8. The left edge of the diagram provides the external microcontroller pins to gain access to the ADC. The eight analog input channels are provided at ADC[7:0], and the ADC reference voltage pins are provided at AREF and AVCC. The key features and registers of the ADC system previously discussed are included in the diagram.

4.4.2 REGISTERS

The key registers for the ADC system are shown in Figure 4.9. It must be emphasized that the ADC system has many advanced capabilities that we do not discuss here. Our goal is to review the basic ADC conversion features of this powerful system. We have already discussed many of the register settings. We will discuss each register in turn (Atmel).

4.4.2.1 ADC Multiplexer Selection Register (ADMUX)

As previously discussed, the ADMUX register contains the ADLAR bit to select left or right justification and the MUX[4:0] bits to determine which analog input will be provided to the analog-to-digital converter for conversion. To select a specific input for conversion is accomplished when a binary equivalent value is loaded into the MUX[4:0] bits. For example, to convert channel ADC7, "00111" is loaded into the ADMUX register. This may be accomplished using the following C instruction:

```
ADMUX = 0x07;
```

The REFS[1:0] bits of the ADMUX register are also used to determine the reference voltage source for the ADC system. These bits may be set to the following values:

- REFS[0:0] = 00: AREF used for ADC voltage reference
- REFS[0:1] = 01: AVCC with external capacitor at the AREF pin
- REFS[1:0] = 10: Reserved
- REFS[1:1] = 11: Internal 2.56 VDC voltage reference with an external capacitor at the AREF pin

4.4.2.2 ADC Control and Status Register (ADCSRA)

The ADCSRA register contains the ADC Enable (ADEN) bit. This bit is the "on/off" switch for the ADC system. The ADC is turned on by setting this bit to a logic one. The ADC Start Conversion (ADSC) bit is also contained in the ADCSRA register. Setting this bit to logic one initiates an ADC. The ADCSRA register also contains the ADC Interrupt flag (ADIF) bit. This bit sets to logic one when the ADC is complete. The ADIF bit is reset by writing a logic one to this bit.

The ADC Prescaler Select (ADPS[2:0]) bits are used to set the ADC clock frequency. The ADC clock is derived from dividing down the main microcontroller clock. The ADPS[2:0] may be set to the following values:

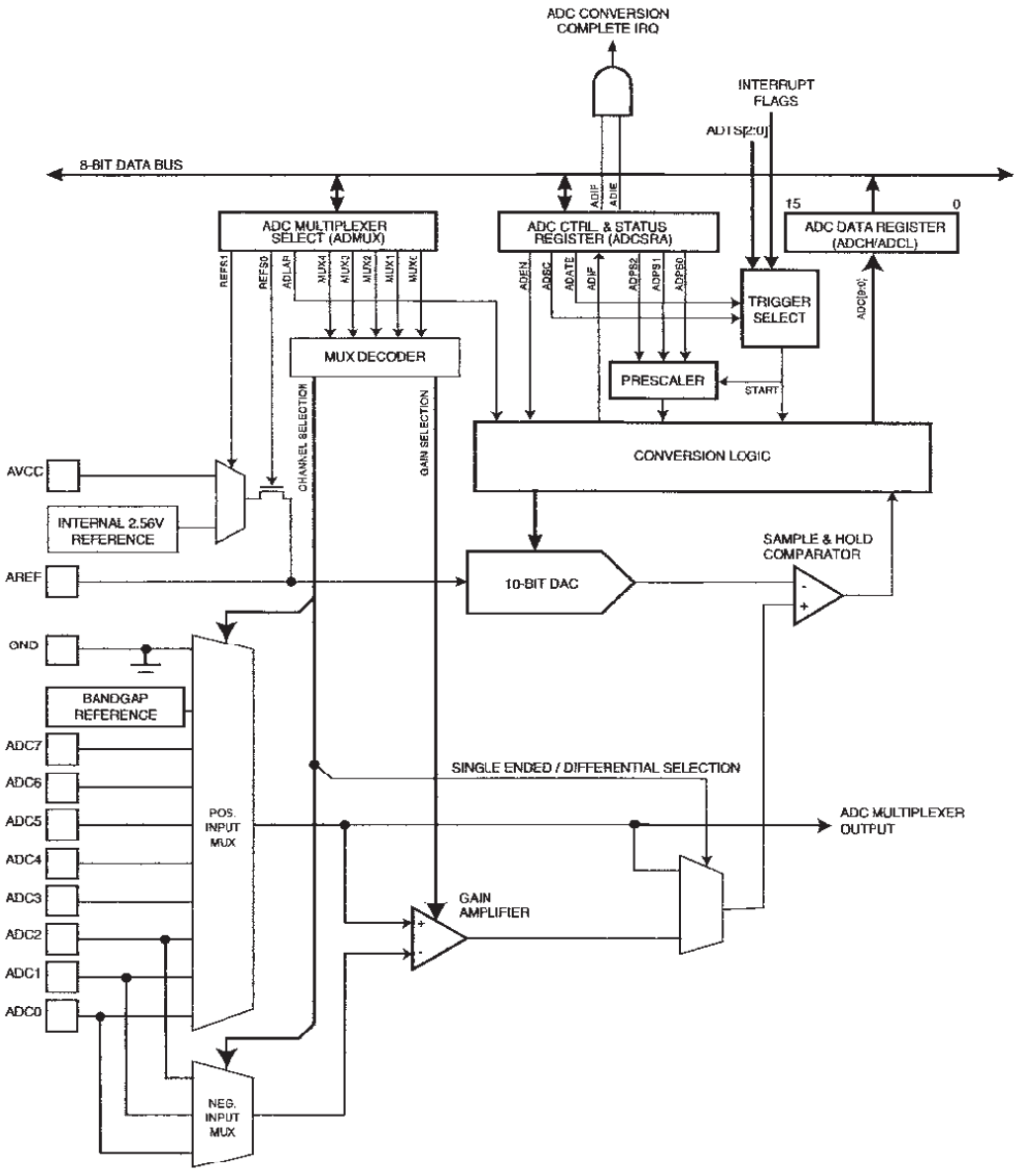


Figure 4.8: Atmel AVR ATmega164 ADC block diagram. (Figure used with permission of Atmel, Incorporated.)

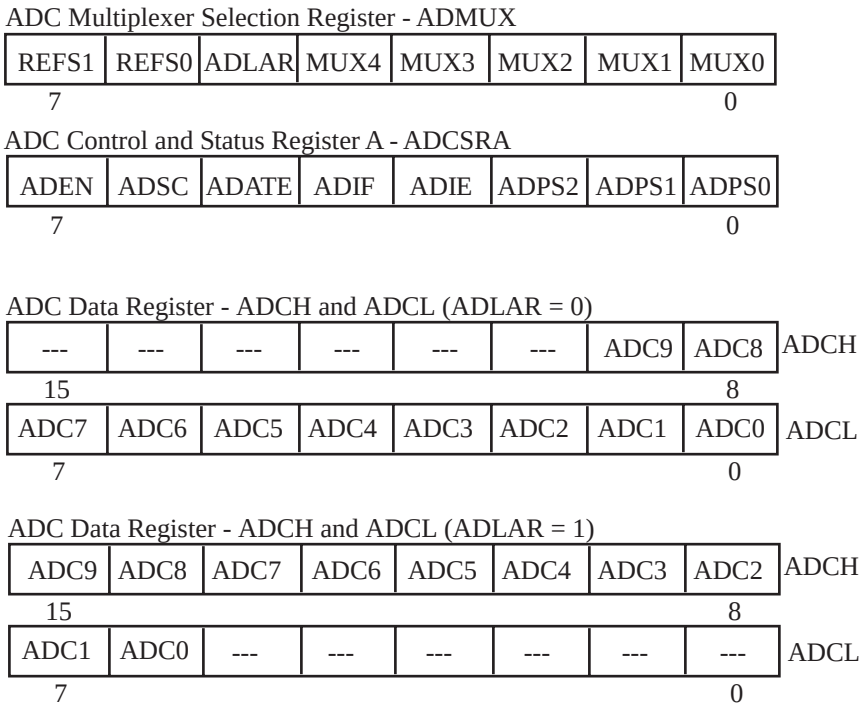


Figure 4.9: ADC Registers. Adapted from Atmel.

- ADPS[2:0] = 000: division factor none
- ADPS[2:0] = 001: division factor 2
- ADPS[2:0] = 010: division factor 4
- ADPS[2:0] = 011: division factor 8
- ADPS[2:0] = 100: division factor 16
- ADPS[2:0] = 101: division factor 32
- ADPS[2:0] = 110: division factor 64
- ADPS[2:0] = 111: division factor 128

4.4.2.3 ADC Data Registers (ADCH, ADCL)

As previously discussed, the ADC Data Register contains the result of the ADC. The results may be left (ADLAR=1) or right (ADLAR=0) justified.

4.4.3 PROGRAMMING THE ADC

Provided below are two functions to operate the ATmega164 ADC system. The first function "InitADC( )" initializes the ADC by first performing a dummy conversion on channel 0. In this particular example, the ADC prescalar is set to 8 (the main microcontroller clock was operating at 10 MHz).

The function then enters a while loop waiting for the ADIF bit to set indicating the conversion is complete. After conversion, the ADIF bit is reset by writing a logic one to it.

The second function, "ReadADC(unsigned char)," is used to read the analog voltage from the specified ADC channel. The desired channel for conversion is passed in as an unsigned character variable. The result is returned as a left justified, 10 bit binary result. The ADC prescalar must be set to 8 to slow down the ADC clock at higher external clock frequencies (10 MHz) to obtain accurate results. After the ADC is complete, the results in the eight bit ADCL and ADCH result registers are concatenated into a 16-bit unsigned integer variable and returned to the function call.

```
//***********************************************************************
//InitADC: initialize analog-to-digital converter
//***********************************************************************

void InitADC( void)
{
ADMUX = 0;                              //Select channel 0
ADCSRA = 0xC3;
 //Enable ADC & start 1st dummy conversion
                                        //Set ADC module prescalar to 8


 //critical for accurate ADC results
while (!(ADCSRA & 0x10));               //Check if conversation is ready
ADCSRA |= 0x10;
 //Clear conv rdy flag - set the bit
}

//***********************************************************************
//ReadADC: read analog voltage from analog-to-digital converter - the desired
//channel for conversion is passed in as an unsigned character variable.  The
//result is returned as a left justified, 10 bit binary result.  The ADC
//prescalar must be set to 8 to slow down the ADC clock at higher external
//clock frequencies (10 MHz) to obtain accurate results.
//***********************************************************************
```

```
unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary voltage

ADMUX = channel;                        //Select channel
ADCSRA |= 0x43;                         //Start conversion
                                        //Set ADC module prescalar to 8
                                        //critical for accurate ADC results
while (!(ADCSRA & 0x10));               //Check if conversion is ready
ADCSRA |= 0x10;                         //Clear Conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL;     //Read 8 low bits first (important)
                                        //Read 2 high bits, multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low
| binary_weighted_voltage_high;
return binary_weighted_voltage;         //ADCH:ADCL
}

//***************************************************************************
```

4.5 EXAMPLES

In this section, we provide a wide variety of applications using the ADC converter beginning with a rain gage type display.

4.5.1 ADC RAIN GAGE INDICATOR

In this example, we construct a rain gage type level display using small light emitting diodes. The circuit configuration is provided in Figure 4.10. We will describe the operation of the LED interface circuit later in the text.

The requirements for this project included:

- Write a function to display an incrementing binary count from 0 to 255 on Port C of the ATmega164.

- Write a function to display a decrementing binary count from 255 to 0 on Port C of the ATmega164.

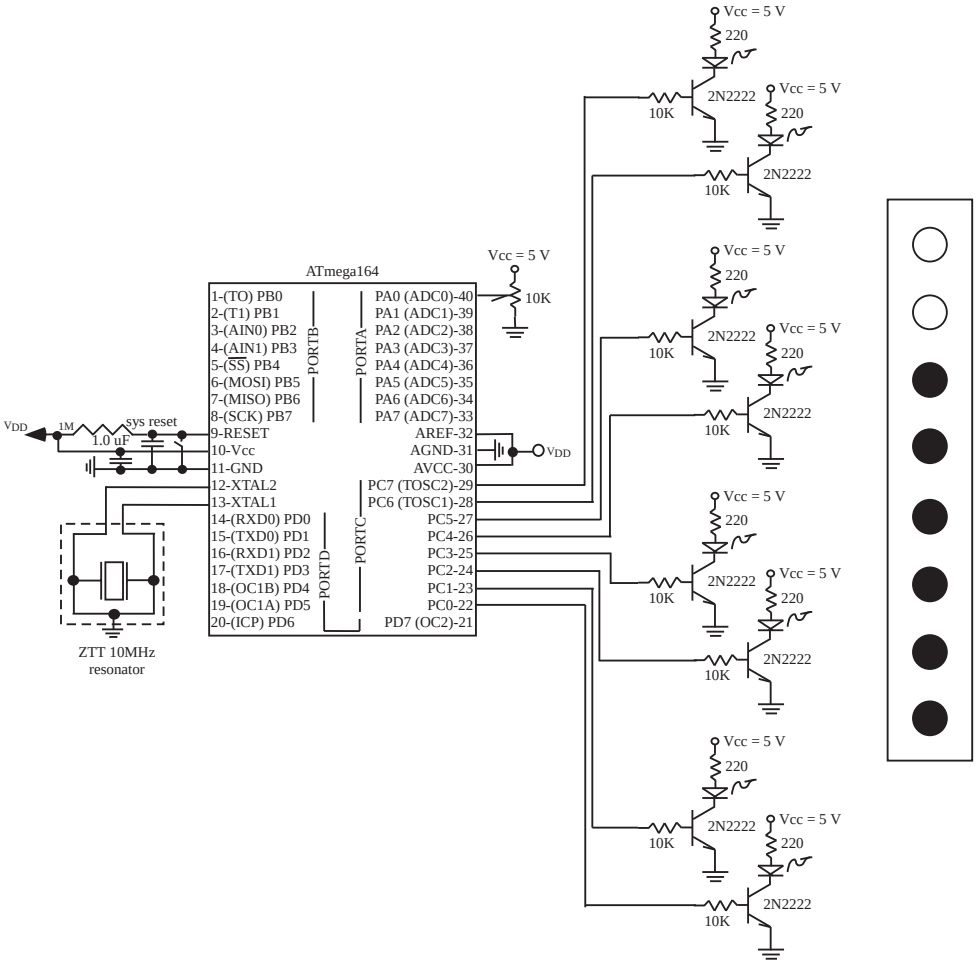


Figure 4.10: ADC with rain gage level indicator.

- Use the on-chip analog-to-digital converter to illuminate up to eight LEDs based on the input voltage. A 10k trimmer potentiometer is connected to the ADC channel to vary the input voltage.

The solution for this project was written by Anthony (Tony) Kunkel, MSEE and Geoff Luke, MSEE, at the University of Wyoming for an Industrial Controls class assignment. A 30 ms delay is provided between PORTC LED display updates. This prevents the display from looking as a series of LEDs that are always on.

```
//*************************************************************************
//Tony Kunkel and Geoff Luke
//University of Wyoming
//*************************************************************************
//This program calls four functions:
//First: Display an incrementing binary count on PORT C from 0-255
//Second: Display a decrementing count on PORT C from 255-0
//Third: Display rain gauge info on PORT C
//Fourth: Delay 30 ms when ever PORT C is updated
//*************************************************************************

//ATMEL register definitions for ATmega16
#include<iom164pv.h>

//function prototypes
void display_increment(void);        //Function
to display increment to PORT C
void display_decrement(void);        //Function
to display decrement to PORT C
void rain_gage(void)
void InitADC(void);                  //Initialize ADC converter
unsigned int ReadADC();              //Read specified ADC channel
void delay_30ms(void);               //Function to delay 30 ms

//*************************************************************************

int main(void)
{

display_increment();         //Display
incrementing binary on PORT C from 0-255
delay_30ms();                //Delay 30 ms
display_decrement();         //Display
decrementing binary on PORT C from 255-0
delay_30ms();                //Delay 30 ms
InitADC();

while(1)
  {
```

```
  rain_gage();              //Display gage info on PORT C
  delay_30ms();             //Delay 30 ms
  }

return 0;
}

//***************************************************************************
//function definitions
//***************************************************************************

//***************************************************************************
//30 ms delay based on an 8MHz clock
//***************************************************************************

void delay_30ms(void)
{
int i, k;

for(i=0; i<400; i++)
  {
  for(k=0; k<300; k++)
{
      asm("nop");           //assembly language nop, requires 2 cycles
}
  }
}

//***************************************************************************
//Displays incrementing binary count from 0 to 255
//***************************************************************************

void display_increment(void)
{
int i;
unsigned char j = 0x00;

DDRC = 0xFF;                //set Port C to output
```

```
for(i=0; i<255; i++)
  {
  j++;                        //increment j
  PORTC = j;                  //assign j to data port
  delay_30ms();               //wait 30 ms
  }
}

//***************************************************************************
//Displays decrementing binary count from 255 to 0
//***************************************************************************

void display_decrement(void)
{
int i;
unsigned char j = 0xFF;

DDRC = 0xFF;                  //set Port C to output

for(i=0; i<256; i++)
  {
  j=(j-0x01);                 //decrement j by one
  PORTC = j;                  //assign char j to data port
  delay_30ms();               //wait 30 ms
  }
}


//***************************************************************************
//Initializes ADC
//***************************************************************************

void InitADC(void)
{
ADMUX = 0;                    //Select channel 0
ADCSRA = 0xC3;                //Enable ADC & start dummy conversion
                              //Set ADC module prescalar
                              //to 8 critical for
                              //accurate ADC results
```

```
while (!(ADCSRA & 0x10));    //Check if conversation is ready
ADCSRA |= 0x10;              //Clear conv rdy flag - set the bit
}

//*********************************************************************
//ReadADC: read analog voltage from analog-to-digital converter -

//the desired channel for conversion is passed in as an unsigned
//character variable.  The result is returned as a left justified,
//10 bit binary result.  The ADC prescalar must be set to 8 to
//slow down the ADC clock at higher external clock frequencies
//(10 MHz) to obtain accurate results.
//*********************************************************************

unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary voltage

ADMUX = channel;                    //Select channel
ADCSRA |= 0x43;                     //Start conversion
                                    //Set ADC module prescalar
                                    //to 8 critical for
                                    //accurate ADC results
while (!(ADCSRA & 0x10));           //Check if conversion is ready
ADCSRA |= 0x10;                     //Clear Conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL; //Read 8 low bits first-(important)
                                    //Read 2 high bits, multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low
| binary_weighted_voltage_high;
return binary_weighted_voltage;     //ADCH:ADCL
}

//*********************************************************************
//Displays voltage magnitude as LED level on PORTC
//*********************************************************************

void rain_gage(void)
```

```
{
unsigned int ADCValue;

ADCValue = readADC(0x00);

DDRC = 0xFF;                    //set Port C to output

if(ADCValue < 128)
  {
  PORTC = 0x01;
  }
else if(ADCValue < 256)
  {
  PORTC = 0x03;
  }
else if(ADCValue < 384)
  {
  PORTC = 0x07;
  }
else if(ADCValue < 512)
  {
  PORTC = 0x0F;
  }
else if(ADCValue < 640)
  {
   PORTC = 0x1F;
   }
else if(ADCValue < 768)
  {
  PORTC = 0x3F;
  }
else if(ADCValue < 896)
  {
  PORTC = 0x7F;
  }
else
  {
  PORTC = 0xAA;
  }
```

```
}

//*********************************************************************
//*********************************************************************
```

4.5.2 ADC RAIN GAGE INDICATOR WITH SPI

As we saw in the previous chapter, the Serial Peripheral Interface (SPI) may be used as an efficient method of displaying data. Recall that in this technique, data to be shifted out is placed into the SPI Data Register (SPDR) and then automatically shifted out of the Master Out Slave In (MOSI) pin of the SPI system.

A rain gage implemented using the SPI is illustrated in Figure 4.11.

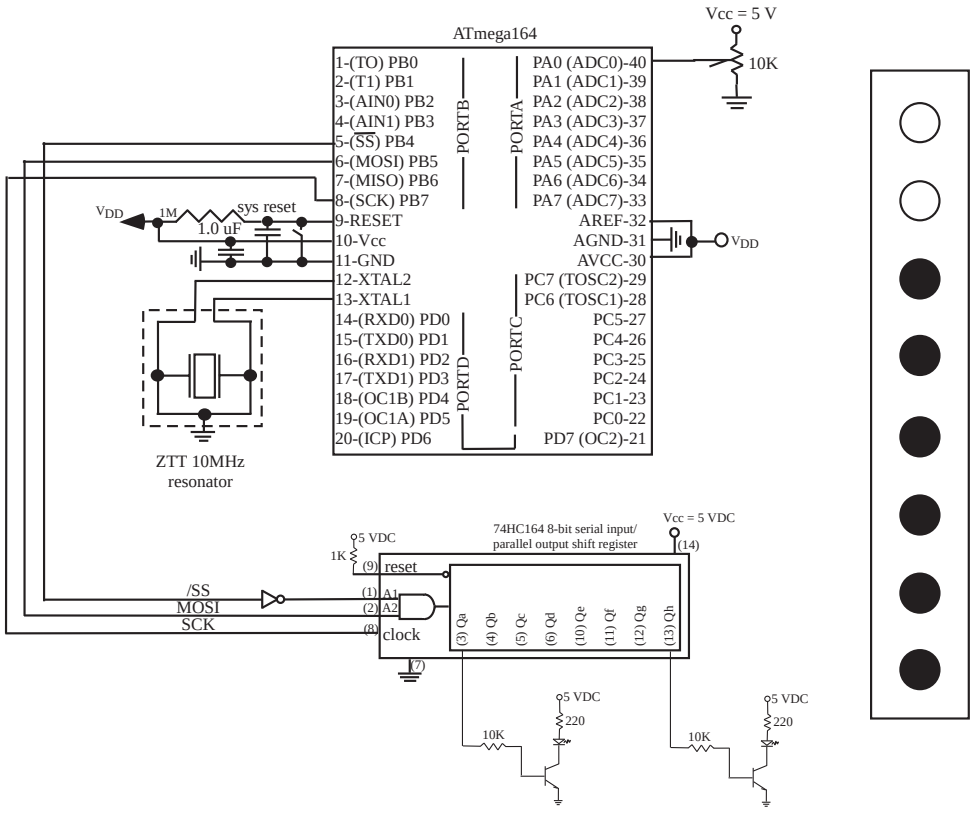


Figure 4.11: ADC with rain gage level indicator implemented via the SPI.

Much of the code used in the previous rain gage example using a parallel port may be re-used for this example. However, function provided in the previous chapter to initialize the SPI and

transmit via the SPI must also be included. Additionally, function **rain_gage** provided in the example must be modified to transmit the control signals to illuminate specific LEDs via the SPI as shown below.

```
//***************************************************************************
//Displays voltage magnitude as LED level via the SPI
//***************************************************************************

void rain_gage(void)
{
unsigned int ADCValue;

ADCValue = readADC(0x00);


if(ADCValue < 128)
  {
  SPDR = 0x01;
  }
else if(ADCValue < 256)
  {
  SPDR = 0x03;
  }
else if(ADCValue < 384)
  {
  SPDR = 0x07;
  }
else if(ADCValue < 512)
  {
  SPDR = 0x0F;
  }
else if(ADCValue < 640)
  {
   SPDR = 0x1F;
   }
else if(ADCValue < 768)
  {
  SPDR = 0x3F;
  }
else if(ADCValue < 896)
```

```
  {

  SPDR = 0x7F;
  }
else
  {
  SPDR = 0xAA;
  }
}

//*********************************************************************
```

4.5.3 TRANSMITTING ADC VALUES VIA THE USART OR SPI

In Chapter 2, we discussed the importance of knowing how variables are stored using a specific compiler. Many compilers will store variables in a similar manner; however, there are slight differences (e.g., the representation of a double variable).

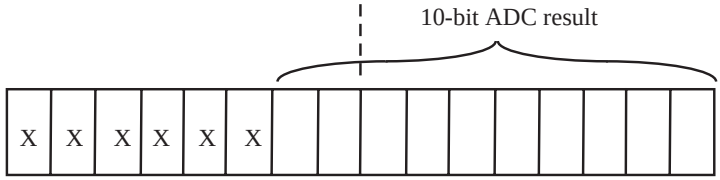
When transmitting results of an analog-to-digital conversion from one microcontroller to another, it is very important to understand the format of the transferred data and how it was stored by the ADC process. As an example, let's investigate the alternatives to transmit the result from an ADC conversion to another microcontroller using either the USART or SPI.

In Figure 4.12a), the result of an ADC conversion is stored as a right-justified 10-bit unsigned integer. To transmit this result to another microcontroller via the USART or SPI, we could consider two different alternatives. One alternative would be to shift the unsigned integer variable holding the ADC result two bits to the right and then casting the result as an unsigned char. This technique is illustrated in Figure 4.12b). The overall result will be an 8-bit ADC representation suitable for transmission via the USART or SPI. However, we lose the two least significant bits of resolution from the ADC conversion. Whether or not this is significant is application dependent.

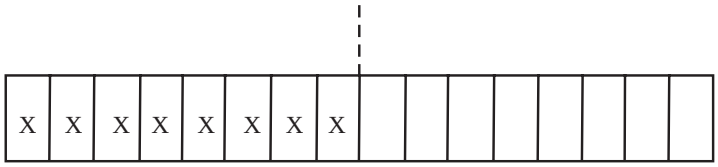
The other alternative would be to split the unsigned integer into two unsigned character variables. The two pieces could then be transmitted via the USART or SPI and then re-assembled back into an unsigned integer variable at the receiving microcontroller. This technique is illustrated in Figure 4.12c).

In the following code example, the function test integer (test_int) was developed to transmit four variables stored in EEPROM (centfail, cycles, doorlock, and closerev) via the SPI to another microcontroller. Note how each individual unsigned int variable is retrieved from EEPROM split into a most significant byte (ms) and least significant byte (ls) as unsigned characters and then transmitted them as two separate bytes via the SPI.

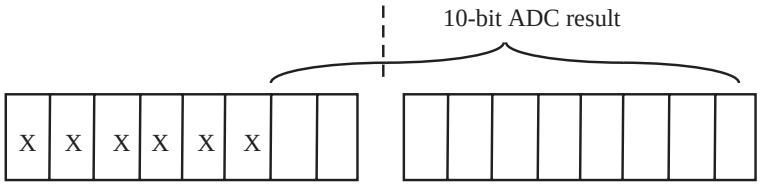
```
//Global declaration of variables stored in EEPROM

//eeprom initialization
```



a) original ADC result



b) shift two bit positions to right - transmit least significant byte



c) parse into byte size pieces, transmit each byte separately, re-assemble in receiving unit

Figure 4.12: Multi-byte serial transmission techniques.

```
#pragma data:eeprom
unsigned int centfail = 12301;    //#centrifugal switch failures
unsigned int cycles   = 12302;    //#cycles
unsigned int doorlock = 12303;    //#times door locked
unsigned int closerev = 12304;    //#close safety reversals
#pragma data:data

:
:
:
//*****************************************************************************
```

```
void test_int()
{
unsigned char   ms_centfail, ls_centfail;
unsigned char   ms_cycles,   ls_cycles;
unsigned char   ms_doorlock, ls_doorlock;
unsigned char   ms_closerev, ls_closerev;
unsigned char   master_control = 0x53;
unsigned int    test_value1,  test_value2;
unsigned int    test_value3,  test_value4;


//configure SPI for data transfer
DDRB = 0xB0;
PORTB = 0xF0;
//SPIE:0,SPE:1,DORD:0,MSTR:1,CPOL:0,CPHA:0,SPR:1,SPR0:1
SPCR = master_control;       //configure for SPI master

//Extract faults from EEPROM

//Centrifugal Switch Failures
EEPROM_READ((unsigned int) &centfail, test_value1); //read EEPROM

//split test_value into two unsigned char variables
ms_centfail = (unsigned char)((test_value1 >> 8) & (0x00FF));
ls_centfail = (unsigned char)(test_value1 & 0x00FF);


//Number of Cycles
EEPROM_READ((int) &cycles, test_value2);//read EEPROM

//split test_value into two unsigned char variables
ms_cycles = (unsigned char)((test_value2 >> 8) & (0x00FF));
ls_cycles = (unsigned char)(test_value2 & 0x00FF);


//Doorlocks
EEPROM_READ((int) &doorlock, test_value3);//read EEPROM

//split test_value into two unsigned char variables
```

```
ms_doorlock = (unsigned char)((test_value3 >> 8) & (0x00FF));
ls_doorlock = (unsigned char)(test_value3 & 0x00FF);


//Close Reversals
EEPROM_READ((int) &closerev, test_value4);//read EEPROM

//split test_value into two unsigned char variables
ms_closerev = (unsigned char)((test_value4 >> 8) & (0x00FF));
ls_closerev = (unsigned char)(test_value4 & 0x00FF);


//transmit data a byte at a time to SPI slave

//centfail
SPDR = ms_centfail;          //load ms_centfail into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag
SPDR = ls_centfail;          //load ls_centfail into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag

//cycles
SPDR = ms_cycles;            //load ms_cycles into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag
SPDR = ls_cycles;            //load ls_cycles into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag

//doorlock
SPDR = ms_doorlock;          //load ms_doorlock into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag
SPDR = ls_doorlock;          //load ls_doorlock into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag

//closerev
```

```
SPDR = ms_closerev;          //load ms_closerev into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag
SPDR = ls_closerev;          //load ls_closerev into SPI data register
while(!(SPSR & 0x80));       //wait for transmission complete SPIF
dummy = SPDR;                //clears SPIF flag
}
//*****************************************************************************
```

At the receiving microcontroller, the two bytes must be reassembled into an unsigned int variable.

4.5.4 ONE-BIT ADC - THRESHOLD DETECTOR

A threshold detector circuit or comparator configuration contains an operational amplifier employed in the open loop configuration. That is, no feedback is provided from the output back to the input to limit gain. A threshold level is applied to one input of the op amp. This serves as a comparison reference for the signal applied to the other input. The two inputs are constantly compared to one another. When the input signal is greater than the set threshold value, the op amp will saturate to a value slightly less than +Vcc as shown in Figure 4.13a). When the input signal falls below the threshold, the op amp will saturate at a voltage slightly greater than –Vcc. If a single-sided op amp is used in the circuit (e.g., LM324), the –Vcc supply pin may be connected to ground. In this configuration, the op map provides for a one-bit ADC circuit.

A bank of threshold detectors may be used to construct a multi-channel threshold detector as shown in Figure 4.13c). This provides a flash converter type ADC. It is a hardware version of a rain gage indicator. In Chapter 1, we provided a 14-channel version for use in a laboratory instrumentation project.

4.6 DIGITAL-TO-ANALOG CONVERSION (DAC)

Once a signal is acquired to a digital system with the help of the analog-to digital conversion process and has been processed, frequently the processed signal is converted back to another analog signal. A simple example of such a conversion occurs in digital audio processing. Human voice is converted to a digital signal, modified, processed, and converted back to an analog signal for people to hear. The process to convert digital signals to analog signals is completed by a digital-to-analog converter. The most commonly used technique to convert digital signals to analog signals is the summation method shown in Figure 4.14.

With the summation method of digital-to-analog conversion, a digital signal, represented by a set of ones and zeros, enters the digital-to-analog converter from the most significant bit to the least significant bit. For each bit, a comparator checks its logic state, high or low, to produce a clean digital bit, represented by a voltage level. Typically, in a microcontroller context, the voltage level is +5 or 0 volts to represent logic one or logic zero, respectively. The voltage is then multiplied by a

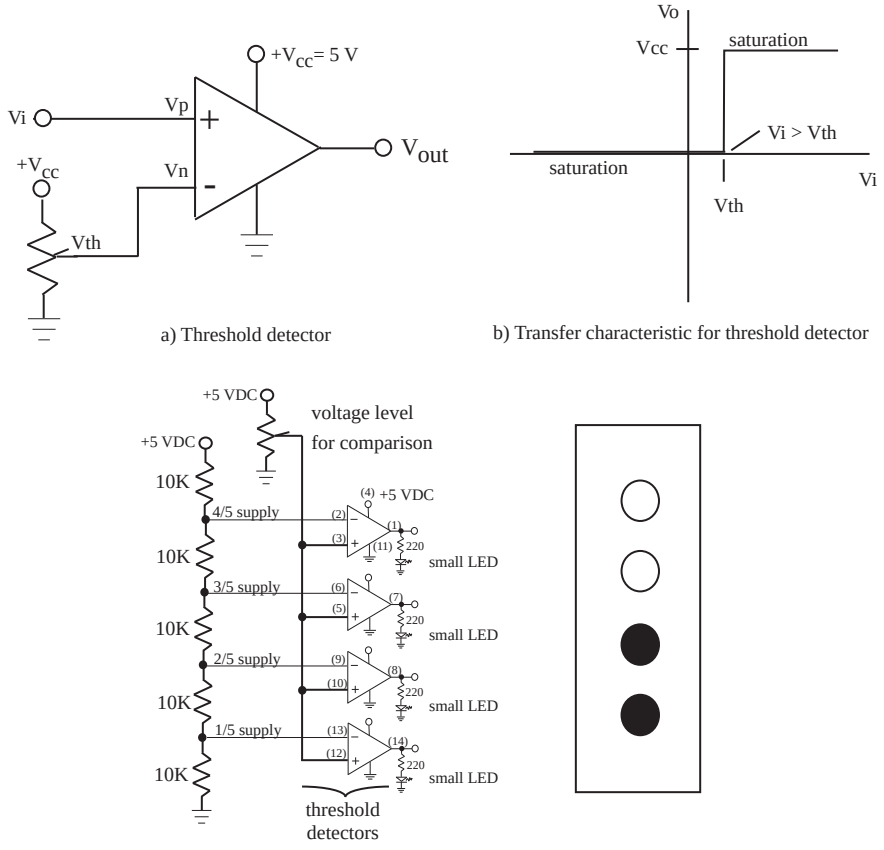


Figure 4.13: One-bit ADC - threshold detector.

scalar value based on its significant position of the digital signal as shown in Figure 4.14. Once all bits for the signal have been processed, the resulting voltage levels are summed together to produce the final analog voltage value. Notice that the production of a desired analog signal may involve further signal conditioning such as a low pass filter to 'smooth' the quantized analog signal and a transducer interface circuit to match the output of the digital-to-analog converter to the input of an output transducer.

A microcontroller can be equipped with a wide variety of DAC configurations including a:

- Single channel, 8-bit DAC connected via a parallel port (e.g., Motorola MC1408P8)
- Quad channel, 8-bit DAC connected via a parallel port (e.g., Analog Devices AD7305)

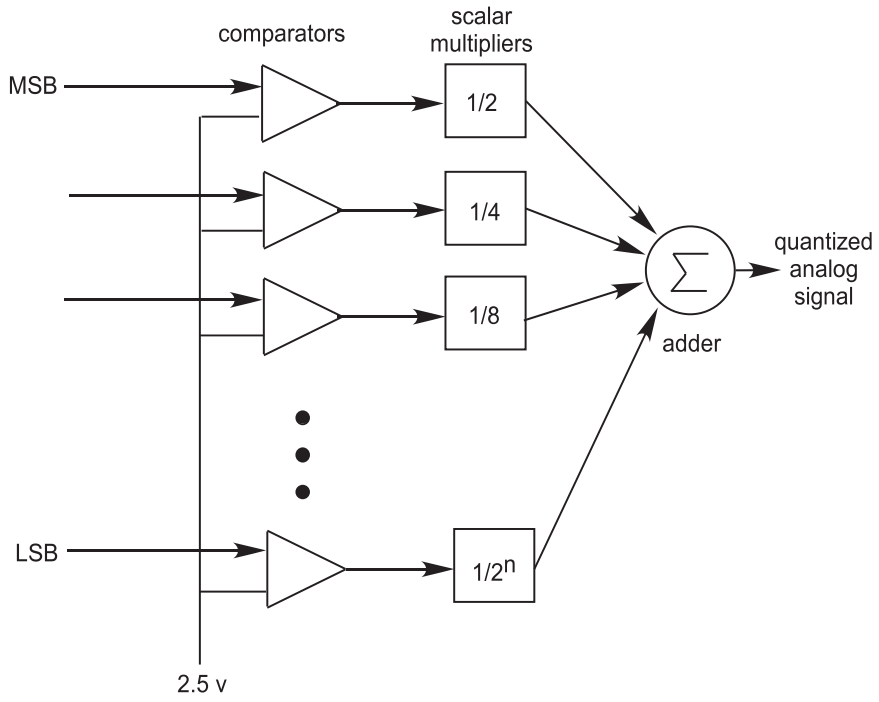


Figure 4.14: A summation method to convert a digital signal into a quantized analog signal. Comparators are used to clean up incoming signals and the resulting values are multiplied by a scalar multiplier and the results are added to generate the output signal. For the final analog signal, the quantized analog signal should be connected to a low pass filter followed by a transducer interface circuit.

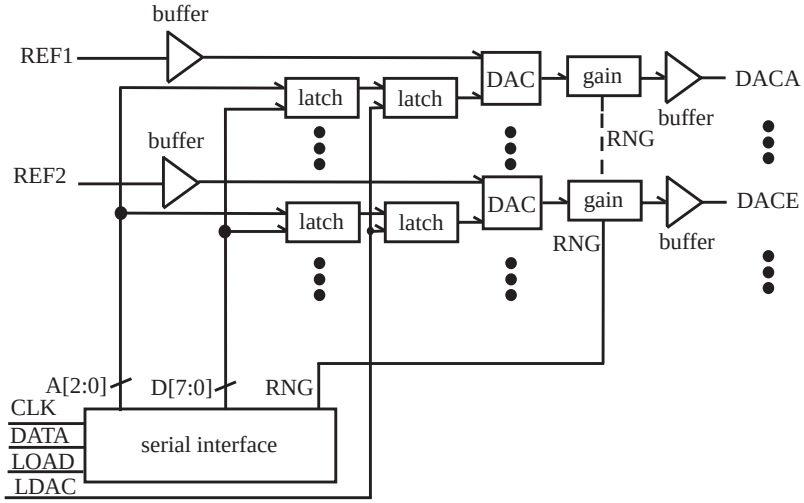
- Quad channel, 8-bit DAC connected via the SPI (e.g., Analog Devices AD7304)
- Octal channel, 8-bit DAC connected via the SPI (e.g., Texas Instrument TLC5628)

Space does not allow an in depth look at each configuration, but we will examine the TLC5628 in more detail.

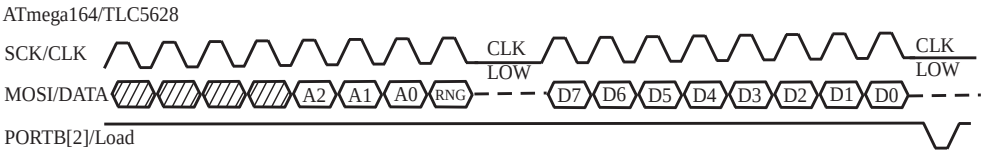
4.6.1 OCTAL CHANNEL, 8-BIT DAC VIA THE SPI

The Texas Instruments (TI) TLC5628 is an eight channel, 8-bit DAC connected to the microcontroller via the SPI. Reference Figure 4.15a). It has a wide range of features packed into a 16-pin chip including (Texas, 1997):

- Eight individual, 8-bit DAC channels,
- Operation from a single 5 VDC supply,



a) TLC5628 octal 8-bit DACs



b) TLC5628 timing diagram

A[2:0]	DAC Updated
000	DACA
001	DACB
010	DACC
011	DACD
100	DACE
101	DACF
110	DACG
111	DACH

D[7:0]	Output Voltage
0000_0000	GND
0000_0001	(1/256) x REF(1+RNG)
:	:
:	:
:	:
:	:
:	:
1111_1111	(255/256) x REF (1+RNG)

c) TLC5628 bit assignments.

Figure 4.15: Eight channel DAC. Adapted from (Texas, 1997).

- Data load via the SPI interface,
- Programmable gain (1X or 2X), and
- A 12-bit command word to program each DAC.

Figure 4.15b) provides the interconnection between the ATmega164 and the TLC5628. The ATmega164 SPI's SCK line is connected to the TLC5628: the MOSI to the serial data line and PORTB[2] to the Load line. As can be seen in the timing diagram, two sequential bytes are sent from the ATmega164 to select the appropriate DAC channel and to provide updated data to the DAC. The Load line is pulsed low to update the DAC. In this configuration, the LDAC line is tied low. The function to transmit data to the DAC is left as an assignment for the reader at the end of the chapter.

4.7 SUMMARY

In this chapter, we presented the differences between analog and digital signals and used this knowledge to discuss three sub-processing steps involved in analog to digital converters: sampling, quantization, and encoding. We also presented the quantization errors and the data rate associated with the ADC process. The dynamic range of an analog-to-digital converter, one of the measures to describe a conversion process, was also presented. We then presented the successive-approximation converter. Transducer interface design concepts were then discussed along with supporting information on operational amplifier configurations. We then reviewed the operation, registers, and actions required to program the ADC system aboard the ATmega164. We concluded the chapter with a discussion of the ADC process and an implementation using a multi-channel DAC connected to the ATmega164 SPI system.

4.8 CHAPTER PROBLEMS

4.1. Given a sinusoid with 500 Hz frequency, what should be the minimum sampling frequency for an analog-to-digital converter if we want to faithfully reconstruct the analog signal after the conversion?

4.2. If 12 bits are used to quantize a sampled signal, what is the number of available quantized levels? What will be the resolution of such a system if the input range of the analog-to-digital converter is 10 V?

4.3. Given the 12 V input range of an analog-to-digital converter and the desired resolution of 0.125 V, what should be the minimum number of bits used for the conversion?

4.4. Perform a trade-off study on the four technologies used for the analog-to-digital conversion. Use cost, conversion time, and accuracy as the list of criteria.

4.5. Investigate the analog-to-digital converters in your audio system. Find the sampling rate, the quantization bits, and the technique used for the conversion.

4.6. A flex sensor provides 10K ohm of resistance for 0 degrees flexure and 40K ohm of resistance for 90 degrees of flexure. Design a circuit to convert the resistance change to a voltage change (Hint: consider a voltage divider). Then design a transducer interface circuit to convert the output from the flex sensor circuit to voltages suitable for the ATmega164 ADC system.

4.7. If an analog signal is converted by an analog-to-digital converter to a binary representation and then back to an analog voltage using a DAC, will the original analog input voltage be the same as the resulting analog output voltage? Explain.

4.8. Derive each of the characteristic equations for the classic operation amplifier configurations provided in Figure 4.4.

4.9. If a resistor was connected between the non-inverting terminal and ground in the inverting amplifier configuration of Figure 4.4a), how would the characteristic equation change?

4.10. A photodiode provides a current proportional to the light impinging on its active area. What classic operational amplifier configuration should be used to current the diode output to a voltage?

4.11. Does the time to convert an analog input signal to a digital representation vary in a successive-approximation converter relative to the magnitude of the input signal?

4.12. Write a function to transmit control signals to the TLC5628 via the ATmega164 SPI system. The function should take as input arguments the DAC channel to be updated, the range, and the data to be passed to the DAC.

REFERENCES

Electrical Signals and Systems, fourth edition, Department of Electrical Engineering, USAF Academy, Colorado, McGraw-Hill Companies, Primis Custom Publishing.

Atmel 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash, ATmega164, ATmega164L, data sheet: 2466L-AVR-06/05, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.

S. Barrett and D. Pack, *Microcontrollers Fundamentals for Engineers and Scientists,* Morgan and Claypool Publishers, 2006.

S. Barrett and D. Paack, *Atmel AVR Microcontroller Primer Programming and Interfacing,* Morgan and Claypool Publishers, 2008.

Roland Thomas and Albert Rosa, *The Analysis and Design of Linear Circuits, Fourth Edition,* Wiley & Sons, Inc., New York, 2003.

Daniel Pack and Steven Barrett, *Microcontroller Theory and Applications: HC12 and S12,* Prentice Hall, 2ed, Upper Saddle River, New Jersey 07458, 2008.

Alan Oppenheim and Ronald Schafer, *Discrete-time Signal Processing, Second Edition,* Prentice Hall, Upper Saddle River, New Jersey, 1999.

John Enderle, Susan Blanchard, and Joseph Bronzino, *Introduction to Biomedical Engineering,* Academic Press, 2000.

L. Faulkenberry, *An Introduction to Operational Amplifiers,* John Wiley & Sons, New York, 1977.

P. Horowitz and W. Hill, *The Art of Electronics,* Cambridge University Press, 1989.

Introduction to Operational Amplifiers with Linear Integrated Circuit Applications, 1982.

D. Stout and M. Kaufman, *Handbook of Operational Amplifier Circuit Design*, McGraw-Hill Book Company, 1976.

S. Franco, *Design with Operational Amplifiers and Analog Integrated Circuits, third edition,* McGraw-Hill Book Company, 2002.

TLC5628C, TLC5628I Octal 8-bit Digital-to-Analog Converters, Texas Instruments, Dallas, TX, 1997.

CHAPTER 5

Interrupt Subsystem

Objectives: After reading this chapter, the reader should be able to

- Understand the need of a microcontroller for interrupt capability.
- Describe the general microcontroller interrupt response procedure.
- Describe the ATmega164 interrupt features.
- Properly configure and program an interrupt event for the ATmega164.
- Use the interrupt system to implement a real time clock.
- Employ the interrupt system as a component in an embedded system.

5.1 INTERRUPT THEORY

A microcontroller normally executes instructions in an orderly fetch-decode-execute sequence as dictated by a user-written program as shown in Figure 5.1. However, the microcontroller must be equipped to handle unscheduled (although planned), higher priority events that might occur inside or outside the microcontroller. To process such events, a microcontroller requires an interrupt system.

The interrupt system onboard a microcontroller allows it to respond to higher priority events. Appropriate responses to these events may be planned, but we do not know when these events will occur. When an interrupt event occurs, the microcontroller will normally complete the instruction it is currently executing and then transition program control to interrupt event specific tasks. These tasks, which resolve the interrupt event, are organized into a function called an interrupt service routine (ISR). Each interrupt will normally have its own interrupt specific ISR. Once the ISR is complete the microcontroller will resume processing where it left off before the interrupt event occurred.

5.2 ATMEGA164 INTERRUPT SYSTEM

The ATmega164 is equipped to handle a powerful and flexible complement of 31 interrupt sources. Three of the interrupts originate from external interrupt sources while the remaining 28 interrupts support the efficient operation of peripheral subsystems aboard the microcontroller. The ATmega164 interrupt sources are shown in Figure 5.2. The interrupts are listed in descending order of priority. As you can see the RESET has the highest priority, followed by the external interrupt request pins

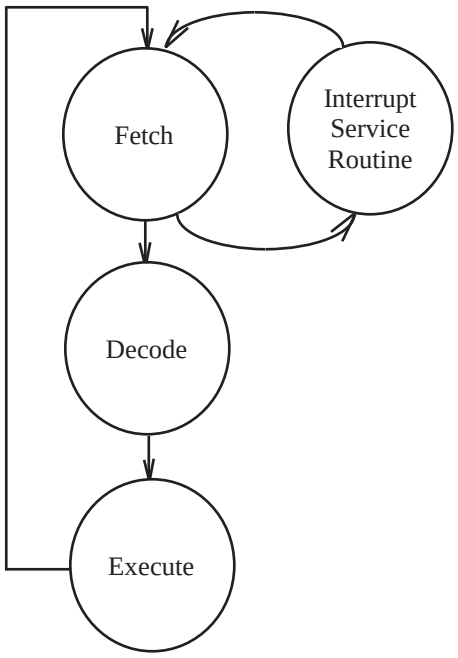


Figure 5.1: Microcontroller Interrupt Response.

INT0 (pin 16), INT1 (pin 17), and INT2 (pin 3). The remaining interrupt sources are internal to the ATmega164.

When an interrupt occurs, the microcontroller completes the current instruction, stores the address of the next instruction on the stack, and starts executing instructions in the designated interrupt service routine (ISR) corresponding to the particular interrupt source. It also turns off the interrupt system to prevent further interrupts while one is in progress. The execution of the ISR is performed by loading the beginning address of the interrupt service routine specific for that interrupt into the program counter. The interrupt service routine will then commence. Execution of the ISR continues until the return from interrupt instruction (reti) is encountered. Program control then reverts back to the main program.

5.3 PROGRAMMING AN INTERRUPT SYSTEM

To program an interrupt the user is responsible for the following actions:

Vector No.	Prog Addr	Source	Interrupt Definition
1	$0000	RESET	External pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	PCINT0	Pin Change Interrupt Request 0
6	$000A	PCINT1	Pin Change Interrupt Request 1
7	$000C	PCINT2	Pin Change Interrupt Request 2
8	$000E	PCINT3	Pin Change Interrupt Request 3
9	$0010	WDT	Watchdog Time-out Interrupt
10	$0012	TIMER2_COMPA	Timer/Counter2 Compare Match A
11	$0014	TIMER2_COMPB	Timer/Counter2 Compare Match B
12	$0016	TIMER2_OVF	Timer/Counter2 Overflow
13	$0018	TIMER1_CAPT	Timer/Counter1 Capture Event
14	$001A	TIMER1_COMPA	Timer/Counter1 Compare Match A
15	$001C	TIMER1_COMPB	Timer/Counter1 Compare Match B
16	$001E	TIMER1_OVF	Timer/Counter1 Overflow
17	$0020	TIMER0_COMPA	Timer/Counter0 Compare Match A
18	$0022	TIMER0_COMPB	Timer/Counter0 Compare Match B
19	$0024	TIMER0_OVF	Timer/Counter0 Overflow
20	$0026	SPI_STC	SPI Serial Transfer Complete
21	$0028	USART0_RX	USART0 Rx Complete
22	$002A	USART0_UDRE	USART0 Data Register Empty
23	$002C	USART0_TX	USART0 Tx Complete
24	$002E	ANALOG_COMP	Analog Comparator
25	$0030	ADC	ADC Conversion Complete
26	$0032	EE_READY	EEPROM Ready
27	$0034	TWI	2-wire Serial Interface
28	$0036	SPM_READY	Store Program Memory Ready
29	$0038	USART1_RX	USART1 Rx Complete
30	$003A	USART1_UDRE	USART1 Data Register Empty
31	$003C	USART1_TX	USART1 Tx Complete

Figure 5.2: Atmel AVR ATmega164 Interrupts. (Adapted from figure used with permission of Atmel, Incorporated.)

- Ensure the interrupt service routine for a specific interrupt is tied to the correct interrupt vector address, which points to the starting address of the interrupt service routine.
- Ensure the interrupt system has been globally enabled. This is accomplished with the assembly language instruction SEI.
- Ensure the specific interrupt subsystem has been locally enabled.
- Ensure the registers associated with the specific interrupt have been configured correctly.

In the next two examples that follow, we illustrate how to accomplish these steps. We use the ImageCraft ICC AVR compiler, which contains excellent support for interrupts. Other compilers have similar features.

5.4 APPLICATION

In this section, we provide two representative samples of writing interrupts for the ATmega164. We preset both an externally generated interrupt event and also one generated from within the microcontroller. The ImageCraft ICC AVR compiler uses the following syntax to link an interrupt service routine to the correct interrupt vector address:

```
#pragma interrupt_handler timer_handler:4

void timer_handler(void)
{
:
:
}
```

As you can see, the #pragma with the reserved word **interrupt_handler** is used to communicate to the compiler that the routine name that follows is an interrupt service routine. The number that follows the ISR name corresponds to the interrupt vector number in Figure 5.2. The ISR is then written like any other function. It is important that the ISR name used in the #pragma instruction matches the name of the ISR in the function body. Since the compiler knows the function is an ISR, it will automatically place the RETI instruction at the end of the ISR.

5.4.1 EXTERNAL INTERRUPTS

The external interrupts INT0 (pin 16), INT1 (pin 17), and INT2 (pin (3) trigger an interrupt within the ATmega164 when an user-specified external event occurs at the pin associated with the specific interrupt. Interrupts INT0 and INT1 may be triggered with a level or an edge signal. Whereas, interrupt INT2 is edge-triggered only. The specific settings for each interrupt is provided in Figure 5.3.

Provided below is the code snapshot to configure an interrupt for INT0. In this specific example, an interrupt will occur when a positive edge transition occurs on the ATmega164 INT0 external interrupt pin.

```
//interrupt handler definition
#pragma interrupt_handler int0_ISR:2

//function prototypes
void int0_ISR(void);
void initialize_interrupt0(void);


//***************************************************************************
```

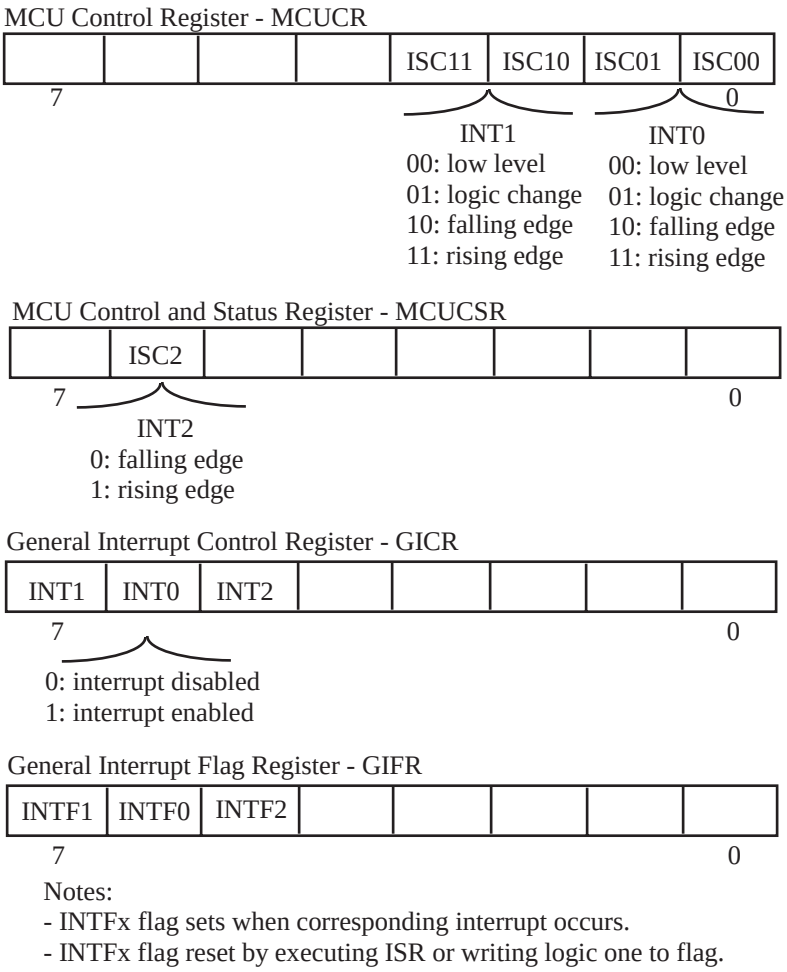


Figure 5.3: Interrupt INT0, INT1, and INT2 Registers.

```
//The following function call should be inserted in the main program to
//initialize the INT0 interrupt to respond to a positive edge trigger.
//This function should only be called once.

:
initialize_interrupt0();
```

```
:



//***************************************************************************

//function definitions

//***************************************************************************
//initialize_interrupt0:  initializes interrupt INT0.
//Note: stack is automatically initialized by the compiler
//***************************************************************************

void initialize_interrupt0(void)       //initialize interrupt 0
{
DDRD = 0xFB;                           //set PD2 (int0) as input
PORTD &= ~0x04;                        //disable pullup resistor PD2
GICR = 0x40;                           //enable int 0
MCUCR = 0x03;                          //set for positive edge trigger
asm("SEI");                            //global interrupt enable
}


//***************************************************************************
//int0_ISR: interrupt service routine for INT0
//***************************************************************************

void int0_ISR(void)
{

//Insert interrupt specific actions here.

}
```

5.4.2 INTERNAL INTERRUPT

In this example, we use Timer/Counter0 to provide prescribed delays within our program. Recall that Timer/Counter0 is an eight bit timer. It rolls over every time it receives 256 timer clock "ticks." There is an interrupt associated with the Timer/Counter0 overflow. If activated, the interrupt will occur every time the contents of the Timer/Counter0 transitions from 255 back to 0 count. We

can use this overflow interrupt as a method of keeping track of real clock time (hours, minutes, and seconds) within a program. In this specific example, we use the overflow to provide precision program delays.

In this example, the ATmega164 is being externally clocked by a 10 MHz ceramic resonator. The resonator frequency is further divided by 256 using the clock select bits CS[2:1:0] in the Timer/Counter Control Register 0 (TCCR0). When CS[2:1:0] are set for [1:0:0], the incoming clock source is divided by 256. This provides a clock "tick" to Timer/Counter0 every 25.6 microseconds. Therefore, the eight bit Timer/Counter0 will rollover every 256 clock "ticks" or every 6.55 ms.

To create a precision delay, we write a function called delay. The function requires an unsigned integer parameter value indicating how many 6.55 ms interrupts the function should delay. The function stays within a while loop until the desired number of interrupts has occurred. For example, to delay one second the function would be called with the parameter value "153." That is, it requires 153 interrupts occurring at 6.55 ms intervals to generate a one second delay.

The code snapshots to configure the Time/Counter0 Overflow interrupt is provided below along with the associated interrupt service routine and the delay function.

```
//function prototypes*************************************************
                                        //delay specified number 6.55ms
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
 //initialize timer0 overflow interrupt


//interrupt handler definition****************************************
                                        //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19



//global variables****************************************************
unsigned int   input_delay;             //counts number of Timer/Counter0
                                        //Overflow interrupts

//main program********************************************************

void main(void)
{
init_timer0_ovf_interrupt();
 //initialize Timer/Counter0 Overflow
```

```
 //interrupt - call once at beginning
                                        //of program
:
:
delay(153);                             //1 second delay

}

//***************************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow interrupt is being
//employed as a time base for a master timer for this project.  The ceramic
//resonator  operating at 10 MHz is divided by 256. The 8-bit Timer0
//register (TCNT0) overflows every 256 counts or every 6.55 ms.
//***************************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0 = 0x04; //divide timer0 timebase
by 256, overflow occurs every 6.55ms
TIMSK = 0x01; //enable timer0 overflow interrupt
asm("SEI");   //enable global interrupt
}

//***************************************************************************
//***************************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//***************************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;                          //increment overflow counter
}

//***************************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay function
//provides the specified delay as the number of 6.55 ms "clock ticks" from
```

```
//the Timer/Counter0 Overflow interrupt.
//Note: this function is only valid when using a 10 MHz crystal or ceramic
//      resonator
//*************************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                          //reset timer0
input_delay = 0;                       //reset timer0 overflow counter
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;
 //wait for specified number of interrupts
  }
}

//*************************************************************************
```

5.5 FOREGROUND AND BACKGROUND PROCESSING

A microcontroller can only process a single instruction at a time. It processes instructions in a fetch-decode-execute sequence as determined by the program and its response to external events. In many cases, a microcontroller has to process multiple events seemingly simultaneously. How is this possible with a single processor?

Normal processing accomplished by the microcontroller is called foreground processing. An interrupt may be used to periodically break into foreground processing, 'steal' some clock cycles to accomplish another event called background processing, and then return processor control back to the foreground process.

As an example, a microcontroller controlling access for an electronic door must monitor input commands from a user and generate the appropriate PWM signal to open and close the door. Once the door is in motion, the controller must monitor door motor operation for obstructions, malfunctions, and other safety related parameters. This may be accomplished using interrupts. In this example, the microcontroller is responding to user input status in the foreground while monitoring safety related status in the background using interrupts as illustrated in Figure 5.4.

5.6 INTERRUPT EXAMPLES

In this section, we provide several varied examples on using interrupts internal and external to the microcontroller.

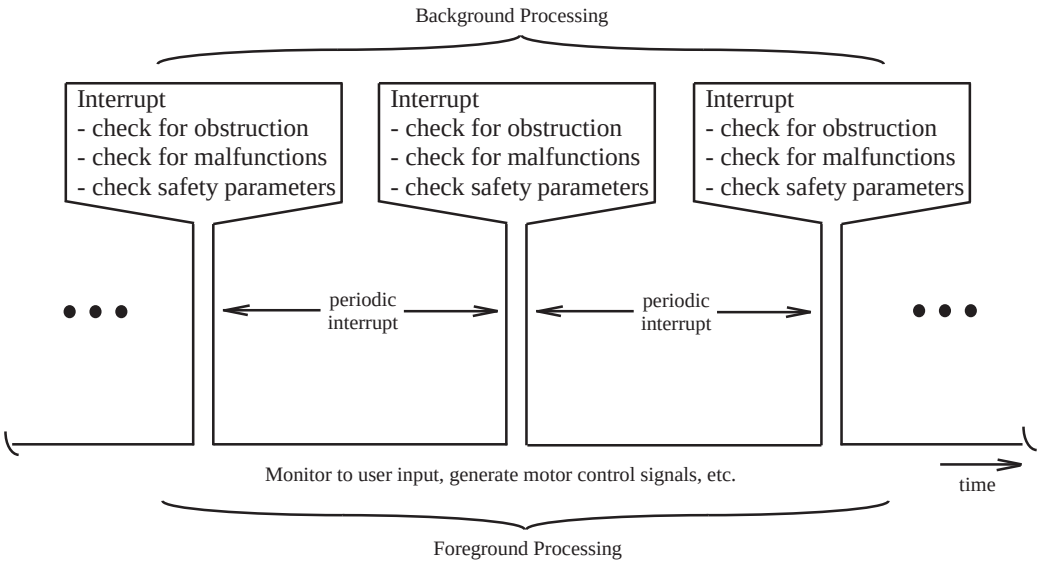


Figure 5.4: Interrupt used for background processing. The microcontroller responds to user input status in the foreground while monitoring safety related status in the background using interrupts.

5.6.1 REAL TIME CLOCK

A microcontroller only 'understands' elapsed time in reference to its timebase clock ticks. To keep track of clock time in seconds, minutes, hours etc., a periodic interrupt may be generated for use as a 'clock tick' for a real time clock. In this example, we use the Timer 0 overflow to generate a periodic clock tick very 6.55 ms. The ticks are counted in reference to clock time variables and may be displayed on a liquid crystal display. This is also a useful technique for generating very long delays in a microcontroller.

```
//function prototypes*****************************************************
                                        //delay specified number 6.55ms
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
//initialize timer0 overflow interrupt
```

```
//interrupt handler definition*********************************************
#pragma interrupt_handler timer0_interrupt_isr:19


//global variables*********************************************************
unsigned int days_ctr, hrs_ctr, mins_ctr, sec_ctr, ms_ctr;

//main program*************************************************************

void main(void)
{
day_ctr = 0; hr_ctr = 0; min_ctr = 0; sec_ctr = 0; ms_ctr = 0;


init_timer0_ovf_interrupt();
 //initialize Timer/Counter0 Overflow

 //interrupt - call once at beginning
                                       //of program
while(1)
  {
  ;                                    //wait for interrupts
  }
}

//***************************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow interrupt is being
//employed as a time base for a master timer for this project.  The ceramic
//resonator  operating at 10 MHz is divided by 256. The 8-bit Timer0 register
//(TCNT0) overflows every 256 counts or every 6.55 ms.
//***************************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0 = 0x04; //divide timer0 timebase
by 256, overflow occurs every 6.55ms
```

```
TIMSK = 0x01; //enable timer0 overflow interrupt
asm("SEI");   //enable global interrupt
}

//***************************************************************************
//***************************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//***************************************************************************

void timer0_interrupt_isr(void)
{

//Update millisecond counter
ms_ctr = ms_ctr + 1;                    //increment ms counter


//Update second counter
//counter equates to 1000 ms at 154
if(ms_ctr == 154)
  {
  ms_ctr = 0;                           //reset ms counter
  sec_ctr = sec_ctr + 1;                //increment second counter
  }

//Update minute counter
if(sec_ctr == 60)
  {
  sec_ctr = 0;                          //reset sec counter
  min_ctr = min_ctr + 1;                //increment min counter
  }

//Update hour counter
if(min_ctr == 60)
  {
  min_ctr = 0;                          //reset min counter
  hr_ctr  = hr_ctr + 1;                 //increment hr counter
  }
```

```
//Update day counter
if(hr_ctr == 24)
  {
  hr_ctr = 0;                          //reset hr counter
  day_ctr  = day_ctr + 1;              //increment day counter
  }
}

//***************************************************************************
```

5.6.2 INTERRUPT DRIVEN USART

Example. You have been asked to evaluate a new positional encoder technology. The encoder provides 12-bit resolution. The position data is sent serially at 9600 Baud as two sequential bytes as shown in Figure 5.5. The actual encoder is new technology and production models are not available for evaluation.

Since the actual encoder is not available for evaluation, another Atmel ATmega164 will be used to send signals in identical format and Baud rate as the encoder. The test configuration is illustrated in Figure 5.6. The ATmega164 on the bottom serves as the positional encoder. The microcontroller is equipped with two pushbuttons at PB0 and PB1. The pushbutton at PB0 provides a debounced input to open a simulated door and increment the positional encoder. The pushbutton at PB1 provides debounced input to close a simulated door and decrement the positional encoder. The current count of the encoder (eight most significant bits) is fed to a digital-to-analog converter (DAC0808) to provide an analog representation.

The positional data from the encoder is sent out via the USART in the format described in Figure 5.5. The TTL compatible serial data from the USART is shifted to an RS-232 compatible signal by the MAX232I and then reconverted back to a TTL compatible signal by the Schottky diode configuration. The top ATmega164 receives the positional data using interrupt driven USART techniques. The current position is converted to an analog signal via the DAC. The transmitted and received signals may be compared at the respective DAC outputs.

Provided below is the code for the ATmega164 that serves as the encoder simulator followed by the code for receiving the data.

```
//***************************************************************************
//author: Steven Barrett, Ph.D., P.E.
//last revised: Aug 15, 2009
//file: encode.c
//target controller: ATMEL ATmega164
//
//ATMEL AVR ATmega164PV Controller Pin Assignments
```

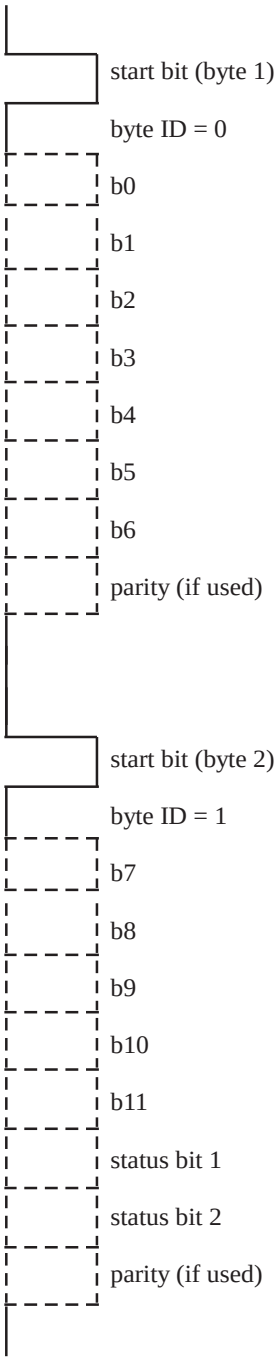


Figure 5.5: Encoder data format. The position data is sent serially at 9600 Baud as two sequential bytes.

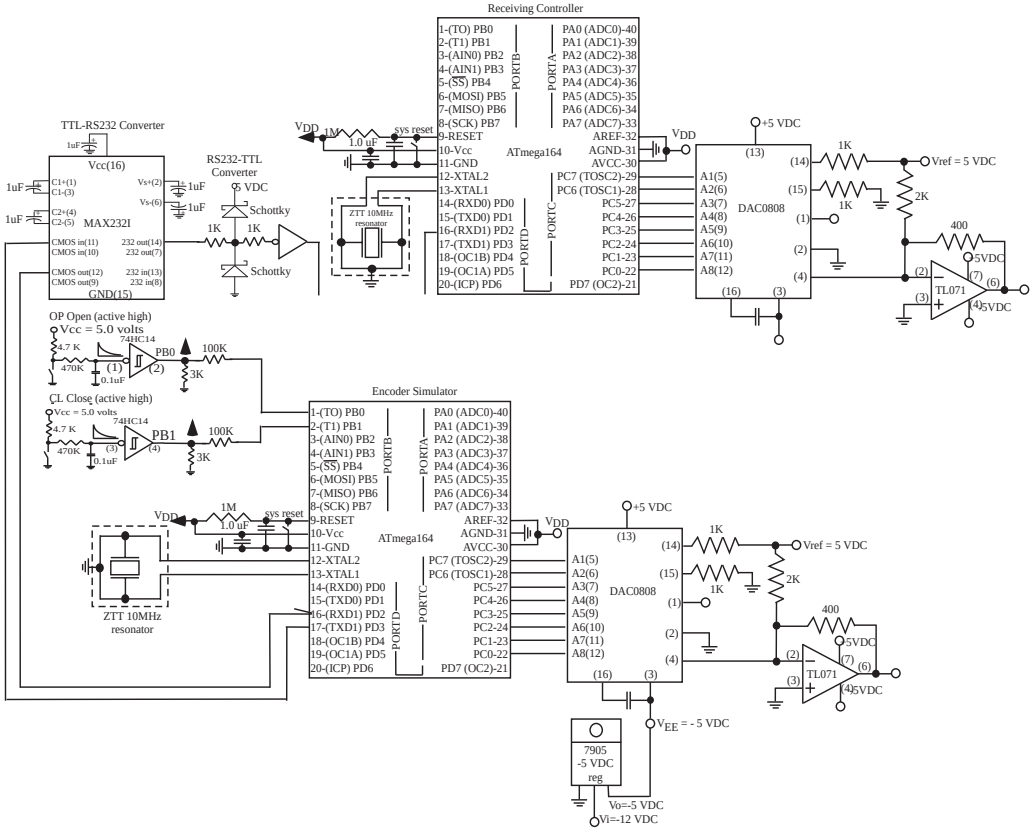


Figure 5.6: Encoder test configuration.

```
//Chip Port Function I/O Source/Dest Asserted Notes
//PORTB:
//Pin 1 PB0 to active high RC debounced switch - Open
//Pin 2 PB1 to active high RC debounced switch - Close
//Pin 9 Reset - 1M resistor to Vcc, tact switch to ground, 1.0 uF to ground
//Pin 10 Vcc - 1.0 uF to ground

//Pin 11 Gnd
//Pin 12 ZTT-10.00MT ceramic resonator connection
//Pin 13 ZTT-10.00MT ceramic resonator connection
//PORTC: Connections to DAC0808
//Pin 30 AVcc to Vcc
```

```
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//*********************************************************************
//include files*******************************************************

//ATMEL register definitions for ATmega324
#include<iom164v.h>
#include<macros.h>

//function prototypes************************************************

 //delay specified number 6.55ms int
void delay(unsigned int number_of_6_55ms_interrupts);
void initialize_ports(void);           //initializes ports
void power_on_reset(void);             //returns system to startup state
void init_timer0_ovf_interrupt(void);
 //used to initialize timer0 overflow
void InitUSART(void);
void USART_TX(unsigned char data);
                                       //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19

//main program*******************************************************
//The main program checks PORTB for user input activity.  If new activity
//is found, the program responds.

//global variables
unsigned char   old_PORTB = 0x08;      //present value of PORTB
unsigned char   new_PORTB;             //new values of PORTB
unsigned int    door_position = 0;

void main(void)
{
power_on_reset();
 //returns system to startup condition
initialize_ports();
 //return LED configuration to default
InitUSART();
                                       //limited startup features
```

```
//main activity loop - processor will continually cycle through loop for new
//activity.  Activity initialized by external signals presented to PORTB or
//PORTD. PORTB[0] is active low, all other inputs PORTB[7:1] are active high
while(1)
  {
  _StackCheck();                        //check for stack overflow
  read_new_input();
 //read input status changes on PORTB
  }
}//end main

//Function definitions
//***************************************************************************
//power_on_reset:
//***************************************************************************

void power_on_reset(void)
{
initialize_ports();                     //initialize ports
init_timer0_ovf_interrupt();            //initialize Timer0 to serve as
                                        //time "clock tick"
}

//***************************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************************

void initialize_ports(void)
{
//PORTA
DDRA=0xff;                              //PORTA[7:0] output
PORTA=0x00;                             //Turn off pull ups

//PORTB
DDRB=0xfc;
 //PORTB[7-2] output, PORTB[1:0] input
PORTB=0x00;                             //disable PORTB pull-up resistors

//PORTC
```

```
DDRC=0xff;                              //set PORTC[7-0] as output
PORTC=0x00;                             //init low

//PORTD
DDRD=0xff;                              //set PORTD[7-0] as output
PORTD=0x00;                             //initialize low
}

//*************************************************************************
//*************************************************************************
//read_new_input: functions polls PORTB for a change in status.  If status
//change has occurred, appropriate function for status change is called
//Pin 1 PB0 to active high RC  debounced switch - Open
//Pin 2 PB1 to active high RC debounced switch - Close
//*************************************************************************

void read_new_input(void)
{
unsigned int   gate_position;
 //measure instantaneous position of gate
unsigned int i;
unsigned char ms_door_position, ls_door_position, DAC_data;

new_PORTB = (PINB);
if(new_PORTB != old_PORTB){
  switch(new_PORTB){                    //process change in PORTB input

    case 0x01:                          //Open
      while(PINB == 0x01)
        {
        //split into two bytes
        ms_door_position=(unsigned char)(((door_position >> 6)
                                          &(0x00FF))|0x01);
        ls_door_position=(unsigned char)(((door_position << 1)
                                          &(0x00FF))&0xFE);
        //TX data to USART
        USART_TX(ms_door_position);
        USART_TX(ls_door_position);
```

```
    //format data for DAC and send to DAC on PORT C
    DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));
    PORTC = DAC_data;

    //increment position counter
    if(door_position >= 4095)
       door_position = 4095;
    else
      door_position++;
    }
  break;

case 0x02:                          //Close
  while(PINB == 0x02)
    {
    //split into two bytes
    ms_door_position=(unsigned char)(((door_position >>6)
                                     &(0x00FF))|0x01);
    ls_door_position=(unsigned char)(((door_position <<1)
                                     &(0x00FF))&0xFE);

    //TX data to USART
    USART_TX(ms_door_position);
    USART_TX(ls_door_position);

    //format data for DAC and send to DAC on PORT C
    DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));


    PORTC = DAC_data;

    //decrement position counter
    if(door_position <= 0)
      door_position = 0;
    else
      door_position--;
    }
    break;
```

```
        default:;                       //all other cases
      }                                 //end switch(new_PORTB)
  }                                     //end if new_PORTB
  old_PORTB=new_PORTB;                  //update PORTB
}

//***************************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay function
//provides the specified delay as the number of 6.55 ms "clock ticks" from
//the Timer0 interrupt.
//Note: this function is only valid when using a 10 MHz crystal or ceramic
//      resonator
//***************************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                           //reset timer0
input_delay = 0;
while(input_delay <= number_of_6_55ms_interrupts)
  {
  }
}

//***************************************************************************
//***************************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is being
//employed as a time base for a master timer for this project. The ceramic
//resonator operating at 10 MHz is divided by 256.  The 8-bit Timer0
//register (TCNT0) overflows every 256 counts or every 6.55 ms.
//***************************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase
by 256, overflow occurs every 6.55ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");   //enable global interrupt
}
```

```
//***************************************************************************
//***************************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//***************************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;                          //input delay processing
}

//***************************************************************************
//void InitUSART(void)
//***************************************************************************

void InitUSART(void)
{
//USART Channel 1 initialization
//System operating frequency: 10 MHz
// Comm Parameters: 8 bit Data, 1 stop, No Parity
// USART Receiver Off
// USART Transmitter On
// USART Mode: Asynchronous
// USART Baud Rate: 9600

UCSR1A=0x00;
UCSR1B=0x08;
UCSR1C=0x06;  //1 stop bit, No parity
UBRR1H=0x00;
UBRR1L=0x40;
}

//***************************************************************************
//void USART_TX(unsigned char data):
//USART Channel 1
//***************************************************************************
void USART_TX(unsigned char data)
{
```

```
// Set USART Data Register
while(!(UCSR1A & 0x20));                //Checks to see if the  data
                                        //register is empty
  UDR1= data;                           //sets the value in ADCH to the
                                        //value in the USART Data Register
}

//***************************************************************************
//***************************************************************************
//end of file
//***************************************************************************
```

Receive ATmega164 code follows.

```
//***************************************************************************
//***************************************************************************
//target controller: ATMEL ATmega164
//Pin 9 Reset - 1M resistor to Vcc, tact switch to ground, 1.0 uF to ground
//Pin 10 Vcc - 1.0 uF to ground
//Pin 11 Gnd
//Pin 12 ZTT-10.00MT ceramic resonator connection
//Pin 13 ZTT-10.00MT ceramic resonator connection
//Pin 30 AVcc to Vcc
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//***************************************************************************
//include files**************************************************************

//ATMEL register definitions for ATmega164
#include<iom164pv.h>
#include<macros.h>

//EEPROM support functions
#include<eeprom.h>

//function prototypes********************************************************

 //delay specified number 6.55ms int
void delay(unsigned int number_of_6_55ms_interrupts);
void initialize_ports(void);            //initializes ports
void power_on_reset(void);              //returns system to startup state
```

```
void init_timer0_ovf_interrupt(void);
 //used to initialize timer0 overflow
void InitUSART(void);
void USART_TX(unsigned char data);
unsigned char USART_RX(void);

                                        //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19
#pragma interrupt_handler USART_RX_interrupt_isr: 29

//main program*****************************************************************
//The main program checks PORTB for user input activity.  If new activity
//is found, the program responds.

//global variables
unsigned char   old_PORTB = 0x08;       //present value of PORTB
unsigned char   new_PORTB;              //new values of PORTB
unsigned char   channel=0;              //ADC channel for conversion
unsigned int    analog_value;           //value from ADC conversion
unsigned int    factory_test = 0;       //activates factory test functions
unsigned int    input_delay = 0;        //insure input lasts for 50 ms
                                        //one interrupt equals 6.55 ms
unsigned int internal_close_timer_num_interrupts;
unsigned int    door_position = 0;
unsigned char   data_rx;
unsigned int    dummy1 = 0x1234;
unsigned int    keep_going =1;
unsigned int    loop_counter = 0;
unsigned int    ms_position, ls_position;

void main(void)
{
power_on_reset();
 //returns system to startup condition
initialize_ports();
 //return LED configuration to default
InitUSART();
                                        //limited startup features
//main activity loop - processor will continually cycle through loop for new
```

```
//activity.  Activity initialized by external signals presented to PORTB or
//PORTD. PORTB[0] is active low, all other inputs PORTB[7:1] are active high

while(1)
  {
  //continuous loop waiting for interrupts
  _StackCheck();                        //check for stack overflow

  }
}//end main

//Function definitions
//***********************************************************************
//power_on_reset:
//***********************************************************************

void power_on_reset(void)
{
initialize_ports();                   //initialize ports
init_timer0_ovf_interrupt();          //initialize Timer0 to serve as
                                      //time "clock tick"
}

//***********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***********************************************************************

void initialize_ports(void)
{
//PORTA
DDRA=0xff; //PORTA[7:0] output
PORTA=0x00;//Turn off pull ups

//PORTB
DDRB=0xfc; //PORTB[7-2] output, PORTB[1:0] input
PORTB=0x00;//disable PORTB pull-up resistors

//PORTC
DDRC=0xff; //set PORTC[7-0] as output
```

```
PORTC=0x00;//init low

//PORTD
DDRD=0xfb; //set PORTD[7-3,1-0] as output, PORTD[2] as input
PORTD=0x00;//initialize low
}

//***********************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay function
//provides the specified delay as the number of 6.55 ms "clock ticks" from
//the Timer0 interrupt.
//Note: this function is only valid when using a 10 MHz crystal or ceramic
//      resonator

//***********************************************************************
//***********************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                           //reset timer0
input_delay = 0;
while(input_delay <= number_of_6_55ms_interrupts)
  {
   }
}

//***********************************************************************
//***********************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is being
//employed as a time base for a master timer for this project.  The ceramic
//resonator operating at 10 MHz is divided by 256.  The 8-bit Timer0
//register (TCNT0) overflows every 256 counts or every 6.55 ms.
//***********************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase
by 256, overflow occurs every 6.55ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
```

```
asm("SEI");   //enable global interrupt
}

//***********************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//***********************************************************************

void timer0_interrupt_isr(void)
{
PORTA ^= 0x01;
input_delay++;     //input delay processing
//PORTA = 0x00;
}

//***********************************************************************
//void InitUSART(void)
//***********************************************************************

void InitUSART(void)
{
//USART Channel 1 initialization
//System operating frequency: 10 MHz
// Comm Parameters: 8 bit Data, 1 stop, No Parity
// USART Receiver On
// USART Transmitter Off
// USART Mode: Asynchronous
// USART Baud Rate: 9600

UCSR1A=0x00;
UCSR1B=0x90;  //RX on, RX interrupt on
UCSR1C=0x06;  //1 stop bit, No parity
UBRR1H=0x00;
UBRR1L=0x40;
}

//***********************************************************************
//USART_RX_interrupt_isr
```

```
//***************************************************************************

void USART_RX_interrupt_isr(void)
{
//Receive USART data
data_rx = UDR1;

//Retrieve door position data from EEPROM
EEPROM_READ((int) &door_position_EEPROM, door_position);

//Determine which byte to update
if((data_rx & 0x01)==0x01)                    //Byte ID = 1
  {
  ms_position = data_rx;

  //Update bit 7
  if((ms_position & 0x0020)==0x0020)          //Test for logic 1
    door_position = door_position | 0x0080;   //Set bit 7 to 1
  else
    door_position = door_position & 0xff7f;   //Reset bit 7 to 0

  //Update remaining bits
  ms_position = ((ms_position<<6) & 0x0f00);
  //shift left 6--blank other bits
  door_position = door_position & 0x00ff;     //Blank ms byte
  door_position = door_position | ms_position; //Update ms byte
  }
else                                  //Byte ID = 0
  {
  ls_position = data_rx;

  //Update ls_position
  ls_position = ((ls_position >> 1) & 0x007f);
  //Shift right 1-blank other bits
  if((door_position & 0x0080)==0x0080)
        //Test bit 7 of curr position
    ls_position = ls_position | 0x0080;       //Set bit 7 to logic 1
  else
    ls_position = ls_position & 0xff7f;       //Reset bit 7 to 0
```

```
  door_position = door_position & 0xff00;      //Blank ls byte
  door_position = door_position | ls_position; //Update ls byte
  }

//format data for DAC and send to DAC on PORT C
DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));


PORTC = DAC_data;
}

//***************************************************************************
//end of file
//***************************************************************************
```

5.7 SUMMARY

In this chapter, we provided an introduction to the interrupt features available aboard the ATmega164. We also discussed how to program an interrupt for proper operation and provided two representative samples: an external interrupt and an internal interrupt.

5.8 CHAPTER PROBLEMS

5.1. What is the purpose of an interrupt?

5.2. Describe the flow of events when an interrupt occurs.

5.3. Describe the interrupt features available with the ATmega164.

5.4. What is the interrupt priority? How is it determined?

5.5. What steps are required by the system designer to properly configure an interrupt?

5.6. How is the interrupt system turned "ON" and "OFF?"

5.7. A 10 MHz ceramic resonator is not available. Redo the example of the Timer/Counter0 Overflow interrupt provided with a timebase of 1 MHz and 8 MHz.

5.8. What is the maximum delay that may be generated with the delay function provided in the text without modification? How could the function be modified for longer delays?

5.9. In the text we provided a 24 hour timer (hh:mm:ss:ms) using the Timer/Counter0 Overflow interrupt. What is the accuracy of the timer? How can it be improved?

5.10. Adapt the 24 hour timer example to generate an active high logic pulse on a microcontroller pin of your choice for three seconds. The pin should go logic high three weeks from now.

5.11. What are the concerns when using multiple interrupts in a given application?

5.12. How much time can background processing relative to foreground processing be implemented?

5.13. What is the advantage of using interrupts over polling techniques?

REFERENCES

Atmel 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash, ATmega164, ATmega164L, data sheet: 2466L-AVR-06/05, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.

S. Barrett and D. Pack (2006) Microcontrollers Fundamentals for Engineers and Scientists. Morgan and Claypool Publishers.

APPENDIX A

ATmega164 Register Set

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0xFF)	Reserved	-	-	-	-	-	-	-	-	
(0xFE)	Reserved	-	-	-	-	-	-	-	-	
(0xFD)	Reserved	-	-	-	-	-	-	-	-	
(0xFC)	Reserved	-	-	-	-	-	-	-	-	
(0xFB)	Reserved	-	-	-	-	-	-	-	-	
(0xFA)	Reserved	-	-	-	-	-	-	-	-	
(0xF9)	Reserved	-	-	-	-	-	-	-	-	
(0xF8)	Reserved	-	-	-	-	-	-	-	-	
(0xF7)	Reserved	-	-	-	-	-	-	-	-	
(0xF6)	Reserved	-	-	-	-	-	-	-	-	
(0xF5)	Reserved	-	-	-	-	-	-	-	-	
(0xF4)	Reserved	-	-	-	-	-	-	-	-	
(0xF3)	Reserved	-	-	-	-	-	-	-	-	
(0xF2)	Reserved	-	-	-	-	-	-	-	-	
(0xF1)	Reserved	-	-	-	-	-	-	-	-	
(0xF0)	Reserved	-	-	-	-	-	-	-	-	
(0xEF)	Reserved	-	-	-	-	-	-	-	-	
(0xEE)	Reserved	-	-	-	-	-	-	-	-	
(0xED)	Reserved	-	-	-	-	-	-	-	-	
(0xEC)	Reserved	-	-	-	-	-	-	-	-	
(0xEB)	Reserved	-	-	-	-	-	-	-	-	
(0xEA)	Reserved	-	-	-	-	-	-	-	-	
(0xE9)	Reserved	-	-	-	-	-	-	-	-	
(0xE8)	Reserved	-	-	-	-	-	-	-	-	
(0xE7)	Reserved	-	-	-	-	-	-	-	-	
(0xE6)	Reserved	-	-	-	-	-	-	-	-	
(0xE5)	Reserved	-	-	-	-	-	-	-	-	
(0xE4)	Reserved	-	-	-	-	-	-	-	-	
(0xE3)	Reserved	-	-	-	-	-	-	-	-	
(0xE2)	Reserved	-	-	-	-	-	-	-	-	
(0xE1)	Reserved	-	-	-	-	-	-	-	-	
(0xE0)	Reserved	-	-	-	-	-	-	-	-	
(0xDF)	Reserved	-	-	-	-	-	-	-	-	
(0xDE)	Reserved	-	-	-	-	-	-	-	-	
(0xDD)	Reserved	-	-	-	-	-	-	-	-	
(0xDC)	Reserved	-	-	-	-	-	-	-	-	
(0xDB)	Reserved	-	-	-	-	-	-	-	-	
(0xDA)	Reserved	-	-	-	-	-	-	-	-	
(0xD9)	Reserved	-	-	-	-	-	-	-	-	
(0xD8)	Reserved	-	-	-	-	-	-	-	-	
(0xD7)	Reserved	-	-	-	-	-	-	-	-	
(0xD6)	Reserved	-	-	-	-	-	-	-	-	
(0xD5)	Reserved	-	-	-	-	-	-	-	-	
(0xD4)	Reserved	-	-	-	-	-	-	-	-	
(0xD3)	Reserved	-	-	-	-	-	-	-	-	
(0xD2)	Reserved	-	-	-	-	-	-	-	-	
(0xD1)	Reserved	-	-	-	-	-	-	-	-	
(0xD0)	Reserved	-	-	-	-	-	-	-	-	
(0xCF)	Reserved	-	-	-	-	-	-	-	-	
(0xCE)	UDR1	USART1 I/O Data Register								190
(0xCD)	UBRR1H	-	-	-	-	USART1 Baud Rate Register High Byte				194/207
(0xCC)	UBRR1L	USART1 Baud Rate Register Low Byte								194/207
(0xCB)	Reserved	-	-	-	-	-	-	-	-	
(0xCA)	UCSR1C	UMSEL11	UMSEL10	-	-	-	UDORD1	UCPHA1	UCPOL1	192/206
(0xC9)	UCSR1B	RXCIE1	TXCIE1	UDRIE1	RXEN1	TXEN1	UCSZ12	RXB81	TXB81	191/205
(0xC8)	UCSR1A	RXC1	TXC1	UDRE1	FE1	DOR1	UPE1	U2X1	MPCM1	190/205
(0xC7)	Reserved	-	-	-	-	-	-	-	-	
(0xC6)	UDR0	USART0 I/O Data Register								190
(0xC5)	UBRR0H	-	-	-	-	USART0 Baud Rate Register High Byte				194/207
(0xC4)	UBRR0L	USART0 Baud Rate Register Low Byte								194/207
(0xC3)	Reserved	-	-	-	-	-	-	-	-	
(0xC2)	UCSR0C	UMSEL01	UMSEL00	-	-	-	UDORD0	UCPHA0	UCPOL0	192/206
(0xC1)	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	191/205

Figure A.1: Atmel AVR ATmega164 Register Set. (Figure used with permission of Atmel, Incorporated.)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0xC0)	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	190/205
(0xBF)	Reserved	-	-	-	-	-	-	-	-	
(0xBE)	Reserved	-	-	-	-	-	-	-	-	
(0xBD)	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-	236
(0xBC)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE	233
(0xBB)	TWDR	2-wire Serial Interface Data Register								235
(0xBA)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	236
(0xB9)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0	235
(0xB8)	TWBR	2-wire Serial Interface Bit Rate Register								233
(0xB7)	Reserved	-	-	-	-	-	-	-	-	
(0xB6)	ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB	158
(0xB5)	Reserved	-	-	-	-	-	-	-	-	
(0xB4)	OCR2B	Timer/Counter2 Output Compare Register B								158
(0xB3)	OCR2A	Timer/Counter2 Output Compare Register A								158
(0xB2)	TCNT2	Timer/Counter2 (8 Bit)								157
(0xB1)	TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20	156
(0xB0)	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20	153
(0xAF)	Reserved	-	-	-	-	-	-	-	-	
(0xAE)	Reserved	-	-	-	-	-	-	-	-	
(0xAD)	Reserved	-	-	-	-	-	-	-	-	
(0xAC)	Reserved	-	-	-	-	-	-	-	-	
(0xAB)	Reserved	-	-	-	-	-	-	-	-	
(0xAA)	Reserved	-	-	-	-	-	-	-	-	
(0xA9)	Reserved	-	-	-	-	-	-	-	-	
(0xA8)	Reserved	-	-	-	-	-	-	-	-	
(0xA7)	Reserved	-	-	-	-	-	-	-	-	
(0xA6)	Reserved	-	-	-	-	-	-	-	-	
(0xA5)	Reserved	-	-	-	-	-	-	-	-	
(0xA4)	Reserved	-	-	-	-	-	-	-	-	
(0xA3)	Reserved	-	-	-	-	-	-	-	-	
(0xA2)	Reserved	-	-	-	-	-	-	-	-	
(0xA1)	Reserved	-	-	-	-	-	-	-	-	
(0xA0)	Reserved	-	-	-	-	-	-	-	-	
(0x9F)	Reserved	-	-	-	-	-	-	-	-	
(0x9E)	Reserved	-	-	-	-	-	-	-	-	
(0x9D)	Reserved	-	-	-	-	-	-	-	-	
(0x9C)	Reserved	-	-	-	-	-	-	-	-	
(0x9B)	Reserved	-	-	-	-	-	-	-	-	
(0x9A)	Reserved									
(0x99)	Reserved	-	-	-	-	-	-	-	-	
(0x98)	Reserved	-	-	-	-	-	-	-	-	
(0x97)	Reserved	-	-	-	-	-	-	-	-	
(0x96)	Reserved	-	-	-	-	-	-	-	-	
(0x95)	Reserved	-	-	-	-	-	-	-	-	
(0x94)	Reserved	-	-	-	-	-	-	-	-	
(0x93)	Reserved	-	-	-	-	-	-	-	-	
(0x92)	Reserved	-	-	-	-	-	-	-	-	
(0x91)	Reserved	-	-	-	-	-	-	-	-	
(0x90)	Reserved	-	-	-	-	-	-	-	-	
(0x8F)	Reserved	-	-	-	-	-	-	-	-	
(0x8E)	Reserved	-	-	-	-	-	-	-	-	
(0x8D)	Reserved	-	-	-	-	-	-	-	-	
(0x8C)	Reserved	-	-	-	-	-	-	-	-	
(0x8B)	OCR1BH	Timer/Counter1 - Output Compare Register B High Byte								137
(0x8A)	OCR1BL	Timer/Counter1 - Output Compare Register B Low Byte								137
(0x89)	OCR1AH	Timer/Counter1 - Output Compare Register A High Byte								137
(0x88)	OCR1AL	Timer/Counter1 - Output Compare Register A Low Byte								137
(0x87)	ICR1H	Timer/Counter1 - Input Capture Register High Byte								138
(0x86)	ICR1L	Timer/Counter1 - Input Capture Register Low Byte								138
(0x85)	TCNT1H	Timer/Counter1 - Counter Register High Byte								137
(0x84)	TCNT1L	Timer/Counter1 - Counter Register Low Byte								137
(0x83)	Reserved	-	-	-	-	-	-	-	-	
(0x82)	TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-	136
(0x81)	TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	135
(0x80)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10	133
(0x7F)	DIDR1	-	-	-	-	-	-	AIN1D	AIN0D	240

Figure A.2: Atmel AVR ATmega164 Register Set. (Figure used with permission of Atmel, Incorporated.)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0x7E)	DIDR0	ADC7D	ADC6D	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	260
(0x7D)	Reserved	-	-	-	-	-	-	-	-	
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	259
(0x7B)	ADCSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0	239
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	258
(0x79)	ADCH	ADC Data Register High byte								259
(0x78)	ADCL	ADC Data Register Low byte								259
(0x77)	Reserved	-	-	-	-	-	-	-	-	
(0x76)	Reserved	-	-	-	-	-	-	-	-	
(0x75)	Reserved	-	-	-	-	-	-	-	-	
(0x74)	Reserved	-	-	-	-	-	-	-	-	
(0x73)	PCMSK3	PCINT31	PCINT30	PCINT29	PCINT28	PCINT27	PCINT26	PCINT25	PCINT24	71
(0x72)	Reserved	-	-	-	-	-	-	-	-	
(0x71)	Reserved	-	-	-	-	-	-	-	-	
(0x70)	TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2	159
(0x6F)	TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1	138
(0x6E)	TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0	110
(0x6D)	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	71
(0x6C)	PCMSK1	PCINT15	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	71
(0x6B)	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	72
(0x6A)	Reserved	-	-	-	-	-	-	-	-	
(0x69)	EICRA	-	-	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	68
(0x68)	PCICR	-	-	-	-	PCIE3	PCIE2	PCIE1	PCIE0	70
(0x67)	Reserved	-	-	-	-	-	-	-	-	
(0x66)	OSCCAL	Oscillator Calibration Register								41
(0x65)	Reserved	-	-	-	-	-	-	-	-	
(0x64)	PRR	PRTWI	PRTIM2	PRTIM0	PRUSART1	PRTIM1	PRSPI	PRUSART0	PRADC	49
(0x63)	Reserved	-	-	-	-	-	-	-	-	
(0x62)	Reserved	-	-	-	-	-	-	-	-	
(0x61)	CLKPR	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0	41
(0x60)	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	60
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	11
0x3E (0x5E)	SPH	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	12
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	12
0x3C (0x5C)	Reserved	-	-	-	-	-	-	-	-	
0x3B (0x5B)	RAMPZ	-	-	-	-	-	-	-	RAMPZ0	15
0x3A (0x5A)	Reserved	-	-	-	-	-	-	-	-	
0x39 (0x59)	Reserved	-	-	-	-	-	-	-	-	
0x38 (0x58)	Reserved	-	-	-	-	-	-	-	-	
0x37 (0x57)	SPMCSR	SPMIE	RWWSB	SIGRD	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	292
0x36 (0x56)	Reserved	-	-	-	-	-	-	-	-	
0x35 (0x55)	MCUCR	JTD	BODS	BODSE	PUD	-	-	IVSEL	IVCE	92/276
0x34 (0x54)	MCUSR	-	-	-	JTRF	WDRF	BORF	EXTRF	PORF	59/276
0x33 (0x53)	SMCR	-	-	-	-	SM2	SM1	SM0	SE	48
0x32 (0x52)	Reserved	-	-	-	-	-	-	-	-	
0x31 (0x51)	OCDR	On-Chip Debug Register								266
0x30 (0x50)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	256
0x2F (0x4F)	Reserved	-	-	-	-	-	-	-	-	
0x2E (0x4E)	SPDR	SPI 0 Data Register								171
0x2D (0x4D)	SPSR	SPIF0	WCOL0	-	-	-	-	-	SPI2X0	170
0x2C (0x4C)	SPCR	SPIE0	SPE0	DORD0	MSTR0	CPOL0	CPHA0	SPR01	SPR00	169
0x2B (0x4B)	GPIOR2	General Purpose I/O Register 2								29
0x2A (0x4A)	GPIOR1	General Purpose I/O Register 1								29
0x29 (0x49)	Reserved	-	-	-	-	-	-	-	-	
0x28 (0x48)	OCR0B	Timer/Counter0 Output Compare Register B								110
0x27 (0x47)	OCR0A	Timer/Counter0 Output Compare Register A								109
0x26 (0x46)	TCNT0	Timer/Counter0 (8 Bit)								109
0x25 (0x45)	TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	108
0x24 (0x44)	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00	110
0x23 (0x43)	GTCCR	TSM	-	-	-	-	-	PSRASY	PSR5SYNC	160
0x22 (0x42)	EEARH	-	-	-	-	EEPROM Address Register High Byte				24
0x21 (0x41)	EEARL	EEPROM Address Register Low Byte								24
0x20 (0x40)	EEDR	EEPROM Data Register								24
0x1F (0x3F)	EECR	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	24
0x1E (0x3E)	GPIOR0	General Purpose I/O Register 0								29
0x1D (0x3D)	EIMSK	-	-	-	-	-	INT2	INT1	INT0	69

Figure A.3: Atmel AVR ATmega164 Register Set. (Figure used with permission of Atmel, Incorporated.)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x1C (0x3C)	EIFR	-	-	-	-	-	INTF2	INTF1	INTF0	69
0x1B (0x3B)	PCIFR	-	-	-	-	PCIF3	PCIF2	PCIF1	PCIF0	70
0x1A (0x3A)	Reserved	-	-	-	-	-	-	-	-	
0x19 (0x39)	Reserved	-	-	-	-	-	-	-	-	
0x18 (0x38)	Reserved	-	-	-	-	-	-	-	-	
0x17 (0x37)	TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2	160
0x16 (0x36)	TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	139
0x15 (0x35)	TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0	110
0x14 (0x34)	Reserved	-	-	-	-	-	-	-	-	
0x13 (0x33)	Reserved	-	-	-	-	-	-	-	-	
0x12 (0x32)	Reserved	-	-	-	-	-	-	-	-	
0x11 (0x31)	Reserved	-	-	-	-	-	-	-	-	
0x10 (0x30)	Reserved	-	-	-	-	-	-	-	-	
0x0F (0x2F)	Reserved	-	-	-	-	-	-	-	-	
0x0E (0x2E)	Reserved	-	-	-	-	-	-	-	-	
0x0D (0x2D)	Reserved	-	-	-	-	-	-	-	-	
0x0C (0x2C)	Reserved	-	-	-	-	-	-	-	-	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	93
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	93
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	93
0x08 (0x28)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	93
0x07 (0x27)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	93
0x06 (0x26)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	93
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	92
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	92
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	92
0x02 (0x22)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	92
0x01 (0x21)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	92
0x00 (0x20)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	92

Notes:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O registers within the address range $00 - $1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the status flags are cleared by writing a logical one to them. Note that the CBI and SBI instructions will operate on all bits in the I/O register, writing a one back into any flag read as set, thus clearing the flag. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses $00 - $3F must be used. When addressing I/O registers as data space using LD and ST instructions, $20 must be added to these addresses. The ATmega164P/324P/644P is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from $60 - $FF, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

Figure A.4: Atmel AVR ATmega164 Register Set. (Figure used with permission of Atmel, Incorporated.)

APPENDIX B

ATmega164 Header File

During C programming, the contents of a specific register may be referred to by name when an appropriate header file is included within your program. The header file provides the link between the register name used within a program and the hardware location of the register.

Provided below is the ATmega164 header file from the ICC AVR compiler. This header file was provided courtesy of ImageCraft Incorporated.

```
#ifndef __iom164pv_h
#define __iom164pv_h

/* ATmega164P header file for
 * ImageCraft ICCAVR compiler
 */

/* i/o register addresses
 * >= 0x60 are memory mapped only
 */

/* 2006/10/01 created
 */

/* Port D */
#define PIND (*(volatile unsigned char *)0x29)
#define DDRD (*(volatile unsigned char *)0x2A)
#define PORTD (*(volatile unsigned char *)0x2B)

/* Port C */
#define PINC (*(volatile unsigned char *)0x26)
#define DDRC (*(volatile unsigned char *)0x27)
#define PORTC (*(volatile unsigned char *)0x28)

/* Port B */
#define PINB (*(volatile unsigned char *)0x23)
#define DDRB (*(volatile unsigned char *)0x24)
#define PORTB (*(volatile unsigned char *)0x25)
```

```
/* Port A */
#define PINA (*(volatile unsigned char *)0x20)
#define DDRA (*(volatile unsigned char *)0x21)
#define PORTA (*(volatile unsigned char *)0x22)

/* Timer/Counter Interrupts */
#define TIFR0 (*(volatile unsigned char *)0x35)
#define  OCF0B    2
#define  OCF0A    1
#define  TOV0     0
#define TIMSK0 (*(volatile unsigned char *)0x6E)
#define  OCIE0B   2
#define  OCIE0A   1
#define  TOIE0    0
#define TIFR1 (*(volatile unsigned char *)0x36)
#define  ICF1     5
#define  OCF1B    2
#define  OCF1A    1
#define  TOV1     0
#define TIMSK1 (*(volatile unsigned char *)0x6F)
#define  ICIE1    5
#define  OCIE1B   2
#define  OCIE1A   1
#define  TOIE1    0
#define TIFR2 (*(volatile unsigned char *)0x37)
#define  OCF2B    2
#define  OCF2A    1
#define  TOV2     0
#define TIMSK2 (*(volatile unsigned char *)0x70)
#define  OCIE2B   2
#define  OCIE2A   1
#define  TOIE2    0

/* External Interrupts */
#define EIFR (*(volatile unsigned char *)0x3C)
#define  INTF2    2
#define  INTF1    1
#define  INTF0    0
```

```
#define EIMSK (*(volatile unsigned char *)0x3D)
#define  INT2     2
#define  INT1     1
#define  INT0     0
#define EICRA (*(volatile unsigned char *)0x69)
#define  ISC21    5
#define  ISC20    4
#define  ISC11    3
#define  ISC10    2
#define  ISC01    1
#define  ISC00    0

/* Pin Change Interrupts */
#define PCIFR (*(volatile unsigned char *)0x3B)
#define  PCIF3    3
#define  PCIF2    2
#define  PCIF1    1
#define  PCIF0    0
#define PCICR (*(volatile unsigned char *)0x68)
#define  PCIE3    3
#define  PCIE2    2
#define  PCIE1    1
#define  PCIE0    0
#define PCMSK0 (*(volatile unsigned char *)0x6B)
#define PCMSK1 (*(volatile unsigned char *)0x6C)
#define PCMSK2 (*(volatile unsigned char *)0x6D)
#define PCMSK3 (*(volatile unsigned char *)0x73)

/* GPIOR */
#define GPIOR0 (*(volatile unsigned char *)0x3E)
#define GPIOR1 (*(volatile unsigned char *)0x4A)
#define GPIOR2 (*(volatile unsigned char *)0x4B)

/* EEPROM */
#define EECR (*(volatile unsigned char *)0x3F)
#define  EEPM1    5
#define  EEPM0    4
#define  EERIE    3
#define  EEMPE    2
```

```
#define  EEMWE    2
#define  EEPE     1
#define  EEWE     1
#define  EERE     0
#define EEDR (*(volatile unsigned char *)0x40)
#define EEAR (*(volatile unsigned int *)0x41)
#define EEARL (*(volatile unsigned char *)0x41)
#define EEARH (*(volatile unsigned char *)0x42)

/* GTCCR */
#define GTCCR (*(volatile unsigned char *)0x43)
#define  TSM      7
#define  PSRASY   1
#define  PSR2     1
#define  PSRSYNC  0
#define  PSR10    0

/* Timer/Counter 0 */
#define OCR0B (*(volatile unsigned char *)0x48)
#define OCR0A (*(volatile unsigned char *)0x47)
#define TCNT0 (*(volatile unsigned char *)0x46)
#define TCCR0B (*(volatile unsigned char *)0x45)
#define  FOC0A    7
#define  FOC0B    6
#define  WGM02    3
#define  CS02     2
#define  CS01     1
#define  CS00     0
#define TCCR0A (*(volatile unsigned char *)0x44)
#define  COM0A1   7
#define  COM0A0   6
#define  COM0B1   5
#define  COM0B0   4
#define  WGM01    1
#define  WGM00    0

/* SPI */
#define SPCR (*(volatile unsigned char *)0x4C)
#define  SPIE     7
```

```
#define  SPE      6
#define  DORD     5
#define  MSTR     4
#define  CPOL     3
#define  CPHA     2
#define  SPR1     1
#define  SPR0     0
#define SPSR (*(volatile unsigned char *)0x4D)
#define  SPIF     7
#define  WCOL     6
#define  SPI2X    0
#define SPDR (*(volatile unsigned char *)0x4E)

/* Analog Comparator Control and Status Register */
#define ACSR (*(volatile unsigned char *)0x50)
#define  ACD      7
#define  ACBG     6
#define  ACO      5
#define  ACI      4
#define  ACIE     3
#define  ACIC     2
#define  ACIS1    1
#define  ACIS0    0

/* OCDR */
#define OCDR (*(volatile unsigned char *)0x51)
#define  IDRD     7

/* MCU */
#define MCUSR (*(volatile unsigned char *)0x54)
#define  JTRF     4
#define  WDRF     3
#define  BORF     2
#define  EXTRF    1
#define  PORF     0
#define MCUCR (*(volatile unsigned char *)0x55)
#define  JTD      7
#define  PUD      4
#define  IVSEL    1
```

```
#define  IVCE     0

#define SMCR (*(volatile unsigned char *)0x53)
#define  SM2      3
#define  SM1      2
#define  SM0      1
#define  SE       0

/* SPM Control and Status Register */
#define SPMCSR (*(volatile unsigned char *)0x57)
#define  SPMIE    7
#define  RWWSB    6
#define  SIGRD    5
#define  RWWSRE   4
#define  BLBSET   3
#define  PGWRT    2
#define  PGERS    1
#define  SPMEN    0

/* Stack Pointer */
#define SP   (*(volatile unsigned int *)0x5D)
#define SPL  (*(volatile unsigned char *)0x5D)
#define SPH  (*(volatile unsigned char *)0x5E)

/* Status REGister */
#define SREG (*(volatile unsigned char *)0x5F)

/* Watchdog Timer Control Register */
#define WDTCSR (*(volatile unsigned char *)0x60)
#define WDTCR (*(volatile unsigned char *)0x60)
#define  WDIF     7
#define  WDIE     6
#define  WDP3     5
#define  WDCE     4
#define  WDE      3
#define  WDP2     2
#define  WDP1     1
#define  WDP0     0
```

```
/* clock prescaler control register */
#define CLKPR (*(volatile unsigned char *)0x61)
#define  CLKPCE   7
#define  CLKPS3   3
#define  CLKPS2   2
#define  CLKPS1   1
#define  CLKPS0   0

/* PRR */
#define PRR0 (*(volatile unsigned char *)0x64)
#define  PRTWI    7
#define  PRTIM2   6
#define  PRTIM0   5
#define  PRUSART1 4
#define  PRTIM1   3
#define  PRSPI    2
#define  PRUSART0 1
#define  PRADC    0

/* Oscillator Calibration Register */
#define OSCCAL (*(volatile unsigned char *)0x66)

/* ADC */
#define ADC  (*(volatile unsigned int *)0x78)
#define ADCL (*(volatile unsigned char *)0x78)
#define ADCH (*(volatile unsigned char *)0x79)
#define ADCSRA (*(volatile unsigned char *)0x7A)
#define  ADEN     7
#define  ADSC     6
#define  ADATE    5
#define  ADIF     4
#define  ADIE     3
#define  ADPS2    2
#define  ADPS1    1
#define  ADPS0    0
#define ADCSRB (*(volatile unsigned char *)0x7B)
#define  ACME     6
#define  ADTS2    2
#define  ADTS1    1
```

```
#define  ADTS0    0
#define ADMUX (*(volatile unsigned char *)0x7C)
#define  REFS1    7
#define  REFS0    6
#define  ADLAR    5
#define  MUX4     4
#define  MUX3     3
#define  MUX2     2
#define  MUX1     1
#define  MUX0     0

/* DIDR */
#define DIDR0 (*(volatile unsigned char *)0x7E)
#define  ADC7D    7
#define  ADC6D    6
#define  ADC5D    5
#define  ADC4D    4
#define  ADC3D    3
#define  ADC2D    2
#define  ADC1D    1
#define  ADC0D    0
#define DIDR1 (*(volatile unsigned char *)0x7F)
#define  AIN1D    1
#define  AIN0D    0

/* Timer/Counter1 */
#define ICR1 (*(volatile unsigned int *)0x86)
#define ICR1L (*(volatile unsigned char *)0x86)
#define ICR1H (*(volatile unsigned char *)0x87)
#define OCR1B (*(volatile unsigned int *)0x8A)
#define OCR1BL (*(volatile unsigned char *)0x8A)
#define OCR1BH (*(volatile unsigned char *)0x8B)
#define OCR1A (*(volatile unsigned int *)0x88)
#define OCR1AL (*(volatile unsigned char *)0x88)
#define OCR1AH (*(volatile unsigned char *)0x89)
#define TCNT1 (*(volatile unsigned int *)0x84)
#define TCNT1L (*(volatile unsigned char *)0x84)
#define TCNT1H (*(volatile unsigned char *)0x85)
#define TCCR1C (*(volatile unsigned char *)0x82)
```

```
#define  FOC1A    7
#define  FOC1B    6
#define TCCR1B (*(volatile unsigned char *)0x81)
#define  ICNC1    7
#define  ICES1    6
#define  WGM13    4
#define  WGM12    3
#define  CS12     2
#define  CS11     1
#define  CS10     0
#define TCCR1A (*(volatile unsigned char *)0x80)
#define  COM1A1   7
#define  COM1A0   6
#define  COM1B1   5
#define  COM1B0   4
#define  WGM11    1
#define  WGM10    0

/* Timer/Counter2 */
#define ASSR (*(volatile unsigned char *)0xB6)
#define  EXCLK    6
#define  AS2      5
#define  TCN2UB   4
#define  OCR2AUB  3
#define  OCR2BUB  2
#define  TCR2AUB  1
#define  TCR2BUB  0
#define OCR2B (*(volatile unsigned char *)0xB4)
#define OCR2A (*(volatile unsigned char *)0xB3)
#define TCNT2 (*(volatile unsigned char *)0xB2)
#define TCCR2B (*(volatile unsigned char *)0xB1)
#define  FOC2A    7
#define  FOC2B    6
#define  WGM22    3
#define  CS22     2
#define  CS21     1
#define  CS20     0
#define TCCR2A (*(volatile unsigned char *)0xB0)
#define  COM2A1   7
```

```
#define  COM2A0   6
#define  COM2B1   5
#define  COM2B0   4
#define  WGM21    1
#define  WGM20    0

/* 2-wire SI */
#define TWBR (*(volatile unsigned char *)0xB8)
#define TWSR (*(volatile unsigned char *)0xB9)
#define  TWPS1    1
#define  TWPS0    0
#define TWAR (*(volatile unsigned char *)0xBA)
#define  TWGCE    0
#define TWDR (*(volatile unsigned char *)0xBB)
#define TWCR (*(volatile unsigned char *)0xBC)
#define  TWINT    7
#define  TWEA     6
#define  TWSTA    5
#define  TWSTO    4
#define  TWWC     3
#define  TWEN     2
#define  TWIE     0
#define TWAMR (*(volatile unsigned char *)0xBD)

/* USART0 */
#define UBRR0H (*(volatile unsigned char *)0xC5)
#define UBRR0L (*(volatile unsigned char *)0xC4)
#define UBRR0 (*(volatile unsigned int *)0xC4)
#define UCSR0C (*(volatile unsigned char *)0xC2)
#define  UMSEL01  7
#define  UMSEL00  6
#define  UPM01    5
#define  UPM00    4
#define  USBS0    3
#define  UCSZ01   2
#define  UCSZ00   1
#define  UCPOL0   0
#define UCSR0B (*(volatile unsigned char *)0xC1)
#define  RXCIE0   7
```

```
#define  TXCIE0   6
#define  UDRIE0   5
#define  RXEN0    4
#define  TXEN0    3
#define  UCSZ02   2
#define  RXB80    1
#define  TXB80    0
#define UCSR0A (*(volatile unsigned char *)0xC0)
#define  RXC0     7
#define  TXC0     6
#define  UDRE0    5
#define  FE0      4
#define  DOR0     3
#define  UPE0     2
#define  U2X0     1
#define  MPCM0    0
#define UDR0 (*(volatile unsigned char *)0xC6)

/* USART1 */
#define UBRR1H (*(volatile unsigned char *)0xCD)
#define UBRR1L (*(volatile unsigned char *)0xCC)
#define UBRR1 (*(volatile unsigned int *)0xCC)
#define UCSR1C (*(volatile unsigned char *)0xCA)
#define  UMSEL11  7
#define  UMSEL10  6
#define  UPM11    5
#define  UPM10    4
#define  USBS1    3
#define  UCSZ11   2
#define  UCSZ10   1
#define  UCPOL1   0
#define UCSR1B (*(volatile unsigned char *)0xC9)
#define  RXCIE1   7
#define  TXCIE1   6
#define  UDRIE1   5
#define  RXEN1    4
#define  TXEN1    3
#define  UCSZ12   2
#define  RXB81    1
```

```
#define  TXB81    0
#define UCSR1A (*(volatile unsigned char *)0xC8)
#define  RXC1     7
#define  TXC1     6
#define  UDRE1    5
#define  FE1      4
#define  DOR1     3
#define  UPE1     2
#define  U2X1     1
#define  MPCM1    0
#define UDR1 (*(volatile unsigned char *)0xCE)


/* bits */

/* Port A */
#define  PORTA7   7
#define  PORTA6   6
#define  PORTA5   5
#define  PORTA4   4
#define  PORTA3   3
#define  PORTA2   2
#define  PORTA1   1
#define  PORTA0   0
#define  PA7      7
#define  PA6      6
#define  PA5      5
#define  PA4      4
#define  PA3      3
#define  PA2      2
#define  PA1      1
#define  PA0      0
#define  DDA7     7
#define  DDA6     6
#define  DDA5     5
#define  DDA4     4
#define  DDA3     3
#define  DDA2     2
#define  DDA1     1
```

```
#define  DDA0     0
#define  PINA7    7
#define  PINA6    6
#define  PINA5    5
#define  PINA4    4
#define  PINA3    3
#define  PINA2    2
#define  PINA1    1
#define  PINA0    0

/* Port B */
#define  PORTB7   7
#define  PORTB6   6
#define  PORTB5   5
#define  PORTB4   4
#define  PORTB3   3
#define  PORTB2   2
#define  PORTB1   1
#define  PORTB0   0
#define  PB7      7
#define  PB6      6
#define  PB5      5
#define  PB4      4
#define  PB3      3
#define  PB2      2
#define  PB1      1
#define  PB0      0
#define  DDB7     7
#define  DDB6     6
#define  DDB5     5
#define  DDB4     4
#define  DDB3     3
#define  DDB2     2
#define  DDB1     1
#define  DDB0     0
#define  PINB7    7
#define  PINB6    6
#define  PINB5    5
#define  PINB4    4
```

```
#define  PINB3    3
#define  PINB2    2
#define  PINB1    1
#define  PINB0    0

/* Port C */
#define  PORTC7   7
#define  PORTC6   6
#define  PORTC5   5
#define  PORTC4   4
#define  PORTC3   3
#define  PORTC2   2
#define  PORTC1   1
#define  PORTC0   0
#define  PC7      7
#define  PC6      6
#define  PC5      5
#define  PC4      4
#define  PC3      3
#define  PC2      2
#define  PC1      1
#define  PC0      0
#define  DDC7     7
#define  DDC6     6
#define  DDC5     5
#define  DDC4     4
#define  DDC3     3
#define  DDC2     2
#define  DDC1     1
#define  DDC0     0
#define  PINC7    7
#define  PINC6    6
#define  PINC5    5
#define  PINC4    4
#define  PINC3    3
#define  PINC2    2
#define  PINC1    1
#define  PINC0    0
```

```
/* Port D */
#define  PORTD7   7
#define  PORTD6   6
#define  PORTD5   5
#define  PORTD4   4
#define  PORTD3   3
#define  PORTD2   2
#define  PORTD1   1
#define  PORTD0   0
#define  PD7      7
#define  PD6      6
#define  PD5      5
#define  PD4      4
#define  PD3      3
#define  PD2      2
#define  PD1      1
#define  PD0      0
#define  DDD7     7
#define  DDD6     6
#define  DDD5     5
#define  DDD4     4
#define  DDD3     3
#define  DDD2     2
#define  DDD1     1
#define  DDD0     0
#define  PIND7    7
#define  PIND6    6
#define  PIND5    5
#define  PIND4    4
#define  PIND3    3
#define  PIND2    2
#define  PIND1    1
#define  PIND0    0

/* PCMSK3 */
#define  PCINT31  7
#define  PCINT30  6
#define  PCINT29  5
#define  PCINT28  4
```

```
#define  PCINT27  3
#define  PCINT26  2
#define  PCINT25  1
#define  PCINT24  0
/* PCMSK2 */
#define  PCINT23  7
#define  PCINT22  6
#define  PCINT21  5
#define  PCINT20  4
#define  PCINT19  3
#define  PCINT18  2
#define  PCINT17  1
#define  PCINT16  0
/* PCMSK1 */
#define  PCINT15  7
#define  PCINT14  6
#define  PCINT13  5
#define  PCINT12  4
#define  PCINT11  3
#define  PCINT10  2
#define  PCINT9   1
#define  PCINT8   0
/* PCMSK0 */
#define  PCINT7   7
#define  PCINT6   6
#define  PCINT5   5
#define  PCINT4   4
#define  PCINT3   3
#define  PCINT2   2
#define  PCINT1   1
#define  PCINT0   0


/* Lock and Fuse Bits with LPM/SPM instructions */

/* lock bits */
#define  BLB12    5
#define  BLB11    4
#define  BLB02    3
```

```
#define  BLB01    2
#define  LB2      1
#define  LB1      0

/* fuses low bits */
#define  CKDIV8   7
#define  CKOUT    6
#define  SUT1     5
#define  SUT0     4
#define  CKSEL3   3
#define  CKSEL2   2
#define  CKSEL1   1
#define  CKSEL0   0

/* fuses high bits */
#define  OCDEN    7
#define  JTAGEN   6
#define  SPIEN    5
#define  WDTON    4
#define  EESAVE   3
#define  BOOTSZ1  2
#define  BOOTSZ0  1
#define  BOOTRST  0

/* extended fuses */
#define  BODLEVEL2 2
#define  BODLEVEL1 1
#define  BODLEVEL0 0


/* Interrupt Vector Numbers */

#define iv_RESET        1
#define iv_INT0         2
#define iv_EXT_INT0     2
#define iv_INT1         3
#define iv_EXT_INT1     3
#define iv_INT2         4
#define iv_EXT_INT2     4
```

```
#define iv_PCINT0       5
#define iv_PCINT1       6
#define iv_PCINT2       7
#define iv_PCINT3       8
#define iv_WDT          9
#define iv_TIMER2_COMPA 10
#define iv_TIMER2_COMPB 11
#define iv_TIMER2_OVF   12
#define iv_TIM2_COMPA   10
#define iv_TIM2_COMPB   11
#define iv_TIM2_OVF     12
#define iv_TIMER1_CAPT  13
#define iv_TIMER1_COMPA 14
#define iv_TIMER1_COMPB 15
#define iv_TIMER1_OVF   16
#define iv_TIM1_CAPT    13
#define iv_TIM1_COMPA   14
#define iv_TIM1_COMPB   15
#define iv_TIM1_OVF     16
#define iv_TIMER0_COMPA 17
#define iv_TIMER0_COMPB 18
#define iv_TIMER0_OVF   19
#define iv_TIM0_COMPA   17
#define iv_TIM0_COMPB   18
#define iv_TIM0_OVF     19
#define iv_SPI_STC      20
#define iv_USART0_RX    21
#define iv_USART0_RXC   21
#define iv_USART0_DRE   22
#define iv_USART0_UDRE  22
#define iv_USART0_TX    23
#define iv_USART0_TXC   23
#define iv_ANA_COMP     24
#define iv_ANALOG_COMP  24
#define iv_ADC          25
#define iv_EE_RDY       26
#define iv_EE_READY     26
#define iv_TWI          27
#define iv_TWSI         27
```

```
#define iv_SPM_RDY      28
#define iv_SPM_READY    28
#define iv_USART1_RX    29
#define iv_USART1_RXC   29
#define iv_USART1_DRE   30
#define iv_USART1_UDRE  30
#define iv_USART1_TX    31
#define iv_USART1_TXC   31

/* */

#endif
```

Author's Biography

STEVEN F. BARRETT

Steven F. Barrett, Ph.D., P.E., received the BS Electronic Engineering Technology from the University of Nebraska at Omaha in 1979, the M.E.E.E. from the University of Idaho at Moscow in 1986, and the Ph.D. from The University of Texas at Austin in 1993. He was formally an active duty faculty member at the United States Air Force Academy, Colorado and is now an Associate Professor of Electrical and Computer Engineering, University of Wyoming. He is a member of IEEE (senior) and Tau Beta Pi (chief faculty advisor). His research interests include digital and analog image processing, computer-assisted laser surgery, and embedded controller systems. He is a registered Professional Engineer in Wyoming and Colorado. He co-wrote with Dr. Daniel Pack six textbooks on microcontrollers and embedded systems. In 2004, Barrett was named "Wyoming Professor of the Year" by the Carnegie Foundation for the Advancement of Teaching and in 2008 was the recipient of the National Society of Professional Engineers (NSPE) Professional Engineers in Higher Education, Engineering Education Excellence Award.

Index