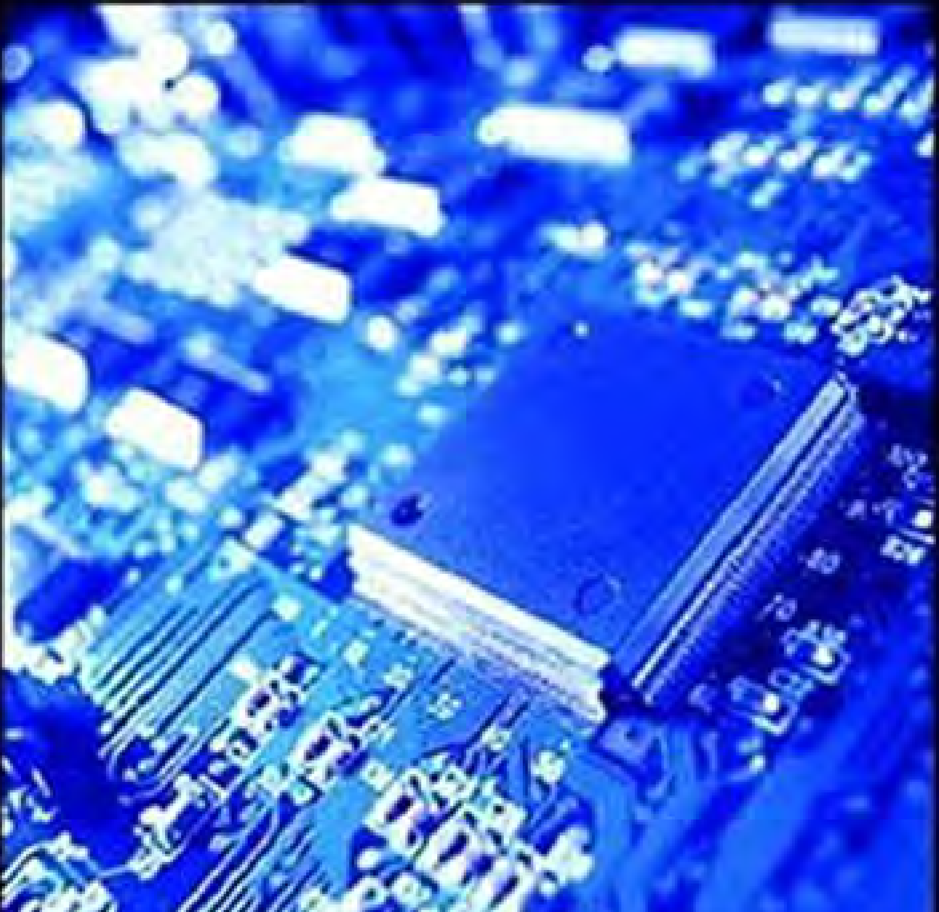
FOURTH EDITION

THE 8051 MICROCONTROLLER



I. SCOTT MACKENZIE

RAPHAEL C.-W. PHAN

THE 8051 MICROCONTROLLER

FOURTH EDITION

I. Scott MacKenzie

York University

Raphael C.-W. Phan

Swinburne University of Technology
(Sarawak Campus)

PEARSON
Prentice
Hall

Pearson Education International

Editor-in-Chief: Vernon Anthony
Production Editor: Rex Davidson
Design Coordinator: Diane Ernsberger
Editorial Assistant: Lara Dimmick
Cover Designer: Candace Rowley
Cover art: Getty Images
Production Manager: Matt Ottenweller
Senior Marketing Manager: Ben Leonard
Marketing Assistant: Les Roberts
Senior Marketing Coordinator: Liz Farrell

This book was set in Times Roman by Laserwords Pte. Ltd. It was printed and bound by R. R. Donnelley & Sons Company. The cover was printed by Coral Graphic Services, Inc.

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd
Pearson Education North Asia Ltd
Pearson Education Canada, Ltd.
Pearson Educacon de Mexico, S.A. de C.V.
Pearson Education -- Japan
Pearson Education Malaysia, Pte. Ltd
Pearson Education, Upper Saddle River, New Jersey

10 9 8 7 6 5 4 3 2 1
ISBN 0-13-205975-4

PREFACE

This book examines the hardware and software features of the MCS-51 family of microcontrollers. The intended audience is college or university students of electronics, computer technology, and electrical or computer engineering, or practicing technicians or engineers interested in learning about microcontrollers.

The means to effectively fulfill that audience's informational needs were tested and refined in the development of this book. In its prototype form, *The 8051 Microcontroller* was the basis of a fifth-semester course for college students in computer engineering. As detailed in Chapter 11, students built an 8051 single-board computer as part of this course. That computer, in turn, has been used as the target system for a final, sixth-semester "project" course in which students design, implement, and document a "product" controlled by the 8051 microcontroller and incorporating original software and hardware.

Because the 8051—like all microcontrollers—contains a high degree of functionality, the book emphasizes architecture and programming rather than electrical details. The software topics are delivered in the context of Intel's assembler (ASM51) and linker/locator (RL51).

Four new chapters are included in this edition, and the main additional feature is the information about using 8051 C programming as an alternative to the assembly language used in earlier editions. Programming in C allows for structured programs and is especially useful in coding big and complex 8051-based projects.

All examples are annotated to assist both the student and the teacher. The examples begin by stating a problem followed by a straightforward solution. Then, following the solution, there is a discussion that explores the inner workings of the problem and the solution. The approach is to explain and to elaborate, taking into account different perspectives that enter into the example.

It is our view that courses on microprocessors or microcontrollers are inherently more difficult to deliver than courses in, for example, digital systems, because a linear sequence of topics is hard to devise. The very first program that is demonstrated to students brings with it significant assumptions, such as a knowledge of the CPU's programming model and addressing modes, the distinction between an address and the content of an address, and so on. For this reason, a course based on this book should not attempt to follow strictly the sequence presented. Chapter 1 is a good starting point, however. It serves as a general introduction to microcontrollers, with particular emphasis on the distinctions between microcontrollers and microprocessors.

Chapter 2 introduces the hardware architecture of the 8051 microcontroller and its counterparts that form the MCS-51 family. Concise examples are presented using short sequences of instructions. Instructors should be prepared at this point to introduce, in parallel, topics from Chapters 3 and 7 and Appendices A and C to support the requisite software knowledge in these examples. Appendix A is particularly valuable, since it contains in a single figure the entire 8051 instruction set.

Chapter 3 introduces the instruction set, beginning with definitions of the 8051's addressing modes. The instruction set has convenient categories of instructions (data transfer, branch, etc.) that facilitate a step-wise presentation. Numerous brief examples demonstrate each addressing mode and each type of instruction.

Chapters 4, 5, and 6 progress through the 8051's on-chip features, beginning with the timers, advancing to the serial port (which requires a timer as a baud rate generator), and concluding with interrupts. The examples in these chapters are longer and more complex than those presented earlier. Instructors are wise not to rush into these chapters; it is essential that students gain solid understanding of the 8051's hardware architecture and instruction set before advancing to these topics.

Many of the topics in Chapter 7 will be covered, by necessity, in progressing through the first six chapters. Nevertheless, this chapter is perhaps the most important for developing in students the potential to undertake large-scale projects. Advanced topics such as assemble-time expression evaluation, modular programming, linking and locating, and macro programming will be a significant challenge for many students. At this point, the importance of hands-on experience cannot be overemphasized. Students should be encouraged to experiment by entering the examples in the chapter into the computer and observing the output and error messages provided by ASM51, RL51, and the object-to-hex conversion utility (OH).

Chapter 8 lays the foundation for C programming the 8051. It highlights differences between this higher-level language compared to assembly language, and differences between conventional C language for computer systems and C for an embedded microcontroller such as the 8051.

Some advanced topics relating to programming methods, style, and the development environment are presented in Chapters 9 and 10. These chapters address larger, more conceptual topics important in professional development environments.

Chapter 11 presents several design examples incorporating selected hardware with supporting software. The software is fully annotated and is the real focus in these examples. The fourth edition includes several additional interfaces: a liquid crystal display (LCD), the 8255, an RS-232 serial interface, a Centronics parallel interface, sensors, relays, and a stepper motor. One of the designs in Chapter 11 is the SBC-51 - the 8051 single-board computer. The SBC-5l can form the basis of a course on the 8051 microcontroller. A short monitor program is included (see Appendix G), which is sufficient to get "up and running." A development environment also requires a host computer, which doubles as a dumb terminal for controlling the SBC-5l after programs have been downloaded for execution.

Many dozens of students have wire-wrapped prototype versions of the SBC during years that Scott has taught 8051-based courses to computer engineering students. Raphael also thanks his Microprocessor Fundamentals, Microprocessor Applications, and Embedded

Microcontrollers students, who enthusiastically undertook assignments and projects based on the 8051.

There is also a new chapter, Chapter 12, on the design and interface examples given in Chapter 11, but with the solutions in C rather than in assembly language.

Chapter 13 presents some more advanced examples of 8051 projects for students and concentrates on the discussion of design choices and the importance of pseudo code in the design process, prior to the actual coding.

Chapter 14 talks briefly about some 8051 derivative devices that are descendants of the 8051 but with enhancements such as increases in speed and memory size, additional built-in peripherals, and enhanced network capabilities and security mechanisms.

Also worth mentioning is the treatment of smart cards and data security in this edition, notably in Chapters 12, 13, and 14, and in Appendix J. This information is included because of the increasing popularity of smart cards using 8-bit microcontrollers such as the 8051 to run security software to protect confidential information.

The book makes extensive use of and builds on Intel's literature on the MCS-51 devices. In particular, Appendix C contains the definitions of all 8051 instructions, and Appendix E contains the 8051 data sheet. Intel's cooperation is gratefully acknowledged.

All the 8051 C examples in this edition have been compiled, debugged, and tested with Keil's ,µision2 IDE, available for download at http://www.keil.com. We also thank the following for their review and invaluable comments, criticism, and suggestions: Dwight Egbert, University of Nevada; Marty Kaliski, Cal Polytech State University; Claude Kansaku, Oregon Institute of Technology; and Ron Tinkham, Santa Fe Community College. Raphael thanks his wife, Grace, for her understanding and patience, and for sacrificing all the nights, weekends, and public holidays to keep him company in writing this edition. In fact, without her gentle nudges, this edition would not have been completed. This edition is dedicated to her.

I. Scott MacKenzie Raphael C.-W.
Phan

CONTENTS

1 INTRODUCTION TO MICROCONTROLLERS 1

2 HARDWARE SUMMARY 17

5 SERIAL PORT OPERATION 111

6 INTERRUPTS 131

7 ASSEMBLY LANGUAGE PROGRAMMING 151

8 8051 C PROGRAMMING 191

9 PROGRAM STRUCTURE AND DESIGN 217

10 TOOLS AND TECHNIQUES FOR PROGRAM DEVELOPMENT 247

11 DESIGN AND INTERFACE EXAMPLES 259

12 DESIGN AND INTERFACE EXAMPLES IN C 319

13 EXAMPLE STUDENT PROJECTS 353

14 8051 DERIVATIVES 377

APPENDICES

BIBLIOGRAPHY 527

INDEX 529

CHAPTER 1

Introduction to Microcontrollers

1.1 INTRODUCTION

Although computers have been with us for only a few decades, their impact has been profound, rivaling that of the telephone, automobile, or television. Their presence is felt by us all, whether computer programmers or recipients of monthly bills printed by a large computer system and delivered by mail. Our notion of computers usually categorizes them as "data processors," performing numeric operations with inexhaustible competence.

We confront computers of a vastly different breed in a more subtle context performing tasks in a quiet, efficient, and even humble manner, their presence often unnoticed. As a central component in many industrial, automotive, and consumer products, we find computers at the supermarket inside cash registers and scales; at home in ovens, washing machines, alarm clocks, and thermostats; at play in toys, VCRs, stereo equipment, and musical instruments; at the office in typewriters and photocopiers; in cars in dashboards and ignition systems; and in industrial equipment such as drill presses and phototypesetters. In these settings computers are performing "control" functions by interfacing with the "real world" to turn devices on and off and to monitor conditions. **Microcontrollers** (as opposed to microcomputers or microprocessors) are often found in applications such as these.

It's hard to imagine the present world of electronic tool toys without the microprocessor. Yet this single-chip wonder has barely reached its 35th birthday. In 1971 Intel Corporation introduced the 8080, the first successful microprocessor. Shortly thereafter, Motorola, RCA, and then MOS Technology and Zilog introduced similar devices: the 6800, 1801, 6502, and Z80, respectively. Alone these integrated circuits (ICs) were rather helpless (and they remain so); but as part of a single-board computer (SBC) they became the central component in useful products for learning about and designing with microprocessors. These SBCs, of which the *D2* by Motorola, *KIM-1* by MOS Technology, and *SDK-85* by Intel are the most memorable, quickly found their way into design labs at colleges, universities, and electronics companies.

A device similar to the microprocessor is the microcontroller. In 1976 Intel introduced the 8748, the first device in the MCS-48^{TM} family of microcontrollers. Within a single integrated circuit containing over 17,000 transistors, the 8748 delivered a CPU, 1K byte of EPROM, 64 bytes of RAM, 27 I/O pins, and an 8-bit timer. This IC, and other MCS-48^{TM} devices that followed, soon became an industry standard in control-oriented applications. Replacement of electromechanical components in products such as washing machines and traffic light controllers was a popular application initially and remains so. Other products where microcontrollers can be found include automobiles, industrial equipment, consumer entertainment products, and computer peripherals. (Owners of an IBM *PC* need only look inside the keyboard for an example of a microcontroller in a minimum-component design.)

The power, size, and complexity of microcontrollers advanced an order of magnitude in 1980 with Intel's announcement of the 8051, the first device in the MCS-51^{TM} family of microcontrollers. In comparison to the 8048, this device contains over 60,000 transistors, 4K bytes ROM, 128 bytes of RAM, 32 I/O lines, a serial port, and two 16-bit timers—a remarkable amount of circuitry for a single IC (see Figure 1-1). New members have been added to the MCS-51^{TM} family, and today variations exist virtually doubling these specifications. Siemens Corporation, a second source for MCS-51^{TM} components, offers the SAB80515, an enhanced 8051 in a 68-pin parade with six 8-bit I/O ports, 13 interrupt sources, and an 8-bit A/D converter with eight input channels. Chapter 14 also discusses several other enhanced variants of the 8051. The 8051 family is well established as one of the most versatile and powerful of the 8-bit microcontrollers, its position as a leading microcontroller entrenched for years to come.

This book is about the MCS-51^{TM} family of microcontrollers. The following chapters introduce the hardware and software architecture of the MCS-51^{TM} family and demonstrate through numerous design examples how this family of devices can participate in electronic designs with a minimum of additional components.

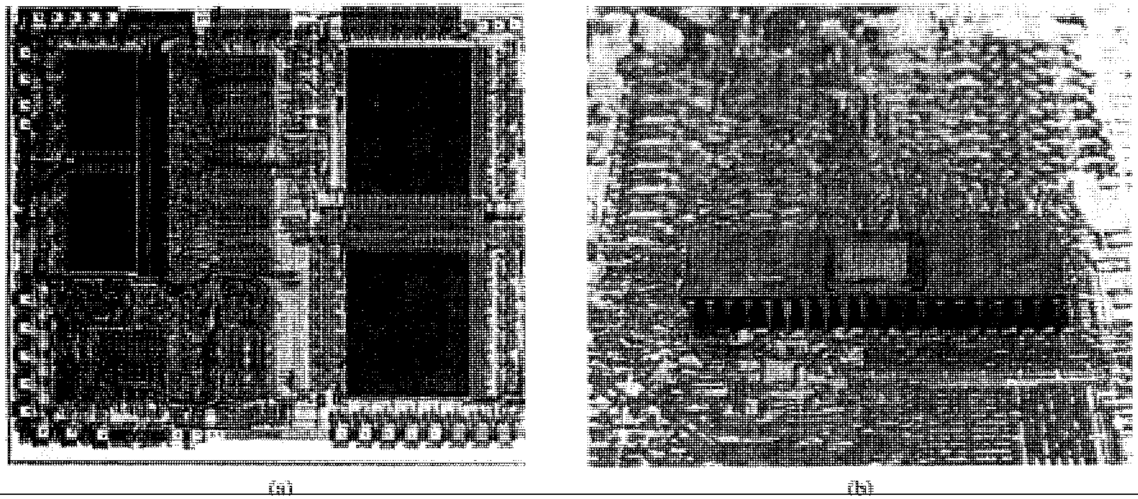


FIGURE 1-1
The 8051 microcontroller. (a) An 8051 die (b) An 8751 with on-chip EPROM (Courtesy Intel Corporation)

In the following sections, through a brief introduction to computer architecture, we shall develop a working vocabulary of the many acronyms and buzz words that prevail (and often confound) in this field. Since many terms have vague and overlapping definitions subject to the prejudices of large corporations and the whims of various authors, our treatment is practical rather than academic. Each term is presented in its most common setting with a straightforward explanation.

1.2 TERMINOLOGY

To begin, a **computer** is defined by two key traits: (1) the ability to be programmed to operate on data without human intervention, and (2) the ability to store and retrieve data. More generally, a **computer system** also includes the **peripheral devices** for communicating with humans, as well as **programs** that process data. The equipment is **hardware,** the programs are **software.** Let's begin with computer hardware by examining Figure 1-2.

The absence of detail in the figure is deliberate, making it representative of all sizes of computers. As shown, a computer system contains a **central processing unit** (CPU) connected to **random access memory** (RAM) and **read-only memory** (ROM) via the **address bus, data bus,** and **control bus. Interface circuits** connect the system buses to **peripheral devices.** Let's discuss each of these in detail.

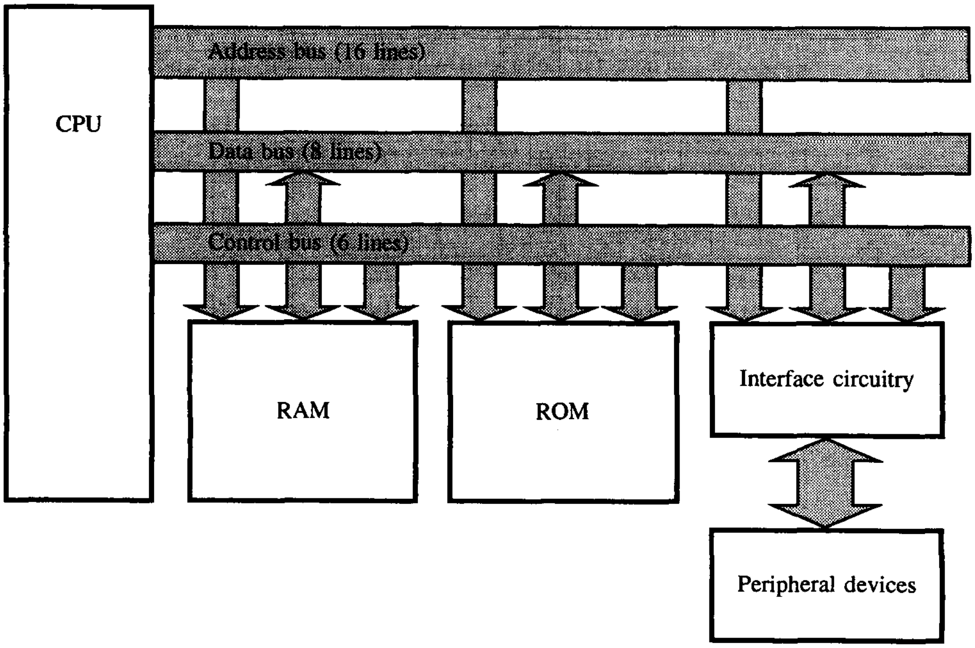


FIGURE 1-2
Block diagram of a microcomputer system

1.3 THE CENTRAL PROCESSING UNIT

The CPU, as the "brain" of the computer system, administers all activity in the system and performs all operations on data. Most of the CPU's mystique is undeserved, since it is just a collection of logic circuits that continuously performs two operations: fetching instructions and executing instructions. The CPU has the ability to understand and execute instructions based on a set of binary codes, each representing a simple operation. These instructions are usually arithmetic (add, subtract, multiply, divide), logic (AND, OR, NOT, etc.), data movement, or branch operations, and are represented by a set of binary codes called the **instruction set.**

Figure 1-3 is an extremely simplified view of the inside of a CPU. It shows a set of **registers** for the temporary storage of information, an **arithmetic and logic unit** (ALU) for performing operations on this information, an **instruction decode and control unit** that determines the operation to perform and sets in motion the necessary actions to perform it, and two additional registers. The **instruction register (IR)** holds the binary code for each instruction as it is executed, and the **program counter (PC)** holds the memory address of the next instruction to be executed.

Fetching an instruction from the system RAM or ROM is one of the most fundamental operations performed by the CPU. It involves the following steps: (a) the contents of the program counter are placed on the address bus, (b) a READ control signal is activated, (c) data (the instruction opcode) are read from RAM and placed on the data bus, (d) the op-code is latched into the CPU's internal instruction register, and (e) the program counter is incremented to prepare for the next fetch from memory. Figure 1-4 illustrates the flow of information for an instruction fetch.

The execution stage involves decoding (or deciphering) the opcode and generating control signals to gate internal registers in and out of the ALU and to signal the ALU to perform

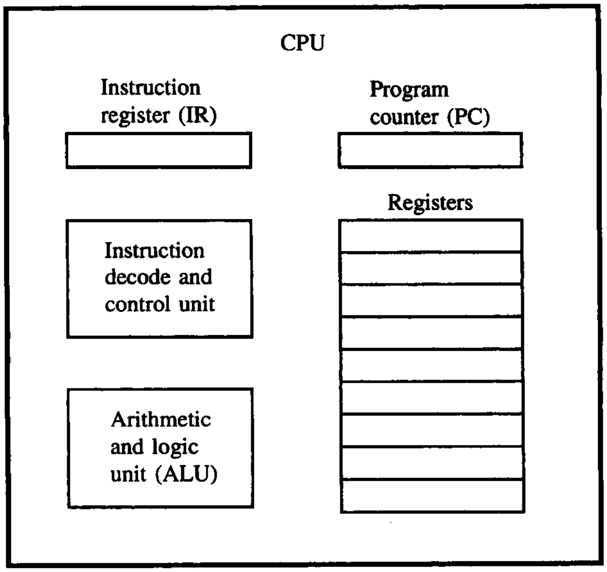

CPU
Instruction register (IR)
Program counter (PC)
Registers
Instruction decode and control unit
Arithmetic and logic unit (ALU)

FIGURE 1-3
The central processing unit (CPU)

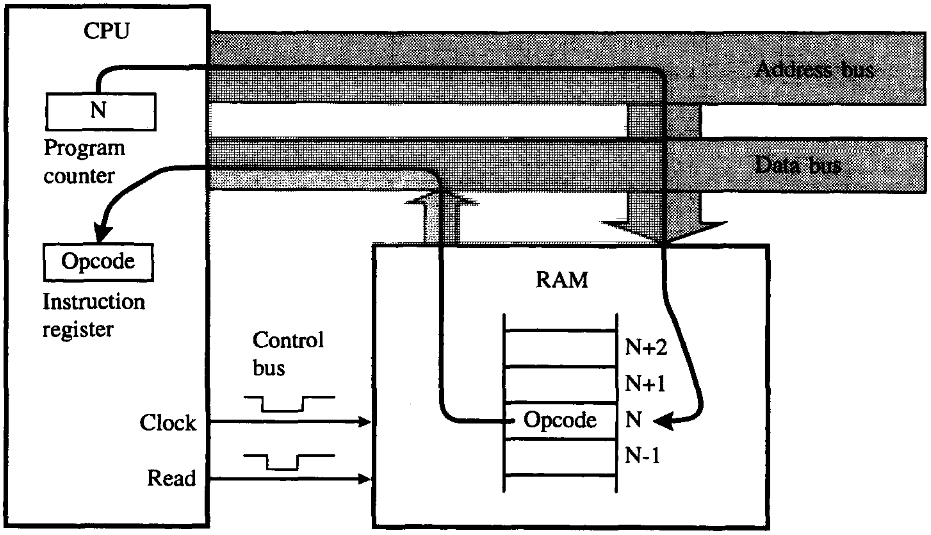


FIGURE 1-4
Bus activity for an opcode fetch cycle

the specified operation. Due to the wide variety of possible operations, this explanation is somewhat limited in scope. It applies to a simple operation such as "increment register." More complex instructions require more steps, such as reading a second and third byte as data for the operation.

A series of instructions combined to perform a meaningful task is called a **program,** or **software,** and herein is the real mystique. The degree to which tasks are efficiently and correctly carried out is determined for the most part by the quality of software, not by the sophistication of the CPU. Programs, then, "drive" the CPU, and in doing so they occasionally go amiss, mimicking the frailties of their authors. Phrases such as "The computer made a mistake" are misguided. Although equipment breakdowns are inevitable, mistakes in results are usually a sign of poor programs or operator error.

1.4 SEMICONDUCTOR MEMORY: RAM AND ROM

Programs and data are stored in memory. The variations of computer memory are so vast, their accompanying terms so plentiful, and technology breakthroughs so frequent, that extensive and continual study is required to keep abreast of the latest developments. The memory devices directly accessible by the CPU consist of semiconductor ICs (integrated circuits) called RAM and ROM. There are two features that distinguish RAM and ROM: first, RAM is read/write memory while ROM is read-only memory; and second, RAM is volatile (the contents are lost when power is removed), while ROM is nonvolatile.

Most computer systems have a disk drive and a small amount of ROM, just enough to hold the short, frequently used software routines that perform input/output operations.

User programs and data are stored on disk and are loaded into RAM for execution. With the continual drop in the per-byte cost of RAM, small computer systems often contain millions of bytes of RAM.

1.5 THE BUSES: ADDRESS, DATA, AND CONTROL

A bus is a collection of wires carrying information with a common purpose. Access to the circuitry around the CPU is provided by three buses: the **address bus, data bus,** and **control bus.** For each read or write operation, the CPU specifies the location of the data (or instruction) by placing an address on the address bus and then activates a signal on the control bus, indicating whether the operation is a read or write. Read operations retrieve a byte of data from memory at the location specified and place it on the data bus. The CPU reads the data and places it in one of its internal registers. For a write operation, the CPU outputs data on the data bus. Because of the control signal, memory recognizes the operation as a write cycle and stores the data in the location specified.

Most small computers have 16 or 20 address lines. Given *n* address lines, each with the possibility of being high (1) or low (0), 2^n locations can be accessed. A 16-bit address bus, therefore, can access $2^{16} = 65{,}536$ locations, and a 20-bit address can access $2^{20} = 1{,}048{,}576$ locations. The abbreviation *K* (for kilo) stands for $2^{10} = 1024$; therefore, 16 bits can address $2^6 \times 2^{10} = 64K$ locations, whereas 20 bits can address 1024K or 1 M locations. The abbreviation M (for mega) stands for $2^{20} = 1024 \times 1024 = 1024K = 1{,}048{,}576$.

The data bus carries information between the CPU and memory or between the CPU and I/O devices. Extensive research effort has been expended in determining the sort of activities that consume a computer's valuable execution time. Evidently computers spend up to two thirds of their time simply moving data. Since the majority of move operations are between a CPU register and external RAM or ROM, the number of lines (the width) of the data bus is important for overall performance. This limitation-by-width is a bottleneck: There may be vast amounts of memory on the system, and the CPU may possess tremendous computational power, but access to the data—data movement between the memory and CPU via the data bus—is bottlenecked by the width of the data bus.

This trait is so important that it is common to add a prefix indicating the extent of this bottleneck. The phrase "16-bit computer" refers to a computer with 16 lines on its data bus. Most computers fit the 4-bit, 8-bit, 16-bit, or 32-bit classification, with overall computing power increasing as the width of the data bus increases.

Note that the data bus, as shown in Figure 1-2, is bidirectional, and the address bus is unidirectional. Address information is always supplied by the CPU (as indicated by the arrow in Figure 1-2), yet data may travel in either direction depending on whether a read or write operation is intended.[1] Note also that the term "data" is used in a general sense: the "information" that travels on the data bus may be the instructions of a program, an address appended to an instruction, or the data used by the program.

[1] Address information is sometimes also provided by direct memory access (DMA) circuitry (in addition to the CPU).

The control bus is a hodgepodge of signals, each having a specific role in the orderly control of system activity. As a rule, control signals are timing signals supplied by the CPU to synchronize the movement of information on the address and data buses. Although there are usually three signals, such as CLOCK, READ, and WRITE, for basic data movement between the CPU and memory, the names and operation of these signals are highly dependent on the specific CPU. The manufacturer's data sheets must be consulted for details.

1.6 INPUT/OUTPUT DEVICES

I/O devices, or "computer peripherals," provide the path for communication between the computer system and the "real world." Without these, computer systems would be rather introverted machines, of little use to the people who use them. Three classes of I/O devices are **mass storage, human interface,** and **control/monitor.**

1.6.1 Mass Storage Devices

Like semiconductor RAMS and ROMs, mass storage devices are players in the arena of memory technology—constantly growing, ever improving. As the name suggests, they hold large quantities of information (programs or data) that cannot fit into the computer's relatively small RAM or "main" memory. This information must be loaded into main memory before the CPU accesses it. Classified according to ease of access, mass storage devices are either **online** or **archival.** Online storage, usually on magnetic disk, is available to the CPU without human intervention upon the request of a program, and archival storage holds data that are rarely needed and require manual loading onto the system. Archival storage is usually on magnetic tapes or disks, although optical discs, such as CD-ROM or WORM technology, are now emerging and may alter the notion of archival storage due to their reliability, high capacity, and low cost.[2]

1.6.2 Human Interface Devices

The union of human and machine is realized by a multitude of human interface devices, the most common being the video display terminal (VDT) and printer. Although printers are strictly output devices that generate hardcopy output, VDTs are really two devices, since they contain a keyboard for input and a CRT (cathode-ray tube) for output. An entire field of engineering, called "ergonomics" or "human factors," has evolved from the necessity to design these peripheral devices with humans in mind, the goal being the safe, comfortable, and efficient mating of the characteristics of people with the machines they use. Indeed, there are more companies that manufacture this class of peripheral device than companies that manufacture computers. For most computer systems, there are at least three of these devices: a keyboard, CRT, and printer. Other human interface devices include the joystick, light pen, mouse, microphone, and loudspeaker.

[2] "CD-ROM" stands for compact-disc read-only memory. "WORM" stands for write-once read-mostly. A CD-ROM contains 700 Mbyte of storage, enough to store the entire 32 volumes of *Encyclopedia Britannica.*

1.6.3 Control/Monitor Devices

By way of control/monitor devices (and some meticulously designed interface electronics and software), computers can perform a myriad of control-oriented tasks, and perform them unceasingly, without fatigue, far beyond the capabilities of humans. Applications such as temperature control of a building, home security, elevator control, home appliance control, and even welding parts of an automobile, are all made possible using these devices.

Control devices are outputs, or **actuators,** that can affect the world around them when supplied with a voltage or current (e.g., motors and relays). Monitoring devices are inputs, or **sensors,** which are stimulated by heat, light, pressure, motion, etc., and convert this energy to a voltage or current read by the computer (e.g., phototransistors, thermistors, and switches). The interface circuitry converts the voltage or current to binary data, or vice versa, and through software an orderly relationship between inputs and outputs is established. The hardware and software interfacing of these devices to microcontrollers is one of the main themes in this book.

1.7 PROGRAMS: BIG AND SMALL

The preceding discussion has focused on computer systems hardware with only a passing mention of the programs, or software, that make them work. The relative emphasis placed on hardware versus software has shifted dramatically in recent years. Whereas the early days of computing witnessed the materials, manufacturing, and maintenance costs of computer hardware far surpassing the software costs, today, with mass-produced LSI (large-scale integrated) chips, hardware costs are less dominant. It is the labor-intensive job of writing, documenting, maintaining, updating, and distributing software that constitutes the bulk of the expense in automating a process using computers.

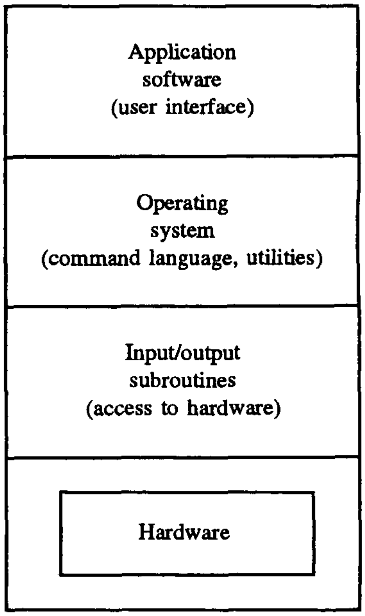
Let's examine the different types of software. Figure 1-5 illustrates three levels of software between the user and the hardware of a computer system: the **application software,** the **operating system,** and the **input/output subroutines.**

At the lowest level, the input/output subroutines directly manipulate the hardware of the system, reading characters from the keyboard, writing characters to the CRT, reading blocks of information from the disk, and so on. Since these subroutines are so intimately linked to the hardware, they are written by the hardware designers and are (usually) stored in ROM. (They are the BIOS—basic input/output system—on the IBM PC, for example.)

To provide close access to the system hardware for programmers, explicit entry and exit conditions are defined for the input/output subroutines. One only needs to initialize values in CPU registers and call the subroutine; the action is carried out with results returned in CPU registers or left in system RAM.

As well as a full complement of input/output subroutines, the ROM contains a start-up program that executes when the system is powered up or reset manually by the operator. The nonvolatile nature of ROM is essential here since this program must exist upon power-up. "Housekeeping" chores, such as checking for options, initializing memory, performing diagnostic checks, etc., are all performed by the start-up program. Last, but not least, a **bootstrap loader** routine reads the first track (a small program) from the disk into RAM and passes control to it. This program then loads the RAM-resident portion of the operating system (a large

FIGURE 1-5
Levels of software



program) from the disk and passes control to it, thus completing the start-up of the system. There is a saying that "the system has pulled itself up by its own bootstraps."

The operating system is a large collection of programs that come with the computer system and provide the mechanism to access, manage, and effectively utilize the computer's resources. These abilities exist through the operating system's **command language** and **utility programs,** which in turn facilitate the development of applications software. If the applications software is well designed, the user interacts with the computer with little or no knowledge of the operating system. Providing an effective, meaningful, and safe user interface is a prime objective in the design of applications software.

1.8 MICROS, MINIS, AND MAINFRAMES

Using their size and power as a starting point, we classify computers as microcomputers, minicomputers, or mainframe computers. A key trait of microcomputers is the size and packaging of the CPU: It is contained within a single integrated circuit—a **microprocessor.** On the other hand, minicomputers and mainframe computers, as well as being more complex in every architectural detail, have CPUs consisting of multiple ICs, ranging from several ICs (minicomputers) to several circuit boards of ICs (mainframes). This increased capacity is necessary to achieve the high speeds and computational power of larger computers.

Typical microcomputers such as the IBM *PC,* Apple *Macintosh,* and Commodore *Amiga* incorporate a microprocessor as their CPU. The RAM, ROM, and interface circuits require many ICs, with the component count often increasing with computing power. Interface circuits vary considerably in complexity, depending on the I/O devices. Driving the loudspeaker contained in most microcomputers, for example, requires only a couple of logic gates. The disk interface, however, usually involves many ICs, some in LSI packages.

Another feature separating micros from minis and mainframes is that microcomputers are single-user, single-task systems-they interact with one user, and they execute one program at a time Minis and mainframes, on the other hand, are multiuser, multitasking systems-they can accommodate many users and programs simultaneously. Actually, the simultaneous execution of programs is an illusion resulting from "time slicing" CPU resources. (Multiprocessing systems, however, use multiple CPUs to execute tasks simultaneously.)

1.9 MICROPROCESSORS VS. MICROCONTROLLERS

It was pointed out above that microprocessors are single-chip CPUs used in microcomputers. How, then, do microcontrollers differ from microprocessors? This question can be addressed from three perspectives: **hardware architecture, applications,** and **instruction set features.**

1.9.1 Hardware Architecture

To highlight the difference between microcontrollers and microprocessors, Figure 1-2 is redrawn in Figure 1-6, showing more detail.

Whereas a microprocessor is a single-chip CPU, a microcontroller contains, in a single IC, a CPU and much of the remaining circuitry of a complete microcomputer system. The components within the dotted line in Figure 1-6 are an integral part of most microcontroller ICs. As well as the CPU, microcontrollers include RAM, ROM, a serial interface, a parallel interface, timer, and interrupt scheduling circuitry—all within the same IC. Of course, the amount of on-chip RAM does not approach that of even a modest microcomputer system; but, as we shall learn, this is not a limitation, since microcontrollers are intended for vastly different applications.

An important feature of microcontrollers is the built-in interrupt system. As control-oriented devices, microcontrollers are often called upon to respond to external stimuli (interrupts) in real time. They must perform fast context switching, suspending one process while executing another in response to an "event." The opening of a microwave oven's door is an example of an event that might cause an interrupt in a microcontroller-based product. Of course, most microprocessors can also implement powerful interrupt schemes, but external components are usually required. A microcontroller's on-chip circuitry includes all the interrupt handling circuitry necessary.

1.9.2 Applications

Microprocessors are most commonly used as the CPU in microcomputer systems. This function is what they are designed for, and this is where their strengths lie. Microcontrollers, however, are found in small, minimum-component designs performing control-oriented activities. These designs were often implemented in the past, using dozens or even hundreds of digital ICs. A microcontroller can aid in reducing the overall component count. All that is required is a microcontroller, a small number of support components, and a control program

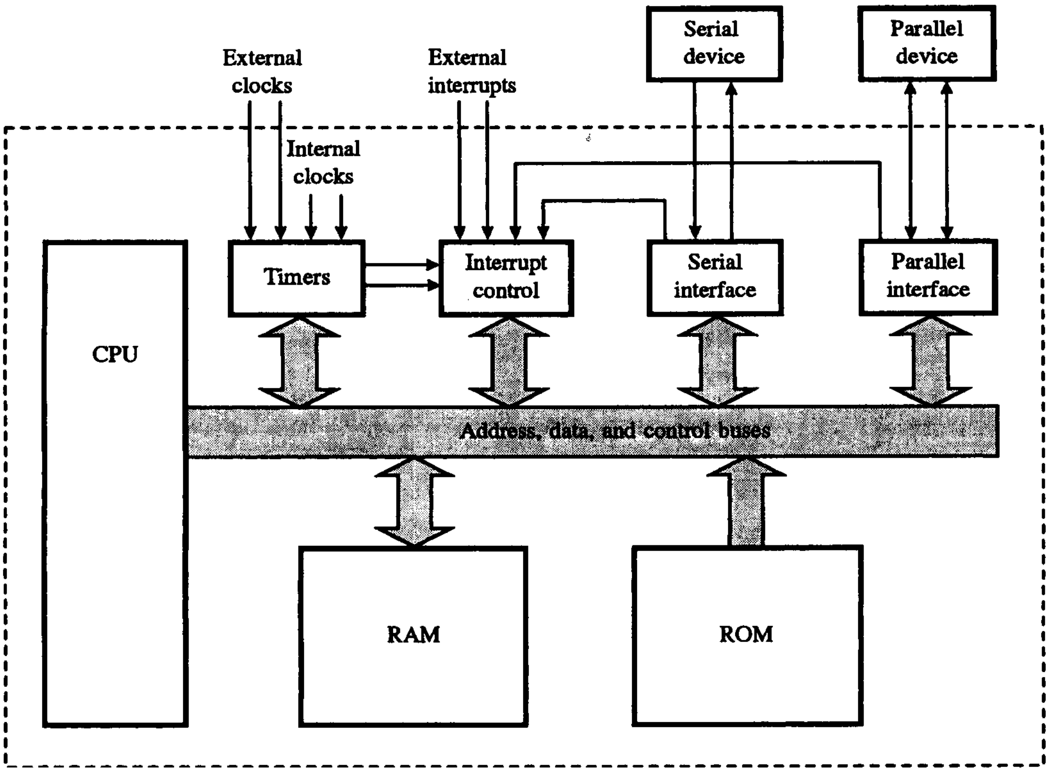


FIGURE 1-6
Detailed block diagram of a microcomputer system

in ROM. Microcontrollers are suited to "control" of I/O devices in designs requiring a minimum component count, whereas microprocessors are suited to "processing" information in computer systems.

1.9.3 Instruction Set Features

Due to the deficiencies in applications, microcontrollers have somewhat different requirements for their instruction sets than microprocessors. Microprocessor instruction sets are "processing intensive," implying they have powerful addressing modes with instructions catering to operations on large volumes of data. Their instructions operate on nibbles, bytes, words, or even double words.[3] Addressing modes provide access to large arrays of data, using address pointers and offsets. Auto-increment and auto-decrement modes simplify stepping through arrays on byte, word, or double-word boundaries. Privileged instructions cannot execute within the user program. The list goes on.

[3] The most common interpretation of these terms is 4 bits = 1 nibble, 8 bits = 1 byte, 16 bits = 1 word, and 32 bits = 1 double word.

Microcontrollers, on the other hand, have instruction sets catering to the control of inputs and outputs. The interface to many inputs and outputs uses a single bit. For example, a motor may be turned on and off by a solenoid energized by a 1-bit output port. Micro-controllers have instructions to set and clear individual bits and perform other bit-oriented operations such as logically ANDing, ORing, or EXORing bits, jumping if a bit is set or clear, and so on. This powerful feature is rarely present in microprocessors, which are usually designed to operate on bytes or larger units of data.

In the control and monitoring of devices (perhaps with a 1-bit interface), microcontrollers have built-in circuitry and instructions for input/output operations, event timing, and enabling and setting priority levels for interrupts caused by external stimuli. Microprocessors often require additional circuitry (serial interface ICs, interrupt controllers, timers, etc.) to perform similar operations. Nevertheless, the sheer processing capability of a microcontroller never approaches that of a microprocessor (all else being equal), since a great deal of the IC's "real estate" is consumed by the on-chip functions—at the expense of processing power, of course.

Since the on-chip real estate is at a premium in microcontrollers, the instructions must be extremely compact, with the majority implemented in a single byte. A design criterion is often that the control program must fit into the on-chip ROM, since the addition of even one external ROM adds too much cost to the final product. A tight encoding scheme for the instruction set is essential. This is rarely a feature of microprocessors; their powerful addressing modes bring with them a less-than-compact encoding of instructions.

1.10 NEW CONCEPTS

Microcontrollers, like other products considered in retrospect to have been a breakthrough, have arrived out of two complementary forces: market need and new technology. The new technology is just that mentioned above: semiconductors with more transistors in less space, mass produced at a lower cost. The market need is the industrial and consumer appetite for more sophisticated tools and toys.[4] This demand encompasses a lot of territory. The most illustrative example, perhaps, is the automobile dashboard. Witness the transformation of the car's "control center" over the past decade—made possible by the microcontroller and other technological developments. Once, drivers were content to know their speed; today they may find a display of fuel economy and estimated time of arrival. Once it was sufficient to know if a seatbelt was unfastened while starting the car; today, we are "told" which seatbelt is the culprit. If a door is ajar, we are again duly informed by the spoken word. (Perhaps the seatbelt is stuck in the door.)

This brings to mind a necessary comment. Microprocessors (and in this sense microcontrollers) have been dubbed "solutions looking for a problem." It seems they have proved so effective at reducing the complexity of circuitry in (consumer) products, that manufacturers are often too eager to include superfluous features simply because they are easy to

[4] 1t is sometimes argued that "market need" is really "market want," spurred on by the self-propelled growth of technology.

design into the product. The result often lacks eloquence—a showstopper initially, but an annoyance finally. The most stark example of this bells-and-whistles approach occurs in the recent appearance of products that talk. Whether automobiles, toys, or toasters, they are usually examples of tackiness and overdesign-1980s art deco, perhaps. Rest assured that once the dust has settled and the novelty has diminished, only the subtle and appropriate will remain.

Microcontrollers are specialized. They are not used in computers per se, but in industrial and consumer products. Users of such products are quite often unaware of the existence of microcontrollers: to them, the internal components are but an inconsequential detail of design. Consider as examples microwave ovens, programmable thermostats, electronic scales, and even cars. The electronics within each of these products typically incorporates a microcontroller interfacing to push buttons, switches, lights, and alarms on a front panel; yet user operation mimics that of the electromechanical predecessors, with the exception of some added features. The microcontroller is invisible to the user.

Unlike computer systems, which are defined by their ability to be programmed and then reprogrammed, microcontrollers are permanently programmed for one task. This comparison results in a stark architectural difference between the two. Computer systems have a high RAM-to-ROM ratio, with user programs executing in a relatively large RAM space and hardware interfacing routines executing in a small ROM space. Microcontrollers, on the other hand, have a high ROM-to-RAM ratio. The control program, perhaps relatively large, is stored in ROM, while RAM is used only for temporary storage. Since the control program is stored permanently in ROM, it has been dubbed **firmware. In** degrees of "firmness," it lies somewhere between software—the programs in RAM that are lost when power is removed—and hardware—the physical circuits. The difference between software and hardware is somewhat analogous to the difference between a page of paper (hardware) and words written on a page (software). Consider firmware as a standard form letter, designed and printed for a single purpose.

1.11 GAINS AND LOSSES: A DESIGN EXAMPLE

The tasks performed by microcontrollers are not new. What is new is that designs are implemented with fewer components than before. Designs previously requiring tens or even hundreds of ICs are implemented today with only a handful of components, including a microcontroller. The reduced component count, a direct result of the microcontroller's programmability and high degree of integration, usually translates into shorter development time, lower manufacturing cost, lower power consumption, and higher reliability. Logic operations that require several ICs can often be implemented within the microcontroller, with the addition of a control program.

One tradeoff is speed. Microcontroller-based solutions are never as fast as the discrete counterparts. Situations requiring extremely fast response to events (a minority of applications) are poorly handled by microcontrollers. For example, consider in Figure 1-7 the somewhat trivial implementation of the NAND operation using an 8051 microcontroller.

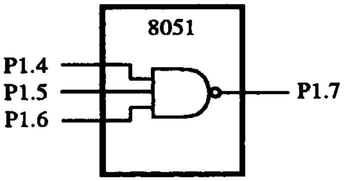


FIGURE 1-7
Microcontroller implementation of a simple logic operation

It is not at all obvious that a microcontroller could be used for such an operation, but it can. The software must perform the operations shown in the flowchart in Figure 1-8. The 8051 assembly language program for this logic operation is shown below.

```
LOOP:  MOV  C,P1.4      ;READ P1.4 BIT INTO CARRY FLAG
       ANL  C,P1.5      ;AND WITH P1.5
       ANL  C,P1.6      ;AND WITH P1.6
       CPL  C           ;CONVERT TO "NAND" RESULT
       MOV  P1.7,C      ;SEND TO P1.7 OUTPUT BIT
       SJMP LOOP        ;REPEAT
```

If this program executes on an 8051 microcontroller, indeed the 3-input NAND function is realized. (It could be verified with a voltmeter or oscilloscope.) The propagation delay from an input transition to the correct output level is quite long, at least in comparison to the equivalent TTL (transistor-transistor logic) circuit. Depending on when the input changed relative to the program sensing the change, the delay is from 3 to 17 microseconds. (This assumes standard 8051 operation using a 12 MHz crystal.) The equivalent TTL propagation delay is on the order of 10 nanoseconds—about three orders of magnitude less. Obviously, there is no contest when comparing the speed of microcontrollers with TTL implementations of the same function.

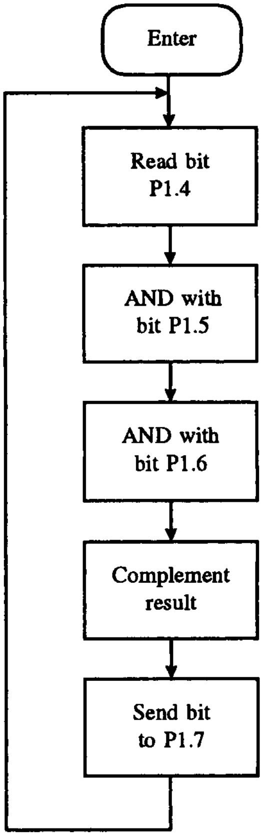
In many applications, particularly those with human operation, whether the delays are measured in nanoseconds, microseconds, or milliseconds is inconsequential. (When the oil pressure drops in your car, do you need to be informed within microseconds?) The logic gate example illustrates that microcontrollers can implement logic operations. Furthermore, as designs become complex, the advantages of the microcontroller-based design begin to take hold. The reduced component count has advantages, as mentioned earlier; but, also, the operations in the control program make it possible to introduce changes in design by modifying only the software. This modification has minimal impact on the manufacturing cycle.

This concludes our introduction to microcontrollers. In the next chapter, we begin our examination of the MCS-51^{TM} family of devices.

PROBLEMS

1.1 What was the first widely used microprocessor? In what year was it introduced and by what company?

FIGURE 1-8
Flowchart for logic gate program



1.2 Two of the smaller microprocessor companies in the 1970s were MOS Technology and Zilog. Name the microprocessor that each of these companies introduced.
1.3 What year was the 8051 microcontroller introduced? What was the predecessor to the 8051, and in what year was it introduced?
1.4 Name the two types of semiconductor memory discussed in this chapter. Which type retains its contents when powered-off? What is the common term that describes this property?
1.5 Which register in a CPU always contains an address? What address is contained in this register?
1.6 During an opcode fetch, what is the information on the address and data buses? What is the direction of information flow on these buses during an opcode fetch?
1.7 How many bytes of data can be addressed by a computer system with an 18-bit address bus and an 8-bit data bus?
1.8 What is the usual meaning of "16-bit" in the phrase "16-bit computer"?
1.9 What is the difference between online storage and archival storage?
1.10 What type of technology is used for archival storage besides magnetic tape and disk?

1.11 With regard to computing systems, what is the goal of the field of engineering known as "human factors"?

1.12 Consider the following human interface devices: a joystick, a light pen, a mouse, a microphone, and a loudspeaker. Which are input devices? Which are output devices?

1.13 Of the three levels of software presented in this chapter, which is the lowest level? What is the purpose of this level of software?

1.14 What is the difference between an actuator and a sensor? Give an example of each.

1.15 What is firmware? Comparing a microcontroller-based system to a microprocessor-based system, which is more likely to rely on firmware? Why?

1.16 What is an important feature of a microcontroller's instruction set that distinguishes it from a microprocessor?

1.17 Name five products not mentioned in this chapter that are likely to use a microcontroller.

CHAPTER 2

Hardware Summary

2.1 MCS-51™ FAMILY OVERVIEW

The MCS-51™ is a family of microcontroller ICs developed, manufactured, and marketed by Intel Corporation. Other IC manufacturers, such as Siemens, Advanced Micro Devices, Fujitsu, and Philips are licensed "second source" suppliers of devices in the MCS-51™ family. Each microcontroller in the family boasts a complement of features suited to a particular design setting.

In this chapter the hardware architecture of the MCS-51™ family is introduced. Intel's data sheet for the entry-level devices (e.g., the 8051AH) is found in Appendix E. This appendix should be consulted for further details, for example, on electrical properties of these devices.

Many of the hardware features are illustrated with short sequences of instructions. Brief descriptions are provided with each example, but complete details of the instruction set are deferred to Chapter 3. See also Appendix A for a summary of the 8051 instruction set or Appendix C for definitions of each 8051 instruction.

The generic MCS-51™ IC is the 8051, the first device in the family offered commercially. Its features are summarized below.

- 4K bytes ROM (factory mask programmed)
- 128 bytes RAM
- Four 8-bit I/O (Input/Output) ports
- Two 16-bit timers
- Serial interface
- 64K external code memory space
- 64K external data memory space
- Boolean processor (operates on single bits)
- 210 bit-addressable locations
- 4 µs multiply/divide

TABLE 2-1
Comparison of MCS-51TM ICs

Part Number	On-Chip Code Memory	On-Chip Data Memory	Timers
8051	4 K ROM	128 bytes	2
8031	0 K	128 bytes	2
8751	4 K EPROM	128 bytes	2
8052	8 K ROM	256 bytes	3
8032	0 K	256 bytes	3
8752	8 K EPROM	256 bytes	3

Other members of the MCS-51^{TM} family offer different combinations of on-chip ROM or EPROM, on-chip RAM, or a third timer. Each of the MCS-51^{TM} ICs is also offered in a low-power CMOS version (see Table 2-1).

The term "8051" loosely refers to the MCS-51^{TM} family of microcontrollers. When discussion centers on an enhancement to the basic 8051 device, the specific part number is used. The features mentioned above are shown in the block diagram in Figure 2-1. (See also Appendix D.)

2.2 ONCE AROUND THE PINS

This section introduces the 8051 hardware architecture from an external perspective—the pinouts (see Figure 2-2). A brief description of the function of each pin follows.

As evident in Figure 2-2, 32 of the 8051's 40 pins function as I/O port lines. However, 24 of these lines are dual-purpose (26 on the 8032/8052). Each can operate as I/O, or as a control line or part of the address or data bus.

Designs requiring a minimum of external memory or other external components use these ports for general purpose I/O. The eight lines in each port can be treated as a unit in interfacing to parallel devices such as printers, digital-to-analog converters, and so on. Or, each line can operate independently in interfacing to single-bit devices such as switches, LEDs, transistors, solenoids, motors, and loudspeakers.

2.2.1 Port 0

Port 0 is a dual-purpose port on pins 32-39 of the 8051 IC. In minimum-component designs, it is used as a general purpose I/O port. For larger designs with external memory, it becomes a multiplexed address and data bus. (See 2.7 External Memory.)

2.2.2 Port 1

Port 1 is a dedicated I/O port on pins 1-8. The pins, designated as P1.0, P1.1, P1.2, etc., are available for interfacing to external devices as required. No alternate functions are as signed

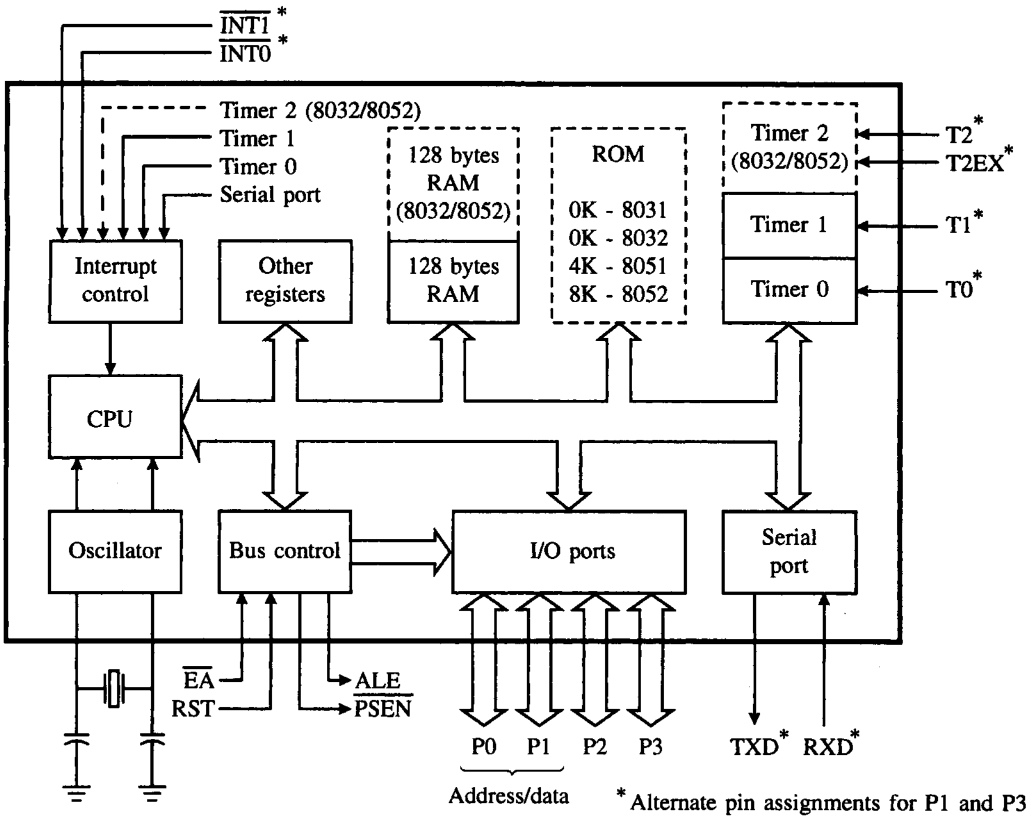


FIGURE 2–1
8051 block diagram

for Port 1 pins; thus, they are used solely for interfacing to external devices. Exceptions are the 8032/8052 ICs, which use P1.0 and P1.1 either as I/O lines or as external inputs to the third timer.

2.2.3 Port 2

Port 2 (pins 21-28) is a dual-purpose port serving as general purpose I/O, or as the high-byte of the address bus for designs with external code memory or more than 256 bytes of external data memory. (See 2.7 External Memory.)

2.2.4 Port 3

Port 3 is a dual-purpose port on pins 10-17. As well as general-purpose i/o, these pins are multifunctional, with each having an alternate purpose related to special features of the 8051. The alternate purpose of the Port 3 and Port 1 pins is summarized in Table 2-2.

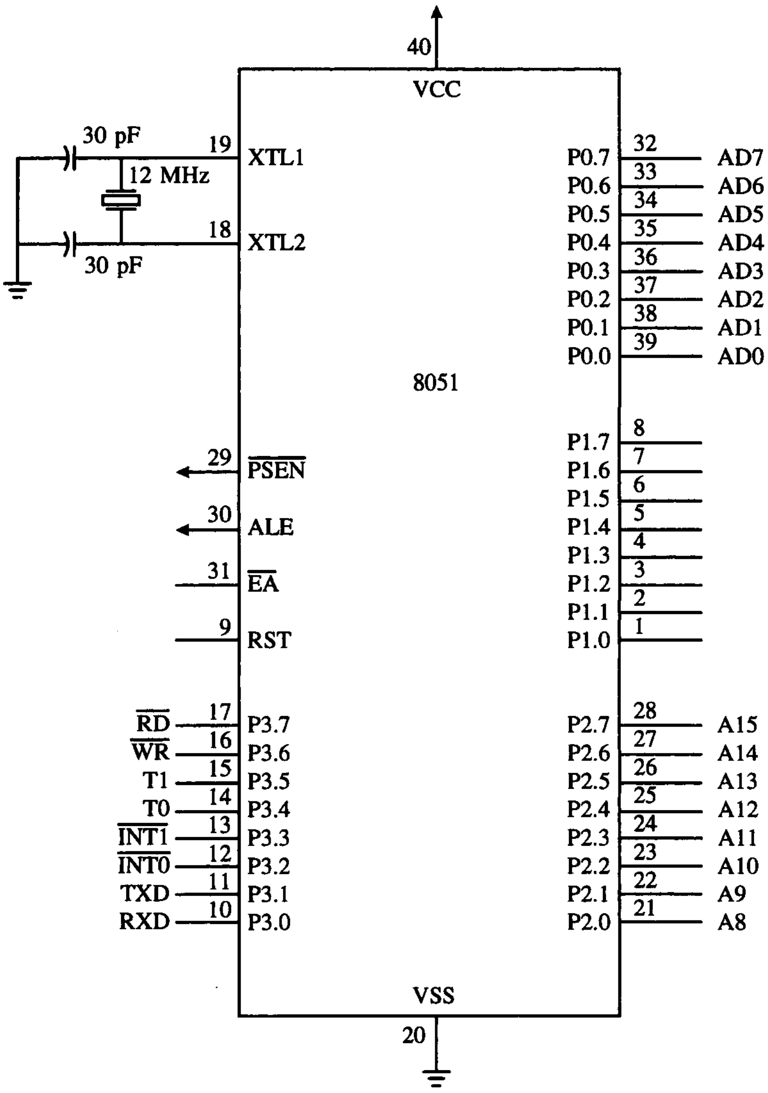


FIGURE 2-2
8051 pinouts

2.2.5 PSEN (Program Store Enable)

The 8051 has four dedicated bus control signals. Program Store Enable ($\overline{\text{PSEN}}$) is an output signal on pin 29. It is a control signal that enables external program (code) memory. It usually connects to an EPROM's (Erasable Programmable Read-Only Memory) Output Enable ($\overline{\text{OE}}$) pin to permit reading of program bytes.

TABLE 2-2
Alternate pin functions for port pins

Bit	Name	Bit Address	Alternate Function
P3.0	RXD	B0H	Receive data for serial port
P3.1	TXD	B1H	Transmit data for serial port
P3.2	$\overline{\text{INT0}}$	B2H	External interrupt 0
P3.3	$\overline{\text{INT1}}$	B3H	External interrupt 1
P3.4	T0	B4H	Timer/counter 0 external input
P3.5	T1	B5H	Timer/counter 1 external input
P3.6	$\overline{\text{WR}}$	B6H	External data memory write strobe
P3.7	$\overline{\text{RD}}$	B7H	External data memory read strobe
P1.0	T2	90H	Timer/counter 2 external input
P1.1	T2EX	91H	Timer/counter 2 capture/reload

The ($\overline{\text{PSEN}}$) signal pulses low during the fetch stage of an instruction, which is stored in external program memory. The binary codes of a program (opcodes) are read from EPROM, travel across the data bus, and are latched into the 8051's instruction register for decoding. When executing a program from internal ROM (8051/8052), ($\overline{\text{PSEN}}$) remains in the inactive (high) state.

2.2.6 ALE (Address Latch Enable)

The ALE output signal on pin 30 will be familiar to anyone who has worked with Intel's 8085, 8088, or 8086 microprocessors. The 8051 similarly uses ALE for demultiplexing the address and data bus. When Port 0 is used in its alternate mode—as the data bus and the low-byte of the address busALE is the signal that latches the address into an external register during the first half of a memory cycle. This done, the Port 0 lines are then available for data input or output during the second half of the memory cycle, when the data transfer takes place. (See 2.7 External Memory.)

The ALE signal pulses at 1/6th the on-chip oscillator frequency and can be used as a general-purpose clock for the rest of the system. If the 8051 is clocked from a 12 MHz crystal, the ALE signal oscillates at 2 MHz. The only exception is during the MOVX instruction, when one ALE pulse is missed. (See Figure 2-11.) This pin is also used for the programming input pulse for EPROM versions of the 8051.

2.2.7 EA (External Access)

The $\overline{\text{EA}}$ input signal on pin 31 is generally tied high (+5 V) or low (ground). If high, the 8051/8052 executes programs from internal ROM when executing in the lower 4K/8K of memory. If low, programs execute from external memory only (and $\overline{\text{PSEN}}$ pulses low accordingly). $\overline{\text{EA}}$ must be tied low for 8031/8032 ICs, since there is no on-chip program memory. If $\overline{\text{EA}}$ is tied low on an 8051/8052, internal ROM is disabled and programs execute from external EPROM. The EPROM versions of the 8051 also use the EA line for the +21 volt supply (V_{pp})for programming the internal EPROM.

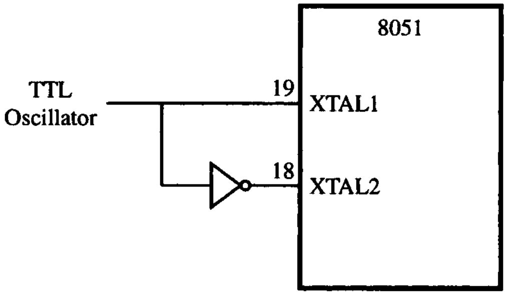


FIGURE 2-3
Driving the 8051 from a TTL oscillator

2.2.8 RST (Reset)

The RST input on pin 9 is the master reset for the 8051. When this signal is brought high for at least two machine cycles, the 8051 internal registers are loaded with appropriate values for an orderly system start-up. For normal operation, RST is low. (See 2.9 Reset Operation.)

2.2.9 On-Chip Oscillator Inputs

As shown in Figure 2-2, the 8051 features an on-chip oscillator that is typically driven by a crystal connected to pins 18 and 19. Stabilizing capacitors are also required as shown.

The nominal crystal frequency is 12 MHz for most ICs in the MCS-51^{TM} family, although the 80C31BH-1 can operate with crystal frequencies up to 16 MHz. The on-chip oscillator needn't be driven by a crystal. As shown in Figure 2-3, a TTL clock source can be connected to XTAL1 and XTAL2.

2.2.10 Power Connections

The 8051 operates from a single +5 volt supply. The V_{cc} connection is on pin 40, and the V_{ss} (ground) connection is on pin 20.

2.3 I/O PORT STRUCTURE

The internal circuitry for the port pins is shown in abbreviated form in Figure 2-4. Writing to a port pin loads data into a port latch that drives a field-effect transistor connected to the port pin. The drive capability is four low-power Schottky TTL loads for Ports 1, 2, and 3; and eight LS loads for Port 0. (See Appendix E for more details.) Note that the pull-up resistor is absent on Port 0 (except when functioning as the external address/data bus). An external pull-up resistor may be needed, depending on the input characteristics of the device driven by the port pin.

There is both a "read latch" and "read pin" capability. Instructions that require a read-modify-write operation (e.g., CPL P1.5) read the latch to avoid misinterpreting the voltage level in the event the pin is heavily loaded (e.g., when driving the base of a transistor). Instructions that input a port bit (e.g., MOV C,P1.5) read the pin. The port latch must contain

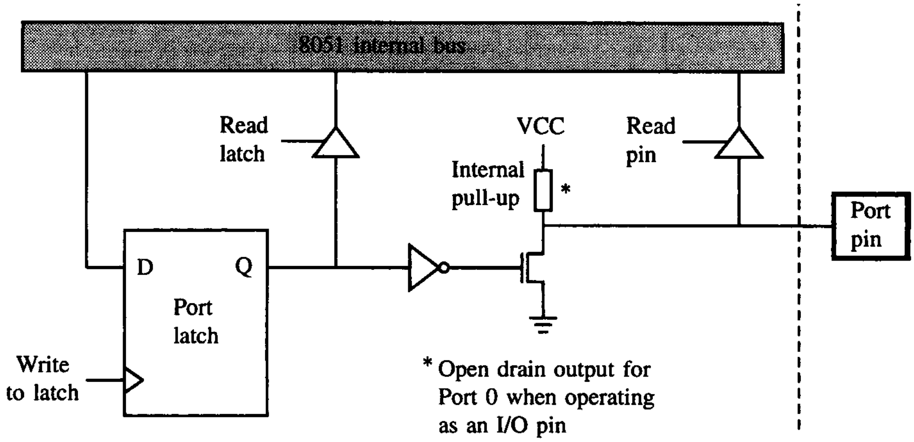


FIGURE 2-4
Circuitry for I/O ports

a 1, in this case, otherwise the FET driver is ON and pulls tile output low. A system reset sets all port latches, so port pins may be used as inputs without explicitly setting tile port latches. If, however, a port latch is cleared (e.g., CLR P1.5), then it cannot function subsequently as an input unless the latch is set first (e.g., SETB P1.5).

Figure 2-4 does not show the circuitry for the alternate functions for Ports 0, 2, and 3. When the alternate function is in effect, the output drivers are switched to an internal address (Port 2), address/data (Port 0), or control (Port 3) signal, as appropriate.

2.4 TIMING AND THE MACHINE CYCLE

The 8051's on-chip oscillator is driven by an external quartz crystal through pins 18 and 19. This crystal has a typical frequency of 12 MHz, meaning that it generates 12 million clock cycles per second. These oscillator clock cycles form the basis of the 8051's timing and synchronization: Every operation performed by the 8051 is in step with these cycles.

With the oscillator clock as reference, the 8051 requires two such clock cycles to perform a single discrete operation, which is either fetching an instruction, decoding, or executing it. This duration of two clock cycles is also called a **state.** Therefore, in order to fully process an instruction, the 8051 would generally require six such states, or 12 clock cycles since it would have to first fetch and decode the instruction before it goes to execute it. This duration of six states is also known as one **machine cycle.** Of course, more complex instructions would take more than one machine cycle to be carried out. Both Appendix B and C provide a list of the number of machine cycles for all the 8051 instructions. This number ranges from one to four machine cycles. Figure 2-5 shows the relationship between oscillator clock cycles (P), states (S), and a machine cycle.

Typically, the 8051's on-chip oscillator, f_{osc} is driven by a 12 MHz crystal, so the period of one clock pulse, $T_{clock} = 1/f_{osc} = 1/12$ MHz = 83.33 ns. One machine cycle consists of 12 such clock pulses, hence its duration is 83.33 ns x 12 = 1 µs.

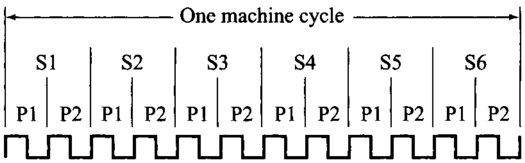


FIGURE 2-5
Relationship between oscillator clock cycles, states, and the machine cycle

2.5 MEMORY ORGANIZATION

Most microprocessors implement a shared memory space for data and programs. This is reasonable, since programs are usually stored on a disk and loaded into RAM for execution; thus both the data and programs reside in the system RAM. Microcontrollers, on the other hand, are rarely used as the CPU in "computer systems." Instead, they are employed as the central component in control-oriented designs. There is limited memory, and there is no disk drive or disk operating system. The control program must reside in ROM.

For this reason, the 8051 implements a separate memory space for programs (code) and data. As shown in Table 2-1, both the code and data may be internal; however, both expand using external components to a maximum of 64K code memory and 64K data memory.

The internal memory consists of on-chip ROM (8051/8052 only) and on-chip data RAM. The on-chip RAM contains a rich arrangement of general-purpose storage, bit-addressable storage, register banks, and special function registers.

Two notable features are: (a) the registers and input/output ports are memory mapped and accessible like any other memory location, and (b) the stack resides within the internal RAM, rather than in external RAM as typical of microprocessors.

Figure 2-6 summarizes the memory spaces for the ROM-less 8031 device without showing any detail of the on-chip data memory. (8032/8052 enhancements are summarized later.)

Figure 2-7 gives the details of the on-chip data memory. As shown, the internal data memory space is divided between **internal RAM** (00H-7FH) and **special function registers** (80H-0FFH). A confusion sometimes arises between the concept of internal (on-chip) data memory and internal RAM. The 8051's internal data memory space has the range from 00H-0FFH, which is 256 bytes. However, only the lower half (00H-7FH) of the internal memory space is for general data while the upper half (80H-0FFH) is mostly for specific purposes and not for general data; hence, only the lower half is considered to be internal RAM. The internal data RAM is further sub-divided into register banks (00H-1FH), bit-addressable RAM (20H-2FH), and general-purpose RAM (30H-7FH). Each of these sections of internal memory is discussed below.

2.5.1 General-Purpose RAM

Although Figure 2-7 shows 80 bytes of general-purpose RAM from addresses 30H to 7FH, the bottom 48 bytes from 00H to 2FH can be used similarly (although these locations have other purposes as discussed below).

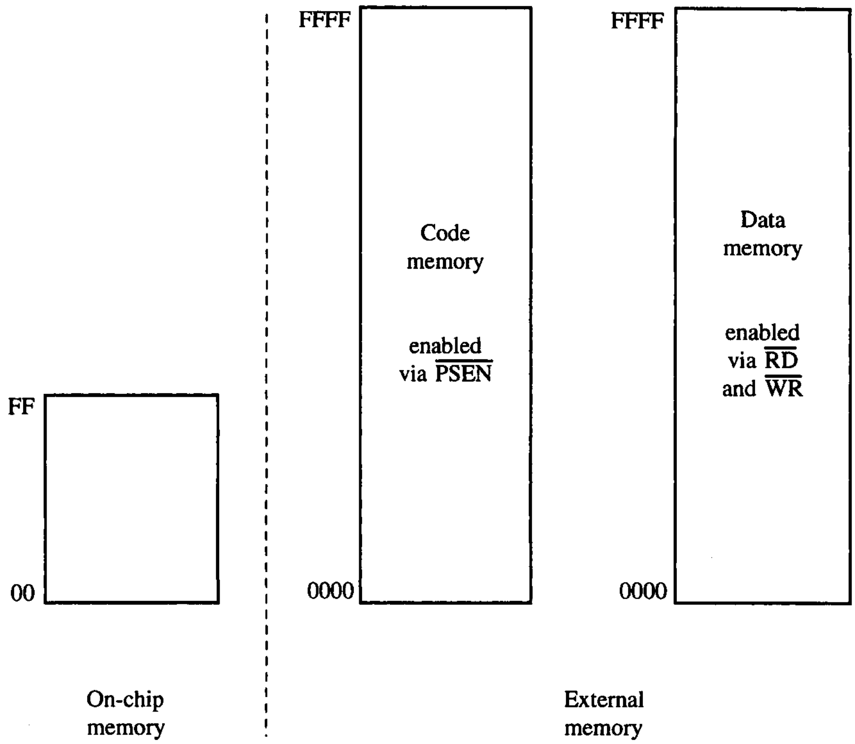

FF
00
On-chip memory
FFFF
0000
Code memory
enabled via $\overline{\text{PSEN}}$
FFFF
0000
Data memory
enabled via $\overline{\text{RD}}$ and $\overline{\text{WR}}$
External memory

FIGURE 2-6
Summary of the 8031 memory spaces

Any location in the general-purpose RAM can be accessed freely using the direct or indirect addressing modes. For example, to read the contents of internal RAM address 5FH into the accumulator, the following instruction could be used:

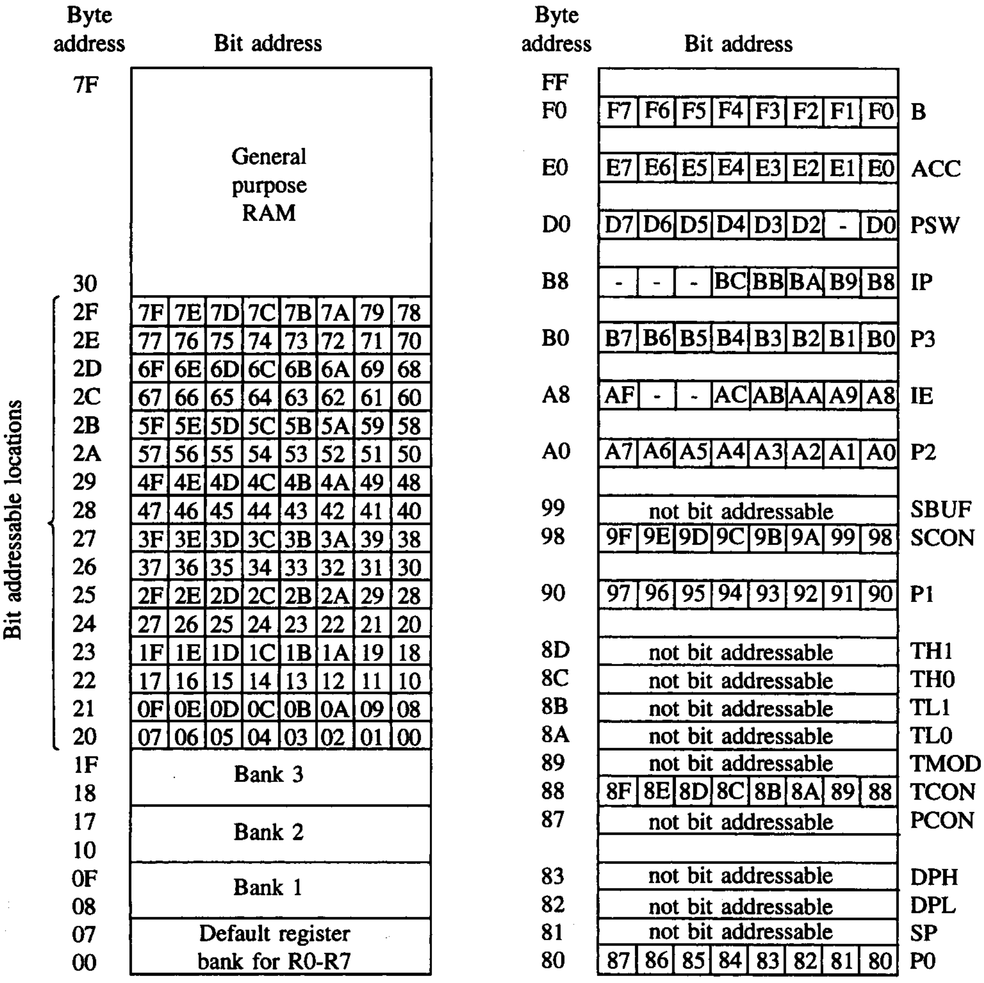
```
MOV A,5FH
```

This instruction moves a byte of data using direct addressing to specify the "source location" (i.e., address 5FH). The destination for the data is implicitly specified in the instruction opcode as the A accumulator. (Note: Addressing modes are discussed in detail in Chapter 3.)

Internal RAM can also be accessed using indirect addressing through R0 or R1. For example, the following two instructions perform the same operation as the single instruction above:

```
MOV R0,#5FH
MOV A,@R0
```

The first instruction uses immediate addressing to move the value 5FH into register R0, and the second instruction uses indirect addressing to move the data "pointed at by R0" into the accumulator.



Byte address	Bit address							
7F	General purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default register bank for R0-R7							
00								

RAM

Byte address	Bit address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	not bit addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	P1
8D	not bit addressable								TH1
8C	not bit addressable								TH0
8B	not bit addressable								TL1
8A	not bit addressable								TL0
89	not bit addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	not bit addressable								PCON
83	not bit addressable								DPH
82	not bit addressable								DPL
81	not bit addressable								SP
80	87	86	85	84	83	82	81	80	P0

SPECIAL FUNCTION REGISTERS

FIGURE 2-7
Summary of the 8051 on-chip data memory

2.5.2 Bit-Addressable RAM

The 8051 contains 210 bit-addressable locations, of which 128 are at byte addresses 20H through 2FH, and the rest are in the special function registers (discussed below).

The idea of individually accessing bits through software is a powerful feature of most microcontrollers. Bits can be set, cleared, ANDed, ORed, etc., with a single instruction. Most microprocessors require a read-modify-write sequence of instructions to achieve the same effect. Furthermore, the 8051 I/O ports are bit-addressable, simplifying the software interface to single-bit inputs and outputs.

There are 128 general-purpose bit-addressable locations at byte addresses 20H through 2FH (8 bits/byte x 16 bytes = 128 bits). These addresses are accessed as bytes or as bits, depending on the instruction. For example, to set bit 67H, the following instruction could be used:

```
SETB 67H
```

Referring to Figure 2-7, note that "bit address 67H" is the most-significant bit at "byte address 2CH." The instruction above has no effect on the other bits at this address. Most microprocessors would perform the same operation as follows:

```
MOV A, 2CH          ;READ ENTIRE BYTE
ORL A,#10000000B    ;SET MOST-SIGNIFICANT BIT
MOV 2CH,A           ;WRITE BACK ENTIRE BYTE
```

EXAMPLE 2.1 What instruction would be used to set bit 3 in byte address 25H?

Solution

```
SETB 2BH
```

Discussion

Byte address 25H is within the bit-addressable area of internal memory (see Figure 2-7). The bit addresses within this byte, starting at bit 0, are 28H, 29H, etc. Bit 3 within byte address 25H is at bit address 2BH.

2.5.3 Register Banks

The bottom 32 locations of internal memory contain the register banks. The 8051 instruction set supports eight registers, R0 through R7, and by default (after a system reset) these registers are at addresses 00H-07H. The following instruction, then, reads the contents of address 05H into the accumulator:

```
MOV A,R5
```

This instruction is a 1-byte instruction using register addressing. Of course, the same operation could be performed in a 2-byte instruction, using the direct address as byte 2:

```
MOV A,05H
```

Instructions using registers R0 to R7 are shorter than the equivalent instructions using direct addressing. Data values used frequently should use one of these registers.

The active register bank may be altered by changing the register bank select bits in the program status word (discussed below). Assuming, then, that register bank 3 is active, the following instruction writes the contents of the accumulator into location 18H:

```
MOV R0,A
```

The idea of "register banks" permits fast and effective "context switching," whereby separate sections of software use a private set of registers independent of other sections of software.

EXAMPLE 2.2 What is the address of register 5 in register bank 3?

Solution

```
1DH
```

Discussion

Register bank 3 occupies internal memory locations 18H to 1FH (see Figure 2-7), with R0 at address 18H, R1 at address 19H, etc. Register 5 (R5) is at address 1DH.

2.6 SPECIAL FUNCTION REGISTERS

Internal registers on most microprocessors are accessed implicitly by the instruction set. For example, "INCA" on the 6809 microprocessor increments the contents of the A accumulator. The operation is specified implicitly within the instruction opcode. Similar access to registers is also used on the 8051 microcontroller. In fact, the 8051 instruction "INC A" performs the same operation.

The 8051 internal registers are configured as part of the on-chip RAM: therefore, each register also has an address.[1] This is reasonable for the 8051, since it has so many registers. As well as R0 to R7, there are 21 special function registers (SFRs) at the top of internal RAM, from addresses 80H to 0FFH. (See Figure 2-7 and Appendix D.) Note that most of the 128 addresses from 80H to 0FFH are not defined. Only 21 SFR addresses are defined (26 on the 8032/8052).

Although the accumulator (A or ACC) may be accessed implicitly as shown previously, most SFRs are accessed using direct addressing. Note in Figure 2-7 that some SFRs are both bit-addressable and byte-addressable. Programmers should be careful when accessing bits versus bytes. For example, the instruction

```
SETB 0E0H
```

sets bit 0 in the accumulator, leaving the other bits unchanged. The trick is to recognize that 0E0H is both the byte address of the entire accumulator and the bit address of the least-significant bit in the accumulator. Since the SETB instruction operates on bits (not bytes), only the addressed bit is affected. Notice that the addressable bits within the SFRs have the five high-order address bits matching those of the SFR. For example, Port 1 is at byte address 90H or 10010000B. The bits within Port 1 have addresses 90H to 97H, or 10010xxxB.

[1] The program counter and the instruction register are exceptions. Since these registers are rarely manipulated directly, nothing is gained by placing them in the on-chip RAM.

EXAMPLE 2.3 What instruction could be used to set the most-significant bit in the B accumulator while leaving the other bits intact?

Solution

```
SETB 0F7H
```

Discussion

The B accumulator is at byte address 0F0H in the special function register space of internal memory (see Figure 2-7). Individual bits are accessible, with bit 0 at address 0F0H, bit 1 at address 0F1H, etc. Bit 7 of the B accumulator is at bit address 0F7H.

The PSW is discussed in detail in the following section. The other SFRs are briefly introduced following the PSW, with detailed discussions deferred to later chapters.

2.6.1 Program Status Word

The program status word (PSW) at address 0D0H contains status bits as summarized in Table 2-3. Each of the PSW bits is examined below.

2.6.1.1 Carry Flag The carry flag (C or CY) is dual-purpose. It is used in the traditional way for arithmetic operations: set if there is a carry out of bit 7 during an add, or set if there is a borrow into bit 7 during a subtract. For example, if the accumulator contains 0FFH, then the instruction

```
ADD A,#1
```

leaves the accumulator equal to 00H and sets the carry flag in the PSW.

TABLE 2–3
PSW (program status word) register summary

Bit	Symbol	Address	Bit Description
PSW.7	CY	D7H	Carry flag
PSW.6	AC	D6H	Auxillary carry flag
PSW.5	F0	D5H	Flag 0
PSW.4	RS1	D4H	Register bank select 1
PSW.3	RS0	D3H	Register bank select 0 00 = bank 0;addresses 00H–07H 01 = bank 1; addresses 08H–0FH 10 = bank 2; addresses 10H–17H 11 = bank 3; addresses 18H–1FH
PSW.2	OV	D2H	Overflow flag
PSW.1	—	D1H	Reserved
PSW.0	P	D0H	Even parity flag

EXAMPLE 2.4 What is the state of the carry flag and the content of the accumulator after execution of the following instruction sequence?

```
MOV R5,#55H
MOV A,#0AAH
ADD A,R5
```

Solution

C = 0, ACC = 0FFH

Discussion

The binary addition that occurs in the third instruction is illustrated below.

```
 01010101 (R5 = 55H)
+10101010 (ACC = 0AAH)
 11111111 (Result in ACC = 0FFH)
```

The addition does not generate a carry of the most-significant bit (bit 7); therefore, the carry bit is cleared. The final result in the accumulator is 0FFH = 255_{10}.

The carry flag is also the "Boolean accumulator," serving as a 1-bit register for Boolean instructions operating on bits. For example, the following instruction ANDs bit 25H with the carry flag and places the result back in the carry flag:

```
ANL C,25H
```

2.6.1.2 Auxiliary Carry Flag When adding binary-coded-decimal (BCD) values, the auxiliary carry flag (AC) is set if a carry was generated out of bit 3 into bit 4 or if the result in the lower nibble is in the range 0AH-0FH. If the values added are BCD, then the add instruction must be followed by DA A (decimal adjust accumulator) to bring results greater than 9 back into range.

EXAMPLE 2.5 What is the state of the auxiliary carry flag and the content of the accumulator after execution of the instruction sequence below?

```
MOV R5,#1
MOV A,#9
ADD A,R5
```

Solution

AC = 1, ACC = 0AH

Discussion

The binary addition that takes place in the third instruction is illustrated below.

```
      1
 00000001       (R5 = 01H)
+00001001       (ACC = 09H)
 00001010       (Result in ACC = 0AH)
```

Although no carry occurred during the binary addition, the low-order nibble of the result is 1010B = 0AH. Since this is greater than 9_{10}, the auxiliary carry bit is set. If the addition instruction is followed by a decimal adjust instruction (DA A), the final result in the accumulator is 0001000B 10H. As a binary-coded decimal number, 10H = 10_{10} which is the correct result of $9_{10} + 1_{10}$.

2.6.1.3 Flag 0 Flag 0 (F0) is a general-purpose flag bit available for user applications.

2.6.1.4 Register Bank Select Bits The register bank select bits (RS0 and RS1) determine the active register bank. They are cleared after a system reset and are changed by software as needed. For example, the following three instructions enable register bank 3 and then move the content of R7 (byte address 1FH) to the accumulator:

```
SETB   RS1
SETB   RS0
MOV    A,R7
```

When the above program is assembled, the correct bit addresses are substituted for the symbols "RS1" and "RS0." Thus, the instruction SETB RS1 is the same as SETB 0D4H.

EXAMPLE 2.6 Illustrate an instruction sequence to make register bank 2 the active register bank. Assume that the previously active register bank is unknown.

Solution

```
SETB RS1
CLR  RS0
```

Discussion

The result of the above instructions is to place $10_2 = 2_{10}$ in the register bank select bits in the program status word. The active register bank becomes bank 2. If it was known that the register bank bits had not changed since the last CPU reset operation, the second instruction would not be necessary; however, since an unknown state was assumed, both register select bits are explicitly initialized.

2.6.1.5 Overflow Flag The overflow flag (OV) is set after an addition or subtraction operation if there was an arithmetic overflow. When signed numbers are added or subtracted, software can examine this bit to determine if the result is in the proper range. When unsigned numbers are added, the OV bit can be ignored. Results greater than +127 or less than -128 will set the OV bit. For example, the following addition causes an overflow and sets the OV bit in the PSW:

```
Hex:         0F          Decimal:     15
            +7F                     +127
            ---                     ----
             8E                      142
```

As a signed number, 8EH represents -116, which is clearly not the correct result of 142; therefore, the OV bit is set.

EXAMPLE 2.7 What is the state of the overflow flag and the content of the accumulator after the execution of the following instruction sequence?

```
MOV R7,#0FFH
MOV A,#0FH
ADD A,R7
```

Solution

OV = 0, ACC = 0EH

Discussion

R7 is initialized with 0FFH, which as a signed number equals -1_{10}. The accumulator is initialized with 0FH, which equals 15_{10}. The result of the addition is 15 + (-1) = 14 = 0EH. Since 14 is within the allowable range for 8-bit signed numbers (-128 to +127), no overflow occurs and the OV bit is cleared. (Note, however, that the C bit is set because the addition generates a carry out of bit 7.)

2.6.1.6 Parity Bit The parity bit (P) is automatically set or cleared each machine cycle to establish even parity with the accumulator. The number of 1-bits in the accumulator plus the P bit is always even. If, for example, the accumulator contains 10101101, P will contain 1 (establishing a total of six 1-bits; i.e., an even number of 1s). The parity bit is most commonly used in conjunction with serial port routines to include a parity bit before transmission or to check for parity after reception.

EXAMPLE 2.8 What is the state of the P bit after execution of the following instruction?

```
MOV A,#55H
```

Solution

P = 0

Discussion

In binary, 55H equals 01010101B. This pattern has four bits equal to 1. Since 4 is an even number, the P bit is set to zero. The total number of bits equal to 1, counting those in the accumulator and the P bit, is four, thus achieving even parity.

2.6.2 B Register

The B register, or accumulator B, at address 0F0H is used along with the accumulator for multiply and divide operations. The MUL AB instruction multiplies the 8-bit unsigned values in A and B and leaves the 16-bit result in A (low-byte) and B (high-byte). The DIV AB instruction divides A by B, leaving the integer result in A and the remainder in B. The B register can also be treated as a general-purpose scratch-pad register. It is bit-addressable through bit addresses 0F0H to 0F7H.

2.6.3 Stack Pointer

The stack pointer (SP) is an 8-bit register at address 81H. It contains the address of the data item currently on the top of the stack. Stack operations include "pushing" data on the stack and "popping" data off the stack. Pushing on the stack increments the SP before writing data, and popping from the stack reads data and then decrements the SP. The 8051 stack is kept in internal RAM and is limited to addresses accessible by indirect addressing. These are the first 128 bytes on the 8031/8051 or the full 256 bytes of on-chip RAM on the 8032/8052.

To reinitialize the SP with the stack beginning at 60H, the following instruction is used:

```
MOV SP,#5FH
```

On the 8031/8051 this would limit the stack to 32 bytes, since the uppermost address of on-chip RAM is 7FH. The value 5FH is used, since the SP increments to 60H before the first push operation.

EXAMPLE 2.9 What instruction would be used to initialize the stack pointer on an 8052 to create a 48-byte stack at the top of internal memory?

Solution

```
MOV SP,#0CFH
```

Discussion

Since the 8052 has 256 bytes of internal memory, a 48-byte stack positioned at the top would occupy locations 0D0H to 0FFH. Since the SP increments before the first item is placed on the stack, it should be initialized to the address just below the starting location, 0CFH.

Designers may choose not to reinitialize the stack pointer and let it retain its default value upon system reset. The reset value of 07H maintains compatibility with the 8051's predecessor, the 8048, and results in the first stack write storing data in location 08H. If the application software does not reinitialize the SP, then register bank 1 (and perhaps 2 and 3) is not available, since this area of internal RAM is the stack.

The stack is accessed explicitly by the PUSH and POP instructions to temporarily store and retrieve data, or implicitly by the subroutine call (ACALL, LCALL) and return (RET, RETI) instructions to save and restore the program counter.

2.6.4 Data Pointer

The data pointer (DPTR), used to access external code or data memory, is a 16-bit register at addresses 82H (DPL, low-byte) and 83H (DPH, high-byte). The following three instructions write 55H into external RAM location 1000H:

```
MOV  A,#55H
MOV  DPTR,#1000H
MOVX @DPTR,A
```

The first instruction uses immediate addressing to load the data constant 55H into the accumulator. The second instruction also uses immediate addressing, this time to load the

16-bit address constant 1000H into the data pointer. The third instruction uses indirect addressing to move the value in A (55H) to the external RAM location whose address is in the DPTR (1000H). The "X" in the mnemonic "MOVX" indicates that the move instruction accesses external data memory.

2.6.5 Port Registers

The 8051 I/O ports consist of Port 0 at address 80H, Port 1 at address 90H, Port 2 at address 0A0H, and Port 3 at address 0B0H. Ports 0, 2, and 3 may not be available for I/O if external memory is used or if some of the 8051 special features are used (interrupts, serial port, etc.). Nevertheless, P1.2 to P1.7 are always available as general purpose I/O lines.

All ports are bit-addressable. This capability provides powerful interfacing possibilities. If a motor is connected through a solenoid and transistor driver to Port 1 bit 7, for example, it could be turned on and off using a single 8051 instruction:

```
SETB P1.7
```

might turn the motor on, and

```
CLR P1.7
```

might turn it off.

The instructions above use the dot operator to address a bit within a bit-addressable byte location. The assembler performs the necessary conversion; thus, the following two instructions are the same:

```
CLR P1.7
CLR 97H
```

The use of predefined assembler symbols (e.g., P1) is discussed in detail in Chapter 7.

In another example, consider the interface to a device with a status bit called BUSY, which is set when the device is busy and clear when it is ready. If BUSY connects to, say, Port 1 bit 5, the following loop could be used to wait for the device to become ready:

```
WAIT: JB P1.5,WAIT
```

This instruction means "if the bit P1.5 is set, jump to the label WAIT." In other words "jump back and check it again."

2.6.6 Timer Registers

The 8051 contains two 16-bit timer/counters for timing intervals or counting events. Timer 0 is at addresses 8AH (TL0, low-byte) and 8CH (TH0, high-byte), and Timer 1 is at addresses 8BH (TLl, low-byte) and 8DH (TH1, high-byte). Timer operation is set by the timer mode register (TMOD) at address 89H and the timer control register (TCON) at address 88H. Only TCON is bit-addressable. The timers are discussed in detail in Chapter 4.

2.6.7 Serial Port Registers

The 8051 contains an on-chip serial port for communicating with serial devices such as terminals or modems, or for interfaces with other ICs with a serial interface (A/D converters, shift registers, nonvolatile RAMs, etc.). One register, the serial data buffer (SBUF) at address 99H, holds both the transmit data and receive data. Writing to SBUF loads data for transmission; reading SBUF accesses received data. Various modes of operation are programmable through the bit-addressable serial port control register (SCON) at address 98H. Serial port operation is discussed in detail in Chapter 5.

2.6.8 Interrupt Registers

The 8051 has a 5-source, 2-priority level interrupt structure. Interrupts are disabled after a system reset and then enabled by writing to the interrupt enable register (1E) at address 0A8H. The priority level is set through the interrupt priority register (IP) at address 0B8H. Both registers are bit-addressable. Interrupts are discussed in detail in Chapter 6.

2.6.9 Power Control Register

The power control register (PCON) at address 87H contains miscellaneous control bits. These are summarized in Table 2-4.

The SMOD bit doubles the serial port baud rate when in Modes 1, 2, or 3. (See Chapter 5.) PCON bits 6, 5, and 4 are undefined. Bits 3 and 2 are general-purpose flag bits available for user applications.

The power control bits, power down (PD) and idle (IDL), were originally available in all MCS-51^{TM} family ICs but are now implemented only in the CMOS versions. PCON is not bit-addressable.

2.6.9.1 Idle Mode An instruction that sets the IDL bit will be the last instruction executed before entering idle mode. In idle mode the internal clock signal is gated off to the CPU, but not to the interrupt, timer, and serial port functions. The CPU status is preserved and all register contents are maintained. Port pins also retain their logic levels. ALE and ($\overline{\text{PSEN}}$) are held high.

Idle mode is terminated by any enabled interrupt or by a system reset. Either condition clears the IDL bit.

2.6.9.2 Power Down Mode An instruction that sets the PD bit will be the last instruction executed before entering power down mode. In power down mode, (1) the on-chip

TABLE 2-4 PCON register summary

Bit	Symbol	Description
7	SMOD	Double-baud rate bit; when set, baud rate is doubled in serial port modes 1, 2, or 3
6		Undefined
5		Undefined
4		Undefined
3	GF 1	General purpose flag bit 1
2	GFO	General purpose flag bit 0
1*	PD	Power down; set to activate power down mode; only exit is
0*	IDL	Idle mode; set to activate idle mode; only exit is an interrupt or system reset

*Only implemented in CMOS versions

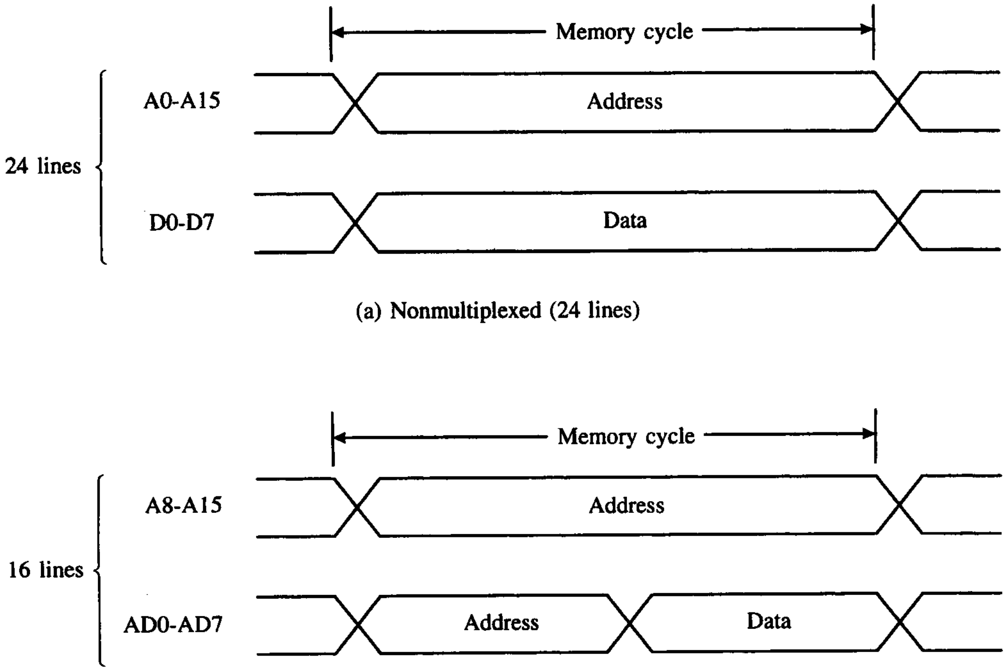
oscillator is stopped, (2) all functions are stopped, (3) all on-chip RAM contents are retained, (4) port pins retain their logic levels, and (5) ALE and ($\overline{\text{PSEN}}$) are held low. The only exit is a system reset.

During power down mode, Vcc can be as low as 2V. Care should be taken not to lower Vcc until after power down mode is entered, and to restore V_{cc} to 5V at least 10 oscillator cycles before the RST pin goes low again (upon leaving power down mode).

2.7 EXTERNAL MEMORY

It is important that microcontrollers have expansion capabilities beyond the on-chip resources to avoid a potential design bottleneck. If any resources must be expanded (memory, I/O, etc.), then the capability must exist. The MCS-51^{TM} architecture provides this in the form of a 64K external code memory space and a 64K external data memory space. Extra ROM and RAM can be added as needed. Peripheral interface ICs can also be added to expand the I/O capability. These become part of the external data memory space using memory-mapped I/O.

When external memory is used, Port 0 is unavailable as an I/O port. It becomes a multiplexed address (A0-A7) and data (D0-D7) bus, with ALE latching the low-byte of the address at the beginning of each external memory cycle. Port 2 is usually (but not always) employed for the high-byte of the address bus.



(a) Nonmultiplexed (24 lines)

(b) Multiplexed (16 lines)

FIGURE 2-8
Multiplexing the address bus (low-byte) and data bus

Before discussing the specific details of multiplexing the address and data buses, the general idea is presented in Figure 2-8. A nonmultiplexed arrangement uses 16 dedicated address lines and eight dedicated data lines, for a total of 24 pins. The multiplexed arrangement combines eight lines for the data bus and the low-byte of the address bus, with another eight lines for the high-byte of the address bus—a total of 16 pins. The savings in pins allows other functions to be offered in a 40-pin DIP (dual inline package).

Here's how the multiplexed arrangement works: during the first half of each memory cycle, the low-byte of the address is provided on Port 0 and is latched using ALE. A 74HC373 (or equivalent) latch holds the low-byte of the address stable for the duration of the memory cycle. During the second half of the memory cycle, Port 0 is used as the data bus, and data are read or written depending on the operation.

2.7.1 Accessing External Code Memory

External code memory is read-only memory enabled by the ($\overline{\text{PSEN}}$) signal. When an external EPROM is used, both Ports 0 and 2 are unavailable as general purpose I/O ports. The hardware connections for external EPROM memory are shown in Figure 2-9.

An 8051 machine cycle is 12 oscillator periods. If the on-chip oscillator is driven by a 12 MHz crystal, a machine cycle is 1 µs in duration. During a typical machine cycle, ALE pulses twice and 2 bytes are read from program memory. (If the current instruction is a 1-byte instruction, the second byte is discarded.) The timing for this operation, known as an opcode fetch, is shown in Figure 2-10.

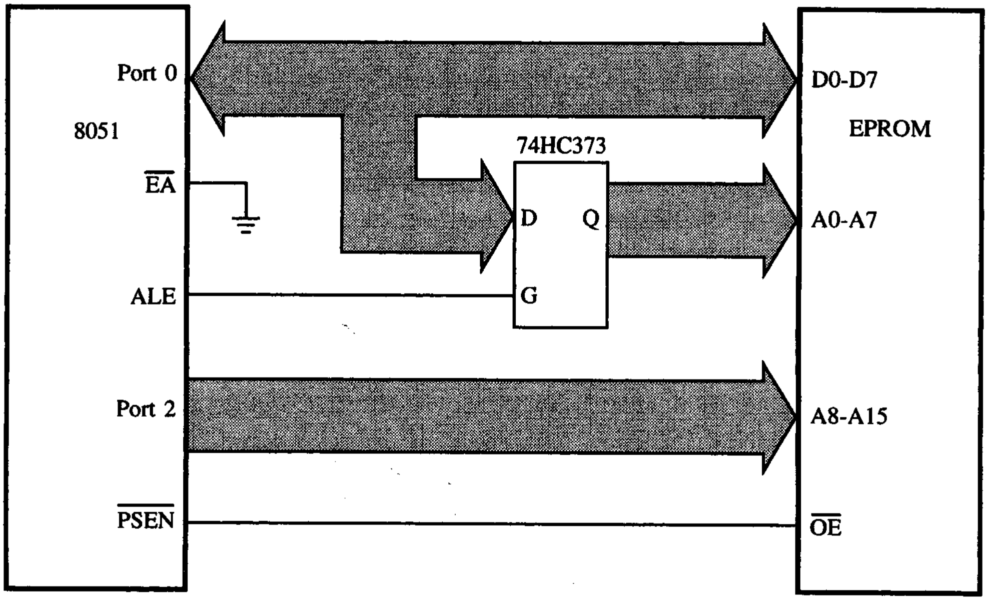
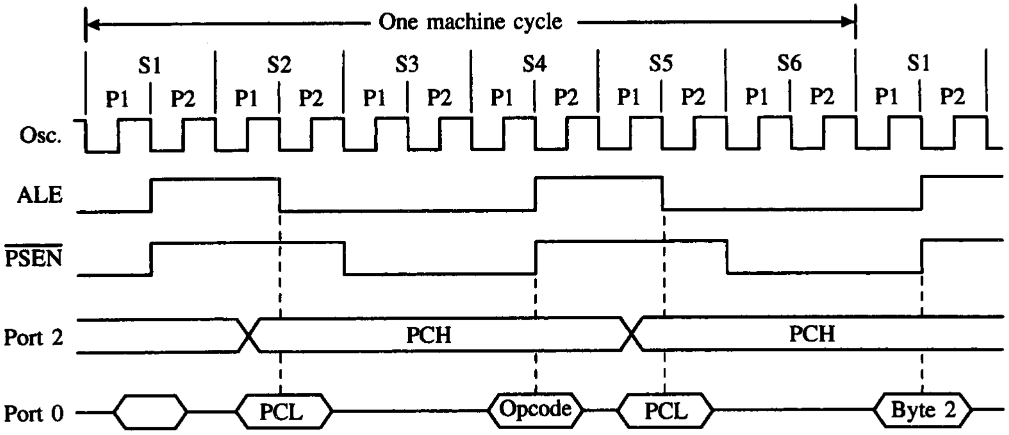


FIGURE 2-9
Accessing external code memory



Note: PCH = Program counter high byte
PCL = Program counter low byte

FIGURE 2-10
Read timing for external code memory

2.7.2 Accessing External Memory

External data memory is read/write memory enabled by the $\overline{\text{RD}}$ and $\overline{\text{WR}}$—the alternate pin functions for P3.7 and P3.6. The only access to external data memory is with the MOVX instruction, using either the 16-bit data pointer (DPTR), R0, or R1 as the address register.

RAMs may be interfaced to the 8051 the same way as EPROMs except the. RD line connects to the RAM's output enable ($\overline{\text{WR}}$) line and $\overline{\text{WR}}$ connects to the RAM's write ($\overline{\text{W}}$) line. The connections for the address and data bus are the same as for EPROMs. Using Ports 0 and 2 as above, up to 64 K bytes of external data RAM can be connected to the 8051.

A timing diagram for a read operation to external data memory is shown in Figure 2-11 for the MOVX A,@DPTR instruction. Notice that both an ALE pulse and a ($\overline{\text{PSEN}}$) pulse are skipped in lieu of a pulse on the RD line to enable the RAM.[2]

The timing for a write cycle (MOVX @DPTR,A) is much the same except the $\overline{\text{WR}}$ line pulses low and data are output on Port 0. ($\overline{\text{RD}}$ remains high.)

Port 2 is relieved of its alternate function (of supplying the high-byte of the address) in minimum component systems, which use no external code memory and only a small amount of external data memory. Eight-bit addresses can access external data memory for small page-oriented memory configurations. If more than one 256-byte page of RAM

[2] If MOVX instructions (and external RAM) are never used, then ALE pulses consistently at 1/6th the crystal frequency.

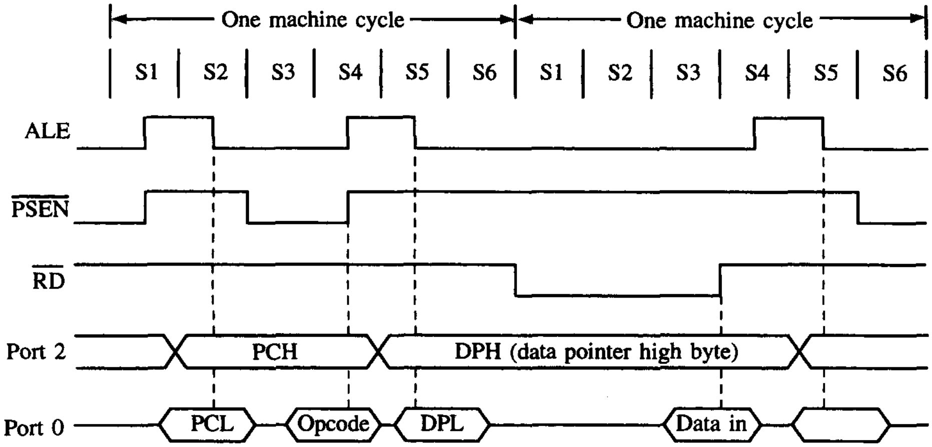


FIGURE 2-11
Timing for MOVX instruction

is used, then a few bits from Port 2 (or some other port) can select a page. For example, a 1K byte RAM (i.e., four 256-byte pages) can be interfaced to the 8051 as shown in Figure 2-12.

Port 2 bits 0 and 1 must be initialized to select a page, and then a MOVX instruction is used to read or write data within that page. For example, assuming P2.0 = P2.1 = 0, the

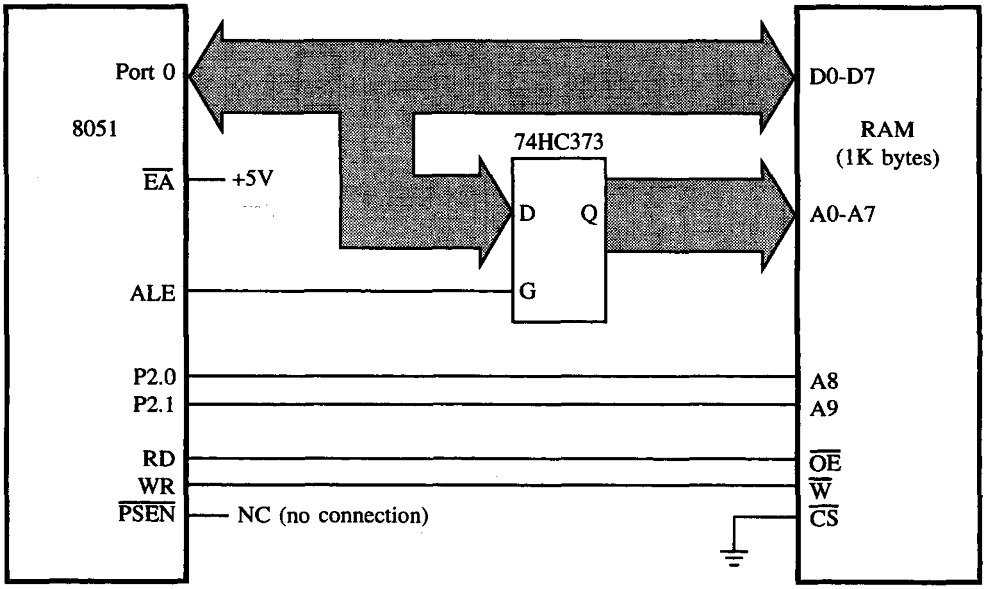


FIGURE 2-12
Interface to 1K RAM

following instructions could be used to read the content of external RAM address 0050H into the accumulator:

```
MOV  R0,#50H
MOVX A,@R0
```

In order to read the last address in this RAM, 03FFH, the two page select bits must be set. The following instruction sequence could be used:

```
SETB P2.0
SETB P2.1
MOV  R0,#0FFH
MOVX A,@R0
```

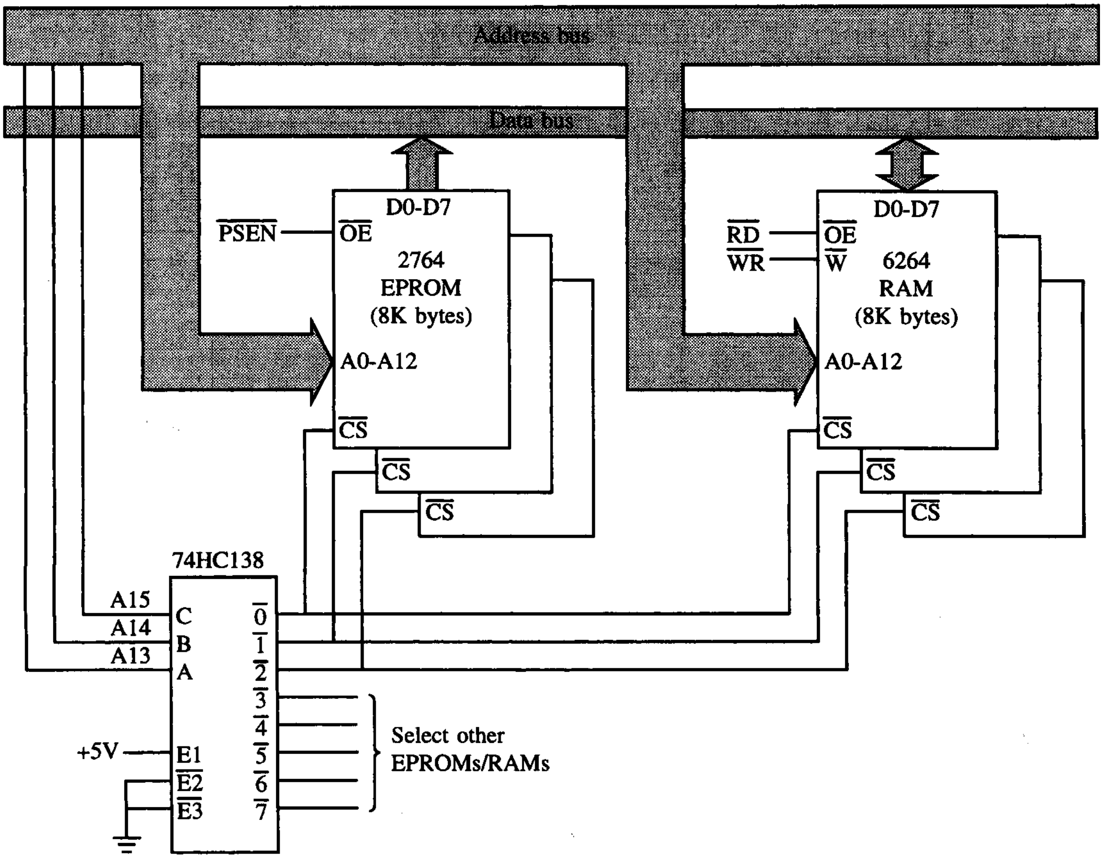
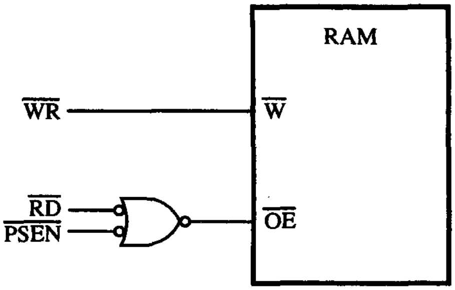


FIGURE 2–13
Address decoding

FIGURE 2-14
Overlapping the external code and data spaces



A feature of this design is that Port 2 bits 2 to 7 are not needed as address bits, as they would be if the DPTR was the address register. P2.2 to P2.7 are available for I/O purposes.

2.7.3 Address Decoding

If multiple EPROMs and/or RAMs are interfaced to an 8051, address decoding is required. The decoding is similar to that required for most microprocessors. For example, if 8K byte EPROMs or RAMs are used, then the address bus must be decoded to select memory ICs on 8K boundaries: 0000H-1FFFH, 2000H-3FFFH, and so on.

Typically, a decoder IC such as the 74HC138 is used with its outputs connected to the chip select ($\overline{\text{RD}}$) inputs on the memory ICs. This is illustrated in Figure 2-13 for a system with multiple 2764 8K EPROMs and 6264 8K RAMs. Remember, due to the separate enable lines ($\overline{\text{PSEN}}$ for code memory, $\overline{\text{RD}}$ and $\overline{\text{WR}}$ for data memory), the 8051 can accommodate up to 64K *each* of EPROM and RAM.

2.7.4 Overlapping the External Code and Data Spaces

Since code memory is read-only, an awkward situation arises during the development of 8051 software. How is software "written into" a target system for debugging if it can only be executed from the "read-only" code space? A common trick is to overlap the external code and data memory spaces. Since $\overline{\text{PSEN}}$ is used to read code memory and $\overline{\text{RD}}$ is used to read data memory, a RAM can occupy code *and* data memory space by connecting its OE line to the logical AND (negative-input NOR) of $\overline{\text{PSEN}}$ and $\overline{\text{RD}}$. The circuit shown in Figure 2-14 allows the RAM IC to be written as data memory and read as data *or* code memory. Thus, a program can be loaded into the RAM (by writing to it as data memory) and executed (by accessing it as code memory).

2.8 8032/8052 ENHANCEMENTS

The 8032/8052 ICs (and the CMOS and/or EPROM versions) offer two enhancements to the 8031/8051 ICs. First, there is an additional 128 bytes of on-chip RAM from addresses

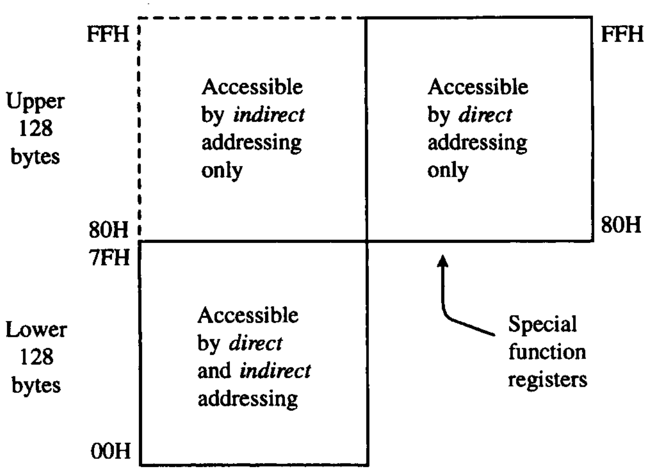

FFH
Upper 128 bytes
Accessible by *indirect* addressing only
Accessible by *direct* addressing only
FFH
80H
7FH
80H
Lower 128 bytes
Accessible by *direct* and *indirect* addressing
Special function registers
00H

FIGURE 2-15
8032/52 memory spaces

80H to 0FFH. So as not to conflict with the SFRs (which have the same addresses), the additional 1/8K of RAM is only accessible using indirect addressing. An instruction such as

```
MOV A,0F0H
```

moves the contents of the B register to the accumulator on all MCS-51™ ICs. The instruction sequence

```
MOV R0,#0F0H
MOV A,@R0
```

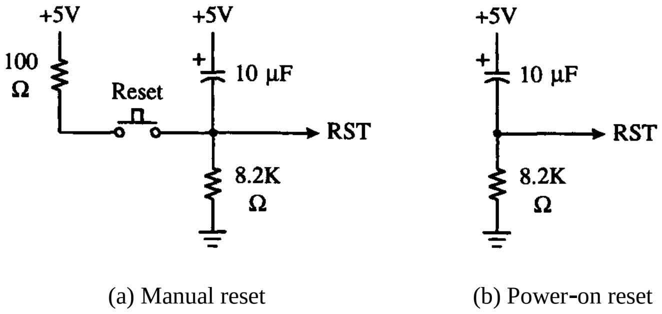
reads into the accumulator the content of internal address 0F0H on the 8032/8052 ICs but is undefined on the 8031/8051 ICs. The internal memory organization of the 8032/8052 ICs is summarized in Figure 2-15.

The second 8032/8052 enhancement is an additional 16-bit timer, Timer 2, which is programmed through five additional special function registers. These are summarized in Table 2-5. See Chapter 4 for more details.

TABLE 2-5
Timer 2 registers

Registration	Address	Description	Bit-Addressable
T2CON	0C8H	Control	Yes
RCAP2L	0CAH	Low-byte capture	No
RCAP2H	0CBH	High-byte capture	No
TL2	0CCH	Timer 2 low-byte	No
TH2	0CDH	Timer 2 high-byte	No

FIGURE 2-16
Two circuits for system reset.
(a) Manual reset
(b) Power-on reset.



2.9 RESET OPERATION

The 8051 is reset by holding RST high for at least two machine cycles and then returning it low. RST may be manually activated using a switch, or it may be activated upon power-up using an RC (resistor-capacitor) network. Figure 2-16 illustrates two circuits for implementing system reset.

The state of all the 8051 registers after a system reset is summarized in Table 2-6. The most important of these registers, perhaps, is the program counter, which is loaded with 0000H. When RST returns low, program execution always begins at the first location in code memory: address 0000H. The content of on-chip RAM is not affected by a reset operation.

TABLE 2-6
Register values after system reset

Register(s)	Contents
Program counter	0000H
Accumulator	00H
B register	00H
PSW	00H
SP	07H
DPTR	0000H
Ports 0-3	0FFH
IP (8031/8051)	XX000000B
IP (8032/8052)	0X000000B
IE (8031/8051)	0XX00000B
E (8032/8052)	XX000000B
Timer registers	00H
SCON	00H
SBUF	00H
PCON (HMOS)	0XXXXXXXB
PCON (CMOS)	0XX00000B

SUMMARY

This chapter has summarized the 8051 hardware architecture. Before developing useful applications, though, we must understand the 8051 instruction set. The next chapter focuses on the 8051 instructions and addressing modes. The discussions of the timer, serial port, and interrupt SFRs were deliberately sparse in this chapter because dedicated chapters follow that examine these in detail.

PROBLEMS

2.1 Name four manufacturers of the 8051 microcontroller, besides Intel.

2.2 Which device in the MCS-51TM family would probably be used for a product that will be manufactured in large quantities with a large on-chip program?

2.3 What instruction could be used to set the least-significant bit at byte address 25H?

2.4 What instruction sequence could be used to place the logical OR of the bits at bit addresses 00H and 01H into bit address 02H?

2.5 What instruction sequence could be used to read bit 0 of Port 0 and write the state of the bit read to bit 0 of Port 3?

2.6 Illustrate an instruction sequence to read bit 0 and bit 1 of Port 0 and write a status condition to bit 0 of Port 3 as follows: If both bits read are 1, write a 1 to the output status bit, otherwise write a B.

2.7 Illustrate an instruction sequence to read bit 0 and bit 1 of Port 0 and write a status condition to bit 0 of Port 3 as follows: If either bit is 1, but not both, write a 1 to the output status bit, otherwise write a 0.

2.8 Illustrate an instruction sequence to read bit 0 and bit 1 of Port 0 and write a status condition to bit 0 of Port 3 as follows: If either bit read is 1, write a 0 to the output status bit, otherwise write a 1.

2.9 For the three preceding questions, illustrate the operation using logic gates.

2.10 What bit addresses are set to 1 as a result of the following instructions?

```
a. MOV 26H,#26H
b. MOV R0,#26H
c. MOV @R0,#7AH
d. MOV A,#13H
e. MOV 30H,#55H
f. XRL 30H,#0AAH
g. SETB P1.1
h. MOV P3,#0CH
```

2.11 What 1-byte instruction has the same effect as the following 2-byte instruc-tion?

```
MOV 0E0H,#55H
```

2.12 Illustrate an instruction sequence to store the value 0ABH in external RAM at address 9A00.

2.13 How many special function registers are defined on the 8052?

2.14 What is the value of the 8051's stack pointer immediately after a system reset?

2.15 What instruction would be used to initialize the stack pointer to create a 64-byte stack at the top of internal memory (a) on the 8031 or (b) on the 8032?

2.16 What instruction would be used to initialize the stack pointer to create a 32-byte stack at the top of memory (a) on the 8051 or (b) on the 8052?

2.17 A certain subroutine makes extensive use of registers R0-R7. Illustrate how this subroutine could switch the active register bank to bank 3 upon entry and restore the previously active register bank upon exit.

2.18 What is the active register bank after execution of each of the following instructions?

```
a. MOV PSW,#0FDH
b. MOV PSW,#18H
c. MOV PSW,#08H
```

2.19 What is the active register bank after execution of each of the following instructions?

```
i. MOV PSW,#0C8H
j. MOV PSW,#50H
k. MOV PSW,#10H
```

2.20 The 8BC31BH-1 can operate using a 16 MHz crystal connected to its XTAL1 and XTAL2 inputs. If MOVX instructions are not used, what is the frequency of the signal on ALE?

2.21 If an 8051 is operating from a 4 MHz crystal, what is the duration of a machine cycle?

2.22 If an 8051 is operating from a 10 MHz crystal, what is the frequency of the waveform on ALE? Assume the software is not accessing external RAM.

2.23 What is the duty cycle of ALE? Assume that software is not accessing external RAM. (Note: Duty cycle is defined as the proportion of time a pulse waveform is high.)

2.24 Section 2.9 states that the 8051 is reset if the RST pin is held high for a minimum of two machine cycles. (Note: As stated in the 8051's DC Characteristics in Appendix E, a "high" on RST is 2.5 volts minimum.)

a) If an 8051 is operating from an 8 MHz crystal, what is the minimum length of time for RST to be high to achieve a system reset?

b) Figure 2-16a shows an RC circuit for a manual reset. While the reset button is depressed, RST = 5 volts and the system is held in a reset state. How long after the reset button is released will the 8051 remain in a reset state?

2.25 How many low-power Schottky loads can be driven by the port line P1.7 on pin 8?

2.26 Name the 8051 control bus signals used to select external EPROMs and external RAMs.

2.27 What is the bit address of the most-significant bit at byte address 25H in the 8051's internal data memory?

2.28 What is the bit address of bit 3 in byte address 2FH in the 8051's internal data memory?

2.29 Some of the bit addressable locations in the 8031's on-chip data memory are bought out as signals on the 8031 IC. Which ones? What are their pin numbers and what are their bit addresses?

2.30 Identify the bit position and byte address for each of the following SETB instructions.

a. `SETB 37H`

b. `SETB 77H`

c. `SETB 0F7H`

2.31 Identify the bit position and byte address for each of the following SETB instructions.

a. `SETB 0A8H`

b. `SETB 84H`

c. `SETB 63H`

2.32 What instruction sets the least-significant bit of the accumulator without affecting the other seven bits?

2.33 What is the state of the P bit in the PSW after execution of each of the following instructions?

a. `MOV A,#55H`

b. `MOV A,#0F8H`

c. `MOV A,#0FFH`

2.34 What is the state of the P bit in the PSW after execution of each of the following instructions?

a. `CLR A`

b. `MOV A,#03H`

c. `MOV A,#0ABH`

2.35 What instruction sequence could be used to copy the content of R7 to external RAM location 100H?

2.36 Illustrate an instruction sequence to read external RAM address 08F5H and place the byte read into the B accumulator.

2.37 Assume the first instruction executed following a system reset is a subroutine call. At what addresses in internal RAM is the program counter saved before branching to the subroutine?

2.38 Consider the instruction MOV SP,#08FH. (a) What is the effect of this instruction if executed on an 8032? (b) What is its effect if executed on an 8031?

2.39 If an 8031 program is designed to use only register bank zero, then the stack pointer need not be initialized. However, if an 8031 program is designed to use all four register banks, then it is imperative that the stack pointer be explicitly initialized. Why?

2.40 What is the difference between the 8051's idle mode and power-down mode?

2.41 What instruction could be used to force the 8051 in power-down mode?

2.42 Illustrate how two 32K-byte static RAMs could be interfaced to the 8051 so that they occupy the full 64K external data space.

2.43 If the contents of the following are:

```
A = 55H
B = 11H
```

```
Internal RAM location 30H = 33H
SP = 00H
```

What would the contents be upon reset?

2.44 Explain the term "I/O expansion."

2.45 What are the differences between memory-mapped I/O and port I/O? Explain.

2.46 An 8051 microcontroller is to execute programs from an external ROM. Specify the control signals and their logic values for enabling external ROM access. Also specify the register (and its initial value after reset) that will identify the first instruction to be fetched.

2.47 The 8051 is said to have 128 bytes of internal data memory. However, referring to the memory map in Figure 2-7, the on-chip data memory of the 8051 is from 00H to 0FFH, giving 256 locations. Why is this so?

2.48 Explain the difference between the stack and the stack pointer (SP) by using an example.

CHAPTER 3

Instruction Set Summary

3.1 INTRODUCTION

Just as sentences are made of words, programs are made of instructions. When programs are constructed from logical, well-thought-out sequences of instructions, fast, efficient, and even elegant programs result. Unique to each family of computers is its instruction set, a repertoire of primitive operations such as "add," "move," or "jump." This chapter introduces the MCS-51^{TM} instruction set through an examination of addressing modes and examples from typical programming situations. Appendix A contains a summary chart of all the 8051 instructions. Appendix C provides a detailed description of each instruction. These appendices should be consulted for subsequent reference.

Programming techniques are not discussed, nor is the operation of the assembler program used to convert assembly language programs (mnemonics, labels, etc.) into machine language programs (binary codes). These topics are the subject of Chapter 7.

The MCS-51^{TM} instruction set is optimized for 8-bit control applications. It provides a variety of fast, compact addressing modes for accessing the internal RAM to facilitate operations on small data structures. The instruction set offers extensive support for 1-bit variables, allowing direct bit manipulation in control and logic systems that require Boolean processing.

As typical of 8-bit processors, 8051 instructions have 8-bit opcodes. This structure provides a possibility of $2^8 = 256$ instructions. Of these, 255 are implemented and 1 is undefined. As well as the opcode, some instructions have one or two additional bytes for data or addresses. In all, there are 139 1-byte instructions, 92 2-byte instructions, and 24 3-byte instructions. The *Opcode Map* in Appendix B shows, for each opcode, the mnemonic, the number of bytes in the instruction, and the number of machine cycles to execute the instruction.

3.2 ADDRESSING MODES

When instructions operate on data, the question arises: "Where are the data?" The answer to this question lies in the 8051's "addressing modes." There are several possible addressing modes and there are several possible answers to the question, such as "in byte 2 of the instruction," "in register R4," "in direct address 35H," or perhaps "in external data memory at the address contained in the data pointer."

Addressing modes are an integral part of each computer's instruction set. They allow specifying the source or destination of data in different ways, depending on the programming situation. In this section, we'll examine all the 8051 addressing modes and give examples of each. There are eight modes available:

- Register
- Direct
- Indirect
- Immediate
- Relative
- Absolute
- Long
- Indexed

3.2.1 Register Addressing

The 8051 programmer has access to eight "working registers," numbered R0 through R7. Instructions using register addressing are encoded using the three least-significant bits of the instruction opcode to specify a register within this logical address space. Thus, a function code and operand address can be combined to form a short (1-byte) instruction. (See Figure 3-1a.)

The 8051 assembly language indicates register addressing with the symbol *Rn* where *n* is from 0 to 7. For example, to add the contents of Register 7 to the accumulator, the following instruction is used

```
ADD A,R7
```

and the opcode is 00101111B. The upper five bits, 00101, indicate the instruction, and the lower three bits, 111, the register. Convince yourself that this is the correct opcode by looking up this instruction in Appendix C.

EXAMPLE 3.1

What is the opcode for the following instruction? What does this instruction do?

```
MOV A,R7
```

Solution

0EFH. This instruction moves the 8-bit content of register 7 (in the active register bank) to the accumulator.

Discussion

Appendix C lists all 8051 instructions, sorted alphabetically by mnemonic. The general form of move byte instructions is

```
MOV destination_byte,source_byte
```

There are 15 variations identified. In this example, we are concerned with MOV A,Rn. The binary opcode appears as 11101rrr. The low-order three bits identify the source register, which is R7 in this example. Substituting "111" for "rrr" yields an opcode of 11101111B = 0EFH.

There are four "banks" of working registers, but only one is active at a time. Physically, the register banks occupy the first 32 bytes of on-chip data RAM (addresses 00H-1FH) with PSW bits 4 and 3 determining the active bank. A hardware reset enables bank 0, but a different bank is selected by modifying PSW bits 4 and 3 accordingly. For example, the instruction

```
MOV PSW,#00011000B
```

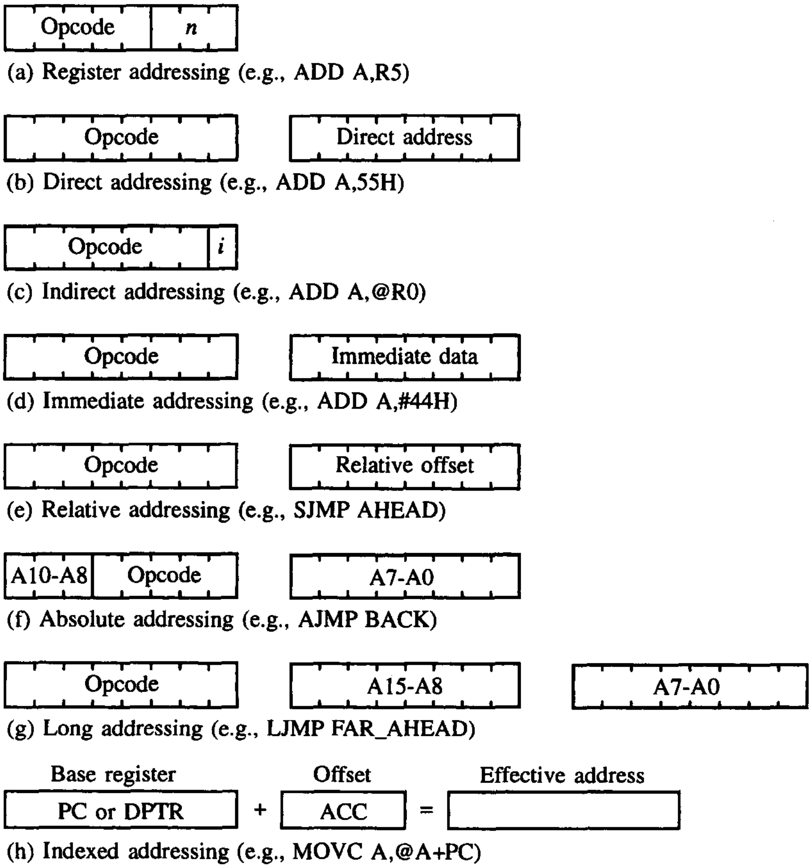

Opcode
n
(a) Register addressing (e.g., ADD A,R5)
Opcode
Direct address
(b) Direct addressing (e.g., ADD A,55H)
Opcode
i
(c) Indirect addressing (e.g., ADD A,@R0)
Opcode
Immediate data
(d) Immediate addressing (e.g., ADD A,#44H)
Opcode
Relative offset
(e) Relative addressing (e.g., SJMP AHEAD)
A10-A8
Opcode
A7-A0
(f) Absolute addressing (e.g., AJMP BACK)
Opcode
A15-A8
A7-A0
(g) Long addressing (e.g., LJMP FAR_AHEAD)
Base register
Offset
Effective address
PC or DPTR
+
ACC
=
(h) Indexed addressing (e.g., MOVC A,@A+PC)

FIGURE 3-1
8051 Addressing modes. (a) Register addressing (b) Direct addressing (c) Indirect addressing (d) Immediate addressing (e) Relative addressing (f) Absolute addressing (g) Long addressing (h) Indexed addressing

activates register bank 3 by setting the register bank, select bits (RS1 and RS0) in PSW bit positions 4 and 3.

Some instructions are specific to a certain register, such as the accumulator, data pointer, etc., so address bits are not needed. The opcode itself indicates the register. These "register-specific" instructions refer to the accumulator as "A," the data pointer as "DPTR," the program counter as "PC," the carry flag as "C," and the accumulator-B register pair as "AB." For example,

```
INC DPTR
```

is a 1-byte instruction that adds to the 16-bit data pointer. Consult Appendix C to determine the opcode for this instruction.

EXAMPLE 3.2

(a) What is the opcode for the following instruction? (b) What does this instruction do?

```
MUL AB
```

Solution

(a) 0A4H. (b) This instruction multiplies the unsigned 8-bit value in the accumulator by the unsigned 8-bit value in register B. The 16-bit product is left in the accumulator (low byte) and register B (high byte).

3.2.2 Direct Addressing

Direct addressing can access any on-chip variable or hardware register. An additional byte is appended to the opcode specifying the location to be used. (See Figure 3-1b.)

Depending on the high-order bit of the direct address, one of two on-chip memory spaces is selected. When bit 7 = 0, the direct address is between 0 and 127 (00H-7FH) and the 128 low-order on-chip RAM locations are referenced. All I/O ports and special function, control, or status registers, however, are assigned addresses between 128 and 255 (80H-0FFH). When the direct address byte is between these limits (bit 7 = 1), the corresponding special function register is accessed. For example, Ports 0 and 1 are assigned direct addresses 80H and 90H, respectively. It is usually not necessary to know the addresses of these registers; the assembler allows for and understands the mnemonic abbreviations ("P0" for Port 0, "TMOD" for timer mode register, etc.). Some assemblers, such as Intel's ASM51, automatically include the definition of predefined symbols. Other assemblers may use a separate source file containing the definitions. As an example of direct addressing, the instruction

```
MOV P1,A
```

transfers the content of the accumulator to Port 1. The direct address of Port 1 (90H) is determined by the assembler and inserted as byte 2 of the instruction. The source of the data, the accumulator, is specified implicitly in the opcode. Using Appendix C as a reference, the complete encoding of this instruction is

```
10001001 - 1st byte (opcode)
10010000 - 2nd byte (address of P1)
```

EXAMPLE 3.3

What are the machine-language bytes for the following instruction?

```
MOV SCON,#55H
```

Solution

75H, 98H, 55H

Discussion

The general form of this instruction is

```
MOV direct,#data
```

As noted in Appendix C, the instruction is three bytes long. The first byte is the opcode, 75H. The second byte is the direct address of the SCON special function register, 98H (see Figure 2-6 or Appendix D). The third byte is the immediate data, 55H.

3.2.3 Indirect Addressing

How is a variable identified if its address is determined, computed, or modified while a program is running? This situation arises when manipulating sequential memory locations, indexed entries within tables in RAM, multiple-precision numbers, or character strings. Register or direct addressing cannot be used, since they require operand addresses to be known at assemble-time.

The 8051 solution is indirect addressing. R0 and R1 may operate as "pointer" registers—their contents indicating an address in internal RAM where data are written or read. The least-significant bit of the instruction opcode determines which register (R0 or R1) is used as the pointer. (See Figure 3-1c.)

In 8051 assembly language, indirect addressing is represented by a commercial "at" sign (@) preceding R0 or R1. As an example, if R1 contains 40H and internal memory address 40H contains 55H, the instruction

```
MOV A,@R0
```

moves 55H into the accumulator.

EXAMPLE 3.4

(a) What is the opcode for the following instruction? (b) What does this instruction do?

```
MOV A,@R0
```

Solution

(a) 0E6H. (b) This instruction moves a byte of data from internal RAM to the accumulator. The data are moved from the location whose address is in R0.

Discussion

This instruction is of the general form

```
MOV A,@Ri
```

The binary opcode, as looked-up in Appendix C, is 1110011i. Since "R0" is specified as the indirect register, "0" is substituted for "i" in the opcode. The opcode is 11100110B = E6H.

Indirect addressing is essential when stepping through sequential memory locations. For example, the following instruction sequence clears internal RAM from address 60H to 7FH:

```
      MOV  R0,#60H
LOOP: MOV  @R0,#0
      INC  R0
      CJNE R0,#80H,LOOP
      (continue)
```

The first instruction initializes R0 with the starting address of the block of memory; the second instruction uses indirect addressing to move 00H to the location pointed at by R0 the third instruction increments the pointer (R0) to the next address; and the last instruction tests the pointer to see if the end of the block has been reached. The test uses 80H, rather than 7FH, because the increment occurs after the indirect move. This ensures the final location (7FH) is written to before terminating.

3.2.4 Immediate Addressing

When a source operand is a constant rather than a variable (i.e., the instruction uses a value known at assemble-time), then the constant can be incorporated into the instruction as a byte of "immediate" data. An additional instruction byte contains the value. (See Figure 3-1d.)

In assembly language, immediate operands are preceded by a number sign (#). The operand may be a numeric constant, a symbolic variable, or an arithmetic expression using constants, symbols, and operators. The assembler computes the value and substitutes the immediate data into the instruction. For example, the instruction

```
MOV A,#12
```

loads the value 12 (0CH) into the accumulator. (It is assumed the constant "12" is in decimal notation, since it is not followed by "H.")

With one exception, all instructions using immediate addressing use an 8-bit data constant for the immediate data. When initializing the data pointer, a 16-bit constant is required. For example,

```
MOV DPTR, #8000H
```

is a 3-byte instruction that loads the 16-bit constant 8000H into the data pointer.

EXAMPLE 3.5 What are the hexadecimal and binary machine language bytes for the following instruction?

```
ADD A,#15
```

Solution

Binary: 00100100B, 00001111B. Hexadecimal: 24H, 0FH.

Discussion

This instruction is of the general form

```
ADD A,#data
```

The opcode, as looked up in Appendix C, is 00100100B = 24H. The second byte of the instruction is the immediate data. This is specified in the instruction as 15_{10} = 00001111 B = 0FH.

3.2.5 Relative Addressing

Relative addressing is used only with certain jump instructions. A relative address (or offset) is an 8-bit signed value, which is added to the program counter to form the address of the next instruction executed. Since an 8-bit signed offset is used, the range for jumping is -128 to +127 locations. The relative offset is appended to the instruction as an additional byte. (See Figure 3-le.)

Prior to the addition, the program counter is incremented to the address following the jump instruction; thus, the new address is relative to the next instruction, *not* the address of the jump instruction. (See Figure 3-2.)

Normally, this detail is of no concern to the programmer, since jump destinations are usually specified as labels and the assembler determines the relative offset accordingly. For example, if the label THERE represents an instruction at location 1040H, and the instruction

```
SJMP THERE
```

is in memory at locations 1000H and 1001H, the assembler will assign a relative offset of 3EH as byte 2 of the instruction (1002H + 3EH = 1040H).

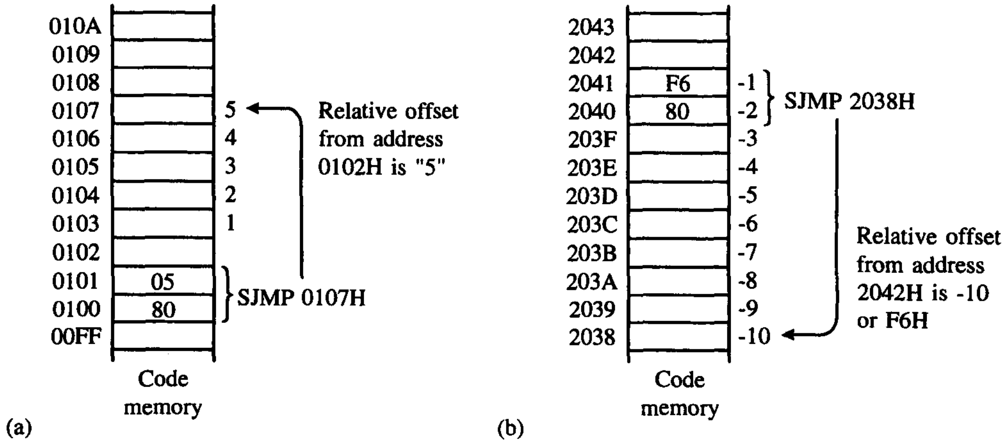


FIGURE 3-2

Calculating the offset for relative addressing. (a) Short jump ahead in memory (b) Short jump back in memory

Relative addressing offers the advantage of providing position-independent code (since "absolute" addresses are not used), but the disadvantage that the jump destinations are limited in range.

EXAMPLE 3.6 The instruction

```
SJMP 9030H
```

is in memory location 9000H and 9001H. What are the machine language bytes for this instruction?

Solution
80H, 2EH

Discussion
This instruction is of the general form

```
SJMP relative
```

As looked up in Appendix C, the instruction is two bytes long, beginning with 10000000B = 80H as the opcode. The second byte is an 8-bit signed value which is the relative offset. Since we are jumping "ahead" in memory for this example, the offset is positive. As shown in Figure 3-2, the source address is the address "after" the jump instruction. The offset is added to this value to obtain the destination address for the jump. Rather than draw a picture and count out the offset, as in Figure 3-2, we can compute the offset arithmetically as follows:

```
source address + offset = destination_address
```

or

```
offset = destination_address - source, address
```

In this example the source address is 9002H (the address *after* the jump instruction) and the destination address is 9030H, so

```
offset = 9030H - 9002H = 2EH
```

EXAMPLE 3.7 An SJMP instruction with a machine language representation of 80H F6H is in memory locations 0802H and 0803H. To what address will the jump occur?

Solution
07FAH

Discussion
The offset for this jump instruction is 0F6H, which is a negative number: The jump is "back" in memory. The source address for the jump is the address following the jump instruction, which for this example is 0804H. To compute the destination address, we simply add the

offset to the source address; however, there is a trick. Since our offset is negative and our address is 16 bits, we must "sign-extend" the offset and express it as a 16-bit number: 0FFF6H. The destination address is computed as follows:

```
  0804H (source address)
+0FFF6H (offset)
  07FAH (destination address)
```

Note: In adding a negative offset to a source address, as above, a carry is generated out of the most-significant bit. This is discarded.

3.2.6 Absolute Addressing

Absolute addressing is used only with the ACALL and AJMP instructions. These 2-byte instructions allow branching within the current 2K page of code memory by providing the 11 least-significant bits of the destination address in the opcode (A10-A8) and byte 2 of the instruction (A7-A0). (See Figure 3-1f.)

The upper five bits of the destination address are the current upper five bits in the program counter, so the instruction following the branch instruction and the destination for the branch instruction must be within the same 2K page, since A15-A11 do not change. (See Figure 3-3.) For example, if the label THERE represents an instruction at address 0F46H, and the instruction

```
AJMP THERE
```

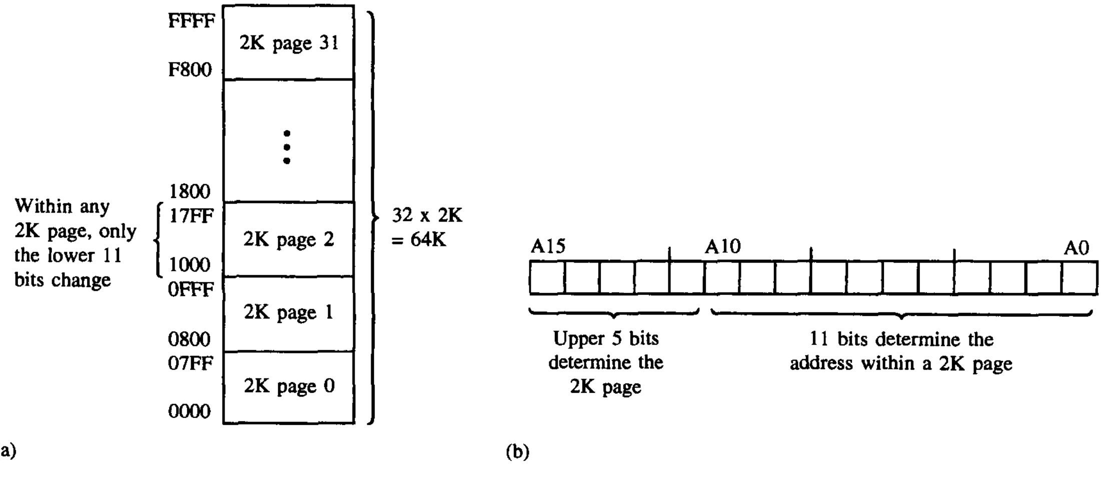

FFFF
F800
2K page 31
1800
17FF
1000
0FFF
0800
07FF
0000
2K page 2
2K page 1
2K page 0
Within any 2K page, only the lower 11 bits change
32 x 2K = 64K
(a)
A15
A10
A0
Upper 5 bits determine the 2K page
11 bits determine the address within a 2K page
(b)

FIGURE 3-3
Instruction encoding for absolute addressing. (a) Memory map showing 2K pages (b) Within any 2K page, the upper five address bits are the same for the source and destination addresses. The lower 11 bits of the destination are supplied in the instruction

is in memory locations 0900H and 0901H, the assembler will encode the instruction as

```
11100001 - 1st byte (Al0-A8 + opcode)
01000110 - 2nd byte (A7-A0)
```

The underlined bits are the low-order 11 bits of the destination address, 0F46H = 0000111101000110B. The upper five bits in the program counter will not change when this instruction executes. Note that both the AJMP instruction and the destination are within the 2K page bounded by 0800H and 0FFFH (see Figure 3-3), and therefore have the upper five address bits in common.

Absolute addressing offers the advantage of short (2-byte) instructions, but has the disadvantages of limiting the range for the destination and providing position-dependent code.

EXAMPLE 3.8 An ACALL instruction is in memory locations 1024H and 1025H. The subroutine to which the call is directed begins in memory location 17A6H. What are the machine language bytes for the ACALL instruction?

Solution

F1H, A6H

Discussion

As found in Appendix C, the encoding for the ACALL instruction is

```
aaa10001 aaaaaaaa
```

The low-order 11 bits of the destination address are inserted into the instruction with bits 10-8 placed in the high-order bits of the opcode and with bits 7-0 forming the second byte of the instruction. The destination address (17A6H) is shown below in binary with the low-order 11 bits identified as a group of three bits (10-8) and a group of eight bits (7-F).

```
00010111 10100110 = 17A6H
 ... aaa aaaaaaaa
```

The task is simply to correctly position the 11 bits from the destination address into the instruction bytes. This positioning is illustrated as follows:

```
lll1000l   10100ll0 = 0F1A6H
aaa............... aaaaaaaa
```

Note: Absolute addressing can only be used if the high-order five bits are the same in both the source and destination addresses. It is this property that identifies the source and destination addresses as falling within the same 2K page.

3.2.7 Long Addressing

Long addressing is used only with the LCALL and LJMP instructions. These 3-byte instructions include a full 16-bit destination address as bytes 2 and 3 of the instruction. (See Figure 3-1g.) The advantage is that the full 64K code space may be used, but the disadvantage is that the instructions are three bytes long and are position-dependent. Position-dependence is a disadvantage because the program cannot execute at different addresses. If, for example, a program begins at 2000H and an instruction such as LJMP 2040H appears, then the program cannot be moved to, say, 4000H. The LJMP instruction would still jump to 2040H, which is not the correct location after the program has been moved.

EXAMPLE 3.9 What are the machine language bytes for the following instruction?

```
LJMP 8AF2H
```

Solution

02H, 8AH, F2H

Discussion

As given in Appendix C, the LJMP instruction is three bytes long with the opcode (02H) in the first byte and the 16-bit destination address in bytes 2 and 3. The high byte of the destination address (8AH) is in byte 2, and the low byte (0F2H) is in byte 3.

3.2.8 Indexed Addressing

Indexed addressing uses a base register (either the program counter or the data pointer) and an offset (the accumulator) in forming the effective address for a JMP or MOVC instruction. (See Figure 3-1h.) Jump tables or look-up tables are easily created using indexed addressing. Examples are provided in Appendix C for the MOVC A,@A+<base-reg> and JMP @A+ DPTR instructions.

EXAMPLE 3.10 What is the opcode for the following instruction?

```
MOVC A,@A+DPTR
```

Solution

93H

Discussion

The answer is found simply by looking up the MOVC instruction in Appendix C. The instruction is only one byte long, with the opcode specifying both the operation and the addressing mode. This instruction moves a byte of data from code memory to the accumulator. The address in code memory is found by adding the index (the present state of the accumulator) to the base register (the data pointer). When the instruction finishes executing, the index is lost because it is overwritten with the value moved from code memory.

3.3 INSTRUCTION TYPES

The 8051 instructions are divided among five functional groups:

- Arithmetic
- Logical
- Data transfer

- Boolean variable
- Program branching

Appendix A provides a quick reference chart showing all the 8051 instructions by functional grouping. Once you are familiar with the instruction set, this chart should prove a handy and quick source of reference. We continue by examining instructions in each functional grouping from Appendix A.

3.3.1 Arithmetic Instructions

The arithmetic instructions are grouped together in Appendix A. Since four addressing modes are possible, the ADD A instruction can be written in different ways:

```
ADD A,7PH        (direct addressing)
ADD A,@R0        (indirect addressing)
ADD A,R7         (register addressing)
ADD A,#35H       (immediate addressing)
```

All arithmetic instructions execute one machine cycle except the INC DPTR instruction (two machine cycles) and the MUL AB and DIV AB instructions (four machine cycles). (Note that one machine cycle takes 1 μs if the 8051 is operating from a 12 MHz clock.)

EXAMPLE 3.11 The accumulator contains 63H, R3 contains 23H, and the PSW contains 00H. (a) What is the hexadecimal content of the accumulator and the PSW after execution of the following instruction?

```
ADD A,R3
```

(b) What is the content of the accumulator in decimal after execution of this instruction?

Solution

(a) ACC = 86H, PSW = 05H. (b) Decimal content of ACC = ? (see discussion).

Discussion

On the surface, this example seems straightforward: Given two values, add them and obtain the sum. However, there are some interesting conceptual issues that are important to understand. Let us begin by expressing the initial values of the ACC and R3 in decimal. By the usual method of conversion to decimal, A = 63H = 01100011B = 99_{10} and R3 = 23H = 00100011B = 35_{10}. So, the result of the addition is $99_{10} + 35_{10} = 134_{10}$. However, there is a problem. If we assume that a two's-complement signed notation is used, the largest positive number that can be expressed in 8 bits is $+127_{10}$. If an unsigned notation is used, the largest possible 8-bit value is $+255_{10}$ and, in this case, the final result of 134_{10} is perfectly OK.

Of course, the 8051 CPU has no special knowledge of whether the data are signed binary, unsigned binary, binary-coded decimal, ASCII, etc. Only you—the programmer—knows for sure. The mechanism to manage different formats of data is provided through the status bits in the PSW. To illustrate this, the addition is worked out below in binary.

```
11...11.
01100011 (ACC = 63H)
```

```
  +00100011 (R3  =  23H)
   10000110 (result stored in ACC = 86H)
```

The result is 10000110B = 86H. Note above that carries occurred out of bits 0, 1, 5, and 6. Carries did not occur out of bits 2, 3, 4, and 7. Because there was no carry out of bit 7, the C (carry) bit in the PSW is not set after the addition.

As for the OV (overflow) bit, the following description appears in Appendix C for the ADD instruction: "the OV bit is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared." This is a formal way of saying, "the OV bit is set if the result is out of range for 8-bit signed numbers." In this example, there was a carry out of bit 6 but not out of bit 7; therefore, the OV bit is set. This makes sense if the data are signed, because the allowable range is -128_{10} to $+127_{10}$ and the result of $99_{10} + 35_{10}$ is "out of range."

From the 8051 CPU's perspective, it is also possible that the data are binary-coded decimal (only the programmer knows for sure!), and it will set or clear the auxiliary carry bit accordingly. Since a carry did not occur out of bit 3, the AC bit is cleared.

Finally, the P bit in the PSW is set or cleared to establish even parity with the accumulator. Because the result in the ACC has three bits equal to one, the P bit is set, bringing the total to four - an even number. The final value in the PSW is 00000101B = 05H. Only the OV and P bits are set; the other bits are cleared (see Table 2-3).

The second question in this example is, "What is the content of the accumulator in decimal after execution of this instruction?" And it is here that we delve into the important concept of the meaning or interpretation of the data on which a CPU like the 8051 operates. Because the original problem did not indicate a format or representation for the original data, we cannot answer the question with certainty, hence the "?" in the solution. There are, however, at least two possible answers. First, if we assume the original data are in unsigned binary notation, then the result is "in range" (because C = 0) and the answer is 134_{10}. Second, if we assume the data are in signed binary notation using twos-complement notation, then the correct answer is "out of range" (because OV = 1). The important point is that the meaning or representation scheme in effect is not a characteristic of the CPU, but, rather, it is determined by the way the data are managed by the software.

EXAMPLE 3.12 Illustrate an instruction sequence to subtract the content of R6 from R7 and leave the result in R7.

Solution

```
MOV  A,R7
CLR  C
SUBB A,R6
MOV  R7,A
```

Discussion

For both addition and subtraction, the accumulator holds one of the values for the operation. So, the first instruction above moves one of the bytes to the accumulator to prepare for the operation. The second instruction clears the carry flag in the program status word. This is required because the only form of the subtract instruction is SUBB—subtract with borrow.

The operation subtracts from the accumulator the source byte and the carry bit. For subtract operations, the carry bit serves as a "borrow" bit. If the state of the carry bit is unknown, it must be explicitly cleared using CLR C before executing SUBB. The third instruction performs the subtraction, leaving the result in the accumulator. The fourth instruction moves the result into R7.

The 8051 provides powerful addressing of its internal memory space. Any location can be incremented or decremented using direct addressing without going through the accumulator. For example, if internal RAM location 7FH contains 40H, then the instruction

```
INC 7FH
```

increments this value, leaving 41H in location 7FH.

EXAMPLE 3.13 Suppose the 8051 did not have an instruction to directly increment an internal RAM location. How could this operation be achieved?

Solution

```
MOVE A,direct
INC  A
MOV direct,A
```

Discussion

The first instruction moves a byte of data from an internal RAM location to the accumulator. The second instruction increments the value read (which is now in the accumulator), and the third instruction writes the result back to internal RAM. Not only is this instruction sequence longer and slower than the single-instruction equivalent (INC direct), the previous value of the accumulator is lost.

One of the INC instructions operates on the 16-bit data pointer. Since the data pointer generates 16-bit addresses for external memory, incrementing it in one operation is a useful feature. Unfortunately, a decrement data pointer instruction is not provided and requires a sequence of instructions such as the following:

```
            DEC   DPL            ;DECREMENT LOW-BYTE OF DPTR
            MOV   R7,DPL         ;MOVE TO R7
            CJNE  R7,#0FFH,SKIP  ;IF UNDERFLOOR TO FF
            DEC   DPH            ;DECREMENT HIGH-BYTE TOO
SKIP: (continue)
```

The high- and low-bytes of the DPTR must be decremented separately; however, the high-byte (DPH) is only decremented if the low-byte (DPL) underflows from 00H to 0FFH.

The MUL AB instruction multiplies the accumulator by the data in the B register and puts the 16-bit product into the concatenated B (high-byte) and accumulator (low-byte) registers. DIV AB divides the accumulator by the data in the B register, leaving the 8-bit quotient in the accumulator and the 8-bit remainder in the B register. For example, if A contains 25 (19H) and B contains 6 (06H), the instruction

```
DIV AB
```

divides the content of A by the content of B. The A accumulator is left with the value 4 and the B accumulator is left with the value 1. (25 ÷ 6 = 4 with a remainder of 1.)

EXAMPLE 3.14 The accumulator contains 55H, the B register contains 22H, and the program status word contains 00H. What are the contents of these registers after execution of the following instruction?

```
MUL AB
```

Solution

```
ACC = 4AH, B = 0BH, PSW = 05H
```

Discussion

You might be able to compute the answer using a calculator; otherwise begin by expressing the original values in decimal: ACC = 55H = 85_{10} and B = 22H = 34_{10}. The product is 85_{10} x 34_{10} = $2{,}890_{10}$ = 0B4AH. The high byte (0BH) is placed in register B and the low byte (4AH) is placed in the accumulator. The P bit in the PSW is set to establish even parity with the accumulator. Since the result is greater than 255_{10}, the overflow flag is set, leaving the PSW containing 05H.

EXAMPLE 3.15 The accumulator contains 1FH. What is the largest value that could be in register B such that the OV bit would *not* be set after executing the following instruction?

```
MUL AB
```

Solution

08H

Discussion

The accumulator contains 1FH = 31_{10}. The overflow flag is set following MUL if the product is greater than 255_{10}. The largest number that 31_{10} can be multiplied by without the product exceeding 255_{10} is 8_{10} = 08H. (Note 31_{10} x 8_{10} = 248_{10}, 31_{10} x 9_{10} = 279_{10}.)

For BCD (binary-coded decimal) arithmetic, ADD and ADDC must be followed by a DA A (decimal adjust) operation to ensure the result is in range for BCD. Note that DAA will not convert a binary number to BCD; it produces a meaningful result only as the second step in the addition of two BCD bytes. For example, if A contains the BCD value 59 (59H), then the instruction sequence

```
ADD A,#1
DA  A
```

first adds 1 to A, leaving the result 5AH, then adjusts the result to the correct BCD value of 60 (60H). (59 + 1 = 60.)

EXAMPLE 3.16 Illustrate how to add two 4-digit binary-coded decimal numbers. The first is in internal memory locations 40H and 41H, and the second is in locations 42H and 43H. The most-significant digits are in locations 40H and 42H. Place the BCD result in locations 40H and 41H.

Solution

```
MOV  A,41H
ADD  A,42H
DA   A
MOV  42H,A
MOV  A,42H
ADDC A,40H
DA   A
MOV  40H,A
```

Discussion

This is an example of "multiprecision arithmetic," whereby a CPU is called upon to perform arithmetic operations on data that are larger than the CPU's natural data size. Even though the 8051's arithmetic instructions operate on bytes, it is possible to add or subtract larger data, for example, 16 bits or 32 bits. This is also true for binary-coded decimal numbers. To add two 4-digit numbers, two byte additions are required.

The trick in multiprecision arithmetic is in propagating carries from byte to byte. During addition, carries are naturally propagated from bit to bit within a byte, but from byte to byte, a carry—if it occurs—is temporarily held in the C bit in the PSW. Since a carry cannot occur into the low-order bytes, the first add instruction above is the generic "ADD." However, "ADDC" is required for the second add operation to include the carry that may have occurred from the low-order byte. Both add instructions are followed by DA A to perform the necessary adjustment for binary-coded decimal values.

3.3.2 Logical Instructions

The 8051 logical instructions (see Appendix A) perform Boolean operations (AND, OR, Exclusive OR, and NOT) on bytes of data on a bit-by-bit basis. If the accumulator contains 00110101B, then the following AND logical instruction

```
ANL A,#01010011B
```

leaves the accumulator holding 00010001B. This is illustrated below.

```
     01010022        (immediate data)
AND  00220102        (original value of A)
     --------
     00010002        (result in A)
```

Since the addressing modes for the logical instructions are the same as those for arithmetic instructions, the AND logical instruction can take several forms:

```
ANL A,55H          (direct addressing)
ANL A,@R0          (indirect addressing)
ANL A,R6           (register addressing)
ANL A,#33H         (immediate addressing)
```

All logical instructions using the accumulator as one of the operands execute in one machine cycle. The others take two machine cycles.

Logical operations can be performed on any byte in the internal data memory space without going through the accumulator. The "XRL direct,#data" instruction offers a quick and easy way to invert port bits, as in

```
XRL P1,#0FFH
```

This instruction performs a read-modify-write operation. The eight bits at Port 1 are read; then each bit read is exclusive ORed with the corresponding bit in the immediate data. Since the eight bits of immediate data are all 1s, the effect is to complement each bit read (e.g., $A \oplus 1 = \overline{A}$). The result is written back to Port 1.

The rotate instructions (RL A and RR A) shift the accumulator one bit to the left or right. For a left rotation, the MSB rolls into the LSB position. For a right rotation, the LSB rolls into the MSB position. The RLC A and RRC A variations are 9-bit rotates using the accumulator and the carry flag in the PSW. If, for example, the carry flag contains 1 and A contains 00H, then the instruction

```
RRC A
```

leaves the carry flag clear and A equal to 80H. The carry flag rotates into ACC.7 and ACC.0 rotates into the carry flag.

The SWAP A instruction exchanges the high and low nibbles within the accumulator. This is a useful operation in BCD manipulations. For example, if the accumulator contains a binary number that is known to be less than 100_{10}, it is quickly converted to BCD as follows:

```
MOV  B,#10
DIV  AB
SWAP A
ADD  A,B
```

Dividing the number by 10 in the first two instructions leaves the tens digit in the low nibble of the accumulator, and the ones digit in the B register. The SWAP and ADD instructions move the tens digit to the high nibble of the accumulator, and the ones digit to the low nibble.

EXAMPLE 3.17 Illustrate two ways to rotate the content of the accumulator three positions to the left Discuss the pros and cons of each method in terms of memory requirements and speed of execution.

Solution

```
(a) RL   A
    RL   A
    RL   A
(b) SWAP A
    RR   A
```

Discussion

Solution (a) is the most obvious approach, whereas solution (b) uses a trick. The SWAP A instruction swaps the low-order and high-order nibbles in the accumulator, and, as noted in Appendix C, this is equivalent to a 4-bit rotate operation. To undo the fourth rotation, a final rotate to the right is used.

Although the effect of solutions (a) and (b) is the same, they are slightly different in terms of memory usage and execution speed. All instructions above are 1-byte, 1-cycle instructions (see Appendix C), so solution (a) uses three bytes of memory and takes three CPU cycles to execute. Solution (b) uses only two bytes of memory and executes in two CPU cycles. The difference may seem minuscule, but in the world of embedded control systems, squeezing every last ounce out of a program is often a design requirement.

EXAMPLE 3.18 Write an instruction sequence to reverse the bits in the accumulator. Bit 7 and bit 0 are swapped, bit 6 and bit 1 are swapped, etc.

Solution

```
        MOV  R7,#8
LOOP:   RLC  A
        XCH  A,0F0H
        RRC  A
        XCH  A,0F0H
        DJNZ R7,LOOP
        XCH  A,0F0H
```

Discussion

The gymnastics required in this example are sometimes called "dancing with bits." The task may seem contrived, but it is amazing how often situations arise that necessitate fiddling with the position of bits within bytes.

The approach taken in the solution is to build the new value in the B register by successively shifting a bit out of the accumulator into the carry bit and then shifting the same bit back into the B register. To reverse the bit pattern, the first shift is "to the left" and the second shift is "to the right." Because the rotate instruction only operates on the accumulator, the B register and the accumulator are exchanged (XCH) following each rotate. A final XCH positions the correct result in the accumulator. Note: The B register is at direct address 0F0H.

3.3.3 Data Transfer Instructions

3.3.3.1 Internal RAM The instructions that move data within the internal memory spaces (see Appendix A) execute in either one or two machine cycles. The instruction format

```
MOV <destination>, <source>
```

allows data to be transferred between any two internal RAM or SFR locations without going through the accumulator. Remember, the upper 128 bytes of data RAM (8032/8052) are accessed only by indirect addressing, and the SFRs are accessed only by direct addressing.

A feature of the MCS-51^{TM} architecture differing from most microprocessors is that the stack resides in on-chip RAM and grows upward in memory, toward higher memory addresses. The PUSH instruction first increments the stack pointer (SP), then copies the byte into the stack. PUSH and POP use direct addressing to identify the byte being saved or restored, but the stack itself is accessed by indirect addressing using the SP register. This indirect addressing means the stack can use the upper 128 bytes of internal memory on the 8032/8052.

The upper 128 bytes of internal memory are not implemented in the 8031/8051 devices. With these devices, if the SP is advanced above 7FH (127), the PUSHed bytes are lost and the POPed bytes are indeterminate.

EXAMPLE 3.19 The stack pointer contains 07H, accumulator A contains 55H, and accumulator B contains 4AH. What internal RAM locations are altered, and what are their new values after executing the following instructions?

```
PUSH ACC
PUSH 0F0H
```

Solution

Address	Contents
08H	**55H**
09H	**4AH**
81H (SP)	**09H**

Discussion

The first instruction pushes the accumulator on the stack. The stack pointer is incremented before the push; so, the content of A (55H) is written to internal RAM location 08H. The second instruction pushes the B accumulator, which is located at internal RAM address 0F0H, on the stack. Again, the stack pointer is incremented before the push, so the content of B (4AH) is written to internal RAM location 09H. The stack pointer is incremented twice as a result of the two instructions, so its final value is 09H.

Data transfer instructions include a 16-bit MOV to initialize the data pointer (DPTR) for look-up tables in program memory, or for 16-bit external data memory accesses.

EXAMPLE 3.20 What is the address of the data pointer in internal RAM?

Solution

DPH is at address 83H. DPL is at address 82H.

Discussion

The data pointer (DPTR) is a 16-bit register occupying two locations in internal RAM. The high byte of the data pointer (DPH) is at address 83H, and the low byte (DPL) is at address 82H.

The instruction format

```
XCH A,<source>
```

causes the accumulator and the addressed byte to exchange data. An exchange "digit" instruction of the form

```
XCHD A,@Ri
```

is similar, but only the low-order nibbles are exchanged. For example, if A contains 0F3H, R1 contains 40H, and internal RAM address 40H contains 5BH, then the instruction

```
XCHD A,@Rl
```

leaves A containing 0FBH and internal RAM location 40H containing 53H.

EXAMPLE 3.21 What are the contents of the A and B accumulators after execution of the following instructions?

```
MOV   0F0H,#12H
MOV   R0,#0F0H
MOV   A,#34H
XCH   A,0F0H
XCHD  A,@R0
```

Solution

A = 14H, B = 32H

Discussion

The first three instructions set the stage for this example, finishing with A = 34H, B = 12H, and R0 = 0F0H. Recall that the B accumulator is located at internal RAM location 0F0H. The fourth instruction exchanges A and B, leaving A = 12H and B = 034H. The fifth instruction swaps the low-order digits of A and B, leaving A = 14H and B = 32H.

3.3.3.2 External RAM The data transfer instructions that move data between internal and external memory use indirect addressing. The indirect address is specified using a 1-byte address (@Ri, where Ri is either R0 or R1 of the selected register bank), or a 2-byte address (@DPTR). The disadvantage in using 16-bit addresses is that all eight bits of Port 2 are used as the high-byte of the address bus. This precludes the use of Port 2 as an I/O port. On the other hand, 8-bit addresses allow access to a few Kbytes of RAM, without sacrificing all of Port 2. (See Chapter 2, "Accessing External Memory.")

All data transfer instructions that operate on external memory execute in two machine cycles and use the accumulator as either the source or destination operand.

The read and write strobes to external RAM (RD and WR) are activated only during the execution of a MOVX instruction. Normally, these signals are inactive (high), and if external data memory is not used, they are available as dedicated I/O lines.

EXAMPLE 3.22 Illustrate an instruction sequence to read the content of external RAM locations 10F4H and 10F5H and place the values read in R6 and R7, respectively.

Solution

```
MOV  DPTR,#10F4H
MOVX A,@DPTR
```

```
MOV  R6,A
INC  DPTR
MOVX A,@DPTR
MOV  R7,A
```

Discussion

The first instruction initializes the data pointer with the first of the two external addresses to be read. The second instruction reads a byte from external memory location 10F4H and places it in the accumulator. The third instruction transfers the byte read to register 6. Note that the MOVX instruction must use the accumulator as either the source or destination of data. The fourth instruction increments the data pointer, leaving it pointing to the second of the two external addresses to be read. The fifth instruction reads a byte from external memory location 10F4H and places it in the accumulator. The sixth instruction transfers the byte read to register 7.

3.3.3.3 Look-Up Tables Two data transfer instructions are available for reading look-up tables in program memory. Since they access program memory, the look-up tables can only be read, not updated. The mnemonic is MOVC for "move constant." MOVC uses either the program counter or the data pointer as the base register and the accumulator as the offset.

The instruction

```
MOVC A,@A+DPTR
```

can accommodate a table of 246 entries, numbered 0 through 255. The number of the desired entry is loaded into the accumulator and the data pointer is initialized to the beginning of the table. The instruction

```
MOVC A,@A+PC
```

works the same way, except the program counter is the base address. The table is usually accessed through a subroutine. First, the number of the desired entry is loaded into the accumulator, and then the subroutine is called. The setup and call sequence would be coded as follows:

```
        MOV  A,#ENTRY_NUMBER
        CALL LOOKUP
             .
             .
             .
LOOKUP: INC  A
        MOVC A,@A+PC
        RET
TABLE:  DB   data, data, data, data, ...
```

The table immediately follows the **RET** instruction in program memory. The **INC** instruction is needed because the PC points to the **RET** instruction when **MOVC** executes. Incrementing the accumulator will effectively bypass the **RET** instruction when the table look-up takes place.

Note however that even though we would expect to get a table of 256 entries when using this technique, we only get 255 entries in this case since we lose one entry due to the effect of the **INC A** instruction. Consider the entry number of 255 that is moved into the accumulator. The **INC A** instruction increments it by 1, causing the value 255 to recycle back to 0 since the accumulator is 8 bits in size. The next instruction, **`MOVC A,@A+PC`**, would attempt to load the accumulator with the value of **`RET`**, which is invalid. Therefore, valid entries are only from 0 to 254.

EXAMPLE 3.23

Write a subroutine called SQUARE to compute the square of an integer between 0 and 9. Enter the subroutine with the integer in A,and return with the square of the integer in A. Write two versions of the subroutine: (a) using a look-up table, and (b) without using a lookup table. Then (c) illustrate a call sequence to convert 6 to its square, 36.

Solution

a. Using a look-up table:

```
SQUARE:  INC   A
         MOVC  A,@A+PC
         RET
TABLE:   DB    0,1,4,9,16,25,36,49,64,81
```

b. Without using a look-up table:

```
SQUARE:  PUSH  0F0H
         MOV   0F0H,A
         MUL   AB
         POP   0F0H
         RET
```

c. Call sequence:

```
         MOV A,#6
         CALL SQUARE
```

Discussion

The solution in (a) and the call sequence in (c) are straightforward implementations of a lookup table in program memory. For this example, however, there is an interesting alternative approach, as shown in (b). By copying the content of A to the B accumulator (in internal RAM location 0FFH), and then multiplying B times A using MUL AB, the square is computed and left in A. Since the B accumulator is overwritten by the second instruction, it is first saved on the stack and then restored from the stack before returning from the subroutine. The PUSH/POP instructions may not be needed, depending on the context; however, it is good programming practice to design subroutines to have as few side effects as possible.

In comparing solutions (a) and (b), we find an interesting tradeoff. Solution (a) is 13 bytes, including the look-up table, whereas solution (b) is only eight bytes. However, solution (a) executes in five CPU cycles, whereas solution (b) executes in 11 cycles. The tradeoff is that solution (a) is faster (which is good!) but consumes more memory (which is bad!), compared with solution (b), which is slower (which is bad!) but consumes less memory (which is good!). This tradeoff would be more pronounced if the look-up table was large, because every entry adds one byte to the size of the routine.

In many situations, the relationship between the index to a table and the entries within a table is not as simple as in this example, and using a look-up table is the only viable implementation.

3.3.4 Boolean Instructions

The 8051 processor contains a complete Boolean processor for single-bit operations. The internal RAM contains 128 addressable bits, and the SFR space supports up to 128 other addressable bits. All port lines are bit-addressable, and each can be treated as a separate single-bit port. The instructions that access these bits are not only conditional branches but also a complete repertoire of move, set, clear, complement, OR, and AND instructions. Such bit operations—one of the most powerful features of the MCS-51^{TM} family of micro-controllersare not easily obtained in other architectures with byte-oriented operations.

The available Boolean instructions are shown in Appendix A. All bit accesses use direct addressing with bit addresses 00H-7FH in the lower 128 locations, and bit addresses 80H-0FFH in the SFR space. Those in the lower 128 locations at byte addresses 20H-2FH are numbered sequentially from bit 0 of address 20H (bit 0FH) to bit 7 of address 2FH (bit 7FH).

Bits may be set or cleared in a single instruction. Single-bit control is common for many I/O devices, including output to relays, motors, solenoids, status LEDs, buzzers, alarms, loudspeakers, or input from a variety of switches or status indicators. If an alarm is connected to Port 1 bit 7, for example, it might be turned on by setting the port bit

```
SETB P1.7
```

and turned off by clearing the port bit

```
CLR Pl.7
```

The assembler will do the necessary conversion of the symbol “P1.7" into the correct bit address, 97H.

Note how easily an internal flag can be moved to a port pin:

```
MOV C,FLAG
MOV P1.0,C
```

In this example, FLAG is the name of any addressable bit in the lower 128 locations or the SFR space. An I/O line (the LSB of Port 1, in this case) is set or cleared, depending on whether the flag bit is 1 or 0.

The carry bit in the program status word (PSW) is used as the single-bit accumulator of the Boolean processor. Bit instructions that refer to the carry bit as "C" assemble as carry-specific instructions (e.g., CLR C). The carry bit also has a direct address, since it resides in the PSW register, which is bit-addressable. Like other bit-addressable SFRs, the PSW bits have predefined mnemonics that the assembler will accept in lieu of the bit address. The carry flag mnemonic is "CY," which is defined as bit address 0D7H. Consider the following two instructions:

```
CLR C
CLR CY
```

Both have the same effect; however, the former is a 1-byte instruction, whereas the latter is a 2-byte instruction. In the latter case, the second byte is the direct address of the specified bit—the carry flag.

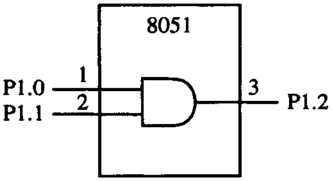


FIGURE 3-4
Simple implementation of a logical AND operation

Logical operations such as AND, OR, NAND, NOR, and NOT are easy to implement for single-bit variables. Furthermore, the variables may be input or output signals on the 8051's I/O ports. Thus, the state of an I/O pin may be directly read or written accompanied by a Boolean operation. Suppose one wanted to compute the logical AND of the input signals on bit 0 and bit 1 of Port 1 and output the result to bit 2 of Port 1. This logical relationship is illustrated in Figure 3-4.

The following instruction sequence creates this logical operation:

```
LOOP:     MOV  C,P1.0            ;n1 = 1 cycle
          ANL  C,P1.1            ;n2 = 2 cycles
          MOV  P1.2,C            ;n3 = 2 cycles
          SJMP LOOP              ;n4 = 2 cycles
```

Bits P1.0 and P1.1 are continuously read with the logical AND of the result continuously presented on P1.2.

If the logic operation in Figure 3-4 is implemented using a typical electronic logic circuit, such as a 74AL508, then the input-to-output propagation delay is on the order of 7 ns. This is the time from an input signal transition to the correct logic level appearing at the output. What is the worst-case propagation delay for the circuit in Figure 3-4 if implemented using the software above? Figure 3-5 will assist in answering this question. The worst-case scenario is that P1.0 changes just after the first instruction, shown at "A" in the figure. The change is not sensed until the next pass through the loop. The correct output state appears at "B" in the figure. The solid line in Figure 3-5 illustrates the sequence of instructions for the worst-case scenario. The number of CPU cycles from "A" to "B," following the solid line, is 11. At 12 MHz operation the worst-case delay is 11 µs. Obviously there is "no contest" in comparing the speed of a microcontroller with speed of electronic logic circuits. The software implementation for Figure 3-4 is about a thousand times slower than a 74LS08 AND gate!

Note that the Boolean instructions include ANL (AND logical) and ORL (OR logical) operations, but not the XRL (exclusive OR logical) operation. An XRL operation is simple

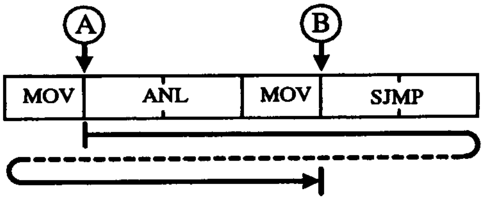


FIGURE 3-5
Instruction sequence for worst-case propagation delay

to implement. Suppose, for example, it is required to form the exclusive OR of two bits, BIT1 and BIT2, and leave the result in the carry flag. The instructions are shown below.

```
          MOV BIT1
          JNB BIT2,SKIP
          CPL C
SKIP:     (continue)
```

First, BIT1 is moved to the carry flag. If BIT2 = 0, then C contains the correct result; that is, BIT1 ⊕ BIT2 = BIT1 if BIT2 = 0. If BIT2 = 1, C contains the complement of the correct result. Complementing C completes the operation.

3.3.4.1 Bit Testing The code in the example above uses the JNB instruction, one of a series of bit-test instructions that jump if the addressed bit is set (JC, JB, JBC) or if the addressed bit is not set (JNC, JNB). In the above case, if BIT2 = 0 the CPL instruction is skipped. JBC (jump if bit set then clear bit) executes the jump if the addressed bit is set, and also clears the bit; thus, a flag can be tested and cleared in a single instruction.

All PSW bits are directly addressable, so the parity bit or the general purpose flags, for example, are also available for bit-test instructions.

3.3.5 Program Branching Instructions

As evident in Appendix A, there are numerous instructions to control the flow of programs, including those that call and return from subroutines or branch conditionally or unconditionally. These possibilities are enhanced further by the three addressing modes for the program branching instructions.

There are three variations of the JMP instruction: SJMP, LJMP, and AJMP (using relative, long, and absolute addressing, respectively). Intel's assembler (ASM51) allows the use of the generic JMP mnemonic if the programmer does not care which variation is encoded. Assemblers from other companies may not offer this feature. The generic JMP assembles to AJMP if the destination contains no forward reference and is within the same 2K page (as the instruction following the AJMP). Otherwise, it assembles to LJMP. The geneneric CALL instruction (see below) works the same way.

The SJMP instruction specifies the destination address as a relative offset, as shown in the earlier discussion on addressing modes. Since the instruction is two bytes long (an opcode plus a relative offset), the jump distance is limited to -128 to +127 bytes relative to the address following the SJMP.

The LJMP instruction specifies the destination address as a 16-bit constant. Since the instruction is three bytes long (an opcode plus two address bytes), the destination address can be anywhere in the 64K program memory space.

The AJMP instruction specifies the destination address as an 11-bit constant. As with SJMP, this instruction is two bytes long, but the encoding is different. The opeode contains three of the 11 address bits, and byte 2 holds the low-order eight bits of the destination address. When the instruction is executed, these 11 bits replace the low-order 11 bits in the PC, and the high-order five bits in the PC stay the same. The destination, therefore, must be within the same 2K block as the instruction following the AJMP. Since there is 64K of code memory space, there are 32 such blocks, each beginning at a 2K address boundary (0000H, 0800H, 1000H, 1800H, etc., up to 0F800H; see Figure 3-3).

In all cases the programmer specifies the destination address to the assembler in the usual way - as a label or as a 16-bit constant. The assembler will put the destination address into the correct format for the given instruction. If the format required by the instruction will not support the distance to the specified destination address, a "destination out of range" message is given.

3.3.5.1 Jump Tables The JMP @A+DPTR instruction supports case-dependent jumps for jump tables. The destination address is computed at execution time as the sum of the 16-bit DPTR register and the accumulator. Typically the DPTR is loaded with the address of a jump table, and the accumulator acts as an index. If, for example, five "cases" are desired, a value from 0 through 4 is loaded into the accumulator and a jump to the appropriate case is performed as follows:

```
MOV  DPTR,#JUMP_TABLE
MOV  A,#INDEX_NUMBER
RL   A
JMP  @A+DPTR
```

The RL A instruction above converts the index number (0 through 4) to an even number in the range 0 through 8 because each entry in the jump table is a 2-byte address:

```
JUMP_TABLE:  AJMP CASE0
             AJMP CASE1
             AJMP CASE2
             AJMP CASE3
```

EXAMPLE 3.24 Suppose the jump table above begins at code memory location 8100H with the following memory assignments:

Address	Content
8100	0l
810l	B8
8102	0l
8103	43
8104	41
8105	76
8106	El
8107	F0

a. What is the beginning and ending address of the 2K block of code memory within which these instructions reside?

b. At what addresses do CASE0 through CASE3 begin?

Solution

a. 8000H to 87FFH

b. CASE0 begins at address 80B8H
CASE1 begins at address 8043H

CASE2 begins at address 8276H
CASE3 begins at address 87F0H

Discussion

This example has more to do with absolute addressing than with jump tables. The jump table consists of a series of four ACALL instructions. Since this instruction uses absolute addressing, the jump destinations must be within the same 2K page as the ACALL instructions. A 2K page is a block of code memory with the upper five address bits the same. The upper five bits in 8100H are 1000B. The other 11 address bits for each CASE routine consist of the upper three bits in the ACALL opcode and the second byte of the ACALL instruction. Consider ACALL CASE3, which is located in locations 8106H (21H) and 8107H (0F0H). The upper three bits of the opcode are 111B and the second byte of the instruction is 11110000B. Concatenating the three binary patterns in the order just given yields 10000111 11110000B = 87F0H, the address of the CASE3 routine.

3.3.5.2 Subroutines and Interrupts There are two variations of the CALL instruction: ACALL and LCALL, using absolute and long addressing, respectively. As with JMP, the generic CALL mnemonic may be used with Intel's assembler if the programmer does not care which way the address is encoded. Either instruction pushes the content of the program counter on the stack and loads the program counter with the address specified in the instruction. Note that the PC will contain the address of the instruction *following* the CALL instruction when it gets pushed on the stack. The PC is pushed on the stack low-byte first, high-byte second. The bytes are popped from the stack in the reverse order. For example, if an LCALL instruction is in code memory at locations 1000H-1002H and the SP contains 20H, then LCALL (a) pushes the return address (1003H) on the internal stack, placing 03H in 21H and 10H in 22H; (b) leaves the SP containing 22H; and (c) jumps to the subroutine by loading the PC with the address contained in bytes 2 and 3 of the instruction.

EXAMPLE 3.25 The following instruction

```
LCALL COSINE
```

is in code memory at addresses 0204H through 0206H, and the subroutine COSINE begins at code memory address 043AH. Assume the stack pointer contains 3AH just before this instruction executes. What internal RAM locations are altered, and what are their new values after execution of the LCALL instruction?

Solution

Address	Contents
3BH	07H
3CH	02H
81H (SP)	3CH

Discussion

Since the LCALL instruction is three bytes long, the instruction following it is at code memory location 0207H. This is the address to which the program must return at the end of the subroutine, and it is the address that is saved on the stack before branching to the subroutine.

Since the stack pointer contains 3AH initially, and, since it is incremented before writing to the stack, the low byte of the return address (02H) is saved at internal memory location 3BH, and the high byte of the return address (02H) is saved at internal memory location 3CH. Since two bytes are saved on the stack, the SP is incremented twice and contains 3CH after the LCALL instruction executes. Note that the stack pointer is a special function register located at internal RAM address 81H.

The LCALL and ACALL instructions have the same restrictions on the destination address as the LJMP and AJMP instructions just discussed.

Subroutines should end with an RET instruction, which returns execution to the instruction following the CALL. There is nothing magical about the way the RET instruction gets back to the main program. It simply "pops" the last two bytes off the stack and places them in the program counter. It is a cardinal rule of programming with subroutines that they should always be entered with a CALL instruction, and they should always be exited with a RET instruction. Jumping in or out of a subroutine any other way usually fouls up the stack and causes the program to crash.

RETI is used to return from an interrupt service routine (ISR). The only difference between RET and RETI is that RETI signals the interrupt control system that the interrupt in progress is done. If there is no interrupt pending at the time RETI is executed, then RETI is functionally identical to RET. Interrupts and the RETI instruction are discussed in more detail in Chapter 6.

EXAMPLE 3.26

The stack pointer contains 1CH just prior to the execution of

```
RET
```

at the end of a subroutine. What is the value of the stack pointer after execution of this instruction?

Solution

1CH

Discussion

The return address for the subroutine is 16 bits or two bytes. The purpose of the RET instruction is to retrieve two bytes from the stack and place them in the program counter, thus allowing the program to continue executing at the location following the call instruction that leads to the subroutine. Regardless of what address this may be, the stack pointer contains two less than its value before RET executed.

EXAMPLE 3.27

XORing Bits

The 8051 instruction set does not include an instruction to XOR two bit values. Write a subroutine XRB that effectively behaves as an instruction to XOR two bits in the form XRB C, P. This means that the two bit values are stored in C and P, respectively, prior to the calling of the subroutine, and the XORed result should be put back in C.

Solution

```
XRB:     MOV 20H,C        ;backup the first bit, x
         ANL C,/P         ;C = x̅y
         MOV 21H,C        ;backup partial result, x̅y
         MOV C,P          ;put second bit, y in C
         ANL C,/20H       ;C = y̅x
         ORL C,21H        ;C = y̅x + x̅y
```

Discussion

There are three approaches to solve this problem. In this example, we use the fact that x⊕y = $\bar{x}y + \bar{y}x$. The other two cases are left as an exercise for the reader in the problems section:

- Change each bit into a byte and then do XOR on the bytes
- Use JB and JNB.

3.3.5.3 Conditional Jumps The 8051 offers a variety of conditional jump instructions. All of these specify the destination address using relative addressing and so are limited to a jump distance of -128 to +127 bytes from the instruction following the conditional jump instruction. Note, however, that the user specifies the destination address the same way as with the other jumps, as a label or 16-bit constant. The assembler does the rest.

There is no 0-bit in the PSW. The JZ and JNZ instructions test the accumulator data for that condition.

The DJNZ instruction (decrement and jump if not zero) is for loop control. To execute a loop *N* times, load a counter byte with *N* and terminate the loop with a DJNZ to the beginning of the loop, as shown below for *N* = 10.

```
         MOV R7,#10
LOOP:    (begin loop)
               .
               .
               .
         (end loop)
         DJNZ R7,LOOP
         (continue)
```

The CJNE instruction (compare and jump if not equal) is also used for loop control. Two bytes are specified in the operand field of the instruction, and the jump is executed only if the two bytes are not equal. If, for example, a character has just been read into the accumulator from the serial port and it is desired to jump to an instruction identified by the label TERMINATE if the character is CONTROL-C (03H), then the following instructions could be used:

```
         CJNE A,#03H,SKIP
         SJMP TERMINATE
SKIP:    (continue)
```

Since the jump occurs only if A ≠ CONTROL-C, a skip is used to bypass the terminating jump instruction except when the desired code is read.

Another application of this instruction is in "greater than" or "less than" comparisons. The two bytes in the operand field are taken as unsigned integers. If the first is less than the second, the carry flag is set. If the first is greater than or equal to the second, the carry flag is cleared. For example, if it is desired to jump to BIG if the value in the accumulator is greater than or equal to 20H, the following instructions could be used:

```
CJNE A,#20H,$+3
JNC   BIG
```

The jump destination for CJNE is specified as "$+3." The dollar sign ($) is a special assembler symbol representing the address of the current instruction. Since CJNE is a 3-byte instruction, "$+3" is the address of the next instruction, JNC. In other words, the CJNE instruction follows through to the JNC instruction *regardless* of the result of the compare. The sole purpose of the compare is to set or clear the carry flag. The JNC instruction decides whether or not the jump takes place. This example is one instance in which the 8051 approach to a common programming situation is more awkward than with most microprocessors; however, as we shall see in Chapter 7, the use of macros allows powerful instruction sequences, such as the example above, to be constructed and executed using a single mnemonic.

SUMMARY

This chapter has presented the 8051 instruction set. The reader is encouraged to refer to Appendix C for more examples of using this instruction set. 0f course, nothing beats practice so the reader should really try out as many programming examples as possible. The next three chapters will discuss more programming examples in terms of interacting with the 8051's on-chip peripherals: the timers, the serial port, and the interrupts.

PROBLEMS

3.1 What are the hexadecimal bytes for the following instructions?

a. `INC DPTR`
b. `MOV  A,#-2`
c. `MOVX @DPTR,A`
d. `CJNE A,#0DH,$+3`
e. `PUSH ACC`
f. `SETB P2.2`

3.2 What are the hexadecimal bytes for the following instructions?

a. `MOV DPH,#84H`
b. `JNB ACC.0,$`
c. `POP DPH`
d. `MOV A,#'='`

e. `XLR    A,#'S'`
f. `CLR    C`

3.3 What instructions are represented by the following machine language bytes?

a. **07EH, 002H**
b. **0C2H, 097H**
c. **013H**
d. **0F6H**
e. **022H**
f. **090H, 080H, 030H**

3.4 What instructions are represented by the following machine language bytes?

a. **0EFH**
b. **012H, 080H, 050H**
c. **0F5H, 08DH**
d. **004H**
e. **083H**
f. **075H, 0BAH, 0E7H**

3.5 List all the 8051's 3-byte instructions with an opcode ending in 5H.

3.6 List all the 8031's 2-byte instructions beginning with 2H.

3.7 Illustrate how the content of internal address 50H could be transferred to the accumulator, using indirect addressing.

3.8 Illustrate two ways to transfer the content of the accumulator to internal RAM address 3CH.

3.9 What opcode is undefined on the 8051?

3.10 How many opcodes are defined on the 8052?

3.11 The following is an 8051 instruction:

```
MOV    50H,#0FFH
```

a. What is the opcode for this instruction?
b. How many bytes long is this instruction?
c. Explain the purpose of each byte of this instruction.
d. How many machine cycles are required to execute this instruction?
e. If an 8051 is operating from a 16 MHz crystal, how long does this instruction take to execute?

3.12 The following is an 8051 instruction:

```
CJNE A,#'Q',AHEAD
```

a. What is the opcode for this instruction?
b. How many bytes long is this instruction?
c. Explain the purpose of each byte of this instruction.
d. How many machine cycles are required to execute this instruction?
e. If an 8051 is operating from a 10 MHz crystal, how long does this instruction take to execute?

3.13 What is the relative offset for the instruction

```
SJMP AHEAD
```

if the instruction is in locations 0400H and 0401H, and the label AHEAD represents the instruction at address 041FH?

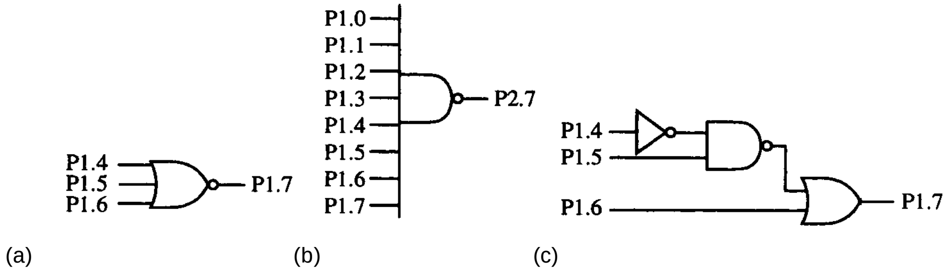


FIGURE 3-6

Logic gate programming problems. (a) 3-input NOR. (b) 8-input NAND. (c) 3-gate logic operation.

3.26 Assume the PSW contains 0C0H and accumulator A contains 40H just before the following instruction executes:

```
RLC A
```

What is the content of accumulator A after this instruction executes?

3.27 Assume the PSW contains 78H and the accumulator contains 81H. What is the content of the accumulator after the following instruction executes?

```
RRC A
```

3.28 What instruction sequence could be used to create a 5 μs low-going pulse on P1.7? Assume P1.7 is high initially and the 8041 is operating from a 12 MHz crystal.

3.29 Write a program to create an 83.3 kHz square wave on P1.0. (Assume 12 MHz operation.)

3.30 Write a program to generate a 4 μs active-high pulse on P1.7 every 200 μs.

3.31 Write programs to implement the logic operations shown in Figure 3-6. For each program, what is the worst-case propagation delay from an input transition to an out-put transition? Assume 12 MHz operation.

3.32 Write programs to implement the logic operations shown in Figure 3-7. For each program, what is the worst-case propagation delay from an input transition to an output transition? Assume 12 MHz operation.

3.33 What is the content of accumulator A after the following instruction sequence executes?

```
MOV A,#7FH
MOV 50H,#29H
```

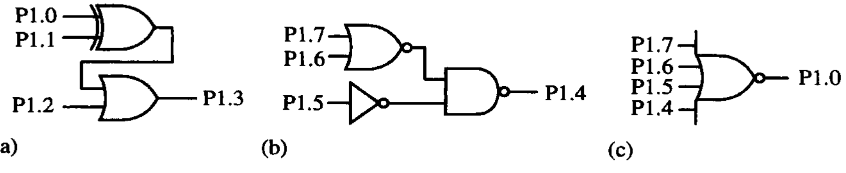


FIGURE 3-7

Logic gate programming problems. (a) 2-gate logic circuit. (b) 3-gate logic circuit.(c) 4-input NOR.

```
SJMP BACK
```

if the instruction is in locations 0A051H and 0A041H, and the label BACK represents the instruction at address 9FE0H?

3.15 Assume the instruction

```
AJMP AHEAD
```

is in code memory at addresses 2FF0H and 2FF1H, and the label AHEAD corresponds to an instruction at address 2F96H. What are the hexadecimal machine language bytes for this instruction?

3.16 Assume the instruction

```
ACALL FACTORIAL
```

is in code memory at locations 06F5H and 06F5H, and the label FACTORIAL corresponds to a subroutine beginning at address 07ABH. What are the machine language bytes for this instruction?

3.17 At a certain point in a program, it is desired to jump to the label EXIT if the accumulator equals the carriage return ASCII code. What instruction(s) would be used?

3.18 At a certain point in a program, it is desired to jump to the label EXIT if the accumulator equals the ASCII code for 'Q' or 'q,' or continue on otherwise. What instruction(s) would be used?

3.19 The instruction

```
SJMP BACK
```

is in code memory at address 0100H and 0101H and the label BACK corresponds to an instruction at address 00AEH. What are the hexadecimal machine language bytes for this instruction?

3.20 The instruction

```
CJNE R7,#'Z',NOTZED
```

is in code memory at addresses 022AH through 022CH. What are the machine language bytes for this instruction?

3.21 What does the following instruction do?

```
SETH 0D7H
```

What is a better way to perform the same operation? Why?

3.22 What is the difference between the following two instructions?

```
INC A
INC ACC
```

3.23 What are the machine language bytes for the instruction

```
LJMP ONWARD
```

if the label ONWARD represents the instruction at address 0A0F6H?

3.24 Assume accumulator A contains 4AH. What is the result in accumulator A after the following instruction executes?

```
XRL A,#0FFH
```

3.25 Assume the accumulator contains 29H. What is the content of the accumulator after the following instruction executes?

```
ORL A,#47H
```

```
MOV R0,#20H
XCHD A, @R0
```

3.34 What are the machine-language bytes for the following instruction?

```
SETB P2.6
```

3.35 What instruction sequence could be used to copy Flag 0 in the PSW to the port pin P1.5?

3.36 Under what circumstances will Intel's assembler (ASM51) convert a generic JMP instruction to LJMP?

3.37 The 8051 internal memory is initialized as follows immediately prior to the execution of an RET instruction:

Internal Address	Contents	SFRs	Contents
0B	9A	SP	0B
0A	78	PC	0200
09	56	A	55
08	34		
07	12		

What is the content of the PC after the RET instruction executes?

3.38 An 8051 subroutine is shown below:

```
SUB:   MOV  R0,#20H
LOOP:  MOV  @R0,#0
       INC  R0
       CJNE R0,#80H,LOOP
       RET
```

a. What does this subroutine do?

b. In how many machine cycles does each instruction execute?

c. How many bytes long is each instruction?

d. Convert the subroutine to machine language.

e. How long does this` subroutine take to execute? (Assume 12 MHz operation.)

3.39 A 4-bit DIP switch and a common-anode 7-segment LED are connected to an 8051 as shown in Figure 3-8. Write a program that continually reads a 4-bit code from the

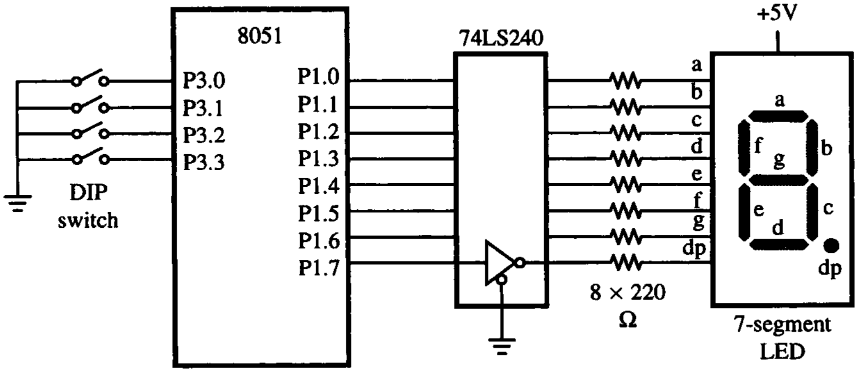


FIGURE 3-8
Interface to a DIP switch and 7-segment LED

DIP switch and updates the LEDs to display the appropriate hexadecimal character. For example, if the code 1100B is read, the hexadecimal character "C" should appear; thus, segments a through g, respectively, should be ON, OFF, OFF, ON, ON, ON, and OFF. Note that setting an 8041 port pin to"1" turns the corresponding segment "ON." (See Figure 3-8.)

3.40 What type of move instruction would you use to move a value into:

a. internal data memory
b. internal code memory
c. external data memory
d. external code memory

3.41 What is a look-up table? What are the advantages of using look-up tables?

3.42 What is the difference between the two instructions below? Describe how each one works in detail.

```
MOV    A,@R0
MOV    A,R0
```

3.43 For the assembly language program given below, find the memory locations affected, and what are the final contents of these affected locations.

```
        MOV    R0,#10H
REP:    MOV    @R0,#55H
        INC    R0
        CJNE   R0,#20H,REP
        MOV    R1,#00H
LOOP:   MOV    @R1,#AAH
        DEC    R1
        CJNE   R1,#5FH,LOOP
END
```

3.44 For the following assembly language program, find the memory locations affected, and what are the new values.

```
        MOV    R0,#7FH
LOOP:   MOV    @R0,#7FH
        DEC    R0
        CJNE   R0,#20H,LOOP
        MOV    R1,#00H
NEXT:   MOV    @R1,#00H
        INC    R1
        CJNE   R1,#5FH,NEXT
        END
```

3.45 Suppose the following memory locations contain the initial values as follows:

30H	55H
78H	00H
7FH	FFH

What are the contents of the memory locations 30H, 78H and 7FH after the execution of each instruction below? Why? (Assume each instruction is independent of the other.)

a. `CPL 7FH`

b. `CLR 78H`

c. `MOV 7FH,78H`

3.46 Write an assembly language program to add all the numbers in internal RAM locations starting from 30H to 6FH. Store the result in internal RAM location 70H. Write comments for each line of code.

3.47 The factorial operation, denoted by the symbol 1' is often found in mathematics, especially in calculations of probability. For example, 5! = 5 x 4 x 3 x 2 x 1. Write an assembly language program to calculate the factorial of a number that is stored in RAM location 44H. Put the result of the calculation into RAM location 77H.

3.48 The accumulator A contains a 5-bit number, x (excluding the number 0). Write a program to use a look-up table to find the result of 20 log10 x. Round your precomputations to integers.

3.49 Write an assembly language program using look-up tables to calculate the exponential function ex of a value x in the accumulator. The result (rounded off to the nearest integer) should be returned in R1 (high byte) and R0 (low byte). For example, if you have the value 2 in the accumulator, your result should be R1 = 0 and R0 = 7.

3.50 Write an assembly language program to perform the multiplication of two numbers in R0 and R1. DO NOT USE the MUL AB instruction! Instead, use other instructions to do so. The higher-order byte of the result should be put in R3 while the lower-order byte of the result should be put in R2.

3.51 What are subroutines? Explain the advantages of using subroutines in your assembly language programs.

3.52 The term nested subroutines refers to the use of two or more subroutines, each one called within another.

a. Write a subroutine called POW that calculates the result of the number in theaccumulator A raised to the power of a number stored in the B register. For example, if A = 2 and B = 3, then your result should be 23 = 8. Return the 16-bit answer in the accumulator, A (for the low byte) and B register (for the high byte).

b. Then, write another subroutine called CALCULATE that calculates the value of 3^4 - 2^3 and returns the 16-bit answer in A (low byte) and B (high byte). Your subroutine should call the POW subroutine to do the exponentiations (raising a number to the power of another number).

3.53 Write a subroutine called SUM that calculates the sum of two numbers in the accumulator A and the B register, and returns the answer in the accumulator, A. Then, write another subroutine called TOTAL that calculates the sum of three numbers in R1, R2, and R3. Do not use the ADD instruction in the TOTAL subroutine. Instead, you should call the SUM subroutine to do the additions. Return the total sum in R5.

3.54 The 8051 provides the XCHD instruction to exchange the lower-order nibbles of two values. Write a subroutine XCHH that exchanges the higher-order nibbles of the two values in A and B.

3.55 Write a subroutine XRB to compute the XOR of two bit values in C and P. The result should be stored in C. Write comments for each line of code.

3.56 The subroutine GUESSME below can be used to perform a very useful operation that is not provided by the 8051's built-in instructions.

a. Write the comments for each line of code.

b. Explain what you think the subroutine is used for.

```
GUESSME:  CLR C
          RRC  A
          DJNZ R0,GUESSME END
```

3.57 Write an assembly language program to perform division of two numbers in R0 and R1. The quotient should be stored in A and the remainder in B. DO NOT USE the DIV AB instruction!

3.58 Example 3.27 considered a subroutine XRB that effectively behaves as an instruction to XOR two bits in the form XRB C, P. This means that the two bit values are stored in C and P prior to the calling of the subroutine, and the X0Red result should be put back in C. Rewrite the subroutine to XOR two bits based on the following methods:

a. Change each bit into a byte and then do XOR on the bytes

b. Use JB and JNB instructions

3.59 Internal memory locations 30H to 39H contain the numbers 0 to 9, respectively. Write the assembly language instructions to reverse the order in which the numbers are stored: 0 is put in 39H, 1 in 38H, etc.

(Hint: Use PUSH and POP instructions.)

3.60 Write the assembly language instructions to add two 16-bit numbers.

CHAPTER 4

Timer Operation

4.1 INTRODUCTION

In this chapter we examine the 8051's on-chip timers. We begin with a simplified view of timers as they are commonly used with microprocessors or microcontrollers.

A timer is a series of divide-by-2 flip-flops that receive an input signal as a clocking source. The clock is applied to the first flip-flop, which divides the clock frequency by 2. The output of the first flip-flop clocks the second flip-flop, which also divides by 2, and so on. Since each successive stage divides by 2, a timer with ***n*** stages divides the input clock frequency by 2^n. The output of the last stage clocks a timer overflow flip-flop, or ***flag,*** which is tested by software or generates an interrupt. The binary value in the timer flip-flops can be thought of as a "count" of the number of clock pulses (or "events") since the timer was started. A 16-bit timer, for example, would count from 0000H-to-0FFFFH. The overflow flag is set on the 0FFFFH-to-0000H overflow of the count.

The operation of a simple timer is illustrated in Figure 4-1 for a 3-bit timer. Each stage is shown as a type-D negative-edge-triggered flip-flop operating in divide-by-2 mode (i.e., the Q output connects to the D input). The flag flip-flop is simply a type-D latch, set by the last stage in the timer. It is evident in the timing diagram in Figure 4-1b that the first stage (Q_0) toggles at 1/2 the clock frequency, the second stage at 1/4 the clock frequency, and so on. The count is shown in decimal and is easily verified by examining the state of the three flip-flops. For example, the count "4" occurs when $Q_2 = 1$, $Q_1 = 0$, and $Q_o = 0$ ($4_{10} = 100_2$).

Timers are used in virtually all control-oriented applications, and the 8051 timers are no exception. There are two 16-bit timers each with four modes of operation. A third 16-bit timer with three modes of operation is added on the 8052. The timers are used for (a) interval timing, (b) event counting, or (c) baud rate generation for the built-in serial

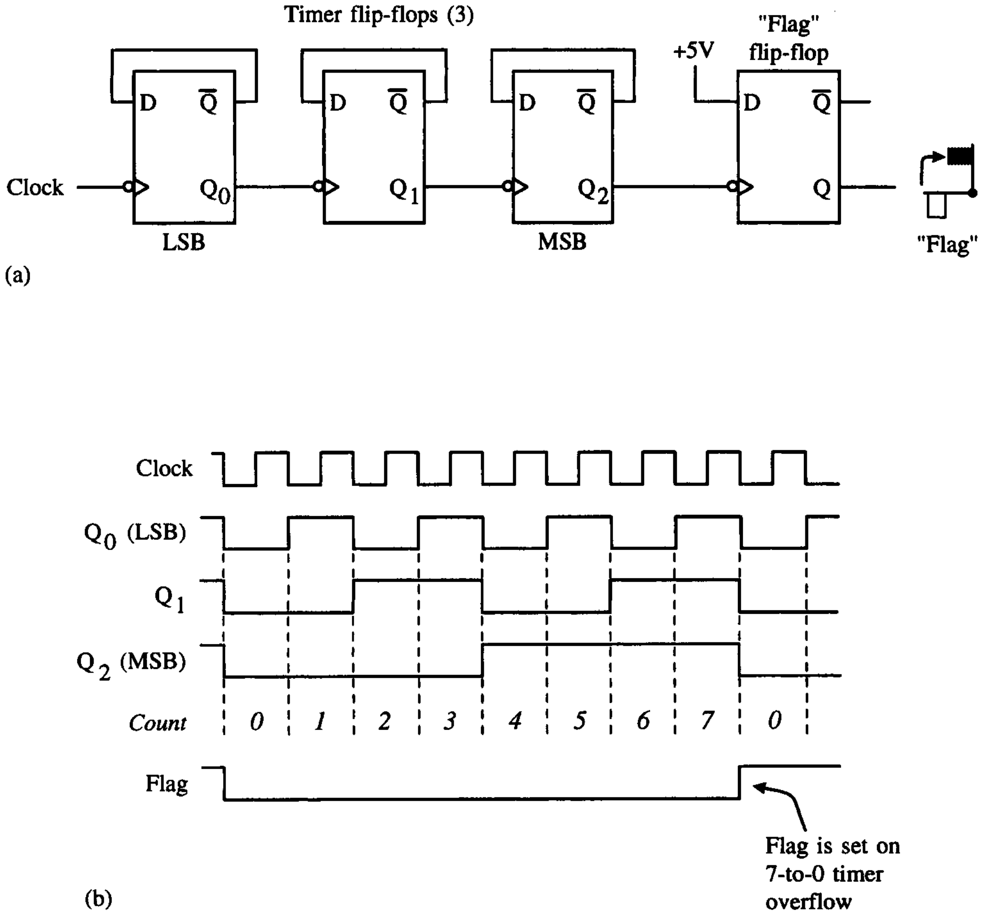


FIGURE 4-1
A 3-bit timer. (a) Schematic. (b) Timing diagram.

port. Each is a 16-bit timer; therefore, the 16th or last stage divides the input clock frequency by $2^{16} = 65,536$.

In interval timing applications, a timer is programmed to overflow at a regular interval and set the timer overflow flag. The flag is used to synchronize the program to perform an action such as checking the state of inputs or sending data to outputs. Other applications can use the regular clocking of the timer to measure the elapsed time between two conditions (e.g., pulse width measurements).

Event counting is used to determine the number of occurrences of an event, rather than to measure the elapsed time between events. An "event" is any external stimulus that provides a 1-to-0 transition to a pin on the 8051 IC. The timers can also provide the baud rate clock for the 8051's internal serial port.

The 8051 timers are accessed using six special function registers. (See Table 4-1.) An additional five SFRs provide access to the third timer in the 8052.

TABLE 4-1
Timer special function registers

Timer SFR	Purpose	Address	Bit-Addressable
TCON	Control	88H	Yes
TMOD	Mode	89H	No
TL0	Timer 0 low-byte	8AH	No
TL1	Timer 1 low-byte	8BH	No
TH0	Timer 0 high-byte	8CH	No
TH1	Timer 1 high-byte	8DH	No
T2CON*	Timer 2 control	C8H	Yes
RCAP2L*	Timer 2 low-byte capture	CAH	No
RCAP2H*	Timer 2 high-byte capture	CBH	No
TL2*	Timer 2 low-byte	CCH	No
TH2*	Timer 2 high-byte	CDH	No

*8032/8052 only

4.2 TIMER MODE REGISTER (TMOD)

The TMOD register contains two groups of four bits that set the operating mode for Timer 0 and Timer 1. (See Table 4-2 and Table 4-3.)

TMOD is not bit-addressable, nor does it need to be. Generally, it is loaded once by software at the beginning of a program to initialize the timer mode. Thereafter, the timer can be stopped, started, and so on by accessing the other timer SFRs.

4.3 TIMER CONTROL REGISTER (TCON)

The TCON register contains status and control bits for Timer 0 and Timer 1 (see Table 4-4). The upper four bits in TCON (TCON.4—TCON.7) are used to turn the timers on and off (TR0, TR1), or to signal a timer overflow (TF0, TF1). These bits are used extensively in the examples in this chapter.

TABLE 4-2
TMOD (timer mode) register summary

Bit	Name	Timer	Description
7	GATE	1	Gate bit. When set, timer only runs while $\overline{\text{INT1}}$ is high
6	C/$\overline{\text{T}}$	1	Counter/timer select bit. 1 = event counter 0 = interval timer
5	M1	1	Mode bit 1 (see Table 4–3)
4	M0	1	Mode bit 0 (see Table 4–3)
3	GATE	0	Timer 0 gate bit
2	C/$\overline{\text{T}}$	0	Timer 0 counter/timer select bit
1	M1	0	Timer 0 M1 bit
0	M0	0	Timer 0 M0 bit

TABLE 4-3 Timer modes

M1	M0	Mode	Description
0	0	0	13-bit timer mode (8048 mode)
0	1	1	16-bit timer mode
1	0	2	8-bit auto-reload mode
1	1	3	Split timer mode: Timer 0: TL0 is an 8-bit timer controlled by timer 0 mode bits; TH0, the same except controlled by timer 1 mode bits Timer 1: stopped

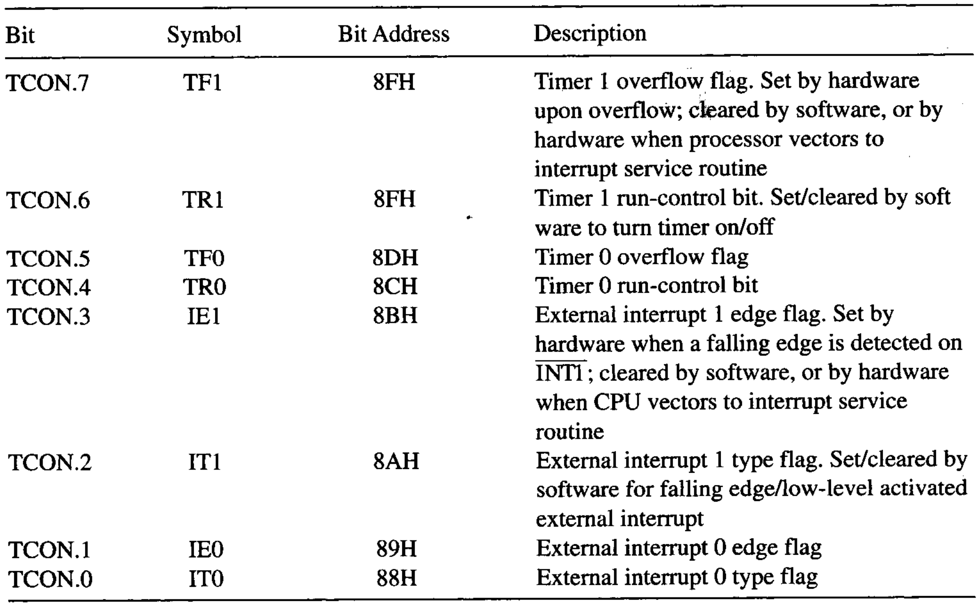
The lower four bits in TCON (TCON.0—TCON.3) have nothing to do with the timers. They are used to detect and initiate external interrupts. Discussion of these bits is deferred until Chapter 6, when interrupts are discussed.

4.4 TIMER MODES AND THE OVERFLOW FLAG

Each timer is discussed below. Since there are two timers on the 8051, the notation "x" is used to imply either Timer 0 or Timer 1; thus, "THx" means either TH1 or TH0, depending on the timer.

The arrangement of timer registers TLx and THx and the timer overflow flags TFx is shown in Figure 4-2 for each mode.

TABLE 4-4
TCON (timer control) register summary



Bit	Symbol	Bit Address	Description
TCON.7	TF1	8FH	Timer 1 overflow flag. Set by hardware upon overflow; cleared by software, or by hardware when processor vectors to interrupt service routine
TCON.6	TR1	8FH	Timer 1 run-control bit. Set/cleared by soft ware to turn timer on/off
TCON.5	TF0	8DH	Timer 0 overflow flag
TCON.4	TR0	8CH	Timer 0 run-control bit
TCON.3	IE1	8BH	External interrupt 1 edge flag. Set by hardware when a falling edge is detected on $\overline{\text{INT1}}$; cleared by software, or by hardware when CPU vectors to interrupt service routine
TCON.2	IT1	8AH	External interrupt 1 type flag. Set/cleared by software for falling edge/low-level activated external interrupt
TCON.1	IE0	89H	External interrupt 0 edge flag
TCON.0	IT0	88H	External interrupt 0 type flag

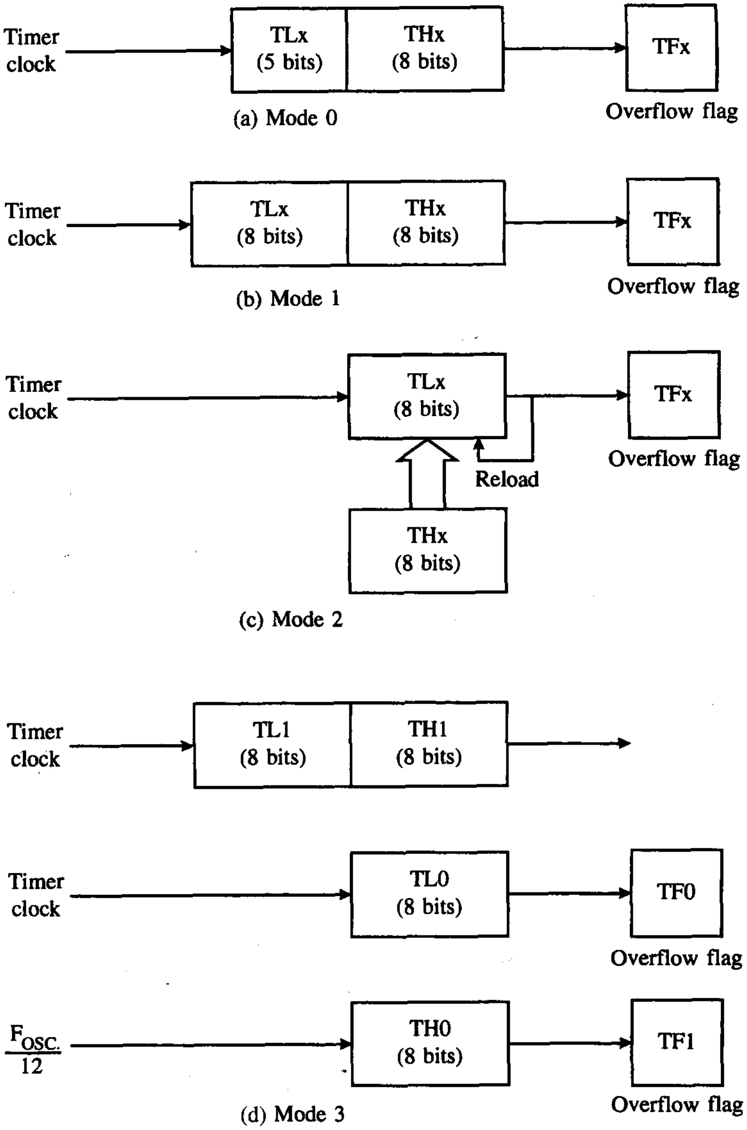


FIGURE 4-2
Timer modes (a) Mode 0 (b) Mode 1 (c) Mode 2 (d) Mode 3

4.4.1 13-Bit Timer Mode (Mode 0)

Mode 0 is a 13-bit timer mode that provides compatibility with the 8051's predecessor, the 8048. It is not generally used in new designs. (See Figure 4-2a.) The timer high-byte (THx) is cascaded with the five least-significant bits of the timer low-byte (TLx) to form a 13-bit timer. The upper three bits of TLx are not used.

4.4.2 16-Bit Timer Mode (Mode 1)

Mode 1 is a 16-bit timer mode and is the same as mode 0, except the timer is operating as a full 16-bit timer. The clock is applied to the combined high and low timer registers (TLx/THx). As clock pulses are received, the timer counts up: 0000H, 0001H, 0002H, etc. An overflow occurs on the 0FFFFH-to-0000H transition of the count and sets the timer overflow flag. The timer continues to count. The overflow flag is the TFx bit in TCON that is read or written by software. (See Figure 4-2b.)

The most-significant bit (MSB) of the value in the timer registers is THx bit 7, and the least-significant bit (LSB) is TLx bit 0. The LSB toggles at the input clock frequency divided by 2, while the MSB toggles at the input clock frequency divided by 65,536 (i.e., 2^{16}.) The timer registers (TLx/THx) may be read or written at any time by software.

4.4.3 8-Bit Auto-Reload Mode (Mode 2)

Mode 2 is 8-bit auto-reload mode. The timer low-byte (TLx) operates as an 8-bit timer while the timer high-byte (THx) holds a reload value. When the count overflows from 0FFH, not only is the timer flag set, but the value in THx is loaded into TLx; counting continues from this value up to the next 0FFH overflow, and so on. This mode is convenient, since timer overflows occur at specific periodic intervals once TMOD and THx are initialized. (See Figure 4-2c.) If TLx contains 4FH, for example, the time counts continuously from 4FH to 0FFH.

4.4.4 Split Timer Mode (Mode 3)

Mode 3 is the split timer mode and is different for each timer. Timer 0 in mode 3 is split into two 8-bit timers. TL0 and TH0 act as separate timers with overflows setting the TF0 and TF1 bits, respectively.

Timer 1 is stopped in mode 3 but can be started by switching it into one of the other modes. The only limitation is that the usual Timer 1 overflow flag, TF1, is not affected by Timer 1 overflows, since it is connected to TH0.

Mode 3 essentially provides an extra 8-bit timer: The 8051 appears to have a third timer. When Timer 0 is in mode 3, Timer 1 can be turned on and off by switching it out of and into its own mode 3. It can still be used by the serial port as a baud rate generator, or it can be used in any way not requiring interrupts (since it is no longer connected to TF1).

4.5 CLOCKING SOURCES

Figure 4-2 does not show how the timers are clocked. There are two possible clock sources, selected by writing to the counter/timer ($C/\overline{T}$) bit in TMOD when the timer is initialized. One clocking source is used for interval timing, the other for event counting.

4.5.1 Interval Timing

If $C/\overline{T} = 0$, continuous timer operation is selected and the timer is clocked from the on-chip oscillator. A divide-by-12 stage is added to reduce the clocking frequency to a value reasonable for most applications.

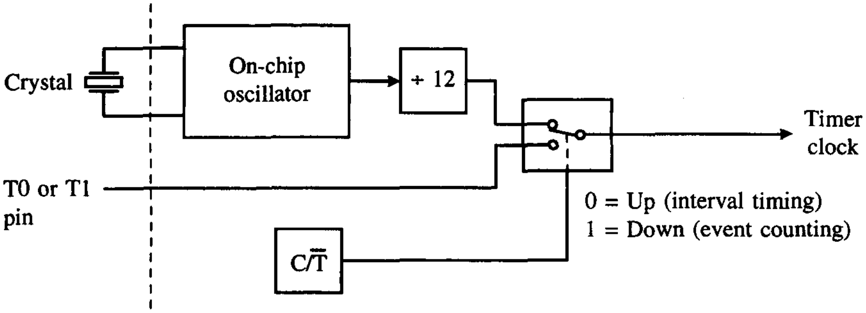


FIGURE 4-3
Clocking source

When continuous timer operation is selected, the timer is used for interval timing. The timer registers (TLx/THx) increment at a rate of 1/12th the frequency of the on-chip oscillator; thus, a 12 MHz crystal would yield a clock rate of 1 MHz. Timer overflows occur after a fixed number of clocks, depending on the initial value loaded into the timer registers, TLx/THx.

4.5.2 Event Counting

If $C/\overline{T}$ = 1, the timer is clocked from an external source. In most applications, this external source supplies the timer with a pulse upon the occurrence of an "event"—the timer is event counting. The number of events is determined in software by reading the timer registers TLx/THx, since the 16-bit value in these registers increments for each event.

The external clock source comes by way of the alternate functions of the Port 3 pins. Port 3 bit 4 (P3.4) serves as the external clocking input for Timer 0 and is known as "T0" in this context. P3.5, or "Tl" is the clocking input for Timer 1. (See Figure 4-3.)

In counter applications, the timer registers are incremented in response to a 1-to-0 transition at the external input, Tx. The external input is sampled during S5P2 of every machine cycle; thus, when the input shows a high in one cycle and a low in the next, the count is incremented. The new value appears in the timer registers during S3P1 of the cycle following the one in which the transition is detected. Since it takes two machine cycles (2 µs) to recognize a 1-to-0 transition, the maximum external frequency is 500 kHz (assuming 12 MHz operation).

4.6 STARTING, STOPPING, AND CONTROLLING THE TIMERS

Figure 4-2 illustrates the various configurations for the timer registers, TLx and THx, and the timer overflow flags, TFx. The two possibilities for clocking the timers are shown in Figure 4-3. We now demonstrate how to start, stop, and control the timers.

The simplest method for starting and stopping the timers is with the run-control bit, TRx, in TCON. TRx is clear after a system reset; thus, the timers are disabled (stopped) by default. TRx is set by software to start the timers. (See Figure 4-4.)

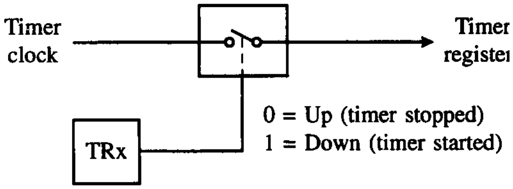


FIGURE 4-4
Starting and stopping the timers

Since TRx is in the bit-addressable register TCON, it is easy to start and stop the timers within a program. For example, Timer 0 is started by

```
SETB TR0
```

and stopped by

```
CLR TR0
```

The assembler will perform the necessary symbolic conversion from "TR0" to the correct bit address. SETB TR0 is exactly the same as SETB 8CH.

Another method for controlling the timers is with the GATE bit in TMOD and the external input INTx. Setting GATE = 1 allows the timer to be controlled by INTx. This is useful for pulse width measurements as follows. Assume INT0 is low but pulses high for a period of time to be measured. Initialize Timer 0 for mode 1, 16-bit timer mode, with

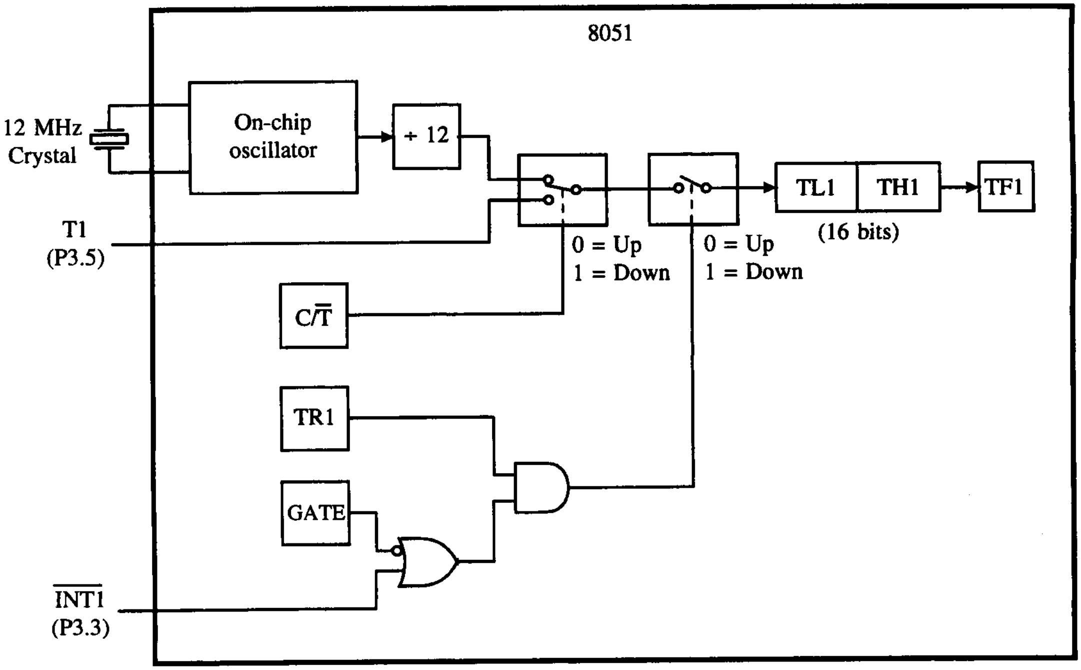


FIGURE 4-5
Timer 1 operating in mode 1

TL0/TH0 = 0000H, GATE = 1, and TR0 = 1. When $\overline{\text{INT0}}$ goes high, the timer is "gated on" and is clocked at a rate of 1 MHz. When $\overline{\text{INT0}}$ goes low, the timer is "gated off' and the duration of the pulse in microseconds is the count in TL0/TH0. ($\overline{\text{INT0}}$ can be programmed to generate an interrupt when it returns low.)

To complete the picture, Figure 4-5 illustrates Timer 1 operating in mode 1 as a 16-bit timer. As well as the timer registers TL1/TH1 and the overflow flag TF1, the diagram shows the possibilities for the clocking source and for starting, stopping, and controlling the timer.

EXAMPLE 4.1 Figure 4-5 illustrates Timer 1 operating in mode 1. Study the figure and identify the 8051's timer registers and control bits shown. Tabulate the bit and byte addresses for each. For the control bits, identify the special function registers that hold them.

Solution

Timer registers:

TH1 at byte address 8DH

TL1 at byte address 8BH

Timer control/mode bits:

TR1 at bit address 8EH (within TCON)

TF1 at bit address 8FH (within TCON)

C/$\overline{\text{T}}$ at bit 6 in TMOD (address 88H)

GATE at bit 7 TMOD (address 88H)

Discussion

Of the four timer control/mode bits shown, only TRL and TF1 are bit-addressable. These bits are usually set and cleared on-the-fly to start and stop the timer or to check its status as appropriate. The C/$\overline{\text{T}}$ and GATE bits are generally written only once, at the beginning of a program to set the timer's mode of operation.

4.7 INITIALIZING AND ACCESSING TIMER REGISTERS

The timers are usually initialized once at the beginning of a program to set the correct operating mode. Thereafter, within the body of a program, the timers are started, stopped, flag bits tested and cleared, timer registers read or updated, and so on, as required in the application.

TMOD is the first register initialized, since it sets the mode of operation. For example, the following instruction initializes Timer 1 as a 16-bit timer (mode 1) clocked by the on-chip oscillator (interval timing):

```
MOV TMOD,#00010000B
```

The effect of this instruction is to set M1 = 0 and M0 = 1 for mode 1, leave C/$\overline{\text{T}}$ = 0 and GATE = 0 for internal clocking, and clear the Timer 0 mode bits. (See Table 4-2.) Of course, the timer does not actually begin timing until its run control bit, TR1, is set.

If an initial count is necessary, the timer registers TL1/TH1 must also be initialized. Remembering that the timers count up and set the overflow flag on an 0FFFFH-to-0000H transition, a 100 µs interval could be timed by initializing TL1/TH1 to 100 counts less than 0000H. The correct value is -100 or 0FF9CH. The following instructions do the job:

```
MOV TL1,#9CH
MOV TH1,#0FFH
```

The timer is then started by setting the run control bit as follows:

```
SETB TR1
```

The overflow flag is automatically set 100 µs later. Software can sit in a "wait loop" for 100 µs using a conditional branch instruction that returns to itself as long as the overflow flag is not set:

```
WAIT:     JNB TF1,WAIT
```

When the timer overflows, it is necessary to stop the timer and clear the overflow flag in software:

```
CLR TR1
CLR TF1
```

4.7.1 Reading a Timer "on the Fly"

In some applications, it is necessary to read the value in the timer registers "on the fly." There is a potential problem that is simple to guard against in software. Since two timer registers must be read, a "phase error" may occur if the low-byte overflows into the high-byte between the two read operations. A value may be read that never existed. The solution is to read the high-byte first, then the low-byte, and then read the high-byte again. If the high-byte has changed, repeat the read operations. The instructions below read the contents of the timer registers TL1/TH1 into registers R6/R7, correctly dealing with this problem.

```
AGAIN: MOV  A,THH
       MOV  R6,TL1
       CJNE A,THH,AGAIN
       MOV R7,A
```

4.8 SHORT, MEDIUM, AND LONG INTERVALS

What is the range of intervals that can be timed? This issue is examined assuming the 8051 is operating from a 12 MHz crystal. The on-chip oscillator is divided by 12 and clocks the timers at a rate of 1 MHz.

The shortest possible interval is limited, not by the timer clock frequency, but by software. Presumably, something must occur at regular intervals, and it is the duration of instructions that limit this for very short intervals. The shortest instruction on the 8051 is one machine cycle or one microsecond. Table 4-5 summarizes the techniques for creating intervals of various lengths. (Operation from a 12 MHz crystal is assumed.)

TABLE 4-5
Techniques for programming timed intervals (12 MHz operation)

Maximum Interval in Microseconds	Technique
≈10	Software tuning
256	8-bit timer with auto-reload
65536	16-bit timer
No limit	16-bit timer plus software loops

EXAMPLE 4.2 Pulse Wave Generation

Write a program that creates a periodic waveform on P1.0 with as high a frequency as possible. What are the frequency and duty cycle of the waveform?

Solution

```
8100                  ORG    8100H
8100 D290     LOOP:   SETB   P1.0 ;one machine cycle
8102 C290             CLR    P1.0 ;one machine cycle
8104 80FA             SJMP   LOOP ;two machine cycles
                             END
```

Discussion

This program creates a pulse waveform on P1.0 with a period of 4 μs. and a frequency of 250 kHz. In each cycle, the signal is high for 1 μs. and low for 3 μs. This corresponds to a duty cycle of 1/4 = 0.25 or 25% (see Figure 4-6).

It might appear at first that the instructions in Figure 4-6 are misplaced, but they are not. The SETB P1.0 instruction, for example, does not actually set the port bit until the end of the instruction, during S6P2.

The period of the waveform can be lengthened by inserting NOP instructions into the loop. Each NOP adds 1 machine cycle or 1 μs to the period of the waveform. For example, adding two NOP instructions after the SETB P1.0 instruction would make the output a square wave (duty cycle = 50%) with a period of 6 μs and a frequency of 166.7 kHz. Beyond a point, software tuning is cumbersome, and a timer is the best choice to create time delays.

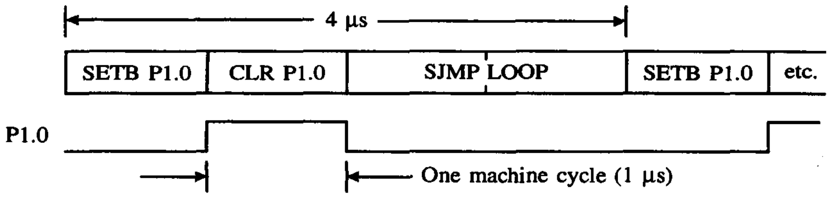


FIGURE 4-6
Waveform for example

EXAMPLE 4.3 **Square Wave Generation**

Write a program that creates a square wave on P1.0 with as high a frequency as possible. What are the frequency and duty cycle of the waveform?

Solution

```
8100          5              ORG 8100H
8100 B290     6       LOOP:  CPL P1.0         ;one machine cycle
8102 80FC     7              SJMP LOOP        ;two machine cycles
              8              END
```

Discussion

The period is 6 µs : high-time = low-time = 3 µs. The frequency is 166.67 kHz and the duty cycle is 50% so this is a square wave, in contrast to the previous example which was not a square wave.

Moderate-length intervals are easily obtained using 8-bit auto-reload mode, mode 2. Since the timed interval is set by an 8-bit count, the longest possible interval before overflow is 2^8 = 256 µs.

EXAMPLE 4.4 **10 kHz Square Wave**

Write a program using Timer 0 to create a 10 kHz square wave on P1.0.

Solution

```
8100            6          ORG   8100H
8100  758902    7          MOV   TMOD,#02H  ;8-bit auto-reload mode
8103  758CCE    8          MOV   TH0,#-50   ;-50 reload value in TH0
8106  D28C      9          SETB  TR0        ;start timer
8108  308DFD   10   LOOP:  JNB   TF0,LOOP   ;wait for overflow
810B  C28D     11          CLR   TF0        ;clear timer overflow flag
810D  B290     12          CPL   P1.0       ;toggle port bit
810F  80F7     13          SJMP  LOOP       ;repeat
               14          END
```

Discussion

The program above creates a square wave on P1.0 with a high-time of 50 µs and a low-time of 50 µs. Since the interval is less than 256 µs, timer mode 2 can be used. An overflow every 50 µs requires a TH0 reload value of 50 counts less than 00H, or -50.

The program uses a complement bit instruction (CPL, line 12) rather than SETB and CLR. Between each complement operation, a delay of 1/2 the desired period (50 µs) is programmed using Timer 0 in 8-bit auto-reload mode. The reload value is specified using decimal notation as -50 (line 8), rather than using hexadecimal notation. The assembler performs the necessary conversion. Note that the timer overflow flag (TF0) is explicitly cleared in software after each overflow (line 11).

Timed intervals longer than 256 µs must use 16-bit timer mode, mode 1. The longest delay is 2^{16} = 65,536 µs or about 0.066 seconds. The inconvenience of mode 1 is that the timer registers must be reinitialized after each overflow, whereas reloading is automatic in mode 2.

EXAMPLE 4.5 **1 kHz Square Wave**

Write a program using Timer 0 to create a 1kHz square wave on P1.0.

Solution

```
8100           6           ORG    8100H
8100  758901   7           MOV    TMOD,#01H   ;16-bit timer mode
8103  75BCFE   8    LOOP:  MOV    TH0,#0FEH   ;-500 (high byte)
8106  758A0C   9           MOV    TL0 #0CH    ;-500 (low byte)
8109  D28C     10          SETB   TR0         ;start timer
810B  308DFD   11   WAIT:  JNB    TF0,WAIT    ;wait for overflow
810E  C28C     12          CLR    TR0         ;stop timer
8110  C28D     13          CLR    TF0         ;clear timer overflow
                                              flag
8112  B290     14          CPL    P1.0        ;toggle port bit
8114  80ED     15          SJMP   LOOP        ;repeat
               16          END
```

Discussion

A 1 kHz square wave requires a high-time of 500 µs and a low-time of 500 µs. Since the interval is longer than 256 µs, mode 2 cannot be used. Full 16-bit timer mode, mode 1, is required. The main difference in the software is that the timer registers, TL0 and TH0, are reinitialized after each overflow (lines 8 and 9).

There is a slight discrepancy in the output frequency in the program above. This results from the extra instructions inserted after the timer overflow to reinitialize the timer. If exactly 1 kHz is required, the reload value for TL0/TH0 must be adjusted somewhat. Such errors do not occur in auto-reload mode, since the timer is never stoppe - it overflows at a consistent rate set by the reload value in TH0.

Since there are two timers in the 8051, a program could make use of them to simultaneously generate two different waveforms on separate port pins.

EXAMPLE 4.6 **Using Two Timers Simultaneously**

Write a program that creates a square wave on P1.0 with a frequency of 10kHz and a square wave on P2.0 with a frequency of 1kHz.

Solution

```
8100         6          ORG 8100H
8100 758912  7          MOV TMOD, #12H  ;timer 1 in mode 1,
             8                          ; timer 0 in mode 2
8103 758CCE  9          MOV TH0, #-50   ;-50 reload value
             10                         ; in TH0
8106 D28C    11         SETB TR0        ;start timer 0
8108 758DFE  12  LOOP:  MOV TH1, #0FEH  ;-500 (high byte)
```

```
810B 758B0C          MOV TL1, #0CH     ;-500 (low byte)
810E D28E            SETB TR1          ;start timer 1
8110 308D04   WAIT:  JNB TF0, NEXT     ;timer 0 overflow?
                                       ; no: check timer 1
8113 C28D            CLR TF0           ;yes: clear timer 0
                                       ; overflow flag
8115 B290            CPL P1.0          ;toggle P1.0
8117 308FF6   NEXT:  JNB TF1, WAIT     ;timer 1 overflow?
                                       ; no: check timer 0
811A C28E            CLR TR1           ;stop timer 1
811C C28F            CLR TF1           ;clear timer 1
                                       ; overflow flag
811E B2A0            CPL P2.0          ;toggle P2.0
8120 80E6            SJMP LOOP         ;repeat
                     END
```

Discussion

Both timers 0 and 1 are used in this case to simultaneously generate two square waves on P1.0 and P2.0, respectively. The value written into TMOD initializes both timers at the same time. Even though the timers are running simultaneously, the testing for overflow has to be done in sequence. Timer 0 is checked first since its period is smaller in order to avoid missing its overflows. Notice that Timer 0 is set to operate in mode 2 auto-reload mode so there is no need to reload the count after every overflow. Meanwhile, Timer 1 operates in mode 1 so its count must be reloaded every time an overflow occurs.

Intervals longer than 0.066 seconds can be achieved by cascading Timer 0 and Timer 1 through software, but this ties up both timers. A more practical approach uses one of the timers in 16-bit mode with a software loop counting overflows. The desired operation is performed every *n* overflows.

EXAMPLE 4.7 Buzzer Interface

A buzzer is connected to P1.7, and a debounced switch is connected to P1.6 (see Figure 4-7). Write a program that reads the logic level provided by the switch and sounds the buzzer for 1 second for each 1-to-0 transition detected.

Solution

```
0064              HUNDRED     EQU    100      ;100 × 10000 us=1 sec.
D8F0              COUNT EQU          -10000
8100                          ORG    8100H
8100   758901           MOV   TMOD, #01H      ;use timer 0 in mode 1
8103   3096FD     LOOP: JNB   P1.6,LOOP       ;wait for 1 input
8106   2096FD     WAIT: JB    P1.6,WAIT       ;wait for 0 input
8109   D297                   SETB   P1.7     ;turn buzzer on
810B   128112                 CALL   DELAY    ;wait 1 second
```

```
810E  C297     14              CLR    P1.7    ;turn buzzer off
8110  80F1     15              SJMP   LOOP
               16  ;
8112  7F64     17  DELAY:      MOV    R7,#HUNDRED
8114  758CD8   18  AGAIN:      MOV    TH0,#HIGH COUNT
8117  758AF0   19              MOV    TL0,#LOW COUNT
811A  D28C     20              SETB   TR0
811C  308DFD   21  WAIT2:      JNB    TF0,WAIT2
811F  C28D     22              CLR    TF0
8121  C28C     23              CLR    TR0
8123  DFEF     24              DJNZ   R7,AGAIN
8125  22       25              RET
               26              END
```

Discussion

The buzzer in Figure 4-7 is a piezo ceramic transducer that vibrates when stimulated with a DC voltage. A typical example is the Projects Unlimited AI-430 that generates a tone of about 3 kHz at 5 volts DC. An inverter is used as a driver since the AI-430 draws 7 mA of current. As indicated in the 8051's DC Characteristics in Appendix E, Port 1pins can sink a maximum of 1.6 mA. The AI-430 costs a few dollars.

The main loop in the software consists of six instructions (lines 10-15). In line 10, a one-instruction loop is executed to wait for the input signal on P1.7 to go high. Then another one-instruction loop (line 11) executes to wait for the input signal to go low. When this happens, the buzzer sounds for 1 second. This is implemented in the next three instructions. First, P1.7 is set to sound the buzzer (line 12); second, a 1-second delay subroutine is called (line 13), and, third, P1.7 is cleared to silence the buzzer (line 14). Then the main loop is executed again (line 15).

The delay subroutine (lines 17-25) uses the technique identified in Table 4-6 as "16-bit timer plus software loops." In theory, delays of any length can be created. In this example, a 1-second delay is created by using a count of —10,000 for TH0/TL0 with 16-bit timer mode. The effect is to create a 10,000 μs delay. This delay (lines 18-23) is enclosed within a loop that executes 100 times, using R7 as a counter. The effect is a 1-second delay.

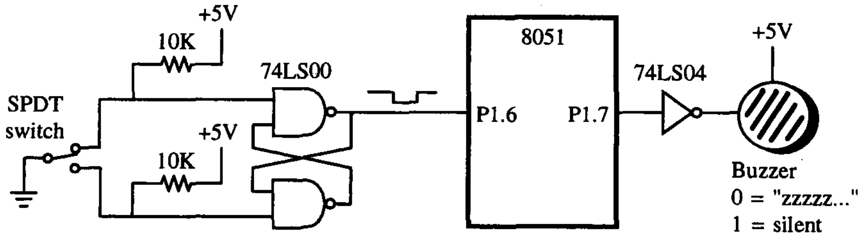


FIGURE 4-7 Buzzer example

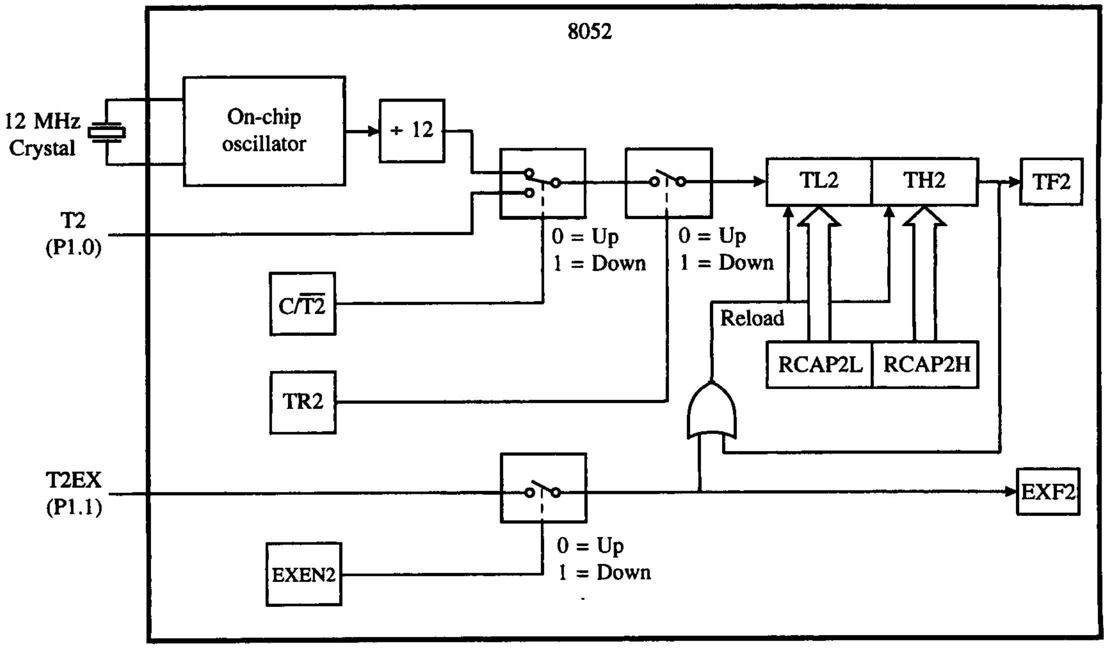


FIGURE 4-8
Timer 2 in 16-bit auto-reload mode

There are two situations not handled in the preceding example. First, if the input toggles during the 1 second that the buzzer is sounding, the transition is not detected, since the software is busy in the delay routine. Second, if the input toggles very quickly-in less than a microsecond-the transition may be missed altogether by the JNB and JB instructions. Problem 5 at the end of this chapter deals with the first situation. The second can only be handled using an interrupt input to "latch" a status flag when a 1-to-0 transition occurs. This is discussed in Chapter 6.

4.9 PRODUCING EXACT FREQUENCIES

As was discussed in the previous section, the output frequencies of the square waves generated so far have slight errors. The errors are due to the rounding off, and overhead caused by the time it takes to execute the instructions themselves.

4.9.1 Eliminating Round-off Errors

Round-off errors occur when the desired period of a certain periodic waveform is not an integer, so it is rounded off to an integer in order that it can be represented in the 8051.

EXAMPLE 4.8 **Round-off Errors**

Suppose a square wave of 3kHz is to be generated. What should the reload value of the timer be? Calculate the round-off error if any, and hence determine what crystal frequency would produce no round-off error.

Solution

A frequency of 3 kHz means the period should be 333.33 µs, so the high-time = low-time = 166.67 s. Since the 8051 can only handle integer count values, the timer reload value should be 167 counts before overflow, or —167. Notice that we have rounded off the desired count to an integer value. The actual period is hence 167 x 2 x 1s = 334 µs so the actual frequency is 2.994 kHz. The round-off error is:

$$\text{Round-off error} = \frac{|f_{\text{desired}} - f_{\text{rounded-off}}|}{f_{\text{desired}}} \times 100\%$$

$$= \frac{|3\text{ kHz} - 2.994\text{ kHz}|}{3\text{ kHz}} \times 100\% = 0.2\%.$$

Discussion

Our previous examples used a crystal frequency of 12 MHz but that causes round-off errors. Assuming that a count of 167 is used, our goal is to calculate the crystal frequency so that an exact square wave with a frequency of 3 kHz is obtained.

Desired period = 333.33 µs so

```
DESIRED HIGH-TIME = DESIRED LOW-TIME
                  = 166.67 µs
                  = DESIRED TIME BETWEEN EVERY TIMER OVERFLOW
                  = NUMBER OF COUNTS × PERIOD OF MACHINE CYCLE
                  = 167 × PERIOD OF MACHINE CYCLE
Hence, PERIOD OF MACHINE CYCLE = 166.67 µs / 167 = 0.9980239 µs
and    MACHINE CYCLE FREQUENCY = 1/0.9980239 µs = 1.00198 MHz
so     CRYSTAL FREQUENCY = 1.00198 MHz × 12 = 12.0238 MHz.
```

4.9.2 Compensating for Overhead Due to Instructions

Instructions take some time to be carried out. The simplest of instructions require one machine cycle while the most complex ones require 4. Therefore, instructions also add to the time delay and if we prefer exact frequencies, we would need to adjust our initial timer counts to compensate for the overhead caused by the execution of these instructions. Example 4.9 considers a very simple demonstration of this.

EXAMPLE 4.9 Rewrite the program of Example 4.5 to compensate for the overhead delay due to instructions in the program.

Solution

```
8H00             H          ORG 8H00H
8H00 75890H      2          MOV TMOD, #0HH ;H6-bit timer mode
8103 758CFE      3   LOOP:  MOV TH0, #0FEH ;-490 (high byte)
```

```
8106 758A0E           MOV TL0, #0EH    ;-490 (low byte)
8109 D28C             SETB TR0         ;start timer
810B 308DFD     WAIT: JNB TF0, WAIT    ;wait for overflow
810E C28C             CLR TR0          ;stop timer
8110 C28D             CLR TF0          ;clear timer overflow flag
8112 B290             CPL P1.0         ; toggle port bit
8114 80ED             SJMP LOOP        ; repeat
                      END
```

Discussion

The program is exactly the same as that in Example 4.5. Only the reload value is different. In this case, a value of -490 was chosen to compensate for the overhead due to instructions. Let's analyze to see why we chose this value.

TABLE 4-6

T2CON (Timer 2 control) register summary

Bit	Symbol	Bit Address	Description
T2CON.7	TF2	CFH	Timer 2 overflow flag. (Not set when TCLK or RCLK = 1.)
T2CON.6	EXF2	CEH	Timer 2 external flag. Set when either a capture or reload is caused by 1-to-0 transition on T2EX and EXEN2 = 1; when timer interrupts are enabled, EXF2 = 1 causes CPU to vector to service routine; cleared by software
T2CON.5	RCLK	CDH	Timer 2 receiver clock. When set, Timer 2 provides serial port receive baud rate; Timer 1 provides transmit baud rate
T2CON.4	TCLK	CCH	Timer 2 transmit clock. When set, Timer 2 provides transmitter baud rate; Timer 1 provides receiver baud rate
T2CON.3	EXEN2	CBH	Timer 2 external enable. When set, capture or reload occurs on 1-to-0 transition of T2EX
T2CON.2	TR2	CAH	Timer 2 run control bit. Set/cleared by software to turn Timer 2 on/off.
T2CON.1	$C/\overline{T2}$	C9H	Timer 2 counter/interval timer select bit. 1 = event counter 0 = interval timer
T2CON.0	$CP/\overline{RL2C}$	C8H	Timer 2 capture/reload flag. When set, capture occurs on 1-to-0 transition of T2EX if EXEN2 = 1; when clear, auto reload occurs on timer overflow or T2EX transition if EXEN2 = 1; if RCLK or TCLK = 1, this bit is ignored

We concentrate on the software loop, which consists of the two MOV instructions, followed by SETB, JNB, two CLRs, CPL and SJMP. The MOV, JNB, and SJMP instructions require 2 machine cycles each, while SETB, CLR and CPL each require 1.

The initial value for P1.0 can be HIGH or LOW. Let's assume that P1.0 is initially LOW. The first two MOV instructions require 2 machine cycles each, while SETB requires 1. The next instruction is JNB which determines the length and duration of the loop, and will be executed repeatedly until the timer 0 overflows. Each execution of JNB requires 2 machine cycles.

When timer 0 overflows, this is the moment in which P1.0 should be complemented to the opposite state immediately. However, in the above program, P1.0 would only be complemented after full execution of the CPL instruction, which is 3 machine cycles later. This extra delay is the overhead due to instructions. Figure 4-8 illustrates this in more detail, where the states of P1.0 and TR0 are indicated. Also notice that the actual high- or low-time of the square wave on P1.0 is between two subsequent CPL instructions. Figure 4-8 shows that this duration is one full timer cycle of 490 µs plus the overhead due to executing the CLR and CPL instructions after the overflow (3 µs), and the execution of the SJMP, MOV and SETB instructions (7 µs.) before the timer is restarted for the next cycle.

Therefore, taking all this into consideration, the actual high- or low-time time is 500 µs., exactly resulting in the desired frequency of 1 kHz for the square wave.

4.10 8052 TIMER 2

The third timer added on the 8052 IC is a powerful addition to the two just discussed. As shown earlier in Table 4-1, five extra special-function registers are added to accommodate Timer 2. These include the timer registers, TL2 and TH2, the timer control register, T2CON, and the capture registers, RCAP2L and RCAP2H.

The mode for Timer 2 is set by its control register, T2CON. (See Table 4-6.) Like Timers 0 and 1, Timer 2 can operate as an interval timer or event counter. The clocking source is provided internally by the on-chip oscillator, or externally by T2, the alternate function of Port 1 bit 0 (P1.0) on the 8052 IC. The C/$\overline{\text{T2}}$ bit in T2CON selects between the internal and external clock, just as the C/$\overline{\text{T}}$ bits do in TCON for Timers 0 and 1. Regardless of the clocking source, there are three modes of operation: auto-reload, capture, and baud rate generator.

4.10.1 Auto-Reload Mode

The capture/reload bit in T2CON selects between the first two modes. When CP/$\overline{\text{RL2}}$ = 0, Timer 2 is in auto-reload mode with TL2/TH2 as the timer registers, and RCAP2L and RCAP2H holding the reload value. Unlike the reload mode for Timers 0 and 1, Timer 2 is always a full 16-bit timer, even in auto-reload mode.

Reload occurs on an 0FFFFH-to-0000H transition in TL2/TH2 and sets the Timer 2 flag, TF2. This condition is determined by software or is programmed to generate an interrupt. Either way, TF2 must be cleared by software before it is set again.

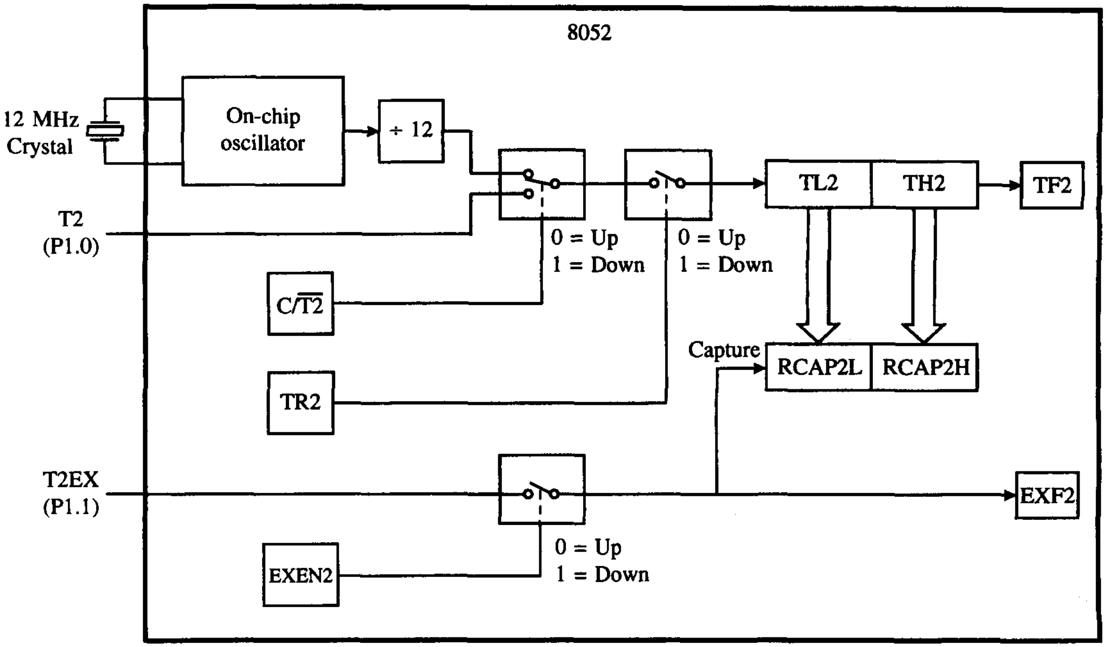


FIGURE 4-9
Timer 2 in 16-bit capture mode

Optionally, by setting EXEN2 in T2CON, a reload also occurs on the 1-to-0 transition of the signal applied to pin T2EX, which is the alternate pin function for P1.1 on the 8052 IC. A 1-to-0 transition on T2EX also sets a new flag bit in Timer 2, EXF2. As with TF2, EXF2 is tested by software or generates an interrupt. EXF2 must be cleared by software. Timer 2 in auto-reload mode is shown in Figure 4-8.

4.10.2 Capture Mode

When CP/$\overline{\text{RL2}}$ = 1, capture mode is selected. Timer 2 operates as a 16-bit timer and sets the TF2 bit upon an 0FFFFH-to-0000H transition of the value in TL2/TH2. The state of TF2 is tested by software or generates an interrupt.

To enable the capture feature, the EXEN2 bit in T2CON must be set. If EXEN2 = 1, a 1-to-0 transition on T2EX (P1.1) "captures" the value in timer registers TL2/TH2 by clocking it into registers RCAP2L and RCAP2H. The EXF2 flag in T2CON is also set and, as stated above, is tested by software or generates an interrupt. Timer 2 in capture mode is shown in Figure 4-9.

4.11 BAUD RATE GENERATION

Another use of the timers is to provide the baud rate clock for the on-chip serial port. This comes by way of Timer 1 on the 8051 IC or Timer 1 and/or Timer 2 on the 8052 IC. Baud rate generation is discussed in Chapter 5.

SUMMARY

This chapter has introduced the 8051 and 8052 timers. The software solutions for the examples presented here feature one common but rather limiting trait. They consume all of the CPU's execution time. The programs execute in wait loops, waiting for a timer overflow. This is fine for learning purposes, but for practical control-oriented applications using microcontrollers, the CPU must perform other duties and respond to external events, such as an operator entering a parameter from a keyboard. In the chapter on interrupts, we shall demonstrate how to use the timers in an "interrupt-driven" environment. The timer overflow flags are not tested in a software loop but generate an interrupt. Another program temporarily interrupts the main program while an action is performed that affects the timer interrupt (perhaps toggling a port bit). Through interrupts, the illusion of doing several things simultaneously is created.

PROBLEMS

4.1 Write an 8051 program that creates a square wave on P1.5 with a frequency of 100 kHz. (Hint: Don't use the timers.)

4.2 What is the effect of the following instruction?

```
SETB 8EH
```

4.3 What is the effect of the following instruction?

```
MOV TMOD,#11010101B
```

4.4 Consider the three-instruction program shown in Example 4.2. What are the frequency and duty cycle of the waveform created on P1.0 for a 16 MHz 8051?

4.5 Rewrite the solution to Example 4.7 to include a "restart" mode. If a 1-to-0 transition occurs while the buzzer is sounding, restart the timing loop to continue the buzz for another second. This is illustrated in Figure 4-10.

4.6 Write an 8051 program to generate a 12 kHz square wave on P1.2 using Timer 0.

4.7 Design a "turnstile" application using Timer 1 to determine when the 10,000th person has entered a fairground. Assume (a) a turnstile sensor connects to TI and generates a pulse each time the turnstile is rotated, and (b) a light is connected to P1.7 that is on when P1.7 = 1, and off otherwise. Count "events" at T1 and turn on the light at P1.7 when the 10,000th person enters the fairground. (See Figure 4-11.)

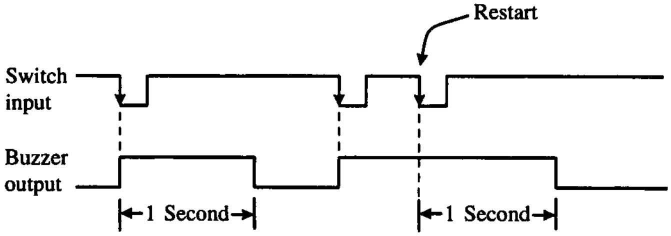


FIGURE 4-10
Timing for modified buzzer example

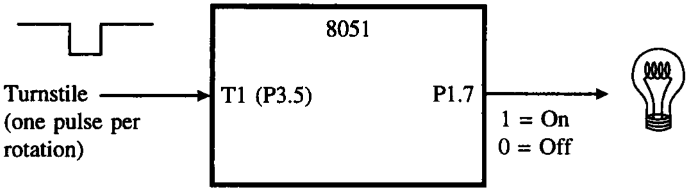


FIGURE 4-11
Turnstile problem

4.8 The international tuning standard for musical instruments is "A above middle C" at a frequency of 440 Hz. Write an 8051 program to generate this tuning frequency and sound a 440 Hz tone on a loudspeaker connected to P1.1. (See Figure 4-12.) Due to rounding of the values placed in TL1/TH1, there is a slight error in the output frequency. What is the exact output frequency, and what is the percentage error? What value of crystal would yield exactly 440 Hz with the program you have written?

4.9 Write an 8051 program to generate a 500 Hz signal on P1.0, using Timer 0. The wave form should have a 30% duty cycle (duty cycle = high-time / period).

4.10 The circuit shown in Figure 4-13 will provide an extremely accurate 60 Hz signal to T2 by tapping the secondary of a power supply transformer. Initialize Timer 2 such that it is clocked by T2 and overflows once per second. Upon each overflow, update a time-of-day value stored in the 8052's internal memory at locations 50H (hours), 51H (minutes), and 52H (seconds). More timer examples and problems are found in Chapter 6.

4.11 a. Write the program to generate a (30% duty cycle) rectangular pulse wave on P1.0, and a square wave on P2.0.
b. Hence, determine the (i) round-off error, and (ii) error due to instruction overhead for both waveforms.

4.12 Write a program to generate a 2.5 kHz waveform on P1.0 with a 20% duty cycle. Show your workings and write comments for each line of the code.

4.13 Consider an external crystal oscillator with a frequency of 16 MHz instead of the standard 12MHz. Write a program to generate a periodic waveform on P1.0 with the highest possible frequency. Hence, calculate the frequency and duty cycle of that waveform.

4.14 Suppose you want to use the 8051's timer to time the duration of 1 hour. What mode should you ask the timer to be in? Why?
4.15 Suppose you are required to generate a square waveform on P1.0 with a frequency of 21 kHz having a duty cycle of 10%, what mode should you use the timer in? Why?

4.16 What is the highest value that an 8051 timer can count to? If you use an 8051's timer as an event counter, what effect will it have on your counting? Why?

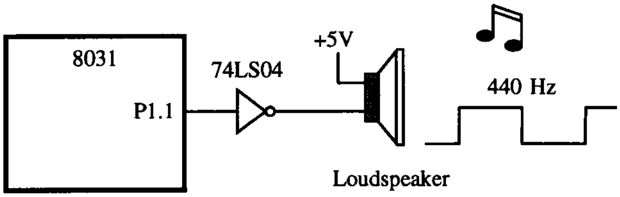


FIGURE 4-12
Loudspeaker interface

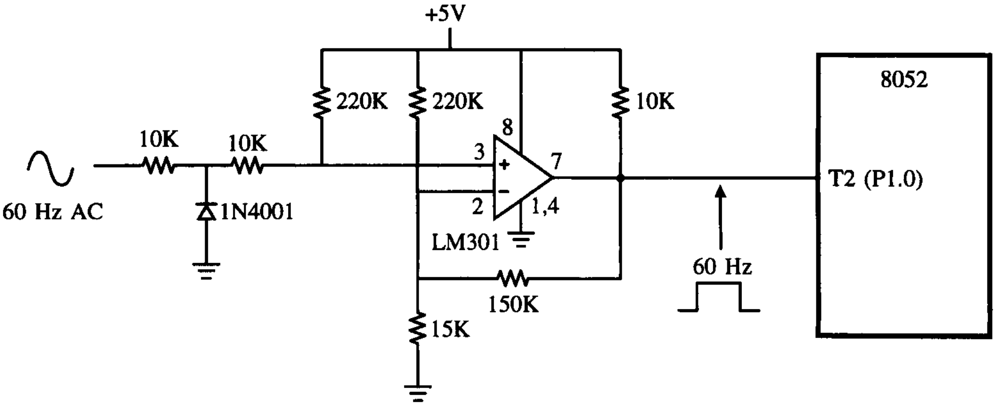


FIGURE 4-13
60 Hz time base

4.17 What do you mean by a 16-bit timer? Explain.

4.18 a. Write a program to generate a 3 kHz waveform on P2.0 with a 30% duty cycle. Show your workings and write comments for each line of the code. Make adjustments to overcome the delay overhead due to instructions.

b. For the program above, is there any truncation error? If so, calculate the percentage error. If not, why do you say so?

CHAPTER 5

Serial Port Operation

5.1 INTRODUCTION

The 8051 includes an on-chip serial port that can operate in several modes over a wide range of frequencies. The essential function of the serial port is to perform parallel-to-serial conversion for output data, and serial-to-parallel conversion for input data.

Hardware access to the serial port is through the TXD and RXD pins introduced in Chapter 2. These pins are the alternate functions for two Port 3 bits, P3.1 on pin 11 (TXD), and P3.0 on pin 10 (RXD).

The serial port features **full duplex** operation (simultaneous transmission and reception) and **receive buffering,** allowing one character to be received and held in a buffer while a second character is received. If the CPU reads the first character before the second is fully received, data are not lost.

The serial port frequency of operation, or **baud rate,** can be fixed (derived from the 8051 on-chip oscillator) or variable. If a variable baud rate is used, Timer 1 supplies the baud rate clock and must be programmed accordingly. (On the 8032/8052, Timer 2 can be programmed to supply the baud rate clock.)

Two special function registers, the serial port buffer register (SBUF) and the serial port control register (SCON), provide software access to the serial port.

5.2 SERIAL COMMUNICATION

Before we move on to discuss the operation of the 8051 serial port, we first touch on some basic concepts of serial communication. Serial communication involves the transmission of bits of data through only one communication line. The data are transmitted bit by bit in either synchronous or asynchronous format. **Synchronous** serial communication transmits one whole block of characters in synchronization with a reference clock while **asynchronous** serial communication randomly transmits one character at any time, independent of any clock. As an example, the transmission of each key press from the keyboard to the computer is asynchronous communication since the rate at which the key presses are entered is not fixed and may occur at any instant. Meanwhile, if two computers were to communicate synchronously, both of them would be synchronized to the same reference clock throughout the duration of the transmission.

5.3 SERIAL PORT BUFFER REGISTER (SBUF)

The serial port buffer register (SBUF) at address 99H is really two buffers. Writing to SBUF loads data to be transmitted, and reading SBUF accesses received data. These are two separate and distinct registers, the transmit write-only register, and the receive read-only register. (See Figure 5-1.)

Notice in Figure 5-1 that a serial-to-parallel shift register is used to clock in the received data before it is transferred to the receive read-only register. This shift register is the key element in providing receive buffering. Only when all 8 bits of the incoming data are received will they be transferred to the receive read-only register. This ensures that while the incoming data are being received, the previous received data are still intact in the receive read-only register.

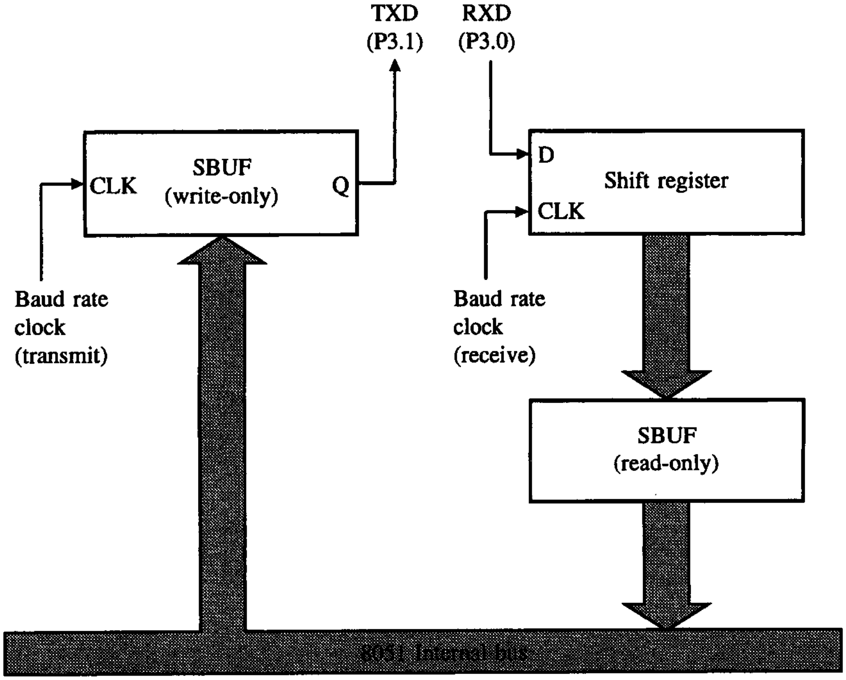


FIGURE 5-1
Serial port block diagram

5.4 SERIAL PORT CONTROL REGISTER (SCON)

The serial port control register (SCON) at address 98H is a bit-addressable register containing status bits and control bits. Status bits indicate the end of a character transmission or reception and are tested in software or programmed to cause an interrupt. Meanwhile, writing to the control bits would set the operating mode for the 8051 serial port. (See Table 5-1 and Table 5-2.)

Before using the serial port, SCON is initialized for the correct mode, and so on. For example, the following instruction

```
MOV   SCON,#01010010B
```

initializes the serial port for mode 1 (SM0/SM1 = 0/1), enables the receiver (REN = 1), and sets the transmit interrupt flag (T1 = 1) to indicate the transmitter is ready for operation.

5.5 MODES OF OPERATION

The 8051 serial port has four modes of operation, selectable by writing 1s or 0s into the SM0 and SM1 bits in SCON. Three of the modes enable asynchronous communications, with each character received or transmitted framed by a start bit and a stop bit. Readers familiar with the operation of a typical RS232C serial port on a microcomputer will find these modes familiar territory. In the fourth mode, the serial port operates as a simple shift register. Each mode is summarized below.

5.5.1 8-Bit Shift Register (Mode 0)

Mode 0, selected by writing 0s into bits SM1 and SM0 of SCON, puts the serial port into 8-bit shift register mode. Serial data enter and exit through RXD, and TXD outputs the shift clock. Eight bits are transmitted or received with the least-significant (LSB) first.

TABLE 5-1
SCON (serial port control) register summary

Bit	Symbol	Address	Description
SCON.7	SM0	9FH	Serial port mode bit 0 (see Table 5-2)
SCON.6	SM1	9EH	Serial port mode bit 1 (see Table 5–2)
SCON.5	SM2	9DH	Serial port mode bit 2. Enables multiprocessor communications in modes 2 & 3; R1 will not be activated if received 9th bit is 0
SCON.4	REN	9CH	Receiver enable. Must be set to receive characters
SCON.3	TB8	9BH	Transmit bit 8. 9th bit transmitted in modes and 3; set/cleared by software
SCON.2	RB8	9AH	Receive bit 8. 9th bit received
SCON.1	TI	99H	Transmit interrupt flag. Set at end of character transmission; cleared by software
SCON.0	RI	98H	Receive interrupt flag. Set at end of character reception; cleared by software

TABLE 5-2
Serial Dort modes

SM0	SM1	Mode	Description	Baud Rate
0	0	0	Shift register	Fixed (oscillator frequency ÷ 12)
0	1	1	8-bit UART	Variable (set by timer)
1	0	2	9-bit UART	Fixed (oscillator frequency ÷ 12 or ÷ 64)
1	1	3	9-bit UART	Variable (set by timer)

The baud rate is fixed at 1/12th the on-chip oscillator frequency. The terms "RXD" and "TXD" are misleading in this mode. The RXD line is used for both data input and output, and the TXD line serves as the clock.

Transmission is initiated by any instruction that writes data to SBUF. Data are shifted out on the RXD line (P3.0) with clock pulses sent out the TXD line (P3.1). Each transmitted bit is valid on the RXD pin for one machine cycle. During each machine cycle, the clock signal goes low on S3P1 and returns high on S6P1. The timing for output data is shown in Figure 5-2.

Reception is initiated when the receiver enable bit (REN) is 1 and the receive interrupt bit (RI) is 0. The general rule is to set REN at the beginning of a program to initialize the serial port and then clear RI to begin a data input operation. When RI is cleared, clock pulses are written out the TXD line, beginning the following machine cycle, and data are clocked in the RXD line. Obviously, it is up to the attached circuitry to provide data on the

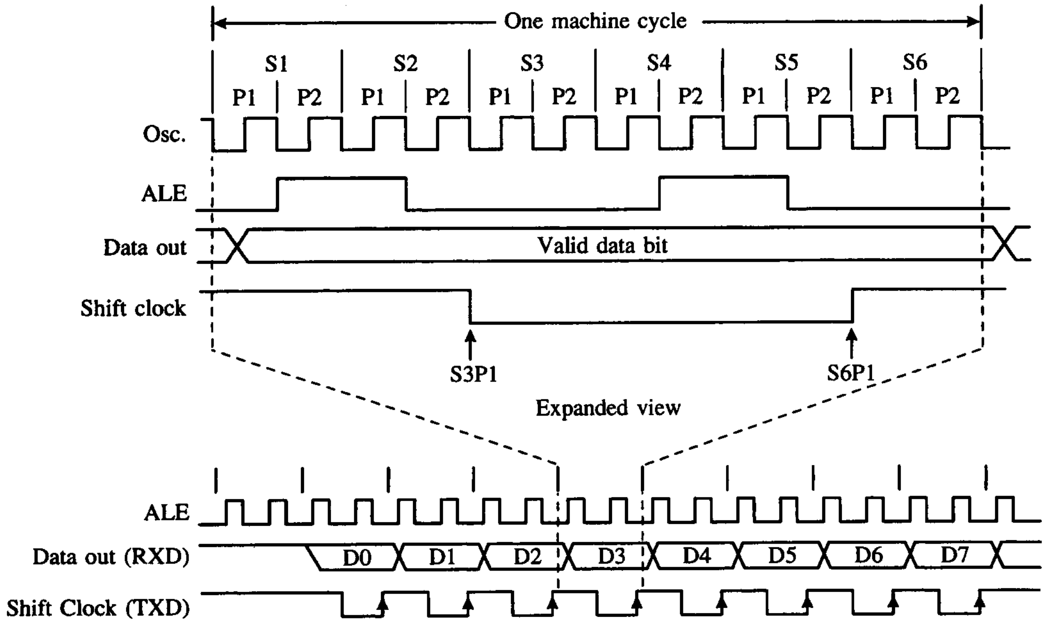


FIGURE 5-2
Serial port transmit timing for mode 0

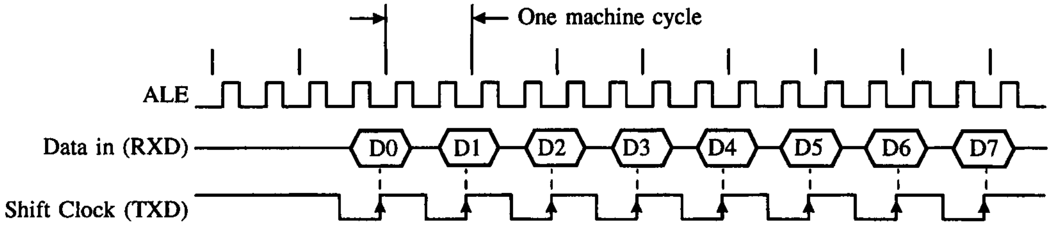


FIGURE 5-3
Serial port receive timing for mode 0

RXD line as synchronized by the clock signal on TXD. The clocking of data into the serial port occurs on the positive edge of TXD. (See Figure 5-3.) Notice that in this mode of operation, the data transfers between the 8051 serial port and the attached circuitry are via synchronous communication where both parties are synchronized to the clock signal on TXD. As will be uncovered in the later sections, the other operating modes of the serial port operate via asynchronous communication.

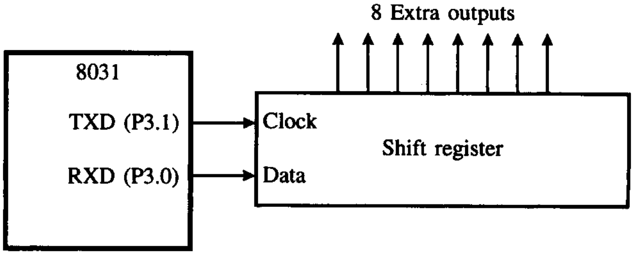
One possible application of shift register mode is to expand the output capability of the 8051. A serial-to-parallel shift register IC can be connected to the 8051 TXD and RXD lines to provide an extra eight output lines. (See Figure 5-4.) Additional shift registers may be cascaded to the first for further expansion.

5.5.2 8-Bit UART with Variable Baud Rate (Mode 1)

In mode 1 the 8051 serial port operates as an 8-bit UART with variable baud rate. A UART, or "universal asynchronous receiver/transmitter," is a device that receives and transmits serial data with each data character preceded by a start bit (low) and followed by a stop bit (high). A parity bit is sometimes inserted between the last data bit and the stop bit. The essential operation of a UART is parallel-to-serial conversion of output data and serial-toparallel conversion of input data.

In mode 1, 10 bits are transmitted on TXD or received on RXD. These consist of a start bit (always 0), eight data bits (LSB first), and a stop bit (always 1). For a receive operation, the stop bit goes into RB8 in SCON. In the 8051, the baud rate is set by the Timer 1 overflow rate; the 8052 baud rate is set by the overflow rate of Timer 1 or Timer 2 or a combination of the two (one for transmit, the other for receive).

FIGURE 5-4
Serial port shift register mode



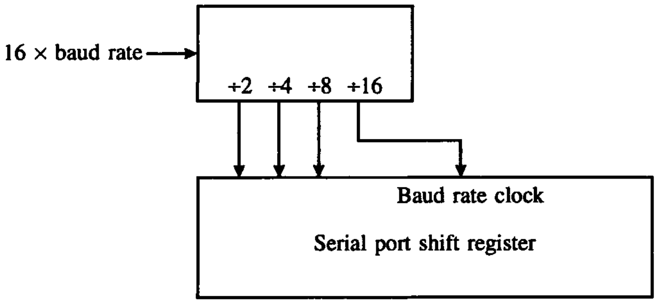


FIGURE 5-5
Serial port clocking

Clocking and synchronizing the serial port shift registers in modes 1, 2, and 3 is established by a 4-bit divide-by-16 counter, the output of which is the baud rate clock. (See Figure 5-5.) The input to this counter is selected through software, as discussed later.

Transmission is initiated by writing to SBUF but does not actually start until the next rollover of the divide-by-16 counter supplying the serial port baud rate. Shifted data are outputted on the TXD line beginning with the start bit, followed by the eight data bits, then the stop bit. The period for each bit is the reciprocal of the baud rate as programmed in the timer. The transmit interrupt flag (TI) is set as soon as the stop bit appears on TXD. (See Figure 5-6.)

Reception is initiated by a 1-to-0 transition on RXD. The divide-by-16 counter is immediately reset to align the counts with the incoming bit stream (the next bit arrives on the next divide-by-16 rollover, and so on). The incoming bit stream is sampled in the middle of the 16 counts.

The receiver includes "false start bit detection" by requiring a 0 state eight counts after the first 1-to-0 transition. If this does not occur, it is assumed that the receiver was triggered by noise rather than by a valid character. The receiver is reset and returns to the idle state, looking for the next 1-to-0 transition.

Assuming a valid start bit was detected, character reception continues. The start bit skipped and eight data bits are clocked into the serial port shift register. When all eight bits have been clocked in, the following occur:

1. The ninth bit (the stop bit) is clocked into RB8 in SCON,
2. SBUF is loaded with the eight data bits, and
3. The receiver interrupt flag (RI) is set.

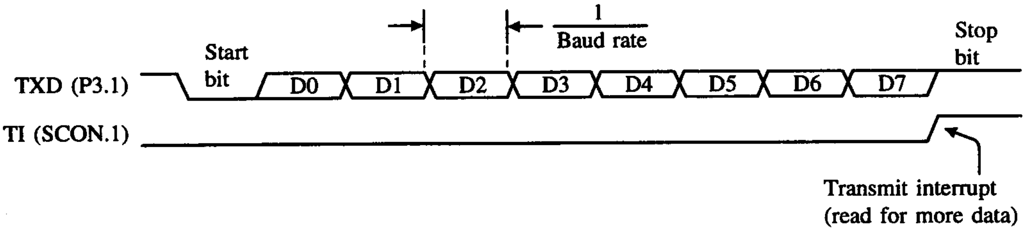


FIGURE 5-6
Setting the serial port T1 flag

These only occur, however, if the following conditions exist:

1. RI = 0, and
2. SM2 = 1 and the received stop bit = 1, or SM2 = 0.

The requirement that RI = 0 ensures that software has read the previous character (and cleared RI). The second condition sounds complicated but applies only in multiprocessor communications mode (see below). It implies, "Do not set RI in multiprocessor communications mode when the ninth data bit is 0."

5.5.3 9-Bit UART with Fixed Baud Rate (Mode 2)

When SM1 = 1 and SM0 = 0, the serial port operates in mode 2 as a 9-bit UART with a fixed baud rate. Eleven bits are transmitted or received: a start bit, eight data bits, a programmable ninth data bit, and a stop bit. On transmission, the ninth bit is whatever has been put in TB8 in SCON (perhaps a parity bit). On reception, the ninth bit received is placed in RB8. The baud rate in mode 2 is either 1/32nd or 1/64th the on-chip oscillator frequency. (See 5.9 Serial Port Baud Rates.)

5.5.4 9-Bit UART with Variable Baud Rate (Mode 3)

Mode 3, 9-bit UART with variable baud rate, is the same as mode 2 except the baud rate is programmable and provided by the timer. In fact, modes 1, 2, and 3 are very similar. The differences lie in the baud rates (fixed in mode 2, variable in modes 1 and 3) and in the number of data bits (eight in mode 1, nine in modes 2 and 3).

5.6 FULL DUPLEX SERIAL COMMUNICATION: ISSUES

The 8051's serial port allows for full duplex operation, which means that both transmission and reception of characters can be done simultaneously. However, there are some issues involved.

The first issue involves the physical connections. Full duplex communication must use two separate lines, one for transmission and one for reception, otherwise the signals from both parties would collide. In mode 0 of the serial port operation, only one line, the RXD, is used for both transmission and reception, hence full duplex cannot be achieved in this mode. In fact, as has been previously discussed in the section on the serial port modes of operation, mode 0 provides only half duplex operation. Meanwhile, all the other three modes use two separate lines, TXD and RXD for transmission and reception respectively, hence they allow for full duplex operation.

Secondly, the timing and synchronization must be considered. A common question asked by the student when told that the serial port can operate in full duplex is: "Suppose the serial port is transmitting a character to a serial device, so both the 8051 and the serial device are in synchronization. What would happen when in the midst of this, an incoming character is detected on the RXD line?" This poses an interesting question. For an answer to this, there is a need to understand the internal structure of the serial port. The 8051 serial port consists of two physically separate SBUF registers as shown in Figure 5-1. The transmit write-only register is clocked by a transmit baud rate clock while the receive read-only register is clocked by a receive baud rate clock. Both clocks are based on the overflow rate of Timer 1, as has been previously discussed (See Section 5.5.2 Mode 1.) Despite this fact, however, both baud rate clocks are separate and independent.

Transmission is initiated by writing to SBUF but only starts when the next rollover of the transmit baud rate clock is detected. Meanwhile, if any incoming character is detected, the 1-to-0 transition that denotes the start bit will immediately reset the receive baud rate clock so that the 8051 is synchronized with the device attached to the serial port.

Towards the end of this chapter, an example that demonstrates the full duplex operation of the serial port will be given.

5.7 INITIALIZATION AND ACCESSING SERIAL PORT REGISTERS

5.7.1 Receiver Enable

The receiver enable bit (REN) in SCON must be set by software to enable the reception of characters. This is usually done at the beginning of a program when the serial port, timers, etc., are initialized. This can be done in two ways. The instruction

```
SETB REN
```

explicitly sets REN, or the instruction

```
MOV SCON,#xxx1xxxxB
```

sets REN and sets or clears the other bits in SCON, as required. (The x's must be 1s or 0s to set the mode of operation.)

5.7.2 The Ninth Data Bit

The ninth data bit transmitted in modes 2 and 3 must be loaded into TB8 by software. The ninth data bit received is placed in RB8. Software may or may not require a ninth data bit, depending on the specifications of the serial device with which communications are established. (The ninth data bit also plays an important role in multiprocessor communications. See below.)

5.7.3 Adding a Parity Bit

A common use for the ninth data bit is to add parity to a character. As discussed in Chapter 2, the P bit in the program status word (PSW) is set or cleared every machine cycle to establish even parity with the eight bits in the accumulator. If, for example, communications require eight data bits plus even parity, the following instructions could be used to transmit the eight bits in the accumulator with even parity added in the ninth bit:

```
MOV C,P              ; PUT EVEN PARITY BIT IN TBS
MOV TBS,C            ; THIS BECOMES THE 9TH DATA BIT
MOV SBUF,A           ; MOVE S BITS FROM ACC TO SBUF
```

If odd parity is required, then the instructions must be modified as follows:

```
MOV C,P           ; PUT EVEN PARITY BIT IN C FLAG
CPL C             ; CONVERT TO ODD PARITY
MOV TB8,C
MOV SBUF,A
```

Of course, the use of parity is not limited to modes 2 and 3. In mode 1, the eight data bits transmitted can consist of seven data bits plus a parity bit. In order to transmit a 7-bit ASCII code with even parity in bit 8, the following instructions could be used:

```
CLR ACC.7         ; ENSURE MSB IS CLEAR
                  ; EVEN PARITY IS IN P
MOV C,P           ; COPY TO C
MOV ACC.7,C       ; PUT EVEN PARITY INTO MSB
MOV SBUF,A        ; SEND CHARACTER
                  ; 7 DATA BITS PLUS EVEN PARITY
```

5.7.4 Interrupt Flags

The receive and transmit interrupt flags (RI and TI) in SCON play an important role in 8051 serial communications. Both bits are set by hardware but must be cleared by software.

Typically, RI is set at the end of character reception and indicates "receive buffer full." This condition is tested in software or programmed to cause an interrupt. (Interrupts are discussed in Chapter 6.) If software wishes to input a character from the device connected to the serial port (perhaps a video display terminal), it must wait until RI is set, then clear RI and read the character from SBUF. This is shown below.

```
WAIT: JNB RI,WAIT ; CHECK RI UNTIL SET
      CLR RI      ; CLEAR RI
      MOV A,SBUF  ; READ CHARACTER
```

TI is set at the end of character transmission and indicates "transmit buffer empty." If software wishes to send a character to the device connected to the serial port, it must first check that the serial port is ready. In other words, if a previous character was sent, wait until transmission is finished before sending the next character. The following instructions transmit the character in the accumulator:

```
WAIT: JNB TI,WAIT ;CHECK TI UNTIL SET
      CLR TI      ;CLEAR TI
      MOV SBUF,A  ;SEND CHARACTER
```

The receive and transmit instruction sequences above are usually part of standard input character and output character subroutines. These are described in more detail in Example 5.2 and Example 5.3.

5.8 MULTIPROCESSOR COMMUNICATIONS

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, nine data bits are received and the ninth bit goes into RB8. The port can be programmed so

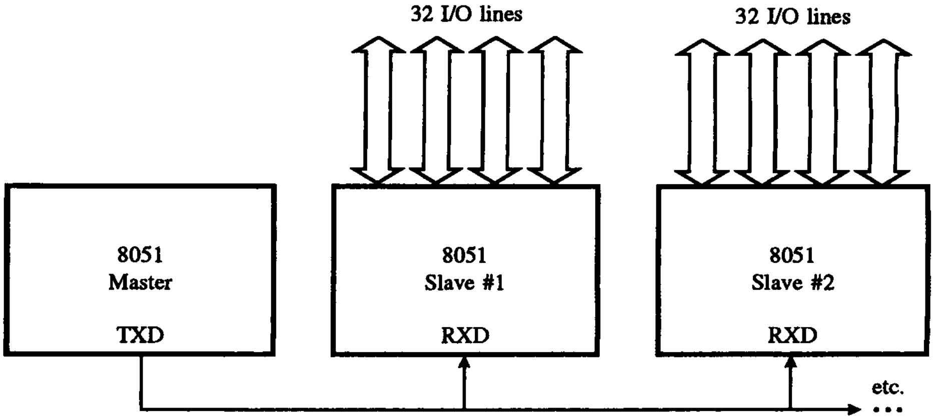


FIGURE 5-7
Multiprocessor communication

that when the stop bit is received, the serial port interrupt is activated only if RB8 = 1. This feature is enabled by setting the SM2 bit in SCON. An application of this is in a networking environment using multiple 8051s in a master/slave arrangement, as shown in Figure 5-7.

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte that identifies the target slave. An address byte differs from a data byte in that the ninth bit is 1 in an address byte and 0 in a data byte. An address byte, however, interrupts all slaves, so that each can examine the received byte to test if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that follow. The slaves that weren't addressed leave their SM2 bits set and go about their business, ignoring the incoming data bytes. They will be interrupted again when the next address byte is transmitted by the master processor. Special schemes can be devised so that once a master/slave link is established, the slave can also transmit to the master. The trick is not to use the ninth data bit after a link has been established (otherwise other slaves may be inadvertently selected).

SM2 has no effect in mode 0, and in mode 1 it can be used to check the validity of the stop bit. In mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

5.9 SERIAL PORT BAUD RATES

As evident in Table 5-2, the baud rate is fixed in modes 0 and 3. In mode 0 it is always the on-chip oscillator frequency divided by 12. Usually a crystal drives the 8051's on-chip oscillator, but another clock source can be used as well. (See Chapter 2.) Assuming a nominal oscillator frequency of 12 MHz, the mode 0 baud rate is 1 MHz. (See Figure 5-8a.)

By default following a system reset, the mode 2 baud rate is the oscillator frequency divided by 64. The baud rate is also affected by a bit in the power control register, PCON.

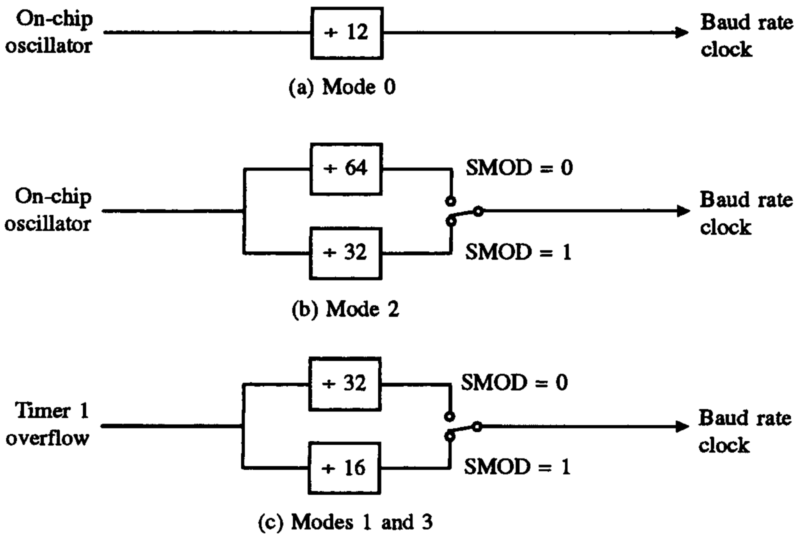


FIGURE 5-8

Serial port clocking sources (a) Mode 0 (b) Mode 2 (c) Modes 1 and 3

Bit 7 of PCON is the SMOD bit. Setting SMOD has the effect of doubling the baud rate in modes 1, 2, and 3. In mode 2, the baud rate can be doubled from a default value of 1/64th the oscillator frequency (SMOD = 0), to 1/32nd the oscillator frequency (SMOD = 1). (See Figure 5-8b.)

Since PCON is not bit-addressable, setting SMOD without altering the other PCON bits requires a "read-modify-write" operation. The following instructions set SMOD:

```
MOV  A,PCON         ;GET CURRENT VALUE OF PCON
SETB ACC.7          ;SET BIT 7 (SMOD)
MOV  PCON,A         ;WRITE VALUE BACK TO PCON
```

The 8051 baud rates in modes 1 and 3 are determined by the Timer 1 overflow rate. Since the timer operates at a relatively high frequency, the overflow is further divided by 32 (16 if SMOD = 1) before providing the baud rate clock to the serial port. The 8052 baud rate in modes 1 and 3 is determined by the Timer 1 or Timer 2 overflow rates, or both.

5.9.1 Using Timer 1 as the Baud Rate Clock

Considering only an 8051 for the moment, the usual technique for baud rate generation is to initialize TMOD for 8-bit auto-reload mode (timer mode 2) and put the correct reload value in TH1 to yield the proper overflow rate for the baud rate. TMOD is initialized as follows:

```
MOV TMOD,#0010xxxxB
```

The x's are 1s or 0s as needed for Timer 0.

This is not the only possibility. Very low baud rates can be achieved by using 16-bit mode, timer mode 2 with TMOD = 0001xxxxB. There is a slight software overhead, however, since the TH1/TL1 registers must be reinitialized after each overflow. This would be performed in an interrupt service routine. Another option is to clock Timer 1 externally using T1 (P3.5). Regardless the baud rate is the Timer 1 overflow rate divided by 32 (or divided by 16, if SMOD = 1).

The formula for determining the baud rate in modes 1 and 3, therefore, is

```
BAUD RATE = TIMER1 OVERFLOW RATE ÷ 32
```

For example, 1200 baud operation requires an overflow rate calculated as follows:

```
1200 = TIMER1 OVERFLOW RATE ÷ 32
TIMER1 OVERFLOW RATE = 38.4 kHz
```

If a 12 MHz crystal drives the on-chip oscillator, Timer 1 is clocked at a rate of 1 MHz or 1000 kHz. Since the timer must overflow at a rate of 38.4 kHz and the timer is clocked at a rate of 1000 kHz, an overflow is required every 1000 ÷ 38.4 = 26.04 clocks. (Round to 26.) Since the timer counts up and overflows on the 0FFH-to-00H transition of the count, 26 counts less than 0 is the required reload value for TH1. The correct value is -26. The easiest way to put the reload value into TH1 is

```
MOV TH1,#-26
```

The assembler will perform the necessary conversion. In this case -26 is converted to 0E6H; thus, the instruction above is identical to

```
MOV TH1,#0E6H
```

Due to rounding, there is a slight error in the resulting baud rate. Generally, a 5% error is tolerable using asynchronous (start/stop) communications. Exact baud rates are possible using an 11.059 MHz crystal. Table 5-3 summarizes the TH1 reload values for the most common baud rates, using a 12.000 MHz or 11.059 MHz crystal.

TABLE 5-3
Baud rate summary

Baud Rate	Crystal Frequency	SMOD	TH1 Reload Value	Actual Baud Rate	Error
9600	12.000 MHz	1	-7 (0F9H)	8923	7%
2400	12.000 MHz	0	-13 (0F3H)	2404	0.16%
1200	12.000 MHz	0	-26 (0E6H)	1202	0.16%
19200	11.059 MHz	1	-3 (0FDH)	19200	0
9600	11.059 MHz	0	-3 (0FDH)	9600	0
2400	11.059 MHz	0	-12 (0F4H)	2400	0
1200	11.059 MHz	0	-24 (0E8H)	1200	0

EXAMPLE 5.1 Initializing the Serial Port

Write an instruction sequence to initialize the serial port to operate as an 8-bit UART at 2400 baud. Use Timer 1 to provide the baud rate clock.

Solution

For this example, four registers must be initialized: SMOD, TMOD, TCON, and TH1. The required values are summarized below.

	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
SCON:	0	1	0	1	0	0	1	0
	GTE	C/T	M1	M0	GTE	C/T	M1	M0
TMOD:	0	0	1	0	0	0	0	0
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
TCON:	0	1	0	0	0	0	0	0
TH1:	1	1	1	1	0	0	1	1

Setting SM0/SM1 = 0/1 puts the serial port into 8-bit UART mode. REN = 1 enables the serial port to receive characters. Setting TI = 1 allows transmission of the first character by indicating that the transmit buffer is empty. For TMOD, setting M1/M0 = 1/0 puts Timer 1 into 8-bit auto-reload mode. Setting TR1 = 1 in TCON turns on Timer 1. The other bits are shown as 0s, since they control features or modes not used in this example.

Discussion

The required TH1 value is that which provides overflows at the rate of 2400 x 32 = 76.8 kHz. Assuming the 8051 is clocked from a 12 MHz crystal, Timer 1 is clocked at a rate of 1 MHz or 1000 kHz, and the number of clocks for each overflow is 1000 ÷ 76.8 = 13.02. (Round to 13.) The reload value is -13 or 0F3H.

The initialization instruction sequence is shown below.

Solution to Example 5.1 8051 Serial Port Example (initialize the serial port)

```
8100                  ORG   8100H
8100  759852   INIT:  MOV   SCON,#52H  ;serial port,mode 1
8103  758920          MOV   TMOD,#20H  ;timer 1, mode 2
8106  758DF3          MOV   TH1,#-13   ;reload count for 2400 baud
8109  D28E            SETB  TR1        ;start timer 1
                      END
```

EXAMPLE 5.2 Output Character Subroutine

Write a subroutine called OUTCHR to transmit the 7-bit ASCII code in the accumulator out the 8051 serial port, with odd parity added as the eighth bit. Return from the subroutine with the accumulator intact, i.e., containing the same value as before the subroutine was called.

Solution

This example and the next illustrate two of the most common subroutines on microcomputer systems with an attached RS232 terminal: output character (OUTCHR) and input character (INCHAR).

```
8100                       ORG 8100H
8100   A2D0        OUTCHR: MOV   C.P        ;put parity bit in C flag
8102   B3                  CPL   C          ;change to odd parity
8103   92E7                MOV   ACC.7.C    ;add to character code
8105   3099FD       AGAIN: JNB   TI.AGAIN    ;Tx empty? no:check again
8108   C299                CLR   TI         ;yes: clear flag and
810A   F599                MOV   SBUF.A     ;send character
810C   C2E7                CLR   ACC.7      ;strip off parity bit and
810E 22                    RET              ; return
                           END
```

Discussion

The first three instructions place odd parity in the accumulator bit 7. Since the P bit in the PSW establishes even parity with the accumulator, it is complemented before being placed in ACC.7. The JNB instruction creates a "wait loop," repeatedly testing the transmit interrupt flag (TI) until it is set. When TI is set (because the previous character transmission is finished), it is cleared and then the character in the accumulator is written into the serial port buffer (SBUF). Transmission begins on the next rollover of the divide-by-l6 counter that clocks the serial port. (See Figure 5-5.) Finally, ACC.7 is cleared so that the return value is the same as the 7-bit code passed to the subroutine.

The OUTCHR subroutine is a building block and is of little use by itself. At a "higher level" this subroutine is called to transmit a single character or a string of characters. For example, the following instructions transmit the ASCII code for the letter "Z" to the serial device attached to the 8051's serial port:

```
MOV A.#'Z'

CALL OUTCHR

(continue)
```

As a natural extension to this idea, Problem 1 at the end of this chapter uses OUTCHR as a building block in an OUTSTR (output string) subroutine that transmits a sequence of ASCII codes (terminated by a NULL byte, 00H) to the serial device attached to the 8051's serial port.

EXAMPLE 5.3 Input Character Subroutine

Write a subroutine called INCHAR to input a character from the 8051's serial port and return with the 7-bit ASCII code in the accumulator. Expect odd parity in the eighth bit received and set the carry flag if there is a parity error.

Solution

```
8100                       ORG  8100H
8100   3098FD   INCHAR:    JNB  RI,$      ;wait for character
8103   C298                CLR  RI        ;clear flag
8105   E599                MOV  A,SBUF    ;read char into A
8107   A2D0                MOV  C,P       ;for odd parity in A,
                                          ;P should be set
8109   B3                  CPL  C         ;complementing correctly
                                          ;indicates if "error"
810A   C2E7                CLR  ACC.7     ;strip off parity
810C   22                  RET
                           END
```

Discussion

This subroutine begins by waiting for the receive interrupt flag (RI) to be set, indicating that a character is waiting in SBUF to be read. When RI is set, the JNB instruction falls through to the next instruction. RI is cleared and the code in SBUF is read into the accumulator. The P bit in the PSW establishes even parity with the accumulator, so it should be set if the accumulator, on its own, correctly contains odd parity in bit 7. Moving the P bit into the carry flag and complementing it leaves CY = 0 if there is no error. On the other hand, if the accumulator contains a parity error, then CY = 1, correctly indicating "parity error." Finally, ACC.7 is cleared to ensure that only a 7-bit code is returned to the calling program.

EXAMPLE 5.4 **Full Duplex Operation**

Write a program that continually transmits characters from a transmit buffer (internal RAM 30H to 4FH). If incoming characters are detected on the serial port, store them in the receive buffer starting at internal RAM location 50H. Assume that the 8051 serial port has already been initialized in mode H.

Solution

This example shows how the serial port can be used in full duplex operation, i.e., transmission and reception are done simultaneously.

```
8100               ORG 8100H
8100 7830          MOV R0, #30H       ;pointer for tx buffer
8102 7950          MOV R1, #50H       ;pointer for rx buffer
8104 209819  LOOP: JB RI, RECEIVE     ;character received?
                                      ; yes: process it
8107 209902        JB TI, TX          ;previous character
                                      ; transmitted?
                                      ; yes: process it
810A 80F8          SJMP LOOP          ; no: continue checking
810C E6      TX:   MOV A, @R0         ;get character from tx
                                      ; buffer
```

```
810D A2D0        MOV C, P            ;put parity bit in C
810F B3          CPL C               ;change to odd parity
8110 92E7        MOV ACC.7, C        ;add to character code
8112 C299        CLR TI              ;clear transmit flag
8114 F599        MOV SBUF, A         ;send character
8116 C2E7        CLR ACC.7           ;strip off parity bit
8118 08          INC R0              ;point to next
                                     ; character in buffer
8119 B850E8      CJNE R0, #50H, LOOP ;end of buffer?
                                     ; no: continue
811C 7830        MOV R0, #30H        ; yes: recycle
811E 80E4        SJMP LOOP           ;continue checking

8120 C298   RX:  CLR RI              ;clear receive flag
8122 E599        MOV A, SBUF         ;read character into A
8124 A2D0        MOV C, P            ;for odd parity in A,
                                     ; P should be set
8126 B3          CPL C               ;complementing
                                     ; correctly indicates
                                     ; “error”
8127 C2E7        CLR ACC.7           ;strip off parity
8129 F7          MOV @R1, A          ;store received
                                     ; character in buffer
812A 09          INC R1              ;point to next location
                                     ; in buffer
812B 80D7        SJMP LOOP           ;continue checking
                 END
```

Discussion

This program first uses the two registers R0 and R1 to point to the transmit and receive buffers respectively. It then checks to see if a character has been received or if the previous character to be transmitted has already been sent out. Notice that reception is processed first. This is because reception is more critical since the 8051 is depending on an external device for an incoming character, which must be processed and stored as soon as possible else it might be overwritten by subsequent incoming characters. Recall that the serial port can hold at most one character in its buffer while a second is being received. If an incoming character has been fully received, it is read from SBUF and checked for parity before being stored into the receive buffer whose current empty location is pointed to by R1. R1 is next incremented to point to the next available location and the program goes back to checking. When the previous transmission is finished, the program first gets the character to be transmitted from the transmit buffer. R0 is used as the pointer to the next character in the transmit buffer. The odd parity is then placed in bit 7 of the code before it is sent to the SBUF for transmission. R1 is then incremented to point to the next character. The program also checks for the end of the transmit buffer, upon which it would recycle back to the beginning of the buffer. Finally, the program goes back to checking for further receptions or transmission.

EXAMPLE 5.5 **Compensating for the Round-off Errors**

Calculate the round-off error for the baud rate of the serial port in Example 5.1. Assuming the same Timer 1 reload value of -13 or 0F3H is to be used, calculate the value of the crystal frequency for an exact baud rate of 2400 baud.

Solution

The round-off error for the baud rate is:

$$\begin{aligned}\text{Round-off error} &= \frac{|\text{baud}_{\text{desired}} - \text{baud}_{\text{rounded-off}}|}{\text{baud}_{\text{desired}}} \times 100\% \\ &= \frac{|2400 - 2403.8|}{2400} \times 100\% \\ &= 0.158\%\end{aligned}$$

$$\begin{aligned}f_{\text{desired}} &= \frac{\text{baud}_{\text{desired}}}{\text{baud}_{\text{rounded-off}}} \times f_{\text{rounded-off}} \\ &= \frac{2400}{2403.8} \times 12 \text{ MHz} \\ &= 11.98 \text{ MHz}\end{aligned}$$

Discussion

The desired baud rate is 2400 baud. The rounded-off Timer 1 reload value is -13 for 13 µs, causing an overflow rate of 76.9kHz, which in turn causes a baud rate of 76.9 kHz ÷ 32 = 2403.8 baud.

In the calculation of the desired crystal frequency to produce an exact baud rate of 2400, we can use the ratio technique, as shown in the solution above. Let's check if causes an exact 2400 baud rate. A crystal frequency of 11.98MHz means that Timer 1 increments at a rate of 11.98 MHz÷12 = 0.998 MHz so the duration of one count is 1/0.998 MHz =1.002 *µs*. The reload value for Timer 1 is -13 so it overflows every 13 x 1.002 µS =1.3026 µs, hence the overflow rate is 1/1.3026 µs = 76.77 kHz. This causes a baud rate of 76.77 kHz ÷ 32 = 2399 baud 2400 baud.

A more commonly used crystal frequency is 11.059 MHz rather than 11.98MHz. In this case, the Timer 1 reload value would have to be -12 instead of -13. To check that it results in an exact baud rate of 2400, note that Timer 1 increments at a rate of 11.059 MHz÷12 = 0.9216 MHz so the duration of one count is 1/0.9216 MHz = 1.085 µs. The reload value for Timer 1 is -12 so it overflows every 12 x 1.085 µs = 13.02 µs, hence the overflow rate is 1/13.02 µs= 76.8 kHz. This causes a baud rate of 76.8 kHz / 32 = 2400 baud.

It is clear then that the crystal frequency to eliminate round-off errors in the baud rate would vary depending on the Timer 1 reload value used.

SUMMARY

This chapter has presented the major details required to program the 8051 serial port. A passing mention has been made in this chapter and in the last chapter of the use of interrupts. Indeed, advanced applications using the 8051 timers or serial ports generally require input/output operations to be synchronized by interrupts. This is the topic of the next chapter.

PROBLEMS

The following problems are typical of the software routines for interfacing terminals (or other serial devices) to a microcomputer. Assume the 8051 serial port is initialized in 8-bit UART mode and the baud rate is provided by Timer 1.

5.1 Write a subroutine called OUTSTR that sends a null-terminated string of ASCII codes to the device (perhaps a VDT) connected to the 8051 serial port. Assume the string of ASCII codes is in external code memory and the calling program puts the address of the string in the data pointer before calling OUTSTR. A null-terminated string is a series of ASCII bytes terminated with a 00H byte.

5.2 Write a subroutine called INLINE that inputs a line of ASCII codes from the device connected to the 8051 serial port and places it in internal data memory beginning at address 50H. Assume the line is terminated with a carriage return code. Place the carriage return code in the line buffer along with the other codes, and then terminate the line buffer with a null byte (00H).

5.3 Write a program that continually sends the alphabet (lowercase) to the device attached to the 8051 serial port. Use the OUTCHR subroutine written earlier.

5.4 Assuming the availability of the OUTCHR subroutine, write a program that continually sends the displayable ASCII set (codes 20H to 7EH) to the device attached to the 8051 serial port.

5.5 Modify the solution to the above problem to suspend and resume output to the screen, using XOFF and XON codes entered on the keyboard. All other codes received should be ignored. (Note: XOFF = CONTROL-S = 13H, XON = CONTROL-Q = 11H)

5.6 Assume the availability of the INCHAR and OUTCHR subroutines and write a program that inputs characters from the keyboard and echoes them back to the screen, converting lowercase characters to uppercase.

5.7 Assume the availability of the INCHAR and OUTCHR subroutines and write a program that inputs characters from the device attached to the 8051 serial port and echoes them back substituting period (.) for any control characters (ASCII codes 00H to 1FH, and 7FH).

5.8 Assume the availability of the OUTCHR subroutine, and write a program that clears the screen on the VDT attached to the 8051 serial port and then sends your name to the VDT 10 times on 10 separate lines. The clear screen function on VDTs is accomplished by transmitting a CONTROL-Z on many terminals or <ESC> [2 J on terminals that support ANSI (American National Standards Institute) escape sequences. Use either method in your solution.

5.9 Figure 5-4 illustrates a technique for expanding the output capability of the 8051. Assuming such a configuration, write a program that initializes the 8051 serial port for shift register mode and then maps the contents of internal memory location 20H to the eight extra outputs, 10 times per second.

5.10 Which type of serial communication does the 8051's serial port support? Simplex, half- or full duplex? Why? Explain the components within the serial port and how they support this type of serial communication.

5.11 Explain how the 8051's input/output (I/O) capability can be expanded by using the serial port. What is the disadvantage of such a method?

5.12 Write a subroutine called OUTCHR9 to transmit a 9-bit code ($C_8C_7C_6C_5C_4C_3$- $C_2C_1C_0$) out the 8051 serial port. Note that C_8 is in the LSB of B register while $C_7C_6C_5C_4C_3C_2C_1C_0$ are in the accumulator. Return from the subroutine with all registers intact.

5.13 Write a subroutine called INCHAR8 to input an 8-bit extended ASCII character from the 8051's serial port and return with the 8-bit code in the accumulator. Expect odd parity in the ninth bit received and set the carry flag if there is a parity error.

5.14 Write a subroutine called OUTCHR8 to transmit the 8-bit extended ASCII code in the accumulator out the 8051 serial port, with odd parity added as the ninth bit. Return from the subroutine with the accumulator intact.

5.15 a. Write an instruction sequence to initialize the serial port to operate as a 9-bit UART at 9600 baud. Use Timer 1 to provide the baud rate clock, assuming that the crystal oscillator frequency, f_{osc} = 12 MHz.

b. Is the resulting baud rate exactly 9600 baud? If not, what is the percentage error (% error)? Hence, calculate the value of $f_o$sc to achieve an exact baud rate of 9600 baud.

CHAPTER 6

Interrupts

6.1 INTRODUCTION

An **interrupt** is the occurrence of a condition—an event—that causes a temporary suspension of a program while the condition is serviced by another program. Interrupts play an important role in the design and implementation of microcontroller applications. They allow a system to respond asynchronously to an event and deal with the event while another program is executing. An **interrupt-driven system** gives the illusion of doing many things simultaneously. Of course, the CPU cannot execute more than one instruction at a time; but it can temporarily suspend execution of one program, execute another, then return to the first program. In a way, this is like a subroutine. The CPU executes another program—the subroutine—and then returns to the original program. The difference is that in an interrupt-driven system, the interruption is a response to an "event" that occurs asynchronously with the main program. It is not known when the main program will be interrupted.

The program that deals with an interrupt is called an **interrupt service routine (ISR)** or **interrupt handler.** The ISR executes in response to the interrupt and generally performs an input or output operation to a device. When an interrupt occurs, the main program temporarily suspends execution and branches to the ISR; the ISR executes, performs the operation, and terminates with a "return from interrupt" instruction; the main program continues where it left off. It is common to refer to the main program as executing at **base-level and** the ISRs as executing at **interrupt-level.** The terms **foreground** (base-level) and **background** (interrupt-level) are also used. This brief view of interrupts is depicted in Figure 6-1, showing (a) the execution of a program without interrupts and (b) execution at base-level with occasional interrupts and ISRs executing at interrupt-level.

A typical example of interrupts is manual input, using a keyboard. Consider an application for a microwave oven. The main program (foreground) might control a microwave power element for cooking; yet, during cooking, the system must respond to manual input

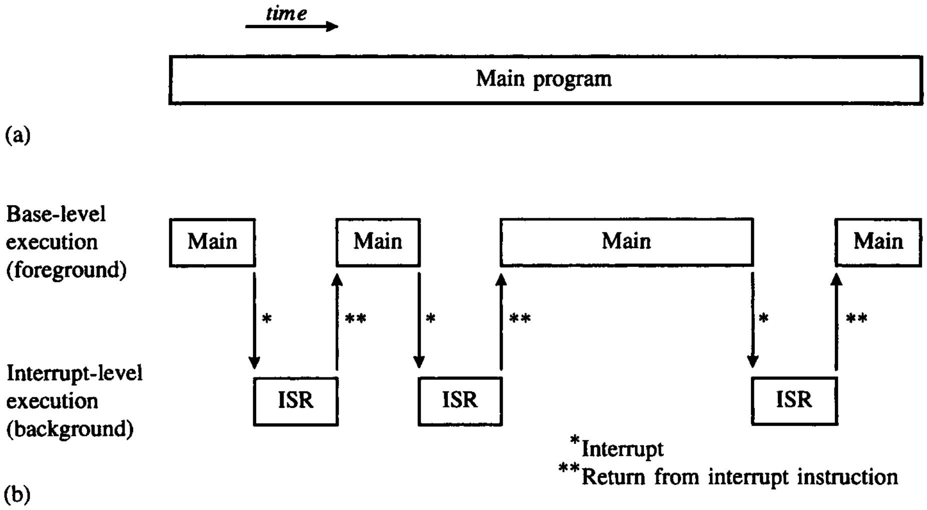

time
Main program
(a)
Base-level execution (foreground)
Main
Main
Main
Main
*
**
*
**
*
**
Interrupt-level execution (background)
ISR
ISR
ISR
*Interrupt
**Return from interrupt instruction
(b)

FIGURE 6-1
Program execution with and without interrupts (a) Without interrupts (b) With interrupts

on the oven's door, such as a request to shorten or lengthen the cooking time. When the user depresses a key, an interrupt is generated (a signal goes from high to low, perhaps) and the main program is interrupted. The ISR takes over in the background, reads the keyboard code(s) and changes the cooking conditions accordingly, and finishes by passing control back to the main program. The main program carries on where it left off. The important point in this example is that manual input occurs "asynchronously"; that is, it occurs at intervals not predictable or controlled by the software running in the system. This is an interrupt.

6.2 8051 INTERRUPT ORGANIZATION

There are five interrupt sources on the 8051: two external interrupts, two timer interrupts, and a serial port interrupt. The 8052 adds a sixth interrupt source from the extra timer. All interrupts are disabled after a system reset and are enabled individually by software.

In the event of two or more simultaneous interrupts or an interrupt occurring while another interrupt is being serviced, there is both a polling sequence and a two-level priority scheme to schedule the interrupts. The polling sequence is fixed but the interrupt priority is programmable.

Let's begin by examining ways to enable and disable interrupts.

6.2.1 Enabling and Disabling Interrupts

Each of the interrupt sources is individually enabled or disabled through the bit-addressable special function register IE (interrupt enable) at address 0A8H. As well as individual enable

TABLE 6-1
IE (interrupt enable) register summary

Bit	Symbol	Bit Address	Description (1 = Enable, 0 = Disable)
IE.7	EA	AFH	Global enable/disable
IE.6	—	AEH	Undefined
IE.5	ET2	ADH	Enable Timer 2 interrupt (8052)
IE.4	ES	ACH	Enable serial port interrupt
IE.3	ET1	ABH	Enable Timer 1 interrupt
IE.2	EX1	AAH	Enable external 1 interrupt
IE.1	ET0	A9H	Enable Timer 0 interrupt
IE.0	EX0	A8H	Enable external 0 interrupt

bits for each interrupt source, there is a global enable/disable bit that is cleared to disable all interrupts or set to turn on interrupts. (See Table 6-1.)

Two bits must be set to enable any interrupt: the individual enable bit and the global enable bit. For example, timer 1 interrupts are enabled as follows:

```
SETB ET1         ;ENABLE Timer 1 INTERRUPT
SETB EA          ;SET GLOBAL ENABLE BIT
```

This could also be coded as

```
MOV  IE,#10001000B
```

Although these two approaches have exactly the same effect following a system reset, the effect is different if IE is written "on-the-fly" in the middle of a program. The first approach has no effect on the other five bits in the IE register, whereas the second approach explicitly clears the other bits. It is fine to initialize IE with a "move byte" instruction at the beginning of a program (i.e., following a power-up or system reset), but enabling and disabling interrupts on-the-fly within a program should use "set bit" and "clear bit" instructions to avoid side effects with other bits in the IE register.

6.2.2 Interrupt Priority

Each interrupt source is individually programmed to one of two priority levels through the bit-addressable special function register IP (interrupt priority) at address 0B8H. (See Table 6-2.)

IP is cleared after a system reset to place all interrupts at the lower priority level by default. The idea of "priorities" allows an ISR to be interrupted by an interrupt if the new interrupt is of higher priority than the interrupt currently being serviced. This is straightforward on the 8051, since there are only two priority levels. If a low-priority ISR is executing when a high-priority interrupt occurs, the ISR is interrupted. A high-priority ISR cannot be interrupted.

TABLE 6-2
IP (interrupt priority) register summary

Bit	Symbol	Bit Address	Description (1 = Higher Level, 0 = Lower Level)
IP.7	—	—	Undefined
IP.6	—	—	Undefined
IP.5	PT2	0BDH	Priority for Timer 2 interrupt (8052)
IP.4	PS	0BCH	Priority for serial port interrupt
IP.3	PT1	0BBH	Priority for Timer 1 interrupt
IP.2	PX1	0BAH	Priority for external 1 interrupt
IP.1	PT0	0B9H	Priority for Timer 0 interrupt
IP.0	PX0	0B8H	Priority for external 0 interrupt

The main program, executing at base level and not associated with any interrupt, can always be interrupted regardless of the priority of the interrupt. If two interrupts of different priorities occur simultaneously, the higher priority interrupt will be serviced first.

6.2.3 Polling Sequence

If two interrupts of the same priority occur simultaneously, a fixed polling sequence determines which is serviced first. The polling sequence is external 0, Timer 0, external 1, Timer 1, Serial Port, Timer 2.

Figure 6-2 illustrates the five interrupt sources, the individual and global enable mechanism, the polling sequence, and the priority levels. The state of all interrupt sources is available through the respective flag bits in the SFRs. Of course, if any interrupt is disabled, an interrupt does not occur, but software can still test the interrupt flag. The timer and serial port examples in the previous two chapters used the interrupt flags extensively without actually using interrupts.

A serial port interrupt results from the logical OR of a receive interrupt (8052) or a transmit interrupt (TI). Likewise, Timer 2 interrupts are generated by a time overflow (TF2) or by the external input flag (EXF2). The flag bits that generate interrupts are summarized in Table 6-3.

TABLE 6-3
Interrupt flag bits

Interrupt	Flag	Sfr Register and Bit Position
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 1	TF1	TCON.7
Timer 0	TF0	TCON.5
Serial port	T1	SCON.1
Serial port	RI	SCON.0
Timer 2	TF2	T2CON.7 (8052)
Timer 2	EXF2	T2CON.6 (8052)

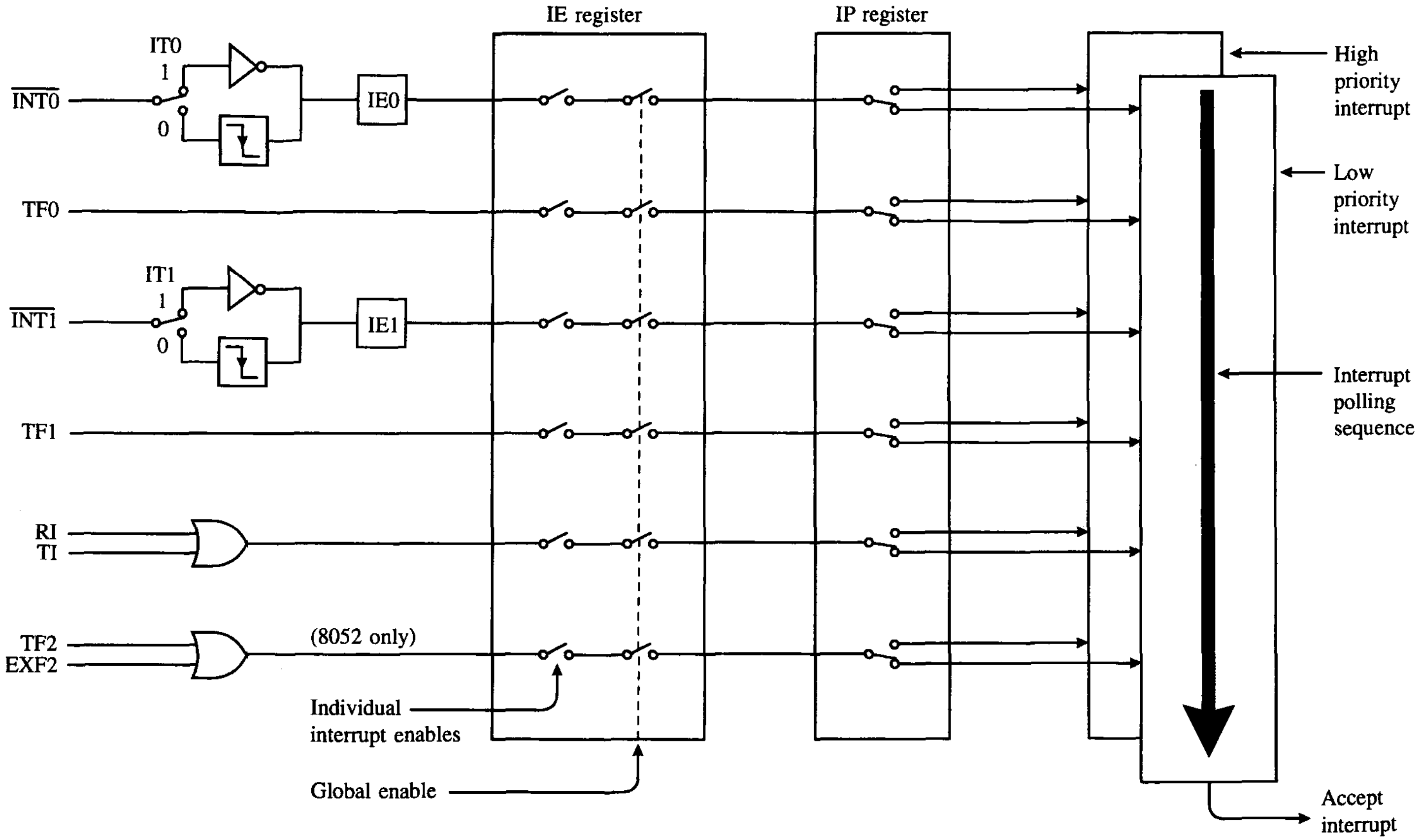


FIGURE 6–2
Overview of 8051 interrupt structure

6.3 PROCESSING INTERRUPTS

When an interrupt occurs and is accepted by the CPU, the main program is interrupted. The following actions occur:

- The current instruction completes execution.
- The PC is saved on the stack.
- The current interrupt status is saved internally.
- Interrupts are blocked at the level of the interrupt.
- The PC is loaded with the vector address of the ISR.
- The ISR executes.

The ISR executes and takes action in response to the interrupt. The ISR finishes with a RETI (return from interrupt) instruction. This retrieves the old value of the PC from the stack and restores the old interrupt status. Execution of the main program continues where it left off.

6.3.1 Interrupt Vectors

When an interrupt is accepted, the value loaded into the PC is called the **interrupt vector.** It is the address of the start of the ISR for the interrupting source. The interrupt vectors are given in Table 6-4.

The system reset vector (RST at address 0000H) is included in this table, since, in this sense, it is like an interrupt: it interrupts the main program and loads the PC with a new value.

When "vectoring to an interrupt," the flag that caused the interrupt is automatically cleared by hardware. The exceptions are RI and TI for serial port interrupts, and TF2 and EXF2 for Timer 2 interrupts. Since there are two possible sources for each of these interrupts, it is not practical for the CPU to clear the interrupt flag. These bits must be tested in the ISR to determine the source of the interrupt, and then the interrupting flag is cleared by software. Usually a branch occurs to the appropriate action, depending on the source of the interrupt.

Since the interrupt vectors are at the bottom of code memory, the first instruction of the main program is often a jump above this area of memory, such as LJMP 0030H.

TABLE 6-4
Interrupt vectors

Interrupt	Flag	Vector Address
System reset	RST	0000H
External 0	IE0	0003H
Timer 0	TF0	000BH
External 1	IE1	0013H
Timer 1	TF0	001BH
Serial port	RI or TI	0023H
Timer 2	TF2 or EXF2	002BH

6.4 PROGRAM DESIGN USING INTERRUPTS

The examples in Chapter 3 and Chapter 4 did not use interrupts but made extensive use of "wait loops" to test the timer overflow flags (TF0, TF1, or TF2) or the serial port transmit and receive flags (TI or RI). The problem in this approach is that the CPU's valuable execution time is fully consumed waiting for flags to be set. This is inappropriate for control-oriented applications where a microcontroller must interact with many input and output devices simultaneously.

In this section, examples are developed to demonstrate practical methods for implementing software for control-oriented applications. The key ingredient is the interrupt. Although the examples are not necessarily bigger, they are more complex, and in recognition of this, we proceed one step at a time. The reader is advised to follow the examples slowly and to examine the software meticulously. Some of the most difficult bugs in system designs often involve interrupts. The details must be understood thoroughly.

Since we are using interrupts, the examples will be complete and self-contained. Each program starts at address 0000H with the assumption that it begins execution following a system reset. The idea is that eventually these programs develop into full-fledged applications that reside in ROM or EPROM.

The suggested framework for a self-contained program using interrupts is shown below.

```
        ORG  0000H     ;RESET ENTRY POINT
        LJMP MAIN
        .              ;ISR ENTRY POINTS
        .
        .
        .
        ORG 0030H      ;MAIN PROGRAM ENTRY POINT
MAIN:   .              ;MAIN PROGRAM BEGINS
        .
        .
```

The first instruction jumps to address 0030H, just above the vector locations where the ISRs begin, as given in Table 6-4. As shown in Figure 6-3, the main program begins at address 0030H.

6.4.1 Small Interrupt Service Routines

Interrupt service routines must begin near the bottom of code memory at the addresses shown in Table 6-4. Although there are only eight bytes between each interrupt entry point, this is often enough memory to perform the desired operation and return from the ISR to the main program.

If only one interrupt source was used, say Timer 0, then the following framework could be used:

```
ORG   0000H   ;RESET
LJMP  MAIN
ORG   000BH   ;Timer 0 ENTRY POINT
```

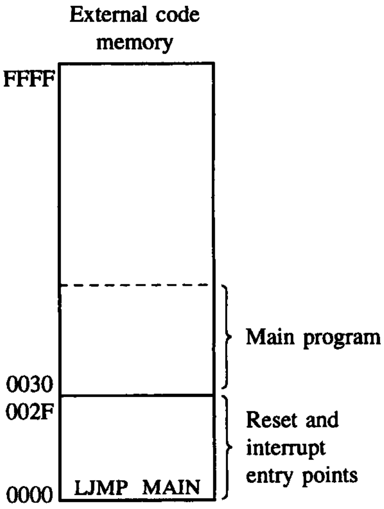


FIGURE 6-3
Memory organization when interrupts are used

```
T0ISR:      .                         ;Timer 0 ISR BEGINS
            .
            RETI                      ;RETURN TO MAIN PROGRAM
MAIN:       .                         ;MAIN PROGRAM
            .
```

If more interrupts are used, care must be taken to ensure they start at the correct location (see Table 6-4) and do not overrun the next ISR. Since only one interrupt is used in the example above, the main program can begin immediately after the RETI instruction.

6.4.2 Large Interrupt Service Routines

If an ISR is longer than eight bytes, it may be necessary to move it elsewhere in code memory or it may trespass on the entry point for the next interrupt. Typically, the ISR begins with a jump to another area of code memory where the ISR can stretch out. Considering only Timer 0 for the moment, the following framework could be used:

```
            ORG     000H      ;RESET ENTRY POINT
            LJMP    MAIN
            ORG     000BH     ;Timer 0 ENTRY POINT
            LJMP    T0ISR
            ORG     0030H     ;ABOVE INTERRUPT VECTORS
MAIN:       .
            .
            .
T0ISR:      .
            .                 ;Timer 0 ISR
            .
            RETI              ;RETURN TO MAIN PROGRAM
```

To keep it simple, our programs will only do one thing at a time initially. The main or foreground program initializes the timer, serial port, and interrupt registers as appropriate, and then does nothing. The work is done totally in the ISR. After the initialize instructions, the main program consists of the following instruction:

```
HERE:      SJMP HERE
```

When an interrupt occurs, the main program is interrupted temporarily while the ISR executes. The RETI instruction at the end of the ISR returns control to the main program, and it continues doing nothing. This is not as farfetched as one might think. In control-oriented applications, the bulk of the work is in fact done in the interrupt routines.

6.5 TIMER INTERRUPTS

Timer interrupts occur when the timer overflow flag, TFx, is set upon overflow of the timer registers, THx/TLx. When the 8051 goes to service this interrupt, the timer overflow flag, TFx is automatically cleared by hardware. Therefore, with interrupts enabled, there is no need to explicitly clear TFx in software as was done in Chapter 4 where timers were used without interrupts.

EXAMPLE 6.1 **Square Wave Using Timer Interrupts**

Write a program using Timer 0 and interrupts to create a 10 kHz square wave on P1.0.

Solution

```
0000          5          ORG  0           ;reset entry point
0000 020030   6          LJMP MAIN        ;jump above interrupt vectors
000B          7          ORG  000BH       ;Timer 0 interrupt vector
000B B290     8  T0ISR:  CPL  P1.0        ;toggle port bit
000D 32       9          RETI
0030         10          ORG  0030H       ;Main program entry point
0030 758902  11  MAIN:   MOV  TMOD,#02H   ;timer 0, mode 2
0033 758CCE  12          MOV  TH0,#-50    ;50 us delay
0036 D28C    13          SETB TR0         ;start timer
0038 75A882  14          MOV  IE,#82H     ;enable timer 0 interrupt
003B 80FE    15          SJMP $           ;do nothing
```

Discussion

With timer interrupts enabled, the event that generates the interrupt is the setting of the timer flag, TFx, upon overflow of the timer registers, THx/TLx. This example appears in Chapter 4 without using interrupts. The bulk of the program is the same except it is now organized into the framework for interrupts.

The solution is a complete program. It could be burned into EPROM and installed in an 8051 single-board computer for execution. Immediately after reset, the program counter is loaded with 0000H. The first instruction executed is LJMP MAIN, which branches over the timer ISR to address 0030H in code memory. The next three instructions (lines 11-13) initialize Timer 0 for 8-bit auto-reload mode with overflows every 50 µs. The MOV instruction in line 14 enables Timer 0 interrupts, so each overflow of the timer generates an interrupt. Of course, the first overflow will not occur for 50 µs, so the main program falls through to the "do-nothing" loop. Each 50 µs an interrupt occurs; the main program is interrupted and the Timer 0 ISR executes. The ISR simply complements the port bit (line 8) and returns to the main program (line 9) where the do-nothing loop executes for another 50 µs.

Note that the timer flag, TF0, is not explicitly cleared by software. When interrupts are enabled, TF0 is automatically cleared by hardware when the CPU services the interrupt.

Since the solution is a "complete" program, we must be aware of how the stack is operating. The return address for the ISR is the location of the SJMP instruction. This address is pushed on the 8051's internal stack prior to vectoring to the interrupt and is popped from the stack when the RETI instruction executes (line 9). Since the SP was not initialized, it defaults to the reset value of 07H. The push operation leaves the return address in internal RAM locations 08H (PC_H) and 09H (PC_L).

EXAMPLE 6.2 **Two Square Waves Using Interrupts**

Write a program using interrupts to simultaneously create 7 kHz and 500 Hz square waves on P1.7 and P1.6.

Solution

The hardware configuration with the timings for the desired waveforms is shown in Figure 6-4.

This combination of outputs would be extremely difficult to generate on a non-interrupt-driven system. Timer 0, providing synchronization for the 7 kHz signal, operates in mode 2, as in the previous example; and timer 1, providing synchronization for the 500 Hz signal, operates in mode 1, 16-bit timer mode. Since 500 Hz requires a high-time of 1 ms

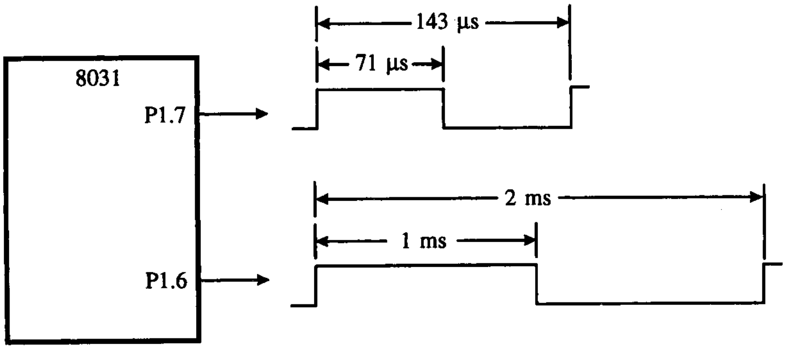


FIGURE 6-4
Waveform example

and low-time of 1 ms, mode 2 cannot be used. (Recall that 256 ms is the maximum timed interval in mode 2 when the 8051 is operating at 12 MHz.) Here's the program:

```
0000                    ORG   0
0000  020030            LJMP  MAIN
000B                    ORG   000BH           ;Timer 0 vector address
000B  02003F            LJMP  T0ISR
001B                    ORG   001BH           ;Timer 1 vector address
001B  020042            LJMP  T1ISR
0030                    ORG   0030H
0030  758912    MAIN:   MOV   TMOD,#12H       ;Timer 1 = mode 1
                                              ;Timer 0 = mode 2
0033  758CB9            MOV   TH0,#-71        ;7 kHz using timer 0
0036  D28C              SETB  TR0
0038  D28F              SETB  TF1             ;force timer 1 interrupt
003A  75A88A            MOV   IE,#8AH         ;enable both timer interrupts
003D  80FE              SJMP  $
                 ;
003F  B297      T0ISR:  CPL   P1.7
0041  32                RETI
0042  C28E      T1ISR:  CLR   TR1
0044  758DFC            MOV   TH1,#HIGH(-1000);1 ms high time &
0047  758B18            MOV   TL1,#LOW(-1000) ; low time
004A  D28E              SETB  TR1
004C  B296              CPL   P1.6
004E  32                RETI
                        END
```

Discussion

Again, the framework is for a complete program that could be installed in EPROM or ROM on an 8051-based product. The main program and the ISRs are located above the vector locations for the system reset and interrupts. Both waveforms are created by "CPL bit" instructions; however, the timed intervals necessitate a slightly different approach for each.

Since the TL1/TH1 registers must be reloaded after each overflow (i.e., after each interrupt), Timer 1 ISR (a) stops the timer, (b) reloads TL1/TH1, (c) starts the timer, and then (d) complements the port bit. Note also that TL1/TH1 are *not* initialized at the beginning of the main program, unlike TH0. Since TL1/TH1 must be reinitialized after each overflow, TF1 is set in the main program by software to "force" an initial interrupt as soon as interrupts are turned on. This effectively gets the 500 Hz waveform started.

The Timer 0 ISR, as in the previous example, simply complements the port bit and returns to the main program. SJMP $ is used in the main program as the abbreviated form of HERE: SJMP HERE. The two forms are functionally equivalent. (See "Special Assembler Symbols" in Chapter 7.)

6.6 SERIAL PORT INTERRUPTS

Serial port interrupts occur when either the transmit interrupt flag (TI) or the receive interrupt flag (RI) is set. A transmit interrupt occurs when transmission of the previous character written to SBUF has finished. A receive interrupt occurs when a character has been completely received and is waiting in SBUF to be read.

Serial port interrupts are slightly different from timer interrupts. The flag that causes a serial port interrupt is not cleared by hardware when the CPU vectors to the interrupt. The reason is that there are two sources for a serial port interrupt, TI or RI. The source of the interrupt must be determined in the ISR and the interrupting flag cleared by software. Recall that with timer interrupts the interrupting flag is cleared by hardware when the processor vectors to the ISR.

EXAMPLE 6.3 Character Output Using Interrupts

Write a program using interrupts to continually transmit the ASCII code set (excluding control codes) to a terminal attached to the 8051's serial port.

Solution

There are 128 7-bit codes in the ASCII chart. (See Appendix F.) These consist of 95 graphic codes (20H to 7EH) and 33 control codes (00H to 1FH, and 7FH). The program shown below is self-contained and executable from EPROM or ROM immediately after a system reset.

```
0000            5           ORG     0
0000  020030    6           LJMP    MAIN
0023            7           ORG     0023H         ;serial port interrupt entry
0023  020042    8           LJMP    SPISR
0030            9           ORG     0030H
0030  758920   10   MAIN:   MOV     TMOD,#20H     ;Timer 1, mode 2
0033  758DE6   11           MOV     TH1,#-26      ;12000 baud reload value
0036  D28E     12           SETB    TR1           ;start timer
0038  759842   13           MOV     SCON,#42H     ;mode 1, set TI to force 1st
               14                                 ; interrupt; send 1st char.
003B  7420     15           MOV     A,#20H        ;send ASCII space first
003D  75A890   16           MOV     IE,90H        ;enable serial port interrupt
0040  80FE     17           SJMP    $             ;do nothing
               18   ;
0042  B47F02   19   SPISR:  CJNE    A,#7FH,SKIP   ;if finished ASCII set,
0045  7420     20           MOV     A,#20H        ; reset to SPACE
0047  F599     21   SKIP:   MOV     SBUF,A        ;send char. to serial port
0049  04       22           INC     A             ;increment ASCII code
004A  C299     23           CLR     TI            ;clear interrupt flag
004C  32       24           RETI
               25           END
```

Discussion

After jumping to MAIN at code address 0030H, the first three instructions initialize Timer 1 to provide a 1200 baud clock to the serial port (lines 10-12). MOV SCON,#42H initializes the serial port for mode 1 (8-bit UART) and sets the TI flag to force an interrupt as soon as interrupts are enabled. Then, the first ASCII graphic code (20H) is loaded into A and serial port interrupts are enabled. Finally, the main body of the program enters a do-nothing loop (SJMP $).

The serial port interrupt service routine does all the work once the main program sets up initial conditions. The first two instructions check the accumulator, and if the ASCII code has reached 7FH (i.e., the last code transmitted was 7EH), reset the accumulator to 20H (lines 19-20). Then, the ASCII code is sent to the serial port buffer (MOV SBUF,A), the code is incremented (INC A), the transmit interrupt flag is cleared (CLR TI), and the ISR is terminated (RETI). Control returns to the main program and SJMP $ executes until TI is set at the end of the next character transmission.

If we compare the CPU's speed to the rate of character transmission, we see that SJMP $ executes for a very large percentage of the time for this program. What is this percentage? At 1200 baud, each bit transmitted takes 1÷1200 = 0.833 ms. Eight data bits plus a start-and-stop bit, therefore, take 8.33 ms or 8333 µs. The worst-case execution time for the SPJSR is found by totaling the number of cycles for each instruction and multiplying by 1 µs (assuming 12 MHz operation). This turns out to be 8 µs. So, of the 8333 µs for each character transmission, only 8 µs are for the interrupt service routine. The SJMP $ instruction executes about 8325 ÷ 8333 x 100 = 99.90% of the time. Since interrupts are used, the SJMP $ instruction could be replaced with other instructions performing other tasks required in the application. Interrupts would still occur every 8.33 ms, and characters would still be transmitted out the serial port as they are in the above program.

6.7 EXTERNAL INTERRUPTS

External interrupts occur as a result of a low-level or negative edge on the $\overline{\text{INT0}}$ or $\overline{\text{INT1}}$ pin on the 8051 IC. These are the alternate functions for Port 3 bits P3.2 (pin 12) and P3.3 (pin 13), respectively.

The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by hardware when vectoring to the ISR only if the interrupt was transition-activated. If the interrupt was level-activated, then the external requesting source controls the level of the request flag, rather than the on-chip hardware.

The choice of low-level-activated interrupts versus negative-edge-activated interrupts is programmable through the IT0 and IT1 bits in TCON. For example, if IT1 = 0, external interrupt 1 is triggered by a detected low at the $\overline{\text{INT1}}$ pin. If IT1 = 1, external interrupt 1 is edge-triggered. In this mode, if successive samples of the $\overline{\text{INT1}}$ pin show a high in one cycle and a low in the next, the interrupt request flag IE1 in ICON is set. Flag bit IE1 then requests the interrupt.

Since the external interrupt pins are sampled once each machine cycle, an input should be held for at least 12 oscillator periods to ensure proper sampling. If the external interrupt is transition-activated, the external source must hold the request pin high for at least one cycle and then hold it low for at least one more cycle to ensure the transition is detected. IE0 and IE1 are automatically cleared when the CPU vectors to the interrupt.

If the external interrupt is level-activated, the external source must hold the request active until the requested interrupt is actually generated. Then it must deactivate the request before the interrupt service routine is completed, or another interrupt will be generated. Usually, an action taken in the ISR causes the requesting source to return the interrupting signal to the inactive state.

EXAMPLE 6.4 Furnace Controller

Using interrupts, design an 8051 furnace controller that keeps a building at 20°C ± 1°C.

Solution

The following interface is assumed for this example. The furnace ON/OFF solenoid is connected to P1.7 such that

```
P1.7 = 1 for solenoid engaged (furnace ON)
P1.7 = 0 for solenoid disengaged (furnace OFF)
```

Temperature sensors are connected to $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ and provide $\overline{\text{HOT}}$ and $\overline{\text{COLD}}$ signals, respectively, such that

$\overline{\text{HOT}}$ = 0 if T> 21°C
$\overline{\text{COLD}}$ = 0 if T< 19°C

The program should turn on the furnace for T < 19°C and turn it off for T > 21°C. The hardware configuration and a timing diagram are shown in Figure 6-5.

```
0000              5             ORG     0
0000    020030    6             LJMP    MAIN
                  7                                  ;EXT 0 vector at 0003H
0003    C297      8   EX0ISR:   CLR     P1.7         ;turn furnace off
0005    32        9             RETI
0013              10            ORG     0013H
0013    D297      11  EX1ISR:   SETB    P1.7         ;turn furnace on
0015    32        12            RETI
0030              13            ORG     30H
0030    75A885    14  MAIN:     MOV     IE,#85H      ;enable external interrupts
0033    D288      15            SETB    IT0          ;negative edge triggered
0035    D28A      16            SETB    IT1
0037    D297      17            SETB    P1.7         ;turn furnace on
0039    20B202    18            JB      P3.2,SKIP    ;if T > 21 degrees,
003C    C297      19            CLR     P1.7         ; turn furnace off
003E    80FE      20  SKIP:     SJMP    $            ;do nothing
                  21            END
```

Discussion

The first three instructions in the main program (lines 14-16) turn on external interrupts and make both $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ negative-edge triggered. Since the current state of the $\overline{\text{HOT}}$ (P3.3) and $\overline{\text{COLD}}$ (P3.3) inputs is not known, the next three instructions (lines 17-19) are required to turn the furnace ON or OFF, as appropriate. First, the furnace is turned ON

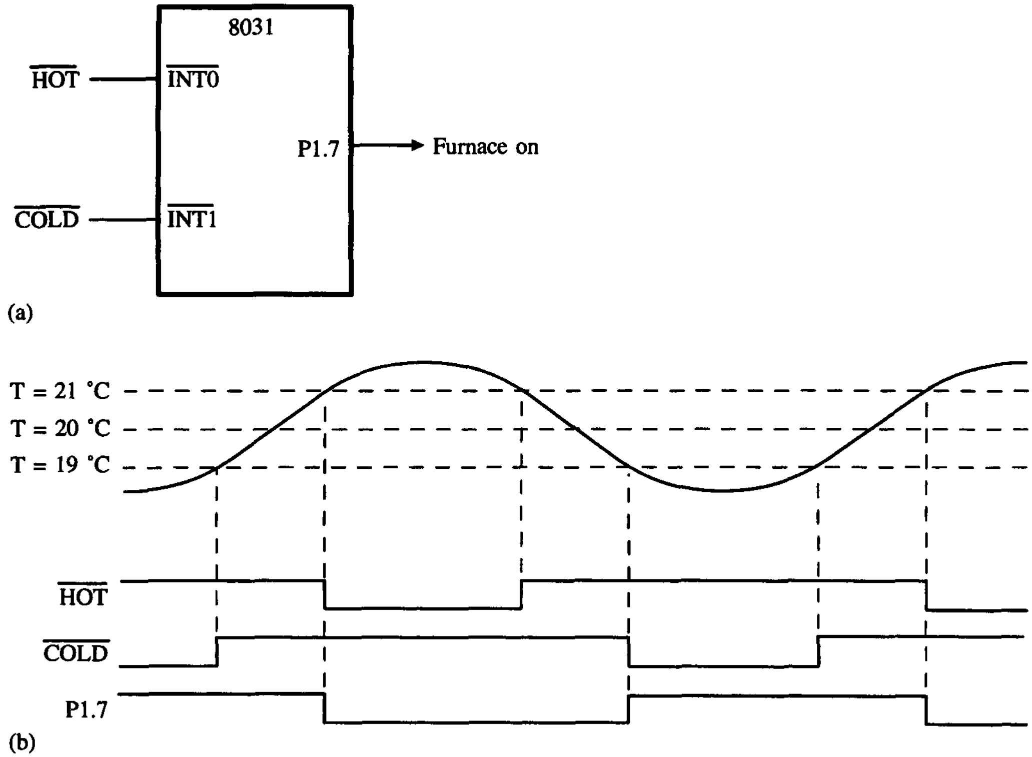


FIGURE 6-5
Furnace example. (a) Hardware connections (b) Timing

(SETB P1.7), and then the $\overline{\text{HOT}}$ input is sampled (JB P3.2,SKIP). If $\overline{\text{HOT}}$ is high, then T < 21°C, so the next instruction is skipped and the furnace is left ON. If, however, $\overline{\text{HOT}}$ is low, then T > 21°C. In this case the jump does not take place. The next instruction turns the furnace OFF (CLR P1.7) before entering the do-nothing loop.

Once everything is set up properly in the main program, little remains to be done. Each time the temperature rises above 21°C or falls below 19°C, an interrupt occurs. The ISRs simply turn the furnace ON (SETB P1.7) or OFF (CLR P1.7), as appropriate, and return to the main program.

Note that an ORG 0003H statement is not necessary immediately before the EX0ISR label. Since the LJMP MAIN instruction is three bytes long, EX0ISR is certain to start at 0003H, the correct entry point for external 0 interrupts.

EXAMPLE 6.5 Intrusion Warning System

Design an intrusion warning system using interrupts that sounds a 400 Hz tone for 1 second (using a loudspeaker connected to P1.7) whenever a door sensor connected to INT0 makes a high-to-low transition.

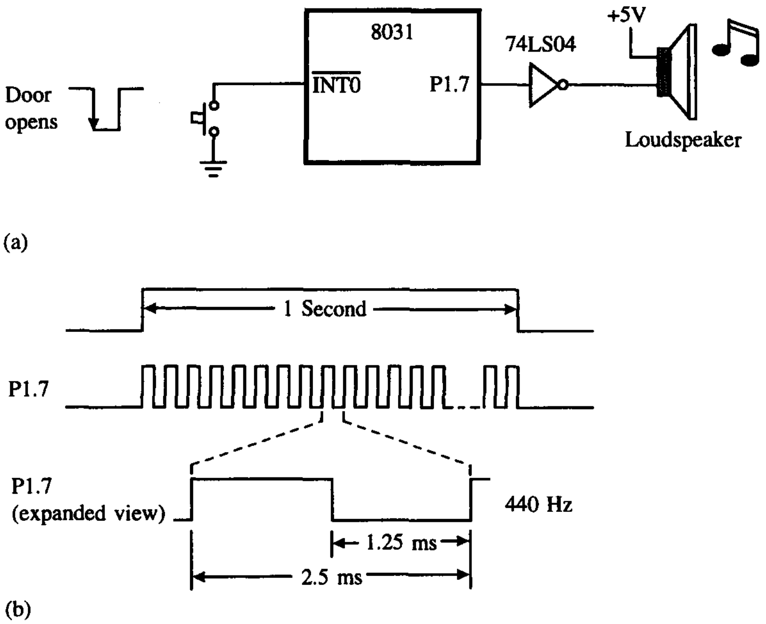


FIGURE 6-6
Loudspeaker interface using interrupts (a) Hardware connections (b) Timing

Solution

The solution to this example uses three interrupts: external 0 (door sensor), Timer 0 (1 second timeout), and Timer 1 (400 Hz tone). The hardware configuration and timings are shown in Figure 6-6.

```
0000                           ORG     0
0000    020030                 LJMP    MAIN        ;3-byte instruction
0003    02003A                 LJMP    EX0ISR      ;EXT 0 vector address
000B                           ORG     000BH       ;Timer 0 vector
000B    020045                 LJMP    T0ISR
001B                           ORG     001BH       ;Timer 1 vector
001B    020059                 LJMP    T1ISR
0030                           ORG     0030H
0030    D288         MAIN:     SETB    IT0         ;negative edge activated
0032    758911                 MOV     TMOD,#11H   ;16-bit timer mode
0035    75A881                 MOV     IE,#81H     ;enable EXT 0 only
0038    80FE                   SJMP    $           ;now relax
                     ;
003A    7F14         EX0ISR:   MOV     R7,#20      ;20 ' 5000 us = 1 second
003C    D28D                   SETB    TF0         ;force timer 0 interrupt
003E    D28F                   SETB    TF1         ;force timer 1 interrupt
0040    D2A9                   SETB    ET0         ;begin tone for 1 second
```

```
0042   D2AB                   SETB   ET1           ;enable timer interrupts
0044   32                     RETI                 ;timer ints will do the work
              ;
0045   C28C     T0ISR:        CLR    TR0           ;stop timer
0047   DF07                   DJNZ   R7,SKIP       ;if not 20th time, exit
0049   C2A9                   CLR    ET0           ;if 20th, disable tone
004B   C2AB                   CLR    ET1           ;disable itself
004D   020058                 LJMP   EXIT
0050   758C3C   SKIP:         MOV    TH0,#HIGH(-50000)   ;0.05 sec. delay
0053   758AB0                 MOV    TL0,#LOW(-5000)
0056   D28C                   SETB   TR0
0058   32       EXIT:         RETI
              ;
0059   C28E     T1ISR:        CLR    TR1
005B   758DFB                 MOV    TH1,#HIGH(-1250)    ;count for 400 Hz
005E   758B1E                 MOV    TL1,#LOW(-1250)
0061   B297                   CPL    P1.7          ;music maestro!
0063   D28E                   SETB   TR1
0065   32                     RETI
                              END
```

Discussion

This is our largest program thus far. Five distinct sections are the interrupt vector locations, the main program, and the three interrupt service routines. All vector locations contain LJMP instructions to the respective routines. The main program, starting at code address 0030H, contains only four instructions. SETB IT0 configures the door sensing interrupt input as negative-edge triggered. MOV TMOD,#11H configures both timers for mode 1, 16-bit timer mode. Only the external 0 interrupt is enabled initially (MOV IE,#81H), so a "door-open" condition is needed before any interrupt is accepted. Finally, SJMP $ puts the main program in a do-nothing loop.

When a door-open condition is sensed (by a high-to-low transition of $\overline{\text{INT0}}$), an external 0 interrupt is generated, EX0ISR begins by putting the constant 20 in R7 (see below), then sets the overflow flags for both timers to force timer interrupts to occur.

Timer interrupts will only occur, however, if the respective bits are enabled in the IE register. The next two instructions (SETB ET0 and SETB ET1) enable timer interrupts. Finally, EX0ISR terminates with a RETI to the main program.

Timer 0 creates the 1 second timeout, and Timer 1 creates the 400 Hz tone. After EX0ISR returns to the main program, timer interrupts are immediately generated (and accepted after one execution of SJMP $). Because of the fixed polling sequence (see Figure 6-2), the Timer 0 interrupt is serviced first. A 1 second timeout is created by programming 20 repetitions of a 50,000 μs timeout. R7 serves as the counter. Nineteen times out of 20, T0ISR operates as follows. First, Timer 0 is turned off and R7 is decremented. Then, TH0/TL is reloaded with -50,000, the timer is turned back on, and the interrupt is terminated. On the 20th Timer 0 interrupt, R7 is decremented to 0 (1 second has elapsed). Both timer interrupts are disabled (CLR ET0, CLR ET1) and the interrupt is terminated. No further timer interrupts will be generated until the next "door-open" condition is sensed.

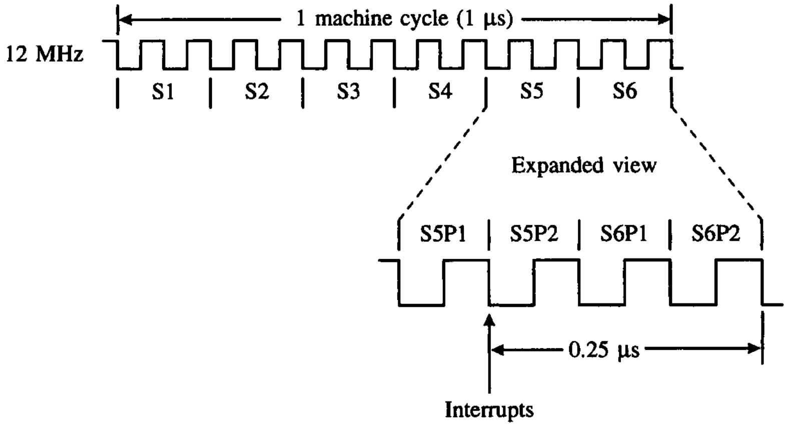


FIGURE 6-7
Sampling of interrupts on S5P2

The 400 Hz tone is programmed using Timer 1 interrupts, 400 Hz requires a period of 1/400 = 2,500 µs or 1,250 high-time and 1,25 µs low-time. Each timer 1 ISR simply puts — 1250 in TH1/TL1, complements the port bit driving the loudspeaker, then terminates.

6.8 INTERRUPT TIMINGS

Interrupts are sampled and latched on S5P2 of each machine cycle. (See Figure 6-7.) They are polled on the next machine cycle, and if an interrupt condition exists, it is accepted if (a) no other interrupt of equal or higher priority is in progress, (b) the polling cycle is the last cycle in an instruction, and (c) the current instruction is not a RETI or any access to IE or IP. During the next two cycles, the processor pushes the PC on the stack and loads the PC with the interrupt vector address. The ISR begins.

The stipulation that the current instruction is not RETI ensures that at least one instruction executes after each interrupt service routine. The timing is shown in Figure 6-8.

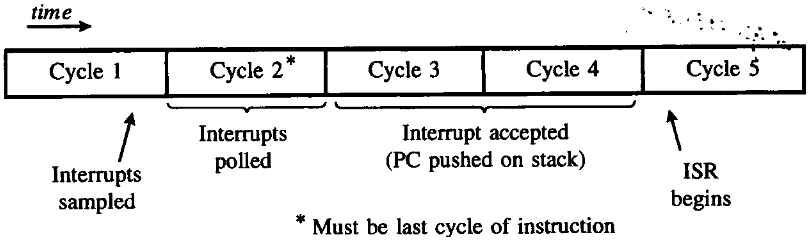


FIGURE 6-8
Polling of interrupts

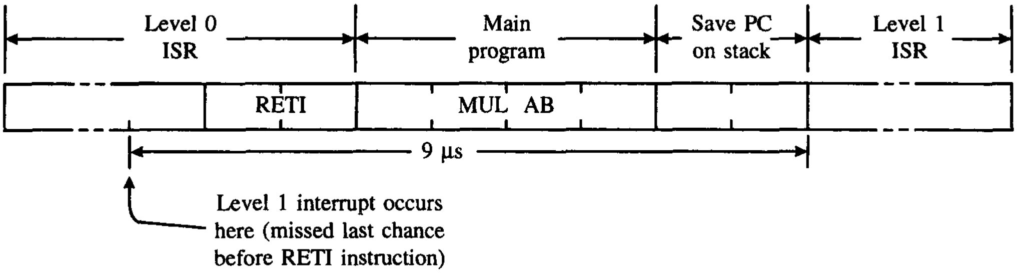

Level 0 ISR
Main program
Save PC on stack
Level 1 ISR
RETI
MUL AB
9 µs
Level 1 interrupt occurs here (missed last chance before RETI instruction)

FIGURE 6-9
Interrupt latency

The time between an interrupt condition occurring and the ISR beginning is called **interrupt latency.** Interrupt latency is critical in many control applications. With a 12 MHz crystal, the interrupt latency can be as short as 3.25 µs on the 8051. An 8051 system that uses one high-priority interrupt will have a worst-case interrupt latency of 9.25 µs (assuming the high-priority interrupt is always enabled). This occurs if the interrupt condition happens just before the RETI of a level 0 ISR that is followed by a multiply instruction (see Figure 6-9).

SUMMARY

This chapter has presented the major details required to embark on the design of interrupt-driven systems with the 805I microcontroller. Readers are advised to begin programming with interrupts in increments. The examples in this chapter serve as a good first contact with 8051 interrupts.

The 8051 single-board computers usually contain a monitor program in EPROM residing at the bottom of code memory. If interrupts are not used in the monitor program, the vector locations probably contain LJMP instructions to an area of CODE RAM where user applications are loaded for execution and debugging. The manufacturer's literature will provide the addresses for programmers to use as entry points for interrupt service routines. Alternatively, users can simply "look" in the interrupt vector locations, using the monitor program's commands for examining code memory locations. The content of code memory address 0003H, for example, will contain the opcode of the first instruction to execute for an external 0 interrupt. If this is an LJMP opcode (22H: see Appendix B), then the next two addresses (0004H and 0005H) contain the address of the ISR, and so on.

Alternately, users can develop self-contained interrupt applications, as shown in the examples. The object bytes can be burned into EPROM and installed in the target system at code address 000EH. When the system is powered up or reset, the application begins execution without the need of a monitor program for loading and starting the application.

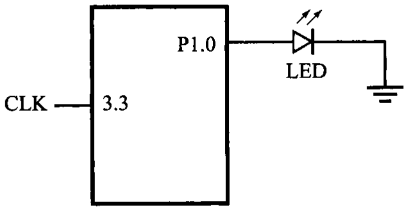


FIGURE 6-10
LED interface using interrupts

PROBLEMS

6.1 Modify Example 6.1 to shut off interrupts and terminate if any key is bit on the terminal.

6.2 Create a 1 kHz square wave on P1.7, using interrupts.

6.3 Create a 7 kHz pulse wave with a 30% duty cycle on P1.6, using interrupts.

6.4 Combine Example 6.1 and Example 6.3 (earlier in the chapter) into one program.

6.5 Modify Example 6.3 to send one character per second. (Hint: use a timer and output the character in the timer ISR.)

6.6 Modify Example 6.5 to include a "restart" mode. If a high-to-low transition occurs while the loudspeaker is sounding, restart the timing loop to continue sounding it for another second. This is illustrated in Figure 6-10.

6.7 Suppose that an external interrupt connected to $\overline{\text{INT0}}$ and a timer 0 interrupt occur at the same time, which one would be serviced first? Why?

6.8 By default, when a serial port interrupt and an external interrupt connected to $\overline{\text{INT1}}$ occur at the same time, the serial port interrupt would only be serviced by the 8051 after the external interrupt has been serviced. How do you set the 8051 to service the serial port interrupt first when both interrupts happen at the same time?

6.9 The IE and IP registers have been initialized to the following values:

```
IE = 10001111
IP = 00001110
```

Suppose that a timer 0 interrupt, a serial port interrupt, and an external $\overline{\text{INT1}}$ interrupt all occur at the same time. Which one would be serviced first? Why?

6.10 What is the difference between small ISRs and large ISRs?

6.11 Referring to Figure 6-10, suppose that pin 3.3 is connected to a clock signal, CLK of frequency 1kHz. Write an interrupt-enabled assembly language program that will send a HIGH to P1.E (turning on the LED) for about 250 s whenever a negativegoing-transition (NGT) of the CLK signal is detected at pin 3.3.

CHAPTER 7

Assembly Language Programming

7.1 INTRODUCTION

This chapter introduces assembly language programming for the 8051 microcontroller. Assembly language is a computer language lying between the extremes of machine language and high-level language. Typical high-level languages like Pascal or C use words and statements that are easily understood by humans, although still a long way from "natural" language. Machine language is the binary language of computers. A machine language program is a series of binary bytes representing instructions the computer can execute.

Assembly language replaces the binary codes of machine language with easy to remember "mnemonics" that facilitate programming. For example, an addition instruction in machine language might be represented by the code "10110011." It might be represented in assembly language by the mnemonic "ADD." Programming with mnemonics is obviously preferable to programming with binary codes.

Of course, this is not the whole story. Instructions operate on data, and the location of the data is specified by various "addressing modes" embedded in the binary code of the machine language instruction. So, there may be several variations of the ADD instruction, depending on what is added. The rules for specifying these variations are central to the theme of assembly language programming.

An assembly language program is not executable by a computer. Once written, the program must undergo translation to machine language. In the example above, the mnemonic "ADD" must be translated to the binary code "10110011." Depending on the complexity of the programming environment, this translation may involve one or more steps before an executable machine language program results. As a minimum, a program called an "assembler" is required to translate the instruction mnemonics to machine language binary codes. A further step may require a "linker" to combine portions of programs from separate files and to set the address in memory at which the program may execute. We begin with a few definitions.

An **assembly language program** is a program written using labels, mnemonics, and so on, in which each statement corresponds to a machine instruction. Assembly language programs, often called source code or symbolic code, cannot be executed by a computer.

A **machine language program** is a program containing binary codes that represent instructions to a computer. Machine language programs, often called object code, are executable by a computer.

An **assembler** is a program that translates an assembly language program into a machine language program. The machine language program (object code) may be in "absolute" form or in "relocatable" form. In the latter case, "linking" is required to set the absolute address for execution.

A **linker** is a program that combines relocatable object programs (modules) and produces an absolute object program that is executable by a computer. A linker is sometimes called a "linker/locator" to reflect its separate functions of combining relocatable modules (linking) and setting the address for execution (locating).

A **segment** is a unit of code or data memory. A segment may be relocatable or absolute. A relocatable segment has a name, type, and other attributes that allow the linker to combine it with other partial segments, if required, and to correctly locate the segment. An absolute segment has no name and cannot be combined with other segments.

A **module** contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. A module may be thought of as a "file" in many instances.

A **program** consists of a single absolute module, merging all absolute and relocatable segments from all input modules. A program contains only the binary codes for instructions (with addresses and data constants) that are understood by a computer.

7.2 ASSEMBLER OPERATION

There are many assembler programs and other support programs available to facilitate the development of applications for the 8051 microcontroller. Intel's original MCS-51^{TM} family assembler, ASM51^{TM}, is no longer available commercially. However, it set the standard to which the others are compared. In this chapter, we focus on assembly language programming as undertaken using the most common features of ASM51. Although many features are standardized, some may not be implemented in assemblers from other companies.

ASM51 is a powerful assembler with all the bells and whistles. It is available on Intel development systems and on the IBM *PC* family of microcomputers. Since these "host" computers contain a CPU chip other than the 8051, ASM51 is called a **cross assembler.** An 8051 source program may be written on the host computer (using any text editor) and may be assembled to an object file and listing file (using ASM51), but the program may not be executed. Since the host system's CPU chip is not an 8051, it does not understand the binary instructions in the object file. Execution on the host computer requires either hardware e-mulation or software simulation of the target CPU. A third possibility is to download the object program to an 8051-based target system for execution. Hardware emulation, software simulation, downloading, and other development techniques are discussed in Chapter 10.

ASM51 is invoked from the system prompt by

```
ASM51 source file [assembler controls]
```

The source file is assembled and any assembler controls specified take effect. (Assembler controls, which are optional, are discussed later in this chapter.) The assembler receives a source file as input (e.g., PROGRAM.SRC) and generates an object file (PROGRAM.OBJ) and listing file (PROGRAM .LST) as output. This is illustrated in Figure 7-1.

Since most assemblers scan the source program twice in performing the translation to machine language, they are described as **two-pass assemblers.** The assembler uses a **location counter** as the address of instructions and the values for labels. The action of each pass is described below.

7.2.1 Pass One

During the first pass, the source file is scanned line-by-line and a **symbol table** is built. The location counter defaults to 0 or is set by the ORG (set origin) directive. As the file is scanned, the location counter is incremented by the length of each instruction. Define data directives (DBs or DWs) increment the location counter by the number of bytes defined. Reserve memory directives (DSs) increment the location counter by the number of bytes reserved.

Each time a label is found at the beginning of a line, it is placed in the symbol table along with the current value of the location counter. Symbols that are defined using equate directives (EQUs) are placed in the symbol table along with the "equated" value. The symbol table is saved and then used during pass two.

7.2.2 Pass Two

During pass two, the object and listing files are created. Mnemonics are converted to opcodes and placed in the output files. Operands are evaluated and placed after the instruction opcodes. Where symbols appear in the operand field, their values are retrieved from the

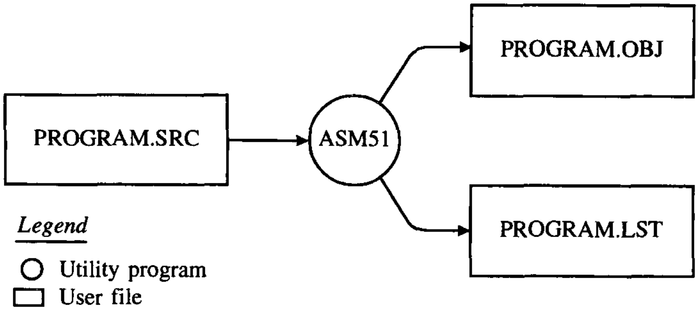


FIGURE 7-1
Assembling a source program

```
ASM(input_file)  /* assemble source program in input_file */
BEGIN
  /***** PASS 1: BUILD THE SYMBOL TABLE *****/
  lc = 0;      /* lc = location counter */
  mnemonic = null;
  open_input_file;
  WHILE (mnemonic != end) DO BEGIN
    get_line();
    scan_line();  /* get label/symbol and mnemonic */
    IF (label) THEN
      enter_in_symbol_table(label, lc);
    CASE mnemonic OF BEGIN
      null, comment, end: ; /* do nothing */
      ORG: lc = operand_value;
      EQU: enter_in_symbol_table(symbol, operand_value);
      DB: WHILE (got_operand) DO increment_lc;
      DS: lc = lc + operand_value;
      1_byte_instruction: lc = lc + 1;
      2_byte_instruction: lc = lc + 2;
      3_byte_instruction: lc = lc + 3;
    END
  END
  /***** PASS 2: CREATE THE OBJECT PROGRAM *****/
  rewind_input_file_pointer;
  lc = 0;
  mnemonic = null;
  open_output_file;
  WHILE (mnemonic != end) DO BEGIN
    get_line();
    scan_line();  /* determine mnemonic op code & value(s) of operands */
    /* Note: Symbols used in operand field are looked-up in the symbol */
    /* table created during pass one.                                  */
    CASE mnemonic OF BEGIN
      null, comment, EQU, END: ; /* do nothing */
      ORG: lc = operand_value;
      DB: WHILE (operand) BEGIN
          put_in_object_file(operand);
          lc = lc + 1;
        END
      DS: lc = lc + operand;
      1_byte_instruction:    put_in_object_file(inst_code);
      2_byte_instruction:    put_in_object_file(inst_code);
                             put_in_object_file(operand);
      1_byte_instruction:    put_in_object_file(inst_code);
                             put_in_object_file(operand high-byte);
                             put_in_object_file(operand low-byte);
      lc = lc + size_of_instruction;
    END
  END
  close_input_file;
  close_output_file;
END
```

FIGURE 7-2
Pseudo code sketch of a two-pass operator

symbol table (created during pass one) and used in calculating the correct data or addresses for the instructions.

Since two passes are performed, the source program may use "forward references," that is, use a symbol before it is defined. This would occur, for example, in branching ahead in a program.

The object file, if it is absolute, contains only the binary bytes (00H-0FFH) of the machine language program. A relocatable object file will also contain a symbol table and other information required for linking and locating. The listing file contains ASCII text codes (20H-7EH) for both the source program and the hexadecimal bytes in the machine language program.

A good demonstration of the distinction between an object file and a listing file is to display each on the host computer's CRT display (using, for example, the TYPE command on MS-DOS systems). The listing file clearly displays, with each line of output containing an address, opcode, and perhaps data, followed by the program statement from the source file. The listing file displays properly because it contains only ASCII text codes. Displaying the object file is a problem, however. The output will appear as "garbage," since the object file contains binary codes of an 8051 machine language program, rather than ASCII text codes.

In Figure 7-2, a sketch of a two-pass assembler is shown written in a pseudo computer language (similar to Pascal or C) to enhance readability.

7.3 ASSEMBLY LANGUAGE PROGRAM FORMAT

Assembly language programs contain the following:

- Machine instructions
- Assembler directives
- Assembler controls
- Comments

Machine instructions are the familiar mnemonics of executable instructions (e.g., ANL). Assembler directives are instructions to the assembler program that define program structure, symbols, data, constants, and so on (e.g., ORG). Assembler controls set assembler modes and direct assembly flow (e.g., $TITLE). Comments enhance the readability of programs by explaining the purpose and operation of instruction sequences.

Those lines containing machine instructions or assembler directives must be written following specific rules understood by the assembler. Each line is divided into "fields" separated by space or tab characters. The general format for each line is as follows:

```
[label:] mnemonic [operand] [,operand] [.. .] [;comment]
```

Only the mnemonic field is mandatory. Many assemblers require the label field, if present, to begin on the left in column 1, and subsequent fields to be separated by space or tab characters. With ASM51, the label field needn't begin in column 1 and the mnemonic field needn't be on the same line as the label field. The operand field must, however, begin on the same line as the mnemonic field. The fields are described below.

7.3.1 Label Field

A label represents the address of the instruction (or data) that follows. When branching to this instruction, this label is used in the operand field of the branch or jump instruction (e.g., SJMP SKIP).

Whereas the term "label" always represents an address, the term "symbol" is more general. Labels are one type of symbol and are identified by the requirement that they must terminate with a colon (:). Symbols are assigned values or attributes, using directives such as EQU, SEGMENT, BIT, DATA, etc. Symbols may be addresses, data constants, names of segments, or other constructs conceived by the programmer. Symbols do not terminate with a colon. In the example below, PAR is a symbol and START is a label (which is a type of symbol).

```
PAR      EQU 500            ;"PAR" IS A SYMBOL WHICH
                            ;REPRESENTS THE VALUE 500
START:   MOV A,#0FFH        ;"START" IS A LABEL WHICH
                            ;REPRESENTS THE ADDRESS OF
                            ;THE MOV INSTRUCTION
```

A symbol (or label) must begin with a letter, question mark, or underscore (_); must be followed by letters, digit, "?", or "_"; and can contain up to 31 characters.[1] Symbols may use upper- or lowercase characters, but they are treated the same. Reserved words (mnemonics, operators, predefined symbols, and directives) may not be used.

7.3.2 Mnemonic Field

Instruction mnemonics or assembler directives go into mnemonic field, which follows the label field. Examples of instruction mnemonics are ADD, MOV, DIV, or INC. Examples of assembler directives are ORG, EQU, or DB. Assembler directives are described later in this chapter.

7.3.3 Operand Field

The operand field follows the mnemonic field. This field contains the address or data used by the instruction. A label may be used to represent the address of the data, or a symbol may be used to represent a data constant. The possibilities for the operand field are largely dependent on the operation. Some operations have no operand (e.g., the RET instruction), while others allow for multiple operands separated by commas. Indeed, the possibilities for the operand field are numerous, and we shall elaborate on these at length. But first, the comment field.

7.3.4 Comment Field

Remarks to clarify the program go into comment field at the end of each line. Comments must begin with a semicolon (;). Entire lines may be comment lines by beginning them with

[1] The reader is reminded that the rules specified in this chapter apply to Intel's ASM51. Other assemblers may have different requirements.

a semicolon. Subroutines and large sections of a program generally begin with a comment block—several lines of comments that explain the general properties of the section of software that follows.

7.3.5 Special Assembler Symbols

Special assembler symbols are used for the register-specific addressing modes. These include A, R0 through R7, DPTR, PC, C, and AB. In addition, a dollar sign ($) can be used to refer to the current value of the location counter. Some examples follow.

```
          SETB C
          INC  DPTR
          JNB  TI,$
```

The last instruction above makes effective use of ASM51's location counter to avoid using a label. It could also be written as

```
HERE:      JNB TI,HERE
```

7.3.6 Indirect Address

For certain instructions, the operand field may specify a register that contains the address of the data. The commercial "at" sign (@) indicates address indirection and may only be used with R0, R1, the DPTR, or the PC, depending on the instruction. For example,

```
    ADD  A,@R0
    MOVC A,@A+PC
```

The first instruction above retrieves a byte of data from internal RAM at the address specified in R0. The second instruction retrieves a byte of data from external code memory at the address formed by adding the contents of the accumulator to the program counter. Note that the value of the program counter, when the add takes place, is the address of the instruction following MOVC. For both instructions above, the value retrieved is placed into the accumulator.

7.3.7 Immediate Data

Instructions using immediate addressing provide data in the operand field that become part of the instruction. Immediate data are preceded with a pound sign (#). For example,

```
CONSTANT EQU 100
         MOV A,#0FEH
         ORL 40H,#CONSTANT
```

All immediate data operations (except MOV DPTR,#data) require eight bits of data. The immediate data are evaluated as a 16-bit constant, and then the low-byte is used. All bits in the high-byte must be the same (00H or 0FFH) or the error message "value will not fit in a byte" is generated. For example, the following instructions are syntactically correct:

```
MOV A,#0FF00H
MOV A,#00FFH
```

But the following two instructions generate error messages:

```
MOV A,#0FE00H
MOV A,#01FFH
```

If signed decimal notation is used, constants from -256 to +255 may also be used. For example, the following two instructions are equivalent (and syntactically correct):

```
MOV A,#-256
MOV A,#0FF00H
```

Both instructions above put 00H into accumulator A.

7.3.8 Data Address

Many instructions access memory locations using direct addressing and require an on-chip data memory address (00H to 7FH) or an SFR address (80H to 0FFH) in the operand field. Predefined symbols may be used for the SFR addresses. For example,

```
MOV A,45H
MOV A,SBUF              ;SAME AS MOV A,99H
```

7.3.9 Bit Address

One of the most powerful features of the 8051 is the ability to access individual bits without the need for masking operations on bytes. Instructions accessing bit-addressable locations must provide a bit address in internal data memory (00H to 7FH) or a bit address in the SFRs (80H to 0FFH).

There are three ways to specify a bit address in an instruction: (a) explicitly by giving the address, (b) using the **dot operator** between the byte address and the bit position, and (c) using a predefined assembler symbol. Some examples follow.

```
SETB 0E7H               ;EXPLICIT BIT ADDRESS
SETB ACC.7              ;DOT OPERATOR (SAME AS ABOVE)
JNB  TI,$               ;"TI" IS A PRE-DEFINED SYMBOL
JNB  99H,$              ;(SAME AS ABOVE)
```

7.3.10 Code Address

A code address is used in the operand field for jump instructions, including relative jumps (SJMP and conditional jumps), absolute jumps and calls (ACALL, AJMP), and long jumps and calls (LJMP, LCALL).

The code address is usually given in the form of a label. For example,

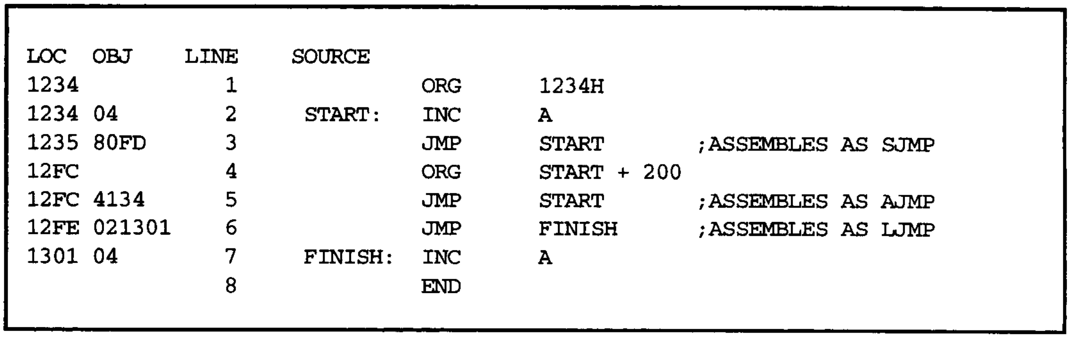
```
HERE:
      SJMP HERE
```

ASM51 will determine the correct code address and insert into the instruction the correct 8-bit signed offset, 11-bit page address, or 16-bit long address, as appropriate.

7.3.11 Generic Jumps and Calls

ASM51 allows programmers to use a generic JMP or CALL mnemonic. "JMP" can be used instead of SJMP, AJMP or LJMP; and "CALL" can be used instead of ACALL or LCALL. The assembler converts the generic mnemonic to a "real" instruction following a few simple rules. The generic mnemonic converts to the short form (for JMP only) if no forward references are used and the jump destination is within -128 locations, or to the absolute form if no forward references are used and the instruction following the JMP or CALL instruction is in the same 2 K block as the destination instruction. If short or absolute forms cannot be used, the conversion is to the long form.

The conversion is not necessarily the best programming choice. For example, if branching ahead a few instructions, the generic JMP will always convert to LJMP even though an SJMP is probably better. Consider the assembled instruction sequence in Figure 7-3 using three generic jumps. The first jump (line 3) assembles as SJMP because the destination is before the jump (i.e., no forward reference) and the offset is less than -128. The ORG directive in line 4 creates a gap of 200 locations between the label START and the second jump, so the conversion on line 5 is to AJMP because the offset is too great for SJMP. Note also that the address following the second jump (12FEH) and the address of START (1234H) are within the same 2K page, which, for this instruction sequence, is bounded by 1000H and 17FFH. This criterion must be met for absolute addressing. The third jump assembles as LJMP because the destination (FINISH) is not yet defined when



```
LOC  OBJ    LINE    SOURCE
1234           1              ORG     1234H
1234 04        2     START:   INC     A
1235 80FD      3              JMP     START        ;ASSEMBLES AS SJMP
12FC           4              ORG     START + 200
12FC 4134      5              JMP     START        ;ASSEMBLES AS AJMP
12FE 021301    6              JMP     FINISH       ;ASSEMBLES AS LJMP
1301 04        7     FINISH:  INC     A
               8              END
```

FIGURE 7-3
Use of the generic JMP mnemonic

the jump is assembled (i.e., a forward reference is used). The reader can verify that the conversion is as stated by examining the object field for each jump instruction. Verify the hexadecimal codes with those found in Appendix C for SJMP, AJMP, and LJMP.

7.4 ASSEMBLE-TIME EXPRESSION EVALUATION

Values and constants in the operand field may be expressed three ways: (a) explicitly (e.g., 0EFH), (b) with a predefined symbol (e.g., ACC), or (c) with an expression (e.g., 2 + 3). The use of expressions provides a powerful technique for making assembly language programs more readable and more flexible. When an expression is used, the assembler calculates a value and inserts it into the instruction.

All expression calculations are performed using 16-bit arithmetic; however, either 8 or 16 bits are inserted into the instruction as needed. For example, the following two instructions are the same:

```
MOV DPTR,#04FFH + 3
MOV DPTR,#0502H            ;ENTIRE 16-BIT RESULT USED
```

If the same expression is used in a "MOV A,#data" instruction, however, the error message "value will not fit in a byte" is generated by ASM51. An overview of the rules for evaluating expressions follows.

7.4.1 Number Bases

The base for numeric constants is indicated in the usual way for Intel microprocessors. Constants must be followed with "B" for binary, "0" or "Q" for octal, "D" or nothing for decimal, or "H" for hexadecimal. For example, the following instructions are the same:

```
MOV A,#15
MOV A,#1111B
MOV A,#0FH
MOV A,#17Q
MOV A,#15D
```

Note that a digit must be the first character for hexadecimal constants in order to differentiate them from labels (i.e., "0A5H" not "A5H").

7.4.2 Character Strings

Strings using one or two characters may be used as operands in expressions. The ASCII codes are converted to the binary equivalent by the assembler. Character constants are enclosed in single quotes 0. Some examples follow.

```
CJNE A,#'Q',AGAIN
SUBB A,#'0'             ;CONVERT ASCII DIGIT TO
                        ;BINARY DIGIT
MOV DPTR,#'AB'
MOV DPTR,#4142H         ;SAME AS ABOVE
```

7.4.3 Arithmetic Operators

The arithmetic operators are

```
+     addition
-     subtraction
*     multiplication
/     division
MOD   modulo (remainder after division)
```

For example, the following two instructions are the same:

```
MOV A,10+10H
MOV A,#1AH
```

The following two instructions are also the same:

```
MOV A,#25 MOD 7
MOV A,#4
```

Since the MOD operator could be confused with a symbol, it must be separated from its operands by at least one space or tab character, or the operands must be enclosed in parentheses. The same applies for the other operators composed of letters.

7.4.4 Logical Operators

The logical operators are

```
OR  logical OR
AND logical AND
XOR logical Exclusive OR
NOT logical NOT (complement)
```

The operation is applied on the corresponding bits in each operand. The operator must be separated from the operands by space or tab characters. For example, the following two instructions are the same:

```
MOV A,#'9' AND 0FH
MOV A,#9
```

The NOT operator only takes one operand. The following three MOV instructions are the same:

```
THREE       EQU 3
MINUS_THREE EQU -3
            MOV A,#(NOT THREE)+1
            MOV A,#MINUS_THREE
            MOV A,#11111101B
```

7.4.5 Special Operators

The special operators are

```
SHR  shift right
SHL  shift left
HIGH high-byte
LOW  low-byte
()   evaluate first
```

For example, the following two instructions are the same:

```
MOV A,#8 SHL 1
MOV A,#12H
```

The following two instructions are also the same:

```
MOV A,#HIGH 1234H
MOV A,#12H
```

7.4.6 Relational Operators

When a relational operator is used between two operands, the result is always false (0000H) or true (0FFFFH). The operators are

```
EQ  =   equals
NE  <>  not equals
LT  <   less than
LE  <=  less than or equal to
GT  >   greater than
GE  >=  greater than or equal to
```

Note that for each operator, two forms are acceptable (e.g., "EQ" or "="). In the following examples, all relational tests are "true":

```
MOV A,#5 = 5
MOV A,#5 NE 4
MOV A,#'X' LT 'Z'
MOV A,#'X' >= 'X'
MOV A,#$ > 0
MOV A,#100 GE 50
```

So, the assembled instructions are all equal to

```
MOV A,#0FFH
```

Even though expressions evaluate to 16-bit results (i.e., 0FFFFH), in the examples above only the low-order eight bits are used, since the instruction is a move byte operation. The result is not considered too big in this case, because as signed numbers the 16-bit value 0FFFFH and the 8-bit value 0FFH are the same (-1).

7.4.7 Expression Examples

The following are examples of expressions and the values that result:

Expression	Result
`B' - `A'	0001H
8/3	0002H
155 MOD 2	0001H
4 * 4	0010H
8 AND 7	0000H
NOT 1	FFFEH
`A' SHL 8	4100H
LOW 65535	00FFH
(8 + 1) * 2	0012H
5 EQ 4	0000H
`A' LT `B'	FFFFH
3 <= 3	FFFFH

A practical example that illustrates a common operation for timer initialization follows: Put -500 into Timer 1 registers TH1 and TL1. In using the HIGH and LOW operators, a good approach is

```
VALUE    EQU -500
         MOV TH1,#HIGH VALUE
         MOV TL1,#LOW VALUE
```

The assembler converts -500 to the corresponding 16-bit value (0FE0CH); then the HIGH and LOW operators extract the high (0FEH) and low (0CH) bytes, as appropriate for each MOV instruction.

7.4.8 Operator Precedence

The precedence of expression operators from highest to lowest is

```
( )
HIGH LOW
* / MOD SHL SHR
+ -
EQ NE LT LE GT GE = <> <=  >=
NOT
AND
OR XOR
```

When operators of the same precedence are used, they are evaluated left to right. Examples:

Expression	Value
HIGH(`A' SHL 8)	0041H
HIGH `A' SHL 8	0000H
NOT `A'-1	FFBFH
`A' OR `A'SHL8	4141H

7.5 ASSEMBLER DIRECTIVES

Assembler directives are instructions to the assembler program. They are not assembly language instructions executable by the target microprocessor. However, they are placed in the mnemonic field of the program. With the exception of DB and DW, they have no direct effect on the contents of memory.

ASM51 provides several categories of directives:

- Assembler state control (ORG, END, USING)
- Symbol definition (SEGMENT, EQU, SET, DATA, IDATA, XDATA, BIT, CODE)
- Storage initialization/reservation (DS, DBIT, DB, DW)
- Program linkage (PUBLIC, EXTRN, NAME)
- Segment selection (RSEG, CSEG, DSEG, ISEG, ESEG, XSEG)

Each assembler directive is presented below, ordered by category.

7.5.1 Assembler State Control

7.5.1.1 ORG (Set Origin) The format for the ORG (set origin) directive is

```
ORG expression
```

The ORG directive alters the location counter to set a new program origin for statements that follow. A label is not permitted. Two examples follow.

```
ORG 100H                    ;SET LOCATION COUNTER TO 100H
ORG ($ + 1000H) AND 0F000H  ;SET TO NEXT 4K BOUNDARY
```

The ORG directive can be used in any segment type. If the current segment is absolute, the value will be an absolute address in the current segment. If a relocatable segment is active, the value of the ORG expression is treated as an offset from the base address of the current instance of the segment.

7.5.1.2 End The format for the END directive is

```
END
```

END should be the last statement in the source file. No label is permitted and nothing beyond the END statement is processed by the assembler.

7.5.1.3 Using The format for the USING directive is

```
USING     expression
```

This directive informs ASM51 of the currently active register bank. Subsequent uses of the predefined symbolic register addresses AR0 to AR7 will convert to the appropriate direct address for the active register bank. Consider the following sequence:

```
USING     3
PUSH      AR7
```

```
USING     1
PUSH      AR7
```

The first push above assembles to PUSH 1FH (R7 in bank 3), whereas the second push assembles to PUSH 0FH (R7 in bank 1).

Note that USING does not actually switch register banks; it only informs ASM51 of the active bank. Executing 8051 instructions is the only way to switch register banks. This is illustrated by modifying the example above as follows:

```
MOV     PSW,#00001000B    ;SELECT REGISTER BANK 3
USING 3
PUSH    AR7               ;ASSEMBLE TO PUSH 1FH
MOV     PSW,#00001000B    ;SELECT REGISTER BANK 1
USING 1
PUSH    AR7               ;ASSEMBLE TO PUSH 0FH
```

7.5.2 Symbol Definition

The symbol definition directives create symbols that represent segments, registers, numbers, and addresses. None of these directives may be preceded by a label. Symbols defined by these directives may not have been previously defined and may not be redefined by any means. The SET directive is the only exception. Symbol definition directives are described below.

7.5.2.1 Segment The format for the SEGMENT directive is shown below.

```
symbol     SEGMENT segment_type
```

The symbol is the name of a relocatable segment. In the use of segments, ASM51 is more complex than conventional assemblers, which generally support only "code" and "data" segment types. However, ASM51 defines additional segment types to accommodate the diverse memory spaces in the 8051. The following are the defined 8051 segment types (memory spaces):

- CODE (the code segment)
- XDATA (the external data space)
- DATA (the internal data space accessible by direct addressing, 00H-7FH)
- IDATA (the entire internal data space accessible by indirect addressing, 00H-7FH, 00H-0FFH on the 8052)
- BIT (the bit space; overlapping byte locations 20H-2FH of the internal data space)

For example, the statement

```
EPROM     SEGMENT     CODE
```

declares the symbol EPROM to be a SEGMENT of type CODE. Note that this statement simply declares what EPROM is. To actually begin using this segment, the RSEG directive is used (see below).

7.5.2.2 EQU (Equate) The format for the EQU directive is

```
Symbol     EQU     expression
```

The EQU directive assigns a numeric value to a specified symbol name. The symbol must be a valid symbol name, and the expression must conform to the rules described earlier. The following are examples of the EQU directive:

```
N27         EQU 27           ;SET N27 TO THE VALUE 27
HERE        EQU $            ;SET "HERE" TO THE VALUE
                             ;OF THE LOCATION COUNTER
CR          EQU ODH          ;SET CR (CARRIAGE RETURN) TO ODH
MESSAGE:    DB `This is a message'
LENGTH      EQU $ - MESSAGE  ;"LENGTH" EQUALS LENGTH OF
                               "MESSAGE"
```

7.5.2.3 Other Symbol Definition Directives The SET directive is similar to the EQU directive except the symbol may be redefined later, using another SET directive.

The DATA, IDATA, XDATA, BIT, and CODE directives assign addresses of the corresponding segment type to a symbol. These directives are not essential. A similar effect can be achieved using the EQU directive; if used, however, they evoke powerful type-checking by ASM51. Consider the following two directives and four instructions:

```
FLAG1 EQU 05H
FLAG2 BIT 05H
          SETB FLAG1
          SETB FLAG2
          MOV  FLAG1,#0
          MOV  FLAG2,#0
```

The use of FLAG2 in the last instruction in this sequence will generate a "data segment address expected" error message from ASM51. Since FLAG2 is defined as a bit address (using the BIT directive), it can be used in a set bit instruction, but it cannot be used in a move byte instruction. Hence, the error. Even though FLAG1 represents the same value (05H), it was defined using EQU and does not have an associated address space. This is not an advantage of EQU, but, rather, a disadvantage. By properly defining address symbols for use in a specific memory space (using the directives BIT, DATA, XDATA, etc.), the programmer takes advantage of ASM51's powerful type-checking and avoids bugs from the misuse of symbols.

7.5.3 Storage Initialization/Reservation

The storage initialization and reservation directives initialize and reserve space in either word, byte, or bit units. The space reserved starts at the location indicated by the current value of the location counter in the currently active segment. These directives may be preceded by a label. The storage initialization/reservation directives are described below.

7.5.3.1 DS (Define Storage) The format for the DS (define storage) directive is

```
[label:] DS expression
```

The DS directive reserves space in byte units. It can be used in any segment type except BIT. The expression must be a valid assemble-time expression with no forward references and no relocatable or external references. When a DS statement is encountered in a program, the location counter of the current segment is incremented by the value of the expression. The sum of the location counter and the specified expression should not exceed the limitations of the current address space.

The following statements create a 40-byte buffer in the internal data segment:

```
        DSEG AT 30H ;PUT IN DATA SEGMENT (ABSOLUTE, INTERNAL)
LENGTH EQU 40
BUFFER: DS LENGTH ;40 BYTES RESERVED
```

The label BUFFER represents the address of the first location of reserved memory. For this example, the buffer begins at address 30H because "AT 30H" is specified with DSEG. (See 7.5.5.2 Selecting Absolute Segments.) This buffer could be cleared using the following instruction sequence:

```
          MOV R7,#LENGTH
          MOV R0,#BUFFER
LOOP:     MOV @R0,#0
          DJNZ R7,LOOP
          (continue)
```

To create a 1000-byte buffer in external RAM starting at 4000H, the following directives could be used:

```
XSTART   EQU 4000H
XLENGTH  EQU 1000
         XSEG AT  XSTART
XBUFFER: DS XLENGTH
```

This buffer could be cleared with the following instruction sequence:

```
      MOV  DPTR,#XBUFFER
LOOP: CLR  A
      MOVX @DPTR
      INC  DPTR
      MOV  A,DPL
      CJNE A,#LOW (XBUFFER + XLENGTH + 1),LOOP
      MOV  A,DPH
      CJNE A,#HIGH(XBUFFER + XLENGTH + 1),LOOP
      (continue)
```

This is an excellent example of a powerful use of ASM51's operators and assemble-time expressions. Since an instruction does not exist to compare the data pointer with an immediate value, the operation must be fabricated from available instructions. Two compares are required, one each for the high- and low-bytes of the DPTR. Furthermore, the compare-and-jump-if-not-equal instruction works only with the accumulator or a register, so the data pointer bytes must be moved into the accumulator before the CJNE instruction. The loop terminates only when the data pointer has reached BUFFER + LENGTH + 1. (The " +1 " is needed because the data pointer is incremented after the last MOVX instruction.)

7.5.3.2 DEBIT The format for the DEBIT (define bit) directive is,

```
[label:]   DEBIT expression
```

The DEBIT directive reserves space in bit units. It can be used only in a BIT segment. The expression must be a valid assemble-time expression with no forward references. When the DEBIT statement is encountered in a program, the location counter of the current (BIT) segment is incremented by the value of the expression. Note that in a BIT segment, the basic unit of the location counter is bits rather than bytes. The following directives create three flags in an absolute bit segment:

```
           BSEG        ;BIT SEGMENT (ABSOLUTE)
KBFLAG:   DEBIT 1      ;KEYBOARD STATUS
PRFLAG:   DEBIT 1      ;PRINTER STATUS
DKFLAG:   DEBIT 1      ;DISK STATUS
```

Since an address is not specified with BSEG in the example above, the address of the flags defined by DEBIT could be determined (if one wishes to do so) by examining the symbol table in the .LST or .M51 files. (See Figure 7-1 and Figure 7-6.) If the definitions above were the first use of BSEG, then KBFLAG would be at bit address 00H (bit 0 of byte address 20H; see Figure 2-6.) If other bits were defined previously using BSEG, then the definitions above would follow the last bit defined. (See 7.5.5.2. Selecting Absolute Segments.)

7.5.3.3 DB (Define Byte) The format for the DB (define byte) directive is

```
[label:]   DB   expression expression][...]
```

The DB directive initializes code memory with byte values. Since it is used to actually place data constants in code memory, a CODE segment must be active. The expression list is a series of one or more byte values (each of which may be an expression) separated by commas.

The DB directive permits character strings (enclosed in single quotes) longer than two characters as long as they are not part of an expression. Each character in the string is converted to the corresponding ASCII code. If a label is used, it is assigned the address of the first byte. For example, the following statements

```
          CSEG AT 0100H
SQUARES: DB 0,1,4,9,16,25  ;SQUARES OF NUMBERS 0-5
MESSAGE: DB 'Login:',0     ;NULL-TERMINATED CHARACTER STRING
```

when assembled, result in the following hexadecimal memory assignments for external code memory:

Address	Contents
0100	00
0101	01

0102	04
0103	09
0104	10
0105	19
0106	4C
0107	6F
0108	67
0109	69
010A	6E
010B	3A
010C	00

7.5.3.4 DW (Define Word) The format for the DW (define word) directive is

```
[label:] DW expression [ ,expression][...]
```

The DW directive is the same as the DB directive except two memory locations (16 bits) are assigned for each data item. For example, the statements

```
CSEG AT 200H
DW    $,`A',1234H,2,`BC'
```

result in the following hexadecimal memory assignments:

Address	Contents
0200	02
0201	00
0202	00
0203	41
0204	12
0205	34
0206	00
0207	02
0208	42
0209	43

7.5.4 Program Linkage

Program linkage directives allow the separately assembled modules (files) to communicate by permitting intermodule references and the naming of modules. In the following discussion, a "module" can be considered a "file." (In fact, a module may encompass more than one file.)

7.5.4.1 Public The format for the PUBLIC (public symbol) directive is

```
PUBLIC symbol [,symbol] [...]
```

The PUBLIC directive allows the list of specified symbols to be known and used outside the currently assembled module. A symbol declared PUBLIC must be defined in the current module. Declaring it PUBLIC allows it to be referenced in another module. For example,

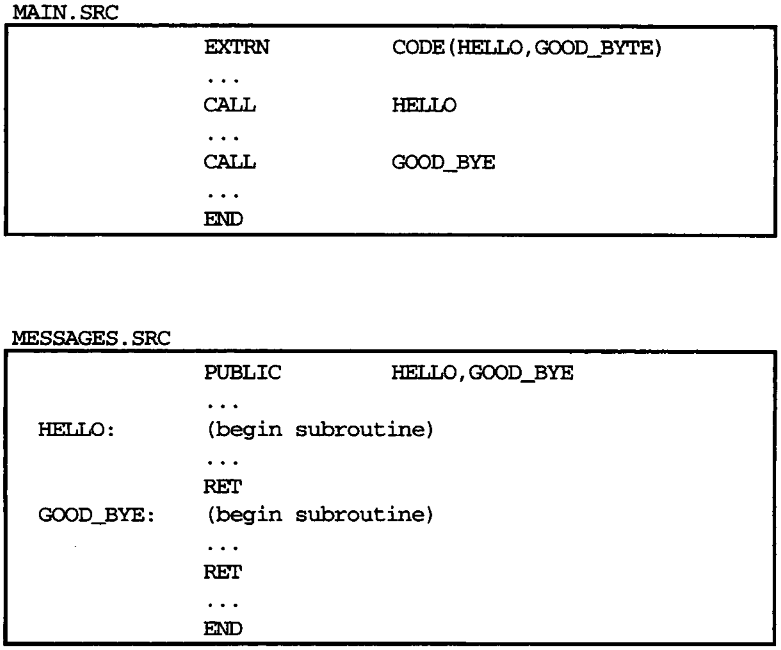
```
PUBLIC INCHAR, OUTCHR, INLINE, OUTSTR
```

7.5.4.2 Extrn The format for the EXTRN (external symbol) directive is

```
EXTRN segment_type(symbol [,symbol] [...], ...)
```

The EXTRN directive lists symbols to be referenced in the current module that are defined in other modules. The list of external symbols must have a segment type associated with each symbol in the list. (The segment types are CODE, XDATA, DATA, IDATA, BIT, and NUMBER. NUMBER is a type-less symbol defined by EQU.) The segment type indicates the way a symbol may be used. The information is important at link-time to ensure symbols are used properly in different modules.

The PUBLIC and EXTRN directives work together. Consider the two files shown in Figure 7-4, MAIN.SRC and MESSAGES.SRC. The subroutines HELLO and GOODBYE are defined in the module MESSAGES but are made available to other modules using the PUBLIC directive. The subroutines are called in the module MAIN even though they are not defined there. The EXTRN directive declares that these symbols are defined in another module.



MAIN.SRC

```
                EXTRN           CODE(HELLO,GOOD_BYTE)
                ...
                CALL            HELLO
                ...
                CALL            GOOD_BYE
                ...
                END
```

MESSAGES.SRC

```
                PUBLIC          HELLO,GOOD_BYE
                ...
   HELLO:       (begin subroutine)
                ...
                RET
   GOOD_BYE:    (begin subroutine)
                ...
                RET
                ...
                END
```

FIGURE 7-4

Use of the EXTRN and PUBLIC assembler directives

Neither MAIN.SRC nor MESSAGE.SRC is a complete program; they must be assembled separately and linked together to form an executable program. During linking, the external references are resolved with correct addresses inserted as the destination for the CALL instructions.

7.5.4.3 Name The format for the NAME directive is

```
NAME module_name
```

All the usual rules for symbol names apply to module names. If a name is not provided, the module takes on the file name (without a drive or subdirectory specifier and without an extension). In the absence of any use of the NAME directive, a program will contain one module for each file. The concept of "modules," therefore, is somewhat cumbersome, at least for relatively small programming problems. Even programs of moderate size (encompassing, for example, several files complete with relocatable segments) needn't use the NAME directive and needn't pay any special attention to the concept of "modules." For this reason, it was mentioned in the definition that a module may be considered a "file," to simplify learning ASM51. However, for very large programs (several thousand lines of code, or more), it makes sense to partition the problem into modules, where, for example, each module may encompass several files containing routines having a common purpose.

7.5.5 Segment Selection Directives

When the assembler encounters a segment selection directive, it diverts the following code or data into the selected segment until another segment is selected by a segment selection directive. The directive may select a previously defined relocatable segment or optionally create and select absolute segments.

7.5.5.1 RSEG (Relocatable Segment) The format for the RSEG (relocatable segment) directive is

```
RSEG segment_name
```

where "segment_name" is the name of a relocatable segment previously defined with the SEGMENT directive. RSEG is a "segment selection" directive that diverts subsequent code or data into the named segment until another segment selection directive is encountered.

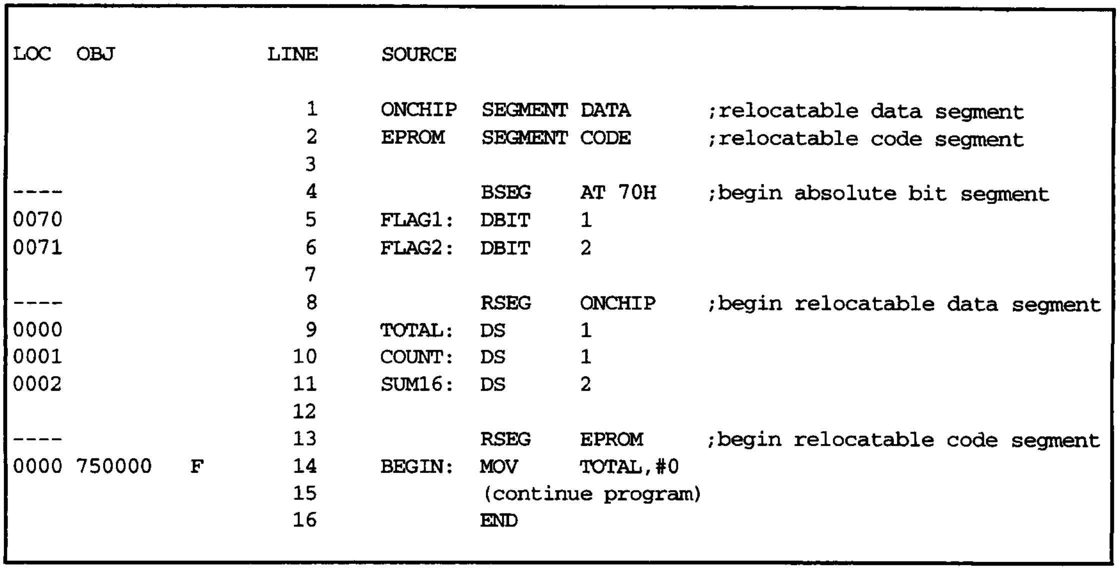
7.5.5.2 Selecting Absolute Segments RSEG selects a relocatable segment. An "absolute" segment, on the other hand, is selected using one of the following directives:

```
CSEG (AT address)
DSEG (AT address)
ISEG (AT address)
BSEG (AT address)
XSEG (AT address)
```

These directives select an absolute segment within the code, internal data, indirect internal data, bit, or external data address spaces, respectively. If an absolute address is provided (by indicating "AT address"), the assembler terminates the last absolute address segment, if any, of the specified segment type and creates a new absolute segment starting at that address. If an absolute address is not specified, the last absolute segment of the specified type is continued. If no absolute segment of this type was previously selected and the absolute address is omitted, a new segment is created starting at location 0. Forward references are not allowed and start addresses must be absolute.

Each segment has its own location counter, which is always set to 0 initially. The default segment is an absolute code segment; therefore, the initial state of the assembler is location 0000H in the absolute code segment. When another segment is chosen for the first time, the location counter of the former segment retains the last active value. When that former segment is reselected, the location counter picks up at the last active value. The ORG directive may be used to change the location counter within the currently selected segment. Figure 7-5 shows examples of defining and initiating relocatable and absolute segments.

The first two lines in Figure 7-5 declare the symbols ONCHIP and EPROM to be segments of type DATA (internal data RAM) and CODE, respectively. Line 4 begins an absolute bit segment starting at bit address 70H (bit 0 of byte address 2EH; see Figure 2-6). Next, FLAG1 and FLAG2 are created as labels corresponding to bit-addressable locations 70H and 71H. RSEG in line 8 begins the relocatable ONCHIP segment for internal data RAM. TOTAL and COUNT are labels corresponding to byte locations. SUM16 is a label corresponding to a word (two-byte) location. The next occurrence of RSEG in line 13 begins the relocatable EPROM segment for code memory. The label BEGIN is the address of the first



```
LOC  OBJ              LINE     SOURCE

                         1     ONCHIP  SEGMENT DATA      ;relocatable data segment
                         2     EPROM   SEGMENT CODE      ;relocatable code segment
                         3
----                     4             BSEG    AT 70H    ;begin absolute bit segment
0070                     5     FLAG1:  DBIT    1
0071                     6     FLAG2:  DBIT    2
                         7
----                     8             RSEG    ONCHIP    ;begin relocatable data segment
0000                     9     TOTAL:  DS      1
0001                    10     COUNT:  DS      1
0002                    11     SUM16:  DS      2
                        12
----                    13             RSEG    EPROM     ;begin relocatable code segment
0000 750000    F        14     BEGIN:  MOV     TOTAL,#0
                        15             (continue program)
                        16             END
```

FIGURE 7-5
Defining and initiating absolute and relocatable segments

instruction in this instance of the EPROM. Note that it is not possible to determine the address of the labels TOTAL, COUNT, SUM 16, and BEGIN from Figure 7-5. Since these labels occur in relocatable segments, the object file must be processed by the linker/locator (see 7.7 Linker Operation) with starting addresses specified for the ONCHIP and EPROM segments. The .M51 listing file created by the linker/locator gives the absolute addresses for these labels. FLAG1 and FLAG2, however, always correspond to bit addresses 70H and 71H because they are defined in an absolute BIT segment.

7.6 ASSEMBLER CONTROLS

Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any effect on the program itself. They can be entered on the invocation line when a program is assembled, or they can be placed in the source file. Assembler controls appearing in the source file must be preceded with a dollar sign and must begin in column 1.

There are two categories of assembler controls: primary and general. Primary controls can be placed in the invocation line or at the beginning of the source program. Only other primary controls may precede a primary control. General controls may be placed anywhere in the source program. Figure 7-6 shows the assembler controls supported by ASM51.

7.7 LINKER OPERATION

In developing large application programs, it is common to divide tasks into subprograms or modules containing sections of code (usually subroutines) that can be written separately from the overall program. The term "modular programming" refers to this programming strategy. Generally, modules are relocatable, meaning they are not intended for a specific address in the code or data space. A linking and locating program is needed to combine the modules into one absolute object module that can be executed.

Intel's RL51 is a typical linker/locator. It processes a series of relocatable object modules as input and creates an executable machine language program (PROGRAM, perhaps) and a listing file containing a memory map and symbol table (PROGRAM.M51). This is illustrated in Figure 7-7.

As relocatable modules are combined, all values for external symbols are resolved with values inserted into the output file. The linker is invoked from the system prompt by

```
RL51 input_list [TO output_list] [location_controls]
```

The input_list is a list of relocatable object modules (files) separated by commas. The output_list is the name of the output absolute object module. If none is supplied, it defaults to the name of the first input file without any suffix. The location_controls set start addresses for the named segments.

For example, suppose three modules or files (MAIN.OBJ, MESSAGES.OBJ, and SUBROUTINES.OBJ) are to be combined into an executable program (EXAMPLE), and that these modules each contain two relocatable segments, one called EPROM of type CODE, and the other called ONCHIP of type DATA. Suppose further that the code segment

NAME	PRIMARY/ GENERAL	DEFAULT	ABBREV.	MEANING
DATE(date)	P	DATE()	DADB	Places string in header (9 char. max.)
DEBUG	P	NODEBUG	NODB	Outputs debug symbol information to object file
NODEBUG	P	NODEBUG	EJ EP	Symbol information not placed in object file
EJECT	G	not applicable		Continue listing on next page
ERRORPRINT(file)	P	NOERRORPRINT	NOEP	Designates a file to receive error messages in addition to the listing file (defaults to console)
NOERRORPRINT	P	NOERRORPRINT	GO	Designates that error messages will be printed in listing file only
GEN	G	GENONLY	NOGE	List only the fully expanded source as if all lines generated by a macro call were already in the source file
GENONLY	G	GENONLY	IC	List only the original source text in the listing file
INCLUDE(file)		not applicable	LI	Designates a file to be included as part of the program
LIST	G	LIST	NOLI	Print subsequent lines of source code in listing file
NOLIST		LIST	MR	Do not print subsequent lines of source code in listing file
MACRO(mem_percent)	G	MACRO(50)	NOMR MO	Evaluate and expand all macro calls. Allocate percentage of free memory for macro processing
NOMACRO	G	MACRO(50)		Do not evaluate macro calls
MOD51	P	MOD51	NOMO	Recognize the 8051-specific predefined special function registers
NOMOD51	P	MOD51		Do not recognize 8051-specific predefined special function registers

FIGURE 7-6

Assembler controls supported by ASM51

NAME	PRIMARY/ GENERAL	DEFAULT	ABBREV.	MEANING
OBJECT(file)	P	OBJECT(source.OBJ)	OJ	Designates file to receive object code
NOOBJECT	P	OBJECT(source.OBJ)	NOOJ	Designates that no object file will be created
PAGING	P	PAGING	PI	Designates that listing file be broken into pages and each will have a header
NOPAGING	P	PAGING	NOPI	Designates that listing file will contain no page breaks
PAGELENGTH(N)	P	PAGELENGT(60)	PL	Sets maximum number of lines in each page of listing file (range = 10 to 65,536)
PAGE WIDTH(N)	P	PAGEWIDTH(120)	PW	Sets maximum number of characters in each line of listing file (range = 72 to 132)
PRINT(file)	P	PRINT(source.LST)	PR	Designates file to receive source listing
NOPRINT	P	PRINT(source.LST)	NOPR	Designates that no listing file will *be* created
SAVE	G	not applicable	SA	Stores current control settings from SAVE stack
RESTORE	G	not applicable	RS	Restores control settings from SAVE stack
REGISTERBANK(rb,...)	P	REGISTERBANK(0)	RB	Indicates one or more banks used in program module
NOREGISTERBANK	P	REGISTERBANK(0)	NORB	Indicates that no register banks are used
SYMBOLS	P	SYMBOLS	SB	Creates a formatted table of all symbols used in program
NOSYMBOLS	P	SYMBOLS	NOSB	Designates that no symbol table is created
TITLE(string)	G	TITLE()	TT	Places a string in allCTubsequent page headers (max. 60 characters)
WORKFILES(path)	P	same as source	WF	Designates alternate path for temporary workfiles
XREF	P	NOXREF	XR	Creates a cross reference listing of all symbols used in program
NOXREF	P	NOXREF	NOXR	Designates that no cross reference list is created

FIGURE 7-6
continued

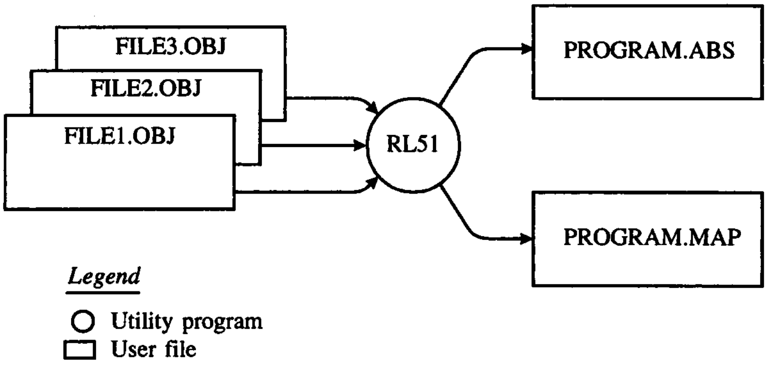


FIGURE 7-7
Linker operation

is to be executable at address 4000H and the data segment is to reside starting at address 30H (in internal RAM). The following linker invocation could be used:

```
RL51 MAIN.OBJ,MESSAGES.OBJ, SUBROUTINES.OBJ TO EXAMPLE & CODE
(EPROM (4000H) ) DATA (ONCHIP (30H) )
```

Note that the ampersand character "&" is used as the line continuation character.

If the program begins at the label START, and this is the first instruction in the MAIN module, then execution begins at address 4000H. If the MAIN module was not linked first, or if the label START is not at the beginning of MAIN, then the program's entry point can be determined by examining the symbol table in the listing file EXAMPLE.M51 created by RL51. By default, EXAMPLE.M51 will contain only the link map. If a symbol table is desired, then each source program must have used the SDEBUG control. (See Figure 7-6.)

7.8 ANNOTATED EXAMPLE: LINKING RELOCATABLE SEGMENTS AND MODULES

Many of the concepts just introduced are now brought together in an annotated example of a simple 8051 program. The source code is split over two files and uses symbols declared as EXTRN or PUBLIC to allow interfile communication. Each file is a module—one named MAIN, the other named SUBROUTINES. The program uses a relocatable code segment named EPROM and a relocatable internal data segment named ONCHIP. Working with multiple files, modules, and segments is essential for large programming projects. A careful examination of the example that follows will strengthen these core concepts and prepare the reader to embark on practical 8051-based designs.

Our example is a simple input/output program using the 8051's serial port and a VDT's keyboard and CRT display. The program does the following:

- Initialize the serial port (once)
- Output the prompt "Enter a command:"

- Input a line from the keyboard, echoing each character as it is received
- Echo back the entire line
- Repeat

Figure 7-8 shows (a) the listing file (ECHO.LST) for the first source file, (b) the listing file (IO.LST) for the second source file, and (c) the listing file (EXAMPLE.M51) created by the linker/locator.

7.8.1 ECHO.LST

Figure 7-8a shows the contents of the file ECHO.LST created by ASM51 when the source file (ECHO.SRC) was assembled. The first several lines in the listing file provide general information on the programming environment. Among other things, the invocation line is restated in an expanded form showing the path to the files. Note the use of the assembler control EP (for ERRORPRINT) on the invocation line. This causes error messages to be sent to the console as well as the listing file. (See Figure 7-6.)

The original source file is shown under the column heading SOURCE, just to the right of the column LINE. As evident, ECHO.SRC contains 22 lines. Lines 1 to 4 contain assembler controls. (See Figure 7-6.) $DEBUG inline 1 instructs ASM51 to place a symbol table in the object file, ECHO.OBJ. This is necessary for hardware emulation or for the linker/locator to create a symbol table in its listing file. $TITLE defines a string to be placed at the top of each page of the listing file. $PAGEW1DTH specifies the maximum width of each line in the listing file. $NOPAGING prevents page breaks (form feeds) from being inserted into the listing file. Most assembler controls affect the look of the output listing file. Some trial and error will usually produce the desire output for printing.

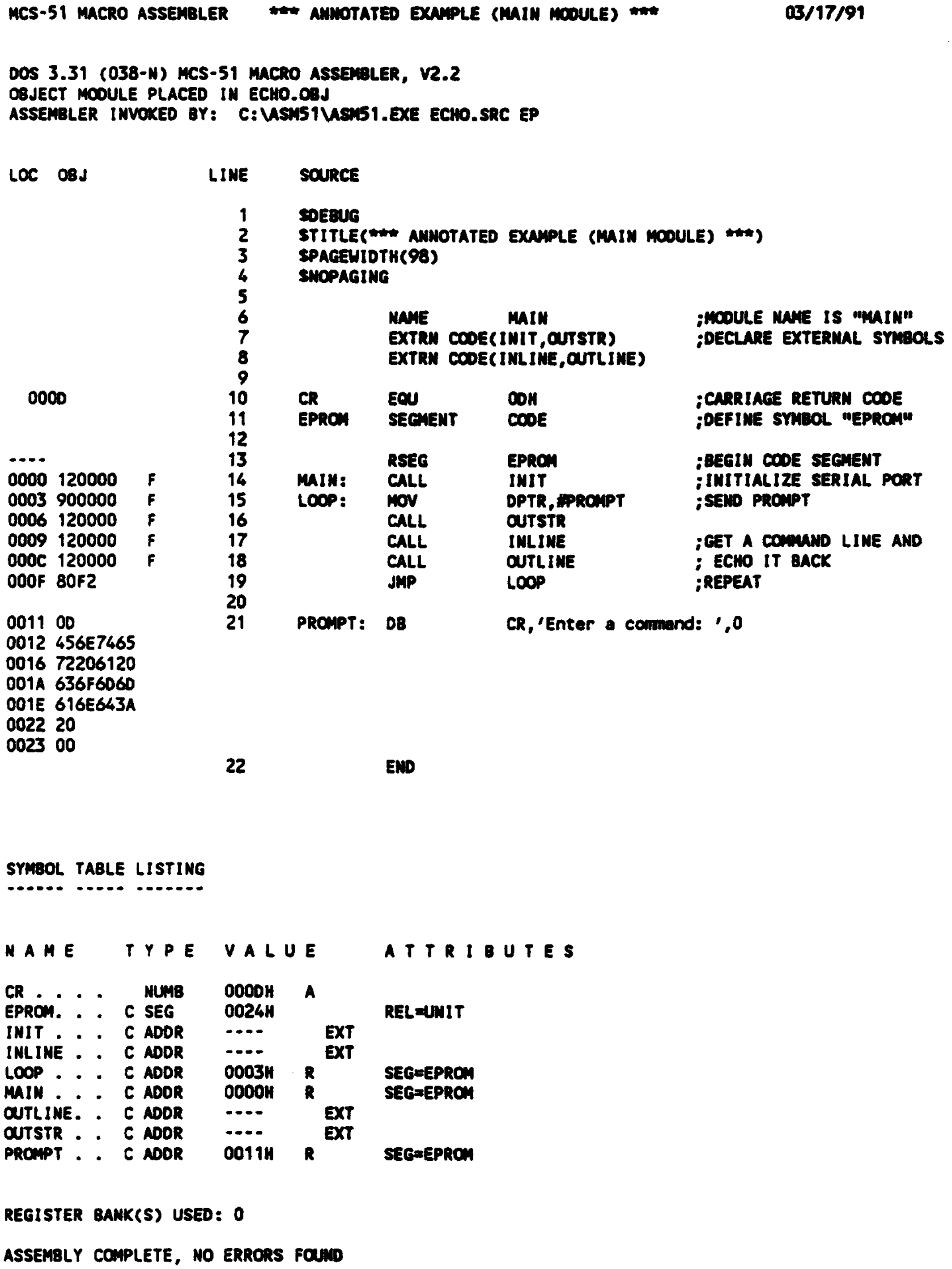
The NAME assembler directive in line 6 defines the current file as part of the module MAIN. For this example, no further instance of the MAIN module is used; however, larger projects may include other files also defined as part of the MAIN module. It may help the reader for the rest of this example to read "file" for the term "module."

Lines 7 and 8 identify the symbols used in the current module but defined elsewhere. Without these EXTRN directives, ASM51 will generate the message "undefined symbol" on each line in the source program where one of these symbols is used. The "segment type" must also be defined for each symbol to ensure its proper use. All of the external symbols defined in this example are of type CODE.

Symbol definitions come next. Line 10 defines the symbol CR as the carriage return ASCII code 0DH. Line 11 defines the symbol EPROM as a segment of type CODE. Recall that the SEGMENT directive defines only what the symbol is—nothing more, nothing less.

The RSEG directive in line 13 begins the relocatable segment named EPROM. Sub Frequent instructions, data constant definitions, and so on, will be placed in the EPROM code segment.

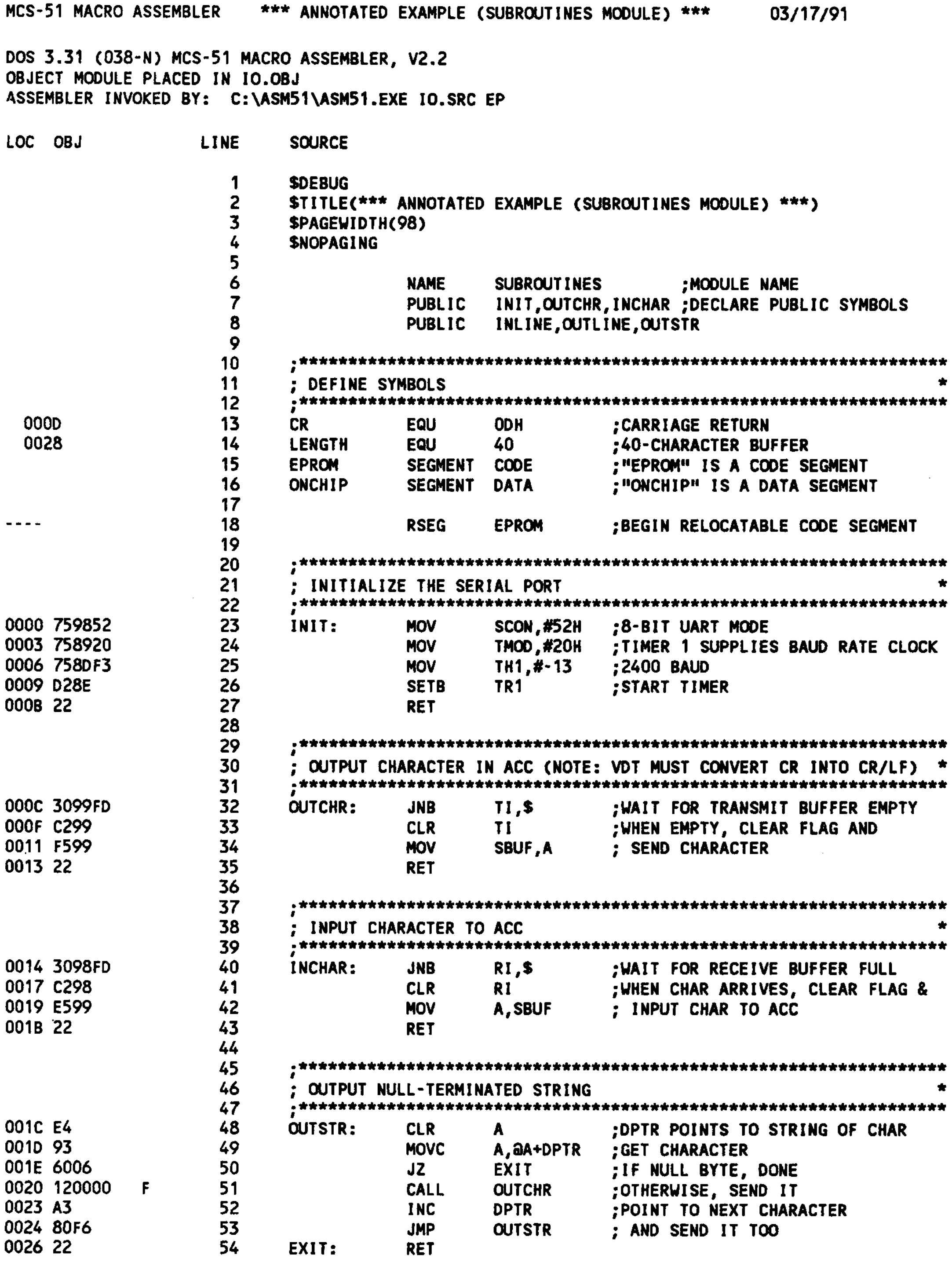
The program begins on line 14 at the label MAIN. The first instruction in the program is a call to the subroutine INIT, which will initialize the 8051's serial port. The assembled code under the OBJ column contains the correct opcode (12H for LCALL); however, bytes 2 and 3 of the instruction (the address of the subroutine) appear as 0000H followed by the letter "F." The linker/locator must "fix" this when the program modules are linked together



```
MCS-51 MACRO ASSEMBLER    *** ANNOTATED EXAMPLE (MAIN MODULE) ***          03/17/91


DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN ECHO.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE ECHO.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $DEBUG
                       2     $TITLE(*** ANNOTATED EXAMPLE (MAIN MODULE) ***)
                       3     $PAGEWIDTH(98)
                       4     $NOPAGING
                       5
                       6             NAME        MAIN              ;MODULE NAME IS "MAIN"
                       7             EXTRN CODE(INIT,OUTSTR)       ;DECLARE EXTERNAL SYMBOLS
                       8             EXTRN CODE(INLINE,OUTLINE)
                       9
  000D                10     CR      EQU         0DH               ;CARRIAGE RETURN CODE
                      11     EPROM   SEGMENT     CODE              ;DEFINE SYMBOL "EPROM"
                      12
----                  13             RSEG        EPROM             ;BEGIN CODE SEGMENT
0000 120000     F     14     MAIN:   CALL        INIT              ;INITIALIZE SERIAL PORT
0003 900000     F     15     LOOP:   MOV         DPTR,#PROMPT      ;SEND PROMPT
0006 120000     F     16             CALL        OUTSTR
0009 120000     F     17             CALL        INLINE            ;GET A COMMAND LINE AND
000C 120000     F     18             CALL        OUTLINE           ; ECHO IT BACK
000F 80F2             19             JMP         LOOP              ;REPEAT
                      20
0011 0D               21     PROMPT: DB          CR,'Enter a command: ',0
0012 456E7465
0016 72206120
001A 636F606D
001E 616E643A
0022 20
0023 00
                      22             END


SYMBOL TABLE LISTING
------ ----- -------


N A M E    T Y P E  V A L U E     A T T R I B U T E S

CR . . . .   NUMB   000DH   A
EPROM. . .  C SEG   0024H            REL=UNIT
INIT . . .  C ADDR  ----      EXT
INLINE . .  C ADDR  ----      EXT
LOOP . . .  C ADDR  0003H   R        SEG=EPROM
MAIN . . .  C ADDR  0000H   R        SEG=EPROM
OUTLINE. .  C ADDR  ----      EXT
OUTSTR . .  C ADDR  ----      EXT
PROMPT . .  C ADDR  0011H   R        SEG=EPROM


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

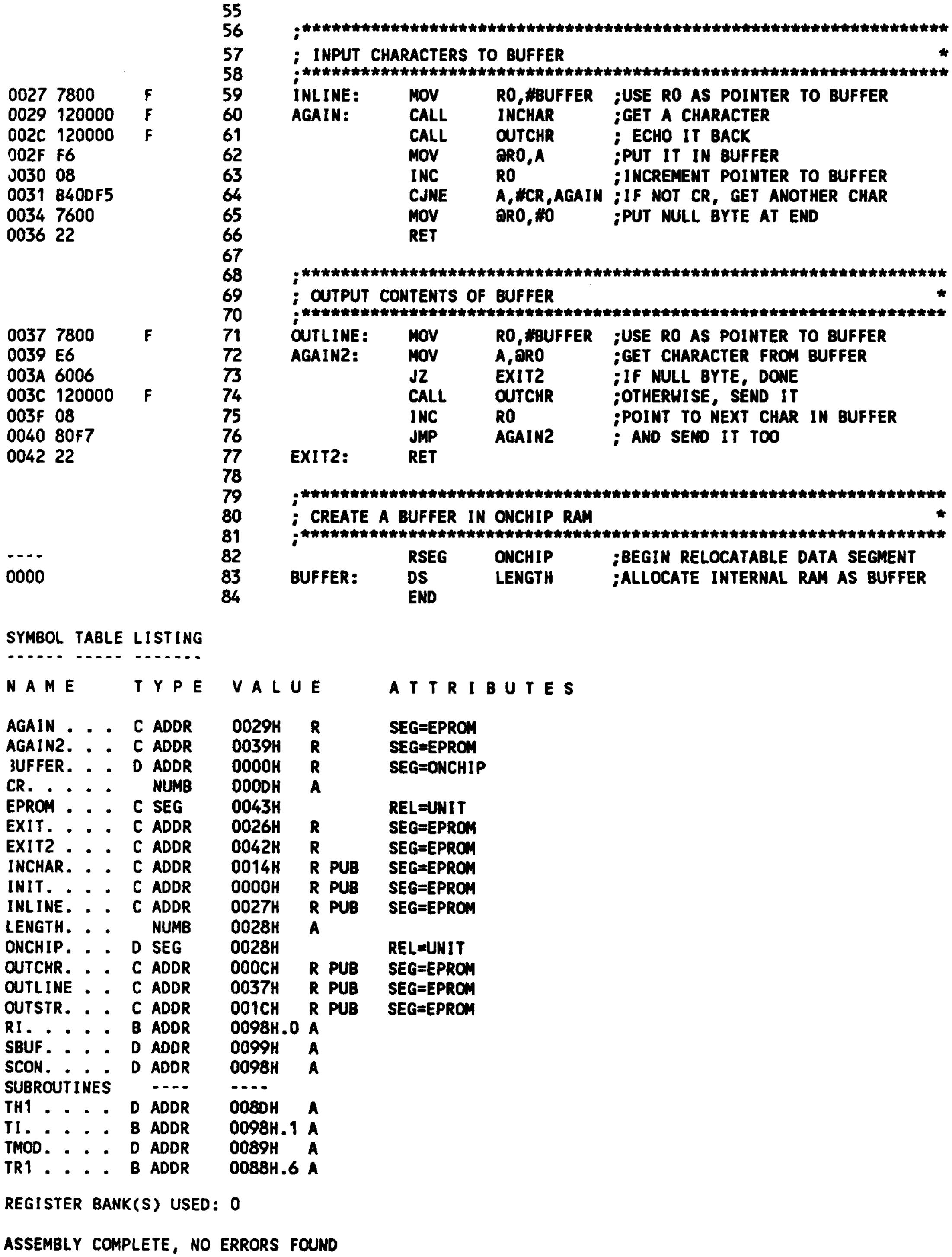
FIGURE 7-8a

Annotated example: linking relocatable segments and modules. (a) ECHO.LST. (b) IO.LST. (c) EXAMPLE.M51.



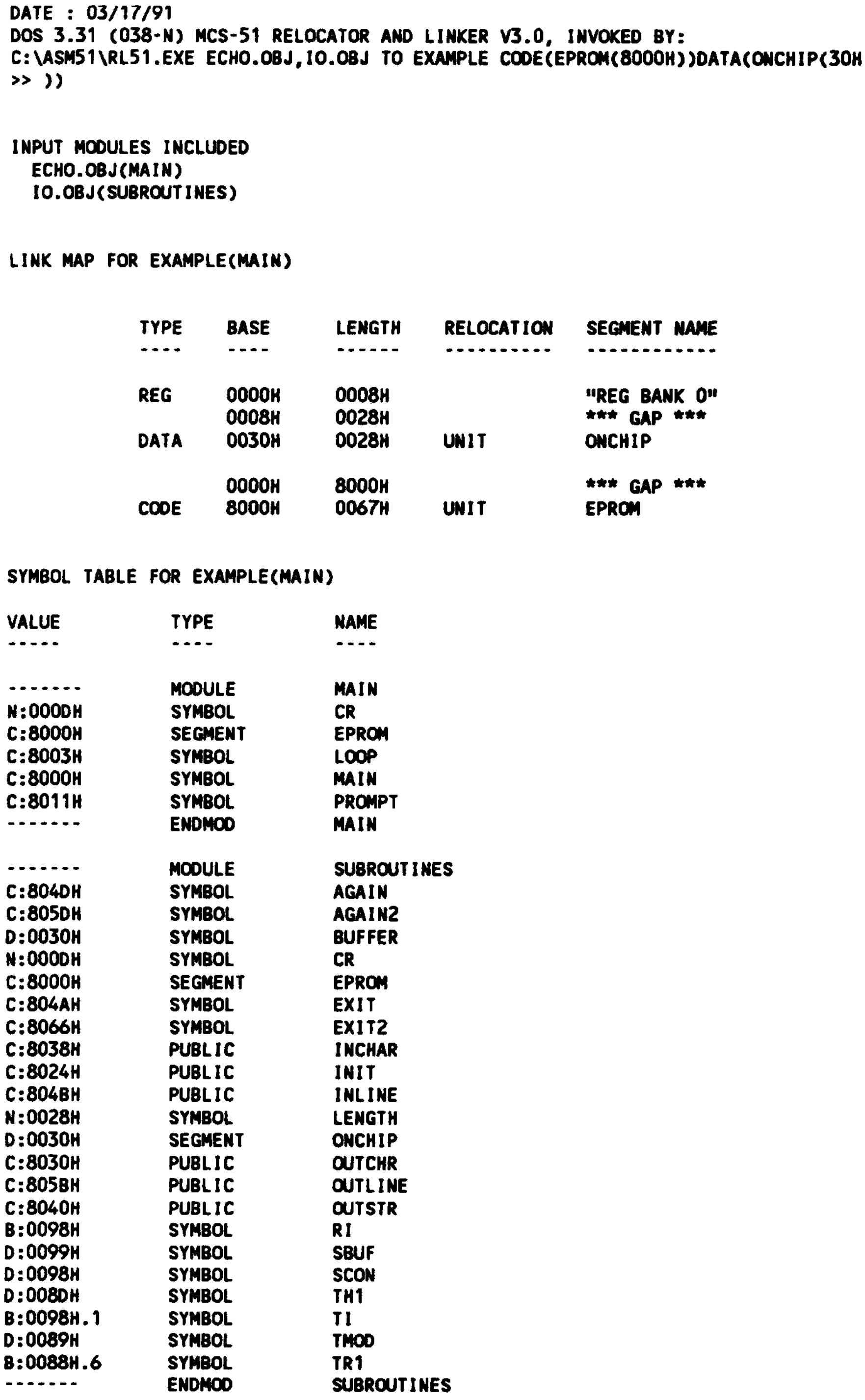
```
MCS-51 MACRO ASSEMBLER    *** ANNOTATED EXAMPLE (SUBROUTINES MODULE) ***      03/17/91

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN IO.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE IO.SRC EP

LOC  OBJ            LINE     SOURCE

                       1     $DEBUG
                       2     $TITLE(*** ANNOTATED EXAMPLE (SUBROUTINES MODULE) ***)
                       3     $PAGEWIDTH(98)
                       4     $NOPAGING
                       5
                       6                 NAME     SUBROUTINES        ;MODULE NAME
                       7                 PUBLIC   INIT,OUTCHR,INCHAR ;DECLARE PUBLIC SYMBOLS
                       8                 PUBLIC   INLINE,OUTLINE,OUTSTR
                       9
                      10     ;******************************************************************
                      11     ; DEFINE SYMBOLS                                                   *
                      12     ;******************************************************************
 000D                 13     CR          EQU      ODH         ;CARRIAGE RETURN
 0028                 14     LENGTH      EQU      40          ;40-CHARACTER BUFFER
                      15     EPROM       SEGMENT  CODE        ;"EPROM" IS A CODE SEGMENT
                      16     ONCHIP      SEGMENT  DATA        ;"ONCHIP" IS A DATA SEGMENT
                      17
----                  18                 RSEG     EPROM       ;BEGIN RELOCATABLE CODE SEGMENT
                      19
                      20     ;******************************************************************
                      21     ; INITIALIZE THE SERIAL PORT                                       *
                      22     ;******************************************************************
0000 759852           23     INIT:       MOV      SCON,#52H   ;8-BIT UART MODE
0003 758920           24                 MOV      TMOD,#20H   ;TIMER 1 SUPPLIES BAUD RATE CLOCK
0006 758DF3           25                 MOV      TH1,#-13    ;2400 BAUD
0009 D28E             26                 SETB     TR1         ;START TIMER
000B 22               27                 RET
                      28
                      29     ;******************************************************************
                      30     ; OUTPUT CHARACTER IN ACC (NOTE: VDT MUST CONVERT CR INTO CR/LF)  *
                      31     ;******************************************************************
000C 3099FD           32     OUTCHR:     JNB      TI,$        ;WAIT FOR TRANSMIT BUFFER EMPTY
000F C299             33                 CLR      TI          ;WHEN EMPTY, CLEAR FLAG AND
0011 F599             34                 MOV      SBUF,A      ; SEND CHARACTER
0013 22               35                 RET
                      36
                      37     ;******************************************************************
                      38     ; INPUT CHARACTER TO ACC                                           *
                      39     ;******************************************************************
0014 3098FD           40     INCHAR:     JNB      RI,$        ;WAIT FOR RECEIVE BUFFER FULL
0017 C298             41                 CLR      RI          ;WHEN CHAR ARRIVES, CLEAR FLAG &
0019 E599             42                 MOV      A,SBUF      ; INPUT CHAR TO ACC
001B 22               43                 RET
                      44
                      45     ;******************************************************************
                      46     ; OUTPUT NULL-TERMINATED STRING                                    *
                      47     ;******************************************************************
001C E4               48     OUTSTR:     CLR      A           ;DPTR POINTS TO STRING OF CHAR
001D 93               49                 MOVC     A,@A+DPTR   ;GET CHARACTER
001E 6006             50                 JZ       EXIT        ;IF NULL BYTE, DONE
0020 120000     F     51                 CALL     OUTCHR      ;OTHERWISE, SEND IT
0023 A3               52                 INC      DPTR        ;POINT TO NEXT CHARACTER
0024 80F6             53                 JMP      OUTSTR      ; AND SEND IT TOO
0026 22               54     EXIT:       RET
```

FIGURE 7-8b
continued



```
                    ;********************************************************************
                    ; INPUT CHARACTERS TO BUFFER                                         *
                    ;********************************************************************
0027 7800      F    INLINE:     MOV     R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0029 120000    F    AGAIN:      CALL    INCHAR      ;GET A CHARACTER
002C 120000    F                CALL    OUTCHR      ; ECHO IT BACK
002F F6                         MOV     @R0,A       ;PUT IT IN BUFFER
0030 08                         INC     R0          ;INCREMENT POINTER TO BUFFER
0031 B40DF5                     CJNE    A,#CR,AGAIN ;IF NOT CR, GET ANOTHER CHAR
0034 7600                       MOV     @R0,#0      ;PUT NULL BYTE AT END
0036 22                         RET

                    ;********************************************************************
                    ; OUTPUT CONTENTS OF BUFFER                                          *
                    ;********************************************************************
0037 7800      F    OUTLINE:    MOV     R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0039 E6             AGAIN2:     MOV     A,@R0       ;GET CHARACTER FROM BUFFER
003A 6006                       JZ      EXIT2       ;IF NULL BYTE, DONE
003C 120000    F                CALL    OUTCHR      ;OTHERWISE, SEND IT
003F 08                         INC     R0          ;POINT TO NEXT CHAR IN BUFFER
0040 80F7                       JMP     AGAIN2      ; AND SEND IT TOO
0042 22             EXIT2:      RET

                    ;********************************************************************
                    ; CREATE A BUFFER IN ONCHIP RAM                                      *
                    ;********************************************************************
----                            RSEG    ONCHIP      ;BEGIN RELOCATABLE DATA SEGMENT
0000                BUFFER:     DS      LENGTH      ;ALLOCATE INTERNAL RAM AS BUFFER
                                END

SYMBOL TABLE LISTING
------ ----- -------

N A M E      T Y P E    V A L U E      A T T R I B U T E S

AGAIN . . .  C ADDR     0029H   R      SEG=EPROM
AGAIN2. . .  C ADDR     0039H   R      SEG=EPROM
BUFFER. . .  D ADDR     0000H   R      SEG=ONCHIP
CR. . . . .    NUMB     000DH   A
EPROM . . .  C SEG      0043H          REL=UNIT
EXIT. . . .  C ADDR     0026H   R      SEG=EPROM
EXIT2 . . .  C ADDR     0042H   R      SEG=EPROM
INCHAR. . .  C ADDR     0014H   R PUB  SEG=EPROM
INIT. . . .  C ADDR     0000H   R PUB  SEG=EPROM
INLINE. . .  C ADDR     0027H   R PUB  SEG=EPROM
LENGTH. . .    NUMB     0028H   A
ONCHIP. . .  D SEG      0028H          REL=UNIT
OUTCHR. . .  C ADDR     000CH   R PUB  SEG=EPROM
OUTLINE . .  C ADDR     0037H   R PUB  SEG=EPROM
OUTSTR. . .  C ADDR     001CH   R PUB  SEG=EPROM
RI. . . . .  B ADDR     0098H.0 A
SBUF. . . .  D ADDR     0099H   A
SCON. . . .  D ADDR     0098H   A
SUBROUTINES    ----     ----
TH1 . . . .  D ADDR     008DH   A
TI. . . . .  B ADDR     0098H.1 A
TMOD. . . .  D ADDR     0089H   A
TR1 . . . .  B ADDR     0088H.6 A

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

FIGURE 7-8c
continued



```
DATE : 03/17/91
DOS 3.31 (038-N) MCS-51 RELOCATOR AND LINKER V3.0, INVOKED BY:
C:\ASM51\RL51.EXE ECHO.OBJ,IO.OBJ TO EXAMPLE CODE(EPROM(8000H))DATA(ONCHIP(30H
>> ))


INPUT MODULES INCLUDED
  ECHO.OBJ(MAIN)
  IO.OBJ(SUBROUTINES)


LINK MAP FOR EXAMPLE(MAIN)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
                    0008H     0028H                  *** GAP ***
            DATA    0030H     0028H     UNIT         ONCHIP

                    0000H     8000H                  *** GAP ***
            CODE    8000H     0067H     UNIT         EPROM


SYMBOL TABLE FOR EXAMPLE(MAIN)

VALUE          TYPE           NAME
-----          ----           ----

-------        MODULE         MAIN
N:000DH        SYMBOL         CR
C:8000H        SEGMENT        EPROM
C:8003H        SYMBOL         LOOP
C:8000H        SYMBOL         MAIN
C:8011H        SYMBOL         PROMPT
-------        ENDMOD         MAIN

-------        MODULE         SUBROUTINES
C:804DH        SYMBOL         AGAIN
C:805DH        SYMBOL         AGAIN2
D:0030H        SYMBOL         BUFFER
N:000DH        SYMBOL         CR
C:8000H        SEGMENT        EPROM
C:804AH        SYMBOL         EXIT
C:8066H        SYMBOL         EXIT2
C:8038H        PUBLIC         INCHAR
C:8024H        PUBLIC         INIT
C:804BH        PUBLIC         INLINE
N:0028H        SYMBOL         LENGTH
D:0030H        SEGMENT        ONCHIP
C:8030H        PUBLIC         OUTCHR
C:805BH        PUBLIC         OUTLINE
C:8040H        PUBLIC         OUTSTR
B:0098H        SYMBOL         RI
D:0099H        SYMBOL         SBUF
D:0098H        SYMBOL         SCON
D:008DH        SYMBOL         TH1
B:0098H.1      SYMBOL         TI
D:0089H        SYMBOL         TMOD
B:0088H.6      SYMBOL         TR1
-------        ENDMOD         SUBROUTINES
```

FIGURE 7-8d
continued

and addresses are set for the relocatable segments. Note, too, that the address under the LOC column is also entered as 0000H. Since the EPROM segment is relocatable, it is not known at assemble-time where the segment will start. All relocatable segments will display 0000H as the starting address in the listing file.

The rest of the program instructions are on lines 15 to 19. A prompt message is sent to the VDT by loading the DPTR with a starting address of the prompt and calling the subroutine OUTSTR. Since the OUTSTR, INLINE, and OUTLINE subroutines are not defined in ECHO.SRC, one can only guess at their operation from the name of the subroutine and the comment lines.

The prompt is a null-terminated ASCII string, which is placed in the EPROM code segment using the DB (define byte) directive on line 21. Since the prompt bytes are constant (i.e., unchanging), it is correct to place them in code memory (even though they are data bytes). The prompt begins with a carriage return to ensure it displays on a new line. (In this example, it is assumed the VDT converts CR to CR/LF.)

All the symbols and labels in ECHO.SRC appear in the symbol table at the bottom of ECHO.LST. Since the EPROM segment is relocatable and the subroutines are external, the VALUE column is not of much use. The value for the symbol EPROM, however, gives the length of the segment, which in this case is 24H or 36 bytes.

7.8.2 IO.LST

Figure 7-8b shows the contents of the file IO.LST—the file containing the input/output subroutines. This module is named SUBROUTINES in line 6. Lines 7 and 8 declare all subroutine names as PUBLIC symbols. This makes these symbols available to other modules. Note that all the subroutines are made public even though only four of them were used in the MAIN module. Perhaps, as the program grows, other modules will be added that may need these subroutines. So, they are all made public.

Lines 13 to 16 define several symbols. Once again, EPROM is used as the name of the code segment. Another segment is used in this module. ONCHIP is defined in line 17 as an internal data segment.

The subroutines are each written in turn beginning at line 20. The comment block beginning each is deliberately brief in this example; however, a more detailed description of a subroutine is usually given. It is useful to provide, for example, entry and exit conditions for each subroutine.

After the last subroutine, a buffer in internal RAM is created using the ONCHIP segment. The segment is started using RSEG (line 82), and the buffer is created using the DS (define storage) directive (line 83). The length of the buffer is assigned to the symbol LENGTH "equated" at the top of the program (line 14) as 40. The placement in the source file of the definition of the symbol LENGTH and of the instance of the segment ONCHIP is largely a matter of taste. Both could also be positioned just before or after the INLINE subroutine, where they are used.

As with the EPROM segment, ONCHIP is given an initial address of 0000H under the LOC column at line 83. Again, the actual location of the ONCHIP segment will not be determined until link-time (see below). The letter "F' appears in numerous locations in IO.LST. Each line so identified contains an instruction using a symbol whose value cannot be determined at assemble-time. The zeros placed in the object file at these locations will be replaced with "absolute" values by the linker/locator.

7.8.3 EXAMPLE.M51

Figure 7-8c shows the contents of the file EXAMPLE.M51 created by the linker/locator program, RL51. The invocation line is repeated near the top of EXAMPLE.M51 and should be examined carefully. Here it is again (leaving out the path):

```
RL51 ECHO.OBJ,IO.OBJ TO EXAMPLE CODE (EPROM (8000H)) & DATA
(ONCHIP (30H))
```

Following the command, the object modules are listed separated by commas in the order they are to be linked. Following the input list, the optional control TO EXAMPLE is specified providing the name for the absolute object module created by RL51. If omitted, the name of the first file in the input list is used (without any file extension). The listing file, in this example, automatically takes on the name EXAMPLE.M51. Finally, the locating controls CODE and DATA specify the names of segments of the associated type and the absolute address at which the segment is to begin. In this example the EPROM code segment begins at address 8000H and the ONCHIP data segment begins at byte address 30H in the 8051's internal RAM.

Following the restatement of the invocation line, EXAMPLE.M151 contains a list of the input modules included by RL51. In this example only two files (ECHO.OBJ and IO.OBJ) and two modules (MAIN and SUBROUTINES) are listed. If the NAME directive had not been used in the source files, the module names would be the same as the file names.

The link map appears next. Both the ONCHIP and EPROM segments are identified, and the starting address and the length (in hexadecimal) are given for each. ONCHIP is identified as a data segment starting at address 30H and 28H (40) bytes in length. EPROM is identified as a code segment starting at address 8000H and 67H (103) bytes in length.

Finally, EXAMPLE.M51 contains a symbol table. All symbols (including labels) used in the program are listed, sorted on a module-by-module basis. All values are "absolute." Remember that the symbol table in the .M51 file can only be created if the $DEBUG assembler control is placed at the top of each source file. The INIT subroutine address (which we noted earlier was absent in ECHO.LST) is identified under the SUBROUTINES module as 8024H. This address is substituted as the code address in any object module using the instruction CALL INIT, as noted earlier in the MAIN module. Knowing the absolute value of labels is important when debugging. When a bug is found, often a temporary "patch" can be made by modifying the program bytes and re-executing the program. If the patch fixes the bug, the appropriate change is made to the source program.

7.9 MACROS

For the final topic in this chapter, we return to ASM51. The macro processing facility (MPL) of ASM51 is a "string replacement" facility. Macros allow frequently used sections of code to be defined once using a simple mnemonic and used anywhere in the program by inserting the mnemonic. Programming using macros is a powerful extension of the techniques described thus far. Macros can be defined anywhere in a source program and subsequently used like any other instruction. The syntax for a macro definition is

```
%*DEFINE (call_pattern) (macro_body)
```

Once defined, the call pattern is like a mnemonic; it may be used like any assembly language instruction by placing it in the mnemonic field of a program. Macros are made distinct from "real" instructions by preceding them with a percent sign, "%". When the source program is assembled, everything within the macro-body, on a character-by-character basis, is substituted for the call-pattern. The mystique of macros is largely unfounded. They provide a simple means for replacing cumbersome instruction patterns with primitive, easy-to-remember mnemonics. The substitution, we reiterate, is on a character-by-character basis-nothing more, nothing less.

For example, if the following macro definition appears at the beginning of a source file,

```
%*DEFINE (PUSH_DPTR)
             (PUSH DPH
              PUSH DPL
              )
```

then the statement

```
%PUSHDPTR
```

will appear in the .LST file as

```
PUSH DPH
PUSH DPL
```

The example above is a typical macro. Since the 8051 stack instructions operate only on direct addresses, pushing the data pointer requires two PUSH instructions. A similar macro can be created to POP the data pointer.

There are several distinct advantages in using macros:

- A source program using macros is more readable, since the macro mnemonic is generally more indicative of the intended operation than the equivalent assembler instructions.
- The source program is shorter and requires less typing.
- Using macros reduces bugs.
- Using macros frees the programmer from dealing with low-level details.

The last two points above are related. Once a macro is written and debugged, it is used freely without the worry of bugs. In the PUSH_DPTR example above, if PUSH and POP instructions are used rather than push and pop macros, the programmer may inadvertently reverse the order of the pushes or pops. (Was it the high-byte or low-byte that was pushed first?) This would create a bug. Using macros, however, the details are worked out once—when the macro is written—and the macro is used freely thereafter, without the worry of bugs.

Since the replacement is on a character-by-character basis, the macro definition should be carefully constructed with carriage returns, tabs, etc., to ensure proper alignment of the macro statements with the rest of the assembly language program. Some trial and error is required.

There are advanced features of ASM51's macro-processing facility that allow for parameter passing, local labels, repeat operations, assembly flow control, and so on. These are discussed below.

7.9.1 Parameter Passing

A macro with parameters passed from the main program has the following modified

format:

```
%*DEFINE (macro_name (parameter_list)) (macro_body)
```

For example, if the following macro is defined,

```
%*DEFINE (CMPA# (VALUE))
       (CINE A,#%VALUE,$ + 3
        )
```

then the macro call

```
%CPA# (20H)
```

will expand to the following instruction in the .LST file:

```
CJNE A,#20H,$ + 3
```

Although the 8051 does not have a "compare accumulator" instruction, one is easily created using the CJNE instruction with "$ +3" (the next instruction) as the destination for the conditional jump. The CMPA# mnemonic may be easier to remember for many programmers. Besides, use of the macro unburdens the programmer from remembering notational details, such as " $ +3."

Let's develop another example. It would be nice if the 8051 had instructions such as

```
JUMP IF ACCUMULATOR GREATER THAN X
JUMP IF ACCUMULATOR GREATER THAN OR EQUAL TO X
JUMP IF ACCUMULATOR LESS THAN X
JUMP IF ACCUMULATOR LESS THAN OR EQUAL TO X
```

but it does not. These operations can be created using CJNE followed by JC or JNC, but the details are tricky. Suppose, for example, it is desired to jump to the label GREATER_THAN if the accumulator contains an ASCII code greater than "Z" (5AH). The following instruction sequence would work:

```
CJNE A,#5BH,$+3
JNC  GREATER_THAN
```

The CJNE instruction subtracts 5BH (i.e., "Z" + 1) from the content of A and sets or clears the carry flag accordingly. CJNE leaves C = 1 for accumulator values 00H up to and including 5AH. (Note: 5AH - 5BH < 0, therefore C = 1; but 5BH - 5BH = 0, therefore C = 0.) Jumping to GREATER_THAN on the condition "not carry" correctly jumps for accumulator values 5BH, 5CH, 5DH, and so on, up to FFH. Once details such as these are worked out, they can be simplified by inventing an appropriate mnemonic, defining a macro, and using the macro instead of the corresponding instruction sequence. Here's the definition for a "jumps if greater than" macro:

```
%*DEFINE (JGT(VALUE, LABEL))
           (CJNE A,#%VALUE+1,$+3 ;JGT
           JNC %LABEL
           )
```

To test if the accumulator contains an ASCII code greater than "Z," as just discussed, the macro would be called as

```
%JGT ('Z',GREATER_THAN)
```

ASM5 1 would expand this into

```
CJNE A,#5BH,$+3      ;JGT
JNC  GREATER_THAN
```

The JGT macro is an excellent example of a relevant and powerful use of macros. By using macros, the programmer benefits by using a meaningful mnemonic and avoiding messy and potentially bug-ridden details.

7.9.2 Local Labels

Local labels may be used within a macro using the following format:

```
%*DEFINE(macro_name [(parameter_list)])
           [LOCAL list_of_local_labels] (macro_body)
```

For example, the following macro definition

```
%*DEFINE (DEC_DPTR) LOCAL SKIP
           (DEC DPL            ;DECREMENT DATA POINTER
           MOV A,DPL
           CJNE A,#0FFH,%SKIP
           DEC DPH
%SKIP:
```

would be called as

```
           %DECDPTR
```

and would be expanded by ASM51 into

```
           DEC DPL             ;DECREMENT DATA POINTER
           MOV A,DPL
           CJNE A, #0FFH, SKIP00
           DEC DPH
SKIP00
```

Note that a local label generally will not conflict with the same label used elsewhere in the source program, since ASM51 appends a numeric code to the local label when the macro is expanded. Furthermore, the next use of the same local label receives the next numeric code, and so on.

The macro above has a potential "side effect." The accumulator is used as a temporary holding place for DPL. If the macro is used within a section of code that uses A for another purpose, the value in A would be lost. This side effect probably represents a bug in the program. The macro definition could guard against this by saving A on the stack. Here's an alternate definition for the DEC_DPTR macro:

```
%*DEFINE (DEC_DPTR) LOCAL SKIP
          (PUSHACC
          DEC DPL              ;DECREMENT DATA POINTER
          MOV A,DPL
          CJNE A,#0FFH,%SKIP
          DEC DPH
%SKIP:    POP ACC
          )
```

7.9.3 Repeat Operations

This is one of several built-in (predefined) macros. The format is

```
%REPEAT (expression) (text)
```

For example, to fill a block of memory with 100 NOP instructions,

```
%REPEAT
(100) (NOP
)
```

7.9.4 Control Flow Operations

The conditional assembly of sections of code is provided by ASM51's control flow macro definition. The format is

```
%IF(expression) THEN (balanced_text)
[ELSE (balanced_text) ]FI
```

For example,

```
INTERNAL EQU 1           ;1 = 8051 SERIAL I/O DRIVERS
                         ;0 = 8251 SERIAL I/O DRIVERS
          .
          .
          %IF (INTERNAL) THEN
(INCHED:   .             ;8051 DRIVERS

OUTCHR:
```

```
                )ELSE
(INCHED:                        ;8251 DRIVERS

OUTCHR:
```

In this example, the symbol INTERNAL is given the value 1 to select I/O subroutines for the 8051's serial port, or the value 0 to select I/O subroutines for an external UART, in this case the 8251. The IF macro causes ASM51 to assemble one set of drivers and skip over the other. Elsewhere in the program, the INCHAR and BUTCHER subroutines are used without consideration for the particular hardware configuration. As long as the program as assembled with the correct value for INTERNAL, the correct subroutine is executed.

SUMMARY

This chapter has presented major concepts of assembly language programming the 8051, including standard program formats and the use of assembler directives and macros to help make a program more readable, compact, and orderly.

As program sizes increase, one wonders if there are other ways to write 8051 programs in a more structured and English-like manner. In the next chapter, we will discuss such an alternative, the 8051 C language.

PROBLEMS

7.1 Recast the following instructions with the operand expressed in binary.

```
MOV A,#255
MOV A,#11Q
MOV A,#11Q
MOV A,#'A'
```

7.2 Recast each of the following instructions with the operand expressed in hexadecimal.

```
MOV A,#255
MOV A,#-3
MOV A,#'z'
MOV A,#'33Q
MOV A,#'$'
MOV A,#64
```

7.3 What is wrong with the coding of the following instruction?

```
ORL 80H,#F0H
```

7.4 Identify the error in the following symbols.

```
?byte.bit
@GOOD_bye
1ST_FLAG
MY_PROGRAM
```

7.5 Recast the following instructions with the expression evaluated as a 16-bit hexadecimal constant.

```
MOV DPTR, # 'C' EQ 48
MOV DPTR,# HIGH'AB'
MOV DPTR,#-1
MOV DPTR,#NOT (257 MOD 256)
```

7.6 Recast each of the following instructions with the source location specified as a hexadecimal address, rather than in symbolic form.

```
MOV A, PSW
MOV A, P0
MOV A, DPH
MOV A, TLC
MOV A, IP
MOV A, TMOD
```

7.7 Recast each of the following instructions with the source location specified in symbolic form, rather than as a hexadecimal address.

```
MOV A,0B0H
MOV A,99H
MOV A,82H
MOV A,85H
MOV A,0A8H
MOV A,0D0H
```

7.8 What are the "segment types" defined by ASM51 for the 8051, and what memory spaces do they represent?

7.9 How could a relocatable segment in external data memory be defined, selected, and a 100-byte buffer created? (Give the segment the name "OFFCHIP" and give the buffer the name "BUFFER.")

7.10 A certain application requires five status bits (FLAG1 to FLAGS). How could a 5-bit buffer be defined in an absolute BIT segment starting at bit address 0811? At what byte address do these bits reside?

7.11 What are two good reasons for making generous use of the EQU directive in assembly language programs?

7.12 What is the difference between the DB and DW directives?

7.13 What are the memory assignments for the following assembler directives:

```
ORG 0FH
DW $ SHL4 4
D5   65535
DW   'C'
```

7.14 What directive is used to select an absolute code segment?

7.15 A file called "ASCII" contains 33 equate directives, one for each control code:

```
NUL EQU 0CH    ;NULL 5YTE
SOH EQU 01H    ;STEDT OF HEADER



US  EQU 7FH    ;UNIT SEPARATOR
DEL EQU 7FH    ;DELETE
```

How could these definitions be made known in another file—a source program—without actually inserting the equates into that file?

7.16 In order for a printout of a listing file to look nice, it is desirable to have each sub-routine begin at the top of a page. How is this accomplished?

7.17 Write the definition for a macro that could be used to fill a block of external data memory with a data constant. Pass the starting address, length, and data constant to the macro as parameters.

7.18 Write the definition for the following macros:

JGE-jump to LABEL if accumulator is greater than or equal to VALUE
JLT-jump to LABEL if accumulator is less than VALUE
JGE-jump to LABEL if accumulator is less than or equal to VALUE
JGE-jump to LABEL if accumulator is outside the range LOWER and UPPER

7.19 Write the definition for a macro called CJNE DPTR that will jump to LABEL if the data pointer does not contain VALUE. Define the macro so that the contents of all registers and memory locations are left intact.

CHAPTER 8

8051 C Programming

8.1 INTRODUCTION

Throughout the earlier chapters, we have discussed how to communicate with the 8051 by using the assembly language. In fact, there is another way by which we can talk to the 8051. This is called 8051 C language, an often-preferred choice when the complexity of a program increases considerably. This chapter introduces the 8051 C language as an alternative to the assembly language programming. The choice of which to use is entirely up to you, the programmer. Factors that commonly influence such a decision are the desired speed, code size, and ease of programming. Since the intent of this chapter is to present the basics of programming the 8051 in C, we will assume that the reader is already familiar with conventional C programming, as is often the case due to C's popularity and widespread usage.

8.2 ADVANTAGES AND DISADVANTAGES OF 8051 C

The advantages of programming the 8051 in C as compared to assembly are:

- Offers all the benefits of high-level, structured programming languages such as C, including the ease of writing subroutines, and others discussed in more detail in Section 9.2
- Often relieves the programmer of the hardware details that the compiler handles on behalf of the programmer
- Easier to write, especially for large and complex programs
- Produces more readable program source codes

Nevertheless, 8051 C, being very similar to the conventional C language, also suffers from the following disadvantages:

- Possesses the disadvantages of high-level, structured programming languages, as mentioned in Section 9.2
- Generally generates larger machine codes
- Programmer has less control and less ability to directly interact with the hardware

To compare between 8051 C and assembly language, consider the solution to Example 4.5, written below in 8051 C language.

```
sbit portbit = P1^0; /* Use variable portbit to refer to P1.0 */
main( )
TMOD = 1;
{
 while (1)
     {
      TH0 = 0xFE;
      TL0 = 0xC;
      TR0 = 1;
      while (TF0 != 1);
      TR0 = 0;
      TF0 = 0;
      portbit = !(P1^0);
      }
}
```

Notice that both the assembly and C language solutions for Example 4.5 require almost the same number of lines. However, the difference lies in the readability of these programs. The C version seems more human than assembly, and is hence more readable. This often helps facilitate the human programmer's efforts to write even very complex programs. The assembly language version is more closely related to the machine code, and though less readable, often results in more compact machine code. As with this example, the resultant machine code from the assembly version takes 83 bytes while that of the C version requires 149 bytes, an increase of 79.5%!

The human programmer's choice of either high-level C language or assembly language for talking to the 8051, whose language is machine language, presents an interesting picture, as shown in Figure 8-1.

8.3 8051 C COMPILERS

We saw in Figure 8-1 that a compiler is needed to convert programs written in 8051 C language into machine language, just as an assembler is needed in the case of programs written in assembly language. A compiler basically acts just like an assembler, except that it is more complex since the difference between C and machine language is far greater than that between assembly and machine language. Hence the compiler faces a greater task to bridge that difference.

Currently, there exist various 8051 C compilers, which offer almost similar functions. All our examples and programs have been compiled and tested with Keil's µVision 2 IDE

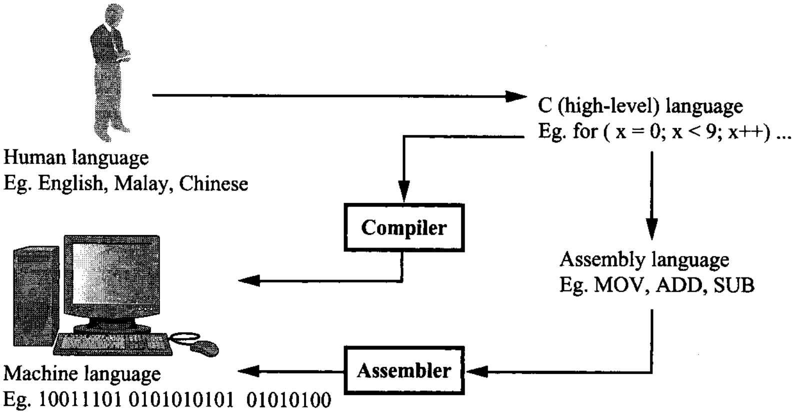


FIGURE 8-1
Conversion between human, high-level, assembly, and machine language

by Keil Software,[1] an integrated 8051 program development environment that includes its C51 cross compiler for C (Refer to Appendix H for a guide to using µVision 2 IDE). A cross compiler is a compiler that normally runs on a platform such as IBM compatible PCs but is meant to compile programs into codes to be run on other platforms such as the 8051.

8.4 DATA TYPES

8051 C is very much like the conventional C language, except that several extensions and adaptations have been made to make it suitable for the 8051 programming environment. The first concern for the 8051 C programmer is the data types. Recall that a data type is something we use to store data. Readers will be familiar with the basic C data types such as int, char, and float, which are used to create variables to store integers, characters, or floating-points. In 8051 C, all the basic C data types are supported, plus a few additional data types meant to be used specifically with the 8051.

Table 8-1 gives a list of the common data types used in 8051 C. The ones in bold are the specific 8051 extensions. The data type **bit** can be used to declare variables that reside in the 8051's bit-addressable locations (namely byte locations 20H to 2FH or bit locations 00H to 7FH). Obviously, these bit variables can only store bit values of either 0 or 1. As an example, the following C statement:

```
bit flag = 0;
```

declares a **bit** variable called flag and initializes it to 0.

The data type **sbit** is somewhat similar to the **bit** data type, except that it is normally used to declare 1-bit variables that reside in special function registers (SFRs). For example:

```
sbit P = 0xD0;
```

[1] Keil Software, Inc., 1501 10$^{a'}$ Street, Suite 110, Plano, TX 75074. Website: http://www.keil.com

TABLE 8-1
Data types used in 8051 C language

Data Type	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2,147,483,648 to +2,147,483,647
unsigned long	32	4	0 to 4,294,967,295
float	32	4	+1.175494E-38 to +3.402823E+38
sbit	1		0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

declares the sbit variable P and specifies that it refers to bit address 0D0H, which is really the LSB of the PSW SFR. Notice the difference here in the usage of the assignment ('=') operator. In the context of **sbit** declarations, it indicates what address the **sbit** variable resides in, while in **bit** declarations, it is used to specify the initial value of the **bit** variable.

Besides directly assigning a bit address to an **sbit** variable, we could also use a previously defined **sfr** variable as the base address and assign our **sbit** variable to refer to a certain bit within that **sfr.** For example:

```
sfr PSW = 0xD0;
sbit P  = PSW^0;
```

This declares an **sfr** variable called PSW that refers to the byte address 0D0H and then uses it as the base address to refer to its LSB (bit 0). This is then assigned to an **sbit** variable, P. For this purpose, the carat symbol (^) is used to specify bit position 0 of the PSW.

A third alternative uses a constant byte address as the base address within which a certain bit is referred. As an illustration, the previous two statements can be replaced with the following:

```
sbit P = 0xD0^0;
```

Meanwhile, the **sfr** data type is used to declare byte (8-bit) variables that are associated with SFRs. The statement:

```
sfr IE = 0xA8;
```

declares an **sfr** variable IE that resides at byte address 0A8H. Recall that this address is where the Interrupt Enable (IE) SFR is located; therefore, the **sfr** data type is just a means to enable us to assign names for SFRs so that it is easier to remember.

The sfr16 data type is very similar to sfr but, while the sfr data type is used for 8-bit SFRs, sfr16 is used for 16-bit SFRs. For example, the following statement:

```
sfr16 DPTR = 0x82;
```

declares a 16-bit variable DPTR whose lower-byte address is at 82H. Checking through the 8051 architecture, we find that this is the address of the DPL SFR, so again, the sfr16 data type makes it easier for us to refer to the SFRs by name rather than address. There's just one thing left to mention. When declaring **sbit, sfr,** or **sfr16** variables, remember to do so outside main, otherwise you will get an error.

In actual fact though, all the SFRs in the 8051, including the individual flag, status, and control bits in the bit-addressable SFRs have already been declared in an include file, called **reg51.h,** which comes packaged with most 8051 C compilers. The contents of reg51.h are listed in Figure 8-2. By using reg51.h, we can refer for instance to the interrupt enable register as simply IE rather than having to specify the address A8H, and to the data pointer as DPTR rather than 82H. All this makes 8051 C programs more human-readable and manageable.

```
/*----------------------

REG51.H
Header file for generic 80051 and 80C31 microcontroller.
Copyright (c) 1988-2001 Keil Elektronik GmbH and Keil Software,
Inc.
All rights reserved.*/

/* BYTE Register */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
```

FIGURE 8-2a
Listing of reg51.h

```
/* BIT Register
*/ /* PSW */
sbit CY    = 0xD7;
sbit AC    = 0xD6;
sbit F0    = 0xD5;
sbit RS1   = 0xD4;
sbit RS0   = 0xD3;
sbit OV    = 0xD2;
sbit P     = OxDO;
/* TCON */
sbit TF1   = 0x8F;
sbit TR1   = 0x8E;
sbit TF0   = 0x8D;
sbit TR0   = 0x8C;
sbit IE1   = 0x8B;
sbit IT1   = 0x8A;
sbit IE0   = 0x89;
sbit IT0   = 0x88;

/* IE */
sbit EA    = 0xAF;
sbit ES    = 0xAC;
sbit ET1   = 0xAB;
sbit EX1   = 0xAA;
sbit ET0   = 0xA9;
sbit EX0   = 0xA8;

/* IP */
sbit PS    = 0xBC;
sbit PT1   = 0xBB;
sbit PX1   = 0xBA;
sbit PT0   = 0xB9;
sbit PX0   = 0xB8;

/* P3 */
sbit RD    = 0xB7;
sbit WR    = 0xB6;
sbit T1    = 0xB5;
sbit T0    = 0xB4;
sbit INT1  = 0xB3;
sbit INT0  = 0xB2;
sbit TXD   = 0xBl;
sbit RXD   = 0xB0;

/* SCON */
sbit SM0   = 0x9F;
sbit SM1   = 0x9E;
sbit SM2   = 0x9D;
sbit REN   = 0x9C;
sbit TB8   = 0x9B;
sbit RB8   = 0x9A;
sbit TI    = 0x99;
sbit RI    = 0x98;
```

FIGURE 8-2b
continued

8.5 MEMORY TYPES AND MODELS

The 8051 has various types of memory space, including internal and external code and data memory. When declaring variables, it is hence reasonable to wonder in which type of memory those variables would reside. For this purpose, several memory type specifiers are available for use, as shown in Table 8-2.

The first memory type specifier given in Table 8-2 is code. This is used to specify that a variable is to reside in code memory, which has a range of up to 64 Kbytes. For example:

```
char code errormsg [1 = "An error occurred" ;
```

declares a char array called errormsg that resides in code memory.

If you want to put a variable into data memory, then use either of the remaining five data memory specifiers in Table 8-2. Though the choice rests on you, bear in mind that each type of data memory affects the speed of access and the size of available data memory. For instance, consider the following declarations:

```
signed int   data num1;
bit          bdata numbit;
unsigned int xdata num2;
```

The first statement creates a signed int variable **num1** that resides in *internal data memory* (00H to 7FH). The next line declares a bit variable **numbit** that is to reside in the bit-addressable memory locations (byte addresses 20H to 2FH), also known as *bdata*. Finally, the last line declares an unsigned int variable called **num2** that resides in external data memory, *xdata*. Having a variable located in the directly addressable internal data memory speeds up access considerably; hence, for programs that are time-critical, the variables should be of type data. For the 8052 and other variants with internal data memory up to 256 bytes, the idata specifier may be used. Note however that this is slower than data since it must use indirect addressing. Meanwhile, if you would rather have your variables reside in external memory, you have the choice of declaring them as pdata or xdata. A variable declared to be in **pdata** resides in the first 256 bytes (a page) of external memory, while if more storage is required, **xdata** should be used, which allows for accessing up to 64 Kbytes of external data memory.

What if when declaring a variable you forget to explicitly specify what type of memory it should reside in, or you wish that all variables are assigned a default memory type

TABLE 8-2
Memory types used in 8051 C language

Memory Type	Description (Size)
code	Code memory (64 Kbytes)
data	Directly addressable internal data memory (128 bytes)
idata	Indirectly addressable internal data memory (256 bytes)
bdata	Bit-addressable internal data memory (16 bytes)
xdata	External data memory (64 Kbytes)
pdata	Paged external data memory (256 bytes)

TABLE 8-3
Memory models used in 8051 C language

Memory Model	Description
Small	Variables default to the internal data memory (data)
Compact	Variables default to the first 256 bytes of external data memory (pdata)
Large	Variables default to external data memory (xdata)

without having to specify them one by one? In this case, we make use of **memory models.** Table 8-3 lists the various memory models that you can use.

A program is explicitly selected to be in a certain memory model by using the C directive, **#pragma**. Otherwise, the default memory model is **small.** It is recommended that programs use the small memory model as it allows for the fastest possible access by defaulting all variables to reside in internal data memory.

The **compact** memory model causes all variables to default to the first page of external data memory while the **large** memory model causes all variables to default to the full external data memory range of up to 64 Kbytes.

8.6 ARRAYS

Often, a group of variables used to store data of the same type need to be grouped together for better readability. For example, the ASCII table for decimal digits would be as in Table 8-4. To store such a table in an 8051 C program, an **array** could be used. An array is a group of variables of the same data type, all of which could be accessed by using the name of the array along with an appropriate index.

The array to store the decimal ASCII table is:

```
   int table([0] =
{0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38, 0x39};
```

TABLE 8-4
ASCII table for decimal digits

Decimal Digit	ASCII Code In Hex
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H

Notice that all the elements of an array are separated by commas. To access an individual element, an index starting from 0 is used. For instance, **`table[0]`** refers to the first element while **`table[9]`** refers to the last element in this ASCII table.

8.7 STRUCTURES

Sometimes it is also desired that variables of different data types but which are related to each other in some way be grouped together. For example, the name, age, and date of birth of a person would be stored in different types of variables, but all refer to the person's personal details. In such a case, a **structure** can be declared. A structure is a group of related variables that could be of different data types. Such a structure is declared by:

```
struct person {
     char name[10];
     int age;
     long DOB;
     };
```

Once such a structure has been declared, it can be used like a data type specifier to create structure variables that have the member's name, age, and DOB. For example:

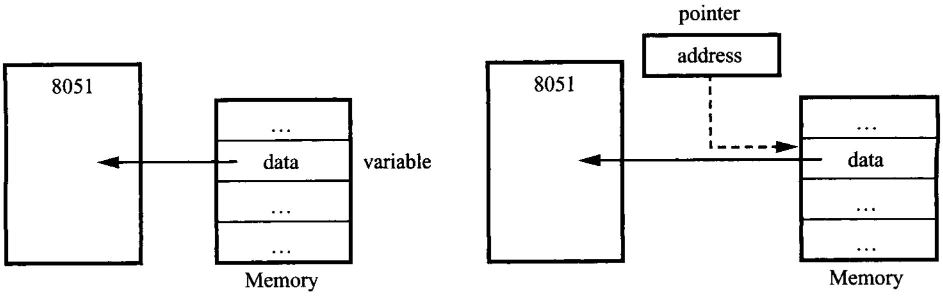
```
struct person grace = {"Grace", 22, 01311980};
```

would create a structure variable grace to store the name, age, and date of birth of a person called Grace. Then in order to access the specific members within the per son structure variable, use the variable name followed by the dot operator (.) and the member name. Therefore, **`grace.name, grace.age, grace.DOB`** would refer to Grace's name, age, and date of birth, respectively.

8.8 POINTERS

When programming the 8051 in assembly, sometimes registers such as R0, R1, and DPTR are used to store the addresses of some data in a certain memory location. When data is accessed via these registers, indirect addressing is used. In this case, we say that R0, R1, or DPTR are used to point to the data, so they are essentially pointers.

Correspondingly in C, indirect access of data can be done through specially defined pointer variables. Many students studying C shy away from pointers. In fact, pointers are simply just special types of variables, but whereas normal variables are used to directly store data, pointer variables are used to store the addresses of the data. Just bear in mind that whether you use normal variables or pointer variables, you still get to access the data in the end. It is just whether you go directly to where it is stored and get the data, as in the case of normal variables, or first consult a directory to check the location of that data before going there to get it, as in the case of pointer variables. Figure 8-3 contrasts between how data is accessed via normal variables and via pointers.



(a) Via Normal Variables

(b) Via Pointer Variables

FIGURE 8-3
Different ways of accessing data. (a) Via normal variables (b) Via pointer variables

Declaring a pointer follows the format:

```
data_type * pointer_name;
```

where

data_type refers to which type of data that the pointer is pointing to
denotes that this is a pointer variable
pointer_name is the name of the pointer

As an example, the following declarations:

```
int * numPtr;
int num;
numPtr = &num;
```

first declares a pointer variable called **numPtr** that will be used to point to data of type int. The second declaration declares a normal variable and is put there for comparison. The third line assigns the address of the num variable to the **numPtr** pointer. The address of any variable can be obtained by using the *address operator*, **&**, as is used in this example. Bear in mind that once assigned, the **numPtr** pointer contains the address of the **num** variable, not the value of its data.

The above example could also be rewritten such that the pointer is straightaway initialized with an address when it is first declared:

```
int num;
int * numPtr = &num;
```

In order to further illustrate the difference between normal variables and pointer variables, consider the following, which is not a full C program but simply a fragment to illustrate our point:

```
int num = 7;
int * numPtr = &num;
printf("%d\n", num);
printf("%d\n", numPtr);
printf("%d\n", &num;)
printf("%d\n", *numPtr);
```

The first line declares a normal variable, `num,` which is initialized to contain the data 7. Next, a pointer variable, **`numPtr,`** is declared, which is initialized to point to the address of num. The next four lines use the **`printf ()`** function, which causes some data to be printed to some display terminal connected to the serial port. The first such line displays the contents of the `num` variable, which is in this case the value 7. The next displays the contents of the **`numPtr`** pointer, which is really some weird-looking number that is the address of the `num` variable. The third such line also displays the address of the num variable because the address operator is used to obtain **`num`**`'s` address. The last line displays the actual data to which the **`numPtr`** pointer is pointing, which is 7. The * symbol is called the *indirection operator,* and when used with a pointer, indirectly obtains the data whose address is pointed to by the pointer. Therefore, the output display on the terminal would show:

```
7
13452 (or some other weird-looking number)
13452 (or some other weird-looking number)
7
```

Later, after we have covered Section 8.10, which discusses how to write complete C programs, you will be able to verify the output for yourself.

8.8.1 A Pointer's Memory Type

Recall that pointers are also variables, so the question arises where they should be stored. When declaring pointers, we can specify different types of memory areas that these pointers should be in, for example:

```
int * xdata numPtr = &num;
```

This is the same as our previous pointer examples. We declare a pointer `numPtr,` which points to data of type int stored in the `num` variable. The difference here is the use of the memory type specifier **xdata** after the * . This specifies that the pointer `numPtr` should reside in external data memory **(xdata),** and we say that the *pointer's memory type* is **xdata.**

8.8.2 Typed Pointers

We can go even further when declaring pointers. Consider the example:

```
int data * xdata numPtr = &num;
```

The above statement declares the same pointer numPtr to reside in external data memory **(xdata),** and this pointer points to data of type int that is itself stored in the variable `num` in internal data memory **(data).** The memory type specifier, **data,** before the * specifies the *data memory type* while the memory type specifier, **xdata,** after the * specifies the pointer memory type.

TABLE 8-5
Data memory type values stored in first byte of untyped pointers

Value	Data Memory Type
1	idata
2	xdata
3	pdata
4	data/bdata
5	code

Pointer declarations where the data memory types are explicitly specified are called *typed pointers.* Typed pointers have the property that you specify in your code where the data pointed by pointers should reside. The size of typed pointers depends on the data memory type and could be of one or two bytes.

8.8.3 Untyped Pointers

When we do not explicitly state the data memory type when declaring pointers, we get *untyped pointers,* which are generic pointers that can point to data residing in any type of memory. Untyped pointers have the advantage that they can be used to point to any data independent of the type of memory in which the data is stored. All untyped pointers consist of 3 bytes, and are hence larger than typed pointers. Untyped pointers are also generally slower because the data memory type is not determined or known until the complied program is run at runtime. The first byte of untyped pointers refers to the data memory type, which is simply a number according to Table 8-5. The second and third bytes are, respectively, the higher-order and lower-order bytes of the address being pointed to.

An untyped pointer is declared just like normal C, where:

```
int * xdata numPtr = &num;
```

does not explicitly specify the memory type of the data pointed to by the pointer. In this case, we are using untyped pointers.

8.9 FUNCTIONS

In programming the 8051 in assembly, we learnt the advantages of using subroutines to group together common and frequently used instructions. The same concept appears in 8051 C, but instead of calling them subroutines, we call them **functions.** As in conventional C, a function must be declared and defined. A function definition includes a list of the number and types of inputs, and the type of the output (return type), plus a description of the internal contents, or what is to be done within that function.

The format of typical function definition is as follows:

```
return_type function_name(arguments) [memory] [reentrant]
                                     [interrupt] [using]
{
    •
    •
 }
```

where[2]

`return_type`	refers to the data type of the return (output) value
`function_name`	is any name that you wish to call the function as
`arguments`	is the list of the type and number of input (argument) values
`memory`	refers to an explicit memory model (small, compact or large)
`reentrant`	refers to whether the function is reentrant (recursive)
`interrupt`	indicates that the function is actually an ISR
`Using`	explicitly specifies which register bank to use

Consider a typical example, a function to calculate the sum of two numbers:

```
int sum(int a,int b)
    {
    return a + b;
    }
```

This function is called sum and takes in two arguments, both of type int. The return type is also int, meaning that the output (return value) would be an int. Within the body of the function, delimited by braces, we see that the return value is basically the sum of the two arguments. In our example above, we omitted explicitly specifying the options: memory, reentrant, interrupt, and us ing. This means that the arguments passed to the function would be using the default small memory model, meaning that they would be stored in internal data memory. This function is also by default non-recursive and a normal function, not an ISR. Meanwhile, the default register bank used is bank 0.

8.9.1 Parameter Passing

In 8051 C, parameters are passed to and from functions and used as function arguments (inputs). Nevertheless, the technical details of where and how these parameters are stored are transparent to the programmer, who does not need to worry about these technicalitie2rheowever, we will briefly discuss in this section how parameters are passed in order to appreciate how the compiler takes care of this on our behalf. In 8051 C, parameters are passed through the registers or through memory. Passing parameters through registers is faster and is the default way in which things are done. The registers used and their purpose are described in more detail in Table 8-6.

TABLE 8-6
Registers used in parameter passing

Number of Argument	Char/1-Byte Pointer	INT/2-Byte Pointer	Long/Float	Generic Pointer
1	R7	R6 & R7	R4--R7	R1-R3
2	R5	R4 & R5	R4-R7	
3	R3	R2 & R3		

[2] The statements delimited by square brackets indicate that they are optional.

TABLE 8-7
Registers used in returning values from functions

Return Type	Register	Description
Bit	Carry Flag (C)	
char/unsigned char/1-byte pointer	R7	
int/unsigned int/2-byte pointer	R6&R7	
long/unsigned long	R4-R7	MSB in R6, LSB in R7 MSB
float	R4-R7	in R4, LSB in R7 32-bit IEEE
generic pointer	R1-R3	format Memory type in R3,MSB in R2, LSB in R1

Since there are only eight registers in the 8051, there may be situations where we do not have enough registers for parameter passing. When this happens, the remaining parameters can be passed through fixed memory locations. To specify that all parameters will be passed via memory, the NOREGPARMs control directive is used. To specify the reverse, use the REGPARMs control directive.

8.9.2 Return Values

Unlike parameters, which can be passed by using either registers or memory locations, output values must be returned from functions via registers. Table 8-7 shows the registers used in returning different types of values from functions.

8.10 SOME 8051 C EXAMPLES

We have so far briefly covered the basic concepts of programming the 8051 in C. In this section, we will look at various examples of how to write such programs. All the 8051 C programs discussed henceforth have been compiled and tested on Keil's µV***ision2,*** which is an 8051 integrated development environment (IDE) that includes source code editing, compiling, and debugging.[3]

8.10.1 The First Program

As is customary in any introductory course on high-level, structured proi,amming languages, the first demonstrative example is a simple program that displays the message "Hello World,"

```
#include <REG51.H>                /* SFR declarations */
#include <stdio.h>                /* Declarations for I/O
                                     functions(eg. printf) */

main ()
{
SCON = 0x52;                      /* serial port, mode 1 */
TMOD = 0x20;                      /* timer 1, mode 2 */
```

[3] A free evaluation version is available at http://www.keil.com.

```
    TH1 = -13;                        /* reload count for 2400
                                         baud */
    TR1 = 1;                          /* start timer 1 */

    while (1)                         /* repeat forever */
          {
          printf ("Hello World\n");   /* Display "Hello World" */
          }
    }
```

The program above continually displays the message "Hello World" to the device attached to the serial port, which is by default connected to a simulated serial window. For further details about interacting with µVision2 IDE, refer to Appendix H or Keil's documentations in µVision2's Help.

8.10.2 Timers

In Chapter 4, ideas of how to interact with the 8051's built-in timers were discussed. As a basis of comparison between programming the 8051 in assembly language and in C, we will give here C solutions for some of the Chapter 4 examples.

EXAMPLE 8.1 **Pulse Wave Generation**

Write a program that creates a periodic waveform on P1.0 with as high a frequency as possible.

Solution

```
    #include <REGS1.H>      /* SFR declarations */
    shit portbit = Pl^0;    /* Use variable portbit to refer to
                               Pi.0 */
    main 0
    {
    while (1)               /* repeat forever */
          {
          portbit = 1;      /* set P1.0 */
          portbit = 0;      /* clear P1.0 */
          }
    }
```

Discussion

This program is actually the C version for Example 4.2. In the program, port P1.0 needs to be set and cleared repeatedly. This is done by simply writing a 1 and 0 to it. Since P1 has already been declared in the `REG51.H` header file, we can refer to its LSB by declaring an shit variable and using that in the main program.

There is one thing that we should note when making use of instructions to generate timing delays and waveforms. In assembly, we could easily do so by taking note of how many machine cycles it takes to execute a certain instruction. In C, however, it is cannot be directly determined how long it would take to execute a certain statement, loop, or functions. All these depend on the specific C compiler used, since different compilers use different levels of optimization and hence generate different combinations of assembly language instructions and subsequently machine code.

The best way is to look at the specific assembly language instructions generated by the compiler for a certain C statement and determine how many machine cycles they would take. In µVision2 IDE, you can disassemble your C program into assembly language by viewing the Disassembly Window. Refer to Appendix H or Keil's Help documentation for more information.

EXAMPLE 8.2 **10 kHz Square Wave**

Write a program using Timer 0 to create a 10kHz square wave on P1.0.

Solution

```
#include <REG51.H>              /* SRR declarations */
shit portbit = P1^0;             /* Use variable portbit to
                                    refer to Pi.H */

main ()
{
TMOD = 2;                       /* 8-bit auto-reload mode */
TH0 = -50;                      /* -50 reload value in TH0
TR0 = 1;                        /* start timer 0 */

while (i)                       /* repeat forever */
      {
      while (TF0 != 1);         /* wait for overflow */
      TF0 = 0;                  /* clear timer overflow flag */
      portbit = !(portbit);     /* toggle Pi.0 */
      }
}
```

Discussion

This is really the C version for Example 4.4. For an infinite loop, the `while` statement is used, with its condition a constant value of 1. In C, 1 denotes TRUE whereas a 0 denotes FALSE. Hence, in this case the condition would always be true, causing the while statement to be repeated indefinitely.

EXAMPLE 8.3 **1 kHz Square Wave**

Write a program using Timer 0 to create a 1 kHz square wave on P1.0.

Solution

```
#include <REGS1.H>              /* SFR declarations */
sbit portbit = P1^0;            /* Use variable portbit to
                                   refer to Pi.0 */
```

```
main ()
{
TMOD = 1;                     /* 16-bit timer mode */
while (1)                     /* repeat forever */
      {
      TH0 = 0xFE;             /* -500 (high byte)
      TL0 = 0x0C;             /* -500 (low byte)
      TR0 = 1;                /* start timer 0 */
      while (TH0 != 1);       /* wait for overflow */
      TR0 = 0;                /* stop timer 0 */
      TF0 = 0;                /* clear timer overflow flag */
      portbit = =(portbit);   /* toggle Pi.0 */
      }
}
```

Discussion

This is the C version for Example 4.5 and is very similar to the previous example.

EXAMPLE 8.4 Buzzer Interface

A buzzer is connected to P1.7, and a debounced switch is connected to P1.6. Write a program that reads the logic level provided by the switch and sounds the buzzer for 1 second for each 1-to-0 transition detected.

Solution

```
#include <REG51.H>            /* SFR declarations */

int hundred = 100;

sbit inbit = P1^6;            /* Use variable inbit to refer to
                                 P1.6 */
sbit outbit = P1^7;           /* Use variable outbit to refer to
                                 Pi.7 */
unsigned char R7;             /* use 8-bit variable to represent
                                 R7 */

void delay(void);             /* Function prototype */
main ()
{
TMOD = 1;                     /* use timer 0 in mode 1 */
while (1)                     /* repeat forever */
      {
      while (inbit!=1);      /* wait for 1 input*/
      while (inbit==1);      /* wait for 0 input*/
      outbit = 1;            /* turn buzzer on */
      delay();               /* wait for 1 second */
      outbit = 0;            /* turn buzzer off */
      }
 }
```

```
void delay(void)
{
R7 = hundred;
do
      {
      TH0 = 0xD8;          /* -10000 (high byte) */
      TL0=  0xF0;          /* -10000 (low byte) */
      TR0 = 1;
      while (TF0 != 1);
      TF0 = 0;
      TR0 = 0;
      R7 -= 1;
while (R7!=0);
}
```

Discussion

This example uses a function called **delay()** together with the main function. Since the function is defined after main, a on-line function prototype should be added prior to main to allow the compiler to know that such a function exists. R7 is also not defined in the **REG51.H** header file, so it is defined here before being used in the **delay()** function. This is done to maintain direct correspondence to the assembly language solution of this example problem in Example 4.7. In reality when one programs in C, rather than using the general purpose registers R0 to R7, one would simply declare variables instead. This is also because the registers R0 to R7 are used by the compiler for parameter passing to and from functions.

8.10.3 Serial Port

In this section, we will give example solutions to the examples previously discussed in Chapter 5 concerning the 8051 serial port.

EXAMPLE 8.5 **Initializing the Serial Port**

Write an instruction sequence to initialize the serial port to operate as an 8-bit UART at 2400 baud. Use Timer 1 to provide the baud rate clock.

Solution

```
#include <REG51> /* SFR declarations */

main ()
{
SCON = 0x52;       /* serial port, mode 1 */
TMOD = 0x20;       /* timer 1, mode 2 */
TH1 = -13;         /* reload count for 2400 baud */
TR1 = 1;           /* start timer 1 */
}
```

EXAMPLE 8.6 Output Character Subroutine

Write a function called OUTCHR to transmit the 7-bit ASCII code in the accumulator out the 8051 serial port, with odd parity added as the 8th bit. Return from the function with the accumulator intact.

Solution

```
#include <REG51.H>    /* SFR declarations */

shit AccMSB = ACC^ 7; /* Use variable AccMSE to refer to
                         ACC.7 */

void OUTCHR(void)
{
CY = P;               /* put parity hit in C flag */
CY = !CY;             /* change to odd parity */
AccMSB = CY;          /* add to character code */
while (TI != 1);      /* Tx empty? no: check again */
TI = 0;               /* yes: clear flag and */
SEUF = ACC;           /* send character */
AccMSB = 0;           /* strip off parity hit */
}
```

Discussion

In the program, the MSB of the accumulator, ACC needs to be accessed so an shit variable, `AccMSE,` is declared that refers to this MSB.

EXAMPLE 8.7 Input Character Subroutine

Write a function called INCHAR to input a character from the 8051's serial port and return with the 7-bit ASCII code in the accumulator. Expect odd parity in the eighth bit received and set the carry flag if there is a parity error.

Solution

```
#include <REG51.H>    /* SFR declarations */
sbit AccMSB = ACC^7;  /* Use variable AccMSB to refer to
                         ACC.7 */

void INCHAR(void)
{
while (RI != 1);      /* wait for character */
RI = 0;               /* clear flag */
ACC = SBUF;           /* read char into accumulator */
CY = P;               /* for odd parity in accumulator, P
                         should he set */
CY = !CY;             /* complementing correctly indicates ii
                         "error" */
AccMSB = 0;           /* strip off parity */
}
```

8.10.4 Interrupts

From our discussion of functions in Section 8.9, we know that interrupt service routines (ISRs) are written in very much the same way as normal functions, with the exception that the "interrupt" statement is used in the function definition. We will now show how the examples from Chapter 6 on interrupts can be solved with programs written in 8051 C.

EXAMPLE 8.8 **Square Wave Using Timer Interrupts**

Write a program using Timer 0 and interrupts to create a 10 kHz square wave on P1.0.

Solution

```
#include <REG51.H>     /* SFR declarations */

sbit portbit = P1^0;   /* Use variable portbit to refer
                          to P1.0 */

main ()
{
TMOD = 0x02;           /* timer 0, mode 2 */
TH0  = -50;            /* 50 ps delay */
TR0  = 1;              /* start timer */
IE   = 0x82;           /* enable timer 0 interrupt */
while(1);              /* repeat forever */
}


void T0ISR(void) interrupt 1
{
portbit = !portbit;    /* toggle port hit Pi.0 */
}
```

Discussion

This program introduces the use of an interrupt function, a special type of function that is automatic whenever the corresponding interrupt occurs. Note that the interrupt statement is followed by an interrupt number, in this case, 1, which refers to a timer/counter 0 interrupt. Table 8-8 gives a list of the various interrupt numbers, types, and vector addresses.

EXAMPLE 8.9 **Two Square Waves Using Timer Interrupts**

Write a program using interrupts to simultaneously create kHz and 500Hz square waves on P1.7 and P1.6.

Solution

```
#include <REG51.H>     /* SFR declarations */

sbit portsev = P1^7; /* Use variable portsev to refer to
                          P1.7 */
sbit portsix = P1^6; /* Use varibhle portsix to refer to
                          P1.6 */
```

TABLE 8-8
Standard 8051 interrupts and interrupt numbers

Interrupt Number	Description	Vector Address
0	External INT 0	0003H
1	Timer/Counter 0	000BH
2	External INT 1	0013H
3	Timer/Counter 1	0018H
4	Serial Port	0023H

```
main ()
{
TMOD = 0x12;          /* timer 1, mode 1; twhile(i);ode 2 */
TH0 = -71;             /* 7kHz using timer 0*/
TR0 = 1;               /* start timer */
TF1 = 1;               /* force timer 1 interrupt */
IE = 0x8A;             /* enable both timer interrupts */
while(1);              /* repeat forever */
}

void T0ISR(void) interrupt 1
{
portsev = !portsev;    /* toghit Pi.6 bit P1.7 */
}

void T1ISR(void) interrupt 3
{
TR1 = 0;
TH1 = OxFC;            /* 1 lms high time & */
TL1 = 0x18;            /*        low time */
TR1 = 1;
portsix = !portsix;    /* toggle port bit P1.6 */
}
```

Discussion

In this example, two interrupt functions are used to service Timer 0 and Timer 1 interrupts. As can be seen from Table 8-8, the corresponding interrupt numbers are 1 and 3, respectively.

EXAMPLE 8.10

Character Output Using Interrupts

Write a program using interrupts to continually transmit the ASCII code set (excluding control codes) to a terminal attached to the 8051's serial port.

Solution

```
#include <REG51.H> /* SFR declarations

*/ main()

{
TMOD = 0x20;       /* timer 1, mode 2 */
TH1  = -26;        /* 12000 baud reload value */
TR1 = 1;            /* start timer */
SCON = 0x42;        /* mode 1, set TI to force 18t interrupt */
ACC  = 0x20;        /* send ASCII space first */
IE   = 0x90;        /* enahle serial port interrupt */
while(1);           /* repeat forever */
}

void SPISR(void) interrupt 4
{
if (ACC == 0x7F)    /* if finished ASCII set */
     ACC = 0x20;    /*      reset to space */
SBUF = ACC;         /* send char. to serial port */
ACC = ACC + 1;      /* increment ASCII code */
TI = 0;             /* clear interrupt flag */
}
```

Discussion

This program uses an interrupt function to service a serial port interrupt request, where the interrupt number used is 4. Note also that even though there is no restriction on what name you should use for an interrupt function, the convention is to use explicit names such as SPISR and T0ISR to refer to these interrupt functions.

EXAMPLE 8.11 **Furnace Controller**

Using interrupts, design an 8051 furnace controller that keeps a building at 20°C ± 1°C.

Solution

```
#include <REG51.H>    /* SFR declarations */

sbit  outbit  =  P1^7;  /*  Use  variahle  outbit  to  refer  to
                           P1.7 */
sbit  hotbit  =  P3^2;  /* Use  variahle  hotbit  to  refer  to  P3.2
                           */

main ()
{
IE = 0x85;              /* enahle external interrupts */
IT0 = 1;                /* negative edge triggered */
IT1 = 1;
outbit = 1;             /* turn furnace on */
if (hotbit != 1)        /* if T > 21 degrees, */
     outbit = 0;        /*   turn it off */
while(1);               /* repeat forever
*/
}

void EX0ISR(void) interrupt 0
{
outhit = 0;             /* turn furnace off */
}
```

```
void EX0ISR(void) interrupt 2
{
outbit = 1;           /* turn furnace on */
}
```

Discussion

Again, as this example shows, writing interrupt functions to service external interrupts is similar to writing such functions for timer and serial port interrupts. The difference lies in the interrupt numbers being used, which can be obtained from Table 8-8. Note again that the interrupt functions have been explicitly named.

EXAMPLE 8.12 **Intrusion Warning System**

Design an intrusion warning system using interrupts that sounds a 400Hz tone for 1 second (using a loudspeaker connected to P1.7) whenever a door sensor connected to INT0 makes a high-to-low transition.

Solution

```
#include <REG51.H>    /* SFR declarations */

sbit outhit = P1^7;  /* use variahle outhit to refer to P1.7 */
unsigned char R7;     /* use 8-bit variahle to represent R7 */

main()
{
IT0 = 1;              /* negative edge activated */
TMOD = 0x11;          /* 16-bit timer mode */
IE = 0x81;            /* enable EXT 0 only */
while(1);             /* repeat forever */
}

void T0ISR(void) interrupt 0
{
R7  = 20;             /* 20 x 5000 μs = 1 second */
TF0 = 1;              /* force timer 0 interrupt */
TF1 = 1;              /* force timer 1 interrupt */
ET0 = 1;              /* hegin tone for 1 second */
ET1 = 1;              /* enahle timer interrupts */
                       /* timer interrupts will do the work */
}

void EX0ISR(void) interrupt 1
{
TR0 = 0;              /* stop timer */
R7 = R7 - 1;          /* decrement R7 */
if (R7 == 0)          /* if 20th time, */
     {
     ET0 = 0;         /* disahle itself */
     ET1 = 0;
     }
else
   {
   TH0 = 0x3C;        /* 0.05 sec. delay */
   TL0 = 0xB0;
   TR0 = 1;
   }
}

void T1ISR(void) interrupt 3
{
TR1 = 0;
TH1 = 0xFB;           /* count for 400 Hz
*/
TL1 = 0x1E;
outbit = !outbit;     /* music maestro! */
TR1 = 1;
}
```

Discussion

This is a combination of several interrupt functions previously discussed.

SUMMARY

This chapter has introduced the concepts of programming the 8051 in C, in contrast to previous chapters, which discussed programming the 8051 in assembly language. Nevertheless, whether it be assembly language or C, both make it possible to write programs, especially complex ones in an organized manner, the structured way. This will be the topic of the next chapter.

PROBLEMS

8.1 How does the 8051 C language differ from the conventional C language in terms of data types?

8.2 Suppose you wish to declare a variable to store a bit value at physical bit address 20H. Describe how you would do so in 8051 C.

8.3 What is the use of the carat (^) symbol in terms of bit addressable locations?

8.4 In how many ways can you declare shit variables? Explain each way by giving an example.

8.5 What is the difference between an **SFR** and an **unsigned char** variable?

8.6 Differentiate between the terms "memory types" and "memory models."

8.7 Explain the difference between a function and an interrupt function.

8.8 Explain, by giving an example, what you understand by a pointer.

8.9 Suppose you would like to have most of your program variables residing in external RAM while a few time-critical variables should reside in the first 128 bytes of internal RAM. Briefly describe how you could achieve this.

8.10 Why is it not advisable to use the accumulator, ACC, when programming in C?

8.11 Write a C program having numbers 1 to 10 in code memory, and that copies the odd (respectively even) numbers to external data memory when P1.0 is set (respectively cleared).

8.12 Write an 8051 C program to continually move a string of ASCII characters stored in internal data memory at location 30H to external data memory at location 1234H.

8.13 a. Show how you create a lookup table in code memory to store the first 10 values of the function f (x) = ex+2 corresponding to x = {1, 2, ... , 10}.

b. Next, declare a pointer that is stored in an indirectly accessible internal RAM location and use that to obtain the value of f(2) and print it out on the screen.

CHAPTER 9

Program Structure and Design

9.1 INTRODUCTION

What makes one program better than the next? Beyond simple views such as "it works," the answer to this question is complex and depends on many factors: maintenance requirements, computer language, quality of documentation, development time, program length, execution time, reliability, security, and so on. In this chapter we introduce the characteristics of good programs and some techniques for developing good programs. We begin with an introduction to structured programming techniques.

Structured programming is a technique for organizing and coding programs that reduces complexity, improves clarity, and facilitates debugging and modifying. The idea of properly structuring programs is emphasized in most programming tasks, and we advance it here as well. The power of this approach can be appreciated by considering the following statement: All programs may be written using only three structures. This seems too good to be true, but it's not. "Statements," "loops," and "choices" form a complete set of structures, and all programs can be realized using only these three structures. Program control is passed through the structures without unconditional branches to other structures. Each structure has one entry point and one exit point. Typically, a structured program contains a hierarchy of subroutines, each with a single entry point and a single exit point.[1]

The purpose of this chapter is to introduce structured programming as applied to assembly language programming and 8051 C programming. This is especially useful in the context of assembly language programming, since although high-level languages (such as Pascal or C) promote structured programming through their statements (WHILE, FOR, etc.) and notational conventions (indentation), assembly language lacks such inherent properties. Nevertheless, assembly language programming can benefit tremendously through

[1] In high-level languages, programs are composed of functions or procedures.

the use of structured techniques. In contrast, since 8051 C is an extension of C, it inherits all the structured techniques. Therefore, as we develop in this chapter structured techniques for the assembly language, we will present the structured properties of 8051 C as a comparison to its assembly language counterpart.

Progressing toward our goal—producing good assembly language programs—the example problems are solved using four methods:

- Flowcharts
- Pseudo code
- Assembly language
- 8051 C language

Solving programming problems in assembly language is, of course, our terminal objective; however, flowcharts and pseudo code are useful tools for the initial stages. Both of these are "visual" tools, facilitating the formulation and understanding of the problem. They allow a problem to be described in terms of "what must be done" rather than "how it is to be done." The solution can often be expressed in flowcharts or pseudo code in machine-independent terms, without considering the intricacies of the target machine's instruction set.

It is unusual for both pseudo code and flowcharts to be used. The preferred choice is largely a matter of personal style. The most common flowcharting symbols are shown in Figure 9-1.

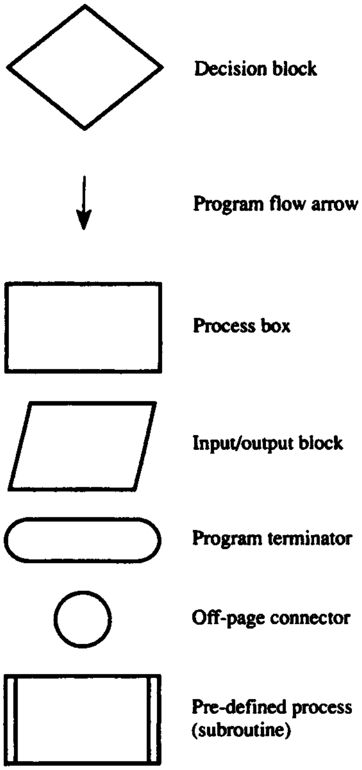


FIGURE 9-1
Common symbols for flowcharting

Pseudo code is just what the name suggests: "sort of a computer language. The idea has been used informally in the past as a convenient way to sketch out solutions to programming problems. As applied here, pseudo code mimics the syntax of Pascal or C in its notation of structure, yet at the same time it encourages the use of natural language in describing actions. Thus, statements such as

```
[get a character from the keyboard]
```

may appear in pseudo code. The benefit in using pseudo code lies in the strict adherence to structure in combination with informal language. Thus we may see

```
IF [condition is true]
     THEN [do statement 1]
     ELSE BEGIN
          [do statement 2]
          [do statement 3]
     END
```

or

```
IF [the temperature is less than 20 degrees Celsius]
     THEN [wear a jacket]
     ELSE BEGIN
          [wear a short sleeve shirt]
          [bring sunglasses]
     END
```

The use of keywords, indentation, and order is essential for the effective use of pseudo code. Our goal is to clearly demonstrate the solution to a programming problem using flowcharts and/or pseudo code, such that the translation to assembly language is easier than direct coding in assembly language. The final product is also easier to read, debug, and maintain. Pseudo code will be defined formally in a later section.

9.2 ADVANTAGES AND DISADVANTAGES OF STRUCTURED PROGRAMMING

The advantages of adopting a structured approach to programming are numerous. These include the following:

- The sequence of operations is simple to trace, thus facilitating debugging.
- There are a finite number of structures with standardized terminology.
- Structures lend themselves easily to building subroutines.
- The set of structures is complete; that is, all programs can be written using three structures.
- Structures are self-documenting and, therefore, easy to read.
- Structures are easy to describe in flowcharts, syntax diagrams, pseudo code, and so on.
- Structured programming results in increased programmer productivity—programs can be written faster.

However, some tradeoffs occur. Structured programming has disadvantages, such as the following:

- Only a few high-level languages (Pascal, C, PL/M) accept the structures directly; others require an extra stage of translation.
- Structured programs may execute more slowly and require more memory than the unstructured equivalent.
- Some problems (a minority) are more difficult to solve using only the three structures rather than a brute-force "spaghetti" approach.
- Nested structures can be difficult to follow.

9.3 THE THREE STRUCTURES

All programming problems can be solved using three structures:

- Statements
- Loops
- Choice

The completeness of the three structures seems unlikely; but, with the addition of nesting (structures within structures), it is easily demonstrated that any programming problem can be solved using only three structures. Let's examine each in detail.

9.3.1 Statements

Statements provide the basic mechanism to do something. Possibilities include the simple assigning of a value to a variable, such as

```
[count = 0]
```

or a call to a subroutine, such as

```
PRINT STRING("Select Option:")
```

Anywhere a single statement can be used, a group of statements, or a **statement block,** can be used. This is accomplished in pseudo code by enclosing the statements between the keywords BEGIN and END as follows:

```
BEGIN
[statement 1]
[statement 2]
[statement 3]
END
```

Note that the statements within the statement block are indented from the BEGIN and END keywords. This is an important feature of structured programming.

9.3.2 The Loop Structure

The second of the basic structures is the "loop," used to repeatedly perform an operation. Adding a series of numbers or searching a list for a value are two examples of programming

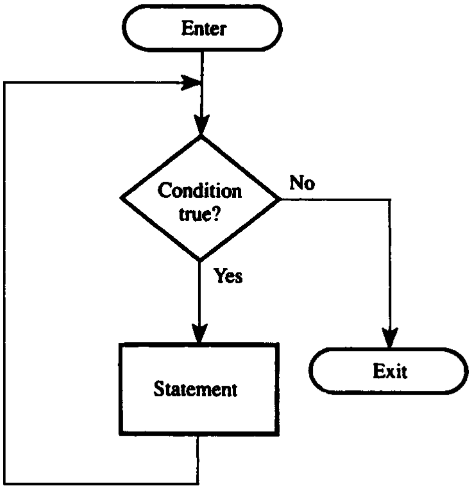


FIGURE 9-2
Flowchart for the WHILE/DO structure

problems that require loops. The term "iteration" is also used in this context. Although there are several possible forms of loops, only two are necessary, WHILE/DO and REPEAT/UNTIL.

9.3.2.1 The WHILE/DO Statement The WHILE/DO statement provides the easiest means for implementing loops. It is called a "statement," since it is treated like a statement—as a single unit with a single entry point (the beginning) and a single exit point (the ending). The pseudo code format is Pseudo Code:

```
WHILE [condition] DO
      [statement]
```

The "condition" is a "relational expression" that evaluates to "true" or "false." If the condition is true, the statement (or statement block) following the "DO" is executed, and then the condition is reevaluated. This is repeated until the relational expression yields a false response; this causes "statement" to be skipped, with program execution continuing at the next statement. The WHILE/DO structure is shown in the flowchart in Figure 9-2.

EXAMPLE 9.1 **WHILE/DO Structure**

Illustrate a WHILE/DO structure such that while the 8051 carry flag is set, a statement is executed.

Solution
Pseudo Code:

```
WHILE[c == 1]DO
     [statement
     ]
```

(*Note:* The double equal sign is used to distinguish the relational operator, which tests for equality, from the assignment operator, the single equal sign [see 9.4 Pseudo Code Syntax].)

Flowchart:

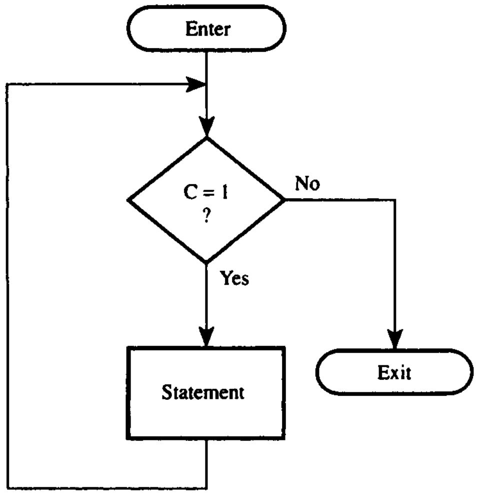


FIGURE 9-3
Flowchart for Example 9.1

Assembly Language:

```
ENTER:      JNC EXIT
STATEMENT:  (statement)
             JMP ENTER
EXIT:       (continue)
```

8051 C:

```
while (c == 1)
       {statement;}
```

As a general rule, the actions within the statement block must affect at least one variable in the relational expression; otherwise a bug in the form of an infinite loop results.

EXAMPLE 9.2 **SUM Subroutine**

Write an 8051 subroutine called SUM to calculate the sum of a series of numbers. Parameters passed to the subroutine include the length of the series in R7 and the starting address of the series in R0. (Assume the series is in 8051 internal memory.) Return with the sum in the accumulator.

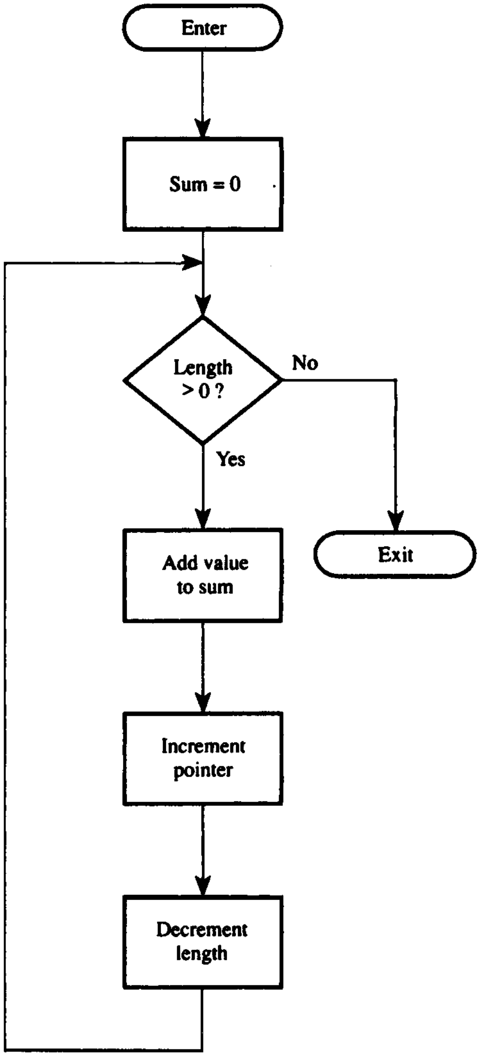
Solution

Pseudo Code:

```
[sum = 0]
WHILE [length > 0] DO BEGIN
                [sum = sum + @pointer)
                [increment pointer]
                [decrement length]
END
```

Flowchart:

FIGURE 9-4
Flowchart for Example 9.2



Assembly language: (closely structured; 13 bytes)

```
SUM:         CLR    A
LOOP:        CJNE   R7,#0,STATEMENT
             JMP    EXIT
STATEMENT:   ADD    A,@R0
             INC    R0
             DEC    R7
             JMP    LOOP
EXIT:        RET
```

(loosely structured; 9 bytes)

```
SUM:         CLR    A
             INC    R7
MORE:        DJNZ   R7,SKIP
             RET
SKIP:        ADD    A,@R0
             INC    R0
             SJMP   MORE
```

Notice above that the loosely structured solution is shorter (and faster) than the closely structured solution. Experienced programmers will, no doubt, code simple examples such as this intuitively and follow a loose structure. Novice programmers, however, can benefit by solving problems clearly in pseudo code first and then progress to assembly language while following the pseudo code structure.

8051 C:

```
void sum(int * start, int length)
{
int result = 0, i = 0;
while (length > 0)
      {
      result = result + start[i];
      i++;
      length--;
      }
}
```

EXAMPLE 9.3 **WHILE/DO Structure**

Illustrate a WHILE/DO structure using the following compound condition: the accumulator not equal to carriage return (0DH) and R7 not equal to 0.

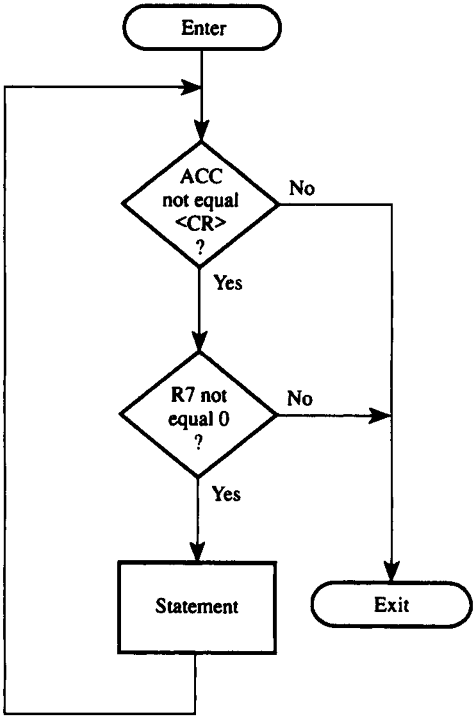
Solution

Pseudo Code:

```
WHILE [ACC!=CR AND R7!=0] DO
      [statement]
```

Flowchart:

FIGURE 9-5
Flowchart for Example 9.3



Assembly Language:

```
ENTER:        CJNE A,#0DH,SKIP
              JMP  EXIT
SKIP:         CJNE R7,#0,STATEMENT
              JMP  EXIT
STATEMENT: one or more statements)
                  .
                  .
                  .
              JMP ENTER
EXIT:         (continue)
```

8051 C:

```
while( (ACC != C) && (R7 != 0) )
      statement;)
```

9.3.2.2 The REPEAT/UNTIL Statement Similar to the WHILE/DO statement is the REPEAT/UNTIL statement, which is useful when the "repeat statement" must be performed at least once. WHILE/DO statements test the condition first; thus, the statement might not execute at all.
Pseudo Code:

```
REPEAT [statement]
UNTIL [condition]
```

Flowchart:

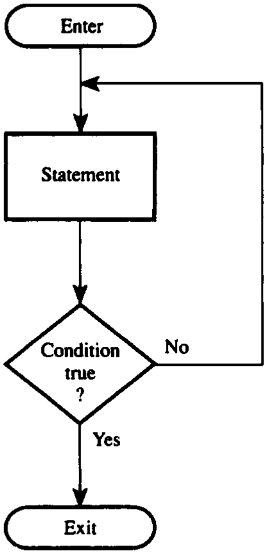


FIGURE 9-6
Flowchart for the REPEAT/UNTIL structure

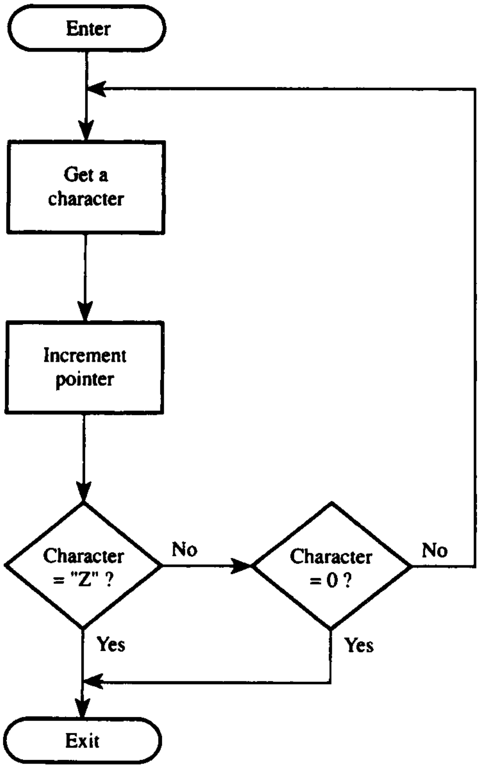
EXAMPLE 9.4 **Search Subroutine**
Write an 8051 subroutine to search a null-terminated string pointed at by R0 and determine if the letter "Z" is in the string. Return with ACC = Z if it is in the string, or ACC = 0 otherwise.

Solution
Pseudo Code:

```
REPEAT
     [ACC = @pointer]
     [increment pointer]
UNTIL [ACC == 'Z' or ACC == 0]
```

Flowchart:

FIGURE 9-7
Flowchart for Example 9.4



Assembly Language:

```
STATEMENT: MOV   A,@R0
           INC   R0
           JZ    EXIT
           CJNE  A,#'Z',STATEMENT
EXIT:      RET
```

8051 C:

```
char statement(char * start)
{
char a;
do
     {
     a = *start;
     start++;
     }
```

```
        while ( (a != 'z') || (a != 0)  );
        return a;
        }
```

9.3.3 The Choice Structure

The third basic structure is that of "choice"—the programmer's "fork in the road." The two most common arrangements are the IF/THEN/ELSE statement and the CASE statement.

9.3.3.1 The IF/THEN/ELSE Statement The IF/THEN/ELSE statement is used when one of two statements (or statement blocks) must be chosen, depending on a condition. The ELSE part of the statement is optional.
Pseudo Code:

```
IF [condition]
        THEN[statement 1]
        ELSE[statement 2]
```

Flowchart:

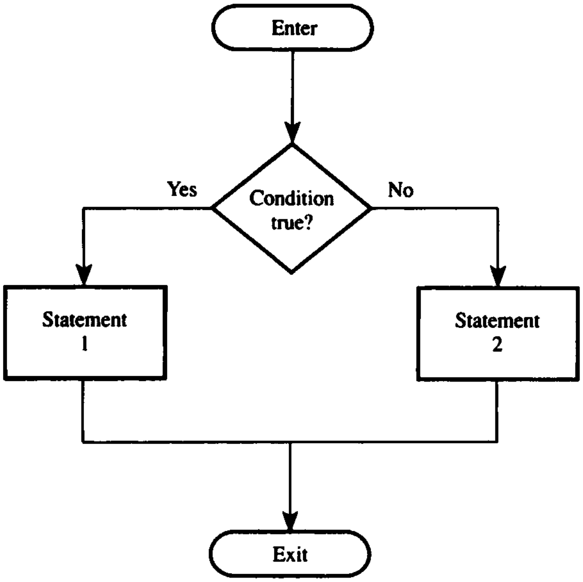


FIGURE 9-8
Flowchart for the IF/THEN/ELSE structure

EXAMPLE 9.5

Character Test
Write a sequence of instructions to input and test a character from the serial port. If the character is displayable ASCII (i.e., in the range 20H to 7EH), echo it as is; otherwise, echo a period (.).

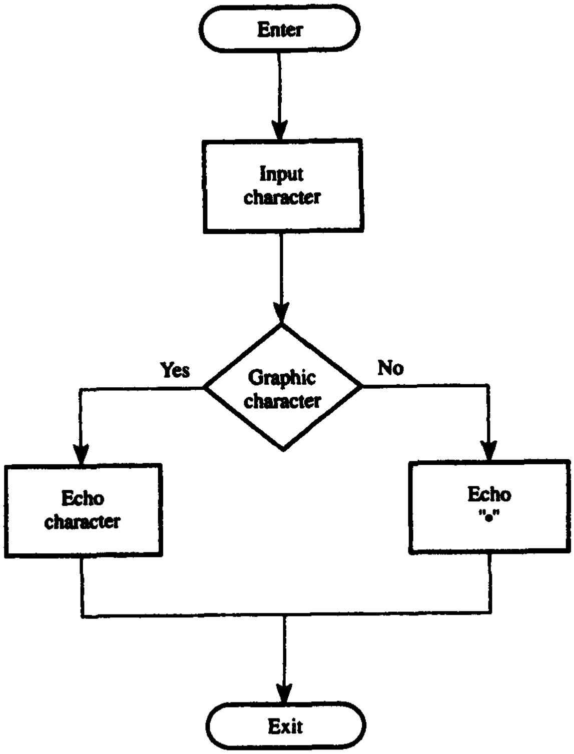
Solution

Pseudo Code:

```
[input character]
IF [character == graphic]
          THEN [echo character]
          ELSE [echo '.']
```

Flowchart:

FIGURE 9-9
Flowchart for Example 9.5



Assembly Language: (closely structured; 14 bytes)

```
ENTER:        ACALL  INCH
              ACALL  ISGRPH
              JNC    STMENT2
STMENT1:      ACALL  OUTCHR
              JMP    EXIT
STMENT2:      MOV    A,#'.'
              ACALL  BUTCHER
EXIT:         (continue)
```

(loosely structured; 10 bytes)

```
               ACALL  INCH
               ACALL  ISGRPH
               JC     SKIP
               MOV    A,#'.'
SKIP:          ACALL  OUTCHR
               (continue)
```

8051 C:

```
while (1)
{
      char a;
      a = inchar();
      if (isgrph(a))
         outchr(a);
      else
         outchar(`.');
}
```

Modify the structure to repeat indefinitely:

```
WHILE [1] DO BEGIN
            [input character]
            IF [character == graphic]
                 THEN [echo character]
                 ELSE [echo '.']
END
```

9.3.3.2 The CASE Statement The CASE statement is a handy variation of the IF/THEN/ELSE statement. It is used when one statement from many must be chosen as determined by a value.

Pseudo Code:

```
CASE [expression] OF
        0:[statement 0]
        1:[statement 1]
        2:[statement 2]
        .
        .
        .
        n:[statement n]
        [default statement]
END_CASE
```

Flowchart:

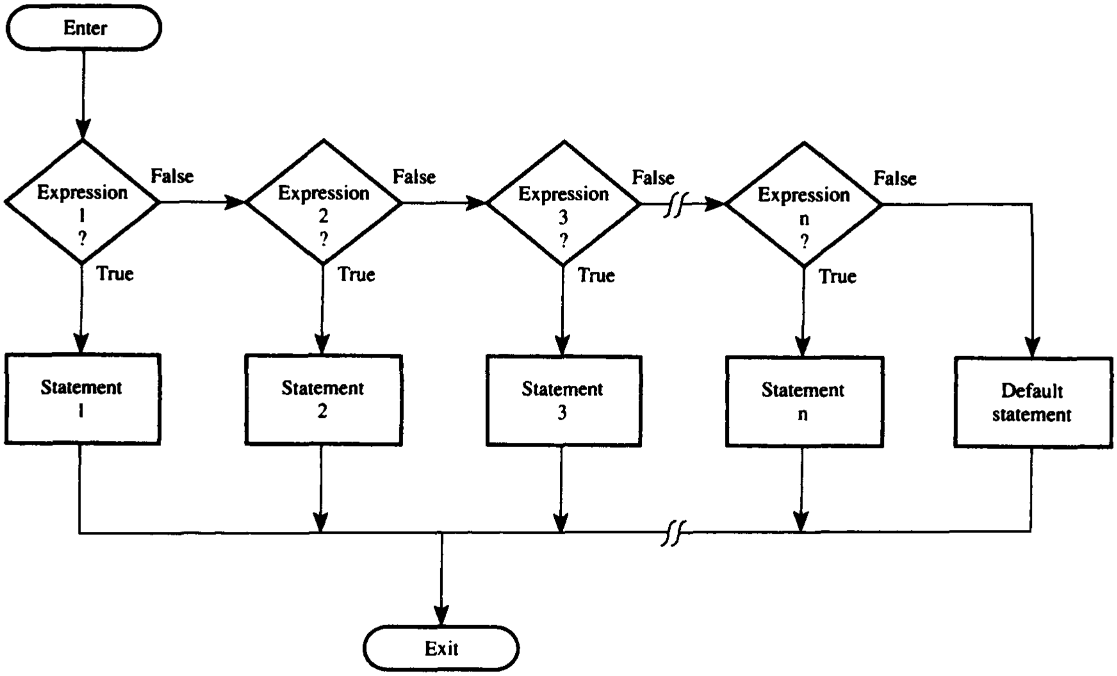


FIGURE 9-10
Flowchart for CASE structure

EXAMPLE 9.6 User Response

A menu-driven program requires a user response of 0, 1, 2, or 3 to select one of four possible actions. Write a sequence of instructions to input a character from the keyboard and jump to ACT0, ACT, ACT2, or ACT3, depending on the user response. Omit error checking.

Solution

Pseudo Code:

```
[input a character]
CASE [character] OF
      '0': [statement 0]
      '1': [statement 1]
      '2': [statement 2]
      `3': [statement 3]
END_CASE
```

Flowchart:

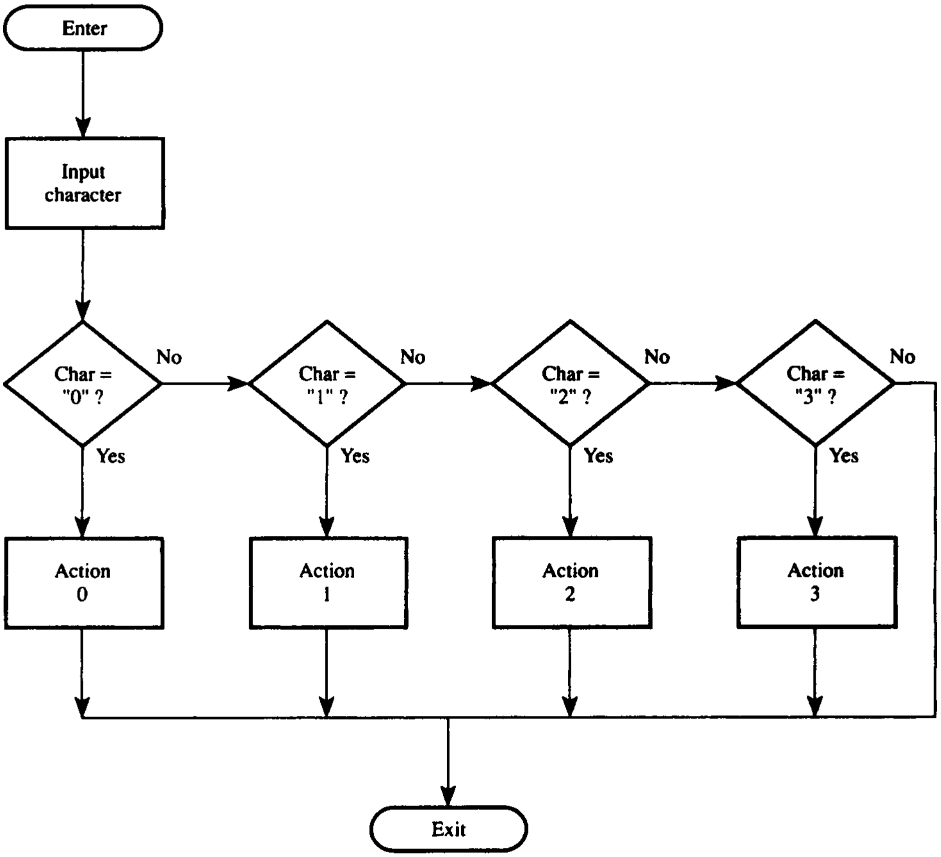


FIGURE 9-11
Flowchart for Example 9.6

Assembly Language: (closely structured)

```
        CALL INCH
        CJNE A,#'0',SKIP1
ACT0:   .
        .
        .
        JMP EXIT
SKIP:   CJNE A,#'1',SKIP2
ACT:    .
        .
        .
        JMP EXIT
```

```
SKIP2:  CJNE A,#'2',SKIP3
ACT2:   .
        .
        .
SKIPS:  CJNE A,#'3',EXIT
ACT3:   .
        .
        .
EXIT:   (continue)
```

(loosely structured)

```
       CALL INCH              ;REDUCE to 2 bit
       ANL  A,#3              ;WORD OFFSET
       RL   A
       MOV  DPTR,#TABLE
       JMP  @A+DPTR
TABLE: AJMP ACT0
       AJMP ACT1
       AJMP ACT2
ACT3:  .
       .
       .
       JMP EXIT
ACT0:  .
       .
       .
       JMP EXIT
ACT1:  .
       .
       .
       JMP EXIT
ACT2:  .
       .
       .
       EXIT:  (continue)
```

8051 C:

```
char a;
a =
inchar();
switch(a)

     case '0':
            statement 0;
            break;

     case '1':
            statement 1;
            break;
```

```
case '2':
      statement 2;
      break;

case '3':
      statement 3;
}
```

9.3.3.3 The GOTO Statement GOTO statements can always be avoided by using the structures just presented. Some times a GOTO statement provides an easy method of terminating a structure when errors occur; however, exercise extreme caution. When the program is coded in assembly language, GOTO statements usually become unconditional jump instructions. A problem will arise, for example, if a subroutine is entered in the usual way (a CALL subroutine instruction), but is exited using a jump instruction rather than a return from subroutine instruction. The return address will be left on the stack, and eventually a stack overflow will occur.

9.4 PSEUDO CODE SYNTAX

Since pseudo code is similar to a high-level language such as Pascal or C, it is worthwhile defining it somewhat more formally, so that, for example, a pseudo code program can be written by one programmer and converted to assembly language by another programmer.

We should acknowledge too that pseudo code is not always the best approach for designing programs. While it offers the advantage of easy construction on a word processor (with subsequent modifications), it suffers from a disadvantage common to other programming languages: pseudo code programs are written line-by-line, so parallel operations are not immediately obvious. With flowcharts, on the other hand, parallel operations can be placed physically adjacent to one another, thus improving the conceptual model (see Figure 9-10).

Before presenting a formal syntax, the following tips are offered to enhance the power of solving programming problems, using pseudo code.

- Use descriptive language for statements.
- Avoid machine dependencies in statements.
- Enclose conditions and statements in brackets: [ ].
- Begin all subroutines with their names followed by a set of parentheses: 0. Parameters passed to subroutines are entered (by name or by value) within the parentheses.
- End all subroutines with RETURN followed by parentheses. Return values are entered within the parentheses.

Examples of subroutines:

```
INCHAR ()        OUTCHR(char)      STRLEN(pointer)
  [statement]      [statement]        [statement]
  . . .            . . .              .  .  .
RETURN (char)    RETURN ( )        RETURN(length)
```

- Use lowercase text except for reserved words and subroutine names.
- Indent all statements from the structure entry and exit points. When a LOOP or CHOOSE structure is started, the statements within the structure appear at the next level of indentation.
- Use the commercial at sign (@) for indirect addressing.

The following is a suggested syntax for pseudo code.

Reserved Words:

```
BEGIN END
REPEAT UNTIL
WHILE DO
IF      THEN           ELSE
CASE OF
RETURN
```

Arithmetic Operators:

```
+ addition
- subtraction
* multiplication
/ division
% modulus (remainder after division)
```

Relational Operators: (result is true or false)

```
== true if values equal to each other
!= true if values not equal
<  true  if first value less than second
<= true if first value <= second value
>  true if first value > second value
>= true if first value >= second value
&& true if both values are true
|| true if either value is true
```

Bitwise Logical Operators:

```
&  logical AND
|  logical OR
^  logical exclusive OR
~  logical NOT (one's complement)
>> logical shift right
<< logical shift left
```

Assignment Operators:

```
=     set equal to
op =  assignment operation shorthand where "op" is one
        of + - * / << >> & ^ |
        e.g.: j += 4 is equivalent to j = j + 4
```

Precedence Operation:

```
( )
```

Indirect Address:

```
@
```

Operator Precedence:

```
( )
~     @
*     /     %
+     -
<<    >>
<      <=    >      >=
==     !=
&
|
&&
||
=     +=    -=     *= etc.
```

Note 1. Do not confuse relational operators with bitwise logical operators. Bitwise logical operators are generally used in assignment statements such as

```
[lower_nibble = byte & 0FH]
```

whereas relational operators are generally used in conditional expressions such as

```
IF [char 1= 'Q' && char !=0DH] THEN  ...
```

Note 2. Do not confuse the relational operator "= =" with the assignment operator "=." For example, the Boolean expression

```
j = = 9
```

is either true or false depending on whether or not j is equal to the value 9, whereas the assignment statement

```
j = 9
```

assigns the value 9 to the variable j.

Structures:

Statement:

```
[do something]
```

Statement Block:

```
BEGIN
        [statement]
```

```
            [statement]
            . . .
     END
```

WHILE/DO:

```
     WHILE [condition] DO
            [statement]
```

REPEAT/UNTIL:

```
     REPEAT
            [statement]
     UNTIL [condition]
```

IF/THEN/ELSE:

```
     IF [condition]
          THEN [statement 1]
          ELSE [statement 2])
```

CASE/OF:

```
     CASE [expression] OF
          1:[statement1]
          2:[statement2]
          3:[statement3]
              .
              .
              .
          n:[statement n]
        [default statement]
        END_CASE
```

9.5 ASSEMBLY LANGUAGE PROGRAMMING STYLE

It is important to adopt a clear and consistent style in assembly language programming. This is particularly important when one is working as part of a team, since individuals must be able to read and understand each other's programs.

The assembly language solutions to problems up to this point have been deliberately sketchy. For larger programming tasks, however, a more critical approach is required. The following tips are offered to help improve assembly language programming style.

9.5.1 Labels

Use labels that are descriptive of the destination they represent. For example, when branching back to repeatedly perform an operation, use a label such as "LOOP," "BACK," "MORE," etc. When skipping over a few instructions in the program, use a label such as "SKIP" or "AHEAD." When repeatedly checking a status bit, use a label such as "WAIT" or "AGAIN."

The choice of labels is restricted somewhat when one is using a simple memory-resident or absolute assembler. These assemblers treat the entire program as a unit, thus limiting the use of common labels. Several techniques circumvent this problem. Common labels can be sequentially numbered, such as SKIP, SKIP, SKIP3 , etc.; or perhaps within subroutines all labels can use the name of the subroutine followed by a number, such as SEND, SEND2, SEND3, etc. There is an obvious loss of clarity here, since the labels SEND2 and SEND3 are not likely to reflect the skipping or looping actions taking place.

More sophisticated assemblers, such as ASM51, allow each subroutine (or a common group of subroutines) to exist as a separate file that is assembled independent of the main program. The main program is also assembled on its own and then combined with the subroutines using a linking and locating program that, among other things, resolves external references between the files. This type of assembler, usually called a "relocatable" assembler, allows the same label to appear in different files.

9.5.2 Comments

The use of comments cannot be overemphasized, particularly in assembly language programming, which is inherently abstract. All lines of code, except those with truly obvious actions, should include a comment.

Conditional jump instructions are effectively commented using a question similar to the flowchart question for a similar operation. The "yes" and "no" answers to the question should appear in comments at the lines representing the "jump" and "no jump" actions. For example, in the INLINE subroutine below, note the style of comments used to test for the carriage return <CR>code.

```
; *************************************************************
;
;           INPUT LINE OF CHARACTERS
;           INLINELINE MUST END WITH <CR>
;           MAXIMUM LENGTH 31 CHARACTERS INCLUDE <CR>
;ENTER:     NO CONDITIONS
;EXIT:      ASCII CODES IN INTERNAL DATA RAM
;           0 STORED AT END OF LINE
;USES       INCHAR, OUTCHR
;
; *************************************************************
;
INLINE:     PUSH 00H                ;SAVE R0 ON STACK
            PUSH 07H                ;SAVE R7 ON STACK
            PUSH ACC                ;SAVE ACCUMULATOR ON STACK
            MOV R0,#60H             ;SET UP BUFFER AT 60H
            MOV R7,#31              ;MAXIMUM LENGTH OF LINE
STMENT:     ACALL INCHAR            ;INPUT A CHARACTER
            ACALL OUTCHR            ;ECHO TO CONSOLE
```

```
             MOV  @R0,A           ;STORE IN BUFFER
             INC  RE              ;INCREMENT BUFFER POINTER
             DEC  R7              ;DECREMENT LENGTH COUNTER
             CJNE A,#0DH, SKIP    ;IS CHARACTER = <CR>?
             SJMP EXIT            ;YES: EXIT
SKIP:        CJNE R7,#0,STMENT    ;NO: GET ANOTHER CHARACTER
EXIT:        MOV  @R0,#0
             POP  ACC             ;RETRIEVE REGISTERS FROM
                                  ;STACK
             POP  07H
             POP  00H
             RET
```

9.5.3 Comment Blocks

Comment lines are essential at the beginning of each subroutine. Since subroutines perform well-defined tasks commonly needed throughout a program, they should be general-purpose and well documented. Each subroutine is preceded by a **comment block,** a series of comment lines that explicitly state

- The name of the subroutine
- The operation performed
- Entry conditions
- Exit conditions
- Name of other subroutines used (if any)
- Name of registers affected by the subroutine (if any)

The INLINE subroutine above is a good example of a well-commented subroutine.

9.5.4 Saving Registers on the Stack

As applications grow in size and complexity, new subroutines are often written that build upon and use existing subroutines. Thus, subroutines are calling other subroutines, which in turn call other subroutines, and so on. These are called "nested subroutines." There is no danger in nesting subroutines so long as the stack has enough room to hold the return addresses. This is not a problem, since nesting beyond several levels is rare.

A potential problem, however, lies in the use of registers within subroutines. As the hierarchy of subroutines grows, it becomes more and more difficult to keep track of what registers are affected by subroutines. A solid programming practice, then, is to save registers on the stack that are altered by a subroutine, and then restore them at the end of the subroutine. Note that the INLINE subroutine shown above saves and retrieves R0, R7, and the accumulator using the stack. When INLINE returns to the calling subroutine, these registers contain the same value as when INLINE was called.

9.5.5 The Use of Equates

Defining constants with equate statements makes programs easier to read and maintain. Equates appear at the beginning of a program to define constants such as carriage return

(<CR>) and line feed (<LF>), or addresses of registers inside peripheral ICs such as STATUS or CONTROL.

The constant can be used throughout the program by substituting the equated symbol for the value. When the program is assembled, the value is substituted for the symbol. A generous use of equates makes a program more maintainable, as well as more readable. If a constant must be changed, only one line needs changing—the line where the symbol is equated. When the program is reassembled, the new value is automatically substituted wherever the symbol is used.

9.5.6 The Use of Subroutines

As programs grow in size, it is important to "divide and conquer"; that is, subdivide large and complex operations into small and simple operations. These small and simple operations are programmed as subroutines. Subroutines are hierarchical in that simple subroutines can be used by more complex subroutines, and so on.

A flowchart references a subroutine using the "predefined process" box. (See Figure 9-1.) The use of this symbol indicates that another flowchart elsewhere describes the details of the operation.

Subroutines are constructed in pseudo code as complete sections of code, beginning with their names and parentheses. Within the parentheses are the names or values of parameters passed to the subroutine (if any). Each subroutine ends with the keyword RETURN followed by parentheses containing the name or value of parameters returned by the subroutine (if any).

Perhaps the simplest example of subroutine hierarchy is the output stung (OUTSTR) and output character (OUTCHR) subroutines. The OUTSTR subroutine (a high-level routine) calls the OUTCHR subroutine (a low-level routine).

The flowcharts, pseudo code, and 8051 assembly and C language solutions are shown below.

Pseudo Code:

```
OUTCHR (char)
     [put odd parity in bit 7]
     REPEAT [test transmit buffer]
     UNTIL [buffer empty]
      [clear transmit buffer empty flag]
     [move char to transmit buffer]
     [clear parity bit]
RETURN ()
OUTSTR (pointer)
      WHILE [(char = @pointer) !=0] BEGIN
           OUTCHR (char)
           [increment pointer]
     END
RETURN()
```

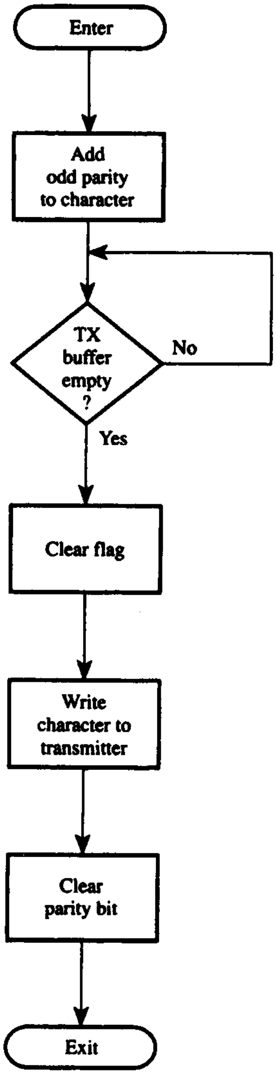
Assembly language:

```
OUTCHR:   MOV C,P           ;PUT PARITY BIT IN C FLAG
          CPL C             ;CHANGE TO ODD PARITY
          MOV ACC.7,C       ;ADD TO CHARACTER
AGAIN:    JNB TI,AGAIN      ;TX EMPTY?
```

```
          CLR  TI           ;YES: CLEAR FLAG AND
          MOV  SBUF,A       ; SEND CHARACTER
          CLR  ACC.7        ;STRIP OFF PARITY BIT AND
          RET               ; RETURN
OUTCHR:   MOV  A,@DPTR      ;GET CHARACTER
          JZ   EXIT         ;IF 0, DONE
          CALL OUTSTR       ;OTHERWISE SEND IT
          INC  DPTR         ;INCREMENT POINTER
          SJMP OUTCHR       ; AND GET NEXT CHARACTER
EXIT:     RET
```

Flowchart:

FIGURE 9-12
Flowchart for OUTCHR subroutine



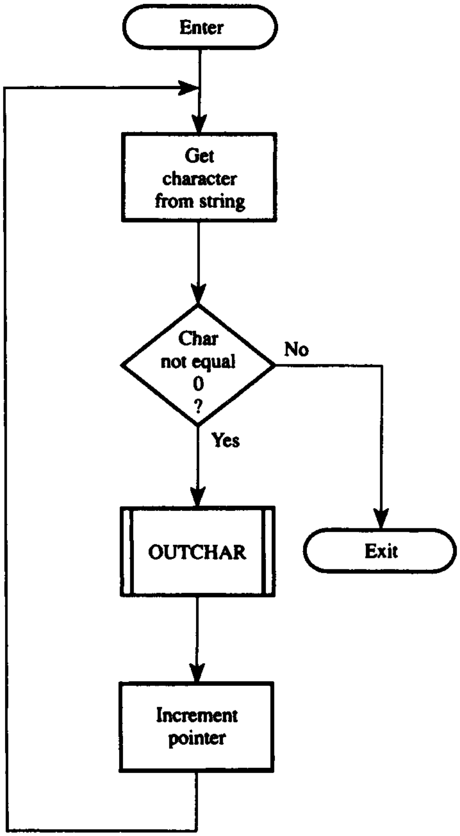


FIGURE 9-13
Flowchart for OUTSTR subroutine

8051C

```
sbit AccMSB = ACC^7;
void outchr (char a)
{
ACC = a;
CY  = P;
CY  = !CY;
AccMSB = CY;
while (TI != 1);
TI = 0;
SBUF = ACC;
AccMSB = 0;
}
void outstr(char * msg)
{
while (*msg !='\0')
    outchr(*p++);
}
```

9.5.7 Program Organization

Although programs are often written piecemeal (i.e., subroutines are written separately from the main program), all programs should be consistent in their final organization. In general, the sections of a program are ordered as follows:

- Equates
- Initialization instructions
- Main body of program
- Subroutines
- Data constant definitions (DB and DW)
- RAM data locations defined using the DS directive

All but the last item above are called the "code segment" and the RAM data locations are called the "data segment." Code and data segments are traditionally separate, since code is often destined for ROM or EPROM, whereas RAM data are always destined for RAM. Note that data constants and strings defined using the DB or DW directives are part of the code segment (not the data segment), since these data are unchanging constants and, therefore, are part of the program.

9.6 8051 C PROGRAMMING STYLE

When programming in 8051 C, it is even more necessary to be clear and consistent. This is because programmers usually write in C when their 8051 programs are large and complex, and involve several programmers working on it at the same time. The following should be noted when programming the 8051 in C.

9.6.1 Comments

In C, comments are enclosed in between the /* and */ symbols. For example, the commented "Hello World" program is given below:

```
main ()
{
SCON = 0x52;       /* perial port, mode 1 */
TMOD = 0x20;       /* timer 1, mode 2 */
TH1  = -13;        /* reload count for 2400 baud */
TR1  = 1;          /* start timer 1 */
while (1)          /* repeat forever */
      {
      printf ("Hello World\n"); /* Dipplay "Hello World" */
      }
}
```

9.6.2 The Use of Defines

You should define constants with aliases so that your program is more readable and it is easier for you to change the constant in future by simply changing one line of code. For example:

```
#define PI 3.1415927
#define MAX 10

int circleArea(int radiup)
{
return (PI * radius * radius);
}

main()
{
int i;
for (i = 0; i < MAX; i++)
    circleArea(i);
}
```

The example program above defines **PI** as the constant $\pi = 3.1415927$ while **MAX** is defined as the constant number 10. **PI** is used to calculate the area of a circle while **MAX** is the maximum number of different circle areas that are being calculated. If, for example, one desires to calculate areas of 20 different circles, then the value of **MAX** can be changed by simply changing the definition line to **#define MAX 20**.

9.6.3 The Use of Functions

C programs, being large and complex, are normally written by several programmers simultaneously. This is possible by using the modular programming approach, which breaks parts of the program into modules and functions. Dividing the program into functions allows for more modularity and makes the overall program clearer and easier to follow. This approach has been adopted in the interface and design example programs discussed in Chapter 12.

9.6.4 The Use of Arrays and Pointers

When it is desired to store a sequence of some related data, it is good to use arrays and have them pointed to by pointers. Recall that pointers store addresses of memory locations. Hence, the address of the first element of an array could be assigned to a pointer. Then each array element could be accessed by adding a suitable offset to the pointer value and dereferencing it. As an example, lookup tables can be optimally implemented by using arrays.

Arrays are also useful when storing strings. For example, the message "This is a welcome message" could be stored in memory as an array by simply using the line: **char * MSG = {"Thip is a welcome message." }** which would use an array of characters to store the string.

9.6.5 Program Organization

To standardize the layout and organization, we have adopted the following program organization when writing 8051 C programs:

- Includes
- Constant Definitions
- Variable Definitions
- Function Prototypes
- Main Function
- Function Definitions

Constant definitions are lines such as **`#define PI 3.1415927`** while variable definitions are those such as **`char A`**. Function prototypes are one-line statements such as **`void HTOA (void)`** that consist of the function name, return type, and parameter list. Meanwhile, function definitions are the full definitions of operations done in the functions, for example:

```
void HTOA(void) {
A = A & 0xF; if
(A>=0xA)
     A = A + 7;
A = A + '0';
}
```

SUMMARY

This chapter has introduced structured programming techniques through flowcharting and pseudo code. Suggestions were offered on designing and presenting programs to enhance their readability. In the next chapter, some of the tools and techniques for developing programs are presented.

PROBLEMS

9.1 Illustrate a WHILE/DO structure such that while the accumulator is less than or equal to 7EH, a statement block is executed.

9.2 Illustrate a WHILE/DO structure using the following compound condition: (accumulator greater than zero and R7 greater than zero) or the carry flag equal to 1. Treat the values in the accumulator and R7 as unsigned integers.

9.3 Write a subroutine to find the place of the most significant 1 in the accumulator. Return with "place" in R7. For example, if ACC = 00010000B, return with R7 = 4.

9.4 Write a subroutine called INLINE to input a line of characters from the console and place them in internal memory, starting at location 60H. Maximum line length is 31 characters, including the carriage return. Put 0 at the end of the line.

CHAPTER 10

Tools and Techniques for Program Development

10.1 INTRODUCTION

In this chapter, the process of developing microcontroller- or microprocessor-based products is described as it follows a series of steps and utilizes a variety of tools. In progressing from concept to product, numerous steps are involved and numerous tools are used. The most common steps and tools are presented as found in typical design scenarios employing the 8051 microcontroller.

Design is a highly creative activity, and in recognition of this we state at the outset that substantial leeway is required for individuals or development teams. Such autonomy may be difficult to achieve for very large or safety-critical projects, however. Admittedly, in such environments the management of the process and the validation of the results must satisfy a higher order. The present chapter addresses the development of relatively small-scale products, such as controllers for microwave ovens, automobile dashboards, computer peripherals, electronic typewriters, or high-fidelity audio equipment.

The steps required and the tools and techniques available are presented and elaborated on, and examples are given. Developing an understanding of the steps is important, but strict adherence to their sequence is not advocated. It is felt that forcing the development process along ordered, isolated activities is usually overstressed and probably wrong. Later in the chapter we will present an all-in-one development scenario, where the available resources are known and called upon following the instinct of the designer. We begin by examining the steps in the development cycle.

10.2 THE DEVELOPMENT CYCLE

Proceeding from concept to product is usually shown in a flow diagram known as the **development cycle,** similar to that shown in Figure 10-1. The reader may notice that

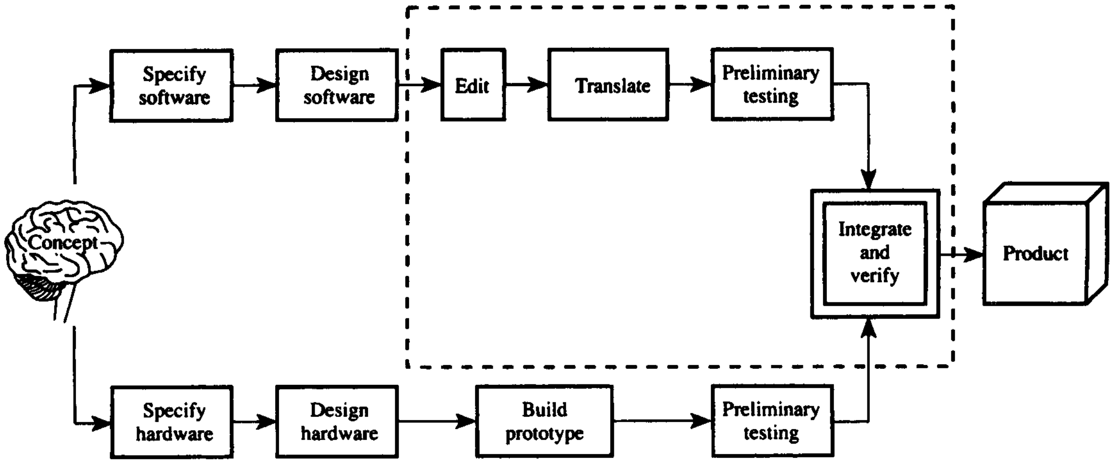


FIGURE 10-1
The development cycle

there is nothing particularly "cyclic" about the steps shown. Indeed, the figure shows the ideal and impossible scenario of "no breakdowns." Of course, problems arise. **Debugging** (finding and fixing problems) is needed at every step in the development cycle with corrections introduced by reengaging in an earlier activity. Depending on the severity of the error, the correction may be trivial or, in the extreme, may return the designer to the concept stage. Thus, there is an implied connection in Figure 10-1 from the output of any step in the development cycle to any earlier step.

The steps along the top path in Figure 10-1 correspond to software development, while those along the bottom correspond to hardware development. The two paths meet at a critical and complicated step called "integration and verification," which leads to acceptance of the design as a "product." Not shown are various steps subsequent to acceptance of the design. These include, for example, manufacturing, testing, distribution, and marketing. The dotted line in Figure 10-1 encompasses the steps of primary concern in this chapter (and book). These will be elaborated in more detail later. But first, we begin by examining the steps in software development.

10.2.1 Software Development

The steps in the top path in Figure 10-1 are discussed in this section, beginning with the specification of the application software.

Specifying Software. Specifying software is the task of explicitly stating what the software will do. This may be approached in several ways. At a superficial level, specifications may first address the user interface; that is, how the user will interact with and control the system. (What effects will result from and be observed for each action taken?) If switches, dials, or audio or visual indicators are employed on the prototype hardware, the explicit purpose and operation of each should be stated.

Formal methods have been devised by computer scientists for specifying software requirements; however, they are not generally used in the design of microcontroller-based applications, which are small in comparison to application software destined for mainframe computers.

Software specifications may also address details of system operation below the user level. For example, a controller for a photocopier may monitor internal conditions necessary for normal or safe operation, such as temperature, current, voltage, or paper movement. These conditions are largely independent of the user interface but still must be accommodated by software.

Specifications can be modularized by system function with entry and exit conditions defined to allow intermodule communication. The techniques described in the previous chapter for documenting subroutines are a reasonable first step in specifying software.

Interrupt-driven systems require careful planning and have unique characteristics that must be addressed at the specification stage. Activities without time-critical requirements may be placed in the foreground loop or in a round-robin sequence for handling by timed interrupts. Time-critical activities generate high-priority interrupts that take over the system for immediate handling. Software specifications may emphasize execution time on such systems. How long does each subroutine or interrupt service routine (ISR) take to execute? How often is each ISR executed? ISRs that execute asynchronously (in response to an event) may take over the system at any time. It may be necessary to block them in some instances or to preempt (interrupt) them in others. Software specifications for such systems must address priority levels, polling sequences, and the possibility of dynamically reassigning priority levels or polling sequences within ISRs.

Designing Software. Designing the software is a task designers are likely to jump into without a lot of planning. There are two common techniques for designing software prior to coding: flowcharts and pseudo code. These were the topic of Chapter 9.

Editing and Translation. The editing and translation of software occur, at least initially, in a tight cycle. Errors detected by the assembler are quickly corrected by editing the source file and reassembling. Since the assembler has no idea of the purpose of the program and checks only for "grammatical" errors (e.g., missing commas, undefined instructions), the errors detected are **syntax errors.** They are also called **assemble-time errors.**

Preliminary Testing. A **run-time** error will not appear until the program is executed by a simulator or in the target system. These errors may be elusive, requiring careful observation of CPU activity at each stage in the program. A **debugger** is a system program that executes a user program for the purpose of finding run-time errors. The debugger includes features such as executing the program until a certain address (a **breakpoint)** is reached, and **single-stepping** through instructions while displaying CPU registers, status bits, or input/output ports.

10.2.2 Hardware Development

For the most part, this book has not emphasized hardware development. Since the 8051 is a highly integrated device, we have focused on learning the 8051's internal architecture and exploiting its on-chip resources through software. The examples presented thus far have used only simple interfaces to external components.

Specifying Hardware. Specifying the hardware involves assigning quantitative data to system functions. For example, a robotic arm project should be specified in terms of number of articulations, reach, speed, accuracy, torque, power requirements, and so on. Designers are often required to provide a specification sheet analogous to that accompanying an audio amplifier or VCR. Other hardware specifications include physical size and weight, CPU speed, amount and type of memory, memory map assignments, I/O ports, optional features, etc.

Designing Hardware. The conventional method of hardware design, employing a pencil and a logic template, is still widely used but may be enhanced through computer-aided design (CAD) software. Although many CAD tools are for the mechanical or civil engineering disciplines, some are specifically geared for electronic engineering. The two most common examples are tools for drawing schematic diagrams and tools for laying out printed circuit hoards (PCBs). Although these programs have a long learning curve, the results are impressive. Some schematic drawing programs produce files that can be read by PCB programs to automatically generate a layout.

Building the Prototype. There are pathetically few shortcuts for the labors of prototyping. Whether breadboarding a simple interface to a bus or port connector on a single-board computer (SBC), or wire wrapping an entire controller board, the techniques of prototyping are only developed with a great deal of practice. Large companies with large budgets may proceed directly to a printed circuit board format, even for the first iteration of hardware design. Projects undertaken by small companies, students, or hobbyists, however, are more likely to use the traditional wire wrapping method for prototypes.

Preliminary Testing. The first test of hardware is undertaken in the absence of any application software. Step-wise testing is important: there's no point in measuring a clock signal using an oscilloscope before the presence of power-supply voltages has been verified. The following sequence may be followed:

- Visual checks
- Continuity checks
- DC measurements
- AC measurements

Visual and continuity checks should occur before power is applied to the board. Continuity checks using an ohmmeter should be conducted from the IC side of the prototype, from IC pin to IC pin. This way, the IC pin-to-socket and socket pin-to-wire connection are both verified. ICs should be removed when power is first applied to the prototype. DC voltages should be verified throughout the board with a voltmeter. Finally, AC measurements are made with the ICs installed to verify clock signals, and so on.

After verifying the connections, voltages, and clock signals, debugging becomes pragmatic: Is the prototype functioning as planned? If not, corrective action may take the designer back to the construction, design, or specification of the hardware.

If the design is a complete system with a CPU, a single wiring error may prevent the CPU from completing its reset sequence: The first instruction after reset may never execute! A powerful debugging trick is to drive the CPU's reset line with a low frequency square wave (≈ kHz) and observe (with an oscilloscope or logic analyzer) bus activity immediately following reset.

Functional testing of the board may require application software or a monitor program to "work" the board through its motions. It is at this stage that software must assist in completing the development cycle.

10.3 INTEGRATION AND VERIFICATION

The most difficult stage in the development cycle occurs when hardware meets software. Some very subtle bugs that eluded simulation (if undertaken) emerge under real-time execution. The problem is confounded by the need for a full complement of resources: hardware such as the PC development system, target system, power supply, cables, and test equipment; and software such as the monitor program, operating system, terminal emulation program, and so on.

We shall elaborate on the integration and verification step by first expanding the area within the dotted line in Figure 10-1. (See Figure 10-2.)

Figure 10-2 shows utility programs and development tools within circles, user files within squares, and "execution environments" within double-lined squares. The use of an editor to create a source file is straightforward. The translation step (from Figure 10-1) is shown in two stages. An assembler (e.g., ASM51) converts a source file to an object file, and a linker/locator (e.g., RL51) combines one or more relocatable object files into a single absolute object file for execution in a target system or simulator. The assembler and linker/locator also create listing files.

The most common filename suffixes are shown in parentheses for each file type. Although any filename and suffix usually can be provided as an argument, assemblers vary in their choice of default suffixes.

If the program was written originally in a single file following an absolute format, linking and locating are not necessary. In this case, the alternate path in Figure 10-2 shows the assembler generating an absolute object file.

It is also possible (although not emphasized in this book) that high-level languages, such as C or PL/M, are used instead of, or in addition to, assembly language. Translation requires a **cross-compiler** to generate the relocatable object modules for linking and locating.

A **librarian** may also participate, such as Intel's LIB51. Relocatable object modules that are general-purpose and useful for many projects (most likely subroutines) may be stored in "libraries." RL51 receives the library name as an argument and searches the library for the code (subroutines) corresponding to previously declared external symbols that have not been resolved at that point in linking/locating.

10.3.1 Software Simulation

Five execution environments are shown in Figure 10-2. Preliminary testing (see Figure 10-1) proceeds in the absence of the target system. This is shown in Figure 10-2 as software simulation. A **simulator** is a program that executes on the development system and imitates the architecture of the target machine. An 8051 simulator, for example, would contain a fictitious (or "simulated") register for each of the special function registers and fictitious memory locations corresponding to the 8051's internal and external memory spaces. Programs

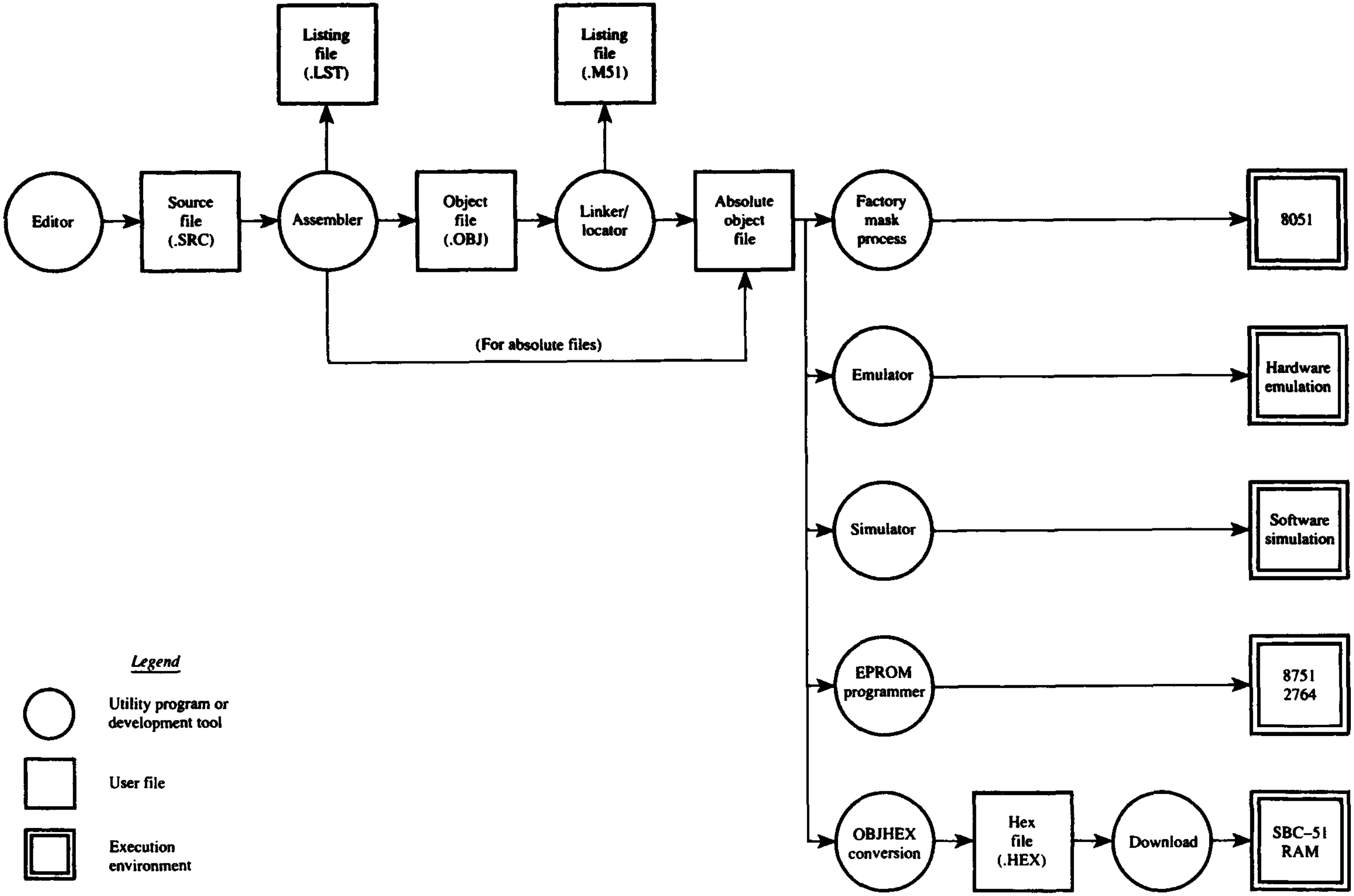


FIGURE 10-2
Detailed steps in the development cycle

are executed in simulation mode with progress presented on the development system's CRT display. Simulators are useful for early testing; however, portions of the application program that directly manipulate hardware must be integrated with the target system for testing.

10.3.2 Hardware Emulation

A direct connection between the development system and the target system is possible through a **hardware emulator** (or **in-circuit emulator).** The emulator contains a processor that replaces the processor IC in the target system. The emulator processor, however, is under the direct control of the development system. This allows software to execute in the environment of the target system without leaving the development system. Commands are available to single-step the software, execute to a breakpoint (or the n^{th} occurrence of a breakpoint), and so on. Furthermore, execution is at full speed, so time-dependent bugs may surface that eluded debugging under simulation.

The main drawback of hardware emulators is cost. PC-hosted units sell in the $2,000 to $7,000 (U.S.) range, which is beyond the budget of most hobbyists and stretches the budgets of most colleges or universities (if equipping an entire laboratory, for example). Companies supporting professional development environments, however, will not hesitate to invest in hardware emulators. The benefit in accelerating the product development process easily justifies the cost.

10.3.3 Execution from RAM

An effective and simple scenario for testing software in the target system is possible, even if a hardware emulator is not available. If the target system contains external RAM configured to overlap the external code space (using the method discussed in Chapter 2; see 2.6.4, Data Pointer), then the absolute object program can be transferred, or "downloaded," from the development system to the target system and executed in the target system.

Intel Hexadecimal Format. As shown in Figure 10-2, an extra stage of translation is required to convert the absolute object file to a standard ASCII format for transmission. Since object files contain binary codes, they cannot be displayed or printed. This weakness is alleviated by splitting each binary byte into two nibbles and converting each nibble to the corresponding hexadecimal ASCII character. For example, the byte 1AH cannot be transmitted to a printer because in ASCII it represents a control character rather than a graphic character. However, the bytes 31H and 41H can be transmitted to a printer because they correspond to graphic or displayable ASCII codes. In fact, these two bytes will print as "1A." (See Appendix F.)

One standard for storing machine language programs in .a displayable or printable format is known as "Intel hexadecimal format." An Intel hex file is a series of lines or "hex records" containing the following fields:

Field	Bytes	Description
Record mark	1	":"indicates start-of-record
Record length	2	number of data bytes in record

Load address	4	starting address for data bytes
Record type	2	00 = data record; 01 = end record
Data bytes	0-16	data
Checksum	2	sum of all bytes in record + checksum = 0

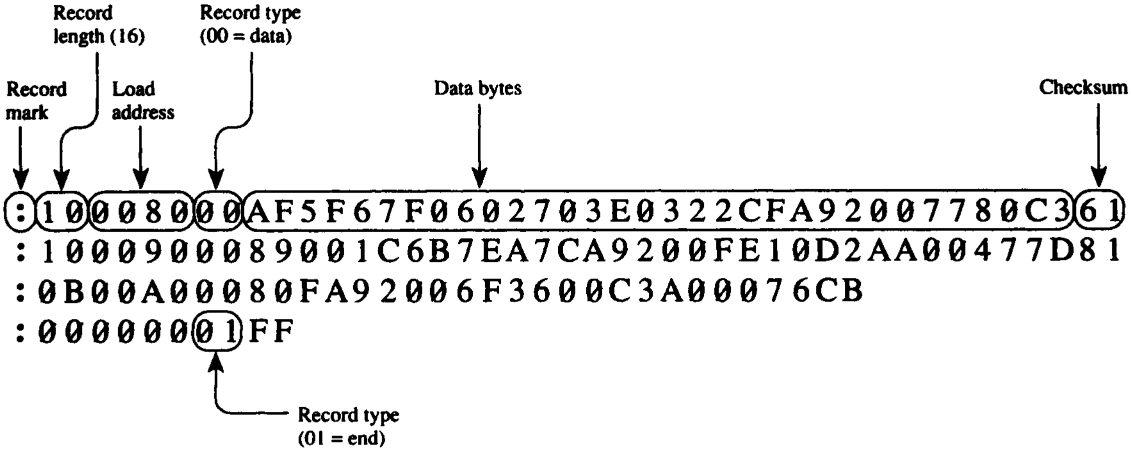
These fields are shown in the Intel hexadecimal file in Figure 10-3. Conversion programs are available that receive an absolute object program as input, convert the machine language bytes to Intel hexadecimal format, and generate a hex file as output. Intel's conversion utility is called OH.

10.3.4 Execution from EPROM

Once a satisfactory degree of performance is obtained through execution in RAM (or through in-circuit emulation), the software is burned into EPROM and installed in the system as **firmware.** Two types of EPROMs are identified in Figure 10-2 as examples. The 8751 is the EPROM version of the 8051, and the 2764 is a common, general-purpose EPROM used in many microprocessor- or microcontroller-based products. Systems designed using an 8751 benefit in that Ports 0 and 2 are available for I/O, rather than functioning as the address and data buses. However, 8751s are relatively expensive compared to 2764s ($30 versus $5, for example).

10.3.5 The Factory Mask Process

If a final design is destined for mass production, then a cost-effective alternative to EPROM is a factory mask ROM, such as the 8051. An 8051 is functionally identical to an 8751; however, code memory cannot be changed on an 8051. The data are permanently entered during the IC manufacturing cycle using a "mask"-essentially a photographic plate that passes or



```
:10008000AF5F67F0602703E0322CFA92007780C361
:1000900089001C6B7EA7CA9200FE10D2AA00477D81
:0B00A00080FA92006F3600C3A00076CB
:00000001FF
```

FIGURE 10-3
Intel hexadecimal format

masks (i.e., blocks) light during a stage of manufacturing. Connections to memory cells in the 8051 are either made or blocked, thus programming each cell as a 1 or 0.

The choice of using an 8751 versus an 8051 is largely economic. A factory mask device is considerably cheaper than the EPROM device; however, there is a large setup fee to produce the mask and initiate a custom manufacturing cycle. A tradeoff point can be identified to determine the feasibility of each approach. For example, if 8751s sell for $25 and 8051s sell for $5 plus a $5,000 setup fee, then the break-even point is

```
25 n = 5 n + 5000
20 n = 5000
   n = 250 units
```

A production run of 250 units or more would justify the use of the 8051 over the 8751.

The situation is more complicated when comparing designs using an 8051 versus an 8031 + 2764, for example. In the latter case, the 8031 + 2764 alternative is much cheaper than an 8751 with on-chip EPROM, so the tradeoff point occurs at much greater quantities. If an 8031 + 2764 sells for, say, $7, then the break-even point is

```
7 n = 5 n + 5000
2 n = 5000
  n = 2500 units
```

A production run of 1000 units would not justify use of the 8051—or so it seems. The use of external EPROM means that Ports 0 and 2 are unavailable for I/O. This may be a critical point that prevents the 8031 + 2764 approach. Even if the loss of on-chip I/O is not a concern, other factors enter. The 8031 + 2764 approach requires two ICs instead of one. This complicates manufacturing, testing, maintenance, reliability, procurement, and a host of other seemingly innocent, but nevertheless real, dimensions of product design. Furthermore, the 8031 + 2764 design will be physically larger than the 8051 design. If the final product necessitates a small form factor, then the 8051 may have to be used, regardless of the additional cost.

10.4 COMMANDS AND ENVIRONMENTS

In this section the overall development environment is considered. We present the notion that at any time the designer is working within an "environment" with commands doing the work. The central environment is the operating system on the host system, which is most likely MS-DOS running on a member of the PC family of microcomputers. As suggested in Figure 10-4, some commands return to MS-DOS upon completion, while others evoke a new environment.

Invoking Commands. Commands are either **resident** (e.g., DIR) or **transient** (e.g., FORMAT, DISKCOPY). A resident command is in memory at all times, ready for execution (e.g., **DIR**). A transient command is an executable disk file that is loaded into memory for execution (e.g., FORMAT).

Application programs are similar to transient commands in that they exist as an executable disk file and are invoked from the MS-DOS prompt. However, there are still many possibilities. Commands or applications may be invoked as part of a batch file, by a function key, or from a menu-driven user interface acting as a front-end for MS-DOS.

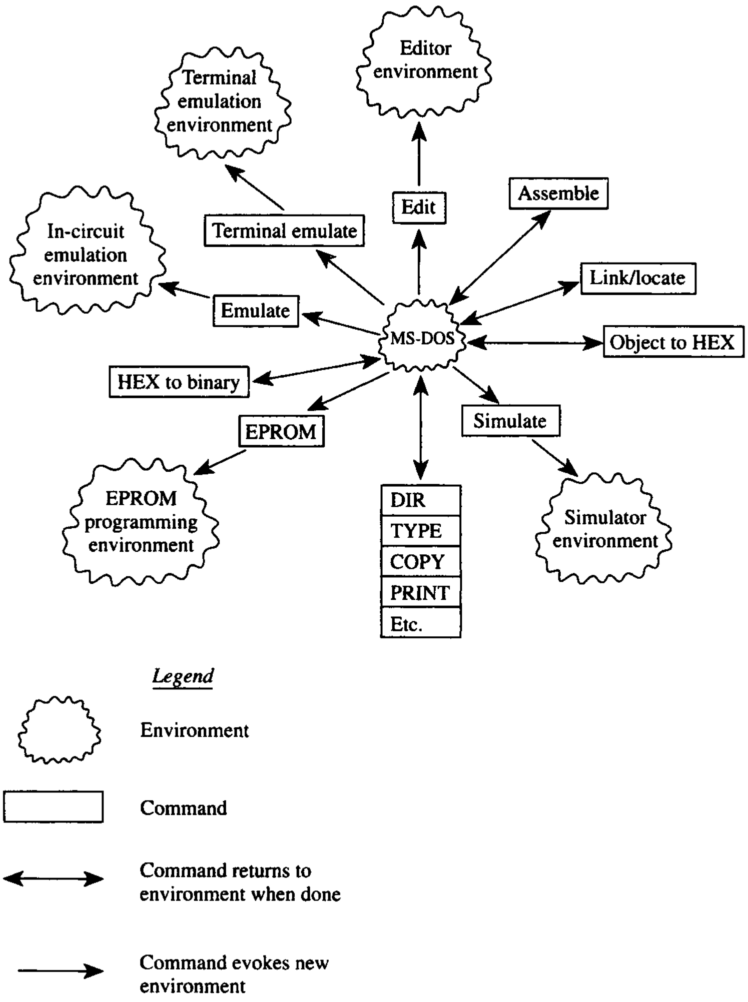


FIGURE 10-4
The development environment

If command arguments are needed, there are many possibilities again. Although arguments are typically entered on the invocation line following the command, some commands have default values for arguments, or prompt the user for arguments. Unfortunately, there is no standard mechanism, such as the "dialogue box" used in the *Macintosh* interface, to retrieve extra information needed for a command or application.

Some applications, such as editors, "take over" the system and bring the user into a new environment for subsequent activities.

Environments. As evident in Figure 10-4, some software tools such as the simulator, in-circuit emulator, or EPROM programmer evoke their own environment. Learning the nuances of each takes time, due to the great variety of techniques for directing the activities of the environment: cursor keys, function keys, first-letter commands, menu highlighting, default paths, and so on. It is often possible to switch among environments while leaving them active. For example, terminal emulators and editors usually allow switching to DOS momentarily to execute commands. The MS-DOS command EXIT immediately brings the user back the suspended environment.

Methodology. As research in artificial intelligence and cognitive science has discovered, modeling human "problem solving" is a slippery business. Humans appear to approach the elements of a situation in parallel, simultaneously weighing possible actions and proceeding by intuition. The methodology suggested here recognizes this human quality. The steps in the development cycle and the tools and techniques afforded by the development environment should be clearly understood, but the overall process should support substantial freedom.

The basic operation of commands is to "translate," "view," or "evoke" (a new environment). The results of translation should be viewed to verify results. We can take the attitude of not believing the outcome of any translation (assembling, EPROM programming, etc.) and verify everything by viewing results. Tools for viewing are commands such as DIR (Were the expected output files created?), TYPE (What's in the output file?), EDIT, PRINT, and so on.

SUMMARY

The tools and techniques available for designing microcontroller-based products have been introduced in this chapter. There is no substitute for experience, however. Success in design requires considerable intuition, a valuable commodity that cannot be delivered in a textbook. The age-old expression "trial and error" still rings true as the main technique employed by designers for turning ideas into real products.

PROBLEMS

10.1 If 8751 EPROMs sell for $30 in any quantity and a mask-programmed 8051 sells for $3 plus a $10,000 setup fee, how many units are necessary to justify use of the 8051 device? What is the savings for projected sales of 3,000 units of the final product if the 8051 is used instead of the 8751?

10.2 Below is an 8051 program in Intel hex format.

```
:10080000758911 7F007E0575A88AD28FD28D80FEF2
```

a. What is the starting address of the program?
b. What is the length of the program?
c. What is the last address of the program?

10.3 The following is a single line from an Intel hex file with an error in the checksum. The incorrect checksum appears in the last two characters as "00." What is the correct checksum?

```
:100800007589117C007F0575A8FFD28FD28D80FE00
```

10.4 The contents of an Intel hex file are shown below.

```
:090100007820765508B880FA2237
:00000001FF
```

Recreate the original source program that this file represents.

CHAPTER 11

Design and Interface Examples

11.1 INTRODUCTION

Many of the 8051's hardware and software features are brought together in this chapter through several design and interface examples. The first is an 8051 single-board computer—the SBC-51—suitable for learning about the 8051 or developing 8051-based products. The SBC-51 uses a substantial monitor program offering basic commands for system operation and user interaction. The monitor program (MON51) is described in detail in Appendix G.

The interface examples are advanced in comparison to those presented in previous chapters. Each example includes a hardware schematic, a statement of the design objective, a software listing of a program that achieves the design objective, and a general description of the operation of the hardware and software. The software listings are extensively commented and should be consulted for specific details.

11.2 THE SBC-51

Several companies offer 8051 single-board computers similar to that described in this section. Surprisingly, the basic design of an 8051 single-board computer does not vary substantially among the various products offered. Since many features are "on-chip," designing an 8051 single-board computer is straightforward. For the most part, only the basic connections to external memory and the interface to a host computer are required.

A monitor program in EPROM is also required. The most basic system requirements, such as examining and changing memory locations or downloading application programs from a host computer, are needed to get "up and running." The SBC-51 described here works together with a simple monitor program to provide these basic functions.

Figure 11-1 contains the schematic diagram for the SBC-51. The entire design includes only 10 ICs yet is powerful and flexible enough to support the development of sophisticated 8051-based products. Central to the operation of the SBC-51 is a monitor program that resides in EPROM and communicates with a video display terminal (VDT) connected to the 8051. The monitor program is described in detail in Appendix G.

The SBC-51 includes, in addition to the standard 80C31 features, 16K bytes of external EPROM, 8.25K bytes of external RAM, an extra 14-bit timer, and 22 extra input/output lines. The configuration shown in Figure 11-1 includes the following components and parts:

- 10 integrated circuits
- 15 capacitors
- 2 resistors
- 1 crystal
- 1 push-button switch
- 3 connectors
- 13 configuration jumpers

Since external memory is used, Port 0 and Port 2 are unavailable for input/output. Although Ports 1 and 3 are partially utilized for special features, some Port 1 and Port 3 lines may be used for input/output purposes, depending on the configuration.

The 80C31 clock source is a 12 MHz crystal connected in the usual way. (See Figure 2-2.) The RST (reset) line is driven by an R-C network for power-on reset and by a push button switch for manual reset. Port 0 doubles as the data bus (D0 to D7) and the low-byte of the address bus (A0 to A7), as discussed earlier. (See 2.7 External Memory.) A 74HC373 octal latch is clocked by ALE to hold the low-byte of the address bus for the duration of a memory cycle. Since the 80C31 does not include on-chip ROM, execution is from external EPROM, and so EA (external access) is connected to ground through configuration jumper X2.

The connection to the host computer or VDT uses a serial RS232C interface. The DB25S connector is wired as a DTE (data terminal equipment) with transmit data (TXD) on pin 2, receive data (RXD) on pin 3, and ground on pin 7. A 1488 RS232 line driver connects to TXD and a 1489 RS232 line receiver connects to RXD. The default connection to the 80C31 is through jumpers X9 and X10 with P3.1 as TXD and P3.0 as RXD. Optionally, through jumpers X11 and X12, the TXD and RXD functions can be provided through software using P1.7 and P1.6.

Port 1 lines 3, 4, and 5 are read by the monitor program upon reset to evoke special features. After reset, however, these lines are available for general-purpose I/O. If the printer interface is used, Port 1 lines 0, 1, and 2 are the handshake signals. If the printer interface is not used, these lines are available for general-purpose I/O.

The 74H4C138 decodes the upper three bits on the address bus (A15 to A13) and generates eight select lines, one for each 8K block of memory. These are called S8K0 (for "select 8K block 0") through to S8K7. Four ICs are selected by these lines: two 2764 EPROMs, a 6264 RAM, and an 8155 RAM/IO/TIMER.

Two 2764 8K by 8 EPROMs are shown in Figure 11-1. The first (labeled "MONITOR EPROM") is selected by S8K0 and resides in the external code space from address

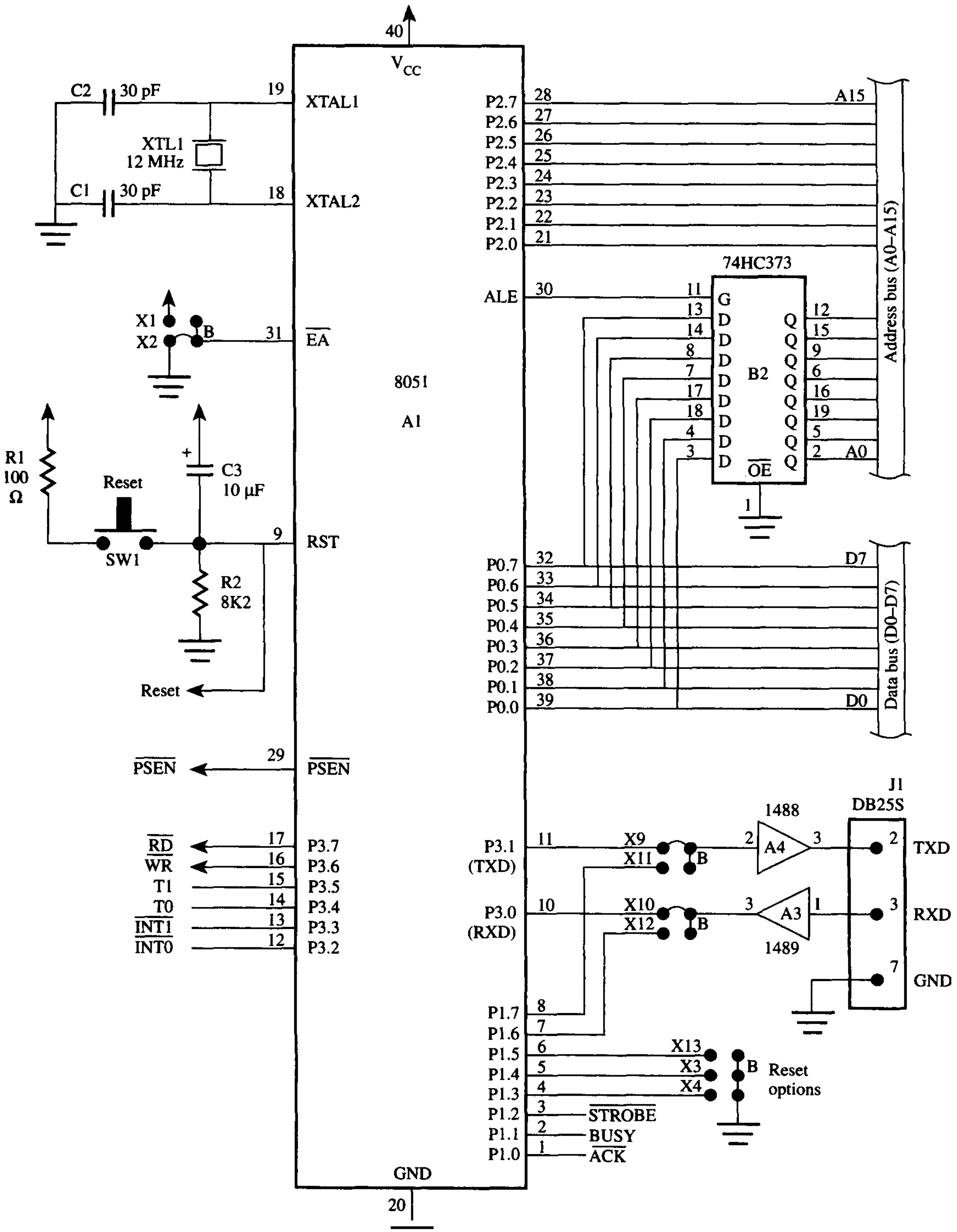


FIGURE 11-1a

An 8051 single-board computer—the SBC-51 (a) Processor and serial port interface; (b) Address decoding, RAM, and EPROM; (c) 8155 and power connections

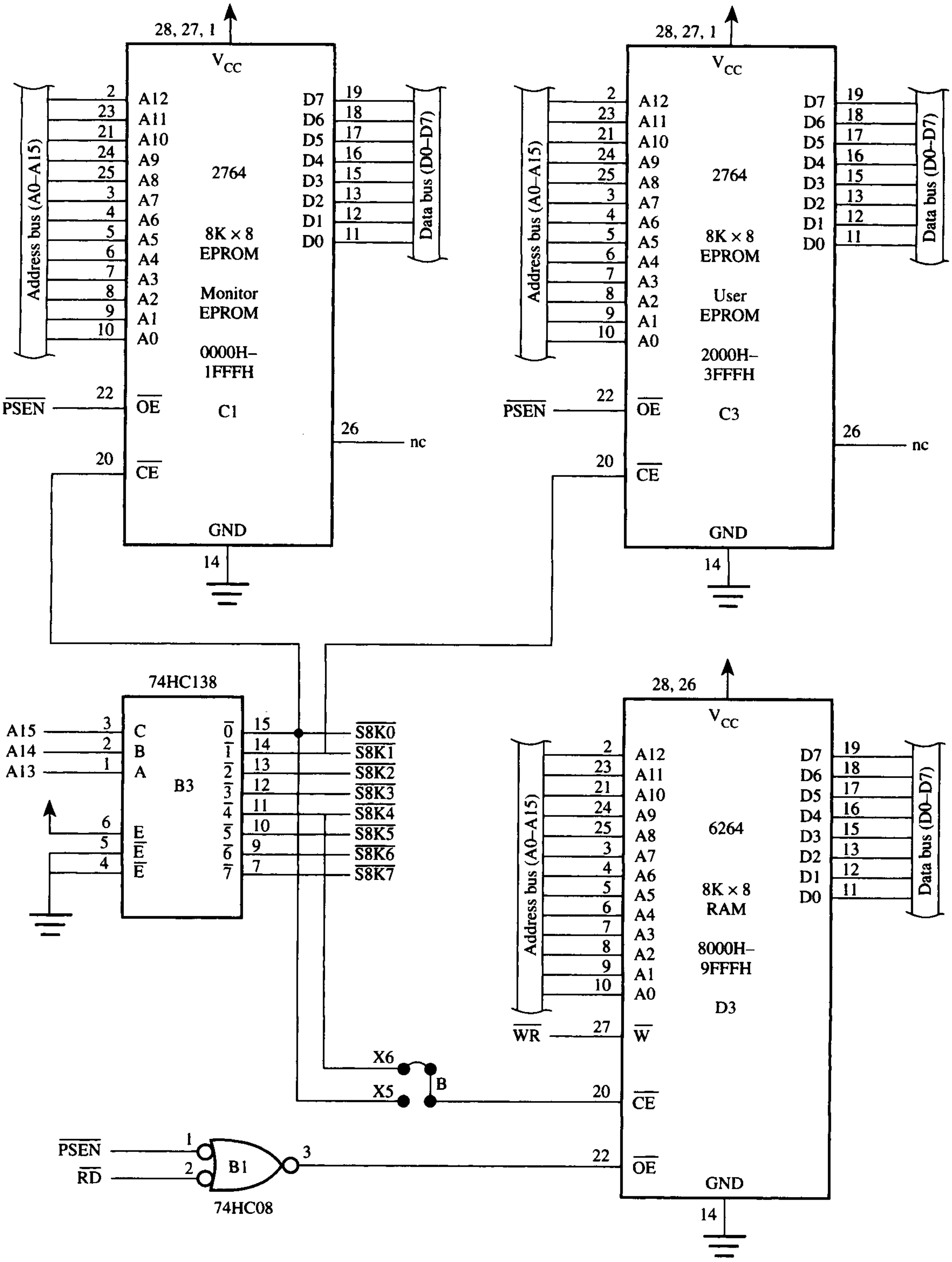
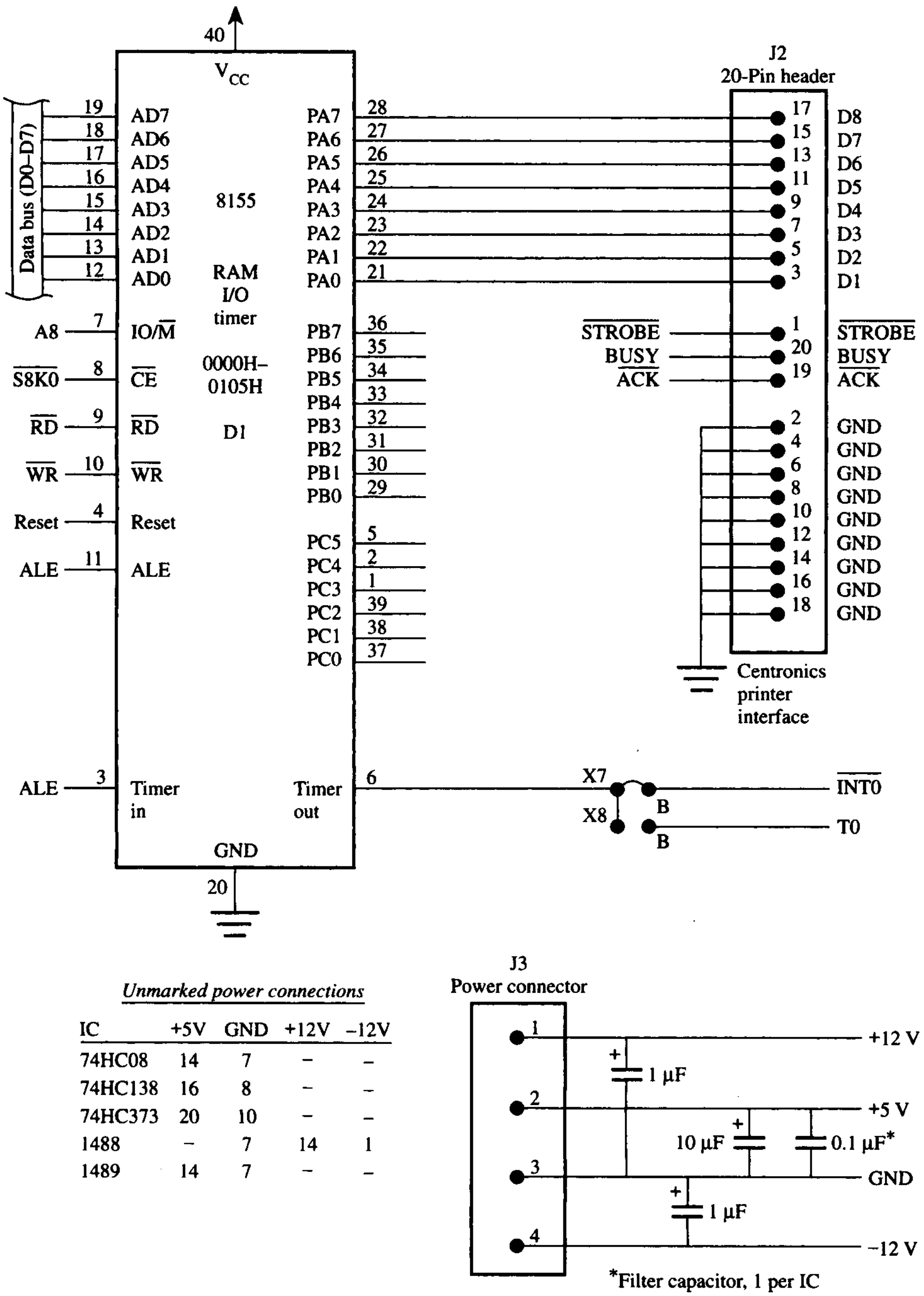


FIGURE 11-1b
continued



Unmarked power connections

IC	+5V	GND	+12V	−12V
74HC08	14	7	–	–
74HC138	16	8	–	–
74HC373	20	10	–	–
1488	–	7	14	1
1489	14	7	–	–

FIGURE 11-1c
continued

0000H to 1FFFH. Since the SBC-51 will begin execution from address 0000H immediately after a system reset, the monitor program must reside in this IC. The second 2764 is labeled "USER EPROM" and is selected by $\overline{\text{S8K1}}$ for execution at addresses 2000H to 3FFFH. This IC is intended for user applications and is not needed for basic system operation. Note that both EPROMs are selected only if $\overline{\text{CE}}$ (chip enable; pin 20) is active (or low) and $\overline{\text{OE}}$ is also active (or low). $\overline{\text{OE}}$ is driven by the 80C31's PSEN line; thus selection is in the external code space, as expected.

The 6264 8K by 8 RAM IC is selected by $\overline{\text{S8K4}}$ (if jumper X6 is installed, as shown), so it resides at addresses 8000H to 9FFFH. The RAM is selected to occupy both the external data space and the external code space using the method described earlier. (See Section 2.7.4 Overlapping the External Code and Data Spaces.) This dual occupancy allows user programs to be loaded (or written) to the RAM as "data memory" and then executed as "code memory."

The 8155 RAM/IO/TIMER is a peripheral interface IC that was added to demonstrate the expansion capabilities of the SBC-51. It is easy to add other peripheral interface ICs in a similar way. The 8155 is selected by $\overline{\text{S8K0}}$ placing it at the bottom of memory. No conflict occurs with the monitor EPROM (which also resides at the bottom of memory, but in the external code space) because the 8155 is further selected for read and write operations using $\overline{\text{RD}}$ and $\overline{\text{WR}}$.

The 8155 contains the following features:

- 256 bytes of RAM
- 22 input/output lines
- 14-bit timer

Address line A8 connects to the 8155's IO/$\overline{\text{M}}$ line (pin 7) and selects the RAM when low and the I/O lines or timer when high. The I/O lines and timer are accessed from six addresses, so the total address range of the 8155 is 0000H to 0105H (256 + 6 addresses). These are summarized below.

Address	**Purpose**
0000H	**first RAM address**
	Other RAM addresses
OOFFH	**last RAM address**
0100H	**Interval/command register**
0101H	**Port A**
0102H	**Port B**
0103H	**Port C**
0104H	**Low-order 8 bits of timer count**
0105H	**High-order 6 bits of timer count & 2 bits of timer mode**

Although the manufacturer's data sheet should be consulted for details of the 8155's operation, configuring the I/O ports is extremely easy. By default all port lines are inputs after a system reset; therefore, no "initialize" operation is needed to read input devices connected to the 8155. To read Port A into the accumulator, for example, the following instruction sequence is used:

```
MOV DPTR,#0100H       ;DPTR points to 8155 Port A
MOVX A,@DPTR          ;read Port A into Acc
```

To program Port A and Port B as outputs, is must first be written into the command register bits 0 and 1, respectively. For example, to configure Port B as an output port and leave Port A and Port C as input, the following instruction sequence is used:

```
MOV DPTR,#0100H       ;8155 command register
MOV A,#0000001OB      ;Port B = output
MOVX @DPTR,A          ;initialize 8155
```

Port C is configured as an output by writing 1s to the command register bits 2 and 3. All three ports would be configured as output as follows:

```
MOV DPTR,#0100H       ;8155 command register
MOV A,#00001111B      ;all ports = output
MOVX @DPTR            ;initialize 8155
```

Port A of the 8155 is shown connected to a 20-pin header labeled "Centronics printer interface." This interface is for demonstration purposes only. MON51 includes a PCHAR (print character) subroutine and directs output to the VDT *and* a parallel printer if CONTROL-Z is entered on the keyboard. (See Appendix G.) Of course, Port A can be used for other purposes if desired.

Power-supply connections are also shown in Figure 11-1. The filter capacitors are particularly important for the +5 volt supply to avoid glitches due to inductive effects when digital devices switch. If the SBC-51 is constructed on a prototype board (for example, by wire wrapping), these capacitors should be considered critical. Place a 10 µF electrolytic capacitor where power enters the prototype board, and 0.01 µF ceramic capacitors beside the socket for each IC, wired between the +5 volt pin and the ground pin.

Since the SBC-51 is small and inexpensive, it is easy to construct a prototype and gain hands-on experience through the monitor program and the interfacing examples in this chapter. Wire wrapping is the most practical method of construction. The SBC-51 is also available assembled and tested on a printed-circuit board (see Figure 11-2).

This concludes our description of the SBC-51. The following sections contain examples of interfaces to peripheral devices that have been developed to connect to the SBC-51 (or a similar 8051 single-board computer).

11.3 HEXADECIMAL KEYPAD INTERFACE

Interfaces to keypads are common for microcontroller-based designs. Keypad input and LED output are an economical choice for a user interface and are often adequate

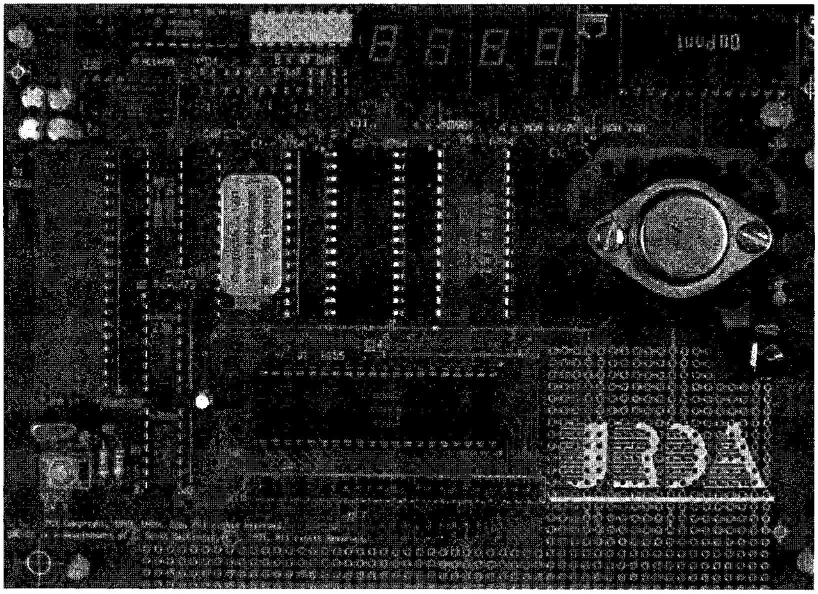


FIGURE 11-2
The printed-circuit board version of the SBC-51 (Courtesy URDA, Inc.)

for complex applications. Examples include the user interface to microwave ovens or automated banking machines. Figure 11-3 shows an interface between Port 1 and a hexadecimal keypad. The keypad contains 16 keys arranged in four rows and four columns. The row lines are connected to Port 1 bits 4-7, the column lines to Port 1 bits 0-3.

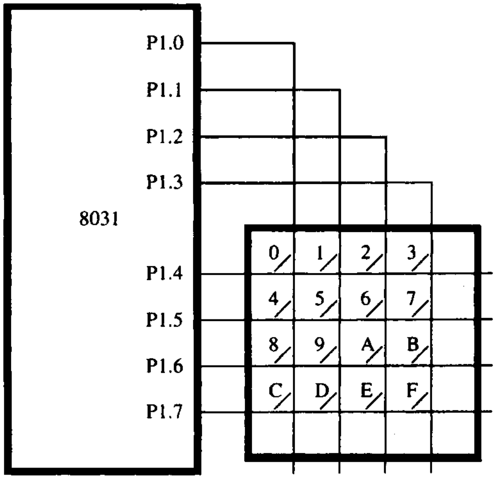


FIGURE 11-3
Interface to hexadecimal keypad

EXAMPLE 11.1

Design Objective

Write a program that continually reads hexadecimal characters from the keypad and echoes the corresponding ASCII code to the console.

On the surface, this example seems quite simple. The software can be divided into the following steps:

1. Get a hexadecimal character from the keypad.
2. Convert the hexadecimal code to ASCII.
3. Send the ASCII code to the VDT.
4. Go to step 1.

In fact, the software solution shown in Figure 11-4 follows this exact pattern (see lines 16-19). Of course, the work is done in the subroutines. Note that steps 2 and 3 above are implemented by calling subroutines in MON51. Of course, the code could have been extracted from MON51 and placed in the listing in Figure 11-4, but that's wasteful. Instead, the MON51 entry points for these subroutines are defined near the top of the listing (in lines 12-13) using the symbols HTOA and OUTCHR, and then the subroutines are called in the MAIN program loop in the usual way. Incidentally, the entry points for MON51 subroutines can be found in the symbol table created by RL51 when MON51 was linked and located. The entry points for HTOA and OUTCHR, for example, are found in Appendix G.

The real challenge for this example is writing the subroutines IN_HEX and GET_KEY. GET_KEY does the work of scanning the row and column lines of the key pad to determine if a key is pressed. If no key is pressed, it returns with C = 0. If a key is pressed, it returns with C = 1 and the hexadecimal code for the key in the accumulator bits 0-3.

IN_HEX performs software debouncing. Since the keypad is a series of mechanical switches, contact closure and release include bounce—the rapid but brief make-and-break of the switch contacts. Debouncing is performed by calling GET_HEX repeatedly until 50 consecutive calls return with C = 1. Any call to GET_HEX returning with C = 0 is interpreted as noise (i.e., bounce) and the counter is reset. After detecting a legitimate key closure, IN_HEX then waits for 50 consecutive calls to GET_HEX returning with C = 0. This ensures a clean key release before the next call to GET_HEX.

The software in Figure 11-4 works, but it is not particularly elegant. Since interrupts are not used, the program's utility within a larger application is limited. A reasonable improvement, therefore, is to redesign the software, using interrupts. An interrupt-driven interface is illustrated in the next example.

11.4 INTERFACE TO MULTIPLE 7-SEGMENT LEDS

An interface to a 7-segment LED display was presented in a problem at the end of Chapter 3. (See Figure 3-8.) Unfortunately, the interface used seven lines on Port 1, so it represents a poor allocation of the 8051's on-chip resources. In this section, we

```
LOC   OBJ               LINE   SOURCE
                           1   $DEBUG
                           2   $NOPAGING
                           3   $NOSYMBOLS
                           4   ;FILE: KEYPAD.SRC
                           5   ;*****************************************************
                           6   ;               KEYPAD INTERFACE EXAMPLE
                           7   ;
                           8   ; This program reads hexadecimal characters from a
                           9   ; keypad attached to Port 1 and echos keys pressed
                          10   ; to the console.
                          11   ;*****************************************************
  033C                    12   HTOA     EQU     033CH    ;MON51 subroutines (V12)
  01DE                    13   OUTCHR   EQU     01DEH
                          14
8000                      15            ORG     8000H
8000 12800B               16   MAIN:    CALL    IN_HEX   ;get code from keypad
8003 12033C               17            CALL    HTOA     ;convert to ASCII
8006 1201DE               18            CALL    OUTCHR   ;echo to console
8009 80F5                 19            SJMP    MAIN     ;repeat
                          20
                          21   ;*****************************************************
                          22   ; IN_HEX - input hex code from keypad with debouncing
                          23   ;          for key press and key release (50 repeat
                          24   ;          operations for each)
                          25   ;*****************************************************
800B 7B32                 26   IN_HEX:  MOV     R3,#50   ;debounce count
800D 128022               27   BACK:    CALL    GET_KEY  ;key pressed?
8010 50F9                 28            JNC     IN_HEX   ;no:  check again
8012 DBF9                 29            DJNZ    R3,BACK  ;yes: repeat 50 times
8014 C0E0                 30            PUSH    ACC      ;save hex code
8016 7B32                 31   BACK2:   MOV     R3,#50   ;wait for key up
8018 128022               32   BACK3:   CALL    GET_KEY  ;key pressed?
801B 40F9                 33            JC      BACK2    ;yes: keep checking
801D DBF9                 34            DJNZ    R3,BACK3 ;no:  repeat 50 times
801F D0E0                 35            POP     ACC      ;recover hex code and
8021 22                   36            RET              ; return
                          37
                          38   ;*****************************************************
                          39   ; GET_KEY - get keypad status
                          40   ;         - return with C = 0 if no key pressed
                          41   ;         - return with C = 1 and hex code in ACC if
                          42   ;           a key is pressed
                          43   ;*****************************************************
8022 74FE                 44   GET_KEY: MOV     A,#0FEH            ;start with column 0
8024 7E04                 45            MOV     R6,#4              ;use R6 as counter
8026 F590                 46   TEST:    MOV     P1,A               ;activate colmn line
8028 FF                   47            MOV     R7,A               ;save ACC
8029 E590                 48            MOV     A,P1               ;read back Port 0
802B 54F0                 49            ANL     A,#0F0H            ;isolate row lines
802D B4F007               50            CJNE    A,#0F0H,KEY_HIT    ;row line active?
8030 EF                   51            MOV     A,R7               ;no: move to next
8031 23                   52            RL      A                  ;    column line
8032 DEF2                 53            DJNZ    R6,TEST
8034 C3                   54            CLR     C                  ;no key pressed
8035 8015                 55            SJMP    EXIT               ;return with C = 0
8037 FF                   56   KEY_HIT: MOV     R7,A               ;save in R6
8038 7404                 57            MOV     A,#4               ;prepare to caculate
803A C3                   58            CLR     C                  ; column weighting
803B 9E                   59            SUBB    A,R6               ;4 - R6 = weighting
803C FE                   60            MOV     R6,A               ;save in R6
803D EF                   61            MOV     A,R7               ;restore scan code
803E C4                   62            SWAP    A                  ;put in low nibble
803F 7D04                 63            MOV     R5,#4              ;use R5 as counter
8041 13                   64   AGAIN:   RRC     A                  ;rotate until 0
```

FIGURE 11-4a
Software for keypad interface

```
8042 5006        65              JNC     DONE          ;done when C = 0
8044 0E          66              INC     R6            ;add 4 until active
8045 0E          67              INC     R6            ; row found
8046 0E          68              INC     R6
8047 0E          69              INC     R6
8048 DDF7        70              DJNZ    R5,AGAIN
804A D3          71      DONE:   SETB    C             ;C = 1 (key pressed)
804B EE          72              MOV     A,R6          ;code in A (whew!!!)
804C 22          73      EXIT:   RET
                 74              END
```

FIGURE 11-4b
continued

demonstrate an interface to four 7-segment LEDs using only three of the 8051's I/O lines. This, obviously, is a much-improved design, particularly if multiple segments must be connected.

Central to the design is the Motorola MC14499 7-segment decoder/driver, which includes much of the circuitry necessary to drive four displays. The only additional components are a 0.015 µF timing capacitor, seven 47 Ohm current-limiting resistors, and four 2N3904 transistors. Figure 11-5 shows the connections between the 80051, the MC14499, and the four 7-segment LEDs.

EXAMPLE 11.2 Design Objective

Assume BCD digits are stored in internal RAM locations 70H and 71H. Copy the BCD digits to the LED display 10 times per second using interrupts.

The software to accomplish the above objective is shown in Figure 11-6. The listing illustrates a number of concepts discussed earlier. The low-level details of sending data to the MC14499 are found in the subroutines UPDATE and OUTS. At a higher level, this example illustrates the design of interrupt-driven applications with a significant amount of foreground *and* background activity (unlike the examples in Chapter 6, which operated only in the background). The interrupts for this example coexist with MON51, which does not itself use interrupts. The monitor program executes in the foreground while the program in Figure 11-6 executes at interrupt-level in the background. When the program is started (e.g., by entering the MON51 command GO8000; see Appendix G), conditions are initialized for the necessary interrupt-initiated updating of the LED displays, and then control quickly passes back to the monitor program. Monitor commands can be executed in the usual way; meanwhile, interrupts are occurring in the background. If, for example, the monitor SET command is used to change internal RAM locations 70H and 71H, the changes are seen immediately (within 0.1 s) on the 7-segment LED displays.
Note the overall structure of the program. The following sections appear in order:

- Assembler controls (lines 1-3)
- Comment block (lines 4-30)

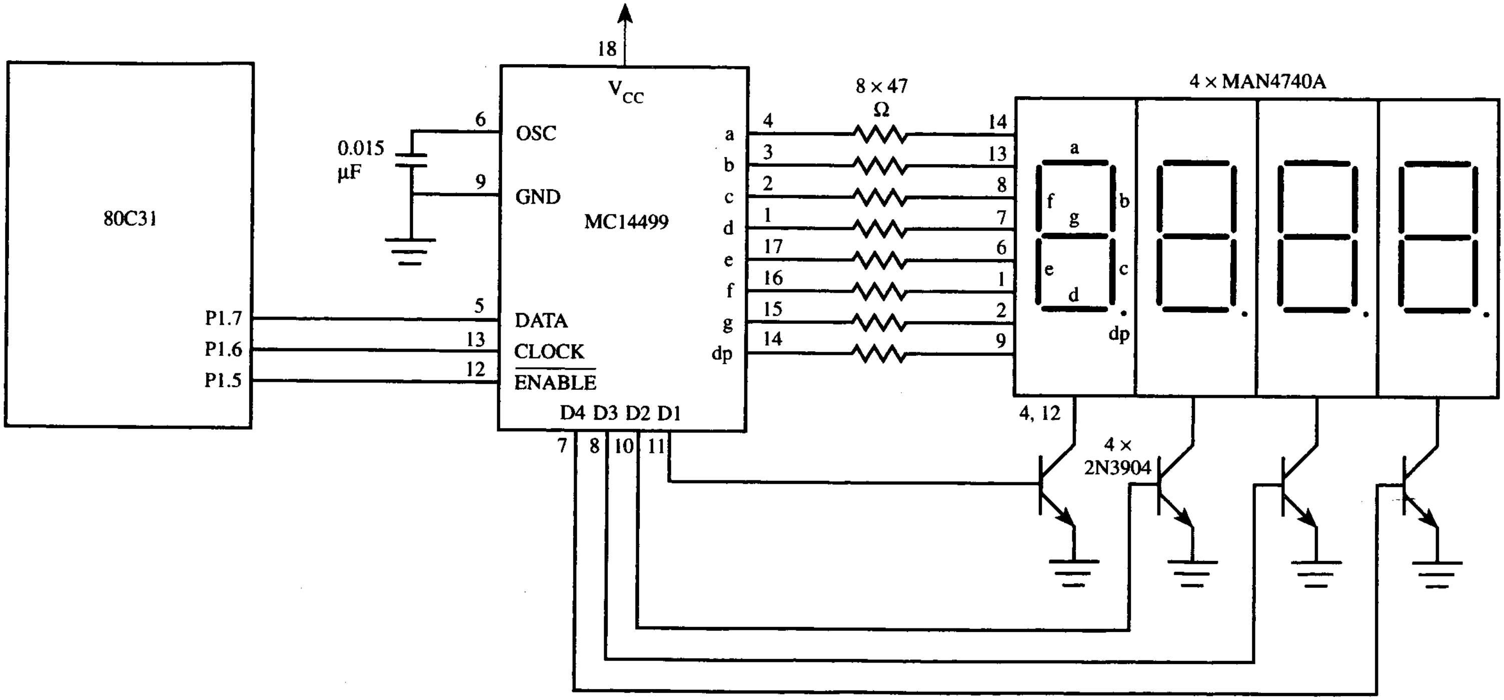
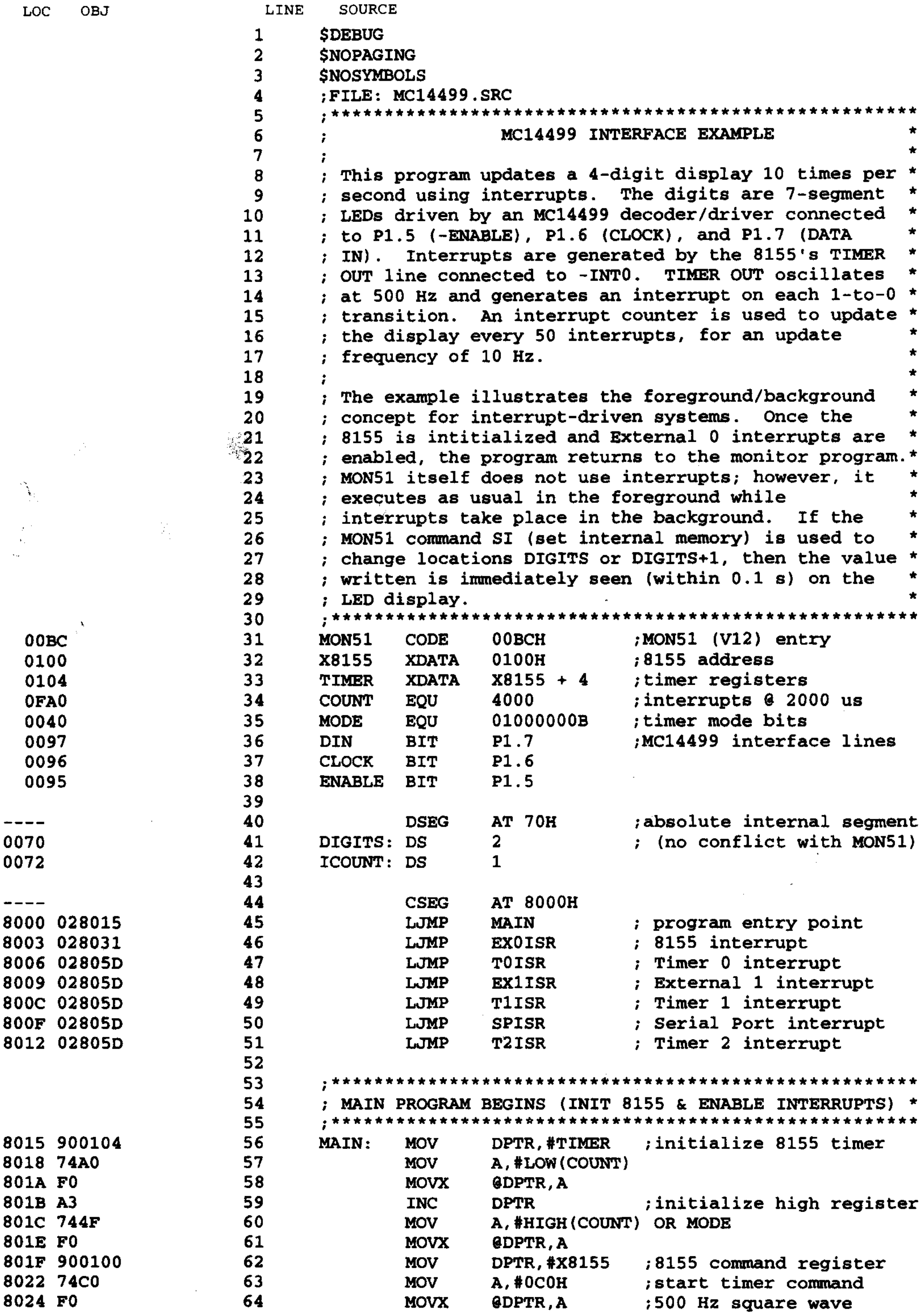


FIGURE 11-5

Interface to MC14499 and four 7-segment LEDs



```
 LOC   OBJ             LINE    SOURCE
                          1     $DEBUG
                          2     $NOPAGING
                          3     $NOSYMBOLS
                          4     ;FILE: MC14499.SRC
                          5     ;*******************************************************
                          6     ;                 MC14499 INTERFACE EXAMPLE            *
                          7     ;                                                      *
                          8     ; This program updates a 4-digit display 10 times per *
                          9     ; second using interrupts.  The digits are 7-segment   *
                         10     ; LEDs driven by an MC14499 decoder/driver connected   *
                         11     ; to P1.5 (-ENABLE), P1.6 (CLOCK), and P1.7 (DATA      *
                         12     ; IN).  Interrupts are generated by the 8155's TIMER   *
                         13     ; OUT line connected to -INT0.  TIMER OUT oscillates   *
                         14     ; at 500 Hz and generates an interrupt on each 1-to-0 *
                         15     ; transition.  An interrupt counter is used to update *
                         16     ; the display every 50 interrupts, for an update       *
                         17     ; frequency of 10 Hz.                                  *
                         18     ;                                                      *
                         19     ; The example illustrates the foreground/background    *
                         20     ; concept for interrupt-driven systems.  Once the      *
                         21     ; 8155 is intitialized and External 0 interrupts are   *
                         22     ; enabled, the program returns to the monitor program.*
                         23     ; MON51 itself does not use interrupts; however, it    *
                         24     ; executes as usual in the foreground while            *
                         25     ; interrupts take place in the background.  If the     *
                         26     ; MON51 command SI (set internal memory) is used to    *
                         27     ; change locations DIGITS or DIGITS+1, then the value *
                         28     ; written is immediately seen (within 0.1 s) on the    *
                         29     ; LED display.                                         *
                         30     ;*******************************************************
  00BC                   31     MON51   CODE    00BCH        ;MON51 (V12) entry
  0100                   32     X8155   XDATA   0100H        ;8155 address
  0104                   33     TIMER   XDATA   X8155 + 4    ;timer registers
  0FA0                   34     COUNT   EQU     4000         ;interrupts @ 2000 us
  0040                   35     MODE    EQU     01000000B    ;timer mode bits
  0097                   36     DIN     BIT     P1.7         ;MC14499 interface lines
  0096                   37     CLOCK   BIT     P1.6
  0095                   38     ENABLE  BIT     P1.5
                         39
----                     40             DSEG    AT 70H       ;absolute internal segment
0070                     41     DIGITS: DS      2            ; (no conflict with MON51)
0072                     42     ICOUNT: DS      1
                         43
----                     44             CSEG    AT 8000H
8000 028015              45             LJMP    MAIN         ; program entry point
8003 028031              46             LJMP    EX0ISR       ; 8155 interrupt
8006 02805D              47             LJMP    T0ISR        ; Timer 0 interrupt
8009 02805D              48             LJMP    EX1ISR       ; External 1 interrupt
800C 02805D              49             LJMP    T1ISR        ; Timer 1 interrupt
800F 02805D              50             LJMP    SPISR        ; Serial Port interrupt
8012 02805D              51             LJMP    T2ISR        ; Timer 2 interrupt
                         52
                         53     ;*******************************************************
                         54     ; MAIN PROGRAM BEGINS (INIT 8155 & ENABLE INTERRUPTS) *
                         55     ;*******************************************************
8015 900104              56     MAIN:   MOV     DPTR,#TIMER   ;initialize 8155 timer
8018 74A0                57             MOV     A,#LOW(COUNT)
801A F0                  58             MOVX    @DPTR,A
801B A3                  59             INC     DPTR          ;initialize high register
801C 744F                60             MOV     A,#HIGH(COUNT) OR MODE
801E F0                  61             MOVX    @DPTR,A
801F 900100              62             MOV     DPTR,#X8155   ;8155 command register
8022 74C0                63             MOV     A,#0C0H       ;start timer command
8024 F0                  64             MOVX    @DPTR,A       ;500 Hz square wave
```

FIGURE 11-6a

Software for MC14499 interface

```
8025 757232                  MOV     ICOUNT,#50     ;initialize int. counter
8028 D2AF                    SETB    EA             ;enable interrupts
802A D2A8                    SETB    EX0            ;enable External 0 int.
802C D288                    SETB    IT0            ;negative-edge triggered
802E 0200BC                  LJMP    MON51          ;return to MON51

                    ;*****************************************************
                    ; EXTERNAL 0 INTERRUPT SERVICE ROUTINE                *
                    ;*****************************************************
8031 D57205         EX0ISR: DJNZ     ICOUNT,EXIT    ;on 50th interrupt,
8034 757232                 MOV      ICOUNT,#50     ; reset counter and
8037 113A                   ACALL    UPDATE         ; refresh LED display
8039 32             EXIT:   RETI

                    ;*****************************************************
                    ; UDATE 4-DIGIT LED DISPLAY (EXECUTION TIME = 84 us)  *
                    ;                                                     *
                    ; ENTER:         Four BCD digits in internal memory   *
                    ;                locations DIGITS and DIGITS+1 (MSD in *
                    ;                high nibble of DIGITS)               *
                    ; EXIT:          MC14499 display updated              *
                    ; USES:          P1.5, P1.6, P1.7                     *
                    ;                All memory locations and regs intact *
                    ;*****************************************************
803A C0E0           UPDATE: PUSH     ACC          ;save Accumulator on stack
803C C295                   CLR      ENABLE       ;prepare MC14499
803E E570                   MOV      A,DIGITS     ;get first two digits
8040 114B                   ACALL    OUT8         ;send two digits
8042 E571                   MOV      A,DIGITS + 1 ;get second byte
8044 114B                   ACALL    OUT8         ;send last two digits
8046 D295                   SETB     ENABLE       ;disable MC14499
8048 D0E0                   POP      ACC          ;restore ACC from stack
804A 22                     RET

                    ;*****************************************************
                    ; SEND 8 BITS IN ACCUMULATOR TO MC14499 (MSB FIRST)   *
                    ;*****************************************************
                            USING    0          ;assume reg. bank 0 enabled
804B C007           OUT8:   PUSH     AR7        ;save R7 on stack
804D 7F08                   MOV      R7,#8      ;use R7 as bit counter
804F 33             AGAIN:  RLC      A          ;put bit in C flag
8050 9297                   MOV      DIN,C      ;send it to MC14499
8052 C296                   CLR      CLOCK      ;3 us low pulse on clock line
8054 00                     NOP                 ;NOPs needed to stretch pulse
8055 00                     NOP                 ; (minimum pulse width is
8056 D296                   SETB     CLOCK      ; is 2 us)
8058 DFF5                   DJNZ     R7,AGAIN   ;repeat until all 8 bits sent
805A D007                   POP      AR7        ;restore R7 from stack
805C 22                     RET

                    ;*****************************************************
                    ; UNUSED INTERRUPTS (ERROR; RETURN TO MONITOR PROGRAM)*
                    ;*****************************************************
                    T0ISR:
                    EX1ISR:
                    T1ISR:
                    SPISR:
805D C2AF           T2ISR:  CLR      EA         ;shut off interrupts &
805F 0200BC                 LJMP     MON51      ; return to MON51
                            END
```

FIGURE 11-6b
continued

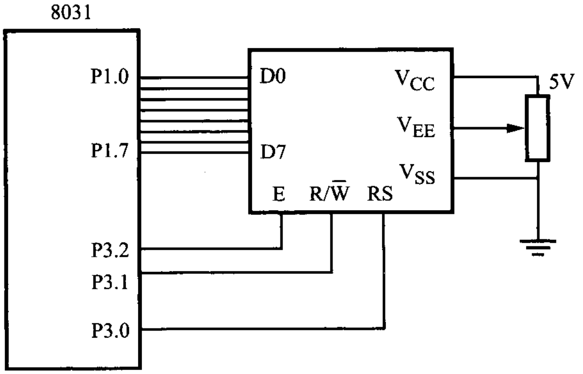
- Definition of symbols (lines 31-38)
- Define storage declarations (lines 40-42)
- Jump table for program and interrupt entry points (lines 44-51)
- Main section (MAIN; lines 56-69)
- External interrupt service routine (EXTOISR; lines 74-77)
- Update LED display subroutine (UPDATE; lines 89-97)
- Output byte subroutine (OUT8; lines 103-113)
- Code to handle unimplemented interrupts (lines 118-123)

The program is written for execution at address 8000H in the SBC-5l's 6264 RAM IC. Since interrupts vector through locations at the bottom of memory, the monitor program includes a jump table redirecting interrupts to addresses starting at address 8000H. (See Appendix G.) The program entry point is conveniently 8000H; however, an LJMP instruction (line 45; see Figure 11-6) passes control to the label MAIN. All the initialize instructions are contained in lines 56-68. The MAIN section terminates by jumping back the monitor program.

11.5 INTERFACE TO LIQUID CRYSTAL DISPLAYS (LCDS)

The previous section saw the use of 7-segment LEDs for display purposes. The 7-segment display is sufficient for displaying numbers and simple characters but some more complex characters require the use of other alternatives such as the liquid crystal display (LCD). One very popular application of LCDs is in scientific calculators. In this section, we will show how to interface to a simple LCD consisting of two lines of 16 characters, each character formed by a 5 x 7 dot matrix. Most LCDs are compatible with the de facto Hitachi HD44780 standard. The connections between the 8051 and an HD44780-compatible LCD are straightforward and are shown in Figure 11-7.

FIGURE 11-7
Interface to an LCD



EXAMPLE 11.3

Design Objective

Assume that ASCII characters are stored in internal RAM locations 30H-7FH. Write a program to continually display these characters on the LCD, 16 characters at a time.

The software listing for this is in Figure 11-8. As previously shown in Figure 11-7, the LCD has three control lines: Register Select (RS), Read/Write (R/$\overline{W}$), and Enable (E). When RS = 0, a command word is to be sent to the LCD while if RS = 1, a data word would be sent

```
               $DEBUG
               $NOSYMBOLS
               $NOPAGING
               ;FILE: LCD.SRC
               ;*****************************************************
               ;                   LCD INTERFACE EXAMPLE           *
               ;                                                   *
               ; This program continually displays on the LCD      *
               ; the ASCII characters stored in internal RAM       *
               ; locations 30H-7FH.                                *
               ;*****************************************************

 00B0                  RS      EQU          P3.0
 00B1                  RW      EQU          P3.1
 00B2                  E       EQU          P3.2
 0090                  DBUS    EQU          P1
 REG                   PTR     EQU          R0
 REG                   COUNT   EQU          R1

8000                   ORG 8000H
8000 1115      MAIN:   ACALL INIT           ;initialize LCD
8002 7916              MOV COUNT, #16       ;initialize char count
8004 7830      STRT:   MOV PTR, #30H        ;initialize pointer
8006 E6        NEXT:   MOV A, @PTR
8007 113A              ACALL DISP           ;display on LCD
8009 08                INC PTR              ;point to next location
800A D904              DJNZ COUNT, TEST     ;is it end of line?
800C 7916              MOV COUNT, #16       ;yes: reinitialize count
800E 1127              ACALL NEW            ; and refresh LCD
8010 B880F3    TEST:   CJNE PTR, #80H, NEXT ;last location?
                                            ;no: go to next
8013 80EF              SJMP STRT            ;yes: go back to start

               ;*****************************************************
               ; Initialize the LCD                                *
               ;*****************************************************

8015 7438      INIT:   MOV A, #38H          ;2 lines, 5 x 7 matrix
8017 113D              ACALL WAIT           ;wait for LCD to be free
```

FIGURE 11-8a
Software for LCD interface

```
8019 C2B0                     CLR RS              ;output a command
801B 114B                     ACALL OUT           ;send it out

801D 740E                     MOV A, #0EH         ;LCD on, cursor on
801F 113D                     ACALL WAIT          ;wait for LCD to be free
8021 C2B0                     CLR RS              ;output a command
8023 114B                     ACALL OUT           ;send it out

8025 7401           NEW:      MOV A, #01H         ;clear LCD
8027 113D                     ACALL WAIT          ;wait for LCD to be free
8029 C2B0                     CLR RS              ;output a command
802B 114B                     ACALL OUT           ;send it out

802D 7480                     MOV A, #80H         ;cursor: line 1, pos. 1
802F 113D                     ACALL WAIT          ;wait for LCD to be free
8031 C2B0                     CLR RS              ;output a command
8033 114B                     ACALL OUT           ;send it out

8035 22                       RET

                    ;*************************************************
                    ; Display data on LCD                           *
                    ;*************************************************

8036 113D           DISP:     ACALL WAIT          ;wait for LCD to be free
8038 D2B0                     SETB RS             ;output a data
803A 114B                     ACALL OUT           ;send it out
803C 22                       RET

                    ;*************************************************
                    ; Wait for LCD to be free                       *
                    ;*************************************************
803D C2B0           WAIT:     CLR RS              ;command
803F D2B1                     SETB RW             ;read
8041 D297                     SETB DBUS.7         ;DB7 = in
8043 D2B2                     SETB E              ;1-to-0 transition to
8045 C2B2                     CLR E               ; enable LCD
8047 2097F3                   JB DBUS.7, WAIT
804A 22                       RET

                    ;*************************************************
                    ; Output to LCD                                 *
                    ;*************************************************
804B F590           OUT:      MOV DBUS, A
804D C2B1                     CLR RW
804F D2B2                     SETB E
8051 C2B2                     CLR E
8053 22                       RET
                              END
```

FIGURE 11-8b
continued

TABLE 11-1
LCD command codes

Code	Description
1	Clear display screen
2	Return to home position
4	Shift cursor to left
5	Shift display right
6	Shift cursor to right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor not blinking
10	Shift cursor to left
14	Shift cursor to right
18	Shift entire display to left
1C	Shift entire display to right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines, 5 X 7 matrix display

instead. $R/\overline{W} = 1$ means that the 8051 would be reading from the LCD whereas $R/\overline{W} = 0$ means that the 8051 is writing to the LCD. Finally, E has to be set high and then brought low to signal to the LCD that its attention is required to process the command or data currently available at its data pins. Table 11-1 shows the command codes that could be issued to the LCD.

The software first calls an INIT subroutine to initialize the display modes and settings before proceeding to display on the LCD a sequence of 16 ASCII characters at a time. Notice that before sending any data to the LCD, a WAIT subroutine is called first to wait as long as the LCD is busy. This can be determined from pin 7 of the LCD's data bus, D7. Only when the LCD is no longer busy would the software call the OUT subroutine to send a byte of data or command to the LCD's data bus.

11.6 LOUDSPEAKER INTERFACE

Figure 11-9 shows an interface between an 8031 and a loudspeaker. Small loudspeakers, such as those found in personal computers or children's toys, can be driven from a single

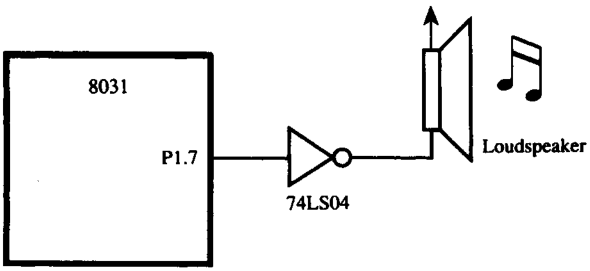


FIGURE 11-9
Interface to a loudspeaker

logic gate, as shown. One side of the loudspeaker's coil connects to +5 volts, the other to the output of a 74LS04 logic inverter. The inverter is required because it has a higher drive capability than the port lines on the 8031.

EXAMPLE 11.4 **Design Objective**

Write an interrupt-driven program that continually plays an A-major musical scale.

Musical melodies are easy to generate from an 8051, using a simple loudspeaker interface. We begin with some music theory. The frequency for each note in an A-major musical scale is given in the comment block at the top of the software listing in Figure 11-10 (lines 14-21). The first frequency is 440 Hz (called "A above middle C"), which is the international reference frequency for musical instruments using the equal-tempered scale (e.g., the piano). The frequency of all other notes can be determined by multiplying this frequency by $2^{n/12,}$ where *n* is the number of steps (or "semitones") to the note being calculated. The easiest example is A', one octave, or 12 steps, above A, which has a frequency of 440 X $2^{12/12}$ = 880 Hz. This is the last note in our musical scale. (See Figure 11-10, line 21.) With reference to the bottom note (or "root") in any major scale in steps is 2, 4, 5, 7, 9, 11, and 12. For example, the note "E" in Figure 11-10 (line 18) is seven steps above the root; thus its frequency is 440 X $2^{7/12}$ = 659.26 Hz.

To create a musical scale, two timings are required: the timing from one note to the next, and the timing for toggling the port bit that drives the loudspeaker. These two timings are vastly different. To play the melody at a rate of four notes/second, for example, a time-out (or interrupt) is needed every 250 ms. To create the frequency for the first note in the scale, a timeout is needed every 1.136 ms. (See Figure 11-10, line 14.)

The software in Figure 11-10 initializes both timers for 16-bit timer mode (line 43) and uses Timer 0 interrupts for the note changes, and Timer 1 interrupts for the frequency of notes. The reload values for the note frequencies are read from a look-up table (lines 90-104). Consult the listing in Figure 11-10 for further details.

11.7 NONVOLATILE RAM INTERFACE

Nonvolatile RAMS (NVRAMs) are semiconductor memories that maintain their contents in the absence of power. NVRAMs incorporate both standard static RAM cells and electrically erasable programmable ROM (EEPROM) cells. Each bit of the static RAM is overlaid with a bit of EEPROM. Data can be transferred back and forth between the two memories.

NVRAMs occupy an important niche in microprocessor- and microcontroller-based applications. They are used to store setup data or parameters that are changed occasionally by the user but must be retained when power is lost.

As an example, many VDT designs avoid the use of DIP switches (which are prone to failure) and use NVRAMs to store setup information such as baud rate, parity on/off, parity odd/even, and so on. Each time the VDT is turned on, these parameters are recalled from NVRAM and the system is initialized accordingly. When a parameter is changed by the user (via the keyboard), the new value is stored in NVRAM.

Modems with an auto-dial feature usually hold phone numbers in internal memory. These phone numbers are often stored in a NVRAM so they will be retained in the event of a power outage. Ten phone numbers with seven digits each can be stored in 35 bytes (by encoding each digits in BCD notation)

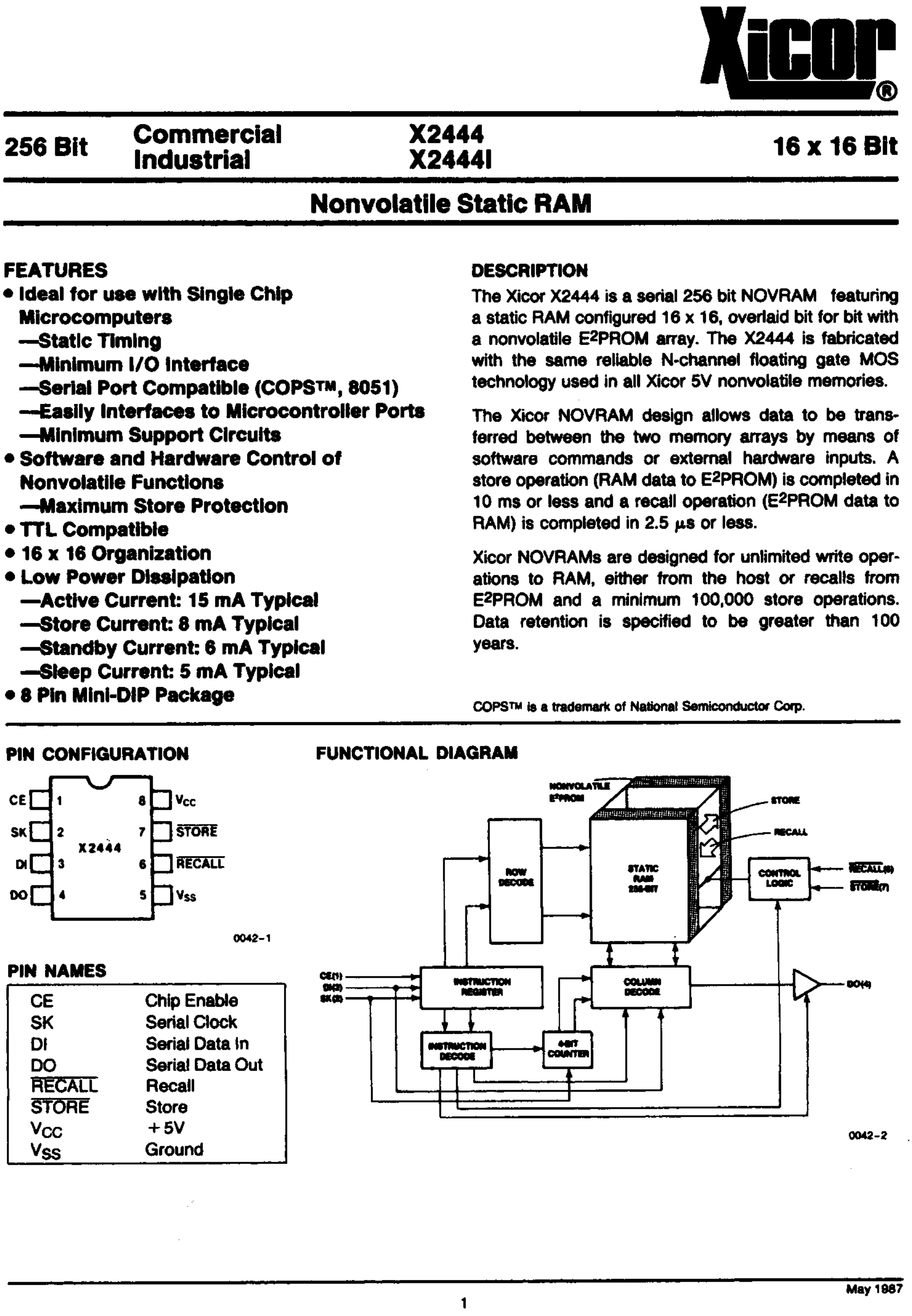
```
                 $debug
                 $nopaging
                 $nosymbols
                 ;FILE: SCALE.SRC
                 ;*******************************************************
                 ;             LOUDSPEAKER INTERFACE EXAMPLE            *
                 ;                                                      *
                 ; This program plays an A major musical scale using    *
                 ; a loudspeaker driven by a inverter through P1.7      *
                 ;*******************************************************
                 ;                                                      *
                 ;  Note  Frequency (Hz)  Period (us)  Period/2 (us)    *
                 ;  ----  --------------  -----------  -------------    *
                 ;   A       440.00          2273          1136         *
                 ;   B       493.88          2025          1012         *
                 ;   C#      554.37          1804           902         *
                 ;   D       587.33          1703           851         *
                 ;   E       659.26          1517           758         *
                 ;   F#      739.99          1351           676         *
                 ;   G#      830.61          1204           602         *
                 ;   A'      880.00          1136           568         *
                 ;*******************************************************
  00BC           MONITOR  CODE  00BCH       ;MON51 (V12) entry point
  3CB0           COUNT    EQU   -50000      ;0.05 seconds per timeout
  0005           REPEAT   EQU   5           ;5 x 0.05 = 0.25 seconds/not

                 ;*******************************************************
                 ; Note: X3 not installed on SBC-51, therefore          *
                 ; interrupts directed to the following jump table      *
                 ; beginning at 8000H                                   *
                 ;*******************************************************
8000                      ORG   8000H       ;RAM entry points for...
8000 028015               LJMP  MAIN        ; main program
8003 02806B               LJMP  EXT0ISR     ; External 0 interrupt
8006 028025               LJMP  T0ISR       ; Timer 0 interrupt
8009 02806B               LJMP  EXT1ISR     ; External 1 interrupt
800C 02803A               LJMP  T1ISR       ; Timer 1 interrupt
800F 02806B               LJMP  SPISR       ; Serial Port interrupt
8012 02806B               LJMP  T2ISR       ; Timer 2 interrupt

                 ;*******************************************************
                 ; MAIN PROGRAM BEGINS                                  *
                 ;*******************************************************
8015 758911      MAIN:    MOV   TMOD,#11H   ;both timers 16-bit mode
8018 7F00                 MOV   R7,#0       ;use R7 as note counter
801A 7E05                 MOV   R6,#REPEAT  ;use R6 as timeout counter
801C 75A88A               MOV   IE,#8AH     ;Timer 0 & 1 interrupts on
801F D28F                 SETB  TF1         ;force Timer 1 interrupt
8021 D28D                 SETB  TF0         ;force Timer 0 interrupt
8023 80FE                 SJMP  $           ;ZzZzZzZz time for a nap

                 ;*******************************************************
                 ; TIMER 0 INTERRUPT SERVICE ROUTINE (EVERY 0.05 SEC.) *
                 ;*******************************************************
8025 C28C        T0ISR:   CLR   TR0                  ;stop timer
8027 758C3C               MOV   TH0,#HIGH (COUNT)    ;reload
802A 758AB0               MOV   TL0,#LOW  (COUNT)
802D DE08                 DJNZ  R6,EXIT              ;if not 5th int, exit
802F 7E05                 MOV   R6,#REPEAT           ;if 5th, reset
8031 0F                   INC   R7                   ;increment note
8032 BF0C02               CJNE  R7,#LENGTH,EXIT      ;beyond last note?
8035 7F00                 MOV   R7,#0                ;yes: reset, A=440 Hz
8037 D28C        EXIT:    SETB  TR0                  ;no:  start timer, go
8039 32                   RETI                       ;     back to ZzZzZzZ
```

FIGURE 11-10a

Software for loudspeaker interface

```

                  ;*****************************************************
                  ; TIMER 1 INTERRUPT SERVICE ROUTINE (PITCH OF NOTES)  *
                  ;                                                   *
                  ; Note: The output frequencies are slightly off due   *
                  ; to the length of this ISR.  Timer reload values     *
                  ; need adjusting.                                     *
                  ;*****************************************************
803A B297         T1ISR:   CPL   P1.7      ;music maestro!
803C C28E                  CLR   TR1       ;stop timer
803E EF                    MOV   A,R7      ;get note counter
803F 23                    RL    A         ;multiply (2 bytes/note)
8040 128050                CALL  GETBYTE   ;get high-byte of count
8043 F58D                  MOV   TH1,A     ;put in timer high register
8045 EF                    MOV   A,R7      ;get note counter again
8046 23                    RL    A         ;align on word boundary
8047 04                    INC   A         ;past high-byte (whew!)
8048 128050                CALL  GETBYTE   ;get low-byte of count
804B F58B                  MOV   TL1,A     ;put in timer low register
804D D28E                  SETB  TR1       ;start timer
804F 32                    RETI            ;time for a rest

                  ;*****************************************************
                  ; GET A BYTE FROM LOOK-UP OF NOTES IN A MAJOR SCALE   *
                  ;*****************************************************
8050 04           GETBYTE: INC   A         ;table look-up subroutine
8051 83                    MOVC  A,@A+PC
8052 22                    RET
8053 FB90         TABLE:   DW    -1136     ;A
8055 FB90                  DW    -1136     ;A (play again; half note)
8057 FC0C                  DW    -1012     ;B (quarter note, etc.)
8059 FC7A                  DW    -902      ;C# - major third
805B FCAD                  DW    -851      ;D
805D FD0A                  DW    -758      ;E  - perfect fifth
805F FD5C                  DW    -676      ;F#
8061 FDA6                  DW    -602      ;G#
8063 FDC8                  DW    -568      ;A'
8065 FDC8                  DW    -568      ;A' (play 4 times; whole note)
8067 FDC8                  DW    -568
8069 FDC8                  DW    -568
  000C            LENGTH   EQU   ($ - TABLE) / 2 ;LENGTH = # of notes

                  ;*****************************************************
                  ; UNUSED INTERRUPTS - BACK TO MONITOR PROGRAM (ERROR) *
                  ;*****************************************************
                  EXT0ISR:
                  EXT1ISR:
                  SPISR:
806B C2AF         T2ISR:   CLR    EA       ;shut off interrupts and
806D 0200BC                LJMP   MONITOR ; return to MON51
                           END
```

FIGURE 11-10b
continued



Xicor®

256 Bit | Commercial Industrial | X2444 X2444I | 16 x 16 Bit

Nonvolatile Static RAM

FEATURES

- Ideal for use with Single Chip Microcomputers
  - —Static Timing
  - —Minimum I/O Interface
  - —Serial Port Compatible (COPS™, 8051)
  - —Easily Interfaces to Microcontroller Ports
  - —Minimum Support Circuits
- Software and Hardware Control of Nonvolatile Functions
  - —Maximum Store Protection
- TTL Compatible
- 16 x 16 Organization
- Low Power Dissipation
  - —Active Current: 15 mA Typical
  - —Store Current: 8 mA Typical
  - —Standby Current: 6 mA Typical
  - —Sleep Current: 5 mA Typical
- 8 Pin Mini-DIP Package

DESCRIPTION

The Xicor X2444 is a serial 256 bit NOVRAM featuring a static RAM configured 16 x 16, overlaid bit for bit with a nonvolatile E^{2}PROM array. The X2444 is fabricated with the same reliable N-channel floating gate MOS technology used in all Xicor 5V nonvolatile memories.

The Xicor NOVRAM design allows data to be transferred between the two memory arrays by means of software commands or external hardware inputs. A store operation (RAM data to E^{2}PROM) is completed in 10 ms or less and a recall operation (E^{2}PROM data to RAM) is completed in 2.5 μs or less.

Xicor NOVRAMs are designed for unlimited write operations to RAM, either from the host or recalls from E^{2}PROM and a minimum 100,000 store operations. Data retention is specified to be greater than 100 years.

COPS™ is a trademark of National Semiconductor Corp.

PIN CONFIGURATION

0042-1

PIN NAMES

CE	Chip Enable
SK	Serial Clock
DI	Serial Data In
DO	Serial Data Out
$\overline{\text{RECALL}}$	Recall
$\overline{\text{STORE}}$	Store
V$_{CC}$	+5V
V$_{SS}$	Ground

FUNCTIONAL DIAGRAM

0042-2

May 1987

1

FIGURE 11-11
Cover page for the X2444 nonvolatile RAM data sheet

The NVRAM used for this interface example is an X2444 manufactured by Xicor,[1] a company that specializes in NVRAMs and EEPROMs. The X2444 contains 256 bits of static RAM overlaid by 256 bits of EEPROM. Data can be transferred back and forth between the two memories either by instructions sent from the processor over the serial interface or by toggling the external STORE and RECALL inputs. Nonvolatile data are retained in the EEPROM, while independent data are accessed and updated in the RAM. The X2444 features are summarized in the first page of its data sheet, reproduced in Figure 11-11.

In this interface example, the STORE and RECALL lines are not used. The various modes of operation are entered by sending the X2444 serial instructions through 8051 ports pins. The interface to the 8051 is shown in Figure 11-12. Only three lines are used:

- P1.0-SK (serial clock)
- P1.1-CE (chip enable)
- P1.2-DI/DO (data input/output)

Instructions are sent to the X2444 by bringing CE high and then clocking an 8-bit op-code into the X2444 via the SK and DI/DO lines. The following opcodes are required for this example:

Instruction	Opcode	Operation
RCL	85H	Recall EEPROM data into RAM
WREN	84H	Set write enable latch
STORE	81H	Store RAM data into EEPROM
WRITE	1AAAA011B	Write data into RAM address AAAA
READ	1AAAA111B	Read data from RAM address AAAA

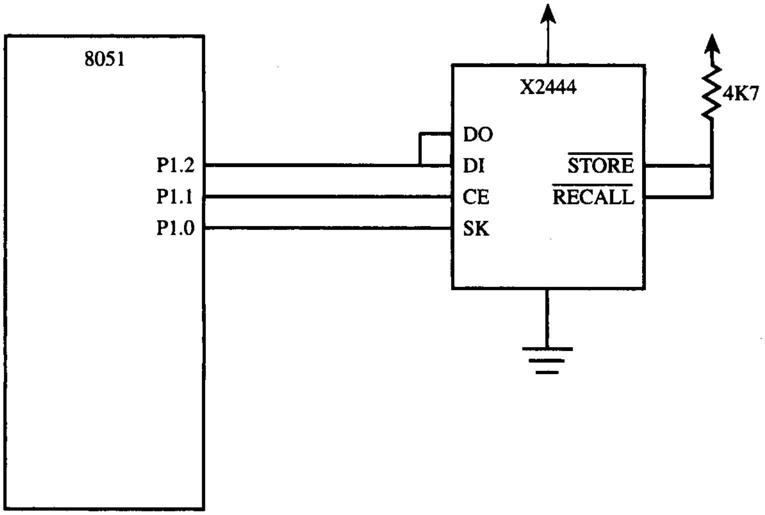


FIGURE 11-12
Interface to X2444 nonvolatile RAM

[1] XICOR, Inc., 851 Buckeye Court, Milpitas, CA 95035.

EXAMPLE 11.5 **Design Objective**

Write the following two programs. The first, called SAVE, copies the contents of 8051 internal locations 60H-7FH to the X2444 EEPROM. The second, called RECOVER, reads previously saved data from the X2444 EEPROM and restores it to locations 60H-7FH. These are two distinct programs. Typically the SAVE program is executed when ever nonvolatile information is changed (for example, by a user altering a configuration parameter). The RECOVER program is executed each time the system is powered up or reset. For this example, the nonvolatile information is kept in the 8051 internal locations 60H-7FH (presumably for access by a control program executing in firmware). The software listing is shown in Figure 11-13.

The operations of saving and recovering data involve the following steps:

Write Data into the X2444

1. **Execute RCL (recall) instruction.**
2. **Execute WREN (set write enable latch) instruction.**
3. **Write data into X2444 RAN.**
4. **Execute STO (store RAM into EEPROM) instruction.**
5. **Execute SLEEP instruction.**

Read Data from X2444

1. **Execute RCL (recall) instruction.**
2. **Read data from X2444 RAM.**
3. **Execute SLEEP instruction.**

As an example of what the software drivers must do, Figure 11-14 illustrates the timing diagram to send the RCL instruction to the X2444. Several of the bits are actually "don't cares" (as specified in the data sheet); however, they are shown as 0s in the figure.

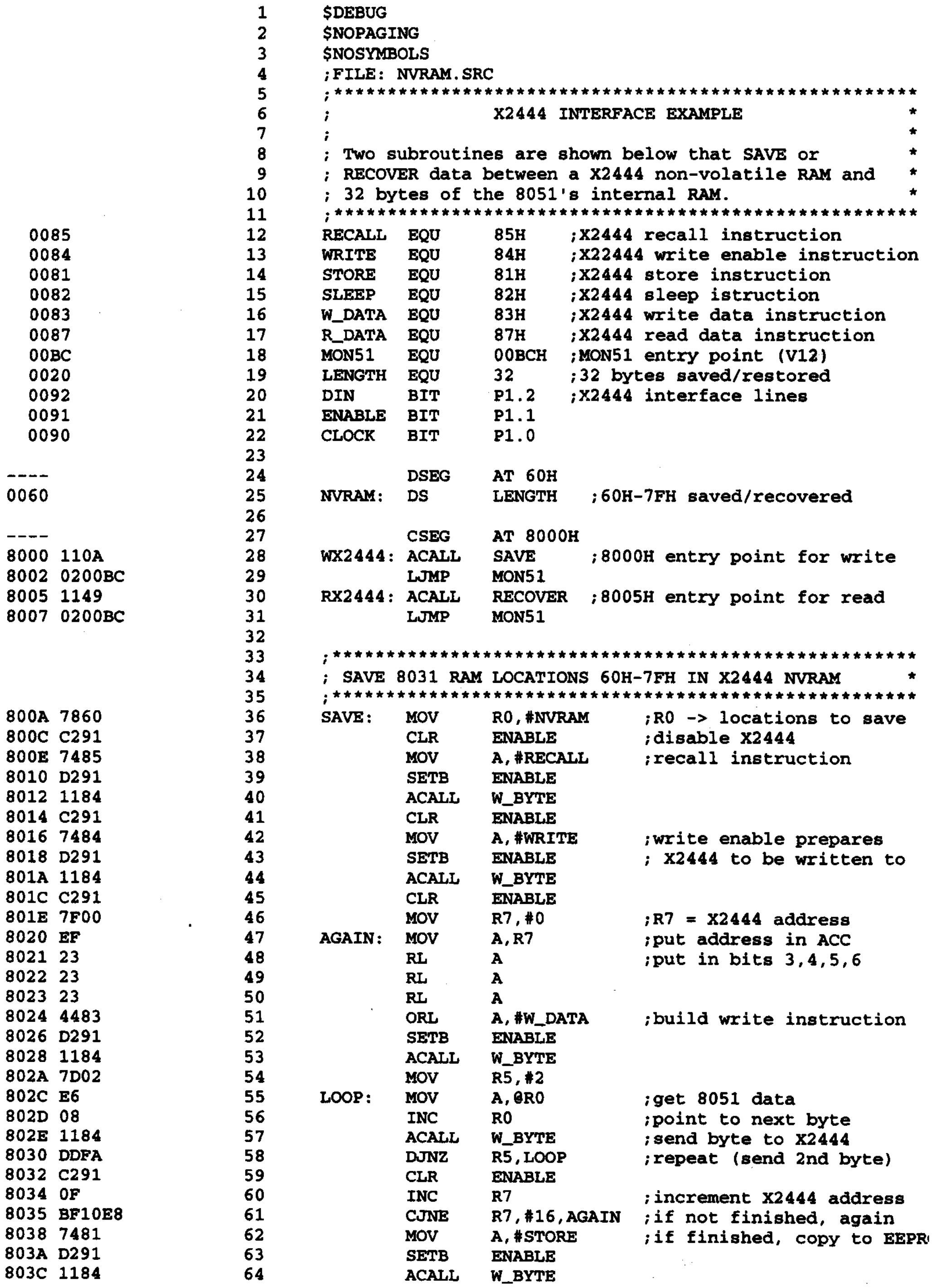
The timing for the WRITE data and READ data instructions is slightly different. For these, the 8-bit opcode is followed immediately by 16 bits of data, and chip enable remains high for all 24 bits. For the read instruction, the eight bits (the opcode) are written to the X2444, then 16 data bits are read from the X2444. Separate subroutines are used for reading eight bits (R_BYTE; lines 106-112) and writing eight bits (W_BYTE; lines 117-123). For specific details, consult the software listing.

11.8 INPUT/OUTPUT EXPANSION

The 8051 has four input/output ports, Port 0 to Port 3. If external memory is used, then all or part of Port 0 and Port 2 would be taken up as data and/or address lines, thereby reducing the number of available I/O lines for general purpose I/O usage. In this section, we will discuss two simple ways to increase the number of I/O lines. This is called expanding the I/O.

11.8.1 Using Shift Registers

Our next example illustrates a simple way to increase the number of input lines on the 8051. Three port lines are used to interface to multiple (in this example, 2) 74HC165 parallel-in serial-out shift registers. (See Figure 11-15.) The additional inputs are sampled



```
              $DEBUG
              $NOPAGING
              $NOSYMBOLS
              ;FILE: NVRAM.SRC
              ;*****************************************************
              ;                X2444 INTERFACE EXAMPLE              *
              ;                                                     *
              ; Two subroutines are shown below that SAVE or        *
              ; RECOVER data between a X2444 non-volatile RAM and   *
              ; 32 bytes of the 8051's internal RAM.                *
              ;*****************************************************
  0085        RECALL  EQU     85H     ;X2444 recall instruction
  0084        WRITE   EQU     84H     ;X22444 write enable instruction
  0081        STORE   EQU     81H     ;X2444 store instruction
  0082        SLEEP   EQU     82H     ;X2444 sleep istruction
  0083        W_DATA  EQU     83H     ;X2444 write data instruction
  0087        R_DATA  EQU     87H     ;X2444 read data instruction
  00BC        MON51   EQU     00BCH   ;MON51 entry point (V12)
  0020        LENGTH  EQU     32      ;32 bytes saved/restored
  0092        DIN     BIT     P1.2    ;X2444 interface lines
  0091        ENABLE  BIT     P1.1
  0090        CLOCK   BIT     P1.0

----                  DSEG    AT 60H
0060          NVRAM:  DS      LENGTH   ;60H-7FH saved/recovered

----                  CSEG    AT 8000H
8000 110A     WX2444: ACALL   SAVE     ;8000H entry point for write
8002 0200BC           LJMP    MON51
8005 1149     RX2444: ACALL   RECOVER  ;8005H entry point for read
8007 0200BC           LJMP    MON51

              ;*****************************************************
              ; SAVE 8031 RAM LOCATIONS 60H-7FH IN X2444 NVRAM      *
              ;*****************************************************
800A 7860     SAVE:   MOV     R0,#NVRAM      ;R0 -> locations to save
800C C291             CLR     ENABLE         ;disable X2444
800E 7485             MOV     A,#RECALL      ;recall instruction
8010 D291             SETB    ENABLE
8012 1184             ACALL   W_BYTE
8014 C291             CLR     ENABLE
8016 7484             MOV     A,#WRITE       ;write enable prepares
8018 D291             SETB    ENABLE         ; X2444 to be written to
801A 1184             ACALL   W_BYTE
801C C291             CLR     ENABLE
801E 7F00             MOV     R7,#0          ;R7 = X2444 address
8020 EF       AGAIN:  MOV     A,R7           ;put address in ACC
8021 23               RL      A              ;put in bits 3,4,5,6
8022 23               RL      A
8023 23               RL      A
8024 4483             ORL     A,#W_DATA      ;build write instruction
8026 D291             SETB    ENABLE
8028 1184             ACALL   W_BYTE
802A 7D02             MOV     R5,#2
802C E6       LOOP:   MOV     A,@R0          ;get 8051 data
802D 08               INC     R0             ;point to next byte
802E 1184             ACALL   W_BYTE         ;send byte to X2444
8030 DDFA             DJNZ    R5,LOOP        ;repeat (send 2nd byte)
8032 C291             CLR     ENABLE
8034 0F               INC     R7             ;increment X2444 address
8035 BF10E8           CJNE    R7,#16,AGAIN   ;if not finished, again
8038 7481             MOV     A,#STORE       ;if finished, copy to EEPR
803A D291             SETB    ENABLE
803C 1184             ACALL   W_BYTE
```

FIGURE 11-13a
Software for X2444 interface

```
803E C291                CLR     ENABLE
8040 7482                MOV     A,#SLEEP        ;put X2444 to sleep
8042 D291                SETB    ENABLE
8044 1184                ACALL   W_BYTE
8046 C291                CLR     ENABLE
8048 22                  RET                     ;DONE!

               ;*****************************************************
               ; RECOVER 8051 RAM LOCATIONS 60H-7FH FROM X2444 NVRAM *
               ;*****************************************************
8049 7860      RECOVER:  MOV     R0,#NVRAM
804B C291                CLR     ENABLE
804D 7485                MOV     A,#RECALL         ;recall instruction
804F D291                SETB    ENABLE
8051 1184                ACALL   W_BYTE
8053 C291                CLR     ENABLE
8055 7F00                MOV     R7,#0             ;R7 = X2444 address
8057 EF        AGAIN2:   MOV     A,R7              ;put address in ACC
8058 23                  RL      A                 ;build read instruction
8059 23                  RL      A
805A 23                  RL      A
805B 4487                ORL     A,#R_DATA
805D D291                SETB    ENABLE
805F 1184                ACALL   W_BYTE            ;send read instruction
8061 7D02                MOV     R5,#2             ; (+ address)
8063 1178      LOOP2:    ACALL   R_BYTE            ;read byte of data
8065 F6                  MOV     @R0,A             ;put in 8051 RAM
8066 08                  INC     R0                ;point to next location
8067 DDFA                DJNZ    R5,LOOP2
8069 C291                CLR     ENABLE
806B 0F                  INC     R7                ;increment X2444 address
806C BF10E8              CJNE    R7,#16,AGAIN2     ;repeat until last
806F 7482                MOV     A,#SLEEP          ;put X2444 to sleep
8071 D291                SETB    ENABLE
8073 1184                ACALL   W_BYTE
8075 C291                CLR     ENABLE
8077 22                  RET                       ;DONE!

               ;*****************************************************
               ; READ A BYTE OF DATA FROM X2444                      *
               ;*****************************************************
8078 7E08      R_BYTE:   MOV     R6,#8      ;use R6 as bit counter
807A A292      AGAIN3:   MOV     C,DIN      ;put X2444 data bit in C
807C 33                  RLC     A          ;build byte in Accumulator
807D D290                SETB    CLOCK      ;toggle clock line (1 us)
807F C290                CLR     CLOCK
8081 DEF7                DJNZ    R6,AGAIN3  ;if not last bit, do again
8083 22                  RET

               ;*****************************************************
               ; WRITE A BYTE OF DATA TO X2444                       *
               ;*****************************************************
8084 7E08      W_BYTE:   MOV     R6,#8      ;use R6 as bit counter
8086 33        AGAIN4:   RLC     A          ;put bit to write in C
8087 9292                MOV     DIN,C      ;put in X2444 DATA IN line
8089 D290                SETB    CLOCK      ;clock bit into X2444
808B C290                CLR     CLOCK
808D DEF7                DJNZ    R6,AGAIN4  ;if not last bit, do again
808F 22                  RET
                         END
```

FIGURE 11-13b
continued

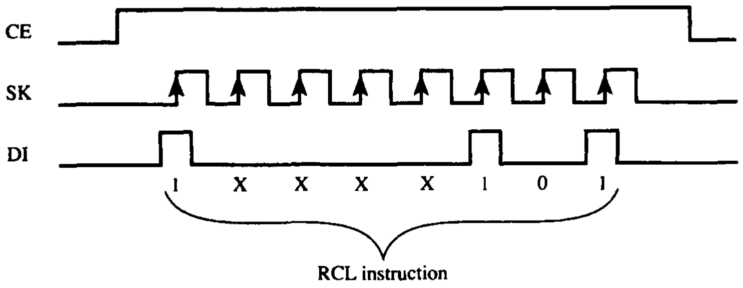


FIGURE 11-14
Timing for the X2444 recall instruction

periodically by pulsing the SHIFT/$\overline{\text{LOAD}}$ line low. The data are then read into the 8051 by reading the DATA IN line and pulsing the CLOCK line. Each pulse on the clock line shifts the data ("down," as shown in Figure 11-15), so the next read to DATA IN reads the next bit, and so on.

EXAMPLE 11.6

Design Objective

Write a subroutine that copies the state of the 16 inputs in Figure 11-15 to 8051 internal RAM locations 25H and 26H.

The software to accomplish this is shown in Figure 11-16. Note that the main program loop consists of calls to two subroutines: GET_BYTES and DISPLAY_RESULTS (lines 34-35). The latter subroutine is included to illustrate a useful technique for debugging when resources are limited. DISPLAY_RESULTS (lines 72-83) reads the data from internal locations 25H and 26H and sends each nibble to the console as a hexadecimal character. This provides a simple visual interface to verify if the program and interface are working. As input lines are toggled high and low, changes will immediately appear on the console (if the interface and program are working properly).

The GET_BYTES subroutine (lines 44-58) takes 112 μs to execute when two 74HC165s are used and the system operates from a 12 MHz crystal. If the inputs were sampled, for example, 20 times per second, GET_BYTES would consume 112 ÷ 50,000 = 0.2% of the CPU's execution time. This would have a minimal impact on the system's resources; however, increasing the number of input lines and/or the sampling rate may start to impact overall system performance. Consult the software listing for further details.

11.8.2 Using the 8255

The 8255 is a programmable peripheral interface (PPI) IC that can be used to expand the 805l's I/O. The 8255 is a 40-pin IC chip with three 8-bit ports known as Port A, Port B, and Port C. There are also two active-low input control signals, RD and WR, used to connect to their 8051 counterparts. Meanwhile, pins D0 to D7 are the data pins used to connect to the data bus of the 8051.

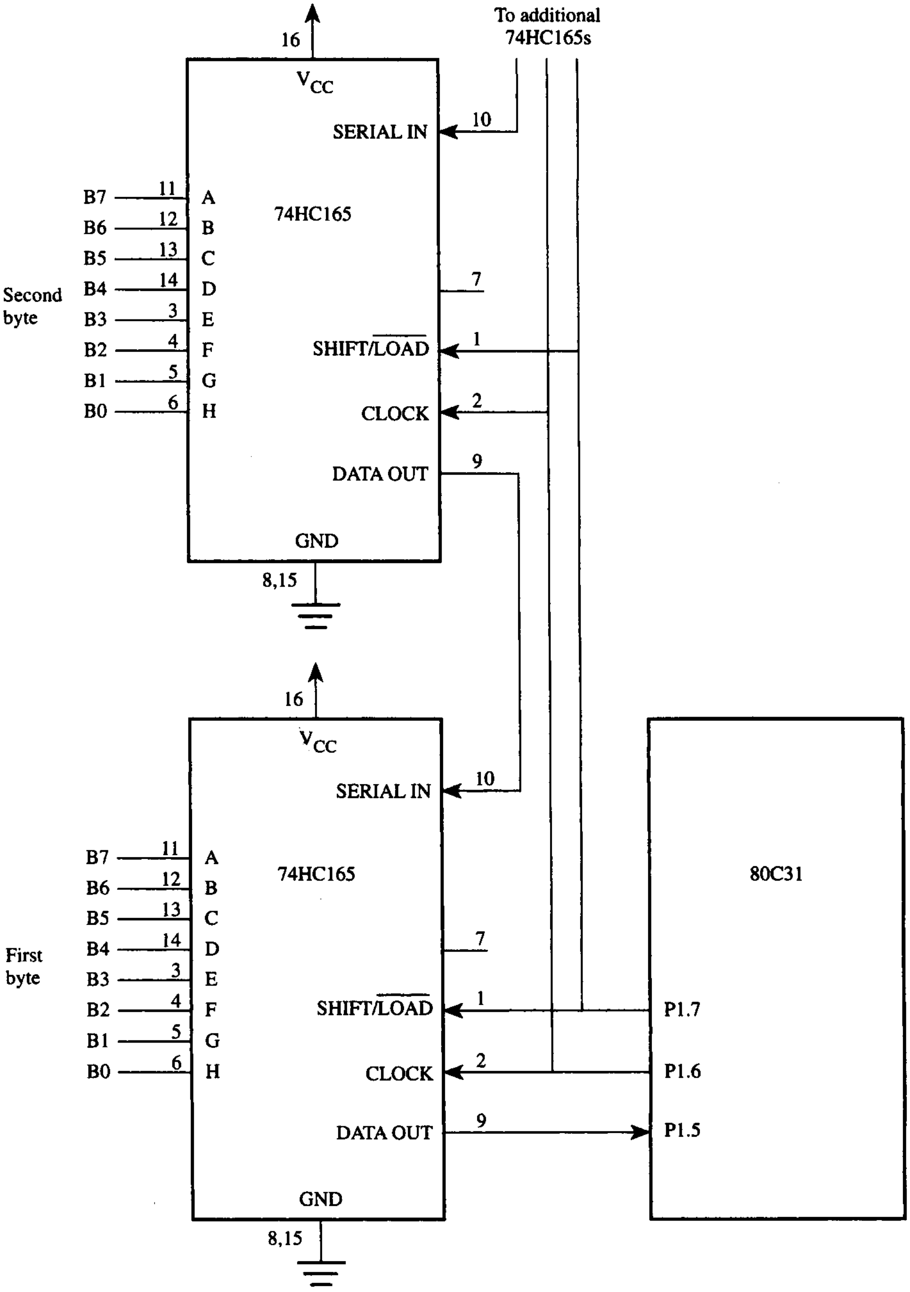


FIGURE 11-15
Interface to two 74HC165s

```
                  $DEBUG
                  $NOSYMBOLS
                  $NOPAGING
                  ;FILE: HC165.SRC
                  ;*****************************************************
                  ;               74HC165 INTERFACE EXAMPLE            *
                  ;                                                    *
                  ; The subroutine GET_BYTES below reads multiple (in  *
                  ; this case two) 74HC165 parallel-in serial-out shift *
                  ; registers attached to P1.7 (SHIFT/LOAD), P1.6      *
                  ; (CLOCK), and P1.5 (DATA OUT).  The bytes read are  *
                  ; placed in bit-addressable locations starting at the *
                  ; byte address BUFFER.                               *
                  ;*****************************************************
 000D             CR      EQU     0DH
 0002             COUNT   EQU     2       ;number of 74HC165s
 0097             SHIFT   BIT     P1.7    ;74HC165 SHIFT/LOAD input
                                          ; 1 = shift, 0 = load
 0096             CLOCK   BIT     P1.6    ;74HC165 CLOCK input
 0095             DOUT    BIT     P1.5    ;74HC165 DATA OUT output
 0282             OUTSTR  CODE    0282H   ;subroutines in MON51(V12)
 028D             OUT2HEX CODE    028DH   ;output byte as two hex char.
 01DE             OUTCHR  CODE    01DEH

8000                      ORG     8000H   ;begin code segment at 8000H
8000 D296                 SETB    CLOCK   ;set interface lines initially ir
8002 D297                 SETB    SHIFT   ; case not already
8004 D295                 SETB    DOUT    ;DOUT must be set (input)

                  ;*****************************************************
                  ; MAIN LOOP (KEPT SMALL FOR THIS EXAMPLE)            *
                  ;*****************************************************
8006 128029               CALL    SEND_HELLO_MESSAGE ;banner message
8009 128011       REPEAT: CALL    GET_BYTES          ;read 74HC165s
800C 128050               CALL    DISPLAY_RESULTS    ;show results
800F 80F8                 JMP     REPEAT             ;loop

                  ;*****************************************************
                  ; GET BYTES FROM 74HC165s & PLACE IN INTERNAL RAM    *
                  ;                                                    *
                  ; Execution time = 112 microseconds (@ 12 MHz).      *
                  ; Execution time for N 74HC165s = 6 + (N x 53) us    *
                  ;*****************************************************
                  GET_BYTES:
8011 7E02                 MOV     R6,#COUNT   ;use R6 as byte counter
8013 7825                 MOV     R0,#BUFFER  ;use R0 as pointer to buffer
8015 C297                 CLR     SHIFT       ;load into 74HC165s by
8017 D297                 SETB    SHIFT       ; pulsing SHIFT/LOAD low
8019 7F08         AGAIN:  MOV     R7,#8       ;use R7 as bit counter
801B A295         LOOP:   MOV     C,DOUT      ;get a bit (put it in C)
801D 13                   RRC     A           ;put in ACC.0 (LSB 1st)
801E C296                 CLR     CLOCK       ;pulse CLOCK line (shifts
8020 D296                 SETB    CLOCK       ; bits toward DATA OUT)
8022 DFF7                 DJNZ    R7,LOOP     ;if not 8th shift, repeat
8024 F6                   MOV     @R0,A       ;if 8th shift, put in buf.
8025 08                   INC     R0          ;increment pointer to buf.
8026 DEF1                 DJNZ    R6,AGAIN    ;get two bytes
8028 22                   RET

                  ;*****************************************************
                  ; SEND HELLO MESSAGE TO CONSOLE (DEBUGGING AID)      *
                  ;*****************************************************
                  SEND_HELLO_MESSAGE:
8029 908030               MOV     DPTR,#BANNER    ;point to hello message
```

FIGURE 11-16a
Software for 75HC165 interface

```
802C 120282                  CALL    OUTSTR          ;send it to console
802F 22                      RET
8030 2A2A2A20        BANNER: DB      '*** TEST 74HC165 INTERFACE ***',CR,0
8034 54455354
8038 20373448
803C 43313635
8040 20494E54
8044 45524641
8048 4345202A
804C 2A2A
804E 0D
804F 00

                     ;*******************************************************
                     ; DISPLAY RESULTS ON CONSOLE (DEBUGGING AID)           *
                     ;*******************************************************
                     DISPLAY_RESULTS:                ;display bytes
8050 7825                    MOV     R0,#BUFFER      ;R0 points to bytes
8052 7E02                    MOV     R6,#COUNT       ;R6 is # of bytes read
8054 E6              LOOP2:  MOV     A,@R0           ;get byte
8055 08                      INC     R0              ;increment pointer
8056 12028D                  CALL    OUT2HEX         ;output as 2 hex char.
8059 7420                    MOV     A,#' '          ;separate bytes
805B 1201DE                  CALL    OUTCHR
805E DEF4                    DJNZ    R6,LOOP2        ;repeat for each byte
8060 740D                    MOV     A,#CR           ;begin a new line
8062 1201DE                  CALL    OUTCHR          ;send CR (LF too!)
8065 22                      RET

                     ;*******************************************************
                     ; CREATE BUFFER IN BIT-ADDRESSABLE INTERNAL RAM        *
                     ;*******************************************************
----                         DSEG    AT 25H  ;on-chip data segment in
0025                 BUFFER: DS      COUNT   ; in bit-addressable space
                             END
```

FIGURE 11-16b
continued

Two input control pins. A0 and Al, are used to select specific ports within the 8255 (See Table 11-2.) However, note that if both A0 and Al are set, this does not select a port but instead selects the control register. The control register is an 8-bit register (See Table 11-3) within the 8255 that is used to specify the mode of operation for all the three ports. This is similar to the 805l's TMOD and SCON registers, which are used to set the operating modes of its timers and serial port, respectively.

The 8255 ports can be directed to operate in any one of four modes, but here we will just use the simplest mode, mode 0 (Basic I/0). This mode sets the ports to provide simple input and output operations similar to the normal 8051 ports 0 to 3. Other more complex modes allow for handshaking, which basically allows two devices (in this case the 8051 and the I/O device) to communicate more intelligently with each other through a series of handshaking signals.

TABLE 11-2
8255 port selection

A1	A0	Selection
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control Register

TABLE 11-3
8255 control register summary

Bit	Group	Description
D7	A	1 = I/O mode 0 = BSR mode
D6	A	Mode selection bit 1
D5	A	Mode selection bit 0 00 = mode 0 01 = mode 1 10 = mode 2 11 = mode 3
D4	A	Port A. 1 = input 0 = output
D3	A	Port C (upper PC7-PC4) 1 = input 0 = output
D2	B	Mode selection bit 0 = mode 0 1 = mode 1
D1	B	Port B 1 = input 0 = output
D0	B	Port C (lower PC3-PC0) 1 = input 0 = output

EXAMPLE 11.7

Design Objective

Write a program that reads the status of the eight switches in Figure 11-17 and for each closed switch, lights the corresponding LED.

Notice that in Figure 11-17, the 8255 has been connected to the 8051 as if it is an external memory device. This concept of connecting I/O devices (in this case the 8255) as memory is called *memory-mapping.* This enables the 8051 to address the ports and control register of the 8A05 as external memory locations. The least two significant bits of the address bus, Al and A0, have been directly connected to the Al and A0 inputs of the 8255; hence, they would be used to select a specific 8255 port or the control register. Meanwhile, address line A8 has been connected to the chip select ($\overline{CS}$) input of the 8255, so an address of the form:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
x	x	x	x	x	x	x	1	x	x	x	x	x	x	A1	A0

would select the 8255 with the Al and A0 bits selecting a specific port or the control register within the 8255. Assuming all the x's as 0, then the addresses 0100H, 0101H, and 0102H would select Ports A, B, and C, respectively, selects the address 0103H selects the control register.

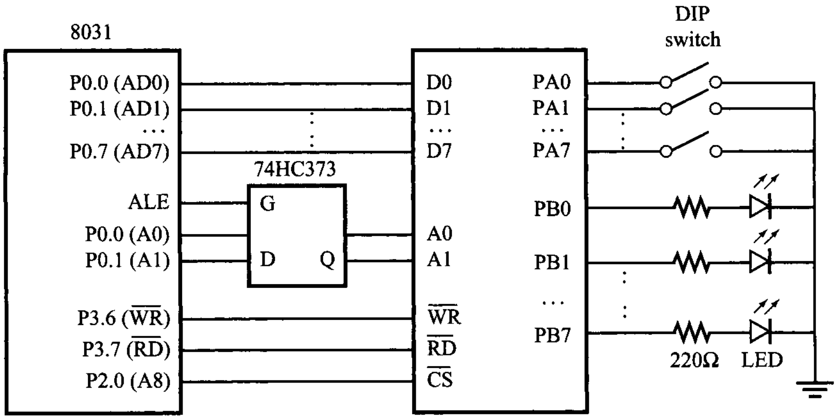


FIGURE 11-17
Interface to an 8255

```
               $DEBUG
               $NOSYMBOLS
               $NOPAGING
               ;FILE: 8255.SRC
               *************************************************
               ;            8255 PPI INTERFACE EXAMPLE          *
               ;                                                *
               ; This program uses the 8255 PPI to connect to   *
               ; 8 switches and 8 corresponding LEDs.  Using    *
               ; the 8255 gives additional 8 I/O pins for I/O   *
               ; usage.                                         *
               ;*************************************************

8000                   ORG 8000H
8000 7490              MOV A, #10010000B   ;Port A=in, Port B=out
8002 900103            MOV DPTR, #0103H    ;point to control reg
8005 F0                MOVX @DPTR, A       ;send control word
8006 900100    LOOP:   MOV DPTR, #0100H    ;point to Port A
8009 E0                MOVX A, @DPTR       ;read switches
800A F4                CPL A               ;complement
800B 900101            MOV DPTR, #0101H    ;point to Port B
800E F0                MOVX @DPTR, A       ;light up LEDs
800F 80F5              SJMP LOOP           ;loop
                       END
```

FIGURE 11-18
Software for 8255 interface

This is a very simple interface example, and the software to do this is given in Figure 11-18. The first step is to select the control register in order to set the mode of operation to 0. This is followed by sending out the appropriate control word to initialize Port A as input and Port B as output. Observe that the switches connected to Port A are in direct correspondence to the LEDs connected to Port B. For example, a closed switch at pin 0 of Port A would give a low, signaling that we should light the corresponding LED at pin 0 of Port B by sending a high to that pin. Hence, Port A is first selected and its contents read into the accumulator, indicating the current status of the switches. Then, all we need to do is to complement the contents of the accumulator, select Port B, and send the contents of the accumulator out to it.

11.9 RS232 (EIA-232) SERIAL INTERFACE

We have learn that the 8051 consists of a built-in serial port for interfacing to serial I/O devices. In fact, we could also connect the 8051 to the serial port of a personal computer (PC). The PC's serial port follows the RS232 or EIA-232 serial interface standard and for this reason, a normal RS232 cable can be used to connect between a PC and the 8051. The RS232 cable is terminated at both ends by a connector (called DB-25) with 25 pins. However, since not all of these pins are used in most PC applications, there is also a different connector version called the DB-9 with only nine more commonly used pins. Whether it is DB-25 or DB-9, the three most important pins are the receive data (RXD), transmit data (TXD), and ground (GND). The RS232 serial interface also allows for handshaking, where in order to establish a communications channel, one device initiates by sending the request to send (RTS) signal to another device and waits for the corresponding clear to send (CTS) signal to be returned. Upon reception of CTS, the two devices can then communicate messages back and forth.

When connecting the 8051 to the RS232 serial interface, one major concern is the difference in voltage levels between them. The 8051 uses the TTL voltage levels where a 5V would indicate a high while a DV indicates a low. Meanwhile, for the RS232, a high is defined as being +3V to +15V while a low is between -5V to -15V. Because of this difference, connections between the 8051 and the RS232 have to be done through line drivers. Line drivers basically function to convert between the two different voltage levels so that a high or low as understood by the 8051 also means a high or low to the RS232 and vice versa. Figure 11-19 shows how the 8051 is connected to the RS232 serial interface via RS232 line drivers such as the 1488/1489.

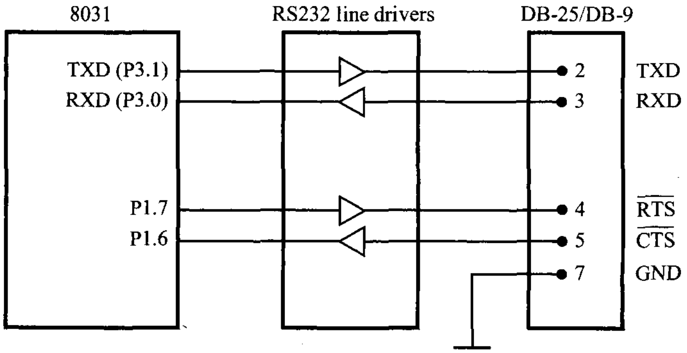


FIGURE 11-19
Interface to RS232

EXAMPLE 11.8 Design Objective

The 8051 is connected to the PC through the RS232 serial interface. Write a program that inputs decimal numbers from the PC attached to the 8051 serial port, and sends the corresponding ASCII code out to the screen.

The software for this is shown in Figure 11-20. First, the FACE subroutine is called for initializing the serial port and to perform the handshaking with the serial interface before an initial message is transmitted. For this purpose, the serial port is set for mode 1, 8-bit UART and the corresponding baud rate to 9600 baud. The RTS signal is then asserted and the 8051 waits for the acknowledging CTS signal from the serial interface. Only when CTS is detected will an initial message be sent out through the serial port.

```
                $DEBUG
                $NOSYMBOLS
                $NOPAGING
                ;FILE: SERIAL.SRC
                ;***********************************************
                ;         RS232 SERIAL INTERFACE EXAMPLE      *
                ;                                             *
                ; This program obtains a decimal number from  *
                ; the PC through the RS232 serial interface   *
                ; and outputs the corresponding ASCII code to *
                ; the screen.                                 *
                ;***********************************************

  020E          INCHAR  EQU     020EH
  01DE          OUTCHR  EQU     01DEH
  0097          RTS     EQU     P1.7
  0096          CTS     EQU     P1.6

8000                    ORG 8000H
8000 12800F             CALL FACE          ;initial & handshake
8003 908043             MOV DPTR, #ASC     ;point to ASCII table
8006 12020E     LOOP:   CALL INCHAR        ;get decimal number
8009 93                 MOVC A, @A+DPTR    ;convert to ASCII
800A 1201DE             CALL OUTCHR        ;output ASCII
800D 80F7               SJMP LOOP          ;loop

                ;***********************************************
                ; Subroutine for initialization & handshaking  *
                ;***********************************************

800F 758920     FACE:   MOV TMOD, #20H  ;timer 1, mode 2
```

FIGURE 11-20a
Software for RS232 interface

```
8012 758D98              MOV TH1, #98H   ;reload count for
                                          9600baud
8015 759852              MOV SCON, #52H  ;serial port, mode 1
8018 90802C              MOV DPTR, #MSG  ;pointer to initial msg
801B D28E                SETB TR1        ;start timer 1

801D C297                CLR RTS         ;assert RTS (handshaking)
801F 2096FD              JB CTS, $       ;wait for CTS

8022 93         OUT:     MOVC A, @A+DPTR ;get characters
8023 1201DE              CALL OUTCHR     ;send out
8026 6003                JZ STOP         ;if end of message, stop
8028 A3                  INC DPTR        ;else, get next character
8029 80F7                SJMP OUT        ;loop
802B 22         STOP:    RET

                ;*********************************************
                ; Initial message and ASCII codes for numbers  *
                ;*********************************************

                ;initial message
802C 504C4541   MSG:     DB 'PLEASE ENTER NUMBER = ', 00H
8030 53452045
8034 4E544552
8038 204E554D
803C 42455220
8040 3D20
8042 00

                ;ASCII codes for decimal numbers
8043 30        ASC:     DB 30H  ;0
8044 31                  DB 31H  ;1
8045 32                  DB 32H  ;2
8046 33                  DB 33H  ;3
8047 34                  DB 34H  ;4
8048 35                  DB 35H  ;5
8049 36                  DB 36H  ;6
804A 37                  DB 37H  ;7
804B 38                  DB 38H  ;8
804C 39                  DB 39H  ;9
                         END
```

FIGURE 11-20b
continued

The program then calls INCHAR to wait for decimal numbers from the serial interface. Once an incoming number is detected, the corresponding ASCII code is retrieved from a lookup table and subsequently sent out through the serial port by calling the OUTCHR subroutine.

TABLE 11-4
DB-25 Centronics parallel interface pins

Pin	Description
1	$\overline{\text{STROBE}}$
2	DATA 0
3	DATA 1
4	DATA 2
5	DATA 3
6	DATA 4
7	DATA 5
8	DATA 6
9	DATA 7
10	$\overline{\text{ACK}}$ (Acknowledge)
11	BUSY
12	PAPER END
13	SLCT (Select)
14	$\overline{\text{AUTOFEED}}$
15	$\overline{\text{ERROR}}$
16	INITIALIZE PRINTER
17	$\overline{\text{SLCTIN}}$ (Select in)
18−25	GND (Ground)

11.10 CENTRONICS PARALLEL INTERFACE

The most common printer interface is the Centronics parallel interface. This has become the de facto standard for parallel interface, just like its RS232 counterpart in the previous section. The Centronics parallel interface specifies 36 signals. This interface can either be terminated by a 36-pin Centronics connector, or even a DB-25 connector, as is more common on most PCs nowadays. (See Table 11-4.) In this section, we will see how to connect the 8051 to a printer through the DB-25 Centronics parallel interface.

The Centronics parallel interface consists of eight data pins to carry a byte of data for printing. When a byte of data has been made available at these data pins, the 8051 sends a STROBE signal that allows the 8051 to inform the printer that data is available at its data pins. If the printer is not busy, it would send a corresponding ACK signal to indicate that it has successfully processed the print request. Otherwise, a BUSY signal would be sent instead.

EXAMPLE 11.9

Design Objective

Write a program to query the printer and then continually send a test message to it for printing. Figure 11-21 shows the connection of the 8051 to a printer via the Centronics parallel interface. The corresponding software is given in Figure 11-22. The program first activates the STROBE signal to the printer and then checks the status of the printer to ensure that it is not busy (BUSY = 0), that there is still paper (PAPER END = 0), that it has been selected (SLOT = 1), and that there is no error (ERROR = 1). If any of the conditions are

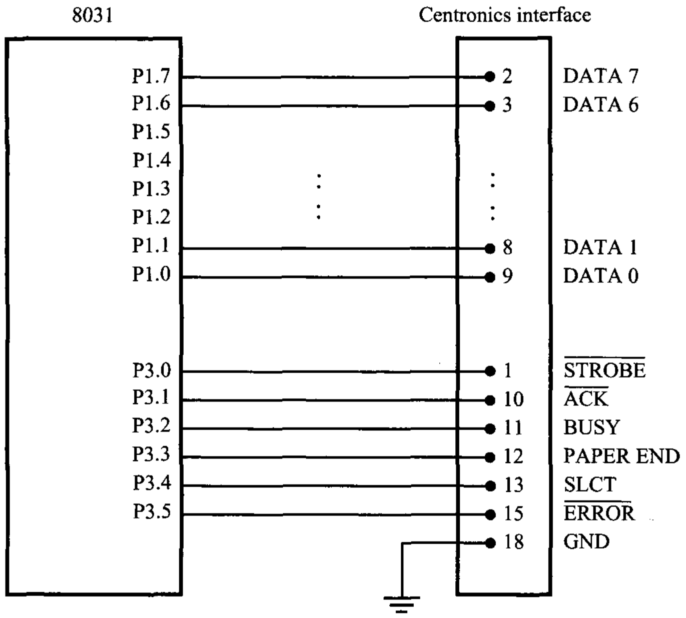


FIGURE 11-21
Interface to Centronics parallel interface

```
                    $DEBUG
                    $NOSYMBOLS
                    $NOPAGING
                    ;FILE: PARALLEL.SRC
                    ;*************************************************
                    ;      CENTRONICS PARALLEL INTERFACE EXAMPLE     *
                    ;                                                *
                    ; This program continually sends a test message *
                    ; to the printer.                                *
                    ;*************************************************

 0090               DAT     EQU   P1
 00B0               STR     EQU   P3.0
 00B1               ACK     EQU   P3.1
 00B2               BUSY    EQU   P3.2
 003C               MASK    EQU   00111100B
 0030               OK      EQU   00110000B

8000                        ORG 8000H
8000 90801C         STRT:   MOV DPTR, #MSG  ;point to test message
8003 75B03C         LOOP:   MOV P3, #MASK   ;activate STROBE, and
                                            ; P3.1-P3.5 = input
```

FIGURE 11-22a
Software for Centronics parallel interface

```
8006 E5B0                          MOV A, P3        ;read printer status
8008 543C                          ANL A, #MASK     ;only P3.1-P3.5
800A B4300D                        CJNE A, #OK, ERR ;any error?
800D E4                  SEND:     CLR ~~A~~ STR    ;no: get ready to send
800E 20B1FD                        JB ACK, $        ;before send, wait for ACK
8011 93                            MOVC A, @A+DPTR  ;get char in test msg
8012 F590                          MOV DAT, A       ;send char to printer
8014 A3                            INC DPTR         ;point to next char
8015 B400EB                        CJNE A, #00H, LOOP ;not end of msg? loop
8018 80E6                          SJMP STRT        ;end of msg, back to start

801A 801F                ERR:      SJMP STOP        ;printer error, stop

                         ;***********************************************
                         ; Test message for the printer                 *
                         ;***********************************************

                         ;test message
801C 54484953            MSG:      DB 'THIS IS A TEST FOR THE PRINTER', 00H
8020 20495320
8024 41205445
8028 53542046
802C 4F522054
8030 48452050
8034 52494E54
8038 4552
803A 00
803B 00                  STOP:     NOP
                                   END
```

FIGURE 11-22b
continued

not satisfied, an error message is generated and the program ends. Otherwise, the 8051 waits for the printer's $\overline{\text{ACK}}$ signal before it proceeds. Once the $\overline{\text{ACK}}$ signal is activated, the current character is obtained from the test message and sent to the printer's data pins via P1.

11.11 ANALOG OUTPUT

Interfacing to the real world often requires generating or sensing analog conditions. Generating and controlling an analog output signal from a microcontroller is easy. This design example uses two resistors, two capacitors, a potentiometer, an LM301 op amp, and MC1408L8 8-bit digital-to-analog converter (DAC). Both ICs are inexpensive and readily available. The eight data inputs to the DAC are driven from Port 1 on the 8031 (see Figure 11-23). After building the circuit and connecting it to the SBC-51, it should be tested using monitor commands. Measure the output voltage at pin 6 of the LM301 (Vo) while writing different values to Port 1 and adjusting the 1K potentiometer. The output should vary from 0 volts (P1 = 00H) to about 10 volts (P1 = 0FFH).

Once the circuit is operating correctly, we are ready to have fun with the interface software. The usual test program is a sawtooth waveform generator that sends a value to the DAC, increments the value, sends it again, and so on (see question 3 at the end of this chapter). However, we will embark on a much more ambitious design—a digitally controlled sine wave generator.

EXAMPLE 11.10

Design Objective

Write a program to generate a sine wave using the DAC interface in Figure 11-23. Use a constant call STEP to set the frequency of the sine wave. Make the program interrupt driven with an update rate of 10 kHz.

Since the number-crunching capabilities of the 8031 are very limited, the only reasonable approach to this problem is to use a look-up table. We need a table with 8-bit values corresponding to one period of a sine wave. The values should start around 127, increase to 255, decrease through 127 to D, and rise back up to 127, following the pattern of a sine wave.

A reasonable rendition of a sine wave requires a relatively large table; so the question arises, how do we generate the table? Manual methods are impractical. The easiest

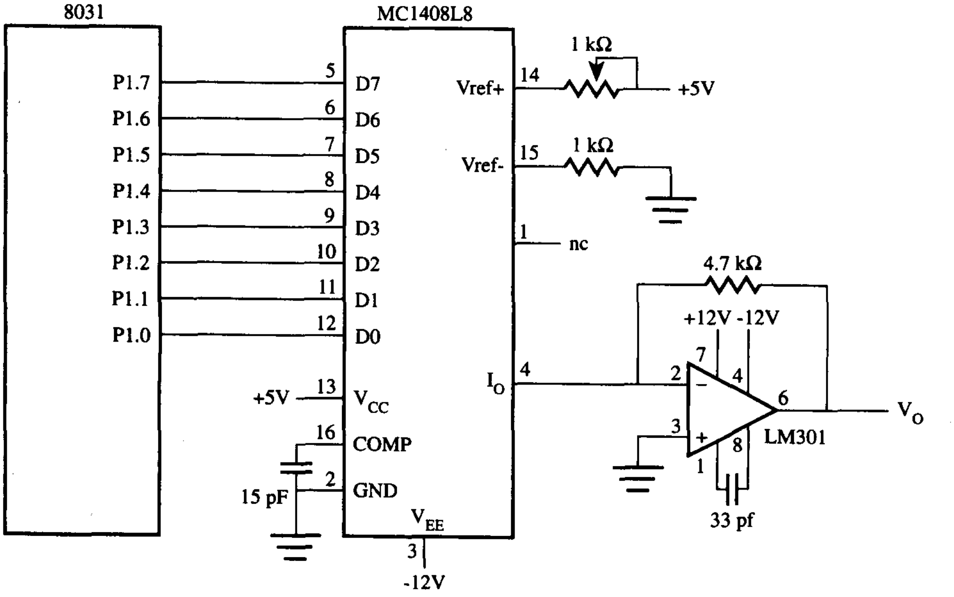
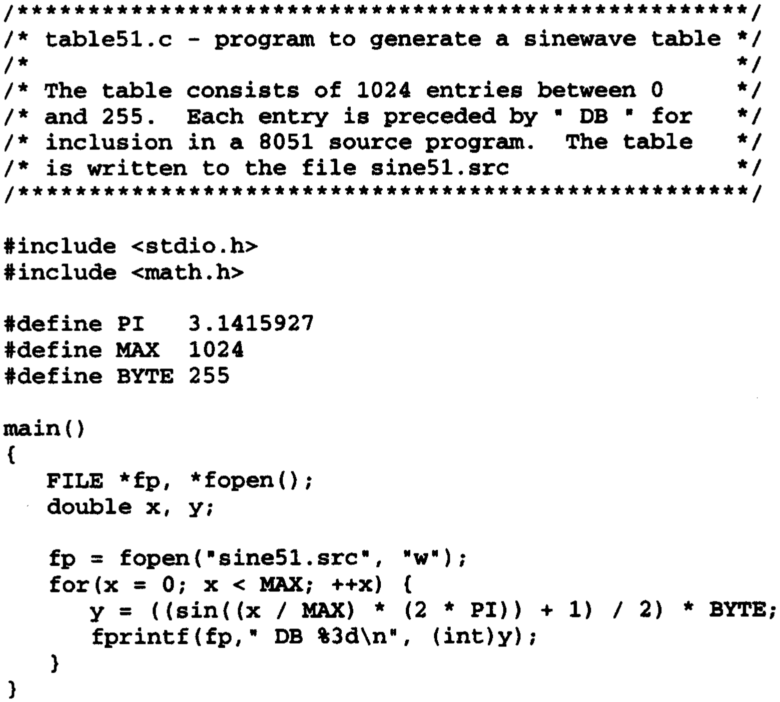


FIGURE 11-23
Interface to MC1408L8



```
/*****************************************************/
/* table51.c - program to generate a sinewave table */
/*                                                   */
/* The table consists of 1024 entries between 0      */
/* and 255.  Each entry is preceded by " DB " for    */
/* inclusion in a 8051 source program.  The table    */
/* is written to the file sine51.src                 */
/*****************************************************/

#include <stdio.h>
#include <math.h>

#define PI   3.1415927
#define MAX  1024
#define BYTE 255

main()
{
   FILE *fp, *fopen();
   double x, y;

   fp = fopen("sine51.src", "w");
   for(x = 0; x < MAX; ++x) {
      y = ((sin((x / MAX) * (2 * PI)) + 1) / 2) * BYTE;
      fprintf(fp," DB %3d\n", (int)y);
   }
}
```

FIGURE 11-24
Software to generate a sine wave table

approach is to write a program in some other high-level language to create the table and save the entries in a file. The table is then imported into our 8031 source program, and off we go. Figure 11-24 is a simple C program called table51.c that will do the job for us. The program generates a 1024-entry sine wave table with values constrained between 0 and 255. The output is written to an output file called sine51.src. Each entry is preceded with "DB" for compatibility with 8031 source code.

The 8031 sine wave program is shown in Figure 11-25. The main loop (lines 36-40) does three things: initializes timer 0 to interrupt every 100 μs, turns on interrupts, and sits in an infinite loop. The timer 0 interrupt service routine (lines 41-51) does all the work. Every 100 μs a value is read from the lookup table using the DPTR and then written to port 1. A constant called STEP is used as the increment through the table. STEP is defined in line 26 as a byte in internal RAM. It must be initialized using a monitor command. Within each ISR, STEP is added to DPTR to get the address of the next sample. The table is ORGed at 8400H (line 69), so it starts on an even 1K boundary. If the DPTR is incremented past 87FFH (the end of the table), it is adjusted to wrap around through the beginning of the table. Since the table is so big, a $NOLIST assembler directive was used after the first five entries (line 77) to shut off output to the

```
                 $debug
                 $nopaging
                 $nosymbols
                 ;FILE: DAC.SRC
                 ;*******************************************************
                 ;              MC1408L8 INTERFACE EXAMPLE              *
                 ;                                                      *
                 ; This program generates a sine wave using a sine      *
                 ; wave look-up table and an interface to a MC1408L8    *
                 ; 8-bit digital-to-analog converter.  The program is   *
                 ; interrupt-driven.                                    *
                 ;                                                      *
                 ; Data are read from a 1024-entry sine wave table and *
                 ; sent to the DAC every 100 us.  Each value sent is    *
                 ; STEP locations past the previous value sent (with    *
                 ; wrap around once the end is reached).   The period   *
                 ; of the sine wave is 100 x (1024 / STEP) us.  For     *
                 ; example, if STEP is 20H, the sine wave has a period *
                 ; of 100 x (1024 / 32) = 3.2 ms and a frequency of     *
                 ; 313 Hz.                                              *
                 ;                                                      *
                 ; Note:  Initialize STEP in internal location 50H      *
                 ; before running the program.                          *
                 ;*******************************************************
  00BC           MONITOR  CODE     00BCH      ;MON51 entry (V12)
  0050           STEP     DATA     50H        ;put STEP in internal RAM

8000                      ORG      8000H      ;start at 8000H
8000 028015               LJMP     MAIN       ;initialize timer
8003 028037               LJMP     EXT0ISR    ;unsused
8006 028022               LJMP     T0ISR      ;every 100 us, update DAC
8009 028037               LJMP     EXT1ISR    ;unused
800C 028037               LJMP     T1ISR      ;unused
800F 028037               LJMP     SPISR      ;unused
8012 028037               LJMP     T2ISR      ;unused
8015 758902      MAIN:    MOV      TMOD,#02H  ;8-bit, auto reload
8018 758C9C               MOV      TH0,#-100  ;100 us delay
801B D28C                 SETB     TR0        ;start timer
801D 75A882               MOV      IE,#82H    ;enable timer 0 interrupts
8020 80FE                 SJMP     $          ;main loop does nothing!
8022 E550        T0ISR:   MOV      A,STEP     ;add STEP to DPTR
8024 2582                 ADD      A,DPL
8026 F582                 MOV      DPL,A
8028 5002                 JNC      SKIP
802A 0583                 INC      DPH
802C 538303      SKIP:    ANL      DPH,#03H   ;wrap around, if necessary
802F 438384               ORL      DPH,#HIGH(TABLE)
8032 E4                   CLR      A
8033 93                   MOVC     A,@A+DPTR  ;get entry
8034 F590                 MOV      P1,A       ;send it
8036 32                   RETI

                 EXT0ISR:                     ;unused interrupts
                 EXT1ISR:
                 T1ISR:
                 T2ISR:
8037 C2AF        SPISR:   CLR      EA         ;turn off interrupts and
8039 0200BC               LJMP     MONITOR    ; return to MON51

                 ;*******************************************************
                 ; The following is a sine wave look-up table.  The     *
                 ; table contains 1024 entries and is ORGed to begin    *
                 ; at 8400H to allow easy wrap-around of the DPTR       *
                 ; once the end of the table is reached.  The entries   *
```

FIGURE 11-25a
8031 Sine wave program

```
                   65     ; are 8-bits each (0 to 255) for output to an 8-bit   *
                   66     ; DAC.  The table was generate from a C program and   *
                   67     ; read into this 8051 program.                        *
                   68     ;*****************************************************
8400               69             ORG      8400H
8400 7F            70     TABLE:  DB       127
8401 80            71             DB       128
8402 81            72             DB       129
8403 81            73             DB       129
8404 82            74             DB       130
                   75     ; Listing turned off after first five entries
                   76     ;----------------------------------------------------
                   77 +1  $NOLIST
                 1093     ; Listing turned back on for last five entries
87FB 7B          1094            DB       123
87FC 7C          1095            DB       124
87FD 7D          1096            DB       125
87FE 7D          1097            DB       125
87FF 7E          1098            DB       126
                 1099            END
```

FIGURE 11-25b
continued

listing file. A $LIST directive was used in line 1092 (not shown) to turn the listing back on for the last five entries. The frequency of the sine wave is controlled by three parameters: STEP, the size of the table, and the timer interrupt period, as explained in lines 16-20 of the listing.

11.12 ANALOG INPUT

Our next design example is an analog input channel. The circuit in Figure 11-26 uses one resistor, one capacitor, a trimpot, and an ADC0804 analog-to-digital converter (ADC). The ADC0804 is an inexpensive ADC (National Semiconductor Corp.) that converts an input voltage to an 8-bit digital word in about 100 μs.

The ADC0804 is controlled by a write input ($\overline{\text{WR}}$) and an interrupt output ($\overline{\text{INTR}}$) A conversion is started by pulsing $\overline{\text{WR}}$ low. When the conversion is complete (100 ms later), the ADC0804 asserts $\overline{\text{INTR}}$, making it low. $\overline{\text{INTR}}$ is de-asserted (high) on the next 1-to-0 transition of $\overline{\text{WR}}$, which initiates the next conversion. $\overline{\text{INTR}}$ and $\overline{\text{WR}}$ connect to the 8031 lines P1.1 and P1.0, respectively. For this example, we use Port A of the 8155 for the data transfer, as shown in the figure.

The ADC0804 operates from an internal clock created by the RC network connecting to pins 19 and 4. The analog input voltage is a differential signal applied to the Vin(+) and Vin(-) inputs on pins 6 and 7. For this example Vin(-) is grounded and Vin(+) is driven from the center tap of the trimpot. Vin(+) will range from 0 to +5 volts, as controlled by the trimpot. Consult the data sheet for a detailed description of the operation of the ADC0804.

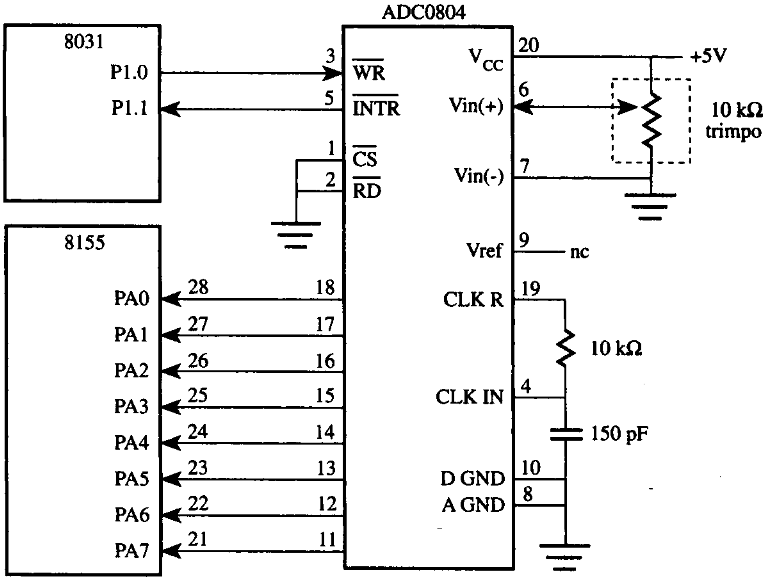


FIGURE 11-26
Interface to ADC0804 ADC

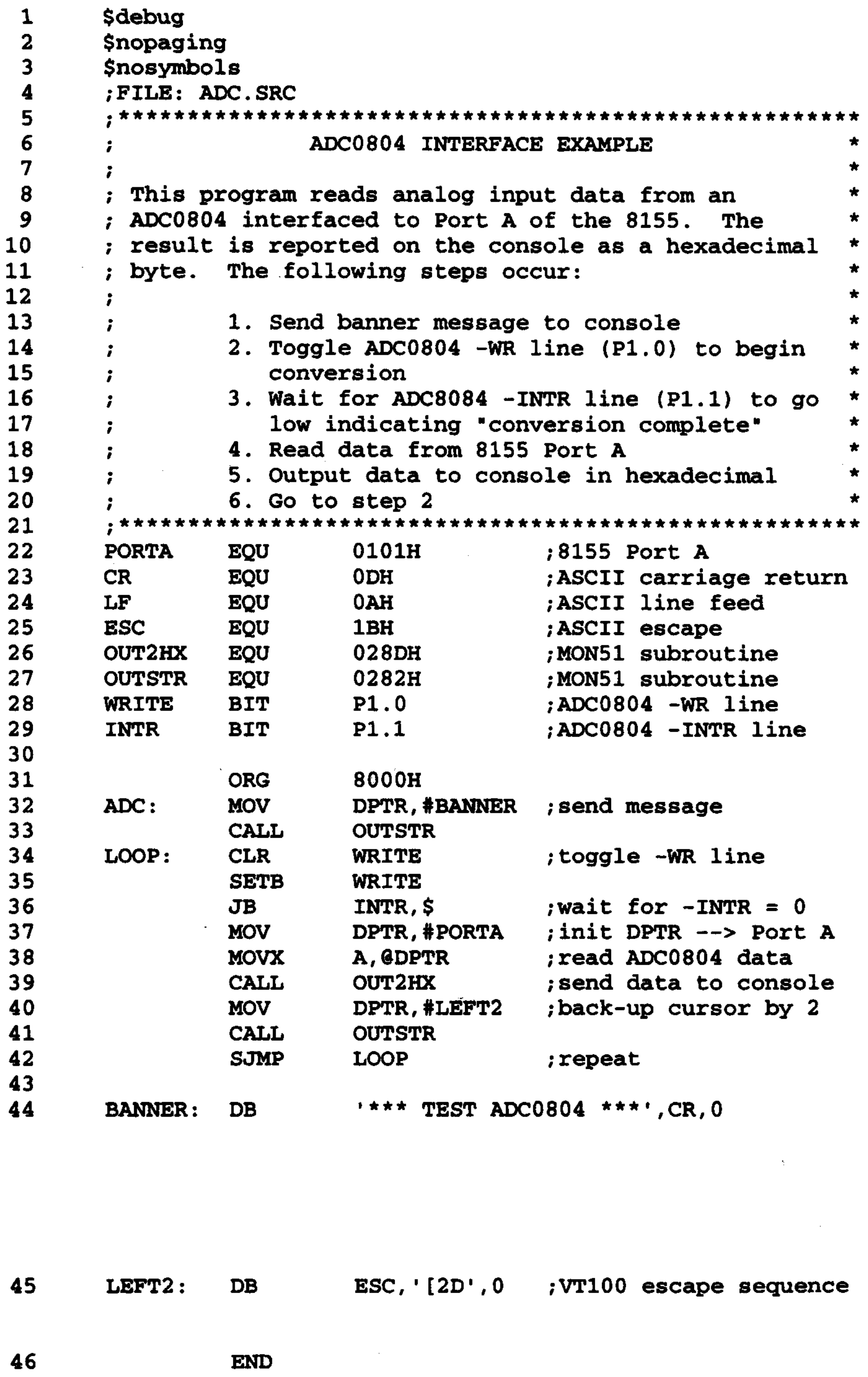
EXAMPLE 11.11

Design Objective

Write a program to continually sense the voltage at the trimpot's center tap (as converted by the ADC0804). Report the result on the console as an ASCII byte.

The program in Figure 11-27 achieves the objective stated above. Since the 8155 ports default to input upon reset, an initialize sequence is not necessary. Port A is at address 0101H in external memory and is easily read using a MOVX instruction. A conversion is started by clearing and setting P1.0 (lines 34-35), the ADC0804's $\overline{\text{WR}}$ input. Then, the program sits in a loop waiting for the ADC0804 to finish the conversion and assert $\overline{\text{INTR}}$ at P1.1 (line 36). The data are read in lines 37 and 38 and then sent to the console, using MON51's OUT2HX subroutine (line 39). As the program runs, a byte is displayed on the console. It will range from 00H to 0FFH as the trimpot is adjusted.

The program in Figure 11-27 is a rough first approximation of the potential for analog input. It is possible to replace the trimpot with other analog inputs. Temperature sensing is achieved using a thermistor—a device with a resistance that varies with temperature. Speech input is possible using a microphone. The ADC80804's conversion period of 100 μs translates into a sampling frequency of 10 kHz. This is sufficient to capture signals with up to 5 kHz bandwidth, roughly equivalent to a voice-grade telephone line. Additional circuitry is required to boost the low-level signals provided by typical microphones to the 0-5 volt range expected by the ADC0804. Additionally, a sample-and-hold circuit is needed to maintain a constant voltage for the duration of each conversion. We'll leave it to the reader to explore these possibilities.



```
$debug
$nopaging
$nosymbols
;FILE: ADC.SRC
;******************************************************
;              ADC0804 INTERFACE EXAMPLE              *
;                                                     *
; This program reads analog input data from an        *
; ADC0804 interfaced to Port A of the 8155.  The      *
; result is reported on the console as a hexadecimal  *
; byte.  The following steps occur:                   *
;                                                     *
;        1. Send banner message to console            *
;        2. Toggle ADC0804 -WR line (P1.0) to begin   *
;           conversion                                *
;        3. Wait for ADC8084 -INTR line (P1.1) to go  *
;           low indicating "conversion complete"      *
;        4. Read data from 8155 Port A                *
;        5. Output data to console in hexadecimal     *
;        6. Go to step 2                              *
;******************************************************
PORTA    EQU      0101H         ;8155 Port A
CR       EQU      0DH           ;ASCII carriage return
LF       EQU      0AH           ;ASCII line feed
ESC      EQU      1BH           ;ASCII escape
OUT2HX   EQU      028DH         ;MON51 subroutine
OUTSTR   EQU      0282H         ;MON51 subroutine
WRITE    BIT      P1.0          ;ADC0804 -WR line
INTR     BIT      P1.1          ;ADC0804 -INTR line

         ORG      8000H
ADC:     MOV      DPTR,#BANNER  ;send message
         CALL     OUTSTR
LOOP:    CLR      WRITE         ;toggle -WR line
         SETB     WRITE
         JB       INTR,$        ;wait for -INTR = 0
         MOV      DPTR,#PORTA   ;init DPTR --> Port A
         MOVX     A,@DPTR       ;read ADC0804 data
         CALL     OUT2HX        ;send data to console
         MOV      DPTR,#LEFT2   ;back-up cursor by 2
         CALL     OUTSTR
         SJMP     LOOP          ;repeat

BANNER:  DB       '*** TEST ADC0804 ***',CR,0
LEFT2:   DB       ESC,'[2D',0   ;VT100 escape sequence
         END
```

FIGURE 11-27
Software for ADC0804 interface

11.13 INTERFACE TO SENSORS

In Chapter 6's section on external interrupts, an example was given on the furnace controller that makes use of a simple temperature sensor. In this section, we will look at a more complex temperature sensor, the DS1620 digital thermometer and thermostat, manufactured by Maxim.[2]

The DS1620 is an 8-pin IC with three thermal alarm outputs, T_{HIGH}, T_{LOW}, and T_{COM}. T_{HIGH} goes high if the measured temperature is greater than or equal to the user-defined high temperature, TH. On the other hand, T_{LOW} goes high if the temperature is less than or equal to the user-defined low temperature, TL. In contrast, T_{COM} goes high when the temperature exceeds TH and only returns low when the temperature is less than TL.

To use the DS1620 along with the 8051, the DS1620 has to be configured for 3-wire communications. This is so called due to the fact that the communication between them is done through a group of three wires, namely the data input/output (DQ) pin, the clock input (CLK/CONV) pin, and the reset input (RST) pin. Communication is initiated by setting the RST, and as clock signals are sent to the DS1620 through the CLK pin, data would be clocked in or out through the DQ pin, with the LSB sent first.

Figure 11-28 shows the connections between the 8051 and the DS1620. Interacting with the DS1620 requires the 8051 to issue commands (see Table 11-5) to it.

FIGURE 11-28
Interface to DS1620

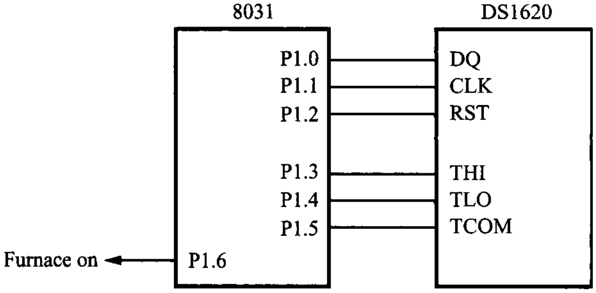


TABLE 11-5
DS1620 command set

Code	Instruction	Description
AAH	Read Temperature	Reads last converted temperature from temperature register
A0H	Read Counter	Reads value of count remaining from counter
A9H	Read Slope	Reads value of the slope accumulator
EEH	Start Convert	Initiates temperature conversion
22H	Stop Convert	Stops temperature conversion
01H	Write TH	Writes TH value into TH register
02H	Write TL	Writes TL value into TL register
A1H	Read TH	Reads TH value from TH register
A2H	Read TL	Reads TL value from TL register
0CH	Write Config	Writes config data to config register
ACH	Read Config	Reads config data from config register

[2] Maxim Integrated Products, Inc., 120 San Gabriel Drive, Sunnyvale, CA 94086.

EXAMPLE 11.12 **Design Objective**

Write a program that uses a temperature sensor to keep the room temperature at 20°C. If the temperature drops below 17°C, turn on the furnace. If the temperature exceeds 23°C, turn off the furnace.

The software for this is shown in Figure 11-29. The furnace is first turned off and the DS1620 is initialized. This requires sending the command 'write config' to the DQ pin, followed by a byte of configuration code for the configuration/status register (see Table 11-6). In our example, we set the configuration code such that the CPU bit is 1, selecting the DS1620 for 3-wire operation with the 8051. Also, the DS16290 is selected for continuous temperature conversion (sensing) by clearing the 1 shot bit to 0. Note that both sending the command byte and the configuration code is done by the SEND subroutine, which sends them bit by bit, LSB first to the DQ pin. Prior to the sending of any command to the DS1620,

```
                $DEBUG
                $NOSYMBOLS
                $NOPAGING
                ;FILE: SENSOR.SRC
                ;************************************************
                ;     TEMPERATURE SENSOR INTERFACE EXAMPLE      *
                ;                                               *
                ; This program uses a temperature sensor to     *
                ; monitor the room temperature.  When the       *
                ; temperature exceeds 23 degrees, the furnace   *
                ; is turned off.  If temperature drops below    *
                ; 17 degrees, the furnace is turned on.         *
                ;************************************************

 0090           DQ      EQU     P1.0
 0091           CLK     EQU     P1.1
 0092           RST     EQU     P1.2

 0093           THI     EQU     P1.3
 0094           TLO     EQU     P1.4
 0095           TCOM    EQU     P1.5

 0096           FURN    EQU     P1.6

8000                    ORG 8000H
8000 C296               CLR FURN                ;turn furnace off

8002 D292       CONF:   SETB RST                ;initiate transfer
8004 740C               MOV A, #0CH             ;write config
8006 128045             CALL SEND               ;send to DS1620
8009 740A               MOV A, #00001010B       ;CPU = 1; 1-shot=0
```

FIGURE 11-29a
Software for interface to DS1620

```
800B 128045                CALL SEND       ;send to DS1620
800E C292                  CLR RST         ;stop transfer

8010 D292                  SETB RST        ;initiate transfer
8012 7401                  MOV A, #01H     ;write TH
8014 128045                CALL SEND       ;send to DS1620
8017 7430                  MOV A, #48      ;TH = 48 x 0.5 = 24 deg C
8019 128045                CALL SEND       ;send to DS1620
801C C292                  CLR RST         ;stop transfer

801E D292                  SETB RST        ;initiate transfer
8020 7402                  MOV A, #02H     ;write TL
8022 128045                CALL SEND       ;send to DS1620
8025 7420                  MOV A, #32      ;TH = 32 x 0.5 = 16 deg C
8027 128045                CALL SEND       ;send to DS1620
802A C292                  CLR RST         ;stop transfer

802C D292          CONV:   SETB RST        ;initiate transfer
802E 74EE                 MOV A, #0EEH    ;start temperature
                                           sensing
8030 128045                CALL SEND       ;send to DS1620
8033 C292                  CLR RST         ;stop transfer

8035 209305        SENS:   JB THI, OFF     ;if T >= 24 degrees, off
8038 209406                JB TLO, ON      ;if T <= 16 degrees, on
803B 80F8                  SJMP SENS       ;loop

803D C296          OFF:    CLR FURN        ;turn furnace off
803F 80F4                  SJMP SENS       ;keep sensing

8041 D296          ON:     SETB FURN       ;turn furnace on
8043 80F0                  SJMP SENS       ;keep sensing
                   ;************************************************
                   ; This subroutine sends a byte of command or    *
                   ; data to the DS1620.                           *
                   ;************************************************

8045 7808          SEND:   MOV R0, #08     ;use R0 as counter
8047 C291          NEXT:   CLR CLK         ;start clock cycle
8049 13                    RRC A           ;rotate A into C, LSB 1st
804A 9290                  MOV DQ, C       ;send out bit to DQ
804C D291                  SETB CLK        ;complete the clock cycle
804E D8F7                  DJNZ R0, NEXT   ;process next bit
8050 22                    RET

                           END
```

FIGURE 11-29b
continued

TABLE 11-6
DS1620 configuration/status register

Bit	Name	Description
7	DONE	Conversion done bit. 1 = done 0 = in progress
6	THF	Temperature high flag. 1 = temperature ≥ TH
5	TLF	Temperature low flag. 1 = temperature ≤ TL
4	NVB	Nonvolatile memory busy flag. 1 = write to memory in progress 0 = memory not busy
3	1	–
2	0	–
1	CPU	CPU use bit. 0 = stand-alone mode (no CPU required) 1 = 3-wire communications mode
0	1SHOT	One-shot mode. 1 = one temperature conversion when receive 'start convert' command 0 = continuous temperature conversion

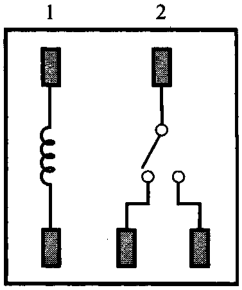
transfer has to be initiated by raising RST to high. When the command has been sent, the transfer is stopped by clearing RST.

After the 'write config' command, the user-defined high temperature, TH, and the user-defined low temperature, TL, is set. Next, a 'start convert' command is issued to direct the DS1620 to start sensing the temperature. The 8051 then waits in an indefinite loop on the status of the THIGH and T_{LOW} outputs and turns the furnace on or off correspondingly.

11.14 INTERFACE TO RELAYS

A relay is a switch whose contacts are opened or closed due to a magnetic field produced by the application of electricity to it. In essence, relays can be thought of as magneto-mechanical

FIGURE 11-30
Internal connections for G6RN



switches and are very useful in applications where it is desired that the opening and closing of switches be controlled electrically.

For the example in this section, we will use the OMRON's[3] single pole double throw (SPDT) relay model G6RN whose internal connections are as shown in Figure 11-30.

EXAMPLE 11.13

Design Objective

Consider a simple pedestrian traffic light system illustrated in Figure 11-31. Design such a traffic light system using the 8051 and a relay to control it.

The schematic diagram showing the connections between the 8051, relay, pedestrian buttons and traffic light LEDs are given in Figure 11-32. The program listing for

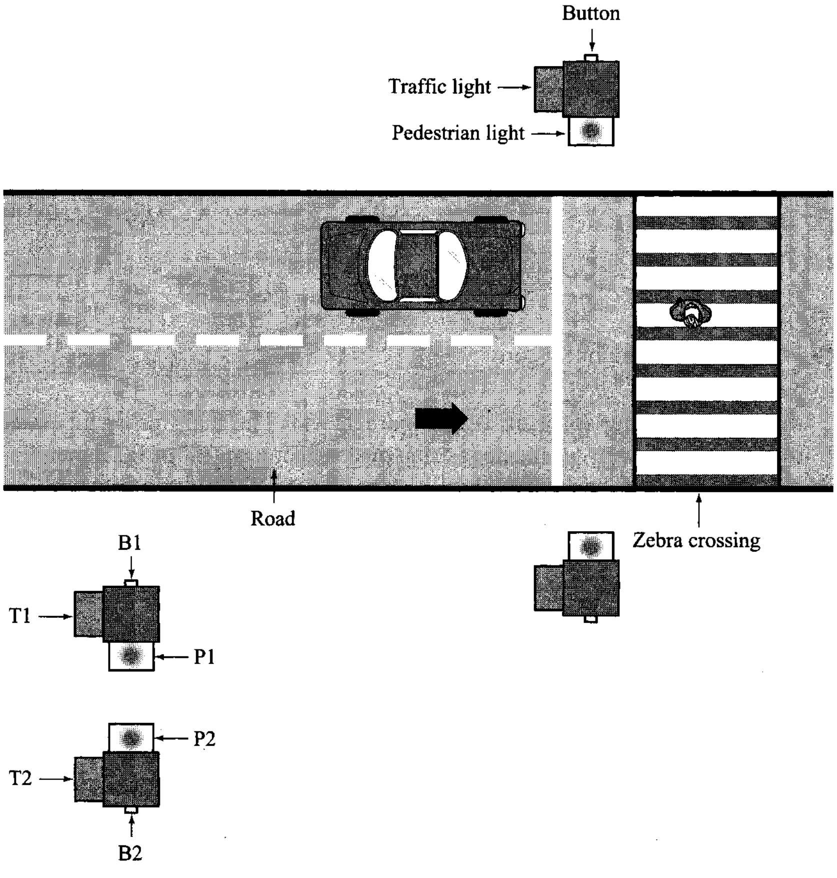


FIGURE 11-31
A pedestrian light system[4]

[3] OMRON Corporation, 28th Fl., Crystal Tower Bldg., 1-2-27, Shiromi, Chuo-ku,Osaka 540 Japan.
[4] Thanks to Joseph, Ken, and Tonny for undertaking this project and for generating the graphics in Figure 11-31.

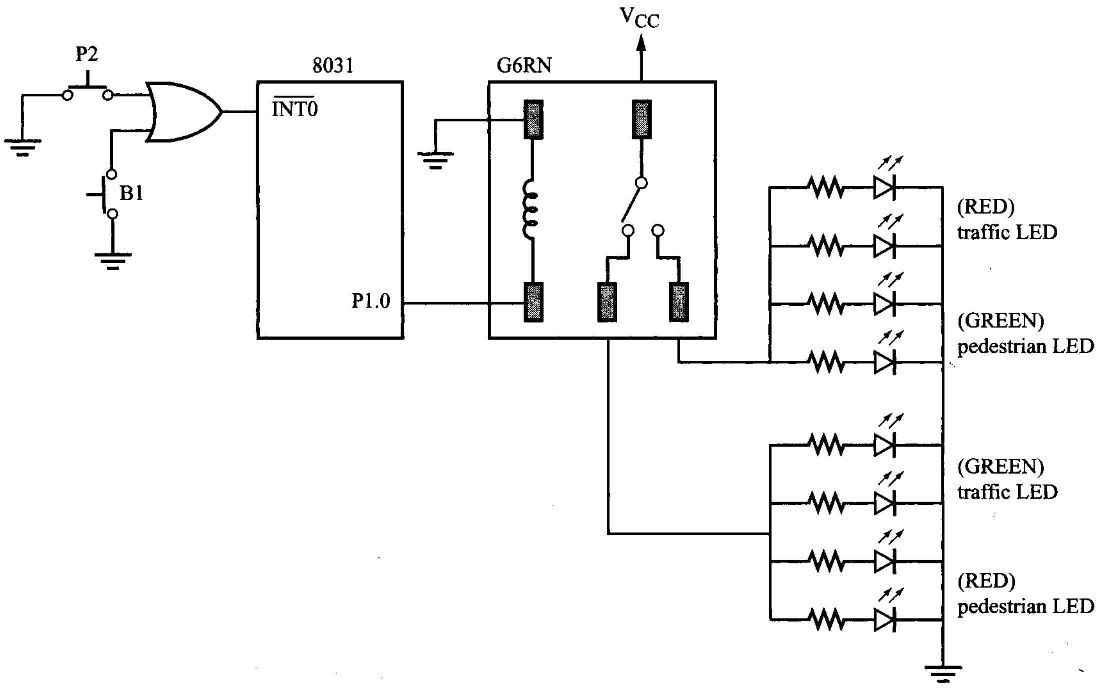


FIGURE 11-32
Interface to G6RN: Pedestrian Light System

the corresponding software is in Figure 11-33. The main program enables the relevant interrupts and waits until a pedestrian pushes a button (B1 or B2 depending on which side of the road he or she is on). Initially, port 1.0 is LOW, and the pole within the relay causes the GREEN traffic light and RED pedestrian light to be on. Once a button is pressed, this generates an interrupt INT0 to the 8051, which calls the corresponding interrupt service routine EX0ISR, the main bulk of the program. Within this ISR, port 1.0 is set, causing

```
$DEBUG
$NOSYMBOLS
$NOPAGING
;FILE: STEPPER.SRC
;*************************************************
; RELAY INTERFACE EXAMPLE:                       *
;                     PEDESTRIAN TRAFFIC LIGHT   *
;                                                *
; This program interacts with a relay to        *
; control some RED and GREEN LEDs in a           *
; pedestrian traffic light system.               *
;                                                *
```

FIGURE 11-33a
Software for interface to the G6RN

```
                    ; Initially, the GREEN traffic light and RED   *
                    ; pedestrian light are on.  If a pedestrian-   *
                    ; crossing button is pressed, a high-to-low    *
                    ; transition is generated at INT0.  This       *
                    ; signals the RED traffic light and the GREEN  *
                    ; pedestrian light to turn on.  A delay of 10  *
                    ; seconds ensues before the GREEN traffic light*
                    ; and RED pedestrian light are turned back on. *
                    ;***********************************************

  000A              TEN      EQU 10
  0064              HUNDRED  EQU 100 ;10 x 100 x 10000 us = 10 secs
  D8F0              COUNT    EQU -10000

0000                         ORG 0
0000 02000B                  LJMP MAIN
                                           ;EXT 0 vector at 0003H
0003 D290           EX0ISR:  SETB P1.0     ;traffic light = RED,and
                                           ; pedestrian light = GREEN
0005 120017                  CALL DELAY    ;wait 10 seconds
0008 C290                    CLR P1.0      ;traffic light = GREEN,
                                           ; pedestrian light = RED
000A 32                      RETI

000B 75A881         MAIN:    MOV IE, #81H  ;enable external 0 int
000E D288                    SETB IT0      ;negative edge triggered
0010 758901                  MOV TMOD, #01 ;timer 0 in mode 1
0013 C290                    CLR P1.0      ;traffic light = GREEN,
                                           ; pedestrian light = RED
0015 80FE                    SJMP $        ;wait forever

                    ;***********************************************
                    ; Subroutine for 10-second delay               *
                    ;***********************************************

0017 7E0A           DELAY:   MOV R6, #TEN
0019 7F64           AGAIN1:  MOV R7, #HUNDRED
001B 758CD8         AGAIN2:  MOV TH0, #HIGH COUNT
001E 758AF0                  MOV TL0, #LOW COUNT
0021 D28C                    SETB TR0
0023 308DFD         WAIT:    JNB TF0, $
0026 C28D                    CLR TF0
0028 C28C                    CLR TR0
002A DFEF                    DJNZ R7, AGAIN2
002C DEEB                    DJNZ R6, AGAIN1
002E 22                      RET

                             END
```

FIGURE 11-33b
continued

the relay's pole to change to its other contact, hence turning on the RED traffic light and the GREEN pedestrian light. A delay of 10 seconds then passes, allowing the pedestrians to cross the road before everything returns to normal and the GREEN traffic light and RED pedestrian light are turned on again.

11.15 STEPPER MOTOR INTERFACE

Various devices such as the dot-matrix printer and the floppy disk drive make use of a special type of motor called the stepper motor. Unlike the conventional DC motor, a stepper motor moves in precise increments or steps. This allows for it to be used in applications that require exact positioning of mechanical parts.

Let's briefly review the basic operation of a stepper motor. A stepper motor has a moving part called the **rotor** that is typically made from permanent magnet material. This rotor is surrounded by the **stator,** which is an electromagnet made up of two windings around some conducting poles. This is shown in more detail in Figure 11-34.

Notice that both the two windings have a center tap, COM, which is typically connected to +5V. Consider next one of these windings shown oriented vertically in Figure 11-34(a), with its two ends denoted by A and B. This vertical winding will be wound round the up and down stator poles, as shown in Figure 11-34(b). The two ends A and B will be alternatively energized, causing current to flow through either one of them and hence turning the stator pole into an electromagnet to attract the rotor magnet in the middle to move in the desired direction. Figure 11-34(b) also shows that the stator A has been energized to attract the rotor magnet towards it in the indicated direction. In the similar way, the horizontal windings' two ends, C and D, would be wound round the left and right stator poles, and these will be alternately energized as well.

A further illustration of the interaction between the rotor and the stator is given in Figure 11-35. Suppose that stator A is first energized and hence becomes an electromagnet to attract the rotor, as shown in Figure 11-35(a). Next, stator D is energized and attracts

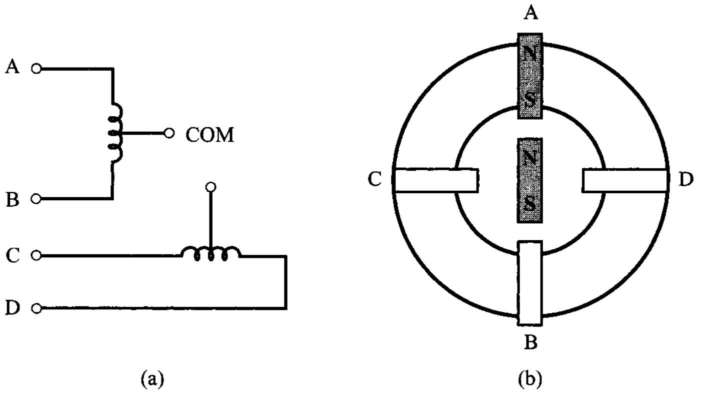


FIGURE 11-34
Configuration of the stator windings of a stepper motor

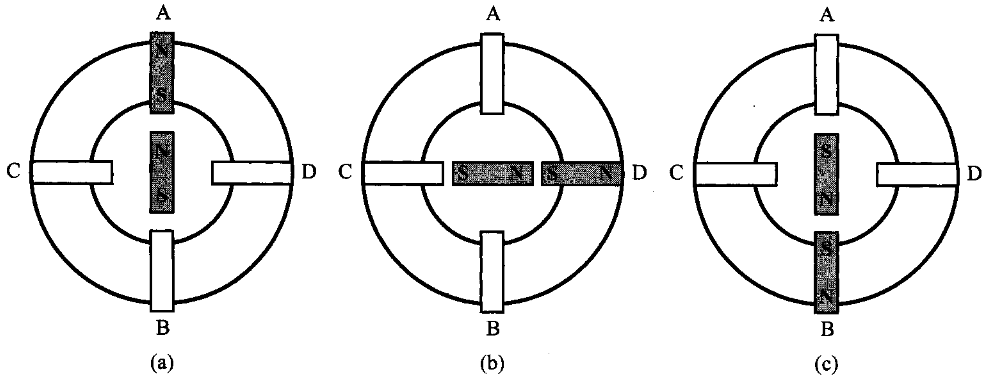


FIGURE 11-35
Rotating the rotor of the stepper motor

the rotor towards it, as in Figure 11-35(b). This is followed by energizing stator B, again attracting the rotor as in Figure 11-35(c). Finally, stator C is energized and similarly attracts the rotor. The energizing of the stators in the sequence A, D, B, C causes the rotor to rotate in clockwise direction to complete one revolution, or the rotor is said to have made a 360° clockwise rotation. In contrast, if the stators are energized in the opposite sequence, the rotor would make a counterclockwise rotation.

Notice that the rotor, when operated in this way, moves to four unique positions as it steps to one full rotation, where each step involves a rotation of 90°. A stepper motor of such type is called a **4-step stepper motor.** For finer resolutions, **8-step stepper motors** can be used, where a step makes a rotation of 45. This is achieved for example by energizing the stator poles not only one at a time but alternatively energizing two adjacent stators at a time. For example, to rotate the stepper motor clockwise, we energize the stator poles according to the sequence A, AD, D, DB, B, BC, C, CA. The first three steps are illustrated in Figure 11-36.

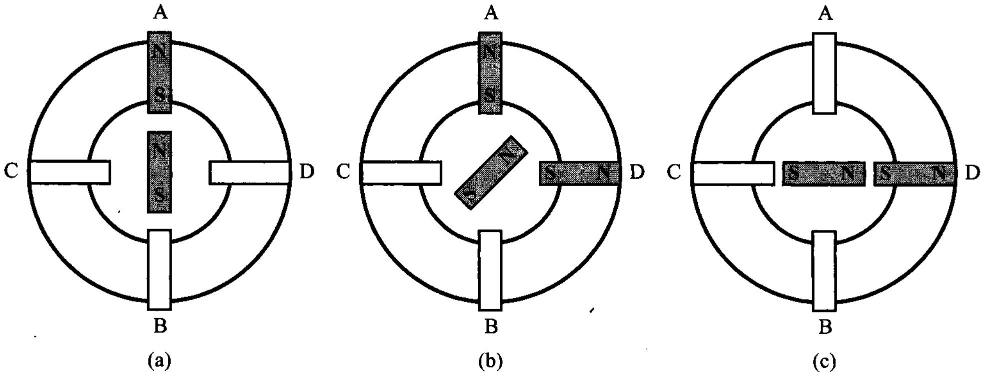


FIGURE 11-36
Half-step rotation of the stepper motor

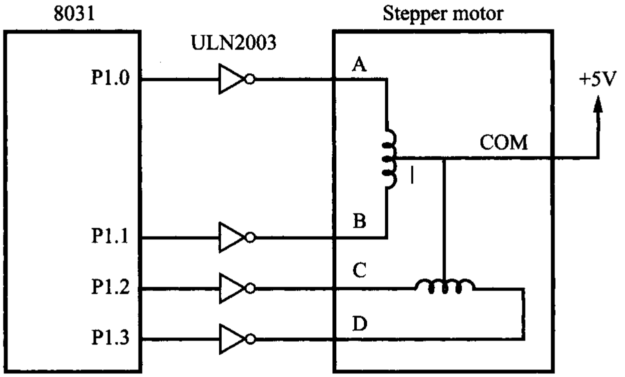


FIGURE 11-37
Interface to the stepper motor

The connection from the 8051 to the windings of the stepper motor has to be made through a stepper motor driver as the currents from the 8051 port pins are insufficient to drive the windings. Figure 11-37 shows how the ULN2003 stepper motor driver interfaces the 8051 to the stepper motor.

With the connections in Figure 11-37, it is clear that to energize a stator pole, a high '1' has to be written to the corresponding port pin connected to it, allowing current to flow through the stator windings thus turning it into an electromagnet. The sequence of steps for both the 4-step and 8-step stepper motors are given in Table 11-7 and Table 11-8, respectively.

TABLE 11-7
Full-step 4-step sequence for clockwise rotation

Step	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	0	0	0	1
3	0	1	0	0
4	0	0	1	0

TABLE 11-8
Half-step 8-step sequence for clockwise rotation

Step	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	1	0	0	1
3	0	0	0	1
4	0	1	0	1
5	0	1	0	0
6	0	1	1	0
7	0	0	1	0
8	1	0	1	0

EXAMPLE 11.14

Design Objective

Write a program to initially rotate the stepper motor clockwise. If a switch connected to INT0 makes a high-to-low transition, change the direction of rotation, in this case it becomes counterclockwise.

The software for this is shown in Figure 11-38. Since the external interrupt 0 ($\overline{\text{INT0}}$) is used to detect the high-to-low transition, a relevant ISR for $\overline{\text{INT0}}$ is included as part of

```
                $DEBUG
                $NOSYMBOLS
                $NOPAGING
                ;FILE: STEPPER.SRC
                ;***********************************************
                ;         STEPPER MOTOR INTERFACE EXAMPLE      *
                ;                                              *
                ; This program uses the 8-step sequence to     *
                ; initially rotate the stepper motor clockwise.*
                ; If a switch connected to INT0 makes a        *
                ; high-to-low transition, the rotation is      *
                ; changed to counterclockwise.                 *
                ;***********************************************

  0009          STEP1      EQU    1001B  ;8-step sequence for
  0008          STEP2      EQU    1000B  ; clockwise rotation
  000C          STEP3      EQU    1100B
  0004          STEP4      EQU    0100B
  0006          STEP5      EQU    0110B
  0002          STEP6      EQU    0010B
  0003          STEP7      EQU    0011B
  0001          STEP8      EQU    0001B
  0000          D          EQU    0      ;direction bit
                                         ; at bit address 0
  0064          HUNDRED    EQU    100    ;100 x 10000 us = 1 sec
  D8F0          COUNT      EQU    -10000

0000                       ORG 0
0000 020006                LJMP MAIN
                                         ;EXT 0 vector at 0003H
0003 B200       EX0ISR:    CPL D         ;complement direction bit
0005 32                    RETI

0006 75A881     MAIN:      MOV IE, #81H  ;enable INT0
0009 D288                  SETB IT0      ;negative edge triggered
000B 758901                MOV TMOD, #01 ;timer 0 in mode 1
000E C200                  CLR D         ;initialize D = 0
0010 120020                CALL CW       ;initial: CW rotation
0013 200005     REPEAT:    JB D, GO_CCW  ;CCW?
0016 120020     GO_CW:     CALL CW       ;no: CW
```

FIGURE 11-38a
Software for stepper motor interface

```
0019 80F8            41                SJMP REPEAT
001B 120035          42    GO_CCW:     CALL CCW     ;yes: CCW
001E 80F3            43                SJMP REPEAT  ;repeat forever
                     44
                     45    ;***********************************************
                     46    ; Subroutine for clockwise (CW) rotation      *
                     47    ;***********************************************
                     48
0020 90005F          49    CW:         MOV DPTR, #SEQ     ;point to table
0023 E4              50                CLR A        ;point to first step
0024 C0E0            51    NEXT:       PUSH ACC     ;backup index in A
0026 93              52                MOVC A,
0027 540F            53                ANL A, #0FH   ;mask off upper 4 bits
0029 F590            54                MOV P1, A    ;send to stepper motor
002B 12004B          55                CALL DELAY   ;wait 1 second
002E D0E0            56                POP ACC      ;restore index into A
0030 04              57                INC A        ;point to next step
0031 B408F0          58                CJNE A, #8, NEXT ;ensure 8 steps
0034 22              59                RET
0046 14              73                DEC A        ;point to next step
0047 B4FFF0          74                CJNE A, #0FFH, NEXT2 ;ensure 8 steps
004A 22              75                RET
                     76
                     77    ;***********************************************
                     78    ; Subroutine for 1-second delay               *
                     79    ;***********************************************
                     80
004B 7F64            81    DELAY:      MOV R7, #HUNDRED
004D 758CD8          82    AGAIN:      MOV TH0, #HIGH COUNT
0050 758AF0          83                MOV TL0, #LOW COUNT
0053 D28C            84                SETB TR0
0055 308DFD          85    WAIT:       JNB TF0, $
0058 C28D            86                CLR TF0
005A C28C            87                CLR TR0
005C DFEF            88                DJNZ R7, AGAIN
005E 22              89                RET
                     90
                     91    ;***********************************************
                     92    ; 8-step sequence pattern for clockwise       *
                     93    ; rotation.                                   *
                     94    ;***********************************************
                     95
005F 09              96    SEQ:       DB STEP1      ;lookup table with
0060 08              97               DB STEP2      ; sequence pattern
0061 0C              98               DB STEP3      ; for clockwise rotation
0062 04              99               DB STEP4
0063 06             100               DB STEP5
0064 02             101               DB STEP6
0065 03             102               DB STEP7
0066 01             103               DB STEP8
                    104
                    105               END
```

FIGURE 11-38b
continued

the program such that the direction bit, D, is complemented at every occurrence of a high-to-low transition at $\overline{\text{INT0}}$. There are also two functions, CW and CCW, to cause the stepper motor to rotate clockwise and anticlockwise, respectively, plus a simple subroutine that generates a delay of 1 second.

The main program initially directs the stepper motor to rotate in a clockwise fashion. Upon completion of one cycle of rotation, the current direction is checked by referring to D, and then either CW or CCW would be called to continue rotating the stepper motor in the direction indicated by D.

SUMMARY

This concludes our examination of interface examples. The designs presented illustrate many of the concepts required to implement sophisticated interfaces using an 8051 microcontroller.

There is no substitute for hands-on experience, however. The examples in this chapter, and those presented earlier, are best understood through a trial-and-error process. Implementing the examples on a real system, such as the SBC-51, is the best way to develop the concepts presented in this book. This book has provided a basis for students to explore further the possibilities of using a microcontroller such as the 8051 in minimum component designs.

PROBLEMS

11.1 Use the loudspeaker interface in Figure 11-9 to repeatedly play the musical melody shown in Figure 11-39.

11.2 Reconfigure the 74HC165 interface in Figure 11-15 and rewrite the accompanying software in Figure 11-16 to use the 8051's serial port (in mode 0) for the clock and data lines.

11.3 If the following program is used with the digital-to-analog output circuit in Figure 11-23, a sawtooth wave results. (Assume 12 MHz operation.)

```
STEP EQU  1
MAIN MOV  P1,A
     ADD  A,#STEP
     SJMP MAIN
```

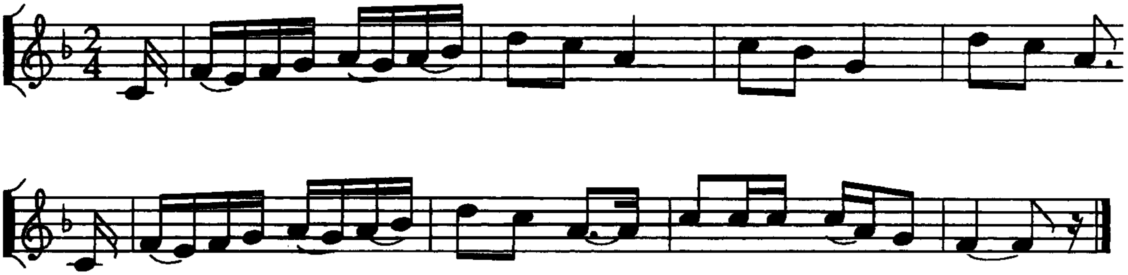


FIGURE 11-39
Musical melody for Problem 11.1

a. What is the frequency of the sawtooth wave?
b. What value for STEP will achieve an output frequency of 10 kHz (approximately)?
c. Derive the equation for frequency, given STEP.

11.4 Several of the programs in this chapter used MON51 subroutines. If a program is to use the subroutine ISDIG (to check if a byte is an ASCII digit), what address from MON51 must be equated to the symbol ISDIG?

11.5 Write a program to create a 1 kHz square wave on P1.7 using the 8155 timer and external 0 interrupts. (Hint: consult an 8155 data sheet.)

11.6 Assume the 74HC165 interface in Figure 11-15 is expanded to 6 ICs, providing 48 additional input lines.

a. How should the program in Figure 11-16 be modified to read the 48 input lines?
b. Where will the data be stored in the 8031's internal RAM?
c. What is the new duration of the GET_BYTES subroutine in Figure 11-16?
d. If GET_BYTES is placed in an interrupt service routine that executes every second, what percentage of the CPU's execution time is spent reading the 48 inputs from the 74HCl65s?

11.7 In Figure 11-25, the constant STEP was defined as a byte of internal data at location 50H using the following assembler directive:

```
STEP DATA  50H
```

This is the correct way to define STEP; however, the following would also work:

```
STEP EQU  50H
```

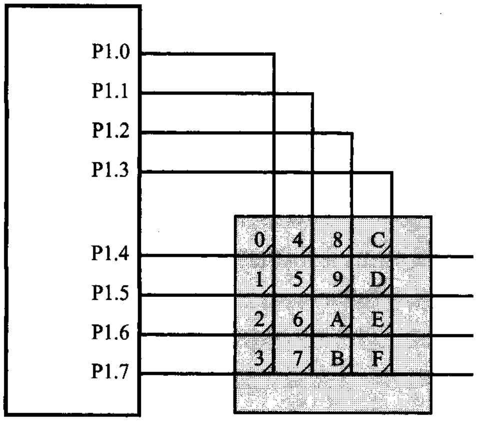
In the latter case, type-checking is not performed by ASM51 when the program is assembled. Give an example of an incorrect use of the label STEP that *would not* generate an assemble-time error if STEP were defined with EQU, but *would* generate an error if STEP were defined properly, using DATA.

11.8 Suppose a 16-key HEX keypad has been connected to the 8051 as shown in Figure 11-40. The columns are numbered from left to right, starting from 0, while the rows are numbered from up to down, also starting from 0. Suppose that when a key-press is detected, the HIT subroutine is called, and R6 contains the column information and the upper 4 bits of P1 contain the row information, where

R6 = 3	Column 0
R6 = 2	Column 1
R6 = 3	Column 2
R6 = 0	Column 3
P1.4 = 0	Row 0
P1.5 = 0	Row 1
P1.6 = 0	Row 2
P1.7 = 0	Row 3

Write the subroutine HIT that uses the values in R6 and P1 to output the value of the key pressed. Show your steps.

FIGURE 11-40
HEX keypad



11.9 Rewrite the software for the LCD in Figure 11-8 to continually display the message "Welcome to the 8051 Microcontroller Experience. Hope you enjoy your reading adventure."

11.10 We have seen how to use the 8255 in mode 0. Research on the more complex modes of the 8255 and hence write the assembly language instructions to initialize the 8255's ports to each of those modes.

11.11 Compare between the serial and parallel interface. What in your opinion is the most significant difference? Hence why do you think serial interfaces are commonly used with keyboards, mice, and modems while parallel interfaces are used with printers?

11.12 Write the corresponding pseudo code for the pedestrian traffic light software in Figure 11-33.

11.13 Research the types and applications of stepper motors. Select three such applications and give specific explanations as to why the stepper motor is used in each of them.

Write the subroutine HIT that uses the values in R6 and P1 to output the value of the key pressed. Show your steps.

CHAPTER 12

Design and Interface Examples in C

12.1 INTRODUCTION

In the previous chapter, several design and interface examples involving the 8051 were presented, with the corresponding interface programs written in assembly language. This chapter will tackle these interface problems by writing similar proof-of-concept programs in 8051 C. This will give a clearer view on the difference and also the similarities between the two languages that we use to interact with the 8051.

12.2 HEXADECIMAL KEYPAD INTERFACE

Recall from the discussion of the previous chapter the hexadecimal keypad interface where the design objective was to write a program to continually read hexadecimal characters from the 4 x 4 keypad and echo the corresponding ASCII code to the console. A similar 8051 C program is shown in Example 12.1.

EXAMPLE 12.1 **Hexadecimal Keypad Interface**

Rewrite in C language the software for the hexadecimal keypad interface in Figure 11-4.

Solution

```
#include <reg51.h>
#include <stdio.h>

unsigned char R3, R5, R6, R7, tempA;
unsigned char bdata A;         /* represents ACC */
sbit aMSB = A^7;               /* variable aMSB to refer to
                                  A.7 */
```

```
sbit aLSB = A^0;               /* variable aLSB to refer to
                                   A.0 */
bit PY;                        /* represents parity bit */
void IN_HEX(void);
void HTOA(void);
void OUTCHR(void);   /* taken & modified from
                                Example 8¯6 */
bit GET_KEY(void);
void RL_A(void);
bit RRC_A(bit);
void PARITY(void);

main( )
{
while (1)                      /* repeat forever */
        {
        IN_HEX();              /* get code from keypad */
        HTOA();                /* convert to ASCII */
        OUTCHR();              /* echo to console */
        }
}

void IN_HEX(void)
{
R3 = 50;                       /* debounce count */
while (R3 != 0)
        {
        if (GET_KEY() == 0)    /* key pressed? */
              R3 = 50;         /* no: check again */
        else
              R3--;            /* yes: repeat 50 times */
        }
tempA = A;                     /* save hex code */
R3 = 50;                       /* wait for key up */
while (R3 != 0)
        {
        if (GET_KEY() == 1)    /* key pressed? */
              R3 = 50;         /* yes: keep checking */
        else
              R3--;            /* no: repeat 50 times */
        }
A = tempA;                     /* recover hex code and return
                                   */
}

bit GET_KEY(void)
{
bit C = 0;                     /* default return value: no key
                                   press */
```

```
A = 0xFE;                            /* start with column 0 */

for (R6 = 4; R6 > 0; R6--)           /* check all 4 columns */
      {
      P1 = A;                        /* activate column line
                                        */
      A = P1;                        /* read back Port 0 */
      tempA = A&(0xF0);              /* isolate row lines */
      if (tempA == 0xF0)             /* row lines active? */
            RL_A();                  /* no: move to next
                                        column line */
      else
            {
            R7 = A;                  /* save in R6 */
            A = 4;                   /* prepare to calculate
                                        col. weighting */
            C = 0;
            A = A - R6;              /* 4 - R6 = weighting */
            R6 = A;                  /* save in R6 */
            A = R7;                  /* restore scan code */
            A = ( R7 << 4 | R7 >> 4 );/* put in low nibble */
              for (R5 = 4; R5 > 0; R5--)/* use R5 as counter
              */ {
                  C = RRC_A(C);      /* rotate ACC until 0 */
                  if (C == 0)
                        break;       /* done when C = 0 */
                  else
                        R6 = R6 + 4; /* add 4 until row found
                                        */
                   }
            C = 1;                   /* key pressed! */
            A = R6;                  /* code in ACC */
            return C;
            }
      }
return C;
}

void RL_A(void)
{
bit tempBit;
tempest = aMSB;                      /* backup MSB of A */
A = A << 1;                          /* rotate A left */
aLSB = tempBit;
}                                    /* rotate aLSB into MSB */

bit RRC_A(bit C)
{
bit tempBit;

tempBit = C;                         /* backup C */
```

```
C = aLSB;                     /* rotate A.0 into C */
A = A >> 1;                   /* rotate A right */
aMSB = tempBit;               /* rotate C into A.7 */
return C; }

void HTOA(void)
{
A = A & 0xF;                  /* ensure upper nibble clear */
if (A >= 0xA)                 /* 'A' to 'F'? */
   A = A + 7;                 /* yes: add extra */
A = A + '0';                  /* no: convert directly */
}


void OUTCHR(void)
{
PARITY();                     /* get even parity of A and put
                                 in PY */
PY = !PY;                     /* change to odd parity */
aMSB = PY;                    /* add to character code */
while (TI != 1);              /* Tx empty? no: check again */
TI = 0;                       /* yes: clear flag and */
SBUF = A;                     /* send character */
aMSB = 0;                     /* strip off parity bit */
}

void PARITY(void)
{
int i;
PY = 0;                       /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )    /* calculate parity of A*/
      PY ^= ( A >> i ) & 1;
}
```

Discussion

Recall that registers R0 to R7 are used for storing parameters in C functions. For this reason, it is often not advisable to use these registers for temporary storage when programing the 8051 in C. Instead, simply declare any 8-bit variable for this purpose. Notice also that in our example above, we intentionally named the temporary variables with the register names R3 to R7 so that it is easier to relate our C program to the assembly language counterpart in Figure 11-4.

Due to the same reason, the accumulator, ACC, is used in most operations and hence is often overwritten with data, so should not be used in your C programs. In our above example, we used a temporary bit-addressable variable A to represent the accumulator.

Since we do not use the ACC but use a representative, A, instead, the parity of A would not be automatically updated in the parity flag, P. Instead, we would need to compute it on our own. Therefore, we slightly modified **`OTCHR()`** to include the calling of a function **`PARITY( )`** that computes the parity of A and stores it in a bit, PY.

In our solution, we also included two functions, **RL_A( )** and **RRC_A( )**, which basically work in the same way as the assembly language instructions **RL A** and **RRC A**. We had to do this in C since no such C statements or functions support rotations. In C, only shift operations (denoted by **<<**) are supported.

12.3 INTERFACE TO MULTIPLE 7-SEGMENT LEDS

In Chapter 11, an assembly language program to interface the 8051 to multiple 7-segment LEDs was presented. Specifically, the program was to copy BCD digits stored in internal RAM locations 70H to 71H, and send them to the LED display 10 times per second using interrupts. What follows will be a rough outline of a possible program written in C. It is left to the reader to fine-tune and adapt it according to his individual preferences.

EXAMPLE 12.2 Interface to Multiple 7-Segment LEDs

Rewrite in C language the software for the multiple 7-segment LED interface in Figure 11-6.

Solution

```
#include <reg5l.h>
#include <stdio.h>

unsigned char bdata A;          /* represents ACC */
int xdata * idata DPTR;         /* DPTR in idata, points to
                                   xdata */
sbit DIN    = P1^0;             /* MC14499 interface lines */
sbit CLOCK  = P1^6;
sbit ENABLE = P1^5;
sbit aLSB = A^0;
sbit aMSB = A^0;
int xdata X8155 = 0x100;        /* 8155 address */
int xdata TIMER = X8155 + 4;    /* timer registers */
int count = 4000;               /* interrupts @ 2000 ps */
unsigned char mode = 0x40;      /* timer mode hits */

int digits, icount;
unsigned char tempA;

void UPDATE(void);
void OUT8(void);
bit RLC_A(bit);

main( )
{
DPTR = TIMER;                  /* initialize 8155 timer */
A = 0xA0;                      /* low byte of count for 500 Hz
                                  square wave */
DPTR[TIMER] = A;
TIMER++;
A = 0xF;                       /* high byte of count */
DPTR[TIMER] = A;
A = 0xC0;                   /* start timer command */
DPTR[X8155] = A;            /* send to 8155 command register */
icount = 50;                /* initialize int. counter */
EA  = 1;                    /* enable  interrupts */
EX0 = 1;                    /* enable  external 0 interrupt */
IT0 = 1;                    /* negative-edge triggered */
while(1);                   /* do nothing */
}

void EX0ISR(void) interrupt 0
{
if (--icount == 0)          /* on 50th interrupt */
    {
     icount = 50;           /* reset counter and */
     UPDATE();              /* refresh LED display */
    }
}

void UPDATE(void)
{
tempA = A;                  /* save A */
ENABLE = 0;                 /* prepare MC14499 */
A = digits;                 /* get first two digits */
OUT8();                     /* send two digits */
A = digits + 1;             /* get second byte */
OUT8();                     /* send last two digits */
ENABLE = 1;                 /* disable MC14499 */
A = tempA;                  /* restore A */
}

void OUT8(void) using 0
{
for (tempA = 8; tempA > 0; tempA--)
        {
        CY = RLC_A(CY);     /* put bit in C flag */
        DIN = CY;           /* send it to MC14499 */
        CLOCK = 0;          /* 3 ps low pulse on clock line */
        CLOCK = 1;
        }
}

bit RLC_A(bit C)
{
bit tempBit;

tempBit = C;                /* backup C */
C = aMSB;                   /* shift A.0 into C */
A = A << 1;                 /* shift A left */
aLSB = tempBit;             /* shift C into A.0 */
return C;
}
```

Discussion

In the 8051, the DPTR is used as a pointer to point to external memory locations. In order to directly relate our C program to its assembly language counterpart, we use the declaration int **`xdata * idata DPTR`** to mean that the **`DPTR`** is a 16-bit pointer stored in idata memory (where SFRs are located) which points to a location in **`xdata`** memory. In assembly language, elements in lookup tables are obtained by the instruction **`MOV @DPTP,A`** or **`MOVC @DPTP,A,`** where the DPTR holds the base (starting) address of the table while A is the index to a specific element. The situation is similar in C. A pointer, which we can call DPTR, can be used to point to the Hirst element of an array (which is the C equivalent of a lookup table). Then, the statement **`DPTR[A]`** would refer to a specific element in the array, where A is the index.

12.4 INTERFACE TO LIQUID CRYSTAL DISPLAYS (LCDS)

Section 11.5 presented an interface example involving an LCD. The program that was written continually retrieved ASCII characters from internal RAM locations 30H-7FH and displayed them on the LCD, 16 characters at a time. The same program, rewritten in C, is shown in Example 12.3.

EXAMPLE 12.3 Interface to LCDs

Rewrite in C language the software for the LCD interface in Figure 11-8.

Solution

```
#include <reg51.h>
#include <stdio.h>

sbit PS = P3^0;
sbit RW = P3^1;
sbit E  = P3^2;
sbit busy = P1^7;
unsigned char bdata A;   /* represents ACC */
unsigned char * ptr;
unsigned char count;
void INIT(void);
void NEW(void);
void DISP(void);
void WAIT(void);
void OUT(void);

main( )
{
INIT();                  /* initialize LCD */
count = 16;              /* initialize character count */
while(1)
       {
       for (ptr = 0x30; ptr < 0x80; ptr++, count--)
       }
```

```
      A = *ptr;           /* get next character */
      DISP();             /* display on LCD */
      if (count==0)
             {
              count = 16; /* if end of line, reinitialize count */
             NEW();       /* and refresh LCD */
             }
      }
   }
}

void INIT(void)
{
A= 0x38;                  /* 2 lines, 5 X 7 matrix */
WAIT();                   /* wait for LCD to he free */
RS = 0;                   /* output a command */
OUT();                    /* send it out */
A = 0x0E;                 /* LCD on, cursor on */
WAIT();                   /* wait for LCD to he free */
RS = 0;                   /* output a command */
OUT();                    /* send it out */
NEW();                    /* refresh LCD display */
}

void NEW(void)
{
A = 0x80;                 /* clear LCD */
WAIT();                   /* wait for LCD to he free */
RS = 0;                   /* output a command */
OUT();                    /* send it out */

A = 0x80;                 /* cursor: line 1, position 1 */
WAIT();                   /* wait for LCD to he free */
RS = 0;                   /* output a command */
OUT();                    /* send it out */
}

void DISP(void)
{
WAIT();                   /* wait for LCD to be free */
RS = 1;                   /* output a data */
OUT();                    /* send it out */
}

void WAIT(void)
{
do
      {
      RS   = 0;           /* command */
      RW   = 1;           /* read */
      busy = 1;           /* make busy bit = input */
      E = 1;                /* l-to-0 transition to */
      E = 0;                /* enable  LCD */
      }
while (busy);               /* if busy, wait */
}
void OUT(void)
{
P1 = A;                     /* get ready output to LCD */
RW = 0;                     /* write */
E  = 1;                     /* l-to-0 transition to */
E  = 0;                     /* enable LCD */
}
```

Discussion

In this example, we need to access some consecutive internal RAM locations. In assembly language, this would be done via indirect addressing. In C, we make use of pointers and since the pointed locations should be in internal RAM, our pointer, say ptr should be defined with the statement **`unsigned char data * ptr`**. We use **`unsigned char`** here since memory locations contain 8-bit values. The rest of the program is straightforward.

12.5 LOUDSPEAKER INTERFACE

Recall that in Section 11.6 in the previous chapter, an interrupt-driven assembly language program to continually play an A-major musical scale was presented. The following Example 12.4 shows the corresponding program in C.

EXAMPLE 12.4 **Loudspeaker Interface**

Rewrite in C language the software for the loudspeaker interface in Figure 11-10.

Solution

```
#include <reg51.h>
#include <stdio.h>

#define LENGTH 12

sbit outbit = P1^7;
int reload;                 /* use this to temporarily store the
                               reload */
                            /* value for the current note    */
int code * PC;              /* PC points to the TABLE */
int REPEAT = 5;             /* reload value = -50000 causes 0.05
                               sec per */
                            /* timeout. Do 5 times, to get 5 x
                                0.05   */
                            /*   = 0.25 seconds per note    */
int ncount, tcount;         /* note counter & timeout counter */
int i, j;
                            /* look-up table of notes in A
                               major scale */
int code TABLE[LENGTH] = {-1136, -1136, -1012, -902, -851,
                           758, -676,
                            -602, -568, -568, -568, -568};
void GETVAL(void);
main()
{
TMOD = 0x11;                /* both timers 16-bit mode */
ncount = 0;                 /* initialize note counter to 0 */
tcount = REPEAT;            /* initialize timeout counter to 5
                               */
IE = 0x8A;                  /* timer 0 & 1 interrupts on */
TF1 = 1;                    /* force Timer 1 interrupt */
TF0 = 1;                    /* force Timer 0 interrupt */
while(l);                   /* ZzZzZzZz time for a nap */
}

void T0ISR(void) interrupt 1
{
TR0 = 0;                    /* stop timer */
TH0 = 0x3C;                 /* HIGH(-50000) */
TL0 = 0xB0;                 /* LOW(-50000) */
if (--tcount == 0)
      {
      tcount = REPEAT;      /* if 5th int, reset */
      ncount++;             /* increment note */
      if (ncount == LENGTH) /* if beyond last note... */
            ncount = O;     /* reset, A = 440Hz */
      }
TR0 = 1;                    /* start timer, go hack to ZzZzZzZ
                               */
}

void T0ISR(void) interrupt 3
{
outbit = !outbit;           /* music maestro! */
TR1 = 0;                    /* stop timer */
reload = ncount;            /* get note counter */
GETBYTE();                  /* get 2 bytes of reload value */
TH1 = reload >> 8;          /* put high byte in timer high
                               register */
TL1 = reload & 0xFF;        /* put low byte in timer low
                               register */
TR1 = 1;                    /* start timer */
}

void GETVAL(void)           /* tahle look-up function */
```

```
{
PC = TABLE;                 /* point to TABLE */
reload = PC[reload];        /* read from TABLE into reload */
}
```

Discussion

The assembly language solution in Figure 11-10 uses a lookup table in code memory. To do the same thing in C, we use an array that is stored in code memory. In our solution above, this array is called TABLE. The task of obtaining elements from this array is performed by a function **`GETVAL( )`** that makes use of a pointer PC to access the array elements. Notice that in contrast to the assembly language solution in Figure 11-10**,** we do not have to read from the table two times. This is because in C, you can define lookup tables with 16-bit elements to store each reload value as an element within it, as has been done in our example above. Thus, we only read once from the table to get the 16-bit reload value. Then, we shift it to the right by 8 bits to get its high byte, and simply mask off the higher 8 bits (by ANDing them with 0) to get its low byte.

12.6 NONVOLATILE RAM INTERFACE

It was shown in Section 11.7 how a nonvolatile RAM (NVRAM) retains its contents even during the absence of power. An interface example was also given on how to copy contents from the 8051 to the NVRAM and to later read back the values stored. The program in C is given by following Example 12.5.

EXAMPLE 12.5 **NVRAM Interface**

Rewrite in C language, the software for the NVRAM interface in Figure 11-13.

Solution

```
#include <reg51.h>
#include <reg5l.h>
#include <stdio.h>
#define RECALL 0x85      /* X2444 recall instruction */
#define WRITE  0x84      /* X2444 write enable  instruction */
#define STORE  0x81      /* X2444 store instruction */
#define SLEEP  0x82      /* X2444 sleep instruction */
#define W_DATA 0x83      /* X2444 write data instruction */
#define R_DATA 0x87      /* X2444 read data instruction */
#define LENGTH 32        /* 32 hytes saved/restored */

unsigned char * R0;      /* used to point to NVRAM locations */
unsigned char R5, R6, R7;
unsigned char bdata A;   /* represents accumulator */
sbit aMSB = A^7;         /* variahle aMSB to refer to A.7 */
sbit aLSB = A^0;         /* variable aLSB to refer to A.0 */
sbit DIN  = P1^2;        /* X2444 interface lines */
sbit ENABLE = P1^1;
sbit CLOCK  = P1^0;
bit C;
unsigned char NVRAM[LENGTH] _at_ 0x60;
void SAVE(void);
void RECOVER(void);
void W_BYTE(void);
void W_BYTE(void);
void RL_A(void); bit
RLC_A(bit);

main()
{
while(1)                    /* repeat forever */
       {
       SAVE();              /* copy from 8051 internal
                               locations 60H-7FH */
                             /*   to X2444 EPROM      */
       RECOVER();           /* read previously saved data from
                               X2444 EPROM to */
                             /*   locations 60H-7FH      */
       }
}

void SAVE(void)
{
R0 = NVRAM;                 /* R0 point to locations to save
                               to */
ENABLE = 0;                 /* disable X2444 */
A = RECALL;                 /* recall instruction */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
A = WRITE;                  /* write enable  prepares X2444 to
                               he written to */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
for (R7 = 0; R7 < 16; R7++)/* R7 = X2444 address */
       {
       A = R7;              /* put address in A */
       RL_A();              /* put in hits 3, 4, 5, 6 */
       RL_A();
       RL_A();
       A = A | W_DATA;      /* build write instruction */
       ENABLE = 1;
       W_BYTE();
       for (R5 = 2; R5 > 0; R5--)
           {
```

```
          A = *R0;        /* get 8051 data */
          R0++;           /* point to next byte */
          W_BYTE();       /* send byte to X2444 */
          }
   ENABLE = 0;
}
A = STORE;                /* if finished, copy to EPROM */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
A = SLEEP;                /* put X2444 to sleep */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
}

void RECOVER(void)
{
R0 = NVRAN;
ENABLE = 0;
A = RECALL;               /* recall instruction */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
for (R7 = 0; R7 < 16; R7++)/* R7 = X2444 address */
      {
      A = R7;             /* put address in A */
      RL_A();             /* build read instruction */
      RL_A();
      RL_A();
      A = A | R_DATA;
      ENABLE = 1;
      W_BYTE();           /* send read instruction
         */
         for (R5 = 2; R5 > 0; R5--)
            {
            R_BYTE();     /* read byte of data */
            *R0 = A;      /* put in 8051 RAM */
            R0++;         /* point to next location */
            }
      ENABLE = 0;
}
A = SLEEP;                /* put X2444 to sleep */
ENABLE = 1;
R_BYTE();
ENABLE = 0;
}

void R_BYTE(void)
{
for (R6 = 8; R6 > 0; R6--) /* use R6 as bit counter */
```

```
      {
      C = DIN;            /* put X2444 data bit in C */
      C = RLC_A(C);       /* build byte in Accumulator */
      CLOCK = 1;          /* toggle clock line (l us) */
      CLOCK = 0;
      }
}

void W_BYTE(void)
{
for (R6 = 8; R6 > 0; R6--) /* use R6 as bit counter */
      {
      C = RLC_A(C);       /* put bit to write in C */
      DIN = C;            /* put in X2444 DATA IN line */
      CLOCK = 1;          /* clock bit into X2444 */
      CLOCK = 0;
      }
}

void RL_A(void)
{
bit tempBit;

tempBit = aMSB;           /* backup MSB of A */
A = A << 1;               /* rotate A left */
aLSB = tempBit;           /* rotate LSB into MSB */
}

bit RLC_A(bit C)
{
bit tempBit;

tempBit = C;              /* backup C */
C = aMSB;                 /* shift ACC.7 into C */
A = A << 1;               /* shift ACC left */
aLSB = tempBit;           /* shift C into ACC.0 */
return C;
}
```

Discussion

This program requires that 32 bytes be allocated in internal data memory locations from 60H-7FH. This is achieved by declaring an array, NVRAM of 32 byte elements, and specifying that it should start at absolute address 60H. The main program continually performs the saving of data bytes from internal data locations 60H-7FH to the NVRAM, and the restoring of previously saved data from the NVRAM to the internal data locations 60H-7FH. Elements are read from the NVRAM array indirectly by way of a pointer, R0. Data are sent serially to and from the NVRAM, and the functions **`W_BYTE ( )`** and **`R_BYTE ( )`** are respectively used for that purpose.

12.7 INPUT/OUTPUT EXPANSION

In the previous chapter, two ways to expand the I/O were demonstrated in assembly language. The following two examples show how the same thing can be done in C. In particular, Example 12.6 shows the method of I/O expansion by using shift registers, while Example 12.7 shows how to expand the I/O by using the 8255.

EXAMPLE 12.6 **Interface to Shift Registers**

Rewrite in C language the software for the Shift Register interface in Figure 11-16.

Solution

```
#include <reg51.h>
#define COUNT 2                       /* numher of shift registers */
unsigned char bdata * R0;
unsigned char R6, R7;
unsigned char xdata * idata DPTR;    /* DPTR in data, points to xdata */
unsigned char bdata A;                /* represents ACC */
sbit aMSB   = A^7;
sbit aLSB   = A^0;
sbit SHIFT = P1^7;                    /* SHIFT/LOAD input: 1 = shift,
                                         0 = load */
sbit CLOCK = P1^6;                    /* CLOCK input */
sbit DOUT  = P1^5;                    /* DATA OUT output */
bit C;
bit PY;                               /* represents PARITY bit */

char * BANNER = {"*** TEST 74HC165 INTERFACE ***\n"};
unsigned char bdata BUFFER[COUNT]; /* buffer to store hytes read */
void GET_BYTES(void);
void SEND_HELLO_MESSAGE(void);
void DISPLAY_RESULTS(void);
bit RRC_A(bit C);
void OUTSTR(void);
void OUTCHR(void);
void OUT2HEX(void);
void SWAP_A(void);
void PARITY(void);
void HTOA(void);
main()
{
CLOCK = 1;                            /* set interface lines initially in   */
SHIFT = 1;                            /* ... case not already */
DOUT = 1;                             /* DOUT must he set (input ) */
SEND_HELLO_MESSAGE();                 /* banner message */
while(l)                              /* loop forever */
        {
        GET_BYTES();                  /* read shift registers */
```

```
             DISPLAY_ RESULTS();                /* show results */
      }
}

void GET_BYTES(void)
{
for (R6 = COUNT; R6 > 0; R6--) /* use R6 as byte counter */
      {
      R0 = 0x25;                /* use R0 as pointer to buffer in bdata */
      SHIFT = 0;                /* load into shift registers hy pulsing... */
      SHIFT = 1;                /* ... SHIFT/LOAD low */
      for (R7 = 8; R7 > 0; R7--)/* use R7 as bit counter */
             {
             C = DOUT;          /* get a bit (put it in C) */
             C = RRC_A(C);      /* put in A.0 (LSB 1st) */
             CLOCK = 0;         /* pulse CLOCK line (shifts hits towards  */
             CLOCK = 1;         /* ... DATA OUT */
             }
      *R0 = A;                  /* if 8th shift, put in buffer */
      R0++;                     /* increment pointer to buffer */
      }
}

void SEND_HELLO_MESSAGE(void)
{
DPTR = BANNER;                  /* point to hello message */
OUTSTR();                       /* send it to console */
}

void DISPLAY_RESULTS(void)
{
R0 = 0x25;                      /* R0 points to bytes */
for (R6 = COUNT; R6 > 0; R6--)  /* use R6 as byte counter */
      {
      A = *R0;                  /* get byte */
      R0++;                     /* increment pointer */
      OUT2HEX();                /* output as 2 hex character */
      A = ' ';                   /* separate hytes */
      OUTSTR();
      }                          /* repeat for each byte */
A = '\n';                        /* begin a new line */
OUTCHR();
}

bit RRC_A(bit C)
{
bit tempBit;

tempBit = C;                /* backup C */
C = aLSB;                   /* rotate A.0 into C */
A = A >> 1;                 /* rotate A right */
```

```
aMSB = tempBit;             /* rotate C into A.7 */
return C;
}

void OUTSTR(void)
{
while (1)
      {
      A = 0;
      A = DPTR[A];          /* get ASCII code */
      if (A == 0)           /* if last code, done */
            break;
      OUTCHR();             /* if not last code, send it */
      A++;                  /* point to next code */
      }
}

void OUTCHR(void)
{
PARITY();                   /* get even parity of A and put in AY */
PY = !PY;                   /* change to odd parity */
aMSB = PY;                  /* add to character code */
while (TI != 1);            /* Tx empty? no: check again */
TI = 0;                     /* yes: clear flag and */
SBUF = A;                   /* send character */
aMSB = 0;                   /* strip off parity bit */
}

void OUT2HEX(void)
{
unsigned char tempA = A;    /* save A in tempA */
SWAP_A();                   /* send high nihble first */
A = A & 0xf;                /* mask off unwanted nihhle */
HTOA();                     /* convert hex nihble to ASCII */
OUTCHR();                   /* send to serial port */
A = tempA;                  /* restore A and send low nibble */
A = A & 0xf;
HTOA();
OUTCHR();
A = tempA;
}

void SWAP_A(void)
{
A = (A >> 4) | (A << 4);    /* swap upper and lower nibbles of A */
}

void PARITY(void)
{
int i;
PY = 0;                     /* initialize parity to 0 */
```

```
for ( i = 0; i < 8 ; i++ ) /* calculate parity of A*/
PY ^= (A >>  i ) & 1;
}

void HTOA(void)
{
A = A & 0xF;                  /* ensure upper nibhle clear */
if ( A >= 0xA )               /* 'A' to 'F'? */
      A = A + 7;              /* yes: add extra */
A = A + '0';                  /* no: convert directly */
}
```

Discussion

The program first sends out a hello message to the display device attached to the serial port. This essentially involves pointing to the message and calling the **OUTSTR( )**, which in turn calls the **OUTSTR( )** function to send the message a character at a time. The function **GETEYTES()** is then called to get two bytes from the shift register. These are immediately displayed by calling the **DISPLAY_RESULTS( )** function. **GETBYTES( )** basically shifts in the byte values bit by bit. Meanwhile, **DISPLAY_RESULTS( )** makes use of the **OUT2HEX( )** function to convert the read byte values from hexadecimal to ASCII format, before calling **OUTSTR( )** to send the ASCII characters out for display.

The design example in Section 11.8.2 of the previous chapter discussed how one could use the 8255 to add three additional 110 ports. As an illustration, an assembly language program was written that read the status of eight switches connected to Port A, and for each closed switch, a corresponding LED connected to one of the Port B pins is lighted. The C program is as below:

EXAMPLE 12.7 Interface to an 8255

Rewrite in C language the softwazre for the 8255 interface in Figure 11-18.

Solution

```
#include <reg51.h>
#include <stdio.h>
unsigned char xdata * idata DPTR; /* DPTR in data, points to
                                     xdata */
unsigned char hdata A;            /* represents ACC */
sbit a_l = A^0;
sbit a_1 = A^1;
sbit a_2 = A^2;
sbit a_3 = A^3;
sbit a_4 = A^4;
sbit a_5 = A^5;
sbit a_6 = A^6;
sbit a_7 = A^7;

void CPL_A();
main()
```

```
{
A = 0x90;                 /* Port A = input, Port B = output */
DPTR = 0x0103;            /* point to control register */
*DPTR = A;                /* send control word */
while(1)
      {
      DPTR = 0x0100;      /* point to Port A */
      A = *DPTR;          /* read switches */
      CPL_A();            /* complement */
      DPTR = 0x0101;      /* point to Port E */
      *DPTR = A;          /* light up LEDs */
      }
}

void CPL_A(void)
{
a_0 = (!a_0);
a_1 = (!a_1);
a_2 = (!a_2);
a_3 = (!a_3);
a_4 = (!a_4);
a_5 = (!a_5);
a_6 = (!a_6);
a_7 = (!a_7);
}
```

Discussion

This example also uses the pointer DPTR to access external memory locations. The operation of complementing the entire 8-bit content in A is done by the function **`CPL_A( )`**, a straightforward function that simply complements each bit within A.

12.8 RS232 (EIA-232) SERIAL INTERFACE

Connecting the 8051 to a PC via the RS232 serial interface was discussed in Section 11.9. The design example was to write a program to input decimal numbers from the PC and then send the corresponding ASCII code back to the PC screen. The C program to do this is given in Example 12.8.

EXAMPLE 12.8 **Interface to RS232**

Rewrite in C language the software for the RS232 interface in Figure 11-20.

Solution

```
#include <reg5l.h>
#include <stdio.h>
unsigned char bdata A;            /* represents ACC */
sbit aMSB = A^7;                  /* variable aMSB to refer to A.7  */
sbit RTS = P1^7;                  /* variable RTS to refer to P1.7  */
sbit CTS = P1^6;                   /* variable CTS to refer to P1.6 */
bit C;                             /* represents carry bit */
unsigned char code * idata DPTR;   /* DPTR in data, points to code */
unsigned char code ASC[10] ={0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36,
                            0x37,0x38, 0x39;    /* ASCII table */
char code * MSG = {"PLEASE 1NTER NUMBER = "};  /* initial message */
void FACE(void);
void INCHAR(void);                 /* taken & modified from Example 8-7 */
void OUTCHR(void);              /* taken & modified from Example 8-6 */
void PARITY(void);
main()
{
FACE();                            /* initialize serial port & do handshake */
DPTR = ASC;                        /* point to ASCII table */
while(1)
      {
      INCHAR();                    /* get decimal number and put in A */
      A = DPTR[A];                 /* convert to ASCII and store back in A */
      OUTCHR();                    /* output ASCII to serial port */
      }
}

void FACE(void)
{
TMOD = 0x20;                       /* timer 1, mode 2 */
TH1 = 0x98;                        /* reload count for 9600 baud */
SC0N = 0x52;                       /* serial port, mode 1 */
DPTR = MSG;                        /* pointer to initial message */
TR1 = 1;                           /* start timer 1 */
RTS = 0;                           /* assert RTS */
while (CTS == 1);                  /* wait for CTS */
do
      {
      A = DPTR[A];                 /* get ASCII characters */
      OUTCHR();                    /* send out */
      DPTR++;                      /* if not end of message, get next character */
      }
while (A != 0);                    /* else if end of message, stop */
}

void INCHAR(void)
{
while (RI != 1);                   /* wait for character */
RI = 0;                            /* clear flag */
A = SBUF;                        /* read char into A */
PARITY();                        /* get even parity of A and put in AY */
C = PY;                          /* for odd parity in accumulator, PY should
                                    be set */
C = !C;                          /* complementing correctly indicates if
                                    "error" */
aMSB = 0;                        /* strip off parity */
}
void OUTCHR(void)
{
PARITY();                        /* get even parity of A and put in PY */
PY = !PY;                        /* change to odd parity */
aMSB = PY;                       /* add to character code */
while (TI != 1);                 /* Tx empty? no: check again */
TI = 0;                          /* yes: clear flag and */
SBUF = A;                        /* send character */
aMSB = 0;                        /* strip off parity bit */
}
void PARITY(void)
{
int i;
PY = 0;                          /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )       /* calculate parity of A*/
      PY ^= ( A >> i ) & 1;
}
```

Discussion

This example employs familiar concepts such as using a temporary bit-addressable variable, A, to represent the accumulator, using an array in code memory to represent a lookup table and using the pointer, DPTR, to access the lookup table. Besides that, we recall that in assembly language, we sometimes wish to store data in consecutive locations in code memory by using the DE or DW assembler directives. This can be implemented in C by using an array that is stored in code memory.We also make use of the function **INCHAR( )**, which is similar to the **INCHAR( )** function discussed in Chapter 8, except that since A is used instead of the accumulator, we need to compute the parity ourselves by calling the **PARITY( )** function, causing the parity to be stored in PY.

12.9 CENTRONICS PARALLEL INTERFACE

Printers interact with computers via the parallel port, and the common standard used is called the Centronics parallel interface. We demonstrated in the previous chapter a program to check the status of the printer before continually sending a test message to it for printing. Example 12.9 shows just how this can be done in C.

EXAMPLE 12.9 **Interface to Centronics Parallel Interface**

Rewrite in C language the software for the parallel interface in Figure 11-22.

Solution

```
 #include <reg5l.h>
 #include <stdio.h>

 unsigned char bdata A;                 /* represents ACC */
 unsigned char MASK = 0x3C;             /* only check 4 of the bits */
 unsigned char OK = 0x30;               /* normal values of the 4 status bits */
 sbit STR = P3^0;
 sbit ACK = P3^1;
 sbit BUSY = P3^2;
 char code * MSG = {"THIS IS A TEST FOR THE PRINTER"}; /* test msg */
 unsigned char code * idata DPTR;       /* DPTR in data, points to code */
 main()
 {
 while (1)
 {
      DPTR = MSG;                       /* point to test message */
      do
       {
       P3 = MASK;                       /* activate STROEE, P3.1-P3.5 as input */
       A = P3;                          /* read printer status */
       A = A & MASK;                    /* only P3.1-P3.5 are wanted */
       if (A != OK)                     /* any error? */
             return;                    /* yes: stop */
       A = 0;                           /* no error, get ready to send */
       while (ACK == 1);                /* before send, wait for ACK */
       A = DPTR[A];                     /* get char in test message */
       P1 = A;                          /* send char to printer */
       DPTR++;                          /* if not end, get next character */

       while ( A != 0 );                /* else if end of message, stop */
       }
}
```

Discussion

Here, the 8-bit variable **MASK** is used to store the masking value, specifying which bits to be masked off (ignored) by being cleared to zero. Bear in mind that a logical AND operation is denoted by one ampersand symbol "&" and not two. Two ampersands "&&" would denote a relational AND operator.

12.10 ANALOG OUTPUT

Interfacing to an analog device often requires the use of ADCs and DACs. In Section 11.11, it was shown how to connect the 8051 to the MC1408L8 DAC to generate an analog sine wave by using a lookup table. Here we will present the corresponding C program.

EXAMPLE 12.10 **Analog Output**

Rewrite in C language the software for the DAC interface in Figure 11-25.

Solution

```
#include <reg51.h>

#define MAX 1024
                        /* truncated TABLE of 1024 entries */
unsigned char TABLE[MAX] = {127, 128, 129, 130, 131, ...};

int data STEP = 1;      /* can he initialized to any increment value */
unsigned char A;        /* represents ACC */
int index = 0;          /* use to point to entries in TABLE */
main( )
{
TMOD = 0x2;             /* 8-bit, auto reload */
TH0 = -100;             /* 100 ms delay */
TR0 = 1;                /* start timer */
IE = 0x82;              /* enable timer 0 interrupt */
while(1);               /* main loop does nothing! */
}

void T0ISR (void) interrupt 1
{
index = index + STEP;   /* add STEP to index */
if ( index > MAX )
      index = 0;        /* if end of TAELE, back to beginning */
A = TABLE[index];       /* get entry */
P1 = A;                 /* send it */
}
```

Discussion

The main program initializes the Timer 0, enables the corresponding Timer 0 interrupt, and then does nothing. The bulk of the program lies in the Timer 0 interrupt function which, upon every occurrence of the interrupt, accesses an entry from **`TABLE[]`**, which is an array of 1024 entries between 0 and 255 that correspond to the magnitudes of one period of a sine wave. (Notice that in the program above, **`TAELE[]`** is shown in truncated form.) When the end of the table is reached, the table index is reinitialized to 0 so that the next entry obtained would again be starting from the beginning of the table. Prior to the running of this program, a separate program similar to that in Figure 11-24 is used to generate the 1024 entries of **`TABLE[]`.**

12.11 ANALOG INPUT

Figure 11-27 demonstrated how an assembly language program is used to use an ADC to read and convert the voltage at a trimpot's center tap into digital form so that the corresponding ASCII code can be output to the console. Given here is the program rewritten in C.

EXAMPLE 12.11 **Analog Input**

Rewrite in C language the software for the ADC interface in Figure 11-27.

Solution

```
#include <reg51.h>
#define PORTA 0x101          /* 8155 Port A */

int xdata * idata DPTR;      /* DPTR in idata, points to xdata */
unsigned char bdata A;       /* represents ACC */
sbit aMSB = A^7;             /* variable aMSE to refer to A.7 */
sbit aLSB = A^0;             /* variable aLSE to refer to A.0 */
sbit WRITE = P1^0;           /* ADC0804 WR line */
sbit INTR = P1^1;            /* ADC0804 INTR line */
bit PY;                      /* represents parity bit */

char * BANNER = ("*** TEST ADC0804 ***\n"};
void OUTSTR(void);
void 0UTCHR(void);
void OUT2HEX(void);
void SWAP_A(void);
void PARITY(void);
void HTOA(void);

main()
{
DPTR = BANNER;               /* send message */
OUTSTR();
while (1)
      {
      WRITE = 0;             /* toggle WR line */
      WRITE = 1;
      while (INTR == 1);     /* wait for INTR = 0 */
      DPTR = PORTA;          /* knit DPTR -> Port A */
      A = *DPTR;             /* read ADC0804 data */
      OUT2HEX();             /* send data to console */
      }
}
```

```
void OUTSTR(void)
{
while (1)
        {
        A = 0;
        A = DPTR[A];        /* get ASCII code */
        if (A == 0)         /* if last code, done */
              break;
        OUTCHR();           /* if not last code, send it */
        A++;                /* point to next code */
}

void OUTCHR(void)
{
PARITY();                   /* get even parity of A and put in
                              AY */
PY = !PY;                   /* change to odd parity */
aMSB = PY;                  /* add to character code */
while (TI != 1);            /* Tx empty? no: check again */
TI = 0;                     /* yes: clear flag and */
SBUF = A;                   /* send character */
aMSB = 0;                   /* strip off parity bit */
}

void OUT2HEX(void)
{
unsigned char tempA = A;    /* save A in tempA */
SWAP_A();                   /* send high nibble first */
A = A & 0xf;                /* mask off unwanted nibble */
HTOA();                     /* convert hex nibble to ASCII */
OUTCHR();                   /* send to serial port */
A = tempA;                  /* restore A and send low nibble */
A = A & 0xf;
HTOA();
OUTCHR(); A = tempA;
}

void SWAP_A(void)
{
A = (A >> 4) | (A << 4);    /* swap upper and lower nibbles of
                               A */
}

void PARITY(void)
{
int i;
Y = 0;                      /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )  /* calculate parity of A*/
PY ^= ( A >> i ) & 1;
}
```

```
void HTOA(void)
{
A = A & 0xF;                /* ensure upper nibble clear */
if ( A >= 0xA )             /* 'A' to 'F'? */
     A = A + 7;             /* yes: add extra */
A = A + '0';                /* no: convert directly */
}
```

Discussion

The main program first displays a message to the console, and then toggles the WR line to begin analog-to-digital conversion. It then waits for the ADC's INTR line to go low indicating that the conversion is complete before it reads data from the ADC via the 8155's port A. The data is next converted to hexadecimal format before being sent out to the console for display.

12.12 INTERFACE TO SENSORS

In this section, we again discuss how the 8051 can interact with a typical sensor, the temperature sensor. Recall that in Chapter 11, we presented an assembly language program to monitor the room temperature by using the DS1620 temperature sensor. When the room temperature is higher than 23°C, the furnace is turned on and the alarm sounds. If the temperature falls below 17°C, the furnace is switched off and the alarm is switched off. We show in Example 12.12 the corresponding C example to accomplish this.

EXAMPLE Interface to DS1620

12.12 Rewrite in C language the software for the sensor interface in Figure 11-29.

Solution

```
#include <reg5l.h>
#include <stdio.h>
unsigned char bdata A;     /* represents ACC */
sbit aMSB = A^7;           /* variable aMSB to refer to A.7 */
sbit aLSB = A^0;           /* variable aLSB to refer to A.0 */
sbit DQ = P1^0;
sbit CLK = P1^1;
sbit RST = P1^2;
sbit THI = P1^3;
sbit TLO = P1^4;
sbit TCOM = P1^5;
sbit FURN = P1^6;
int i;
void SEND(void);
bit RRC_A(bit);
```

```
main()
{
FURN = 0;                    /* turn furnace off */
RST = 1;                     /* initiate transfer */
A = 0xC;                     /* write config */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 1;                       /* write TH */
SEND();                      /* send to DS1620 */
A = 44;                      /* TH = 44 X 0.5 °C = 22 °C */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 2;                       /* write TL */
SEND();                      /* send to DS1620 */
A = 36;                      /* TL = 36 X 0.5 °C = 18 °C */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 0xEE;                    /* start temperature sensing */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
while(1)                     /* keep sensing forever */
       {
       if (THI == 1)         /* if T>=22 °C, furnace = off */
            FURN = 0;
       if (TLO == 1)         /* if T<=18 °C, furnace = on */
            FURN = 1;
       }
}

void SEND(void)
{
for (i = 8; i > 0; i--)   /* loop for all 8 bits */
       {
       CLK = 0;              /* start clock cycle */
       CY = RRC_A(CY);       /* rotate A into CY, LSB first */
       DQ = CY;              /* send out bit to DB */
       CLK = 1;              /* complete the clock cycle */
       }
}
```

```
bit RRC_A(bit C)
{
bit tempBit;
tempBit = C;            /* backup C */
C = aLSB;               /* rotate A.0 into C */
A = A >> 1;             /* rotate A right */
aMSB = tempBit;         /* rotate C into A.7 */
return C;
}
```

Discussion

Most of the functionality in this example is done by the function **send()**, which serially (starting with the LSB) sends an 8-bit data to the port pin connected to the DQ (data) input of the sensor.

12.13 INTERFACE TO RELAYS

In the last chapter, a very interesting application of relays was presented, that of a pedestrian traffic light system. The 8051 electrically controlled the relay to switch between two states, either to turn on the RED traffic light and GREEN pedestrian light in order to allow pedestrians to cross the road, or to turn on the GREEN traffic light and RED pedestrian light. The program written in C to control all this is given in Example 12.13.

EXAMPLE 12.13 **Interface to a Relay: Pedestrian Traffic Light System**

Rewrite in C language the software for the relay interface in Figure 11-33.

Solution

```
#include <reg51.h>
#include <stdio.h>

sbit LEDs = P1^0;
int thousand = 1000;  /* 1000 X 10000 us = 10 secs */
int count;            /* to store the count values */

void delay(void);

main( )

{
IE = 0x81;            /* enable  INT0 */
IT0 = 1;              /* negative edge triggered */
TMOD = 1;             /* timer 0 in mode 1 */
LEDs = 0;             /* initially, traffic = GREEN,
                         pedestrian = RED */
while (1);            /* wait forever */
}

void delay(void)      /* 10-second delay */
{
for (count = thousand; count > 0; count –)
        {
        TH0 = 0x6C;
        TL0 = 0x70;
        TR0 = 1;
        while(TF0 != 1);
        TF0 = 0;
        TR0 = 0;
        }
}

void EX0ISR(void) interrupt 0
{
LEDs = 1;             /* traffic light = RED, pedestrian =
                         GREEN */
delay();              /* wait 10 seconds */
LEDs = 0;             /* traffic light = GREEN, pedestrian =
                         RED */
}
```

Discussion

Comparing this program with its assembly language counterpart in the previous chapter, it would seem that they are very similar. The only difference lies in the amount of delay generated by the **`delay( )`** function, which is supposed to wait for 10 seconds to allow pedestrians to cross the road before the traffic and pedestrian lights switch back to their initial states. Nevertheless, the actual delay generated by this **`delay( )`** function depends on which 8051 C compiler is used. In order to determine the exact amount of delay generated, we should view the corresponding assembly language instructions generated by the compiler. In Keil's µVision2 IDE, this is done by choosing the Disassembly Window from the View menu.

12.14 STEPPER MOTOR INTERFACE

The stepper motor finds interesting applications where precision-positioning of components is required. We saw in Chapter 11 an example assembly language program that demonstrated how to initially rotate a stepper motor clockwise, and whenever a switch connected to external interrupt 0 (INT0) causes a high-to-low transition, the direction of rotation should be reversed. In this section, we present the equivalent C program.

EXAMPLE 12.14 **Interface to a stepper motor**

Rewrite in C language the software for the stepper motor interface in Figure 11-38.

Solution

```
#include <reg51.h>
#include <stdio.h>


unsigned char A;                  /* ACC mirror */
unsigned char tempA;              /* temp variable */
unsigned char code * idata DPTR; /* DATR in data, points to
                                     code */
bit D;                            /* direction bit for current
                                     direction */
int HUNDRED = 100;                /* 100 X 10000 us = 1 sec */
                                  /* 8-step sequence for
                                     clockwise rotation */
unsigned char code SEQ[8] = {0x9, 0x8, 0xC, 0x4, 0x6, 0x2,
0x3, 0x1};

void CW(void);
void CCW(void);
void delay(void);

main()
{
IE = 0x81;                        /* enable  INTO */
IT0 = 1;                          /* negative edge triggered */
TMOD = 1;                         /* timer 0 in mode 1 */
D = 0;                            /* initialize D = 0 for
                                     clockwise rotation */
CW();                             /* initial rotation is
                                     clockwise */
while(1)                          /* repeat forever */
      {
      if (D != 1)                 /* previously clockwise? */
            CCW();                /* no: change to clockwise */
      else
            CW();                 /* yes: change to
                                     counterclockwise */


void CW(void)                     /* function for clockwise
                                     rotation */
{
DATR = SEQ;                       /* point to start of table */
for (A = 0; A < 8; A++)
       {
       tempA = A;                 /* backup index in A */
       A = DPTR[A];        /* get step pattern into 4 LSBs of
                              A */
       A = A | (P1&0xF0); /* retain 4 MSBs of A1 and put into
                              A */
       P1 = A;              /* send to stepper motor */
       delay();            /* wait for 1 sec */
       A = tempA;          /* restore index into A */
       }
}
void CCW(void)             /*function for counterclockwise
                              rotation */
{
DPTR = SEQ;                /* point to start of tahle */
for (A = 7; A >= 0; A==)
       {
       tempA = A;          /* hackup index in A */
       A = DPTR[A];        /* get step pattern into 4 LSBs of
                              A */
       A = A | (P1&0xF0); /* retain 4 MSBs of P1 and put into
                              A */
       P1 = A;              /* send to stepper motor */
       delay();            /* wait for 1 sec */
       A = tempA;          /* restore index into A */
}
}

void delay(void)           /* 1 second delay */
{
for (tempA = HUNDRED; tempA > 0; tempA--)
       {
       TH0 = 0x6C;
       TL0 = 0x70;
       TR0 = 1;
       while(TF0 != 1);
       TF0 = 0;
       TR0 = 0;
       }
}

void EX0ISR(void) interrupt 0
{
D = !D;                    /* complement direction bit */
}
```

Discussion

The program basically consists of the **cw()** and **ccw()** functions that directly interact with the stepper motor and direct it to rotate clockwise and counterclockwise respectively. The only difference between these two functions, though, is just the sequence of step patterns being written to the stepper motor. In between the writing of a step pattern to the stepper motor, the **`delay( )`** function is called to give ample time for the stepper motor to digest each step pattern.

PROBLEMS

12.1 Rewrite the software for the LCD to continually display the message "Welcome to the 8051 Microcontroller Experience. Hope you enjoy your reading adventure."

12.2 Imagine that the stepper motor is used to automatically turn the knob of a safe. Based on a sequence of user-supplied numbers, the stepper motor turns the knob by the specified number of steps clockwise, then counter-clockwise, clockwise, etc. For example, if the sequence is 8, 0, 1, 6, 0, 9, 9, then the stepper motor first turns clockwise by 8 steps, turns counter-clockwise by 0 steps, turns clockwise again by 1 step, and so on. Write a function **`safe (int * seq, int seqsize)`** to receive as input a sequence of numbers that have been stored in an array of size seqsize, and then turns the stepper motor according to the input sequence. Assume the availability of the **`CW ()`** and the **`ccw ( )`** functions.

12.3 The 8051 microcontroller is commonly used in smart cards, which are cards not only with built-in memory but also a built-in microcontroller as its brain, making it capable of running programs from within itself. Some of the information contained in smart cards is confidential, hence there is a need for some form of protection from unauthorized access. This is usually done by encrypting the information. Encryption is the process of transforming confidential information (normally called the plaintext) into an unintelligible form (called the ciphertext). Once in this form, even if someone were to eavesdrop or spy on it, he would not be able to understand its meaning. The only way to get back the original information is to perform the reverse process, called decryption. The Caesar cipher is one method of encryption that dates back to the days of the Roman empire, and is named after its inventor Julius Caesar. Though simple by today's standards, and no longer in use to protect information, an understanding of how it works helps to give us a basic idea of encryption. A more detailed treatment of security and encryption will be given in Chapter 13.

Consider the English alphabet:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Imagine if we shifted all the alphabets 3 positions to the left, such that we obtain a new set of alphabets:

```
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

Then, given any confidential message, we replace each character from the first set of alphabets with a character from the second set of alphabets. For example, every BA' we see will be replaced with a 'D', every 'B' with an BE', every 'C' with an 'F' and so on.

So the message:

```
MEET ME TONIGHT
```

would be encrypted to:

```
PHHW PH WRQLJKW
```

Write a function CaesarEncrypt (char * plain, char * cipher) that takes in two message strings (arrays of characters), plain and cipher, respectively. Encrypt the message plain and save the encrypted message in cipher.

12.4 With reference to the previous question, write the corresponding function Caes - arDecrypt (char * plain, char * cipher) to decrypt a message stored in cipher and save the decrypted message in plain.

CHAPTER 13

Example Student Projects

13.1 INTRODUCTION

Students are often eager to do work on microcontroller and other robot-based projects. When assigned the task of constructing 8051-based projects, one often wonders in amazement at the enthusiasm and keen interest shown by the students. The student reader has also seen some design and interface problems in Chapters 11 and 12. As an extension of that, we will attempt in this chapter to outline potential student projects that would help to tie together all the concepts discussed in the earlier chapters of this book.

We will start with one of the simplest such projects: an 8051-based home security system. Other more advanced-level projects would include a simple elevator system, a Tic-TacToe game, an 8051-based calculator, a micromouse, a soccer-playing robot, and a smart card application.

13.2 HOME SECURITY SYSTEM

One of the simplest projects to build up the design skills in the student is the 8051-based home security system. This is basically a demonstration of how to interact with I/O devices through the 8051's I/O ports.

13.2.1 Project Description

A sample project description for such a home security system could be:

Design an 8051 microcontroller-based home security system that has 64 Kbyte external code memory and 64 Kbyte external data memory. Assume you only have 8 K EPROMs and 8 K RAMs available.

Your security system should be able to detect when any of these sections of the house have been opened or intruded by a burglar:

1. *gate*
2. *front door*
3. *back door*
4. *any window*

When one of them has been opened, turn on an alarm and also use a seven-segment display to display the corresponding house section. For example, if the gate is open, display 1. If any window is open, display 4.

13.2.2 System Specifications

The first thing for the student in tackling this project is to determine the required components for the system. Based on the project description, it is clear that the following are needed:

- 1 x 8051 microcontroller
- 8 x 8Kbyte ROM/EPROM/EEPROM
- 8 x 8Kbyte RAM
- 4 x sensors
- 1 x alarm
- 1 x 7-segment display

The requirement was that the system should have 64 Kbyte of external code memory, but since only 8 Kbyte ROM chips are available, we would have to make do with having eight of them in cascade. The same goes for implementing the external data memory. We need to detect intrusions at four different points of the house as outlined in the project description, hence we would need at least four sensors which could be infrared, light, or even switch sensors. Besides these, the project also requires the use of an alarm and a seven-segment display to indicate the point of intrusion. Notice that the list above does not include the components needed for the external crystal clock circuit and reset circuit. For details of such components, please consult Figure 2-2 in Chapter 2.

13.2.3 System Design

The next step involves making design choices to come up with a system that meets the requirements and specifications previously outlined. This includes deciding where to connect what to, what to use the 8051 I/O ports for; culminating in an overall schematic diagram showing the 8051 and how its pins (I/O, control, and clock) are connected to external memory and various 1/0 devices such as the sensors, alarm, and seven-segment display. Students might also want to use a 74LS47 BCD-to-seven-segment decoder to minimize the I/O port pins.

13.2.4 Software Design

Having decided on the hardware connections, the student can then proceed to designing and programming the software to control the 8051 and how it should interact with the external devices. This is often done with reference to the overall schematic diagram. Some questions that you should ask yourself, and hence the corresponding programming decisions that you should make, include:

- How should I sense the intrusion points? Via polling or interrupt?
- What priority should I assign to each intrusion point?
- When an intrusion occurs, what should I do?
- Should my system be programmed to detect more than one intrusion at a time?
- Should I include a capability to reset or deactivate the system?
- Should I allow the home owner to customize the system?
- Would the system have any security? For example, how should I ensure only the home owner can customize or deactivate the system?

Often, in coming up with such questions, you need to put yourself in the shoes of the actual user to be able to better understand the situation and the problem that you are trying to solve with the system.

Given below is a sample pseudo code for this home security system:

Pseudo Code:

```
WHILE [1] DO BEGIN
       WHILE [sensor == false] DO
             [wait]
       CASE [sensor] OF
                 `gate': [display = 0]
             frontdoor': (display = 1]
             `backdoor': [display = 2]
              `windows': [display = 3]
       END_CASE
       [alarm = on]
      IF [reset == TRUE && password == TRUE]
                THEN [alarm = off]
END
```

The pseudo code above shows that the program is in an infinite loop. As long as no sensor is triggered, the program waits and does nothing. When a sensor is triggered, it is checked to see which intrusion point it corresponds to, after which a number from 0 to 3 is displayed on the seven-segment display to indicate the point of intrusion. Next, the alarm is turned on. The program also allows for a reset mode where if the correct owner password is keyed in, the security system is reset and the alarm is turned off.

13.3 ELEVATOR SYSTEM

An elevator system is another interesting project that students could undertake. For simplicity, we will only consider a three-floor elevator system.

13.3.1 Project Description

The project description follows:

Design an 8051-based elevator system that supports three floors, namely ground, first floor, and second floor

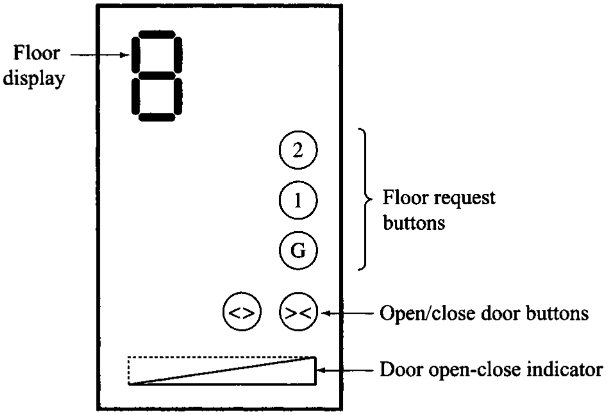


FIGURE 13-1
Inside the elevator

Your system should consist of two parts:

1. *Inside the Elevator*
Figure 13-1 shows the inside of the elevator. The floor request buttons, G, 1, and 2, are used by the elevator passenger to request the floor to which he wants to travel. The open (respectively close) door buttons are used by the passenger to open (or close) the elevator door. Meanwhile, there is also a corresponding door open-close indicator, which is basically a series of eight LEDs arranged from left to right, flashing one at a time. When the flashing LED is leftmost, this indicates that the door is closed. When the flashing LED is rightmost, the door is fully open.

2. *Outside the Elevator*
Figure 13-2 shows the outside of the elevator. The floor indicators are available at each floor to indicate whether the elevator is currently at that floor. The summon elevator buttons are used by the waiting passenger outside the elevator to summon the elevator to his floors, and to indicate which direction the passenger wants to travel, up or down.

13.3.2 System Specifications

A thorough read through the project description reveals that we would require:

- 1 x 8051 microcontroller
- 9 x switch buttons (five for inside, four for outside)

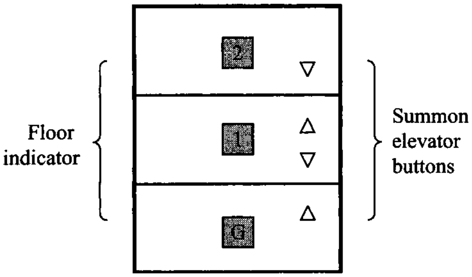


FIGURE 13-2
Outside the elevator

- 11 x LEDs (eight for door open-close indicator, three for floor indicators)
- 1 x 7-segment display

It is clear that this project merely involves the interaction with switches and LEDs, plus a seven-segment display. Therefore, it is not much different from the home security system in the previous section. The only thing here is that we are considering a totally different scenario, and the sequence of happenings (opening door, elevator moving, etc.) is more complex than the home security system.

13.3.3 System Design

The system design stage in this case would involve deciding which port pins to connect to the switch buttons, the LEDs, and the seven-segment display. Also, since the elevator system does not require the use of external memory, all four ports of the 8051 are available for I/O purposes so we have plenty to spare.

13.3.4 Software Design

The resultant program for this elevator system would be more complex because various scenarios have to be considered, including:

- Is the elevator currently on the same floor as that on which the up/down button request is made?
- Is the elevator going in the same direction as the request?
- If not, but the requesting floor is nearer, would it stop and service this first or ignore it and finish servicing an existing request?
- Would the elevator be servicing the nearest requesting floors first or would it be constantly moving in one direction from the lowest requesting floor to the highest, before changing direction and repeating the process?
- If there is no request, what should the elevator be doing?

There are various alternatives to handling the problem and previous students have come up with all sorts of ways to tackle this. A possible program could follow along the lines of the pseudo code below:
Pseudo Code:

```
[start at ground floor]
WHILE [1] DO BEGIN
      WHILE [elevator summon == FALSE] DO
             [wait]
      IF [summoning floor == current floor]
           THEN BEGIN
                 [door = open]
                 WHILE [open door button == TRUE] DO
                       [wait]
                 [door = close]
           END
      [direction = up]
```

```
BEGIN
      WHILE [ [summoning floor != current floor] &&
              [requested floor != current floor] ] DO BEGIN
                    IF [current floor == TOP_FLOOR]
                          THEN [direction = down]
                    IF [current floor == TOP_FLOOR]
                          THEN [direction = up]
                    IF [direction == up]
                             THEN [current floor = current floor + 1]
                             ELSE [current floor = current floor - 1]
                    [floor indicator = current floor]
                    [floor display = current floor]
      END
      BEGIN
              [door = open]
              WHILE [open door button == TRUE] DO
                    [wait]
              [door = close]
              END
      END
END
```

Here, the current floor is the floor on which the elevator is currently at; the summoning floor is the floor on which a waiting passenger outside the elevator has pressed the summon elevator button; while the requested floor refers to the floor requested by the passenger from within the elevator by pressing the floor request button.

The program is in an infinite loop and the elevator is initially on the ground floor, waiting to be summoned by people who are in need of the elevator on their floor. When the elevator is summoned, the program checks if the elevator is already on that floor, and if so would open the elevator door, wait for a while, and only close the door when the open door button is no longer pressed. The elevator then moves up.

As the elevator is going up from one floor to the next, the program checks if there are any elevator summons on the current floor, or if any passenger within the elevator had requested to go to that floor. If not, the elevator keeps going to the next floor. If the current floor is already the top or bottom floor, the direction of travel is reversed. At the same time, the floor indicators outside the elevator as well as the floor displays within the elevator would be updated to correspond to the current floor.

If the current floor is either the summoning floor or the requested floor, then the elevator opens its door, and waits for the passenger to release the open door button before it closes the elevator doors.

13.4 TIC-TAC-TOE

A more challenging project for students would be the development of games, and the corresponding artificial intelligence (AI) that would allow the program to compete with humans and with each other. One such project is the developing the Tic-Tac-Toe game based on the 8051.

TABLE 13-1
Playing modes

MODE	P1.3 (M1)	P1.4 (M1)
0 - Human vs Human	0	0
1 - Human vs AI	0	1
2 - AI vs Human	1	0

13.4.1 Project Description

First, the project description and rules of the game have to be outlined: *Design an 8051-based Tic-Tac-Toe game that consists of two parts:*

1. Tic-Tac-Toe Game Environment
At the start of the game, the player(s) selects one of three playing modes: Human vs Human, Human vs AI, and AI vs Human. The playing mode is input via two port pins, P1.3 and P1.4, and is shown in more detail in Table 13-1.

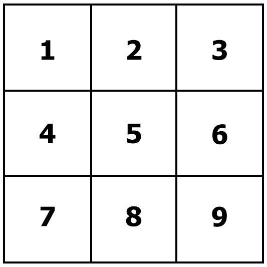
There should be a 3 x 3 Tic-Tac-Toe panel (see Figure 13-3) to indicate which box has already been selected by a player. A selected box would either light up as yellow (Player 1) or red (Player 2). When a vertical, horizontal, or diagonal row of three boxes of the same color has been selected, the corresponding player wins the game. Otherwise, if all boxes have been selected but no such row occurs, the game ends in a draw Each of the nine boxes basically consists of a yellow and red LED, and all these LEDs are connected to the 8051 through Ports 0, 1, and 2 as indicated in Figure 13-4.

Each player takes turns to interact with a 3 x 3 matrix keypad to select a certain box during his turn. The connections of the 8051 to the keypad are via Port 3. Player 1 indicates the end of his turn by setting P0.4 (Elf), which lights up an end-of-turn LED. Similarly, Player 2 ends his turn by setting his end-of-turn LED connected to P0.3 (E2).

At the end of the game, the results will be indicated on LEDs connected to P2.3 and P2.4, whose details are in Table 13-2.

2. Tic-Tac-Toe Artificial Intelligence (AI) Component
The AI component should be able to play as either player 1 or player 2 and should attempt as best as it could to win over the human player.

FIGURE 13-3
Tic-Tac-Toe



1	2	3
4	5	6
7	8	9

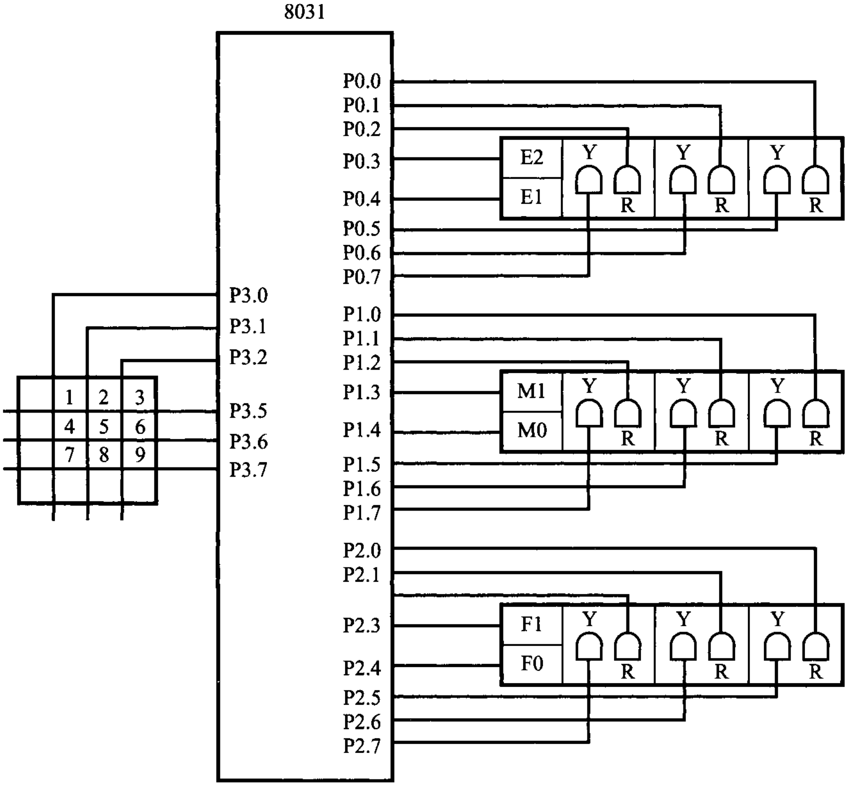


FIGURE 13-4
Connecting the 8051 to the Tic-Tac-Toe pane

13.4.2 System Specifications

The Tic-Tac-Toe system consists of mostly LEDs and a 3 X 3 keypad for interacting with the human player. A detailed list of components needed is as follows:

- 1 x 8051 microcontroller
- 1 x keypad (3 x 3)
- 11 x yellow LEDs
- 11 x red LEDs

TABLE 13-2
Final results indicated on LEDs connected to P1.3 and P1.4

MODE	P2.3 (F1)	P2.4 (F0)
Player 1 (Yellow) Wins	1	0
Player 2 (Red) Wins	0	1
Draw	1	1

So you see, there really is nothing much to the Tic-Tac-Toe hardware. What is more important is the logical and clear organization of the LEDs to avoid confusions on the part of the human player.

13.4.3 Software Design

In writing the game environment component, some questions come to mind:

- How do you determine the end of a player's turn?
- How should you check and hence run a playing mode?
- How do you determine if the game was won by a player or ended in a draw?

Meanwhile, for the AI part of the project:

- How do you determine the best box position?
- How do you block an opponent's bid for a three-in-one row?
- Are there any more advanced and subtle techniques to fool the opponent into selecting a box to your advantage?
- Should the AI strategy be fixed or should it adaptively change based on the current situation?

The pseudo code for a possible Tic-Tac-Toe game environment is given below:
Pseudo Code:

```
WHILE [input == FALSE]
      [wait]
[current player = yellow]
CASE [mode] OF
      `0': BEGIN
            WHILE [1] DO BEGIN
            [check current player color]
              WHILE DO [player input == FALSE]
                  [wait]
            [selected box == player color]
            IF [3-in-a-row == TRUE]
                  THEN BEGIN
                      [display winner]
                         [end game]
                  END
               IF [all boxes selected == TRUE]
                  THEN BEGIN
                       [display draw]
                         [end game]
                  END
            END
      END

      `1': BEGIN
            [current player = human]
            WHILE [1] DO BEGIN
                  [check current player color]
```

```
             IF [current player == AI]
                   THEN [call AI]
             WHILE DO [input == FALSE]
                   [wait]
             [selected box == player color]
             IF [3-in-a-row == TRUE]
                   THEN BEGIN
                        [display winner]
                        [end game]
                   END
             IF [all boxes selected == TRUE]
                    THEN BEGIN
                        [display draw]
                        [end game]
                   END
                   current player = other player]
             END
END
`2': BEGIN
      [current player = AI]
      WHILE [1] DO BEGIN
             [check current player color]
             IF [current player == AI]
                  THEN [call AI]
            WHILE DO [input == FALSE]
                   [wait]
             [selected box == player color]
              IF [3-in-a-row == TRUE]
                    THEN BEGIN
                          [display winner]
                          [end game]
                   END
             IF [all boxes selected == TRUE]
                    THEN BEGIN
                           [display draw]
                           [end game]
                   END
             [current player = other player]
             END
      END
END_CASE
```

This program waits for the player to give an input to start playing Tic-Tac-Toe. Once an input is detected, it checks to see what mode of play has been selected. If it is mode 0, which is "Human vs Human," then the program goes into an infinite loop and checks to see who is the current player and waits for the current player to make his turn. When an input has been made, meaning a certain box has been selected, the program displays changes the color of that box to the corresponding color of the current player. After the current player has taken his or her turn, the program checks to see if there are any three boxes of the same color in a vertical, horizontal, or diagonal row. In such a situation, the winner is displayed and the game ends. Otherwise, the program checks if all boxes have been selected, which means that the game is a draw and this result is also displayed.

Otherwise, if mode 1, which is "Human vs AI," has been selected, the first player to start is the human player. Going into an infinite loop, the program then checks the current player. If the current player is the AI, the AI function would be called to calculate the best box to choose. Whether it is the AI or a human player, when a box has been selected by a player, it is displayed with the player's color. The program proceeds to check for three equally-colored boxes in the same row, upon which it has detected a winner, displays the result, and ends the game.

Mode 2 with "AI vs Human" is also similar to mode 1, with the difference that the first player to start is the AI.

Meanwhile, the pseudo code for a possible Tic-Tac-Toe's AI is given below:

Pseudo Code:

```
IF [opponent's 2-in-one row == TRUE]
       THEN [block opponent's chance]
ELSE BEGIN
       [calculate best box position]
       WHILE [box == taken] DO
             [calculate next best box position]
       [select box]
       END
```

The AI looks simple enough. The main priority is to first block an opponent's attempt at getting three boxes in a row so if two boxes in a row are detected, then the AI's choice would be to select the third box in order to block the attempt and prevent the opponent from winning. Next, if there are not two boxes in a row, that means the AI is still safe and has more flexibility to choose a box based on its calculations. If the best box has already been taken, it calculates the next best box and so on until it finds a free box upon which it immediately selects. Simple it may seem, but the real task of programming the AI lies in the calculation of the best box positions. Here is what distinguishes the best AI from among the rest.

The separation of the software into two parts helps ease the programmer's task. This also provides an interesting exercise on **modular programming,** where a program is divided into modules, each written by a separate programmer. Once the programming is done, the modules are integrated into one complete program. Modular programming allows for several parts of a program to be developed concurrently and is best suited for large and complex programming projects.

13.5 CALCULATOR

Calculators are so common these days that we sometimes tend to take them for granted. As a further exercise in 8051-based system design and interfacing, an 8051-based would be a contenting project. Of course, to keep it simple, only the basic operations would be considered.

13.5.1 Project Description

The project description proceeds as follows:

Design an 8-bit 8051-based calculator to perform simple arithmetic operations such as:

- *Addition*
- *Subtraction*
- *Multiplication*
- *Division*
- *Logical AND*
- *Logical OR*
- *Logical NOT*
- *Logical XOR*
- *Square*
- *Square root*
- *Cube*
- *Cubic root*
- *Inversion*
- *Exponentiation*
- *Factorial*
- *Modulo operation*
- *Percentage*
- *Pi*
- *Basic memory operations*

Your calculator should be able to support 16-bit answers, which should be displayed on either LCD or 7-segment displays. User input should be via a matrix keypad whose layout should be designed.

13.5.2 System Specifications

The calculator requires a matrix keypad for input, and either an LCD or seven-segment display for output. Besides these, there should at least be one push button switch for switching on the calculator, and a corresponding LED to indicate that the calculator is on. Therefore, the components needed for this system would be:

- 1 x 8051 microcontroller
- 1 x matrix keypad
- 1 x LCD or 5 x 7-segment displays
- 1 x LED
- 1 x push button switch

13.5.3 Software Design

The software to control this system would be similar to that of operating systems such as Windows or MS-DOS, in which all arithmetic and logical operations as well as interactions with input and output devices are handled by the program. In writing this program, the following should be considered:

- How do you determine if the key pressed by the user is a number or an operation?
- How should you handle operator precedence?
- Should you allow for accumulated operations?
- Should the calculator operate with unsigned or signed numbers?
- Should the calculator support decimal points?
- How would you implement each type of arithmetic or logical operation?
- Would you use lookup tables?

A possible 8051 calculator could operate based on the following pseudo code:
Pseudo Code:

```
WHILE (1) DO BEGIN
       WHILE [keypress == FALSE]
             [wait]
       CASE [keypress] OF
             `numeric': BEGIN
                   [save numeric value]
                   [format for 7-segment display]
                   [send to 7-segment display]
                   END
             `operation': BEGIN
                   [determine operation]
                   [save operation]
                   END
             `memory operation': BEGIN
                   [determine operation]
                   [perform operation]
                   END
             `equal': BEGIN
                   [recall saved numeric values and operations]
                   [perform operations]
                   [format result for 7-segment display]
                   [send to 7-segment display]
                   END
       END_CASE
END
```

The program waits indefinitely for a key press. When a key press is detected, it is checked to see if it corresponds to a numeric value, arithmetic or logical operation, a memory operation, or the equal key. When the former two are detected, the program checks specifically which value or operation they are and saves them. A numeric value would also be formatted and sent to the seven-segment display. If a memory operation is requested, the program immediately determines what memory operation is requested and services it. Finally, if the `equal` key is pressed, it means the user is ready to see the results, so all previously saved numeric values and operations are recalled and performed according to precedence. The result is then formatted and sent to the 7-segment display.

13.6 MICROMOUSE

When you connect the 8051 (the brain), some sensors (the eyes, ears, etc.), and some motors (the hands and feet), you get a basic robot. One such robot is the **micromouse,** which is the name for a robot that imitates a mouse and which tries to solve a maze. Solving a maze requires that the micromouse start from one end and slowly learn, mostly by trial and error as well as memorizing, the correct path to the other end of the maze. Once it has found the correct path and reached the other end of the maze, it should be able to follow this path the next time it is put in the maze. Micromouse contests are popular and held in many areas around the world. In this section, we will consider a micromouse project.

13.6.1 Project Description

As usual, we describe details of the project:

Design an 8051 microcontroller-based micromouse that should have the ability to move forward, turn left and right as well as make U-turns. It should be able to solve a maze as fast as possible and once solved, should store the correct path in its memory.

13.6.2 System Specifications

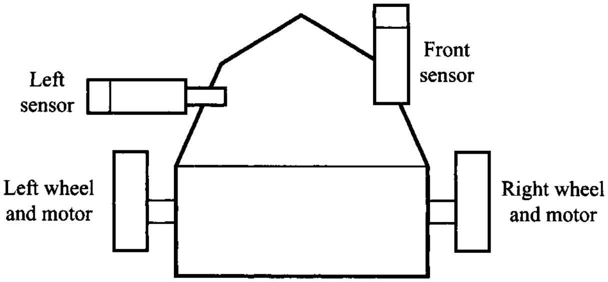
In order to move forward and make turns, the micromouse should have two motors attached, respectively, to its left and right wheels. It should also be able to know if it has reached a dead end, or if there is a wall on its left or right if it wanted to turn in that direction. For that, it would need some sensors. Two infrared sensors would suffice. Probably we could also include some LEDs to indicate what the micromouse is currently supposed to do, that is whether it is going forward, turning left, right, or making a U-turn. The micromouse would also need some large external memory for memorizing paths previously traversed. Therefore, the components for the micromouse are:

- 1 x 8051 microcontroller
- 2 x motors
- 2 x infrared sensors
- 1 x 64Kbyte RAM
- 2 x LEDs

13.6.3 System Design

The main part of the system design stage is to decide where to put the infrared sensors. The infrared sensor is typically made of two parts arranged side by side: an infrared LED to emit infrared light and an infrared detector (phototransistor) to detect the presence of any infrared light. If a wall is in the way of the sensor, the infrared light emitted by the infrared LED would be reflected and subsequently detected by the infrared detector. Otherwise, if no wall is present, there would be no reflection and hence no infrared light would be detected. This is how the micromouse would know if there are any walls in a certain direction.

FIGURE 13-5
Top view of a micromouse



One of the sensors should be put on the front of the mouse to detect if it has any walls blocking its forward movement. The other sensor should be put on either the left or right side of the mouse. Suppose you decide to put it on the left side, as shown in Figure 13-5.

The next question asked is: If you only have a sensor on one side, how would you be able to sense for the presence of walls on the other side? The answer is: you don't have to do that directly. Let's consider this with some example situations, as given in Examples 13.1 to 13.3.

EXAMPLE 13.1 The Micromouse Reaches a Left Corner

How would the micromouse know that it has reached a left corner?

Solution

Front sensor senses wall. Left sensor senses no wall.

Discussion

In this case, the front sensor of the micromouse detects a wall blocking its forward movement. Meanwhile, the left sensor senses no wall. Hence it will know that it has reached a corner where it cannot move forward but can turn left.

Of course, it could be possible that the micromouse has reached a T-junction, in which case it could have turned left or right. Nevertheless, since left is free to turn to, the micromouse should try the left path first and come back later to try the right one.

EXAMPLE 13.2 The Micromouse Reaches a Right Corner

How would the micromouse know that it has reached a right corner?

Solution

Both front and left sensors sense walls.

Discussion

Since both sensors sense the presence of walls, the micromouse can neither move forward nor turn left. Hence, it guesses that there should be no wall on its right and so makes a turn to the right.

However, it could be possible that the micromouse has reached a dead end, in which case even its right side would have a wall. We will consider this in the next example.

EXAMPLE 13.3 The Micromouse Reaches a Dead End

How would the micromouse know that it has reached a dead end?

Solution

Previously, both front and left sensors sensed walls. After it makes a right turn, the front sensor again senses a wall.

Discussion

Previously, both the front and left sensors sensed walls, so the micromouse turned right. However, after making the turn, its front sensor still senses a wall. This could only happen if it were at a dead end. In this case, it should turn right a second time, so that it has effectively made two right turns. This is essentially a U-turn. It can now proceed forward to trace back its steps and leave the dead end.

13.6.4 Software Design

The software for the micromouse is somewhat complicated and requires some deep thinking. Several issues include:

- How would the micromouse detect left corners, right corners, and dead ends?
- How do you make the micromouse turn left or right?
- In what order should the micromouse test out all the possible paths? Should the nearest or furthest branches be fully tested first?
- How should the micromouse remember previously traversed paths?

The pseudo code for the micromouse program is as follows:
Pseudo Code:

```
WHILE [1] DO BEGIN
       [left motor = right motor = forward]
      IF [front sensor == wall]
           IF [left sensor == no wall]
                   THEN BEGIN
                        [left motor = backward]
                        [right motor = forward]
                   END
                   ELSE BEGIN
                        [left motor = forward]
                        [right motor = backward]
                        IF [front sensor == wall]
                              THEN BEGIN
                                     [left motor = forward]
                                       [right motor = backward]
                              END
                   END
       [memorize path]
END
```

The issue of detecting corners and dead ends has just been discussed in Examples 13.1 to 13.3. What about turning left and right? This can easily be solved. If the micromouse is to go forward, both motors should turn forward. If it wants to go left, the left motor should go backwards while the right motor should go forward. Similarly, to turn right, the left motor should go forward while the right motor should go backwards.

The pseudo code above shows that the program goes into an infinite loop, and initially goes forward. If it senses a wall in front, it proceeds to sense if there is a wall on its left. If not, it turns left. Otherwise, it turns right. After turning right, it senses if there is a wall in front. If there still is then it is at a dead end, so it turns right again. Now that it has done the required turns, it can proceed to go forward. During all this while, the micromouse is also memorizing the paths that it has previously traversed.

The main challenge in programming the micromouse lies in the one line: [memorize path]. How should the micromouse memorize the paths to ensure that the next time it comes to the same T-junction or crossroad, it would not choose paths with dead ends? This will be the subject of immersing thought and vigorous programming trials.

13.7 A SOCCER-PLAYING ROBOT

The previous section discussed how an 8051 could be used as the brain for a mouse-like robot, enabling it to find its way through a maze and hence solve it. In fact, the 8051 can be used to control any robot, and is among the popular microcontrollers that are used by robot enthusiasts as robot brains. Robot contests are held around the world, and are very interesting sights indeed. Participating robots would be required to do all sorts of tasks ranging from playing simple sports to more complicated tasks such as climbing stairs. In this section, we will discuss how an 8051 is used to control a soccer-playing robot.

13.7.1 Project Description

Design an 8051 microcontroller-based robot to take part in a robo-soccer contest, lasting 10 minutes. During each match, two robots playing as blue and red are put in a grey-carpeted arena with randomly placed blue and red balls of 15mm in diameter. A common goal is located at one end of the arena and is illuminated by bright lights. Each robot should detect the presence of balls, sense the ball colors, and collect balls of its own color. However, it should only store one ball at a time. Once it has collected a ball, it should search for the common goal and kick the ball into the goal.

13.7.2 System Specifications

Similar to the micromouse, the robot should have wheels attached to motors. It would also need some light sensors (LED and phototransistor pairs) to detect the goal, and ball colors, plus some contact sensors such as switches to detect walls or the opponent robot. Since the robot is to stand alone, it should have nonvolatile memory. In this case, an EPROM would be a good choice since this would allow the program to be overwritten at will, which is often desirable. The required components are then:

- 1 x 8051 microcontroller
- 2 x motors
- 4 x wheels

- 2 x light sensors (LED and phototransistor pair)
- 4 x switches
- 1 x EPROM

13.7.3 System Design

In designing this robot system, where to put the light and contact sensors is important. For example, the light sensors to sense the ball colors should be well shaded from their external surroundings, otherwise it would affect the color detection. This is especially so when the robot is very near the goal. The brightness of the goal lights would interfere with the light from the sensor's LED and cause confusion.

Another design issue is the kicking mechanism. Various methods exist to kick the ball, and watching different students think up different ideas would be part of the fun. Some would use motors to swing a shaft to make contact with the ball, similar to how baseball players hit with their bats. However, the limitation of this technique is that the kick would be less powerful than if an elastic energy were used. Hence, some would prefer to attach the shaft to a piece of rubber band, and use a motor to stretch it. Then when the robot is ready to kick, the motor moves into a position such that the rubber band is released, forcing the shaft to swing towards the ball and kick it out of the robot.

13.7.4 Software Design

Some issues related to the robot software are:

- How would the robot detect walls and opponent robots?
- How should the robot turn?
- When the robot reaches a corner, should it turn right or left? Should it always turn to one direction or should it alternate, or turn randomly?
- How should randomness be implemented?
- How would the robot detect ball colors?
- How would the robot detect the direction of the goal?
- How should the robot kick?
- What strategy should the robot use against its opponent? Should it be more offensive, defensive, or switch strategies halfway through the match?
- How should the robot move around the arena? Should it move randomly, or should it move in a certain pattern? If so, what would be the best pattern to ensure that it covers as much ground as possible?

Given below is a sample pseudo code for the robot playing blue:
Pseudo Code:

```
WHILE [1] DO BEGIN
      [walk in spiral pattern]
      IF [contact sensor == TRUE]
            THEN BEGIN
                    [reverse]
                   [turn left or right]
            END
      IF [light sensor == BLUE]
           IF [balls collected < 1]
                 THEN BEGIN
                      [collect ball]
                      [sense goal lights]
                IF [goal detected == TRUE]
                      THEN BEGIN
                           [stop]
                           [kick]
                    END
                END
      IF [light sensor == RED]
          IF [time left == 5 minutes]
                IF [balls collected < 1]
                      THEN BEGIN
                           [collect ball]
                           [go straight forward]
                           IF [wall detected]
                                [release ball]
                    END
END
```

The program directs the robot to walk in a spiral pattern, believed by some to be a good pattern that allows the robot to cover much ground. When a contact sensor returns TRUE, this indicates that it has either moved against a wall or an opponent, upon which the robot reverses and either turns right or left, away from the obstacle, before proceeding with its forward movement. If the light sensor detects a BLUE ball, which is its own color, it checks to see if it already has a ball within its storage. Otherwise, it collects the ball and immediately goes in search of the goal. Once the goal is detected, the robot stops and kicks the ball in. If the light sensor senses a RED ball, the robot checks if there are only 5 minutes left in the match. If so, it will switch to a more offensive strategy where it collects an opponent ball, goes straight forward, expecting to bump into a wall sooner or later. When it does, it releases the ball. Balls right next to the wall are usually much harder to collect. This would be the BLUE robot's strategy against the opponent.

13.8 A SMART CARD APPLICATION

Microcontrollers are finding increasing usage in **smart cards,** which are cards that integrate both memory and a microcontroller within its physical limits. Imagine having a computer so small that we can wrap it up with a piece of plastic that fits into our wallet: That's a smart card. Smart cards are being used as national identity cards, driving licenses, passport information, and electronic cash (e-cash). Since these smart cards contain personal as well as sensitive financial information, and would be issued to the public, it is vital that the information contained within it be protected from being accessed or tampered with by unauthorized parties. The following subsection briefly describes basic security concepts for a better understanding of how the information within smart cards can be protected.

13.8.1 Basic Security Concepts

Information security typically requires that **confidentiality, authentication,** and **integrity** to be guaranteed. Confidentiality ensures that only authorized parties can access and view the information. Authentication proves the identity of a party while integrity allows one to determine whether information has been modified without permission.

All these are possible by using cryptographic techniques such as **encryption, digital signatures,** and **message authentication.** Encryption is the process of transforming some confidential information into unintelligible form in order to protect it from being accessed. Standard encryption methods are the Data Encryption Standard (DES) and its variant, the triple-DES. Incidentally, the latter is also used in many current Automatic Teller Machine (ATM) cards to protect the user Personal Identification Numbers (PINs).

Meanwhile, a digital signature is similar in concept to its physical handwritten counterpart. However, instead of using a handwritten signature or a thumbprint, a digital signature uses some bits of secret data to prove the identity of a party. Message authentication is done by using some secret bits of data to perform an operation on the message to obtain what is called **a message authentication code (MAC).** This is similar to a checksum and is used to check if the message has been modified. Since the MAC can only be obtained when you know the secret data, this ensures that unauthorized parties cannot modify the message without being detected because the MAC would be different.

13.8.2 Project Description

Recall that in the problems section of the previous chapter, we discussed a very simple encryption method called the Caesar cipher. We will now consider a slightly more complicated and generalized version, the **polyalphabetic cipher.** Even so, this encryption method is also not in use today to protect information because it is very easy to crack by using computer programs. However, for ease of illustration, we will consider it in this section, and only touch on currently secure methods in the problems section at the end of this chapter.

The ***polyalphabetic cipher*** *is similar to the Caesar cipher except that instead of shifting the original plaintext set of alphabets a fixed three positions to the left, it is shifted left by a variable number of positions. For example, shifting by 10 positions to the left, the plaintext alphabet set would be replaced by a ciphertext alphabet set as follows:*

Plaintext alphabets	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Ciphertext alphabets	K L M N O P Q R S T U V W X Y Z A B C D E F G H I J

so the message THIS IS SECRET is encrypted into DRSC SC ***COMBOD.***

Design an 8051 microcontroller-based smart card that encrypts by using the polyalphabetic cipher, null-terminated messages stored in its ***EPROM*** *starting at location 1234H.*

13.8.3 System Specifications

The smart card system is really quite simple, consisting of just the 8051 and an EPROM, whether external or built into the microcontroller, in which case we would be using an 8071 instead of an 8051. We might choose to add in some extra features, for example an LCD to display both plaintext and ciphertext messages. The components are:

- 1 x 8051 microcontroller
- 1 x EPROM
- 1 x LCD

13.8.4 Software Design

When writing the encryption software, several issues might be:

- How would the number of shifts be determined? Would it be randomly chosen, or would the user be allowed to specify?
- Are there any number of shifts that should not be permitted? For example, shifting by 0, 26, 52, or any multiple of 26 positions would cause the ciphertext to be exactly the same as the ciphertext, resulting in no encryption at all.

Pseudo Code:

```
BEGIN
      [determine number of shifts]
      [location = 1234H]
      REPEAT
            BEGIN
                  [get character from location]
                    [encrypt character]
                  [store back in location]
            END
      UNTIL [character == NULL]
END
```

The software is straightforward enough. First, the number of shifts is determined, and the first location is set to 1234H. The program then gets a character from the current location, encrypts it with the polyalphabetic cipher and stores the encrypted character back into the current location, hence overwriting the previously stored character. This is repeated until a NULL character is detected, indicating the end of the message.

SUMMARY

We have described some advanced 8051-based projects that we feel would be challenging as well as interesting to students. Possible solutions are discussed in an orderly fashion starting from a careful study of the project description to a specification of the system requirements and components. This is followed by a subsection on software design which considers some programming decisions and the corresponding pseudo code description of the software.

PROBLEMS

13.1 Draw the schematic diagram for the home security system project in Section 13.2 showing all the connections between the 8051 and the alarm, sensors, seven-segment display, and external memory chips. Hence, write the software (in assembly or C) to control this system.

13.2 Draw the schematic diagram for the elevator system project in Section 13.3 showing all the connections between the 8051 and the LEDs, switches, and seven-segment display.

13.3 Observe the behavior of several elevator systems in your university or any building near you. In what order do these systems service the elevator requests? Why do you think such an order is used?

13.4 Draw the schematic diagram for the Tic-Tac-Toe project in Section 13.4 showing all the connections between the 8051, the keypad, the LEDs, and switches. Hence, write the software (in assembly or C) for this system.

13.5 Draw the schematic diagram for the calculator project in Section 13.5 showing all the connections between the 8051 and the keypad, display, LED, and switch.

13.6 Write the assembly language subroutine or C function to perform any two of the arithmetic operations listed in the project description of the 8051 calculator.

13.7 Draw the schematic diagram for the micromouse project in Section 13.6 showing all the connections between the 8051 and the motors, sensors, LEDs, and external memory.

13.8 Give some thought to the problem of programming the micromouse:

a. Testing out all possible paths: In which order do you think they should be tested out? Nearest branches first, or the furthest? Why?

b. Memorizing previously traversed paths: What do you think is the best way for the micromouse to achieve this? Why?

13.9 When using smart cards for personal identification, it is often desired that the photo of the individual also be embedded within the card. However, one limitation of the smart card is its small memory size. Research currently used techniques of embedding photos into smart cards.

13.10 One day in class, you chance upon a folded piece of paper, written by a certain classmate to another classmate of the opposite sex. "Interesting," you think to yourself as the smile on your face widens. But the smile turns to disbelief when you see that the rest of the message reads as follows:

atih spit idcxywi

You have just learnt about the Caesar cipher in class the other day, so you guess that the classmate must have used that. But after sitting down and trying to decipher the message, you find that it was not the Caesar cipher. Nevertheless, you are sure it must be something similar, just that unlike the Caesar cipher where you shift by three positions to the right, you do not know how many shifts your classmate used. A polyalphabetic cipher was used!

Suddenly it dawns on you: use the 8051 to decipher it! Write an 8051 program to decipher the message.

Hint: You would need to perform what we call a **dictionary attack:** Try all possible shifts and compare each deciphered word with words in a dictionary to see if you get any matches. Once you get matches for all three words, you have most probably deciphered the message successfully. Assume that you have a dictionary with 1000 entries including the words of the original message, and which is represented by the following array of characters:

```
char * dictionary = { "a" , "able", "about", ...} ;
```

where dictionary [0] gives you "a" , dictionary [1] gives you "able", etc.
Assume all characters are in lowercase, with the ASCII codes given in the table below:

TABLE 13-3 Partial ASCII table

	ASCII Code		
Character	Decimal	Hex	Binary
a	97	61H	01100001
b	98	62H	01100010
c	99	63H	01100011
d	100	64H	01100100
e	101	65H	01100101
f	102	66H	01100110
g	103	67H	01100111
. . .			

Write the pseudo code for such a program.

13.11 Write the corresponding 8051 C solution to Problem 13.10. Hence, use it to decode the message.

13.12 The Advanced Encryption Standard (AES) is a recent encryption standard chosen to replace the DES and triple-DES in future security applications. Research on how the AES works and write an 8051 program (whether in assembly or C) to implement the AES. A brief description of the AES is also given in Appendix J.

CHAPTER 14

8051 Derivatives

14.1 INTRODUCTION

Since the advent of the MCS-51^{TM} family of microcontroller ICs, newer and more advanced versions have sprung up. These **8051 derivatives** have additional memory, built-in I/0 such as ADCs and DACs, and other extended peripherals. In this section, we will briefly review some of these derivatives.

14.2 MCS-151TM AND MCS-251TM

Intel has produced advanced versions of the MCS-51^{TM} family. These are the MCS-151TM, with five times increased performance, followed by the MCS-251^{TM} family with 15 times performance increase. The MCS-251 ^{TM} family has an advanced architecture that increases the efficiency for 8051 C language programming, an extended instruction set that includes 16- and 32-bit arithmetic and logic instructions, plus increased memory size options (see Table 14-1). Here, **One-Time Programmable (OTP) ROM** refers to EPROM-like code memory but without the quartz glass window for erasing the contents. This reduces the packaging cost but prevents the OTP ROM from being erased with ultraviolet (UV) light; hence, it can only be programmed once. ROMless versions are also available.

14.3 MICROCONTROLLERS WITH FLASH MEMORY AND NVRAM

In Chapter 2 we were introduced to the MCS-51^{TM} with different types of on-chip code memory ranging from the ROMless 8031, through to the 8051 with on-chip ROM, and the 8751 with on-chip EPROM.

TABLE 14-1
Comparison of MCS-251^{TM} ICs

Part Number	On-Chip Code Memory	On-Chip Data Memory
80251SA	8K ROM/OTPROM	1K
80251SB	16K ROM/OTPROM	1K
80251SP	8K ROM/OTPROM	512 bytes
80251SQ	16K ROM/OTPROM	512 bytes
80251TA	8K ROM	1K
80251TB	16K ROM	1K
80251TP	8K ROM	512 bytes
80251TQ	16K ROM	512 bytes

Even with an 8751 which allows the programmer to reprogram it with a PROM programmer, you would have to first erase it with an ultra-violet (UV) EPROM eraser before you could program it again. In contrast, the 8951 is an 8051 derivative with on-chip **flash memory** that basically works like an EPROM but its contents can be electrically erased by the PROM programmer itself; hence, it does not require a separate UV-EPROM eraser. One such manufacturer of the 8951 is Atmel Corporation.[1]

Besides derivatives that make use of flash memory, others such as Maxim Integrated Product's[2] DS5000 have an NVRAM for code memory. The advantage of NVRAM over flash memory is that you can change your program in code memory one byte at a time, instead of having to erase it entirely before reprogramming. The DS5000's NVRAM also allows programs to be reloaded into code memory even via the PC's serial port, eliminating the need for a separate PROM programmer.

14.4 MICROCONTROLLERS WITH ADCS AND DACS

One of the most common features of 8051 derivatives are the built-in analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) that reduce the need for connecting to external ADCs and DACs when interfacing the 8051 to analog I/O devices. An example of such an 8051 derivative is the SAB80C515A microcontroller manufactured by Siemens, which has a built-in 10-bit ADC. More advanced variants are the collection of 8051 derivatives from Atmel that also function as MP3 players, including the AT89C51SND1C and AT89C51SND2C that even come with the recently popular USB 1.1 interface to PCs.

14.5 HIGH-SPEED MICROCONTROLLERS

The 8051 runs at 12 clocks per machine cycle. Certain high-speed derivatives such as the MCS-151^{TM} and MCS-251^{TM} run at only two clocks per machine cycle, and hence are capable of executing more instructions within a given amount of time.

[1] Atmel Corporation, 2125 O'Nel Drive, San Jose, CA 95131

[2] Maxim Integrated Products, Inc. includes the Dallas Semiconductors

Maxim also produces various types of 8051 derivatives, including high-speed microcontrollers, network microcontrollers, and secure microcontrollers. Its high-speed microcontroilers run at four clocks per machine cycle compared to the 8051's 12 clocks, while its ultra high-speed microcontroller, the DS89C420, runs at one clock per machine cycle. Other enhancements over the 8051 include more interrupt sources and increased memory sizes.

14.6 NETWORK MICROCONTROLLERS

The network microcontrollers manufactured by Maxim support various network protocols such as the Ethernet and Controller Area Network (CAN). Other network microcontrollers such as the 83751 by Philips[3] support the Inter-Integrated Circuit (I^2C) network interface while the COM20051 by Standard Microsystems[4] supports the ARCNET token ring network protocol. Such network protocols allow several microcontrollers and other processors to be connected as a network to share and exchange data. Atmel's ATWebSEG-32 is an 8051 derivative that supports internet connection (TCP/IP) and the Ethernet.

14.7 SECURE MICROCONTROLLERS

In the previous chapter, we had a look at how the 8051 can be used as the brain in smart cards. This included a discussion of how confidential information could be protected via software encryption. In fact, encryption can be done by dedicated security hardware, and several 8051 derivatives such as Maxim's secure 8051 microcontrollers have been produced to achieve this.

The security system in the smart card that allows encryption, digital signatures, and message authentication to be performed is commonly called a **public-key infrastructure (PKI).** This PKI is commonly embedded within secure microcontrollers and is supported by hardware peripherals built into these microcontrollers, for instance Maxim's DS5240, which has a Modulo Arithmetic Accelerator (MAA) to support modulo arithmetic operations, used extensively in PKIs. Other secure microcontrollers such as Maxim's DS5000 support hardware encryption of the programs loaded into its code memory. By encrypting these programs, even if the smart card is tampered with and its programs accessed without authorization, the attacker would not be able to understand the meaning of the programs.

SUMMARY

In this chapter, we have described the many derivatives of the 8051. These enhanced versions of the 8051 typically have better types of on-chip memory, larger memory size, higher speed, and more built-in peripherals to support interactions with analog I/O devices, network, and security applications.

[3] Philips Semiconductors, 811 E. Argues Avenue, Box 3409, Sunnyvale, CA 94088

[4] Standard Microsystems Corporation, 80 Arkay Drive, Hauppauge, NY 11788

That being said, the final decision of selecting the most suitable 8051 *or* derivative for your project depends on a set of criteria such as the amount and type of on-chip ROM and RAM, speed, as well as advanced I/O, network, and security requirements.

PROBLEMS

14.1 Conduct research on the various 8051 derivative manufacturers and list out other derivatives with built-in ADCs or DACs.

14.2 Conduct research on the 8051 manufacturers and list out other secure microcontrollers. Include their security features.

14.3 Are there other enhancements to the 8051 that are not discussed in this chapter? List out the enhancements if any, and examples of such derivatives.

A

Quick Reference Chart

MNEMONIC		DESCRIPTION
Arithmetic Operations		
ADD	A,source	add source to A
ADD	A,#data	
ADDC	A,source	add with carry
ADDC	A,#data	
SUBB	A,source	subtract from A
SUBB	data	with borrow
INC	A	increment
INC	source	
DEC	A	decrement
DEC	source	
INC	DPTR	increment DPTR
MUL	AB	multiply A & B
DIV	AB	divide A by B
DA	A	decimal adjust A
Logical Operations		
ANL	A,source	logical AND
ANL	A,#data	
ANL	dlrect,A	
ANL	direct,#data	
ORL	A,source	logical OR
ORL	A,#data	
ORL	direct,A	
ORL	direct,#data	
XRL	A,source	logical XOR

MNEMONIC		DESCRIPTION
XRL	A,#data	
XRL	direct,A	
XRL	direct,#data	
CLR	A	clear A
CPL	A	complement A
RL	A	rotate A left
RLC	A	(through C)
RR	A	rotate A right
RRC	A	(through C)
SWAP	A	swap nibbles

LEGEND

Rn	register addressing using R0-R7
direct	8-bit internal address (00H-0FFH)
@Ri	indirect addressing using R0 or R1
source	any of [Rn,direct,@Ri]
dest	any of [Rn,direct,@Ri]
#data	8-bit constant included in inst.
#data 16	16-bit constant
bit	8-bit direct address of bit
rel	signed 8-bit offset
addr11	11-bit address in current 2k page
addr16	16-bit address

FIGURE A-1
Quick reference chart

B

Opcode Map

Instruction Code Summary

H L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	JBC bit, rel	JB bit, rel	JNB bit, rel	JC rel	JNC rel	JZ rel	JNZ rel	SJMP rel	MOV DPTR, # data 16	ORL C, /bit	ANL C, /bit	PUSH dir	POP dir	MOVX A, @DPTR	MOVX @DPTR, A
1	AJMP (P0)	ACALL (P0)	AJMP (P1)	ACALL (P1)	AJMP (P2)	ACALL (P2)	AJMP (P3)	ACALL (P3)	AJMP (P4)	ACALL (P4)	AJMP (P5)	ACALL (P5)	AJMP (P6)	ACALL (P6)	AJMP (P7)	ACALL (P7)
2	LJMP addr16	LCALL addr16	RET	RETI	ORL dir, A	ANL dir, A	XRL dir, A	ORL C, bit	ANL C, bit	MOV bit, C	MOV C, bit	CPL bit	CLR bit	SETB bit	MOVX A, @R0	MOVX @R0, A
3	RR A	RRC A	RL A	RLC A	ORL dir, # data	ANL dir, # data	XRL dir, # data	JMP @A+DPTR	MOVC A, @A+PC	MOVC A, @A+DPTR	INC DPTR	CPL C	CLR C	SETB C	MOVX A, @R1	MOVX @R1, A
4	INC A	DEC A	ADD A, # data	ADDC A, # data	ORL A, # data	ANL A, # data	XRL A, # data	MOV A, # data	DIV AB	SUBB A, # data	MUL AB	CJNE A, # data, rel	SWAP A	DA A	CLR A	CPL A
5	INC dir	DEC dir	ADD A, dir	ADDC A, dir	ORL A, dir	ANL A, dir	XRL A, dir	MOV dir, # data	MOV dir, dir	SUBB A, dir		CJNE A, dir, rel	XCH A, dir	DJNZ dir, rel	MOV A, dir	MOV dir, A
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	MOV @R0, # data	MOV dir, @R0	SUBB A, @R0	MOV @R0, dir	CJNE @R0, # data, rel	XCH A, @R0	XCHD A, @R0	MOV A, @R0	MOV @ R0, A
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, @R1	MOV @R1, # data	MOV dir, @R1	SUBB A, @R1	MOV @R1, dir	CJNE @R1, # data, rel	XCH A, @R1	XCHD A, @R1	MOV A, @R1	MOV @ R1, A
8	INC R0	DEC R0	ADD A, R0	ADDC A, R0	ORL A, R0	ANL A, R0	XRL A, R0	MOV R0, # data	MOV dir, R0	SUBB A, R0	MOV R0, dir	CJNE R0, # data, rel	XCH A, R0	DJNZ R0, rel	MOV A, R0	MOV R0, A
9	INC R1	DEC R1	ADD A, R1	ADDC A, R1	ORL A, R1	ANL A, R1	XRL A, R1	MOV R1, # data	MOV dir, R1	SUBB A, R1	MOV R1, dir	CJNE R1, # data, rel	XCH A, R1	DJNZ R1, rel	MOV A, R1	MOV R1, A
A	INC R2	DEC R2	ADD A, R2	ADDC A, R2	ORL A, R2	ANL A, R2	XRL A, R2	MOV R2, # data	MOV dir, R2	SUBB A, R2	MOV R2, dir	CJNE R2, # data, rel	XCH A, R2	DJNZ R2, rel	MOV A, R2	MOV R2, A
B	INC R3	DEC R3	ADD A, R3	ADDC A, R3	ORL A, R3	ANL A, R3	XRL A, R3	MOV R3, # data	MOV dir, R3	SUBB A, R3	MOV R3, dir	CJNE R3, # data, rel	XCH A, R3	DJNZ R3, rel	MOV A, R3	MOV R3, A
C	INC R4	DEC R4	ADD A, R4	ADDC A, R4	ORL A, R4	ANL A, R4	XRL A, R4	MOV R4, # data	MOV dir, R4	SUBB A, R4	MOV R4, dir	CJNE R4, # data, rel	XCH A, R4	DJNZ R4, rel	MOV A, R4	MOV R4, A
D	INC R5	DEC R5	ADD A, R5	ADDC A, R5	ORL A, R5	ANL A, R5	XRL A, R5	MOV R5, # data	MOV dir, R5	SUBB A, R5	MOV R5, dir	CJNE R5, # data, rel	XCH A, R5	DJNZ R5, rel	MOV A, R5	MOV R5, A
E	INC R6	DEC R6	ADD A, R6	ADDC A, R6	ORL A, R6	ANL A, R6	XRL A, R6	MOV R6, # data	MOV dir, R6	SUBB A, R6	MOV R6, dir	CJNE R6, # data, rel	XCH A, R6	DJNZ R6, rel	MOV A, R6	MOV R6, A
F	INC R7	DEC R7	ADD A, R7	ADDC A, R7	ORL A, R7	ANL A, R7	XRL A, R7	MOV R7, # data	MOV dir, R7	SUBB A, R7	MOV R7, dir	CJNE R7, # data, rel	XCH A, R7	DJNZ R7, rel	MOV A, R7	MOV R7, A

2Byte	3Byte
2Cycle	4Cycle

FIGURE B–1
Opcode map

C

Instruction Definitions[1]

LEGEND

Symbol	Interpretation
←	is replaced by ...
()	the contents of . . .
(( ))	the data pointed at by . . .
rrr	one of eight registers; 000 = R0, 001 = R1, etc.
dddddddd	data bits
aaaaaaaa	address bits
bbbbbbbb	address of a bit
	indirect addressing using R0 (i = 0) or R1 (i = 1)
eeeeeeee	8-bit relative address

ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7-5, and the second byte of the instruction.

[1] Adapted from 8-Bit Embedded Controllers (270645). Santa Clara, CA: Intel Corporation, 1991, by permission of Intel Corporation.

The subroutine called must therefore start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label "SUBRTN" is a program memory location 0345H. After executing the instruction,

```
ACALL   SUBRTN
```

at location 0123H, the SP contains 09H, internal RAM locations 08H and 09H contain 25H and 01H, respectively, and the PC contains 0345H.

Bytes: 2

Cycles: 2

Encoding: aaal000l aaaaaaaa

Note: aaa = A10-A8 and aaaaaaaa = A7-A0 of the destination address.

Operation: (PC)• •(PC) + 2

(SP) ← (SP) + 1

((SP)) ← (PC7-PC0)

(SP) ← (SP) + 1

((SP)) ← (PC15-PC8)

(PC10-PC0) ←page address

ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the accumulator,leaving the result in the accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number is produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source-operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AH (10101010B). The instruction,

```
ADD    A,R0
```

leaves 6DH (01101110B) in the accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A,Rn

Bytes: 1

Cycles: 1

Encoding: 00101rrr

Operation: (A) ← (A) + (Rn)

ADD A,direct

Bytes: 2

Cycles: 1

Encoding: 00100101 aaaaaaaa

Operation: (A) ← (A) + (direct)

ADD A,@Ri

Bytes: 1

Cycles: 1

Encoding: 0010011i

Operation: (A) ← (A) + ((Ri))

ADD A,#data

Bytes: 2

Cycles: 1

Encoding: 00100100 dddddddd

Operation: (A) ← (A) + #data

ADDC A,<src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag, and the accumulator contents, leaving the result in the accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carryout of bit 6 but not out of bit 7, or a carryout of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number is produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source-operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The instruction,

```
ADDC A,Rn
```

leaves 6EH (01101110B) in the accumulator with AC cleared and both the carry flag and OV set to 1.

ADDC A,Rn

Bytes:	1
Cycles:	1
Encoding:	00110rrr
Operation:	(A) ← (A) + (C) + (Rn)

ADDC A,direct

Bytes:	2
Cycles:	1
Encoding:	00110101 aaaaaaaa
Operation:	(A) ← (A) + (C) + (direct)

ADDC A,@Ri

Bytes:	1
Cycles:	1
Encoding:	0011011i
Operation:	(A) ← (A) + (C) + (Ri)

ADDC A,#data

Bytes:	2
Cycles:	1
Encoding:	00110100 dddddddd
Operation:	(A) ← (A) + (C) + #data

AJMP addr11

Function: Absolute Jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC *(after* incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2K block of program memory as the first byte of the instruction following AJMP.

Example: The label "JMPADR" is at program memory location 0123H. The instruction,

```
AJMP JMPADR
```

is at location 0345H and loads the PC with 0123H.

Bytes: 2

Cycles: 2

Encoding: aaa0000l aaaaaaaa

Note: aaa = A10-A8 and aaaaaaaa = A7-A0 of the destination address.

Operation: (PC) ← (PC) + 2

(PC10-PC0) ← page address

ANL <dest-byte>,<src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 55H (010101011B), then the instruction,

```
ANL A,R0
```

leaves 41H (01000001H) in the accumulator.

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared is either a constant contained in the instruction or a value computed in the accumulator at run-time. The instruction,

```
ANL P1,#01110011B
```

clears bits 7, 3, and 2 of output Port 1.

ANL A,Rn

Bytes:	1
Cycles:	1
Encoding:	01011rrr
Operation:	(A) ← (A) AND (Rn)

ANL A,direct

Bytes:	2
Cycles:	1
Encoding:	0101010l aaaaaaaa
Operation:	(A) ← (A) AND (direct)

ANL A,@Ri

Bytes:	1
Cycles:	1
Encoding:	0101011i
Operation:	(A) ← (A) AND ((Ri))

ANL A,#data

Bytes:	2
Cycles:	1
Encoding:	01010100 dddddddd
Operation:	(A) ← (A) AND #data

ANL direct,A

Bytes:	2

Cycles: 1

Encoding: 01010010 aaaaaaaa

Operation: (direct) ← (direct) AND (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding: 01010011 aaaaaaaa dddddddd

Operation: (direct) ← (direct) AND #data

ANL C,<src-bit>

Function: Logical-AND for bit variables

Description: If the Boolean value of the source bit is a logical O, then clear the carry flag; otherwise, leave the carry flag in its current state. A slash (/) preceding the operand in the assembly language program indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected*. No other flags are affected.

Only direct addressing is allowed for the source operand.

Example: *Set* the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0:

```
MOV C,P1.0      ;LOAD C WITH INPUT PIN STATE
ANL C,ACC.7     ;AND CARRY WITH ACC BIT 7
ANL C,/OV       ;AND WITH INVERSE OF OV FLAG
```

ANL C,bit

Bytes: 2

Cycles: 2

Encoding: 10000010 bbbbbbbb

Operation: (C) ← (C) AND (bit)

ANL C,bit

Bytes: 2

Cycles:2

Encoding: 10110000 bbbbbbbb

Operation: (C) ← (C) AND NOT(bit)

CALL (See ACALL, or LCALL) CJNE

<dest-byte>,<src-byte>,rel

Function: Compare and Jump if Not Equal

Description: CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte>is less than the unsigned integer value of <src-byte>; otherwise, the carry flag is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence

```
            CJNE R7,#60H,NOT_EQ
;           . . .  . . .        ;R7 = 60H
NOTEQ:      JC REGROW            ;IF R7 < 60H
;           . . .  . . .         ;R7 > 60H
REG LOW     . . .  . . .         ;R7 < 60H
```

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 34H, then the instruction,

```
WAIT:       CJNE A,P1,WAIT
```

clears the carry flag and continues with the next instruction, since the accumulator does equal the data read from Port 1. (If some other value is inputted on P1, the program loops at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding: 1011010l aaaaaaaa eeeeeeee

Operation: (PC) ← (PC) + 3

```
IF (A)<>(direct)
THEN
(PC) ← (PC) + relative address
IF (A) <(direct)
THEN
    (C) ←1
ELSE
    (C) ←0
```

CJNE A#data,rel

Bytes: 3

Cycles: 2

Encoding: 10110100 dddddddd eeeeeeee

Operation:

```
(PC) ← (PC) + 3
IF (A) <> data
THEN
(PC) ← (PC) + relative address
IF (A) < data
THEN
(C) ← 1
ELSE
(C) ← 0
```

CJNE Rn,#data,rel

Bytes: 3

Cycles: 2

Encoding: 10111rrr dddddddd eeeeeeee

Operation:

```
(PC) ← (PC) + 3
IF (Rn) <> data
    THEN
        (PC) ← (PC) + relative address
IF (Rn)<data
    THEN
        (C) ← 1
    ELSE
        (C) ← 0
```

CJNE @Ri,#data,rel

Bytes: 3

Cycles: 2

Encoding: 1011011i eeeeeeee

Operation:

```
(PC) ← (PC) + 3
  IF ((Ri)) <> data
      THEN
          (PC) ← (PC) + relative address
  IF ((Ri)) < data
      THEN
          (C) ← 1
      ELSE
          (C) ← 0
```

CLR A

Function: Clear Accumulator

Description: The accumulator is cleared (all bits set to 0). No flags are affected.

Example: The accumulator contains 5CH (01011100B). The instruction,

```
CLR   A
```

leaves the accumulator set to 00H (00000000B).

Bytes: 1

Cycles: 1

Encoding: 11100100

Operation: (A) ← 0

CLR bit

Function: Clear bit

Description: The indicated bit is cleared (reset to 0). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

Example: Port 1 has previously been written with 5DH (01011101B). The instruction,

```
CLR P1.2
```

leaves the port set to 59H (01011001B).

CLR C

Bytes: 1

Cycles: 1

Encoding: 11000011

Operation: (C) ← 0

CLR bit

Bytes: 2

Cycles: 1

Encoding: 11000010 bbbbbbbb

Operation: (bit) ← 0

CPL A

Function: Complement Accumulator

Description: Each bit of the accumulator is logically complemented (Vs complement). Bits that previously contained a 1 are changed to a 0 and vice versa. No flags are affected.

Example: The accumulator contains 5CH (01011100B). The instruction,

```
CPL     A
```

leaves the accumulator set to 0A3H (10100011B).

Bytes: 1

Cycles: 1

Encoding: 11110100

Operation: (A) ← NOT(A)

CPL bit

Function: Complement bit

Description: The bit variable specified is complemented. A bit that was a 1 is changed to 0 and vice versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data is from the output data latch, *not* the input pin.

Example: Port 1 has previously been written with 5BH (01011011B). The instructions,

```
CPL      P1.1
CPL      P1.2
```

leave the port set to 5BH (01011011B).

CPL C

Bytes: 1

Cycles: 1

Encoding: 10110011

Operation: (C) ← NOT(C)

CPL bit

Bytes: 2

Cycles: 1

Encoding: 1010010 bbbbbbbb

Operation: (bit) ← NOT(bit)

DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the 8-bit value in the accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two 4-bit digits. Any ADD or ADDC instruction may be used to perform the addition.

If accumulator bits 3-0 are greater than 9 (xxxx1010-xxxx1111), or if the AC flag is 1, 6 is added to the accumulator, producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order 4-bit field propagated through all high-order bits, but it does not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed 9 (1010xxxx-1111xxxx), these high-order bits are incremented by 6, producing the proper BCD digit in the high-order bits, but not clearing the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 99, allowing precision decimal addition. OV is not affected.

All of the above occurs during one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the accumulator, depending on initial accumulator and PSW conditions.

Note: DA A cannot simply convert a hexadecimal number in the accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example: The accumulator holds the value 56H (01010l10B), representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B), representing the packed BCD digits of the decimal 67. The carry flag is set. The instructions,

```
ADDC A,R3
DA   A
```

first perform a standard 2s-complement binary addi-tion, resulting in the value 0BEH (1011110B) in the accumulator. The carry and auxiliary-carry flag are cleared.

The decimal adjust instruction then alters the accu-mulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag is set by the decimal adjust instruction, indicating that a decimal overflow occurr-ed. The true sum of 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the accumulator initially holds 30H (representing the digits 30 decimal), then the instructions,

```
ADD    A,#99H
DA     A
```

leave the carry set and 29H in the accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 - 1 = 29.

Bytes: 1

Cycles: 1

Encoding: 11010100

Operation: (Assume the contents of the accumulator are BCD.)

```
IF [[(A3-A0)>9] AND [(AC = 1]]
   THEN (A3-A0) ← (A3-A0) + 6
AND
  IF [[A7-A4)>9] AND [(C) = 1]]
  THEN (A7-A4) ← (A7-A4) + 6)
```

DEC BYTE

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H underflows to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data is from the output data latch, *not* the input pins.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instructions,

```
DEC @R0
DEC R0
DEC @R0
```

leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1

Cycles: 1

Encoding: 00010100

Operation: (A) ← (A) - 1

DEC Rn

Bytes: 1

Cycles: 1

Encoding: 00011rrr

Operation: (Rn) ← (Rn) - 1

DEC direct

Bytes: 2

Cycles: 1

Encoding: 00010101 aaaaaaaa

Operation: (direct) ← (direct) - 1

DEC @Rib

Bytes: 1

Cycles: 1

Encoding: 0001011i

Operation: ((Ri)) ← ((Ri)) — 1

DIV AB

Function: Divide

Description: DIV AB divides the unsigned 8-bit integer in the accumulator by the unsigned 8-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared.

Exception: If B originally contained 00H, the values returned in the accumulator and B-register are undefined and the overflow flag is set. The carry flag is cleared in any case.

Example: The accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The instruction,

```
DIV AB
```

leaves 13 in the accumulator (0DH or 00001101B) and the value 17 (11H or 0001000B) in B, since 251 = 13 x 18 + 17. Carry and OV are both cleared.

Bytes: 1

Cycles: 4

Encoding: 10000100

Operation: (A) ← QUOTIENT OF (A)/(B)

(B) ← REMAINDER OF (A)/(B)

DJNZ <byte>,<rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by the first operand, andbranches to the address indicated by the second operand if the resulting value is not 0. An original value of 00H underflows to 0FFH. No flags are affected. The branch destination is computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data are read from the output data latch, *not* the input pins.

Example: Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The instructions,

```
DJNZ 40H,LABEL1
DJNZ 40H,LABEL2
DJNZ 40H,LABEL3
```

cause a jump to the instruction at LABEL2 with the values 00H, 6FH, and 15H in the 3 RAM locations. The first jump is not taken because the result was 0.

This instruction provides a simple way to execute a program loop a given number of times, or to add a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instructions,

```
        MOV R2,#8
TOGGLE: CPL P2.7
        DJNZ R2,TOGGLE
```

toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse lasts three machine cycles, two for DJNZ and one to alter the pin.

DJNZ Rn,rel

Bytes: 2

Cycles: 2

Encoding: 11011rrr eeeeeeee

Operation:

```
(PC) ← (PC) + 2
(Rn) ← (Rn) — 1
IF (Rn)0
   THEN
         (PC) ← (PC) + byte_2
```

DJNZ direct,rel

Bytes: 3

Cycles: 2

Encoding: 11010101 aaaaaaaa eeeeeeee

Operation:

```
(PC) ← (PC) + 2
(direct) ← (direct) — 1
IF (direct) <> 0
   THEN
         (PC) ← (PC) + byte_2
```

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data is from the output data latch, *not* the input pins.

Example: Register 0 contains 7EH (0111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instructions,

```
INC @R0
INC R0
INC @R0
```

leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H.

INC A

Bytes: 1

Cycles: 1

Encoding: 00000100

Operation: (A) ← (A) + 1

INC Rn

Bytes: 1

Cycles: 1

Encoding: 00001rrr

Operation: (Rn) ← (Rn) + 1

INC direct

Bytes: 2

Cycles: 1

Encoding: 00000101 aaaaaaaa

Operation: (direct) ← (direct) + 1

INC @Ri

Bytes: 1

Cycles: 1

Encoding: 0000011i

Operation: ((Ri)) ← ((Ri)) + 1

INC DPTR

Function: Increment Data Pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH). No flags are affected.

This is the only 16-bit register that can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The instructions,

```
INC DPTR
INC DPTR
INC DPTR
```

change DPH and DPL to 13H and 01H.

Bytes: 1

Cycles: 2

Encoding: 10100011

Operation: (DPTR) ← (DPTR) + 1

JB bit, rel

Function: Jump if Bit set

Description: If the indicated bit is a 1, jump to the address indicated; otherwise, proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Example: The data present at input port 1 are 11001010B. The accumulator holds 56H (01010110B). The instructions,

```
JB P1.2,LABEL1
JB ACC.2,LABEL2
```

cause program execution to branch to the instruction at LABEL2.

Bytes: 3

Cycles: 2

Encoding: 0100000 bbbbbbbb eeeeeeee

Operation:

```
(PC) ← (PC) + 3
IF (bit) = 1
    THEN
        (PC) ← (PC) + byte_2
```

JBC bit,rel

Function: Jump if Bit set and Clear bit

Description: If the indicated bit is 1, clear it and branch to the address indicated; otherwise proceed with the next instruction. *The bit is not cleared if it is already a 0.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: The accumulator holds 56H (01010110B). The instructions,

```
JBC ACC.3,LABEL1
JBC ACC.2,LABEL2
```

cause program execution to continue at the instruction identified by LABEL2, with the accumulator modified to 52H (01010010B).

Bytes: 3

Cycles: 2

Encoding: 00010000 bbbbbbbb eeeeeeee

Operation:

```
(PC) ← (PC) + 3
IF(bit) = 1
   THEN
         (bit) ← 0
(PC) ← (PC) + byte_2
```

JC rel

Function: Jump if Carry is set

Description: If the carry flag is set, branch to the address indicated; otherwise roceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The instructions,

```
JC LABEL1
CPL C
JC LABEL2
```

set the carry, and cause program execution to continue at the instruction identified by LABEL2.

Bytes: 2

Cycles: 2

Encoding: 01000000 eeeeeeee

Operation:

```
(PC) ← (PC) + 2
IF (C) = 1
        THEN
              (PC) + (PC) + byte_2
```

JMP <dest>(See SJMP, AJMP, or LJMP)

JMP@A+DPTR

Function: Jump indirect

Description: Add the 8-bit unsigned contents of the accumulator with the 16-bit pointer, and load the resulting sum to the program counter. This is the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the accumulator nor the data pointer is altered. No flags are affected.

Example: An even number from 0 to 6 is in the accumulator. The following instructions branch to 1 of 4 AJMP instructions in a jump table starting at JMP_TBL:

```
                MOV DPTR,#JMP_TBL
                JMP @A + DPTR
       JMP_TBL: AJMP LABEL0
                AJMP LABEL1
                AJMP LABEL2
                AJMP LABEL3
```

If the accumulator equals 04H when starting this sequence, execution jumps to LABEL2. Remember that AJMP is a two-byte instruction, so the jump instruction starts at every other address.

Bytes: 1

Cycles: 2

Encoding: 01110011

Operation: (PC) ← (PC) + (A) + (DPIR)

JNB bit,rel

Function: Jump if Bit Not set

Description: If the indicated bit is a 0, branch to the indicated address; otherwise, proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified*. No flags are affected.

Example: The data present at input Port 1 are 110010108. The accumulator holds 56H (01010110B). The instructions,

```
JNB P1.3,LABEL1
JNB ACC.3,LABEL2
```

cause program execution to continue at the instruction at LABEL2.

Bytes: 3

Cycles: 2

Encoding: 00110000 bbbbbbbb eeeeeeee

Operation:

```
(PC) ← (PC) + 3
IF (bit) = 0
        THEN
            (PC) ← (PC) + byte_2
```

JNC rel

Function: Jump if Carry not set

Description: If the carry flag is a 0, branch to the address indicated; otherwise, proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The instructions,

```
JNC LABEL1
CPL C
JNC LABEL2
```

clear the carry flag and cause program execution to continue at the instruction identified by LABEL2.

Bytes: 2

Cycles: 2

Encoding: 01010000 eeeeeeee

Operation:

```
(PC) ← (PC) + 2
IF (C) = 0
    THEN
        (PC) ← (PC) + byte_2
```

JNZ rel

Function: Jump if accumulator Not Zero

Description: If any bit of the accumulator is a 1, branch to the indicated address; otherwise, proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally holds 00H. The instructions,

```
JNZ LABEL1
INC A
JNZ LABEL2
```

set the accumulator to 01H and continue at LABEL2.

Bytes: 2

Cycles: 2

Encoding: 01110000 eeeeeeee

Operation:

```
(PC) ← (PC) + 2
IF (A)<>0
        THEN
              (PC) ← (PC) + byte_2
```

JZ rel

Function: Jump if accumulator Zero

Description: If all bits of the accumulator are 0, branch to the indicated address; otherwise, proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally holds 01H. The instructions,

```
JZ LABEL1
DEC A
JZ LABEL2
```

change the accumulator to 00H and cause program execution to continue at the instruction identified by LABEL2.

Bytes: 2

Cycles: 2

Encoding: 01100000 eeeeeeee

Operation:

```
(PC) ← (PC) + 2
IF (A) = 0
       THEN
              (PC) ← (PC) + byte_2
```

LCALL addr16

Function: Long Call to subroutine

Description: LCALL calls a subroutine located at the indicated address. The instruction adds 3 to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low-byte first), incrementing the stack pointer by 2. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K-byte program memory address space. No flags are affected.

Example: Initially the stack pointer equals 07H. The label "SUBRTN" is assigned to program memory location 1234H. After executing the instruction,

```
LCALL     SUBRTN
```

at location 0123H, the stack pointer contains 09H, internal RAM locations 08H and 09H contain 26H and 01H, and the PC contains 1234H.

Bytes: 3

Cycles: 2

Encoding: 00010010 aaaaaaaa aaaaaaaa

Note: Byte 2 contains address bits 16-8, byte 3 contains address bits 7-0.

Operation: (PC) ← (PC) + 3

(SP) ← (SP) + 1

((SP)) ← (PC7-PC0)

(SP) ← (SP) + 1

((SP)) ← (PC15-PC8)

(PC) ← addr15 - addr0

LJMP addr16

Function: Long Jump

Description: LJMP causes an unconditional branch to the indicated address by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label "JMPADR" is assigned to the instruction at program memory location 1234H. The instruction,

```
LJMP JMPADR
```

at location 0123H loads the program counter with 1234H.

Bytes: 3

Cycles: 2

Encoding: 00010010 aaaaaaaa aaaaaaaa

Note*:* Byte 2 contains address bits 16-8, byte 3 contains address bits 7-0.

Operation: (PC) ← addr16-addr0

MOV<dest-byte>,<src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input Port 1 are 11001010B (0CAH). The instructions,

```
MOV R0,#30H     ;R0← 30H
MOV A,@R0       ;A← 40H
MOV R1,A;       ;R1← 40H
MOV B,@R1       ;B← 10H
MOV @R1,P1      ;RAM← 40H)
                ; ← 0CAH
MOVE P2,P1      ;P2← 00CAH
```

leave the value 30H in register 0, 40H in both the accumulator and register 1, 10H in register B, and 0CAH (11001010B) both in RAM location 40H and output on Port 2.

MOV A,Rn

Bytes: 1

Cycles: 1

Encoding: 11101ra

Operation: (A) ← (Rn)

MOV A,direct

Bytes: 2

Cycles: 1

Encoding: 11100101 aaaaaaaa

Operation: (A) ← (direct)

Note: MOV A,ACC is not a valid instruction.

MOV A,@Ri

Bytes: 1

Cycles: 1

Encoding:	1110011i
Operation:	(A) ← ((Ri))

MOV A,#data

Bytes:	2
Cycles:	1
Encoding:	01110100 dddddddd
Operation:	(A) ← #data

MOV Rn,A

Bytes:	1
Cycles:	1
Encoding:	01111rrr
Operation:	(Rn) ← (A)

MOV Rn,direct

Bytes:	2
Cycles:	2
Encoding:	10101rrr
Operation:	(Rn) ← (direct)

MOV Rn,#data

Bytes:	2
Cycles:	1
Encoding:	01111rrr dddddddd
Operation:	(Rn) ← #data

MOV direct,A

Bytes:	2
Cycles:	1
Encoding:	11110101 aaaaaaaa
Operation:	(direct) ← A

MOV direct,Rn

Bytes: 2

Cycles: 2

Encoding: 10001m aaaaaaaa

Operation: (direct) ← (Rn)

MOV direct,direct

Bytes: 3

Cycles: 2

Encoding: 10000101 aaaaaaaa aaaaaaaa

Note: Byte 2 contains the source address;
Byte 3 contains the destination address.

Operation: (direct) ← (direct)

MOV direct,@Ri

Bytes: 2

Cycles: 2

Encoding: 1000011i aaaaaaaa

Operation: (direct) ← ((Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding: 01110101 aaaaaaaa dddddddd

Operation: (direct) ← #data

MOV @Ri,A

Bytes: 1

Cycles: 1

Encoding: 1111011i

Operation: ((Ri)) ← A

MOV @Ri,direct

Bytes:	2
Cycles:	2
Encoding:	1010011i aaaaaaaa
Operation:	((Ri)) ← (direct)

MOV @Ri,#data

Bytes:	2
Cycles:	1
Encoding:	0111011i dddddddd
Operation:	((Ri)) ← #data

MOV <dest-bit>,<src-bit>

Function: Move bit variable

Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input Port 2 is 11000101B. The data previously written to output Port 1 are 35H (00110101B). The instructions,

```
MOV P1.3,C
MOV C,P3.3
MOV P1.2,C
```

leave the carry cleared and change Port 1 to 39H (00111001B).

MOV C,bit

Bytes:	2
Cycles:	1
Encoding:	10100010 bbbbbbbb
Operation:	(C) ← (bit)

MOV bit,C

Bytes:	2

Cycles: 2

Encoding: 10010010 bit address

Operation: (bit) ← (C)

MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: The data pointer is loaded with the 16-bit constant indicated. The 16-bit constant is located in the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

Example: The instruction,

```
MOV DPTR,#1234H
```

loads the value 1234H into the data pointer: DPH holds 12H and DPL holds 34H.

Bytes: 3

Cycles: 2

Encoding: 10010000 dddddddd dddddddd

Note: Byte 2 contains immediate data bits 15-8, byte 3 contains bits 7-0.

Operation: (DPTR) ← #data 16

MOVC A,@A+<base-reg>

Function: Move code byte or constant byte.

Description: The MOVC instructions load the accumulator with a code byte or constant byte from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit accumulator contents and the contents of a 16-bit base register, which may be either the data pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added to the accumulator; otherwise, the base register is not altered. Sixteen-bit addition is performed so a carryout from the low-order 8 bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the accumulator. The following subroutine translates the value in the accumulator to 1 of 4 values defined by the DB (define byte) directive.

```
REL_PC:     INC  A
            MOVC A,@A + PC
            RET
            DB   66H
DB 77H
DB 88H
DB 99H
```

If the subroutine is called with the accumulator equal to 01H, it returns with 77H in the accumulator. The INC A before the MOVC instruction is needed to "get around" the RET instruction above the table. If several bytes of code separate the MOVC from the table, the corresponding number should be added to the accumulator instead.

MOVC A,@A+DPTR

Bytes: 1

Cycles: 2

Encoding: 10010011

Operation: (A) ← ((A) + (DPTR))

MOVC A,@A+PC

Bytes: 1

Cycles: 2

Encoding: 10000011

Operation: (PC) ← (PC) + 1
(A) ← ((A) + (PC))

MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the accumulator and a byte of external data memory; hence the "X" appended to the MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the data pointer generates a 16-bit address. P2 outputs the high-order eight address bits (the contents of DPH), while P0 multiplexes the low-order eight bits (DPL) with data. The P2 special function register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be addressed via the data pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example: An external 256-byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/TIMER) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal 1/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence

```
MOVX A,@Ri
MOVX @R0,A
```

copies the value 56H into both the accumulator and external RAM location 12H.

MOVX A,@Ri

Bytes:	1
Cycles:	2
Encoding:	1110001i
Operation:	(A) ← ((Ri))

MOVX A,@DPTR

Bytes:	1
Cycles:	2
Encoding:	11100000
Operation:	(A) ← ((DPTR))

MOVX @Ri,A

Bytes:	1
Cycles:	2
Encoding:	11110011
Operation:	((Ri)) ← (A)

MOVX @DPTR,A

Bytes: 1

Cycles: 2

Encoding: 11110000

Operation: (DPTR) ← (A)

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned 8-bit integers in the accumulator and register B. The low-order byte of the 16-bit product is left in the accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

```
MUL AB
```

gives the product 12,800 (3200H), so B is changed to 32H (00110010B) and the accumulator is cleared. The overflow flag is set, and carry is cleared.

Bytes: 1

Cycles: 4

Encoding: 10100100

Operation: (B) ← HIGH BYTE OF (A) x (B)

(A) ← LOW BYTE OF (A) x (B)

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no register or flags are affected.

Example: It is desired to produce a low-going output pulse on bit 7 of Port 2 lasting exactly five cycles. A simple SETB/CLR sequence generates a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instructions,

```
CLR P2.7
NOP
```

```
NOP
NOP
NOP
SETB P2.7
```

Bytes: 1

Cycles: 1

Encoding: 00000000

Operation: (PC) ← (PC) + 1

ORL <dest-byte>,<src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The 2 operands allow 6 addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101) then the instruction,

```
ORL A,R0
```

leaves the accumulator holding the value 0D7H (11010111B).When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the accumulator at run-time. The instruction,

```
ORL P1,00110010B
```

sets bits 5, 4, and 1 of output Port 1.

ORL A,Rn

Bytes: 1

Cycles: 1

Encoding: 01001m

Operation: (A) ← (A) OR (Rn)

ORL A,direct

Bytes: 2

Cycles: 1

Encoding: 01000101 aaaaaaaa

Operation: (A) ← (A) OR (direct)

ORL A,@Ri

Bytes: 1

Cycles: 1

Encoding: 0100011i

Operation: (A) ← (A) OR ((Ri))

ORL A,#data

Bytes: 2

Cycles: 1

Encoding: 01000100 dddddddd

Operation: (A) ← (A) OR #data

ORL direct,A

Bytes: 2

Cycles: 1

Encoding: 01000010 aaaaaaaa

Operation: (direct) ← (direct) OR (A)00

ORL direct,#data

Bytes: 3

Cycles: 2

Encoding: 01000011 aaaaaaaa dddddddd

Operation: (direct) ← (direct) OR #data

ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, or OV = 0.

```
MOV C,P1.0   ;LOAD CY WITH INPUT PIN Pi.0
ORL C,ACC.7 ;OR CY WITH THE ACC, BIT7
ORL C,/OV    ;OR CY WITH INVERSE OF OV
```

ORL C,bit

Bytes: 2

Cycles: 2

Encoding: 01110010 bbbbbbbb

Operation: (C) ← (C) OR (bit)

ORL C,/bit

Bytes: 2

Cycles: 2

Encoding: 10100000 bbbbbbbb

Operation: (C) ← (C) OR NOT(bit)

POP direct

Function: Pop from stack

Description: The contents of the internal RAM location addressed by the stack pointer are read, and the stack pointer is decremented by 1. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The stack pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The instructions,

```
POP     DPH
POP     DPL
```

leave the stack pointer equal to the value 30H and the data pointer set to 0123H. At this point the instruction,

```
        POP     SP
```

leaves the stack pointer set to 20H. Note that in this special case the stack pointer is decremented to 2FH before being loaded with the value popped (20H).

Bytes: 2

Cycles: 2

Encoding: 11010000 aaaaaaaa

Operation: (direct) ← ((SP))

(SP) ← (SP) — 1

PUSH direct

Function: Push onto stack

Description: The stack pointer is incremented by 1. The contents of the indicated variable are then copied into the internal RAM location addressed by the stack pointer. Otherwise, no flags are affected.

Example: On entering an interrupt routine, the stack pointer contains 09H. The data pointer holds the value 0123H. The instructions,

```
        PUSH     DPL
        PUSH     DPH
```

leave the stack pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Bytes: 2

Cycles: 2

Encoding: 11000000 aaaaaaaa

Operation: (SP) ← (SP) + 1

((SP)) ← (direct)

RET

Function: Return from subroutine

Description: RET pops the high- and low-order bytes of the PC successively from the stack, decrementing the stack pointer by 2. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The stack pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

```
RET
```

leaves the stack pointer equal to the value 09H. Program execution continues at location 0123H.

Bytes: 1

Cycles: 2

Encoding: 00100010

Operation: (PC1-PC8) ← ((SP))

(SP) ← (SP) — 1

(PC7-PC0) ← ((SP))

(SP) ← (SP) — 1

RETI

Function: Return from interrupt

Description: RETI pops the high- and low-order bytes of the PC successively from the stack and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by 2. No other registers are affected; the PSW is *not* automatically restored to its preinterrupt status. Program execution continues at the resulting address, which is generally an instruction immediately after the point at which the interrupt request is detected. If a lower- or same-level interrupt is pending when the RETI instruction is executed, then one instruction is executed before the pending interrupt is processed.

Example: The stack pointer originally contains the value 0BH. An interrupt is detected during the instruction ending at location 0123H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

```
RETI
```

leaves the stack pointer equal to 09H and returns program execution to location 0123H.

Bytes: 1

Cycles: 2

Encoding: 00110010

Operation: (PC15-PC8) ← ((SP))

(SP) ← (SP) — 1

(PC7-PC0) ← ((SP))

(SP) ← (SP) — 1

RL A

Function: Rotate Accumulator Left

Description: The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The accumulator holds the value 0C5H (11000101B). The instruction,

```
RL      A
```

leaves the accumulator holding the value 8BH (10001011B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding: 00100011

Operation: (An + 1) ← (An), n = 0-6

(A0) ← (A7)

RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The accumulator holds the value 0C5H (11000101B), and the carry is 0. The instruction,

```
RLC  A
```

leaves the accumulator holding the value 8BH (10001011B) with the carry set.

Bytes: 1

Cycles: 1

Encoding: 00110011

Operation: (An + 1) ← (An), n = 0-6

(A0) ← (C)

(C) ← (A7)

RR A

Function: Rotate Accumulator Right

Description: The eight bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The accumulator holds the value 0C5H (11000101B) with the carry unaffected. The instruction,

```
RR      A
```

leaves the accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding: 00000011

Operation: (An) ← (An+1), n = 0 — 6

(A7) ← (A0)

RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The accumulator holds the value 0C5H (11000101B), the carry is 0. The instruction,

```
RRC  A
```

leaves the accumulator holding the value 62H (01100010B) with the carry set.

Bytes: 1

Cycles: 1

Encoding: 00010011

Operation: (An) ← (An + 1), n = 0 — 6

(A7) ← (C)

(C) ← (A0)

SETB <bit>

Function: Set Bit

Description: SETB sets the indicated bit to 1. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The instructions,

```
SETB C
SETB P1.0
```

leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 1

Encoding: 11010011

Operation: (C) ← 1

SETB bit

Bytes: 2

Cycles: 1

Encoding: 11010010 bbbbbbbb

Operation: (bit) ← 1

SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

Example: The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,

```
SJMP RELADR
```

assembles into location 0100H. After the instruction is executed, the PC contains the value 0123H.

(Note: Under the above conditions, the instruction following SJMP is at 0102H. Therefore, the displacement byte of the instruction is the relative offset (0123H - 0102H = 21H. Put another way, an SJMP with a displacement of 0FEH is a 1-instruction infinite loop.)

Bytes: 2

Cycles: 2

Encoding: 10000000 eeeeeeee

Operation: (PC) ← (PC) + 2

(PC) ← (PC) + byte_2

SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C is set before executing a SUBB instruction, this indicates that a borrow is needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the accumulator along with the source operand.) AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not into bit 6.

When subtracting signed integers, OV indicates that a negative number is produced when a negative value is subtracted from a positive value, or a positive number is produced when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

```
SUBB A,R2
```

leaves the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A,Rn

Bytes: 1

Cycles: 1

Encoding: 10011rrr

Operation: (A) ← (A) — (C) — (Rn)

SUBB A,direct

Bytes: 2

Cycles: 1

Encoding: 10010101 aaaaaaaa

Operation: (A) ← (A) — (C) — (direct)

SUBB A,@Ri

Bytes: 1

Cycles: 1

Encoding: 1001011i

Operation: (A) ← (A) — (C) — ((Ri))

SUBB A,#data

Bytes: 2

Cycles: 1

Encoding: 10010100 dddddddd

Operation: (A) ← (A) — (C) — #data

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (4-bit fields) of the accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The accumulator holds the value 0C5H (11000101B). The instruction,

```
SWAP A
```

leaves the accumulator holding the value 5CH (01011100B).

Bytes: 1

Cycles: 1

Encoding: 11000100

Operation: (A3-A0) ↔ (A7-A4)

XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the accumulator with the contents of the indicated variable, at the same time writing the original accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

```
XCH A,@R0
```

leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A,Rn

Bytes: 1

Cycles: 1

Encoding: 11001rr

Operation: (A) ↔ (Rn)

XCH A,direct

Bytes: 2

Cycles: 1

Encoding: 11000101 aaaaaaaa

Operation: (A) ↔ (direct)

XCH A,@Ri

Bytes: 1

Cycles: 1

Encoding: 1100011i

Operation: (A) ↔ ((Ri))

XCHD A,@R0

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the accumulator (bits 0-3), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (011110101B). The instruction,

```
XCHD A,@R0
```

leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the accumulator.

Bytes: 1

Cycles: 1

Encoding: 1101011i

Operation: (A3-A0) ↔ ((Ri3-Ri0))

XRL<dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B), then the instruction,

```
XRL A,Rn
```

leaves the accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined in the accumulator at run-time. The instruction,

```
XRL P1,#00110001B
```

complements bits 5, 4, and 0 of output Port 1.

XRL A,Rn

Bytes:	1
Cycles:	1
Encoding:	01101rrr
Operation:	(A) ← (A) ⊕ (Rn)

XRL A,direct

Bytes:	2
Cycles:	1
Encoding:	01100101 aaaaaaaa
Operation:	(A) ← (A) ⊕ (direct)

XRL A,@R0

Bytes:	1
Cycles:	1
Encoding:	0110011i
Operation:	(A) ← (A) ⊕ ((Ri))

XRL A,#data

Bytes:	2
Cycles:	1
Encoding:	01100100 dddddddd
Operation:	(A) ← (A) ⊕ #data

XRL direct,A

Bytes: 2

Cycles: 1

Encoding: 01100010 aaaaaaaa

Operation: (direct) ← (direct) ⊕ (A)

XRL direct,#data

Bytes: 3

Cycles: 2

Encoding: 01100011 aaaaaaaa dddddddd

Operation: (direct) ← (direct) ⊕ #data

D

Special Function Registers[1]

The 8051's special function registers are shown in the SFR memory map in Figure D-1. Blank locations are reserved for future products and should not be written to. The SFRs identified with an asterisk contain bits that are defined as mode or control bits. These registers and their bit definitions are described on the following pages.

Some bits are identified as "not implemented." User software should not write 1s to these bits, since they may be used in future MCS51™ products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

PCON (POWER CONTROL REGISTER)

Symbol:	PCON
Function:	Power control and miscellaneous features
Bit Address:	87H
Bit-Addressable:	No
Summary:	

7	6	5	4	3	2	1	0
SMOD	-	-	-	GF1	GF0	PD	IDL

[1] Adapted from *8-Bit Embedded Controllers* (270645). Santa Clara, CA: Intel Corporation, 1991, by permission OF intel corporation.

Bit Definitions:

Bit Symbol — **Bit Description**

SMOD — Double baud rate. If timer 1 is used to generate baud rate and SMOD = 1, the baud rate is doubled when the serial port is used in modes 1, 2, or 3.

—Not implemented; reserved for future use.

—Not implemented; reserved for future use.

—Not implemented; reserved for future use.

GF1 — General purpose flag bit 1.

GF0 — General purpose flag bit 0.

PD — Power down bit. Setting this bit activates power down operation in the CMOS version of the 8051.[2]

IDL — Idle mode bit. Setting this bit activates idle mode operation in the CMOS versions of the 8051.[2]

8 Bytes

F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW*								D7
C8	T2CON*		RCAP2L	RCAP2H	TL2	TH2			CF
C0									C7
B8	IP*								BF
B0	P3								B7
A8	IE*								AF
A0	P2								A7
98	SCON*	SBUF							9F
90	P1								97
88	TCON*	TMOD*	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PSW*	87

↑

Bit addressable

*SFRs containing mode or controll bits

FIGURE D-P3

Special function register memory map

[2] 1f 1s are writen to PD and IDL at the same time, PD take precedence

TCON (TIMER/COUNTER CONTROL REGISTER)

Symbol:	TCON
Function:	Timer/Counter Control
Bit Address:	88H
Bit-Addressable:	Yes

Summary:

7	6	5	4	3	2	1	0
TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**

Bit Definitions:

Bit		**Bit**	
Symbol	**Position**	**Address**	**Description**
TF1	TCON.7	8FH	Timer 1 overflow flag. Set by hardware when the timer/counter 1 overflow, cleared by software or by hardware as processor vectors to the interrupt service routine.
TR1	TCON.6	8EH	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 ON/OFF.
TF0	TCON.5	8DH	Timer overflow flag. (See TF1)
TR0	TCON.4	8CH	Timer 0 run control fit. (See TR1)
IE1	TCON.3	8BH	External interrupt 1 edge flag. Set by hardware when external interrupt edge is detected; cleared by hardware when interrupt is processe.
IT1	TCON.2	8AH	Interrupt 1 type control bit. Set/cleared by software to specify failing-edge/low-level triggered exteITll interrupt.
IE0	TCON.1	89H	External interrupt 0 edge flag. (See IE1)
IT0	TCON.0	88H	Interrupt 0 type control bit. (See IT1)

SCON (SERIAL CONTROL REGISTER)

Symbol:	SCON
Function:	Serial Port Control
Bit Address:	98H
Bit-Addressable:	Yes

TABLE D-1

SM0	SM1	Mode	Description	Baud Rate
0	0	0	Shift Register	F^{OSC} ¸ 12*
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	F^{OSC} ¸ 64 or F^{OSC} ¸ 32
1	1	3	9SMt UART	Variable

*F^{OSC} is the oscillator frequency of the 8051 IC. Typically, this is derived from a crystal source of 12 MHz.

Summary:

7	6	5	4	3	2	1	0
SM0	**SM1**	**SM2**	**REN**	**TB8**	**RB8**	**TI**	**RI**

Bit Definition

Bit		**Bit**	
Symbol	**Position**	**Address**	**Description**
SM0	SCON.7	9FH	Serial port mode bit 0. (See Table D-l)
SM1	SCON.6	9EH	Serial port mode bit 1. (See Table D-l)
SM2	SCON.5	9DH	Serial port mode bit 2. Enable the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3, if SM2 is set to 1, then RI will not be activated if the received ninth data bit (RB8) is 0. In mode l, if SM2 = l, then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0.
REN	SCON.4	9CH	Receiver enable. Set/cleared by software to enable/disable reception.
TB8	SCON.3	9BH	Transmit bit 8. The ninth bit that will be transmitted in modes 2 and 3. Set/cleared by software.
RB8	SCON.2	9AH	Receive bit 8. In modes 2 and 3, RB8 is the ninth data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, TB8 is not used.
TI	SCON.1	99H	Transmit interrupt. Set by hardware at the end of the eighth bit time in mode 0, or at the beginning of the stop bit in the other modes. Must be cleared by software .

RI	SCON.0	98H	Receive interrupt. Set by hardware at the end of the eighth bit time in mode 0, or halfway through the stop bit time in the other modes (except see SM2). Must be cleared by software.

IE (INTERRUPT ENABLE REGISTER)

Symbol:	IE
Function:	Interrupt Enable
Bit Address:	A8H
Bit-Addressable:	Yes
Summary:	

7	6	5	4	3	2	1	0
EA	—	ET2	ES	ET1	EX1	ET0	EX0

Bit Definitions:

Bit		**Bit**	
Symbol	**Position**	**Address**	**Description 1=enable, 0=disable)**
EA	IE.7	0AFH	Enable/disable all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit
—	IE.6	0AEH	Not implemented; reserved for future use.
ET2	IE.5	0ADH	Enable/disable timer 2 overflow or capture interrupt (80 X 2 only).
ES	IE.4	0ACH	Enable/disable serial port interrupt.
ET1	IE.3	0ABH	Enable/disable timer 1 overflow interrupt.
EX1	IE.2	0AAH	Enable/disable external interrupt 1.
ET0	IE.1	0A9H	Enable/disable timer 0 overflow interrupt.
EX0	IE.0	0A8H	Enable/disable external interrupt 0.

IP (INTERRUPT PRIORITY REGISTER)

Symbol:	IP
Function:	Interrupt Priority

Bit Address: 0B8H

Bit-Addressable: Yes

Summary:

7	6	5	4	3	2	1	0
–	–	PT2	PS	PT1	PX1	PT0	PX0

Bit Definitions:

Bit		**Bit**	**Description**
Symbol	**Position**	**Address**	**(1 = high priority, 0 = low priority)**
—	P.7	0BFH	Not implemented reserved for future use.
—	IP.6	0BEH	Not implemented; reserved for future use.
PT2	IP.5	0BDH	Timer 2 interrupt priority level (80 X 2 only).
PS	IP.4	0BCH	Serial port interrupt priority level.
PT1	IP.3	0BBH	Timer 1 interrupt priority level.
PX1	IP.2	0BAH	External interrupt 1 priority level.
PT0	IP.1	0B9H	Timer 0 interrupt priority level.
PX0	IP.0	0B8H	External interrupt 0 priority level.

T2CON (TIMER/COUNTER 2 CONTROL REGISTER)

SyIP.H: T2CON

Function: Timer/Counter 2 Control

Bit Address: C8H

Bit-Addressable: Yes

Summary:

7	6	5	4	3	2	1	0
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

Bit Definitions:

Bit		**Bit**	
Symbol	**Position**	**Address**	**Description**
TF2	T2CON.7	0CFH	Timer 2 overflow flag. Set by hardware and cleared by software. TF2 cannot be set when either RCLK = 1 or TCLK = 1.

EXF2	T2CON.6	0CEH	Timer 2 external flag. Set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the timer interrupt service routine. EXF2 must be cleared by software.
RCLK	T2CON.5	0CDH	Receive clock. When set, causes the serial port to use timer 2 overflow pulses for its receive clock in modes 1 and 3. RCLK = 0 causes timer 1 overflow to be used for the receive clock.
TCLK	T2CON.4	0CCH	Transmit clock. When set, causes the serial port to use timer overflow pulses for its transmit clock in modes 1 and 3. TCLK = 0 causes timer 1 overflows to be used for the transmit clock.
EXEN2	T2CON.3	0CBH	Timer 2 external enable flag. When set, allows a capture of reload to occur as a result of negative transition on T2EX if timer 2 is not being used to clock to serial port. EXEN2 = 0 causes timer 2 to ignore events at T2EX.
TR2	T2CON.2	0CAH	Timer 2 run bit. Software START/STOP control for timer 2; A logic 1 starts the timer.
C/T2	T2CON.1	0C9H	Counter/timer select for timer 2. 0 = internal timer, 1 = external event counter (falling edge triggered).
CP/RL2	T2CON.0	0C8H	Capture/reload flag. When set captures will occur on negative transitions at T2EXCPIRL2 EXEN2 = 1. When cleared, auto-reloads will occur either with timer 2 overflows or on negative transitions at T2EX when EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to autoreload on timer 2 overflows.

PSW (PROGRAM STATUS WORD)

Symbol:	PSW
Function:	Program Status
Bit Address:	0D0H
Bit Addressable:	Yes

TABLE D-2

RS1	RS0	Bank	Active Addresses
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

Summary:

7	6	5	4	3	2	1	0
CY	AC	F0	RS1	RS1	OV	—	P

Bit Definitions:

Bit	Bit		
Symbol	**Position**	**Address**	**Description**
CY	PSW.7	0D7H	Carry flag. Set if there is a carry-out of bit 7 during an add, or set if there is a borrow into bit 7 during a subtract.
AC	PSW.6	0D6H	Auxiliary carry flag. Set during add instructions if there is a carry-out of bit 3 into bit 4 or if the result in the lower nibble is in the range 0AH to 0FH.
F0	PSW.5	0D5H	Flag 0. Available to the user for general purposes.
RS1	PSW.4	0D4H	Register bank select bit 1. (See Table D-2)
RS0	PSW.3	0D3H	Register bank select bit 0. (See Table D-2)
OV	PSW.2	0D2H	Overflow flag. Set after an addition or subtraction operation if there was anarithmetic overflow (i.e., the signed result is greater than 127 or less than -128).
—	PSW.1	0D1H	
P	PSW.0	0D0H	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of "1" bits in the accumulator.

E

8051 Data Sheet[1]

[1] Reprinted with permission of Intel Corporation from *8-Bit Embeded Controllers* (270645). Santa Clara, CA: Intel Corporation, 1991.

intel®

MCS®-51
8-BIT CONTROL-ORIENTED MICROCOMPUTERS
8031/8051
8031AH/8051AH
8032AH/8052AH
8751H/8751H-8

- **High Performance HMOS Process**
- **Internal Timers/Event Counters**
- **2-Level Interrupt Priority Structure**
- **32 I/O Lines (Four 8-Bit Ports)**
- **64K Program Memory Space**
- **Security Feature Protects EPROM Parts Against Software Piracy**
- **Boolean Processor**
- **Bit-Addressable RAM**
- **Programmable Full Duplex Serial Channel**
- **111 Instructions (64 Single-Cycle)**
- **64K Data Memory Space**

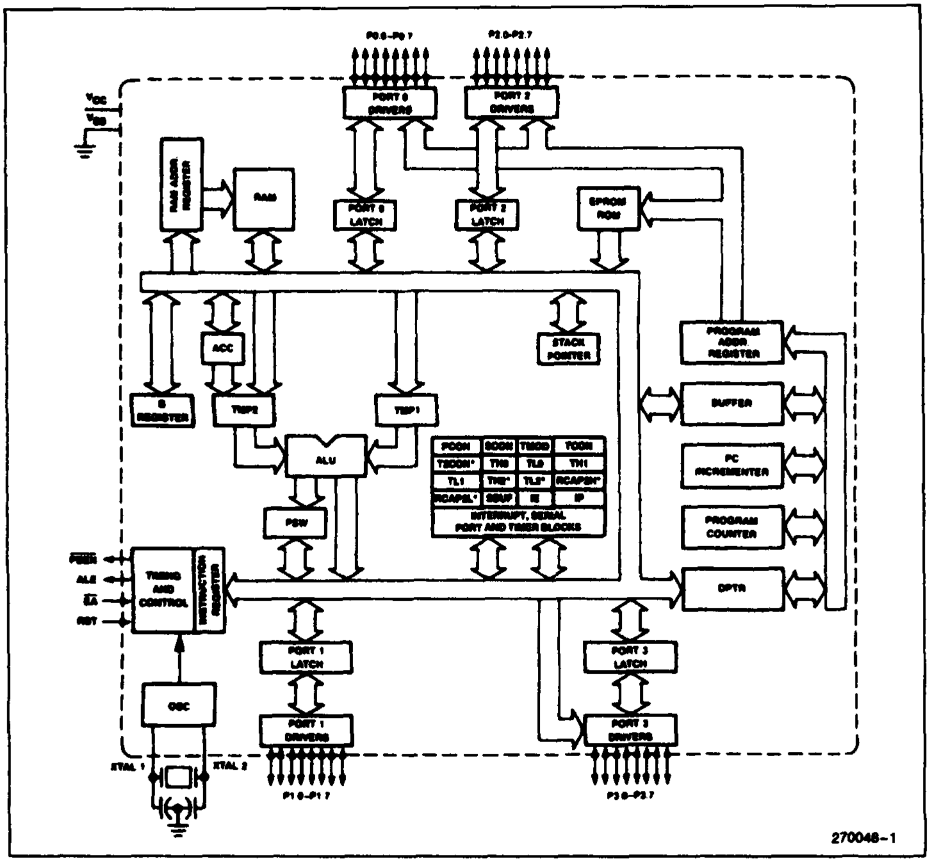
The MCS®-51 products are optimized for control applications. Byte-processing and numerical operations on small data structures are facilitated by a variety of fast addressing modes for accessing the internal RAM. The instruction set provides a convenient menu of 8-bit arithmetic instructions, including multiply and divide instructions. Extensive on-chip support is provided for one-bit variables as a separate data type, allowing direct bit manipulation and testing in control and logic systems that require Boolean processing.

The 8051 is the original member of the MCS-51 family. The 8051AH is identical to the 8051, but it is fabricated with HMOS II technology.

The 8751H is an EPROM version of the 8051AH; that is, the on-chip Program Memory can be electrically programmed, and can be erased by exposure to ultraviolet light. It is fully compatible with its predecessor, the 8751-8, but incorporates two new features: a Program Memory Security bit that can be used to protect the EPROM against unauthorized read-out, and a programmable baud rate modification bit (SMOD). The 8751H-8 is identical to the 8751H but only operates up to 8 MHz.

The 8052AH is an enhanced version of the 8051AH. It is backwards compatible with the 8051AH and is fabricated with HMOS II technology. The 8052AH enhancements are listed in the table below. Also refer to this table for the ROM, ROMless, and EPROM versions of each product.

Device	Internal Memory		Timers/ Event Counters	Interrupts
	Program	Data		
8052AH	8K x 8 ROM	256 x 8 RAM	3 x 16-Bit	6
8051AH	4K x 8 ROM	128 x 8 RAM	2 x 16-Bit	5
8051	4K x 8 ROM	128 x 8 RAM	2 x 16-Bit	5
8032AH	none	256 x 8 RAM	3 x 16-Bit	6
8031AH	none	128 x 8 RAM	2 x 16-Bit	5
8031	none	128 x 8 RAM	2 x 16-Bit	5
8751H	4K x 8 EPROM	128 x 8 RAM	2 x 16-Bit	5
8751H-8	4K x 8 EPROM	128 x 8 RAM	2 x 16-Bit	5



270048-1

Figure 1. MCS®-51 Block Diagram

PACKAGES

Part	Prefix	Package Type
8051AH/ 8031AH	P D N	40-Pin Plastic DIP 40-Pin CERDIP 44-Pin PLCC
8052AH/ 8032AH	P D N	40-Pin Plastic DIP 40-Pin CERDIP 44-Pin PLCC
8751H/ 8751H-8	D	40-Pin CERDIP

PIN DESCRIPTIONS

V_{CC}: Supply voltage.

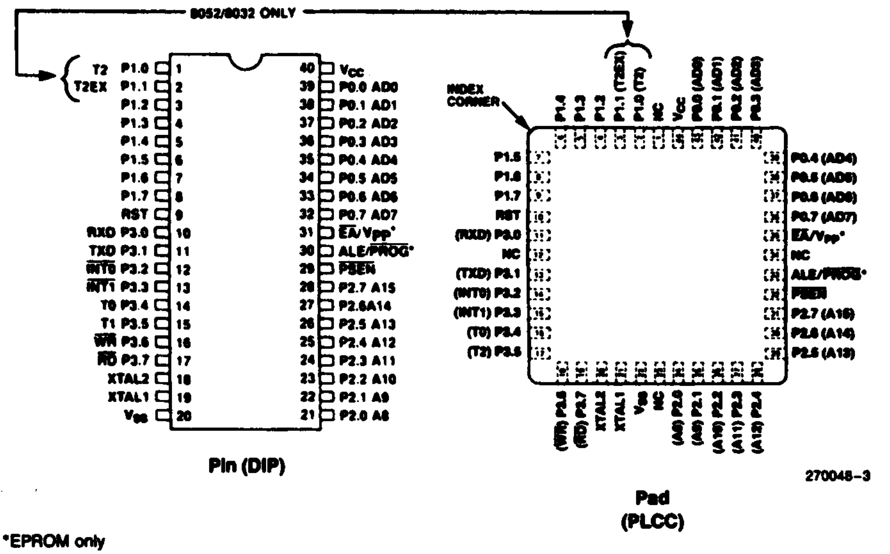
V_{SS}: Circuit ground.

Port 0: Port 0 is an 8-bit open drain bidirectional I/O port. As an output port each pin can sink 8 LS TTL inputs.

Port 0 pins that have 1s written to them float, and in that state can be used as high-impedance inputs.

Port 0 is also the multiplexed low-order address and data bus during accesses to external Program and Data Memory. In this application it uses strong internal pullups when emitting 1s and can source and sink 8 LS TTL inputs.

Port 0 also receives the code bytes during programming of the EPROM parts, and outputs the code bytes during program verification of the ROM and EPROM parts. External pullups are required during program verification.



Pin (DIP)

270048–3

Pad (PLCC)

*EPROM only

Figure 2. MCS®-51 Connections

Port 1: Port 1 is an 8-bit bidirectional I/O port with internal pullups. The Port 1 output buffers can sink/source 4 LS TTL inputs. Port 1 pins that have 1s written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 1 pins that are externally being pulled low will source current (I_{IL} on the data sheet) because of the internal pullups.

Port 1 also receives the low-order address bytes during programming of the EPROM parts and during program verification of the ROM and EPROM parts.

In the 8032AH and 8052AH, Port 1 pins P1.0 and P1.1 also serve the T2 and T2EX functions, respectively.

Port 2: Port 2 is an 8-bit bidirectional I/O port with internal pullups. The Port 2 output buffers can sink/source 4 LS TTL inputs. Port 2 pins that have 1s written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 2 pins that are externally being pulled low will source current (I_{IL} on the data sheet) because of the internal pullups.

Port 2 emits the high-order address byte during fetches from external Program Memory and during accesses to external Data Memory that use 16-bit addresses (MOVX @DPTR). In this application it uses strong internal pullups when emitting 1s. During accesses to external Data Memory that use 8-bit addresses (MOVX @Ri), Port 2 emits the contents of the P2 Special Function Register.

Port 2 also receives the high-order address bits during programming of the EPROM parts and during program verification of the ROM and EPROM parts.

Port 3: Port 3 is an 8-bit bidirectional I/O port with internal pullups. The Port 3 output buffers can sink/source 4 LS TTL inputs. Port 3 pins that have 1s written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 3 pins that are externally being pulled low will source current (I_{IL} on the data sheet) because of the pullups.

Port 3 also serves the functions of various special features of the MCS-51 Family, as listed below:

Port Pin	Alternative Function
P3.0	RXD (serial input port)
P3.1	TXD (serial output port)
P3.2	$\overline{\text{INT0}}$ (external interrupt 0)
P3.3	$\overline{\text{INT1}}$ (external interrupt 1)
P3.4	T0 (Timer 0 external input)
P3.5	T1 (Timer 1 external input)
P3.6	$\overline{\text{WR}}$ (external data memory write strobe)
P3.7	$\overline{\text{RD}}$ (external data memory read strobe)

RST: Reset input. A high on this pin for two machine cycles while the oscillator is running resets the device.

ALE/$\overline{\text{PROG}}$: Address Latch Enable output pulse for latching the low byte of the address during accesses to external memory. This pin is also the program pulse input ($\overline{\text{PROG}}$) during programming of the EPROM parts.

In normal operation ALE is emitted at a constant rate of $\frac{1}{6}$ the oscillator frequency, and may be used for external timing or clocking purposes. Note, however, that one ALE pulse is skipped during each access to external Data Memory.

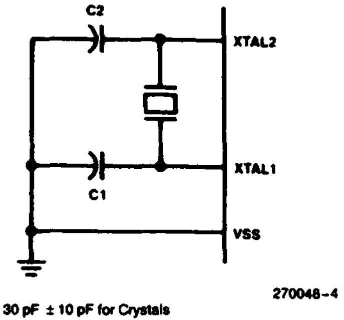
$\overline{\text{PSEN}}$: Program Store Enable is the read strobe to external Program Memory.

When the device is executing code from external Program Memory, $\overline{\text{PSEN}}$ is activated twice each machine cycle, except that two $\overline{\text{PSEN}}$ activations are skipped during each access to external Data Memory.

$\overline{\text{EA}}$/V_{PP}: External Access enable $\overline{\text{EA}}$ must be strapped to V_{SS} in order to enable any MCS-51 device to fetch code from external Program memory locations starting at 0000H up to FFFFH. $\overline{\text{EA}}$ must be strapped to V_{CC} for internal program execution.

Note, however, that if the Security Bit in the EPROM devices is programmed, the device will not fetch code from any location in external Program Memory.

This pin also receives the 21V programming supply voltage (VPP) during programming of the EPROM parts.



C1, C2 = 30 pF ± 10 pF for Crystals
= 40 pF ± 10 pF for Ceramic Resonators

Figure 3. Oscillator Connections

XTAL1: Input to the inverting oscillator amplifier.

XTAL2: Output from the inverting oscillator amplifier.

OSCILLATOR CHARACTERISTICS

XTAL1 and XTAL2 are the input and output, respectively, of an inverting amplifier which can be configured for use as an on-chip oscillator, as shown in Figure 3. Either a quartz crystal or ceramic resonator may be used. More detailed information concerning the use of the on-chip oscillator is available in Application Note AP-155, "Oscillators for Microcontrollers."

To drive the device from an external clock source, XTAL1 should be grounded, while XTAL2 is driven, as shown in Figure 4. There are no requirements on the duty cycle of the external clock signal, since the input to the internal clocking circuitry is through a divide-by-two flip-flop, but minimum and maximum high and low times specified on the Data Sheet must be observed.

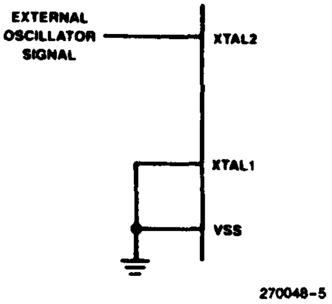


Figure 4. External Drive Configuration

DESIGN CONSIDERATIONS

If an 8751BH or 8752BH may replace an 8751H in a future design, the user should carefully compare both data sheets for DC or AC Characteristic differences. Note that the V_{IH} and I_{IH} specifications for the $\overline{\text{EA}}$ pin differ significantly between the devices.

Exposure to light when the EPROM device is in operation may cause logic errors. For this reason, it is suggested that an opaque label be placed over the window when the die is exposed to ambient light.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ...... 0°C to 70°C

Storage Temperature .... −65°C to +150°C

Voltage on $\overline{EA}$/V_{PP} Pin to V_{SS} ... −0.5V to +21.5V

Voltage on Any Other Pin to V_{SS} .... −0.5V to +7V

Power Dissipation 1.5W

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

***WARNING:** *Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

Operating Conditions: T_A (Under Bias) = 0°C to +70°C; V_{CC} = 5V ±10%; V_{SS} = 0V

D.C. CHARACTERISTICS (Under Operating Conditions)

Symbol	Parameter		Min	Max	Units	Test Conditions
V_{IL}	Input Low Voltage (Except $\overline{EA}$ Pin of 8751H & 8751H-8)		−0.5	0.8	V	
V_{IL1}	Input Low Voltage to $\overline{EA}$ Pin of 8751H & 8751H-8		0	0.7	V	
V_{IH}	Input High Voltage (Except XTAL2, RST)		2.0	V_{CC} + 0.5	V	
V_{IH1}	Input High Voltage to XTAL2, RST		2.5	V_{CC} + 0.5	V	XTAL1 = V_{SS}
V_{OL}	Output Low Voltage (Ports 1, 2, 3)*			0.45	V	I_{OL} = 1.6 mA
V_{OL1}	Output Low Voltage (Port 0, ALE, $\overline{PSEN}$)*					
		8751H, 8751H-8		0.60 0.45	V V	I_{OL} = 3.2 mA I_{OL} = 2.4 mA
		All Others		0.45	V	I_{OL} = 3.2 mA
V_{OH}	Output High Voltage (Ports 1, 2, 3, ALE, $\overline{PSEN}$)		2.4		V	I_{OH} = −80 μA
V_{OH1}	Output High Voltage (Port 0 in External Bus Mode)		2.4		V	I_{OH} = −400 μA
I_{IL}	Logical 0 Input Current (Ports 1, 2, 3, RST) 8032AH, 8052AH All Others			−800 −500	μA μA	V_{IN} = 0.45V V_{IN} = 0.45V
I_{IL1}	Logical 0 Input Current to $\overline{EA}$ Pin of 8751H & 8751H-8 Only			−15	mA	V_{IN} = 0.45V
I_{IL2}	Logical 0 Input Current (XTAL2)			−3.2	mA	V_{IN} = 0.45V
I_{LI}	Input Leakage Current (Port 0) 8751H & 8751H-8 All Others			 ±100 ±10	 μA μA	 0.45 ≤ V_{IN} ≤ V_{CC} 0.45 ≤ V_{IN} ≤ V_{CC}
I_{IH}	Logical 1 Input Current to $\overline{EA}$ Pin of 8751H & 8751H-8			500	μA	V_{IN} = 2.4V
I_{IH1}	Input Current to RST to Activate Reset			500	μA	V_{IN} < (V_{CC} − 1.5V)
I_{CC}	Power Supply Current: 8031/8051 8031AH/8051AH 8032AH/8052AH 8751H/8751H-8			 160 125 175 250	 mA mA mA mA	All Outputs Disconnected; $\overline{EA}$ = V_{CC}
C_{IO}	Pin Capacitance			10	pF	Test freq = 1 MHz

***NOTE:**

Capacitive loading on Ports 0 and 2 may cause spurious noise pulses to be superimposed on the V_{OL}s of ALE and Ports 1 and 3. The noise is due to external bus capacitance discharging into the Port 0 and Port 2 pins when these pins make 1-to-0 transitions during bus operations. In the worst cases (capacitive loading > 100 pF), the noise pulse on the ALE line may exceed 0.8V. In such cases it may be desirable to qualify ALE with a Schmitt Trigger, or use an address latch with a Schmitt

A.C. CHARACTERISTICS

Under Operating Conditions;
Load Capacitance for Port 0, ALE, and PSEN = 100 pF;
Load Capacitance for All Other Outputs = 80 pF

Symbol	Parameter	12 MHz Oscillator		Variable Oscillator		Units
		Min	Max	Min	Max	
1/TCLCL	Oscillator Frequency			3.5	12.0	MHz
TLHLL	ALE Pulse Width	127		2TCLCL − 40		ns
TAVLL	Address Valid to ALE Low	43		TCLCL − 40		ns
TLLAX	Address Hold after ALE Low	48		TCLCL − 35		ns
TLLIV	ALE Low to Valid Instr In					
	8751H		183		4TCLCL − 150	ns
	All Others		233		4TCLCL − 100	ns
TLLPL	ALE Low to $\overline{\text{PSEN}}$ Low	58		TCLCL − 25		ns
TPLPH	$\overline{\text{PSEN}}$ Pulse Width					
	8751H	190		3TCLCL − 60		ns
	All Others	215		3TCLCL − 35		ns
TPLIV	$\overline{\text{PSEN}}$ Low to Valid Instr In					
	8751H		100		3TCLCL − 150	ns
	All Others		125		3TCLCL − 125	ns
TPXIX	Input Instr Hold after $\overline{\text{PSEN}}$	0		0		ns
TPXIZ	Input Instr Float after $\overline{\text{PSEN}}$		63		TCLCL − 20	ns
TPXAV	$\overline{\text{PSEN}}$ to Address Valid	75		TCLCL − 8		ns
TAVIV	Address to Valid Instr In					
	8751H		267		5TCLCL − 150	ns
	All Others		302		5TCLCL − 115	ns
TPLAZ	$\overline{\text{PSEN}}$ Low to Address Float		20		20	ns
TRLRH	$\overline{\text{RD}}$ Pulse Width	400		6TCLCL − 100		ns
TWLWH	$\overline{\text{WR}}$ Pulse Width	400		6TCLCL − 100		ns
TRLDV	$\overline{\text{RD}}$ Low to Valid Data In		252		5TCLCL − 165	ns
TRHDX	Data Hold after $\overline{\text{RD}}$	0		0		ns
TRHDZ	Data Float after $\overline{\text{RD}}$		97		2TCLCL − 70	ns
TLLDV	ALE Low to Valid Data In		517		8TCLCL − 150	ns
TAVDV	Address to Valid Data In		585		9TCLCL − 165	ns
TLLWL	ALE Low to $\overline{\text{RD}}$ or $\overline{\text{WR}}$ Low	200	300	3TCLCL − 50	3TCLCL + 50	ns
TAVWL	Address to $\overline{\text{RD}}$ or $\overline{\text{WR}}$ Low	203		4TCLCL − 130		ns
TQVWX	Data Valid to $\overline{\text{WR}}$ Transition					
	8751H	13		TCLCL − 70		ns
	All Others	23		TCLCL − 60		ns
TQVWH	Data Valid to $\overline{\text{WR}}$ High	433		7TCLCL − 150		ns
TWHQX	Data Hold after $\overline{\text{WR}}$	33		TCLCL − 50		ns
TRLAZ	$\overline{\text{RD}}$ Low to Address Float		20		20	ns
TWHLH	$\overline{\text{RD}}$ or $\overline{\text{WR}}$ High to ALE High					
	8751H	33	133	TCLCL − 50	TCLCL + 50	ns
	All Others	43	123	TCLCL − 40	TCLCL + 40	ns

NOTE:
*This table does not include the 8751-8 A.C. characteristics (see next page).

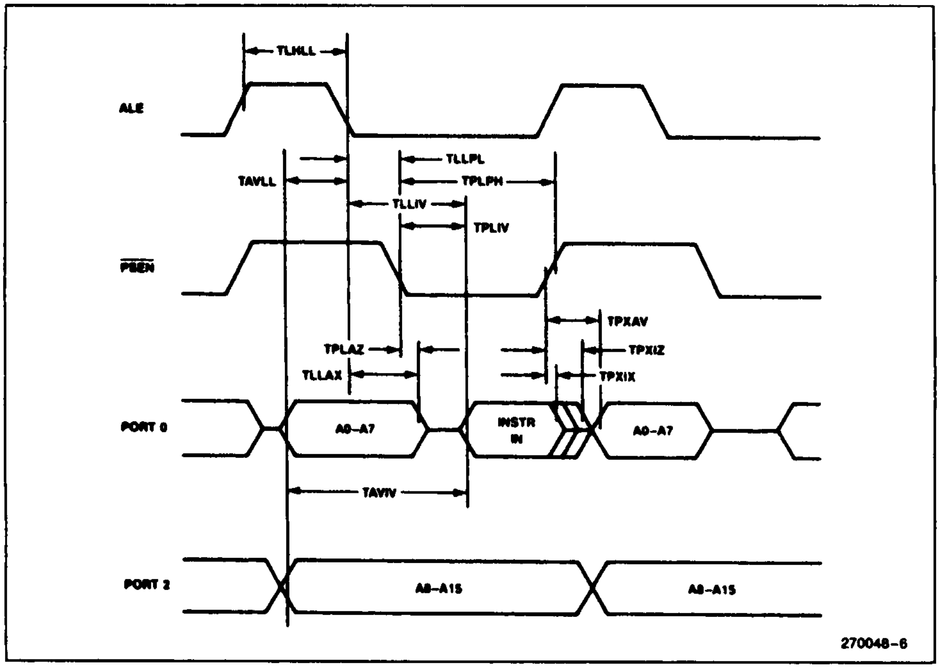
This Table is only for the 8751H-8

A.C. CHARACTERISTICS

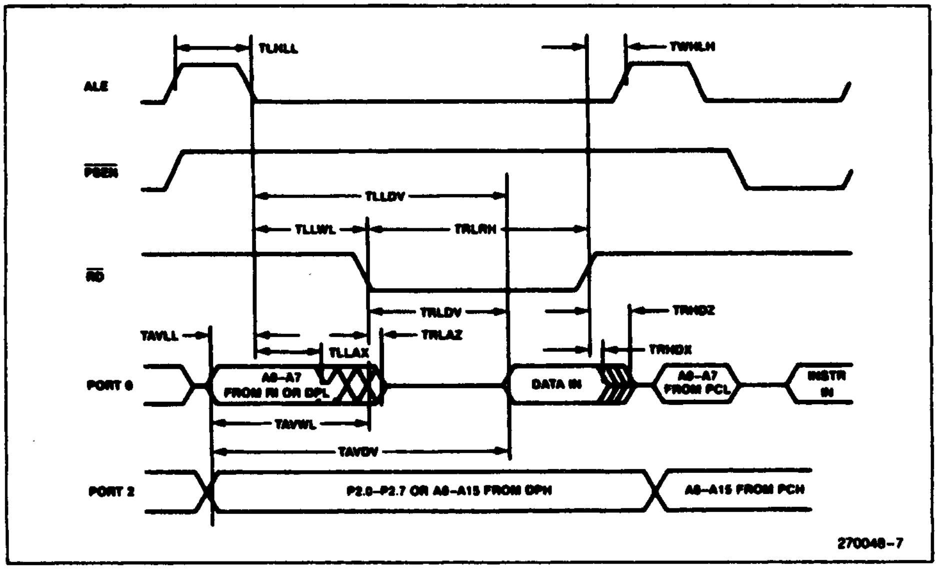
Under Operating Conditions;
Load Capacitance for Port 0, ALE, and PSEN = 100 pF;
Load Capacitance for All Other Outputs = 80 pF

Symbol	Parameter	8 MHz Oscillator		Variable Oscillator		Units
		Min	Max	Min	Max	
1/TCLCL	Oscillator Frequency			3.5	8.0	MHz
TLHLL	ALE Pulse Width	210		2TCLCL − 40		ns
TAVLL	Address Valid to ALE Low	85		TCLCL − 40		ns
TLLAX	Address Hold after ALE Low	90		TCLCL − 35		ns
TLLIV	ALE Low to Valid Instr In		350		4TCLCL − 150	ns
TLLPL	ALE Low to $\overline{\text{PSEN}}$ Low	100		TCLCL − 25		ns
TPLPH	$\overline{\text{PSEN}}$ Pulse Width	315		3TCLCL − 60		ns
TPLIV	$\overline{\text{PSEN}}$ Low to Valid Instr In		225		3TCLCL − 150	ns
TPXIX	Input Instr Hold after $\overline{\text{PSEN}}$	0		0		ns
TPXIZ	Input Instr Float after $\overline{\text{PSEN}}$		105		TCLCL − 20	ns
TPXAV	$\overline{\text{PSEN}}$ to Address Valid	117		TCLCL − 8		ns
TAVIV	Address to Valid Instr In		475		5TCLCL − 150	ns
TPLAZ	$\overline{\text{PSEN}}$ Low to Address Float		20		20	ns
TRLRH	$\overline{\text{RD}}$ Pulse Width	650		6TCLCL − 100		ns
TWLWH	$\overline{\text{WR}}$ Pulse Width	650		6TCLCL − 100		ns
TRLDV	$\overline{\text{RD}}$ Low to Valid Data In		460		5TCLCL − 165	ns
TRHDX	Data Hold after $\overline{\text{RD}}$	0		0		ns
TRHDZ	Data Float after $\overline{\text{RD}}$		180		2TCLCL − 70	ns
TLLDV	ALE Low to Valid Data In		850		8TCLCL − 150	ns
TAVDV	Address to Valid Data In		960		9TCLCL − 165	ns
TLLWL	ALE Low to $\overline{\text{RD}}$ or $\overline{\text{WR}}$ Low	325	425	3TCLCL − 50	3TCLCL + 50	ns
TAVWL	Address to $\overline{\text{RD}}$ or $\overline{\text{WR}}$ Low	370		4TCLCL − 130		ns
TQVWX	Data Valid to $\overline{\text{WR}}$ Transition	55		TCLCL − 70		ns
TQVWH	Data Valid to $\overline{\text{WR}}$ High	725		7TCLCL − 150		ns
TWHQX	Data Hold after $\overline{\text{WR}}$	75		TCLCL − 50		ns
TRLAZ	$\overline{\text{RD}}$ Low to Address Float		20		20	ns
TWHLH	$\overline{\text{RD}}$ or $\overline{\text{WR}}$ High to ALE High	75	175	TCLCL − 50	TCLCL + 50	ns

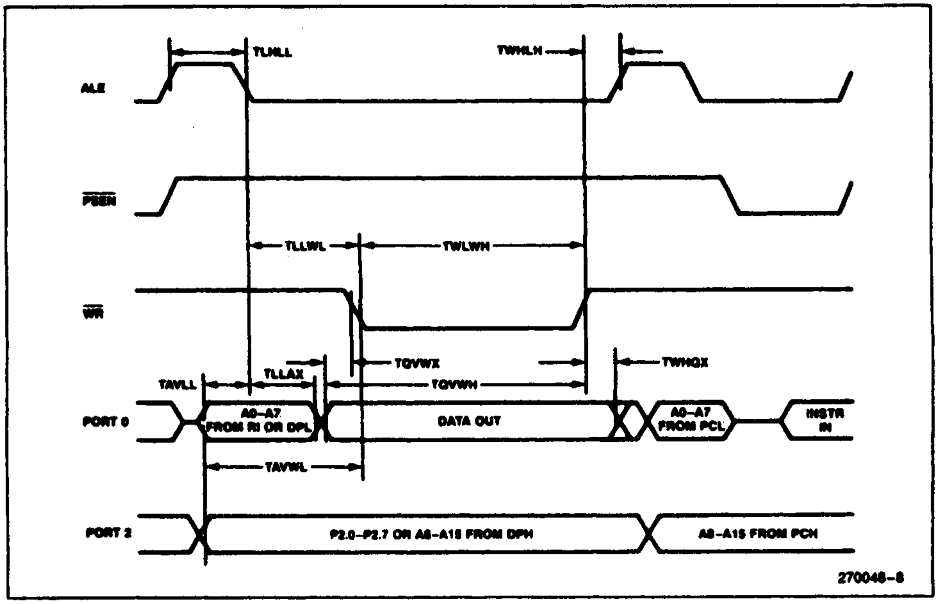
EXTERNAL PROGRAM MEMORY READ CYCLE



EXTERNAL DATA MEMORY READ CYCLE



EXTERNAL DATA MEMORY WRITE CYCLE

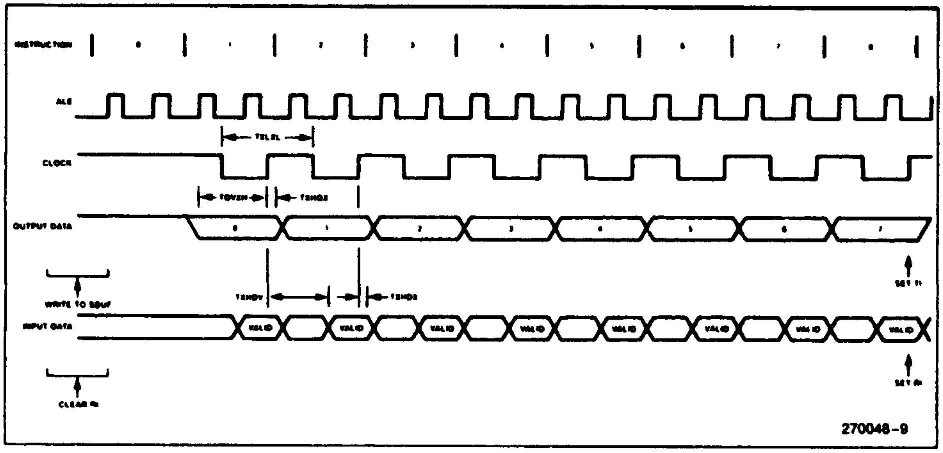


SERIAL PORT TIMING—SHIFT REGISTER MODE

Test Conditions: T_A = 0°C to 70°C; VCC = 5V ±10%; VSS = 0V; Load Capacitance = 80 pF

Symbol	Parameter	12 MHz Oscillator		Variable Oscillator		Units
		Min	Max	Min	Max	
TXLXL	Serial Port Clock Cycle Time	1.0		12TCLCL		µs
TQVXH	Output Data Setup to Clock Rising Edge	700		10TCLCL – 133		ns
TXHQX	Output Data Hold after Clock Rising Edge	50		2TCLCL – 117		ns
TXHDX	Input Data Hold after Clock Rising Edge	0		0		ns
TXHDV	Clock Rising Edge to Input Data Valid		700		10TCLCL – 133	ns

SHIFT REGISTER TIMING WAVEFORMS

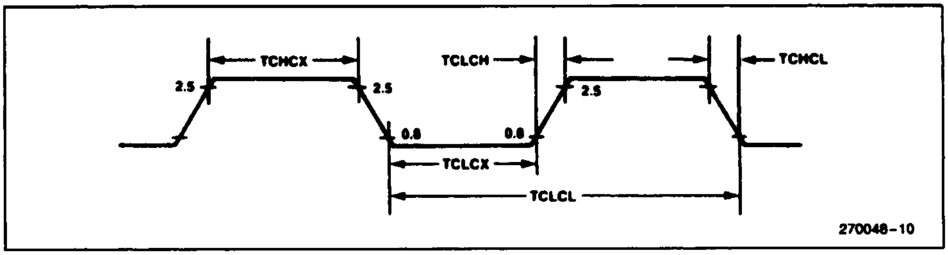


270048-9

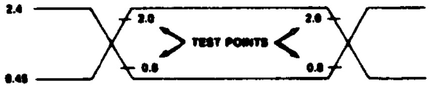
EXTERNAL CLOCK DRIVE

Symbol	Parameter	Min	Max	Units
1/TCLCL	Oscillator Frequency (except 8751H-8) 8751H-8	3.5 3.5	12 8	MHz MHz
TCHCX	High Time	20		ns
TCLCX	Low Time	20		ns
TCLCH	Rise Time		20	ns
TCHCL	Fall Time		20	ns

EXTERNAL CLOCK DRIVE WAVEFORM



A.C. TESTING INPUT, OUTPUT WAVEFORM



270048-11

A.C. Testing: Inputs are driven at 2.4V for a Logic "1" and 0.45V for a Logic "0". Timing measurements are made at 2.0V for a Logic "1" and 0.8V for a Logic "0".

EPROM CHARACTERISTICS

Table 3. EPROM Programming Modes

Mode	RST	$\overline{\text{PSEN}}$	ALE	$\overline{\text{EA}}$	P2.7	P2.6	P2.5	P2.4
Program	1	0	0*	VPP	1	0	X	X
Inhibit	1	0	1	X	1	0	X	X
Verify	1	0	1	1	0	0	X	X
Security Set	1	0	0*	VPP	1	1	X	X

NOTE:
"1" = logic high for that pin
"0" = logic low for that pin
"X" = "don't care"
"VPP" = +21V ±0.5V
*ALE is pulsed low for 50 ms.

Programming the EPROM

To be programmed, the part must be running with a 4 to 6 MHz oscillator. (The reason the oscillator needs to be running is that the internal bus is being used to transfer address and program data to appropriate internal registers.) The address of an EPROM location to be programmed is applied to Port 1 and pins P2.0–P2.3 of Port 2, while the code byte to be programmed into that location is applied to Port 0. The other Port 2 pins, and RST, $\overline{\text{PSEN}}$, and $\overline{\text{EA}}$ should be held at the "Program" levels indicated in Table 3. ALE is pulsed low for 50 ms to program the code byte into the addressed EPROM location. The setup is shown in Figure 5.

Normally $\overline{\text{EA}}$ is held at a logic high until just before ALE is to be pulsed. Then $\overline{\text{EA}}$ is raised to +21V, ALE is pulsed, and then $\overline{\text{EA}}$ is returned to a logic high. Waveforms and detailed timing specifications are shown in later sections of this data sheet.

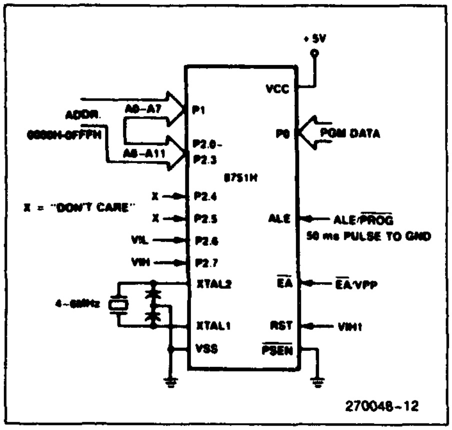


Figure 5. Programming Configuration

Note that the $\overline{\text{EA}}$/VPP pin must not be allowed to go above the maximum specified VPP level of 21.5V for any amount of time. Even a narrow glitch above that voltage level can cause permanent damage to the device. The VPP source should be well regulated and free of glitches.

Program Verification

If the Security Bit has not been programmed, the on-chip Program Memory can be read out for verification purposes, if desired, either during or after the programming operation. The address of the Program Memory location to be read is applied to Port 1 and pins P2.0–P2.3. The other pins should be held at the "Verify" levels indicated in Table 3. The contents of the addressed location will come out on Port 0. External pullups are required on Port 0 for this operation.

The setup, which is shown in Figure 6, is the same as for programming the EPROM except that pin P2.7 is held at a logic low, or may be used as an active-low read strobe.

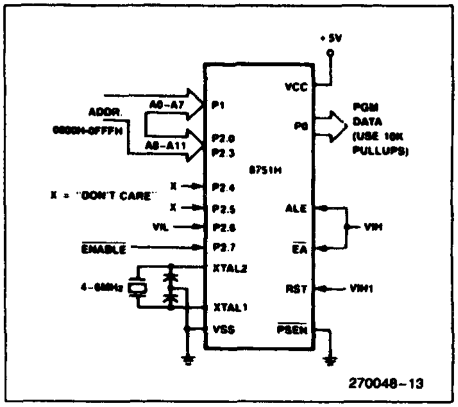


Figure 6. Program Verification

EPROM Security

The security feature consists of a "locking" bit which when programmed denies electrical access by any external means to the on-chip Program Memory. The bit is programmed as shown in Figure 7. The setup and procedure are the same as for normal EPROM programming, except that P2.6 is held at a logic high. Port 0, Port 1, and pins P2.0–P2.3 may be in any state. The other pins should be held at the "Security" levels indicated in Table 3.

Once the Security Bit has been programmed, it can be cleared only by full erasure of the Program Memory. While it is programmed, the internal Program Memory can not be read out, the device can not be further programmed, and it **can not execute out of external program memory**. Erasing the EPROM, thus clearing the Security Bit, restores the device's full functionality. It can then be reprogrammed.

Erasure Characteristics

Erasure of the EPROM begins to occur when the chip is exposed to light with wavelengths shorter than approximately 4,000 Angstroms. Since sunlight and fluorescent lighting have wavelengths in this range, exposure to these light sources over an extended time (about 1 week in sunlight, or 3 years in room-level fluorescent lighting) could cause inadvertent erasure. If an application subjects the device to this type of exposure, it is suggested that an opaque label be placed over the window.

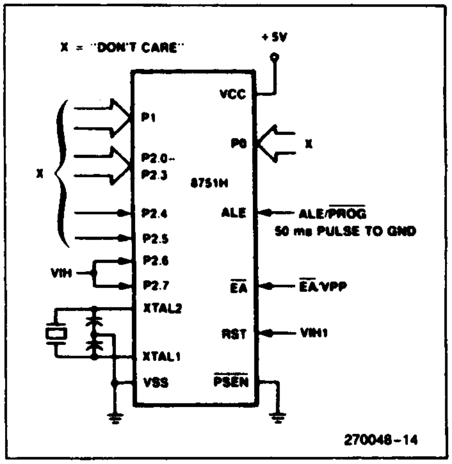


Figure 7. Programming the Security Bit

The recommended erasure procedure is exposure to ultraviolet light (at 2537 Angstroms) to an integrated dose of at least 15 W-sec/cm^2. Exposing the EPROM to an ultraviolet lamp of 12,000 μW/cm^2 rating for 20 to 30 minutes, at a distance of about 1 inch, should be sufficient.

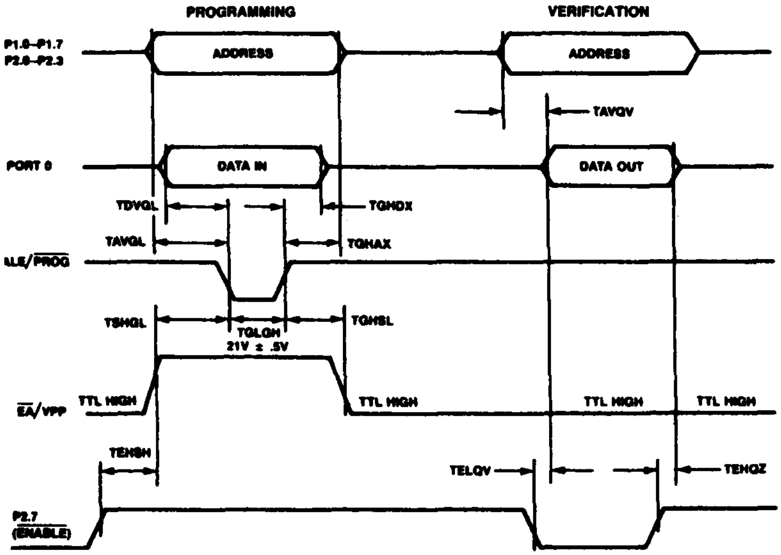
Erasure leaves the array in an all 1s state.

EPROM PROGRAMMING AND VERIFICATION CHARACTERISTICS

T_A = 21°C to 27°C; VCC = 5V ±10%; VSS = 0V

Symbol	Parameter	Min	Max	Units
VPP	Programming Supply Voltage	20.5	21.5	V
IPP	Programming Supply Current		30	mA
1/TCLCL	Oscillator Frequency	4	6	MHz
TAVGL	Address Setup to $\overline{\text{PROG}}$ Low	48TCLCL		
TGHAX	Address Hold after $\overline{\text{PROG}}$	48TCLCL		
TDVGL	Data Setup to $\overline{\text{PROG}}$ Low	48TCLCL		
TGHDX	Data Hold after $\overline{\text{PROG}}$	48TCLCL		
TEHSH	P2.7 ($\overline{\text{ENABLE}}$) High to VPP	48TCLCL		
TSHGL	VPP Setup to $\overline{\text{PROG}}$ Low	10		μs
TGHSL	VPP Hold after $\overline{\text{PROG}}$	10		μs
TGLGH	$\overline{\text{PROG}}$ Width	45	55	ms
TAVQV	Address to Data Valid		48TCLCL	
TELQV	$\overline{\text{ENABLE}}$ Low to Data Valid		48TCLCL	
TEHQZ	Data Float after $\overline{\text{ENABLE}}$	0	48TCLCL	

EPROM PROGRAMMING AND VERIFICATION WAVEFORMS



270048-15

For programming conditions see Figure 5.

For verification conditions see Figure 6.

DATA SHEET REVISION HISTORY

The following are the key differences between this and the -004 version of this data sheet.

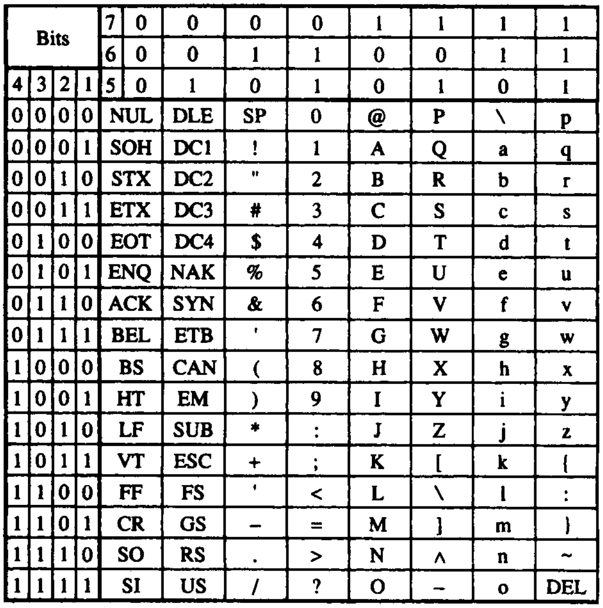
1. Data sheet status changed from "Preliminary" to "Production".
2. LCC package offering deleted.
3. Maximum Ratings Warning and Data Sheet Revision History revised.

The following are the key differences between this and the -003 version of this data sheet:

1. Introduction was expanded to include product descriptions.
2. Package table was added.
3. Design Considerations added.
4. Test Conditions for I_{IL1} and I_{IH} specifications added to the DC Characteristics.
5. Data Sheet Revision History added.

F

ASCII Code Chart



Bits				7	0	0	0	0	1	1	1	1
				6	0	0	1	1	0	0	1	1
4	3	2	1	5	0	1	0	1	0	1	0	1
0	0	0	0		NUL	DLE	SP	0	@	P	\	p
0	0	0	1		SOH	DC1	!	1	A	Q	a	q
0	0	1	0		STX	DC2	"	2	B	R	b	r
0	0	1	1		ETX	DC3	#	3	C	S	c	s
0	1	0	0		EOT	DC4	$	4	D	T	d	t
0	1	0	1		ENQ	NAK	%	5	E	U	e	u
0	1	1	0		ACK	SYN	&	6	F	V	f	v
0	1	1	1		BEL	ETB	'	7	G	W	g	w
1	0	0	0		BS	CAN	(	8	H	X	h	x
1	0	0	1		HT	EM	)	9	I	Y	i	y
1	0	1	0		LF	SUB	*	:	J	Z	j	z
1	0	1	1		VT	ESC	+	;	K	[	k	{
1	1	0	0		FF	FS	'	<	L	\	l	:
1	1	0	1		CR	GS	–	=	M	]	m	}
1	1	1	0		SO	RS	.	>	N	^	n	~
1	1	1	1		SI	US	/	?	O	–	o	DEL

FIGURE F-1
ASC1I code chart

G

MON51—An 8051 Monitor Program

This appendix contains the listing for an 8051 monitor program (MON51) along with a general description of its design and operation. Many of the concepts in assembly language programming developed earlier in short examples (see Chapter 7) can be reinforced by reviewing this appendix. The source and listing files for MON51 are contained on the diskette accompanying this text.

MON51 is a monitor program written for the 8051 microcontroller and, more specifically, for the SBC-51 single-board computer (see Chapter 11). We begin with a description of the purpose of a monitor program and then give a summary of MON51 commands. The overall operation of MON51 and some design details are also described. The final pages of this appendix contain listings of each assembled source file, the listing created by RL51, and a dump of MON51 in Intel hexadecimal format.

A monitor program is not an operating system. It is a small program with commands that provide a primitive level of system operation and user interaction. The major difference is that operating systems are found on larger computers with disk drives, while monitor programs are found on small systems, such as the SBC-51, with a keyboard (or keypad) for input and a CRT (or LEDs) for output. Some single-board computers also provide mass storage using an audio cassette interface.

MON51 is an 8051 assembly language program approximately two kilobytes in length. It was written on an Intel iPDS100 development system using an editor (CREDIT), a cross assembler (ASM51), and a linker/locator (RL51). Testing and debugging were performed using a hardware emulator (EMV51) connected to an Intel SDK-51TM single-board computer. It is worth mentioning that hardware emulators are such powerful tools that the first version of MON51 burned into EPROM was essentially "bug-free."

MON51 was developed using modular programming techniques. Ten source files were used, including nine program files and one macro definition file. To keep the listing files relatively short for this appendix, the "create symbol table" option was switched off in each source file using the $NOSYMBOLS assembler control. (See MAIN.LST, line 4 on p. 466.) However, since the $DEBUG assembler control was also used, the listing file created by the linker/locator, RL51, contains a complete, absolute symbol table for the entire program. The linker/locator absolute output was placed in V12 (for "version 12") and the listing file was placed in V12.M51. The OH (object-to-hex) utility was used to convert the absolute output to an Intel hex file, V12.HEX, suitable for printing, downloading to a target system, or burning into EPROM. (Spaces have been inserted into the hex file to improve readability.)

The files appearing in this appendix are identified below.

File	**Page**
ASM51 Listings	
MAIN.LST	466
GETPAR.LST	470
IO.LST	473
CONVRT.LST	478
LOAD.LST	479
DUMP.LST	481
SFR.LST	484
IS.LST	488
SET.LST	489
Macro Definitions	
MACROS.SRC	493
RL51 Listing	
V12.M51	494
Hex File	
V12.HEX	496

COMMANDS AND SPECIFICATIONS

MON51 uses the 8051's on-chip serial port for input/output. It requires an RS232C VDT operating at 2400 baud with seven data bits, one stop bit, and odd parity. Command line editing is possible using the BACKSPACE or DELETE keys to back up and correct mistakes. VT100 escape sequences are used (see below).

The following four commands are supported:

DUMP	dump the contents of a range of memory locations to the console
SET	examine single memory locations and set their contents to a new value
GO	go to user program at a specified address
LOAD	load an Intel hex-formatted file

COMMAND FORMAT

Commands are entered using the following format:

<c1><c2><p1>,<p2>,<p3>,<p4><CR>

where <c1>and<c2>form a 2-character command

<p1>to<p4>are command parameters

<CR>is the RETURN key

The following general comments describe command entry on MON51:
When the system is powered up or reset, the "V12>" prompt appears on the VDT.

- Commands are entered on the keyboard and followed by pressing the RETURN key.
- During command entry, mistakes may be corrected using the BACKSPACE or DELETE key.
- Maximum line length is 20 characters.
- CONTROL-C terminates a command at any time and returns control to the prompt.
- An empty command line (RETURN key only) defaults to the SET command.
- Each command uses from 0 to 4 parameters entered in hexadecimal. *(Note:* The "H" is not needed.)
- Parameters are separated by commas.
- Any parameter may be omitted by entering two commas.
- Parameters omitted will default to their previous value.
- CONTROL-P toggles an enable bit for printer output. *(Note:* If printer output is enabled, the printer must be connected and on-line; otherwise MON51 waits for the printer and no output is sent to the console.)

COMMON SYNTAX

Dump

Format: D<m><start>,<end><CR>

Where: D is the command identifier

<m> is the memory space selector as follows:

B—bit-addressable memory

C—code memory

I—internal data memory

R—special function registers

X-extemal data memory

<start> is the start address

<end> is the end address

<CR> is the RETURN key

(Note: Output to the console may be suspended by entering CONTROL-S (XOFF). CONTROL-Q (XON) resumes output.)

Example: DX8000,8100<CR>dumps the contents of 257 external data locations from addresses 8000H to 8100H

Set

Format: S<m><address><CR> followed by one of

<value>

<SP>

—

<CR>

Q

Where: S is the command identifier

<m> is the memory space selector as follows:

B—bit-addressable memory

C—code memory

I—internal data memory

R—special function registers

X—external data memory

<addressees the address of a memory location to examine and/or set to a value

<value> is a data value to write into memory

<SP> is the space bar used to write same value into the next location—examine previous location

<CR> is the return key used to examine the next location

Q quit and return to the MON51 prompt

Example: SI90<CR>followed by A5<CR>sets Port 1 to A5H

Go

Format: GO<address><CR>

Where: GO is the command identifier

<address> is the address in code memory to begin

execution <CR> is the RETURN key

Example: GO8000<CR> loads 8000H into the program counter

Lo

Format: LO<CR>

Where: LO is the command identifier

<CR>is the RETURN key

Example: LO<CR> The message "<host download to SBC-51>" appears. MON51 receives an Intel hex-formatted file at the serial port and writes the bytes received into external data memory. When the transfer is complete, the message "EOF—file download OK" appears. *Note:* Use the DUMP command to verify that the transfer was successful.

GENERAL OPERATION OF MON51

The general operation of MON51 is described by the following steps:

1. Initialize registers and memory locations.
2. Send prompt to VDT display.
3. Get a command line from VDT keyboard.
4. Decode command.
5. Execute command
6. Go to step 2.

Although numerous subroutines and program modules participate in the overall operation of MON51, the basic framework to implement the above steps is found in MAIN.LST. The instructions to initialize registers and memory locations (step 1) are lines 171 to 200 (see pp. 467-68). The prompt is sent to the VDT display in lines 201 and 202, and then a command is inputted from the VDT keyboard in lines 203 and 204 (steps 3 and 4). The command is decoded in lines 205 to 224. MON51 is designed to support 26 commands, one for each letter in the alphabet. Only four are implemented, however, while the others are for future expansion or custom applications. Decoding the command entails (a) using the first character in the input line buffer to look up the command's address from the table in lines 227 to 252, (b) placing the command's address on the stack (1 byte at a time), and (c) popping the address into the program counter using the RET instruction (line 225).

The code for the DUMP, SET, and LOAD commands is contained in separate files, while the code for the GO command is found in MAIN.LST in lines 268 to 274. Numerous subroutines are used throughout MON51 and can be found in the various listings provided.

During command decoding, the subroutine GETPAR is called (line 224). This subroutine gets parameters from the input line (which are stored in the line buffer as ASCII characters) and converts them to binary, placing the results in internal RAM at four 16-bit locations starting at the label PARMTR (see GETPAR.LST, line 58 on p. 471). These parameters are needed by the commands DUMP, SET, and GO.

After each command is executed, control is passed back to MON51 at the label GETCMD (line 200 in MAIN.LST) and the above steps are repeated. User programs may terminate, using the address of GETCMD (00BCH) as the destination of an LJMP instruction.

MON51 contains numerous comments, and each subroutine and file contains a comment block explaining the general operation of the section of code that follows. Readers interested in understanding the intricacies of MON51 operation are directed to the listing files and comment lines for further details. In the following section, the overall design of MON51 is discussed.

THE DESIGN OF MON51

Developing an understanding of the "design" (as opposed to the operation) is also important, since MON51 incorporates—on a relatively large scale—many of the concepts developed earlier in brief examples. The following paragraphs elaborate on key design features that can be adopted in developing software for 8051-based designs. In a sense, then, the present appendix is like a case study—describing the design details of an existing 8051 software product.

Assembler controls, as mentioned in Chapter 7, are used primarily to control the format and content of the listing files created by ASM51 and RL51. At the top of each of the nine program listings, several assembler controls appear. These were chosen primarily to produce listing files suitable for reproduction in this appendix. Note in MAIN.LST, for example, that the control $NOLIST is used in line 6. This was used because line 7 (not listed) contained the control $INCLUDE(MACROS.SRC), which directed ASM51 to the file containing the macro definitions (pp. 493-94). Turning the "list" option off and on just before and after the $INCLUDE statement prevented ASM51 from needlessly putting the macro definitions into MAIN.LST.

Many of the assembler directives supported by ASM51 appear throughout the MON51 listing files. Recall (Section 7.5 Assembler Directives) that ASM51 supports five categories of directives:

- Assembler state control
- Symbol definition
- Storage initialization/reservation
- Program linkage
- Segment selection

Examples from each of these categories appear in the listings.

Assembler State Control. The only assembler state control directives used in MON51 are END and USING. The ORG directive was never used, since the code and data segments were designed to be relocatable with addresses established only at link-time.[1] An instance of USING can be found in GETPAR.LST, line 108 on p. 472.

Symbol Definitions. Many symbols are defined throughout MON51, as, for example, in MAIN.LST, lines 115 to 127 (pp. 466-67). Note in particular the definitions for EPROM, ONCHIP, and BITRAM. EPROM is defined as the name for a code segment. Similarly, ONCHEP is defined as the data segment and BITRAM is defined as the bit segment. All three are, by definition, relocatable. Recall that the segment type "DATA" corresponds to the 8051 on-chip data space accessible by direct addressing (00H to 7FH). All instructions in MON51 go in the EPROM code segment, all on-chip data locations are defined in the ONCHIP data segment, and all bit locations are defined in the BITRAM segment.

Storage Initialization/Reservation. The define byte directive (DB) is used throughout MON51 to initialize code memory with byte constants. Usually, the definition is for ASCII character strings sent to the console as part of MON51 's normal operation. For example, the prompt characters for MON51 are defined in MAIN.LST on lines 288 and 289.

[1] The only exception to this is the absolute definition of the stack using the DSEG directive (see p. 469, line 288).

Other definitions include the escape sequence sent to the console to back up the cursor. MON51 is designed to work with VT100 compatible terminals. If the BACKSPACE or DELETE key is pressed while you are entering a line, the escape sequence <ESC>0 [(or hexadecimal bytes 1BH, 5BH, and 44H)] must be sent to the terminal. These bytes are defined as a null-terminated ASCII string (see IO.LST, line 144 on p. 475) using the DB directive. Once the code for the BACKSPACE or DELETE key is detected (see lines 17 and 18 on p. 473 and lines 133 and 135 on p. 475), the pointer to the input line buffer is decremented twice, and the "back-up cursor" escape sequence is sent to the console, using the output string (OUT-SIR) subroutine. (See lines 136 to 139 on p. 475.)

Storage locations are reserved using either the define storage (DS) or define bit (DBIT) directives. The first instance of a DS directive defines a 24-byte stack (see line 288 in MAIN.LST) in an absolute data segment starting at address 08H in internal RAM. MON51 does not use register banks 1 to 3, so internal RAM addresses 08H to 1FH have been reserved for the stack. The default value of 07H for the stack pointer ensures that the first write to the stack is at address 08H. A 20-byte buffer for the command line is reserved at line 296 in MAIN.LST. Several bit locations are reserved in the BITRAM bit segment at the end of MAIN.LST.

Program Linkage. The PUBLIC and EXTRN directives are used near the beginning of most source files. PUBLIC declares which symbols defined in a particular file are to be made available for use in other source files. EXTRN declares which symbols are used in the current source file but are defined in another source file. The NAME directive is not used in MON51; therefore, each file is a module with the file name serving as the module name.

Several of the files for MON51 hold the source code for subroutines used elsewhere. For example, IO.LST (pp. 473-78) contains the source code for the input/output subroutines INCHAR, OUTCHR, INLINE, OUTSTR, OUTHEX, and OUT2HX. These subroutines are all declared as "public" in line 26 (p. 473). Other files that use these subroutines must contain a statement declaring these symbols as external code address symbols (e.g., MAIN,LST, line 108 on p. 466). Where these subroutines are called, the address for the LCALL or ACALL instruction is unknown by ASM51, so "F" appears in the listing file to indicate that linking/locating is required to "fix" the instruction. For example, in MAIN,LST the prompt is sent to the console in line 202, using ACALL OUTSTR. This instruction appears in the listing as 1100H with an "F' beside it. The linker/locator, RL51, will determine the correct absolute address of the OUTSTR subroutines and fix the instruction. In this example, since the addressing mode is absolute within the current 2K page, the fix substitutes 11 bits into the instruction—three bits in the upper byte and all eight bits in the lower byte.

Segment Selection. The segment selection directives are RSEG to select a relocatable segment, and the five directives to select an absolute segment of a specified memory space (DSEG, BSEG, XSEG, CSEG, and ISEG). The RSEG is used at least once in each file to begin the EPROM code segment (e.g., MAIN,LST, line 129, p. 467). Other instances of RSEG appear as necessary to select the ONCHIP data segment (e.g., MAIN,LST, line 295) or the BITRAM bit segment (e.g., MAIN,LST, line 306, p. 470). The only instance of an absolute segment is the definition of the stack at absolute internal RAM address 08H (MAIN,LST, line 287, p. 469).

OPTION JUMPERS

The SBC-51 has three jumpers on Port 1 that are read by software to evoke special features. MON51 reads these jumpers after a system reset and sets or clears corresponding bit locations (see MAIN.LST lines 171 to 176). Each jumper has a special purpose as outlined below.

Jumper	Installed	Purpose
X3	NO	normal execution
	YES	jump to 2000H upon reset
X4	NO	interrupt jump table at 80xxH
	YES	interrupt jump table at 20xxH
X13	NO	normal execution
	YES	software UART (see below)

Caution should be exercised when you are interfacing I/O circuitry to Port 1. If the attached interface presents a logic 0 to the corresponding pin during a system reset, this is interpreted by MON51 as if the jumper were installed, and the option listed above takes effect.

Reset Entry Point. Jumper X3 can be installed to force a jump to 2000H immediately after a system reset (see MAIN.LST, line 199). This is useful in order to have both the MON51 EPROM and a user EPROM installed simultaneously and to select which executes upon reset. If, for example, the user program does not use a VDT, then it is inconvenient to power up in "MON51 mode" just to enter GO2000H to evoke the user program. The installation of jumper X3 avoids the need for this.

Interrupt Jump Tables. Although MON51 does not use interrupts, user applications that coexist with MON51 can adopt an interrupt-driven design. Since MON51 resides in EPROM starting at address 0000H *and* since all interrupts vector to locations near the bottom of memory, a jump table was devised to allow user applications executing at other addresses to employ interrupts. Based on the default hardware configuration of the SBC-51, user applications generally execute either in RAM starting at 8000H or in EPROM starting at 2000H. If a user application enables interrupts and subsequently an interrupt occurs and is accepted, the interrupt vector address (i.e., entry point) is one of 0003H, 000BH, and 0013H, etc., depending on the source of the interrupt (see Table 6-4). At each interrupt vector address in MON51, there is a short instruction sequence that jumps to a location either in the user EPROM at 20xxH or in RAM at 80xxH, depending on whether or not jumper X4 is installed on the SBC-51 (see MAIN.LST, lines 150 to 183). The correct entry point for each interrupt is listed below.

INTERRUPT ENTRY POINT

Interrupt Source	X4 Installed	X4 Not Installed
(default entry point)	2000H	8000H
External 0	2003H	8003H
Timer 0	2006H	8006H
External 1	2009H	8009H
Timer 1	200CH	800CH
Serial Port	200FH	800FH
Timer 2	2012H	8012H

The entry points above are spaced three locations apart, leaving just enough room for an LJMP instruction to the interrupt service routine. If operation is from a 12 MHz crystal, 6 μs is added to the interrupt latency because of the overhead of this scheme (due to the JNB and LJMP instructions in MON51 and the LJMP instruction in the user program). Of course, if user applications do not use interrupts, programs may begin at 2000H or 8000H and proceed through the above entry points.

For an example of a user application using interrupts that coexists with MON51, see the MC14499 interface project in Chapter 11.

Software UART. When jumper X13 is installed, MON51 implements a software UART for serial input/output instead of using the 8051's on-chip serial port. This feature is useful for developing interfaces to serial devices other than the default terminal. Note that the baud rate is 1200 if the software UART is used. The operation of the software UART is described in IO.LST (p. 477, 11.219-278).

THE LINK MAP AND SYMBOL TABLE

One of the most important listings is that produced by the linker/locator, RL51. This listing, V12.M51 (pp. 494-96), contains a link map and symbol table. The link map shows the starting address and length of the three relocatable segments used by MON51 (p. 495). The addresses are absolute at this stage, since the segments have been located by RL51 based on the specifications provided in the invocation line (p. 494). The symbol table gives the absolute values assigned to all relocatable symbols (as assigned by RL51 at link-time). It is partitioned alphabetically by module (i.e., file) in the order linked.

As an example, the absolute address of the OUTSTR subroutine is determined by looking up OUTSTR in the IO module (p. 495-96). The address of OUTSTR in MON51 version 12 is 0282H.

INTEL HEX FILE

The final listing is that produced by the OH (object-to-hex) conversion utility. The listing V12.HEX (pp. 496-98) contains the machine-language bytes of the MON51 program in Intel hexadecimal format. For example, the OUTSTR subroutine starts at address 0282H. On p. 497, the first three bytes of this subroutine appear as E4H, 93H, and 60H. Referring directly to the OUTSTR subroutine in the IO.LST confirms that these values are correct (see p. 476, 11.156-158).

MAIN MODULE FOR 8051 MONITOR PROGRAM

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN MAIN.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE MAIN.SRC EP

LOC  OBJ            LINE     SOURCE
                       1     $debug
                       2     $title(*** MAIN MODULE FOR 8051 MONITOR PROGRAM ***)
                       3     $pagewidth(98)
                       4     $nosymbols              ;symbol table in .M51 file created by RL51
                       5     $nopaging
                       6     $nolist                 ;next line contains $include(macros.src)
                      76     ;previous line contains $list
                      77     ;*********************************************************************
                      78     ;                                                                    *
                      79     ;                               M O N 5 1                            *
                      80     ;                                                                    *
                      81     ;                        (An 8051 Monitor Program)                   *
                      82     ;                                                                    *
                      83     ; Copyright (c) I. Scott MacKenzie, 1988, 1991                       *
                      84     ; Dept. of Computing & Information Science                           *
                      85     ; University of Guelph                                               *
                      86     ; Guelph, Ontariao                                                   *
                      87     ; Canada NIG 2WI                                                     *
                      88     ;                                                                    *
                      89     ; VERSION 12 - April 1991                                            *
                      90     ;                                                                    *
                      91     ; COMMANDS: D - dump memory to console                               *
                      92     ;           G - go to user program                                   *
                      93     ;           L - load hex file                                        *
                      94     ;           S - set memory to value                                  *
                      95     ;                                                                    *
                      96     ; RESET OPTIONS:                                                     *
                      97     ;                                                                    *
                      98     ; JUMPER  INSTALLED  RESULT                                          *
                      99     ; ------  ---------  -----------------------------------------       *
                     100     ;   X3       NO      Execute MON51                                   *
                     101     ;   X3       YES     Jump to 2000H upon reset                        *
                     102     ;   X4       NO      Interrupt jump table in user RAM (8000H)        *
                     103     ;   X4       YES     Interrupt jump table in user EPROM (2000H)      *
                     104     ;   X13      NO      Serial I/O using P3.1 (TXD) and P3.0 (RXD)      *
                     105     ;   X13      YES     Serial I/O using P1.7 (TXD) and P1.6 (RXD) and *
                     106     ;                    interrupts (except if X4 installed)             *
                     107     ;*********************************************************************
                     108             extrn   code(outchr, inline, outstr, htoa, atoh, getpar)
                     109             extrn   code(load, dump, setcmd, serial_io_using_interrupts)
                     110             extrn   data(parmtr, tb_count)
                     111             extrn   bit(t_flag, r_flag, r_idle)
                     112             public  getcmd, linbuf, buflen, endbuf, notice
                     113             public  error, p_bit, riot, ram, rom, x13_bit
                     114
 0014                115     buflen  equ     20      ;maximum line length
 0007                116     bel     equ     07H     ;ASCII bell code
 000D                117     cr      equ     0DH     ;ASCII carriage return code
 000A                118     lf      equ     0AH     ;ASCII line feed code
 0100                119     riot    xdata   0100h   ;8155 RAM/IO/TIMER SBC51
 8000                120     ram     equ     8000h   ;RAM address for user software
 2000                121     rom     code    2000h   ;EPROM address for user firmware
                     122     eprom   segment code
                     123     onchip  segment data
                     124     bitram  segment bit
 0095                125     x13     bit     p1.5    ;reset option
```

```
  0094                   x3      bit     p1.4
  0093                   x4      bit     p1.3

 ----                            rseg    eprom
 0000 020000      F      main:   ljmp    skip            ;jump above interrupt vectors
 0003 300034      F              jnb     x4_bit,rom3     ;external 0 interrupt entry point
 0006 028003                     ljmp    ram+3
 0009 0000                       dw      0
 000B 30002F      F              jnb     x4_bit,rom6     ;timer 0 interrupt entry point
 000E 028006                     ljmp    ram+6
 0011 0000                       dw      0
 0013 30002A      F              jnb     x4_bit,rom9     ;external 1 interrupt entry point
 0016 028009                     ljmp    ram+9
 0019 0000                       dw      0
 001B 300025      F              jnb     x4_bit,rom12    ;timer 1 interrupt entry point
 001E 020000      F              ljmp    check           ;check jumper X13 first
 0021 0000                       dw      0
 0023 300020      F              jnb     x4_bit,rom15    ;serial port interrupt entry point
 0026 02800F                     ljmp    ram+15
 0029 0000                       dw      0
 002B 30001B      F              jnb     x4_bit,rom18    ;timer 2 interrupt entry point
 002E 028012                     ljmp    ram+18

                         ;*****************************************************************
                         ; Timer 1 interrupts are used for serial I/O if jumper X13 is     *
                         ; installed.  P1.7 is used for TXD and P1.6 is used for RXD.      *
                         ; (See IO module)                                                 *
                         ;*****************************************************************
 0031 200003      F      check:  jb      x13_bit,ram12   ;if X13 installed, serial I/O using
 0034 020000      F              ljmp    serial_io_using_interrupts      ;interrupts
 0037 02800C             ram12:  ljmp    ram+12          ;if X13 not installed, use RAM table
 003A 022003             rom3:   ljmp    rom+3           ;if X4 is installed, interrupts
 003D 022006             rom6:   ljmp    rom+6           ; are directed to a jump table in
 0040 022009             rom9:   ljmp    rom+9           ; user EPROM at 2003H (otherwise
 0043 02200C             rom12:  ljmp    rom+12          ; directed to jump table in user
 0046 02200F             rom15:  ljmp    rom+15          ; RAM at 8003H; see above)
 0049 022012             rom18:  ljmp    rom+18

 004C 436F7079           notice: db      'Copyright (c) I. Scott MacKenzie, 1988, 1991'
 0050 72696768
 0054 74202863
 0058 2920492E
 005C 2053636F
 0060 7474204D
 0064 61634B65
 0068 6E7A6965
 006C 2C203139
 0070 38382C20
 0074 31393931


                         ;*****************************************************************
                         ; Copy state of pins used for reset jumpers to flag bits.  Reset  *
                         ; jumpers are only read upon reset.  Subsequent tests are on the  *
                         ; flag bits.                                                      *
                         ;*****************************************************************
 0078 A294               skip:   mov     c,x3
 007A 9200        F              mov     x3_bit,c
 007C A293                       mov     c,x4
 007E 9200        F              mov     x4_bit,c
 0080 A295                       mov     c,x13
 0082 9200        F              mov     x13_bit,c
 0084 200003      F              jb      x3_bit,skip4    ;If x3 jumper installed,
 0087 022000                     ljmp    rom             ; jump to user EPROM at 2000H
 008A 200017      F      skip4:  jb      x13_bit,skip5   ;If x13 jumper installed,
008D C200     F              clr    r_flag          ; r_flag = 0 (no character waiting)
008F D200     F              setb   r_idle          ; r_idle = 1 (receiver idle)
0091 D200     F              setb   t_flag          ; t_flag = 1 (ready to transmit)
0093 75000B   F              mov    tb_count,#11    ; 11 bits sent (start + 8 + stop)
0096 758920                  mov    tmod,#20H       ; 8-bit auto-reload mode
0099 758D98                  mov    th1,#-104       ; 1 / (8 x 0.0012) us for 1200 baud
009C D28E                    setb   tr1             ; start timer 1 (interrupts coming)
009E D2AF                    setb   ea              ; turn on timer 1 interrupts - serial
00A0 D2AB                    setb   et1             ; I/O uses P1.7 (TXD) and P1.6 (RXD)
00A2 800B                    sjmp   skip6           ; and skip over serial port init
00A4 759852          skip5:  mov    scon,#01010010B ;If x13 not installed, initialize
00A7 758DF3                  mov    th1,#0F3H       ; for 2400 baud reload value
00AA 758920                  mov    tmod,#20H       ;auto reload mode
00AD D28E                    setb   tr1             ;start timer
00AF C200     F      skip6:  clr    p_bit           ;default = no printer output
00B1 900100                  mov    dptr,#riot      ;address of 8155
00B4 7401                    mov    a,#1            ;Port A = output
00B6 F0                      movx   @dptr,a         ;initialize printer port
00B7 900000   F              mov    dptr,#hello     ;send hello message to console
00BA 1100     F              acall  outstr
00BC 758107          getcmd: mov    sp,#stack - 1   ;initialize stack pointer
00BF 900000   F              mov    dptr,#prompt    ;send prompt to console
00C2 1100     F              acall  outstr
00C4 7800     F              mov    r0,#linbuf      ;R0 points to line buffer
00C6 1100     F              acall  inline          ;input command line from console
00C8 E500     F              mov    a,linbuf        ;get first character
00CA B40D03                  cjne   a,#cr,skip7     ;empty line?
00CD 020000   F              jmp    setcmd          ;yes: default to "set" command
                     skip7:                         ;check for alphabetic character
00D0 B44100        +2        cjne   a,#'A',$+3      ;JOR
00D3 404C          +2        jc     error
00D5 B45B00        +2        cjne   a,#'Z'+1,$+3
00D8 5047          +2        jnc    error
00DA 541F                    anl    a,#1fh          ;reduce to 5 bits
00DC 14                      dec    a               ;reduce ASCII 'A' to 0
00DD 23                      rl     a               ;adjust to word boundary
00DE F8                      mov    r0,a            ;save
00DF 04                      inc    a
00E0 900000   F              mov    dptr,#table     ;command address table
00E3 93                      movc   a,@a+dptr       ;get address low byte of command
00E4 C0E0                    push   acc             ;save on stack
00E6 E8                      mov    a,r0            ;restore accumulator
00E7 93                      movc   a,@a+dptr       ;get address high byte of command
00E8 C0E0                    push   acc             ;sneaky: pushing address on stack
00EA 1100     F              acall  getpar          ;get parameters from linbuf
00EC 22                      ret                    ;pop command address into PC and
                                                    ; off we go!
00ED 0000     F      table:  dw     error           ;A (Note: most commands undefined)
00EF 0000     F              dw     error           ;B
00F1 0000     F              dw     error           ;C
00F3 0000     F              dw     dump            ;Dump memory locations
00F5 0000     F              dw     error           ;E
00F7 0000     F              dw     error           ;F
00F9 0000     F              dw     go              ;Go to user program
00FB 0000     F              dw     error           ;H
00FD 0000     F              dw     error           ;I
00FF 0000     F              dw     error           ;J
0101 0000     F              dw     error           ;K
0103 0000     F              dw     load            ;Load hex file
0105 0000     F              dw     error           ;M
0107 0000     F              dw     error           ;N
0109 0000     F              dw     error           ;O
010B 0000     F              dw     error           ;P
010D 0000     F              dw     error           ;Q
```

```
010F 0000     F             dw      error           ;R
0111 0000     F             dw      setcmd          ;Set memory to value
0113 0000     F             dw      error           ;T
0115 0000     F             dw      error           ;U
0117 0000     F             dw      error           ;V
0119 0000     F             dw      error           ;W
0118 0000     F             dw      error           ;X
011D 0000     F             dw      error           ;Y
011F 0000     F             dw      error           ;Z
0121 900000   F     error:  mov     dptr,#emess     ;unimplemented command
0124 1100     F             acall   outstr
0126 E500     F             mov     a,linbuf        ;send out 1st two characters in
0128 1100     F             acall   outchr          ; line buffer too
012A E500     F             mov     a,linbuf + 1
012C 1100     F             acall   outchr
012E 808C                   jmp     getcmd

                    ;*******************************************************************
                    ; GO to user program command.  This command is small enough to      *
                    ; include with the main module.                                     *
                    ;                                                                   *
                    ;  FORMAT:  GO<address>                                             *
                    ;                                                                   *
                    ;*******************************************************************
0130 7400     F     go:     mov     a,#low(getcmd)  ;if "ret" from user program,
0132 C0E0                   push    acc             ; save "getcmd" address on stack
0134 7400     F             mov     a,#high(getcmd)
0136 C0E0                   push    acc
0138 C000     F             push    parmtr + 1      ;push address of user program on
013A C000     F             push    parmtr          ; stack
013C 22                     ret                     ;pop address into PC (Off we go!)

                    ;*******************************************************************
                    ; ASCII null-terminated strings                                     *
                    ;*******************************************************************
013D 0D             hello:  db      cr,'MON51',0
013E 4D4F4E35
0142 31
0143 00
0144 0D             prompt: db      cr,'V12>',0
0145 5631323E
0149 00
014A 07             emess:  db      bel,cr,'Error: Invalid Command - ',0
0148 0D
014C 4572726F
0150 723A2049
0154 6E76616C
0158 69642043
015C 6F6D6D61
0160 6E64202D
0164 20
0165 00

                    ;*******************************************************************
                    ; Reserve 24 bytes for the stack in an absolute segment starting at *
                    ; 08H.  (Note: MON51 does not use register banks 1-3.)              *
                    ;*******************************************************************
----                                dseg at 8
0008                stack:          ds      24

                    ;*******************************************************************
                    ; Create a line buffer for monitor commands in a relocatable data   *
                    ; segment.  See linker/locator listing (.M51) for absolute address  *
                    ; assigned.                                                         *
                    ;*******************************************************************
```

```
0000              296     linbuf:      ds      buflen         ;input command line goes here
  0014            297     endbuf       equ     $
                  298
                  299     ;*********************************************************************
                  300     ; Create 1-bit flags.  If p_bit flag set, OUTCHR transmits  to the  *
                  301     ; console and the printer; if clear, output only sent to the        *
                  302     ; console (default).  "x" bits are set to the state of the reset    *
                  303     ; jumpers as read immediately after reset.  Consult the link map    *
                  304     ; to determine the exact (i.e., absolute) location of these bits.   *
                  305     ;*********************************************************************
----              306                  rseg    bitram
0000              307     p_bit:       dbit    1
0001              308     x3_bit:      dbit    1
0002              309     x4_bit:      dbit    1
0003              310     x13_bit:     dbit    1
                  311                  end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

GET PARAMETERS FROM INPUT LINE

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN GETPAR.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE GETPAR.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $debug
                       2     $title(*** GET PARAMETERS FROM INPUT LINE ***)
                       3     $pagewidth(98)
                       4     $nosymbols
                       5     $nopaging
                       6     ;*********************************************************************
                       7     ;                                                                   *
                       8     ; GETPAR.SRC - GET PARAMETERS                                       *
                       9     ;                                                                   *
                      10     ; This routine scans the characters in the input buffer "linbuf"    *
                      11     ; and extracts the parameters used by the monitor commands.  The    *
                      12     ; input line must be of the form:                                   *
                      13     ;                                                                   *
                      14     ;  <C1><C2><P1>,<P2>,<P3>,<P4><CR>0                                 *
                      15     ;                                                                   *
                      16     ; where <C1> and <C2> are the 2-character command and <P1> to <P4>  *
                      17     ; are up to 4 parameters required by the commands.  Scanning begins *
                      18     ; after the 2-character command (i.e., at linbuf+2).                *
                      19     ;                                                                   *
                      20     ; Parameters are entered as hex ASCII codes separated by            *
                      21     ; commas.  The hex ASCII codes are converted to 16-bit binary       *
                      22     ; values and placed in memory beginning at "parmtr" in the order    *
                      23     ; high byte/low byte.                                               *
                      24     ;                                                                   *
                      25     ; Any parameters omitted retain their value from the previous       *
                      26     ; command.  Any illegal hex ASCII code terminates the scanning.     *
                      27     ;                                                                   *
                      28     ;*********************************************************************
                      29             extrn code(ishex, atoh)
                      30             extrn data(linbuf)
```

```
                                 public getpar, parmtr, reg_sp, bit_sp, int_sp, ext_sp, cde_sp

 0004                    numpar  equ     4               ;4 input parameters
                         eprom   segment code
                         onchip  segment data
                         bitram  segment bit

----                             rseg    eprom
0000 7800        F       getpar: mov     r0,#linbuf+2    ;R0 points to input line data
0002 7900        F               mov     r1,#parmtr      ;R1 points to parameters destination
0004 7F04                        mov     r7,#numpar      ;use R7 as counter
0006 B62C02              getp2:  cjne    @r0,#',',getp3  ;parameter missing?
0009 8008                        sjmp    getp4           ;yes: try to convert next parameter
000B 1100        F       getp3:  acall   gethx           ;no:  convert it
000D 300008      F               jnb     found,getp5     ;if illegal byte, exit
0010 B62C05                      cjne    @r0,#',',getp5  ;next character should be ','
0013 08                  getp4:  inc     r0
0014 09                          inc     r1
0015 09                          inc     r1              ;point to next parameter destination
0016 DFEE                        djnz    r7,getp2        ; and let's try for another
0018 1100        F       getp5:  acall   space           ;determine memory space selected
001A 22                          ret

                         ;*****************************************************************
                         ; Create space for the parameters in the "onchip" data segment.   *
                         ;*****************************************************************
----                             rseg    onchip
0000                     parmtr: ds      numpar*2        ;2 bytes required for each parameter

                         ;*****************************************************************
                         ; GETHX - GET HeX word from "linbuf"                              *
                         ;                                                                 *
                         ;       enter:  R0 points to string                               *
                         ;               R1 points to 2-byte destination                   *
                         ;       exit:   hex word in memory                                *
                         ;               R0 points to next non-hex character in string     *
                         ;               'found' = 1 if legal value found or 0 otherwise   *
                         ;       uses:   atoh, insrt, ishex                                *
                         ;                                                                 *
                         ;*****************************************************************
----                             rseg    eprom
001B C200        F       gethx:  clr     found           ;default = no value found
001D E6                          mov     a,@r0
001E 1100        F               acall   ishex           ;any value?
0020 5013                        jnc     gethx9          ;no:  return
0022 D200        F       gethx2: setb    found           ;set value found flag
0024 E4                          clr     a
0025 F7                          mov     @r1,a           ;clear destination value
0026 09                          inc     r1
0027 F7                          mov     @r1,a
0028 19                          dec     r1
0029 E6                  gethx3: mov     a,@r0           ;get value
002A 1100        F               acall   ishex           ;value found?
002C 5007                        jnc     gethx9          ;no:  return
002E 1100        F               acall   atoh            ;yes: attempt to convert it
0030 1100        F               acall   insrt           ; and insert into parameter
0032 08                          inc     r0              ;increment buffer pointer
0033 80F4                        sjmp    gethx3          ;repeat until illegal value
0035 22                  gethx9: ret

                         ;*****************************************************************
                         ; Create space for a "found" bit which is set as the command line  *
                         ; is scanned.                                                      *
                         ;*****************************************************************
```

```
----                     rseg    bitram
0000             found:  dbit    1

                 ;*******************************************************************
                 ; INSRT - INSeRT nibble from A into 16-bit value pointed at by R1   *
                 ;                                                                   *
                 ;       enter:  ACC.0 to ACC.3 contains hex nibble                  *
                 ;               R1 points to 2-byte RAM location                    *
                 ;       exit:   16-bit value shifted left 4 and nibble in ACC       *
                 ;                inserted into least significant digit              *
                 ;                                                                   *
                 ;*******************************************************************
----                     rseg    eprom
                         using   0
0036 C007        insrt:  push    AR7             ;use R7 as counter
0038 C000                push    AR0             ;use R0 to point to lsb
003A A801                mov     r0,1            ;R0 <-- R1
003C 08                  inc     r0
003D 7F04                mov     r7,#4           ;4 rotates
003F C4                  swap    a
0040 33          insrt2: rlc     a
0041 C0E0                push    acc
0043 E6                  mov     a,@r0
0044 33                  rlc     a
0045 F6                  mov     @r0,a
0046 E7                  mov     a,@r1
0047 33                  rlc     a
0048 F7                  mov     @r1,a
0049 D0E0                pop     acc
004B DFF3                djnz    r7,insrt2
004D D000                pop     AR0             ;R0
004F D007                pop     AR7             ;R7
0051 22                  ret

                 ;*******************************************************************
                 ; SPACE - determine memory SPACE selector as per second character   *
                 ;          in command line as follows:                              *
                 ;                                                                   *
                 ;               B = bit space                                       *
                 ;               C = code space                                      *
                 ;               I = internal data                                   *
                 ;               R = Special Functions Registers                     *
                 ;               X = external data                                   *
                 ;                                                                   *
                 ;*******************************************************************
0052 C200     F  space:  clr     bit_sp          ;default: no space selected
0054 C200     F          clr     cde_sp          ; clear all bits
0056 C200     F          clr     int_sp
0058 C200     F          clr     reg_sp
005A C200     F          clr     ext_sp
005C E500     F          mov     a,linbuf+1      ;memory space selector in command
005E B44202              cjne    a,#'B',spac2    ;determine space and set selector bit
0061 D200     F          setb    bit_sp
0063 B44302      spac2:  cjne    a,#'C',spac3
0066 D200     F          setb    cde_sp
0068 B44902      spac3:  cjne    a,#'I',spac4
006B D200     F          setb    int_sp
006D B45202      spac4:  cjne    a,#'R',spac5
0070 D200     F          setb    reg_sp
0072 B45802      spac5:  cjne    a,#'X',spac6
0075 D200     F          setb    ext_sp
0077 22          spac6:  ret
```

```
                  158     ;*******************************************************************
                  159     ; Create space for the memory space selector bits in the 8031      *
                  160     ; bit-addressable space.  See link map for exact address of each of *
                  161     ; these bits.                                                      *
                  162     ;*******************************************************************
----              163             rseg    bitram
0001              164     bit_sp: dbit    1
0002              165     cde_sp: dbit    1
0003              166     int_sp: dbit    1
0004              167     reg_sp: dbit    1
0005              168     ext_sp: dbit    1
                  169             end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

INPUT/OUTPUT ROUTINES

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN IO.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE IO.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $debug
                       2     $title(*** INPUT/OUTPUT ROUTINES ***)
                       3     $pagewidth(98)
                       4     $nopaging
                       5     $nosymbols
                       6     ;*******************************************************************
                       7     ;                                                                  *
                       8     ; IO.SRC - INPUT/OUTPUT SUBROUTINES                                *
                       9     ;                                                        SM 03/91 *
                      10     ;*******************************************************************
  0007                11     bel     equ     07H     ;ASCII bell code
  000D                12     cr      equ     0DH     ;ASCII carriage return code
  001B                13     esc     equ     1BH     ;ASCII escape code
  000A                14     lf      equ     0AH     ;ASCII line feed code
  0003                15     etx     equ     3       ;control-C (terminate input, get next cmd)
  0010                16     dle     equ     10h     ;control-P (output to printer as well)
  007F                17     del     equ     7FH     ;delete
  0004                18     bs      equ     4       ;back space (VT100)
  0097                19     TX_pin  bit     p1.7
  0096                20     RX_pin  bit     p1.6
                      21
                      22             extrn   code(getcmd, htoa, toupper)
                      23             extrn   data(linbuf)
                      24             extrn   bit(p_bit, x13_bit)
                      25             extrn   number(endbuf, riot)
                      26             public  inchar, outchr, inline, outstr, outhex, out2hx
                      27             public  serial_io_using_interrupts, t_flag, r_flag
                      28             public  tb_count, r_idle
                      29
                      30     eprom   segment code
                      31     onchip  segment data
                      32     bitram  segment bit
                      33
```

```
                    ;*******************************************************************
                    ; The following buffers & flags are needed for character I/O using  *
                    ; interrupts.  Interrupts are used only if jumper X13 is installed. *
                    ; Jumpers X9 and X10 should be removed and jumpers X11 and X12      *
                    ; should be installed.  This connects the RS232 TXD and RXD lines   *
                    ; to P1.7 and P1.6 (rather than to P3.1 and P3.0).                  *
                    ;*******************************************************************
----                                rseg    onchip
0000                t_buff:         ds      1       ;transmit_buffer (character to send)
0001                r_buff:         ds      1       ;receive_buffer (character received)
0002                ti_count:       ds      1       ;transmit_interrupt_count
0003                ri_count:       ds      1       ;receive_interrupt_count
0004                tb_count:       ds      1       ;transmit_bit_count
0005                rb_count:       ds      1       ;receive_bit_count
----                                rseg    bitram
0000                t_flag:         dbit    1       ;1 = end of character transmission
0001                r_flag:         dbit    1       ;1 = end of character reception
0002                r_idle:         dbit    1       ;1 = idle state

                    ;*******************************************************************
                    ; OUTCHR -   OUTput CHaRacter to serial port                        *
                    ;            with odd parity added in ACC.7                         *
                    ;                                                                   *
                    ;  enter:    ASCII code in accumulator                              *
                    ;  exit:     character written to SBUF; <cr> sent as <cr><lf>; all  *
                    ;            registers intact                                       *
                    ;                                                                   *
                    ;*******************************************************************
----                        rseg    eprom
0000 C0E0           outchr: push    acc
0002 A2D0           nl:     mov     c,p             ;add odd parity
0004 B3                     cpl     c
0005 92E7                   mov     acc.7,c
0007 20000D    F            jb      x13_bit,out1    ;if X13 installed, use interrupts
000A 3000FD    F            jnb     t_flag,$        ;wait for transmitter ready
000D C2AB                   clr     et1             ;begin "critical section"
000F C200      F            clr     t_flag          ;>>>  clear flag and ...
0011 F500      F            mov     t_buff,a        ;>>>  load data to transmit
0013 D2AB                   setb    et1             ;end "critical section"
0015 8007                   sjmp    out2
                                                    ;if X13 not installed, test TI
0017 3099FD         out1:   jnb     ti,$            ;wait for transmitter ready
001A C299                   clr     ti              ;clear TI flag
001C F599                   mov     sbuf,a          ;done!
001E C2E7           out2:   clr     acc.7           ;remove parity bit
0020 B40D04                 cjne    a,#cr,out3
0023 740A                   mov     a,#lf
0025 80DB                   jmp     nl              ;if <cr>, send <lf> as well
0027 D0E0           out3:   pop     acc
0029 300003    F            jnb     p_bit,out4      ;if printer control = off, exit
002C 120000    F            call    pchar           ;if on, print character
002F 22             out4:   ret

                    ;*******************************************************************
                    ; INCHR -   INput CHaRacter from serial port                        *
                    ;                                                                   *
                    ;  enter:   no conditions                                           *
                    ;  exit:    ASCII code in ACC.0 to ACC.6; ACC.7 cleared; control-C  *
                    ;           aborts to prompt                                        *
                    ;                                                                   *
                    ;*******************************************************************
0030 20000D    F    inchar: jb      x13_bit,in1     ;if X13 installed, use interrupts
0033 3000FD    F            jnb     r_flag,$        ;wait for receive_flag to be set
0036 C2AB                   clr     et1             ;begin "critical section"
```

```
0038 C200       F           clr     r_flag          ;>>> clear receive_flag and ...
003A E500       F           mov     a,r_buff        ;>>> read receive_buffer
003C D2AB                   setb    et1             ;end "critical section"
003E 8007                   sjmp    in2
                                                    ;if X13 not installed, test RI
0040 3098FD         in1:    jnb     ri,$            ;wait for receiver interrupt
0043 C298                   clr     ri              ;clear RI flag
0045 E599                   mov     a,sbuf          ;done!
0047 C2E7           in2:    clr     acc.7           ;clear parity bit (no error checking)
0049 B40303                 cjne    a,#etx,in3      ;if control-C,
004C 020000     F           jmp     getcmd          ; warm start
004F 22             in3:    ret

                    ;*******************************************************************
                    ; INLINE -  INput LINE of characters                               *
                    ;           line must end with <cr>; maximum size set by buflen;   *
                    ;           <bs> or <del> deletes previous character               *
                    ;                                                                  *
                    ;  enter:   R0 points to input buffer in internal data RAM         *
                    ;  exit:    ASCII codes in internal RAM; 0 stored at end of line   *
                    ;  uses:    inchar, outchr                                         *
                    ;                                                                  *
                    ;*******************************************************************
0050 120000     F   inline: call    inchar          ;get character from console
0053 120000     F           call    touppr          ;convert to uppercase, if necessary
0056 B41004                 cjne    a,#dle,inlin7   ;is it control-P?
0059 B200       F           cpl     p_bit           ;yes: toggle printing flag bit
005B 80F3                   jmp     inline
005D 120000     F   inlin7: call    outchr          ;echo character to port
0060 F6                     mov     @r0,a           ;put in buffer
0061 08                     inc     r0
0062 B80008     F           cjne    r0,#endbuf,inlin2 ;overflow?
0065 900000     F           mov     dptr,#inlin8
0068 1100       F           acall   outstr
006A 020000     F           jmp     getcmd
006D B40402         inlin2: cjne    a,#bs,inlin3    ;back space?
0070 8003                   sjmp    inlin5
0072 B47F0A         inlin3: cjne    a,#del,inlin6   ;delete?
0075 18             inlin5: dec     r0              ;yes: back up pointer twice
0076 18                     dec     r0
0077 900000     F           mov     dptr,#backup    ;back up cursor
007A 120000     F           call    outstr
007D 80D1                   jmp     inline
007F B40DCE         inlin6: cjne    a,#cr,inline    ;character = <cr> ?
0082 7600                   mov     @r0,#0          ;yes: store 0
0084 22                     ret                     ; and return
0085 1B             backup: db      esc,'[D',0      ;VT100 sequence to backup cursor
0086 5B44
0088 00
0089 07             inlin8: db      bel,cr,'Error: Command too long',cr,0
008A 0D
008B 4572726F
008F 723A2043
0093 6F6D6D61
0097 6E642074
009B 6F6F206C
009F 6F6E67
00A2 0D
00A3 00


                    ;*******************************************************************
                    ; OUTSTR -  OUTput a STRing of characters                          *
                    ;                                                                  *
                    ;  enter:   DPTR points to character string in external code       *
                    ;           memory; string must end with 00H                       *
                    ;  exit:    characters written to SBUF                        *
                    ;  uses:    outchr                                            *
                    ;                                                             *
                    ;**************************************************************
00A4 E4             outstr: clr     a
00A5 93                     movc    a,@a+dptr       ;get ASCII code
00A6 6006                   jz      outst2          ;if last code, done
00A8 120000    F            call    outchr          ;if not last code, send it
00AB A3                     inc     dptr            ;point to next code
00AC 80F6                   sjmp    outstr         ; and get next character
00AE 22             outst2: ret

                    ;**************************************************************
                    ; OUT2HX - OUTput 2 HeX characters to serial port             *
                    ; OUTHEX - OUTput 1 HEX character to serial port              *
                    ;                                                             *
                    ;  enter:   accumulator contains byte of data                 *
                    ;  exit:    nibbles converted to ASCII & sent out serial port;*
                    ;           OUT2HX sends both nibbles; OUTHEX only sends lower*
                    ;           nibble; all registers intact                      *
                    ;  uses:    outchr, htoa                                      *
                    ;                                                             *
                    ;**************************************************************
00AF C0E0           out2hx: push    acc             ;save data
00B1 C4                     swap    a               ;send high nibble first
00B2 540F                   anl     a,#0FH          ;make sure upper nibble clear
00B4 120000    F            call    htoa            ;convert hex nibble to ASCII
00B7 120000    F            call    outchr          ;send to serial port
00BA D0E0                   pop     acc
00BC C0E0           outhex: push    acc             ;send low nibble
00BE 540F                   anl     a,#0FH
00C0 120000    F            call    htoa
00C3 120000    F            call    outchr
00C6 D0E0                   pop     acc
00C8 22                     ret

                    ;**************************************************************
                    ; PCHAR -   Print CHARacter                                   *
                    ;           send character to Centronics interface on 8155;  *
                    ;           implements handshaking for -ACK, BUSY, and -STROBE*
                    ;                                                             *
                    ;  enter:   ASCII code in accumulator                         *
                    ;  exit:    character written to 8155 Port A                  *
                    ;                                                             *
                    ;**************************************************************
  0092              strobe  equ     92h             ;8051 port 1 interface line for
  0091              busy    equ     91h             ; printer interface
  0090              ack     equ     90h

00C9 C0E0           pchar:  push    acc
00CB C083                   push    dph
00CD C082                   push    dpl
00CF 2091FD         wait:   jb      busy,wait       ;wait for printer ready, i.e.,
00D2 3090FA                 jnb     ack,wait        ; BUSY = 0 AND -ACK = 1
00D5 900000    F            mov     dptr,#riot + 1  ;8155 Port A address
00D8 C2E7                   clr     acc.7           ;clear 8th bit, if set
00DA B40D02                 cjne    a,#cr,pchar2    ;substitute <lf> for <cr>
00DD 740A                   mov     a,#lf
00DF F0             pchar2: movx    @dptr,a         ;send data to printer
00E0 D292                   setb    strobe          ;toggle strobe bit
00E2 C292                   clr     strobe
00E4 D292                   setb    strobe
00E6 D082                   pop     dpl
00E8 D083                   pop     dph
```

```
00EA D0E0                    pop    acc
00EC 22                      ret

                ;*********************************************************************
                ; The following code, which executes upon a Timer 1 interrupt,       *
                ; implements a full duplex software UART.  Timer 1 interrupts occur *
                ; at a rate of 8 times the baud rate, or every 1 / (8 x 0.0012) =   *
                ; 104 microseconds.  For transmit, a bit is output on P1.7 every 8  *
                ; interrupts.  For receive, a bit is read in P1.6 every 8           *
                ; interrupts starting 12 interrupts after the start bit is          *
                ; detected.  At 12 MHz, worse case execution time for this routine  *
                ; is 53 us.  (This occurs for the last interrupt for simultaneous   *
                ; transmit and receive operations.)                                 *
                ;*********************************************************************
                serial_io_using_interrupts:
00ED C0E0                    push   acc
00EF C0D0                    push   psw
00F1 A200      F             mov    c,r_idle        ;if r_idle = 1 & p1.6 = 1, no RX
00F3 8296                    anl    c,RX_pin        ; activity, therefore ...
00F5 4027                    jc     tx1             ; check for TX activity
00F7 30000A    F             jnb    r_idle,rx1      ;else, if r_idle = 0 & p1.6 = x,
                                                    ; reception in progress, check it out
                                                    ;else, r_idle = 1 & p1.6 = 0, there-
                                                    ; fore, start bit detected
00FA C200      F             clr    r_idle          ;clear r_idle (begin reception)
00FC 75000C    F             mov    ri_count,#12    ;11 interrupts to first data bit
00FF 750009    F             mov    rb_count,#9     ;9 bits received (includes stop bit)
0102 801A                    sjmp   tx1
0104 D50017    F     rx1:    djnz   ri_count,tx1    ;decrement receive_interrupt_count
0107 750008    F             mov    ri_count,#8     ;if 8th interrupt, reload and
010A E500      F             mov    a,r_buff        ; read next serial bit on RXD (P1.6)
010C A296                    mov    c,RX_pin
010E 13                      rrc    a
010F F500      F             mov    r_buff,a
0111 D5000A    F             djnz   rb_count,tx1    ;decrement receive_bit_count
0114 750009    F             mov    rb_count,#9     ;if 9th bit received, done
0117 33                      rlc    a               ;re-align bits (discard stop bit) and
0118 F500      F             mov    r_buff,a        ; save ASCII code in receive_buffer
011A D200      F             setb   r_idle
011C D200      F             setb   r_flag          ;done! (now check for TX activity)
011E 20002C    F     tx1:    jb     t_flag,tx4      ;if t_flag = 1, nothing to transmit
0121 E500      F             mov    a,tb_count      ;check transmit_bit_count
0123 840B09                  cjne   a,#11,tx2       ;if 11, first time here, therefore
0126 C297                    clr    TX_pin          ; begin with start bit (p1.7 = 0)
0128 1500      F             dec    tb_count        ; decrement transmit_bit_count
012A 750008    F             mov    ti_count,#8     ;8 interrupts for each data bit
012D 801E                    sjmp   tx4
012F D5001B    F     tx2:    djnz   ti_count,tx4    ;if transmit_interrupt_count not 0
0132 750008    F             mov    ti_count,#8     ; exit, otherwise reset count to 8 &
0135 E500      F             mov    a,t_buff        ; get next data bit to transmit
0137 D3                      setb   c               ;(ensures 9th bit = stop bit)
0138 13                      rrc    a               ;right rotate (sends LSB first)
0139 9297                    mov    TX_pin,c        ;put bit on p1.7 (TXD)
013B F500      F             mov    t_buff,a        ;save in t_buff for next rotate
013D E500      F             mov    a,tb_count      ;check for stop bit (stretch count)
013F B40203                  cjne   a,#2,tx3        ;if count = 1, stop bit just sent
0142 750010    F             mov    ti_count,#16    ;stretch count to 16 (2 stop bits)
0145 D50005    F     tx3:    djnz   tb_count,tx4    ;if last bit,
0148 75000B    F             mov    tb_count,#11    ; reset count and
014B D200      F             setb   t_flag          ; hoist flag (done!)
014D D0D0            tx4:    pop    psw
014F D0E0                    pop    acc
0151 32                      reti
                             end
```

```
REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

CONVERSION SUBROUTINES

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN CONVRT.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE CONVRT.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $debug
                       2     $title(*** CONVERSION SUBROUTINES ***)
                       3     $pagewidth(98)
                       4     $nopaging
                       5     $nosymbols
                       6     $nolist                 ;next line contains $include(macros.src)
                      76     ;previous line contains $list
                      77     ;*****************************************************************
                      78     ;                                                                *
                      79     ; CONVRT.SRC - CONVERSION SUBROUTINES                             *
                      80     ;                                                                *
                      81     ;*****************************************************************
                      82             public  atoh, htoa, touppr, tolowr
                      83             extrn code(isalph)
                      84
                      85     eprom   segment code
----                  86             rseg    eprom
                      87
                      88     ;*****************************************************************
                      89     ; ATOH - Ascii TO Hex conversion                                  *
                      90     ;                                                                *
                      91     ;       enter:  ASCII code in accumulator (assume hex character)  *
                      92     ;       exit:   hex nibble in ACC.0 - ACC.3; ACC.4 - ACC.7 cleared *
                      93     ;                                                                *
                      94     ;*****************************************************************
0000 C2E7             95     atoh:   clr     acc.7           ;ensure parity bit is off
                      96                                     ;'0' to '9'?
0002 B43A00           97 +2          cjne    a,#'9'+1,$+3           ;JLE
0005 4002             98 +2          jc      atoh2
0007 2409             99             add     a,#9            ;no:  adjust for range A-F
0009 540F            100     atoh2:  anl     a,#0FH          ;yes: convert directly
000B 22              101             ret
                     102
                     103     ;*****************************************************************
                     104     ; HTOA - Hex TO Ascii conversion                                  *
                     105     ;                                                                *
                     106     ;       enter:  hex nibble in ACC.0 to ACC.3                      *
                     107     ;       exit:   ASCII code in accumulator                         *
                     108     ;                                                                *
                     109     ;*****************************************************************
000C 540F            110     htoa:   anl     a,#0FH          ;ensure upper nibble clear
                     111                                     ;'A' to 'F'?
000E B40A00          112 +2          cjne    a,#0AH,$+3             ;JLT
0011 4002            113 +2          jc      htoa2
0013 2407            114             add     a,#7            ;yes: add extra
0015 2430            115     htoa2:  add     a,#'0'          ;no:  convert directly
0017 22              116             ret
                     117
```

```
                        118     ;***********************************************************************
                        119     ; TOUPPR - convert character TO UPPeRcase                              *
                        120     ;                                                                      *
                        121     ;       enter:  ASCII code in accumulator                              *
                        122     ;       exit:   converted to uppercase if alphabetic (left as is       *
                        123     ;                otherwise)                                            *
                        124     ;       uses:   isalph                                                 *
                        125     ;                                                                      *
                        126     ;***********************************************************************
0018 120000    F        127     touppr: call    isalph          ;is character alphabetic?
001B 5002               128             jnc     skip            ;no:  leave as is
001D C2E5               129             clr     acc.5           ;yes: convert to uppercase by
001F 22                 130     skip:   ret                     ;     clearing bit 5
                        131
                        132     ;***********************************************************************
                        133     ; TOLOWR - convert character TO LOWeRcase                              *
                        134     ;                                                                      *
                        135     ;       enter:  ASCII code in accumulator                              *
                        136     ;       exit:   converter to lowercase if alphabetic (left as is       *
                        137     ;                otherwise)                                            *
                        138     ;       uses:   isalph                                                 *
                        139     ;                                                                      *
                        140     ;***********************************************************************
0020 120000    F        141     tolowr: call    isalph          ;is character alphabetic?
0023 5002               142             jnc     skip2           ;no:  leave as is
0025 D2E5               143             setb    acc.5           ;yes: convert to lowercase by setting
0027 22                 144     skip2:  ret                     ;     bit 5
                        145             end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

LOAD INTEL HEX FILE

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN LOAD.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE LOAD.SRC EP


LOC  OBJ            LINE     SOURCE
                       1     $debug
                       2     $title(*** LOAD INTEL HEX FILE ***)
                       3     $pagewidth(98)
                       4     $nopaging
                       5     $nosymbols
                       6     ;***********************************************************************
                       7     ;                                                                      *
                       8     ; LOAD.SRC - LOAD INTEL HEX FILE COMMAND                               *
                       9     ;                                                                      *
                      10     ;  FORMAT: LO - load and return to MON51                               *
                      11     ;          LG - load and go to user program in RAM                     *
                      12     ;                                                                      *
                      13     ;***********************************************************************
                      14             extrn   code (inchar, outstr, atoh, getcmd,touppr)
                      15             extrn   data (linbuf,parmtr)
                      16             extrn   number (ram)
                      17             public  load
                      18
  0007                19     bel     equ     07H             ;ASCII bell code
  000D                20     cr      equ     0DH             ;ASCII carraige code
  000A                  lf      equ     0AH            ;ASCII line feed code
  001A                  eof     equ     1AH            ;ASCII control-z (end-of-file)
  0003                  cancel  equ     03H            ;ASCII control-c (cancel)
                        eprom   segment code
----                            rseg    eprom

0000 900000     F       load:   mov     dptr,#mess1
0003 1100       F               acall   outstr
0005 1100       F       load1:  acall   inchar
0007 B41A02                     cjne    a,#eof,skip1
000A 8031                       sjmp    done
000C B40303             skip1:  cjne    a,#cancel,skip2
000F 020000     F               jmp     getcmd
0012 B43AF0             skip2:  cjne    a,#':',load1    ;each record begins with a ':'
0015 7900                       mov     r1,#0           ;initialize checksum to 0
0017 1100       F               acall   gethex          ;get byte count from serial port
0019 F5F0                       mov     b,a             ;use B as byte counter
001B 6020                       jz      done            ;if B = 0, done
001D 05F0               load2:  inc     b               ;no:
001F 1100       F               acall   gethex          ;get address high byte
0021 F583                       mov     dph,a
0023 1100       F               acall   gethex          ;get address low byte
0025 F582                       mov     dpl,a
0027 1100       F               acall   gethex          ;get record type (ignore)
0029 1100       F       load4:  acall   gethex          ;get data byte
002B F0                         movx    @dptr,a         ;store in 8031 external memory
002C A3                         inc     dptr
002D 15F0                       dec     b               ;repeat until count = 0
002F E5F0                       mov     a,b
0031 70F6                       jnz     load4
0033 E9                         mov     a,r1            ;checksum should be zero
0034 60CF                       jz      load1           ;if so, get next record
0036 900000     F       error:  mov     dptr,#mess4     ;if not, error
0039 1100       F               acall   outstr
003B 800F                       sjmp    wait
003D E500       F       done:   mov     a,linbuf+1      ;Load and Go command?
003F 1100       F               acall   touppr
0041 B44703                     cjne    a,#'G',skip     ;no:  normal end to command
0044 020000     F               ljmp    ram             ;yes: go to user program
0047 900000     F       skip:   mov     dptr,#mess3
004A 1100       F               acall   outstr
004C 1100       F       wait:   acall   inchar          ;wait for eof before returning
004E B41AFB                     cjne    a,#eof,wait
0051 020000     F               jmp     getcmd

                        ;**********************************************************************
                        ; Get two characters from serial port and form a hex byte.  Also     *
                        ; add byte to checksum in R1.                                         *
                        ;**********************************************************************
0054 1100       F       gethex: acall   inchar          ;get first character
0056 1100       F               acall   atoh            ;convert to hex
0058 C4                         swap    a               ;put in upper nibble
0059 F8                         mov     r0,a            ;save it
005A 1100       F               acall   inchar          ;get second character
005C 1100       F               acall   atoh            ;convert to hex
005E 48                         orl     a,r0            ;OR with first nibble
005F FA                         mov     r2,a            ;save byte
0060 29                         add     a,r1            ;add byte to checksum
0061 F9                         mov     r1,a            ;restore checksum in R1
0062 EA                         mov     a,r2            ;retrieve byte
0063 22                         ret
0064 486F7374           mess1:  db      'Host download to SBC-51',cr
0068 20646F77
006C 6E6C6F61
```

```
0070 6420746F
0074 20534243
0078 2D3531
007B 00
007C 5E5A203D        83             db      '^Z = end-of-file',cr,'^C = cancel',cr,0
0080 20656E64
0084 2D6F662D
0088 66696C65
008C 0D
008D 5E43203D
0091 2063616E
0095 63656C
0098 0D
0099 00
009A 656F6620        84     mess3:  db      'eof - file downloaded OK',0
009E 2D206669
00A2 6C652064
00A6 6F776E6C
00AA 6F616465
00AE 64204F4B
00B2 00
00B3 07              85     mess4:  db      bel,cr,'Error: ^Z Terminates',0
00B4 0D
00B5 4572726F
00B9 723A205E
00BD 5A205465
00C1 726D696E
00C5 61746573
00C9 00
                     86             end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

DUMP MEMORY TO CONSOLE

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN DUMP.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE DUMP.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $debug
                       2     $title(*** DUMP MEMORY TO CONSOLE ***)
                       3     $pagewidth(98)
                       4     $nopaging
                       5     $nosymbols
                       6     ;*********************************************************************
                       7     ;                                                                    *
                       8     ; DUMP MEMORY COMMAND                                                *
                       9     ;                                                                    *
                      10     ;       FORMAT: D<char><start>,<end>                                 *
                      11     ;                                                                    *
                      12     ;               D       - dump memory command                        *
                      13     ;               <char>  - memory space selector as follows:          *
                      14     ;                               I = internal data RAM                *
                      15     ;                               X = external data RAM                *
                      16     ;                               C = code memory                      *
                      17     ;                               B = bit address space                *
```

```
                  ;                               R = SFRs                          *
                  ;               <start> - starting address to dump                *
                  ;               <end>   - ending address to dump                  *
                  ;                                                                 *
                  ;******************************************************************
                          extrn code (getcmd, outchr, htoa, out2hx, rsfr, isgrph)
                          extrn code (getval, inchar, outhex)
                          extrn data (parmtr, linbuf)
                          extrn bit (bit_sp, int_sp, reg_sp, x13_bit, r_flag)
                          public dump, outdat, outadd

                  onchip  segment data
                  eprom   segment code
 000D             cr      equ     0DH
 0013             xoff    equ     13H
 0020             space   equ     ' '

----                      rseg    eprom
0000 850083   F   dump:   mov     dph,parmtr      ;starting address to dump
0003 850082   F           mov     dpl,parmtr+1
0006 750000   F   dump1:  mov     ascbuf,#0       ;initialize ASCII string to NULL
0009 7900     F           mov     r1,#ascbuf      ;R1 = pointer into ASCII string
000B 200005   F           jb      x13_bit,dump1a  ;check keyboard status, which is
000E 30000C   F           jnb     r_flag,dump2    ; r_flag if x13 jumper installed, or
0011 8003                 sjmp    dump1b
0013 309807       dump1a: jnb     ri,dump2        ; ri if x13 jumper not installed
0016 1100     F   dump1b: acall   inchar
0018 B41302               cjne    a,#xoff,dump2   ;if Control-S
001B 1100     F           acall   inchar          ; wait for next key
001D 1100     F   dump2:  acall   getval          ;read byte of data
001F C0E0                 push    acc
0021 1100     F           acall   outadd
0023 D0E0         dump3:  pop     acc
0025 1100     F           acall   insert          ;add to ASCII buffer
0027 1100     F           acall   outdat          ;send to console
0029 40D8                 jc      dump1           ;if C, new line
002B 1100     F           acall   getval
002D C0E0                 push    acc
002F 80F2                 sjmp     dump3

                  ;******************************************************************
                  ; OUTADD - OUTput ADDress to console                              *
                  ;                                                                 *
                  ;       enter:  DPTR contains address; one memory space selector  *
                  ;                bit set                                          *
                  ;       exit:   appropriate size (8-bit or 16-bit) address sent to *
                  ;                console; followed by " = " if "set" command active *
                  ;       uses:   out2hx, outchr                                    *
                  ;                                                                 *
                  ;******************************************************************
0031 200020   F   outadd: jb      bit_sp,outad5   ;if bit, register, or internal space
0034 20001D   F           jb      reg_sp,outad5   ; only send 8-bit address
0037 20001A   F           jb      int_sp,outad5
003A E583                 mov     a,dph
003C 1100     F           acall   out2hx
003E E582         outad2: mov     a,dpl
0040 1100     F           acall   out2hx
0042 E500     F           mov     a,linbuf        ;get command character
0044 B45308               cjne    a,#'S',outad3
0047 7420         outad4: mov     a,#space
0049 1100     F           acall   outchr
004B 743D                 mov     a,#'='          ;send '=' if "set" command
004D 1100     F           acall   outchr
004F 7420         outad3: mov     a,#space
```

```
0051 1100     F          acall   outchr
0053 22                  ret
0054 E583         outad5: mov    a,dph          ;address must be in range
0056 60E6                jz      outad2         ; 00H to FFH for bit, register,
0058 020000   F          jmp     getcmd         ; or internal memory spaces

                  ;*****************************************************************
                  ; OUTDAT - OUTput DATa to console                                 *
                  ;                                                                 *
                  ;       enter:  data in acc; one memory space selector bit set   *
                  ;       exit:   appropriate size value (1 bit or 8 bit) sent to  *
                  ;                console; ASCII buffer flushed if dump command and *
                  ;                end of boundary reached                          *
                  ;       uses:   outhex, outasc                                    *
                  ;                                                                 *
                  ;*****************************************************************
005B C4           outdat: swap   a              ;this gets tricky!
005C 200002   F          jb      bit_sp,outdt2  ;if bit space,
005F 1100     F          acall   outhex         ;only send nibble
0061 C4           outdt2: swap   a
0062 1100     F          acall   outhex
0064 E500     F          mov     a,linbuf       ;if "set" command,
0066 6453                xrl     a,#'S'         ; don't check <end>
0068 600D                jz      outad4         ; send ' = '
006A E583                mov     a,dph          ;DPTR = <end>?
006C 6500     F          xrl     a,parmtr+2
006E 7011                jnz     outdt3
0070 E582                mov     a,dpl
0072 6500     F          xrl     a,parmtr+3
0074 700B                jnz     outdt3
0076 200005   F          jb      reg_sp,outdt8  ;if bit or register space,
0079 200002   F          jb      bit_sp,outdt8  ; don't send ASCII
007C 1100     F          acall   outasc         ;yes: flush buffer
007E 020000   F   outdt8: jmp    getcmd         ;  & get next command
0081 A3           outdt3: inc    dptr           ;no:  onwards
0082 200012   F          jb      reg_sp,outdt7  ;if register space, one byte/line
0085 300006   F          jnb     bit_sp,outdt6  ;if bit space, check for 8-bit
0088 E582                mov     a,dpl          ; boundary
008A 5407                anl     a,#07H
008C 6009                jz      outdt7
008E E582         outdt6: mov    a,dpl          ;if internal, external, or code
0090 540F                anl     a,#0FH         ; space, check for 16-byte boundary
0092 C3                  clr     c
0093 7007                jnz     outdt4
0095 1100     F          acall   outasc         ;if there, flush ASCII buffer
0097 740D         outdt7: mov    a,#cr
0099 1100     F          acall   outchr
009B D3                  setb    c
009C 22           outdt4: ret

                  ;*****************************************************************
                  ; INSERT - INSERT ASCII code into buffer                          *
                  ;                                                                 *
                  ;       enter:  byte of data in accumulator                       *
                  ;       exit:   ASCII code in buffer ('.' substituted for control *
                  ;                codes)                                           *
                  ;       uses:   isgrph                                            *
                  ;                                                                 *
                  ;*****************************************************************
009D C0E0         insert: push   acc
009F 1100     F          acall   isgrph         ;is it a displayable character?
00A1 4002                jc      insrt2         ;yes: leave as is
00A3 742E                mov     a,#'.'         ;no:  substitute period
00A5 F7           insrt2: mov    @r1,a
```

```
00A6 09               146             inc     r1
00A7 7700             147             mov     @r1,#0          ;null character at end
00A9 D0E0             148             pop     acc
00AB 22               149             ret
                      150
                      151     ;*******************************************************************
                      152     ; OUTASC - OUTput ASCii codes to console                            *
                      153     ;                                                                   *
                      154     ;       enter:  -                                                   *
                      155     ;       exit:   buffer of ASCII graphic codes sent to console       *
                      156     ;                                                                   *
                      157     ;*******************************************************************
00AC 7420             158     outasc: mov     a,#space
00AE 1100       F     159             acall   outchr
00B0 7900       F     160             mov     r1,#ascbuf
00B2 E7               161     out3:   mov     a,@r1
00B3 7001             162             jnz     out2
00B5 22               163             ret
00B6 1100       F     164     out2:   acall   outchr
00B8 09               165             inc     r1
00B9 80F7             166             sjmp    out3
                      167
                      168     ;*******************************************************************
                      169     ; Create a buffer in internal RAM to hold ASCII codes to be dumped  *
                      170     ; to console for DUMP command.                                      *
                      171     ;*******************************************************************
----                  172             rseg    onchip
0000                  173     ascbuf: ds      17              ;ascii buffer
                      174             end

REGISTER BANK(S) USED: 0
```

READ AND WRITE SFRs

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN SFR.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE SFR.SRC EP


LOC  OBJ            LINE     SOURCE

                       1     $debug
                       2     $title (*** READ AND WRITE SFRs ***)
                       3     $pagewidth(98)
                       4     $nopaging
                       5     $nosymbols
                       6     ;*******************************************************************
                       7     ;                                                                   *
                       8     ; SFR.SRC - READ AND WRITE SPECIAL FUNCTION REGISTERS               *
                       9     ;                                                                   *
                      10     ;*******************************************************************
                      11
                      12             public rsfr, wsfr
                      13
                      14     eprom   segment code
----                  15             rseg    eprom
                      16
                      17     ;*******************************************************************
                      18     ;                                                                   *
                      19     ; RSFR - Read Special Function Register                             *
                      20     ;                                                                   *
```

```
                  ;       enter:  R0 contains address of SFR to read              *
                  ;       exit:   accumulator contains value and C = 0            *
                  ;               if invalid SFR, C = 1                           *
                  ;                                                               *
                  ;****************************************************************

                  ; Let's get lazy and define a macro to do all the work.

                  rsfr:
0000 B88003    +2         cjne   r0,#80h,SKIP00      ;p0
0003 E580      +2         mov     a,80h
0005 22        +1         ret
               +2 SKIP00:
0006 B88103    +2         cjne   r0,#81h,SKIP01      ;sp
0009 E581      +2         mov     a,81h
000B 22        +1         ret
               +2 SKIP01:
000C B88203    +2         cjne   r0,#82h,SKIP02      ;dpl
000F E582      +2         mov     a,82h
0011 22        +1         ret
               +2 SKIP02:
0012 B88303    +2         cjne   r0,#83h,SKIP03      ;dph
0015 E583      +2         mov     a,83h
0017 22        +1         ret
               +2 SKIP03:
0018 B88803    +2         cjne   r0,#88h,SKIP04      ;tcon
001B E588      +2         mov     a,88h
001D 22        +1         ret
               +2 SKIP04:
001E B88903    +2         cjne   r0,#89h,SKIP05      ;tmod
0021 E589      +2         mov     a,89h
0023 22        +1         ret
               +2 SKIP05:
0024 B88A03    +2         cjne   r0,#8ah,SKIP06      ;tl0
0027 E58A      +2         mov     a,8ah
0029 22        +1         ret
               +2 SKIP06:
002A B88B03    +2         cjne   r0,#8bh,SKIP07      ;tl1
002D E58B      +2         mov     a,8bh
002F 22        +1         ret
               +2 SKIP07:
0030 B88C03    +2         cjne   r0,#8ch,SKIP08      ;th0
0033 E58C      +2         mov     a,8ch
0035 22        +1         ret
               +2 SKIP08:
0036 B88D03    +2         cjne   r0,#8dh,SKIP09      ;th1
0039 E58D      +2         mov     a,8dh
003B 22        +1         ret
               +2 SKIP09:
003C B89003    +2         cjne   r0,#90h,SKIP0A      ;p1
003F E590      +2         mov     a,90h
0041 22        +1         ret
               +2 SKIP0A:
0042 B89803    +2         cjne   r0,#98h,SKIP0B      ;scon
0045 E598      +2         mov     a,98h
0047 22        +1         ret
               +2 SKIP0B:
0048 B89903    +2         cjne   r0,#99h,SKIP0C      ;sbuf
004B E599      +2         mov     a,99h
004D 22        +1         ret
               +2 SKIP0C:
004E B8A003    +2         cjne   r0,#0a0h,SKIP0D      ;p2
0051 E5A0      +2         mov     a,0a0h
0053 22        +1         ret
```

```
                        85 +2  SKIP0D:
0054 B8A803             86 +2          cjne   r0,#0a8h,SKIP0E       ;ie
0057 E5A8               87 +2          mov     a,0a8h
0059 22                 88 +1          ret
                        89 +2  SKIP0E:
005A B8B003             90 +2          cjne   r0,#0b0h,SKIP0F       ;p3
005D E5B0               91 +2          mov     a,0b0h
005F 22                 92 +1          ret
                        93 +2  SKIP0F:
0060 B8B803             94 +2          cjne   r0,#0b8h,SKIP10       ;ip
0063 E5B8               95 +2          mov     a,0b8h
0065 22                 96 +1          ret
                        97 +2  SKIP10:
0066 B8D003             98 +2          cjne   r0,#0d0h,SKIP11       ;psw
0069 E5D0               99 +2          mov     a,0d0h
006B 22                100 +1          ret
                       101 +2  SKIP11:
006C B8E003            102 +2          cjne   r0,#0e0h,SKIP12       ;acc
006F E5E0              103 +2          mov     a,0e0h
0071 22                104 +1          ret
                       105 +2  SKIP12:
0072 B8F003            106 +2          cjne   r0,#0f0h,SKIP13       ;b
0075 E5F0              107 +2          mov     a,0f0h
0077 22                108 +1          ret
                       109 +2  SKIP13:
0078 D3                110             setb    c                ;C = 1 if invalid SFR
0079 22                111             ret
                       112
                       113     ;*******************************************************************
                       114     ;                                                                  *
                       115     ; WSFR - Write Special Function Register                           *
                       116     ;                                                                  *
                       117     ;         enter:  R0 contains address of SFR                       *
                       118     ;                 accumulator contains value to write              *
                       119     ;         exit:   value written to SFR and C = 0                   *
                       120     ;                 if invalid SFR, C = 1                            *
                       121     ;                                                                  *
                       122     ;*******************************************************************
                       123
                       124     wsfr:
007A B88003            125 +2          cjne   r0,#80h,SKIP14        ;p0
007D F580              126 +2          mov     80h,a
007F 22                127 +1          ret
                       128 +2  SKIP14:
0080 B88103            129 +2          cjne   r0,#81h,SKIP15        ;sp
0083 F581              130 +2          mov     81h,a
0085 22                131 +1          ret
                       132 +2  SKIP15:
0086 B88203            133 +2          cjne   r0,#82h,SKIP16        ;dpl
0089 F582              134 +2          mov     82h,a
008B 22                135 +1          ret
                       136 +2  SKIP16:
008C B88303            137 +2          cjne   r0,#83h,SKIP17        ;dph
008F F583              138 +2          mov     83h,a
0091 22                139 +1          ret
                       140 +2  SKIP17:
0092 B88803            141 +2          cjne   r0,#88h,SKIP18        ;tcon
0095 F588              142 +2          mov     88h,a
0097 22                143 +1          ret
                       144 +2  SKIP18:
0098 B88903            145 +2          cjne   r0,#89h,SKIP19        ;tmod
009B F589              146 +2          mov     89h,a
009D 22                147 +1          ret
                       148 +2  SKIP19:
```

```
009E B88A03        149 +2           cjne    r0,#8ah,SKIP1A       ;tl0
00A1 F58A          150 +2           mov      8ah,a
00A3 22            151 +1           ret
                   152 +2   SKIP1A:
00A4 B88B03        153 +2           cjne    r0,#8bh,SKIP1B       ;tl1
00A7 F58B          154 +2           mov      8bh,a
00A9 22            155 +1           ret
                   156 +2   SKIP1B:
00AA B88C03        157 +2           cjne    r0,#8ch,SKIP1C       ;th0
00AD F58C          158 +2           mov      8ch,a
00AF 22            159 +1           ret
                   160 +2   SKIP1C:
00B0 B88D03        161 +2           cjne    r0,#8dh,SKIP1D       ;th1
00B3 F58D          162 +2           mov      8dh,a
00B5 22            163 +1           ret
                   164 +2   SKIP1D:
00B6 B89003        165 +2           cjne    r0,#90h,SKIP1E       ;p1
00B9 F590          166 +2           mov      90h,a
00BB 22            167 +1           ret
                   168 +2   SKIP1E:
00BC B89803        169 +2           cjne    r0,#98h,SKIP1F       ;scon
00BF F598          170 +2           mov      98h,a
00C1 22            171 +1           ret
                   172 +2   SKIP1F:
00C2 B89903        173 +2           cjne    r0,#99h,SKIP20       ;sbuf
00C5 F599          174 +2           mov      99h,a
00C7 22            175 +1           ret
                   176 +2   SKIP20:
00C8 B8A003        177 +2           cjne    r0,#0a0h,SKIP21       ;p2
00CB F5A0          178 +2           mov      0a0h,a
00CD 22            179 +1           ret
                   180 +2   SKIP21:
00CE B8A803        181 +2           cjne    r0,#0a8h,SKIP22       ;ie
00D1 F5A8          182 +2           mov      0a8h,a
00D3 22            183 +1           ret
                   184 +2   SKIP22:
00D4 B8B003        185 +2           cjne    r0,#0b0h,SKIP23       ;p3
00D7 F5B0          186 +2           mov      0b0h,a
00D9 22            187 +1           ret
                   188 +2   SKIP23:
00DA B8B803        189 +2           cjne    r0,#0b8h,SKIP24       ;ip
00DD F5B8          190 +2           mov      0b8h,a
00DF 22            191 +1           ret
                   192 +2   SKIP24:
00E0 B8D003        193 +2           cjne    r0,#0d0h,SKIP25       ;psw
00E3 F5D0          194 +2           mov      0d0h,a
00E5 22            195 +1           ret
                   196 +2   SKIP25:
00E6 B8E003        197 +2           cjne    r0,#0e0h,SKIP26       ;acc
00E9 F5E0          198 +2           mov      0e0h,a
00EB 22            199 +1           ret
                   200 +2   SKIP26:
00EC B8F003        201 +2           cjne    r0,#0f0h,SKIP27       ;b
00EF F5F0          202 +2           mov      0f0h,a
00F1 22            203 +1           ret
                   204 +2   SKIP27:
00F2 D3            205              setb     c                 ;if reached here, invalid
00F3 22            206              ret                        ;SFR, set carry flag
                   207              end


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN IS.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE IS.SRC EP


LOC  OBJ            LINE     SOURCE
                       1     $title (*** IS ROUTINES ***)
                       2     $debug
                       3     $pagewidth (98)
                       4     $nopaging
                       5     $nosymbols
                       6     $nolist                 ;next line contains $include(macros.src)
                      76     ;previous line contains $list
                      77     ;*******************************************************************
                      78     ;                                                                  *
                      79     ; IS.SRC - IS SUBROUTINES                                          *
                      80     ;                                                                  *
                      81     ; These subroutines test a byte to see if it matches a certain     *
                      82     ; condition.  If so, they return with the carry flag set; if not,  *
                      83     ; they return with the carry flag clear.  These subroutines only   *
                      84     ; alter the carry flag; the accumulator is left intact.            *
                      85     ;                                                                  *
                      86     ;*******************************************************************
                      87
                      88             public  isgrph, ishex, isdig, isalph
                      89     eprom   segment code
----                  90             rseg    eprom
                      91
                      92     ;*******************************************************************
                      93     ;                                                                  *
                      94     ; ISGRPH - IS the byte an ascii GRaPHic code (i.e., in the range   *
                      95     ;              20H to 7EH)                                         *
                      96     ;                                                                  *
                      97     ;       enter:  ASCII code in accumulator                          *
                      98     ;       exit:   C = 1 if code is ASCII graphic character           *
                      99     ;               C = 0 if code is ASCII control character           *
                     100     ;                                                                  *
                     101     ;*******************************************************************
0000 B42000          102     isgrph: cjne    a,#20h,$+3      ;set C if < space
0003 10D703          103             jbc     cy,isgrp2       ;if set, clear and return
0006 B47F00          104             cjne    a,#7fh,$+3      ;set C, if graphic
0009 22              105     isgrp2: ret
                     106
                     107     ;*******************************************************************
                     108     ;                                                                  *
                     109     ; ISHEX - IS character ascii HEX?                                  *
                     110     ;                                                                  *
                     111     ;       enter:  ASCII code in ACC                                  *
                     112     ;       exit:   C = 1 if in range 0-9, a-f, or A-F                 *
                     113     ;               C = 0 otherwise                                    *
                     114     ;       uses:   isdigt                                             *
                     115     ;                                                                  *
                     116     ;*******************************************************************
000A C0E0            117     ishex:  push    acc
000C 120000     F    118             call    isdig
000F 400B            119             jc      skip            ;if digit, then ishex = true
0011 D2E5            120             setb    acc.5           ;convert to lowercase
0013 B46100          121             cjne    a,#'a',$+3      ;C = 1 if < 'a'
0016 10D703          122             jbc     cy,skip         ;if 1, clear and return
0019 B46700          123             cjne    a,#'f'+1,$+3    ;carry set if hex
001C D0E0            124     skip:   pop     acc
001E 22              125             ret
```

```
                        126
                        127     ;***********************************************************************
                        128     ;                                                                       *
                        129     ; ISDIG - IS ascii code a DIGit?                                        *
                        130     ;                                                                       *
                        131     ;       enter:  ASCII code in accumulator                               *
                        132     ;       exit:   C = 1 if code is character in range 0-9                 *
                        133     ;               C = 0 otherwise                                         *
                        134     ;                                                                       *
                        135     ;***********************************************************************
001F B43000             136     isdig:  cjne    a,#'0',$+3      ;carry set if < 0
0022 10D703             137             jbc     cy,skip2        ;if set, clear and return
0025 B43A00             138             cjne    a,#'9'+1,$+3    ;carry set if digit
0028 22                 139     skip2:  ret
                        140
                        141     ;***********************************************************************
                        142     ;                                                                       *
                        143     ; ISALPH - IS ascii code an ALPHabetic character                        *
                        144     ;                                                                       *
                        145     ;       enter:  ASCII code in accumulator                               *
                        146     ;       exit:   C = 1 if code is character in range a-z or A-Z          *
                        147     ;               C = 0 otherwise                                         *
                        148     ;                                                                       *
                        149     ;***********************************************************************
0029 B46100             150 +2  isalph: cjne   a,#'a',$+3       ;JIR
002C 4005               151 +2          jc      SKIP00
002E B47B00             152 +2          cjne    a,#'z'+1,$+3
0031 400C               153 +2          jc      yes
                        154 +2  SKIP00:
0033 B44100             155 +2          cjne   a,#'A',$+3       ;JIR
0036 4005               156 +2          jc      SKIP01
0038 B45B00             157 +2          cjne    a,#'Z'+1,$+3
003B 4002               158 +2          jc      yes
                        159 +2  SKIP01:
003D C3                 160             clr     c               ;if reached here, can't be alpha
003E 22                 161             ret
003F D3                 162     yes:    setb    c               ;must be alphabetic character
0040 22                 163             ret
                        164             end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

SET MEMORY TO VALUE

```
DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN SET.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE SET.SRC EP


LOC  OBJ              LINE     SOURCE

                         1     $title (*** SET MEMORY TO VALUE ***)
                         2     $debug
                         3     $pagewidth (98)
                         4     $nopaging
                         5     $nosymbols
                         6     $nolist                  ;next line contains $include(macros.src)
                        76     ;previous line contains $list
                        77      ;*******************************************************************
                        78      ;                                                                  *
                        79      ; SET.SRC - SET MEMORY TO VALUE                                    *
                        80      ;                                                                  *
                        81      ;       FORMAT: S<char><add><cr>                                   *
                        82      ;                                                                  *
                        83      ;               S       - set command                              *
                        84      ;               <char>  - memory space selector                    *
                        85      ;                         B = bit address space                    *
                        86      ;                         R = SFRs                                 *
                        87      ;                         I = internal data RAM                    *
                        88      ;                         X = external data RAM                    *
                        89      ;                         C = code ROM                             *
                        90      ;               <add>   - address to set                           *
                        91      ;               <cr>    - carriage return                          *
                        92      ;                                                                  *
                        93      ;       FOLLOWED BY:    <cr>     - examine next                    *
                        94      ;                       <value>  - write data                      *
                        95      ;                       -        - examine previous                *
                        96      ;                       Q        - quit                            *
                        97      ;                       <sp>     - write same value into next      *
                        98      ;                                  address                         *
                        99      ;                                                                  *
                        100     ;*******************************************************************
                        101             extrn   code (rsfr, wsfr, outadd, getpar, getcmd)
                        102             extrn   code (inline, outhex, outdat, outstr)
                        103             extrn   bit (bit_sp, cde_sp, int_sp, reg_sp, ext_sp)
                        104             extrn   data (parmtr, linbuf)
                        105             public  setcmd, getval
                        106
 0007                   107     bel     equ     07H
 000D                   108     cr      equ     0DH
                        109     eprom   segment code
----                    110             rseg    eprom
                        111
0000 850083     F       112     setcmd: mov     dph,parmtr      ;address to examine and/or set
0003 850082     F       113             mov     dpl,parmtr+1
0006 1100       F       114     set4:   acall   getval
0008 C0E0               115             push    acc
000A 1100       F       116             acall   outadd
000C D0E0               117             pop     acc
000E 1100       F       118             acall   outdat
0010 7800       F       119             mov     r0,#linbuf+2    ;start address of data
0012 1100       F       120             acall   inline
0014 E500       F       121             mov     a,linbuf+2      ;check response character
0016 B45103             122             cjne    a,#'Q',set6     ;if 'Q', quit set command
0019 020000     F       123             ljmp    getcmd
001C B42D0F             124     set6:   cjne    a,#'-',set7     ;if '-', decrement address pointer
001F C0E0               125 +1          push   acc            ;DEC_DPTR
0021 1582               126 +1          dec     dpl
0023 E582               127 +1          mov     a,dpl
0025 B4FF02             128 +2          cjne    a,#0FFH,SKIP00
0028 1583               129 +1          dec     dph
002A D0E0               130 +2  SKIP00:  pop     acc
002C 80D8               131             sjmp    set4
002E B40D03             132     set7:   cjne    a,#0dh,set8     ;if <cr>, examine next location
0031 A3                 133             inc     dptr
0032 80D2               134             sjmp    set4
0034 1100       F       135     set8:   acall   getpar          ;otherwise a value has been entered
0036 E500       F       136             mov     a,parmtr+1      ;convert value and
0038 1100       F       137             acall   putval          ; put it into memory
003A A3                 138             inc     dptr            ;check out next location
003B 80C9               139             sjmp    set4
                        140
```

```
                    ;*******************************************************************
                    ; PUTVAL - PUT VALue into memory                                    *
                    ;                                                                   *
                    ;       enter:  DPTR contains address; ACC contains value; one      *
                    ;                memory space selector bit set as follows:          *
                    ;                 bit_sp = 1 for bit address space                  *
                    ;                 cde_sp = 1 for code ROM                           *
                    ;                 int_sp = 1 for internal data RAM                  *
                    ;                 reg_sp = 1 for SFRs                               *
                    ;                 ext_sp = 1 for external data RAM                  *
                    ;               If bit address space selected then                  *
                    ;                 R4 = byte address                                 *
                    ;                 R5 = read byte value                              *
                    ;               (register bank 0 assumed)                           *
                    ;               Bit will be inserted into read byte                 *
                    ;                 value and byte value will be                      *
                    ;                 written back to byte address                      *
                    ;       exit:   value written into memory                           *
                    ;                                                                   *
                    ;*******************************************************************
003D A882           putval: mov     r0,dpl          ;put address in R0 also
003F 300002    F            jnb     int_sp,put2
0042 F6                     mov     @r0,a           ;internal data RAM
0043 22                     ret
0044 300002    F    put2:   jnb     ext_sp,put3
0047 F0                     movx    @dptr,a         ;external data RAM
0048 22                     ret
0049 300001    F    put3:   jnb     cde_sp,put4
004C 22                     ret                     ;sorry, can't write into code space
004D 30001E    F    put4:   jnb     bit_sp,put5
0050 5401                   anl     a,#1            ;make sure bits 1-7 = 0
0052 530007                 anl     0,#7            ;reduce R0 to bit address
                                                    ;writing byte value
0055 7DFE                   mov     r5,#0feh        ;build mask in R5
0057 08                     inc     r0
0058 D802           put9:   djnz    r0,put7
005A 8006                   sjmp    put8
005C 23             put7:   rl      a               ;adjust bit position
005D CD                     xch     a,r5
005E 23                     rl      a               ;adjust mask
005F CD                     xch     a,r5
0060 80F6                   sjmp    put9
0062 CC             put8:   xch     a,r4            ;byte value read earlier
0063 5D                     anl     a,r5            ;turn off bit of interest
0064 4C                     orl     a,r4            ;set if entered value = 1
                                                    ;byte value to write in accumulator
0065 A803                   mov     r0,3            ;recover byte address
0067 B88000                 cjne    r0,#80h,$+3     ;SFR address?
006A 5005                   jnc     put10           ;yes if C = 1
006C F6                     mov     @r0,a           ;Done!! Whew!
006D 22                     ret
006E 300005    F    put5:   jnb     reg_sp,put6
0071 1100      F    put10:  acall   wsfr
0073 4001                   jc      put6
0075 22                     ret
0076 900000    F    put6:   mov     dptr,#put11     ;no memory space selected
0079 1100      F            acall   outstr          ;unrecoverable error, send message
007B 020000    F            ljmp    getcmd          ; and get another command
007E 07             put11:  db      bel,cr,'Error: no memory space selected',cr,0
007F 0D
0080 4572726F
0084 723A206E
0088 6F206D65
008C 6D6F7279
0090 20737061
```

```
0094 63652073
0098 656C6563
009C 746564
009F 0D
00A0 00

                    ;*****************************************************************
                    ; GETVAL - GET VALue from memory                                  *
                    ;                                                                 *
                    ;       enter:  DPTR contains address; One memory space selector  *
                    ;                 bit set (see PUTVAL)                            *
                    ;       exit:   Value in ACC, C = 0                               *
                    ;               If bit address space selected then                *
                    ;                 R3 = byte address                               *
                    ;                 R4 = read byte value                            *
                    ;                 (register bank 0 assumed)                       *
                    ;               If invalid SRF address, advance                   *
                    ;                 DPTR until valid address found                  *
                    ;                                                                 *
                    ;*****************************************************************
00A1 A882           getval: mov     r0,dpl          ;put addres in R0 also

00A3 300002    F            jnb     int_sp,get2
00A6 E6                     mov     a,@r0           ;internal data RAM
00A7 22                     ret                     ;C = 0
00A8 300002    F    get2:   jnb     ext_sp,get3
00AB E0                     movx    a,@dptr         ;external data RAM
00AC 22                     ret                     ;C = 0
00AD 300003    F    get3:   jnb     cde_sp,get4
00B0 E4                     clr     a
00B1 93                     movc    a,@a+dptr       ;code ROM
00B2 22                     ret                     ;C = 0
00B3 30002D    F    get4:   jnb     bit_sp,get5
                                                    ;So you want to read a bit
                                                    ;value, do you?
                                                    ;This gets tricky!!
00B6 5300F8                 anl     0,#0f8h         ;reduce R0 to byte address
00B9 B88000                 cjne    r0,#80h,$+3     ;if >= 80H, read SFR
00BC 400A                   jc      get6
00BE AB00                   mov     r3,0            ;save byte address in R3
00C0 1100      F            acall   rsfr            ;first get byte value
00C2 FC                     mov     r4,a            ;save byte value in R4
00C3 500D                   jnc     get8            ;now extract bit
00C5 A3                     inc     dptr            ;if C = 1, try next SFR
00C6 80D9                   sjmp    getval
00C8 E8             get6:   mov     a,r0            ;internal RAM, therefore
00C9 03                     rr      a               ;build byte address in acc
00CA 03                     rr      a               ;by translating as shown
00CB 03                     rr      a
00CC D2E5                   setb    acc.5           ;eg: bit 35H is at byte address 26H
00CE F8                     mov     r0,a
00CF FB                     mov     r3,a            ;save byte address in R3
00D0 E6                     mov     a,@r0           ;read byte value
00D1 FC                     mov     r4,a            ;save byte value in R4
00D2 AF82           get8:   mov     r7,dpl          ;extract bit by rotating
00D4 530707                 anl     7,#7            ;right DPL mod 8 times
00D7 0F                     inc     r7
00D8 DF02           get9:   djnz    r7,get10
00DA 8003                   sjmp    get11
00DC 03             get10:  rr      a
00DD 80F9                   sjmp    get9
00DF 5401           get11:  anl     a,#1            ;here's your bit
00E1 C3                     clr     c
00E2 22                     ret
```

```
00E3 300009    F        258      get5:   jnb     reg_sp,get7
00E6 1100      F        259      get13:  acall   rsfr
00E8 5004               260              jnc     get12            ;if invalid SFR address,
00EA A3                 261              inc     dptr             ;increment DPTR & try again
00EB 08                 262              inc     r0
00EC 80F8               263              sjmp    get13
00EE 22                 264      get12:  ret
00EF 8085               265      get7:   jmp     put6             ;Error: no memory space selected
                        266              end

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

MACROS.SRC

```
;*******************************************************************
; JLT - Jump to "label" if accumulator Less Than "value"          *
;*******************************************************************
%*define(jlt(value,label))
       (cjne   a,#%value,$+3             ;JLT
        jc     %label)


;*******************************************************************
; JGT - Jump to "label" if accumulator Greater Than "value"       *
;*******************************************************************
%*define(jgt(value,label))
        (cjne  a,#%value+1,$+3           ;JGT
        jnc    %label)


;*******************************************************************
; JLE - Jump to "label" if accumulator Less than or Equal to "value"*
;*******************************************************************
%*define(jle(value,label))
        (cjne  a,#%value+1,$+3           ;JLE
        jc     %label)


;*******************************************************************
; JGE - Jump to "label" if accumulator Greater than or Equal to    *
;       "value"                                                    *
;*******************************************************************
%*define(jge(value,label))
        (cjne  a,#%value,$+3             ;JGE
        jnc    %label)


;*******************************************************************
; JOR - Jump to "label" if accumulator Out of Range of "lower_value"*
;       and "upper_value"                                          *
;*******************************************************************
%*define(jor(lower_value,upper_value,label))
        (cjne  a,#%lower_value,$+3       ;JOR
        jc     %label
        cjne   a,#%upper_value+1,$+3
        jnc    %label)
```

```
;*********************************************************************
;
; JIR - Jump to "label" if accumulator In Range of "lower_value" and*
;       "upper_value"                                               *
;*********************************************************************
;
%*define(jir(lower_value,upper_value,label)) local skip
        (cjne   a,#%lower_value,$+3       ;JIR
        jc      %skip
        cjne    a,#%upper_value+1,$+3
        jc      %label
%skip:  )

;*********************************************************************
;
; DECREMENT DPTR                                                    *
;*********************************************************************
;
%*define(dec    dptr) local skip
        (push   acc             ;DEC_DPTR
        dec     dpl
        mov     a,dpl
        cjne    a,#0FFH,%skip
        dec     dph
%skip:  pop     acc)

;*********************************************************************
;
; PUSH DPTR ONTO STACK                                              *
;*********************************************************************
;
%*define(push   dptr)
        (push   dpl             ;PUSH_DPTR
        push    dph)


;*********************************************************************
;
; POP DPTR FROM STACK                                               *
;*********************************************************************
;
%*define(pop    dptr)
        (pop    dph             ;POP_DPTR
        pop     dpl)

;*********************************************************************
;
%*define (enable_register_bank(n))
        (%if (%n) then (setb    rs0) else (clr    rs0) fi
       %if (%n gt 1) then (setb   rs1) else (clr    rs1) fi)

;*********************************************************************
;
%*define (send_message(p))
        (push   dpl
        push    dph
        mov    dptr,#%p
        call   outstr
        pop    dph
        pop    dpl)
```

V12.M51

```
DATE : 04/21/91
DOS 3.31 (038-N) MCS-51 RELOCATOR AND LINKER V3.0, INVOKED BY:
C:\ASM51\RL51.EXE MAIN.OBJ,GETPAR.OBJ,IO.OBJ,CONVRT.OBJ,LOAD.OBJ,DUMP.OBJ,SFR.
>> OBJ,IS.OBJ,SET.OBJ TO V12 DATA(ONCHIP(30H))CODE(EPROM(0))


INPUT MODULES INCLUDED
  MAIN.OBJ(MAIN)
  GETPAR.OBJ(GETPAR)
  IO.OBJ(IO)
  CONVRT.OBJ(CONVRT)
  LOAD.OBJ(LOAD)
  DUMP.OBJ(DUMP)
  SFR.OBJ(SFR)
  IS.OBJ(IS)
  SET.OBJ(SET)


LINK MAP FOR V12(MAIN)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ----    ----      ------    ----------   ------------

            REG     0000H     0008H                  "REG BANK 0"
            DATA    0008H     0018H     ABSOLUTE
            BIT     0020H     0001H.5   UNIT         BITRAM
                    0021H.5   000EH.3                *** GAP ***
            DATA    0030H     0033H     UNIT         ONCHIP

            CODE    0000H     0703H     UNIT         EPROM
```

Note: The following symbol table has been abreviated for this printout. Only symbols declared as "public" are shown.

```
SYMBOL TABLE FOR V12(MAIN)

VALUE          TYPE           NAME
-----          ----           ----

-------        MODULE         MAIN
N:0014H        PUBLIC         BUFLEN
D:0044H        PUBLIC         ENDBUF
C:0121H        PUBLIC         ERROR
C:00BCH        PUBLIC         GETCMD
D:0030H        PUBLIC         LINBUF
C:004CH        PUBLIC         NOTICE
B:0020H        PUBLIC         P_BIT
N:8000H        PUBLIC         RAM
X:0100H        PUBLIC         RIOT
C:2000H        PUBLIC         ROM
B:0020H.3      PUBLIC         X13_BIT
-------        ENDMOD         MAIN

-------        MODULE         GETPAR
B:0020H.5      PUBLIC         BIT_SP
B:0020H.6      PUBLIC         CDE_SP
B:0021H.1      PUBLIC         EXT_SP
C:0166H        PUBLIC         GETPAR
B:0020H.7      PUBLIC         INT_SP
D:0044H        PUBLIC         PARMTR
B:0021H        PUBLIC         REG_SP
-------        ENDMOD         GETPAR

-------        MODULE         IO
C:020EH        PUBLIC         INCHAR
C:022EH        PUBLIC         INLINE
C:028DH        PUBLIC         OUT2HX
```

```
C:01DEH        PUBLIC         OUTCHR
C:029AH        PUBLIC         OUTHEX
C:0282H        PUBLIC         OUTSTR
B:0021H.3      PUBLIC         R_FLAG
B:0021H.4      PUBLIC         R_IDLE
C:02CBH        PUBLIC         SERIAL_IO_USING_INTERRUPTS
B:0021H.2      PUBLIC         T_FLAG
D:0050H        PUBLIC         TB_COUNT
-------        ENDMOD         IO

-------        MODULE         CONVRT
C:0330H        PUBLIC         ATOH
C:033CH        PUBLIC         HTOA
C:0350H        PUBLIC         TOLOWR
C:0348H        PUBLIC         TOUPPR
-------        ENDMOD         CONVRT

-------        MODULE         LOAD
C:0358H        PUBLIC         LOAD
-------        ENDMOD         LOAD

-------        MODULE         DUMP
C:0422H        PUBLIC         DUMP
C:0453H        PUBLIC         OUTADD
C:047DH        PUBLIC         OUTDAT
-------        ENDMOD         DUMP

-------        MODULE         SFR
C:04DDH        PUBLIC         RSFR
C:0557H        PUBLIC         WSFR
-------        ENDMOD         SFR

-------        MODULE         IS
C:05FAH        PUBLIC         ISALPH
C:05F0H        PUBLIC         ISDIG
C:05D1H        PUBLIC         ISGRPH
C:05DBH        PUBLIC         ISHEX
-------        ENDMOD         IS

-------        MODULE         SET
C:06B3H        PUBLIC         GETVAL
C:0612H        PUBLIC         SETCMD
-------        ENDMOD         SET
```

V12.HEX

```
:10 0000 00 02 00 78 30 02 34 02 80 03 00 00 30 02 2F 02 80   A8
:10 0010 00 06 00 00 30 02 2A 02 80 09 00 00 30 02 25 02 00   9A
:10 0020 00 31 00 00 30 02 20 02 80 0F 00 00 30 02 1B 02 80   ED
:10 0030 00 12 20 03 03 02 02 CB 02 80 0C 02 20 03 02 20 06   DE
:10 0040 00 02 20 09 02 20 0C 02 20 0F 02 20 12 43 6F 70 79   57
:10 0050 00 72 69 67 68 74 20 28 63 29 20 49 2E 20 53 63 6F   D2
:10 0060 00 74 74 20 4D 61 63 4B 65 6E 7A 69 65 2C 20 31 39   5B
:10 0070 00 38 38 2C 20 31 39 39 31 A2 94 92 01 A2 93 92 02   5E
:10 0080 00 A2 95 92 03 20 01 03 02 20 00 20 03 17 C2 0B D2   85
:10 0090 00 0C D2 0A 75 50 0B 75 89 20 75 8D 98 D2 8E D2 AF   0F
:10 00A0 00 D2 AB 80 0B 75 98 52 75 8D F3 75 89 20 D2 8E C2   B4
:10 00B0 00 00 90 01 00 74 01 F0 90 01 3D 51 82 75 81 07 90   1C
:10 00C0 00 01 44 51 82 78 30 51 2E E5 30 B4 0D 03 02 06 12   FE
:10 00D0 00 B4 41 00 40 4C B4 5B 00 50 47 54 1F 14 23 F8 04   53
:10 00E0 00 90 00 ED 93 C0 E0 E8 93 C0 E0 31 66 22 01 21 01   69
```

```
:10 00F0 00 21 01 21 04 22 01 21 01 21 01 30 01 21 01 21 01   DD
:10 0100 00 21 01 21 03 58 01 21 01 21 01 21 01 21 01 21 01   A6
:10 0110 00 21 06 12 01 21 01 21 01 21 01 21 01 21 01 21 01   D9
:10 0120 00 21 90 01 4A 51 82 E5 30 31 DE E5 31 31 DE 80 8C   AB
:10 0130 00 74 BC C0 E0 74 00 C0 E0 C0 45 C0 44 22 0D 4D 4F   07
:10 0140 00 4E 35 31 00 0D 56 31 32 3E 00 07 0D 45 72 72 6F   4B
:10 0150 00 72 3A 20 49 6E 76 61 6C 69 64 20 43 6F 6D 6D 61   FF
:10 0160 00 6E 64 20 2D 20 00 78 32 79 44 7F 04 B6 2C 02 80   02
:10 0170 00 08 31 81 30 04 08 B6 2C 05 08 09 09 DF EE 31 B8   D2
:10 0180 00 22 C2 04 E6 B1 DB 50 13 D2 04 E4 F7 09 F7 19 E6   02
:10 0190 00 B1 DB 50 07 71 30 31 9C 08 80 F4 22 C0 07 C0 00   E9
:10 01A0 00 A8 01 08 7F 04 C4 33 C0 E0 E6 33 F6 E7 33 F7 D0   94
:10 01B0 00 E0 DF F3 D0 00 D0 07 22 C2 05 C2 06 C2 07 C2 08   A2
:10 01C0 00 C2 09 E5 31 B4 42 02 D2 05 B4 43 02 D2 06 B4 49   B1
:10 01D0 00 02 D2 07 B4 52 02 D2 08 B4 58 02 D2 09 22 C0 E0   B7
:10 01E0 00 A2 D0 B3 92 E7 20 03 0D 30 0A FD C2 AB C2 0A F5   DC
:10 01F0 00 4C D2 AB 80 07 30 99 FD C2 99 F5 99 C2 E7 B4 0D   96
:10 0200 00 04 74 0A 80 DB D0 E0 30 00 03 12 02 A7 22 20 03   2E
:10 0210 00 0D 30 0B FD C2 AB C2 0B E5 4D D2 AB 80 07 30 98   61
:10 0220 00 FD C2 98 E5 99 C2 E7 B4 03 03 02 00 BC 22 12 02   A2
:10 0230 00 0E 12 03 48 B4 10 04 B2 00 80 F3 12 01 DE F6 08   77
:10 0240 00 B8 44 08 90 02 67 51 82 02 00 BC B4 04 02 80 03   E3
:10 0250 00 B4 7F 0A 18 18 90 02 63 12 02 82 80 D1 B4 0D CE   C6
:10 0260 00 76 00 22 1B 5B 44 00 07 0D 45 72 72 6F 72 3A 20   C4
:10 0270 00 43 6F 6D 6D 61 6E 64 20 74 6F 6F 20 6C 6F 6E 67   7D
:10 0280 00 0D 00 E4 93 60 06 12 01 DE A3 80 F6 22 C0 E0 C4   F4
:10 0290 00 54 0F 12 03 3C 12 01 DE D0 E0 C0 E0 54 0F 12 03   F1
:10 02A0 00 3C 12 01 DE D0 E0 22 C0 E0 C0 83 C0 82 20 91 FD   7C
:10 02B0 00 30 90 FA 90 01 01 C2 E7 B4 0D 02 74 0A F0 D2 92   B4
:10 02C0 00 C2 92 D2 92 D0 82 D0 83 D0 E0 22 C0 E0 C0 D0 A2   2D
:10 02D0 00 0C 82 96 40 27 30 0C 0A C2 0C 75 4F 0C 75 51 09   E0
:10 02E0 00 80 1A D5 4F 17 75 4F 08 E5 4D A2 96 13 F5 4D D5   D9
:10 02F0 00 51 0A 75 51 09 33 F5 4D D2 0C D2 0B 20 0A 2C E5   69
:10 0300 00 50 B4 0B 09 C2 97 15 50 75 4E 08 80 1E D5 4E 1B   70
:10 0310 00 75 4E 08 E5 4C D3 13 92 97 F5 4C E5 50 B4 02 03   A3
:10 0320 00 75 4E 10 D5 50 05 75 50 0B D2 0A D0 D0 D0 E0 32   A2
:10 0330 00 C2 E7 B4 3A 00 40 02 24 09 54 0F 22 54 0F B4 0A   11
:10 0340 00 00 40 02 24 07 24 30 22 12 05 FA 50 02 C2 E5 22   9E
:10 0350 00 12 05 FA 50 02 D2 E5 22 90 03 BC 51 82 51 0E B4   2C
:10 0360 00 1A 02 80 31 B4 03 03 02 00 BC B4 3A F0 79 00 71   80
:10 0370 00 AC F5 F0 60 20 05 F0 71 AC F5 83 71 AC F5 82 71   DD
:10 0380 00 AC 71 AC F0 A3 15 F0 E5 F0 70 F6 E9 60 CF 90 04   25
:10 0390 00 0B 51 82 80 0F E5 31 71 48 B4 47 03 02 80 00 90   11
:10 03A0 00 03 F2 51 82 51 0E B4 1A FB 02 00 BC 51 0E 71 30   9F
:10 03B0 00 C4 F8 51 0E 71 30 48 FA 29 F9 EA 22 48 6F 73 74   73
:10 03C0 00 20 64 6F 77 6E 6C 6F 61 64 20 74 6F 20 53 42 43   BA
:10 03D0 00 2D 35 31 0D 5E 5A 20 3D 20 65 6E 64 2D 6F 66 2D   E2
:10 03E0 00 66 69 6C 65 0D 5E 43 20 3D 20 63 61 6E 63 65 6C   DC
:10 03F0 00 0D 00 65 6F 66 20 2D 20 66 69 6C 65 20 64 6F 77   3F
:10 0400 00 6E 6C 6F 61 64 65 64 20 4F 4B 00 07 0D 45 72 72   1E
:10 0410 00 6F 72 3A 20 5E 5A 20 54 65 72 6D 69 6E 61 74 65   20
:10 0420 00 73 00 85 44 83 85 45 82 75 52 00 79 52 20 03 05   07
:10 0430 00 30 08 0C 80 03 30 98 07 51 0E B4 13 02 51 0E D1   CB
:10 0440 00 B3 C0 E0 91 53 D0 E0 91 8F 91 7D 40 DB D1 B3 C0   08
:10 0450 00 E0 80 F2 20 05 20 20 08 1D 20 07 1A E5 83 51 8D   39
:10 0460 00 E5 82 51 8D E5 30 B4 53 08 74 20 31 DE 74 3D 31   9E
:10 0470 00 DE 74 20 31 DE 22 E5 83 60 E6 02 00 BC C4 20 05   84
:10 0480 00 02 51 9A C4 51 9A E5 30 64 53 60 DD E5 83 65 46   B4
:10 0490 00 70 11 E5 82 65 47 70 0B 20 08 05 20 05 02 91 CE   9A
:10 04A0 00 02 00 BC A3 20 08 12 30 05 06 E5 82 54 07 60 09   4B
```

```
:10 0480 00 E5 82 54 0F C3 70 07 91 CE 74 0D 31 DE 03 22 C0   94
:10 04C0 00 E0 B1 D1 40 02 74 2E F7 09 77 00 D0 E0 22 74 20   09
:10 04D0 00 31 DE 79 52 E7 70 01 22 31 DE 09 80 F7 B8 80 03   FE
:10 04E0 00 E5 80 22 B8 81 03 E5 81 22 B8 82 03 E5 82 22 B8   43
:10 04F0 00 83 03 E5 83 22 B8 88 03 E5 88 22 B8 89 03 E5 89   68
:10 0500 00 22 B8 8A 03 E5 8A 22 B8 8B 03 E5 8B 22 B8 8C 03   D4
:10 0510 00 E5 8C 22 B8 8D 03 E5 8D 22 B8 90 03 E5 90 22 B8   D2
:10 0520 00 98 03 E5 98 22 B8 99 03 E5 99 22 B8 A0 03 E5 A0   BD
:10 0530 00 22 B8 A8 03 E5 A8 22 B8 B0 03 E5 B0 22 B8 B8 03   F2
:10 0540 00 E5 B8 22 B8 D0 03 E5 D0 22 B8 E0 03 E5 E0 22 B8   50
:10 0550 00 F0 03 E5 F0 22 D3 22 B8 80 03 F5 80 22 B8 81 03   AE
:10 0560 00 F5 81 22 B8 82 03 F5 82 22 B8 83 03 F5 83 22 B8   8D
:10 0570 00 88 03 F5 88 22 B8 89 03 F5 89 22 B8 8A 03 F5 8A   A9
:10 0580 00 22 B8 8B 03 F5 8B 22 B8 8C 03 F5 8C 22 B8 8D 03   2F
:10 0590 00 F5 8D 22 B8 90 03 F5 90 22 B8 98 03 F5 98 22 B8   0B
:10 05A0 00 99 03 F5 99 22 B8 A0 03 F5 A0 22 B8 A8 03 F5 A8   ED
:10 05B0 00 22 B8 B0 03 F5 B0 22 B8 B8 03 F5 B8 22 B8 D0 03   1A
:10 05C0 00 F5 D0 22 B8 E0 03 F5 E0 22 B8 F0 03 F5 F0 22 D3   2D
:10 05D0 00 22 B4 20 00 10 D7 03 B4 7F 00 22 C0 E0 12 05 F0   3F
:10 05E0 00 40 0B D2 E5 B4 61 00 10 D7 03 B4 67 00 D0 E0 22   1D
:10 05F0 00 B4 30 00 10 D7 03 B4 3A 00 22 B4 61 00 40 05 B4   0F
:10 0600 00 7B 00 40 0C B4 41 00 40 05 B4 5B 00 40 02 C3 22   B3
:10 0610 00 D3 22 85 44 83 85 45 82 D1 B3 C0 E0 91 53 D0 E0   95
:10 0620 00 91 7D 78 32 51 2E E5 32 B4 51 03 02 00 BC B4 2D   D5
:10 0630 00 0F C0 E0 15 82 E5 82 B4 FF 02 15 83 D0 E0 80 D8   B8
:10 0640 00 B4 0D 03 A3 80 D2 31 66 E5 45 D1 4F A3 80 C9 A8   7C
:10 0650 00 82 30 07 02 F6 22 30 09 02 F0 22 30 06 01 22 30   F1
:10 0660 00 05 1E 54 01 53 00 07 7D FE 08 D8 02 80 06 23 CD   E5
:10 0670 00 23 CD 80 F6 CC 5D 4C A8 03 B8 80 00 50 05 F6 22   4F
:10 0680 00 30 08 05 B1 57 40 01 22 90 06 90 51 82 02 00 BC   0B
:10 0690 00 07 0D 45 72 72 6F 72 3A 20 6E 6F 20 6D 65 6D 6F   37
:10 06A0 00 72 79 20 73 70 61 63 65 20 73 65 6C 65 63 74 65   2E
:10 06B0 00 64 0D 00 A8 82 30 07 02 E6 22 30 09 02 E0 22 30   F1
:10 06C0 00 06 03 E4 93 22 30 05 2D 53 00 F8 B8 80 00 40 0A   59
:10 06D0 00 AB 00 91 DD FC 50 0D A3 80 D9 E8 03 03 03 D2 E5   04
:10 06E0 00 F8 FB E6 FC AF 82 53 07 07 0F DF 02 80 03 03 80   AD
:10 06F0 00 F9 54 01 C3 22 30 08 09 91 DD 50 04 A3 08 80 F8   A1
:03 0700 00 22 80 85                                          CF
:00 0000 01                                                   FF
```

H

A Guide to Keil's µVISION2 IDE

INTRODUCTION

Keil's µVision2 Integrated Development Environment (IDE)[1] is software that allows the 8051 C programmer to edit, compile, run, and debug 8051 C programs. This appendix presents a brief guide to using the µVision2 IDE.

THE µVISION2 WORKSPACE

When you double-click the µVision2 IDE, you will see a screen similar to the one shown in Figure H-1. The main windows that appear are the Workspace, the Edit window, and the Output window.

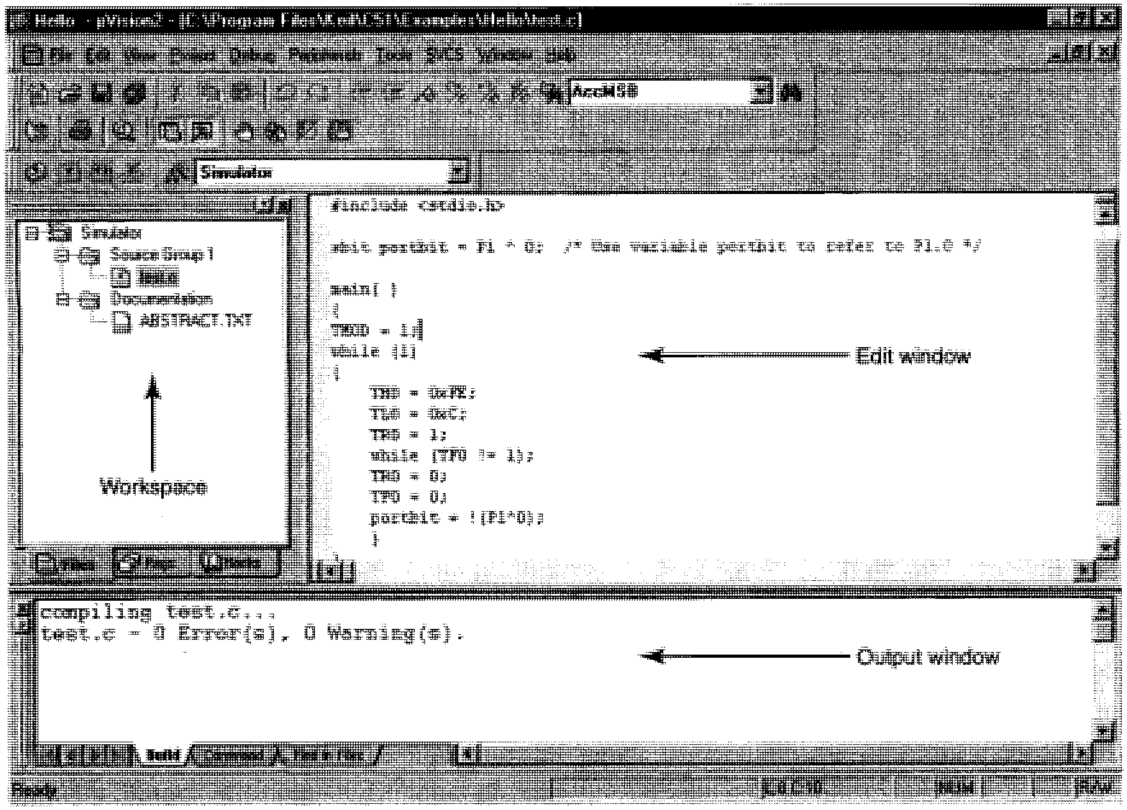
The Workspace is a pane with three tabs that open different windows—Files, Regs, and Books. The Files window (see Figure H-2) allows you to manage the source code files that you want to include in your current project. Right-click the Simulator option and then choose "Select Device for Target 'Simulators" to indicate the target 8051 or derivative device for which you want to write a C program. To add a file to your project, simply right-click the "Source Group 1" option and browse for the desired C file.

If you prefer to create a new program, then select New from the File menu and start keying the code for your program. When you have finished, save it as a C file and add it to your project. In Figure H-2, you can see that the sole file included in the project is test.c. Double-click a file name to open it and it will display in the Edit window, as shown in Figure H-1.

COMPILING AND DEBUGGING

When you have finished editing your program, the next thing you want to do is compile it to see if there are any compile errors. For this, you could select "Build target" from the Project

[1] An evaluation version is available for download at http://www.keil.com/demo/.


Hello - µVision2 - [C:\Program Files\Keil\C51\Examples\Hello\test.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
Simulator
Simulator
Source Group 1
test.c
Documentation
ABSTRACT.TXT
Workspace
Files Regs Books
Edit window
Output window
Build Command Find in Files
Ready
L:8 C:10
NUM
R/W

```
#include <stdio.h>

sbit portbit = P1 ^ 0;  /* Use variable portbit to refer to P1.0 */

main( )
{
TMOD = 1;
while (1)
{
    TH0 = 0xFE;
    TL0 = 0xC;
    TR0 = 1;
    while (TF0 != 1);
    TR0 = 0;
    TF0 = 0;
    portbit = !(P1^0);
    }
```

```
compiling test.c...
test.c - 0 Error(s), 0 Warning(s).
```

FIGURE H-1
µVision2 IDE

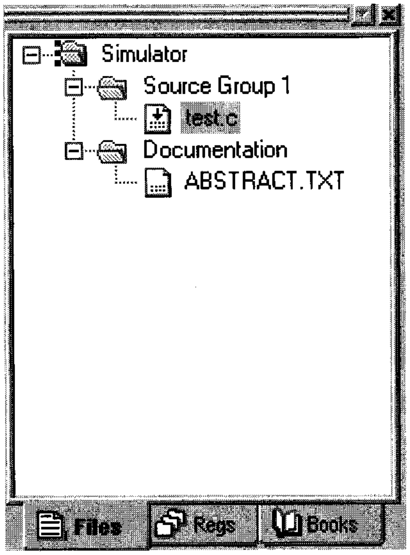


FIGURE H-2
The Files window

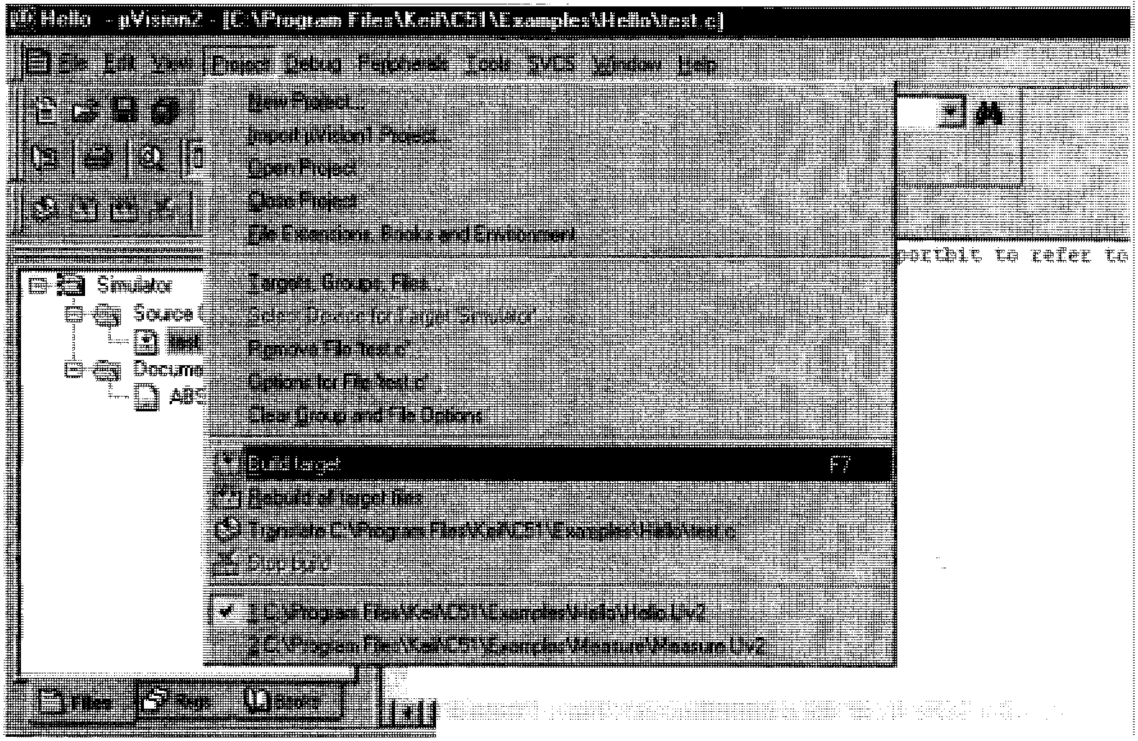

Hello - µVision2 - [C:\Program Files\Keil\C51\Examples\Hello\test.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
New Project...
Import µVision1 Project...
Open Project
Close Project
File Extensions, Books and Environment
Targets, Groups, Files...
Select Device for Target 'Simulator'
Remove File 'test.c'
Options for File 'test.c'
Clear Group and File Options
Build target F7
Rebuild all target files
Translate C:\Program Files\Keil\C51\Examples\Hello\test.c
Stop build
1 C:\Program Files\Keil\C51\Examples\Hello\Hello.Uv2
2 C:\Program Files\Keil\C51\Examples\Measure\Measure.Uv2
Simulator
Source G
test
Docume
ABS
portbit to refer to
Files
Regs
Books

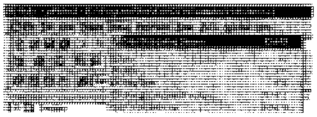
FIGURE H-3
Select "Build target" from the Project menu

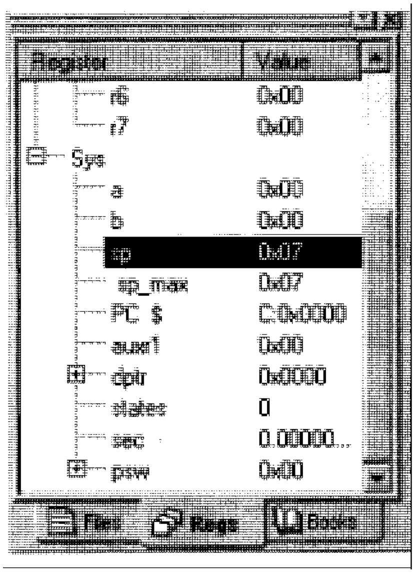
menu (see Figure H-3). This will compile all recently updated files in your project. Alternatively, you could select "Translate ... " to compile only the C file you have currently highlighted, or "Rebuild all target files" to compile all files in your project, regardless of whether or not you have just updated them. The Output tab should display 0 errors.

Before downloading the compiled program into the ROM of your 8051, it is advisable to test and debug it to determine if it executes as desired. Select "Start/Stop Debug Session" from the Debug menu (see Figure H-4). If you want to run the program from beginning to the end, select "Go" from the Debug menu. Otherwise, if you would like to execute one C statement at a time, select the "Step" option.

In the debugging session, as you are stepping through or running your program, you can choose to view the status of registers by clicking on the Regs tab to open the Regs window (see Figure H-5).

FIGURE H-4
Select "Start/Stop Debug Session" from the Debug menu



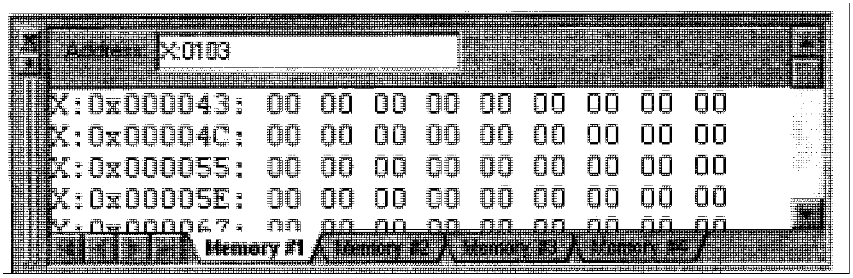


Register	Value
r6	0x00
r7	0x00
Sys	
a	0x00
b	0x00
sp	0x07
sp_max	0x07
PC $	C:0x0000
auxr1	0x00
dptr	0x0000
states	0
sec	0.00000...
psw	0x00

FIGURE H-5
Regs window

Figure H-5 shows the Regs window with the stack pointer (SP) and its contents highlighted. You can also view the memory contents by selecting "Memory Window" from the View menu. The Memory window (see Figure H-6) normally appears at the lower-right corner of the screen. Figure H-6 shows the contents of external memory locations starting from 0103H. If you would rather look at the contents of internal data memory, then click in the text box to the right of "Address:" and key D:0000, where D refers to internal data memory and 0000 specify the starting location you want to view. Keying C instead of D enables you to view code memory. To modify a certain memory location, simply double-click the location and choose "Modify Memory at ... " (see Figure H-7) and key the new value into the edit box that appears. You can enter the values in decimal or in hex, using standard C notation.

Viewing the contents and status of the on-chip peripherals is also easy by selecting them from the Peripherals menu. As an example, Figure H-8 shows how to view the contents of Port 0.



Address: X:0103

```
X:0x000043:  00  00  00  00  00  00  00  00  00
X:0x00004C:  00  00  00  00  00  00  00  00  00
X:0x000055:  00  00  00  00  00  00  00  00  00
X:0x00005E:  00  00  00  00  00  00  00  00  00
X:0x000067:  00  00  00  00  00  00  00  00  00
```

FIGURE H-6
Memory window

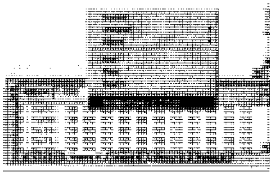


FIGURE H-7
Modifying a memory location

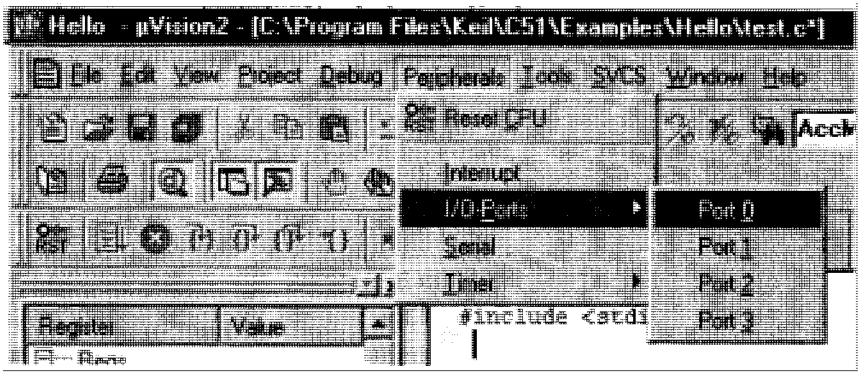
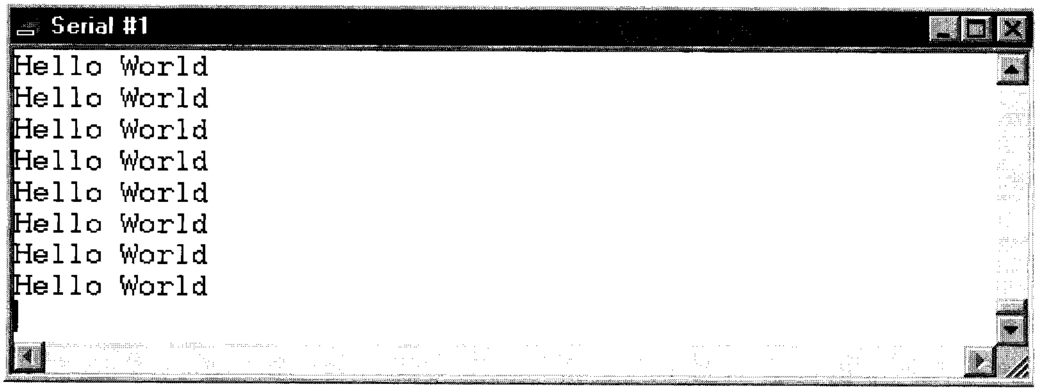


FIGURE H-8
Viewing the contents and status of a peripheral

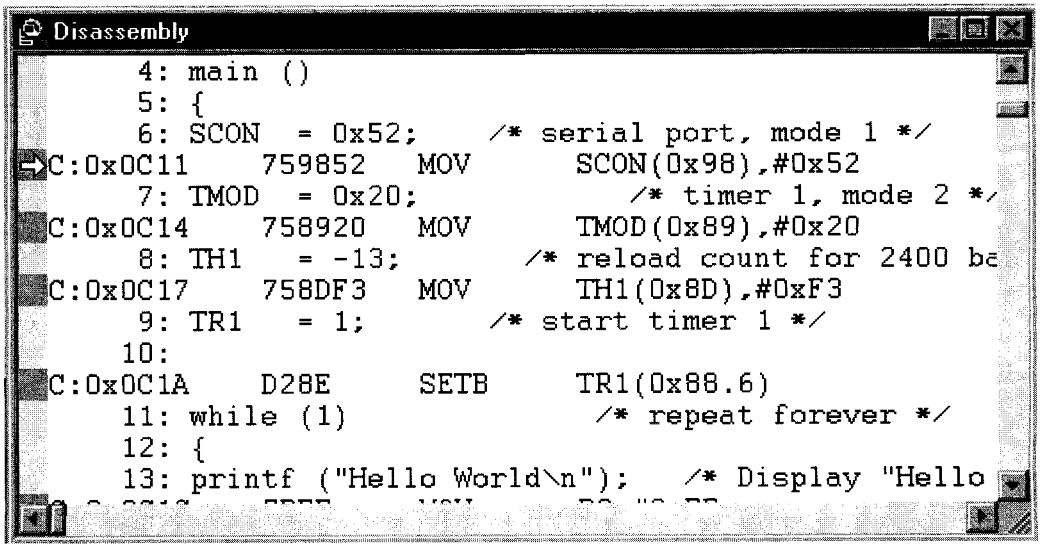
Recall that in the first "Hello World" C program in Chapter 8, mention was made that in µVision2 IDE, the device connected to the 8051's serial port is a simulated serial window by default. By using this serial window, you can view whatever messages that you send with the print f statement to the serial port. To access this serial window, select "Serial Window #1" from the View menu during the debugging session. Figure H-9 shows the serial window when the "Hello World" program in Chapter 8 is run.



```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

FIGURE H-9
Serial window

In the section on timers in Chapter 8, you learned that you could direct the µVision2's C compiler to disassemble your C programs into assembly language. To view the corresponding assembly language version of your program, select "Disassembly Window" from the View menu. Figure H-10 shows the disassembly of our "Hello World" program, with the original C statements shown right above their corresponding assembly language instructions.



```
     4: main ()
     5: {
     6: SCON  = 0x52;    /* serial port, mode 1 */
C:0x0C11    759852   MOV      SCON(0x98),#0x52
     7: TMOD  = 0x20;            /* timer 1, mode 2 */
C:0x0C14    758920   MOV      TMOD(0x89),#0x20
     8: TH1   = -13;      /* reload count for 2400 ba
C:0x0C17    758DF3   MOV      TH1(0x8D),#0xF3
     9: TR1   = 1;       /* start timer 1 */
    10:
C:0x0C1A    D28E     SETB     TR1(0x88.6)
    11: while (1)              /* repeat forever */
    12: {
    13: printf ("Hello World\n");   /* Display "Hello
```

FIGURE H-10
Disassembly window

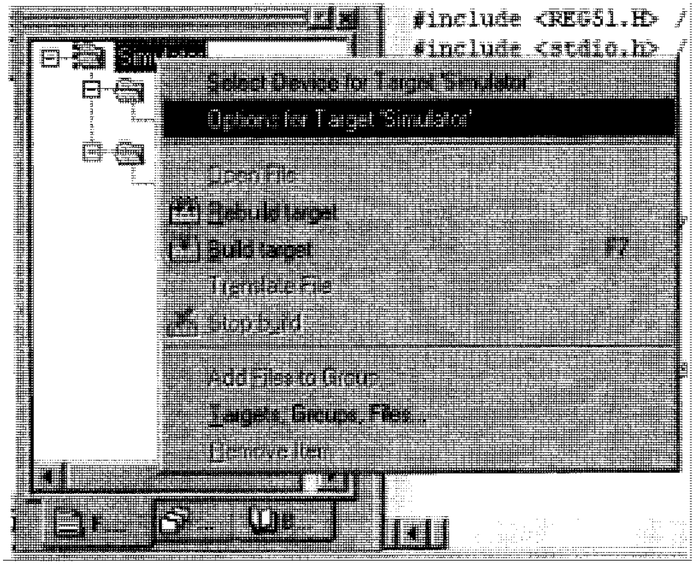
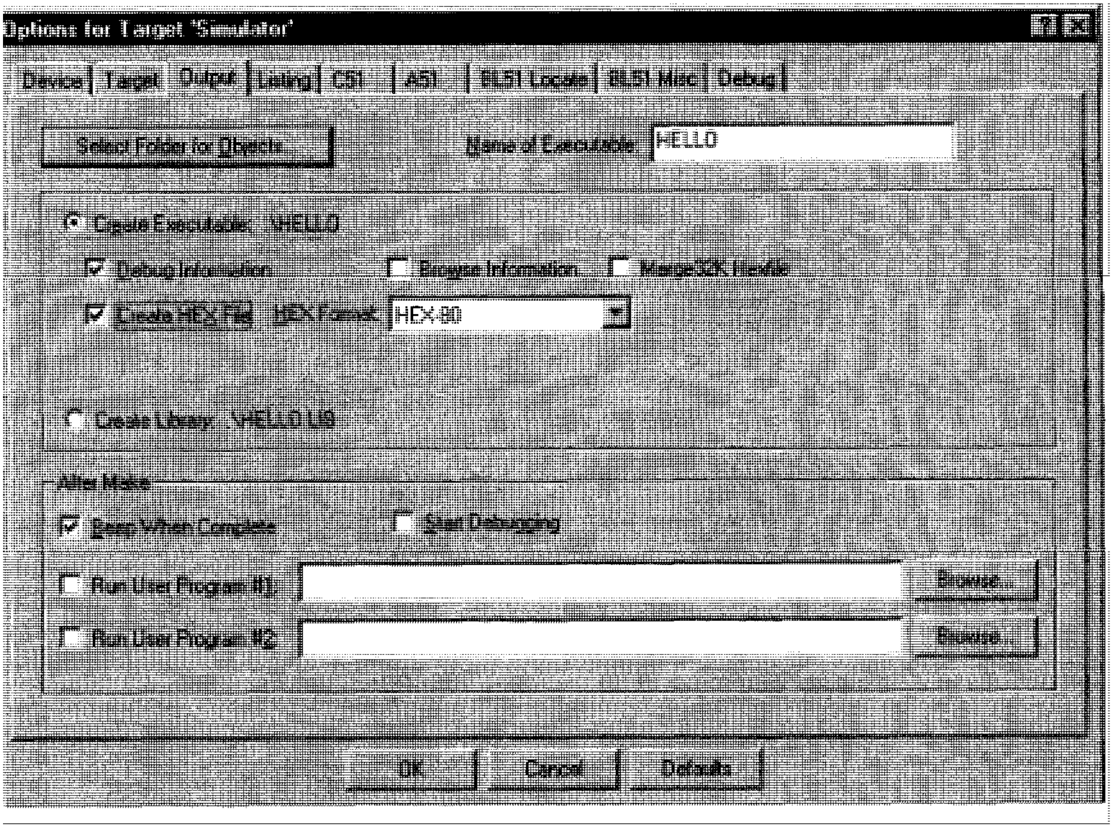


FIGURE H-11
Options for the target



Name of Executable: HELLO

● Create Executable: .\HELLO

☑ Debug Information ☐ Browse Information ☐ Merge32K Hexfile

☑ Create HEX File HEX Format: HEX-80

○ Create Library: .\HELLO.LIB

☑ Beep When Complete ☐ Start Debugging

☐ Run User Program #1:

☐ Run User Program #2:

FIGURE H-12
Output options tab

Once you are satisfied with the debugging and simulation of your program, you can direct the µVision2 to generate the machine code as a HEX file. Right-click the Simulator option in the Files window, and then select "Options for Target ... " (see Figure H-11). Click the Output tab and check the "Create HEX File" option (see Figure H-12).

With this option selected, the next time you build the target, a HEX file would be created in the same directory as your C file. You can then download this HEX file into ROM, and the 8051 will be ready to execute the program the moment you switch it on.

I

A Guide to the 8052 Simulator

INTRODUCTION

The 8052 Simulator for Windows is software that simulates the entire operation of the 8051 or 8052 microcontroller, plus all its registers, internal RAM locations, I/O ports, timers, serial ports, and interrupts. During the initial stages of developing an 8051—or 8052—based system, you can test the assembled or compiled program entirely in software with the help of this simulator, without having to worry about the actual hardware. This appendix provides a brief guide to using the 8052 Simulator.

The 8052 Simulator

The main window of the 8052 Simulator is as shown in Figure I-1. The first thing you need to do is to load an already assembled or compiled program that has been saved in Intel HEX format. Most 8051 assemblers and compilers have an option to save the assembled or compiled output in this format. Figure 1-2 shows how this is done by selecting the "Open Intel-Standard File" from the File menu and then browsing for the Intel HEX file.

The loaded program is now ready to be tested. First, you can choose to view various components of the 8051, such as the internal RAM, I/O ports, special function registers (SFRs), and timers. Select each of these from the View menu, as shown in Figure 1-3.

Figure 1-4 shows the Internal Memory (RAM) window that was selected from the View menu. All the rows of the first column represent the addresses of 16 memory locations, the contents of which are displayed in the 16 columns on the right. For example, the first row consists of the internal memory locations 00H to 0FH, and as can be seen in Figure 1-4, the content of each is 0. To modify any location, click it and then key the new value in hexadecimal form. Alternatively, you can choose to check (set) or leave unchecked (clear) any of the 8 bits in that location.

'By Vault Information Services, 8174 S. Holly PMB 272, Littleton, CO 80122, U.S.A.
E-mail: sim8052@vaultbbs.com.

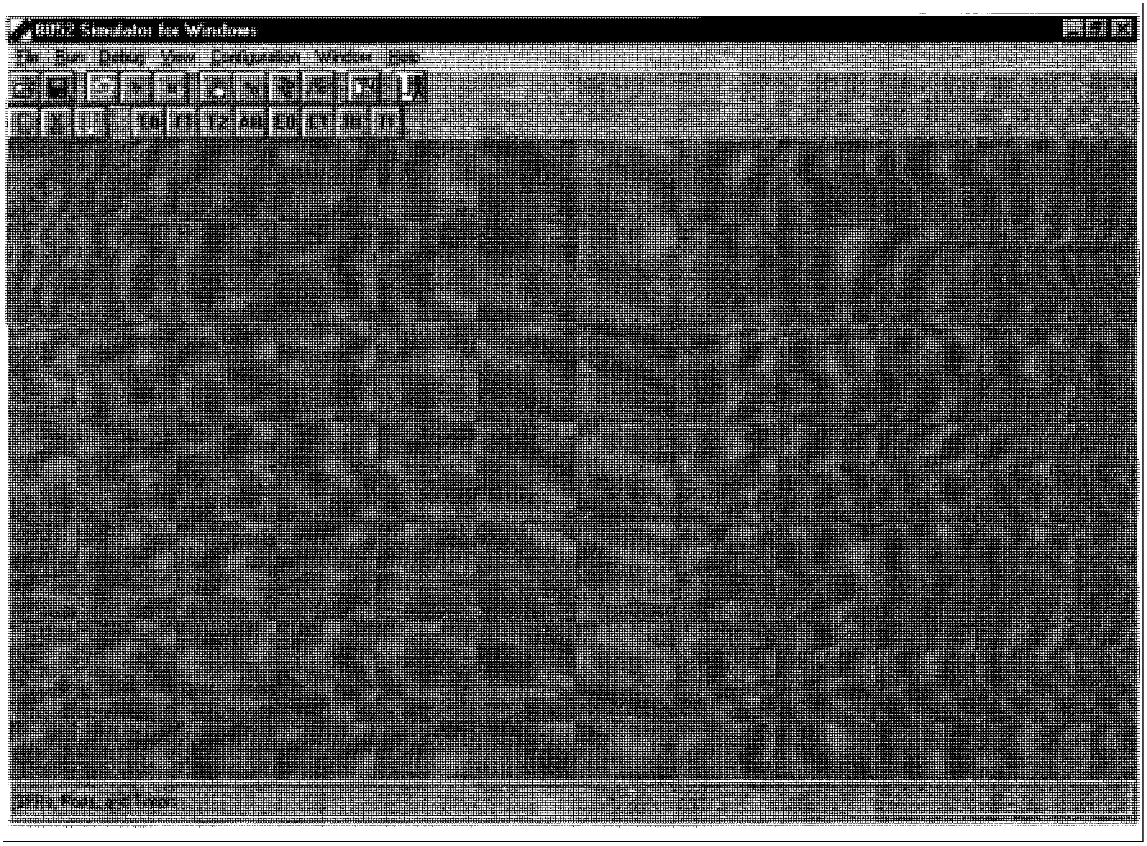

8052 Simulator for Windows
File Run Debug View Configuration Window Help
T0 T1 T2 A/D C0 C1 IN TI
SFRs, Ports and Timers

FIGURE I-1
8052 Simulator

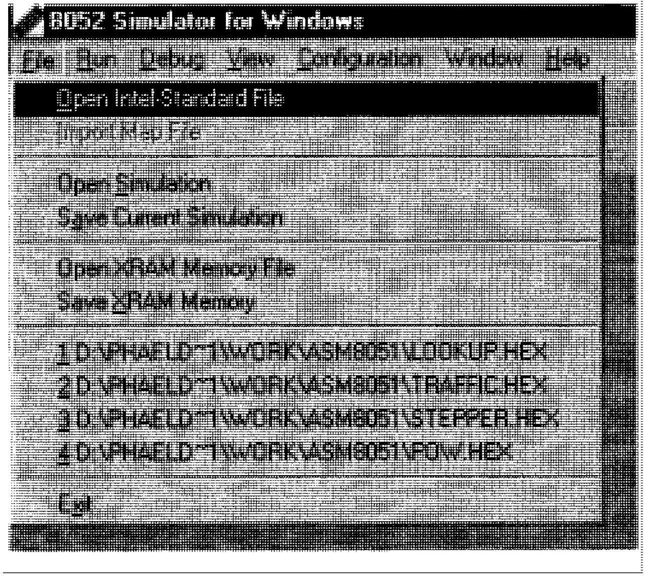

8052 Simulator for Windows
File Run Debug View Configuration Window Help
Open Intel-Standard File
Import Map File
Open Simulation
Save Current Simulation
Open XRAM Memory File
Save XRAM Memory
1 D:\PHAELD~1\WORK\ASM8051\LOOKUP.HEX
2 D:\PHAELD~1\WORK\ASM8051\TRAFFIC.HEX
3 D:\PHAELD~1\WORK\ASM8051\STEPPER.HEX
4 D:\PHAELD~1\WORK\ASM8051\POW.HEX
Exit

FIGURE 1-2
Opening an Intel HEX file

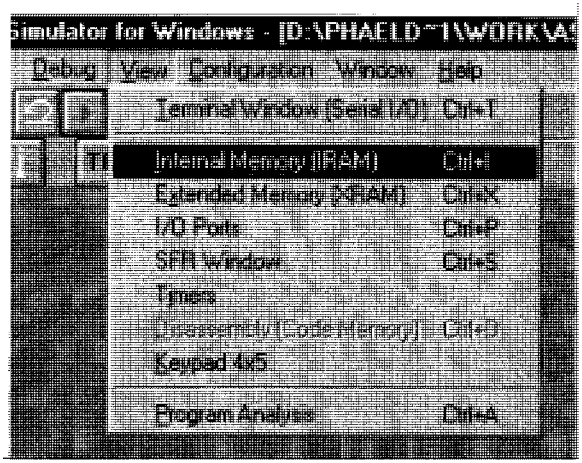
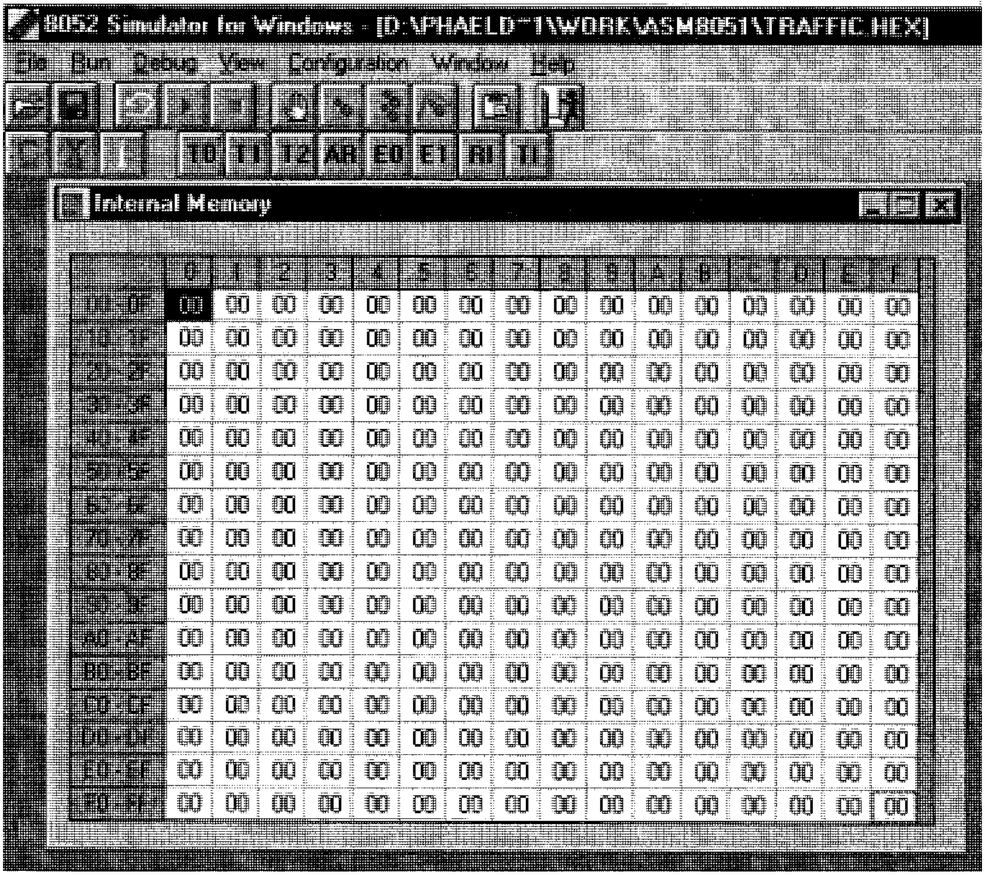
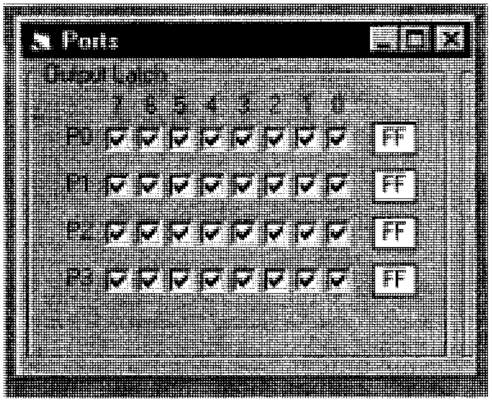

Simulator for Windows - [D:\PHAELD~1\WORK\AS
Debug
View
Configuration
Window
Help
Terminal Window (Serial I/O) Ctrl+T
Internal Memory (IRAM) Ctrl+I
Extended Memory (XRAM) Ctrl+X
I/O Ports Ctrl+P
SFR Window Ctrl+S
Timers
Disassembly (Code Memory) Ctrl+D
Keypad 4x5
Program Analysis Ctrl+A

FIGURE 1-3
Viewing internal components of the 8051



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00 - 0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10 - 1F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20 - 2F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30 - 3F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40 - 4F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50 - 5F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60 - 6F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70 - 7F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80 - 8F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90 - 9F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0 - AF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0 - BF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0 - CF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0 - DF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0 - EF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0 - FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

FIGURE 1-4
Viewing internal RAM

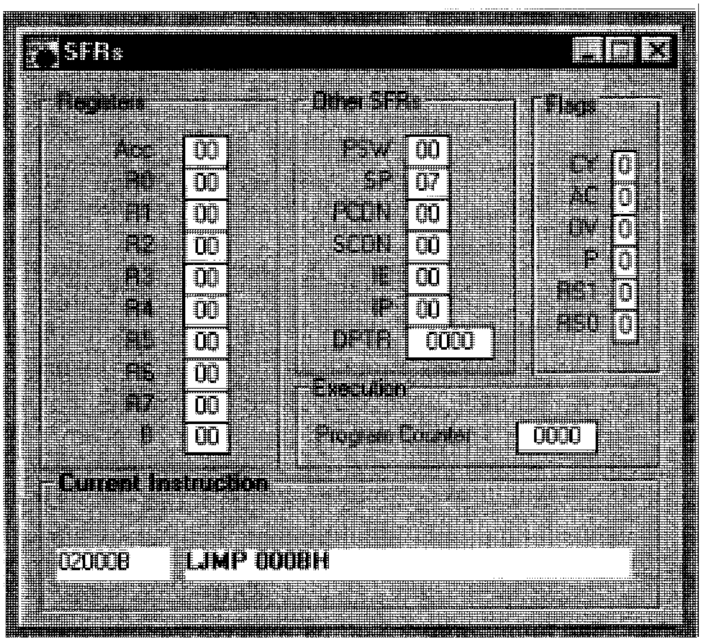


	7	6	5	4	3	2	1	0	
P0	☑	☑	☑	☑	☑	☑	☑	☑	FF
P1	☑	☑	☑	☑	☑	☑	☑	☑	FF
P2	☑	☑	☑	☑	☑	☑	☑	☑	FF
P3	☑	☑	☑	☑	☑	☑	☑	☑	FF

FIGURE 1-5
Viewing I/O ports

You can choose to view the contents of up to 64K of external RAM from the same View menu. Selecting "I/O Ports" from the View menu displays the Ports window (see Figure 1-5), which is straightforward enough. The contents of all four I/O ports are shown in the figure. You can modify their contents using a process similar to that used to modify internal or external RAM.

Figure 1-6 shows the SFRs window, which displays the contents of all the SFRs in the 8051's upper 128 internal RAM locations. Again, their contents can be modified by



Registers
Acc 00
R0 00
R1 00
R2 00
R3 00
R4 00
R5 00
R6 00
R7 00
B 00

Other SFRs
PSW 00
SP 07
PCON 00
SCON 00
IE 00
IP 00
DPTR 0000

Flags
CY 0
AC 0
OV 0
P 0
RS1 0
RS0 0

Execution
Program Counter 0000

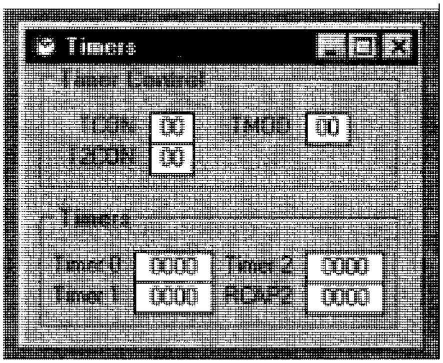
Current Instruction
020000 LJMP 0000H

FIGURE 1-6
Viewing SFRs

clicking on the white boxes and then keying the new value in hex. Notice also that at the bottom of the SFRs window is the "Current Instruction" section. The box at left in this area shows the location in code memory of the next instruction to be executed while the right box shows that next instruction.

Also available for viewing is the Timers window (see Figure I-7), which shows the contents of the TCON and TMOD registers as well as the timer count values of Timers 0, 1, and 2 (for the 8052).

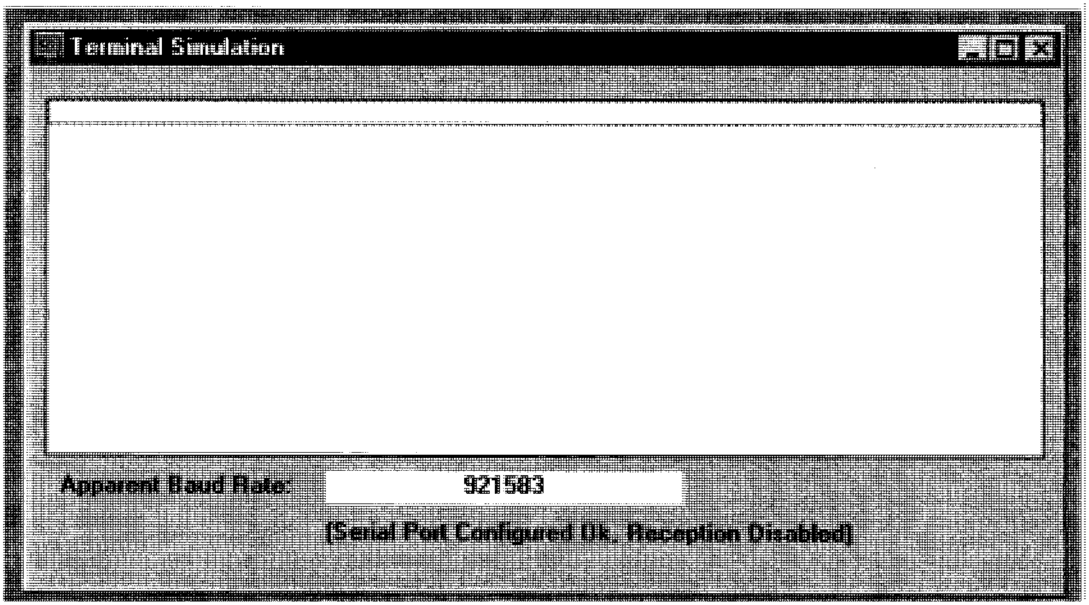
The serial port is also simulated by a terminal window (see Figure I-8) that displays whatever characters are sent to the serial port. The full version of the 8052 Simulator also



TCON 00 TMOD 00
T2CON 00

Timer 0 0000 Timer 2 0000
Timer 1 0000 RCAP2 0000

FIGURE 1-7
Viewing timers



Apparent Baud Rate: 921583

[Serial Port Configured Ok, Reception Disabled]

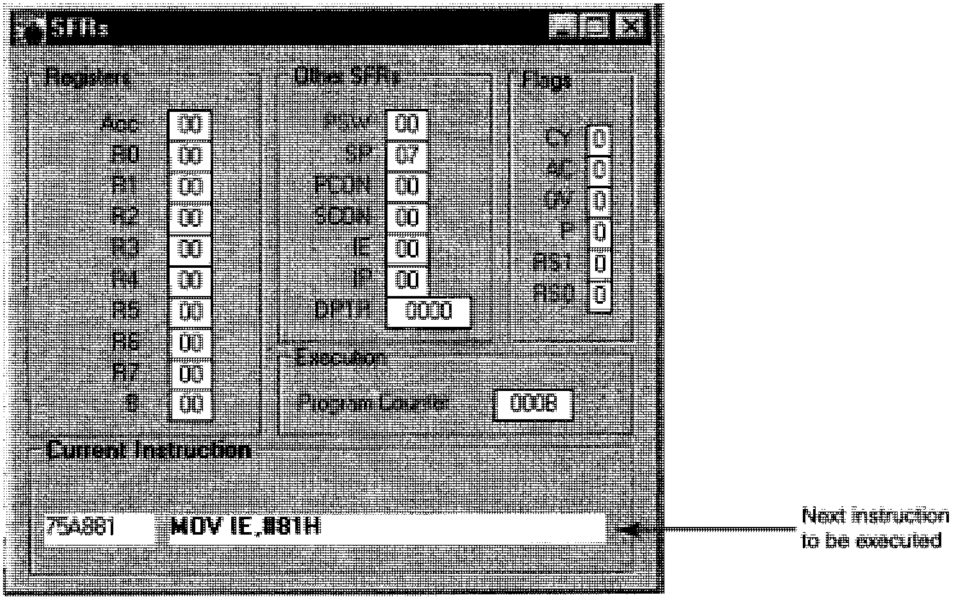
FIGURE 1-8
Viewing the terminal window

supports connection of the 8051's serial port to the computer's COMM/serial port, so that any data sent to or received from the 8051 serial port would be sent to or received from the computer's COMM/serial port.

Let's assume that the HEX file corresponding to the assembly language program in Figure 11-33 has been loaded. This is the program for the pedestrian traffic light system. To test execute the program, first display the relevant windows using the View menu. For example, you could display the internal RAM, I/O ports, and SFRs windows. Next, you could choose to run the entire program right by choosing the "Execute Program" from the Run menu. You can stop the program execution at any time by selecting "Stop" from the same menu. Alternatively, you could execute each instruction one by one by choosing "Single Step" from the Debug menu, or use F8, the shortcut key for the option that will allow you to run each step separately.

For example, the first instruction in the traffic light program is to jump to MAIN. Press F8 once to execute this instruction, and then press F8 again to execute the next instruction, which is to move the value 81H into the IE register (see Figure 1-9). You will see that the content displayed in the IE register's is now 81H. In a similar way, you can step through the rest of the program one step at a time, checking the suitable SFR, internal RAM locations, I/O ports, or timers to see if they are changed as expected. To reset the program to the beginning, simply choose "Reset Program" from the Run menu.

At any time during the program execution or as you execute it one step at a time, you can select "Program Analysis" from the View menu to display useful information such as the number of instructions so far executed and how many machine cycles have been used.



SFRs

Registers

Acc	00
R0	00
R1	00
R2	00
R3	00
R4	00
R5	00
R6	00
R7	00
B	00

Other SFRs

PSW	00
SP	07
PCON	00
SCON	00
IE	00
IP	00
DPTR	0000

Flags

CY	0
AC	0
OV	0
P	0
RS1	0
RS0	0

Execution

Program Counter 0008

Current Instruction

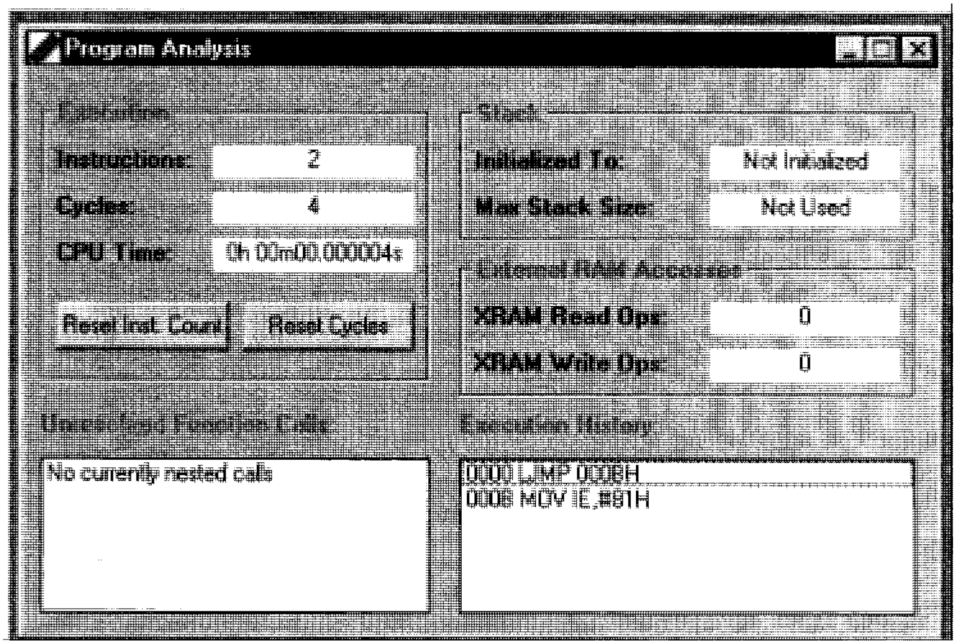
75A881 MOV IE,#81H

Next instruction to be executed

FIGURE 1-9
Checking the program one step at a time

There is also an Execution History area in the Program Analysis window that lists all previously executed instructions (see Figure I-10.)

An added feature in the 8052 Simulator is the 4 x 5 keypad simulator (see Figure I-11). This simulates the functions of a 4 x 5 keypad, similar to the 4 x 4 hexadecimal keypad discussed in Chapter 11. You can customize the names of any key simply by right-clicking the key you want to change and then keying the new name.



Instructions: 2

Cycles: 4

CPU Time: 0h 00m00.000004s

Initialized To: Not Initialized

Max Stack Size: Not Used

XRAM Read Ops: 0

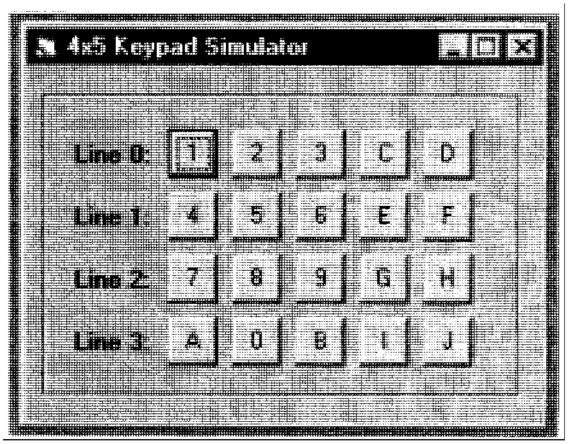
XRAM Write Ops: 0

No currently nested calls

```
0000 LJMP 0008H
0008 MOV IE,#81H
```

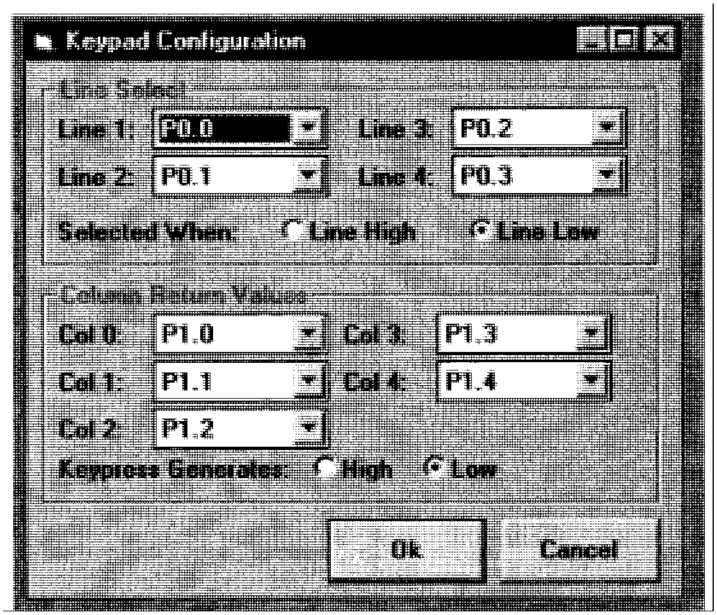
FIGURE 1-10
The Program Analysis window

FIGURE I-11
The keypad simulator



Line 0:	1	2	3	C	D
Line 1:	4	5	6	E	F
Line 2:	7	8	9	G	H
Line 3:	A	0	B	I	J

You can also customize which port pins will be used to connect the rows and columns by selecting "Keypad Configuration" from the Configuration menu. This is shown in Figure 1-12.



Line Select

Line 1: P0.0
Line 3: P0.2
Line 2: P0.1
Line 4: P0.3

Selected When: ○ Line High ● Line Low

Column Return Values

Col 0: P1.0
Col 3: P1.3
Col 1: P1.1
Col 4: P1.4
Col 2: P1.2

Keypress Generates: ○ High ● Low

FIGURE 1-12
The Keypad Configuration window

J

The Advanced Encryption Standard

INTRODUCTION

As the 8051 is used more and more in smart cards, where the security of information is almost always important, it is most often used to execute software or hardware implementations of encryption methods. The current standard for encryption is the Advanced Encryption Standard (AES), a recent and secure method that was officially adopted in 2000 by the U.S. National Institute of Standards & Technology (NIST) for encrypting confidential, nonclassified information in the United States. It is expected that the AES will replace the Data Encryption Standard in ATM cards, smart cards, online transactions, and other security-related applications. Therefore, the study of the 8051 would not be complete without a discussion of the AES. This appendix is intended to acquaint the reader on how the AES works so that you will understand how the AES can be implemented with the 8051.

BLOCK ENCRYPTION

These days, messages and information are digitally stored in files in the computer in terms of bits of 1 s and 0s. These messages vary in size from a few bytes to a few hundred megabytes. For that reason, current encryption methods specify a standard input block size, typically 128 bits. A message that is to be encrypted first must be broken down into blocks of 128 bits, and then each block is encrypted. The encrypted blocks are then concatenated to form the encrypted message. See Figure J-1 for an illustration of this process.

HOW THE AES WORKS

For the purpose of describing the AES, its designers represented the 128-bit input block as a 4 x 4 array of bytes. An element of this array is denoted by a_1, where i and j are the row and column indices respectively, numbered from 0 to 3, as shown in Figure J-2.

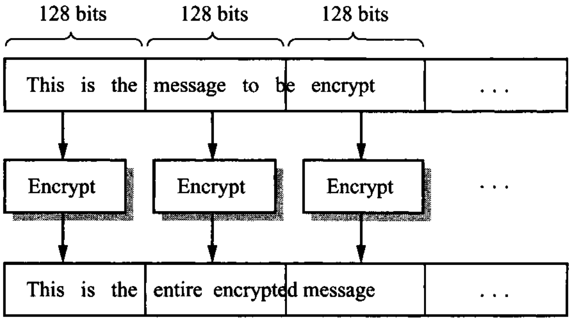


FIGURE J-1
Encrypting a message one block at a time

In order to use the AES for encrypting the 128-bit block, the user is prompted for a password, called the **secret key.** This key could be either 128, 192, or 256 bits. For ease of description in this example, we will henceforth concentrate on AES with a 128-bit key. Details of using the AES with the other two key versions are similar.

This 128-bit secret key is put through a series of operations, collectively called the **key schedule.** The purpose is to use the secret key to generate a total of 11 different keys of also 128 bits in size, each of which is called a **round key.**

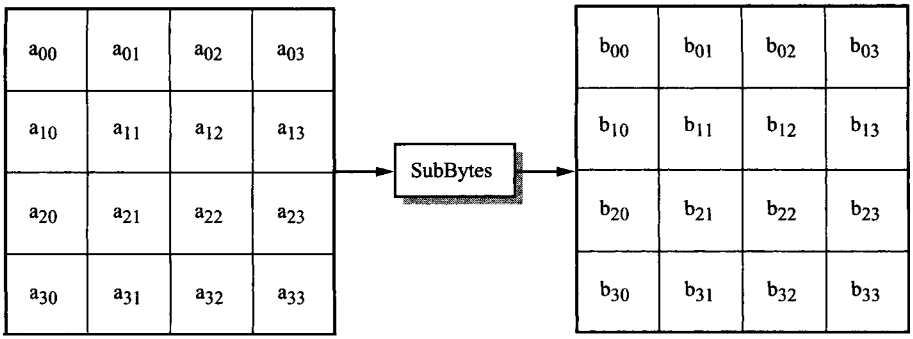
Now let's look at how the AES encryption works. Before we start, remember that even though there are numerous specific names involved, the AES is designed to be suitable for implementation on 8-bit processors and microcontrollers such as the 8051, so its operations are simple and straightforward.

The AES encryption basically involves iterating an operation, called a **round,** for a total of 10 times. Within each round operation are the following 4 smaller operations: **SubBytes, ShiftRows, MixColumns** and **AddRoundKey.**

The SubBytes operation is based on a table, called the **substitution box** or s-box. Each byte, of the 4 x 4 array previously shown in Figure J-2 is replaced with an entirely different byte, b_1. How is this b_1 obtained? The original a_1 is used as an index to refer to a corresponding element in the s-box table. That element is b_1. Figure J-3 shows the SubBytes operation.

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

FIGURE J-2
Representing a 128-bit block of the AES



a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

b_{00}	b_{01}	b_{02}	b_{03}
b_{10}	b_{11}	b_{12}	b_{13}
b_{20}	b_{21}	b_{22}	b_{23}
b_{30}	b_{31}	b_{32}	b_{33}

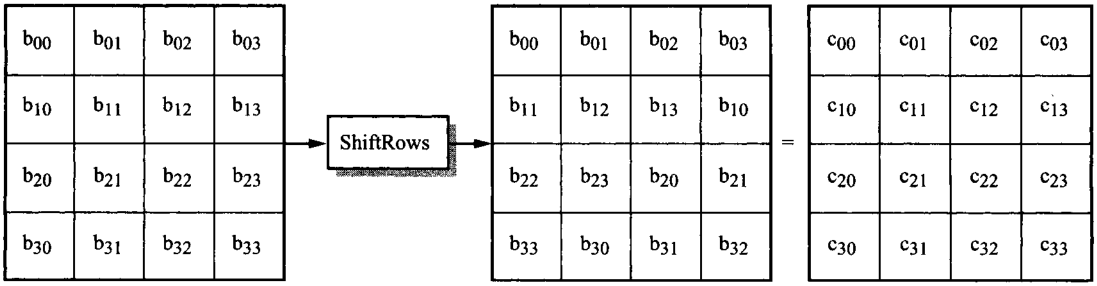
FIGURE J-3
The SubBytes operation

The ShiftRows operation is even simpler. In the 4 x 4 array, the first row is left unchanged. The second row is rotated left by one byte (8 bits), the third row is rotated left by 2 bytes, and the fourth row is rotated left by 3 bytes (see Figure J-4).

The MixColumns operation is a bit more involved, and we won't go into the technical details here. The interested reader is referred to the AES documentation in the Bibliography for more details. Simply explained, this operation replaces each column with an entirely different column based on a series of operations that can be efficiently implemented on the 8051. Figure J-5 illustrates the MixColumns operation, where the shaded column shows that the first column at the input is replaced with a new column appearing also in the first column at the output.

The AddRoundKey operation takes all 128 bits of the 4 x 4 array and exclusiveORs them with a 128-bit round key that was generated from the secret key, as previously mentioned.

We have described all the operations within the AES. To summarize, the AES encryption takes a 128-bit input block, and puts it through 10 rounds of operations, each of which consists of the SubBytes, ShiftRows, MixColumns, and AddRoundKey operations applied in

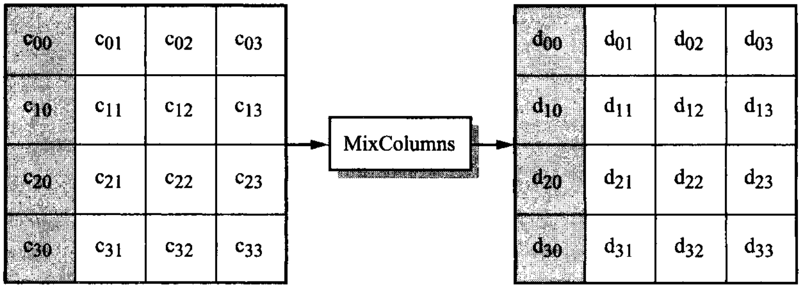


b_{00}	b_{01}	b_{02}	b_{03}
b_{10}	b_{11}	b_{12}	b_{13}
b_{20}	b_{21}	b_{22}	b_{23}
b_{30}	b_{31}	b_{32}	b_{33}

b_{00}	b_{01}	b_{02}	b_{03}
b_{11}	b_{12}	b_{13}	b_{10}
b_{22}	b_{23}	b_{20}	b_{21}
b_{33}	b_{30}	b_{31}	b_{32}

c_{00}	c_{01}	c_{02}	c_{03}
c_{10}	c_{11}	c_{12}	c_{13}
c_{20}	c_{21}	c_{22}	c_{23}
c_{30}	c_{31}	c_{32}	c_{33}

FIGURE J-4
The ShiftRows operation



c_{00}	c_{01}	c_{02}	c_{03}
c_{10}	c_{11}	c_{12}	c_{13}
c_{20}	c_{21}	c_{22}	c_{23}
c_{30}	c_{31}	c_{32}	c_{33}

d_{00}	d_{01}	d_{02}	d_{03}
d_{10}	d_{11}	d_{12}	d_{13}
d_{20}	d_{21}	d_{22}	d_{23}
d_{30}	d_{31}	d_{32}	d_{33}

FIGURE J-5
The MixColumns operation

sequence. However, an extra AddRoundKey operation is added prior to the first round, and the MixColumns operation is omitted from the last round. These slight adjustments are design choices to make encryption and decryption similar in structure. This similarity is desired because it allows for more compact hardware and software implementations. An illustration of the entire AES encryption process is shown in Figure J-6.

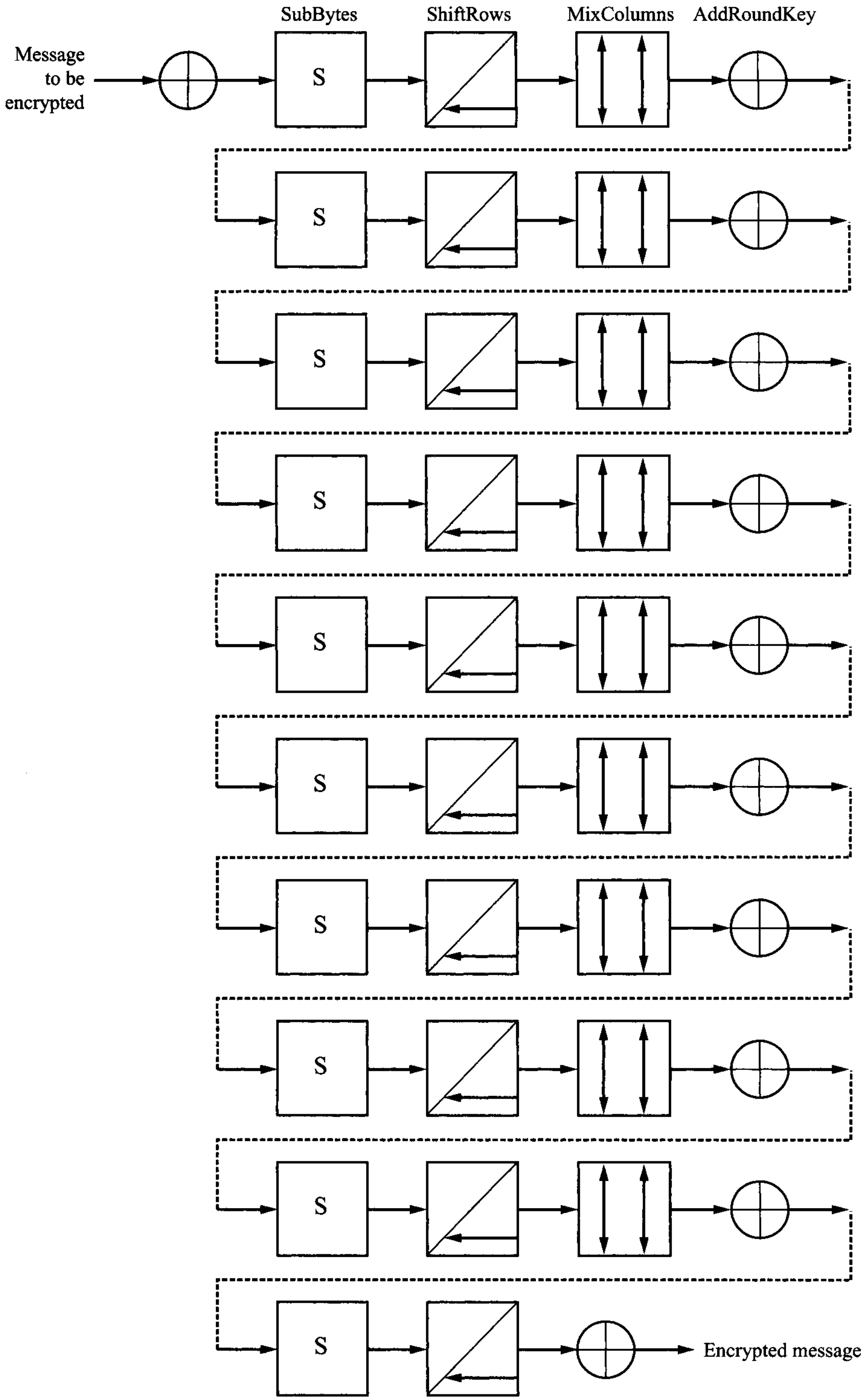


FIGURE J-6
The AES encryption

K

Sources of 8051 Development Products

Allen Systems, 2151 Fairfax Road, Columbus, OH 43221

Product:	FX-31 8052-BASIC SBC
Description:	8052 single-board computer with built-in BASIC interpreter
Product:	CA-51
Description:	8051 cross assembler
Host Computer:	IBM *PC* and compatibles
Product:	DP-31/535
Description:	Single-board computer based on Siemens 80535 CPU

Applied Microsystems Corp., 5020 148th Ave. N. E., P.O. Box 97002, Redmond, WA 98073-9702

Product:	EC7000
Description:	8051 microcontroller emulator
Host Computer:	IBM *PC* and compatibles

Aprotek, 1071-A Avenida Acaso, Camarillo, CA 93010

Product:	PA8751
Description:	8751 programming adapter
Features:	Adapts 8751 EPROM microcontrollers to any EPROM programmer as a 2732

Avoset Systems, Inc., 804 South State St., Dover, DE 19901

Product:	XASM51
Description:	8051 cross assembler
Host Computer:	IBM *PC* and compatibles
Product:	AVSIM51
Description:	8051 family simulator
Host Computer:	IBM *PC* and compatibles

Binary Technology, Inc., B.O. Box 67, Meridan, NH 03770

Product:	SIBEC-Il
Description:	8052 single-board computer with built-in BASIC interpreter

Cybernetic Micro Systems, Inc., P.O. Box 3000, San Gregorio, CA 94074

Product:	CYS8051
Description:	8051 cross assembler
Host Computer:	IBM *PC* and compatibles
Features:	Does not generate relocatable modules
Product:	SIM8051
Description:	8051 simulator
Host Computer:	IBM *PC* and compatibles
Product:	CYS8051
Description:	EPROM programmer for 8751
Interface:	RS232 serial

Decmation, Inc., 3375 Scott Blvd., Suite #236, Santa Clara, CA 95054

Product:	ASM51, PLM51 SIM51, etc.
Description:	8051 development software
Host Computer:	VAX, PDP-11, IBM PC and compatibles

HiTech Equipment Corp., 9560 Black Mountain Road, San Diego, CA 92126

Product: 8051 SIM
Description: 8051 simulator
Host Computer: IBM *PC/XT* or Z80 CP/M microcomputers

Huntsville Microsystems, Inc., P.O. Box 12415, 4040 South Memorial Parkway, Huntsville, AL 35802

Product: SBE-31, HMI-200-8051
Description: 8051 emulator
Interface: Serial
Host Computer: Various computers with CP/M or MS-DOS operating system

Keil Software, Inc., 1501 10" Street, Suite 110, Plano, TX 75074

Product: *μ*Vision2 IDE
Description: 8051 C compiler and debugger
Host Computer: IBM *PC* and compatibles

Logical Systems Corp., 6184 Beall Station, Syracuse, NY 13217

Product: UPA8751
Description: 8751 programming adaptor
Features: Adapts 8751 to 2732 socket for installation into any EPROM programmer
Product: SIM51
Description: 8051 simulator and debugger
Host Computer: IBM *PC* (MS-DOS, CP/M) and compatibles

Micromint, Inc., 4 Park Street, Vernon, CT 06066

Product: BCC52
Description: 8052 single-board computer with built-in interpreter

Nohau Corp., 51 East Campbell Ave., Suite 107E, Campbell, CA 95008

Product: EMV51-PC

Description: 8051 emulator

Host Computer: IBM *PC* and compatibles

Relational Memory Systems, Inc., P.O. Box 6719, San Jose, CA 95150

Product: ASM51 RLINK, RLOC, GLIB, OBJCON

Description: 8051 software development tools

Host Computer: IBM *PC* and compatibles

Scientific Engineering Laboratories, Inc., 104 Charles Street, Suite 143, Boston, MA 02114

Product: XPAS51

Description: 8051 PASCAL cross compiler

Host Computer: IBM *PC* and compatibles

Single Board Systems, B.O. Box 3788, Salem, OR 97306

Product: SBS-52

Description: 8052 single-board computer with built-in BASIC interpreter

Software Development Systems, Inc., 3110 Woodstock Drive, Downers Grove, IL 60515

Product: A51

Description: 8051 cross assembler

Host Computer: Various systems running MS-DOS, Xenix, or Unix

Universal Cross Assemblers, P.O. Box 384, Bedford, Nova Scotia B4A 2X3, Canada

Product: CROSS-16

Description: 8051 cross assembler

Host Computer: IBM *PC* and compatibles

URDA, Inc., 1811 Jancey St., Suite #200, Pittsburgh, PA 15206

Product:	SBC-51
Description:	PC board version of SBC-51 described in this text

Z-World, 2065 Martin Ave. #110, Santa Clara, CA 95050

Product:	IBM PC co-processor
Description:	Co-processor plugs into *PC* or *PC/AT;* runs Intel's ISIS operating system and software development tools

IC MANUFACTURERS

The following companies are manufacturers and/or developers of the 8051 and derivative ICs.

Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051
Siemens Components, Inc., 2191 Laurelwood Road, Santa Clara, CA 95054
Signetics/Philips, 811 East Argues Ave., Sunnyvale, CA 94088-3409
Advanced Micro Devices, Inc., 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088-3453
Fujitsu Microelectronics, Inc., 3320 Scott Blvd., Santa Clara, CA 95054-3197

BIBLIOGRAPHY

ARTICLES

Ball, S. Embedded debugging tricks. *Circuit Cellar* (1995, September): 20-23.

Boyet, H., and R. Katz. The 8051 one-chip microcomputer. *Byte* (1981, December): 288-311.

Cantrell, T. Audio processor chips for the masses. *Circuit Cellar* (1995, November): 70-76.

Cantrell, T. Chip on patrol. *Circuit Cellar* (1995, June): 64-71.

Cantrell, T. Plan '251 for outer space!: Intel's 8xC251SB. *Circuit Cellar* (1995, March): 72-77.

Cantrell, T. UFO alert. *Circuit Cellar* (1995, January): 100-107.

Cheung, H. DRAM on an 8031: It's not as hard as you'd think. *Circuit Cellar* (1994, September): 24-31.

Ciarcia, S. Build the BASIC-52 computer/controller. *Byte* (1985, August): 105-117.

Ciarcia, S. Build a gray-scale video digitizer—Part 1: Display/receiver. *Byte* (1987, May): 95-106.

Ciarcia, S. Build a gray-scale video digitizer—Part 2: *Byte* (1987, June): 129-138.

Ciarcia, S. Build an intelligent serial EPROM programmer. *Byte* (1986, October): 103-119.

Ciarcia, S. Build a trainable infrared master controller. *Byte* (1987, March): 113-123.

Ciarcia, S. Why microcontrollers?—Part 1. *Byte* (1988, August): 239-245.

Ciarcia, S. Why microcontrollers?—Part 2. *Byte* (1988, September): 303-323.

Collier, M., and F. Gweme. Preventing the ultimate blow: A portable checking unit for 8751s. *Circuit Cellar* (1994, September): 48-43.

Dinwiddle, G. An 8031 in-circuit emulator. *Byte* (1986, July): 181-194.

Dybowski, J. Atmel's AT29C2051 flash-based microcontroller. *Circuit Cellar* (1995, February): 76-80.

Dybowski, J. Beef up the 8052 with the DS87C520. *Circuit Cellar,* no. 52 (1994, November): 76-82.

Dybowski, J. Embedded development. *Circuit Cellar* (1995, March): 78-84.

Messick, P., and J. Battle. Build a MIDI input for your Casio SK-1 *Keyboard* (1987, August): 34-40.

Natarajan, K. S., and C. Eswarn Design of a CCITT V.22 modem. *Microprocessors and Microsystems* 12(9) (1988, November): 532-535.

National Institute of Standards and Technology (NIST). *Advanced Encryption Standard, Federal Information Processing Standard (FIPS) 197*. (2002). Online: http://csrc.nist.gov/encryption/aes.

Schenker, J. Controller chip cuts keyboard redesign to weeks. *Electronics* 61(2) (1988, January 21): 42E-42F.

Vaidya, D. M. Microsystem design with the 8052-BASIC microcontroller. *Microprocessors and Microsystems* 9(8) (1985, October): 405-411.

Vaidya, D. M. Software development for the 8052-BASIC microcontroller. *Microprocessors and Microsystems* 9(10) (1985, December): 481-485.

Wallace, H. No emulator? Try a one-wire debugger. *Circuit Cellar* (1995, January): 20-23.

Warner, W. Use a single-chip microprocessor as the heart of your position controller *EDN* (1988, Sept. 1) 161-168.

Yeskey, D. In-circuit 8051 emulator is a tool for the masses. *Electronic Design* (1988, January 7): 202.

BOOKS

8-bit Embedded Controllers (270645). Santa Clara, CA: Intel, 1991.

Ayala, K. J. *The 8051 Microcontroller: Architecture, Programming, and Applications.* New York: West, 1991.

Barnett, R. H. *The 8051 Family of Microcontrollers.* New Jersey: Prentice Hall, 1995.

Boyet, H., and R. Katz. *The 8051: Programming Interfacing Applications.* New York: MTI Publications, 1981.

Cx51 Compiler: Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers. Keil Software, 2001.

Getting Started with µVision2 and the C51 Microcontroller Development Tools. Keil Software, 2001.

Mazidi, M. A., and J. G. Mazidi. *The 8051 Microcontroller and Embedded Systems.* New Jersey: Prentice Hall, 2000.

MCS-51 Macro Assembler User's Guide (9800937-03). Santa Clara, CA: Intel, 1983.

Schultz, T. C and the 8051: *Hardware, Modular Programming, and Multitasking.* New Jersey: Prentice Hall, 1998.

Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* New York: John Wiley & Sons, 1996.

Stallings, W. *Cryptography and Network Security: Principles and Practice. New Jersey:* Prentice Hall, 1999.

Stewart, J. W., and K. X. Miao. *The 8051 Microcontroller: Hardware, Software and Interfacing.* New Jersey: Prentice Hall, 1999.

Yeralan, S., and A. Ahluwalia. *Programming and Interfacing the 8051 Microcontroller.* Massachusetts: Prentice Hall, 1995.

INDEX