Microcontrollers

ENGIN 2223

Alyssa J. Pasquale, Ph.D.

College of DuPage

Contents

List of Figures

List of Tables

1 Changelog

This section will be modified when changes are made to this textbook.

Table 1.1: Table of changes made to this textbook.

Date	Chapter(s)	Description of Change(s)
2021/02/15	all	Changed license to CC-BY-NC-SA
2021/07/12	all	Modified formatting and moved all figures to be in-line with the text
2021/07/12	4.10	Included clarifying details on top-down vs. bottom-up design
2021/07/12	4.11	Updated list of simulation software tools used in digital systems
2021/07/12	4.11	Added caveat about limitations to Tinkercad's Arduino simulation
2021/07/12	5.8	Added more explanation about what is meant by random-access memory and included information on its opposite (sequential-access memory)
2021/07/12	9.1	Included more information on each type of chip package

2021/07/12	15	Changed terminology used in serial communication architecture (see author note)
2021/07/14	note	Added attribution information
2022/03/23	14	Added more information on fast PWM and phase-correct PWM
2022/04/18	6	Added a reference for packed BCD addition in the half carry flag section
2022/04/20	10	Corrected a typo in output low current section and changed the wording in the current-level compatibility section to be more clear how current and logic levels are related.
2022/04/27	all	Made minor formatting changes

2 Introduction

This book contains all of the information you will need to learn about the ATmega328P microcontroller as well as about embedded system design. First, microcontrollers and embedded systems will be described and explained at a zoomed-out level. Then, each of the peripheral features of the microcontroller will be explored in more detail. These topics include I/O port registers, analog to digital conversion, interrupts, timers/counters, clock systems, pulse-width modulation, serial communication, memory addressing, CPU registers and condition codes, and a basic overview of assembly language. Some chapters have additional practice problems to aid in studying the material.

This book should be used in conjunction with the ATmega328P datasheet[1] as well as with the course lab manual. A list of other useful and recommended textbooks is included in chapter 3.

[1] Atmel, "ATmega328/P Datasheet Complete," November 2016

2.1 Author Note

The original versions of this textbook contained racist terminology for terms used in serial communication. I am sincerely sorry for using this terminology, and am making a concerted effort to become more educated. This has been changed as of 2021. However, due to the fact that the ATmega328P microcontroller still uses this terminology in their documentation, I will do my best to try to alleviate confusion due to the differences in terminology. There is a great article from Boston University that explains this situation.

2.2 License & Attribution Information

This book is licensed under creative commons as CC-BY-SA-NC. This license allows reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix,

3
Course Prerequisites

The topics in Table 3.1 should be well-understood before beginning this course.

Table 3.1: Prerequisite topics to understand before taking microcontrollers.

Prerequisite Topics
Number systems (especially decimal, binary and hexadecimal)
Binary arithmetic, both signed and unsigned
Detection of overflow in binary arithmetic (signed and unsigned)
Hexadecimal addition
ASCII code
Boolean logic
AND, OR, NOT, XOR, NAND, NOR gates
Multiplexers and decoders
Combinational and sequential logic design
Shift registers (especially PIPO, SIPO and PISO)

Textbooks that may assist you in learning the material for this course are listed in Table 3.2.

Table 3.2: Recommended textbooks.

Title	Author
Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment	David Russell
The Atmel AVR Microcontroller: MEGA and XMEGA in Assembly and C	Han-Way Huang
AVR Microcontroller and Embedded Systems: Using Assembly and C	Muhammad Ali Mazidi, Janice Mazidi
Atmel AVR Microcontroller Primer: Programming and Interfacing	Steven F. Barrett, Daniel J. Pack

4
Introduction to Microcontrollers & Embedded System Design

To UNDERSTAND MICROCONTROLLERS, one must first understand computers. What is a computer? These ubiquitous devices may be very familiar, but to understand their design more analytically, their components must be understood. A block diagram, shown in Figure 4.1, breaks down the components of a computer.

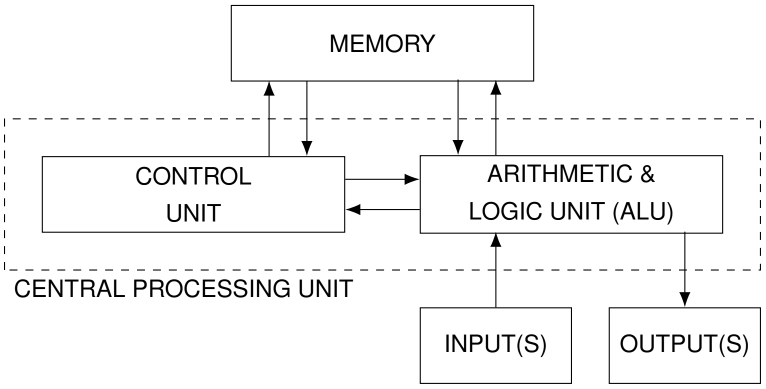


Figure 4.1: A block diagram of a computer, consisting of memory, a central processing unit, inputs and outputs.

4.1 Computer Components

The main computer components, as highlighted in this block diagram, include: memory, a control unit, an arithmetic and logic unit (ALU), inputs, outputs, and bus lines. Each of these components are explained briefly in this chapter, and then in more detail in chapter 5.

Computer Memory

Computer memory contains all of the information necessary to allow the computer to boot up, run programs, and access data. The instructions that tell the computer how to boot up is known as firmware. Program memory is referred to as software. As discussed later, this is much different from how microcontroller memory is organized. However, memory still plays a large role in storing program instructions and variable data information.

Control Unit

The control unit reads and interprets program instructions. It also sends control signals through the control bus. These signals instruct the computer to read or write to memory, control the timing of data transfer, and otherwise sequences the computer processor to take the necessary steps in the necessary timing to carry out the program.

The control unit uses something called a program counter (which is explored in more detail in chapter 5) to keep track of the current instruction, and a special piece of memory known as the status register (which is explained in chapter 6) to keep track of the most recently executed operation.

Arithmetic and Logic Unit (ALU)

As may seem obvious from the name, the arithmetic and logic unit (ALU) is capable of performing many arithmetic (addition, subtraction, multiplication) and logic (AND, XOR, OR) functions. The functions that are available in the ALU dictate the types of instructions that are possible to execute on the computer. The ALU together with the control unit form the central processing unit (CPU) of a computer.

Input and Output (I/O) Devices

Computers use a large number of input and output devices. Input devices gather information from the outside world to help the computer make decisions about what processes to carry out. Common computer inputs include: keyboard, mouse, touch screen, microphone, switches, and light detectors. Output devices allow the computer to display information for humans to see. Common computer outputs include: monitor, speakers, haptic feedback, LEDs, displays, and buzzers.

Bus Lines

Bus lines are interconnections between components. Bus lines are simply connections of multiple "normal" wires. Bus lines in this book are indicated by very thick lines. However, they may also be drawn as lines with a slash. A number next to the slash indicates the number of bits in each bus. Both of these possibilities are shown schematically in Figure 4.2.

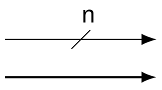


Figure 4.2: Two possible schematics of bus lines.

Many "micro" devices exist today: microcomputers, microcontrollers, and microprocessors. Understanding this hierarchy of devices, outlined in Figure 4.3, will put microcontrollers in context, after which they will be focused on exclusively.

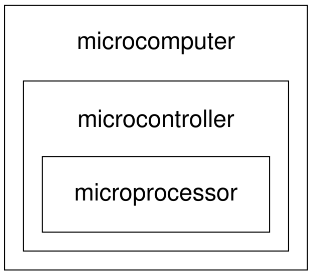


Figure 4.3: Hierarchy of "micro" devices: microcomputers consist of microcontrollers, which consist of microprocessors.

4.2 *Microprocessor*

A microprocessor is a single integrated circuit (IC), generally consisting of just the CPU. If you have ever built a computer, this is the part that you purchase from AMD or Intel, and place into the motherboard. Thermal paste is put on top to thermally link it to a heat sink, which allows heat to safely dissipate away from the sensitive electronics inside of the processor. Less expensive, less powerful microprocessors are available for tasks that are not as extensive as those needed in general purpose computing.

Advantages of microprocessors

- Small
- Inexpensive
- Customizable and versatile: the user can decide what and how many peripheral devices to include

Disadvantages of microprocessors

- No on-chip memory
- No peripheral functions included
- Cannot use directly with I/O devices

4.3 *Microcontroller*

A microcontroller is a microprocessor that includes memory and I/O functionality. These are used in embedded systems, and are ubiquitous in today's technological world. Five of the major 8-bit microcontrollers are outlined in Table 4.1.

Vendor	Microcontroller(s)
Freescale Semiconductor	68HC08/68HC11
Intel	8051
Zilog	Z8
Microchip	AVR*
	PIC

Table 4.1: Five of the major 8-bit microcontrollers and their manufacturers.

Each of the microcontrollers listed in Table 4.1 has a specific instruction set, their own set of peripheral devices and I/O pins, and are generally not interchangeable. *Note: Atmel, the original producer of AVR microcontrollers, was acquired by Microchip in 2016. Many AVR manuals and datasheets still contain the Atmel logo.

4.4 *Embedded Systems*

Whereas a PC or a laptop is a general use machine (used for games, Internet, music, word processing, etc.) embedded systems refer to single-function devices. There are many examples of these in the world around us, but a good rule of thumb is that an embedded system is capable of performing computations without the use of an operating system (such as Windows, Linux, macOS, iOS, etc.). Embedded systems can be found in watches, MP3 players, vending machines, and more. In these examples, a full computer would be detrimental to the operation of the device. Imagine having to boot up Windows to run a dishwasher!

4.5 *Choosing a Microcontroller*

How does one choose a microcontroller to use in an embedded system project? It first must meet all project requirements and it must include peripherals and accessories that make it relatively simple to develop products around the microcontroller. In addition, it is important to ensure that the microcontroller not only is available now, but will also be available into the future. Some considerations in making this determination are outlined in Table 4.2.

Project Requirements
Speed
Packaging (DIP, surface-mount, etc.)
Power consumption
Memory
Peripherals (timers, ADC, etc.)
Number of I/O pins
Ease of upgrade
Cost
Additional Accessories
An available assembler
A debugger
A compiler for high-level programming languages such as C
Technical support

Table 4.2: Project requirement and additional accessory considerations when choosing a microcontroller.

4.6 *Atmel AVR Microcontroller & Arduino*

The microcontroller that will be used in this class is the AVR ATmega328P. It is packaged in an Arduino Uno which contains extra features including (but not limited to): power regulator, bootloader, USB connection, I/O pins connected to headers, and the availability of the Arduino IDE for writing C code. The Arduino is a relatively inexpensive microcontroller package. By the end of this class, you will have all of the information you need to develop your own embedded systems and "smart projects" at home using this platform.

The ATmega328P datasheet[1] is the single most useful and important document to understanding everything that needs to be known about the microcontroller. It is available for download on the Microchip website. The first time you look at it, it may seem overwhelming. Perhaps you only understand a few words here and there. Do not be intimidated. Every time you refer to the datasheet, you will find that you understand a little more. Eventually, your knowledge will expand, and more and more of the datasheet will be part of your understanding. At the very least, become comfortable with the features of the ATmega328P, which are detailed on the front page of the datasheet. Some of this front page has been paraphrased in Table 4.3

[1] Atmel, "ATmega328/P Datasheet Complete," November 2016.

Table 4.3: ATmega328P datasheet front page.

High Performance, Low Power Atmel AVR 8-Bit Microcontroller Family
Advanced RISC Architecture

131 Powerful Instructions
32×8 General Purpose Working Registers
Up to 20 MIPS Throughput at 20MHz
High Endurance Non-volatile Memory Segments
32KBytes Flash program memory
1KBytes EEPROM
2KBytes Internal SRAM
Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
Data retention: 20 years at 85°C/100 years at 25°C
Peripheral Features
Two 8-bit Timer / Counters
One 16-bit Timer / Counter
Six PWM Channels
6-channel 10-bit ADC in PDIP Package
Programmable Serial USART
Primary/Secondary SPI Serial Interface
Byte-oriented 2-wire Serial Interface (Philips I^2C compatible)
Programmable Watchdog Timer with Separate On-chip Oscillator
On-chip Analog Comparator
Interrupt and Wake-up on Pin Change
Special Microcontroller Features
Power-on Reset and Programmable Brown-out Detection
Internal Calibrated Oscillator
External and Internal Interrupt Sources
Six Sleep Modes
Power Consumption at 1MHz, 1.8V, 25°C
Active Mode: 0.2mA
Power-down Mode: 0.1μA
Power-save Mode: 0.75μA (Including 32kHz RTC)

4.7 *Embedded System Programming*

Computers and microcontrollers are useless without operating instructions, which are written in various programming languages. There is a hierarchy of language types, from machine language to high-level programming languages such as C. This hierarchy is depicted in Figure 4.4.

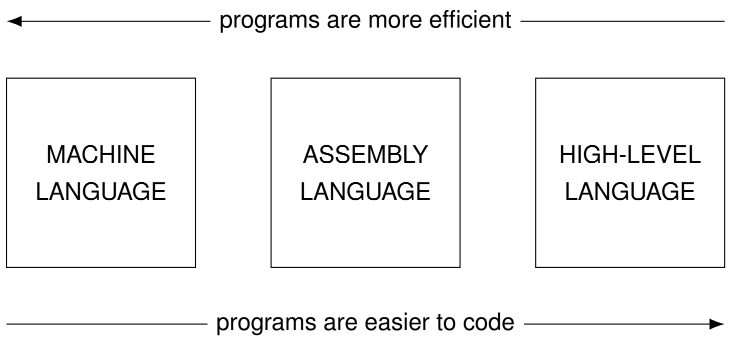


Figure 4.4: Hierarchy of language types and their relation to efficiency and simplicity.

Machine Language

Machine language is the instruction set of the microprocessor converted into binary. For example, the instruction 0001 1101 0100 1111 on the ATmega328P adds numbers from two registers (15 and 20) together. This code is then routed through hardware (the digital logic "guts" of the microprocessor) via the instruction decoder to route the appropriate control, address, and data signals to the associated hardware to carry out the desired task. Machine language can be very tedious to write and difficult to debug. It used to be accomplished using punch cards and magnetic tape.

Assembly Language

Assembly language uses mnemonic codes to refer to each instruction rather than the actual binary code. The above machine instruction could be rewritten into assembly as `ADD 15,20`. Assembly is converted into machine code by a program called an assembler. Families of microcontrollers and microprocessors have specific instructions that they are capable of carrying out. For this reason, assembly code cannot be copy/pasted directly from an AVR microcontroller to an Intel processor, for example. The ATmega328P, being an AVR microcontroller, uses the AVR instruction set. Being familiar with the allowable instructions on a microcontroller means that code can be written very efficiently and compactly; assembly code generally uses less memory and takes less time to execute than code written using higher-level languages. AVR assembly is discussed in more detail in chapter 19.

High-Level Programming Language

High-level programming languages use functions to accomplish what assembly and machine language does. Addition can be carried out using familiar arithmetic symbols, for example: `f = 152 + 38;` High-level language code is converted into assembly and from there to machine language by a compiler. Examples of high-level languages are C, C++, Python and Visual Basic.

Using high-level languages means that more time can be spent working on algorithms rather than on the specifics of what machine instructions to call in assembly. Initialization occurs without specifically having to do so; the stack, stack pointer, memory addresses, etc. are all allocated automatically by the compiler. In addition, high-level programs in the same language can more-or-less be recycled from one microcontroller to another (with caveats, not all registers have the same names, some functions may be a bit different, etc.). C concepts that are pertinent to microcontroller programming is discussed in more detail in chapter 18.

4.8 Compilers

Compilers ensure that the high-level programming language code is correct, both in syntax and in memory allocation. Errors or warnings are usually displayed in these cases, and codes with errors are not loaded onto the microcontroller. Once the code is correct, the compiler takes the code and converts it into assembly language, and from there generates a HEX file which contains all of the machine code that needs to go into memory on the microcontroller.

4.9 Embedded System vs. Computer Program Design

Designing for embedded systems is very different from writing programs for computer (desktop, laptop, and mobile) applications. This has a lot to do with the resources, especially memory, available on each type of device. These differences are outlined in Table 4.4.

Table 4.4: Embedded system vs. computer application programming and resource differences.

	Embedded System	**Computer**
Program Data Location	ROM	RAM
RAM Capacity	2 kB (ATmega328P)	Nearly unlimited
ROM Capacity	32 kB (ATmega328P)	Nearly unlimited
Use of Peripherals (ADC, clock, etc)	Frequent	Rare
Assembly Code Usage	Frequent	Rare

Desktop Memory

Desktop computers and laptops have a nearly unlimited supply of memory of all types.

Onboard flash memory is used to store non-changing information containing instructions that tell the computer what to do when it boots up. This is usually known as BIOS (basic I/O system).

Random-Access Memory (RAM), while somewhat a misnomer, stores program data while the program is running. For example, clicking on a PDF copies the entire Adobe program into RAM where it runs from there. This happens because there are many gigabytes of RAM available on most computers, and RAM tends to be faster than other computer memory types.

Read-Only Memory (ROM), which historically has mostly been made from magnetic disk drives, and these days is being usurped by solid-state memory, contains program executable storage and non-variable data files.

Embedded Systems Memory

Embedded systems have much more limited memory capacity and options.

Flash memory is non-volatile and stores program instructions and defined (constant) data.

RAM is volatile memory that contains variable data. Because RAM is a limited resource, it is necessary to be aware of size constraints while writing applications. For example, an array with 500 float values uses more RAM than the ATmega328P microcontroller can support.

Use of On-Chip Features

With embedded systems, a lot of peripheral features are used to interface with and control I/O devices, which doesn't happen on a PC. Configuring each of these peripheral functions requires a high level of understanding of each of their individual control registers.

ATmega328P Peripheral Features

- Analog to Digital Converters
- External Interrupts
- 8- and 16-Bit Timers
- Watchdog Timer
- USART, SPI, and TWI Serial Communication

Use of Assembly Code

Assembly code is frequently, but not always, used to program instructions onto a microcontroller. Assembly code uses less memory than equivalent C code, and their programs tend to be more efficient. Using assembly leads to great knowledge of the microcontroller.

4.10 *Top-Down Design / Bottom-Up Implementation*

For embedded control projects it is important to **design** before writing code or wiring up hardware. Rather than diving in head first and starting to wire things up and write code, it is important to first understand the problem. Break down the project into as many modules or parts as you can think of, and consider how they interconnect or interrelate. This is called top-down design (as if you are looking at the project from above and looking at each individual part). Once each module or part has been defined, then design and implement each one individually, rather than trying to tackle the entire project at once. This is called bottom-up implementation. This design process is explained in Table 4.5.

Table 4.5: Top-down design and bottom-up implementation process.

Top-Down Design
Understand the problem completely
– STOP – do not write any code yet!
– Specify requirements
– Think about possible errors
– Think about the "what", not the "how"
– Don't get caught up in minutiae
Design in levels
– Break the problem into parts
– Then break those parts into even more parts
– Build a block diagram of all parts and how they interrelate
– Start refining details
Bottom-Up Implementation
Implement one subsystem at a time
– Implement simpler subsystems first
– Integrate everything at the end

At every step, it is important to keep detailed documentation of the project. Engineers keep notebooks in order to record ideas, thoughts, requirements, things that don't work, and things that do work.

The benefits of this design strategy is that it helps to clarify the problem. You can't design something if you're not completely sure what it should do. In addition, as parts get broken down into smaller pieces, they become less complicated. You may realize that parts of the solution may be reusable (e.g. a module built for one component may have have code that can be used elsewhere in the design). Finally, when breaking a system down into parts, more than one person can work on the solution.

4.11 Design Tools

Design tools are always available to aid in project design and implementation.

One important design tool is the use of **flowcharts**. Flowcharts describe the steps that a program must implement, so that it is simpler to write the corresponding software code. Another benefit of flowcharts is that they are language-independent. In other words, given a flowchart, the program could then be implemented using the C programming language as easily as in Assembly. An example flowchart is shown in Figure 4.5.

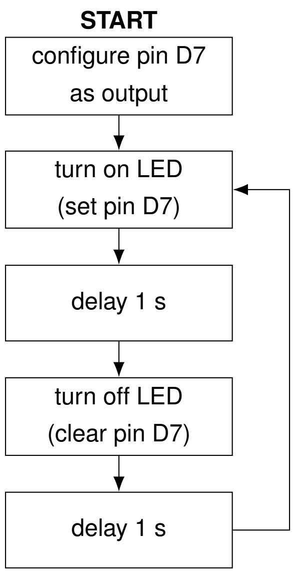


Figure 4.5: Example flowchart of a circuit that blinks an LED on and off with one second delays in between.

Another important design tool is **circuit simulation software** which tests circuit functionality before wiring any hardware or writing any software code. Working with simulation software may be a large part of your future job. There are many different software packages that exist. You may have used CircuitVerse, Logisim, or NI Multisim in Introduction to Digital Systems, which are hardware simulation packages. Tinkercad has circuit simulation software that allows testing of both the hardware and software of the Arduino Uno. This gives you the benefit of trying out new designs without necessarily having any access to an Arduino or any other components. (Note that not all Arduino functionality is possible on Tinkercad.

Notably, many of the serial communication protocols have not been implemented in Tinkercad as of early 2020.)

4.12 Debugging

Once a project has been designed implementation is begun, it will inevitably be time to start debugging. In order to make debugging as painless as possible, be smart about how code is implemented. Work on high-level functions first. Once they are working perfectly, add other subroutines only one at a time. If more than one bit of extra functionality is added at once, and something doesn't work, it will be difficult to determine what component of the design doesn't work. In the process, keep code well-commented, and document design decisions, research, implementation ideas, and notes about what does and doesn't work (and why) in a design notebook.

If something doesn't work (which will inevitably happen in any design), rather than diving in headfirst and changing code, find out what the code is actually doing. Then, and only then, can you change it to what it should be doing. This requires an intimate understanding of both the software and hardware components of your design. It may first need to be determined if hardware or software is at fault. For this reason it is imperative to test hardware before integrating it with software, to avoid problems later on. Several hardware testing and software debugging steps are suggested in Table 4.6.

Hardware Testing
Check that LEDs light before using them
Check pushbuttons with a multimeter on continuity mode
Use a tabletop multimeter or digital logic probe (for static signals)
Use an oscilloscope (for time-varying signals)
Software Debugging
Code walkthroughs: have your peers take a look at your code
Walk through the code step-by-step and see what results
Comment out lines of code one line at a time until the program works, then uncomment one line at a time until it doesn't work

Table 4.6: Hardware testing and software debugging procedures.

5
General Principles of Microcontrollers

The central processing unit (CPU) of a microcontroller contains many parts that are crucial to carrying out all possible instructions, including the arithmetic and logic unit, registers, program counter, instruction decoder, and memory.

5.1 CPU Architectures

There are two main types of CPU architectures used in computing devices. **Harvard** architectures keep program and data memory separate. This allows the microcontroller to simultaneously read an instruction and write information to data memory. A simplified Harvard architecture CPU block diagram is shown in Figure 5.1. This simultaneous instruction and data access increases the speed of the device. Separate storage also means that program and data memories can be different widths. Data memory is restricted to 8 bits, but program memory is 16 bits on the ATmega328P.

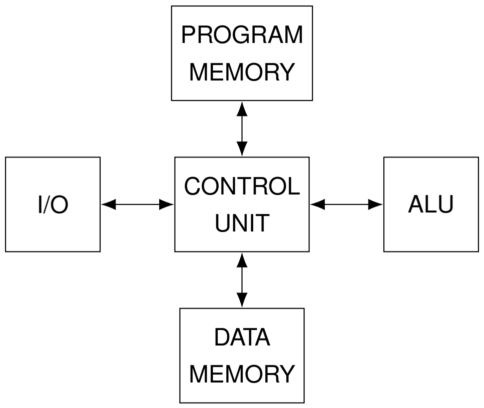


Figure 5.1: A Harvard CPU architecture features separate program and data memory.

In contrast, **von Neumann** architecture (named after mathematician and physicist John von Neumann) addresses program and data memory with a single bus. A simplified von Neumann architecture CPU block diagram is shown in Figure 5.2. It is used extensively on PCs because RAM and ROM are external to the CPU and using separate wire-traces for each on the motherboard would be expensive.

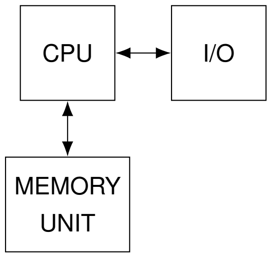


Figure 5.2: A von Neumann CPU architecture features a single bus for program and data memory.

ATmega328P Microcontroller

The ATmega328P microcontroller makes use of a Harvard architecture, meaning that program memory and data memory are stored in different locations. A block diagram of the ATmega328P CPU is shown in Figure 5.3. (Note that while virtually all of the interconnects are in actuality buses, bus notation is not used in this block diagram for simplicities sake.)

5.2 Reduced Instruction Set Computing (RISC)

Computers in the 1980s started to be programmed with every conceivable instruction, not all of which were used, which led to very complicated instruction codes and CPUs. Reduced instruction set computing (RISC) processors use only a limited number of instructions and have a fixed instruction size; most ATmega328P instructions are 16 bits (some are 32 bits).

Computers making use of RISC generally feature many registers, because data cannot be manipulated directly in memory (data must first be loaded into a register). RISC processors have a small instruction set, which is not a problem when using a high-level programming language such as C (but can make assembly coding more tedious). Most RISC instructions can be completed within a single clock cycle.

5.3 Machine Instructions

Instructions form the basis of all digital computing. A piece of binary data, called an instruction, contains information about the specific operation to be carried out. There are arithmetic operations such as

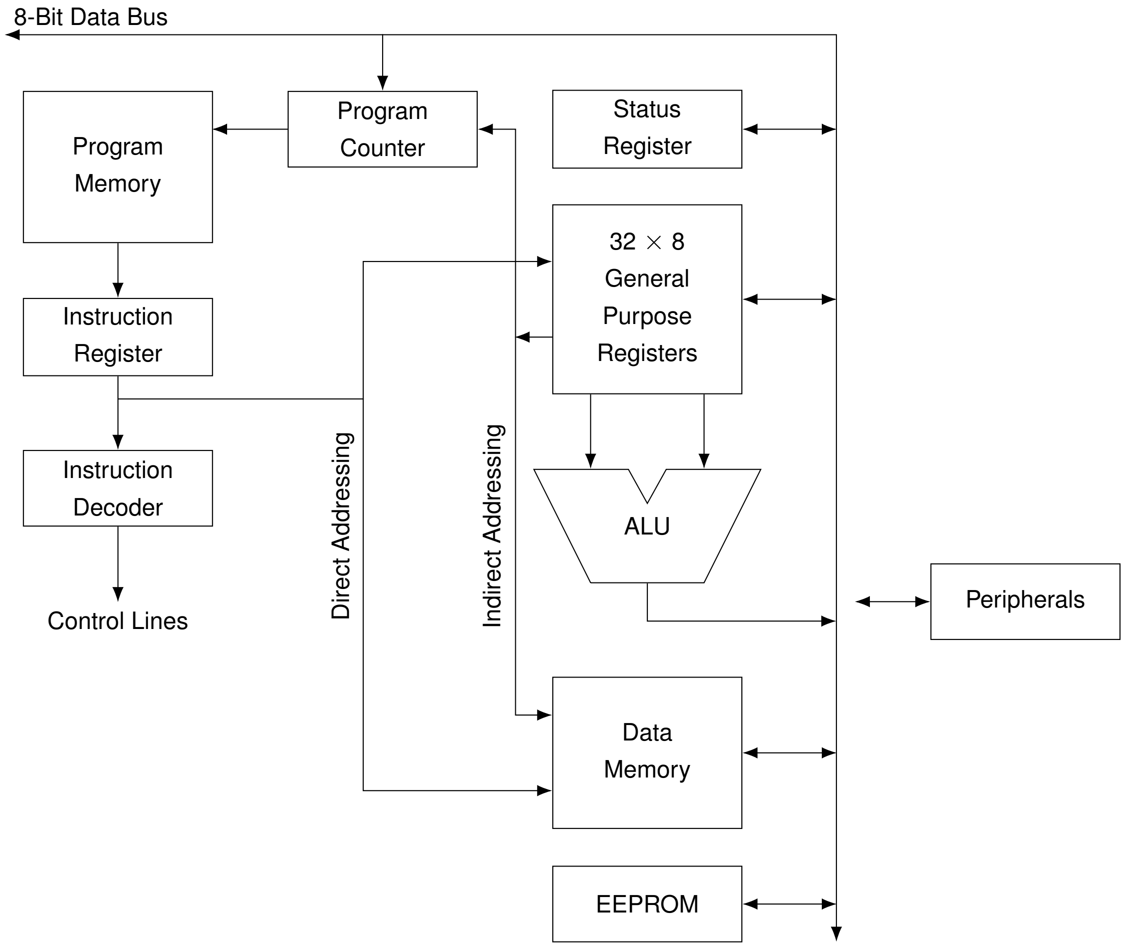


Figure 5.3: Block diagram of the ATmega328P central processing unit (CPU).

addition and subtraction, branch operations such as jump to another piece of code, data transfer operations such as loading or copying data from one place to another in memory, bit instructions such as clearing and shifting data, and control instructions such as sleeping or resetting the watchdog timer. In addition to containing information about the operation (known as the opcode), the instruction also contains information about what data is to be manipulated (known as the operand or operands). Each of these instructions is then parsed in hardware by the instruction decoder, which then routes the appropriate control and data signals to the arithmetic and logic unit (ALU).

In general, for every n opcodes, there must be a $\lceil \log_2(n) \rceil$ bit number to store it. The AVR instruction set for 8-bit microcontrollers, which is used for the ATmega328P microcontroller, has approximately 130 unique instructions.[1] $\lceil \log_2(130) \rceil = 8$, which means that 8 bits are needed in each 16-bit instruction to specify which operation will occur. This leaves only 8 bits for the remaining operand(s). There

[1] Atmel, "AVR Instruction Set Manual," November 2016.

are 32 general purpose registers that can be used to temporarily store data and many instructions exist that require the contents of two of these registers. In that case, 8 bits is insufficient to properly address both of these registers (each of which requires 5 bits to address).

To get around this limitation, the AVR instruction set uses the concept of variable length instructions, or **expanding opcodes**. Instructions that have many or lengthy operands are designed to have short opcodes while instructions with few or no operands are designed to have long opcodes. Most AVR instructions are 16 bits, with a few 32-bit instructions used as needed. There are 32 general purpose (GP) registers, and the ATmega328P additionally has 32,768 bytes of program memory and 2,048 bytes of data memory, which need to be addressed. In addition, there are other specialty registers such as a status register and stack register that can be affected by machine instructions.

To highlight the use of expanding opcodes, consider the following instructions which run the gamut from having a long opcode to having a short opcode.

- The instruction `CLI` is used to clear the global interrupt flag in the status and control register `SREG`. Because this instruction requires no operands, the full 16 bits of instruction are used to store the opcode.

- The instruction `NEG` converts the contents of a single GP register into a 2's complement number. Because there are 32 GP registers, 5 bits are required for the operand, leaving 11 bits remaining for the opcode.

- Most instructions have two GP registers as operands. For example, the instruction `ADD` will sum the contents of two GP registers. 10 bits are required to address both of the operands, leaving 6 bits for the opcode.

- A few instructions require very long operands. In order to properly address up to 4 MB of memory, it is necessary to use 22 bits of data for the operand. In order to accomplish this, a 32-bit instruction is used. 22 bits are dedicated to the memory address with the other 10 bits used for the opcode.

While expanding opcodes allows for much greater flexibility in instruction operands, it also makes for more difficult instruction decoding. This is because there is no longer a fixed position and length for each opcode within each instruction.

5.4 *Instruction Decoder*

The instruction decoder translates the instruction into appropriate control and address signals. This can take place using combinational logic gates or by using a programmed ROM. As an example, the `ADD` (add without carry) instruction requires two general purpose (GP) registers as operands (one of which is also designated as the destination register where the result will be saved). The instruction decoder will address these two GP registers and route their contents to adder hardware in the ALU, then route the solution back to the destination register.

5.5 *Arithmetic and Logic Unit (ALU)*

The ALU contains all of the hardware that is necessary to perform arithmetic and logic operations. While most ALU hardware designs are proprietary, taking a look at the instruction set gives a good idea of the type of hardware that must be included within the ALU. For example, the AVR ALU must contain, at the very least, an adder, logic gates such as AND, OR, XOR, and NOT, and other similar hardware.

5.6 *Registers*

Registers are ubiquitous in microcontroller design. **General purpose registers** contain data that we can immediately operate on, and allows us to store data with easy access to the ALU before saving results back to memory. **Memory address registers** or **pointer registers** tell the microcontroller where to look in program data for the next instruction to perform. **Status registers** (`SREG` on the ATmega328P) store information related to the most recent operation that has been executed on the microcontroller. **I/O registers** and extended I/O registers (sometimes called **peripheral registers**) store information regarding the operation of the I/O pins and their peripheral features. Some of these, `DDRxn`, `PORTxn`, and `PINxn` are discussed in chapter 10. In addition, there is a **stack pointer register** that is used to hold memory addresses in short term storage.

Register Architectures

There are four main types of register architectures: serial in / serial out (SISO), parallel in / parallel out (PIPO), serial in / parallel out (SIPO) and parallel in / serial out (PISO). They are defined by their size, which has to do with the number of flip-flops that they consist of, and ultimately tells us how many bits of data can be stored within them. Data can be loaded either serially or in parallel. A shift operation can be added to registers to increase their functionality, at the expensive of added electronics.

Serial in / serial out (SISO) registers can be used in data buffering, storing data temporarily while it is in between its source and its destination. Serial data is sent to the input of the first flip-flop, and at each clock cycle the signal is shifted through subsequent flip-flops. Registers with serial input take the longest time for data to transmit from one end to the other. Registers with serial output additionally take longer than parallel output devices to access the data stream at the end. A schematic of a 4-bit SISO register is shown in Figure 5.4.

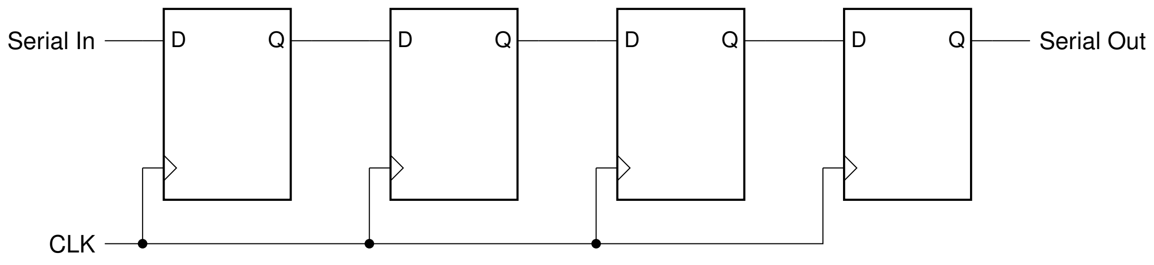


Figure 5.4: Schematic of a 4-bit serial in / serial out (SISO) register.

Parallel in / parallel out (PIPO) registers are used in general purpose I/O registers, among other applications. Data is immediately made accessible to all flip-flops simultaneously. In addition, all of the outputs are immediately available. At each clock cycle, the outputs are updated to reflect any new data available on the inputs. PIPO shift registers are available that have control signals to either parallel load the data or to shift data in either direction. A schematic of a 4-bit PIPO (non-shift) register is shown in Figure 5.5.

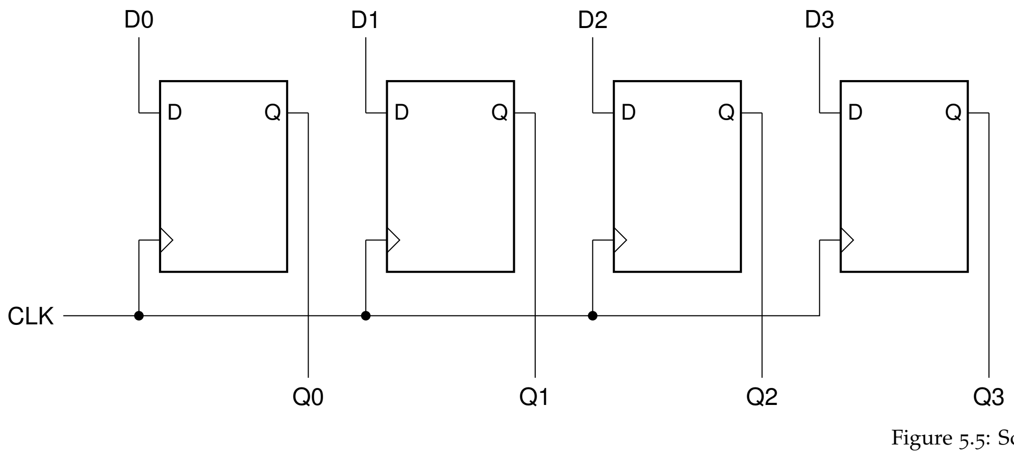


Figure 5.5: Schematic of a 4-bit parallel in / parallel out (PIPO) register.

Serial in / parallel out (SIPO) registers are used to input serial data from a serial communication protocol, and then output that data either to a general purpose register or memory. Serial data is sent to the input of the first flip-flop, and at each clock cycle the signal is shifted through subsequent flip-flops. All outputs are immediately available. A schematic of a 4-bit SIPO register is shown in Figure 5.6.

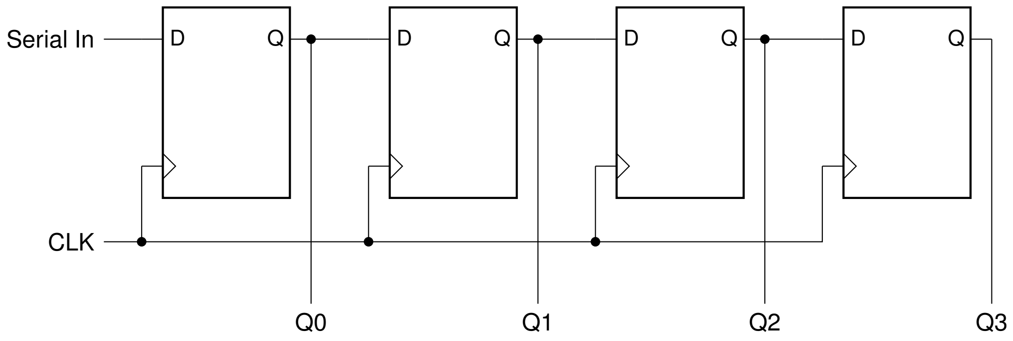


Figure 5.6: Schematic of a 4-bit serial in / parallel out (SIPO) register.

Parallel in / serial out (PISO) registers are used to input data from a general purpose register or memory and to output that data serially through a serial communication protocol. A schematic of a 2-bit PISO register is shown in Figure 5.7 (only two bits are shown to save space). A control signal $\overline{\text{W}}$/S is used to control if the data loads into the flip-flops (which occurs when the signal is held LOW) or if data shifts through the flip-flops (which occurs when the signal is held HIGH).

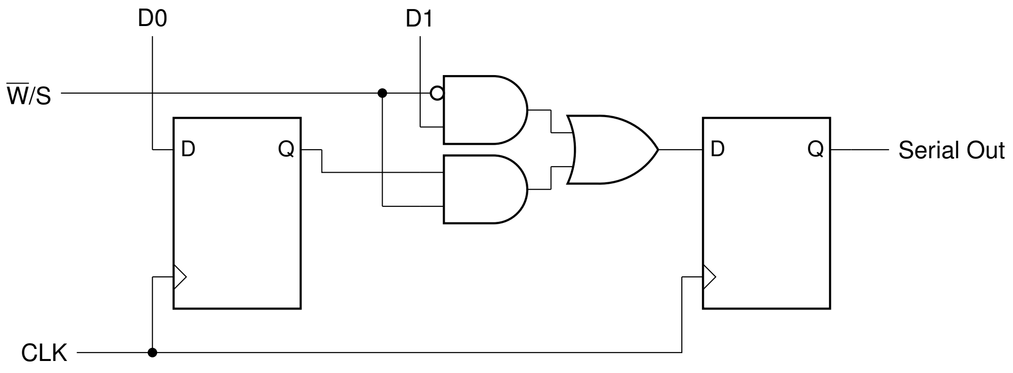


Figure 5.7: Schematic of a 2-bit parallel in / serial out (PISO) register.

5.7 *Program Counter (PC)*

The program counter contains the address (in memory) of the first byte of the instruction to be executed. It is incremented after every instruction. A schematic, shown in Figure 5.8, shows the inner workings of a program counter. When the power is switched on, the program counter is forced to 0 and the instruction fetch starts from address 0 ($\overline{\text{reset}}$ = 0 forces the flip-flops to a value of 0).

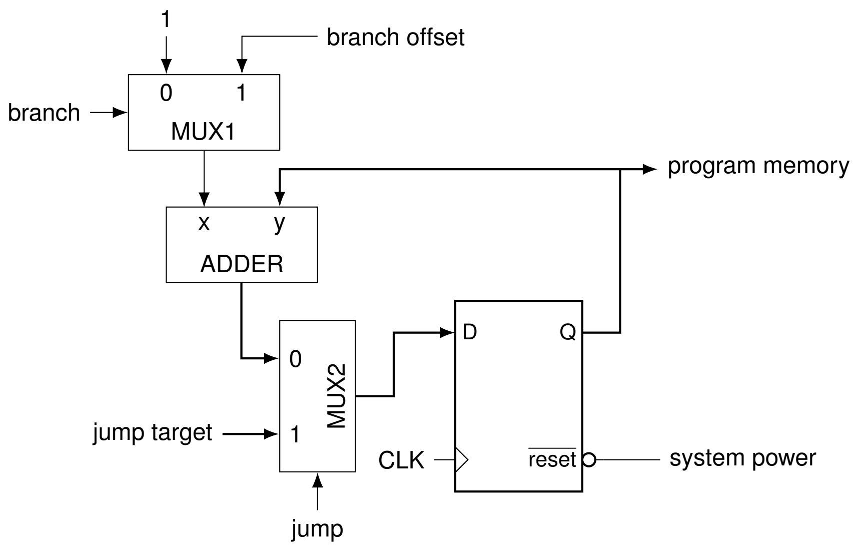


Figure 5.8: Schematic of a program counter (PC).

If the instruction is a conditional branch instruction (which is discussed in chapter 6), branch = 1, and the sum of the current PC value (`Qs`) is added to the branch offset (which is passed through MUX1) and that value is sent to `Ds` to update the PC value.

If the instruction is a jump instruction, then the value of jump target (which is passed through MUX2) is loaded into `Ds` to update the PC value.

If the instruction is not a program flow control instruction (i.e. it is not a branch or jump instruction), then the PC is incremented by 1 after each instruction is fetched.

5.8 Memory

The two types of memory are volatile and non-volatile. This refers to whether or not data can persist after power has been removed from the microcontroller or computer. Before discussing these types of memory systems, however, it is important to note that the way that memory is addressed (i.e. how each piece of memory is stored or recalled) is also of critical importance to the operation of a microcontroller. This is discussed in chapter 7.

Volatile Memory

Volatile memory is used for temporary storage because data is unable to persist after power has been removed from the device. It is usually called random-access memory (RAM). This concept of random-access refers to the ability to obtain data from any arbitrary (or random) address in memory at any given time. Types of RAM include static RAM (SRAM), dynamic RAM (DRAM), and registers.

RAM differs from sequential-access memory, in which the data must be accessed in the same order in which it was stored. Sequential-access memory is the type of memory obtained when using rolls of magnetic tape (for example). A tape reader must spool through the entire reel to read from beginning to end in memory, and cannot skip around.

SRAM is the type of volatile memory used in the ATmega328P. A schematic of SRAM is given in Figure 5.9, and the modes of operation are explained in Table 5.1. Each inverter in the schematic must be connected to Vcc and ground in order to retain its data.

Static RAM Characteristics

- Uses flip-flops
- Faster than DRAM
- More expensive than DRAM
- Low power consumption

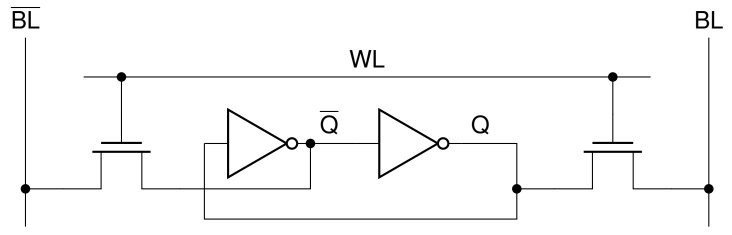


Figure 5.9: Schematic of a Static RAM cell.

Table 5.1: Description of Static RAM modes of operation.

Modes	Description
Standby	WL = LOW, mosfets are OFF (open switches) Q and $\overline{Q}$ will not change and are floating

Write	WL = HIGH, mosfets are ON (closed switches)
	BL and $\overline{BL}$ are asserted
	Q = BL, $\overline{Q}$ = $\overline{BL}$

Read	WL = HIGH, mosfets are ON (closed switches) read data from BL and $\overline{BL}$

DRAM is the type of volatile memory used in computers due to its small size and low cost. A schematic of DRAM is given in Figure 5.10, with the modes of operation explained in Table 5.2. If the capacitor is not continuously refreshed with either a value of one or zero, it will discharge and memory will be lost.

Dynamic RAM Characteristics

- Uses one transistor and one capacitor
- Slower than SRAM
- Cheaper than SRAM
- Takes less space than SRAM
- Consumes more power than SRAM
- Requires constant refreshing

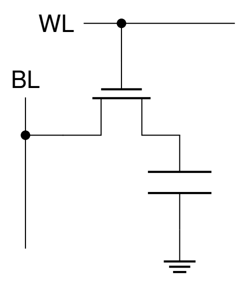


Figure 5.10: Schematic of a Dynamic RAM cell.

Modes	Description
Write	WL = HIGH, mosfets are ON (closed switches) BL is asserted, must wait for cap to charge or discharge
Read	WL = HIGH, mosfets are ON (closed switches) charge stored on cap goes to BL value must be re-written after every read

Table 5.2: Description of Dynamic RAM modes of operation.

A subset of DRAM known as pseudostatic RAM (PSRAM) uses memory cells similar to that of DRAM, but includes circuitry that allows it to refresh itself, which means that it has the same ease-of-use as SRAM.

Non-Volatile Memory

Non-volatile memory is used for more permanent storage because the data persists after power has been removed from the device. Over the years, non-volatile memory has taken on many different diverse forms: paper punch cards; wax cylinders and records; optical CD and DVD disks; magnetic disks, drives and tape; and semiconductor drives. Because non-volatile memory doesn't require power to hold on to stored data, it is used to store firmware on computers, and program instructions and constants on microcontrollers. It is usually referred to as read-only memory (ROM), due to the fact that in regular operation the memory is effectively read-only. Writing to ROM typically requires a higher voltage than a read operation requires, and in addition write operations take more time than read operations.

The first types of ROM included mask ROM, which was manufactured in a clean room in a manner similar to the fabrication of integrated circuits. This meant that all of the bits of memory were hard-wired onto the device and could not be erased or rewritten after the fact. Mask ROM is therefore inefficient for small projects that may require debugging, or for embedded systems that require firmware updates or other types of memory changes.

EPROM is known as electrically programmable ROM. These chips generally contain a quartz window over the integrated circuit; exposure to UV light for an extended period of time erases the memory and sets all bits to a value of 1. An external programmer is used to electrically program all of the desired data to memory. EPROM has the advantage of being re-writable, but suffers from needing to be physically removed from a circuit to be erased and reprogrammed. In addition, the quartz window is expensive to make and EPROM chips can be more expensive than other types of memory. However, they were an improvement over one-time-programmable (PROM) chips, which, as the name implies, could only be programmed a single time and could not be erased and re-written.

Flash memory is used as program memory in the ATmega328P. It uses floating-gate transistors, one of which is shown in Figure 5.11 in "NOR" architecture. CG stands for control gate, and is where control signals are asserted. FG stands for floating gate, and is where electrons are either injected or removed to store data. S stands for source, which is generally the low potential (or ground) connection of a transistor. D is the drain, which is generally the high potential connection of a transistor.

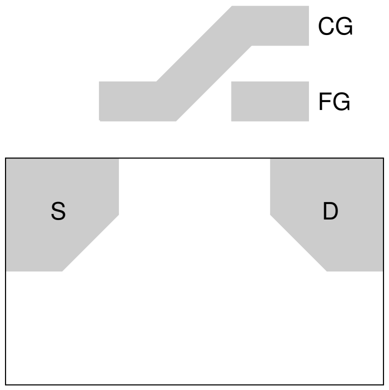


Figure 5.11: Schematic of a flash floating-gate cell.

The modes of operation of the flash floating-gate cell are detailed in Table 5.3.

Mode	Description
Clear	≈ 12 V applied to CG, channel on electrons travel from S to D in the process they tunnel through oxide to FG
Set	≈ -12 V applied to CG electrons tunnel from FG to CG and are removed
Read	5 V applied to CG if FG has electrons, a channel will not form (0) if FG has no electrons, a channel will form (1) from S to D

Table 5.3: Description of flash modes of operation.

Flash ROM is a subset of **EEPROM**, also known as electrically erasable programmable ROM. Today, there is little difference between EEPROM and flash architectures. However, the ATmega328P contains both flash and EEPROM storage for non-volatile memory. Flash is where program data and an optional bootloader are stored. The EEPROM on the ATmega328P is used for data storage that can be re-written but is not intended to change frequently. (For example, a passcode for a garage door opener can be reprogrammed, but mostly remains the same, and must be saved during power outages.) A fuse bit on the ATmega328P can be configured to allow the EEPROM storage to remain programmed in between chip erases, allowing additional functionality for the microcontroller.

5.9 Instruction Execution Process & Timing

This is the process by which the microcontroller executes each instruction. In this fashion, the CPU is nothing more than a very complicated finite state machine. The state diagram is shown in Figure 5.12.

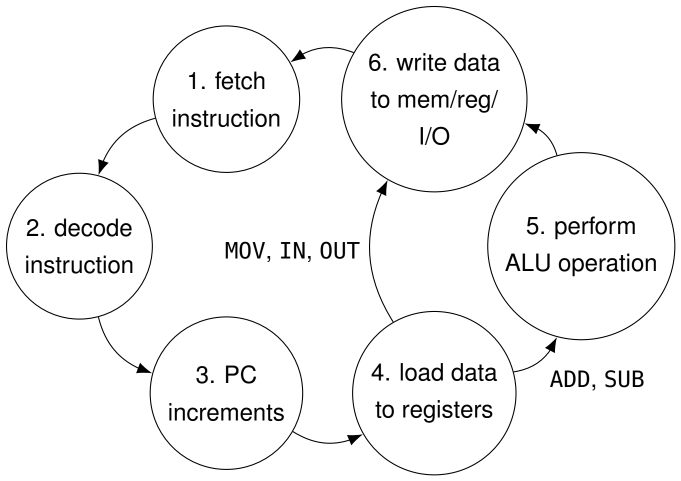


Figure 5.12: State diagram of microcontroller instruction execution process.

1. The next instruction in memory is read, indicated by the address stored in the program counter
2. The instruction is decoded into a set of commands or signals for each of the components in the processor
3. The program counter increments so that it points to the next location in memory
4. Data is loaded from memory (or input device(s)) into register(s), the location of this data is usually stored in the instruction code as an operand
5. If the ALU is required to execute the operation, the processor instructs the hardware to carry this out
6. The result is written back to a memory location, to a register, or even to an output device
7. Jump back to step 1

The ATmega328P uses a pipelining concept to speed up the rate at which the CPU can process the next instruction. This is enabled by its Harvard architecture, which allows the CPU to fetch the next instruction from memory while executing the current instruction. This is known as a parallel instruction fetch and is depicted in Figure 5.13.

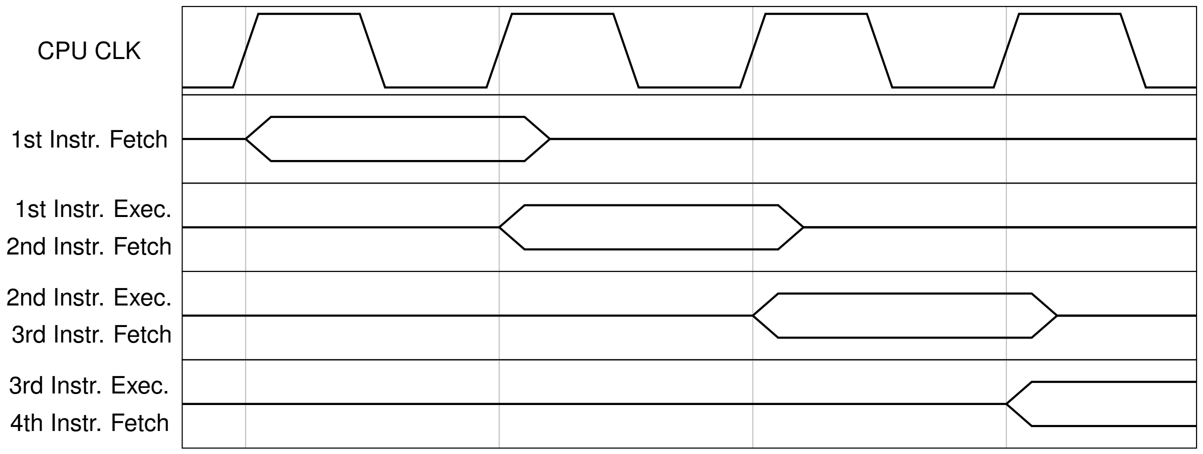


Figure 5.13: Timing diagram of parallel instruction fetch process.

Figure 5.14 shows the internal timing for the CPU, and shows how most of the instructions available on the ATmega328P are able to occur within a single clock cycle.

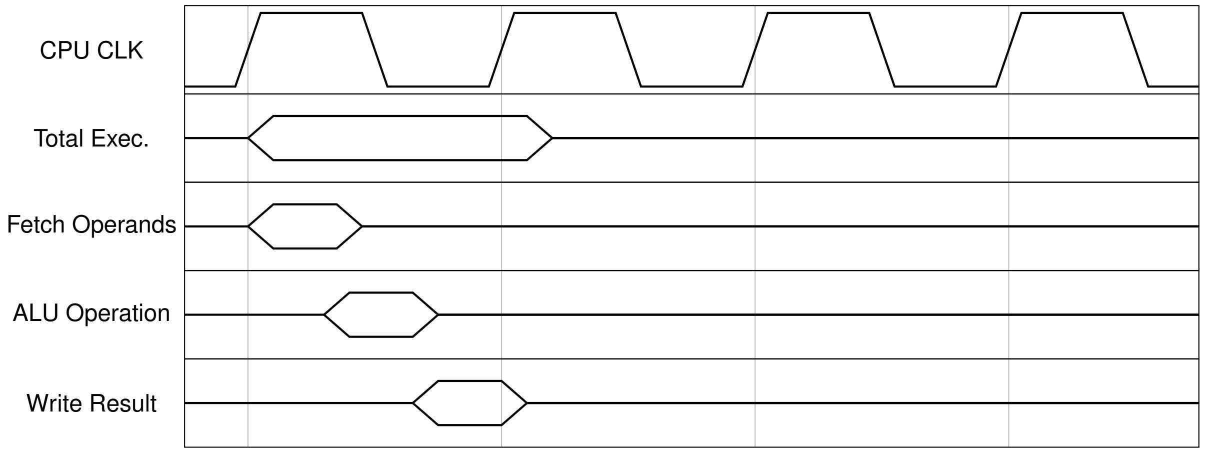


Figure 5.14: Timing diagram of single cycle ALU operation timing.

6
Status Register (`SREG`)

The status register on the ATmega328P contains information about the result of the most recently executed instruction on the microcontroller. The information in `SREG` can be used to control the flow of the program being executed by way of conditional branch instructions (which is discussed in chapter 19). All of the machine instructions that are capable of affecting the flags in `SREG` are listed in the AVR instruction manual [1]. The status register contains eight bits (known as flags) as shown in Table 6.1.

[1] Atmel, "AVR Instruction Set Manual," November 2016.

bit:	7	6	5	4	3	2	1	0
flag:	`I`	`T`	`H`	`S`	`V`	`N`	`Z`	`C`

Table 6.1: Contents of the status register (`SREG`).

6.1 `I` – Global Interrupt Enable Flag

This bit, as was discussed in chapter 13, the global interrupt enable flag is used to enable and disable interrupts. It must be set to 1 to globally enable interrupts, and it must be cleared to disable interrupts. This bit is automatically cleared by hardware on the ATmega328P when an interrupt service routine is invoked, and is subsequently set again when the execution of the ISR has been completed.

6.2 `T` – Bit Copy Storage Flag

There are instructions available in the AVR instruction set that use this bit as a source or destination for the operated bit. These two operations are `BST` (bit store from bit in register to `T` flag in `SREG`), and `BLD` (bit load from the `T` flag in `SREG` to a bit in register).

6.3 H – Half Carry Flag

The half carry flag is useful when dealing with binary coded decimal (BCD) numbers. The half carry flag is set if a carry was generated by the least significant 4 bits of the byte in the most recently executed instruction. In BCD arithmetic, this carry has a value of 16. The most significant bit (MSB) of 16 (the number 1) belongs in the 10's place of the BCD number. The remaining value of 6 needs to be added back in to the least significant nibble. An example of BCD addition with the half carry flag is shown in Table 6.2.

	1	1	1	**1**					
	0	0	1	1	1	0	0	1	$= 39_{10}$
+	0	1	0	0	1	0	0	0	$= 48_{10}$
	1	0	0	0	0	0	0	1	$= 81_{10}$ WRONG!
+	0	0	0	0	0	1	1	0	$= 6_{10}$
	1	0	0	0	0	1	1	1	$= 87_{10}$ Correct!

Table 6.2: BCD arithmetic, adding 39 and 48 yields an incorrect sum unless 6 is added again into the least significant nibble of the result.

An algorithm for "packed BCD addition" (when two BCD digits are represented in an 8-bit binary number) is provided in an application note by Microchip.[2]

[2] Atmel, "AVR204: BCD Arithmetics," January 2003.

6.4 N – Negative Flag

This bit is only generated with signed numbers. The negative flag is the MSB of the result of the most recently executed operation. A value of 0 indicates a positive number, and a value of 1 indicates a negative number.

6.5 V – 2's Complement Overflow Flag

This bit is only generated with signed numbers. When set, the most recently executed operation resulted in an overflow condition. When cleared, the most recently executed operation did not result in an overflow. There are different equations for its generation based on the particular instruction that is in use.

In **signed addition**, an overflow occurs if the addition of two negative numbers results in a positive number, or when the addition of two positive numbers results in a negative number. Overflow is not possible when adding a positive to a negative number.

In **signed subtraction**, an overflow occurs if a positive number minus a negative number results in a negative number, or when a negative number minus a positive number results in a positive number. Overflow is not possible when subtracting a positive number from a positive number, or when subtracting a negative number from a negative number.

In **signed multiplication**, an overflow occurs if not all of the overflow bits are equal.

6.6 S – Sign Flag

The sign flag is always an exclusive OR operation between the N and V flags,

$$S = N \oplus V.$$

It gives the true sign of the result of the most recent operation, as indicated in Table 6.3.

N	V	S	
0	0	0	sign is positive
0	1	1	sign is negative (overflow with false positive answer)
1	0	1	sign is negative
1	1	0	sign is positive (overflow with false negative answer)

Table 6.3: Information about the sign flag (S) flag based on the negative flag (N) and the 2's complement overflow flag (V).

6.7 Z – Zero Flag

This bit is set if the result of the most recently executed operation is zero.

6.8 C – Carry Flag

This bit is set if the most recently executed operation resulted in a carry or a borrow. This flag is particularly useful when adding numbers that require more than 8-bits to store. For example, when adding 16-bit numbers, the microcontroller must work 8-bits at a time due to the size limitation of the CPU registers. A carry on the least significant byte (carry flag of 1) indicates that one must be added to the most significant byte.

In unsigned addition and subtraction, the presence of a carry or borrow indicates overflow.

6.9 Practice Problems

1. Determine the values of the C, Z, N, V and S flags for the following.

   (a) 1010 1101 + 0101 0011

C=1, Z=1, N=0, V=0, S=0

(b) 0000 1010 − 0000 1111 C=1, Z=0, N=1, V=0, S=1

(c) 0101 0101 + 0101 0010 C=0, Z=0, N=1, V=1, S=0

7
Memory Addressing Modes

Memory maps of microcontrollers indicate what addresses are used for what type of memory. They are important to understand the functionality of the microcontroller memory space, and how the memory can be accessed by hardware.

7.1 Flash Program Memory

The ATmega328P contains 32 kBytes of non-volatile program flash memory. Because all of the instructions are 16-bits or 32-bits wide, data memory is formatted as 16 kBytes of 16-bit words. This means that 14 bits ($\log_2 16384$) are required to address the full program memory space on the ATmega328P. Indeed, the program counter on the ATmega328P is a 14-bit counter.

Program memory space is divided into two sections: an application section and a bootloader section. The size of the bootloader section is configured with the high fuse byte, as discussed in chapter 9. A map of the program memory on the ATmega328P is shown in Figure 7.1.

7.2 SRAM Data Memory

The volatile data memory on the ATmega328P is composed of 2 kBytes of SRAM. This data memory space, the memory map of which is given in Figure 7.1, is split into several sections.

32 General Purpose Registers

These 32 general purpose registers contain information to be entered into the ALU on the microcontroller. They are used to execute instructions. Registers R26 to R31 (known as `X`, `Y`, and `Z`) are indirect

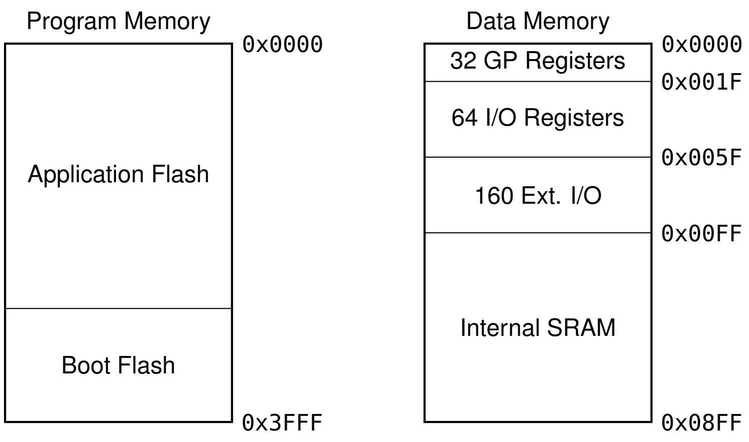


Figure 7.1: Memory map of the flash program memory and SRAM data memory on the ATmega328P.

addressing pointer registers. They are used in pairs to indirectly address program memory.

64 I/O Registers

In addition to the general purpose registers, data memory contains 64 I/O registers. This memory space contains the `PINx`, `PORTx`, and `DDRx` registers, as well as `SREG` and registers that control the operation of timer / counter 0, SPI communication, external interrupts, among others. These registers can be directly addressed using assembly instructions. Using assembly commands such as `LOAD`, data from the 64 I/O registers can be loaded into the general purpose registers to allow further operations to take place.

160 Extended I/O Registers

There are a lot more peripheral units than can be supported with 64 memory locations, therefore there are an additional 160 extended I/O registers that control all of the other peripheral features of the ATmega328P, including serial communication, interrupts, timer / counters, ADC, and more. These memory locations can only be accessed indirectly using load and store instructions in assembly (which takes more clock cycles than the direct addressing that can be used on the 64 I/O registers).

Internal SRAM

The remaining data space is used to store variable data in memory, as well as to house the stack.

7.3 *EEPROM Data Memory*

In addition to the above mentioned memory spaces, 1 kByte of EEPROM non-volatile data memory is available as a separate data space from SRAM. EEPROM is used to save data that can be changed but should remain stored in memory even in the absence of power to the device.

7.4 *Memory Addressing*

Memory addressing refers to the ways in which a microcontroller accesses data to carry out an instruction. The instruction contains operands that specify the data to be operated on, while the program counter provides the address for the next instruction in program memory. Memory addressing modes are determined when a microcontroller is designed, and cannot be changed by a programmer.

In a simple device such as an EPROM chip, memory addressing occurs directly by means of an internal decoder. A memory address is asserted on the address pins, and the internal circuitry asserts the contents of that address onto the output pins of the device. This is depicted for a 65 kB memory in Figure 7.2.

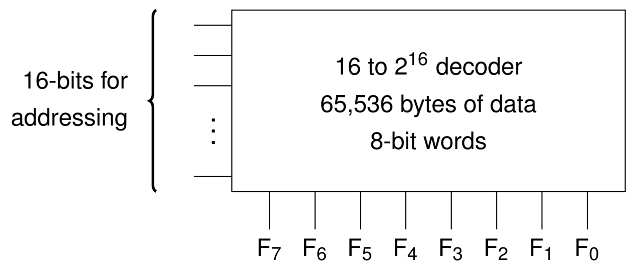


Figure 7.2: Memory addressing occurs in a simple system with a decoder. A 16 to 2^{16} decoder is capable of addressing 65,536 words of data.

In general, memory can be addressed **directly** or **indirectly**. Direct addressing occurs when the memory address is stored as part of the instruction itself. Direct addressing is typically fast (the data to be operated on in memory is pointed to directly), and has the flexibility of allowing variable data to be manipulated, rather than constant data (which is used in immediate addressing). Indirect addressing occurs when a pointer is used instead of the memory address. Indirect addressing enables a microcontroller to access memory contents even if the number of addresses available exceeds the size of the instruction itself, but suffers the drawback of being slower than direct addressing (first the pointer must be loaded, then the contents of the memory address that the pointer points to is operated on).

General Purpose Register Addressing

General purpose register addressing occurs when an assembly instruction contains the address of one or more general purpose register to be operated upon. Instructions with only a single register (`Rd`) operand address memory in a way known as **direct, single register addressing**, which is shown schematically in Figure 7.3.

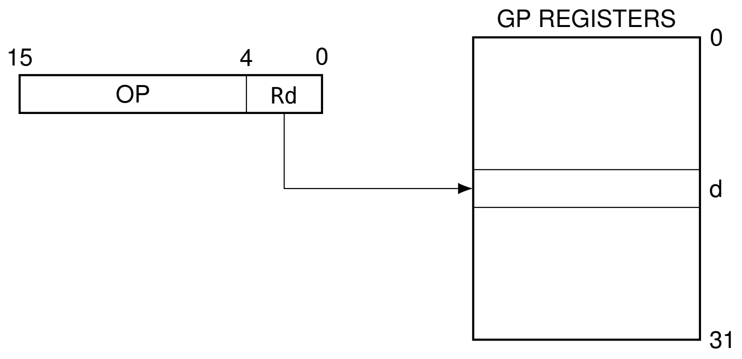


Figure 7.3: Direct, single register addressing occurs when an instruction contains a single general purpose register as an operand.

An instruction that directly addresses a single GP register is `NEG`, which takes the two's complement value of a register and saves the result back into that same register. It is capable of addressing all of the 32 GP registers using 5 bits in the instruction. The remaining 11 bits are used for the opcode.

Instructions with two general purpose register operands (`Rd` and `Rr`) address memory in a way known as **direct, two register addressing**, which is shown schematically in Figure 7.4. The result of these instructions is always stored in `Rd`, the destination register.

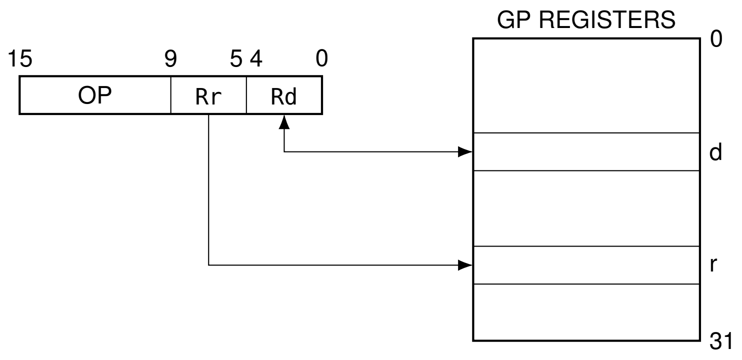


Figure 7.4: Direct, two register addressing occurs when an instruction contains two general purpose registers as operands.

An instruction that directly addresses a single GP register is `ADD`, which takes the value stored in a source register, adds it to the value stored in a destination register, and saves the result back into the destination register. It is capable of addressing all of the 32 GP registers, which means that 10 bits of the instruction are used for register addressing. The remaining 6 bits are used for the opcode.

Immediate Addressing

In immediate addressing, the instruction contains data to be operated on immediately. In other words, the data to be operated on is part of the instruction. Because the data is part of the instruction, immediate addressing can be fast. However, it is also less flexible when it comes to changing or repurposing code.

Most of the instructions that contain immediate addressing in the ATmega328P allow for the immediate data to take on values between 0–255. This requires 8 bits of data, which means that only 8 bits remain in the instruction for the opcode and the other operand. The number of GP registers that can be addressed is therefore limited to registers 16–31, which means that 4 bits are used to address the register and the remaining 4 bits are used as the opcode.

The `AND` and `ANDI` instructions demonstrate the difference between two-register addressing and immediate addressing. `AND` is a logical AND operation that occurs between two GP registers, `Rd` and `Rr`. The data is fetched from each of the registers, routed to the ALU, and then stored in the destination register. All of the GP registers can be addressed in this instruction, leaving 6 bits for the opcode. The machine instruction is:

```
0010 00rd dddd rrrr,
```

where `001000` is the opcode, `rrrrr` indicates the binary value of the source register, and `ddddd` indicates the binary value of the destination register.

`ANDI` is a logical AND with an immediate. This means that the microcontroller performs a logical AND between the contents of GP register `Rd` and a constant (the immediate), with the result stored in the destination register. 8 bits of the instruction are used for the immediate, 4 bits are used for the GP register, leaving 4 bits for the opcode. The machine instruction for `ANDI` is:

```
0111 KKKK dddd KKKK,
```

where `0111` is the opcode, `dddd` indicates the binary value of the destination register (to be added to 16; the instruction uses GP registers 16–31), and `KKKKKKKK` indicates the binary value of the immediate data.

Tying this together with GP register addressing, assembly code using AVR instructions can be written to load immediate data into GP registers, add the data together, and store it in a destination register. The assembly code with descriptions is given in Table 7.1.

`LDI r20, 230`	Load the number 230 into GP register 20
`LDI r22, 15`	Load the number 15 into GP register 22
`ADD r20, r22`	Add the contents of GP registers 20 and 22, store the result in register 20

Table 7.1: Assembly code that loads immediate data to GP registers, adds the contents together, and stores the result in a destination register.

I/O Register Addressing

I/O register addressing, shown schematically in Figure 7.5, is used when the instruction contains the address of the I/O register to be operated on. As there are 64 I/O registers, each instruction requires 6 bits to indicate the memory location. Therefore, this mode cannot be used to access extended I/O memory.

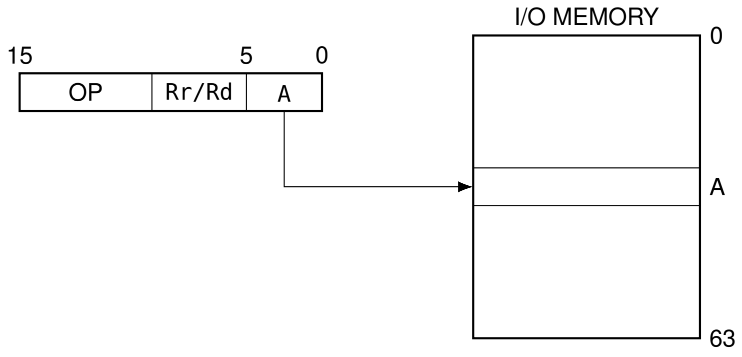


Figure 7.5: I/O register addressing occurs when an instruction contains an I/O registers as an operand.

In **I/O direct addressing**, instructions contain an I/O address (`A`) as well as a destination general purpose register (`Rd`) and/or a source general purpose register (`Rr`).

Notable instructions that use direct I/O addressing are `IN` and `OUT`, which either loads data from an I/O register into a destination GP register or stores the contents from a source register into one of the I/O registers, respectively. 6 bits are used to address the I/O register, 5 bits are used to address the (source or destination) GP register, and the remaining 5 bits are used for the opcode.

Assembly code can be written to load data from `PIN` registers into GP registers, perform a subtraction, and store the result in a destination register. The assembly code with descriptions is given in Table 7.2.

`IN r5, PIND`	Load the data from port D into GP register 5
`IN r6, PINB`	Load the data from port B into GP register 6
`SUB r5, r6`	Subtracts the contents of GP register 6 from the contents of register 5 and stores the result in register 5

Table 7.2: Assembly code that loads `PIN` data to GP registers, subtracts the contents, and stores the result in a destination register.

Data Memory Addressing

Data memory addressing is used to access the data memory (SRAM). These instructions contain two 16-bit words; the least significant word contains the 16-bit data address. **Data direct addressing**, shown schematically in Figure 7.6, contains destination and/or source general purpose registers (`Rd` and/or `Rr`). Because there are two instruction words to decode, these instructions require at least 2 clock cycles to execute.

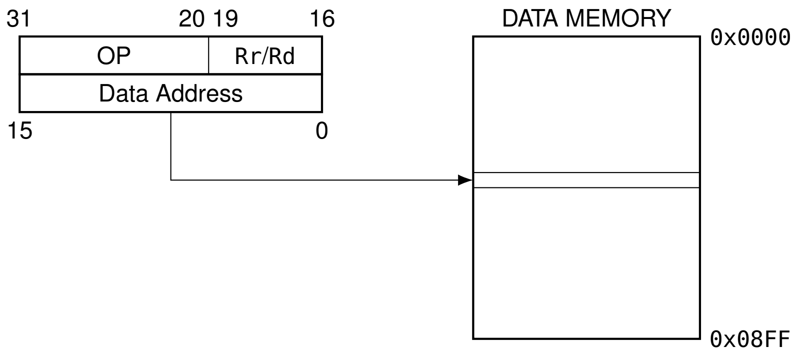


Figure 7.6: Data direct addressing occurs when an instruction contains a 16-bit data address as an operand.

An instruction that uses direct data memory addressing is `LDS`, which loads data from a given address in SRAM into a destination GP register. 16 bits are available for the data memory address, which means that it is limited to addressing the first 64 KB of data stored in SRAM. This is not a problem on the ATmega328P which only has 2 KB of memory. The remaining part of the instruction contains 5 bits to address the GP register, with the remaining 11 bits used for the opcode. The machine instruction for `LDS` is:

```
1001 000d dddd 0000
kkkk kkkk kkkk kkkk,
```

where `10010000000` is the opcode, `ddddd` indicates the binary value of the destination register, and `kkkkkkkkkkkkkkkk` indicates the address in SRAM to be accessed.

Data indirect addressing occurs when the operand address is the contents of the `X`, `Y` or `Z` pointer register. This mode, shown schematically in Figure 7.7, is used to access extended I/O registers or the internal SRAM. This type of addressing is also used when the address of a desired memory location is not known at the time that a program is written. Just as there are pros and cons to using direct and indirect addressing for the GP registers, there are pros and cons to using direct and indirect addressing to access data memory.

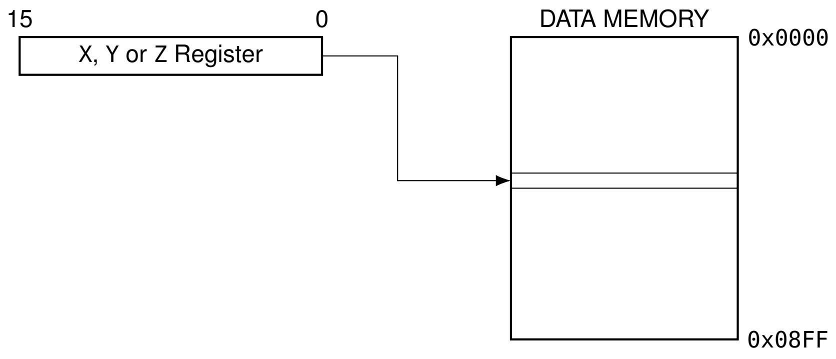


Figure 7.7: Data indirect addressing occurs when an operand address is the contents of an indirect addressing pointer register.

Indirect data memory addressing can also occur with any one of the following modifications:

- displacement – add a constant value to the address in pointer register `Y` or `Z`,
- post-increment – add one to the value in pointer register `Y` or `Z` after the operation, and
- pre-decrement – subtract one from the value in pointer register `Y` or `Z` before the operation.

Data indirect with displacement mode, shown in Figure 7.8 is used when the operand address is the result of the contents of `Y` or `Z` (indirect addressing pointer registers) added to the address contained in 6-bits (`q`) of the instruction word. This mode is useful for writing position independent code, since an address register can be used to stored a base address, with data access using displacements relative to the base.

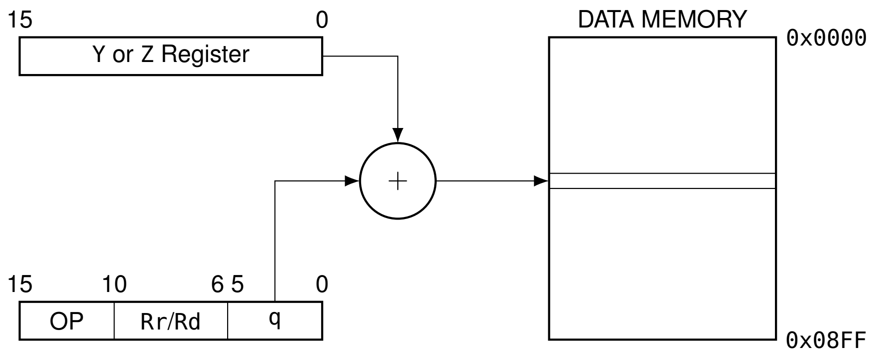


Figure 7.8: Data indirect addressing with displacement occurs when an operand address is the result of the contents of an indirect addressing pointer register added to 6 bits (`q`) contained in the instruction word.

As an example, say that the contents of an array are stored in a known position in SRAM. The address of element 0 (150 in the example in Table 7.3) can be stored in either pointer register `Y` or `Z`. If element 15 needs to be accessed, the displacement offset would be equal to 15. This assembly code is given in Table 7.3. Note that data indirect with displacement is not available with pointer register `X`.

`CLR r29`	Clear the HIGH byte of pointer register `Y`
`LDI r28, 150`	Load the value 150 into the LOW byte of pointer register `Y`
`LDD r4, Y+15`	Loads the value stored in address 165 (`Y` + 15) into GP register 4

Table 7.3: Assembly code that loads indirect from pointer register `Y` with displacement.

Data indirect with pre-decrement mode is used when the operand address is the contents of the `X`, `Y` or `Z` register, which is automatically decremented before the operation. This addressing mode is similarly useful for accessing array contents, for example. It is depicted schematically in Figure 7.9.

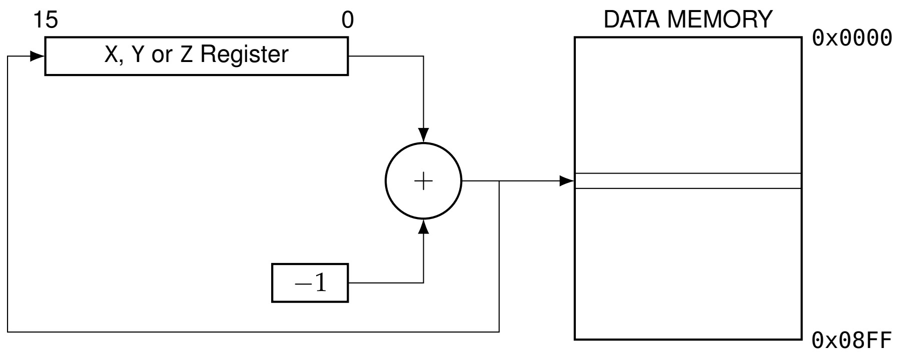


Figure 7.9: Data indirect with pre-decrement addressing occurs when an operand address is the contents of an indirect addressing pointer register, which is decremented prior to the instruction.

Data indirect with post-increment addressing is used when the operand address is the contents of the `X`, `Y` or `Z` register, which is automatically incremented after the operation. This addressing mode is also useful for accessing array contents. It is depicted schematically in Figure 7.10.

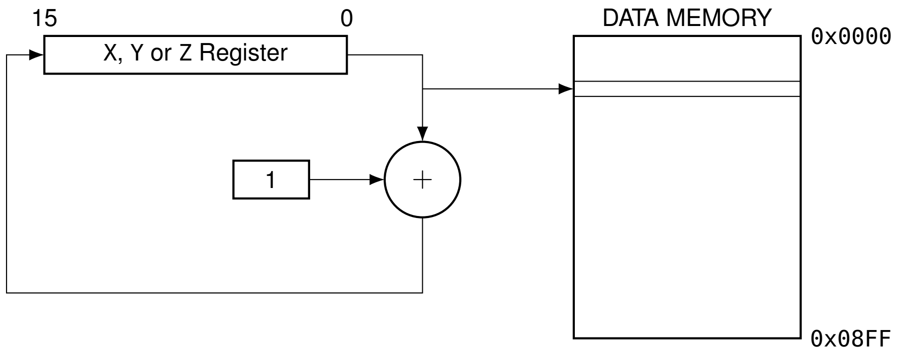


Figure 7.10: Data indirect with post-increment addressing occurs when an operand address is the contents of an indirect addressing pointer register, which is incremented after the instruction.

Program Memory Addressing

Program memory addressing is used to access the full program memory space. It requires at least 14 bits to address the total flash memory section. The `Z` pointer register is used to refer to the address in program memory. Because program memory stores 16-bit words, and the destination register can only store 8 bits of data, the `Z` register additionally specifies if the LOW byte (accessed if Z_{LSB} is 0) or HIGH byte (accessed if Z_{LSB} is 1) should be returned. Program memory addressing can also be done with a post-increment or with a pre-decrement. Program memory addressing is shown schematically in Figure 7.11.

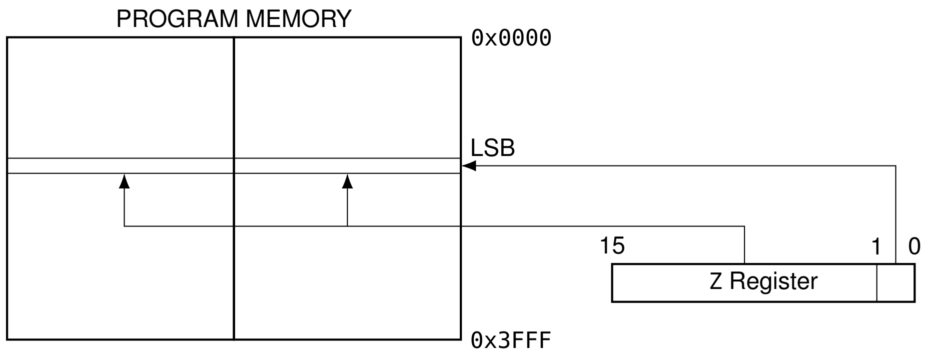


Figure 7.11: Program memory addressing accesses the full memory space given by the address in pointer register `Z`. The LSB of `Z` indicates if the low or high byte should be returned as a result.

In **direct program addressing**, instructions can contain a pointer (indirect) to be loaded to the program counter, or an 16-bit word (immediate) following the instruction to load to the program counter. The program counter can also be incremented by a constant `k`. Program execution will then continue at that space in program memory.

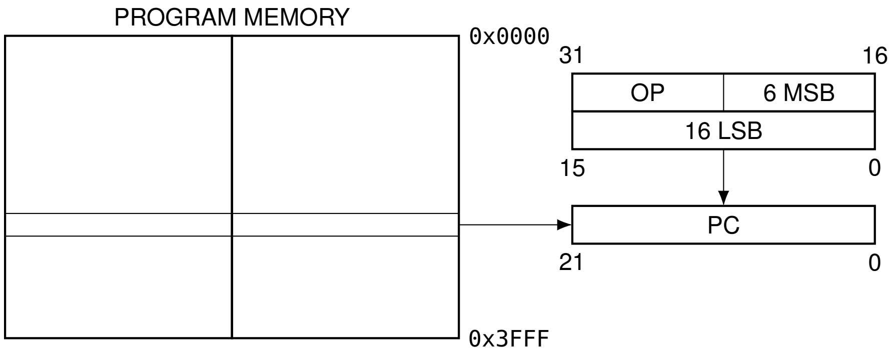


Figure 7.12: Direct program addressing accesses the full memory space given by a pointer loaded into the program counter.

The types of instructions that use direct programming addressing include `JMP`, which causes the program counter to load an immediate value and continue executing instructions from that location in memory and `EIJUMP`, which has the program counter jump to the memory location stored in pointer register `Z`.

7.5 *Bit Addressing*

Bit addressing is used to access specific bits in registers (especially `SREG`). Because most registers are 8 bits, the instruction requires 3 bits (`b`) to specify which bit (from 0–7) is to be read from/written to.

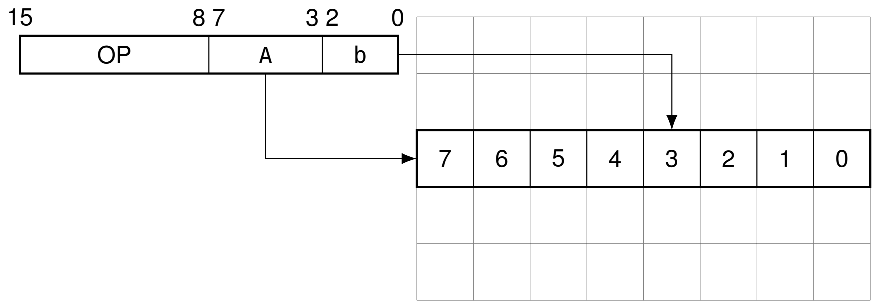


Figure 7.13: Bit addressing is used to access specific bits in registers.

The instruction `CLI`, for example, is used to clear the global interrupt flag bit in `SREG`, which prevents interrupts from executing on the microcontroller. The global interrupt flag bit can be set again using the `SEI` instruction. These instructions do not require any operands, and thus use 16-bit opcodes.

7.6 *The Stack & Stack Operations*

The stack is a special area of RAM reserved for temporary data storage. It cannot overlap with the general purpose, I/O, or extended I/O memory registers. Therefore, the memory address at which it starts must be greater than or equal to `0x0100`. Usually, the stack is initialized at the highest available address in RAM. The current location of the most recent data byte in the stack is known as the stack pointer (`SP`). The stack pointer consists of two 8-bit registers that exist in the I/O space.

The stack itself can be visualized as a stack of plates in a cafeteria; new plates can be added to the stack, and when plates are accessed, the last one in is always the first one out (last-in, first-out).

Data can be added to the stack using the `PUSH` instruction. When this occurs, new information is placed into temporary storage, and the value of `SP` is decremented. Removing data from the stack occurs using the `POP` instruction. The value of `SP` is incremented during this instruction. The contents of the stack, as well as the locatio of the stack pointer, is depicted schematically in Figure 7.14 after both a `PUSH` and `POP` operation.

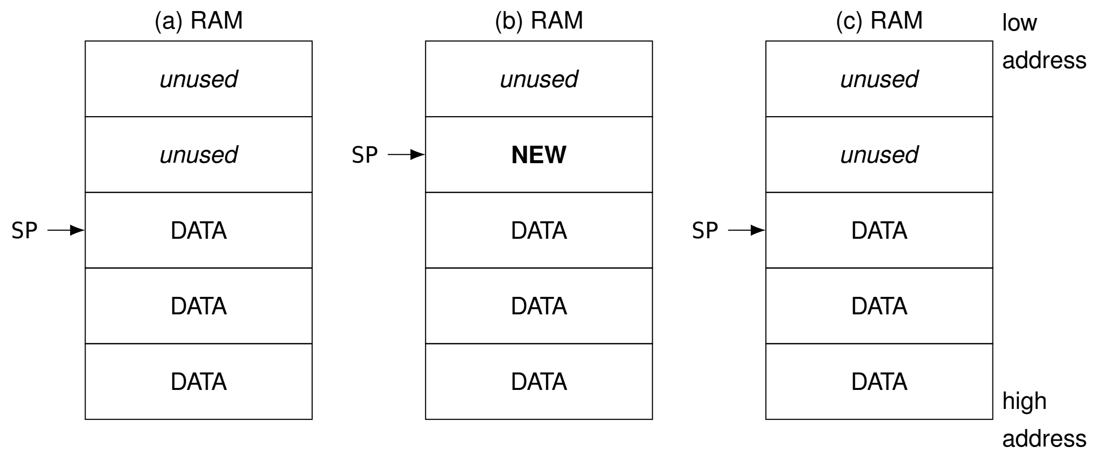


Figure 7.14: Stack operations. (a) Memory in the stack, (b) after a `PUSH` instruction, and (c) after a `POP` instruction.

Use of the stack is essential during a subroutine call (external function or an ISR). In order to return to the exact spot in memory where the program left to execute the subroutine, the return address is `PUSH`ed to the top of the stack before the location of the subroutine is accessed. After the subroutine is complete, the program goes back to the location and the return address is `POP`ed out of the stack.

Stack Problems

Care must be taken when allocating data memory to be used in the stack. **Stack overflow** is a situation in which too much data is pushed onto the stack which causes the `SP` to point to an address outside of the stack memory area. **Stack underflow** is a situation in which too much data is popped from the stack so that the `SP` points to an address below the stack bottom. A **stack collision** occurs when the stack is allocated to memory addresses which overlap with data registers.

7.7 *Practice Problems*

1. How many bytes in memory can be accessed if 6 bits are used in the instruction for the memory address? 64

2. In indirect addressing, a 16-bit pointer register is used to address memory. How many bytes can be accessed in this scenario? 64 K

3. Use the table below and `Y = 0x0103` to determine the value stored in the destination register, as well as the value of pointer register `Y`, after each of the following subsequent operations.

Address	SRAM Contents
`0x0100`	`0xC1`
`0x0101`	`0xFB`
`0x0102`	`0x3F`
`0x0103`	`0xE9`
`0x0104`	`0x95`
`0x0105`	`0x9A`
`0x0106`	`0x1C`
`0x0107`	`0xDC`

(a) `LD r4, Y` `r4 = 0xE9, Y = 0x0103`

(b) `LD r5, Y+` `r5 = 0xE9, Y = 0x0104`

(c) `LD r6, Y` `r6 = 0x95, Y = 0x0104`

(d) `LD r7, Y+3` `r7 = 0xDC, Y = 0x0107`

(e) `LD r8, Y-` `r8 = 0x1C, Y = 0x0106`

(f) `LDS r9, 0x0102` `r9 = 0x3F, Y = 0x0106`

8
Model Microcontroller

To better understand how a microcontroller works, a simple model microcontroller will be designed. It will have an ALU to perform all required operations, two data registers, four input devices, and four output devices. Each of the registers, input devices, and output devices has its own numerical label to determine with which component each instruction should interact. This model microcontroller is shown in Figure 8.1.

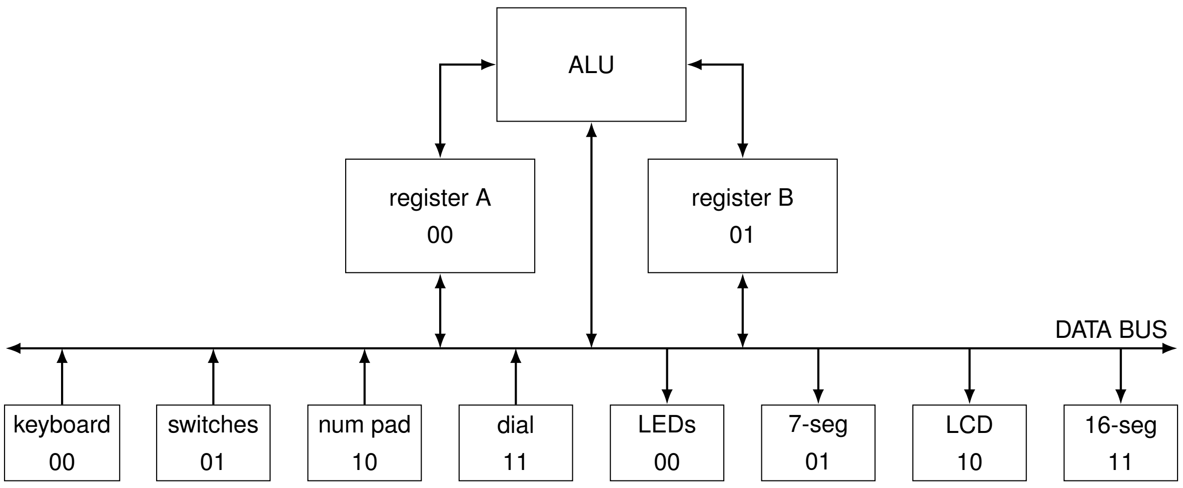


Figure 8.1: A block diagram of the model microcontroller with input and output devices, two registers, and an ALU.

8.1 Microcontroller Instructions

Microcontrollers are capable of carrying out specific machine instructions. An instruction consists of an operation code (opcode), which dictates what function to do (add, subtract, XOR, etc.). The ALU contains specialized hardware to accomplish each of these functions. Each source of data used to carry out the function is known as an operand. A full listing of the ATmega328P instructions is available in the AVR instruction manual.

8.2 *Microcontroller Operation Codes (Opcodes)*

Recall that for every n opcodes, there must be a $\lceil \log_2(n) \rceil$ bit number to store it, assuming that expanding opcodes are not used. If there are 40 operations, 6 bits of the instruction code are necessary to store the opcode. The model microcontroller will have 7 operations. Therefore, 3 bits are necessary for storing this information. However 4 bits will be used to create 8-bit instructions.

It is now possible to come up with a list of instructions that the model microcontroller can perform, and indicate the operands that are required for each. These instructions are given in Table 8.1.

Instruction	**Opcode**	**Description**	**Operands**
`IN`	`0000`	Load data from input	I/O(A), `Rd`
`OUT`	`0001`	Store data to output	I/O(A), `Rr`
`MOV`	`0010`	Copy data	`Rr`, `Rd`
`ADD`	`0011`	Add data	`Rr`, `Rd`
`SUB`	`0100`	Subtract data	`Rr`, `Rd`
`AND`	`0101`	Logical AND	`Rr`, `Rd`
`OR`	`0110`	Logical OR	`Rr`, `Rd`

Table 8.1: Listing and description of the instructions available from a model microcontroller.

`IN` – Load data from input to register

The `IN` instruction loads data from an input device to one of the registers. In order to carry out this instruction, the designation of the input device (A) as well as the designation of the destination register (Rd) need to be known.

`OUT` – Store data from register to output

The `OUT` instruction takes data from a source register and sends it to one of the output devices. In order to carry out this instruction, the designation of the output device (A) as well as the designation of the source register (Rr) need to be known.

`MOV` – Copy data from one register to another

The `MOV` instruction takes data from a source register and copies it into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

`ADD` – *Add data from register* `Rr` *to* `Rd` *and store in* `Rd`

The `ADD` instruction takes data from two registers, adds the contents, and stores the sum into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

`SUB` – *Subtract data in register* `Rr` *from* `Rd` *and store in* `Rd`

The `SUB` instruction takes data from a source register, subtracts it from the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

`AND` – *Logical AND data in register* `Rr` *with* `Rd` *and store in* `Rd`

The `AND` instruction takes data from a source register, performs a logical AND operation with the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

`OR` – *Logical OR data in register* `Rr` *with* `Rd` *and store in* `Rd`

The `OR` instruction takes data from a source register, performs a logical OR operation with the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

Each of the model microcontroller instructions can be expressed in machine code (binary), assembly (using an opcode and operands), or using register-transfer language (depicting the movement of data from one register to another). This is outlined in Table 8.2.

Table 8.2: Model microcontroller instructions in machine code, register transfer language, and assembly.

Machine Instruction	**Assembly**	**Register Transfer**
`0000 AAdd`	`IN Rd,A`	`Rd` ← I/O (A)
`0001 rrAA`	`OUT A,Rr`	I/O (A) ← `Rr`
`0010 rrdd`	`MOV Rd,Rr`	`Rd` ← `Rr`
`0011 rrdd`	`ADD Rd,Rr`	`Rd` ← `Rd + Rr`
`0100 rrdd`	`SUB Rd,Rr`	`Rd` ← `Rd − Rr`

8.3 *Model Microcontroller Program*

Now a set of instructions that the model microcontroller can understand can be created. A sample set of instructions is provided in Table 8.3. The program will be written in machine code, assembly language, and register transfer language.

Table 8.3: Program instructions for the model microcontroller.

Step	Instruction
1	Input data from the switches into register A
2	Input data from the dial into register B
3	Output the contents of register A to the LCD screen
4	Output the contents of register B to the LEDs
5	Add the contents of both registers, and store the result in register A
6	Output the contents of register A to the 16-seg. display

Machine Code

The binary numbers in the machine code are what's actually saved to the microcontroller memory. Using hexadecimal makes the list of instructions somewhat less unwieldy. Based on the sample program defined in Table 8.3, the corresponding machine instructions are given in Table 8.4.

Table 8.4: Program instructions for the model microcontroller in machine code given in both binary and hexadecimal.

Step	Machine Code	Hex
1	`0000 0100`	`0x04`
2	`0000 1101`	`0x0D`
3	`0001 0010`	`0x12`
4	`0001 0100`	`0x14`
5	`0011 0100`	`0x34`
6	`0001 0011`	`0x13`

Assembly Language & Register Transfer Language

Assembly language allows instructions to be invoked while having a somewhat easier time reading, understanding, and debugging the program contents. Register transfer language is another abstraction that allows for the visualization of movement of data from one location to another, along the location of the arrow. Based on the sample program defined in Table 8.3, the corresponding machine instructions are given in Table 8.5.

Step	Assembly	Register Transfer Language
1	`IN A,switches`	A ← I/O (switches)
2	`IN B,dial`	B ← I/O (dial)
3	`OUT LCD,A`	I/O (LCD) ← A
4	`OUT LEDs,B`	I/O (LEDs) ← B
5	`ADD A,B`	A ← A+B
6	`OUT 16seg,A`	I/O (16seg) ← A

Table 8.5: Program instructions for the model microcontroller in assembly language and register transfer language.

Model Microcontroller Arithmetic & Logic Unit (ALU)

The ALU of the model microcontroller is depicted schematically in Figure 8.2. This schematic diagram shows all of the internal hardware that is required to carry out each operation. The intricacies of the instruction decoder have been omitted for simplicities sake.

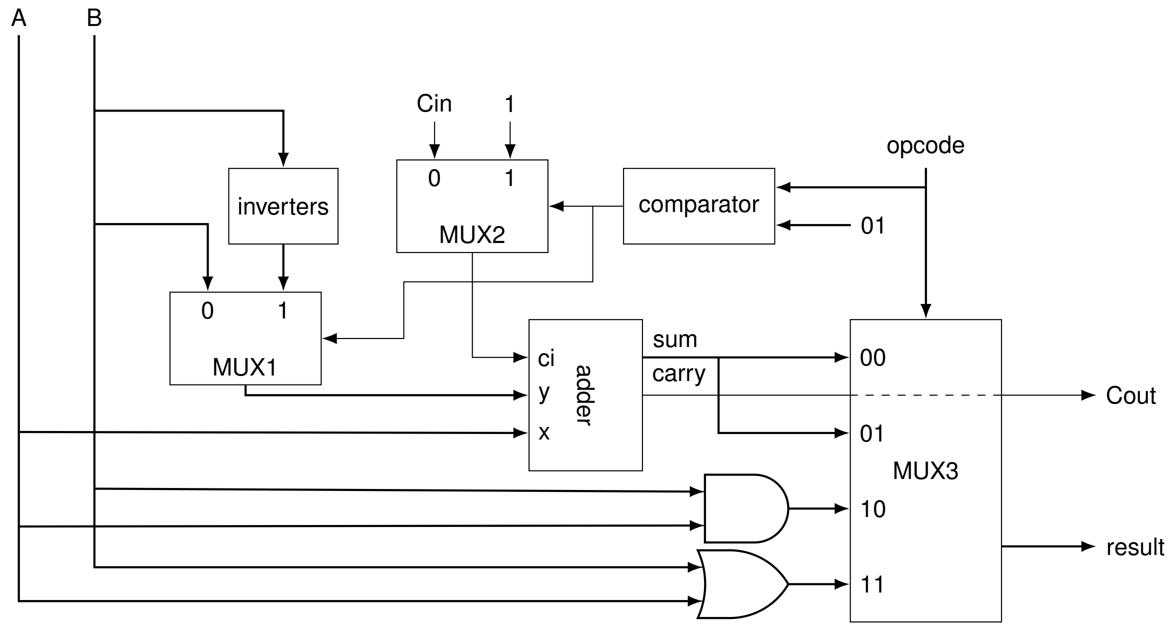


Figure 8.2: Hardware schematic of a simple arithmetic and logic unit (ALU).

Table 8.6 describes the operation of the hardware of each of these instructions.

Table 8.6: Listing and description of the four opcodes available on the simple arithmetic and logic unit.

Opcode 00: Add
Opcode 00 is sent to the comparator, which outputs a value of 0
MUX2 output: Cin, which enters the adder
MUX1 output: B, which enters the adder

A also enters the adder MUX3 output: sum The carry is sent as Cout
Opcode 01: Subtract
Opcode 01 is sent to the comparator, which outputs a value of 1 MUX2 output: 1, which enters the adder MUX1 output: B′, which enters the adder (this creates a 2's complement value of B) A also enters the adder MUX3 output: (A + (−B)) The carry is sent as Cout
Opcode 10: AND
Opcode 10 is sent to the comparator, which outputs a value of 0 A and B both go to the AND gate MUX3 output: AB
Opcode 11: OR
Opcode 11 is sent to the comparator, which outputs a value of 0 A and B both go to the OR gate MUX3 output: A+B

8.4 *Practice Problems*

Using the available instructions for the model microcontroller, write assembly commands to execute the following tasks.

1. Load data from the keyboard into register A.

```
IN A,keyboard
```

2. Load data from the number pad into register B.

```
IN B,numpad
```

3. Subtract the contents of register A from register B and store the result in register B.

```
SUB B,A
```

4. Move the contents of register B to register A.

```
MOV A,B
```

5. Display the contents of register A on the 7-segment display.

```
OUT 7seg,A
```

9
The ATmega328P Microcontroller

The ATmega328P microcontroller is featured on the Arduino Uno board and contains a number of peripheral functions that will be described in detail in this book.

9.1 Pinout Diagrams

The ATmega328P is available in four different packages. The 28-pin plastic dual in-line package (PDIP) is used in breadboarded designs. The pinout diagram of the ATmega328P's PDIP integrated circuit package is shown in Figure 9.1.

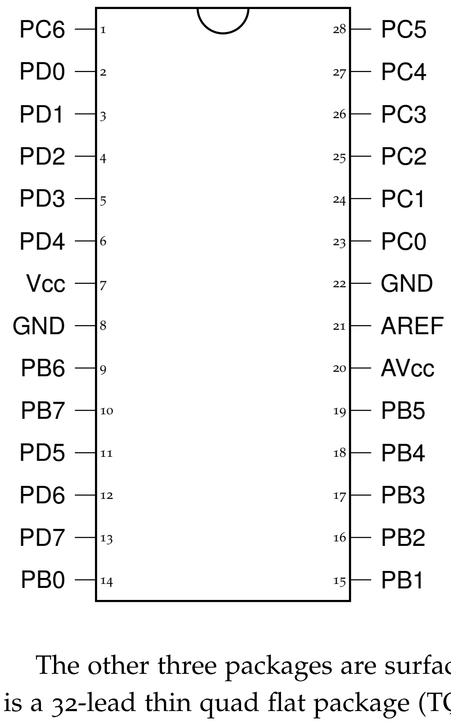


Figure 9.1: Pinout diagram for 28-pin PDIP package.

The other three packages are surface mount designs. First, there is a 32-lead thin quad flat package (TQFP). The close spacing of pins allows for more connections to be made (relative to the DIP architecture) in the same (or less) area of space. This chip has four more pins than the PDIP. Two of the pins are extra ground connections. The other two include connections to additional analog to digital conversion channels. However, this type of pin configuration can be much more complicated for use in hobbyist projects, especially without knowledge of surface mount soldering techniques. The pinout diagram for the TQFP integrated circuit chip is shown in Figure 9.2.

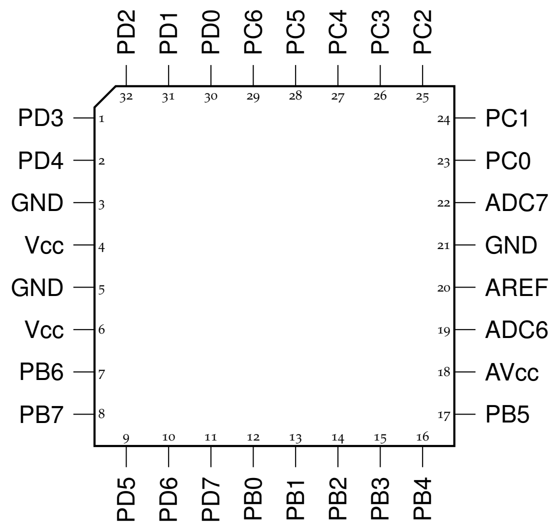


Figure 9.2: Pinout diagram for 32-lead TQFP package.

The second surface mount package is a 28-pad quad flat no-leads package (QFN) which, as the name implies, has no leads protruding from the integrated circuit chip. As with the 28-pin PDIP format, this lacks the two extra analog to digital conversion channels that are available in the 32-pin chips. The pinout diagram for the 28-pad QFN integrated circuit chip is shown in Figure 9.3.

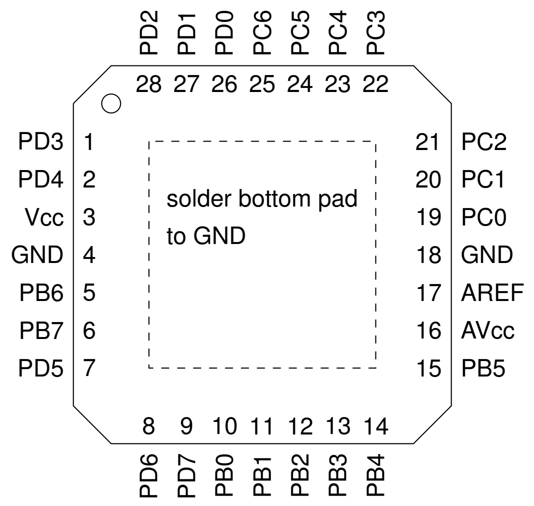


Figure 9.3: Pinout diagram for 28-pad QFN package.

The final surface mount format is a 32-pad QFN package. This chip also contains no physical leads, and, as a 32-pad device, contains the two extra analog to digital converter channels. This format's pinout diagram is given in Figure 9.4.

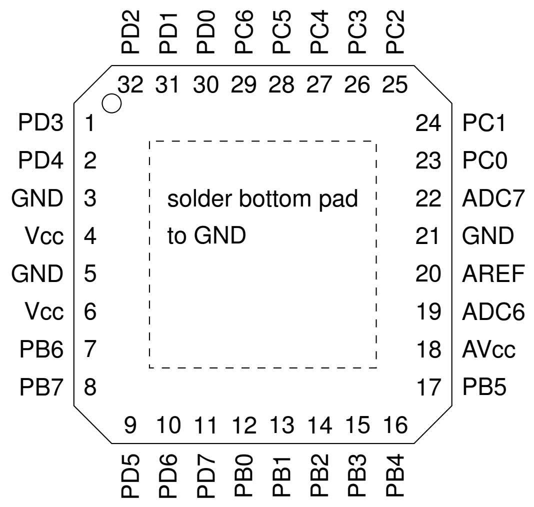


Figure 9.4: Pinout diagram for 32-pad QFN package.

As mentioned in chapter 10, each of these pins serves multiple purposes. This allows a pin to act either as an input/output interface or to connect an external crystal oscillator, or to provide a pulsed waveform, for example. Importantly, the Vcc pin must be connected to a proper operating voltage, which is between 2.7 V and 5.5 V. Pin PC6 can either be used as an active LOW reset pin or as a general purpose I/O pin, depending on the fuse settings.

9.2 *Writing Programs to Memory*

By connecting the Arduino Uno to a computer with a USB cable and using the Arduino IDE, programs and data can be written to the ATmega328P memory using a serial communication protocol called USART. A section of program memory known as the **bootloader** instructs the microcontroller to receive external information and write it to program memory. The Arduino Uno can also be programmed using another serial protocol known as SPI. Otherwise, a conventional non-volatile memory programmer can be used to program the microcontroller.

A non-volatile memory programmer can be used to program the ATmega328P microcontroller chip external to the Arduino Uno. While the Arduino package has many features and is convenient to use, it is useful to be able to program individual microcontroller chips for small-size, low-power, or low-cost situations.

9.3 *Fuse Bytes*

Although the microcontroller contains flash memory, EEPROM, SRAM, and general purpose registers, it also contains non-volatile fuse bytes to affect the functioning of the microcontroller external to the program operations. There are three fuse bytes used to control the microcontroller: the extended fuse byte, the low fuse byte, and the high fuse byte. Programmed fuses have a value of 0, while unprogrammed fuses have a value of 1.

Extended Fuse Byte

This fuse configures the brown-out detection unit. If a power supply fluctuates, it is possible for the microcontroller to malfunction if the voltage supplied to the Vcc pin dips below a certain value. The brown-out detection unit resets the microcontroller while the voltage is underneath a given threshold as programmed by the extended fuse byte.

High Fuse Byte

This fuse configures features that are relevant to programming and debugging the ATmega328P microcontroller. Among other things, the bits in the high fuse byte control the ability to use serial programming, to use the watchdog timer (which is explained in chapter 14), to preserve EEPROM memory when writing new program memory to the device, and to configure the amount of memory set aside to act as a bootloader.

Low Fuse Byte

This fuse configures features that are relevant to the clock source used to operate the microcontroller. There are two clock sources internal to the ATmega328P, but the device may also be clocked from an external source. The start-up time is also affected by changing the low fuse byte. Clock sources require a sufficiently high value of Vcc before it starts oscillating, and it must oscillate a sufficient number of times before the clock can be considered stable. A time-out delay is used after device reset to ensure a sufficient value of Vcc.

9.4 *Practice Problems*

1. The reset pin is active-______.

   LOW

2. What is the operating voltage range of the ATmega328P?

   2.7–5.5 V

3. How many fuse bytes are available on the ATmega328P?

   3

4. What are the names of the fuse bytes?

   extended, high, low

5. True or false: Upon power-up, the CPU starts working immediately.

   FALSE

6. What is the purpose of brown-out detection?

   To protect the CPU from malfunction if the power supply dips below a given voltage level.

10
I/O Port Registers

COMPUTERS ARE ONLY INTERESTING insofar as they can interact with input and output devices. The gatekeeper between a microcontroller and I/O devices are I/O pins. These I/O pins are bidirectional. Through associated I/O registers, the pins can be configured as input pins (which allows the microcontroller to read their corresponding logic level) or output pins (which allows the microcontroller to send data to them). Figure 10.1 shows a schematic of how this works.

The ATmega328P has three I/O ports which connect to pins and other peripheral features. These ports are `PORTB`, `PORTC`, and `PORTD`. These ports, and their associated Arduino pins, are outlined in Table 10.1 Ports B and D are gateways to the digital pins on the ATmega328P microcontroller. Port C gives access to the analog pins on the ATmega328P. The analog pins can be used digitally; they are called analog pins because they connect to analog to digital converters, which is discussed in chapter 11.

`PORTB`								
bit:	7	6	5	4	3	2	1	0
pin:	–	–	D13	D12	D11	D10	D9	D8
`PORTC`								
bit:	7	6	5	4	3	2	1	0
pin:	–	–	A5	A4	A3	A2	A1	A0
`PORTD`								
bit:	7	6	5	4	3	2	1	0
pin:	D7	D6	D5	D4	D3	D2	D1	D0

Table 10.1: ATmega328P data ports and their corresponding Arduino pins.

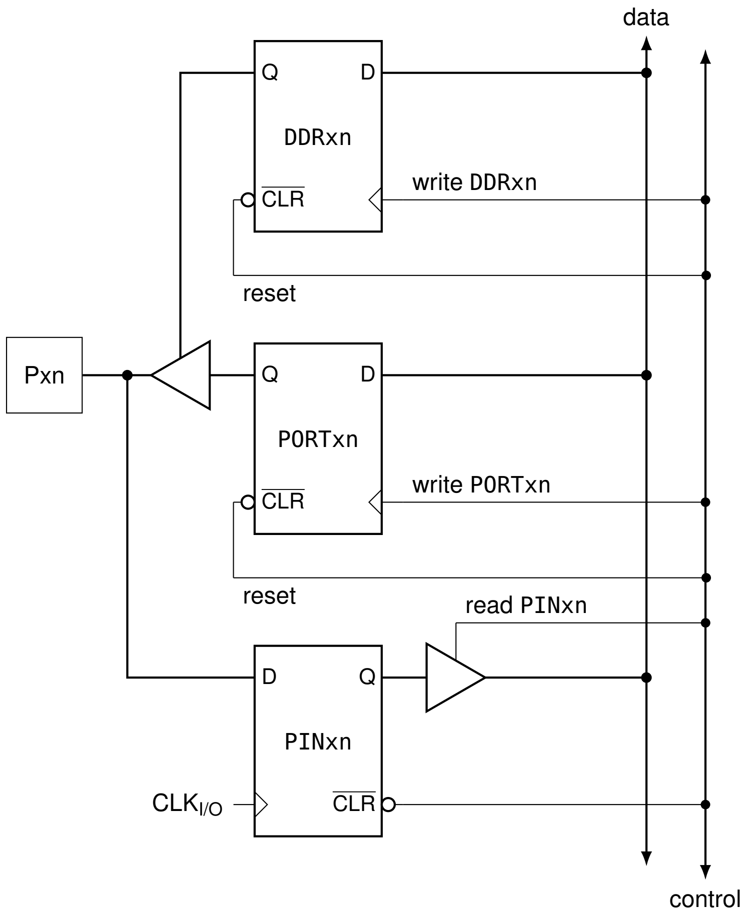


Figure 10.1: Hardware schematic of I/O port registers: DDRxn controls the data direction, PORTxn controls the data sent to output pins, and PINxn contains the data state of input pins.

10.1 Electrical Characteristics

It is important to understand the electrical characteristics of I/O pins in order to interface external devices with the microcontroller properly. The two characteristics of primary concern are **voltage-level compatibility** and **current-level compatibility**.

Voltage-Level Compatibility

To ensure that the voltage levels of external devices are able to properly interface with the ATmega328P, the voltages required for a logic HIGH and logic LOW must be known for both devices. (The values for the ATmega328P can be found in the Common DC Characteristics section of the datasheet.) The following four voltages must be compared and checked for compatibility:

- **Input HIGH voltage** (V_{IH}) – This is the voltage level that is treated as a logic 1 when applied to the input of a device.

- **Input LOW voltage** (V_{IL}) – This is the voltage level that is treated as a logic 0 when applied to the input of a device.
- **Output HIGH voltage** (V_{OH}) – This is the output voltage level when a digital circuit outputs a logic 1. The output HIGH voltage of circuit X must be greater than or equal to the input HIGH voltage of circuit Y, in order for circuit X to properly drive circuit Y.
- **Output LOW voltage** (V_{OL}) – This is the output voltage level when a digital circuit outputs a logic 0. The output LOW voltage of circuit X must be less than or equal to the input LOW voltage of circuit Y, in order for circuit X to properly drive circuit Y.

Current-Level Compatibility

In order for the ATmega328P to properly drive external devices, the current drive capability must be understood. The ATmega328P is capable of supplying current when the output voltage is HIGH (current flows from the microcontroller to the device), or sinking current when the output level is LOW (current flows from the device to the microcontroller).

Each I/O pin on the ATmega328P is capable of sourcing and sinking 40 mA. The maximum current rating of the microcontroller is 200 mA. All logic devices have four current values that are involved in determining how much current is necessary to properly derive the device:

- **Input HIGH current** (I_{IH}) – This is the input current that must flow into the input pin to create a HIGH logic level.
- **Input LOW current** (I_{IL}) – This is the input current that must flow out of the input pin to create a LOW logic level.
- **Output HIGH current** (I_{OH}) – This is the output current that flows out of the output pin when the output logic level is HIGH.
- **Output LOW current** (I_{OL}) – This is the output current that flows into the output pin when the output logic level is LOW.

10.2 Internal Pull-Up Resistors

There are internal pull-up resistors built in to each digital I/O pin on the ATmega328P. They have a value of approximately 36 kΩ. In order to make use of the pull-up resistors, write a value of 1 to a pin when it is configured as an input. The input pull-up resistors can be globally turned off by setting bit 4 (`PUD`, pull-up disable) in the

Microcontroller Control Register (MCUCR), but by default this bit is configured to enable the usage of the internal pull-ups. The I/O pin hardware that controls the input pull-up is shown in Figure 10.2.

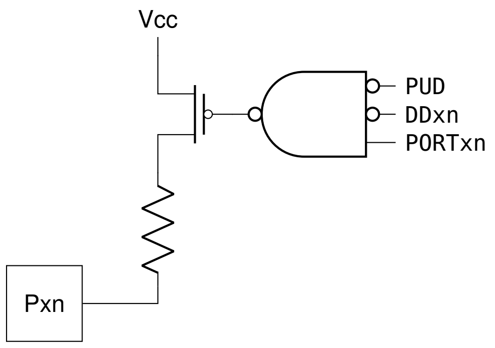


Figure 10.2: Hardware schematic of I/O port input pull-up resistors.

10.3 *Alternate Pin Functions*

All of the I/O pins on the ATmega328P microcontroller serve alternative purposes. This increases the functionality of the microcontroller without requiring extra space for extra pins. Some of these alternative functions include acting as sources for external interrupts, providing functionality for serial communication, converting analog to digital signals, etc. Each pin has extra hardware that allows for this extra functionality. Multiplexers route control signals to either have the pins act as regular I/O pins, or to function with their alternate purpose. The I/O pin multiplexers are shown schematically in Figure 10.3.

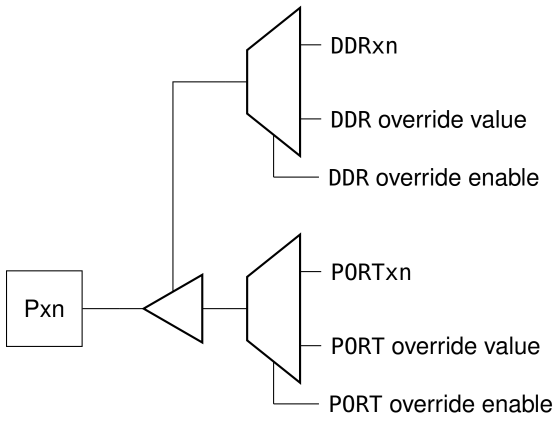


Figure 10.3: Hardware schematic of alternate I/O port functionality.

10.4 *Practice Problems*

1. True or false: All bits of Port B are accessible on the ATmega328P. FALSE

2. True or false: To change the data direction of pins on Port B, the register `DDRB` is used.

TRUE

3. What is the difference between `PORTC=0x00` and `DDRC=0x00`?

`PORTC=0x00` sets the output value of all pins on Port C to be equal to logic LOW. `DDRC=0x00` sets the directionality of all pins on Port C to be input pins.

4. True or false: All of the ATmega328P ports have 8 bits.

TRUE

5. True or false: Upon power-up, the I/O pins are configured as output ports.

FALSE

6. True or false: The `PORTx` register is used to send data out to ATmega328P pins.

TRUE

7. True or false: The `PINx` register is used to bring data into the CPU from ATmega328P pins.

TRUE

11 Analog to Digital Conversion

As discussed in chapter 12, many sensors output analog values. Table 11.1 gives a list of various types of digital and analog sensors.

Digital	**Analog**
Pushbutton	Potentiometer
Toggle Switch	Microphone
Keypad	Temperature sensor
Encoder	Photo-diode, -resistor

Table 11.1: Listing of digital and analog sensors.

Analog values must be converted into digital values in order to work with the digital functionality of the microcontroller. By nature, analog signals are continuous, and they contain an infinite number of points, as shown in Figure 11.1. The voltage values can take on any real number between V_{MIN} (usually ground) and V_{MAX} (Vcc). To convert these into digital signals, the analog signal must be sampled, quantized, and encoded.

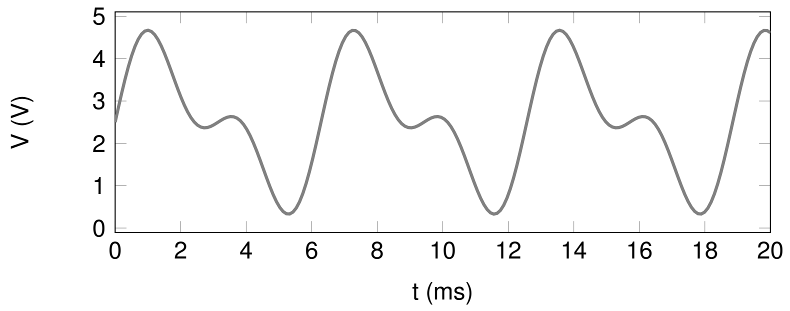


Figure 11.1: An analog signal takes on continuous values and contains an infinite number of points.

11.1 Sampling

An infinite number of points can't be digitized because it would require infinite processing power and infinite memory. Therefore, the analog signal must be sampled periodically. This sample rate is very important when doing signal processing (audio, video, etc.). It is inversely related to conversion accuracy. The fine details of sampling rate are beyond the scope of this course; you will very likely study the topic in a signals and systems electrical engineering course. Suffice it to say, when the ADC on the ATmega328P is used, the sample rate will be approximately 9,000× per second. Figure 11.2 shows the same analog signal sampled every millisecond. At each point in time, the sampled data is held in memory and then quantized and encoded.

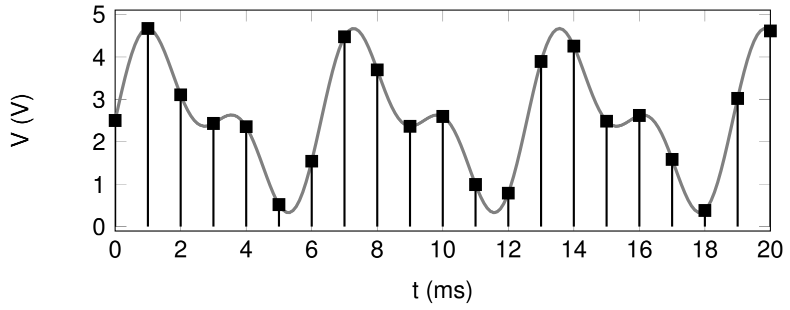


Figure 11.2: An analog signal sampled every millisecond.

11.2 Quantization

Because digital data cannot take on a continuum of values, the sampled analog data must be set equal to one of a number of discrete values. Figure 11.3 shows an analog signal that has been quantized. At every sampled point in the signal, that value is rounded down to the nearest possible division of $V_{MAX} \div 2^n$, where n refers to the number of bits of system resolution.

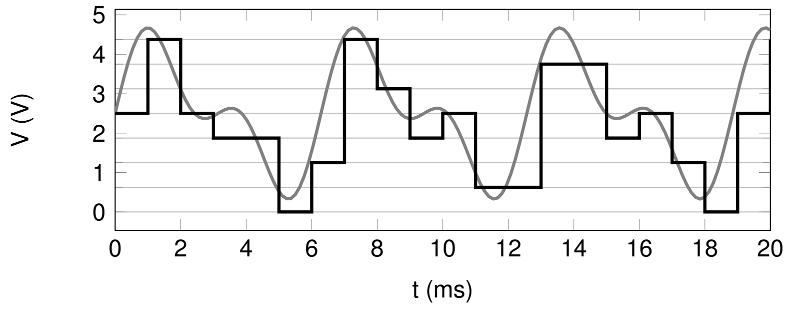


Figure 11.3: An analog signal sampled every 1 ms that has been quantized in a 3-bit system.

The number of possible discrete values available depends on the **resolution** of the system. The resolution of an ADC is represented in bits. In general, higher resolution is better, meaning that more accuracy can be obtained in the final result. A 2-bit system has $2^2 = 4$ possible discrete values; an 8-bit system has $2^8 = 256$ possible values; the ADC on the ATmega328P is 10 bits, meaning there are $2^{10} = 1024$ discrete values.

Other ADC metrics include the **step size**, which is the minimum change in voltage required to change the output value. The step size

$$\Delta V = \frac{V_{MAX} - V_{MIN}}{2^n}.$$

Quantization Error

The **quantization error** refers to the difference between the actual and quantized voltage: error = $V_{\text{quantized}} - V_{\text{actual}}$. Quantization error can be reduced by increasing the sample rate, and by increasing the resolution of the ADC, as shown in Figure 11.4.

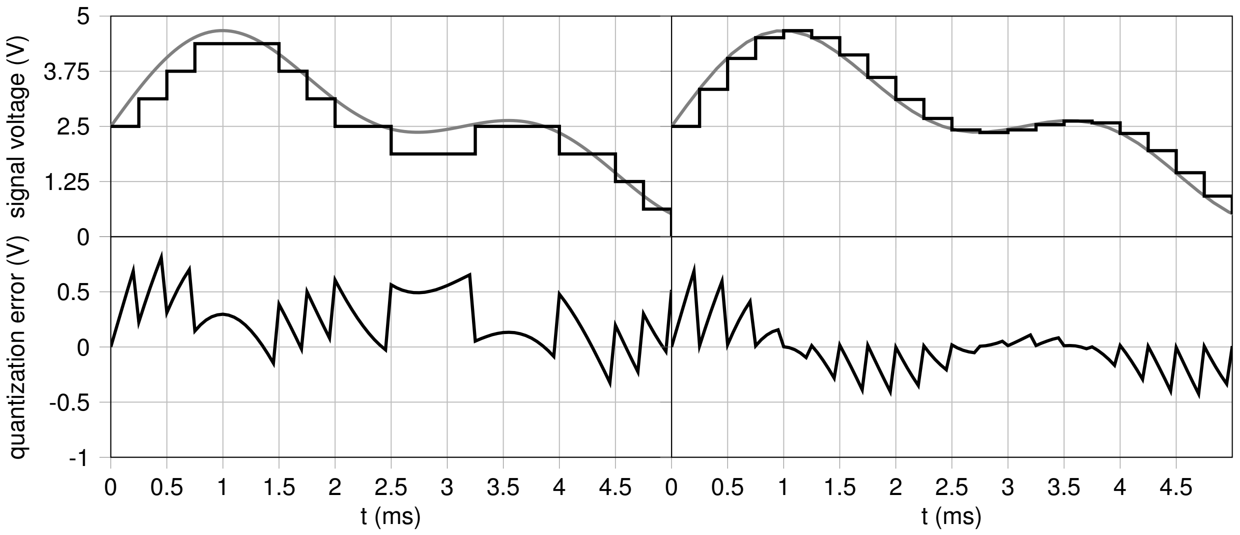


Figure 11.4: Quantized analog signals and associated quantization error for a 3-bit ADC (left) and an 8-bit ADC (right). The sample rate is 4× per millisecond.

11.3 *Encoding*

The encoding part of ADC comes when the sampled and quantized signal is then converted into a binary number. In hardware, this can be accomplished with a priority encoder.

11.4 ADC Architectures

There are many different hardware approaches to realizing an ADC. Each architecture has tradeoffs between size, resolution, and data processing time. While only three architectures are outlined here, there are many other possibilities that exist to convert analog signals to digital values.

Direct Conversion (Flash) ADC

A flash ADC, as shown in Figure 11.5 uses analog comparators to quantize an analog input signal.

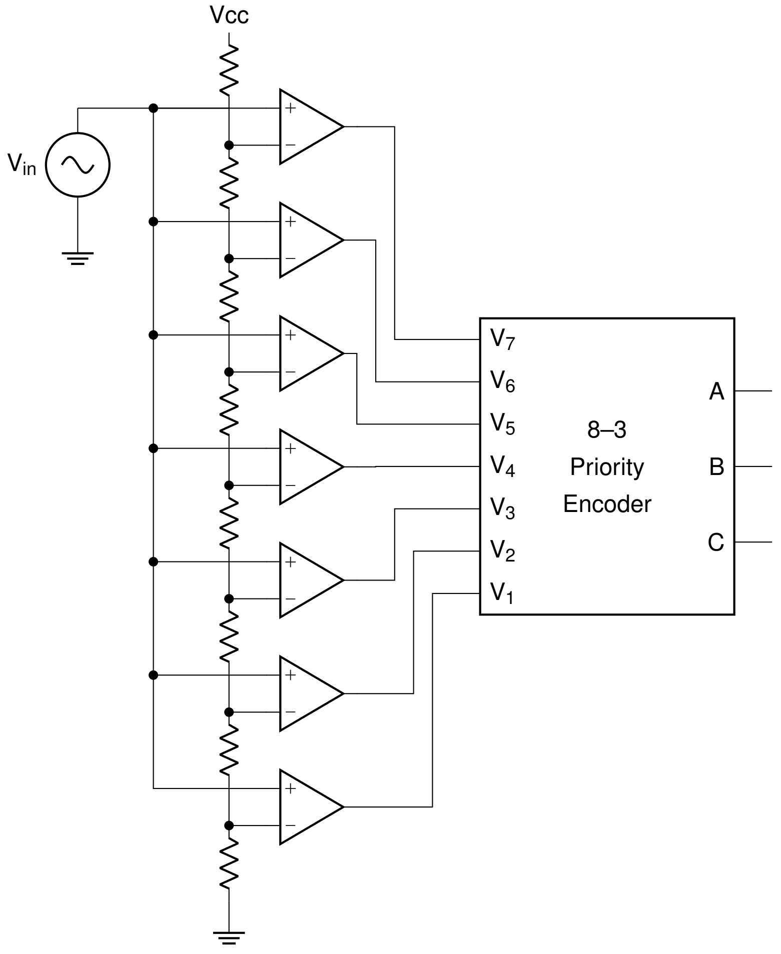


Figure 11.5: A 3-bit flash analog to digital converter.

Each comparator output (V_n) goes to Vcc if the applied voltage is greater than $\frac{n}{2^m} \times$ Vcc, where n is the number of the comparator, and m is the number of bits of system resolution. For example, the output V_6 becomes one when $V_{in} > \frac{6}{2^3} \times$Vcc. An encoder then takes the quantized signal and converts it into a binary output.

Flash ADCs are quick; they only take a single clock cycle to compute the output. However, they consume relatively high amounts of power and also require $2^m - 1$ comparators, which means they require a form factor that is unreasonably large for high resolution data converters. A 10-bit ADC would require 1023 analog comparators!

Successive Approximation Register (SAR) ADC

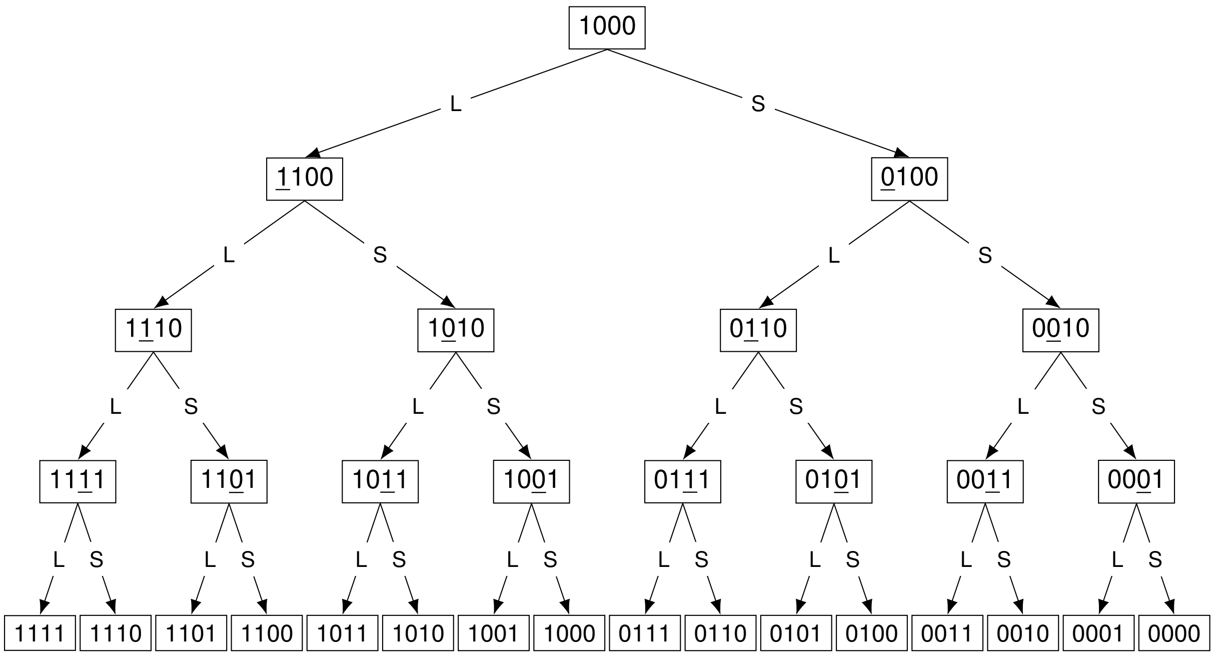


Figure 11.6: Process flow for a 4-bit successive approximation register analog to digital converter.

The successive approximation register is the type used in the ATmega328P microcontroller. It works by comparing the analog input to half of the value of V_{MAX}, then depending on the result, comparing in intervals of half the remaining difference until the final result is determined. It requires a properly calibrated digital to analog converter (DAC), but only requires a single analog comparator, so the form factor is relatively small. It takes m clock cycles to compute a final result, where m is the number of bits of system resolution. The SAR ADC in the ATmega328P requires 13 clock cycles to execute. An SAR ADC consumes much less power than a flash ADC.

A process flow of how a 4-bit SAR ADC calculates values is shown in Figure 11.6. The analog input value is first compared with one half of the maximum value. If the analog input is larger than this value (indicated by the letter L), then it is compared to successively larger approximations of the value. If the analog input is smaller than this value (indicated by the letter S), then it is compared to successively smaller approximations of the value until settling on the result.

Pipelined ADC

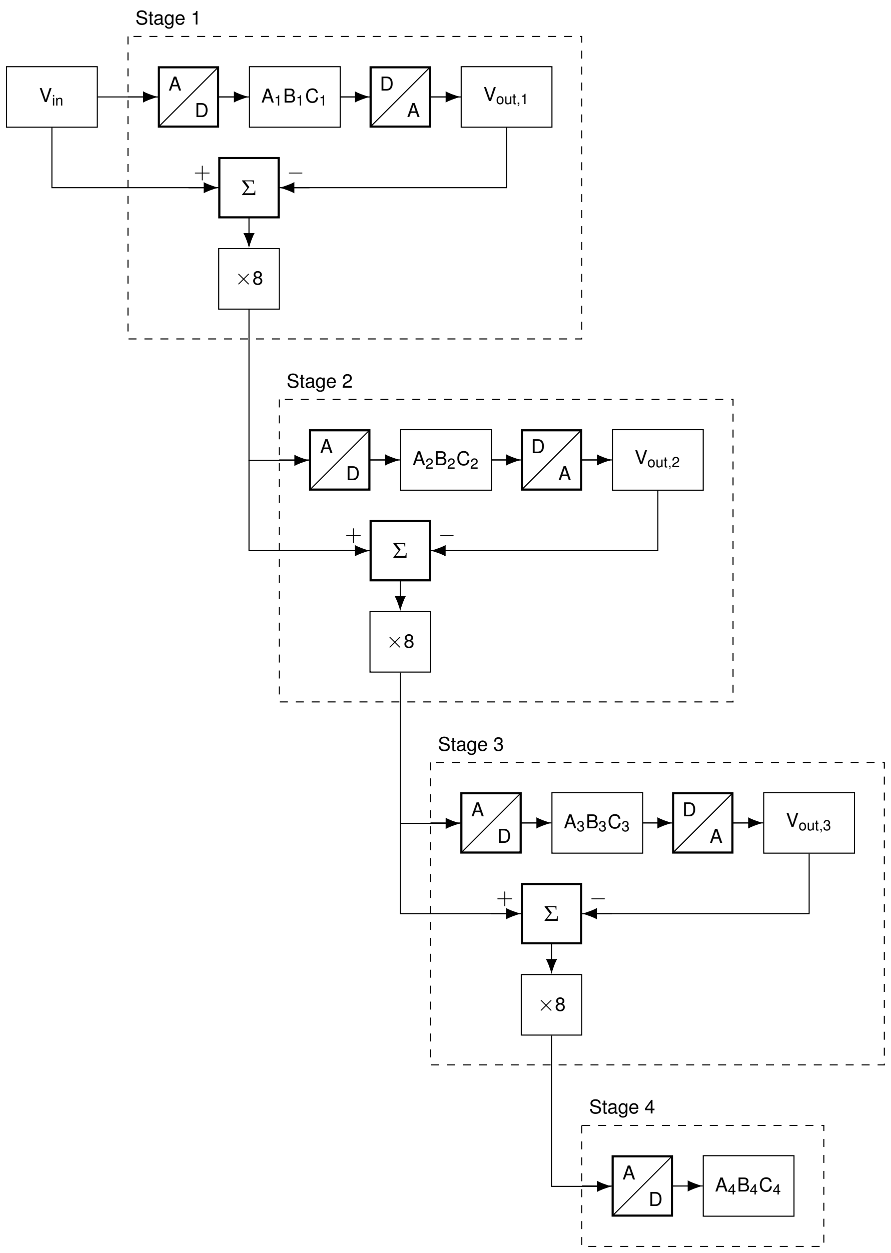


Figure 11.7: A 12-bit pipelined ADC with 3-bit flash ADC stage.

A pipelined ADC has q stages, where q is equal to the total device resolution divided by stage resolution. Each stage (which has m bits of resolution) uses an m-bit flash ADC to calculate m bits of the result at a time. For example, in Figure 11.7, a 12-bit pipelined ADC uses four 3-bit flash stages. The first stage calculates the first 3 most significant bits of the result. This is then subtracted from the input value, and then the next 3 significant bits are calculated. This process iterates until the final result has been calculated, giving an output of the form $\underline{A_1B_1C_1}$ $\underline{A_2B_2C_2}$ $\underline{A_3B_3C_3}$ $\underline{A_4B_4C_4}$. The pipelined ADC requires only q clock cycles to compute the output.

11.5 The ADC on the ATmega328P

The ATmega328P contains a 10-bit successive approximation ADC. It is connected to an 8-channel analog multiplexer that selects from eight different input sources (including all of the pins in PORT C). The minimum voltage level is 0 V (ground) and the maximum voltage level can be selected from different sources (either an externally applied reference voltage AREF, the internal value of Vcc, or 1.1 V). A block diagram of the ADC is shown in Figure 11.8.

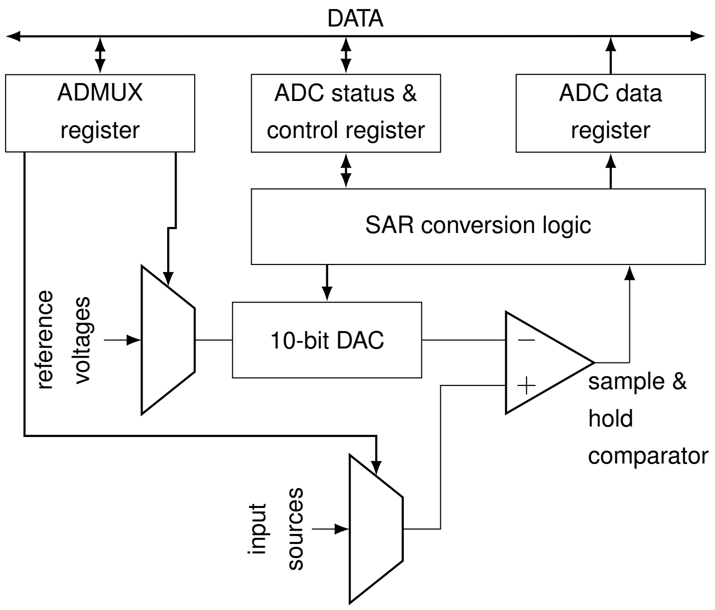


Figure 11.8: Analog to digital converter block diagram.

The number of available input sources depends on the package of the ATmega328P. The 28-pin chips do not contain pins for ADC6 and ADC7. However, the 32-pin chips do.

11.6 Digital to Analog Conversion

While the ATmega328P does not contain a standalone digital to analog converter (DAC) module to provide analog voltage levels, it is important to understand the concept of digital to analog conversion. A DAC performs the reverse function of an ADC, and as indicated previously in this chapter, is a crucial component included in SAR and pipelined ADC architectures. The analog voltage output of a DAC is given by

$$V_{\text{OUT}} = \text{Vcc} \times \frac{\text{value}}{2^n},$$

where n is equal to the the resolution of the converter.

A digital datastream is sampled and held, in which case the digital value is then converted into an analog voltage level. An R-2R ladder is one of many common architectures used to convert a binary value into a voltage, and is depicted schematically in Figure 11.9.

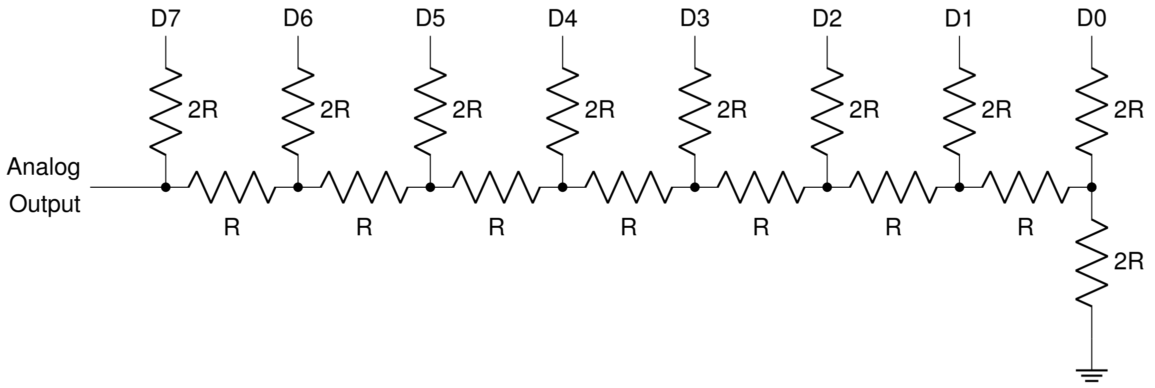


Figure 11.9: Schematic of a 3-bit R-2R digital to analog converter.

11.7 *Practice Problems*

1. Find the step size for an 8-bit ADC, if Vref = 1.28 V. — $\Delta V = 5$ mV

2. Given the situation in Question 1, calculate the output if the analog input is 0.7 V. — `1000 1100`

3. Given the situation in Question 1, calculate the output if the analog input is 1.0 V. — `1100 1000`

4. How many bits of resolution does the ATmega328P microcontroller have? — 10 bits

5. True or false: The output of most sensors is analog. — TRUE

6. Calculate the step size for the following ADCs, given Vref = 5 V.

   (a) 8-bit ADC — $\Delta V = 19.5$ mV

   (b) 10-bit ADC — $\Delta V = 4.9$ mV

(c) 12-bit ADC ΔV = 1.2 mV

(d) 16-bit ADC ΔV = 15.3 μV

7. Given a Vref of 2.56 V, find the corresponding Vin for each of the 8-bit ADC outputs.

(a) `1111 1111` Vin = 2.55 V

(b) `1001 1001` Vin = 1.53 V

(c) `0110 1100` Vin = 1.08 V

12
Sensors & Sensor Calibration

In order to obtain information about the world around us, sensors must be used. Sensors take information about the external environment and convert (transduce) them into electrical signals. These electrical signals are voltage, current, and resistance. Devices with electrical properties dependent on a particular physical quantity are chosen (photoresistors, temperature-sensitive transistors, piezoelectric ceramics) to measure that property. Figure 12.1 shows example block diagrams of this property in a temperature sensor (such as the TMP36) and a photoresistor.

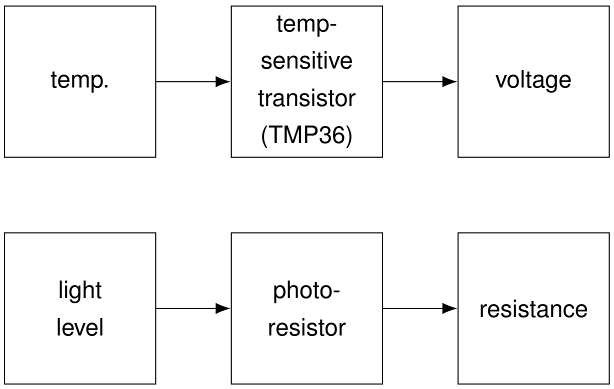


Figure 12.1: Diagram of how the TMP36 temperature sensor transduces temperature to voltage, as well as how a photoresistor transduces light level to resistance.

Several types of sensors are listed in Table 12.1.

Table 12.1: Various types of sensors.

Light Level Sensors
Photoresistor
Photodiode
Phototransistor
Distance, Speed, Acceleration Sensors

Ultrasonic detector
Accelerometer
Wheel encoder / tachometer
Sound Intensity
Microphone
Digital Sensors
Switches
Toggles
Pushbuttons
Environmental Conditions
Temperature sensor
Humidity sensor
Barometer

Some sensors provide digital data output. For example, a push-button is capable of generating a digital logic value of HIGH or LOW depending on whether or not the button is pressed. Simple digital sensors such as this can be directly connected to any pin on the AT-mega328P that has been configured as an input pin.

Of sensors that provide digital data output, some have been built to send data using a serial communication protocol. Devices that make use of USART, SPI, or TWI can interface with the ATmega328P using serial I/O, as is discussed in chapter 15. Making sense of this data will require consulting the sensor datasheet.

Most sensors, however, provide analog information, which may be in the form of a voltage output or a changing resistance. Sensors that output a voltage can interface directly with the ATmega328P using an analog pin. Sensors that output a resistance must first be configured with another resistor in series (R_C) in order to convert the variable resistance into an analog voltage value, as shown in Figure 12.2.

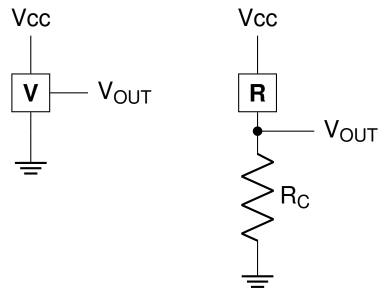


Figure 12.2: Schematics of how to connect sensors that output (V) voltage and (R) resistance to the ADC on the ATmega328P.

12.1 *Choosing Resistor Values*

With sensors that output variable resistance values, it is important to correctly chose the value of R_C in order to obtain an appropriate output value. Using the schematic in Figure 12.2, an equation between the output voltage, Vcc, and both resistors is given by

$$V_{OUT} = \text{Vcc}\,\frac{R_C}{R_C + R}.$$

The output voltage will be maximum (V_{MAX}) when the sensor resistance is lowest, and it will be minimum (V_{MIN}) when the sensor resistance is greatest. A value of R_C must be chosen such that $V_{MAX}-V_{MIN}$ (the contrast) is maximum.

In the specific case of the photoresistor, the maximum resistance in the device corresponds to when it is dark (R_{DARK} = 1 MΩ), and the minimum resistance in the device corresponds to when it is light (R_{LIGHT} = 10 kΩ). Using the above equation to determine V_{MAX} and V_{MIN} as a function of R_C, it is found that

$$V_{MAX} = \text{Vcc}\,\frac{R_C}{R_C + 10\text{k}\Omega}, \text{ and } V_{MIN} = \text{Vcc}\,\frac{R_C}{R_C + 1\text{M}\Omega}.$$

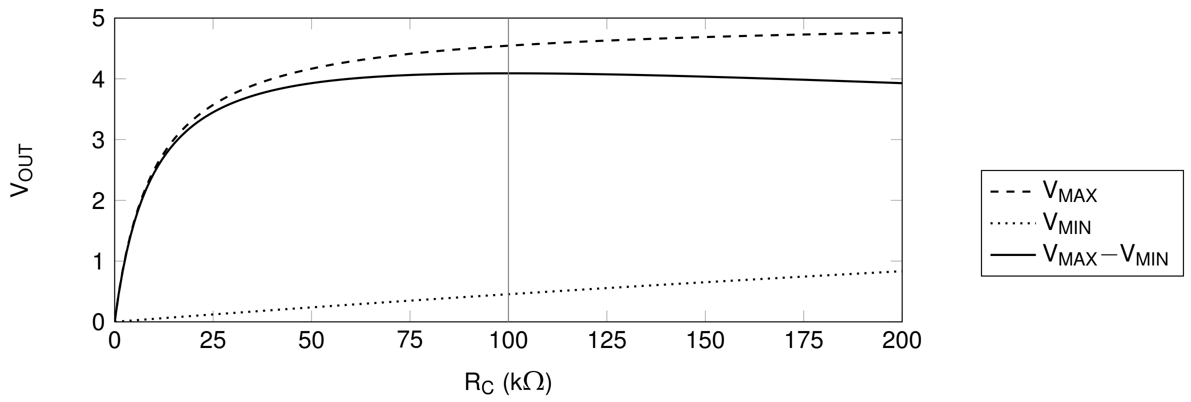


Figure 12.3: The maximum and minimum voltages and contrast as a function of the value of R_C.

Plotting the photoresistor equations for V_{MAX}, V_{MIN} and the difference between them, it can be shown in Figure 12.3 that the contrast between these two signals occurs when R_C = 100 kΩ. That is the value of R_C that should be chosen for that particular application. This process must be carried out individually for every sensor used.

After determining the value of R_C, it is next imperative to ensure that the power rating of the resistor is sufficient. From the equation P = IV, it can be shown that the power is maximum when the total resistance is minimized. Referring to the photoresistor example, the maximum power can be calculated as 0.2 mW, meaning that a 1/4 W resistor is sufficient to ensure that none of the components will overheat and melt.

12.2 Sensor Calibration

In order to obtain meaningful information from sensors, they must be characterized and calibrated. In other words, it is critical to know how to convert the output voltage into the physical quantity that it is meant to represent. This can be accomplished by checking the sensor datasheet or by performing a manual calibration.

Datasheet Calibration

Some sensors have datasheets which explain how the output corresponds to the physical quantity. For example, the TMP36 temperature sensor datasheet states that the output has a slope of 10 mV/°C and that an output voltage of 750 mV indicates a temperature of 25°C. From this, the equation between temperature (in °C) and voltage (in V) can be derived as

$$T = 100 \times V - 50.$$

As discussed in chapter 11, the ATmega328P has a 10-bit analog to digital converter (ADC), which outputs a value of 0 when the input voltage is 0 V, and a value of 1023 when the input voltage is 5 V. Therefore an equation can be derived directly between the value of the ADC and the temperature. This data, as shown in Figure 12.4, yields a relationship between temperature and ADC value of

$$\text{T} = 0.489 \times \text{AC} - 50.$$

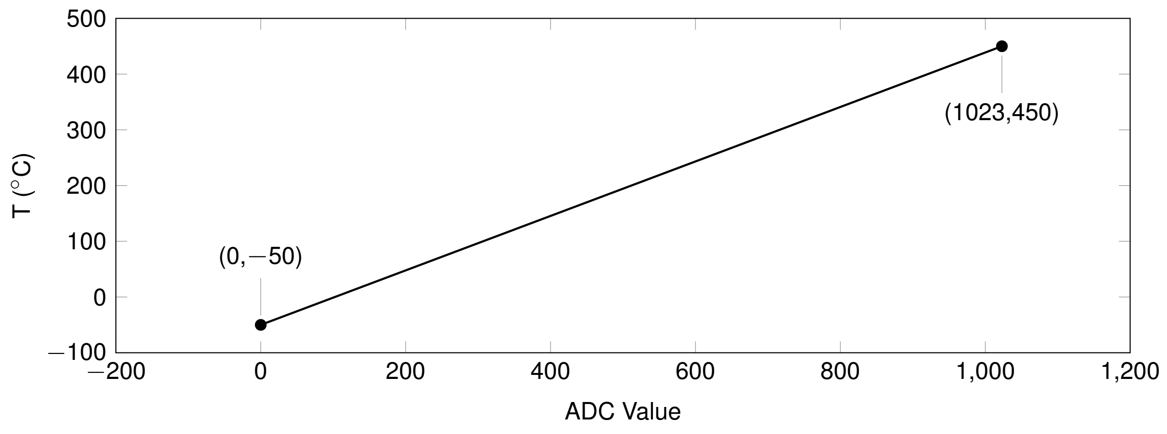


Figure 12.4: Linear relationship between temperature and ADC value of the TMP36 temperature sensor, given by the TMP36 datasheet.

This can be programmed onto the ATmega328P, avoiding the use of floating-point math, using the equation

```
T = 500L*ADC>>10 − 50.
```

One-Point Calibration

When put into practice, the sensor may not give the output that that was expected. If there is a discrepancy between the output of the sensor and that of a trustworthy reference, the data must be further calibrated. One-point calibration used when slope is correct but there's an offset between measured and actual data (as measured by a trustworthy reference). If the measured data is greater than the actual data, the offset must be subtracted, and if the measured data is less than the actual data, the offset must be added. This is shown graphically in Figure 12.5.

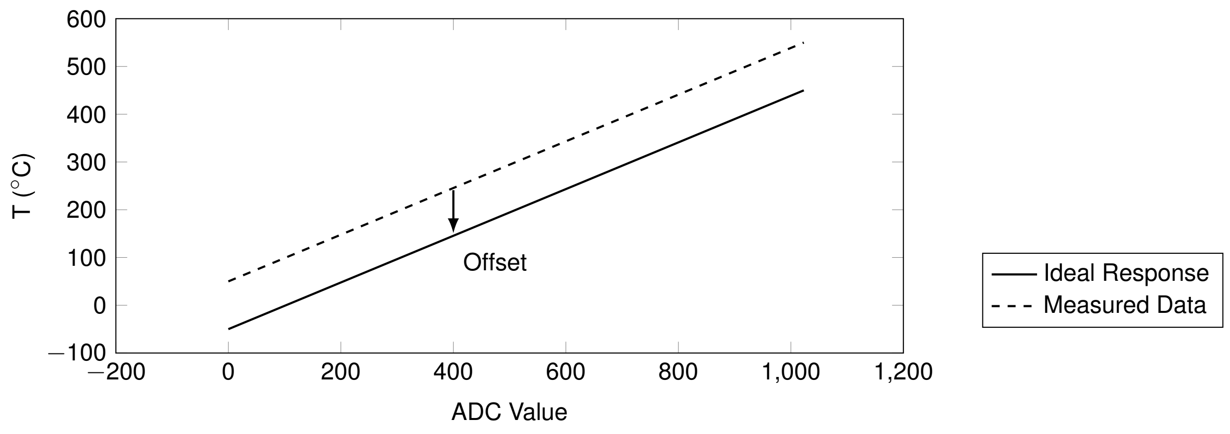


Figure 12.5: Example of one-point calibration; finding an offset value changes the measured response to the ideal (expected) result.

Multiple-Point Calibration

If the output response of a sensor is not known, it must be found using a reference instrument to measure the physical quantity and compare it to the ADC values output by the sensor. Use as many known reference points as possible, and use plotting software such as Excel to find a best fit line, keeping in mind that the best fit line may not always be linear. This best fit line will then be used in software to convert the ADC value into the corresponding physical quantity. A hypothetical example is shown in Figure 12.6 with a linear relationship found between sound intensity and ADC values.

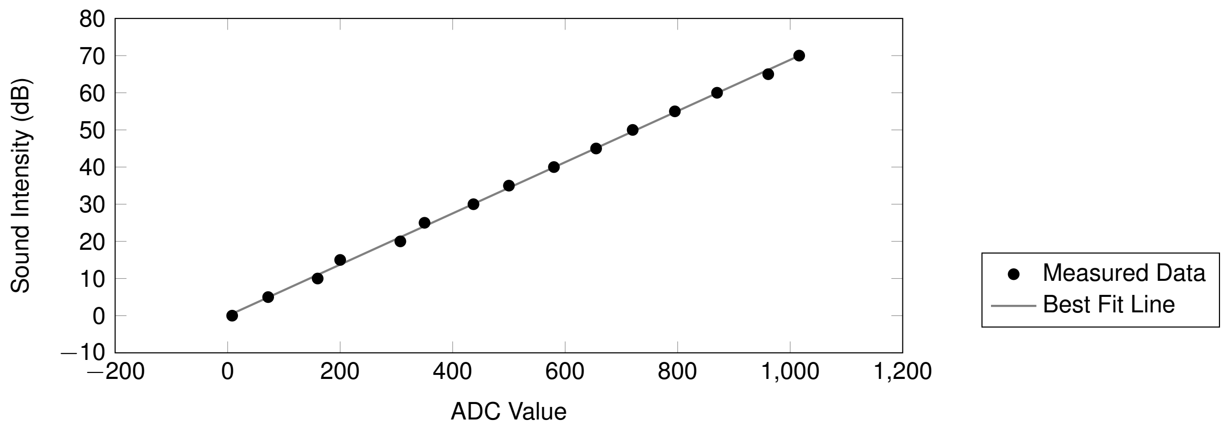


Figure 12.6: Example of multiple point calibration to determine the ADC value given at different sound intensity levels.

12.3 *Mitigating Fluctuating Data & Sensor Noise*

Sensor data can fluctuate over time due to environmental conditions or noise. It may be important to perform a rolling average to obtain a steady, reliable readout. A rolling average uses n previous values (stored in an array) to calculate the current average value. Care must be taken in choosing an appropriate value for n, which must be specifically tailored to each different sensor and situation in which it is to be used. Table 12.2 explains what happens if n is small or large.

Table 12.2: Issues that can result if n is chosen to be either small or large..

If n is small...
Program uses less memory
Program takes less time to initialize
Sensor is more susceptible to noise
If n is large...
Program takes more memory
Program takes more time to initialize
Sensor is less susceptible to noise
If n is too large, the sensor is not sensitive to short-term or quick changes

Sensor data from a temperature sensor that has been averaged out using three different values of n (as well as the raw data, which has an n value of 1) is shown in Figure 12.7.

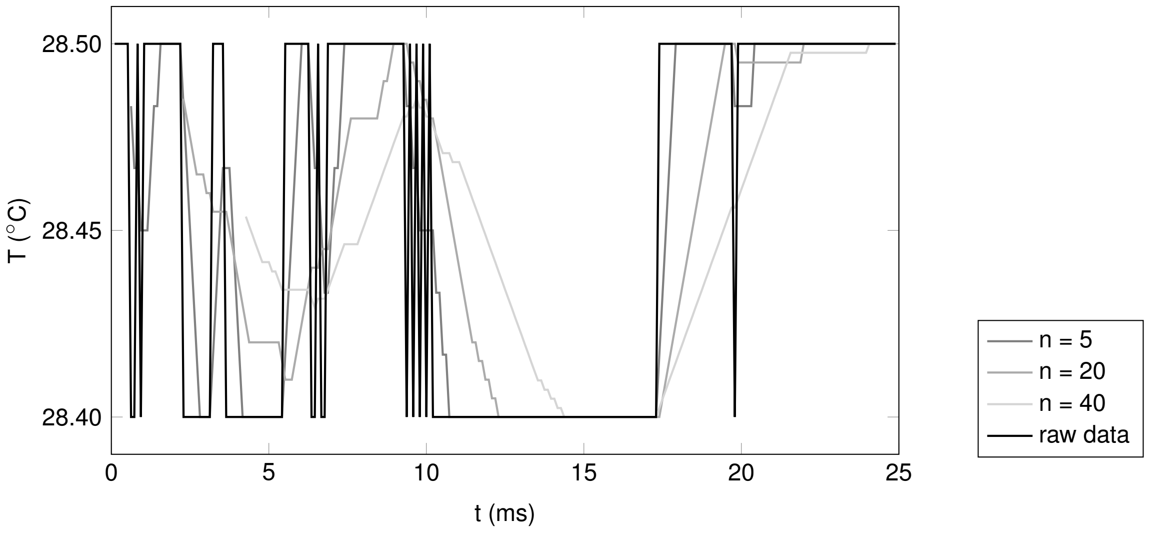
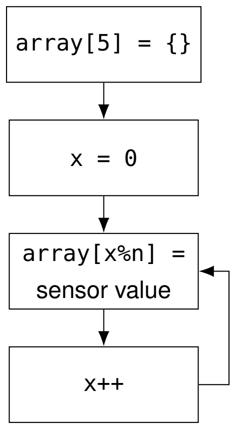


Figure 12.7: Temperature data subjected to rolling average with different values of n.

Circular Buffer

A circular buffer is a method of taking a rolling average to clean up sensor noise and fluctuations. It is simply an array of n sensor values, where the first value in is also the first value out. (This type of situation is known as FIFO: First In First Out.) A flowchart and sample array is shown in Figure 12.8 to explain this process.



array index	0	1	2	3	4
step 1	17.2				
step 2	17.2	16.5			
step 3	17.2	16.5	17.8		
step 4	17.2	16.5	17.8	16.9	
step 5	17.2	16.5	17.8	16.9	17.0
step 6	18.0	16.5	17.8	16.9	17.0
step 7	18.0	18.2	17.8	16.9	17.0

Figure 12.8: Flow chart and example array of values in a circular buffer.

13
Interrupts

An interrupt is an important asynchronous event that requires immediate attention. For example, in monitoring an oil refinery, many sensors are checked in sequence and cycled through (a system known as **polling**). If a fire breaks out, it would not be prudent to wait until the fire sensor is checked to know about it. It would also not be wise to only check the fire sensor, or to check it excessively, because the program must do other things as well. Instead: the fire sensor generates an **interrupt request** if a fire is detected.

13.1 *Program Flow*

An example program flow that compares continuous polling to interrupts is shown in Figure 13.1.

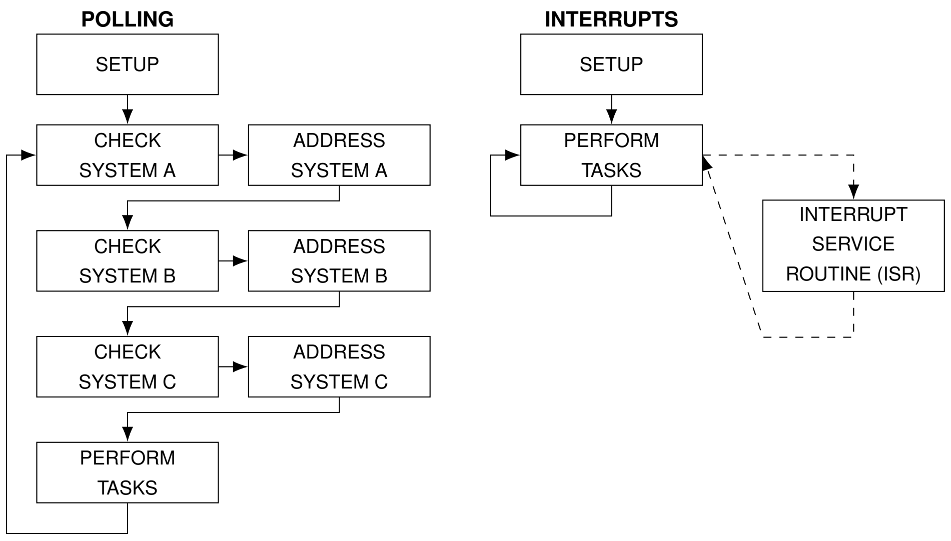


Figure 13.1: Program flow of software that uses continuous polling to monitor systems, vs. software that uses interrupts to asynchronously handle important events.

Today's embedded systems have many features that require many systems to be monitored. For example, a smart phone contains at the very least a power switch, home button, screen, keypad, app buttons, and microphone. If all of these inputs were continuously polled to check their status, the phone would never be able to accomplish other tasks.

Interrupts are used for performance; asynchronously handling important events ensures that the system can carry out its regular functionality. In addition, immediate action can occur upon a changing input, rather than waiting for the software flow to poll that particular input. Interrupts allow a system timer to be used to trigger updates at regular intervals. Interrupts can also be used to wake a device up from a low power mode, allowing the device to save power when not in use. The drawbacks to interrupts deal with their asynchronous nature. Because an interrupt can be invoked at any time (even in the middle of another operation), code needs to be carefully written to reflect that fact.

13.2 Interrupt Service Routine (ISR)

An interrupt service routine (ISR) is a subroutine that the microcontroller executes when an interrupt is invoked. It is at times referred to as an interrupt handler. ISRs should be as short and fast as possible, so as to deal with the asynchronous event without detracting from the functionality of the program in general. Because an ISR is never formally invoked in software (it is only asynchronously invoked upon a specific event), the function must take no arguments, must not return any values, and cannot use any other functions that themselves use interrupts. Any variable that must be shared with an ISR must be global, `volatile`, and protected.

When the interrupt is invoked, program goes to the memory location of the associated **interrupt vector**. Each of these vectors can have an associated subroutine written for it in the Arduino IDE. The group of memory locations that holds the ISR addresses is called the **interrupt vector table**. (An abridged version of the ATmega328P interrupt vector table is given in Table 13.1.) The lower the address of the ISR location, the higher its **priority**. If two interrupts are triggered simultaneously, the one with the highest priority will be serviced first, followed by the second.

13.3 ISR Execution

When an ISR is invoked, the microcontroller...

1. Finishes executing the current instruction

Vector	Address	Source	Interrupt Definition
1	`0x0000`	`RESET`	External pin, power-on reset
2	`0x0002`	`INT0`	External interrupt request 0
3	`0x0004`	`INT1`	External interrupt request 1
4 – 6	`0x0006` – `0B`	`PCINTx`	Pin change interrupt requests
7	`0x000C`	`WDT`	Watchdog time-out interrupt
8 – 17	`0x000E` – `21`	`TIMERx`	Timer / counter interrupts
18 – 21	`0x0022` – `29`		Serial communication interrupts
22 – 26	`0x002A` – `33`		Other peripheral interrupts (ADC, EEPROM, etc.)

Table 13.1: Abridged version of the ATmega328P interrupt vector table.

2. Saves the address of the next instruction in the stack
3. Jumps to interrupt vector table
4. Loads memory address of the associated ISR into the program counter
5. Clears global interrupt enable flag in `SREG` (`I=0`)
6. Runs the ISR subroutine until its end (`RETI` = return from interrupt instruction)
7. Sets global interrupt enable flag in `SREG` (`I=1`)
8. Loads the memory address to return to from the stack into the program counter
9. Executes the next instruction
10. Either
    (a) Continues fetch/execute as normal
    (b) Services the next interrupt (if there is one)

13.4 *ISR Categories*

Interrupt service routines can be categorized as external (coming from an outside source) or internal (coming from a hardware peripheral). External interrupts include pin interrupts, which are associated with certain pins on the ATmega328P. Pins D2 and D3 can invoke interrupts, each with their own associated ISR. These are the interrupts `INT0` and `INT1`. Different invoking situations (rising edge, falling edge, low level, toggle) means that the exact status of the input when the ISR is invoked can be known. However, they are only available on two pins. Pin-change interrupts are available on all pins on the ATmega328P, however they do not have unique ISRs, so extra software is required to determine exactly what pin was changed and what condition it is in.

Internal peripheral interrupts use timers to create routine tasks, create interrupts upon successful transmission or receipt of serial communications, notify the program when an ADC conversion has been completed, or upon successful write to EEPROM memory.

Reset

Reset is a special category of interrupt on the ATmega328P. It is the highest priority ISR, and cannot be disabled. When the ATmega328P is first powered on, initial values of the program counter, flip-flops, I/O control registers, etc. are unknown. The reset ISR sets these critical registers to initial values upon reset. The program counter is set to `0x0000` upon reset (shown in Figure 13.2), therefore it begins by servicing the reset subroutine before carrying out the rest of the program. There are several reset sources on the ATmega328P microcontroller. One of them occurs when the microcontroller is first powered on. An external button on the Arduino is connected to the $\overline{\texttt{RESET}}$ pin on the ATmega328P, which creates a reset condition when the pin is held low for longer than a specified minimum amount of time. In addition, brown-out resets and watchdog resets can be enabled to trigger the reset condition.

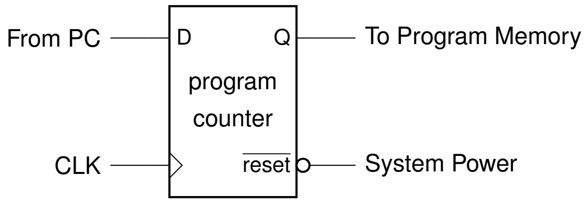


Figure 13.2: The program counter is initialized to a value of `0x0000` when system power is shut off.

13.5 *Enabling / Disabling Interrupts*

At times, it is useful to enable or disable interrupts. This can happen globally, in that all interrupts are enabled/disabled, or locally, in which only individual interrupts are enabled/disabled. When the ATmega328P resets, all interrupts are disabled to allow the device to reset without interrupt. Interrupts can be globally disabled by clearing the interrupt flag in `SREG` (`I=0`). Interrupts can be globally enabled by setting the interrupt flag in `SREG` (`I=1`). Interrupts are globally disabled upon entering an ISR, and then globally enabled once the ISR has been serviced. **Nonmaskable** interrupts cannot be disabled, for example the reset interrupt.

13.6 Practice Problems

1. Which of the two techniques (interrupts or polling) requires more system resource usage on the microcontroller? polling

2. What is the memory address of the watchdog timer interrupt? `0x000C`

3. True or false: While servicing an interrupt service routine, interrupts are globally enabled. FALSE

4. Is the reset interrupt maskable or nonmaskable? nonmaskable

5. When two interrupts are invoked simultaneously, the one with the ______ memory address will be serviced first. lower

6. If `INT0` and `INT1` are invoked simultaneously, which will be serviced first? `INT0`

14
Clocks, Timers / Counters & Pulse-Width Modulation

A MICROCONTROLLER NEEDS A CLOCK to keep time for all of its hardware components and peripherals. A clock signal is a square wave that oscillates between 0 V and Vcc at regular intervals. A clock is also useful to create time delays in code (for example, to blink an LED or to check sensors at regular intervals). A list of different technologies that can be used to generate clock sources is provided in Table 14.1.

Table 14.1: A list of various clock sources.

Ceramic Resonators
– Piezoelectric ceramic material – Internal vibrations create an oscillating voltage at a specific frequency – Not accurate enough to be used for a CPU clock (0.5% tolerance)
Crystal Oscillators
– Piezoelectric crystal – Internal vibrations create an oscillating voltage at a specific frequency – Highly accurate and can be used for system clocks (0.001% tolerance)
RC Circuits
– Amplifiers – 555 timers – RLC circuits

Not all of these technologies directly generate a clock signal. Some of them (such as an RLC circuit) generate a sinusoidal wave and some (such as piezoelectric crystals) may not generate the proper voltage levels necessary for a digital clock.

All microcontrollers contain circuitry to condition an input oscillation and turn it into a clock signal. They also have the capability to change the clock frequency. **Clock multipliers** increase the frequency by using a phase-locked loop. The advantage of this is to have faster code execution. However, all microcontrollers have maximum frequencies beyond which they will not work reliably, which must be taken into consideration before multiplying the frequency of any clock signal. Increasing the clock frequency also has the side effect of increasing the power consumption of the microcontroller.

Clock dividers (prescalers) decrease the frequency. They use a timer to count to a particular prescaler value (2, 8, 1024, etc) and when the timer reaches that value, an overflow causes a pin to toggle. That toggle becomes the new clock signal. Slower clocks indicate a longer time to execute code but use less power.

14.1 ATmega328P Clock

The ATmega328P has two internal clocks: an 8 MHz RC oscillator and a 128 kHz lower power oscillator. An external clock (connected to the `XTAL1` pin), or a crystal or ceramic oscillator (connected to the `XTAL1` and `XTAL2` pins) may be used instead of one of the internal oscillators. The clock source is selected by changing fuse bits in the low fuse byte. The ATmega328P is rated for a fastest clock speed of 20 MHz. It has no clock multiplier, but is capable of using a global prescaler to divide the clock frequency.

The clock control unit, as shown in Figure 14.1, routes the clock signals to all device peripherals, as different peripherals require their own clock signals. The CPU clock (CLK_{CPU}) is routed to all parts involved with core operations. The I/O clock ($\text{CLK}_{\text{I/O}}$) is routed to all parts involved with input and output operations, including timer / counters, serial communication peripherals and external interrupts. The flash clock ($\text{CLK}_{\text{FLASH}}$) is routed to the program memory. When externally programming the chip, it is important to have access to program memory without the CPU running. The asynchronous clock (CLK_{ASY}) allows the asynchronous timer / counter to be clocked directly from an external source. Finally, the ADC clock (CLK_{ADC}) is used to power the ADC with the CPU clock off to reduce noise and increase the precision of the analog to digital conversion process.

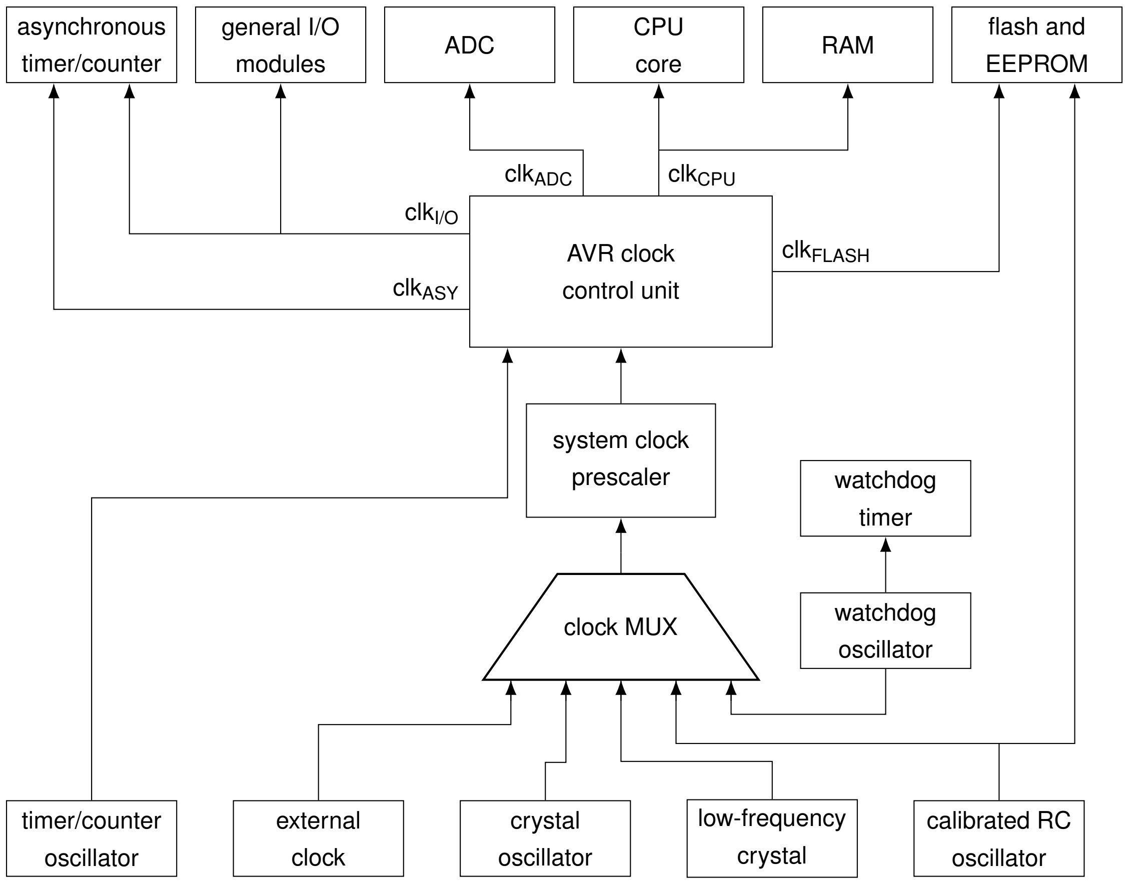


Figure 14.1: Schematic of the AVR clock distribution system.

14.2 Timer / Counters

Timer / counters are integral in the use of timed interrupts and pulse-width modulation (PWM). A counter counts up values from $0 \to 2^n - 1$, where n is the resolution of the counter. A timer is simply a counter which is clocked with the CPU clock. Timer / counters are used to generate waveforms (for example in PWM), measure time intervals, generate interrupts at specific intervals, and capture or count external events.

There are three timer / counters on the ATmega328P: Timer / counter 0 (TCNT0) and timer / counter 2 (TCNT2) are 8-bit counters. timer / counter 1 (TCNT1) is a 16-bit counter. Each counter can be assigned its own prescaler value to allow them to overflow at different frequencies.

There are different clock sources that can be used for each timer / counter to provide flexibility in timing. The first option, available on all three timer / counters, is to use the CPU clock. It is possible to prescale the clock to slow down the timing intervals, however there are a finite number of prescalers available to use.

The second option, available on timer / counters 0 and 1, is to use an external clock. The microcontroller then synchronizes the external clock to the system clock. This limits the range of frequencies of the external clock. Because it is sampled by the CPU clock, it must be slower than half the frequency of the CPU clock. (In fact, the ATmega328P datasheet recommends that the maximum external clock frequency be no greater than the CPU clock divided by 2.5.) The external clock frequency cannot be prescaled. While an external clock is being used, due to the synchronization of the CPU clock, this is not a truly asynchronous timing option.

The last option, available only on timer / counter 2, is to use a truly asynchronous oscillator. That is, the oscillator will independently clock the timer / counter without any synchronization to the CPU clock. Timing events (such as interrupts) will by definition occur outside of the timescale of the CPU clock, yet the CPU clock drives all of the hardware that deals with those events. Therefore, it is still necessary to limit the frequency of the asynchronous clock. It is recommended that the frequency be at least four times less than the CPU clock. In particular, the asynchronous timer / counter is meant for 32,768 Hz oscillators which, when prescaled, can be used to develop real time clocks.

Each of the timer / counter units can be simplified as a block diagram as shown in Figure 14.2

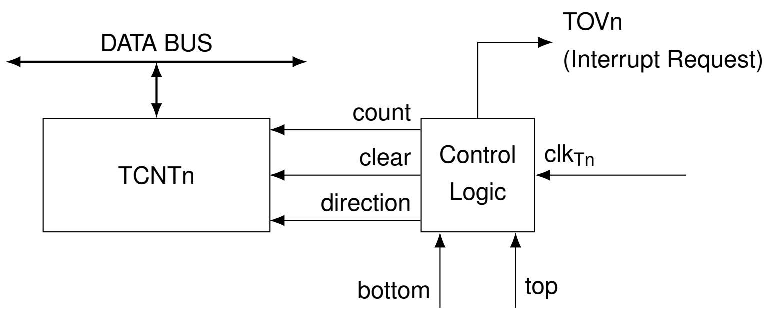


Figure 14.2: Block diagram of each timer / counter unit on the ATmega328P.

The definitions given in Table 14.2 are important to understand the operation of each timer / counter.

Each timer / counter has two units. The **output compare unit** functionality allows the timer / counter to generate square waves with differing frequencies and duty cycles. It is capable of generating PWM signals. The **input capture unit** (which is available with TCNT1) captures input signals and can calculate their frequencies and duty cycles.

Name	Definition
`bottom`	`0x00` (or `0x0000`)
`max`	`0xFF` (or `0xFFFF`)
`top`	highest value in the count sequence
`direction`	select between increment and decrement
`count`	signal to increment/decrement by 1
$\texttt{clk}_{\texttt{Tn}}$	timer / counter clock

Table 14.2: Important definitions to understand the operation of the timer / counter system.

Output Compare Unit

The output compare unit block diagram is shown in Figure 14.3. The value on the timer / counter register `TCNTn` is continuously compared with the values in output compare registers `OCRnA` and `OCRnB`. A match will set the corresponding output compare flag `OCFnA` or `OCFnB`. If interrupts are enabled, this will generate an output compare interrupt. Output signals on `OCnA` and `OCnB` can be generated based on the vaues of `WGMn` and `COMnx`, which are initialized in the timer / counter control registers.

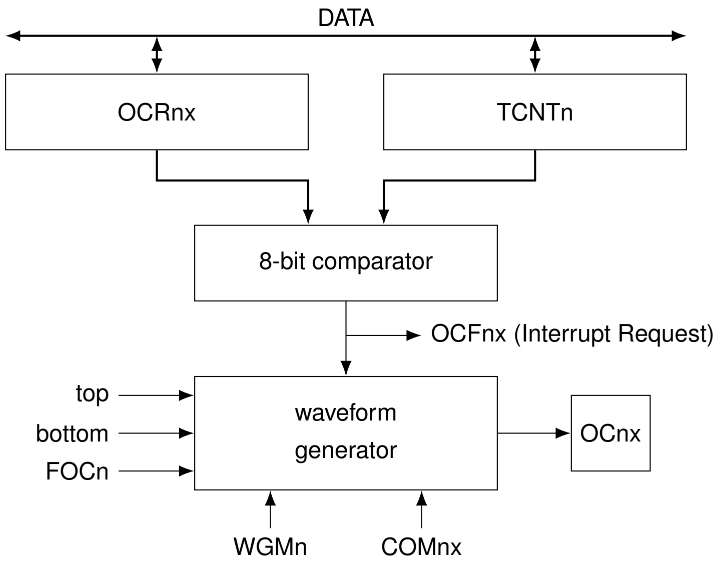


Figure 14.3: Block diagram of the timer / counter output compare unit.

Input Capture Unit (Timer / Counter 1 Only)

The input capture unit, as shown in Figure 14.4, is capable of triggering a capture from input pin `ICP1` based on the value of the `ICES` bit as configured in the control register. When `ICES = 0`, an input capture is triggered on a falling edge of the signal. Otherwise, an input capture is triggered on a rising edge. At that time, the value of `TCNT1` is written to the input capture register ICR1. At the same time, the input capture flag ICF1 is set to trigger an interrupt.

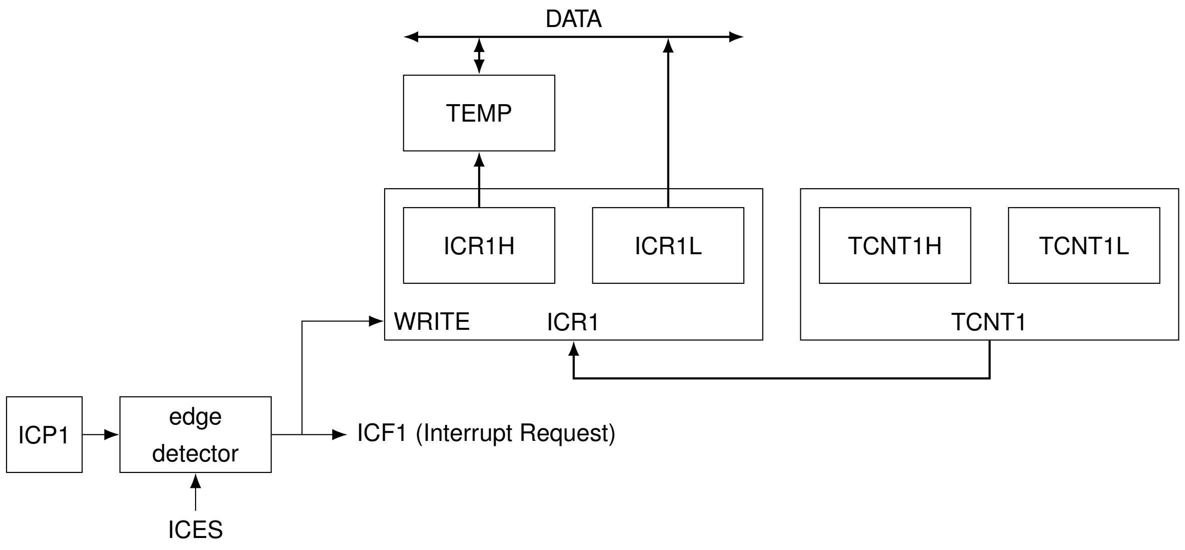


Figure 14.4: Block diagram of the timer / counter 1 input capture unit.

This unit can be used to measure the period and duty cycle of devices that output a PWM signal. To calculate the waveform period, the time at which the first falling edge occurs (T_1) is saved. The next falling edge, which occurs at time T_2 is similarly saved. The difference between these two values, multiplied by the period of the timer / counter, is the period of the waveform, given by

$$T_{INPUT} = (T_2 - T_1) \times T_{TC},$$

where T_{TC} is the period of the timer / counter clock. An example of this is shown in Figure 14.5, where

$$T_{INPUT} = (25 - 15) \times T_{TC} = 10 \times T_{TC}.$$

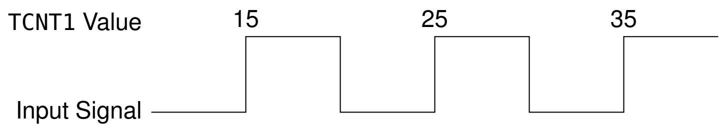


Figure 14.5: Using the timer / counter 1 input capture unit to calculate the period of an input signal.

To measure duty cycle, a rising edge, a falling edge, and the next rising edge are necessary. The period is measured as the difference in time between subsequent rising edges. The duty cycle is the amount of time the signal is HIGH (difference between rising and falling edges) divided by the total period. An example, shown in Figure 14.6, indicates that the input wave period is

$$T_{INPUT} = (25 - 15) \times T_{TC} = 10 \times T_{TC},$$

the amount of time that the input signal is HIGH is

$$T_{ON} = (18 - 15) \times T_{TC} = 3 \times T_{TC},$$

and therefore the duty cycle can be calculated as

$$\text{Duty Cycle} = T_{ON} \div T_{INPUT} = \frac{3}{10} = 30\%$$

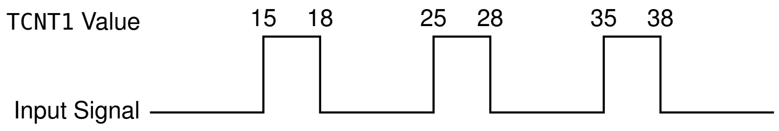


Figure 14.6: Using the timer / counter 1 input capture unit to calculate the period and duty cycle of an input signal.

14.3 Timer / Counter Modes of Operation

Using these two units, the timer / counter can carry out one of its many modes of operation. These are **normal** mode, **clear timer on compare match (CTC)** mode, **fast PWM** mode, **phase-correct PWM** mode, and **phase- and frequency-correct PWM** mode.

Normal Mode

Normal mode is capable of creating square waves with 50% duty cycle with varying frequencies. The frequency of the output wave depends on the value of the prescaler as well as the resolution of the timer / counter used. The timer / counter counts up from 0 to $(2^n - 1)$. An overflow flag (`TOVn`) is set when `TCNTn` becomes 0 after overflowing. The period of the output wave is

$$T_{NORMAL} = N \times 2^n \times T_{I/O},$$

where N is the value of the prescaler being used, n is the resolution of the timer / counter (either 8 or 16), and $T_{I/O}$ is the period of the I/O clock.

Clear Timer on Compare Match (CTC) Mode

CTC mode can generate square waves with 50% duty cycle with much more diversity in the possible output frequency. The timer/ counter counts up from 0 to the value stored in register `OCRnA`, with an interrupt generated when `TCNTn` becomes 0 after overflowing. The period of the output wave is

$$T_{CTC} = N \times (\texttt{OCRnA} + 1) \times T_{I/O},$$

where N is the value of the prescaler being used, `OCRnA` is the value saved in the corresponding register, and $T_{I/O}$ is the period of the I/O clock. An example waveform that can be created using CTC mode is shown in Figure 14.7.

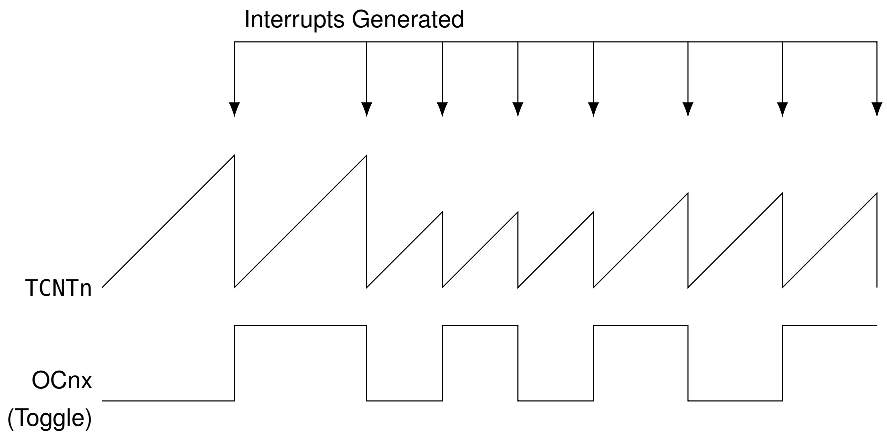


Figure 14.7: Clear timer on compare match (CTC) mode waveform.

14.4 *Pulse-Width Modulation (PWM)*

Pulse-width modulation (PWM) is used to generate digital signals with varying average voltage levels. Instead of outputting an analog value, which is not possible without a digital to analog converter (DAC), a digital pulse with varying frequency and duty cycle is output.

Duty Cycle & Average Voltage

As discussed, the duty cycle is the fraction of time that a signal is HIGH. By changing the duty cycle (D), the effective intensity (average voltage ($\overline{V}$) of a signal can be varied from OFF (duty cycle = 0%) to ON (duty cycle = 100%). The duty cycle can be calculated as

$$D = T_{HIGH} \div (T_{HIGH} + T_{LOW}),$$

where T_{HIGH} is the amount of time that the signal has a logic HIGH level, and T_{LOW} is the amount of time that the signal has a logic LOW level. The average voltage can be calculated as

$$\overline{V} = D \times V_{MAX} + (1 - D) \times V_{MIN},$$

where D is the duty cycle, V_{MAX} is the maximum signal voltage (usually Vcc), and V_{MIN} is the minimum signal voltage (usually 0 V).

Figure 14.8 shows five PWM waveforms, all with a period of 20 ms. Waveform A has a duty cycle of 0% and average voltage of

0 V. Waveform B has a duty cycle of 25% and a corresponding average voltage of 0.25 × 5 V = 1.25 V. Waveform C has a duty cycle of 50%, and average voltage of 0.5 × 5 V = 2.5 V. Waveform D has a duty cycle of 75%, giving it an average voltage of 0.75 × 5 V = 3.75 V. Waveform E has a duty cycle of 100% and average voltage of 5 V.

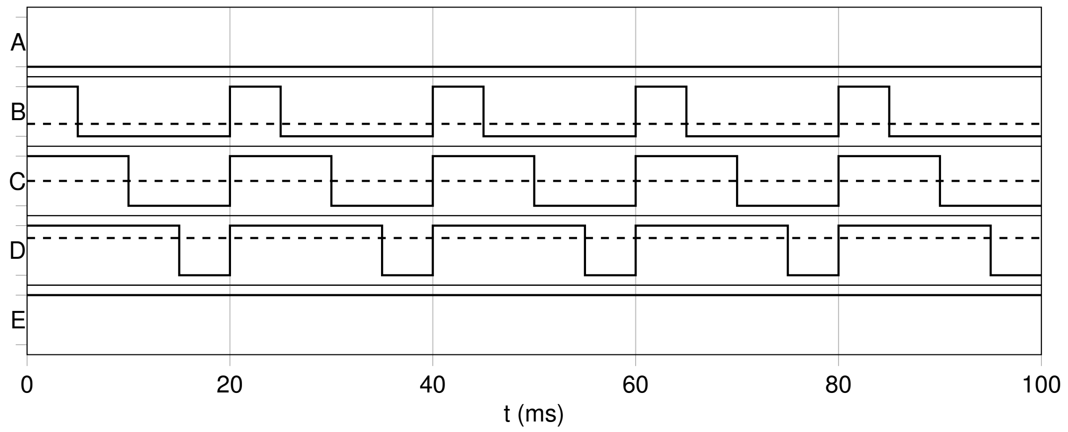


Figure 14.8: Pulse-width modulation waveforms with various duty cycles. The average voltage is indicated with a dashed line.

PWM Frequency

The PWM frequency is equal to the number of complete cycles that occur per unit of time. It is independent of the waveform duty cycle. Figure 14.9 shows waveforms, all with 25% duty cycles, with different frequencies. Waveform A has a frequency of 200 Hz, waveform B has a frequency of 100 Hz, waveform C has a frequency of 50 Hz, and waveform D has a frequency of 20 Hz.

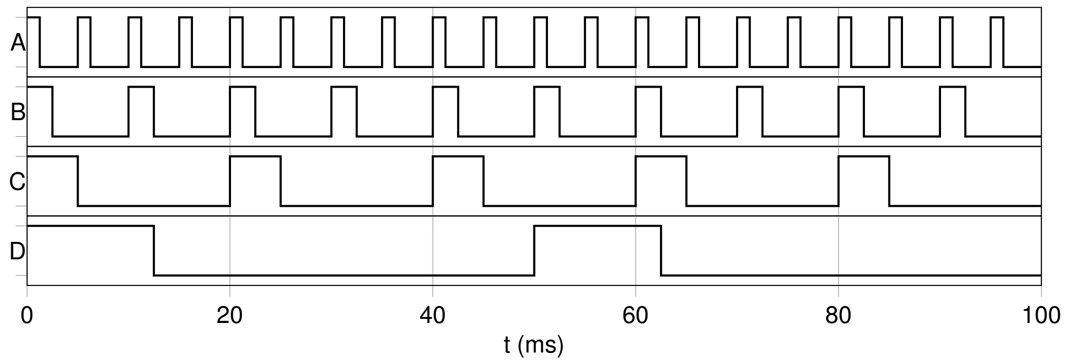


Figure 14.9: Pulse-width modulation waveforms with various frequencies. All have a duty cycle of 25%.

Fast PWM

In fast PWM mode, the timer / counter counts from `BOTTOM` to `TOP`, and then restarts again from `BOTTOM`. The value of `TOP` is either `MAX`

(0xFF or 0xFFFF) or the value given in register OCRnA. Timer / counter 1 allows 8-bit, 9-bit, 10-bit or 16-bit resolution in addition to using OCR1A or ICR1 as values of TOP. If the timer / counter reaches MAX, the overflow flag (TOVn) will be set. If OCRnA is used to determine the value of TOP, then the output compare flag (OCFnA) will be set when the contents of the timer/counter is equal to TOP. In these cases, the overflow flag will not be set; overflow interrupts therefore cannot be used when TOP is not equal to MAX.

The period of a fast PWM signal is

$$T = N \times (\texttt{TOP} + 1) \times T_{I/O},$$

where N is the value of the prescaler being used. Fast PWM has the advantage of having up to double the frequency of phase-correct PWM, but has the drawback of having only half the resolution.

An example fast PWM waveform is shown in Figure 14.10.

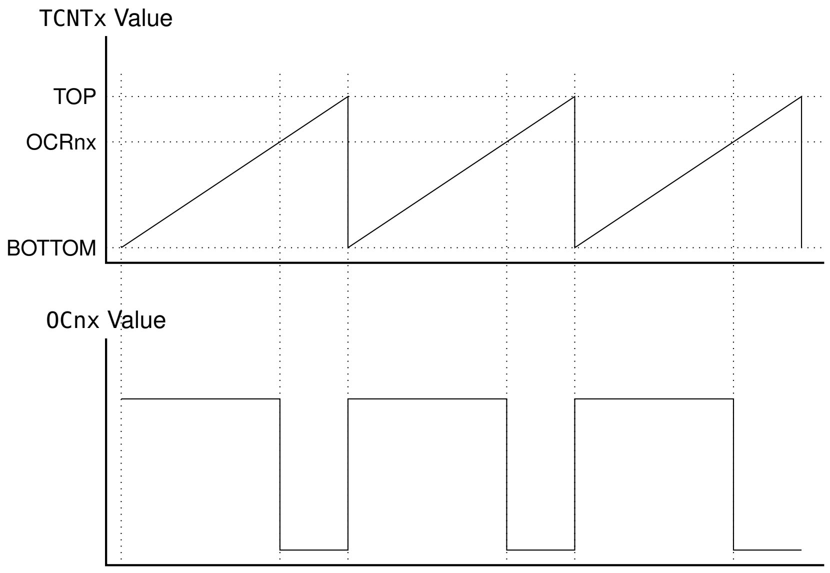


Figure 14.10: Example of a fast PWM waveform.

The frequency of a fast PWM waveform is changed in a "normal mode" fashion by using a prescaler and MAX, based on the equation

$$f = f_{I/O} \;/\; (N \times 2^n),$$

where n is the resolution of the timer/counter and N is the prescaler. In this manner, PWM can be used on both pins OCnA and OCnB with independent duty cycles, given by the equation

$$D = \texttt{OCRnx} \;/\; \texttt{MAX}.$$

Alternatively, more flexibility in the PWM waveform can be obtained by altering the value of TOP along with the prescaler, based on the equation

$$f = f_{I/O} \;/\; (N \times (\mathtt{TOP} + 1)).$$

In this case, PWM can be used only on pin `OCnB`, with a duty cycle given by the equation

$$D = \mathtt{OCRnB} \;/\; \mathtt{OCRnA}.$$

Phase-Correct PWM

In phase-correct PWM mode, the timer / counter increments from `BOTTOM` to `TOP`, and then decrements back to `BOTTOM`. The value of `TOP` is either `MAX` (`0xFF` or `0xFFFF`) or the value given in register `OCRnA`. Timer / counter 1 allows 8-bit, 9-bit, 10-bit or 16-bit resolution in addition to using `OCR1A` or `ICR1` as values of `TOP`. When the timer / counter is equal to `BOTTOM`, the overflow flag (`TOVn`) will be set. If `OCRnA` is used to determine the value of `TOP`, then the output compare flag (`OCFnA`) will be set when the contents of the timer/counter is equal to the value stored in `OCRnA`.

The period of a phase-correct PWM signal is

$$T = 2 \times N \times \mathtt{TOP} \times T_{I/O},$$

where N is the value of the prescaler being used. Phase-correct PWM has the advantage of having up to double the resolution of fast PWM, but has the drawback of having only half the frequency. The symmetric shape of the output waveform makes phase-correct PWM ideally suited for motors. Fast PWM, having an asymmetric output response, can possibly lead to glitches in the output if the frequency or duty cycle are changed abruptly.

An example phase-correct PWM waveform is shown in Figure 14.11.

The frequency of a phase-correct PWM waveform is changed in a "normal mode" fashion by using a prescaler and `MAX`, based on the equation

$$f = f_{I/O} \;/\; (2 \times N \times 2^n),$$

where n is the resolution of the timer/counter and N is the prescaler. In this manner, PWM can be used on both pins `OCnA` and `OCnB` with independent duty cycles, given by the equation

$$D = \mathtt{OCRnx} \;/\; \mathtt{MAX}.$$

Alternatively, more flexibility in the PWM waveform can be obtained by altering the value of `TOP` along with the prescaler, based on the equation

$$f = f_{I/O} \;/\; (2 \times N \times \mathtt{TOP}).$$

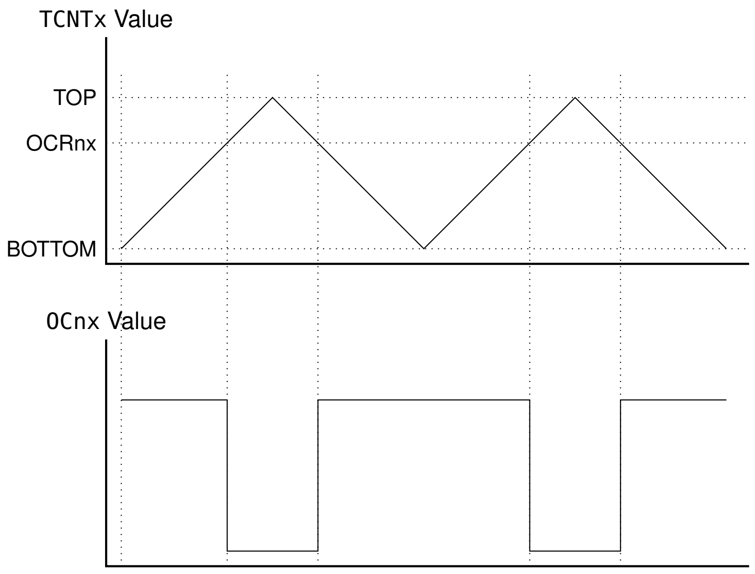


Figure 14.11: Example of a phase-correct PWM waveform.

In this case, PWM can be used only on pin `OCnB`, with a duty cycle given by the equation

$$D = \texttt{OCRnB / OCRnA}.$$

Phase- and Frequency-Correct PWM

Phase- and frequency-correct PWM is much the same as phase-correct PWM, except that the output periods are all symmetrical. This modality is only available on timer / counter 1 on the ATmega328P. It is to be used when the PWM frequency needs to be changed throughout the operation of the system. Two registers, `OCR1A` and `ICR1` may be used to set the value of `TOP`. The period of a phase- and frequency-correct PWM signal is the same of that given for a phase-correct PWM waveform.

14.5 Watchdog Timer (WDT)

The watchdog timer (WDT) is a separate timer on AVR microcontrollers that allow for system resets if the program is unresponsive after a certain period of time. If the WDT isn't refreshed within this specified time period, the device will automatically reset. In system reset mode, the device automatically resets if the WDT hasn't been refreshed. In interrupt mode, the WDT triggers an interrupt. This can be used to wake up from a sleep mode, or can also be used as a general system timer. In interrupt and system reset mode, an interrupt is first triggered before device reset. This allows for a safe shutdown by saving important parameters in the interrupt before reset.

14.6 Practice Problems

Assume a clock frequency of 16 MHz to answer all of the following questions.

1. How many timer / counters are available on the ATmega328P?

3

2. True or false: Timer / counter 0 is a 16-bit counter.

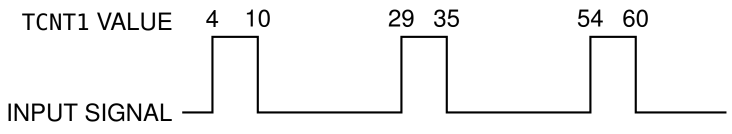
FALSE

3. On timer / counter 0 and timer / counter 2, in normal mode, when the counter rolls over, it goes from ______ to ______.

`0xFF` to `0x00`

4. What is the period and duty cycle of the following waveform, given $T_{I/O} = 1\ \mu s$?

$T = 25\ \mu s$, $D = 24\%$



5. In normal mode, what is the longest possible delay that can be obtained using timer / counter 0 or timer / counter 2? With what prescaler value is this achieved?

$T_{MAX} = 16.38$ ms, N = 1024

6. In normal mode, what is the longest possible delay that can be obtained using timer / counter 1? With what prescaler value is this achieved?

$T_{MAX} = 4.19$ s, N = 1024

7. In normal mode, what delay results with timer / counter 1 and a prescaler N = 8?

32.768 ms

8. In normal mode, what prescaler must be used with timer / counter 2 to achieve a delay of 0.512 ms? Is this possible with timer / counter 0? Why or why not?

32, Not possible with TCNT0 because it does not have a prescaler option of 32.

9. In CTC mode, what delay is obtained when `OCR1A` = 1499 and the prescaler N = 64?

6 ms

10. In CTC mode using timer / counter 2 and a prescaler N = 32, what value must be loaded into `OCR2A` to obtain a delay of 0.38 ms?

189

11. In CTC mode, what prescaler value must be used if `OCR0A` = 140 to obtain a delay of ≈9 ms?

1024

15 Serial Communication

Serial communication protocols allow data to be sent with far fewer wires than with parallel I/O. The schematic difference between serial and parallel communication protocols is shown in Figure 15.1.

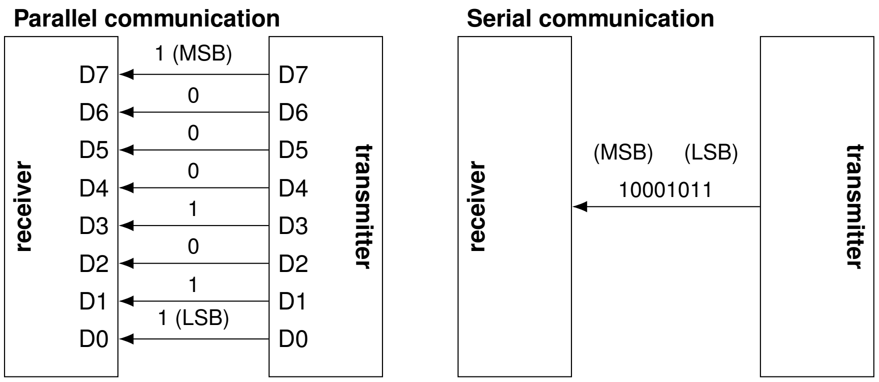


Figure 15.1: Schematic difference between parallel I/O (which uses many wires) and serial I/O (which can use as few as one wire).

While parallel data transmission allows many bits to be sent simultaneously, it requires as many wires as bits. This can be prohibitive in a microcontroller as the number of I/O ports is quite limited. In addition, parallel I/O can suffer from noise (crosstalk) and signal reflections if the source and destination are not close. Long distance synchronization can be difficult to achieve if there is significant delay in transmitting from one device to another.

Serial data is sent one bit at a time. Few wires are used, which minimizes crosstalk, meaning that it can be used over long distances. It is also cheaper to implement. There are many features of serial communication that may change the configuration and number of wires required to implement the communication protocol.

15.1 Simplex & Duplex

Simplex communication is capable of sending data in only one direction. Consider for example a radio. Information is broadcast to the radio receiver, but no information is transmitted back. It is simply a unidirectional flow of data. A duplex communication system consists of two devices that can communicate with each other in both directions (i.e. as receivers and as transmitters).

In **half-duplex** communication, only one wire is present to send and receive data, therefore information can be sent in both directions, but only one direction at a time. Half-duplex communication can be considered similar to speaking on a walkie-talkie; one individual speaks at a time and indicates the end of their message by saying "over."

In **full-duplex** communication, information can be sent in both directions simultaneously, which requires the use of one more wire than half-duplex communication. Full-duplex communication is possible using phones where two people can speak simultaneously (although limitations of humans may make it difficult for both parties to understand; this complication is not present in computing devices).

15.2 Architecture

Many serial protocols use a primary-secondary configuration. The primary device has unidirectional control over other devices (secondaries). In some serial communication protocols, the primary selects the active secondary device, and also supplies the clock signal. Some serial protocols allow multiple primary devices. In addition, some protocols allow the roles of primary and secondary to be changed between message transmissions.

15.3 Data Transfer Protocol

In serial communication, data can be sent synchronously or asynchronously. **Synchronous** communication requires a clock signal to be provided by the primary (which requires the existence of another wire to carry the clock signal). With synchronous communication, once (8-bit) data transfer is initiated, the receiver has only to wait 8 clock cycles to obtain the information.

In **asynchronous** communication, `START` and `STOP` bits are required to signal to the receiver that transfer has been initiated and completed. Certain serial protocols allow the user to specify whether data should be sent LSB-first or MSB-first. In asynchronous communication, `START` and `STOP` bits are required to specify when data communication has commenced or completed. In addition, data rate, electrical signal definition for HIGH and LOW, and handshaking protocols may also need to be defined in each serial protocol.

Serial communication protocols supported by the ATmega328P include USART (universal synchronous / asynchronous receiver / transmitter), SPI (serial peripheral interface), and TWI (two-wire interface).

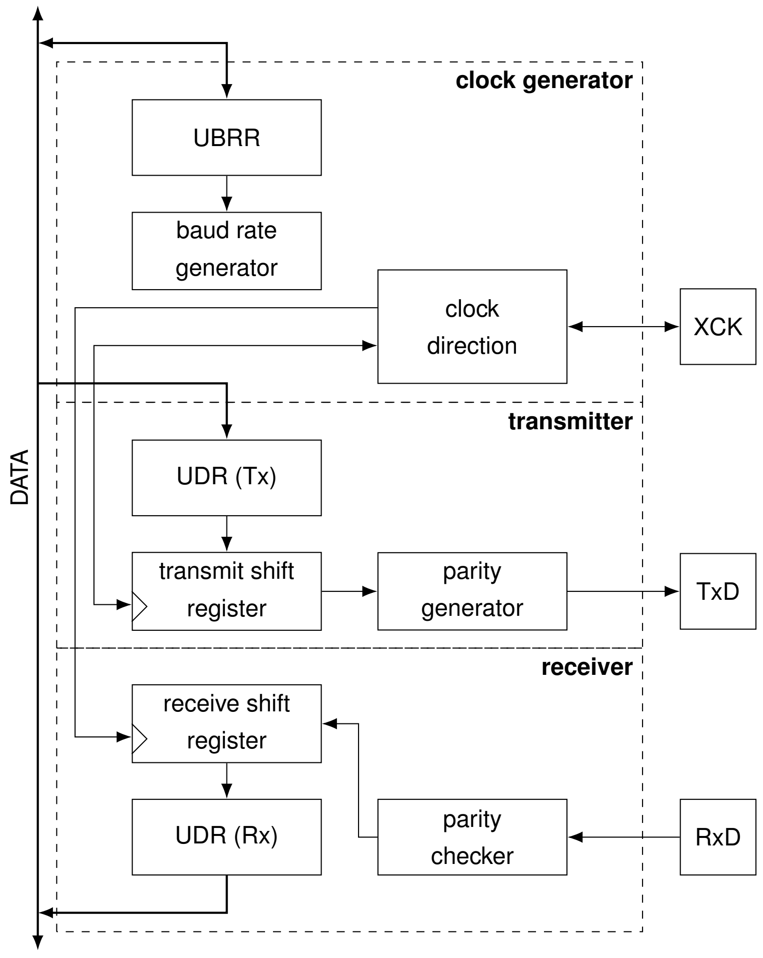


Figure 15.2: USART block diagram.

15.4 *Universal Synchronous / Asynchronous Receiver / Transmitter (USART)*

USART is commonly included in microcontrollers. The ATmega328P USART is capable of operating either asynchronously or synchronously. In asynchronous mode, two I/O pins are used, one to transmit data (`TXD`), and one to receive data (`RXD`). This indicates that USART is a full-duplex system. In synchronous mode, an additional pin is required for the clock signal (XCK). A block diagram of USART features and functionality is given in Figure 15.2.

The USART on the ATmega328P is capable of sending data in chunks of 5, 6, 7, 8, or 9 bits with one or two STOP bits. It is capable of odd and even parity generation and checking. In addition, it is capable of detecting data overrun and framing errors. When enabled, interrupts can be generated upon completion of transmission or receiving data, or when the transfer data register is empty. The baud rate can be set using the USART baud rate register (UBRR), and generates baud rates of

$$\text{BAUD} = f_{OSC} \div [16 \times (\texttt{UBRR} + 1)]$$

in asynchronous normal mode,

$$\text{BAUD} = f_{OSC} \div [8 \times (\texttt{UBRR} + 1)]$$

in asynchronous double-speed mode, and

$$\text{BAUD} = f_{OSC} \div [2 \times (\texttt{UBRR} + 1)]$$

in synchronous primary mode.

Receiving Data

The USART checks the received signal at every clock cycle. In asynchronous communication (in which the USART acts similarly to a shift register, which is capable of operating within multiple protocols such as RS-232 and RS-485), a baud rate is agreed upon by both devices beforehand. Otherwise, a clock signal is supplied. If the signal is LOW for a long enough time, it is registered as a START bit. (If, however, the signal was LOW for less than half of the bit rate, it is considered to be a spurious signal and ignored.) After receiving the START bit, the remaining character is clocked in to the receive shift register and into UDR to be sent to the data bus. A busy flag is set during this process to signal that the device is busy receiving data. The USART then signals that it has received new data, and may send an interrupt to the processor to take further action.

Transmitting Data

An 8-bit character is deposited into the transmit shift register from the data bus. At this point, the USART generates a START bit, which is sent, followed by the character to be transmitted. If requested, a parity bit is sent, followed by a STOP bit. During this process, a "busy" flag is set, signaling that the device is busy transmitting data. If enabled, an interrupt is generated once data has shifted out and no new data exists in the transmit buffer.

15.5 *Serial Peripheral Interface (SPI)*

SPI communication on the ATmega328P is capable of full-duplex synchronous communication using only four wires. The Arduino can be configured as either a primary or a secondary, and can send bytes either MSB first or LSB first.

The ATmega328P SPI bus uses four logic signals, given in Table 15.1. Each of the signals is associated with a particular pin on Port B. (Please note that there is an inconsistency between the pin names and the descriptions, see the author note at the start of this book for more information.)

Table 15.1: SPI defined logic signals.

Name	Pin	Description
`SCK`	`D13`	Serial clock (output from primary)
`MOSI`	`D11`	Primary output, secondary input
`MISO`	`D12`	Primary input, secondary output
$\overline{\texttt{SS}}$	`D10`	Secondary select (active low, output from primary)

The primary and secondary devices are connected as shown in Figure 15.3. As can be seen, shift registers exist on both the primary and secondary devices. As a data byte is transmitted from the primary to the secondary along the `MOSI` line, the data that was previously stored in the secondary's shift register transmits from the secondary to the primary along the `MISO` line.

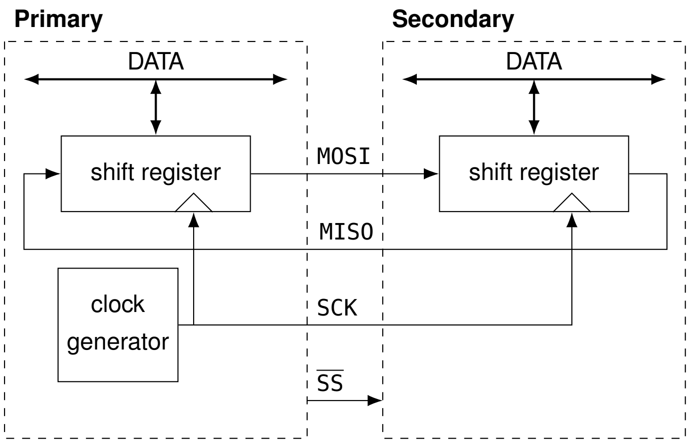


Figure 15.3: SPI primary-secondary connection diagram.

SPI Primary-Secondary Configuration

Multiple secondary devices can be supported with SPI. This can be accomplished independently as shown in Figure 15.4. Independent secondary devices all require their own secondary select signal. Therefore using many independent secondary devices can require a a large number of I/O pins.

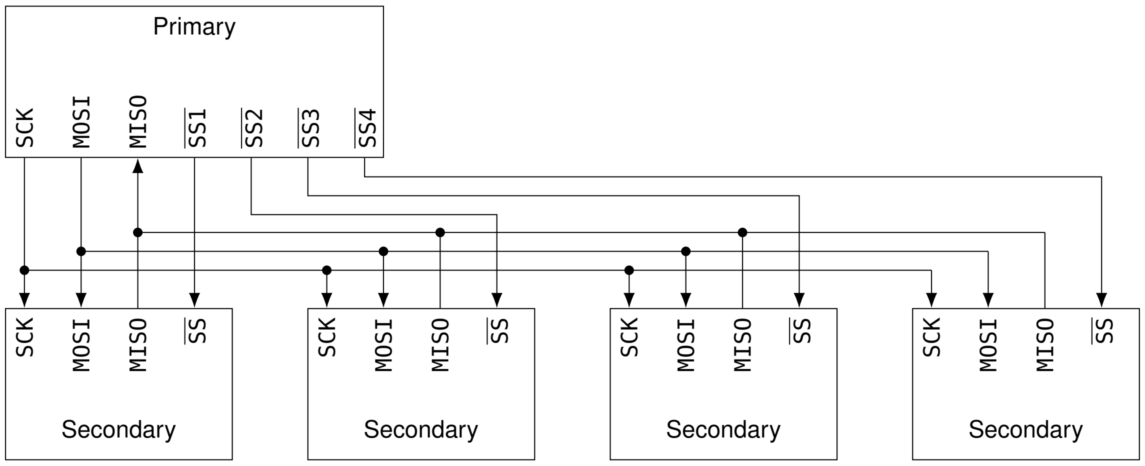


Figure 15.4: Independent secondary configuration in SPI.

Multiple secondary devices can also be connected in a daisy-chained configuration as shown in Figure 15.5. Daisy-chained secondary devices have the output of one secondary feeding into the input of the next, sharing a common secondary select signal.

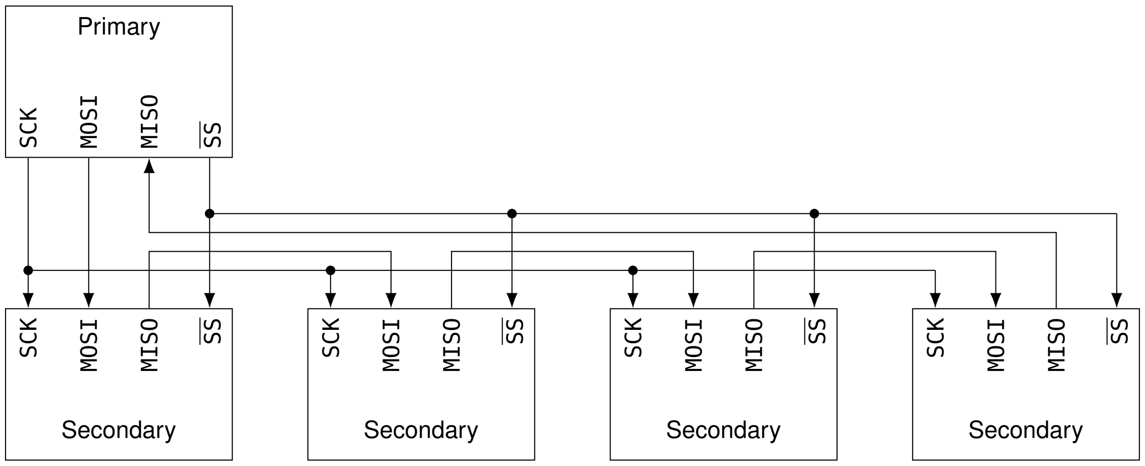


Figure 15.5: Daisy-chained secondary configuration in SPI.

SPI Clock Polarity & Phase

Whereas in USART communication, the receiver and transmitter must agree upon a bit rate before commencing communication, in SPI the primary and secondary must agree on a clock polarity and phase. The clock polarity indicates the idle value of the clock signal. When idle, the `SCK` will be kept LOW if the polarity bit (known as

CPOL) in the SPI control register is 0. Otherwise, if CPOL is 1, the clock signal will be kept HIGH when idle.

The clock phase, configured by the CPHA bit in the SPI control register, indicates whether sampling will occur on leading edges (when CPOL is 0, the leading edge is a rising edge, otherwise it is a falling edge) or trailing edges (when CPOL is 0, the trailing edge is a falling edge) of the clock signal. Together with CPOL, this determines if data will be sampled on rising or falling clock ticks.

Advantages & Disadvantages of SPI

The SPI protocol has a number of advantages and disadvantages. It is capable of full-duplex communication, is not limited in the number of bits that can be transmitted in each message, uses lower power than protocols like TWI, is not limited to the maximum clock speed, has very simple hardware interfacing, and has simple software implementation.

However, the disadvantages is that SPI does require more pins than other protocols (notably TWI), does not have hardware for receipt acknowledgement from secondary devices (i.e. the primary could be transmitting to nowhere and not know it), does not have any error checking, and can only work over short distances.

15.6 Two-Wire Interface (TWI)

Two-wire interface (TWI), also referred to as inter-integrated circuit (I^2C), is a serial protocol used to connect with one or more secondary devices (connected as shown in Figure 15.6) using only two wires. One wire is a bidirectional data line called SDA, which indicates that TWI is capable of half-duplex communication. The other wire, SCL (also bidirectional), carries the clock signal, indicating that TWI is a synchronous communication system. Both of these pins reside in Port C on the ATmega328P.

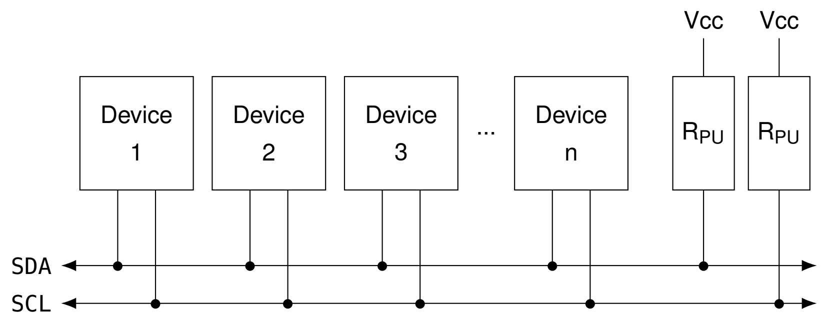


Figure 15.6: Device configuration using the TWI protocol.

TWI Module

The TWI hardware consists of several modules that work together to ensure efficient and accurate operation of the communication system. This overview is depicted schematically in Figure 15.7. The two pins `SCL` and `SDA` interface the microcontroller with external devices. The **bus interface unit** contains four units. `TWDR`, the data and address shift register, which contains the address or data bytes to be transmitted, or the address or data bytes that have been received. The `START/STOP` controller generates and detects `START` and `STOP` bits used for framing the received and transmitted data. Spike suppression filters out short bursts of data that may otherwise interfere with serial communication. Finally, the arbitration detection unit monitors communications to ensure that only one primary is communicating at a time in a multi-primary system.

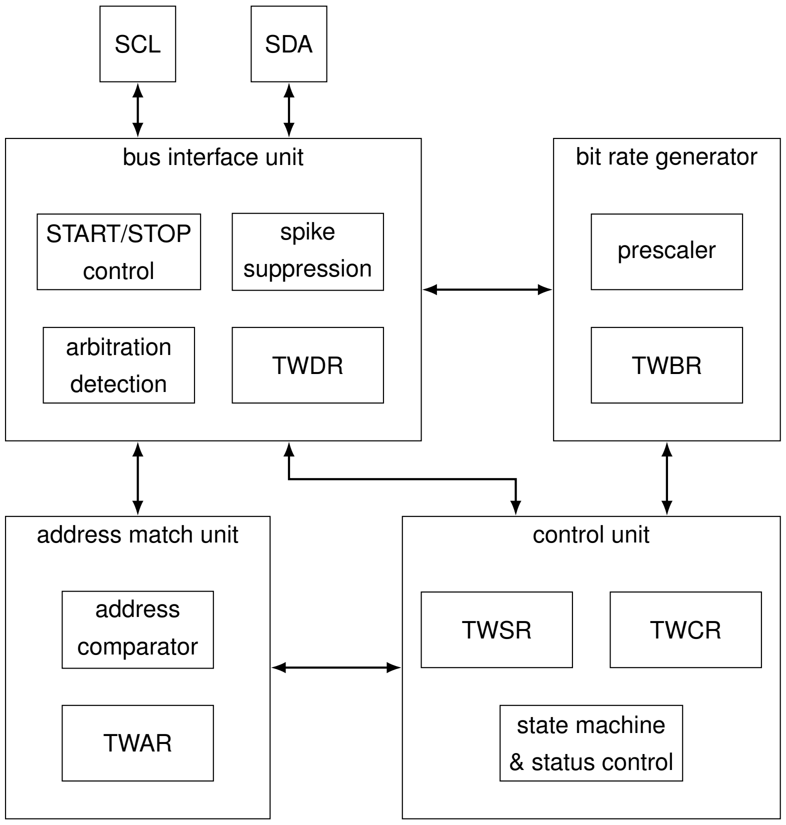


Figure 15.7: Overview schematic of the TWI module on the ATmega328P.

The **bit rate generator** controls the period of the clock signal on `SCL` when the device is operating in primary mode. The bit rate is stored in a register known as `TWBR`, and together with a prescaler bit they control the period given by the equation

$$T_{\mathrm{SCL}} = \mathrm{T\_CPU} \times [16 + (2 \times \mathtt{TWBR} \times N)],$$

where N is the value of the prescaler.

The **address match unit** checks to ensure that received address bytes match the 7-bit address given in the address register `TWAR`.

Finally, the **control unit** monitors the TWI bus and generates signals in response to the settings that have been detected by all of the other units. `TWSR` is the TWI status register, which contains data corresponding to the status of the most recently executed communication operation on the bus. The TWI control register, `TWCR`, contains the bus specification and settings.

TWI Communication Process

While TWI has the advantage of using fewer wires than SPI communication, the drawback is in the complexity of using the TWI protocol. Communication begins when the primary transmits a `START` bit followed by the 7-bit address of the secondary with which it intends to communicate and a bit indicating a read or write operation. (To be assigned a unique secondary address for a TWI-compatible device, a fee must be paid to NXP Semiconductors.) The corresponding secondary device (if it is properly connected to the interface bus) then sends an acknowledge bit, which the primary can use to ensure that it is indeed communicating with the correct device. The primary then continues to send a clock signal while either transmitting messages to the secondary or receiving data from the secondary. A `STOP` bit is transmitted when communication with the secondary is complete.

TWI Status Codes

Multiple status codes exist for all TWI communication modes (primary transmit, primary receive, secondary transmit, and secondary receive). These codes give information about the status of the bus and connected hardware. They can be used to generate error messages, or to stop program flow if proper handshaking and acknowledgement between primary and secondary has not been successful.

15.7 Serial Communication Errors

There are several transmission errors that can occur in serial communication. In an **overrun error**, all of the receiving buffer registers and shift registers are full but haven't yet been read by the CPU, when a new `START` bit is detected. In other words, data is being transmitted faster than it can be received.

In a **framing error**, a received character is improperly framed by `START` and `STOP` bits. This indicates a synchronization problem, faulty transmission, or a BREAK condition (BREAK = LOW for more than the duration of an entire character).

A **parity error** indicates that an odd number of bits have changed value during the transmission, indicating noise or other transmission issues. It can be detected by a parity detector circuit.

Parity Error Detection

Parity refers to the number of 1's in a variable or data stream. Parity is ODD if there is an odd number of 1's, and EVEN if there is an even number of 1's. In order to create a data stream with a particular parity (which is decided upon by both the transmitter and receiver beforehand), a parity bit must be generated. Parity checking is a simple way to check for transmission errors, although it is not foolproof (it is capable of producing false positives, but not false negatives). It is simple to implement, requiring only XOR or XNOR gates, and only requires a single bit added to a signal. However it is only able to detect an error if an odd number of bits are corrupted during transmission.

Even Parity: Success
Person A wants to send `1011`
Compute parity: $P = 1 \oplus 0 \oplus 1 \oplus 1 = 1$
Send signal `10111` (the last bit is P, the parity bit)
Person B receives `10111`
Compute parity: $1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$ = even
Transmission assumed successful
Even Parity: Failure
Person A wants to send `1011`
Compute parity: $P = 1 \oplus 0 \oplus 1 \oplus 1 = 1$
Send signal `10111` (the last bit is P, the parity bit)
Person B receives `10011`
Compute parity: $1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1$ = odd
Transmission was not successful

Table 15.2: Example of error detection using parity-checking.

15.8 Practice Problems

1. Is the USART capable of full-duplex, half-duplex, or simplex communication?

full-duplex

2. Does the USART operate synchronously or asynchronously?

it can operate either synchronously or asynchronously

3. True or false: the SPI bus requires an external clock to operate. FALSE

4. The secondary select line on the SPI bus is active ______. LOW

5. Using the SPI bus, data is received and transmitted in chunks of how many bits? 8

6. How many specific secondary devices can be supported on the TWI bus? $2^7 = 128$

7. True or false: the TWI bus requires an external clock to operate. FALSE

8. Calculate the parity of the data byte `1101 1001`. odd

9. Calculate the parity of the data byte `0101 0011`. even

10. Generate an odd parity bit for the data byte `1101 0011`. $P = 0$

11. Generate an even parity bit for the data byte `1101 0011`. $P = 1$

16
Power Management & Sleep Modes

Embedded systems frequently run on batteries. In system designs for applications when it is monetarily or physically difficult to frequently change batteries, it is important to design the project to consume as little power as possible. The ATmega328P is built from CMOS technology; CMOS devices use no power when they are not switching. There are sleep modes available that reduce power consumption by turning off the clock signal to peripheral units. In addition, a power reduction register (PRR) can also be used to disable modules (by turning off the associated clock signal) that are not needed.

16.1 Sleep Modes

There are six sleep modes available on the ATmega328P, as shown in Table 16.1.

Name	**clk_{CPU}**	**$\text{clk}_{\text{FLASH}}$**	**$\text{clk}_{\text{I/O}}$**	**clk_{ADC}**	**clk_{ASY}**
Idle			×	×	×
ADC Noise Reduction				×	×
Power-down					
Power-save					×
Standby					
Extended Standby					×

Table 16.1: Sleep modes on the ATmega328P, indicating the active peripheral clocks.

The **idle mode** stops the CPU clock but allows the SPI, USART, analog comparator, ADC, TWI, timer / counters, watchdog timer, and system interrupts to continue operating.

ADC noise reduction mode stops the CPU clock, I/O clock, and flash clock. The peripherals that can continue to run under this restriction are the ADC, external interrupts, TWI, timer / counter 2 (clocked asynchronously), and the watchdog timer. As is indicated from the name, shutting down superfluous clock signals leads to lower noise in the ADC, which leads to more accurate results.

Power-down mode shuts off all generated clocks, allowing only asynchronous modules to run. **Power-save mode** is virtually identical, except that timer / counter 2, if enabled and clocked asynchronously, will continue running during sleep.

When clocked from an external source, **standby mode** can be used; it is identical to power-down mode except that the external oscillator is kept running. Similarly, **extended standby mode** is identical to power-save except that an external source is used to clock the system and the eternal clock will continue running during sleep.

16.2 Wake-Up Sources

It is important for the microcontroller to wake up from sleep modes in order to continue with normal operations as needed. The wake-up sources differ depending on the sleep mode that is used. These sources are listed in Table 16.2.

Table 16.2: Wake-up sources for each of the sleep modes on the ATmega328P.

Idle	external & pin-change interrupts, TWI Address Match, timer / counter 2, SPM/EEPROM ready, ADC, WDT, other I/O
ADC Noise Reduction	external & pin-change interrupts, TWI Address Match, timer / counter 2, SPM/EEPROM ready, ADC, WDT
Power-down	external & pin-change interrupts, TWI Address Match, WDT
Power-save	external & pin-change interrupts, TWI Address Match, timer / counter 2, WDT
Standby	external & pin-change interrupts, TWI Address Match, WDT
Extended Standby	external & pin-change interrupts, TWI Address Match, timer / counter 2, WDT

17
Control Systems & Feedback

When controlling a device, it is important to use some form of feedback to monitor the current status and see if changes need to be made.

Consider, for example, that you are sitting inside of a room with only a window. The window can only be opened and closed completely. If it gets too warm in the room, you open the window. When it gets too cold, you close the window. This leads to a temperature response as shown in Figure 17.1.

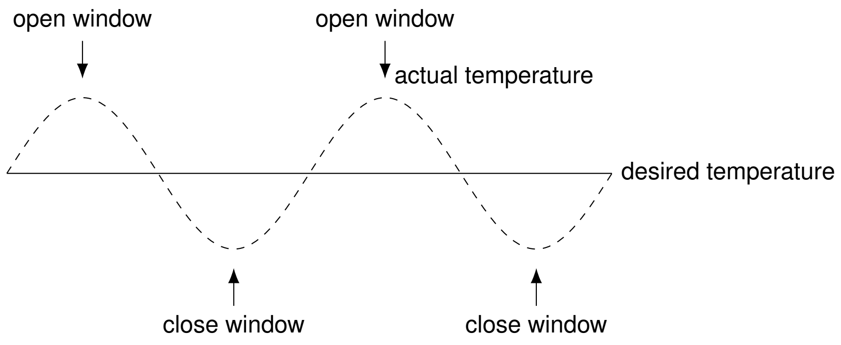


Figure 17.1: The temperature response in the room with a single window.

This is known as a negative feedback system. When you start to feel uncomfortable (negative feedback), you take an action that changes the surroundings to better suit your needs. However, in this situation, not only does the temperature fluctuate between extremes without ever settling on an ideal temperature, it also requires you to continue moving over to the window to open and close it. It would be better to have an automated system to take care of this functionality.

In a **closed-loop negative feedback** system, a setpoint is required. This is the value at which the system will attempt to keep the value of the output (temperature, in the case of the window example). When the measured output value is greater than the setpoint, action will be taken to attempt to decrease the output value. When the measured output value is less than the setpoint, action will be taken to attempt to increase the output value. Instead of turning a system on and off completely (or opening a window fully or closing it fully), which leads to oscillating behavior, it is necessary to use a proportional feedback mechanism to more finely tune the desired changes in output value.

For example, a home central air-conditioning unit, when initially switched on in the middle of summer, may have a setpoint of 20°C, but the surrounding temperature is 30°C. This signals to the AC to turn on at great intensity. When turned on in the evening, the surrounding temperature of 25°C signals to the AC to turn on but without as much intensity as before. This is known as proportional feedback because the amount of action requested at the output is proportional to the difference (hence the term negative feedback) between the setpoint value r(t) and the actual value y(t) (known as the process variable). This difference is also known as the error e(t).

$$e(t) = r(t) - y(t)$$

17.1 *Proportional Feedback Control*

In proportional feedback control, the error is multiplied by a constant known as K_P. The new value of our output u(t) becomes

$$u(t) = y(t) + K_P\, e(t),$$

where y(t) is the value of the process variable (i.e. what the value is right now). This is depicted schematically in Figure 17.2.

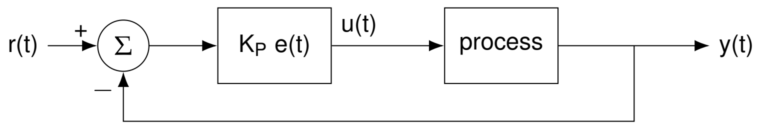


Figure 17.2: Block diagram of proportional control.

The constant K_P is chosen with careful consideration. Values that are too low lead to sluggish, unresponsive feedback (this is known as an overdamped system). Values that are too high become unstable and can oscillate rapidly between values (this is known as an underdamped system). In the worst case, an underdamped system won't ever settle down to a proper value. Overdamped (red curve corresponding to $K_P = 0.1$) and underdamped (blue curve corresponding to $K_P = 1.8$) conditions are shown in Figure 17.3. When K_P is just right, the output is stable and has no oscillations. This is known as a **critically damped** system and is the desired outcome of most control systems.

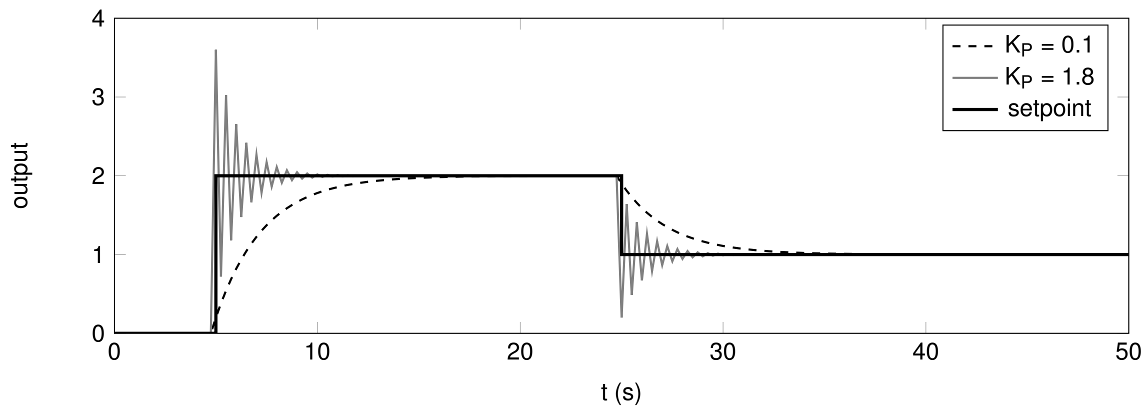


Figure 17.3: Proportional control output. The setpoint value is the thick black curve. The outputs are shown for a small value of K_P (dashed line, overdamped system) and a large value of K_P (gray solid line, underdamped system).

17.2 *Proportional-Integral (PI) Control*

In addition to the proportionality constant K_P that considers the current amount of error in the system, an integral term K_I is included that takes into account past values of the error.

$$\text{past error} = \int_0^t e(\tau)d\tau$$
$$u(t) = y(t) + K_P\, e(t) + K_I \int_0^t e(\tau)d\tau$$

This is depicted schematically in Figure 17.4.

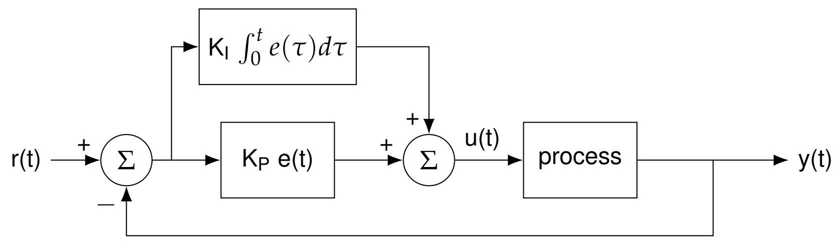


Figure 17.4: Block diagram of proportional-integral control.

This integral constant (which must be less than K_P) helps the feedback system achieve a steady-state value much quicker, based on the fact that it helps compensate for errors that have not yet been cleaned up by the linear proportionality constant K_P. However, a careful value of K_I still needs to be chosen, due to the fact that a poorly chosen value can still lead to an overdamped (K_I too low) or underdamped (K_I too high) output.

Figure 17.5 shows a PI control system with K_P = 0.1 that would ordinarily lead to a highly overdamped system which responds very slowly to change (indicated by the red curve). By adding integral control to the system, it is able to respond much quicker to change, with the drawback that it introduces overshoot.

An integral constant can be necessary to eliminate **steady-state error**, which refers to error in a system that persists even when the system has reached a stable state.

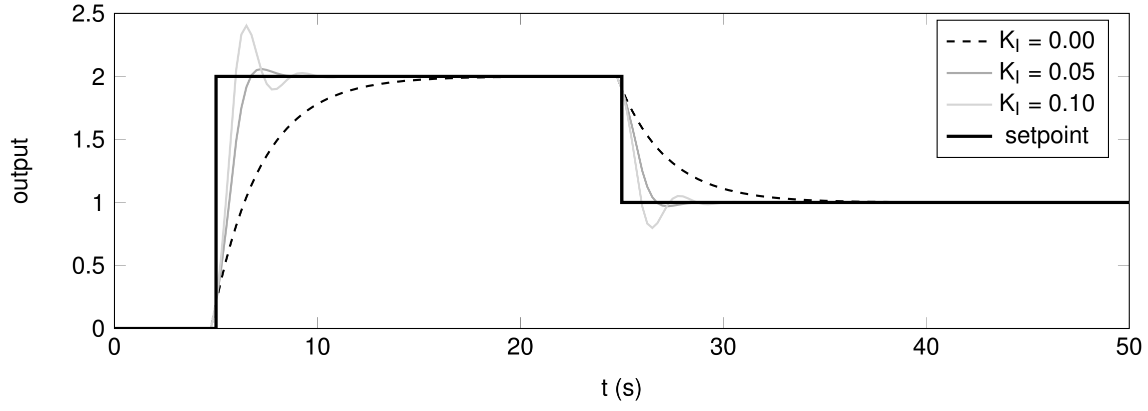


Figure 17.5: Proportional-integral control output. The setpoint value is the thick black curve. All outputs have K_P = 0.1 The dashed curve has no integral constant, the dark gray curve has K_I = 0.05, and the light gray curve has K_I = 0.10.

17.3 *Proportional-Integral-Derivative (PID) Control*

A full proportional-integral-derivative (PID) feedback system takes into account also anticipated future values of the error based on taking a derivative of the current error.

$$\text{future error} = \frac{de(t)}{dt}$$

$$u(t) = y(t) + K_P\, e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt}$$

This is depicted schematically in Figure 17.6.

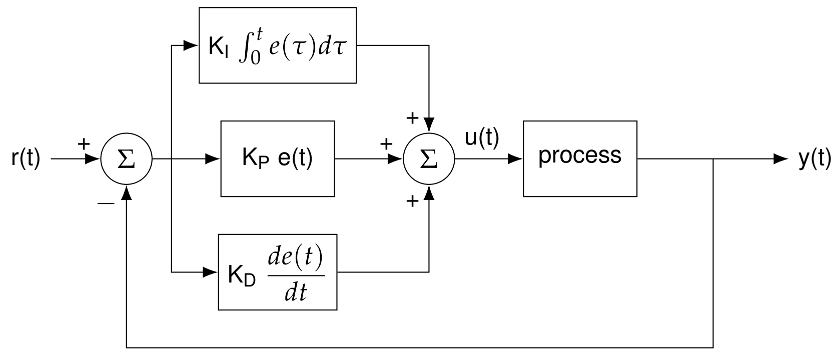


Figure 17.6: Block diagram of proportional-integral-derivative control.

Adding a derivative proportionality constant K_D may make the feedback mechanism much more sensitive to noise, and is only recommended in cases where noise will be minimum.

Example data with proportional control only, PI control, and PID control is shown in Figure 17.7.

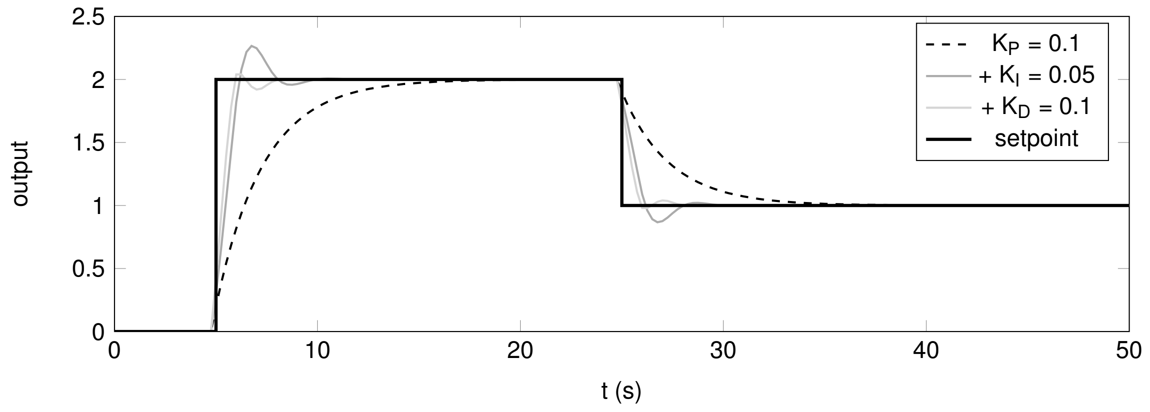


Figure 17.7: Proportional-integral-derivative control output. The setpoint value is the thick black curve. The dashed curve corresponds to only a proportional term with $K_P = 0.1$. The dark gray curve adds an integral term of $K_I = 0.05$. The light gray curve adds a derivative term of $K_D = 0.10$.

Values for each of the constants must be chosen specifically for each application in which they are to be used. Changing any other part of the process may also require a change in the constants.

18
C Concepts for Microcontrollers

The C programming language is an invaluable tool for programming microcontrollers using a high-level language. As discussed in chapter 4, writing C programs for embedded systems is quite different from writing programs for general computing applications. This chapter is not intended to serve as an exhaustive reference for C; it will merely outline the most important concepts in programming C for microcontroller applications.

18.1 Standard Datatypes

All variables in C must be declared before usage. This declaration includes the datatype that the variable takes on, which indicates how much memory the microcontroller must use to store the variable and the minimum and maximum values it can take on. The list of datatypes given in Table 18.1 is valid for the Arduino Uno, and may differ somewhat from other Arduino microcontrollers or compilers other than that of the Arduino IDE.

Table 18.1: C datatypes used on the Arduino Uno using the Arduino IDE.

Datatype	Memory	Data Range
Integer Datatypes		
`char`	8-bits	-128 to 127
`unsigned char`	8-bits	0 to 255
`int`	16-bits	$-32{,}768$ to 32,767
`unsigned int`	16-bits	0 to 65,535
`long`	32-bits	$-2{,}147{,}483{,}648$ to 2,147,483,647
`unsigned long`	32-bits	0 to 4,294,967,295
Floating-Point Datatypes		
`float`	32-bits	-3.4×10^{38} to 3.4×10^{38}
`double`	32-bits	-3.4×10^{38} to 3.4×10^{38}

Integer datatypes are used to represent whole numbers. **Floating-point** datatypes are capable of representing fractional numbers as well as numbers which are too large or too small to fit into the constraints of the integer datatypes. The two types used on the Arduino Uno with the Arduino IDE, `float` and `double`, are both single-precision type floating-point numbers. These numbers have a single sign bit, 8 exponent bits, and 23 fractional bits. The number they represent is

$$(-1^{\mathtt{s}}) \times (1.\mathtt{f}) \times 2^{(\mathtt{e}-127)},$$

where `s` is the sign bit, `f` is the fractional number, and `e` is the exponent component. There are a couple of downsides to working with floating-point numbers (apart from requiring 4 bytes of memory for each variable). First, floating-point arithmetic is very slow. In addition, floats only have 6–7 decimal digits of precision. That means the total number of digits, not the number to the right of the decimal point. Finally, floating-point numbers are not exact, and may yield strange results when compared. For example 6.0 / 3.0 may not equal 2.0.

Practice Problems

1. Indicate the datatype you would use for the following variables...

   (a) ...temperature (in either degrees C or F).

   `int`

   (b) ...the number of days in a week.

   `unsigned char`

   (c) ...the number of days in a year.

   `unsigned int`

(d) ...the number of months in a year. `unsigned char`

(e) ...an address of 64K RAM space. `unsigned int`

(f) ...the age of a person in years. `unsigned char`

18.2 Variable Scope & Keywords

The `scope` of a variable refers to what functions can access the variable. A variable defined within a function can only be accessed by that function. The following code

```
void setup() {
  unsigned char j = 15;
}

void loop() {
  // this function cannot access j
}
```

shows an example of variable scope. The variable `j` can only be accessed within the `setup()` function in which it is defined.

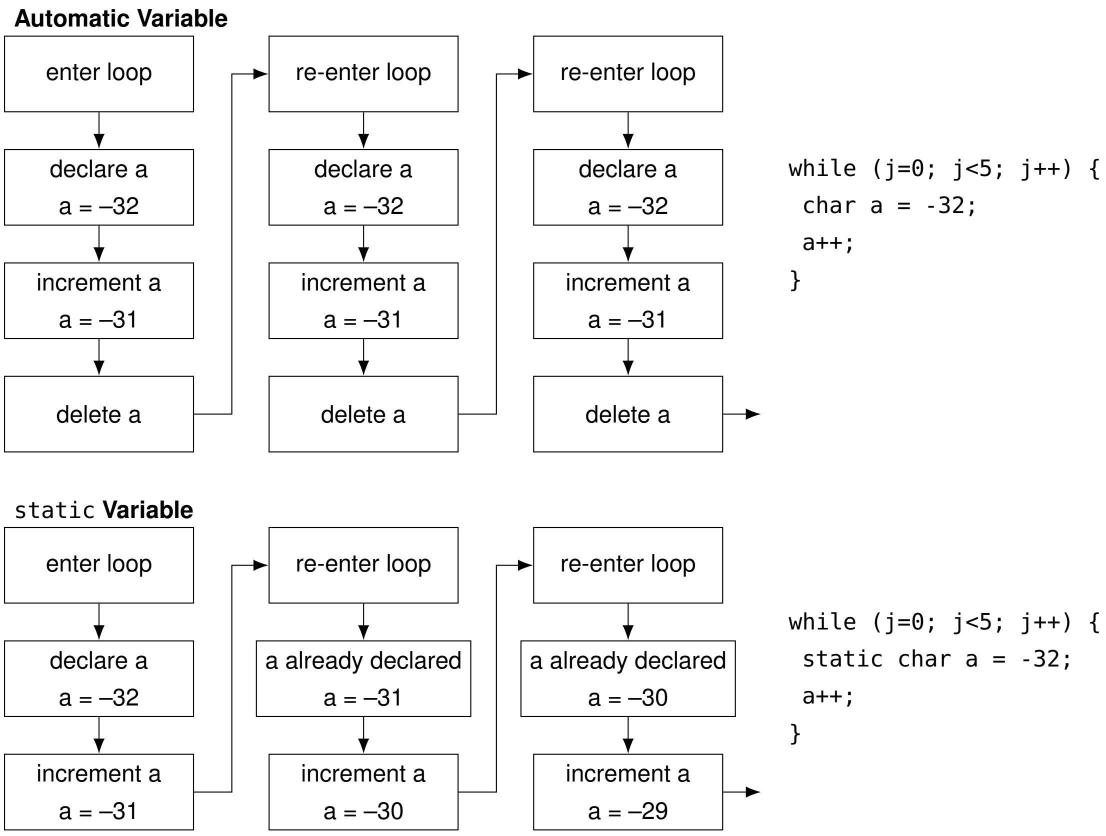
A variable that is defined outside of all functions can be accessed by every function in the code, and is referred to as a **global variable**. Global variables should only be used in situations where it is necessary; otherwise, it is recommended that the scope of variables be limited by keeping them within the function in which they are to be used.

`volatile` Variables

When a compiler takes C code and translates it into assembly language, it attempts to optimize that code by leaving out any unused variables and converting unchanging variables to constants, among other optimizations. At times, it may appear that a global variable is unused by functions, especially in the case of an interrupt service routine (ISR). An ISR is never formally invoked (or called) by the `void loop()` function in the Arduino IDE, and it may appear to the compiler as if any variables that are used within the ISR are unused (in which case it does not save them in memory) or unchanging (in which case it saves it in program memory as a constant value). By creating a `volatile` variable, the compiler knows not to discard the variable or to treat it as a constant. All datatypes can be saved as `volatile` variables.

`static` Variables

The keyword `static` in front of a variable refers to how long the variable is active in memory. Variables without this keyword are known as automatic, meaning that they come into existence when they are declared, and then expire whenever the function or loop in which they reside has finished running. A `static` variable exists in memory for as long as the program is running. This means that, even if they are declared with a certain value inside of a function or a loop, they can be changed for as long as the program runs. This allows for non-global variables that can change within a function or a loop. Consider the following examples, which show the difference between automatic and `static` variables. Figure 18.1 depicts the difference between `static` and automatic variables.



```
while (j=0; j<5; j++) {
 char a = -32;
 a++;
}
```

```
while (j=0; j<5; j++) {
 static char a = -32;
 a++;
}
```

Figure 18.1: A flowchart representation of the difference between automatic and `static` variables.

`const` *Variables*

The keyword `const` is used to denote a variable whose value will not change. The keyword can be used with all datatypes. This keyword is convenient to use with data arrays to define the number of elements to be stored into the array. Using a variable allows the number to be stored in a single location which can easily be changed while debugging the software code. Additionally, if an array is defined using a variable to denote the number of elements, that variable **must** be a `const` variable. Most compilers will give a warning if attempting to assign a new value to a `const` variable inside of C code.

18.3 Arrays

When variables are related to each other, it may be prudent to store the data in an array. An array, which must be initialized with the number of values to be given in the array, can store any type of variable (`char`, `int`, `long`, `float`). Each element in the array is defined by its index. Index numbers always go from 0 to (n−1), where n is the number of values in the array. The syntax for defining an array is

```
datatype arrayName[sizeofArray];
```

Consider the array `myArray[6]` consisting of six `unsigned char` variables shown in Table 18.2.

`unsigned char myArray[6] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6}`

index:	0	1	2	3	4	5
value:	`0xFC`	`0x60`	`0xDA`	`0xF2`	`0x66`	`0xB6`

Table 18.2: An array of six `unsigned char` variables.

To access an element of this array, a new variable can be assigned the value of one of the elements. For example, `unsigned char a = myArray[5]` will save the 6th element of the array into variable `a`. Note that `a` and `myArray[]` are of the same datatype. To save a new value into the array, an element inside of the array can be assigned a new value. For example, `myArray[1] = 13` saves the value of 13 into the 2nd element of the array `myArray[]`.

While arrays can be populated with any type of variable, **character arrays** using ASCII formatting (not to be confused with `char` arrays!) require a special precaution in that they must contain a null character (`\0`) at the end. Therefore, if they are assigned with the number of elements, that number must be n+1. Optionally, the number in square brackets can be left out.

Multi-Dimensional Arrays

Data can be stored into a multi-dimensional array. An array of integers, for example, `int a[n][m]` has `n` rows and `m` columns of data. This means that `n`×`m` elements can be stored into the array. Memory considerations must be taken into account before defining large arrays, which is especially true in multi-dimensional arrays, as memory can expand non-linearly with the addition of new elements. An example two-dimensional array is given in Table 18.3.

```
unsigned char wholeNums[2][5] = {
                 {1, 2, 3, 4, 5},
                 {6, 7, 8, 9, 10} };
```

n:	0	1	2	3	4
m = 0:	1	2	3	4	5
m = 1:	6	7	8	9	10

Table 18.3: An two-dimensional array consisting of ten `unsigned char` variables.

18.4 Bitwise Operations

While there are many types of operators used in programming (the assignment operator, arithmetic operators, logical operators, etc.), bitwise operations are used frequently in microcontroller programming. Bitwise means each bit in one variable is compared individually with the corresponding bit in the other variable. They are used to manipulate binary values, especially when only some values in a data byte need to be changed while leaving others alone. For this reason, they are used frequently when changing or accessing the values stored in I/O registers `DDRxn`, `PORTxn` and `PINxn`.

Bitwise AND: &

The bitwise AND operator, &, acts as a series of AND gates operating between the two operands, as shown in Figure 18.2. Because taking a logical AND with the number 0 results in a 0, the bitwise AND operation is used to selectively clear bits while leaving others alone.

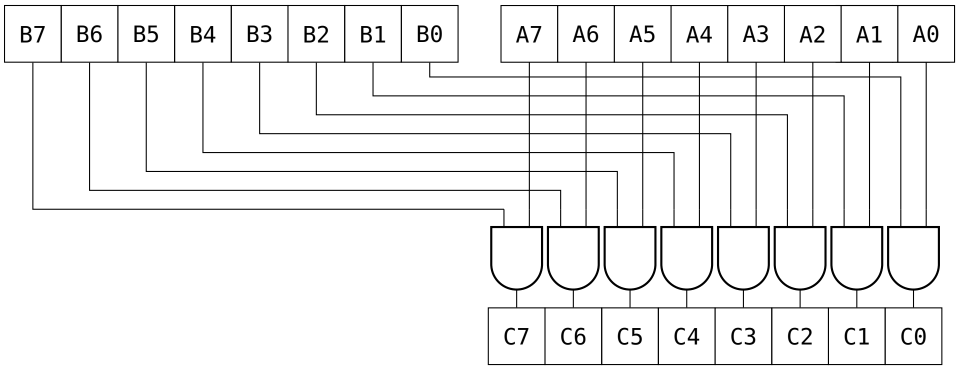


Figure 18.2: A bitwise AND operation takes the logical AND of each corresponding bit of the two operands.

Bitwise OR: |

The bitwise OR operator, |, acts as a series of OR gates operating between the two operands, as shown in Figure 18.3. Because taking a logical OR with the number 1 results in a 1, the bitwise OR operation is used to selectively set bits while leaving others alone.

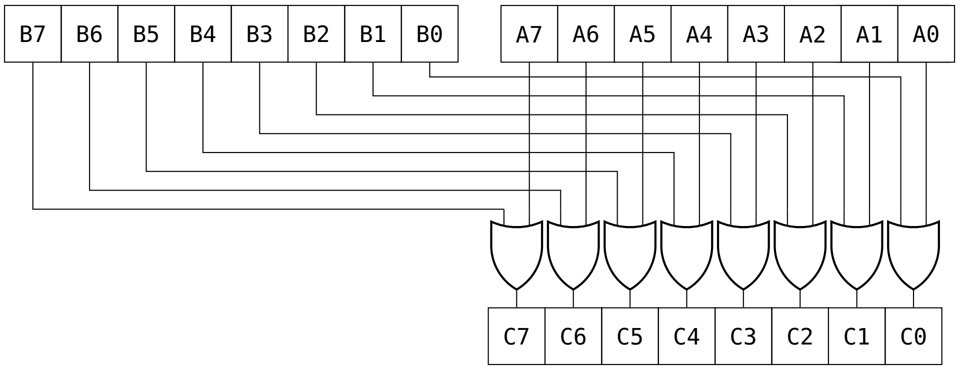


Figure 18.3: A bitwise OR operation takes the logical OR of each corresponding bit of the two operands.

Bitwise XOR: ˆ

The bitwise XOR operator, ˆ, acts as a series of XOR gates operating between the two operands, as shown in Figure 18.4. Because taking a logical XOR with the number 1 results in a toggle, the bitwise XOR operation is used to selectively toggle bits while leaving others alone.

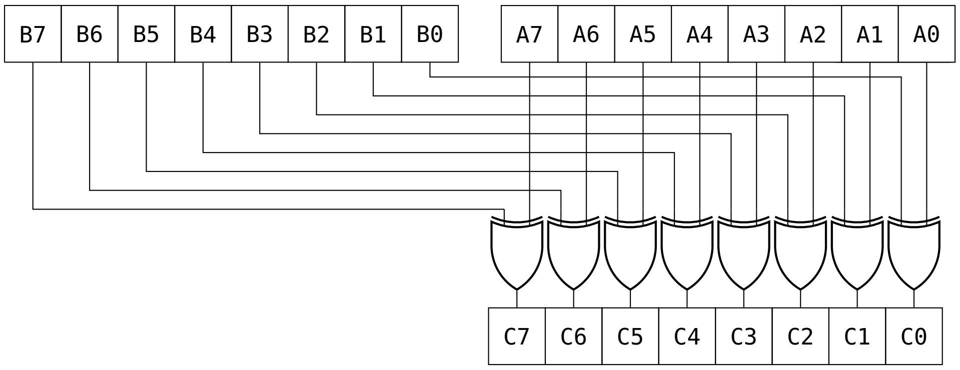


Figure 18.4: A bitwise XOR operation takes the logical OR of each corresponding bit of the two operands.

Bitwise NOT: ˜

The bitwise NOT operator, ˜, acts to invert each individual bit of the operand, as shown in Figure 18.5. The bitwise NOT operation is used to toggle all bits simultaneously.

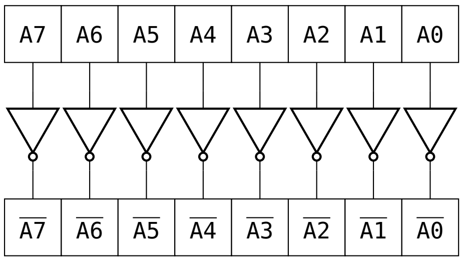


Figure 18.5: A bitwise NOT operation takes the logical NOT of each bit of the operand.

Bitshift Right: »

The bitshift right operator, », shifts the operand to the right a specific number of places. When being bitshifted to the right, the number 0 will be shifted in to the most significant bit. Figure 18.6 shows an example bitshift right operation.

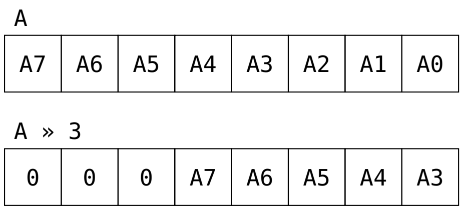


Figure 18.6: An example bitshift right operation.

Bitshift Left: «

The bitshift left operator, «, shifts the operand to the left a specific number of places. When being bitshifted to the left, the number 0 will be shifted in to the least significant bit. Figure 18.7 shows an example bitshift left operation.

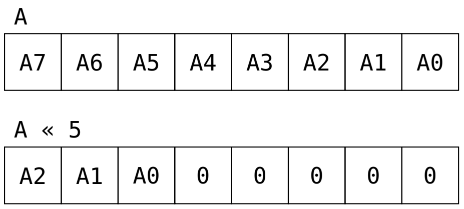


Figure 18.7: An example bitshift left operation.

Practice Problems

1. Find the content of `PORTB` after the following operations:

   (a) `PORTB = 0x37 & 0xCA;` `PORTB = 0x02`

   (b) `PORTB = 0x37 | 0xCA;` `PORTB = 0xFF`

   (c) `PORTB = 0x37 ˆ 0xCA;` `PORTB = 0xFD`

2. To selectively clear certain bits, which bitwise operator should be used? bitwise AND

3. To selectively set certain bits, which bitwise operator should be used? bitwise OR

4. Indicate the data on the ports for each of the following operations. (Note: the operations are independent of each other.)

   (a) `PORTD = 0xF0 & 0x45;` `PORTD = 0x40`

   (b) `PORTD = 0x0F & 0x56;` `PORTD = 0x06`

   (c) `PORTB = 0xF0 ˆ 0x76;` `PORTB = 0x86`

   (d) `PORTB = 0x0F ˆ 0x90;` `PORTB = 0x9F`

   (e) `PORTC = 0xF0 | 0x91;` `PORTC = 0xF1`

   (f) `PORTC = 0x0F | 0x99;` `PORTC = 0x9F`

(g) `PORTD = 0x65 >> 2;` `PORTD = 0x19`

(h) `PORTD = 0x39 << 2;` `PORTD = 0xE4`

(i) `PORTB = 0xD4 >> 3;` `PORTB = 0x1A`

(j) `PORTB = 0xA7 << 3;` `PORTB = 0x38`

18.5 Comparison & Boolean Operators

Comparison operators are used to compare two variables or values. They are useful when decisions need to be made based on how one variable compares to another. Comparison operators return Boolean values (`TRUE` and `FALSE`) based on the result of the operation. The comparison operators are shown in Table 18.4, with `a = 50` and `b = -10` used to show examples.

Table 18.4: Comparison operators with examples.

Operator	Description	Example	Result
`==`	Equal To	`a == b`	`FALSE`
`!=`	Not Equal To	`a != b`	`TRUE`
`>`	Greater Than	`a > b`	`TRUE`
`>=`	Greater Than or Equal To	`a >= b`	`TRUE`
`<`	Less Than	`a < b`	`FALSE`
`<=`	Less Than or Equal To	`a <= b`	`FALSE`

Boolean operators are used to make logical decisions based on the results of two or more comparison operations. Table 18.5 lists all of the Boolean operators. It is important to note that Boolean operators use TWO symbols (with the exception of NOT), whereas bitwise operators only use one.

Table 18.5: Boolean operators.

Operator	Name	Description

`&&`	AND	`a && b = TRUE` if both `a` and `b` are `TRUE`
		`a && b = FALSE` if either `a` or `b` is `FALSE`

`||`	OR	`a || b = TRUE` if either `a` or `b` is `TRUE`
		`a || b = FALSE` if both `a` and `b` are `FALSE`

`!`	NOT	`!a = TRUE` if `a` is `FALSE`
		`!a = FALSE` if `a` is `TRUE`

Practice Problems

1. Find result of the following operations:

   (a) `(-5 > 0) || (3 <= 8)` TRUE

   (b) `(1 >= 2) && (5 >= 0)` FALSE

   (c) `(14 != 38) && (20 >= -10)` TRUE

   (d) `(0 == 3) || !(5 != 8)` FALSE

18.6 Compound Operators

While not necessary to use in a microcontroller program, compound operators provide a convenient shorthand for arithmetic and bitwise operations. Using `x = x + 1` as an example, the instructions are to find the number stored in the variable `x`, add the number 1 to it, and then store the result into the variable `x`. This code may have an inefficiency in that the location of variable `x` has to be looked up twice if the compiler does not recognize that two parts of the expression are identical. It can be replaced with a compound operation instead: `x += 1`.

Operator	**Description**	**Example**	**Longhand Syntax**
`++`	Increment	`a++`	`a = a + 1`
`- -`	Decrement	`a- -`	`a = a - 1`
`+=`	Compound Addition	`a+=4`	`a = a + 4`
`-=`	Compound Subtraction	`a-=12`	`a = a - 12`
`*=`	Compound Multiplication	`a*=10`	`a = a * 10`
`/=`	Compound Division	`a/=5`	`a = a / 5`
`%=`	Compound Modulo	`a%=3`	`a = a % 3`
`&=`	Compound Bitwise AND	`a&=0xFC`	`a = a & 0xFC`
`|=`	Compound Bitwise OR	`a|=0xFC`	`a = a | 0xFC`
`ˆ=`	Compound Bitwise XOR	`aˆ=0xFC`	`a = a ˆ 0xFC`
`»=`	Compound Bitshift Right	`a»=3`	`a = a » 3`
`«=`	Compound Bitshift Left	`a«=6`	`a = a « 6`

Table 18.6: Compound operators with examples.

Practice Problems

1. Find the final value of each variable after the compound operation has been executed.

(a) `unsigned char a = 38;`
`a/=9;` `a = 4`

(b) `char n = -9;`
`n++;` `n = -8`

(c) `unsigned char j = 12;`
`j%=7;` `j = 5`

(d) `unsigned char x = 0xAC;`
`x^=0xE9;` `x = 0x45`

(e) `unsigned char num = 22;`
`num&=199;` `num = 6`

18.7 Control Flow: Conditional

A microcontroller can execute instructions using three different paradigms: sequential, conditional, and iterative. The **sequential flow** of code is what normally occurs in C programs. The program execution starts at the first line of code and carries out every subsequent line in order.

At times, however, it is necessary to execute code in a different fashion depending on the value of one or more variables (for example, in chapter 17 the need for turning up the heat based on the temperature of the room is discussed). This type of control flow is known as **conditional flow**. The two types of conditional flow functions are `If` & `If/Else` and `Switch Case`.

`If` & `If/Else` Statements

The `if` statement is a conditional statement that executes a different set of code based on the result of a Boolean and/or comparison operation. For example, when continuously polling a pushbutton to determine its state, the code may need to turn on an LED if the button was pushed. A flowchart showing how an `if` statement works is given in Figure 18.8.

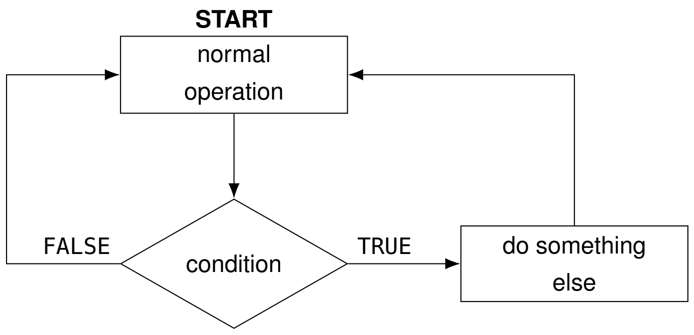


Figure 18.8: A flowchart depicting the operation of microcontroller during an `if` statement.

The syntax of this example is:

```
if (condition) {
  //do something else
}
// normal operation
```

It is possible that different blocks of code should be implemented based on the result of a condition. For example, to take the absolute value of a number, the magnitude of the input variable is compared with 0. If the variable is greater than zero, then the output will be the variable itself. Otherwise (which is where the `else` comes into action) the output should be the negative value of that variable. A flowchart showing the operation an `if/else` flow is given in Figure 18.9.

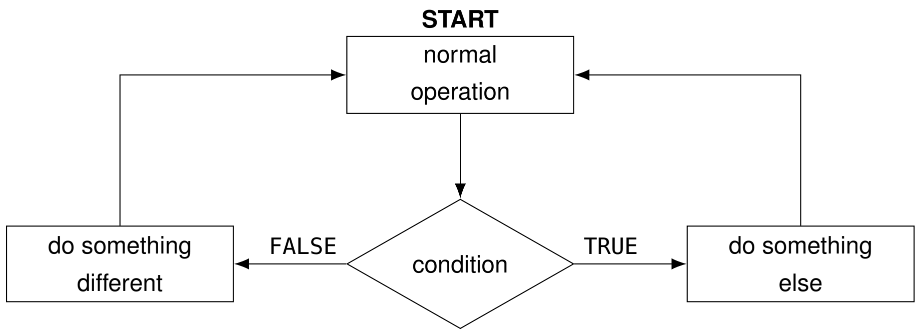


Figure 18.9: A flowchart depicting the operation of microcontroller during an `if/else` statement.

The syntax of this example is:

```
if (condition) {
  //do something else
}
else {
  //do something different
}
// normal operation
```

If more than two results are possible based on the conditional statement, one or more `else if` block can be included in the conditional flow. For example, a machine counting the number of widgets sorted per second may instruct a conveyor belt to speed up or slow down to varying degrees based on comparing the input variable (widgets sorted per second) to a series of setpoints. A flowchart showing the operation an `if/else if/else` flow is given in Figure 18.10.

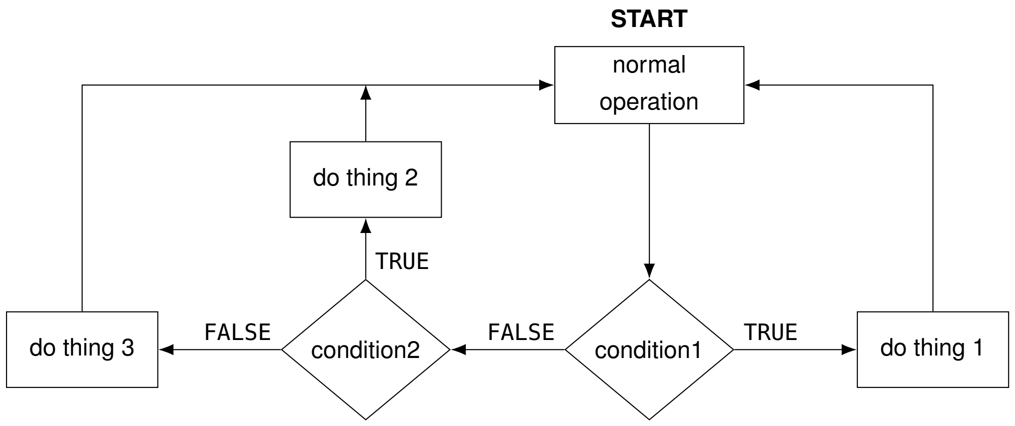


Figure 18.10: A flowchart depicting the operation of microcontroller during an `if/else if/else` statement.

The syntax of this example is:

```
if (condition1) {
  //do thing 1
}
else if (condition2) {
  //do thing 2
}
else {
  //do thing 3
}
// normal operation
```

`Switch` Case

If there are many repeated `else if` blocks, it can be more efficient to replace the `if` statement with a `switch` case statement instead. The flowchart and associated code syntax is given in Figure 18.11.

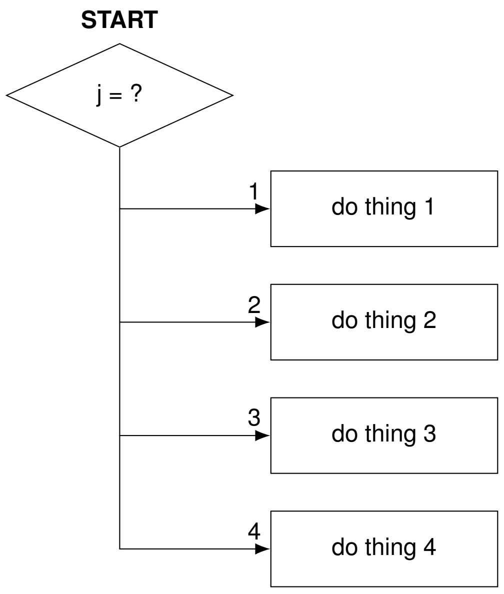


Figure 18.11: Flowchart and associated code for a `switch` case statement.

The syntax for the switch case shown in the flowchart in Figure 18.11 is:

```
switch(j) {
  case 1:
    // do thing 1
    break;
  case 2:
    // do thing 2
    break;
  case 3:
    // do thing 3
    break;
  case 4:
    // do thing 4
    break;
}
```

18.8 Control Flow: Iterative

Iterative flow of code is used to repeat identical (or nearly identical) functions a certain number of times (or infinitely). In chapter 12, the concept of a circular buffer was discussed for taking rolling averages. An average requires all of the elements of an array to be summed together before being divided by the number of array entries. Rather than having one line of code for each array element (which would be inefficient on many levels), a loop can be used to repeat code a certain number of times before returning to sequential flow. Three types of iterative flow are the `for` loop, `while` loop, and `do/while` loop.

`for` Loop

`for` loops are best used when a given segment of code needs to be iterated a given number of times (such as the example of summing up the circular buffer, which must be iterated a number of times equal to the number of elements in the array). The flowchart of a `for` loop is shown in Figure 18.12.

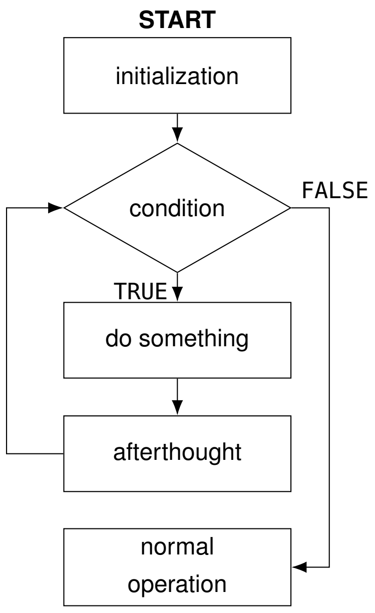


Figure 18.12: Flowchart indicating the usage of a `for` loop.

The syntax of a `for` loop,

```
for (initialization; condition; afterthought) {
  // code goes here
}
```

indicates that after an optional initialization, in which any variables required within the loop that aren't already declared are declared, a conditional statement is checked. If the conditional is TRUE, the contents of the loop will be executed. If the conditional is FALSE, the software will exit the loop. An optional afterthought is executed exactly once every time the loop ends.

The initialization, condition, and afterthought generally relate to iteration of the loop. Continuing with the circular buffer example, the following code could be used.

```
const unsigned char n = 5;
unsigned char arrayValues[n] = {20, 30, 50, 100, 10};
unsigned char arrayAverage = 0;
for (unsigned char j = 0; j < n; j++) {
  static unsigned char arraySum = 0;
  arraySum += arrayValues[j];
  arrayAverage = arraySum / n;
}
```

An infinite `for` loop is used when code needs to be repeated indefinitely, and can be written by leaving the initialization, condition, and afterthought blank as follows.

```
for ( ; ; ) {
  // this code will be repeated an infinite number of times
}
```

`while` Loop

A second type of loop is the `while` loop. It is recommended to use a `while` loop when code needs to be repeated until a given condition is true. This effectively allows the normal execution of code to be delayed until a particular condition is met.

The syntax of a `while` loop is:

```
// optional initialization
while (condition) {
  // code goes here
  // optional afterthought
}
```

An example flowchart of a `while` loop is given in Figure 18.13.

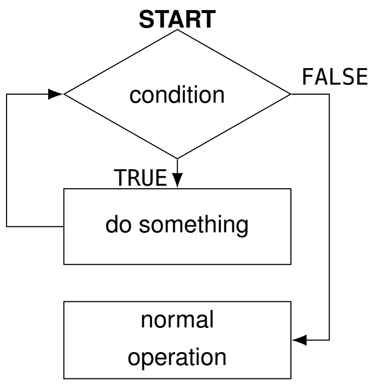


Figure 18.13: Flowchart indicating the usage of a `while` loop.

Using SPI communication, data must be transmitted completely from the `SPDR` register before the rest of the code can continue to execute. It is possible to determine when the code has been successfully completed when bit 7 in the `SPSR` register has been set. This is a perfect situation in which to use a `while` loop, as follows.

```
while (!(SPSR & (1 « 7))); // nothing occurs until bit 7 is
set
```

An infinite `while` loop can be written by making the condition equal to 1 as follows

```
while (1) {
  // this code will be repeated an infinite number of times
}
```

`do/while` Loop

A third type of loop is the `do/while`, which is very similar to a `while` loop, except that the body of the code is executed once before the condition is checked, as shown in the flowchart in Figure 18.14.

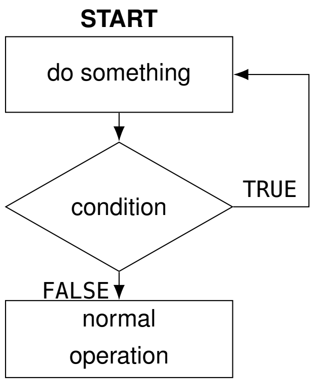


Figure 18.14: Flowchart indicating the usage of a `do/while` loop.

The syntax of a `do/while` loop follows.

```
// optional initialization
do {
  // code goes here
  // optional afterthought
} while (condition);
```

As with the `while` loop, to create an infinite `do/while` loop, the condition should be set to 1 as follows

```
do {
  // this code will be repeated an infinite number of times
} while (1);
```

Practice Problems

1. Find the final value of the variable x after the loop has been executed.

   (a) `for (unsigned char x = 0; x > 6; x++) { }`

   `x = 0`

   (b) `unsigned int x = 1;`
   `while (x <= 5) { x*=2; }`

   `x = 8`

   (c) `unsigned int x = 0;`
   `do { x++; } while(x <= 20);`

   `x = 21`

19 Assembly

Assembly can be used to program the ATmega328P; it requires a detailed understanding of the AVR instruction set.[1] In order to access all of the information stored in general purpose (GP) registers, data memory, and program memory, an understanding of memory addressing (discussed in chapter 7) is fundamental. The instructions allowed on the microcontroller can be divided into several categories.

[1] Atmel, "AVR Instruction Set Manual," November 2016.

19.1 Data Transfer Instructions

Before any meaningful instructions can take place, data must be loaded into the GP registers. This data can come from program memory, data memory (including the I/O registers), another GP register, or it can be an immediate data value. Instructions that load data into GP registers can:

- move – copy a register or pair of registers into a GP register,
- load – load immediate data into a GP register; load data directly or indirectly (with or without displacement, post-increment, or pre-decrement) from data memory into a GP register; load data indirectly from program memory into a GP register,
- in – input data from an I/O register into a GP register, and
- pop – take data from the stack and save it into a GP register.

Data can also be stored from a GP register into memory. These instructions can:

- store – store data directly or indirectly (with or without displacement, post-increment, or pre-decrement) from a GP register into data memory; store data indirectly from a GP register into program memory,

- out – output data from a GP register into an I/O register, and
- push – take data from a GP register and save it in the stack.

Practice Problems

1. Which general purpose registers can be used with instruction `LD`?

`r0 – r31`

2. Which general purpose registers can be used with instruction `LDI`?

`r16 – r31`

19.2 Arithmetic & Logic Instructions

Arithmetic and logic instructions allow the microcontroller to add, subtract, and multiply. Data can be complemented using either one's or two's complement. Logical operations include AND, OR, and XOR. Some of these instructions have the option to be used with or without a carry and using signed, unsigned, or fractional values. All of these instructions are carried out within one or two clock cycles. Most all of these instructions affect flags in `SREG`.

Of the arithmetic and logic instructions, many of them either use direct, two register addressing or immediate addressing with direct, one register addressing. For example, the `AND` instruction is used to perform the logical AND operation on the contents of two GP registers (a source and destination) and save the results in the destination register. The `ANDI` instruction is used to perform the logical AND operation on the contents of one GP register with an immediate and save the results back into the GP register.

The logic instructions are essentially equivalent to bitwise operators. Table 19.1 shows the logical assembly instructions and their equivalent C counterparts.

Table 19.1: Logical instructions can be used to emulate bitwise operators.

Operation	**Instruction(s)**
AND	`AND`, `ANDI`, `CBR`
OR	`OR`, `ORI`, `SBR`
XOR	`EOR`
NOT	`COM`

Notice that not all assembly instructions allow for immediate addressing. Shifting bits (as accomplished using bitshift operators in C) is discussed below, as bitshifting is accomplished using bit manipulation instructions. It is also possible to set an individual bit in a GP register using `SBR`, which is similar to using a bitwise OR with a single high bit. Similarly, an individual bit in a GP register can be cleared using `CBR`, which is similar to using a bitwise AND with a single low bit.

Addition and subtraction operations can take place with or without a carry. For example, `ADD` simply adds the contents of two GP registers (a source and a destination) and saves the result into the destination register. `ADC` adds the contents of two GP registers (a source and a destination) and the contents of the `C` flag in `SREG`, and then saves the result in the destination register.

The arithmetic and logic instructions also contain instructions that can:

- increment a register – `Rd` $\leftarrow$ `Rd` + 1,
- decrement a register – `Rd` $\leftarrow$ `Rd` $-$ 1,
- clear a register – `Rd` $\leftarrow$ `0x00`, and
- set a register – `Rd` $\leftarrow$ `0xFF`.

Practice Problems

1. Determine the contents of the destination register (in decimal) after each subsequent operation.

   (a) `LDI r17, 0x80` r17 = 128

   (b) `LDI r18, 0x15` r18 = 21

   (c) `COM r18` r18 = 234

   (d) `AND r18, r17` r18 = 128

   (e) `ORI r17, 0x03` r17 = 131

   (f) `EOR r17, r18` r17 = 3

   (g) `SER r18` r18 = 255

   (h) `SUB r18, r17` r18 = 252

   (i) `NEG r18` r18 = 4

   (j) `MUL r17, r18` r17 = 12

19.3 Branch (Control Flow) Instructions

Just as there are conditional and iterative control flow operations in C, there are instructions that allow for these same processes in AVR assembly. It is important to have a solid understanding of branch instructions and memory addressing, as these are fundamental to carrying out conditional logic in assembly.

One subset of branch instructions are jump instructions, which tell the microcontroller to move to a different part of memory. Essentially, these instructions change the value of the program counter so that a different part of the program memory will be addressed on the next clock cycle. The two most important jump instructions in AVR assembly are

- `JMP` – (absolute jump) jumps to a location in program memory, this instruction can address up to 4 M of memory using a 22-bit operand; and
- `RJMP` – (relative jump) jumps to a relative location in program memory, this instruction moves forward or backward in memory space using a relative change in the program counter (note that `RJMP` is not always capable of addressing the entire memory space).

There are also call instructions, which are used for executing subroutines. Locations in program memory are saved into the stack to ensure that the code is able to return to the same point once it has finished executing the subroutine.

Whereas in C a conditional statement is made in syntax and curly brackets are used to denote the start and end of the conditionally or iteratively executed subroutine, in assembly this is done using logical instructions, compare instructions, and branch instructions.

Two important compare instructions are

- `CP` – compare, which compares the contents of two GP registers `Rd` and `Rr`, and
- `CPI` – compare with immediate, which compares the contents of a GP register `Rd` with a constant.

Branch instructions cause the code to jump to a different address in memory if a certain condition is met. There many possible branch conditions, including

- branch if a flag is set or clear – these instructions will jump to a different address in memory if a flag in `SREG` is either 0 or 1,
- branch if two numbers are equal (`BREQ`) – this instruction will jump to a different address in memory if two numbers are the same,

- branch if two numbers are not equal (BRNE) – this instruction will jump to a different address in memory if two numbers are not the same,
- branch if one number is greater than or equal to another – there are different instructions to use for signed (BRGE) and unsigned (BRSH) values, and
- branch if one number is less than another – there are different instructions to use for signed (BRLT) and unsigned (BRLO) values.

The argument of a branch instruction is the name of a subroutine in the code, which is defined using a colon, as follows:

```
;define a subroutine known as function1
function1:
  ;compare two registers, repeat function1 if they are equal
  CP r1, r2
  BREQ function1
```

Conditional Control Flow

As seen in chapter 18, conditional control flow in C took on the form of **if statements** and **switch cases**. There isn't really a switch case equivalent in assembly; a series of mutually exclusive if statements can be used instead.

A simple if statement can be executed by using a compare instruction and a branch instruction. This works for checking to see if two values are equal, unequal, greater than, or less than, and making a decision based on the result.

The following example branches to the conditiontrue memory address if the value of register r1 $\geq$ r2, as shown in the flowchart in Figure 19.1.

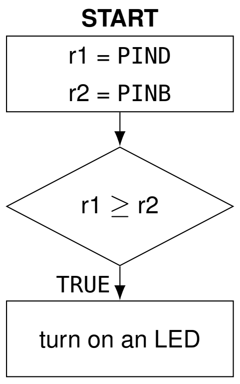


Figure 19.1: An example of conditional control flow (equivalent to a simple if statement) in assembly.

The values stored in the two GP registers come from the data stored in the `PINB` and `PIND` registers. If the condition is satisfied, an LED connected to pin 5 in port C is turned on. The associated assembly code is as follows:

```
;configure PORTC pin 5 as output
SBI DDRC, 5
;read contents of PINB and PIND
IN r1, PINB
IN r2, PIND
;compare r1 and r2
CP r1, r2
BRSH conditiontrue
;condition false
JMP end
;condition true subroutine
conditiontrue:
  SBI PORTC, 5
end:
```

This code is using unsigned values in registers `r1` and `r2`, which is why the `BRSH` instruction is used. Had signed values been necessary, the `BRGE` instruction would have been used instead.

This code can easily be tailored to act as an **if / else** statement using the lines of code between the `BRSH` instruction and the `conditiontrue` subroutine. Series of nested if / else statements can be used to generate an **if / elseif / else** situation.

More complicated if statements can be written by cleverly configuring compares and jumps. The following example uses GP registers `r16` and `r17` (which both obtain their data from I/O registers). In C, the conditional statement inside of the if would have looked like `((r16 < 100) && (r17 >= 50))`. If the condition is satisfied, an LED connected to pin 5 in port C is turned on. Otherwise, the LED is turned off. The flowchart is shown in Figure 19.2.

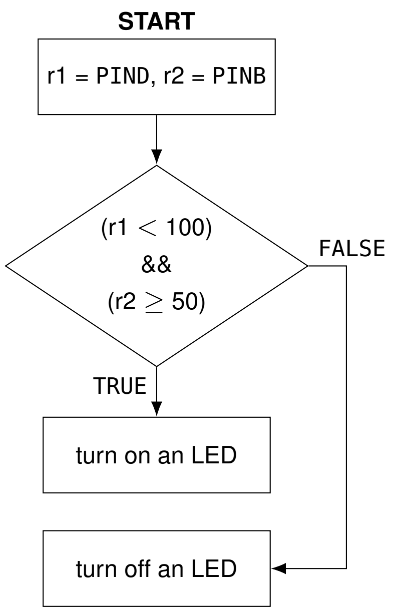


Figure 19.2: An example of conditional control flow (equivalent to a more complicated `if/else` statement) in assembly.

The associated assembly code is as follows:

```
;configure PORTC pin 5 as output
SBI DDRC, 5
;read contents of PINB and PIND
IN r16, PINB
IN r17, PIND
;compare r16 with 100
CPI r16, 100
;if r16 >=100, condition is false
BRSH conditionfalse
;compare r17 with 50
CPI r17, 50
;if r17 < 50, condition is false
BRLO conditionfalse
conditiontrue:
  SBI PORTC, 5
  JMP end
conditionfalse:
  CBI PORTC, 5
end:
```

Note that GP registers must be at least 16 due to the use of instructions using immediate addressing. Note also that unsigned values were used in both GP registers.

Iterative Control Flow

The equivalent of a **for loop** can be accomplished in assembly by initializing a GP register to take on a particular value, executing the iterative code, executing the afterthought to the GP register, and then comparing the value of the GP register to the conditional statement.

The following example demonstrates how to increment the `PORTB` register from 0–10; perhaps it is connected to a 7-segment display via a BCD to 7-segment decoder. This code will enable the decoder to increment from 0–10. The flowchart is shown in Figure 19.3

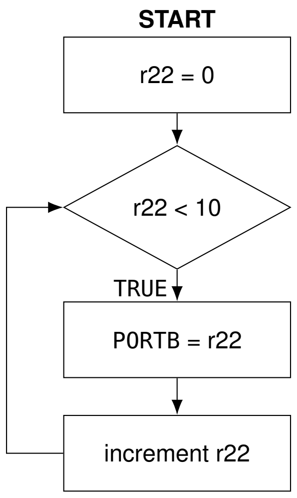


Figure 19.3: An example of iterative control flow (equivalent to a `for` loop) in assembly.

The associated assembly code is as follows:

```
;configure PORTB pins as outputs
LDI r23, 0x0F
OUT DDRB, r23
;initialization:  clear register r22
CLR r22
loop1:
  OUT PORTB, r22
  ;afterthought:  increment r22
  INC r22
  ;conditional:  compare r22 with 10, repeat if it’s lower
  CPI r22, 10
  BRLO loop1
```

Note the use of GP registers greater than 16 for the instructions using immediate addressing (`LDI` and `CPI`).

As with C code, the for configuration is used when a segment code should be iterated a given number of times before returning to normal operation. When a condition needs to be met before executing a segment of code, a **while** or **do/while** configuration should be used. The assembly code will still use compare and branch instructions, the only difference from the for configuration will be the ordering of each exact instruction.

The following example turns on an LED (connected to port C pin 0) if a toggle switch (connected to port B pin 0) is turned on. Otherwise, if the toggle switch is turned off, the LED turns off. The flowchart is shown in Figure 19.4.

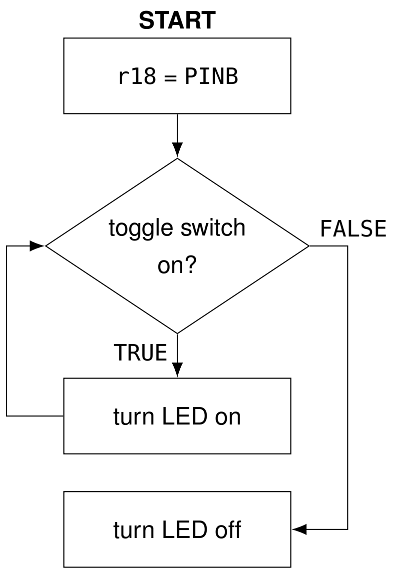


Figure 19.4: Flowchart indicating the usage of a `while` loop.

The assembly code follows

```
SBI DDRC, 0
main:
JMP checktrue
conditiontrue:
  SBI PORTC, 0
checktrue:
  IN r18, PINB
  ANDI r18, 0x01
  CPI r18, 0x01
  BREQ conditiontrue
CBI PORTC, 0
JMP main
```

A do/while loop executes the contents of the loop first before checking the conditional statement. In this case, the `JMP checktrue` instruction can be removed to simulate a do/while loop.

Control Flow & SREG

Compare and branch instructions operate by determining whether or not a particular flag was set in `SREG`. For example, conditional logic that would have been represented as `if (a == b)` in C code is carried out using a compare instruction on two GP registers, and then a branch instruction if the two registers are equal (`BREQ`), as follows:

```
CP r1, r2
BREQ subroutine1
```

The compare instruction executes the operation `r1` − `r2`. If the result is zero (indicating that the two values are equal), the `Z` flag in `SREG` will be set. Therefore, the `BREQ` instruction checks to see if the `Z` flag in `SREG` had been set as a result of the previous instruction and uses that to decide whether or not to change the value of the program counter. The flags that are checked for several logical comparisons are shown in Table 19.2.

Comparison	Instruction	Flag
`a == b`	`BREQ`	`Z` = 1
`a != b`	`BRNE`	`Z` = 0
`a >= b`	`BRSH` (unsigned)	`C` = 0
`a >= b`	`BRGE` (signed)	`N` ⊕ `V` = 0
`a < b`	`BRLO` (unsigned)	`C` = 1
`a < b`	`BRLT` (signed)	`N` ⊕ `V` = 1

Table 19.2: Branch instructions check the status of flags in `SREG`.

Because flag checks are how the branch instructions know whether or not to change the value of the program counter, it can be seen why there are no "greater than" or "less than or equal to" instructions. With unsigned values, a greater than situation would occur if `Z` = 0 AND `C` = 0, so both flags would need to be logically compared. It is possible to do this in assembly manually, but it is much easier to change the value to compare with. Similarly, with unsigned values, a less than or equal to situation would occur if `C` XOR `Z` is TRUE.

Practice Problems

1. Determine the contents of `SREG` after each subsequent operation.

   (a) `LDI r18, 20` `0x00`

   (b) `LDI r19, 13` `0x00`

(c) `CP r18, r19` `0x20`

(d) `CPI r19, 200` `0x01`

(e) `NEG r18` `0x35`

(f) `LDI r18, 220` `0x00`

(g) `LDI r19, 57` `0x00`

(h) `ADD r18, r19` `0x21`

2. Which instruction(s) will branch if `Z` = 1? `BREQ`

3. Which instruction(s) will branch if `Z` = 0? `BRNE`

4. Which instruction(s) will branch if `C` = 1? `BRCS`, `BRLO`

5. Which instruction(s) will branch if `C` = 0? `BRCC`, `BRSH`

19.4 Bit Manipulation Instructions

Bit manipulation instructions consist of instructions that allow registers to be manipulated on the bit level. This consists of shifting, swapping, setting, and clearing instructions.

Shift instructions come in a few flavors: logical shift, arithmetic shift, and rotate through carry. A **logical shift** is equivalent to bitshift operations in C. When a register is logically shifted right (using the `LSR` instruction), the MSB is replaced with a 0. Assuming that the result does not overflow, this operation is equivalent to multiplying a binary value (signed or unsigned) by two. When a register is logically shifted left (using the `LSL` instruction), the LSB is replaced by a 0. A logical shift right is equivalent to dividing an unsigned binary value by two. While C code is able to perform multiple bitshifts at a time (for example, to bitshift a value 3 times to the right, »3 is used), each shift instruction in assembly executes a single time. Iterative control flow can be used to execute multiple bit shifts.

In an **arithmetic shift** right (which is the `ASR` instruction, no such arithmetic shift left exists as it would be equivalent to logical shift left), the sign bit is shifted in to the MSB of the register. This is equivalent to dividing a signed or unsigned binary value by two by preserving the sign bit.

Rotate through carry instructions can occur to the left (using the ROL instruction) or right (using the ROR instruction), and execute a rotation (where the register wraps around so the MSB is moved to the LSB, or vice versa) through the previous value of the carry bit. A rotate left through carry is depicted graphically in Figure 19.5.

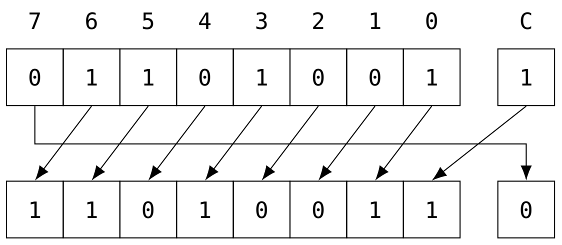


Figure 19.5: Rotate left through carry.

A rotate right through carry is depicted graphically in Figure 19.6. These instructions are particularly useful when dealing with values that are stored in two registers (i.e., the numbers are greater than 8 bits). When rotating, the value that is rotated out is saved into the carry flag of SREG and is therefore ready to be rotated in to the next byte without the need for additional processing.

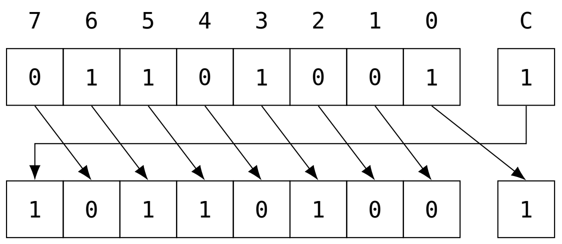


Figure 19.6: Rotate right through carry.

The SWAP instruction swaps the high and low nibbles on the operand register.

The T flag in SREG is a bit used to store a single bit of data. Storing data to this flag is accomplished using the BST instruction, while retrieving data from this flag is accomplished using the BLD instruction.

The last set of bit manipulation pertain to setting and clearing bits. Bits can be set and cleared individually in an I/O register using SBI and CBI, individual bits in SREG can be set and cleared using BSET and BCLR, and each individual flag in SREG can be set or cleared using one of the many instructions that accomplish those tasks. Individual bits can be cleared or set in a GP register by using ANDI or ORI similar to bitwise AND and bitwise OR operations in C.

19.5 *Miscellaneous Instructions*

The miscellaneous instructions on the ATmega328P are `BREAK` (break), which is used by the on-chip debug system; `NOP` (no operation), which spends one clock cycle doing nothing; `SLEEP` (sleep), which sets the device into sleep mode; and `WDR` (watchdog reset), which resets the watchdog timer.

20
Index