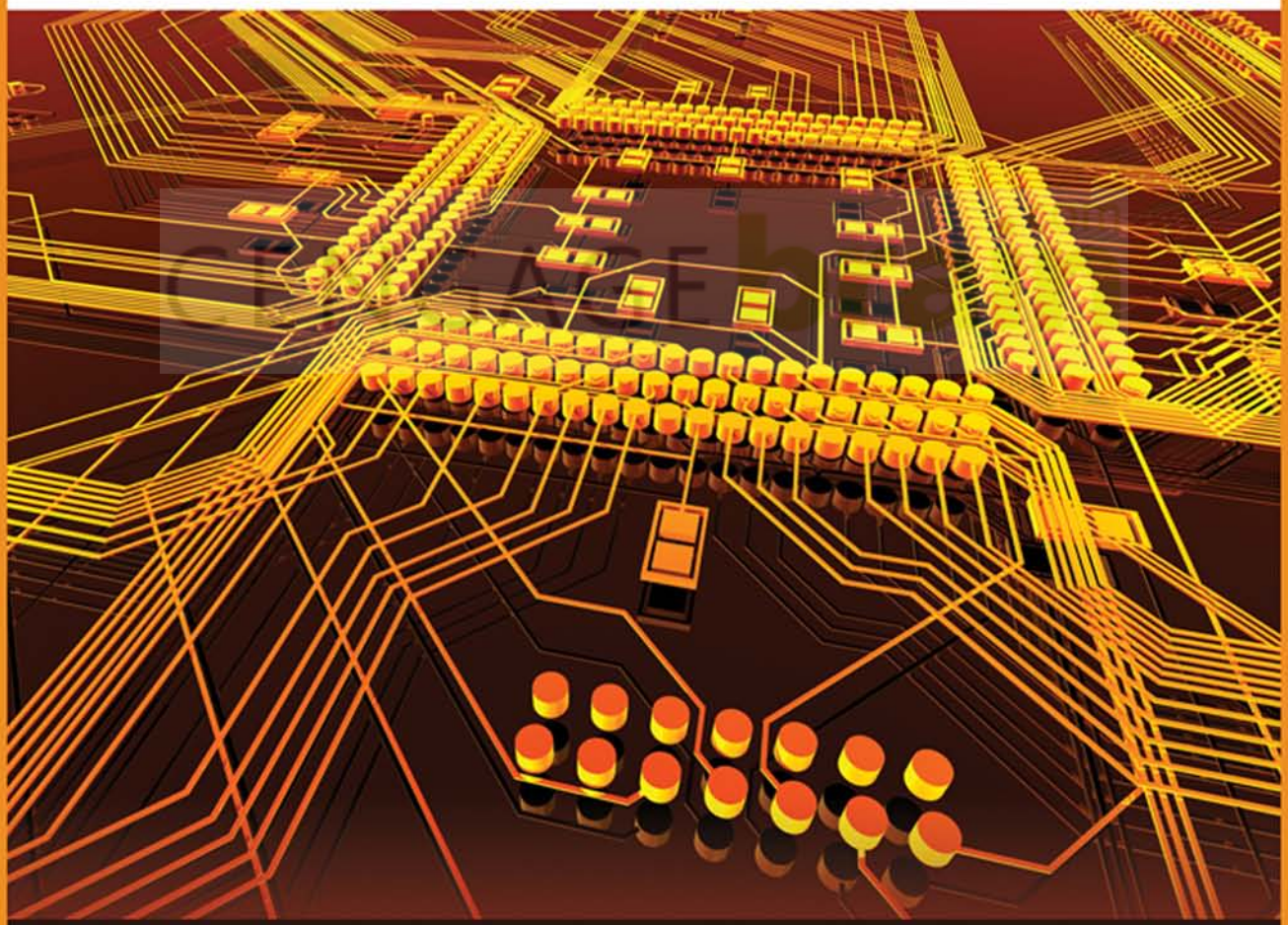
THIRD EDITION

EMBEDDED MICROCOMPUTER SYSTEMS

REAL TIME INTERFACING



JONATHAN W. VALVANO

Embedded Microcomputer Systems: Real Time Interfacing, Third Edition
Jonathan W. Valvano

Publisher, Global Engineering: Christopher M. Shortt

Acquisitions Editor: Swati Meherishi

Assistant Development Editor: Yumnam Ojen Singh

Editorial Assistant: Tanya Altieri

Team Assistant: Carly Rizzo

Marketing Manager: Lauren Betsos

Media Editor: Chris Valentine

Content Project Manager: D. Jean Buttrom

Production Service: RPK Editorial Services, Inc.

Copyeditor: Shelly Gerger-Knechtl

Proofreader: Becky Taylor

Indexer: Shelly Gerger-Knechtl

Compositor: Glyph International Ltd.

Senior Art Director: Michelle Kunkler

Cover Designer: Andrew Adams

Cover Image: © Cybrain/Shutterstock

Internal Designer: Carmela Periera

Senior Rights Acquisitions Specialist: Mardell Glinski-Schultz

Text and Image Permissions Researcher: Kristiina Paul

First Print Buyer: Arethea L. Thomas

Library of Congress Control Number: 2010938462

ISBN 13: 978-1-111-42625-5

ISBN-10: 1-111-42625-2

Cengage Learning
200 First Stamford Place, Suite 400
Stamford, CT 06902
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
international.cengage.com/region.

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your course and learning solutions, visit **www.cengage.com/engineering.**

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com.**

Printed in the United States of America
1 2 3 4 5 6 7 14 13 12 11 10

1 Microcomputer-Based Systems

Chapter 1 objectives are to introduce:

- Embedded systems
- Practical aspects of digital logic
- Architecture of the Freescale MC9S12 family
- Parallel port input/output operations

The overall objective of this book is to teach the design of embedded systems. It is an effective approach to learn new techniques by doing them. But, the dilemma in teaching a laboratory-based topic such as embedded systems is that there is a tremendous volume of details that first must be learned before microcomputer hardware and software systems can be designed. The approach taken in this book is to learn by doing, starting with very simple problems and building up to more complex systems later in the book.

We will begin with a short section introducing some terminology and the basic components of a computer system. In order to understand the context of our designs, we will overview the general characteristics of embedded systems. It is in these discussions that we develop a feel for the range of possible embedded applications. Because courses taught using this book typically have a lab component, we will review some practical aspects of digital logic. We then introduce the specific architectures of the Freescale MC9S12 family. For more detailed information concerning your specific microcomputer, refer to the respective Freescale manual. Data sheets for the devices we will use can be found in PDF format at **http://users.ece.utexas.edu/~valvano**. At the end of the chapter, we will discuss prototyping methods to build embedded systems and present a simple example with binary inputs and outputs.

Even though we will design systems based specifically on the MC9S12C32, these solutions can, with little effort, be implemented on other versions of the MC9S12 family. If your overall goal is to develop assembly language software, then the C code can serve to clarify the software algorithms. If your overall goal is to develop C code, then I strongly advise you to learn enough assembly language so that you can understand the machine code that your compiler generates. From this understanding, you can evaluate, debug, and optimize your system.

1.1 Computer Architecture

A *computer* system combines a processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports, as shown in Figure 1.1. *Software* is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed. The *processor* executes the software by retrieving and interpreting these instructions one at a time. A *microprocessor* is a small processor, where small refers to size (i.e., it fits in your hand) and not computational ability. For example, the Intel Pentium and the PowerPC are microprocessors. A *microcomputer* is a small computer, where again small refers to size (i.e., you can carry it) and not computational ability. For example, a desktop PC is a microcomputer. A very small microcomputer, called a *microcontroller*, contains all the components of a computer (processor, memory, I/O) on a single chip. The Freescale MC9S12C32 used in this book is a microcontroller. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from an 8-bit 6811 running at 2 MHz with 512 bytes of memory to a personal computer with a state-of-the-art 64-bit processor running at multi-GHz speeds having terabytes of storage.

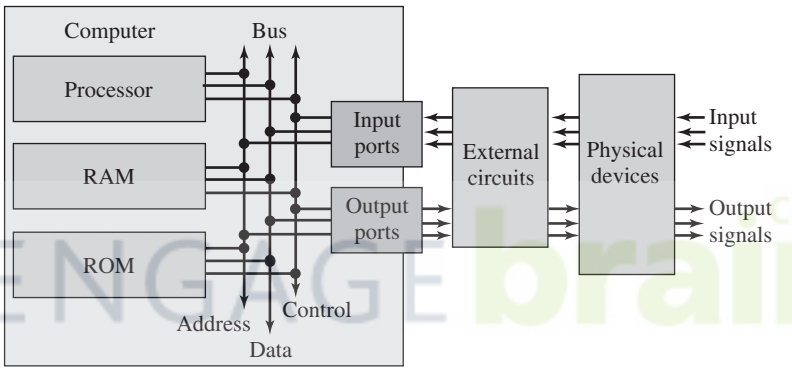


Figure 1.1
The basic components of a computer system include processor, memory, and I/O.

The computer can store information in *RAM* by writing to it, or it can retrieve previously stored data by reading from it. Most RAMs are *volatile*, meaning if power is interrupted and restored, the information in the RAM is lost. Information is programmed into *ROM* using techniques more complicated than writing to RAM. On the other hand, retrieving data from a ROM is identical to retrieving data from RAM. ROMs are *nonvolatile*, meaning if power is interrupted and restored, the information in the ROM is retained. Some ROMs are programmed at the factory and can never be changed. A Programmable ROM (*PROM*) can be erased and reprogrammed by the user, but the erase/program sequence is typically 10000 times slower than the time to write data into a RAM. Some PROMs are erased with ultraviolet light and programmed with voltages, whereas electrically erasable PROMs (*EEPROM*) are both erased and programmed with voltages. Flash ROM is a popular type of EEPROM. For most of the systems in this book, we will store instructions and constants in ROM and place variables and temporary data in RAM.

Checkpoint 1.1: What are the differences between a microcomputer, a microprocessor, and a microcontroller?

The external devices attached to the computer provide functionality for the system. An *input port* is hardware on the computer that allows information about the external

world to be entered into the computer. The computer also has hardware called an *output port* to send information out to the external world. An *interface* is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator moves the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist in either digital or analog form. In general, we can classify I/O interfaces into four categories

parallel—binary data is available simultaneously on a group of lines
serial—binary data is available one bit at a time on a single line
analog—data is encoded as an electrical voltage, current, or power
time—data is encoded as a period, frequency, pulse width, or phase shift

Checkpoint 1.2: What are the differences between an input port and an input interface?

Checkpoint 1.3: List three input interfaces available on a personal computer.

Checkpoint 1.4: List three output interfaces available on a personal computer.

In this book, numbers that start with $ (e.g., $64) are specified in hexadecimal, which is base 16. In C, we start hexadecimal numbers with 0x (e.g., 0x64). Intel assembly language adds an "H" at the end to specify hexadecimal (e.g., 64H). Texas Instruments uses "h" (e.g., 64h).

In a system with *memory-mapped I/O*, as shown in Figure 1.1, the I/O ports are connected to the processor in a manner similar to memory. I/O ports are assigned addresses, and the software accesses I/O using reads and writes to the specific I/O addresses. The software inputs from an input port using the same instructions as it would if it were reading from memory. Similarly, the software outputs from an output port using the same instructions as it would if it were writing to memory. The processor, memory, and I/O are connected together by an address bus, a data bus, and a control bus. Together, these buses direct the data transfer between the various modules in the computer. A *bus* is defined as a collection of signals, which are grouped for a common purpose. For example, the *address bus* on the 9S12 is 16 signals (A15-A0), which together specify the memory address ($0000 to $FFFF) that is currently being accessed. The address specifies both which module (input, output, RAM or ROM) as well as which cell within the module will communicate with the processor. The *data bus* contains the information that is being transferred, which on the 9S12 is 16 bits (D15-D0), but it can transfer either 8-bit or 16-bit data. The *control bus* specifies the timing and the direction of the transfer. We call a complete data transfer a *bus cycle*. In a simple computer system, like the 9S12, only two types of transfers are allowed, as shown in Table 1.1. In this simple system, the processor always controls the address (where to access), the direction (read or write), and the control (when to access.) The MC9S12C32 has 2048 bytes of RAM located at addresses $3800 to $3FFF. Figure 1.2 illustrates how the

Table 1.1 Simple computers generate two types of cycles.

Type	Address Driven by	Data Driven by	Transfer
Memory Read Cycle	Processor	RAM, ROM or Input	Data copied to processor
Memory Write Cycle	Processor	Processor	Data copied to Output or RAM

processor fetches the 8-bit contents of location $3800 using a read cycle. Assume memory at address $3800 has the value $98. The processor first places the RAM address $3800 on the address bus, then the processor issues a read command on the control bus. The memory will respond by placing its $98 information on the data bus, and lastly the processor will accept the data and terminate the read command.

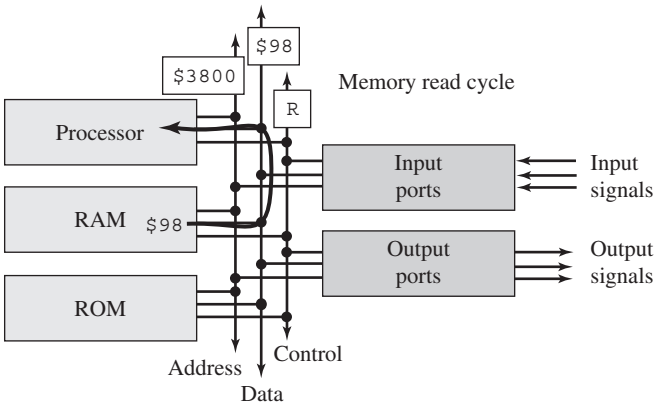


Figure 1.2
A memory read cycle copies data from RAM, ROM, or an input device into the processor.

Figure 1.3 illustrates how the processor stores the 8-bit value $25 into RAM location $3800 using a write cycle. The processor first places the RAM address $3800 on the address bus. Next, the processor places the $25 information on the data bus, and then the processor issues a write command on the control bus. The memory will respond by storing the $25 information into the proper place, and after the processor is sure the memory has captured the data, it will terminate the write command.

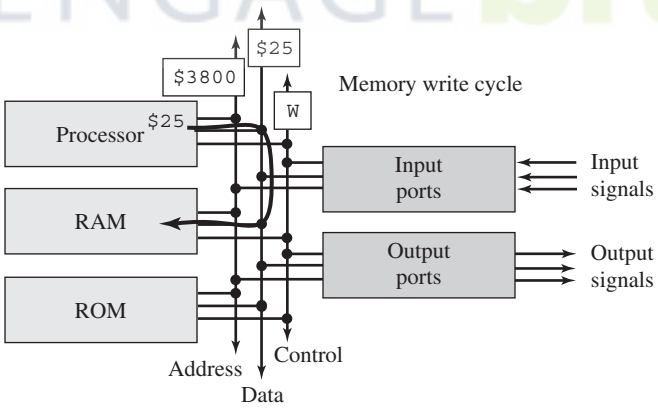


Figure 1.3
A memory write cycle copies data from the processor into RAM or an output device.

You see that if we wish to transfer data from an input device into RAM, we must first transfer it from input to the processor, then from the processor into RAM. In some microcontrollers, such as the ARM Cortex-M3 and in all desktop PCs, we can transfer data directly from input to RAM or from RAM to output using *direct memory access* (DMA). The *bandwidth* of an I/O device is the number of information bytes/sec that can be transferred. Because DMA is faster, we will use this method to interface high bandwidth devices such as disks and networks. During a read DMA cycle (Figure 1.4), data flows directly from the memory to the output device. Many systems that support DMA also allow data to be transferred from memory to memory (See Chapter 10).

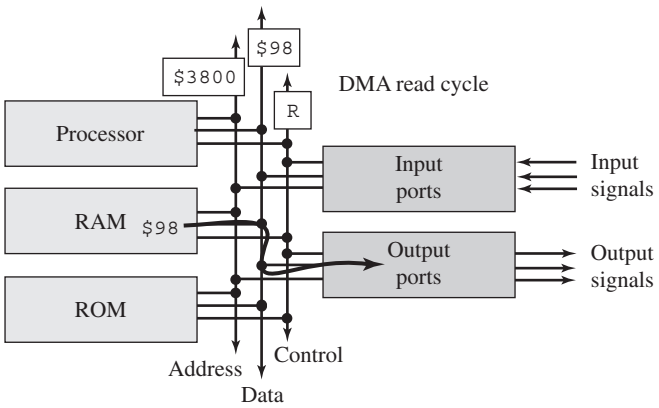


Figure 1.4
A DMA read cycle copies data from RAM, ROM, or an input device into an output device.

During a write DMA cycle (Figure 1.5) data flows directly from the input device to memory.

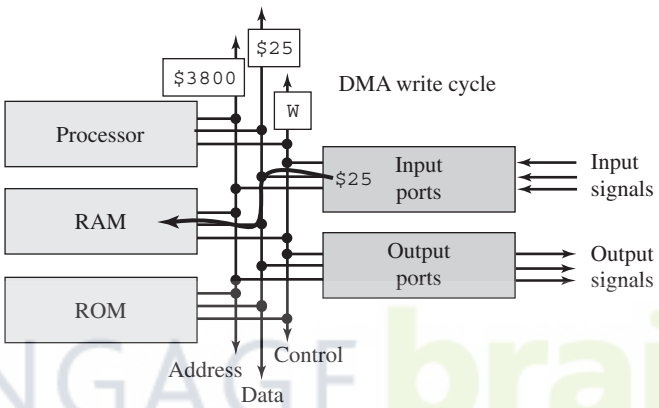


Figure 1.5
A DMA write cycle copies data from the input device into RAM or an output device.

Input/output devices are important in all computers, but they are especially significant in an embedded system. In a computer system with *I/O-mapped I/O*, the control bus signals that activate the I/O are separate from those that activate the memory devices. These systems have a separate address space and separate instructions to access the I/O devices. The original Intel 8086 had four control bus signals: MEMR, MEMW, IOR, and IOW. MEMR and MEMW were used to read and write memory, while IOR and IOW were used to read and write I/O. The Intel x86 refers to any of the processors that Intel has developed based on this original architecture. Even though we do not consider the personal computer (PC) an embedded system, there are embedded systems developed on this architecture. One such platform is called the PC/104 Embedded-PC. The Intel x86 processors continue to implement this separation been memory and I/O. Currently, there are 32 to 64 memory address lines, but only 16 of those lines are used to access I/O devices. The other address lines are not used during an I/O bus cycle. Rather than use the regular memory access instructions, the Intel x86 processor uses special `in` and `out` instructions to access the I/O devices. The advantages of I/O-mapped I/O are that software can not inadvertently access I/O when it thinks it is accessing memory. In other words, it protects I/O devices from common software bugs, such as bad pointers, stack overflow, and buffer overflows. In contrast, systems with memory-mapped I/O are easier to design, and the software is easier to write.

The processor within the 9S12 microcontroller has four major components, as illustrated in Figure 1.6. The *bus interface unit* (BIU) reads data from the bus during a read cycle, and writes data onto the bus during a write cycle. The 9S12 has a single processor and

does not support DMA. Therefore, the BIU always drives the address bus and the control signals of the bus. The *effective address register* (EAR) contains the memory address used to fetch the data needed for the current instruction.

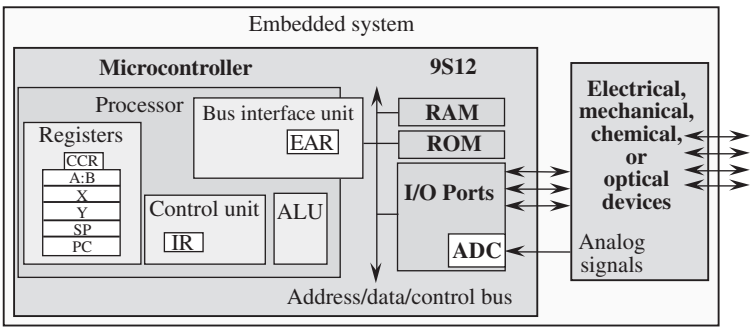


Figure 1.6
The four basic the components of 9S12 processor.

The *control unit* (CU) orchestrates the sequence of operations in the processor. The CU issues commands to the other three components. The *instruction register* (IR) contains the *operation code* (or *opcode*) for the current instruction. Most 9S12 opcodes are 8 bits wide, but some are 16 bits. Most instructions have two parts, the opcode that defines the function to perform, and an *operand* that specifies the data to be used. In an embedded system the software is converted to machine code, which is a list of instructions, and stored in nonvolatile memory. When the system is running, instructions one at a time are fetched from memory and executed.

The *registers* are high-speed storage devices located in the processor. Registers do not have addresses like regular memory, but rather they have specific functions explicitly defined by the instruction. *Accumulators* are registers that contain data. *Index registers* contain addresses. The *program counter* (PC) points to the memory containing the instruction to execute next. In an embedded system, the PC usually points into nonvolatile memory (e.g., ROM, EPROM, or EEPROM). The information stored in nonvolatile memory (e.g., the instructions) is not lost when power is removed. The *stack pointer* (SP) points to the RAM and defines the stack. The stack is an extremely important component of software development and can be used to pass parameters, save temporary information, and implement local variables. The internal RAM of the 9S12 is volatile memory, meaning its information is lost when power is removed. On some systems, such as calculators and PDAs, a separate battery powers the RAM, creating nonvolatile RAM. The *condition code register* (CCR) contains the status of the previous operation, as well as some operating mode flags such as the interrupt enable bit. This register is called the *flag register* on the Intel computers.

The *arithmetic logic unit* (ALU) performs arithmetic and logic operations. Addition, subtraction, multiplication, and division are examples of arithmetic operations. And, or, exclusive or, and shift are examples of logical operations.

Checkpoint 1.5: For what do the acronyms CU DMA BIU ALU stand?

In general, the execution of an instruction goes through four phases. First, the computer fetches the machine code for the instruction by reading the value in memory pointed to by the program counter (PC). Some instructions are only one byte long, while others are two or more bytes. After each byte of the instruction is fetched, the PC is incremented. At this time, the instruction is decoded, and the effective address is determined (EAR). Many instructions require additional data, and during phase 2, the data is retrieved from memory at the effective address. Next, the actual function for this instruction is performed. Often, the computer bus is idle at this time, because no additional data is required. During the last phase, the results are written back to memory. All instructions have a phase 1, but the other three phases

may or may not occur for any specific instruction. Each of the phases may require one or more bus cycles to complete. Each bus cycle reads or writes one piece of data. The 9S12 bus cycle can transfer 8-bit or 16-bit data.

Phase	Function	R/W	Address	Comment
1	Instruction fetch	read	PC++	Put into IR,
2	Data read	read	EAR	Data passes through ALU,
3	Operation	none		ALU operations, set CCR
4	Data store	write	EAR	Results stored in memory

1.2 Embedded Computer Systems

An *embedded computer system* is an electronic system that includes a microcomputer such as the Freescale 9S12 that is configured to perform a specific dedicated application, drawn previously as Figure 1.6. To better understand the expression *embedded microcomputer system*, consider each word separately. In this context, the word embedded means "hidden inside so one can't see it." The software that controls the system is programmed or fixed into ROM and is not accessible to the user of the device. Even so, *software maintenance*, which is verification of proper operation, updates, fixing bugs, adding features, extending to new applications, updating end user configurations, is still extremely important. In this book, we will develop techniques that facilitate this important aspect of system design. Embedded systems have these four characteristics.

First, embedded systems typically perform a single function. Consequently, they solve only a limited range of problems. For example, the embedded system in a microwave oven may be reconfigured to control different versions of the oven within a similar product line. Still, a microwave oven will always be a microwave oven, and you can't reprogram it to be a dishwasher. What makes each embedded system unique are the I/O ports of the microcontroller and the external devices interfaced to them.

Second, embedded systems are tightly constrained. There are typically very specific performance parameters within which the system must operate. For example, a cell-phone carrier typically gets 832 radio frequencies to use in a city, a hand-held video game must cost less than $50, an automotive cruise control system must operate the vehicle within 3 mph of the set-point speed, and a portable MP3 player must operate for 12 hours on one battery charge.

Third, many embedded systems must operate in *real time*. In a real-time computer system, we can put an upper bound on the time required to perform the input-calculation-output sequence. A real-time system can guarantee a worst-case upper bound on the response time between when the new input information becomes available and when that information is processed. Another real-time requirement that exists in many embedded systems is the execution of periodic tasks. A periodic task is one that must be performed at equal time intervals. A real-time system can put a small and bounded limit on the interval between when a task should be run and when it is actually run. Because of the real-time nature of these systems, microcontrollers like the 9S12 have a rich set of features to handle all aspects of time.

The fourth characteristic of embedded systems is their small memory requirements. There are exceptions to this rule, such as those that process video or audio, but most have memory requirements measured in thousands of bytes.

There have been two trends in the microcontroller field. The first trend is to make microcontrollers smaller, cheaper, and with lower power. The Microchip PIC and Texas Instruments MSP430 families are good examples of this trend. Size, cost, and power are critical factors for high-volume products, where the products are often disposable. On the other end of the spectrum is the trend of larger RAM and ROM, faster processing, and increasing integration of complex I/O devices, such as Ethernet, radio, graphics, and audio.

It is common for one device to have multiple microcontrollers, where the operational tasks are distributed and the microcontrollers are connected in a local area network (LAN). These high-end features are critical for consumer electronics, medical devices, automotive controllers, and military hardware, where performance and reliability are more important than cost. However, small size and low power continue as important features for all embedded systems.

Checkpoint 1.6: What is an embedded system?

The computer engineer has many design choices to make when building a real-time embedded system. Often, defining the problem, specifying the objectives, and identifying the constraints are harder than actual implementations. In this book, we will develop computer engineering design processes, introducing fundamental methodologies for problem specification, prototyping, testing, and performance evaluation.

In this book, we will refer to devices such as the Freescale MC9S12C32 simply as 9S12. The different versions of the microcomputers contain varying amounts of memory and input/output (I/O) devices. A typical automobile now contains an average of ten microcontrollers. In fact, upscale homes may contain as many as 150 microcontrollers, and the average consumer now interacts with microcontrollers up to 300 times a day. As shown in Figure 1.7, the general areas that employ embedded microcomputers encompass every field of engineering:

- Communications
- Automotive
- Military
- Medical
- Consumer
- Machine control

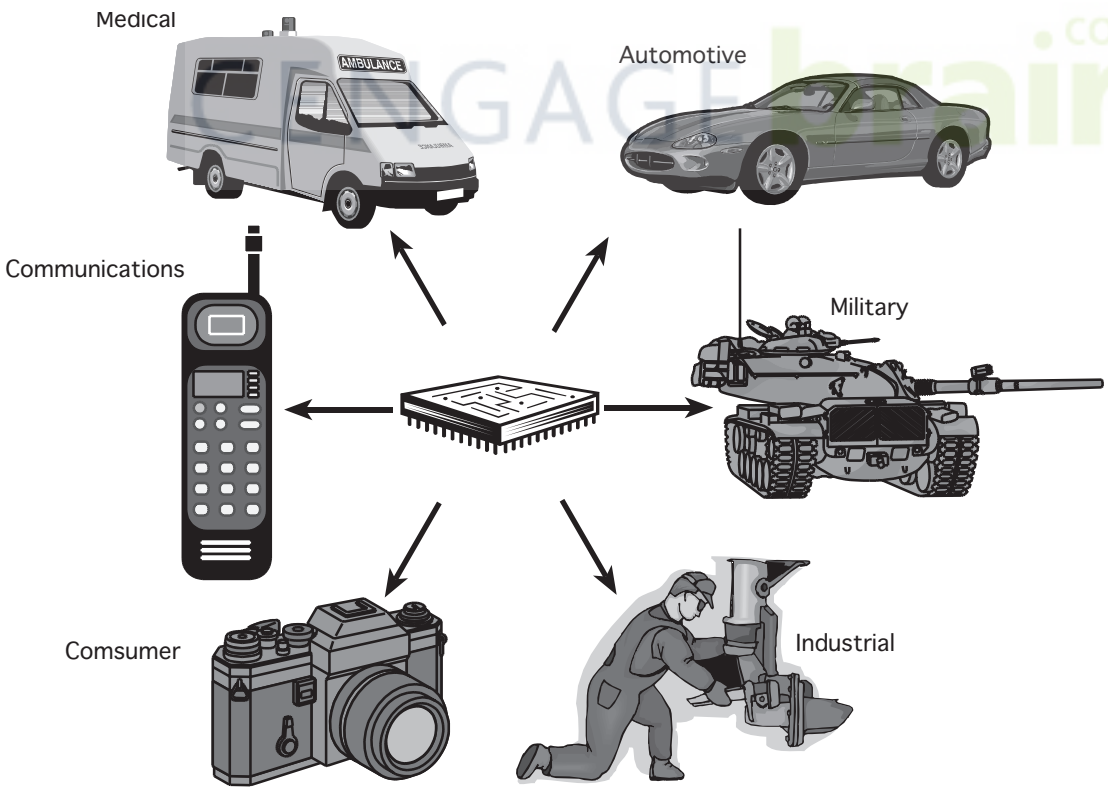


Figure 1.7
Example embedded computer systems.

Table 1.2 presents typical embedded microcomputer applications and the function performed by the embedded microcomputer. Each microcomputer accepts inputs, performs calculations, and generates outputs. We must also learn how to interface a wide range of inputs and outputs that can exist in either digital or analog form.

Checkpoint 1.7: There is a microcontroller embedded in an alarm clock. List three operations the software must perform.

In contrast, a *general-purpose computer system* typically has a keyboard, disk, and graphics display and can be programmed for a wide variety of purposes. Typical general-purpose applications include word processing, electronic mail, business accounting, scientific computing, and data base systems. General-purpose computers have the opposite of the four characteristics previously listed. First, they can perform a wide and dynamic range

Table 1.2
Embedded system applications.

	Function Performed by the Microcomputer
Consumer:	
Washing machine	Controls the water and spin cycles
Exercise equipment	Measures speed, distance, calories, heart rate
Remote controls	Accepts key touches, sends infrared pulses
Clocks and watches	Maintains the time, alarm, and display
Games and toys	Entertains the user, joystick input, video output
Audio/video electronics	Interacts with the operator, enhances performance
Communication:	
Phone answering machines	Plays outgoing and saves incoming messages
Telephone system	Switches signals and retrieves information
Cellular phones, pagers	Interacts with key pad, microphone, and speaker
Automotive:	
Automatic braking	Optimizes stopping on slippery surfaces
Noise cancellation	Improves sound quality, removing noise
Theft deterrent devices	Allows keyless entry, controls alarm
Electronic ignition	Controls sparks and fuel injectors
Power windows and seats	Remembers preferred settings for each driver
Instrumentation	Collects and provides necessary information
Military:	
Smart weapons	Recognizes friendly targets
Missile guidance systems	Directs ordnance at the desired target
Global positioning systems	Determines where you are on the planet
Surveillance	Collects information about enemy activities
Industrial:	
Point-of-sale systems	Accepts inputs and manages money
Set-back thermostats	Adjusts day/night thresholds saving energy
Traffic control systems	Senses car positions, controls traffic lights
Robot systems	Inputs from sensors, controls the motors
Inventory systems	Inputs from bar code readers, prints labels
Automatic sprinklers	Controls the wetness of the soil
Medical:	
Infant apnea monitors	Detects breathing, alarms if stopped
Glucose monitors	Measures blood sugar levels in diabetics
Cardiac monitors	Measures heart function, alarms if problem
Drug delivery	Administers proper doses
Cancer treatments	Controls doses of radiation, drugs, or heat
Pacemakers	Sends pulses to the heart to make it beat
Prosthetic devices	Increases mobility for the handicapped

of functions. Because the general-purpose computer has a removable disk or network interface, new programs can easily be added to the system. The user of a general-purpose computer does have access to the software that controls the machine. In other words, the user decides which operating system to run and which applications to launch. Second, they are loosely constrained. For example, the Java machine used by a web browser will operate on a extremely wide range of computer platforms. Third, general-purpose machines do not run in real time. Yes, we would like the time to print a page on the printer to be fast, and we would like web page to load quickly, but there are no guaranteed response times for these types of activities. In fact, the real-time tasks that do exist (such as sound recording, burning CDs, and graphics) are actually performed by embedded systems built into the computer. Fourth, general-purpose computers employ billions, if not trillions, of memory cells.

The most common type of general-purpose computer is the personal computer (e.g., the Pentium-based IBM-PC compatible and Macintosh). Computers more powerful than the personal computer can be grouped in the workstation ($10,000 to $50,000 range) or the supercomputer categories (above $50,000). See the web site **www.top500.org** for a list of the fastest computers on the planet. These computers often employ multiple processors and have much more memory than the typical personal computer. The workstations and supercomputers are used for handling large amounts of information (business applications), running large simulations (weather forecasting), searching (**www.google.com**), or performing large calculations (scientific research). This book will not cover the general-purpose computer, although many of the basic principles of embedded computers do apply to all types of systems.

The I/O interfaces are a crucial part of an embedded system because they provide necessary functionality. Most personal computers have the same basic I/O devices (mouse, keyboard, display, CD, USB, etc.) In contrast, there is no common set of I/O that all embedded system have. The software together with the I/O ports and associated interface circuits give an embedded computer system its distinctive characteristics. A *device driver* is a set of software functions that facilitate the use of an I/O port. Another name for device driver is *application programmer interface* (API). In this book we will study a wide range of I/O ports supported by the Freescale microcomputers. Parallel ports provide for digital input and/or outputs. Serial ports include the synchronous *Serial Peripheral Interface* (SPI) and asynchronous *Serial Communications Interface* (SCI), which have a wide range of software selectable baud rates and are optimized to minimize CPU overhead. The SPI also enables synchronous communication between the microcontroller and peripheral devices such as

Sensors
Liquid Crystal Display (LCD) and light emitting diode (LED) displays
Analog to digital converters (ADC) and digital to analog converters (DAC)
Other microprocessors

Analog to digital converters convert analog voltages to digital numbers, and they are available on the 9S12 with 8-bit and 10-bit resolution. The timer features on the Freescale microcomputers include

Fixed periodic rate interrupts
Computer Operating Properly (COP) protection against software failures
Pulse accumulator for external event counting or gated time accumulation
Pulse Width Modulated outputs (PWM)
Event counter system for advanced timing operations
Input capture used for period and pulse width measurement
Output compare used for generating signals and frequency measurement

1.3 The Design Process

1.3.1 Top-Down Design

In this section, we introduce the design process. The process is called *top-down*, because we start with the high-level designs and work down to low-level implementations. The basic approach is introduced here, and the details of these concepts are presented throughout the remaining chapters of the book. As we learn software/hardware development tools and techniques, we can place them into the framework presented in this section. As illustrated in Figure 1.8, the development of a product follows an analysis-design-implementation-testing cycle. For complex systems with long life-spans, we traverse multiple times around the development cycle. For simple systems, a one-time pass may suffice.

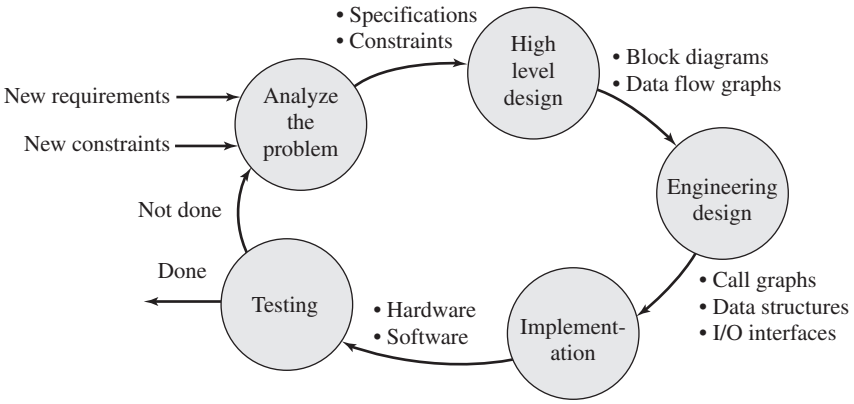


Figure 1.8
System development cycle.

During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information. A *requirement* is a general parameter that the system must satisfy. We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed *specifications*. In general, specifications are detailed parameters describing how the system should work. For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device. For example, suppose we wish to build a motor controller. During the analysis phase, we would determine obvious specifications such as range, stability, accuracy, and response time. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of operation, display readability, and reliability. Often, improving the performance of one parameter can be achieved only by decreasing the performance of another. This art of compromise defines the tradeoffs an engineer must make when designing a product. A *constraint* is a limitation, within which the system must operate. The system may be constrained as to such factors as cost, safety, compatibility with other products, use of specific electronic and mechanical parts as employed in other devices, interfaces with other instruments and test equipment, and development schedule. The following measures are often considered during the analysis phase of a project:

Safety: The risk to humans or the environment
Accuracy: The difference between desired and actual parameter performance
Precision: The number of distinguishable measurements
Resolution: The smallest change that can be reliably detected
Response time: The time difference between triggering event and resulting action
Bandwidth: The amount of information processed per time unit
Maintainability: The flexibility with which the device can be modified
Testability: The ease with which proper operation of the device can be verified

Compatibility: The conformity of the device to existing standards
Mean time between failure: The reliability of the device
Size and weight: The physical space required by the system
Power: The amount of energy it takes to operate the system
Nonrecurring engineering cost (NRE cost): The one-time cost to design and test the product
Unit cost: The cost required to manufacture one additional product
Time-to-prototype: The time required to design, build, and test an example system
Time-to-market: The time required to deliver the product to the customer
Human factors: The degree to which our customers enjoy or appreciate the product

Checkpoint 1.8: What's the difference between a requirement and a specification?

The following is one possible outline of a *software requirements document*. IEEE publishes a number of templates that can be used to define a project (IEEE STD 830-1998). A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.

1. **Overview**
   - **1.1. Objectives:** Why are we doing this project? What is the purpose?
   - **1.2. Process:** How will the project be developed?
   - **1.3. Roles and Responsibilities:** Who will do what? Who are the clients?
   - **1.4. Interactions with Existing Systems:** How will it fit in?
   - **1.5. Terminology:** Define terms used in the document.
   - **1.6. Security:** How will intellectual property be managed?
2. **Function Description**
   - **2.1. Functionality:** What will the system do precisely?
   - **2.2. Scope:** List the phases and what will be delivered in each phase.
   - **2.3. Prototypes:** How will intermediate progress be demonstrated?
   - **2.4. Performance:** Define the measures and describe how they will be determined.
   - **2.5. Usability:** Describe the interfaces. Be quantitative if possible.
   - **2.6. Safety:** Explain any safety requirements and how they will be measured.
3. **Deliverables**
   - **3.1. Reports:** How will the system be described?
   - **3.2. Audits:** How will the clients evaluate progress?
   - **3.3. Outcomes:** What are the deliverables? How do we know when the system is done?

During the **high-level design** phase, we build a conceptual model of the hardware and software system. It is in this model that we employ as much abstraction as appropriate. The project is broken in modules or subcomponents. Modular design will be presented in Chapter 2. During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide whether the project has a high enough potential for profit. A *data flow graph* is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components and the ovals are software modules. We use data flow graphs in the high-level design, because they describe the overall operation of the system while hiding the details of how it works. Issues such as safety (e.g., Isaac Asimov's first Law of Robotics: "*A robot may not harm a human being, or, through inaction, allow a human being to come to harm*") and testing (e.g., we need to verify that our system is operational) should be addressed during the high-level design.

An example data flow graph for a motor controller is shown in Figure 1.9. The requirement of the system is to deliver power to a motor so that the speed of the motor equals the desired value set by the operator using a keypad. In order to make the system easier to use and to assist in testing, a *liquid crystal display* (LCD) is added. The sensor converts motor speed into an electrical voltage. The amplifier converts this signal into the 0 to +5V voltage range required by the ADC. The ADC converts analog voltage into a digital sample. The ADC routines, using the ADC and timer hardware, collect samples and calculate voltages. Next, this software uses a table data structure to convert voltage into measured speed. The user will be able to select the desired speed using the Keypad interface. The desired and measured speed data are passed to the Controller software, which will adjust the power output in such a manner as to minimize the difference between the measured speed and the desired speed. Finally, the power commands are output to the *actuator* module. The actuator interface converts the digital control signals to power delivered to the motor. The measured speed and speed error will be sent to the LCD module. The solution to this problem will be presented in Chapter 13.

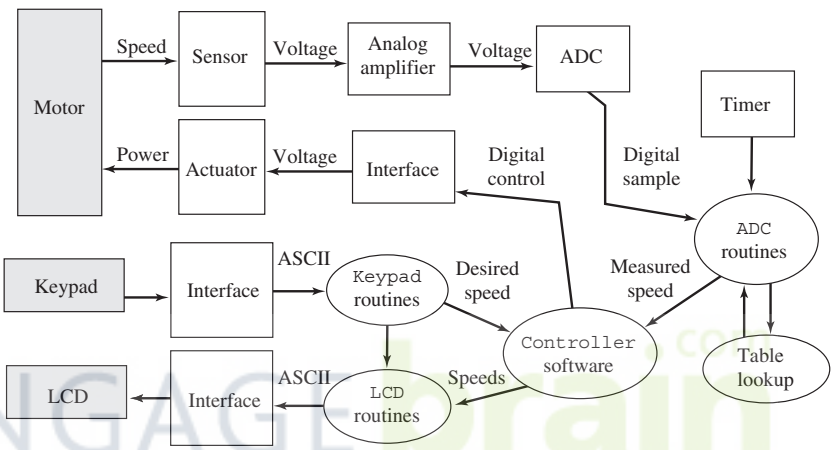


Figure 1.9
A data flow graph showing how signals pass through a motor controller.

The next phase is **engineering design**. We begin by constructing a preliminary design. This system includes the overall top-down hierarchical structure, the basic I/O signals, shared data structures, and the overall software scheme. At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design. Next, we finish the top-down hierarchical structure, and build mock-ups of the mechanical parts (connectors, chassis, cables, etc.) and user software interface. Sophisticated 3-D CAD systems can create realistic images of our system. Detailed hardware designs must include mechanical drawings. It is a good idea to have a *second source*, which is an alternative supplier that can sell us our parts if the first source can't deliver on time. *Call-graphs* are a graphical way to define how the software/hardware modules interconnect. A hierarchical system will have a tree-structured call graph. *Data structures* include both the organization of information and mechanisms to access the data. Again safety and testing should be addressed during this low-level design.

A call-graph for this motor controller is shown in Figure 1.10. Again, rectangles represent hardware components and ovals show software modules. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups and placed at the bottom of the graph. A high-level call-graph, like the one shown in Figure 1.10, shows only the high-level hardware/software modules. A detailed call-graph would include each software function and I/O port. Normally, hardware is passive and the software initiates hardware/software communication, but as we will learn in Chapter 4, it is possible for the hardware to interrupt the software and cause certain software modules to be run. In this system, the timer hardware will cause the `ADC` software to collect a sample at a regular rate.

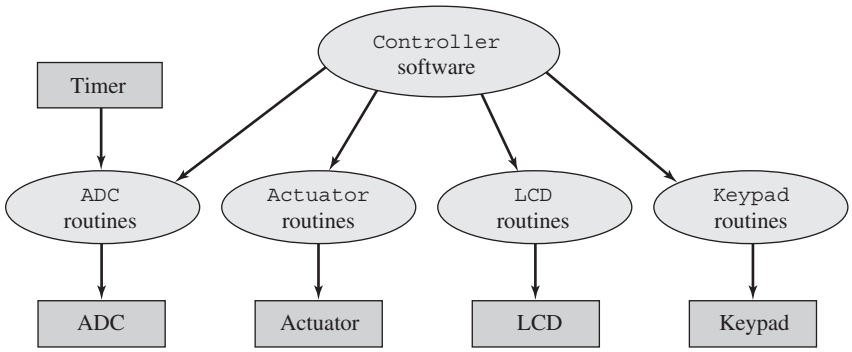


Figure 1.10
A call flow graph for a motor controller.

The `Controller` software calls the `Keypad` routines to get the desired speed, calls the `ADC` software to get the motor speed at that point, determines what power to deliver to the motor, and updates the actuator by sending the power value to the `Actuator` interface. The `Controller` software calls the `LCD` routines to display the status of the system. As we will see in Chapters 11–15, acquiring data, calculating parameters, and outputting results at a regular rate is strategic when performing digital processing in embedded systems.

Checkpoint 1.9: What confusion could arise if two software modules were allowed to access the same I/O port? This situation would be evident on a call-graph if the two software modules had arrows pointing to the same I/O port.

Observation: If module A calls module B, and B returns data, then a data flow graph will show an arrow from B to A, but a call-graph will show an arrow from A to B.

The next phase is **implementation**. An advantage of a top-down design is that implementation of subcomponents can occur concurrently. During the initial iterations of the development cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator than to construct a physical device out of actual components. Rapid prototyping is important in the early stages of product development. This allows for more loops around the analysis-design-implementation-testing cycle, which in turn leads to a more sophisticated product.

Recent software and hardware technological developments have made significant impacts on the software development process for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even through the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evaluate the performance of our system. First, we debug the system and validate basic functions. Next, we use careful measurements to optimize

performance, such as static efficiency (memory requirements), dynamic efficiency (execution speed), accuracy (difference between truth and measured), and stability (consistent operation). Debugging techniques are presented in Chapter 2.

Maintenance is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints. To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the development cycle.

1.3.2 Bottom-Up Design

Figure 1.8 describes top-down design as a cyclic process, beginning with a problem statement and ending up with a solution. With a *bottom-up* design we begin with solutions and build up to a problem statement. Many innovations begin with an idea, "what if . . .?" In a bottom-up design, one begins with designing, building, and testing low-level components. Figure 1.11 illustrates a two-level process, combining three subcomponents to create the overall product. This hierarchical process could have more levels and/or more components at each level. The low-level designs can occur in parallel. The design of each component is cyclic, iterating through the design-build-test cycle until the performance is acceptable. Bottom-up design may be inefficient because some subsystems may be designed, built, and tested but never used. As the design progresses the components are fit together to make the system more and more complex. Only after the system is completely built and tested does one define the overall system specifications. The bottom-up design process allows creative ideas to drive the products a company develops. It also allows one to quickly test the feasibility of an idea. If one fully understands a problem area and the scope of potential solutions, then a top-down design will arrive at an effective solution most quickly. On the other hand, if one doesn't really understand the problem or the scope of its solutions, a bottom-up approach allows one to start off by learning about the problem.

Observation: A good engineer knows both bottom-up and top-down design methods, choosing the approach most appropriate for the situation at hand.

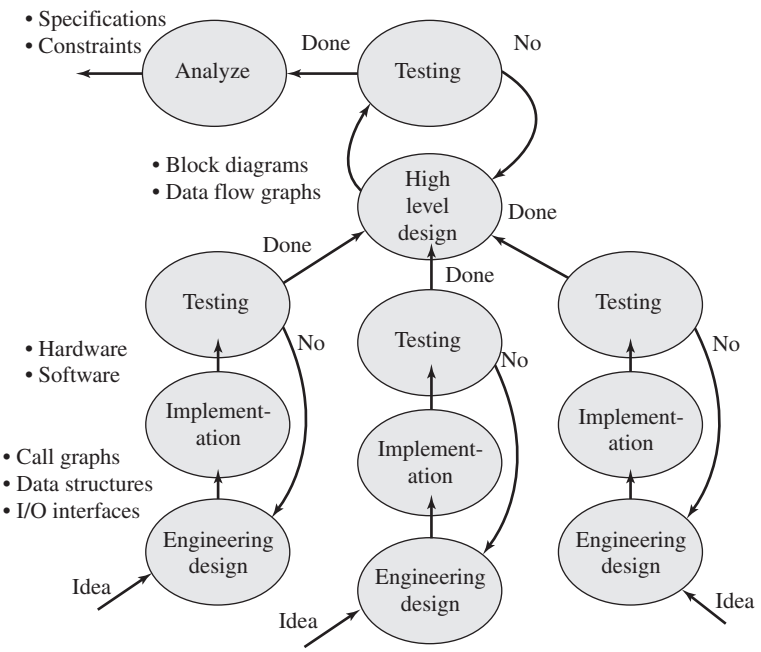


Figure 1.11 System development process illustrating bottom-up design.

1.4 Digital Logic and Open Collector

Normal digital logic has two states: high and low. There are four currents of interest, as shown in Figure 1.12, when analyzing whether the inputs of the next stage are loading the output. I_{IH} and I_{IL} are the currents required of an input when high and low, respectively. Similarly, I_{OH} and I_{OL} are the maximum currents available at the output when high and low. In order for the output to properly drive all the inputs of the next stage, the maximum available output current must be larger than the sum of all the required input currents for both the high and low conditions.

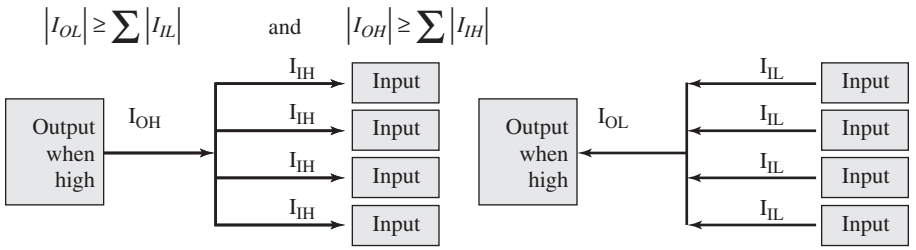


Figure 1.12
Sometimes one output must drive multiple inputs.

When we design circuits using devices all from a single logic family, we can define *fan out* as the maximum number of inputs one output can drive. For *transistor-transistor logic* (TTL) logic we can calculate *fan out* from the input and output currents:

$$\text{fan out} = \text{minimum } ((I_{OH}/I_{IH}), (I_{OL}/I_{IL}))$$

The fan out of high speed *complementary metal-oxide semiconductor* (CMOS) devices like the MC9S12C32 is determined by capacitive loading and not by the currents. Figure 1.13 shows a simple model of a CMOS interface. The ideal voltage of the output device is labeled V_1. For interfaces in close proximity, the resistance R results from the output impedance of the output device, and the capacitance C results from the input capacitance of the input device. However, if the interface requires a cable to connect the two devices, both the resistance and capacitance will be increased by the cable. The voltage labeled V_2 is the effective voltage as seen by the input. The *slew rate* of a signal is the slope of the voltage versus time during the time when the logic level switches between low and high. A similar parameter is the *transition time*, which is the time it takes for an output to switch from one logic level to another. In Figure 1.13, the transition time is defined as the time it takes V_2 to go from 1.75 to 3.25 V. There is a capacitive load for each CMOS input connected to a CMOS output. As this capacitance increases, the slew rate decreases, which will increase the transition time. The time constant of this simple circuit is $\tau = R*C$. Let T be the pulse width of the digital signal. If T is large compared

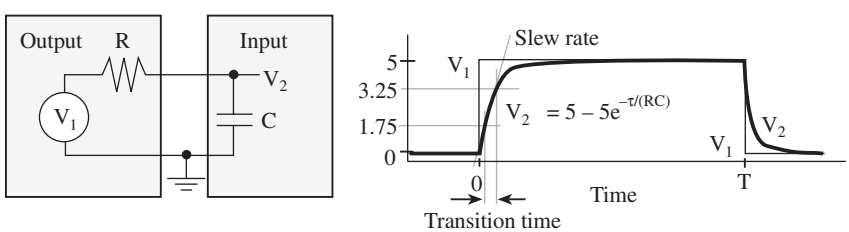


Figure 1.13
Capacitance loading is an important factor when interfacing CMOS devices.

to τ, then the CMOS interface functions properly. For circuits that mix devices from one family with another, we must look individually at the input and output currents, voltages, and capacitive loads. Table 1.3 shows typical current values for the various digital logic families. The MC9S12C32 allows full drive (a low R providing 10 mA) and reduced drive (a larger R providing a smaller 2 mA) modes.

Table 1.3
The input and output currents of various digital logic families and microcomputers.

Family	Example	I_{OH}	I_{OL}	I_{IH}	I_{IL}	Fan Out
Standard TTL	7404	0.4 mA	16 mA	40 μA	1.6 mA	10
Schottky TTL	74S04	1 mA	20 mA	50 μA	2 mA	10
Low-power Schottky TTL	74LS04	0.4 mA	4 mA	20 μA	0.4 mA	10
High-speed CMOS	74HC04	4 mA	4 mA	1 μA	1 μA	
Freescale microcomputer	MC68HC11E	0.8 mA	1.6 mA	1 μA	1 μA	
Freescale microcomputer	MC9S12C32	10 mA	10 mA	1 μA	1 μA	
Intel microcomputer	87C51 P0	7 mA	3.2 mA	10 μA	10 μA	
	87C51 P1,P2,P3	60 μA	1.6 mA		50 μA	

Observation: For TTL devices the logic low currents are much larger than the logic high currents.

Figure 1.14 compares the input and output voltages for many of the digital logic families. V_{IL} is the voltage below which an input is considered a logic low. Similarly, V_{IH} is the voltage above which an input is considered a logic high. V_{OH} is the output voltage when the signal is high. In particular, if the output is a logic high, and the current is less than I_{OH}, then the voltage will be greater than V_{OH}. Similarly, V_{OL} is the output voltage when the signal is low. In particular, if the output is a logic low and the current is less than I_{OL}, then the voltage will be less than V_{OL}. The maximum output current specification on the 9S12 is 25 mA, which is the current above which it will

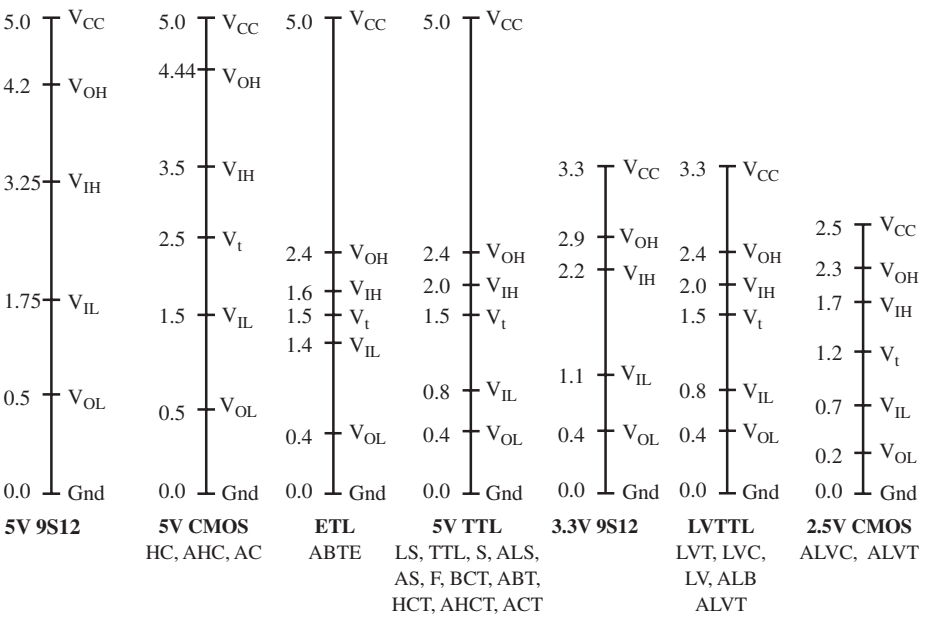


Figure 1.14
Voltage thresholds for various digital logic families.

cause damage. Normally, we design the system so the output currents are less than I_{OH} and I_{OL}. V_t is the typical threshold voltage, which is the voltage at which the input usually switches between logic low and high. Formally however, an input is considered as indeterminate for voltages between V_{IL} and V_{IH}. The five parameters that affect our choice of logic families are

- Power supply voltage (e.g., +5 V, 3.3 V etc.)
- Power supply current (e.g., will the system need to run on batteries?)
- Speed (e.g., clock frequency and propagation delays)
- Output drive, I_{OL}, I_{OH} (e.g., does it need to drive motors or lights?)
- Noise immunity (e.g., electromagnetic field interference)
- Temperature (e.g., 0 to 70 C)

Common error: If the voltage applied to a high-speed CMOS input pin exists between V_{IL} and V_{IH} for extended periods of time, permanent damage may occur.

Checkpoint 1.10: The 9S12 is an HC device. How will the 9S12 interpret an input pin as the input voltage changes from 0, 1, 2, 3, 4, to 5V? That is, for each voltage, will it be considered as a logic low, as a logic high, or as indeterminate?

Checkpoint 1.11: Considering both voltage and current, can the output of a 74HC04 drive the input of a 74LS04?

Checkpoint 1.12: Considering both voltage and current, can the output of a 74LS04 drive the input of a 74HC04?

A very important concept used in computer technology is *tristate logic*, which has three output states: high, low, and off. Other names for the off state are HiZ, floating, and tristate. As shown in Figure 1.15, the 74HC245 chip has eight bidirectional tristate drivers. The triangle shape, drawn with a signal on the top or the bottom of the triangle, is used to specify tristate control output in logic diagrams. The 74HC245 chip is active when the output enable, **OE**, input is low, and the direction is controlled by the **DIR** input. For example, if **OE**=0 and **DIR**=0, then the **B1**–**B8** are inputs and **A1**–**A8** are outputs, and each output $\mathbf{A_n}$ equals the corresponding $\mathbf{B_n}$ input. Conversely, if **OE**=0 and **DIR**=1, then the **A1**-**A8** are inputs and **B1**-**B8** are outputs, and each output $\mathbf{B_n}$ equals the corresponding $\mathbf{A_n}$ input. When **OE** is high, all the outputs will be off (floating). Devices like the 74HC245 are used in microcomputer systems because of the large output current and bidirectional tristate outputs. Tables 1.4 and 1.5 illustrate the wide range of technologies available for digital logic design. Not all logic families have the same choice of logic functions. $\mathbf{t_{pd}}$ is the propagation delay from input to output.

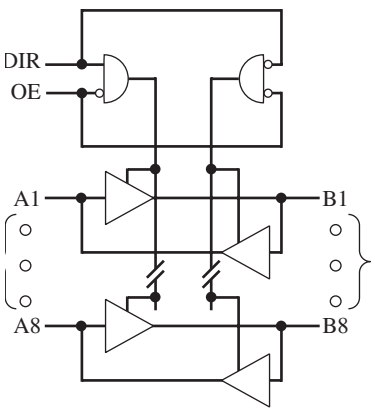


Figure 1.15
Block diagram of a 74HC245 tristate driver.

Family Technology	V_{IL} V_{IH}	V_{OL} V_{OH}	I_{OL}	I_{OH}	I_{CC}	t_{pd}
LVT—Low-Voltage BiCMOS	LVTTL	LVTTL	64	−32	190	3.5
ALVC—Advanced Low-Voltage CMOS	LVTTL	LVTTL	24	−24	40	3.0
LVC—Low-Voltage CMOS	LVTTL	LVTTL	24	−24	10	4.0
ALB—Advanced Low-Voltage BiCMOS	LVTTL	LVTTL	25	−25	800	2.0
AC—Advanced CMOS	CMOS	CMOS	12	−12	20	8.5
AHC—Advanced high-Speed CMOS	CMOS	CMOS	4	−4	20	11.9
LV—Low-Voltage CMOS	LVTTL	LVTTL	8	−8	20	14
Units			mA	mA	μA	ns

Table 1.4
Comparison of the output drive, power supply current, and speed of various 3.3 V logic '245 gates.

Family Technology	V_{IL} V_{IH}	V_{OL} V_{OH}	I_{OL}	I_{OH}	I_{CC}	t_{pd}
AHC—Advanced High-Speed CMOS	CMOS	CMOS	8	−8	0.04	7.5
AHCT—Advanced High-Speed CMOS	TTL	CMOS	8	−8	0.04	7.7
AC—Advanced CMOS	CMOS	CMOS	24	−24	0.04	6.5
ACT—Advanced CMOS	TTL	CMOS	24	−24	0.04	8.0
HC—High-Speed CMOS Logic	CMOS	CMOS	6	−6	0.08	21
HCT—High-Speed CMOS Logic	TTL	CMOS	6	−6	0.08	30
ABT—Advanced BiCMOS	TTL	TTL	64	−32	0.25	3.5
74F—Fast Logic	TTL	TTL	64	−15	120	6.0
BCT—BiCMOS	TTL	TTL	64	−15	90	6.6
AS—Advanced Schottky Logic	TTL	TTL	64	−15	143	7.5
ALS—Advanced Low-Power Schottky	TTL	TTL	24	−15	58	10
LS—Low Power Schottky logic	TTL	TTL	24	−15	95	12
S—Schottky Logic	TTL	TTL	64	−15	180	9
TTL—Transistor-Transistor Logic	TTL	TTL	16	−0.4	22	22
Units			mA	mA	mA	ns

Table 1.5
Comparison of the output drive, power supply current, and speed of various 5 V logic '245 gates.

The 74LS04 is a low-power Schottky NOT gate, as shown in Figure 1.16. It is called Schottky logic because the devices are made from Schottky transistors. The output is *high* when the transistor Q4 is active, driving the output to +5V. The output is *low* when the transistor Q5 is active, driving the output to 0.

The 74HC04 is a high-speed CMOS NOT gate, as shown in Figure 1.17. The output is *high* when the transistor Q1 is active, driving the output to +5V. The output is *low* when the transistor Q2 is active, driving the output to 0. Since the 9S12 is made with high-speed CMOS logic, its outputs behave like the Q1/Q2 "push/pull" transistor pair. The 9S12 output ports are not inverting. That is, when you write a "1" to an output port, then the output voltage goes high. Similarly, when you write a "0" to an output port, then the output voltage goes low. Analyses of the circuit in Figure 1.17 reveal some of the basic properties of high-speed CMOS logic. First, because of the complementary nature of the P-channel (the one on the top) and the N-channel (the one on the bottom) transistors, when the input is constant (continuously high or continuously low), the supply current, I_{cc}, is very low. Second, the gate will require supply current only when the output switches from low to high or from high to low. This observation leads to the design rule that the power required to run a high-speed CMOS system is linearly related to the frequency of its clock, because the frequency of the clock determines the number of transitions per second. Along the same lines,

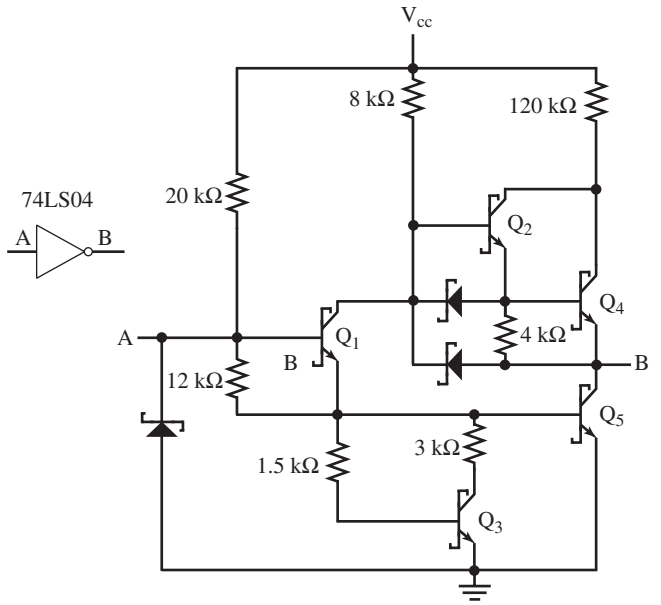


Figure 1.16
Transistor implementation of a low-power Schottky NOT gate.

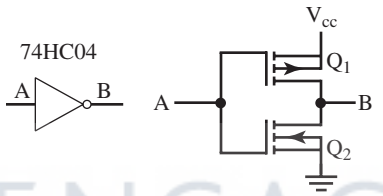


Figure 1.17
Transistor implementation of a high-speed CMOS NOT gate.

we see that if the voltage on input A exists between V_{IL} and V_{IH} for extended periods of time, then both Q1 and Q2 are partially active, causing a short from V_{cc} to ground. This condition can cause permanent damage to the transistors. Third, since the input A is connected to the gate of the two MOS transistors, the input currents will be very small (1 μA). In other words, the input impedance (input voltage divided by input current) of the gate is very high. Normally, a high input impedance is a good thing, except when the input is not connected. If the input is not connected, then it takes very little input currents to cause the logic level to switch.

> ***Common error:*** If unused input pins on a CMOS microcontroller are left unconnected, then the input signal may oscillate at high frequencies depending on the EM fields in the environment, wasting power unnecessarily.

> ***Maintenance tip:*** It is a good design practice to connect unused CMOS inputs to ground or connect them to +5 V.

Open collector logic has outputs with two states: low and off. The 7405 is a TTL open collector NOT gate, as shown in Figure 1.18. When drawing logic diagrams, we add the "x" on the output to specify open collector logic. It is called open collector because the collector pin of Q3 is not connected, or left open. The output is *off* when there is no active transistor driving the output. In other words, when the input is low, the output floats. This "not driven" condition is called the open collector state. The output is low when the transistor Q3 is active, driving the output to 0.

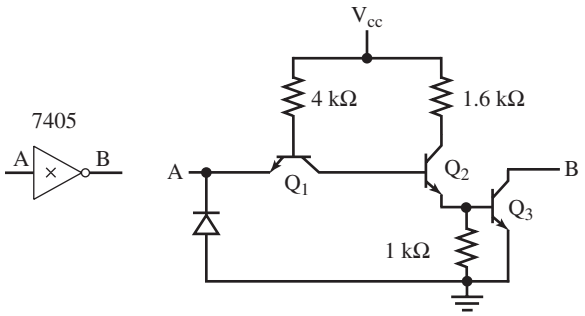


Figure 1.18
Transistor implementation of a regular TTL open collector NOT gate.

The 74HC05 is a high-speed CMOS open collector NOT gate, as shown in Figure 1.19. The output is *off* when there is no active transistor driving the output. In other words, when the input is low, the output floats. The output is low when the transistor Q2 is active driving the output to 0. Technically, the 74HC05 implements *open drain* rather than open collector, because it is the drain pin of Q2 that is left open. In this book, we will use the terms open collector and open drain interchangeably to refer to digital logic with two output states (low and off). The data sheets of the 9S12 refer to open collector logic as *wire or mode* (WOM).

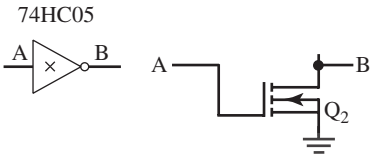


Figure 1.19
Transistor implementation of a high-speed CMOS open collector NOT gate.

Because of the multiple uses of open collector, many microcomputers can implement open collector logic. All the ports of the Intel 8051 are inherently open collector. The 9S12 PORTS also supports open collector outputs by setting the SWOM bit.

> ***Observation:*** The 7405 and 74HC05 are inverting open collector examples, but most microcomputer output ports are not inverting. That is, when you write a "1" to an open-collector output port, then the output floats, and when you write a "0" to an open-collector output port, then the output voltage goes low.

In general, we can use an **open collector NOT** gate to control the current to a device, such as a relay, a *light emitting diode* (LED), a solenoid, a small motor, or a small light. We used the open collector NOT gate in the LED interface shown in Figure 1.20 to control the current to our diode. When input to the 7405 is high (p = 1, which means +5 V), the output is low (q=0, which means 0 V). In this state, a 10 mA current is applied to the diode, and it lights up. But, when the input is low (p = 0, which means +0 V), the output floats (q = HiZ, which is neither high or low). This floating output state causes the LED current to be zero, and the diode is dark. The resistor choice will be explained in Chapter 8.

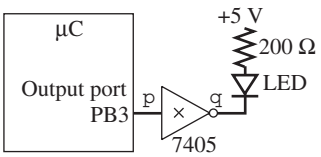


Figure 1.20
Open collector used to interface a light emitting diode.

When needed for digital logic, we can convert an open collector output to a digital signal using a pull-up resistor from the output to +5V. In this way, when the open collector output floats, the signal will be a digital high. How do we select the value of the pullup resistor? In general the smaller the resistor, the larger the I_{OH} it will be able to supply when the output is high. On the other hand, a larger resistor does not waste as much I_{OL} current when the output is low. One way to calculate the value of this pull-up resistor is to first determine the required output high voltage, V_{out}, and output high current, I_{out}. To supply a current of at least I_{out} at a voltage above V_{out}, the resistor must be less than:

$$R \leq (+5 - V_{out})/I_{out}$$

As an example, we will calculate the resistor value for the situation where the circuit needs to drive five regular TTL loads. We see from Figure 1.14 that V_{out} must be above V_{IH} (2V) in order for the TTL inputs to sense a high logic level. We can add a safety factor and set V_{out} at 3 V. In order for the high output to drive all five TTL inputs, I_{out} must be more than five I_{IH}. From Table 1.3, we see that I_{IH} is 40 μA, so I_{out} should be larger than 5•40 μA or 0.2 mA. For this situation the resistor must be less than 10 kΩ.

Another example of open collector logic occurs when interfacing switches to the microcontroller. The circuit in the left of Figure 1.21 shows a mechanical switch with one terminal connected to ground. In this circuit, when the switch is pressed, the voltage `r` is zero. When the switch is not processed, the signal `r` floats. The circuit in the middle of Figure 1.21 shows the mechanical switch with a 10 k pull-up resistor attached the other side. When the switch is pressed, the voltage at `s` still goes to zero, because the resistance of the switch (less than 0.1 Ω) is much less than the pull-up resistor. But now, when the switch is not pressed, the pull-up resistor creates a +5 V at `s`. This circuit is shown connected to an input pin of the microcontroller. The software, by reading the input port, can determine whether or not the switch is processed. If the switch is pressed, the software will read zero, and if the switch is not pressed, the software will read one. The circuit on the right of Figure 1.21 also interfaces a mechanical switch to the microcontroller, but it implements positive logic using a pull-down resistor. The signal `t` will be high if the switch is pressed and low if it is released.

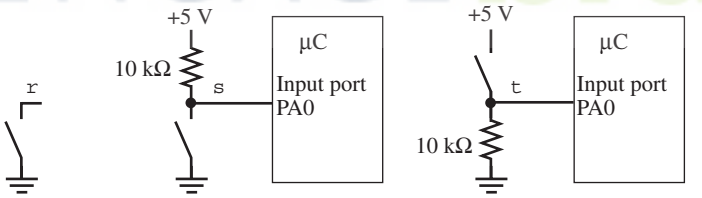


Figure 1.21
Switch interface.

Observation: Some of the ports on the 9S12 implement pull-up or pull-down resistors, so the interfaces shown in Figure 1.21 can be made without the resistor.

1.5 Digital Representation of Numbers

1.5.1 Fundamentals

Information is stored on the computer in binary form. A binary *bit* can exist in one of two possible states. In *positive logic*, the presence of a voltage is called the 1, true, asserted, or high state. The absence of a voltage is called the 0, false, not asserted, or low state. Figure 1.22 shows the output of a typical complementary metal-oxide semiconductor (CMOS) circuit. The left side shows the condition with a true bit, and the right side shows a false bit. The output of each digital circuit consists of a p-type transistor "on top of" an n-type transistor. In digital circuits, each transistor is essentially on or off. If the transistor is *on*, it is equivalent to a short circuit between its two output pins. Conversely, if the transistor is *off*, it is equivalent to an open circuit between its outputs pins.

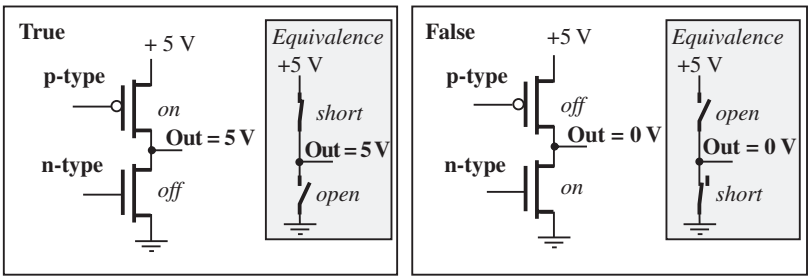


Figure 1.22
A binary bit is true if a voltage is present and false if the voltage is 0.

On a 9S12 powered with 5 V supply, a voltage between 3.25 and 5 V is considered high, and a voltage between 0 and 1.75 V is considered low. Separating the two regions by 1.5 V allows digital logic to operate reliably at very high speeds. The design of transistor-level digital circuits is beyond the scope of this book. However, it is important to know that digital data exist as binary bits and encoded as high and low voltages.

Numbers are stored on the computer in binary form. In other words, information is encoded as a sequence of 1's and 0's. On most computers, the memory is organized into 8-bit bytes. This means each 8-bit byte stored in memory will have a separate address. *Precision* is the number of distinct or different values. We express precision in alternatives, decimal digits, bytes, or binary bits. *Alternatives* are defined as the total number of possibilities. For example, an 8-bit number scheme can represent 256 different numbers. An 8-bit *digital to analog converter* (DAC) can generate 256 different analog outputs. An 8-bit *analog to digital converter* (ADC) can measure 256 different analog inputs. We use the expression $4^1/_2$ decimal digits to mean 20,000 alternatives and the expression 4¾ decimal digits to mean 40,000 alternatives. The ½ decimal digit means twice the number of alternatives or one additional binary bit. The ¾ decimal digit means four times as many alternatives or two additional binary bits. For example, a voltmeter with a range of 0.00 to 9.99 V has a three decimal digit precision. Let the operation [[x]] be the greatest integer of x. E.g., [[2.1]] is rounded up to 3. Tables 1.6 and 1.7 illustrate various representations of precision.

Table 1.6
Relationship between bits, bytes, and alternatives as units of precision.

Binary Bits	Bytes	Alternatives
8	1	256
10		1024
12		4096
16	2	65536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	[[n/8]]	2^n

Table 1.7
Definition of decimal digits as a unit of precision.

Decimal Digits	Alternatives
3	1000
$3^1/_2$	2000
$3^3/_4$	4000
4	10000
$4^1/_2$	20000
$4^3/_4$	40000
5	100000
n	10^n

Observation: A good rule of thumb to remember is $2^{10 \bullet n} \approx 10^{3 \bullet n}$.

Checkpoint 1.13: How many binary bits correspond to $2^1/_2$ decimal digits?

Checkpoint 1.14: About how many decimal digits can be presented in a 64-bit 8-byte number? You can answer this without a calculator, just using the "rule of thumb."

The *hexadecimal* number system uses base 16 as opposed to our regular decimal number system, which uses base 10. Hexadecimal is a convenient mechanism for humans to represent binary information, because it is extremely simple for us to convert back and forth between binary and hexadecimal. Hexadecimal number system is often abbreviated as "hex." A *nibble* is defined as 4 binary bits, which will be one hexadecimal digit. In mathematics, a subscript of 2 means binary, but in assembly language we will use the prefix **%** to signify binary numbers. The hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In assembly language, we use the prefix `$` to signify hexadecimal, and in C, we use the prefix `0x`. To convert from binary to hexadecimal, you simply separate the binary number into groups of four binary bits (starting on the right), then convert each group of four bits into one hexadecimal digit. For example, if you wished to convert `%10100111`, first you would group it into nibbles `1010 0111`, then you would convert each group `1010=A 0111=7`, yielding the result of `$A7`. To convert hexadecimal to binary, you simply substitute the 4-bit binary for each hexadecimal digit. For example, if you wished to convert `$B5D1`, you substitute `B=1011 5=0101 D=1101 1=0001`, yielding the result of `%1011010111010001`.

Checkpoint 1.15: Convert the binary number %111011101011 to hexadecimal.

Checkpoint 1.16: Convert the hex number $3800 to binary.

Checkpoint 1.17: How many binary bits does it take to represent $12345?

A great deal of confusion exists over the abbreviations we use for large numbers. In 1998, the International Electrotechnical Commission (IEC) defined a new set of abbreviations for the powers of 2, as shown in Table 1.8. These new terms are endorsed by the Institute of Electrical and Electronics Engineers (IEEE) and International Committee for Weights and Measures (CIPM) in situations where the use of a binary prefix is appropriate. The confusion arises over the fact that the mainstream computer industry (such as Microsoft, Apple, and Dell) continues to use the old terminology. According to the companies that market to consumers, a 1 GHz is 1,000,000,000 Hz, but 1 Gbyte of memory is 1,073,741,824 bytes. The correct terminology is to use the SI-decimal abbreviations to represent powers of 10 and the IEC-binary abbreviations to represent powers of 2. The scientific meaning of 2 kilovolts is 2000 volts, but 2 kibibytes is the proper way to specify 2048 bytes. The term **kibibyte** is a contraction of a kilo binary byte and is a unit of information or computer storage abbreviated KiB.

$$1 \text{ KiB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$$

$$1 \text{ MiB} = 2^{20} \text{ bytes} = 1{,}048{,}576 \text{ bytes}$$

$$1 \text{ GiB} = 2^{30} \text{ bytes} = 1{,}073{,}741{,}824 \text{ bytes}$$

Table 1.8 Common abbreviations for large numbers.

Value	SI	Decimal	Value	IEC	Binary
1000^1	k	kilo-	1024^1	Ki	kibi-
1000^2	M	mega-	1024^2	Mi	mebi-
1000^3	G	giga-	1024^3	Gi	gibi-
1000^4	T	tera-	1024^4	Ti	tebi-
1000^5	P	peta-	1024^5	Pi	pebi-
1000^6	E	exa-	1024^6	Ei	exbi-
1000^7	Z	zetta-	1024^7	Zi	zebi-
1000^8	Y	yotta-	1024^8	Yi	yobi-

These abbreviations also can be used to specify the number of binary bits. The term **kibibit** is a contraction of kilo binary bit and is a unit of information or computer storage abbreviated Kibit.

$$1 \text{ Kibit} = 2^{10} \text{ bits} = 1024 \text{ bits}$$

$$1 \text{ Mibit} = 2^{20} \text{ bits} = 1{,}048{,}576 \text{ bits}$$

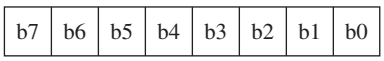
$$1 \text{ Gibit} = 2^{30} \text{ bits} = 1{,}073{,}741{,}824 \text{ bits}$$

A **mebibyte** (1 MiB is 1,048,576 bytes) is approximately equal to a megabyte (1 MB is 1,000,000 bytes), but mistaking the two has nonetheless led to confusion and even legal disputes. In the engineering community, it is appropriate to use terms that have a clear and unambiguous meaning.

1.5.2 8-Bit Numbers

A byte contains 8 bits as shown in Figure 1.23, where each bit $b_7, \ldots, b_0$ is binary and has the value 1 or 0. We specify b_7 as the *most significant bit* or MSB, and b_0 as the least significant bit or LSB.

Figure 1.23
8-bit binary format.



If a byte is used to represent an unsigned number, then the value of the number is

$$N = 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

Notice that the significance of bit n is 2^n. There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0, and the largest is 255. For example, %00001010 is 8+2 or 10.

Checkpoint 1.18: Convert the binary number %01101010 to unsigned decimal.

Checkpoint 1.19: Convert the hex number $32 to unsigned decimal.

The *basis* of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. The basis represents the "places" in a "place-value" system. For positive integers, the basis is the infinite set {1, 10, 100, ...}, and the "values" can range from 0 to 9. Each positive integer has a unique set of values such that the dot-product of the **value-vector** times the **basis-vector** yields that number. For example, 2345 is (. . ., 2,3,4,5)•(. . ., 1000,100,10,1), which is 2*1000+3*100+4*10+5. For the unsigned 8-bit number system, the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128\}$$

The values of a binary number system can only be 0 or 1. Even so, each 8-bit unsigned integer has a unique set of values such that the dot-product of the values times the basis yields that number. For example, 69 is (0,1,0,0,0,1,0,1)•(128,64,32,16,8,4,2,1), which equals 0*128+1*64+0*32+0*16+0*8+1*4+0*2+1*1.

Checkpoint 1.20: Give the representations of decimal 35 in 8-bit binary and hexadecimal.

Checkpoint 1.21: Give the representations of decimal 200 in 8-bit binary and hexadecimal.

One of the first schemes to represent signed numbers was called *one's complement*. It was called one's complement because to negate a number, you complement (logical not) each bit. For example, if 25 equals 00011001 in binary, then −25 is 11100110. An 8-bit one's complement number can vary from −127 to +127. The most significant bit is a sign bit, which is

1 if and only if the number is negative. The difficulty with this format is that there are two zeros +0 is 00000000, and −0 is 11111111. Another problem is that ones complement numbers do not have basis elements. These limitations led to the use of two's complement.

The *two's complement* number system is the most common approach used to define signed numbers. It was called two's complement because to negate a number, you complement each bit (like one's complement), then add 1. For example, if 25 equals 00011001 in binary, then −25 is 11100111. If a byte is used to represent a signed two's complement number, then the value of the number is

$$N = -128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

Observation: One usually means two's complement when one refers to signed integers.

There are 256 different signed 8-bit numbers. The smallest signed 8-bit number is −128, and the largest is 127. For example, %10000010 equals −128+2 or −126.

Checkpoint 1.22: Are the signed and unsigned decimal representations of the 8-bit hex number $35 the same or different?

For the signed 8-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, -128\}$$

Observation: The most significant bit in a two's complement signed number will specify the sign.

Notice that the same binary pattern of %11111111 could represent either 255 or −1. It is very important for the software developer to keep track of the number format. The computer can not determine whether the 8-bit number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly instructions you select to operate on the number. Some operations like addition, subtraction, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, multiply, divide, and shift right (divide by 2) require separate hardware (instruction) for unsigned and signed operations. For example, the 9S12 multiply instruction, `mul`, operates only on unsigned values. So if you use the `mul` instruction, you are implementing unsigned arithmetic. The 9S12 has both unsigned, `emul`, and signed, `emuls`, multiply instructions. So if you use the `emuls` instruction, you are implementing signed arithmetic.

Observation: To take the negative of a two's complement signed number we first complement (flip) all the bits, then add 1.

Checkpoint 1.23: Give the representations of –35 in 8-bit binary and hexadecimal.

Checkpoint 1.24: Why can't you represent the number 200 using 8-bit signed binary?

Common error: An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers.

Maintenance tip: To improve the clarity of your software, always specify the format of your data (signed versus unsigned) when defining or accessing the data.

1.5.3 Character Information

We can use bytes to represent characters with the **American Standard Code for Information Interchange** (ASCII) code. Standard ASCII is actually only 7 bits, but is stored using 8-bit bytes with the most significant byte equal to 0. Some computer systems use the 8th bit of the ASCII code to define additional characters such as graphics and letters in other alphabets. The 7-bit ASCII code definitions are given in the Table 1.9. For example, the letter "V" is in the $50 row and the 6 column. Putting the two together yields hexadecimal $56.

Table 1.9
Standard 7-bit ASCII.

		BITS 4 to 6						
		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
B	1	SOH	XON	!	1	A	Q	a	q
I	2	STX	DC2	"	2	B	R	b	r
T	3	ETX	XOFF	#	3	C	S	c	s
S	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
0	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
3	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	S1	US	/	?	O	_	o	DEL

Checkpoint 1.25: How is the character 0 represented in ASCII?

One way to encode a character string is to use null-termination. In this way, the characters of the string are stored one right after the other, and the end of the string is signified by the NUL character (0). For example, the string "Valvano" is encoded as the following eight bytes: $56,$61,$6C,$76,$61,$6E,$6F,$00.

Checkpoint 1.26: How is "Hello World" encoded as a null-terminated ASCII string?

1.5.4 16-Bit Numbers

A *word* or *double byte* contains 16 bits, where each bit b15, . . . , b0 is binary and has the value 1 or 0, as shown in Figure 1.24.

Figure 1.24
16-bit binary format.

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

If a word is used to represent an unsigned number, then the value of the number is

$$N = 32768 \bullet b_{15} + 16384 \bullet b_{14} + 8192 \bullet b_{13} + 4096 \bullet b_{12} + 2048 \bullet b_{11} + 1024 \bullet b_{10} + 512 \bullet b_9 + 256 \bullet b_8 + 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

There are 65536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0, and the largest is 65535. For example, %0010000110000100 or $2184 is 8192+256+128+4 or 8580. For the unsigned 16-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$$

There are also 65536 different signed 16-bit numbers. The smallest two's complement signed 16-bit number is −32768 and the largest is 32767. For example, %1101000000000100 or $D004 is −32768+16384+4096+4 or −12284. If a word is used to represent a signed two's complement number, then the value of the number is

$$N = -32768 \bullet b_{15} + 16384 \bullet b_{14} + 8192 \bullet b_{13} + 4096 \bullet b_{12} + 2048 \bullet b_{11} + 1024 \bullet b_{10} + 512 \bullet b_9 + 256 \bullet b_8 + 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

For the signed 16-bit number system the basis is

$$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768\}$$

Common error: An error will occur if you use 16-bit operations on 8-bit numbers, or use 8-bit operations on 16-bit numbers.

Maintenance tip: To improve the clarity of your software, always specify the precision of your data when defining or accessing the data.

When we store 16-bit data into memory, it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the *big endian* approach, which stores the most significant part first. Intel microcomputers implement the *little endian* approach, which stores the least significant part first. The PowerPC is *biendian*, because it can be configured to efficiently handle both big and little endian. Figure 1.25 shows two ways to store the 16-bit number 1000 ($03E8) at locations $50–$51.

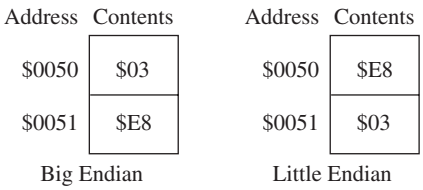


Figure 1.25 Example of big and little endian formats of a 16-bit number.

We also can use either the big endian or the little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. Figure 1.26 shows the big and little endian formats that could be used to store the 32-bit number $12345678 at locations $50–$53.

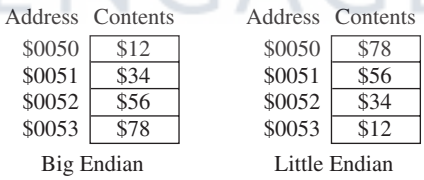


Figure 1.26 Example of big and little endian formats of a 32-bit number.

In the foregoing two examples, we normally would not pick out individual bytes (e.g., the $12) but rather capture the entire multiple byte data as one nondivisible piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big and little endian schemes store the data in first-to-last sequence. For example, if we wish to store the four ASCII characters '9S12' as a string, which is $3953313200 at locations $50–$54, then the ASCII '9' =$39 comes first in both big and little endian schemes.

The terms "big and little endian" comes from Jonathan Swift's satire *Gulliver's Travels*. In Swift's book, a Big Endian refers to people who crack their egg on the big end. The Lilliputians were Little Endians, because they insisted that the only proper way is to break an egg on the little end. The Lilliputians considered the Big Endians as inferiors. The Big and Little Endians fought a long and senseless war over which end is best to crack an egg.

1.5.5 Fixed-Point Numbers

We will use fixed-point numbers when we wish to express values in our software that have noninteger values. A **fixed-point number** contains two parts. The first part is a **variable integer**, called **I**. This integer may be signed or unsigned. An unsigned fixed-point number

is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On the 9S12, we typically use 8 bits or 16 bits. Extended precision can be implemented, but the execution speed will be slower because the calculations will have to be performed using software algorithms rather than hardware instructions. This integer part is saved in memory and is manipulated by software. These manipulations include but are not limited to add, subtract, multiply, divide, convert to BCD and convert from BCD. The second part of a fixed-point number is a **fixed constant**, called . This value is fixed, and cannot be changed during execution of the program. The fixed constant is not stored in memory. Usually we specify the value of this fixed content using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the two parts:

fixed-point number **I•**

The **resolution** of a number is the smallest difference that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant ( ). Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001V is really the same thing as an integer with units of mV. When interacting with a human operator, it is usually convenient to use **decimal fixed-point**. With decimal fixed-point the fixed constant is a power of 10.

$$\text{decimal fixed-point number} = \mathbf{I} \bullet 10^{\mathbf{m}} \text{ for some constant integer m}$$

Again, the integer **m** is fixed and is not stored in memory. Decimal fixed-point is easy to display, whereas **binary fixed-point** is easier to use when performing mathematical calculations. With binary fixed-point the fixed constant is a power of 2.

$$\text{binary fixed-point number} = \mathbf{I} \bullet 2^{\mathbf{n}} \text{ for some constant integer n}$$

Observation: If the range of numbers is known and small, then the numbers can be represented in a fixed-point format.

Checkpoint 1.27: Give an approximation of , using the decimal fixed-point (= 0.001) format.

Checkpoint 1.28: Give an approximation of , using the binary fixed-point ($= 2^{-8}$) format.

An 8-bit ADC has a range of 0 to +5 V, and its resolution is about 5 V/256 or 0.02 V. It would be appropriate to store voltages as 16-bit unsigned fixed-point numbers with a resolution of 0.01V. For the 9S12C32, which has a 10-bit ADC and a range of 0 to +5 V, its resolution is about 5 V/1024 or 0.005 V. It would be appropriate to store voltages on the 9S12C32 as 16-bit unsigned fixed-point numbers with a resolution of 0.001 V. The resolution is choosen so that no information is lost.

It is very important to carefully consider the order of operations when performing multiple integer calculations. For example, assume we wished to calculate M=(53*N)/100, where M and N are integers. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. In this example, if N is an 8-bit unsigned number, then 53*N can overflow the 0 to 255 range. One solution of the overflow problem is promotion. **Promotion** is the action of increasing the inputs to a higher precision, performing the calculation at the higher precision, checking for overflow, then demoting the result back to the lower precision. In this example, the 53, N, and 100 are all converted to 16-bit unsigned numbers. (53*N)/100 is calculated in 16-bit precision. The result can be verified to be in the 0 to 255 range, then converted back to 8-bit precision. The other error is called **drop-out**. Drop-out occurs during a right shift or a divide, and the consequence is that an intermediate

result loses its ability to represent all of the values. To avoid drop-out, it is very important to divide last when performing multiple integer calculations. If we divided first, e.g., M=53*(N/100), then the values of M would be only 0, 53, or 106. We could have calculated M=(53*N+50)/100 to implement rounding to the closest integer. The value 50 is selected because it is about one half of the divisor.

When adding or subtracting two fixed-point numbers with the same , we simply add or subtract their integer parts. First, let x,y,z be three fixed-point numbers with the same . Let x=I• , y=J• , and z=K• . To perform z=x+y, we simply calculate K=I+J. Similarly, to perform z=x−y, we simply calculate K=I−J. When adding or subtracting fixed-point numbers with different fixed parts, we must first convert two the inputs to the format of the result before adding or subtracting. This is where binary fixed-point is more convenient, because the conversion process involves shifting rather than multiplication/division.

For multiplication, we have z=x•y. Again, we substitute the definitions of each fixed-point parameter and solve for the integer part of the result

$$K=I•J•$$

For division, we have z=x/y. Again, we substitute the definitions of each fixed-point parameter and solve for the integer part of the result

$$K=I/J/$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about overflow and drop-out.

We can use these fixed-point algorithms to perform complex operations using the integer functions of our 9S12. For example, consider the following digital filter calculation.

$$y=x-0.0532672•x_1+x_2+0.0506038•y_1-0.9025•y_2$$

In this case, the variables y, y_1, y_2, x, x_1, and x_2 are all integers, but the constants will be expressed in binary fixed-point format. The value −0.0532672 will be approximated by $-14•2^{-8}$. The value 0.0506038 will be approximated by $13•2^{-8}$. Lastly, the value −0.9025 will be approximated by $-231•2^{-8}$. The fixed-point implementation of this digital filter is

$$y=(256•x\ -14•x_1+256•x_2\ +13•y_1-231•y_2)\ >>\ 8$$

Common error: Lazy or incompetent programmers use floating-point in many situations where fixed-point would be preferable.

Observation: As the fixed constant is made smaller, the accuracy of the fixed-point representation is improved, but the variable integer part also increases. Unfortunately, larger integers require more bits for storage and calculations.

Checkpoint 1.29: Using a fixed constant of 10^{-3}, rewrite the digital filter $y=x-0.0532672•x_1+x_2+0.0506038•y_1-0.9025•y_2$ in decimal fixed-point format.

1.6 Common Architecture of the 9S12

In 1968, two unhappy engineers named Bob Noyce and Gordon Moore left the Fairchild Semiconductor Company and created their own company, which they called Integrated Electronics (Intel). Working for Intel in 1971, Federico Faggin, Ted Hoff, and Stan Mazor invented the first single-chip microprocessor, the Intel 4004 (Figure 1.27). It was a four-bit processor designed to solve a very specific application for a Japanese company called Busicon. Busicon backed out of the purchase, so Intel decided to market it as a "general purpose" microprocessing system. The product was a success, which lead to two more powerful microprocessors: the Intel 8008 in 1974, and the Intel 8080 also in 1974. Both the Intel 8008 and the Intel 8080 were 8-bit microprocessors using N-channel metal-oxide semiconductor (NMOS) technology. Seeing the long-term potential for this technology,

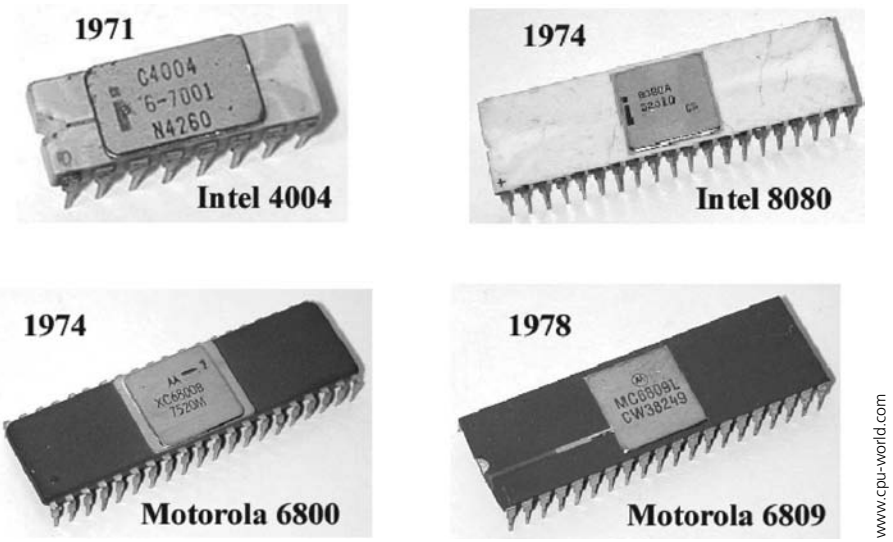


Figure 1.27
The first microprocessors (**http://www.cpu-world.com**).

Motorola released its MC6800 in 1974, which was also an 8-bit processor with about the same capabilities as the 8080. Although similar in computing power, the 8080 and 6800 had very different architectures. The 8080 used isolated I/O and handled addresses in a fundamentally different way from data. Isolated I/O defines special hardware signals and special instructions for input/output. On the 8080, certain registers had capabilities designed for addressing, whereas other registers had capabilities for data manipulation. In contrast, the 6800 used memory-mapped I/O and handled addresses and data in a similar way. As defined previously, input/output on a system with memory-mapped I/O is performed in a manner similar to accessing memory.

During the 1980s and 1990s, Motorola (von Neumann architecture) and Intel (Harvard architecture) traveled down similar paths. The microprocessor families from both companies developed bigger and faster products: Intel 8085, 8088, 80x86, . . . and the Motorola 6809, 68000, 680x0. . . . During the early 1980's another technology emerged—the microcontroller. In sharp contrast to the microprocessor family, which optimized computational speed and memory size at the expense of power and physical size, the microcontroller devices minimized power consumption and physical size, striving for only modest increases in computational speed and memory size. Out of the Intel architecture came the 8051 family (**www.semiconductors.philips.com**), and out of the Motorola architecture came the 6805, 6811, and 6812 microcontroller family (**www. freescale.com**). Many of the same fundamental differences that existed between the original 8-bit Intel 8080 and Motorola 6800 have persisted during forty years of microprocessor and microcontroller developments. In 1999, Motorola shipped its 2 billionth MC68HC05 microcontroller. In 2004, Motorola spun off its microcontroller products as Freescale Semiconductor, and remained No. 2 in market share in microcontrollers overall and No. 1 in market share in microcontrollers for automotive applications (Gartner Dataquest). Microchip is the No. 1 supplier of 8-bit microcontrollers.

In this section, common features of the 9S12 are presented. In subsequent sections, information specific to each processor is presented.

1.6.1 Registers

The 9S12 registers are depicted in Figure 1.28. Registers A and B concatenated together form a 16-bit accumulator, Register D, with Register A containing the most significant byte. Typically Registers A and B contain data (numbers) whereas Registers X and Y contain addresses (pointers.) On the 9S12, the SP (stack pointer) points to the top element of the stack. Register PC (program counter) points to the current instruction.

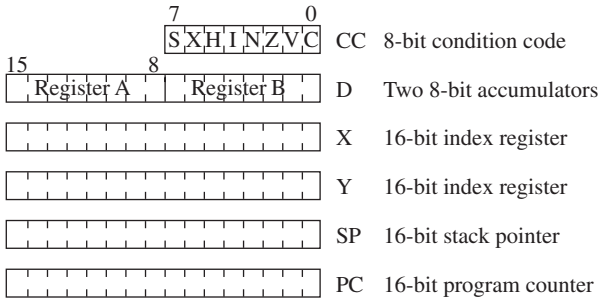


Figure 1.28
The 9S12 has six registers.

The condition code bits are shown in Figure 1.29. The N, Z, V, and C bits signify the status of the previous ALU operation. Many instructions set these bits to signify the result of the operation. When S=1, the stop instruction is disabled. When X=0, XIRQ interrupts are allowed. Once X is set to zero, the software cannot set it back to 1. The H bit is used for binary coded decimal (BCD) addition. When I=0, IRQ interrupts are enabled. Interrupts are discussed in Chapter 4.

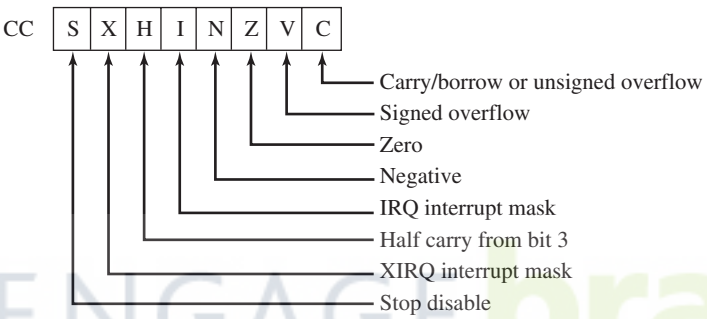


Figure 1.29
The 9S12 condition code bits.

Checkpoint 1.30: To prevent (disable) interrupts, what value should the I bit be?

Checkpoint 1.31: List all the registers that can hold 16-bit addresses.

1.6.2 Terminology

This chapter focuses on the 9S12 architecture, but now we introduce a few simple instructions in order to understand how the microcomputer works. When describing the action of an assembly instruction we will use the following notation.

```
w is a signed 8-bit -128 to +127 or unsigned 8-bit 0 to 255
n is a signed 8-bit -128 to +127
u is a unsigned 8-bit 0 to 255
W is a signed 16-bit -32787 to +32767 or unsigned 16-bit 0 to 65535
N is a signed 16-bit -32787 to +32767
U is a unsigned 16-bit 0 to 65535
=[addr] specifies an 8-bit read from addr
={addr} specifies a 16-bit read from addr using "big endian"
=<addr> specifies a 32-bit read from addr using "big endian"
[addr]= specifies an 8-bit write to addr
{addr}= specifies a 16-bit write to addr using "big endian"
<addr>= specifies a 32-bit write to addr using "big endian"
```

Assembly language instructions have four fields. The *label field* is optional and starts in the first column; it is used to identify the position in memory of the current instruction. You must choose a unique name for each label. The *opcode field* specifies the microcomputer

command to execute. The *operand field* specifies where to find the data to execute the instruction. We will see that opcodes have 0, 1, 2, or 3 operands. The *comment field* is also optional and is ignored by the computer, but it allows you to describe the software making it easier to understand. Good programmers add comments to explain the software. The `ldaa` instruction reads 8 bits of data from memory and places them in register A. The `staa` instruction stores the 8-bit value from register A into memory. The `ldx` instruction reads 16 bits of data from memory and places them in register X. The `stx` instruction stores the 16-bit value from register X into memory. The first two assembly instructions copy the contents of Port A (location 0 on the 9S12) to memory location $3800. The next two assembly instructions move a 16-bit value from locations $3802–$3803 into locations $3804–$3805.

```
label    opcode    operand    comment

here     ldaa      $0000      ;RegA=[$0000]
         staa      $3800      ;[$3800]=RegA
         ldx       $3802      ;RegX={3802}
         stx       $3804      ;{3804}=RegX
```

As we will learn in further along in the book, it is much better to add comments to explain how—or even better, why—we do the action. But for now we are learning what the instruction is doing, so in this chapter, comments will describe what the instruction does. The assembly language instructions (like the forgoing example) are translated into machine instructions. The `ldaa $0000` instruction is translated into 2 bytes of machine code:

```
Object code    instruction      comment

$96 $00        ldaa $0000       ;RegA=[$0000]
```

1.6.3 Addressing Modes

A fundamental issue in program development is the differentiation between data and address. It is in assembly language programming in general and addressing modes in specific that this differentiation becomes clear. When we put the number 1000 into register X, whether this is data or address depends on how the 1000 is used. Most instructions access memory to fetch parameters or save results. The addressing mode is the format the instruction uses to specify the memory location to read or write data. All instructions begin by fetching the machine instruction (opcode and operand) pointed to by the PC. Some instructions operate completely within the processor and require no memory data fetches. These instructions have no operand and are classified as *inherent*. For example, the `clra` instruction places a zero into register A. If the data is found in the instruction itself, the instruction uses *immediate* addressing mode. If the instruction uses the absolute address to specify the memory data location, the instruction uses either *direct* or *extended* addressing mode. Notice in Table 1.10 that the `ldaa` instruction can be used with the immediate, direct, and extended addressing modes. In particular, the `ldaa` instruction means "load into register A" and the addressing mode specifies the source of the data to be used. Many computers, including the 9S12, use *PC-relative* addressing mode to encode branch instructions. PC-relative addressing makes the object code smaller and relocatable. Normally, the computer executes one instruction after another as they are listed in memory, except the branch instructions, which cause the program to jump to another place. For example, the `bra $F042` instruction is an unconditional branch, causing the program to jump to location

Table 1.10
Simple addressing modes.

9S12 code	opcode	operand	comment
$87	clra		;Reg A = 0 (inherent)
$86 24	ldaa	#36	;Reg A = $24 (immediate)
$96 24	ldaa	36	;Reg A = [$0024] (direct)
$B6 08 01	ldaa	$0801	;Reg A = [$0801] (extended)
$20 $40	bra	$F042	;Jump to $F042

$F042. The addressing mode is called PC-relative because the machine code contains the address difference between where the program is now and the address to which the program will jump. There are many more addressing modes, but for now, these five addressing modes, as illustrated in Table 1.10, are enough to get us started.

Checkpoint 1.32: What is the addressing mode used for?

In this section, these five addressing modes are introduced. These simple addressing modes are sufficient to understand most of the software presented in this book. The more complicated addressing modes are presented in Chapter 2.

1. Inherent addressing mode has no operand field. Sometimes there is no data for the instruction at all. For example, the `stop` instruction halts execution. Sometimes the data for the instruction is implied. For example, the `clrb` instruction sets register B to zero. In this case, the data value of zero is implied. On the other hand, sometimes the data must be fetched from memory, but the address of the data is implied. For example, the `pula` instruction will pop an 8-bit data from the stack and store it in register A. In particular, the data value pointed to by the SP is read from memory and stored into register A.

2. Immediate addressing mode uses a fixed data constant. The data itself is included in the machine code. For example, the `ldaa #36` instruction will store a data value of 36 into register A (Figure 1.30). Notice that the "36" itself is encoded in the machine code for the `ldaa #36` instruction. In assembly code, this mode is signified by the # sign.

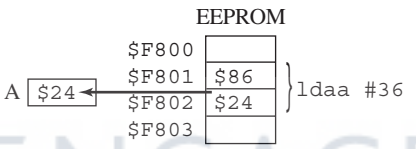


Figure 1.30
Example of the immediate addressing mode.

Observation: With immediate mode addressing, the information is stored in the machine code.

Common error: It is illegal to use the immediate addressing mode with instructions that store data into memory (e.g., staa).

Checkpoint 1.33: What is the difference between `ldaa #36` and `ldaa #$24`?

3. Direct-page addressing mode uses an 8-bit address to access from addresses 0 to $00FF. In many computer systems outside the Freescale family, this addressing mode is called *zero-page*. On the 9S12, they reference the I/O ports. In assembly language, the **<** operator forces direct addressing. Figure 1.31 illustrates the execution of the `ldaa 36` instruction.

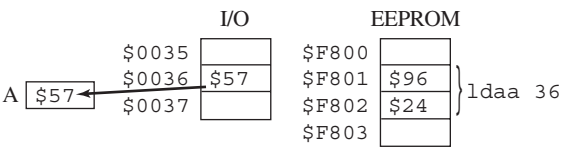


Figure 1.31
Example of the direct-page addressing mode.

Observation: With direct and extended mode addressing, a fixed pointer to the information is stored in the machine code. The data itself may change dynamically, but its location is fixed.

Checkpoint 1.34: What is the difference between `ldaa #36` and `ldaa 36`?

4. Extended addressing mode uses a 16-bit address to access all memory and I/O devices. In many computer systems outside the Freescale family this addressing mode is called *direct*, because it can directly access all of memory. In this book, we will adhere to the Freescale terminology (direct means 8-bit addressing and extended means 16-bit addressing). The **>** operator forces extended addressing. In general, when the address happens to fall in the 0 to $00FF range, the assembler will automatically use direct addressing, otherwise it uses extended addressing. Figure 1.32 illustrates the execution of the `ldaa $0801` instruction.

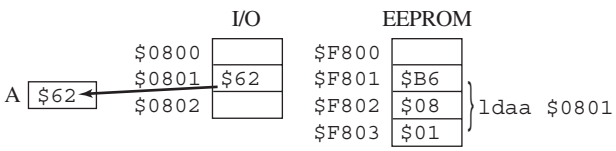


Figure 1.32
Example of the extended addressing mode.

Common error: It is wrong to assume the < and > operators affect the amount of data that is transferred. The < and > operators will affect the addressing mode; that is how the address is represented.

Observation: Accesses to Registers A, B, and CC transfer 8 bits, whereas accesses to Registers D, X, Y, SP, and PC transfer 16 bits regardless of the addressing mode.

Checkpoint 1.35: What is the difference between `ldx #$0801` and `ldx $0801`?

Checkpoint 1.36: What is the difference between the instruction `ldaa $12` and the instruction `ldx $12`?

5. PC-relative addressing mode is used for the branch and branch to subroutine instructions. Stored in the machine code is not the absolute address of where to branch, but the 8-bit signed offset relative distance from the current PC value. When the branch address is being calculated, the PC already points to the next instruction. Calculating relative offsets gives first-time programmers a lot of trouble, but lucky for us the assembler calculates it for us. It is explained here in order to better understand how the computer works, rather than being necessary for us to do while programming. The address of the next instruction is (location of instruction)+(number of bytes in the machine code). In the following example, assume the branch instruction is located at address $F880. The destination address is before the current instruction, which is called a backward jump.

```
bra $F840
```

The operand field for PC relative addressing is an 8-bit value called **rr,** which is calculated using the equation: (destination address)−(location of instruction)−(size of the instruction). Since the bra op code is one byte ($20) and the operand is one byte, this instruction requires two bytes and the **rr** field is

```
$F840 - $F880 - 2 = -$42 = $BE
```

and the object code for this instruction will be $20BE. Again assume the branch instruction is located at address $F880. This time the destination address is after the current instruction, which is called a forward jump.

```
bra $F8C8
```

The **rr** field is

```
$F8C8 - $F880 - 2 = $46
```

and the object code for this instruction will be $2046.

Common error: Since not every instruction supports every addressing mode, it would be a mistake to use an addressing mode not available for that instruction.

Observation: Some of the conditional branch instructions on the 9S12 require a different number of cycles to execute depending on whether or not the branch is taken. The cycle time when accessing external memory on a 9S12 depends on the speed of the external memory. It also depends on whether the address is an even number or an odd number. These facts complicate the task of predetermining how long a 9S12 program will take to execute.

Observation: Relative addressing within a program block is essential for implementing relocatable code.

1.6.4 Numbering Scheme Used by Freescale

The numbering scheme used by Freescale allows you to quickly determine the type of ROM used as the main program memory in the device, as shown in Table 1.11. In most cases, the value at the end of the part number specifies the size of the program memory (e.g., MC68HC711E20 is 20K and MC9S12C32 is 32K), but sometimes for the 6811 it does not (e.g., MC68HC11D3 is 4K and MC68HC711E9 is 12K). The letter (or letters) placed after the 11 or 12 and before the final number specifies the series. For example, the MC68HC711E9 belongs to the "E" series. Please note that most Freescale microcontrollers have two or three memory modules, and the numbering scheme refers only to the main program memory. For example, according to the numbering scheme the MC9S12DP512 has 512K bytes flash EEPROM, but it also has 4K bytes of regular EEPROM and 12K bytes of RAM.

Table 1.11 Freescale uses a numbering scheme to specify the type of main memory.

Number	Main Memory	Examples
None	ROM	MC68HC11E9 MC68HC12D60
7	EPROM	MC68HC711E9 MC68HC711D3
8	EEPROM	MC68HC811E2 MC68HC812A4
9	Flash EEPROM	MC68HC912B32 MC9S12C32

Checkpoint 1.37: The MC9S12A64 has how much and what type of main memory?

1.7 9S12 Architecture Details

The microcontrollers in the 9S12 family differ by the amount of memory and by the types of I/O modules. All 9S12 microcontrollers have a 16-bit central processing unit (HCS12 or HCS12X), SIM (a system integration module), RAM (volatile random access memory), Flash EEPROM (nonvolatile electrically erasable programmable read-only memory), and a PLL (phase-locked loop). The 9S12 microcontrollers are configured with zero, one, or more of the following modules: asynchronous serial communications interface (SCI), serial peripheral interface (SPI), inter-integrated circuit (I^2C), key wakeup, 16-bit timer, a pulse-width modulation (PWM), 10-bit or 12-bit analog-to-digital converter (ADC), 8-bit digital-to-analog converter (DAC), liquid-crystal display driver (LCD), controller area network (CAN 2.0), universal serial bus (USB 2.0) interface, Ethernet (MAC FEC 10/100) interface, memory expansion logic, and XGate. The PLL allows the software to increase or decrease the execution speed. XGate is an I/O coprocessor and is appropriate for systems requiring high-speed I/O. Typically, the SPI and I^2C modules allow the 9S12 to communicate with other dedicated peripheral devices, whereas typically, the SCI, CAN, USB, and Ethernet allow the 9S12 to communicate with other computers. The ADC module (together with sensors and analog amplifiers) can be used to collect information. The PWM module can be used to control delivered

power to motors and lights. The PWM also can be used as an analog output. We can use key wake-up modules to interface digital inputs to the 9S12. Currently, the 9S12 family consists of a large number of devices with a wide range of memory and I/O configurations, only some of which are shown in Table 1.12. This book will focus on the MC9S12C32 and MC9S12DP512, because they are low-cost full-featured devices with minimal memory, making them ideally suitable for educational use. The fundamental concepts learned in this book can be adapted easily to other members of the Freescale microcontroller family.

Series	ROM	RAM	I/O pins	Features
MC9S12A	32 to 512	2 to 12	59 to 91	I^2C, SCI, SPI, ADC
MC9S12B	64 to 128	2 to 4	59 to 91	Automotive/Industrial with CAN
MC9S12C	32 to 128	2 to 4	31 to 60	CAN, SCI, SPI, ADC
MC9S12D	32 to 512	4 to 12	59 to 91	Automotive/Industrial with CAN
MC9S12E	32 to 128	2 to 16	58 to 90	General purpose, 3 V with D/A
MC9S12GC	16 to 128	1 to 4	31 to 60	Low-cost, Low-pin count
MC9S12H	128 to 256	6 to 12	83 to 117	LCD/H-Bridge drivers with CAN
MC9S12HZ	64 to 256	4 to 12	58 to 85	I^2C, SCI, SPI, ADC, LCD
MC9S12NE	64	8	80	Ethernet, I^2C
MC9S12UF	32	3 to 5	75 to 77	30 MHz, USB, SCI
MC9S12XA	128 to 256	11 to 32	59 to 119	XGate, 40 MHz, I^2C, ADC, SCI
MC9S12XD	64 to 512	4 to 32	59 to 119	XGate, 40 MHz, I^2C, CAN, ADC
MC9S12XE	128 to 1000	12 to 64	59 to 152	XGate, 40 MHz, SCI, SPI, I^2C, ADC
MC9S12XS	64 to 256	4 to 12	44 to 91	XGate, 40 MHz, CAN, SCI, SPI, ADC

Table 1.12
The 9S12 family includes many series; memory is defined in kibibytes.

1.7.1 9S12C32 Architecture

The 9S12C32, with its port structure shown in Figure 1.33, is one of the smaller and lower-cost members of the 9S12 family. It has 2 kibibytes of RAM and 32 kibibytes of EEPROM. Modules include an asynchronous serial communications interface (SCI) module; a serial peripheral interface (SPI) module; an 8-channel, 16-bit timer module; a pulse-width modulation (PWM) module; an 8-channel, 10-bit analog-to-digital converter (ATD); a 1 Mbps controller area network (CAN 2.0) module; memory expansion logic; and a phase-locked loop (PLL). The PLL allows the software to increase or decrease the execution speed. The PWM module can utilize either Port P or Port T as selected by the MODRR register. Ports J and P also support keypad wake-up interrupts.

The address space of the input/output devices and the RAM can be mapped on any 2-KiB boundary by software. In this book, we will use the address map shown in Table 1.13 for the single-chip MC9S12C32.

The 9S12C32 is available in three quad flat package (QFP) sizes. The larger chip packages have more pins, as shown in Table 1.14.

One of the smallest systems based on the 9S12 family is the 24-pin Nanocore module from TechArts, as shown in Figure 1.34. This system includes a voltage regulator, a run/load switch, a BDM header, RS232 drivers for the SCI port, the eight pins of Ports T, and the eight pins of Port AD.

Program 1.1 and Table 1.15 define some of the parallel ports for the 9S12C32. A full list of I/O ports can be found in the reference manual at **http://users.ece.utexas.edu/~valvano/Datasheets**. A *pseudo-operation code* is a command to the assembler and usually does not create machine executable code. Other names for pseudo-operation code are *pseudo-op* and *assembly directive*. The `equ` is a pseudo-op that creates a mapping from a

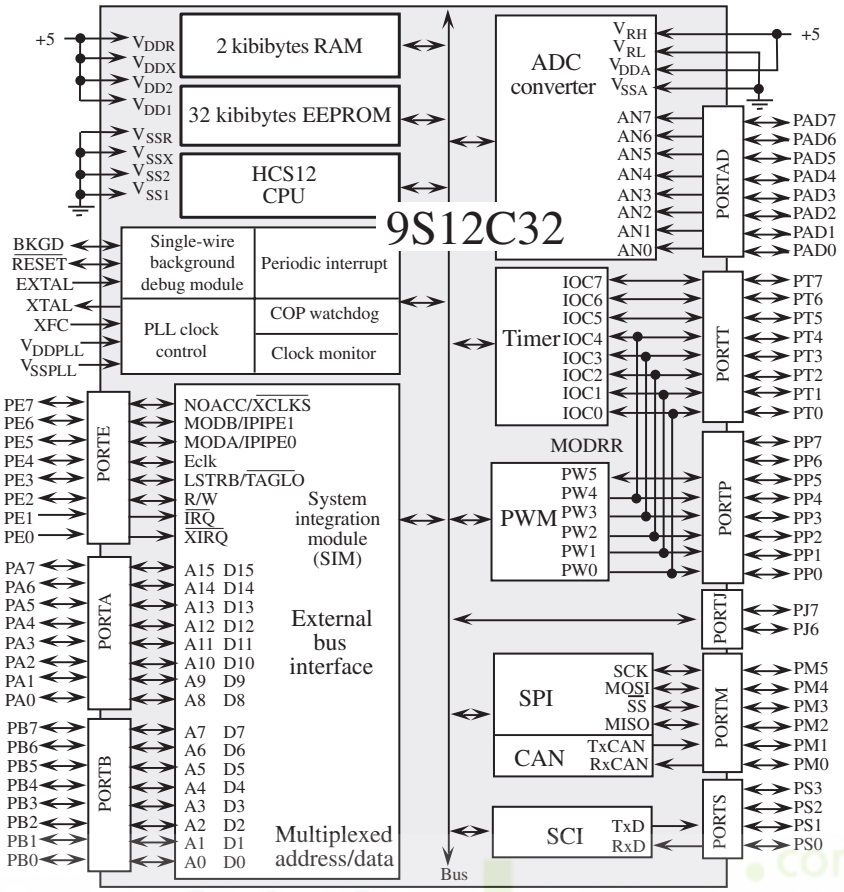


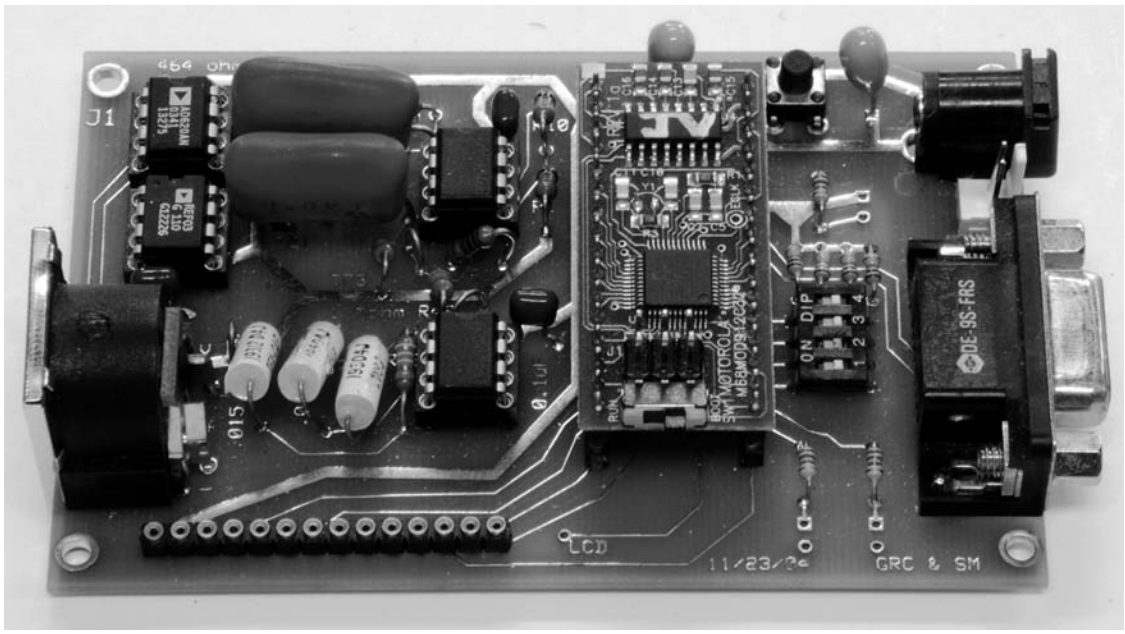
Figure 1.33
Block diagram of a Freescale 9S12C32.

Address	Size	Device	Device	Contents
$0000 to $03FF	1K	I/O	Input/output devices	
$3800 to $3FFF	2K	RAM	Random access memory	Variables and stack
$4000 to $7FFF	16K	EEPROM	Electrically erasable PROM	Programs and fixed constants
$C000 to $FFFF	16K	EEPROM	Electrically erasable PROM	Programs and fixed constants

Table 1.13
The MC9S12C32 has 32K of EEPROM and 2K bytes of RAM.

Port	48-pin	52-pin	80-pin	Shared Functions
Port A	PA0	PA2-PA0	PA7-PA0	Address/data bus
Port B	PB4	PB4	PB7-PB0	Address/data bus
Port E	PE7, PE4, PE1, PE0	PE7, PE4, PE1, PE0	PE7-PE0	System integration module
Port J	—	—	PJ7, PJ6	Key wake-up
Port M	PM5-PM0	PM5-PM0	PM5-PM0	SPI, CAN
Port P	PP5	PP5-PP3	PP7-PP0	Key wake-up, PWM
Port S	PS1-PS0	PS1-PS0	PS3-PS0	SCI
Port T	PT7-PT0	PT7-PT0	PT7-PT0	Timer, PWM
Port AD	PAD7-PAD0	PAD7-PAD0	PAD7-PAD0	Analog-to-digital converter

Table 1.14
The 9S12C32 has nine external I/O ports.



Courtesy of Jonathan Valvano

Figure 1.34
The TechArts NC12C32 Nanocore used to build an EKG monitor (see Figure 12.49).

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
$0250	0	0	PM5	PM4	PM3	PM2	PM1	PM0	PTM
$0251	0	0	PM5	PM4	PM3	PM2	PM1	PM0	PTIM
$0252	0	0	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN
$0270	PAD7	PAD6	PAD5	PAD4	PAD3	PAD2	PAD1	PAD0	PTAD
$0271	PAD7	PAD6	PAD5	PAD4	PAD3	PAD2	PAD1	PAD0	PTIAD
$0272	DDRAD7	DDRAD6	DDRAD5	DDRAD4	DDRAD3	DDRAD2	DDRAD1	DDRAD0	DDRAD

Table 1.15
Some 9S12C32 Parallel ports.

symbol and a value. Given the code in Program 1.1 on page 40, the instructions `staa $0240` and `staa PTT`, produce the exact same machine code; therefore, they perform the exact same function when executed. We prefer `staa PTT`, because it is easier to understand. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output. Notice that Port M is only six bits wide. A pin on Port AD (`PTAD`) can be used as a digital input if the corresponding bit in the `ATDDIEN` is set to 1 and the bit in the `DDRAD` is cleared to 0. A pin on Port AD (`PTAD`) can be used as a digital output if the corresponding bit in the `DDRAD` is set to 1. The pins on Port AD can be used as analog input if the ADC is enabled and the corresponding bit in the `ATDDIEN` is cleared to 0.

1.7.2 9S12DP512 Architecture

Figure 1.35 shows the port structure of the 9S12DP512. Although the 9S12DP512 has 512 KiB EEPROM, only 48 kibibytes of it is directly addressable using standard 16-bit addressing modes. The remaining 464 KiB must be accessed using the paged memory process.

```
; port name definitions
ATDDIEN  equ $008D  ; Input Enable
DDRAD    equ $0272  ; Direction
DDRM     equ $0252  ; Direction
DDRT     equ $0242  ; Direction
PTAD     equ $0270  ; I/O
PTIAD    equ $0271  ; Input
PTM      equ $0250  ; I/O
PTIM     equ $0251  ; Input
PTT      equ $0240  ; I/O
PTIT     equ $0241  ; Input
```

```
#define _P(n) *(unsigned char volatile *)(n)
#define ATDDIEN _P(0x008D)
#define DDRAD   _P(0x0272)
#define DDRM    _P(0x0252)
#define DDRT    _P(0x0242)
#define PTAD    _P(0x0270)
#define PTIAD   _P(0x0271)
#define PTM     _P(0x0250)
#define PTIM    _P(0x0251)
#define PTT     _P(0x0240)
#define PTIT    _P(0x0241)
```

Program 1.1
Definitions of some of the 9S12C32 I/O ports.

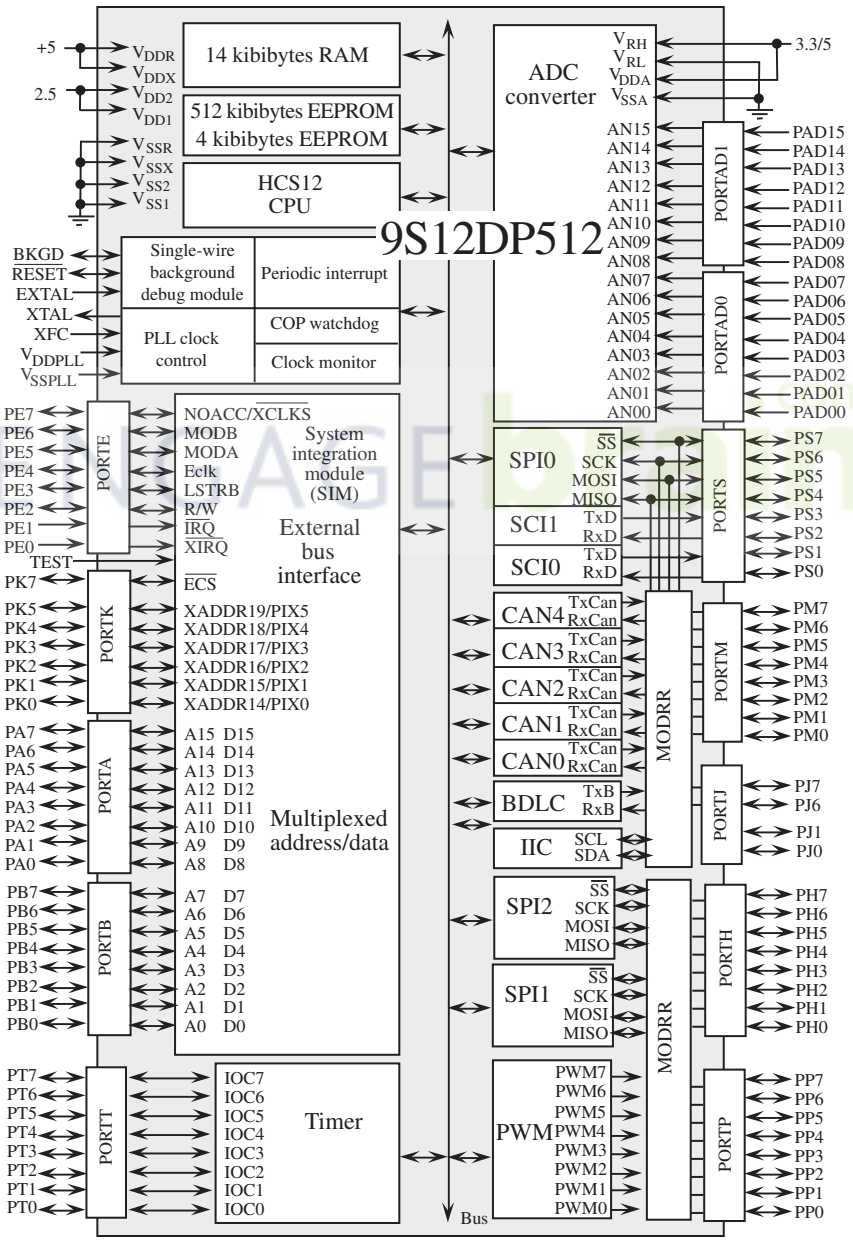


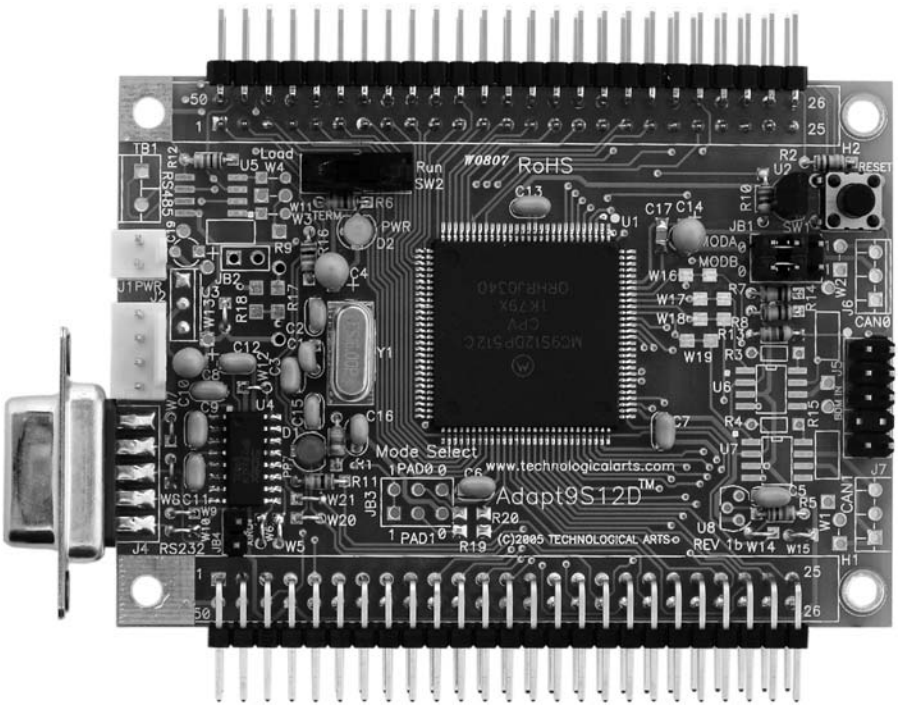
Figure 1.35
Block diagram of a Freescale 9S12DP512.

Table 1.16 shows the I/O pins that exist on the 9S12DP512 chip. Other versions of the D-family, such as DG DJ DT, are at a lower cost than the DP512, because they have less memory, fewer CAN, fewer SPI, and no J1850 ports. All the DP512 examples in this book will apply to these other members of the D-family. All pins except PE5 and PE6 are available on the TechArts Adapt module shown in Figure 1.36.

Table 1.16
The 9S12DP512 is a 112 pin module with 91 I/O pins

Chip	Special I/O (Priority Order)	Standard I/O
PAD15-PAD8	ADC	Digital input
PAD7-PAD0	ADC	Digital input
PA7-PA0		Digital I/O
PB7-PB0		Digital I/O
PE7-PE0	IRQ and XIRQ	Digital I/O
PH7-PH4	SPI2	Digital I/O and key wake-up
PH3-PH0	SPI1	Digital I/O and key wake-up
PJ7-PJ6	CAN4 I^2C or CAN0	Digital I/O and key wake-up
PJ1-PJ0		Digital I/O and key wake-up
PK7, PK5-PK0		Digital I/O
PM7-PM6	CAN3 or CAN4	Digital I/O
PM5-PM4	CAN2 CAN0 CAN4 or SPI0	Digital I/O
PM3-PM2	CAN1 CAN0 or SPI0	Digital I/O
PM1-PM0	CAN0 or BDLC	Digital I/O
PP7-PP4	PWM or SPI2	Digital I/O and key wake-up
PP3-PP0	PWM or SPI1	Digital I/O and key wake-up
PS7-PS4	SPI0	Digital I/O
PS3-PS2	SCI1	Digital I/O
PS1-PS0	SCI0	Digital I/O
PT7-PT0	Input capture or output compare	Digital I/O

Figure 1.36
A 9S12DP512 system from Technological Arts (#AD9S12DP512M0).



A 9S12DP512 system from Technological Arts (#AD9S12DP512MO).

The 9S12DP512 has 91 I/O pins, some of which are listed in Program 1.2. Many of the pins can be configured to implement complex I/O functions, but in this section, the pins are used as simple digital inputs or outputs. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output. Program 1.2 defines four of these parallel ports. A full list of I/O ports can be found in the reference manual at **http://users.ece.utexas.edu/~valvano/Datasheets/**. Table 1.17 shows all 91 digital I/O pins that can be used on the 9S12DP512. Pins PE1, PE0, and Ports `PORTAD1`, `PORTAD0` are input only. The module routine register (MODRR) will be explained later, when it is needed. In C, we add the `volatile`, so the compiler will not optimize the code involving I/O ports. More precisely, it tells the compiler the value may change beyond the control of the software itself. In particular, each time the value is needed, it will be reread from the port.

```
; port name definitions
DDRH     equ $026A  ; Direction
DDRJ     equ $0262  ; Direction
DDRM     equ $0252  ; Direction
DDRP     equ $025A  ; Direction
DDRT     equ $0242  ; Direction
PTH      equ $0260  ; I/O
PTIH     equ $0261  ; Input
PTJ      equ $0268  ; I/O
PTIJ     equ $0269  ; Input
PTM      equ $0250  ; I/O
PTIM     equ $0251  ; Input
PTP      equ $0258  ; I/O
PTIP     equ $0259  ; Input
PTT      equ $0240  ; I/O
PTIT     equ $0241  ; Input
```

```
#define _P(n) *(unsigned char volatile *)(n)
#define DDRH _P(0x0262)
#define DDRJ _P(0x026A)
#define DDRM _P(0x0252)
#define DDRP _P(0x025A)
#define DDRT _P(0x0242)
#define PTH  _P(0x0260)
#define PTIH _P(0x0261)
#define PTJ  _P(0x0268)
#define PTIJ _P(0x0269)
#define PTM  _P(0x0250)
#define PTIM _P(0x0251)
#define PTP  _P(0x0258)
#define PTIP _P(0x0259)
#define PTT  _P(0x0240)
#define PTIT _P(0x0241)
```

Program 1.2
Definitions of some of the 9S12DP512 I/O ports.

Figure 1.37 shows the Dragon12 board from Wytec (**http://www.evbplus.com/index.html**). This board provides an integrated approach to teaching embedded systems where most of the I/O devices are included on the board itself. This board includes the 9S12DG512 and the following additional components: 16 by 2 LCD display module with LED backlight module, 4-digit, multiplexed 7-segment display module, 4 by 4 matrix keypad, eight LEDs connected, an 8-position DIP switch, four pushbutton switches, IR transceiver with built-in 38 kHz oscillator, RS485 communication port with terminal block, a speaker driven by timer (or PWM or DAC for alarm), voice and music applications, a potentiometer trimmer pot for analog input, dual SCIs with DB9 connectors, dual 10-bit DAC for testing SPI interface and generating analog waveforms, I^2C-based Real Time Clock DS1307 with backup battery, dual H-Bridge with motor feedback or incremental encoder interface, four robot servo outputs, an opto-coupler output, a DPDT form C relay, a temperature sensor, a light sensor, a logic probe with LED indicator, a solderless breadboard, and a DB9 RS232 cable for connecting to a PC serial port. The Dragon 12 is an excellent platform to teach embedded systems. This board is very cost effective, much cheaper than purchasing the components individually. If embedded systems is taught in more than one class, this board supports both a simple introduction to embedded systems as well as a more sophisticated embedded systems lab including CAN, I^2C, motors, and servos.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
$0001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
$0002	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
$0003	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	DDRB
$0008	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	PORTE
$0009	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0	DDRE
$0032	PK7	0	PK5	PK4	PK3	PK2	PK1	PK0	PORTK
$0033	DDRK7	0	DDRK5	DDRK4	DDRK3	DDRK2	DDRK1	DDRK0	DDRK
$008F	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PORTAD0
$012F	PAD15	PAD14	PAD13	PAD12	PAD11	PAD10	PAD09	PAD08	PORTAD1
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
$0247	0	MODRR6	MODRR5	MODRR4	MODRR3	MODRR2	MODRR1	MODRR0	MODRR
$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
$0249	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTIS
$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
$0250	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTM
$0251	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTIM
$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
$0258	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTP
$0259	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTIP
$025A	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
$0260	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTH
$0261	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTIH
$0262	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
$0268	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTJ
$0269	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTIJ
$026A	DDRJ7	DDRJ6	0	0	0	0	DDRJ1	DDRJ0	DDRJ

Table 1.17
9S12DP512 parallel ports.

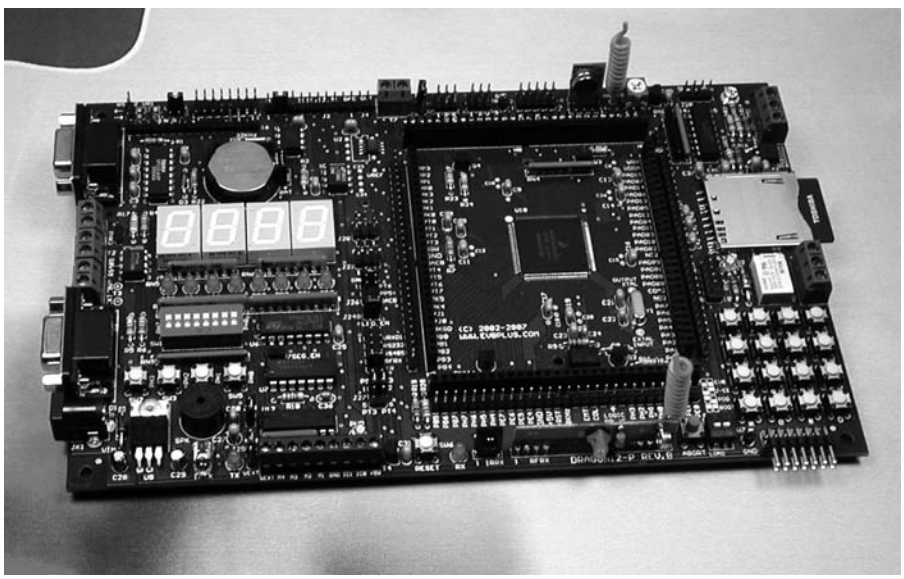


Figure 1.37
Dragon12-plus board from Wytec.

Dragon12-plus board from Wytec

1.7.3 9S12E128 Architecture

Figure 1.38 shows the port structure of the 9S12E128, which has 8 KiB of RAM and 128 KiB of flash EEPROM. This member of the 9S12 family has 12 input capture/output compare timer pins, 12 pulse-width modulated output pins, 16 ADC inputs, two DAC outputs, one SPI modules three SCI modules, and one I^2C interface. There are two sizes of the 9S12E128 chip: one with 80 pins and the other with 112 pins. The 112-pin chip has 92 I/O pins, some of which are listed in Table 1.18. We clear (0) a bit in the direction register to make that pin an input, and set it (1) to make it an output.

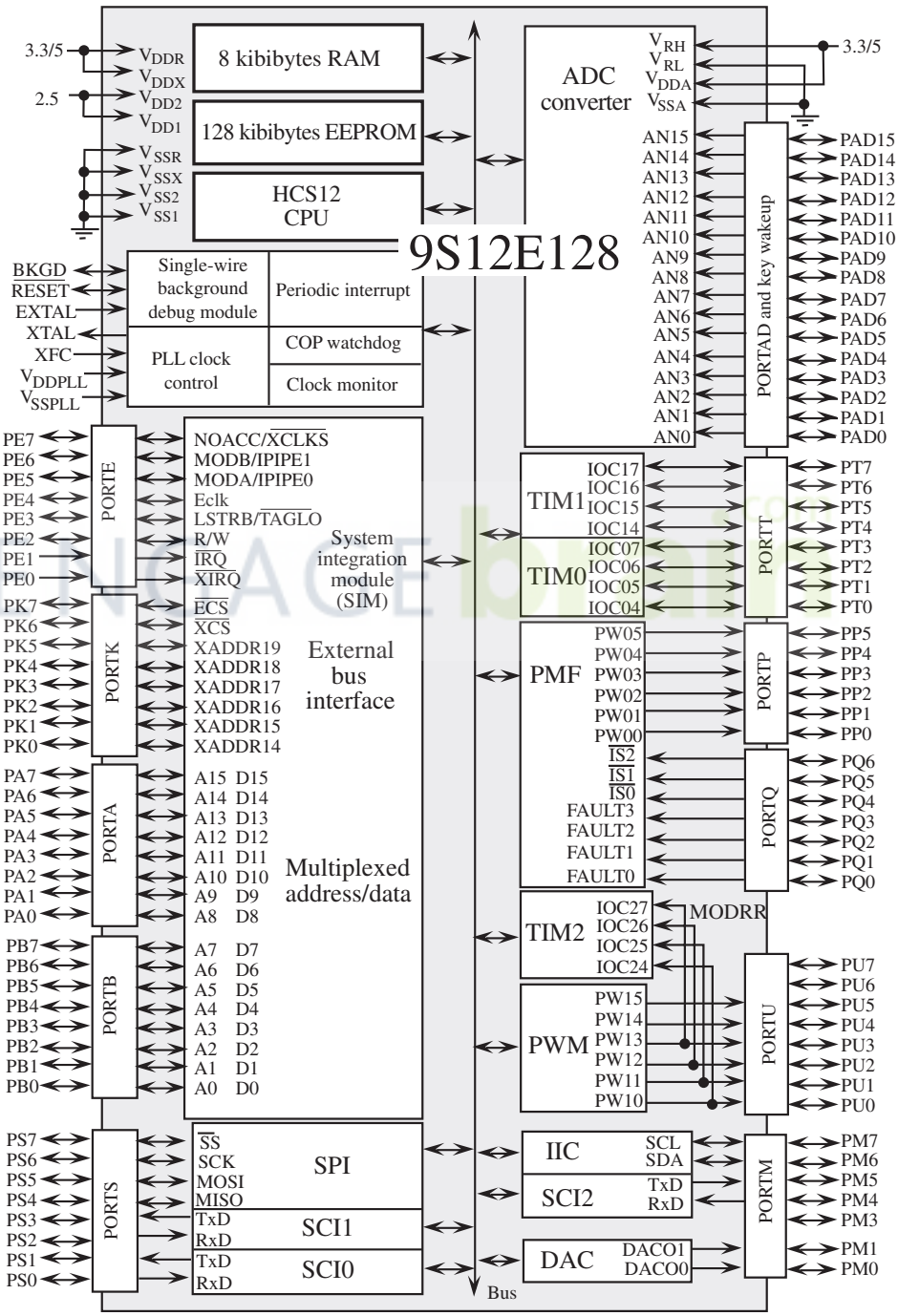


Figure 1.38
Block diagram of a Freescale 9S12E128.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
$0001	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
$0002	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
$0003	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	DDRB
$0008	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0	PORTE
$0009	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0	DDRE
$0032	PK7	PK6	PK5	PK4	PK3	PK2	PK1	PK0	PORTK
$0033	DDRK7	DDRK6	DDRK5	DDRK4	DDRK3	DDRK2	DDRK1	DDRK0	DDRK
$008C	Bit 15	14	13	12	11	10	9	Bit 8	ATDDIEN0
$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATDDIEN1
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0241	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTIT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
$0249	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTIS
$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
$0250	PM7	PM6	PM5	PM4	PM3	0	PM1	PM0	PTM
$0251	PM7	PM6	PM5	PM4	PM3	0	PM1	PM0	PTIM
$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	0	DDRM1	DDRM0	DDRM
$0258	0	0	PP5	PP4	PP3	PP2	PP1	PP0	PTP
$0259	0	0	PP5	PP4	PP3	PP2	PP1	PP0	PTIP
$025A	0	0	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
$0260	0	PQ6	PQ5	PQ4	PQ3	PQ2	PQ1	PQ0	PTQ
$0261	0	PQ6	PQ5	PQ4	PQ3	PQ2	PQ1	PQ0	PTIQ
$0262	0	DDRQ6	DDRQ5	DDRQ4	DDRQ3	DDRQ2	DDRQ1	DDRQ0	DDRQ
$0268	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	PTU
$0269	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	PTIU
$026A	DDRU7	DDRU6	DDRU5	DDRU4	DDRU3	DDRU2	DDRU1	DDRU0	DDRU
$0270	PAD15	PAD14	PAD13	PAD12	PAD11	PAD10	PAD09	PAD08	PTADH
$0271	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PTADL
$0274	DDRAD15	DDRAD14	DDRAD13	DDRAD12	DDRAD11	DDRAD10	DDRAD09	DDRAD08	DDRADH
$0275	DDRAD07	DDRAD06	DDRAD05	DDRAD04	DDRAD03	DDRAD02	DDRAD01	DDRAD00	DDRADL

Table 1.18
9S12E128 Parallel ports.

A pin on Port AD (PTAD) can be used as a digital input if the corresponding bit in the ATDDIEN is set to 1 and the bit in the DDRAD is cleared to 0. A pin on Port AD (PTAD) can be used as a digital output if the corresponding bit in the DDRAD is set to 1. The pins on Port AD can be used as analog input if the ADC is enabled and the corresponding bit in the ATDDIEN is cleared to 0.

1.7.4 Operating Modes

The 9S12 can operate in one of eight modes, where the mode is selected by the values of the three signals BKGD, MODA, and MODB that exist when the device starts up after a reset. In this book, we will only study the three modes shown in Table 1.19. In *single chip mode*, the 9S12 contains the four major building blocks required to make a complete computer system: processor, I/O, RAM, and EEPROM. For most of the book, we will be using single chip mode, where all ports are available for input/output. Because the 9S12 family is available with flash EEPROM ranging from 32 to 512 KiB, most embedded projects can be developed directly in single chip mode. On the other hand, the 9S12 family RAM sizes only range from 2 to 32 KiB. In Chapter 9, we will use the two expanded modes to interface external devices to Ports A, B, and E. For embedded

BKGD	MODB	MODA	Mode	Port A	Port B
1	0	0	Normal Single Chip	In/Out	In/Out
1	0	1	Normal Expanded Narrow	A15-8/D15-8/D7-0	A7-A0
1	1	1	Normal Expanded Wide	A15-8/D15-8	A7-0/D7-0

Table 1.19
The 9S12 has eight operating modes, but we will use normal single chip mode.

systems that require a large amount of read/write storage, we will use expanded modes to interface external RAM to the system. *Expanded narrow mode* creates a 16-bit address bus and 8-bit data bus, while *expanded wide mode* implements both the address bus and data bus as 16 bits.

We use flash EEPROM during development because it takes only minutes to perform an edit/assemble/download cycle. For delivered projects, we simply program our code into the flash EEPROM and embed the device into our final product. In single chip mode, the 9S12 implements a complete microcomputer, where all of its I/O ports are available. This mode is used for the final product with the application software programmed into the EEPROM.

1.8 Phase-Lock-Loop (PLL)

Normally, the execution speed of a microcontroller is determined by an external crystal. Some MC9S12C32 boards have an 8 MHz crystal creating a 4 MHz E clock. The MC9S12DP512 shown in Figure 1.36 has a 16 MHz crystal creating a 8 MHz E clock. The 9S12 has a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Program 1.3 will increase the E clock from 8 MHz to 24 MHz. The `OSCCLK` is the frequency of the crystal. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the E clock will require less power to operate and generate less heat. Speeding up the E clock obviously allows for more calculations per second.

```
; 9S12DP512 with a 16 MHz crystal
PLL_Init
  movb #$02,SYNR
  movb #$01[1],REFDV
; PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
  movb #$00,CLKSEL
  movb #$D1,PLLCTL     ; turn on PLL
  brclr CRGFLG,#$08,*  ; stabilized?
  bset CLKSEL,#$80     ; Switch to PLL
  rts
```

[1] Use `00` for the 9S12C32 or 9S12E128 with an 8 MHz crystal.
[2] Use `00` for the 9S12C32 or 9S12E128 with an 8 MHz crystal.

```
// 9S12DP512 with a 16 MHz crystal
void PLL_Init(void){
  SYNR = 0x02;
  REFDV = 0x01[2];
// PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
  CLKSEL = 0x00;
  PLLCTL = 0xD1;  // turn on PLL
  while((CRGFLG&0x08) == 0){}
  CLKSEL |= 0x80;  // Switch to PLL clock
}
```

Program 1.3
These programs active the phase lock loop, setting the E clock to 24 MHz.

1.9 Parallel I/O Ports

1.9.1 Basic Concepts of Input and Output Ports

A *parallel I/O port* is a simple mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once. An input port, which allows the software to read external digital signals, is usually read only. That means a read cycle access from the port address returns the values existing on the inputs at that time. In particular, the tristate driver (triangle-shaped circuit in Figure 1.39) will drive the input signals onto the data bus during a read cycle from the port address. A write cycle access to an input port usually produces no effect. A simple fixed input port, such as pin PAD0 on the 9S12DP512 or pin PE0 on the MC9S12C32, behaves similarly to the circuit shown in Figure 1.39. The digital values existing on the input pins are copied into the microcomputer when the software executes a read from the port address.

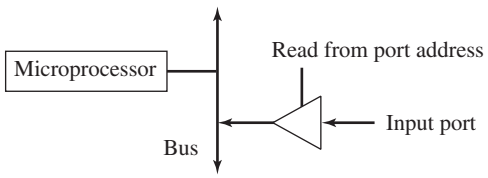


Figure 1.39
A read-only input port allows the software to read external digital signals.

A latched input port behaves similarly to the circuit shown in Figure 1.40. The digital values existing on the input pins are copied into an internal latch on an appropriate edge of the external control signal. At a later time, the data is transferred to the microcomputer when the software executes a read from the latch address. Notice that this latched input port also supports the regular input function. In other words, the software has the option of reading the port address to get information directly from the input port pins or from the latch address to get information that existed at the time of the previous active edge of the external control signal. None of the 9S12 pins can be latched.

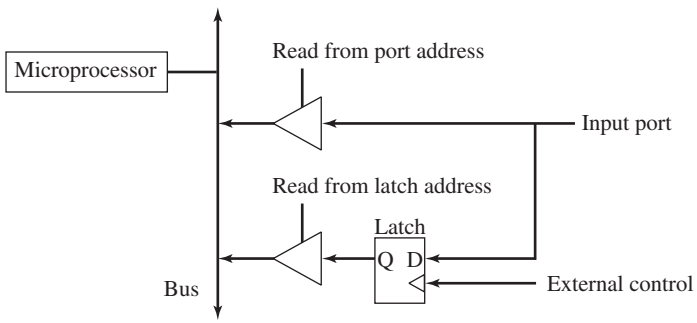


Figure 1.40
A latched input port allows the software to read external digital signals that are captured via the external control signal.

Checkpoint 1.38: What happens if the software writes to an input port?

Often the latched input port allows the software to select the strategic edge to affect the latch function. In other words, the software specifies whether the rise or fall of an external control signal latches the input data. It is important to remember that when the software reads from the latch address, it obtains the values of the input signals occurring at the time of the active edge of the external control signal. In this way, the external device can provide the data at the input and issue an edge on the control signal latching the data into the computer;

the software can process the data at a later time without requiring the external device to maintain the valid data at the input. One of the limitations of this particular configuration is that it does not include a way for the software to signal back (acknowledge) to the external hardware that the previous data has been read by the software. In Chapter 3, we will develop more sophisticated handshaking protocols that provide for two-way synchronization.

Although an input device usually involves just the software reading the port, an output port can participate in both the read and write cycles very much like a regular memory. Figure 1.41 describes a "readable output port." For an 8-bit output port, there will be 8 D flip flops to hold the values on the output pins. A write cycle to the port address will affect the values on the output pins. In particular, the microcomputer places information on the data bus and that information is clocked into the D flip flops. Since it is a readable output, a read cycle access from the port address returns the current values existing on the port pins. A "write-only output port" does not allow software to read the current values. There are no output-only ports on the 9S12.

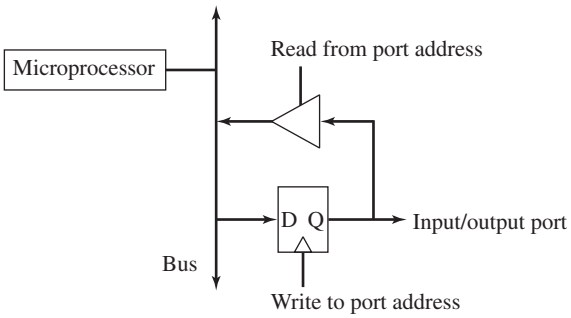


Figure 1.41
A readable output port allows the software to generate external digital signals.

Checkpoint 1.39: What happens if the software reads from an output port as shown in Figure 1.41?

To make the microcontroller more marketable, most ports can be software-specified to be either inputs or outputs. Freescale uses the concept of a direction register to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1), as shown in Figure 1.42. We define a *ritual* as a program executed during startup that initializes hardware and software. If the ritual software makes direction bit zero, the port pin behaves like a simple input, and if it makes the direction bit one, the pin becomes a readable output. Most of the 9S12 ports operate as shown in to Figure 1.42.

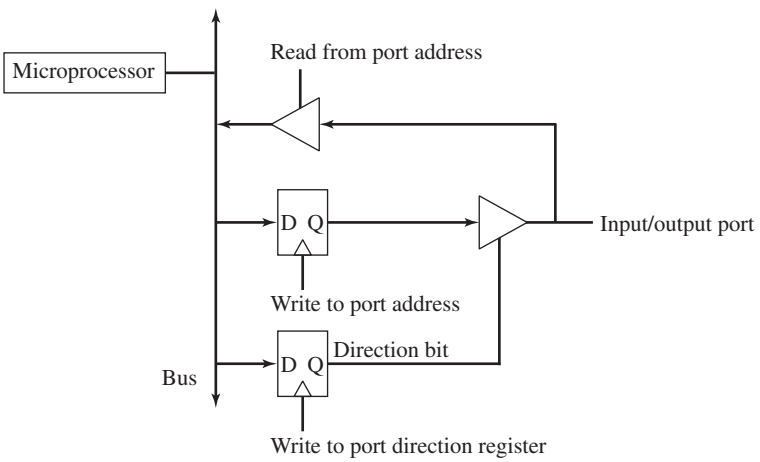
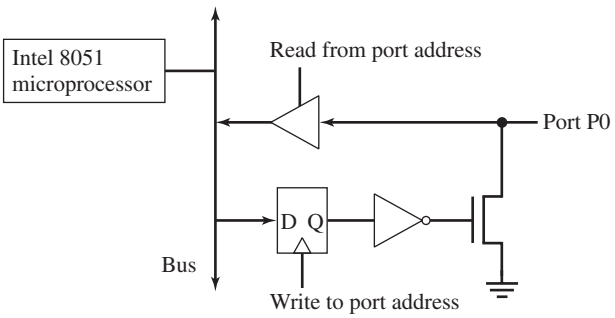


Figure 1.42
A bidirectional port can be configured as a read-only input port or a readable output port.

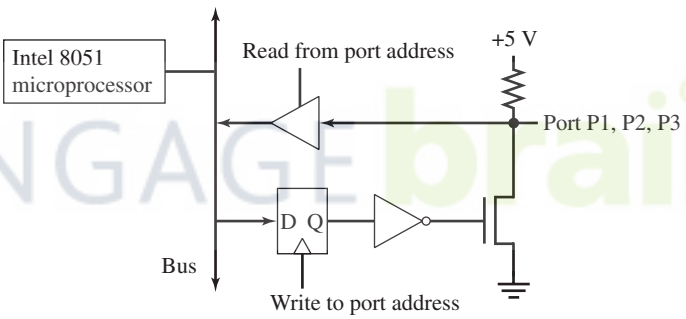
Intel uses open collector outputs to determine whether a pin is an input or an output, as shown in Figure 1.43. Port P0 on the Intel 8051 is open collector. If the software writes a "1" to the port, it behaves like a simple input port. If Port P0 is to be used as an output, you will need to add external pull-up resistors.

Figure 1.43
A bidirectional port is sometimes implemented with an open collector readable output port.



On the Intel 8051, Ports P1, P2, and P3 are open collector with internal pull-up resistors, as shown in Figure 1.44. If the software writes a "1" to the port, it behaves like a simple input port with the pullup to +5 V. If the software writes a "0" to the port, the output will be pulled low, and if it writes a "1", the output will be pulled up by the internal resistor.

Figure 1.44
Some of the bidirectional ports on the Intel 8051 have internal pull-up resistors.



Common error: An output port that is created with open collector outputs with pull-up may not have enough I_{OH} to drive its external circuits.

Common error: Many program errors can be traced to confusion between I/O ports and regular memory. For example, you should not write to an input port, and sometimes you cannot read from an output port.

Observation: If a port pin is configured as a readable output but external loading causes the pin voltage to be different from the value written by the software, then a read from the port will return the voltage level at the pin and not the value last written by the software. This fact can be used by the software to detect excess loading by the external circuits.

1.9.2 Introduction to I/O Programming and the Direction Register

On most embedded microcomputers, the I/O ports are memory mapped. This means the software accesses an input/output port simply by reading from or writing to the appropriate address. To make our software more readable, we include symbolic definitions for the I/O ports, as previously shown in Programs 1.1 and 1.2. We use the direction register (e.g., `DDRT`) to specify which pins are input and which are output. Typically, we write to the data

register (e.g., PTT) *once* during the initialization phase. We use the data register to perform input/output on the port. Conversely, we read and write the data register *multiple times* to perform input and output, respectively, during the running phase. To initialize an I/O port, we set its direction register. The direction register specifies bit for bit whether the corresponding pins are input (0) or output (1). To make pins 7–4 input and pins 3–0 output, you should set the direction register to $0F, as shown in Program 1.4.

Program 1.4
Software that initializes pins 7–4 to input and pins 3–0 to output on the 9S12.

```
ldaa #$0F  ;PTT 7-4 input
staa DDRT  ;PTT 3-0 output
```

```
DDRT = 0x0F;
```

If a pin is programmed as an input, then a write to that port has no effect on that pin. If a pin is programmed as an output, a write to that port will set or clear that pin. Notice that most 9S12 I/O ports have two I/O addresses, e.g., PTT and PTIT. On the 9S12, we have two options when reading pins that are configured as outputs. If a pin on the 9S12 is configured as an output, a read from the regular port address (e.g., PTT) will return the value that was previously written. In addition to the regular port addresses, the 9S12 has separate input addresses (e.g., PTIT), which are read-only. If a pin on the 9S12 is configured as an output, a read from the input address returns the value that exists on the pin at that time. We could compare PTT to PTIT to determine if any output pins are damaged or to detect excess loading. When a pin is an input, reading PTT or PTIT returns the same value, which is of course the input on that pin.

Checkpoint 1.40: Does the entire port need to be defined as input or output, or can some pins be input while others are output?

Observation: We can create open collector outputs (zero, hiZ) by setting the data port to zero at the beginning of our software, then setting the direction register to the complement of the desired output whenever we want to change the output.

Example 1.1: Design an embedded system that flashes LEDs in a 0101, 0110, 1010, 1001 binary repeating pattern.

Solution: Some problems are so unusual that they require the engineer to invent completely original solutions. Most of the time, however, the engineer can solve even complex problems by building the system from components that already exist. Creativity will still be required in selecting the proper components, making small changes in their behavior (tweaking), arranging them in an effective and efficient manner, then verifying the system satisfies both the requirements and constraints. When young engineers begin their first job, they are sometimes surprised to see that education does not stop with college graduation, but rather is a life-long activity. In fact, it is the educational goal of all engineers to continue to learn both processes (rules about how to solve problems) and products (hardware software components). As the engineer becomes more experienced, he or she has a larger toolbox from which processes and components can be selected.

The hardest step for most new engineers is the first one, where to begin? We begin by analyzing the problem to create a set of specifications and constraints. This system will need four LEDs, and the computer must be able to activate and deactivate them. Since the problem didn't specify power source, speed, color, or brightness, we could either put off these decisions until the engineering design stage in order to simplify the design or minimize cost, or we could go back to the customer and ask for additional requirements. In this case, the customer didn't care about power, speed, color or brightness, but did think minimizing cost was a good idea. Due to the nature of this book, we will constrain all our designs

to include the 9S12. Because we have +5 V microcomputer systems, we will specify the system to run on +5 V power. We have in our stockroom lost-cost red LEDs with 2.2 V, 10 mA specification, so we will use them. Table 1.20 summarizes the specifications and constraints. We will use standard 5% resistors to minimize cost.

Table 1.20
Specifications and constraints of the LED output system.

Specifications	Constraints
Repeating pattern of 5, 6, 10, 9	9S12 based
Four 2.2 V, 10 mA red LEDs	Minimize cost
+5 V power supply	Standard 5% resistors

It is often difficult to distinguish whether a parameter is a specification or a constraint. In actuality, when designing a system it often doesn't matter into which category a parameter falls, because the system must satisfy all specifications and constraints. Nevertheless, when documenting the device it is better to categorize parameters properly. Specifications generally define in a quantitative manner the overall system objectives as given to us by our customers. Constraints, on the other hand, generally define the boundary space within which we must search for a solution to the problem. If we must use a particular component, it is often considered a constraint. In this case we will be using the 9S12. Constraints also are often defined as an inequality, such as the cost must be less than $50, or the battery must last for at least one week. Specifications on the other hand are often defined as a quantitative number, and the system satisfies the requirement if the system operates within a specified *tolerance* of that parameter. Tolerance can be defined as a percentage error or as a range with minimum and maximum values. For example, our LED output system is acceptable if it has four LEDs, but unacceptable if it has three or five of them. Similarly, it will be OK as long as the LED current is between 8 and 12 mA. If the current drops below 8 mA, we won't be able to see the LED, and if it goes above 12 mA, it might damage the LED.

Observation: Defining realistic tolerances on our specifications will have a profound effect on system cost.

Checkpoint 1.41: What are the effects of specifying a tighter tolerance (e.g., 1% when the problem asked for 5%)?

Checkpoint 1.42: What are the effects of specifying a looser tolerance (e.g., 10% when the problem asked for 5%)?

The next step is the high-level design. The data flow graph in Figure 1.45 shows information as it flows from the `controller` software to the four LEDs. The data flow graph will be important during the subsequent design phases because the hardware blocks can be considered as a preliminary hardware block diagram of the system. The call graph, also shown in Figure 1.45, illustrates this is master/slave configuration in which the `controller` software will manipulate the four LEDs.

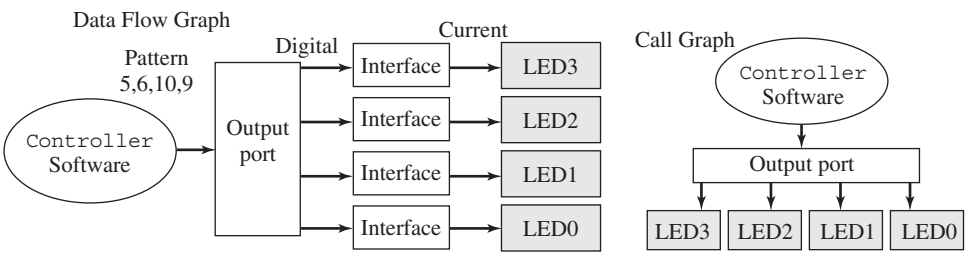


Figure 1.45
Data flow graph and call graph of the LED output system.

The hardware design of this system simply involves using four copies of the LED interface presented earlier in Figure 1.20. We can use the 7405 because its I_{OL} (16mA) is enough to drive the 10 mA LED. Notice the similarity of the data flow graph in Figure 1.45 with the hardware circuit in Figure 1.46. The data flow graph and call graph in this example look similar because this system just performs output. We will need to tweak the circuit, adjusting the value of the resistor to produce the desired 2.2 V, 10 mA specification. If the V_{OL} of the 7405 is 0.4 V, and the voltage across the LED is 2.2 V, then the voltage across the resistor should be 5-2.2-0.4 V or 2.4 V. We calculate the resistor value using Ohm's Law—R is 2.4 V/10 mA or 240 Ω. Using standard resistor values with a 5% tolerance will make the product both cheaper and easier to build. One good way to tell whether a resistor value is standard is to go online and see which resistor values are in stock (e.g., **www.digikey.com, www.jameco.com, www.mouser.com**). In particular, 220 Ω and 270 Ω are two standard resistor values near 240 Ω. If we were to use 220 Ω, then the LED current would be (5-2.2-0.4 V)/220 Ω or 10.9 mA. Similarly, if we were to use 270 Ω, then the LED current would be (5-2.2-0.4 V)/270 Ω or 8.9 mA. Both would have been acceptable, but we will use the 220 Ω resistor because it is closer. It would have been more expensive to design this system with 240 Ω resistors.

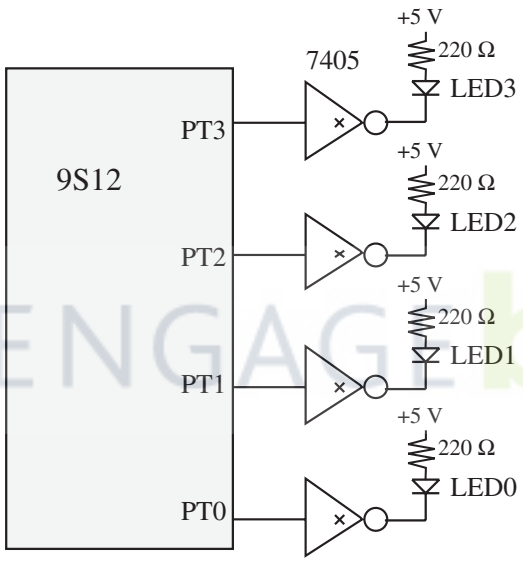


Figure 1.46
Hardware circuit for the LED output system.

The software design of this system also involves using examples presented earlier with some minor tweaking. The only data required in this problem is the 5,6,10,9 sequence. In Chapter 3 we will consider solutions to this type of problem using data structures, but in this first example, we will take a simple approach, not using a data structure. Figure 1.47 illustrates a software design process using *flowcharts*. We start with a general approach on the left. Flowchart 1 shows that the software will initialize the output port and perform the output sequence. As we design the software system, we fill in the details. This design process is called *successive refinement*. It is also classified as top-down, because we begin with high-level issues and end with the low-level. In Flowchart 2, we set the direction register, then output the 5,6,10,9. It is at this stage that we figured out how to create the repeating sequence. Flowchart 3 fills in the remaining details. To output the pattern 0101 to the LEDs, we will output a 5 to PTT on the 9S12. *Pseudo-code* is similar to high-level languages but lacks a rigid syntax. This means we can utilize whatever syntax we like. Flowcharts are good when the software involves complex algorithms with many decisions points causing control paths. On the other hand, pseudo-code may be better when the software is more sequential and involves complex mathematical calculations.

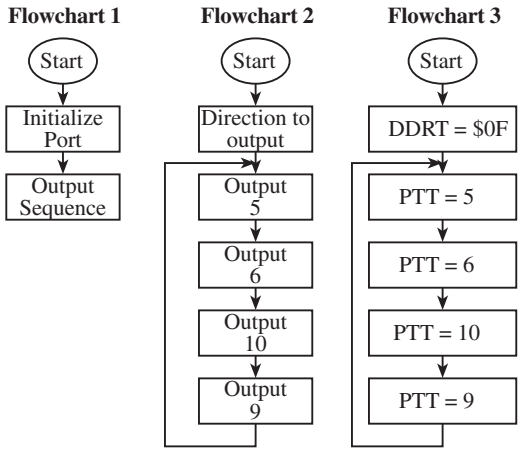


Figure 1.47
Software design for the LED output system using flowcharts.

Many software developers use pseudo-code rather than flowcharts, because the pseudo-code itself can be embedded into the software as comments. Program 1.5 shows the assembly code and C implementations. Notice the similarity in structure between Flowchart 3 and this code. Information following the semicolon is a comment, which allows programmers to document their software, but is ignored when converting to machine code. `org` is a pseudo-op instructing the assembler to place the subsequent software at the specified address. The first `org` specifies that the machine code for this system will be loaded into flash EEPROM. Making the bottom four bits of the port outputs was previously presented as Program 1.4. In order to set the LEDs to the pattern 0101, we first bring the value of 5 into Register A, then we output the 5 by storing register A to the port. We create the repeating pattern by using an unconditional branch at the end, so the 5,6,10,9 output pattern occurs over and over. The *reset vector* on the 9S12 is a 16-bit value stored at $FFFE and $FFFF. This 16-bit value is loaded into the program counter when the system starts up after a reset. In particular, this value specifies where our software will start execution. The last two lines of this program define the reset vector so our program will start at `Main`.

Program 1.5
Assembly and C software for the LED output system.

```
     org   $4000 ;ROM
Main ldaa  #$0F  ;make PT3-0
     staa  DDRT  ;outputs
Controller
     ldaa  #5
     staa  PTT   ;set 0101
     ldaa  #6
     staa  PTT   ;set 0110
     ldaa  #10
     staa  PTT   ;set 1010
     ldaa  #9
     staa  PTT   ;set 1001
     bra   Controller
     org   $FFFE
     fdb   Main  ;Reset vector
```

```
void main(void){// make PT3-0
  DDRT = 0x0F;  // outputs
  while(1){
    PTT = 5;    // 0101
    PTT = 6;    // 0110
    PTT = 10;   // 1010
    PTT = 9;    // 1001
  }
}
```

In order to test the system, we need to build a prototype. One option is simulation. A screen shot of this implementation can be seen as Figure 1.48. A second option is to use a development system like the one shown in Figure 1.49. In this approach, you build the external circuits on a protoboard and use the debugger to download and test the software.

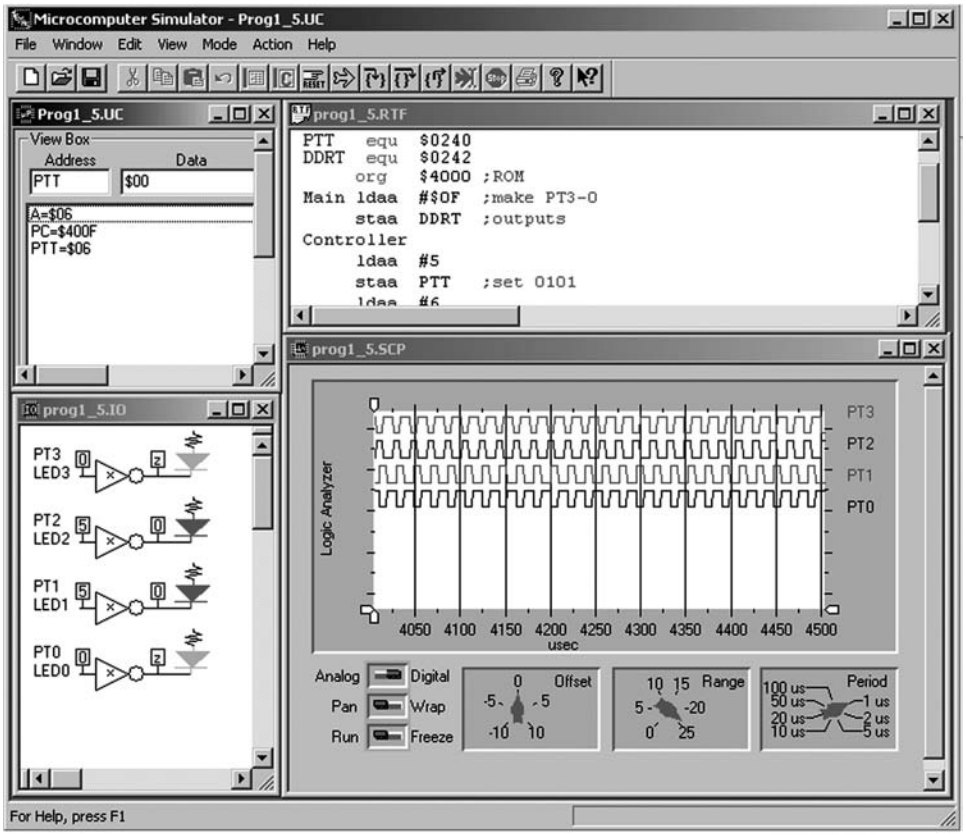


Figure 1.48
TExaS simulation of the LED output system.

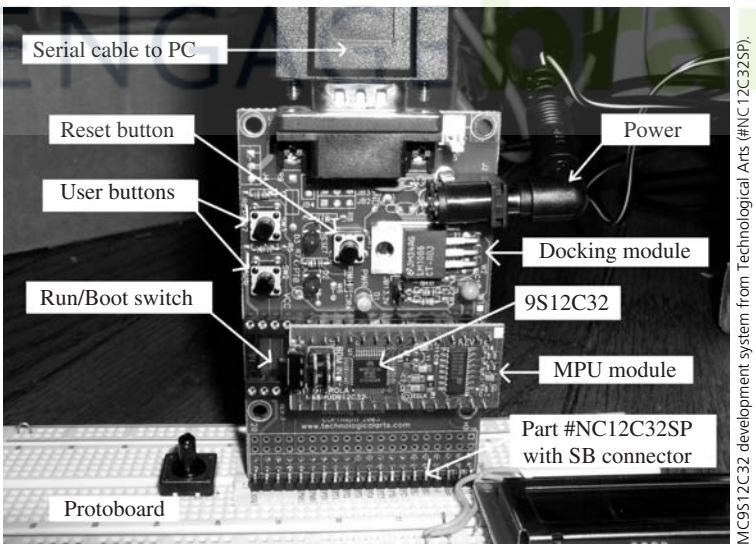


Figure 1.49
MC9S12C32 development system from Technological Arts (#NC12C32SP).

A third approach, is typically used after a successful evaluation with one of the previous methods. In this approach, we design a printed circuit board (PCB) including both the external circuits and the microcontroller itself. The 9S12, using the background debug module, has facilities to download software onto the microcontroller.

During the testing phase of the project, we observe that all four of the LEDs are continuously on. We use the software debugger to single-step our program, which correctly outputs the 0101, 0110, 1010, 1001 binary repeating pattern. During this single stepping, the

LEDs do come on and off in the proper pattern. Using a voltmeter on the circuit, we observe the 0.4V signal on the output of the 7405 whenever the software wishes to turn the LED on. We run the system at full speed again and observe two 7405 outputs on the oscilloscope, collecting data presented as Figure 1.50. Simulation, shown in Figure 1.48, also seems to be completely functional. All the electronic tests show the system is running properly, but our eyes still see all four LEDs continuously on.

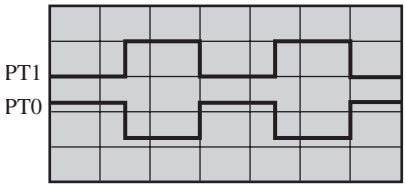


Figure 1.50
Oscilloscope waveforms collected during the testing of the LED output system. The voltage sensitivity is 5V/division, and the time base is 1μs/division.

Checkpoint 1.43: What is the error in this design, and how do we fix it?

Portability is a measure of how easy it is to convert software that runs on one machine to run on another machine.

Observation: In general, C code is more portable than assembly language.

1.10 Choosing a Microcontroller

I chose to focus this book on the 9S12, because year after year Freescale continues to be a major supplier of microcontrollers. Sometimes, the computer engineer is faced with the task of selecting the microcontroller for the project. When faced with this decision, some engineers consider only those devices for which they have hardware and software experience. Fortunately, this blind approach often yields an effective and efficient product, because many microcontrollers overlap in their cost and performance. In other words, if a familiar microcontroller can implement the desired functions for the project, then it is often efficient to bypass that more perfect piece of hardware in favor of a faster development time. On the other hand, sometimes we wish to evaluate all potential candidates. It may be cost-effective to hire or train the engineering personnel so that they are proficient in a wide spectrum of potential microcontroller devices. There are many factors to consider when selecting a microcontroller, including the following:

- Labor costs include training, development, and testing
- Material costs include parts and supplies
- Manufacturing costs depend on the number and complexity of the components
- Maintenance costs involve revisions to fix bugs and perform upgrades
- ROM size must be big enough to hold instructions and fixed data for the software
- RAM size must be big enough to hold locals, parameters, and global variables
- EEPROM must hold nonvolatile fixed constants that are field-configurable
- Processor must be capable of performing all calculations in real time
- I/O bandwidth determines the input/output rate
- 8-, 16-, or 32-bit data size should match most of the data to be processed
- Numerical operations may be required, such as multiply, divide, signed, floating point
- Special functions may be required, such as multiply/accumulate, fuzzy logic, complex numbers
- Parallel ports are needed for the input/output digital signals
- Serial ports are used to interface with other computers or I/O devices
- Timer functions are used to generate signals, measure frequency, measure period

- Pulse-width modulation is used for the output signals in many control applications
- ADC is used to convert analog inputs to digital numbers
- Package size and environmental issues affect many embedded systems
- Second source availability increases the chance parts can be purchased
- Availability of high-level language cross-compilers, simulators, emulators is important
- Power requirements, are important because many systems will be battery operated

When considering speed, it is best to compare time to execute a benchmark program similar to your specific application, rather than just comparing bus frequency. One of the difficulties is that the microcomputer selection depends on the speed and size of the software, but the software cannot be written without the computer. Given this uncertainty, it is best to select a family of devices with a range of execution speeds and memory configurations. In this way a prototype system with large amounts of memory and peripherals can be purchased for software and hardware development, and once the design is in its final stages, the specific version of the computer can be selected now that the memory and speed requirements for the project are known. In conclusion, while this book focuses on the 9S12 microcontroller, it is expected that once the study of this book is completed, the reader will be equipped with the knowledge needed to select the proper microcontroller and complete the software design.

1.11 Exercises

For these questions, substitute your specific 9S12 for *YourComputer* as appropriate.

1.1 Is RAM volatile or nonvolatile?

1.2 Assuming *YourComputer* is running in expanded mode, what are the names of its bus signals?

1.3 Consider *YourComputer* with memory-mapped I/O. Assume there is no direct memory access (DMA) on your microcomputer. For this problem, we specify four classes of devices: *processor*, *RAM*, *ROM*, and *I/O*.

a) List the devices that can drive the address bus during a CPU read cycle.
b) List the devices that can drive the address bus during a CPU write cycle.
c) List the devices that can drive the data bus during a CPU read cycle.
d) List the devices that can drive the data bus during a CPU write cycle.
e) List the devices that can drive the R/W line during a CPU read cycle.
f) List the devices that can drive the R/W line during a CPU write cycle.
g) List the devices that can receive the information from data bus during a CPU read cycle.
h) List the devices that can receive the information from data bus during a CPU write cycle.

1.4 How many 74LS low-power Schottky inputs can an output of *YourComputer* drive?

1.5 Consider the current it takes to power a +5 V 74xx245 (I_{CC}) as shown in Table 1.5. What is the qualitative difference between the CMOS devices and the non-CMOS devices? What is the explanation for the difference?

1.6 How many alternatives does a 14-bit ADC have?

1.7 If a system uses a 12-bit ADC, about how many decimal digits will it have?

1.8 If a system requires $3\frac{1}{2}$ decimal digits of precision, what is the smallest number of bits the ADC needs to have?

1.9 How many alternatives does a 16-bit ADC have?

1.10 If a system uses an 11-bit ADC, about how many decimal digits will it have?

1.11 If a system requires $2\frac{3}{4}$ decimal digits of precision, what is the smallest number of bits the ADC needs to have?

1.12 If a system requires $4\frac{3}{4}$ decimal digits of precision, what is the smallest number of bits the ADC needs to have?

1.13 Convert the following decimal numbers to 8-bit unsigned binary: 25, 63, 125, and 200.

1.14 Convert the following decimal numbers to 8-bit signed binary: 25, 63, −125, and −2.

1.15 Convert the following hex numbers to unsigned decimal: $25, $63, $A3, and $FE.

1.16 Convert the 16-bit binary number %0010000001101010 to unsigned decimal.

1.17 Convert the 16-bit hex number $1234 to unsigned decimal.

1.18 Convert the unsigned decimal number 1234 to 16-bit hexadecimal.

1.19 Convert the unsigned decimal number 10000 to 16-bit binary.

1.20 Convert the 16-bit hex number $1234 to signed decimal.

1.21 Convert the 16-bit hex number $ABCD to signed decimal.

1.22 Convert the signed decimal number 1234 to 16-bit hexadecimal.

1.23 Convert the signed decimal number −10000 to 16-bit binary.

1.24 List the registers in *YourComputer*.

1.25 How much RAM and ROM are in *YourComputer*? What are the specific address ranges of these memory components?

1.26 What are the bits in the CCR of *YourComputer*?

1.27 What are the bidirectional ports on *YourComputer*?

1.28 What are the unidirectional ports on *YourComputer* (if any)?

1.29 What is a direction register?

1.30 Why does the microcomputer have direction registers?

D1.31 Design the circuit that interfaces a 3 V, 5 mA LED to *YourComputer*.

D1.32 Design the circuit that interfaces a 2.5 V, 1 mA LED to *YourComputer*.

D1.33 Redesign the switch interface shown in the middle of Figure 1.21 assuming the signal `S` is to be connected to a 74LS04 instead of the input port of the microcontroller. In particular, change the 10 kΩ resistor value to the appropriate value for low-power Schottky logic.

D1.34 Write software that initializes 9S12 Port T so pins 7,5,3,1 are output and the rest are input.

D1.35 Write software that initializes 9S12 Port T so pins 5,4 are output and the rest are input.

D1.36 You can make the 6811 Port C generate open collector outputs by setting appropriate bit in the direction register, DDRC, to 1 and setting the CWOM bit in the PIOC register to 1. This open collector mode can be found in some of the ports of the Intel 8051 microcontroller family, but it is missing in the 9S12.

Freescale claims you can easily upgrade 6811 systems to the more powerful 9S12. You have just graduated from the Prestigious University, and the first task at your new high-paying job at *We Are Nerds, INC,* is to upgrade an existing *WAN INC* 6811 system to the 9S12. It seems easy, so you begin by reading all about the 9S12. The 9S12 has more instructions and addressing modes, but luckily all existing 6811 assembly instructions and addressing modes are still available on the 9S12. The 9S12 has more parallel ports, more serial ports, more input captures, and more output compares.

So far so good, but all of a sudden BAM it hits you—looking at you square in the face is a big problem. The existing system uses open collector output mode with the 6811 bit CWOM set, but the 9S12 has no open collector modes on any of its parallel port outputs. So now you, the young engineering genius from PU, must solve your first engineering problem.

Lucky for you, the engineers that worked on the problem previously were also from PU. They implemented the low-level access to PORTC as a device driver, and organized the software with a layered approach. This layered system will allow you to come in and replace the low (hardware access) level, without having to modify the upper levels. In particular, here are the existing low-level routines in both assembly and C.

```
;Initialize Parallel Port
;Input: Reg B specifies which port bits will be input(0) or
output(1)
;Outputs: none
Pinit  ldaa PIOC
       oraa #$20 ; set CWOM so outputs are open collector
       staa PIOC
       stab DDRC ; 1 means open collector output, 0 means input
       ldaa #$FF
       stab PORTC ; any output pins are initialized to HiZ
       rts
;Set output (only output pins are effected)
;Input: Reg B specifies new output values
;Outputs: none
Pout   stab PORTC ; modifies output pins only
       rts
; Get input
; Input: none
; Outputs: Reg B returned with current values of both inputs
and outputs
Pin    ldab PORTC
       rts
void Pinit(unsigned char direction){
  PIOC |= 0x20; // set CWOM so outputs are open collector
  DDRC = direction; // 1 means open collector output, 0 means input
  Pout(0xFF);} // any output pins are initialized to HiZ
void Pout(unsigned char data){
  PORTC = data;} // modifies output pins only
unsigned char Pin(void){
  return PORTC;}
```

Your specific task: Rewrite the three device driver routines to run on the 9S12, either in assembly or in C. You may use any 9S12 assembly language instructions and addressing modes. Refer to the 9S12 parallel port using the symbol PTT and to the direction register using the symbol DORT. A global variable will be required.

Test your answer with this example: Assume the four input signals (C7–C4) are set by external hardware to C7=0,C6=1,C5=0, C4=0. Assume the four output signals (C3–C0) have +5 V pull-up resistors.

```
void main(void){ unsigned char info;
    Pinit(0x0F);  // C7-C4 inputs(HiZ), C3-C0 outputs are HiZ
    info=Pin();   // info set to 0x4F (because of pullups)
    Pout(0x00);   // C7-C4 still inputs(HiZ), but C3-C0 are low
    info=Pin();   // info set to 0x40
    Pout(0x05);   // C7-C4,C2,C0 are HiZ, but C3,C1 are low
    info=Pin(); } // info set to 0x45
```

1.12 Lab Assignments

The labs in this book involve the following steps:

Part a) During the analysis phase of the project, determine additional specifications and constraints. In particular, discover which microcontroller you are to use, whether you are to develop in assembly language or in C, and whether the project is to be simulated then built, just built, or just simulated. For example, inputs can be created with switches, and outputs can be generated with LEDs. The SCI can be interfaced to a PC, and a communication program such as HyperTerminal can be used to interact with the system.

Part b) Design, build, and test the hardware interfaces. Use a computer-aided-drawing (CAD) program to draw the hardware circuits. Label all pins, chips, and resistor values. In this chapter, there will be one switch for each input and one LED for each output. Connect the switch interfaces to the 9S12 input pins, and connect the LED interfaces to the 9S12 output pins. Pressing the switch will signify a high input logic value. You should activate the LED to signify a high output logic value.

Part c) Design, implement, and test the software that initializes the I/O ports and performs the specified function. Often a main program is used to demonstrate the system.

Lab 1.1. The overall objective is to create a `not` gate. The system has one digital input and one digital output, such that the output is the logical complement of the input. Implement the design such that the complement function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `coma lsra` and `lsla` instructions.

Lab 1.2. The overall objective is to create a 3-input `and` gate. The system has three digital inputs and one digital output, such that the output is the logical `and` of the three inputs. Implement the design such that the `and` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `anda lsra` and `lsla` instructions.

Lab 1.3. The overall objective is to create a 3-input `or` gate. The system has three digital inputs and one digital output, such that the output is the logical `or` of the three inputs. Implement the design such that the `or` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `oraa lsra` and `lsla` instructions.

Lab 1.4. The overall objective is to create a 2-input `exclusive or` gate. The system has two digital inputs and one digital output, such that the output is the logical `exclusive or` of the two inputs. Implement the design such that the exclusive or function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `eora lsra` and `lsla` instructions.

Lab 1.5. The overall objective is to create a 3-input `voting` logic. The system has three digital inputs and one digital output, such that the output is high if and only if two or more inputs are high. This means the output will be low if two for more inputs are low. Implement the design such that the `voting` function occurs in the software of the 9S12. If you are writing in assembly, you may wish to investigate the `anda oraa lsra` and `lsla` instructions.