The 8051 Microcontroller

Third Edition

I. Scott MacKenzie

Library of Congress Cataloging-in-Publication Data
MacKenzie, I. Scott.
The 8051 microcontroller / I. Scott MacKenzie—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-02-373660-7
1. Intel 8051 (Computer)—Programming. 2. Digital control systems. I. Title. II. Title: Eight thousand fifty-one microcontroller.
QA76.8.I27M23 1995
004.165—dc20 94-8278
CIP

Cover photo: Lester Lefkowitz/Tony Stone Worldwide
Editor: Dave Garza
Production Editor: Stephen C. Robb
Cover Designer: Julia Z. Van Hook
Production Manager: Pamela D. Bennett

This book was set in Times Roman and Helvetica Bold Condensed by Compset, Inc. and was printed and bound by RR Donnelley & Sons Company. The cover was printed by Phoenix Color.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-02-373660-7

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S. A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

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 or computer technology, 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* has been the basis of a fifth semester course for college students in computer engineering. As detailed in Chapter 10, students build 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.

Since 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).

It is my 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.) which 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 over-emphasized. 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).

Some advanced topics relating to programming methods, style, and the development environment are presented in Chapters 8 and 9. These chapters address larger, more conceptual topics important in professional development environments.

Chapter 10 presents several design examples incorporating selected hardware with supporting software. The software is fully annotated and is the real focus in these examples. The second edition includes two additional interfaces; a digital-to-analog output interface using an MC1408 8-bit DAC, and an analog-to-digital input interface using an ADC0804 8-bit ADC. One of the designs in Chapter 10 is the SBC-51—the 8051 single-board computer. The SBC-51 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-51 after programs have been downloaded for execution.

Many dozens of students have wire-wrapped prototype versions of the SBC-51 during the years that I have taught 8051-based courses to computer engineering students. Shortly after the release of the first edition of this text, URDA, Inc. (Pittsburgh, Pennsylvania) began manufacturing and marketing a PC-board version of the SBC-51. This has proven to be a cost-effective solution to implementing a complete lecture-plus-lab package for teaching the 8051 microcontroller to technology students. Contact URDA at 1-800-338-0517 for more information.

Finally, each chapter contains questions further exploring the concepts presented. This new edition includes 128 end-of-chapter questions—almost double the number in the first edition. A solutions manual is available to instructors from the publisher.

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. I also thank the following persons who reviewed the manuscript and offered invaluable comments, criticism, and suggestions: Antony Alumkal, Austin Community College; Omer Farook, Purdue University—Calumet; David Jones, Lenoir Community College; Roy Seigel, DeVry Institute; and Chandra Sekhar, Purdue University—Calumet.

I. Scott MacKenzie

CONTENTS

APPENDIXES

1

INTRODUCTION TO MICROCONTROLLERS

1.1 INTRODUCTION

Although computers have only been with us for 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 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; 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 tools and toys without the microprocessor. Yet this single-chip wonder has barely reached its twentieth 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.

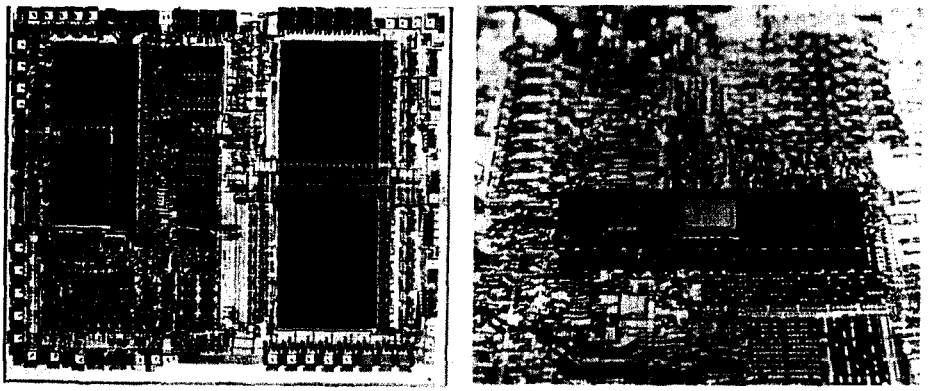
A device similar to the microprocessor is the microcontroller. In 1976 Intel introduced the 8748, the first device in the MCS-48™ family of microcontrollers. Within a single integrated circuit containing over 17,000 transistors, the 8748 delivered a CPU,

1K byte of EPROM, 64 bytes of RAM, 27 I/O pins, and an 8-bit timer. This IC, and other MCS-48™ 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™ 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™ family, and today variations exist virtually doubling these specifications. Siemens Corporation, a second source for MCS-51™ components, offers the SAB80515, an enhanced 8051 in a 68-pin package with six 8-bit I/O ports, 13 interrupt sources, and an 8-bit A/D converter with 8 input channels. 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™ family of microcontrollers. The following chapters introduce the hardware and software architecture of the MCS-51™ family, and demonstrate through numerous design examples how this family of devices can participate in electronic designs with a minimum of additional components.

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



(a) (b)

FIGURE 1–1
The 8051 microcontroller. (a) An 8051 die. (b) An 8751 EPROM. (Courtesy Intel Corp. Copyright 1991.)

(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.

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

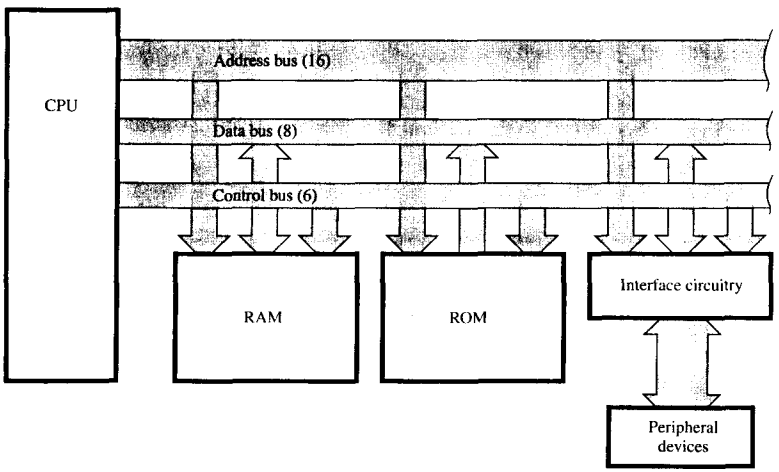


FIGURE 1–2
Block diagram of a microcomputer system

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 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 opcode 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 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 "incre-

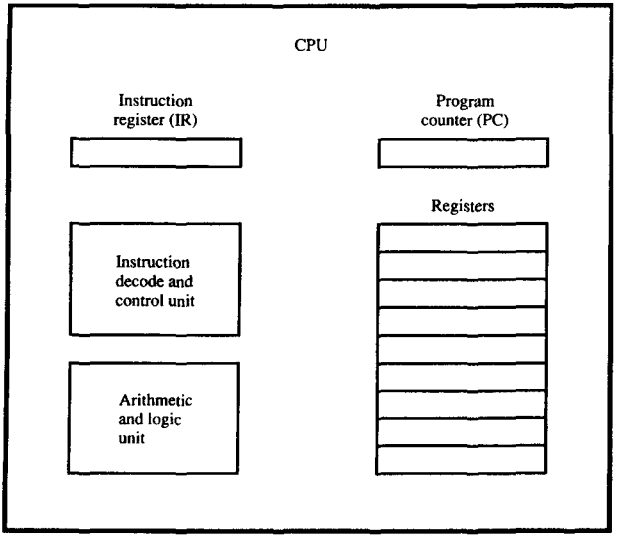


FIGURE 1–3
The central processing unit (CPU)

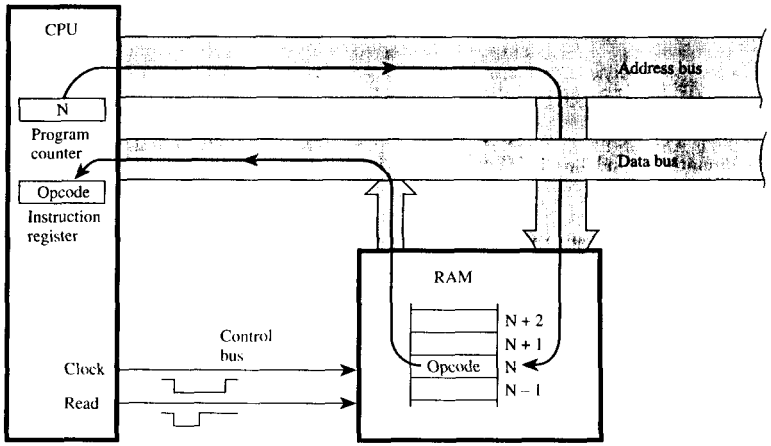


FIGURE 1–4
Bus activity for an opcode fetch cycle

ment 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 $2^6 \times 2^{10} = 64$K locations, while 20 bits can address 1024K (or 1 Meg) locations.

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.

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. Al-

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

though 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 man 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, or loudspeaker.

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 per-

[2]"CD-ROM" stands for compact-disc read-only memory. "WORM" stands for write-once read-mostly. A CD-ROM contains 550 Mbytes of storage, enough to store the entire 32 volumes of the Encyclopedia Britannica.

form 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.

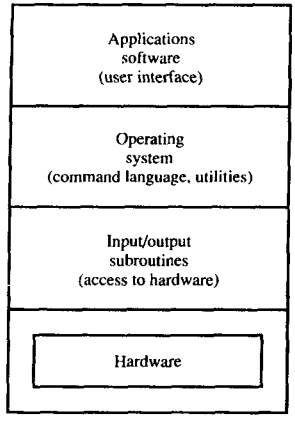
Control devices are outputs, or **actuators,** that can affect the world around them when supplied with a voltage or current (e.g., motors and relays). Monitoring devices are inputs, or **sensors,** that are stimulated by heat, light, pressure, motion, etc., and convert this to a voltage or current read 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.**

FIGURE 1–5
Levels of software



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 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 one of the prime objectives in the design of applications software.

1.8 MICROS, MINIS, AND MAINFRAMES

Using this as a starting point, we classify computers by their size and power 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 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 showing more detail (see Figure 1–6).

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 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 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.

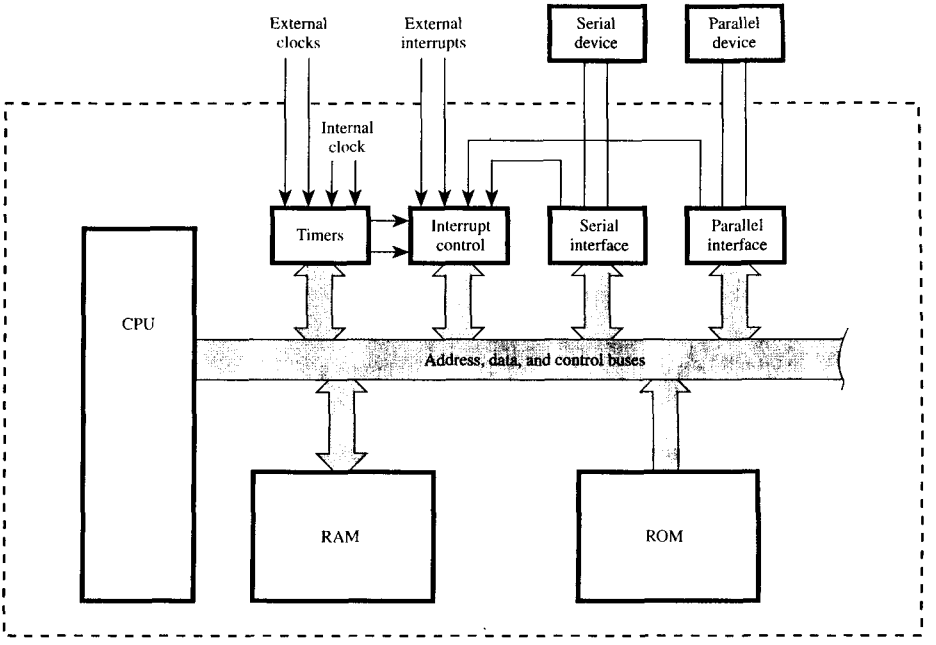


FIGURE 1–6
Detailed block diagram of a microcomputer system

1.9.3 Instruction Set Features

Due to the differences 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.

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.

[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 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 be 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 encompasses a lot of territory. The most illustrative, 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 proven 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 design into the product. The result often lacks eloquence—a show-stopper 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.

---

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

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. As examples, consider 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. As an example, consider in Figure 1–7 the somewhat trivial implementation of the NAND operation using an 8051 microcontroller.

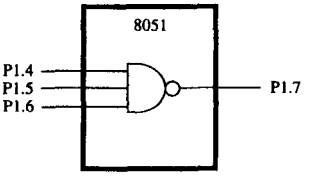


FIGURE 1–7
Microcontroller implementation of a simple logic operation

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

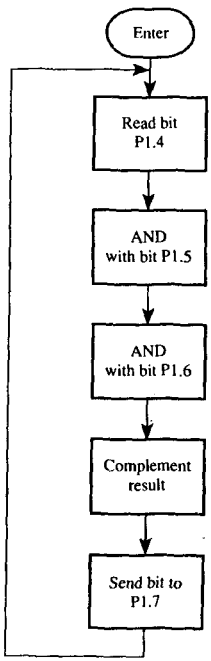


FIGURE 1–8
Flowchart for logic gate program

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 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™ family of devices.

PROBLEMS

1. What was the first widely used microprocessor? In what year was it introduced and by what company?
2. Two of the smaller microprocessor companies in the 1970s were MOS Technology and Zilog. Name the microprocessor that each of these companies introduced.
3. What year was the 8051 microcontroller introduced? What was the predecessor to the 8051 and in what year was it introduced?
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?
5. Which register in a CPU always contains an address? What address is contained in this register?
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?
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?
8. What is the usual meaning of "16-bits" in the phrase "16-bit computer"?
9. What is the difference between online storage and archival storage?
10. What type of technology is used for archival storage besides magnetic tape and disk?
11. With regard to computing systems, what is the goal of the field of engineering known as "human factors"?
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?
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?
14. What is the difference between an actuator and a sensor? Give an example of each.
15. What is firmware? Comparing a microcontroller-based system to a microprocessor-based system, which is more likely to rely on firmware? Why?
16. What is an important feature of a microcontroller's instruction set that distinguishes it from a microprocessor?
17. Name five products not mentioned in this chapter that are likely to use a microcontroller.

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-51™ ICs

PART NUMBER	ON-CHIP CODE MEMORY	ON-CHIP DATA MEMORY	TIMERS
8051	4K ROM	128 bytes	2
8031	0K	128 bytes	2
8751	4K EPROM	128 bytes	2
8052	8K ROM	256 bytes	3
8032	0K	256 bytes	3
8752	8K EPROM	256 bytes	3

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

The term "8051" loosely refers to the MCS-51™ family of microcontrollers. When discussion centers on an enhancement to the basic 8051 device, the specific part number

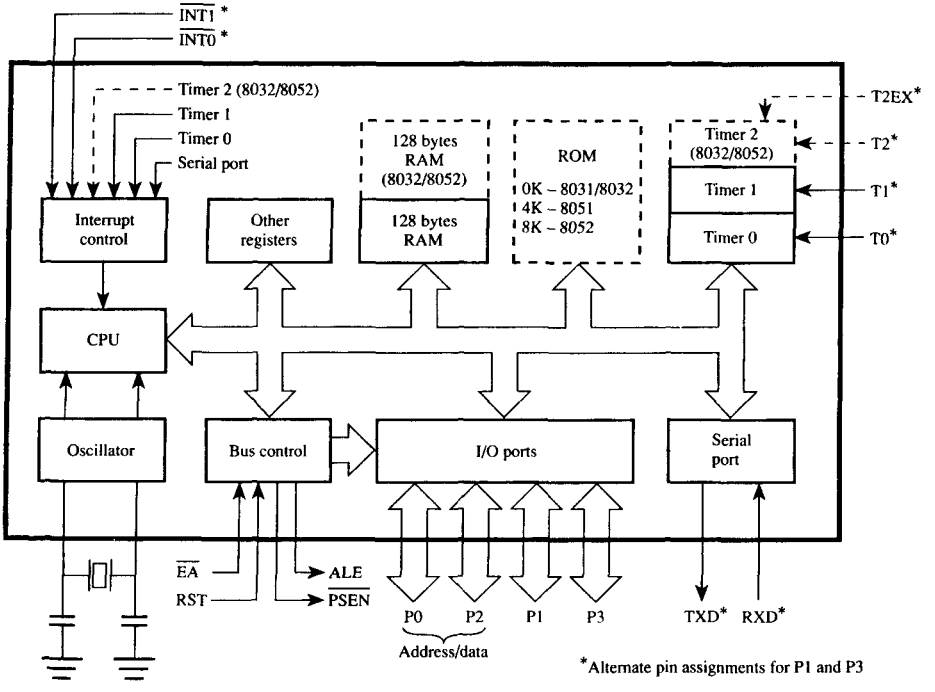


FIGURE 2–1
8051 block diagram

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.

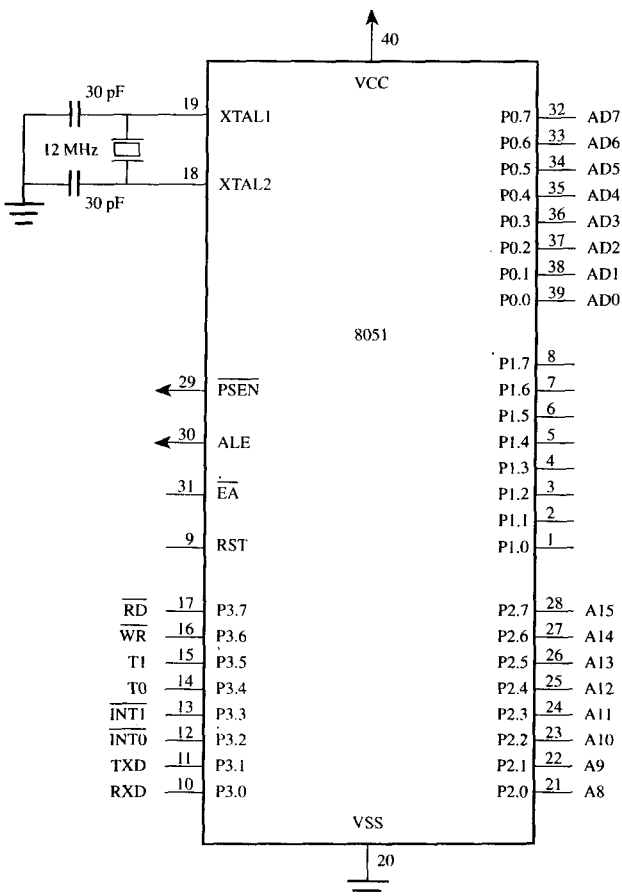


FIGURE 2–2
8051 pinouts

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 2.6 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 assigned 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 2.6 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.

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

2.2.5 $\overline{\text{PSEN}}$ (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 Output Enable ($\overline{\text{OE}}$) pin to permit reading of program bytes.

The $\overline{\text{PSEN}}$ signal pulses low during the fetch stage of an instruction. 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 bus—ALE 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 2.6 External Memory.)

The ALE signal pulses at a rate of 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 2–10.) This pin is also used for the programming input pulse for EPROM versions of the 8051.

2.2.7 $\overline{\text{EA}}$ (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 $\overline{\text{EA}}$ line for the +21 volt supply (V_{PP}) for programming the internal EPROM.

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. (See 2.8 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™ 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 4 low-power Schottky TTL loads for Ports 1, 2, and 3; and 8 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.

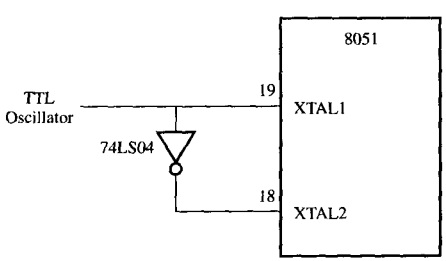
There is both a "read latch" and "read pin" capability. Instructions that require a read-modify-write operation (e.g., CPL P1.5) read the latch to avoid misinterpreting the voltage level in the event the pin is heavily loaded (e.g., when driving the base of a transistor). Instructions that input a port bit (e.g., MOV C,P1.5) read the pin. The port latch must contain a 1, in this case, otherwise the FET driver is ON and pulls the output low. A system reset sets all port latches, so port pins may be used as inputs without explicitly setting the 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 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,

FIGURE 2–3
Driving the 8051 from a TTL oscillator



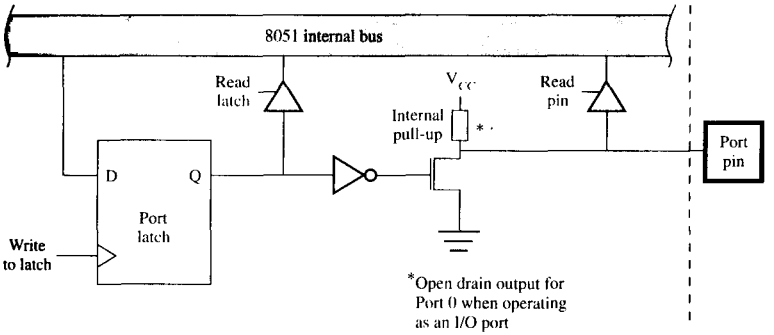


FIGURE 2–4
Circuitry for I/O ports

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 2–5 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 2–6 gives the details of the on-chip data memory. As shown, the internal memory space is divided between register banks (00H–1FH), bit-addressable RAM (20H–2FH), general-purpose RAM (30H–7FH), and special function registers (80H–FFH). Each of these sections of internal memory is discussed below.

2.4.1 General Purpose RAM

Although Figure 2–6 shows 80 bytes of general purpose RAM from addresses 30H to 7FH, the bottom 32 bytes from 00H to 2FH can be used similarly (although these locations have other purposes as discussed below).

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
```

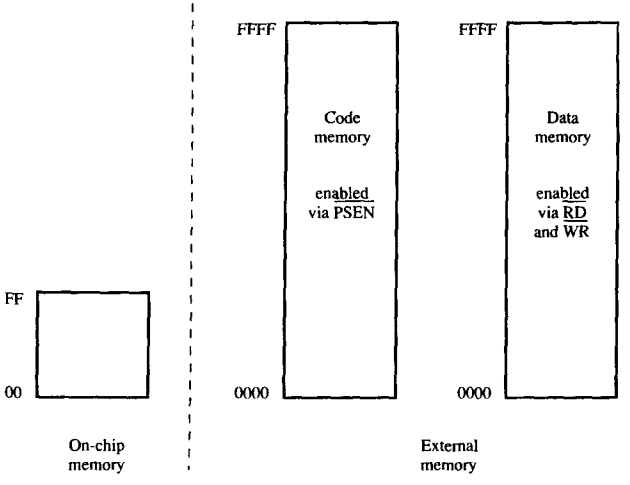


FIGURE 2–5
Summary of the 8031 memory spaces

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.

2.4.2 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 $\times$ 16 bytes = 128 bits). These addresses are accessed as bytes

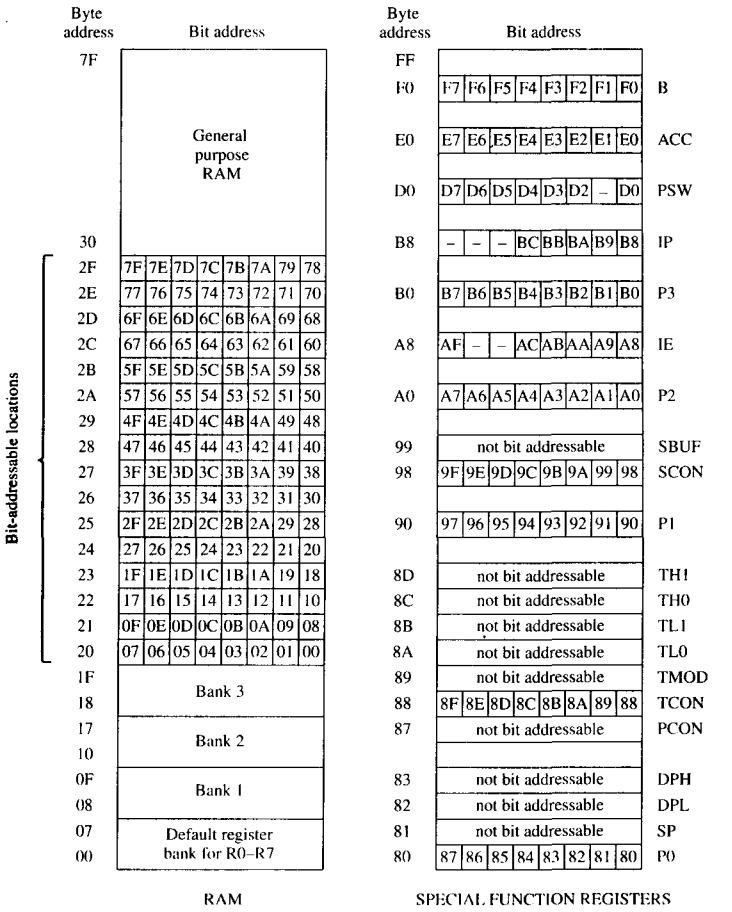


FIGURE 2–6
Summary of the 8051 on-chip data memory

or as bits, depending on the instruction. For example, to set bit 67H, the following instruction could be used:

```
SETB 67H
```

Referring to Figure 2–6, 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
```

2.4.3 Register Banks

The bottom 32 locations of internal memory contain the register banks. The 8051 instruction set supports 8 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 and faster 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.

2.5 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 FFH. (See Figure 2–6 and Appendix D.) Note that most of the 128 addresses from 80H to FFH are not defined. Only 21 SFR addresses are defined (26 on the 8032/8052).

Although the accumulator (A) may be accessed implicitly as shown previously, most SFRs are accessed using direct addressing. Note in Figure 2–6 that some SFRs are

[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.

both bit-addressable and byte-addressable. Designers 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 E0H 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.

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.

2.5.1 Program Status Word

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

2.5.1.1 Carry Flag

The carry flag (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 FFH, then the instruction

```
ADD  A,#1
```

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

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
```

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

BIT	SYMBOL	ADDRESS	BIT DESCRIPTION
PSW.7	CY	D7H	Carry flag
PSW.6	AC	D6H	Auxiliary 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

2.5.1.2 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.

2.5.1.3 Flag 0

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

2.5.1.4 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 contents 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.

2.5.1.5 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.

2.5.1.6 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 10101101B, P will contain 1 (establishing a total of 6 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.

2.5.2 B Register

The B register at address F0H 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 di-

vides 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 F0H to F7H.

2.5.3 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.

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.

2.5.4 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).

2.5.5 Port Registers

The 8051 I/O ports consist of Port 0 at address 80H, Port 1 at address 90H, Port 2 at address A0H, and Port 3 at address B0H. 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 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.

As 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."

2.5.6 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 (TL1, 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.

2.5.7 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.

2.5.8 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 (IE) at address A8H. The priority level is set through the interrupt priority register (IP) at address B8H. Both registers are bit-addressable. Interrupts are discussed in detail in Chapter 6.

2.5.9 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™ family ICs but are now implemented only in the CMOS versions. PCON is not bit-addressable.

2.5.9.1 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.

2.5.9.2 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 oscillator is stopped, (2) all

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	GF1	General purpose flag bit 1
2	GF0	General purpose flag bit 0
1*	PD	Power down; set to activate power down mode; only exit is reset
0*	IDL	Idle mode; set to activate idle mode; only exit is an interrupt or system reset

*Only implemented in CMOS versions

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

During power down mode, V_{CC} can be as low as 2V. Care should be taken not to lower V_{CC} 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).

2.6 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™ 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.

Before discussing the specific details of multiplexing the address and data buses, the general idea is presented in Figure 2–7. 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,

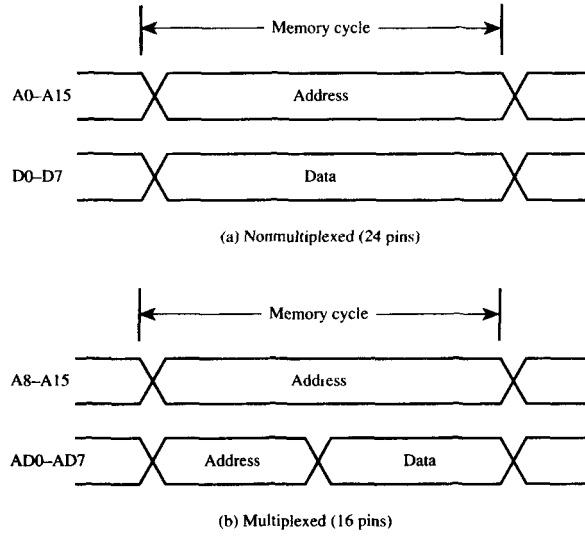


FIGURE 2–7
Multiplexing the address bus (low-byte) and data bus

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.

2.6.1 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 2–8.

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 2–9.

2.6.2 Accessing External Data 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

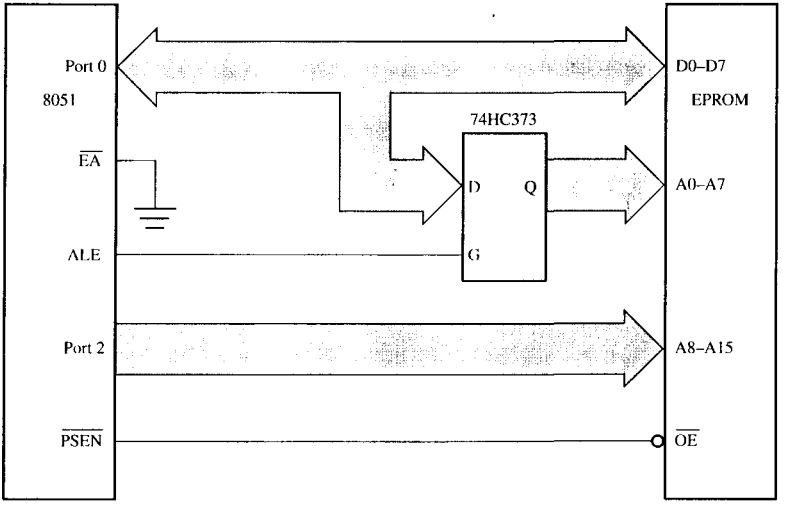


FIGURE 2–8
Accessing external code memory

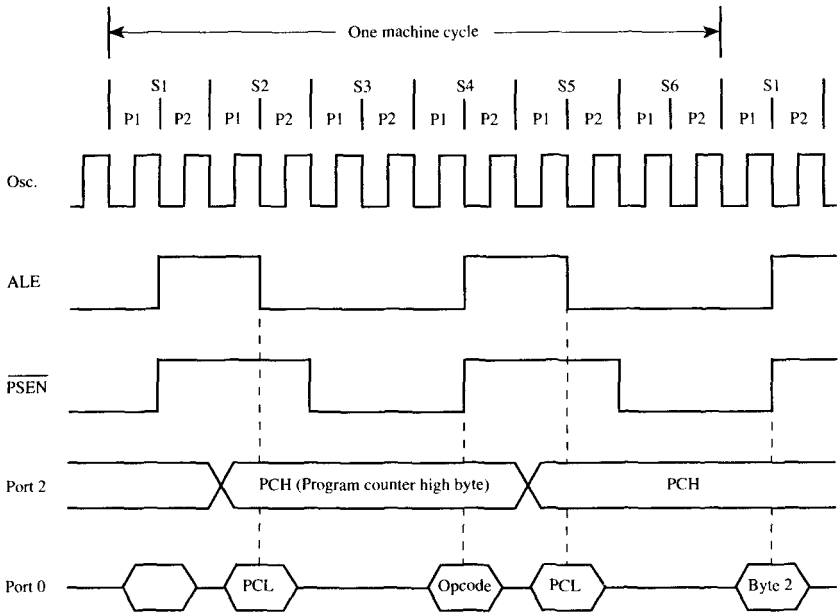


FIGURE 2–9
Read timing for external code memory

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 $\overline{RD}$ line connects to the RAM's output enable ($\overline{OE}$) line and $\overline{WR}$ connects to the RAM's write ($\overline{W}$) line. The connections for the address and data bus are the same as for EPROMs. Using Ports 0 and 2 as above, up to 64K 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 2–10 for the MOVX A,@DPTR instruction. Notice that both an ALE pulse and a $\overline{PSEN}$ pulse are skipped in lieu of a pulse on the $\overline{RD}$ line to enable the RAM.[2]

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

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

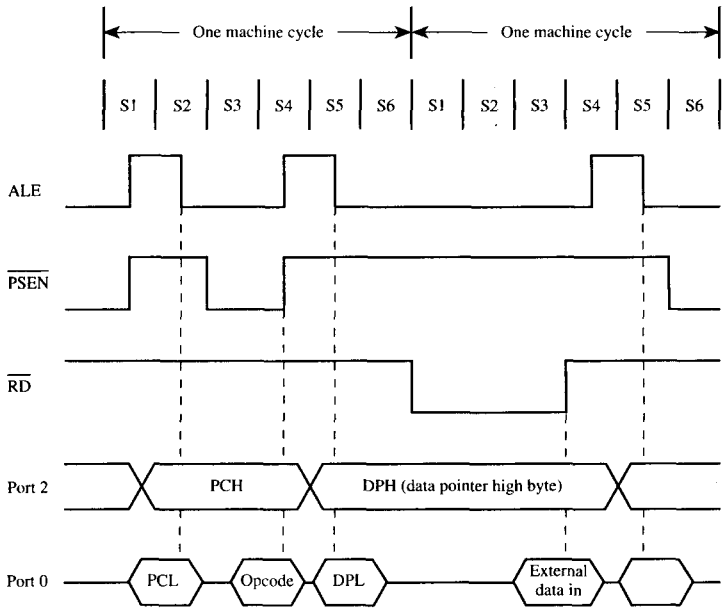


FIGURE 2–10
Timing for MOVX instruction

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 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 2–11.

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 following instructions could be used to read the contents 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
```

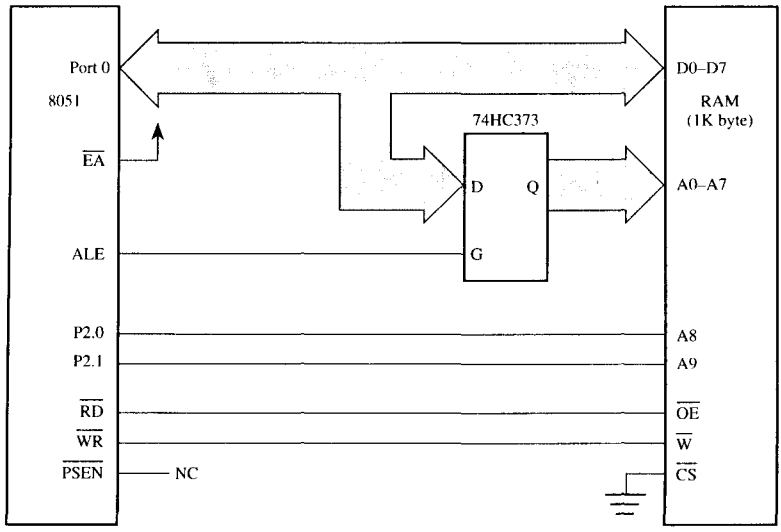


FIGURE 2–11
Interface to 1K RAM

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.

2.6.3 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 ($\overline{CS}$) inputs on the memory ICs. This is illustrated in Figure 2–12 for a system with multiple 2764 8K EPROMs and 6264 8K RAMs. Remember, due to the separate enable lines ($\overline{PSEN}$ for code memory, $\overline{RD}$ and $\overline{WR}$ for data memory), the 8051 can accommodate up to 64K *each* of EPROM and RAM.

2.6.4 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 $\overline{PSEN}$ is used to read code memory and $\overline{RD}$ is

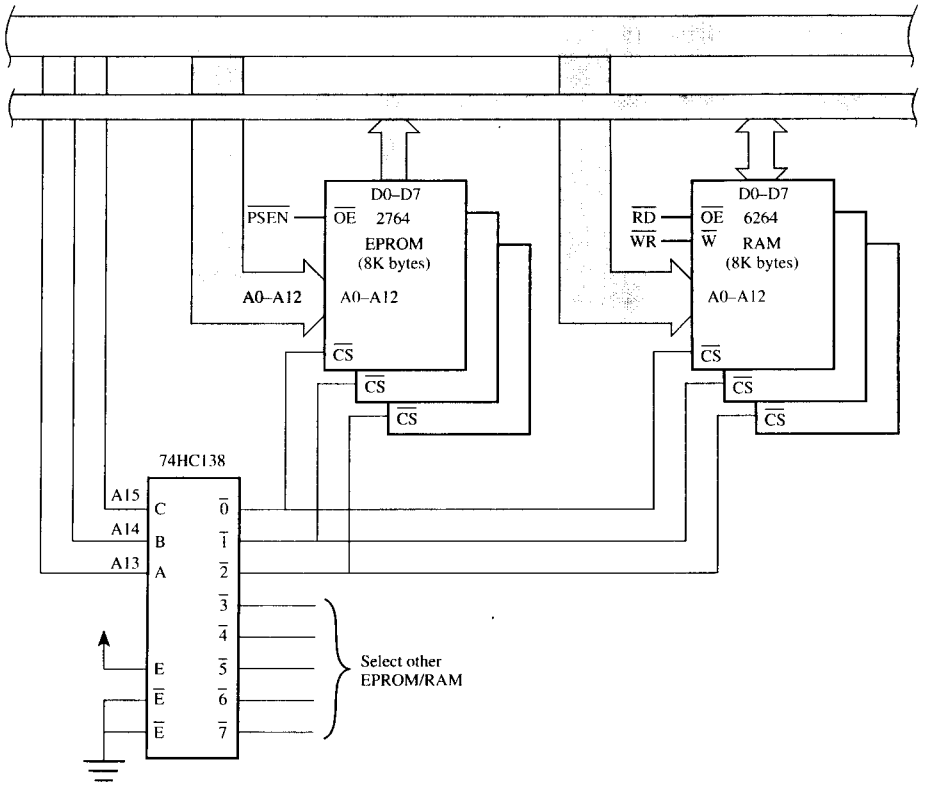


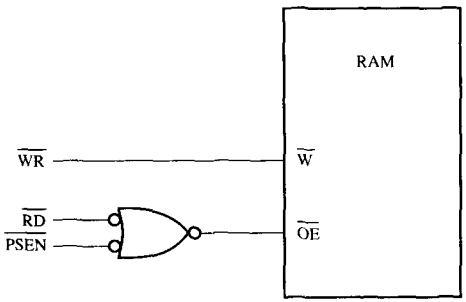
FIGURE 2–12
Address decoding

used to read data memory, a RAM can occupy code *and* data memory space by connecting its $\overline{OE}$ line to the logical AND (negative-input NOR) of $\overline{PSEN}$ and $\overline{RD}$. The circuit shown in Figure 2–13 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).

2.7 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 80H to FFH. So as not to conflict with the SFRs (which have the same ad-

FIGURE 2–13
Overlapping the external code and data spaces



dresses), 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 contents of internal address F0H 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 2–14.

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.

2.8 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 may be activated upon power-

FIGURE 2–14
8032/52 memory spaces

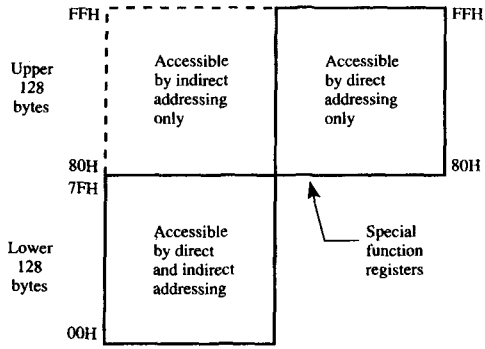


TABLE 2–5
Timer 2 registers

REGISTER	ADDRESS	DESCRIPTION	BIT-ADDRESSABLE
T2CON	C8H	Control	Yes
RCAP2L	CAH	Low-byte capture	No
RCAP2H	CBH	High-byte capture	No
TL2	CCH	Timer 2 low-byte	No
TH2	CDH	Timer 2 high-byte	No

up using an R-C (resistor-capacitor) network. Figure 2–15 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.

2.9 SUMMARY

This chapter has summarized the 8051 hardware architecture. Before developing useful applications, though, we must understand the 8051 instruction set. The next chapter fo-

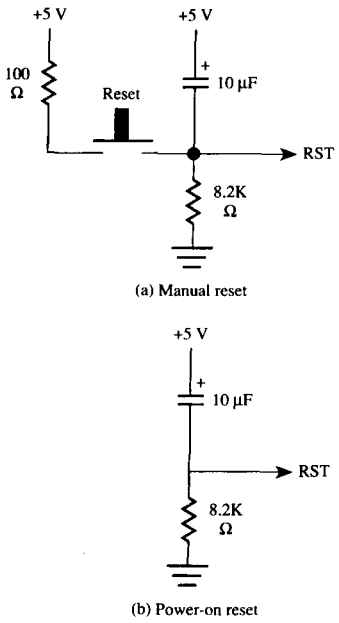


FIGURE 2–15
Two circuits for system reset. (a) Manual Reset (b) Power-on Reset

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	FFH
IP (8031/8051)	XXX00000B
IP (8032/8052)	XX000000B
IE (8031/8051)	0XX00000B
IE (8032/8052)	0X000000B
Timer registers	00H
SCON	00H
SBUF	00H
PCON (HMOS)	0XXXXXXXB
PCON (CMOS)	0XXX0000B

cuses on the 8051 instructions and addressing modes. The discussions of the timer, serial port, and interrupt SFRs were deliberately sparse in this chapter, since dedicated chapters follow that examine these in detail.

PROBLEMS

1. Name four manufacturers of the 8051 microcontroller, besides Intel.
2. Which device in the MCS-51™ family would probably be used for a product that will be manufactured in large quantities with a large on-chip program?
3. What instruction could be used to set the least-significant bit at byte address 25H?
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?
5. What bit addresses are set to one as a result of the following instructions?

```
MOV  R0,#26H
MOV  @R0,#7AH
```

6. What 1-byte instruction has the same effect as the following 2-byte instruction?

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

7. Illustrate an instruction sequence to store the value 0ABH in external RAM at address 9A00H.
8. How many special function registers are defined on the 8052?
9. What is the value of the 8051's stack pointer immediately after a system reset?
10. What instruction could be used to initialize the 8031 SP to create a 64-byte stack at the top of internal RAM?

11. 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.
12. The 80C31BH–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?
13. If an 8051 is operating from a 4 MHz crystal, what is the duration of a machine cycle?
14. 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.
15. 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.)
16. Section 2.8 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 2–15a 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?
17. How many low-power Schottky loads can be driven by the port line P1.7 on pin 8?
18. Name the 8051 control bus signals used to select external EPROMs and external RAMs.
19. What is the bit address of the most-significant bit at byte address 25H in the 8051's internal data memory?
20. What instruction sets the least-significant bit of the accumulator without affecting the other 7 bits?
21. Assuming the following instruction has just executed,

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

    what is the state of the P bit in the program status word?
22. What instruction sequence could be used to copy the contents of R7 to external RAM location 100H?
23. 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?
24. What is the difference between the 8051's idle mode and power-down mode?
25. What instruction could be used to force the 8051 into power-down mode?
26. Illustrate how two 32K-byte static RAMs could be interfaced to the 8051 so that they occupy the full 64K external data space.

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™ 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™ 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 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's 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 8 "working registers," numbered R0 through R7. Instructions using register addressing are encoded using the three least-significant bits of the instruction opcode to indicate 1 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.

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
```

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,

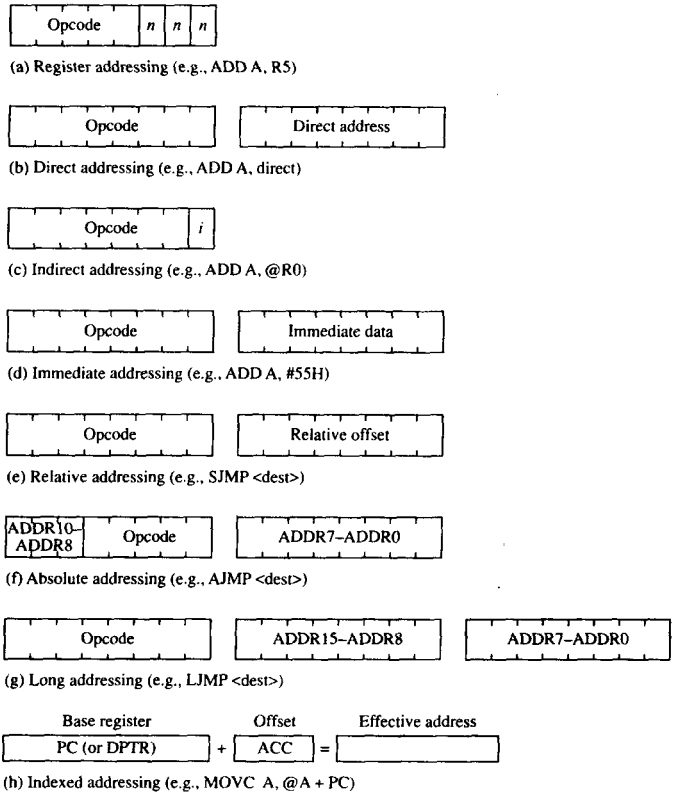


FIGURE 3–1
8051 Addressing modes. (a) Register addressing (b) Direct addressing (c) Indirect addressing (d) Immediate addressing (e) Relative addressing (f) Absolute addressing (g) Long addressing (h) Indexed addressing.

```
INC  DPTR
```

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

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–FFH). 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 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.). As an example of direct addressing, the instruction

```
MOV  P1,A
```

transfers the contents 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)
```

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 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,@R1
```

moves 55H into the accumulator.

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.

3.2.5 Relative Addressing

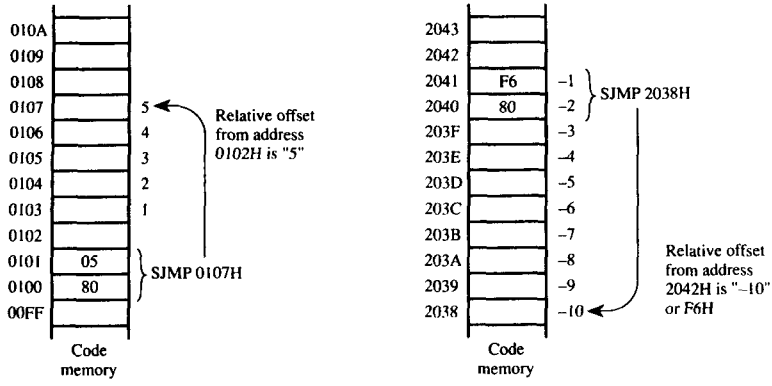
Relative addressing is used only with certain jump instructions. A relative address (or offset) is an 8-bit signed value, which is added to the program counter to form the address of the next instruction executed. Since an 8-bit signed offset is used, the range for jumping is −128 to +127 locations. The relative offset is appended to the instruction as an additional byte. (See Figure 3–1e.)

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).



(a) Short jump ahead in memory

(b) Short jump back in memory

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.

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
```

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

```
11100001 - 1st byte (A10-A8 + opcode)
01000110 - 2nd byte (A7-A0)
```

The underlined bits are the low-order 11 bits of the destination address, 0F46H = 000011110100011OB. The upper 5 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.

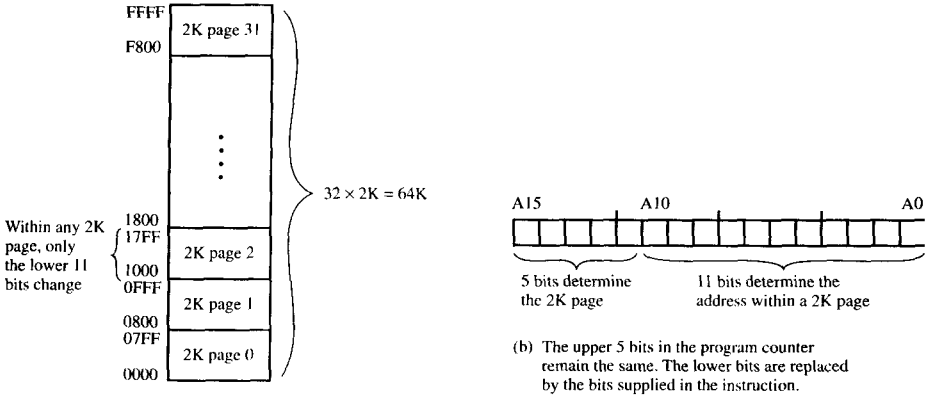
Absolute addressing offers the advantage of short (2-byte) instructions, but has the disadvantages of limiting the range for the destination and providing position-dependent code.

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 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.

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.



(a) 64K memory map divided into 32 2K pages

(b) The upper 5 bits in the program counter remain the same. The lower bits are replaced by the bits supplied in the instruction.

FIGURE 3–3
Instruction encoding for absolute addressing. (a) Memory map showing 2K pages (b) Within any 2K page, the upper 5 address bits are the same.

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

All arithmetic instructions execute in 1 machine cycle except the INC DPTR instruction (2 machine cycles) and the MUL AB and DIV AB instructions (4 machine cycles). (Note that one machine cycle takes 1 μs if the 8051 is operating from a 12 MHz clock.)

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.

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 UNDERFLOW 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 FFH.

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 contents of A by the contents 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.)

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 DA A will not convert a binary number to BCD; it produces a meaningful result only as the second step in the addition of 2 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.)

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.

```
        01010011                (immediate data)
AND     00110101                (original value of A)
        00010001                (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.

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™ 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 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.

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.

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 F3H, R1 contains 40H, and internal RAM address 40H contains 5BH, then the instruction

```
XCHD A,@R1
```

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

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 8 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 Data Memory.")

All data transfer instructions that operate on external memory execute in 2 machine cycles and use the accumulator as either the source or destination operand.

The read and write strobes to external RAM ($\overline{RD}$ and $\overline{WR}$) 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.

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 256 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 used as the base address, and the table is accessed through a subroutine. First, the number of the desired entry is loaded into the accumulator, then the subroutine is called. The setup and call sequence would be coded as follows:

```
            MOV  A,ENTRY_NUMBER
            CALL LOOK_UP
            .
            .
            .
LOOK_UP:    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.

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™ family of microcontrollers—are 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–FFH 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 00H) 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  P1.7
```

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, while 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.

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 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  C,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 $\oplus$ 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 generic 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 opcode contains 3 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 F800H; 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
```

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 contents 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.

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 a 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 left 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.

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.

PROBLEMS

1. What is the hexadecimal opcode for the following instruction?

```
INC  DPTR
```

2. What is the hexadecimal opcode for the following instruction?

```
DEC  R6
```

3. What instruction is represented by the opcode 5DH?
4. What instruction is represented by the opcode FFH?
5. List all the 8051's 3-byte instructions with an opcode ending in 5H.
6. Illustrate how the contents of internal address 50H could be transferred to the accumulator, using indirect addressing.
7. What opcode is undefined on the 8051?
8. The following is an 8051 instruction:

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

   a) What is the opcode for this instruction?

   b) How many bytes long in 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?

9. 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?

10. What is the relative offset for the instruction

```
SJMP BACK
```

   if the instruction is in locations A050H and A051H, and the label BACK represents the instruction at address 9FE0H?

11. 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?

12. 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?

13. 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?

14. What does the following instruction do?

```
SETB 0D7H
```

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

15. What is the difference between the following two instructions?

```
INC  A
INC  ACC
```

16. What are the machine-language bytes for the instruction

```
LJMP ONWARD
```

if the label ONWARD represents the instruction at address A0F6H?

17. Assume accumulator A contains 5AH. What is the result in accumulator A after the following instruction executes?

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

18. Assume the PSW contains 0C0H and accumulator A contains 50H just before the following instruction executes:

```
RLC  A
```

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

19. 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 8051 is operating from a 12 MHz crystal.
20. Write a program to create an 83.3 kHz square wave on P1.0. (Assume 12 MHz operation.)
21. Write a program to generate a 4 μs active-high pulse on P1.7 every 200 μs.
22. Write programs to implement the logic operations shown in Figure 3–4.
23. For part (a) above, what is the worst-case propagation delay time from an input transition to an output transition?
24. What is the content of accumulator A after the following instruction sequence executes?

```
MOV  A,#7FH
MOV  50H,#29H
MOV  R0,#50H
XCHD A,@R0
```

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

```
SETB P2.6
```

26. What instruction sequence could be used to copy Flag 0 in the PSW to the port pin P1.5?
27. Under what circumstances will Intel's assembler (ASM51) convert a generic JMP instruction to LJMP?
28. The 8051 internal memory is initialized as follows immediately prior to the execution of a 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?

29. 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.)

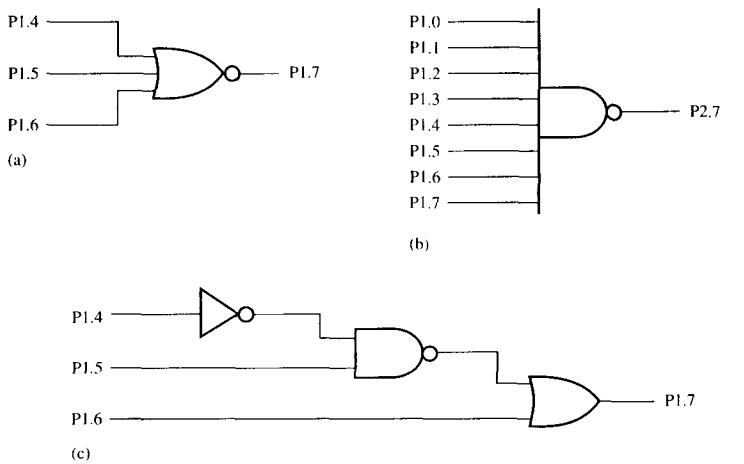


FIGURE 3–4
Logic gate programming problems. (a) 3-input NOR (b) 8-input NAND (c) 3-gate logic operation

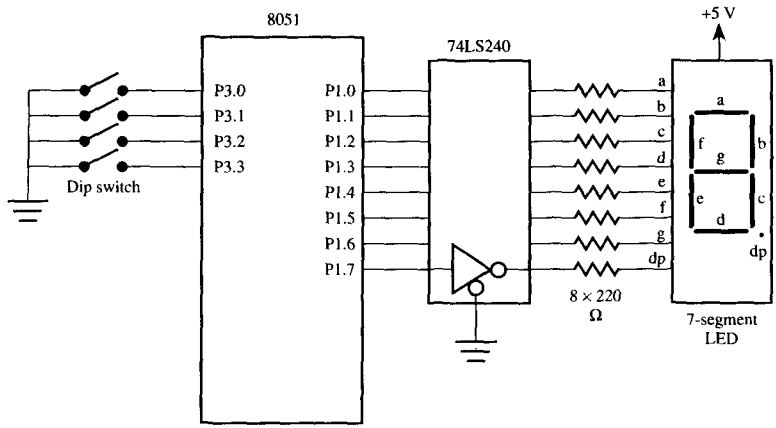


FIGURE 3–5
Interface to a DIP switch and 7-segment LED

30. A 4-bit DIP switch and a common-anode 7-segment LED are connected to an 8051 as shown in Figure 3–5. Write a program that continually reads a 4-bit code from the 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 8051 port pin to "1" turns the corresponding segment "ON." (See Figure 3–5.)

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-two 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 FFFFH. The overflow flag is set on the FFFFH-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-two mode (i.e., the $\overline{Q}$ 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 $Q_0 = 0$ ($4_{10} = 100_2$).

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 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 ap-

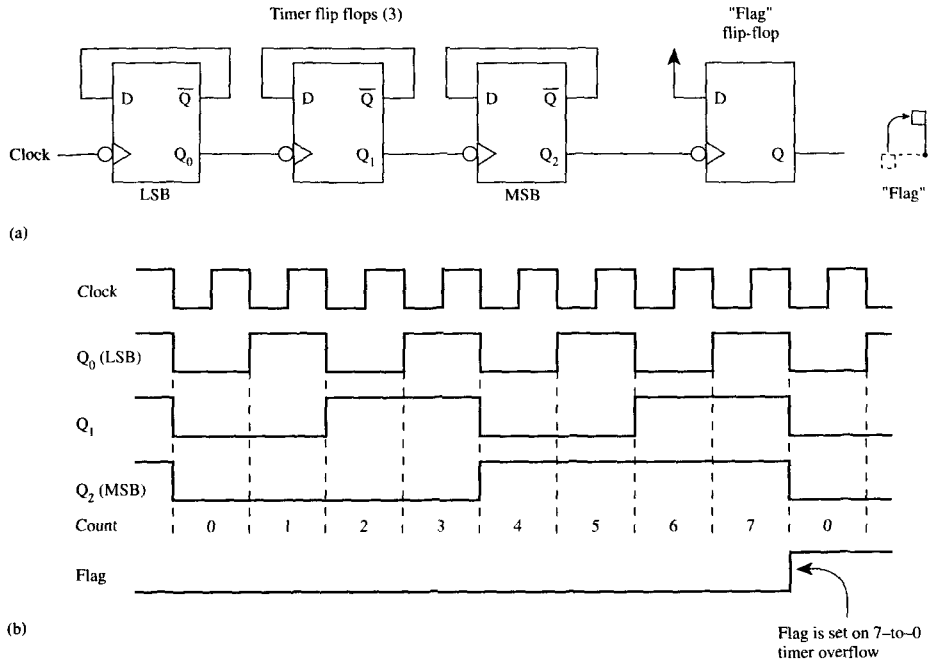


FIGURE 4–1
A 3-bit timer. (a) Schematic (b) Timing diagram.

plications 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 5 SFRs provide access to the third timer in the 8052.

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.

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

TABLE 4–2
TMOD (timer mode) register summary

BIT	NAME	TIMER	DESCRIPTION
7	GATE	1	Gate bit. When set, timer only runs while $\overline{INT1}$ is high
6	C/$\overline{T}$	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/$\overline{T}$	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

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.

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.

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.

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	8EH	Timer 1 run control bit. Set/cleared by software 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 $\overline{\text{INT 1}}$; 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

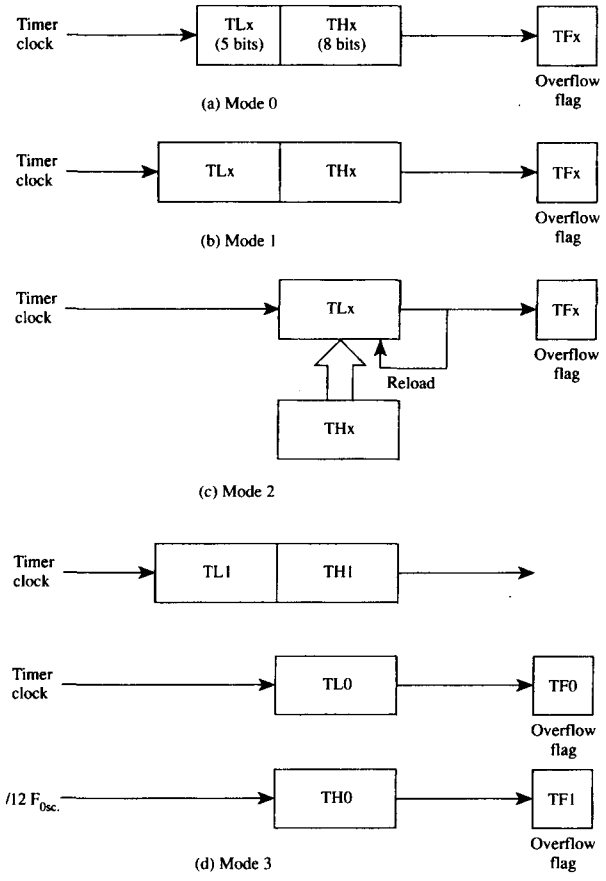


FIGURE 4–2
Timer modes. (a) Mode 0 (b) Mode 1 (c) Mode 2 (d) Mode 3.

4.4.2 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 FFFFH-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 FFH to 00H, 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 FFH-to-00H transition, 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.)

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.

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

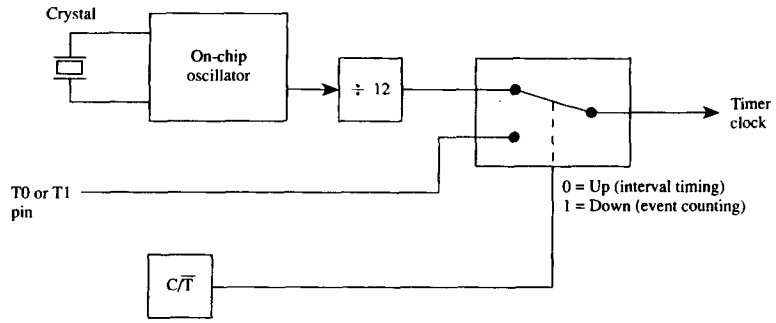


FIGURE 4–3
Clocking source

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 "T1," 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.)

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

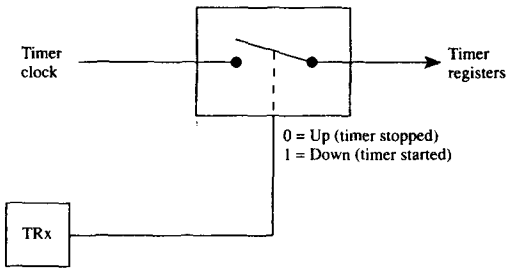
```
SETB TR0
```

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.

FIGURE 4–4
Starting and stopping the timers



Another method for controlling the timers is with the GATE bit in TMOD and the external input $\overline{INTx}$. Setting GATE = 1 allows the timer to be controlled by $\overline{INTx}$. This is useful for pulse width measurements as follows. Assume $\overline{INT0}$ is low but pulses high for a period of time to be measured. Initialize Timer 0 for mode 2, 16-bit timer mode, with TL0/TH0 = 0000H, GATE = 1, and TR0 = 1. When $\overline{INT0}$ goes high, the timer is "gated on" and is clocked at a rate of 1 MHz. When $\overline{INT0}$ goes low, the timer is "gated off" and the duration of the pulse in microseconds is the count in TL0/TH0. ($\overline{INT0}$ 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.

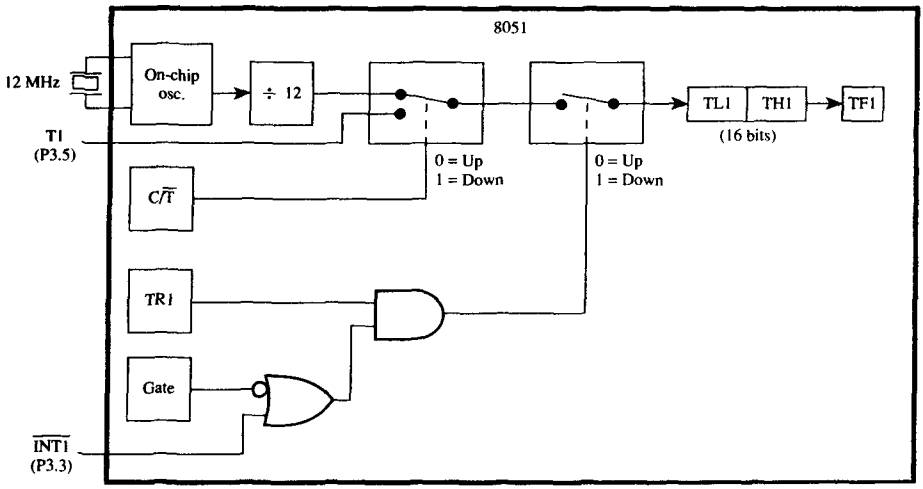


FIGURE 4–5
Timer 1 operating in mode 1

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 $C/\overline{T}$ = 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 FFFFH-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 FF9CH. 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,TH1
          MOV  R6,TL1
          CJNE A,TH1,AGAIN
          MOV  R7,A
```

4.8 SHORT INTERVALS 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.)

Example 4–1: *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?

Very short intervals (i.e., high frequencies) can be programmed without using the timers. Here's the program:

```
8100             5          ORG     8100H
8100 D290        6    LOOP: SETB    P1.0   ;one machine cycle
8102 C290        7          CLR     P1.0   ;one machine cycle
8104 80FA        8          SJMP    LOOP   ;two machine cycles
                 9          END
```

This program creates a pulse waveform on P1.0 with a period of 4 μs: high-time = 1 μs, low-time = 3μs. *The frequency is 250 kHz and the duty cycle is 25%.* (See Figure 4–6.)

It might appear at first that the instructions in Figure 4–6 are misplaced, but they're 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 output signal can be lengthened somewhat by inserting NOP (no operation) instructions into the loop. Each NOP adds 1 μs to the period of the output sig-

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

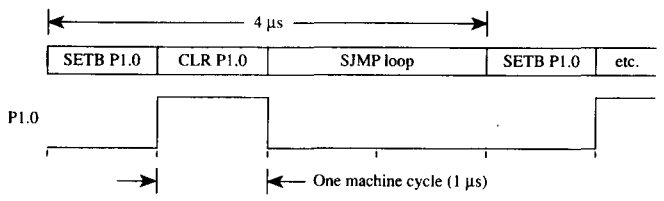


FIGURE 4–6
Waveform for example

nal. For example, adding two NOP instructions after SETB P1.0 would make the output a square wave with a period of 6 μs and a frequency of 166.7 kHz. Beyond a point, however, "software tuning" is cumbersome and a timer is the best choice for delays.

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 4–2: 10 kHz Square Wave

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

A 10 kHz square wave requires a high-time of 50 μs and a low-time of 50 μs. Since this 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. Here's the program:

```
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
```

This program uses a complement bit instruction (CPL) rather than the SETB and CLR bit instructions in the previous example. 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 may be specified using decimal notation as −50, rather than using hexadecimal notation. The assembler performs the necessary conversion. Note that the timer overflow flag (TF0) must be explicitly cleared by software after each overflow.

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 4–3: 1 kHz Square Wave

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

A 1 kHz square wave requires a high-time of 500 μs and a low-time of 500 μs. Since this 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.

```
8100             6              ORG     8100H
8100 758901      7              MOV     TMOD,#01H    ;16-bit timer mode
8103 758CFE      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
```

There is a slight error 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 registers TL0/TH0 must be adjusted somewhat. Such errors do not occur in auto-reload mode, since the timer is never stopped—it overflows at a consistent rate set by the reload value in TH0.

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 4–4: 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.

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 1 pins can sink a maximum of 1.6 mA. The AI-430 costs a few dollars.

Creating software delays is one of the most common programming tasks given to students of microprocessors. The usual method of decrementing a count within a loop is

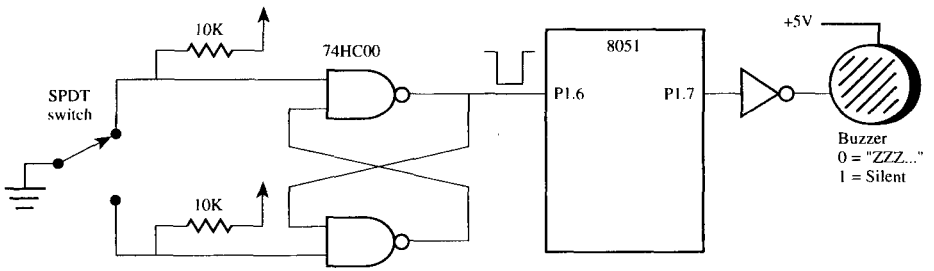


FIGURE 4–7
Buzzer example

not necessary on the 8051, since it has built-in timers. A 1-second delay subroutine using Timer 0 is shown in this example.

A 1-to-0 transition on P1.6 is detected by waiting for a 1 (JNB P1.6,LOOP) and then waiting for a 0 (JB P1.6,WAIT). Here's the program:

```
  0064          6     HUNDRED  EQU     100          ;100 x 10000 us = 1 sec.
  2710          7     COUNT    EQU     10000
8100            8              ORG     8100H
8100 758901     9              MOV     TMOD,#01H    ;use timer 0 in mode 1
8103 3096FD     10    LOOP:    JNB     P1.6,LOOP    ;wait for 1 input
8106 2096FD     11    WAIT:    JB      P1.6,WAIT    ;wait for 0 input
8109 D297       12             SETB    P1.7         ;turn buzzer on
810B 128112     13             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 758C27     18    AGAIN:   MOV     TH0,#HIGH COUNT
8117 758A10     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
```

There are two situations not handled in the example above. First, if the input toggles during the one 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 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/$\overline{T2}$ bit in T2CON selects between the internal and external clock, just as the C/$\overline{T}$ 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.

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

4.9.1 Auto-Reload Mode

The capture/reload bit in T2CON selects between the first two modes. When CP/$\overline{RL2}$ = 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 FFFFH-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.

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.

4.9.2 Capture Mode

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

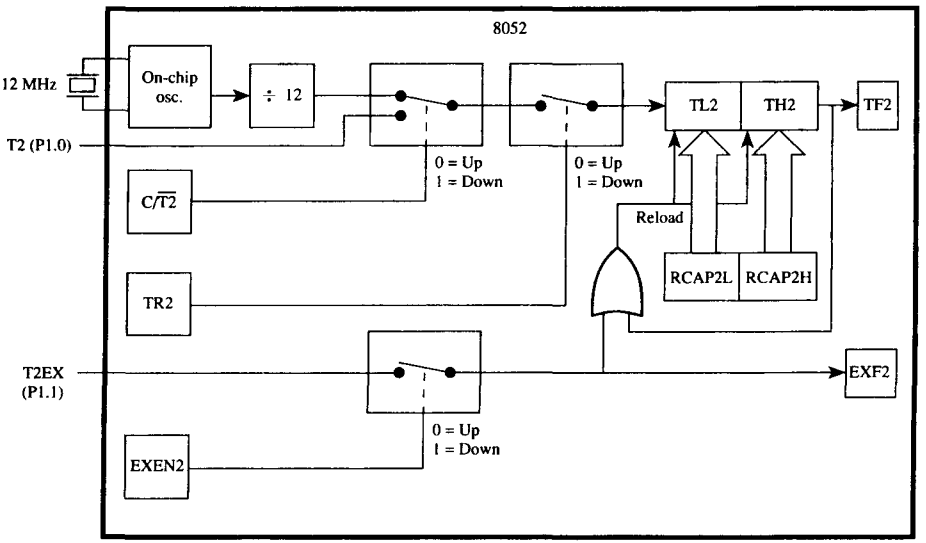


FIGURE 4–8
Timer 2 in 16-bit auto-reload mode

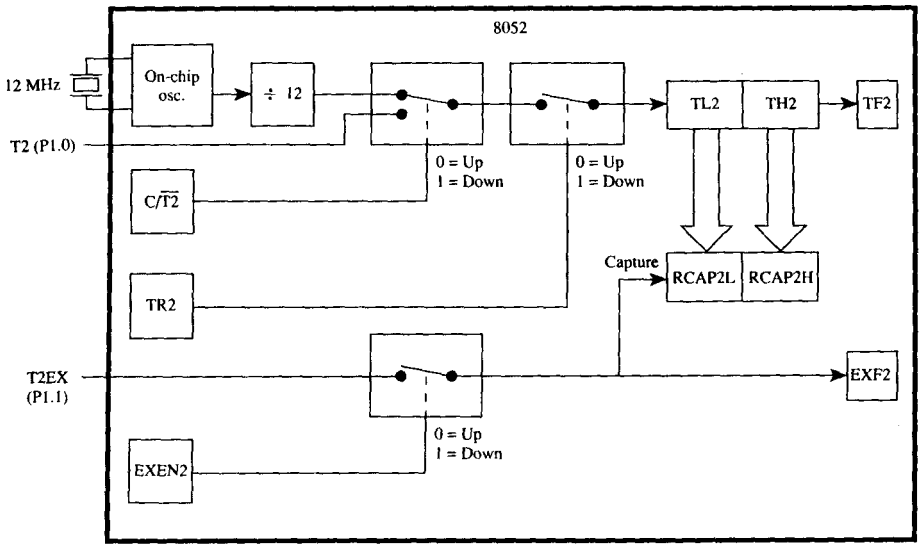


FIGURE 4–9
Timer 2 in 16-bit capture mode

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.

4.10 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.

4.11 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.

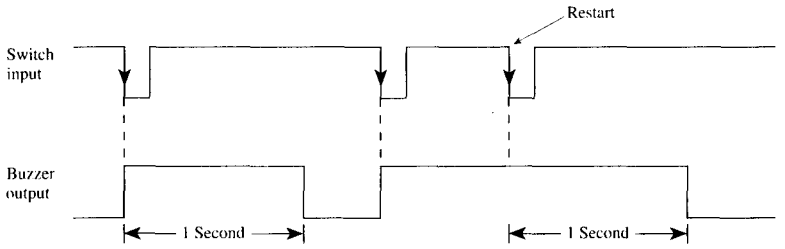


FIGURE 4–10
Timing for modified buzzer example

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

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.)
2. What is the effect of the following instruction?

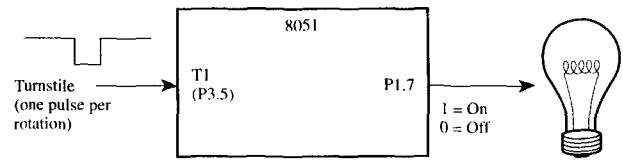
```
SETB 8EH
```

3. What is the effect of the following instruction?

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

4. Consider the three-instruction program shown in Example 4–1. What is the frequency and duty cycle of the waveform created on P1.0 for a 16 MHz 8051?
5. Rewrite the solution to Example 4–4 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.
6. Write an 8051 program to generate a 12 kHz square wave on P1.2 using Timer 0.
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 T1 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.)
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?

FIGURE 4–11
Turnstile problem



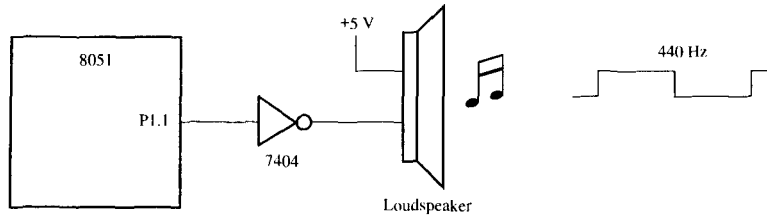


FIGURE 4–12
Loudspeaker interface

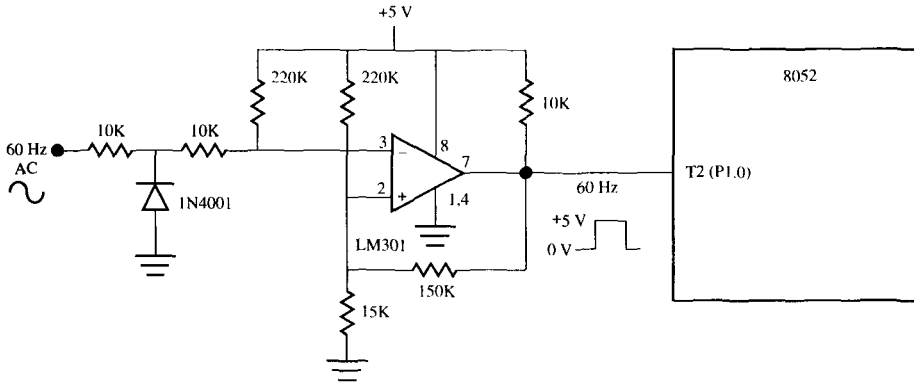


FIGURE 4–13
60 Hz time base

9. Write an 8051 program to generate a 500 Hz signal on P1.0 using Timer 0. The waveform should have a 30% duty cycle (duty cycle = high-time ÷ period).
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.

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.

Two special function registers provide software access to the serial port, SBUF and SCON. The serial port buffer (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.)

The serial port control register (SCON) at address 98H is a bit-addressable register containing status bits and control bits. Control bits set the operating mode for the serial port, and status bits indicate the end of a character transmission or reception. The status bits are tested in software or programmed to cause an interrupt.

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.)

5.2 SERIAL PORT CONTROL REGISTER

The mode of operation of the 8051 serial port is set by writing to the serial port mode register (SCON) at address 99H. (See Table 5–1 and Table 5–2.)

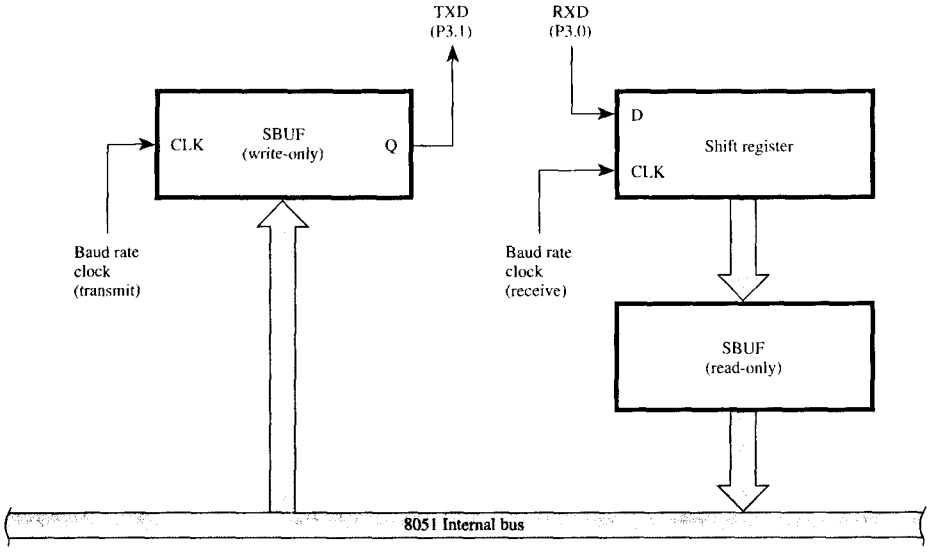


FIGURE 5–1
Serial port block diagram

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.

5.3 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.

5.3.1 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; RI 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

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 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.)

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.

TABLE 5–2
Serial port 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)

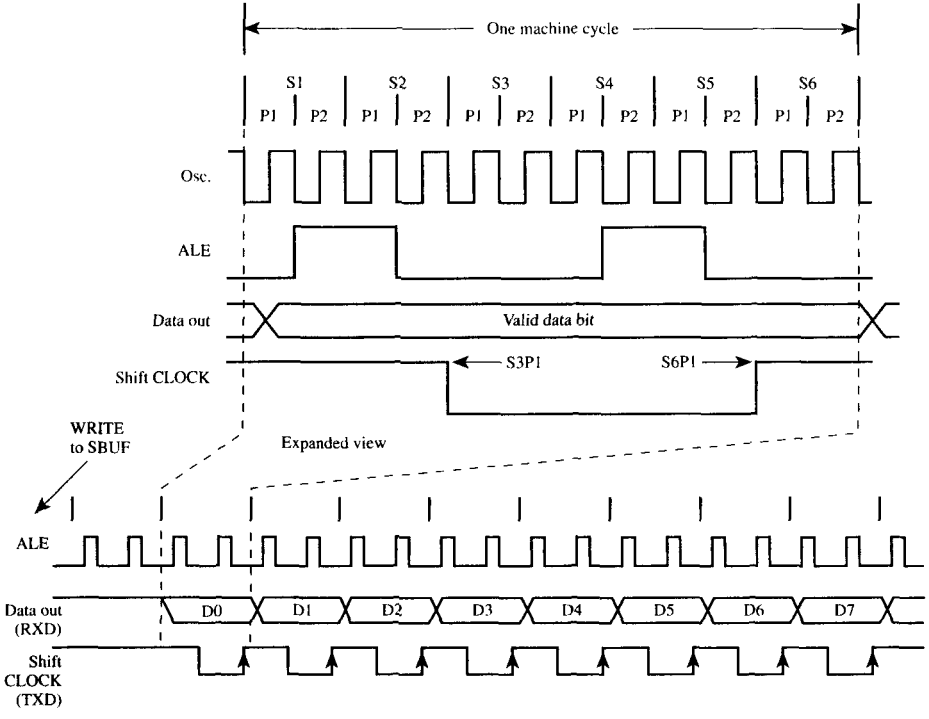


FIGURE 5–2
Serial port transmit timing for mode 0

5.3.2 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-to-parallel 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).

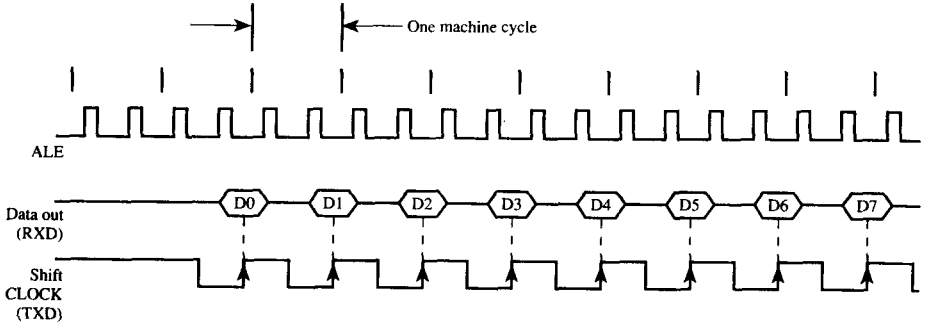


FIGURE 5–3
Serial port receive timing for mode 0

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

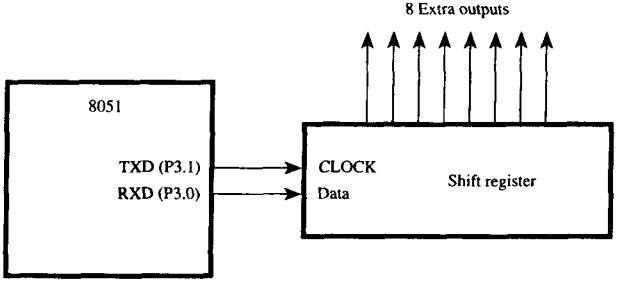


FIGURE 5–4
Serial port shift register mode

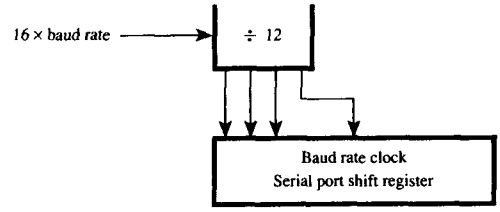


FIGURE 5–5
Serial port clocking

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 is 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.

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."

5.3.3 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 pro-

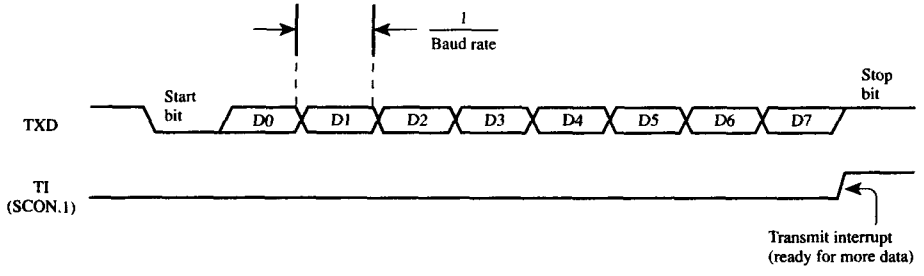


FIGURE 5–6
Setting the serial port TI flag

grammable 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 5.6 Serial Port Baud Rates.)

5.3.4 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).

5.4 INITIALIZATION AND ACCESSING SERIAL PORT REGISTERS

5.4.1 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.)

5.4.2 The 9th 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 is established. (The ninth data bit also plays an important role in multiprocessor communications. See below.)

5.4.3 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 requires 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 TB8
MOV  TB8,C            ; THIS BECOMES THE 9TH DATA BIT
MOV  SBUF,A           ; MOVE 8 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
```

5.4.4 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.

5.5 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 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

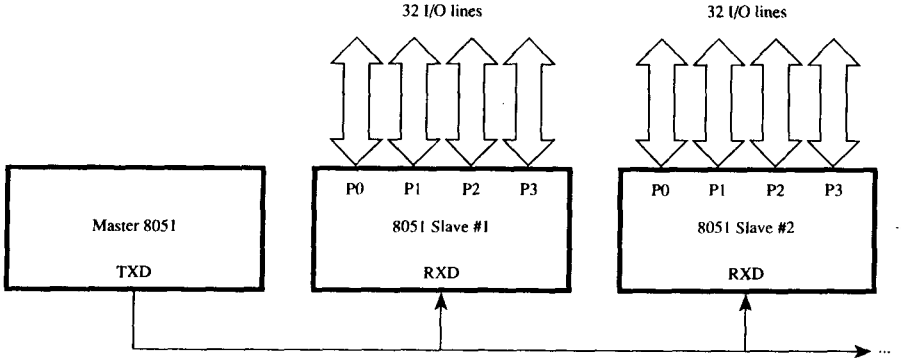


FIGURE 5–7
Multiprocessor communication

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.

5.6 SERIAL PORT BAUD RATES

As evident in Table 5–2, the baud rate is fixed in modes 0 and 2. 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. 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.)

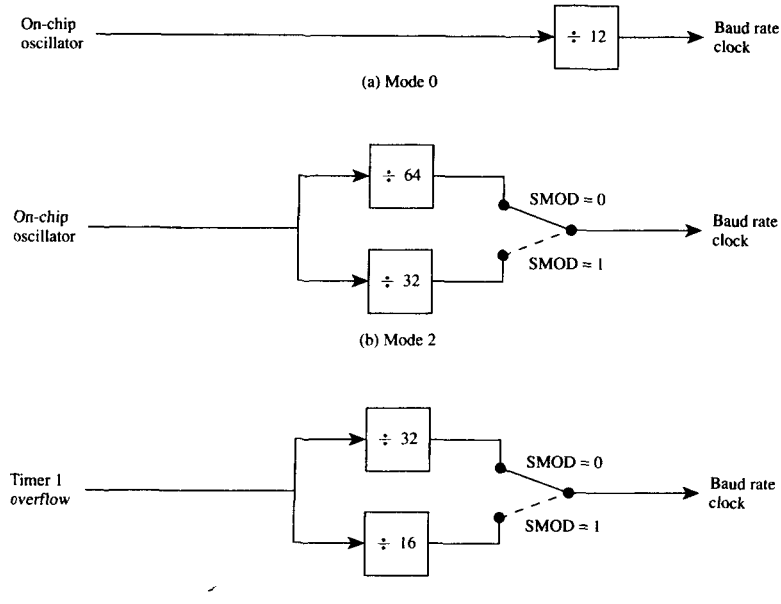


FIGURE 5–8
Serial port clocking sources. (a) Mode 0 (b) Mode 2 (c) Modes 1 and 3.

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.

5.6.1 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 = TIMER 1 OVERFLOW RATE ÷ 32
```

For example, 1200 baud operation requires an overflow rate calculated as follows:

```
1200 = TIMER 1 OVERFLOW RATE ÷ 32
TIMER 1 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 FFH-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 (F9H)	8923	7%
2400	12.000 MHz	0	−13 (F3H)	2404	0.16%
1200	12.000 MHz	0	−26 (E6H)	1202	0.16%
19200	11.059 MHz	1	−3 (FDH)	19200	0
9600	11.059 MHz	0	−3 (FDH)	9600	0
2400	11.059 MHz	0	−12 (F4H)	2400	0
1200	11.059 MHz	0	−24 (E8H)	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.

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.

The required TH1 value is that which provides overflows at the rate of 2400 × 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.

Example 5–1: 8051 Serial Port Example (initialize the serial port)

```
8100             5              ORG     8100H
8100 759852      6     INIT:    MOV     SCON,#52H   ;serial port, mode 1
8103 758920      7              MOV     TMOD,#20H   ;timer 1, mode 2
8106 758DF3      8              MOV     TH1,#-13    ;reload count for 2400 baud
8109 D28E        9              SETB    TR1         ;start timer 1
                10              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 8th bit. Return from the subroutine with the accumulator intact, i.e., containing the same value as before the subroutine was called.

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             5              ORG     8100H
8100 A2D0        6    OUTCHR:   MOV     C,P          ;put parity bit in C flag
8102 B3          7              CPL     C            ;change to odd parity
8103 92E7        8              MOV     ACC.7,C      ;add to character code
8105 3099FD      9    AGAIN:    JNB     TI,AGAIN     ;Tx empty? no: check again
8108 C299       10              CLR     TI           ;yes: clear flag and
810A F599       11              MOV     SBUF,A       ;     send character
810C C2E7       12              CLR     ACC.7        ;strip off parity bit and
810E 22         13              RET                  ;  return
                14              END
```

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-16 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.

```
8100             5              ORG     8100H
8100 3098FD      6    INCHAR:   JNB     RI,$         ;wait for character
8103 C298        7              CLR     RI           ;clear flag
8105 E599        8              MOV     A,SBUF       ;read char into A
```

```
8107 A2D0      9            MOV     C,P          ;for odd parity in A,
               10                                ;   P should be set
8109 B3        11           CPL     C            ;complementing correctly
               12                                ; indicates if "error"
810A C2E7      13           CLR     ACC.7        ;strip off parity
810C 22        14           RET
               15           END
```

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 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.

5.7 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.

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.
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).
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.
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. 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)
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.
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).
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.
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.

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, while cooking, the system must respond to manual input 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

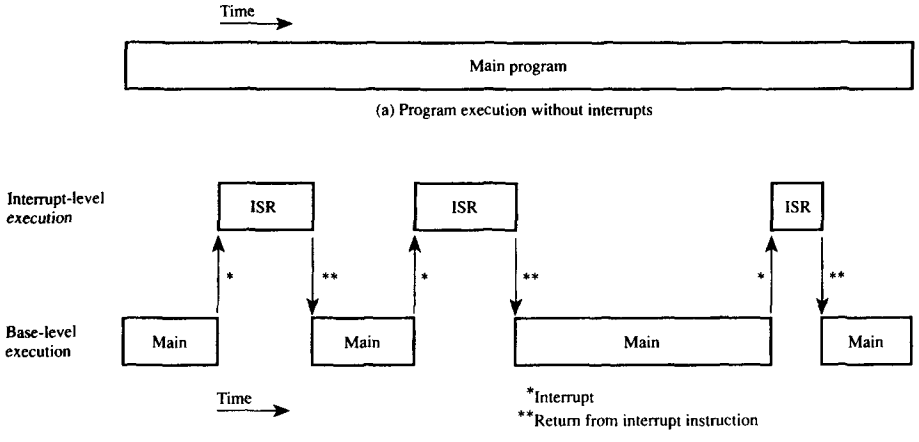


FIGURE 6–1
Program execution with and without interrupts. (a) Without interrupts (b) With interrupts.

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 individ-

ual enable 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.

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.

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

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

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 (RI) or a transmit interrupt (TI). Likewise, Timer 2 interrupts are generated by a timer overflow (TF2) or by the external input flag (EXF2). The flag bits that generate interrupts are summarized in Table 6–3.

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.

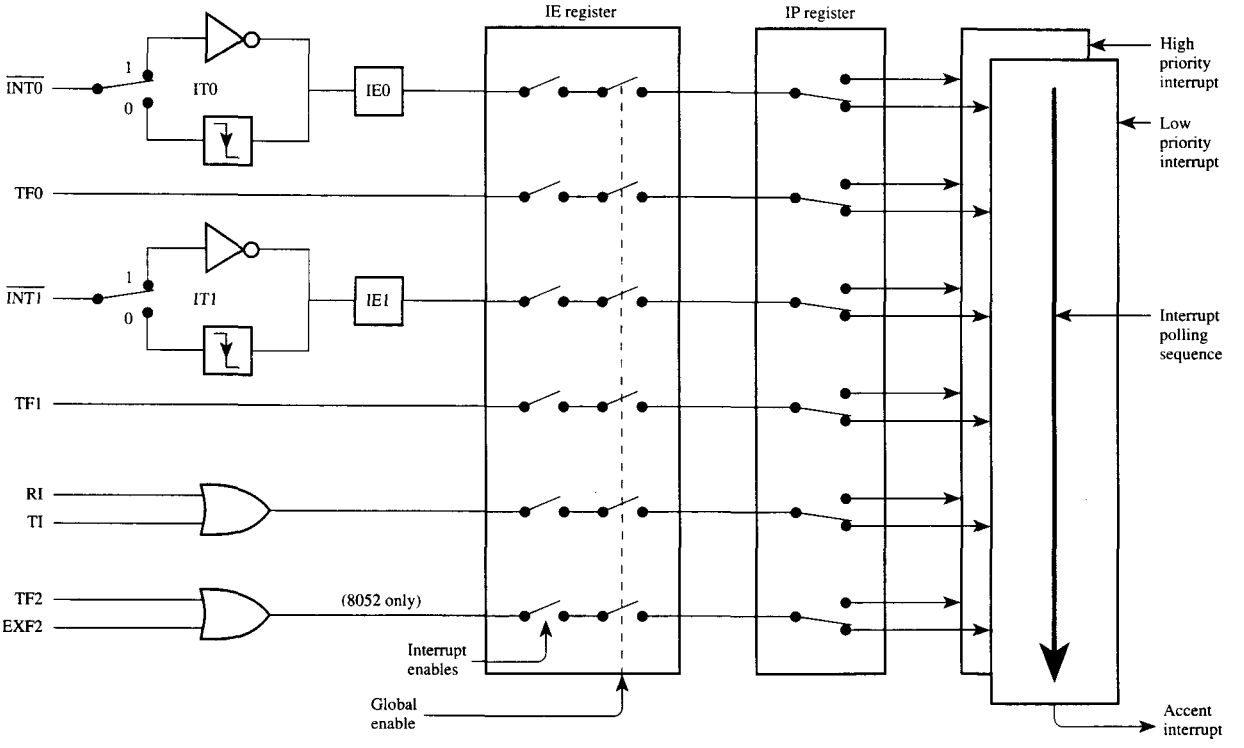


FIGURE 6–2
Overview of 8051 interrupt structure

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)

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.

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 inappropri-

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	TF1	001BH
Serial port	RI or TI	0023H
Timer 2	TF2 or EXF2	002BH

ate 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.

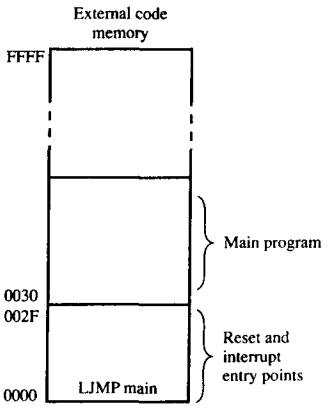


FIGURE 6–3
Memory organization when using interrupts

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
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  0000H              ;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 pro-

gram, and it continues doing nothing. This is not as farfetched as one might think. In many control-oriented applications, the bulk of the work is in fact done in the interrupt service routines.

Example 6–1 A Square Wave Using Timer Interrupts

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

Timer interrupts occur when the timer registers (TLx/THx) overflow and set the overflow flag (TFx). This example appears in Chapter 4 without using interrupts (see example). The bulk of the program is the same except it is now organized into the framework for interrupts. Here's the program:

```
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
               16               END
```

This is a complete program, which 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 IE,#82H instruction 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 and returns to the main program 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 vectors to the interrupt.

Incidentally, the return address in the main program is the location of the SJMP instruction. This address gets pushed on the 8051's internal stack prior to vectoring to each interrupt, and gets popped from the stack when the RETI instruction executes at the end of the ISR. Since the SP was not initialized, it defaults to its reset value of 07H. The push operation leaves the return address in internal RAM locations 08H (PC_L) and 09H (PC_H).

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.

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 and low-time of 1 ms, mode 2 cannot be used. (Recall that 256 μs is the maximum timed interval in mode 2 when the 8051 is operating at 12 MHz.) Here's the program:

```
0000            5               ORG     0
0000 020030     6               LJMP    MAIN
000B            7               ORG     000BH       ;Timer 0 vector address
000B 02003F     8               LJMP    T0ISR
001B            9               ORG     001BH       ;Timer 1 vector address
001B 020042    10               LJMP    T1ISR
0030           11               ORG     0030H
0030 758912    12      MAIN:    MOV     TMOD,#12H   ;Timer 1 = mode 1
               13                                   ;Timer 0 = mode 2
0033 758CB9    14               MOV     TH0,#-71    ;7 kHz using timer 0
0036 D28C      15               SETB    TR0
0038 D28F      16               SETB    TF1         ;force timer 1 interrupt
003A 75A88A    17               MOV     IE,#8AH     ;enable both timer intrrpts
003D 80FE      18               SJMP    $
               19      ;
003F B297      20      T0ISR:   CPL     P1.7
0041 32        21               RETI
0042 C28E      22      T1ISR:   CLR     TR1
0044 758DFC    23               MOV     TH1,#HIGH(-1000)  ;1 ms high time &
0047 758B18    24               MOV     TL1,#LOW(-1000)   ; low time
004A D28E      25               SETB    TR1
004C B296      26               CPL     P1.6
004E 32        27               RETI
               28               END
```

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, then

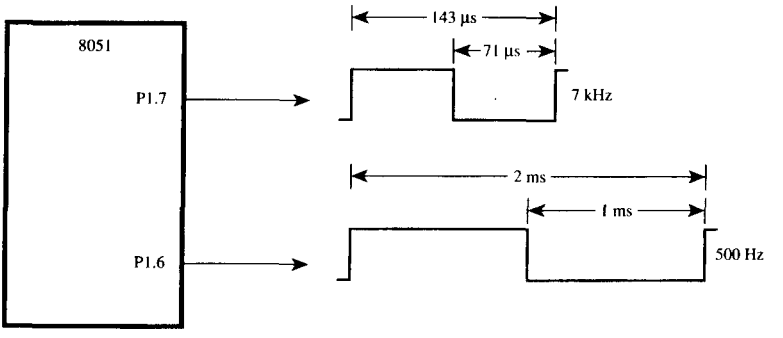


FIGURE 6–4
Waveform example

(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.)

6.5 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.

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
```

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 SPISR 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 × 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.

6.6 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 TCON 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 1 cycle, and then hold it low for at least 1 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.

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 off
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
```

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 (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 $\overline{\text{INT0}}$ makes a high-to-low transition.

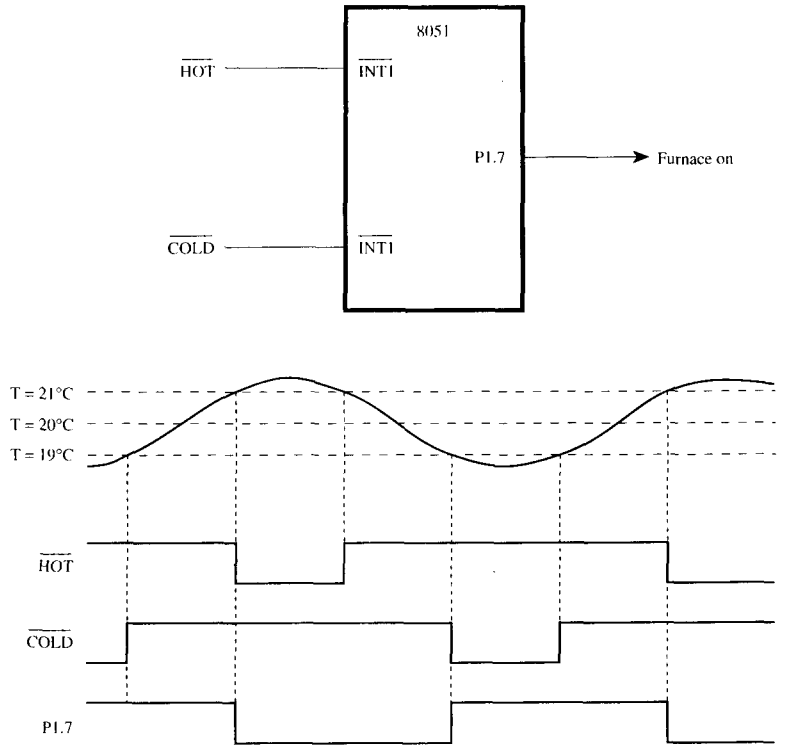


FIGURE 6–5
Furnace example. (a) Hardware connections (b) Timing.

The solution to this example uses three interrupts: external 0 (door sensor), Timer 0 (400 Hz tone), and Timer 1 (1 second timeout). The hardware configuration and timings are shown in Figure 6–6.

```
0000              5              ORG     0
0000 020030       6              LJMP    MAIN        ;3-byte instruction
0003 02003A       7              LJMP    EX0ISR      ;EXT 0 vector address
000B              8              ORG     000BH       ;Timer 0 vector
000B 020045       9              LJMP    T0ISR
001B             10              ORG     001BH       ;Timer 1 vector
001B 020059      11              LJMP    T1ISR
0030             12              ORG     0030H
0030 D288        13     MAIN:    SETB    IT0         ;negative edge activated
0032 758911      14              MOV     TMOD,#11H   ;16-bit timer mode
```

```
0035 75A881  15              MOV     IE,#81H      ;enable EXT 0 only
0038 80FE    16              SJMP    $            ;now relax
             17      ;
003A 7F14    18      EX0ISR: MOV     R7,#20       ;20 x 5000 us = 1 second
003C D28D    19              SETB    TF0          ;force timer 0 interrupt
003E D28F    20              SETB    TF1          ;force timer 1 interrupt
0040 D2A9    21              SETB    ET0          ;begin tone for 1 second
0042 D2AB    22              SETB    ET1          ;enable timer interrupts
0044 32      23              RETI                 ;timer ints will do the work
             24      ;
0045 C28C    25      T0ISR:  CLR     TR0          ;stop timer
0047 DF07    26              DJNZ    R7,SKIP      ;if not 20th time, exit
0049 C2A9    27              CLR     ET0          ;if 20th, disable tone
004B C2AB    28              CLR     ET1          ;disable itself
004D 020058  29              LJMP    EXIT
0050 758C3C  30      SKIP:   MOV     TH0,#HIGH(-50000)  ;0.05 sec. delay
0053 758AB0  31              MOV     TL0,#LOW(-50000)
0056 D28C    32              SETB    TR0
0058 32      33      EXIT:   RETI
             34      ;
0059 C28E    35      T1ISR:  CLR     TR1
005B 758DFB  36              MOV     TH1,#HIGH(-1250)   ;count for 400 Hz
005E 758B1E  37              MOV     TL1,#LOW(-1250)
0061 B297    38              CPL     P1.7         ;music maestro!
0063 D28E    39              SETB    TR1
0065 32      40              RETI
             41              END
```

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 RET1 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/TL0 is reloaded with −50000, the timer is turned back on and the interrupt is terminated. On the 20th Timer 0 interrupt, R7 is decremented to 0 (1 second

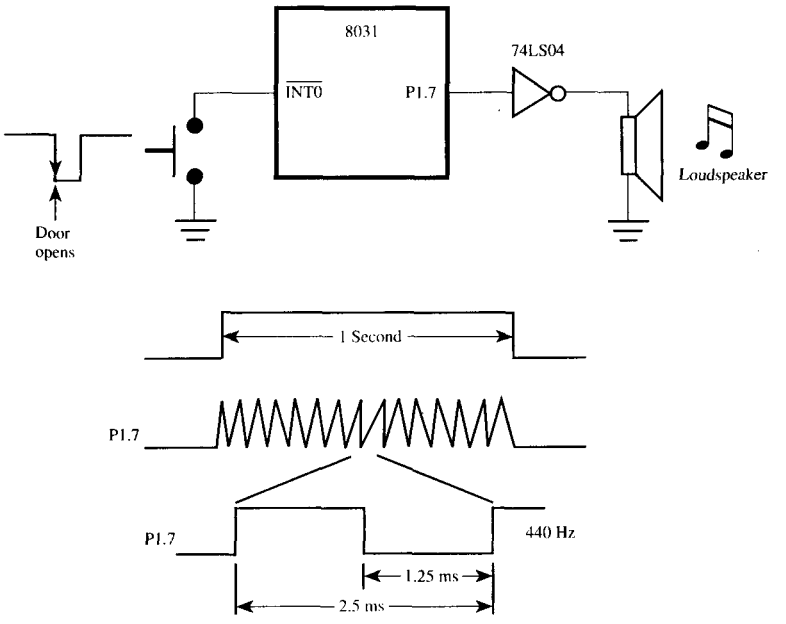


FIGURE 6–6
Loudspeaker interface using interrupts. (a) Hardware connections (b) Timing.

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.

The 400 Hz tone is programmed using Timer 1 interrupts. 400 Hz requires a period of 1/400 = 2,500 μs, or 1,250 μs high-time and 1,250 μs low-time. Each timer 1 ISR simply puts −1250 in TH1/TL1, complements the port bit driving the loudspeaker, then terminates.

6.7 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.

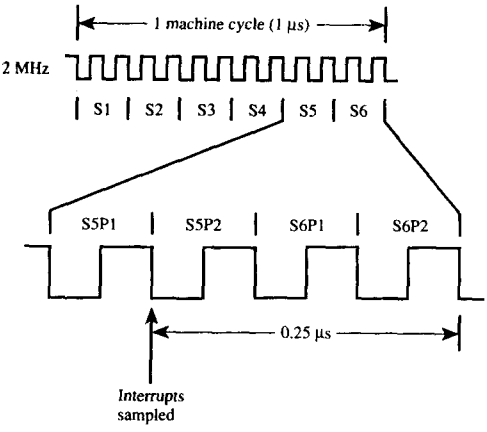


FIGURE 6–7
Sampling of interrupts on S5P2

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).

6.8 SUMMARY

This chapter has presented the major details required to embark on the design of interrupt-driven systems with the 8051 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.

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

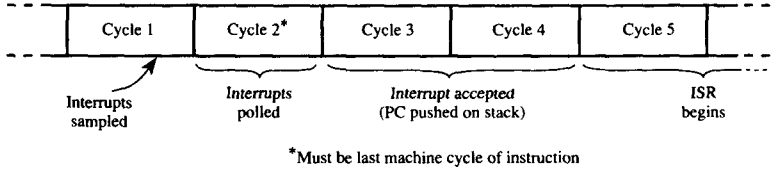


FIGURE 6–8
Polling of interrupts

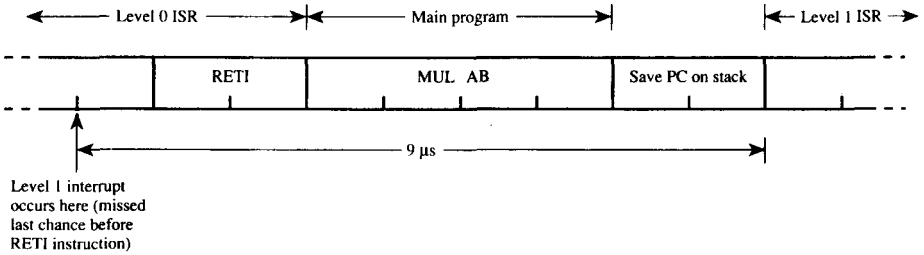


FIGURE 6–9
Interrupt latency

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 (0003H and 0004H) 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 0000H. 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.

PROBLEMS

1. Modify Example 1 to shut off interrupts and terminate if any key is hit on the terminal.
2. Create a 1 kHz square wave on P1.7 using interrupts.
3. Create a 7 kHz pulse wave with a 30% duty cycle on P1.6 using interrupts.
4. Combine Example 6–1 and Example 6–3 (earlier in the chapter) into one program.
5. Modify Example 6–3 to send one character per second. (Hint: use a timer and output the character in the timer ISR.)

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™family assembler, ASM51, is 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 emulation 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 9.

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 re-

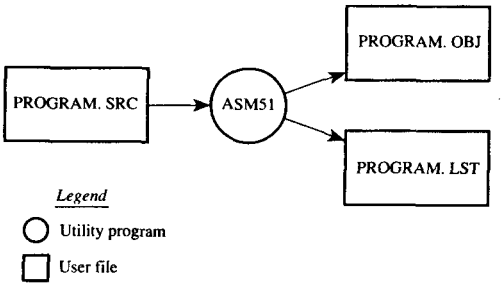


FIGURE 7–1
Assembling a source program

ceives 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 (DB or DW) increment the location counter by the number of bytes defined. Reserve memory directives (DS) 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 (EQU) 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 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, when branching ahead in a program.

The object file, if it is absolute, contains only the binary bytes (00H–FFH 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.

A sketch of a two-pass assembler is shown in Figure 7–2 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 con-

```
ASM(input_file) /* assemble source program in input_file */

BEGIN

   /* pass 1: build the symbol table */

   [lc = 0]              /* lc = location counter; default to 0 */
   [mnemonic = null]
   [open input_file]
   WHILE [mnemonic != end] DO BEGIN
      [get line from input_file]
      [scan line and get label/symbol and mnemonic]
      IF [label] THEN [enter "label = lc" into symbol table]
      CASE [mnemonic] OF
         null, comment, END:
               [do nothing]
         ORG:  [lc = operand]
         EQU:  [enter "symbol = operand" into symbol table]
         DB:   [increment lc by number of bytes defined]
         DW:   [increment lc by twice the number of words defined]
         DS:   [lc = lc + operand]
         1_byte_instruction:    [lc = lc + 1]
         2_byte_instruction:    [lc = lc + 2]
         3_byte_instruction:    [lc = lc + 3]
   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 from input_file]
      [scan line and determine mnemonic op code and value(s) of operand(s)]

      /*   Note: If symbols are used in operand field,
       *   their values are looked-up in the symbol table
       *   created during pass one.
       */

      CASE [mnemonic] OF
         null, comment, EQU, END:
               [do nothing]
         ORG:  [lc = operand]
         DB:   [put bytes into object_file and increment lc by # of bytes]
         DW:   [put words into object_file and inc lc by twice # of words]
         DS:   [lc = lc + operand]
         1_byte_instruction:    [put opcode into output_file]
         2_byte_instruction:    [put opcode into output_file]
                                [put low-byte of operand into output_file]
         3_byte_instruction:    [put opcode into output_file]
                                [put high-byte of operand into output_file]
                                [put low-byte of operand into output_file]
   END
   [close input_file]
   [close output_file]

END
```

FIGURE 7–2
Pseudo code sketch of a two-pass assembler

stants, 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, digits, "?", or "_"; and can contain up to 31 characters.[1] Symbols may use upper- or lower-case 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 in the 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 in the comment field at the end of each line. Comments must begin with a semicolon (;). Entire lines may be comment lines by beginning them with 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. As well, a dollar sign ($) can be used to refer to the current value of the location counter. Some examples follow.

---

[1] The reader is reminded that the rules specified in this chapter apply to Intel's ASM51. Other assemblers may have different requirements.

```
        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 FFH) 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 using signed decimal notation, constants from −256 to +256 may 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 2K block as the destination instruction. If short or absolute forms cannot be used, the conversion is to the long form.

```
MCS-51 MACRO ASSEMBLER    GENERIC

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN GENERIC.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE GENERIC.SRC EP

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 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 (12FCH) 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 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, "O" 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 ('). 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,#10H
```

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 (FFFFH). 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 FFFFH and the 8-bit value FFH 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. 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 (FE0CH); then the HIGH and LOW operators extract the high (FEH) 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' SHL 8	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, BSEG, 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,#00011000B  ;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–FFH 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  0DH                 ;SET CR (CARRIAGE RETURN) TO 0DH
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; however, if used 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,A
            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 XBUFFER + LENGTH + 1. (The "+1" is needed because the data pointer is incremented *after* the last MOVX instruction.)

7.5.3.2 DBIT

The format for the DBIT (define bit) directive is,

```
[label:]    DBIT expression
```

The DBIT 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 DBIT 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:     DBIT 1        ;KEYBOARD STATUS
PRFLAG:     DBIT 1        ;PRINTER STATUS
DKFLAG:     DBIT 1        ;DISK STATUS
```

Since an address is not specified with BSEG in the example above, the address of the flags defined by DBIT 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 below, MAIN.SRC and MESSAGES.SRC. The subroutines HELLO and GOOD_BYE 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_BYE)
                    . . .
                    CALL    HELLO
                    . . .
                    CALL    GOOD_BYE
                    . . .
                    END
```

MESSAGES.SRC

```
                    PUBLIC HELLO,GOOD_BYE
                    . . .
HELLO:              (begin subroutine)
                    . . .
                    RET
GOOD_BYE:           (begin subroutine)
                    . . .
                    RET
                    . . .
                    END
```

Neither MAIN.SRC nor MESSAGES.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.

```
LOC  OBJ            LINE    SOURCE

                       1    ONCHIP      SEGMENT     DATA      ;A RELOCATABLE DATA SEGMENT
                       2    EPROM       SEGMENT     CODE      ;A RELOCATABLE CODE SEGMENT
                       3
----                   4                BSEG        AT 70H    ;BEGIN ABSOLUTE BIT SEGMENT
0070                   5    FLAG1:      DBIT        1
0071                   6    FLAG2:      DBIT        1
                       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 EPROM SEGMENT
0000 750000    F      14    BEGIN:      MOV         TOTAL,#0
                      15    ;           (continue program)
                      16                END
```

FIGURE 7–4
Defining and initiating absolute and relocatable segments

The ORG directive may be used to change the location counter within the currently selected segment. Figure 7–4 shows examples of defining and initiating relocatable and absolute segments.

The first two lines in Figure 7–4 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 (2-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 instruction in this instance of the EPROM. Note that it is not possible to determine the address of the labels TOTAL, COUNT, SUM16, and BEGIN from Figure 7–4. 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 one.

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 7–5 shows the assembler controls supported by ASM51.

NAME	PRIMARY/ GENERAL	DEFAULT	ABBREV.	MEANING
DATE(date)	P	DATE()	DA	Places string in header (9 char. max.)
DEBUG	P	NODEBUG	DB	Outputs debug symbol information to object file
NODEBUG	P	NODEBUG	NODB	Symbol information not placed in object file
EJECT	G	not applicable	EJ	Continue listing on next page
ERRORPRINT(file)	P	NOERRORPRINT	EP	Designates a file to receive error messages in addition to the listing file (defaults to console)
NOERRORPRINT	P	NOERRORPRINT	NOEP	Designates that error messages will be printed in listing file only
GEN	G	GENONLY	GE	Generates a full listing of the macro expansion process including macro calls in the listing file
GENONLY	G	GENONLY	GO	List only the fully expanded source as if all lines generated by a macro call were already in the source file
NOGEN	G	GENONLY	NOGE	List only the original source text in the listing file
INCLUDE(file)	G	not applicable	IC	Designates a file to be included as part of the program
LIST	G	LIST	LI	Print subsequent lines of source code in listing file
NOLIST	G	LIST	NOLI	Do not print subsequent lines of source code in listing file
MACRO[(mem_percent)	P	MACRO(50)	MR	Evaluate and expand all macro calls. Allocate percentage of free memory for macro processing
NOMACRO	P	MACRO(50)	NOMR	Do evaluate macro calls
MOD51	P	MOD51	MO	Recognize the 8051-specific predefined special registers
NOMOD51	P	MOD51	NOMO	Do not recognize the 8051-specific predefined special registers
OBJECT[(file)]	P	OBJECT(source.OBJ)	OJ	Designate 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 will 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	PAGELENGTH(60)	PL	Sets maximum number of lines in each page of listing file (range = 10 to 65,536)
PAGEWIDTH(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 setting for LIST and GEN
RESTORE	G	not applicable	RS	Restores control setting 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 banks are used
SYMBOLS	P	SYMBOLS	SB	Creates a formatted table of all symbols used in program
NOSYMBOLS	P	SYMBOLS	NOSB	No symbol table created
TITLE(string)	G	TITLE()	TT	Places a string in all subsequent page headers (maximum 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	No cross reference list created

FIGURE 7–5
Assembler controls supported by ASM51

7.7 LINKER OPERATION

When 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 7–6.

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_file][location_controls]
```

The input_list is a list of relocatable object modules (files) separated by commas. The output_file 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 2 relocatable segments, one called EPROM of type CODE, and the other called ONCHIP of type DATA. Suppose further that the code segment 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

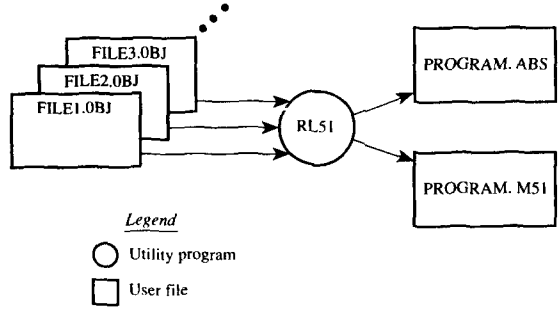


FIGURE 7–6
Linker operation

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 $DEBUG control. (See Figure 7–5.)

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 inter-file 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 7–7 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 7–7a 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 7–5.)

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 7–5.) $DEBUG in line 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. $PAGEWIDTH specifies the maximum width of each line in the listing file. $NOPAGING prevents page breaks (form feeds) from being inserted in the listing file. Most assembler controls affect the look of the output listing file. Some trial-and-error will usually produce the desired output for printing.

```
MCS-51 MACRO .   (BLER    *** ANNOTATED EXAMPLE (MAIN MODULE) ***          03/17/91          (a)

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 636F6D6D
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 7–7
Annotated example: linking relocatable segments and modules. (a) ECHO.LST (b) IO.LST (c) EXAMPLE.M51.

```
MCS-51 MACRO ASSEMBLER    *** ANNOTATED EXAMPLE (SUBROUTINES MODULE) ***      05/17/91          (b)

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      0DH          ;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 7–7
continued

```
                        55
                        56      ;*******************************************************************
                        57      ; INPUT CHARACTERS TO BUFFER                                      *
                        58      ;*******************************************************************
0027 7800       F       59      INLINE:     MOV     R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0029 120000     F       60      AGAIN:      CALL    INCHAR      ;GET A CHARACTER
002C 120000     F       61                  CALL    OUTCHR      ; ECHO IT BACK
002F F6                 62                  MOV     @R0,A       ;PUT IT IN BUFFER
0030 08                 63                  INC     R0          ;INCREMENT POINTER TO BUFFER
0031 B40DF5             64                  CJNE    A,#CR,AGAIN ;IF NOT CR, GET ANOTHER CHAR
0034 7600               65                  MOV     @R0,#0      ;PUT NULL BYTE AT END
0036 22                 66                  RET
                        67
                        68      ;*******************************************************************
                        69      ; OUTPUT CONTENTS OF BUFFER                                       *
                        70      ;*******************************************************************
0037 7800       F       71      OUTLINE:    MOV     R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0039 E6                 72      AGAIN2:     MOV     A,@R0       ;GET CHARACTER FROM BUFFER
003A 6006               73                  JZ      EXIT2       ;IF NULL BYTE, DONE
003C 120000     F       74                  CALL    OUTCHR      ;OTHERWISE, SEND IT
003F 08                 75                  INC     R0          ;POINT TO NEXT CHAR IN BUFFER
0040 80F7               76                  JMP     AGAIN2      ; AND SEND IT TOO
0042 22                 77      EXIT2:      RET
                        78
                        79      ;*******************************************************************
                        80      ; CREATE A BUFFER IN ONCHIP RAM                                   *
                        81      ;*******************************************************************
----                    82                  RSEG    ONCHIP      ;BEGIN RELOCATABLE DATA SEGMENT
0000                    83      BUFFER:     DS      LENGTH      ;ALLOCATE INTERNAL RAM AS BUFFER
                        84                  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 7–7
continued

```
DATE : 03/17/91                                                    (c)
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 7–7
continued

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. Subsequent 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 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 7–7b 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 7–7c 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.M51 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

```
                %PUSH_DPTR
```

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))
              (CJNE   A,#%VALUE,$ + 3
              )
```

then the macro call

```
              %CMPA#(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 pro-

grammers. 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 "jump 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)
```

ASM51 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

```
            %DEC_DPTR
```

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
            (PUSH  ACC
            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
(INCHAR:    .                           ;8051 DRIVERS
            .
OUTCHR:     .
            .
            ) ELSE
(INCHAR:    .                           ;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 OUTCHR subroutines are used without consideration for the particular hardware configuration. As long as the program was assembled with the correct value for INTERNAL, the correct subroutine is executed.

PROBLEMS

1. Recast the following instructions with the operand expressed in binary.

```
MOV  A,#255
MOV  A,#11Q
MOV  A,#1AH
MOV  A,#'A'
```

2. What is wrong with the coding of the following instruction?

```
ORL  80H,#F0H
```

3. Identify the error in the following symbols.

```
?byte.bit
@GOOD_bye
1ST_FLAG
MY_PROGRAM
```

4. Recast the following instructions with the expression evaluated as a 16-bit hexadecimal constant.

```
MOV  DPTR,#'0' EQ 48
MOV  DPTR,#HIGH 'AB'
MOV  DPTR,#-1
MOV  DPTR,#NOT (257 MOD 256)
```

5. What are the "segment types" defined by ASM51 for the 8051, and what memory spaces do they represent?
6. 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 "XBUFFER.")
7. A certain application requires five status bits (FLAG1 to FLAG5). How could a 5-bit buffer be defined in an absolute BIT segment starting at bit address 08H? At what byte address do these bits reside?
8. What are two good reasons for making generous use of the EQU directive in assembly language programs?
9. What is the difference between the DB and DW directives?
10. What are the memory assignments for the following assembler directives:

```
ORG  OFH
DW   $ SHL 4
DB   65535
DW   '0'
```

11. What directive is used to select an absolute code segment?
12. A file called "ASCII" contains 33 equate directives, 1 for each control code:

```
NUL          EQU  00H                    ;NULL BYTE
SOH          EQU  01H                    ;START OF HEADER
.
.
.
US           EQU  1FH                    ;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?

13. In order for a printout of a listing file to look nice, it is desirable to have each subroutine begin at the top of a page. How is this accomplished?
14. 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.
15. 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

JLE—jump to LABEL if accumulator is less than or equal to VALUE

JOR—jump to LABEL if accumulator is outside the range LOWER and UPPER

16. 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.

8

PROGRAM STRUCTURE AND DESIGN

8.1 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 the idea 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. 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 the use of structured techniques.

[1]In high-level languages, programs are composed of functions or procedures.

9

TOOLS AND TECHNIQUES FOR PROGRAM DEVELOPMENT

9.1 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.

9.2 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 9–1. The reader may notice that 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 cor-

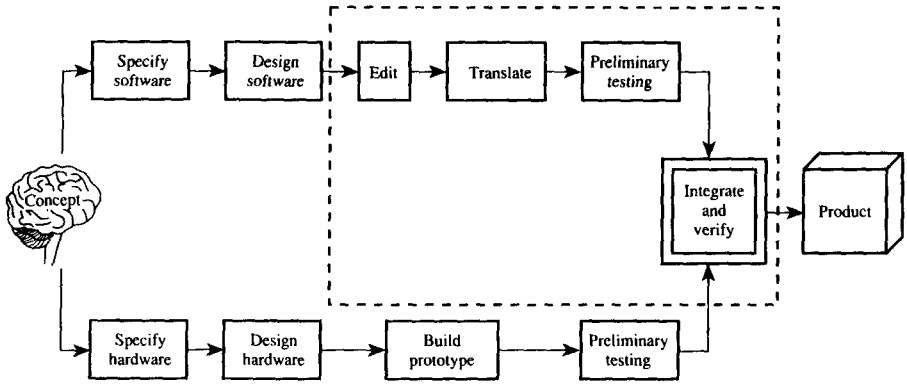


FIGURE 9–1
The development cycle

rections 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 9–1 from the output of any step in the development cycle to any earlier step.

The steps along the top path in Figure 9–1 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 9–1 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.

9.2.1 Software Development

The steps in the top path in Figure 9–1 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 8.

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.

9.2.2 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 boards (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 (≈1 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.

9.3 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 9–1. (See Figure 9–2.)

Figure 9–2 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 9–1) 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 9–2 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.

9.3.1 Software Simulation

Five execution environments are shown in Figure 9–2. Preliminary testing (see Figure 9–1) proceeds in the absence of the target system. This is shown in Figure 9–2 as software simulation. A **simulator** is a program that executes on the development system and

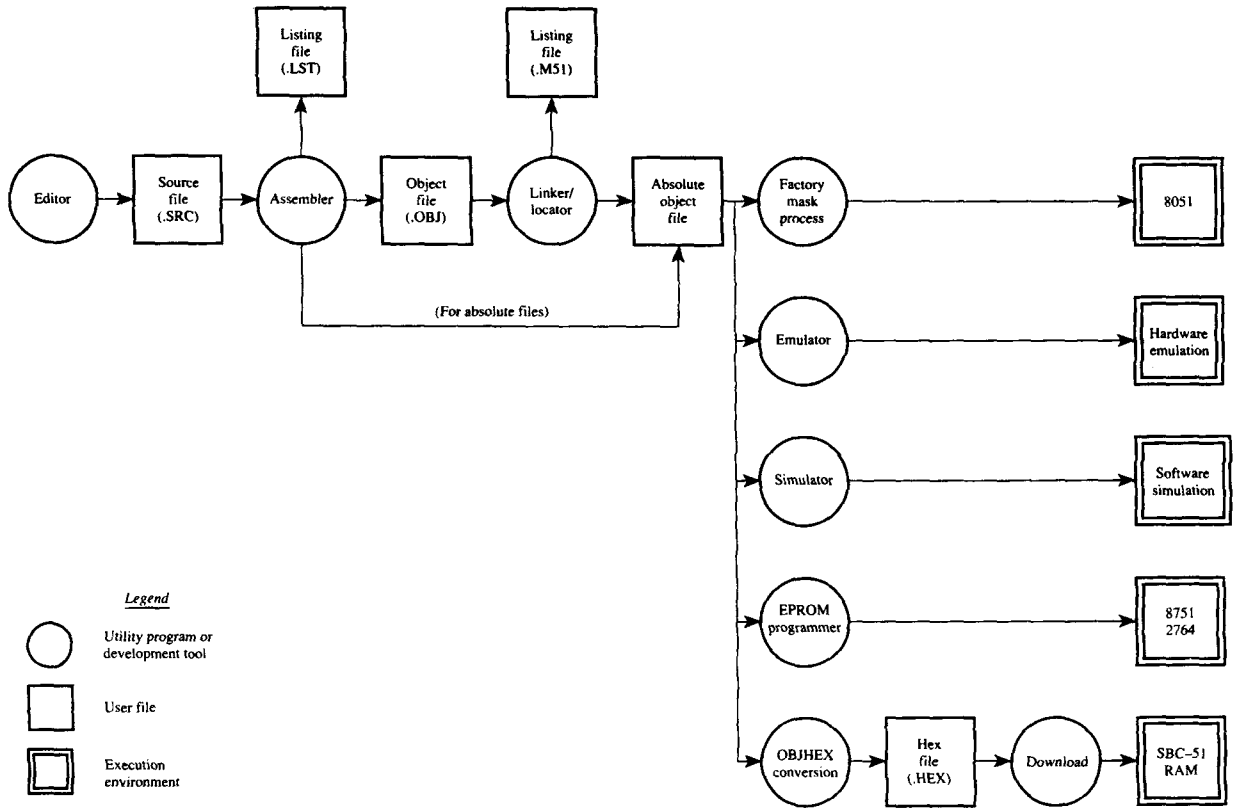


FIGURE 9–2
Detailed steps in the development cycle

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 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.

9.3.2 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.

9.3.3 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., Overlapping the External Code and Data Spaces), 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 9–2, a 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 9–3. 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.

9.3.4 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 9–2 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).

9.3.5 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

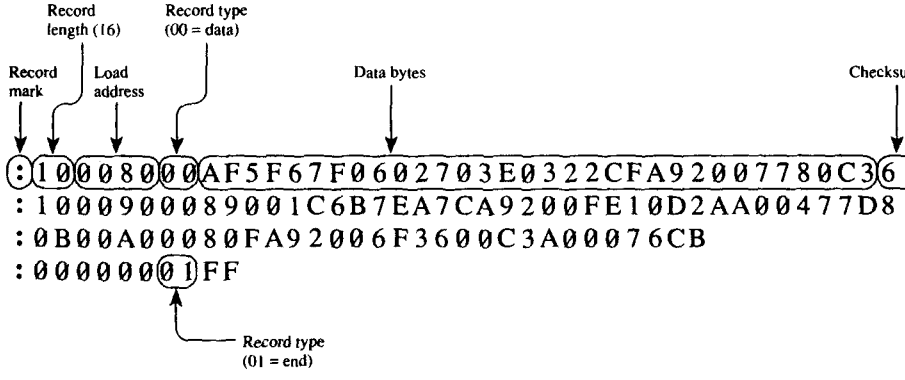


FIGURE 9–3
Intel hexadecimal format

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 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
```

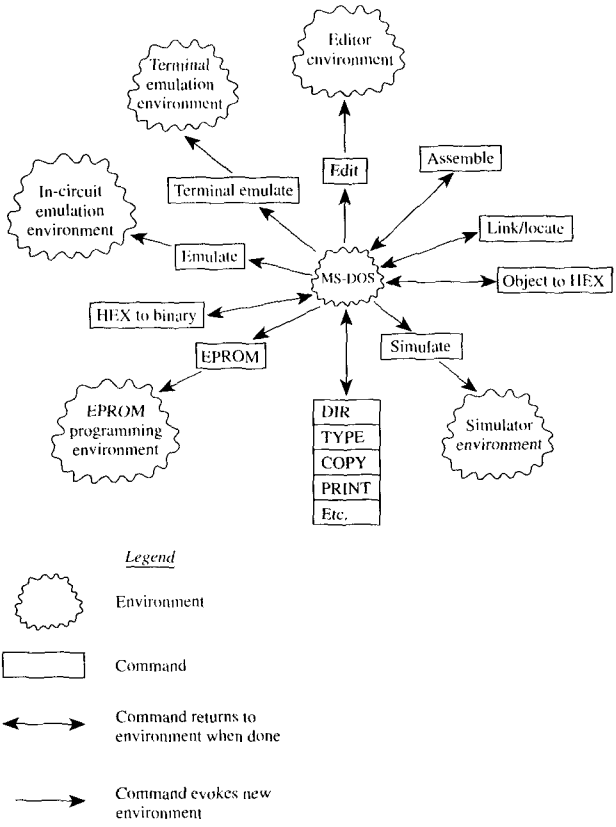
A production run of 1000 units would not justify use of the 8051—or so it seems. The use of external EPROM means that Ports 0 and 2 are unavailable for I/O. This may be a critical point that prevents the 8031 + 2764 approach. Even if the loss of on-chip I/O is not a concern, other factors enter. The 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.

9.4 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 9–4, 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).

FIGURE 9–4
The development environment



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.

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 9–4, 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.

9.5 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

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?
2. Below is an 8051 program in Intel hex format.

```
:100800007589117F007E0575A88AD28FD28D80FEF2
:10081000C28C758C3C758AB0DE087E050FBF09025C
:100820007F00D28C32048322FB90FC0CFC7AFCAD5E
:0A083000FD0AFD5CFDA6FDC8FDC831
:00000001FF
```

(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?

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
```

4. The contents of an Intel hex file are shown below.

```
:0901000078207655088880FA2237
:00000001FF
```

Recreate the original source program that this file represents.

10

DESIGN AND INTERFACE EXAMPLES

10.1 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.

10.2 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 10–1 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.

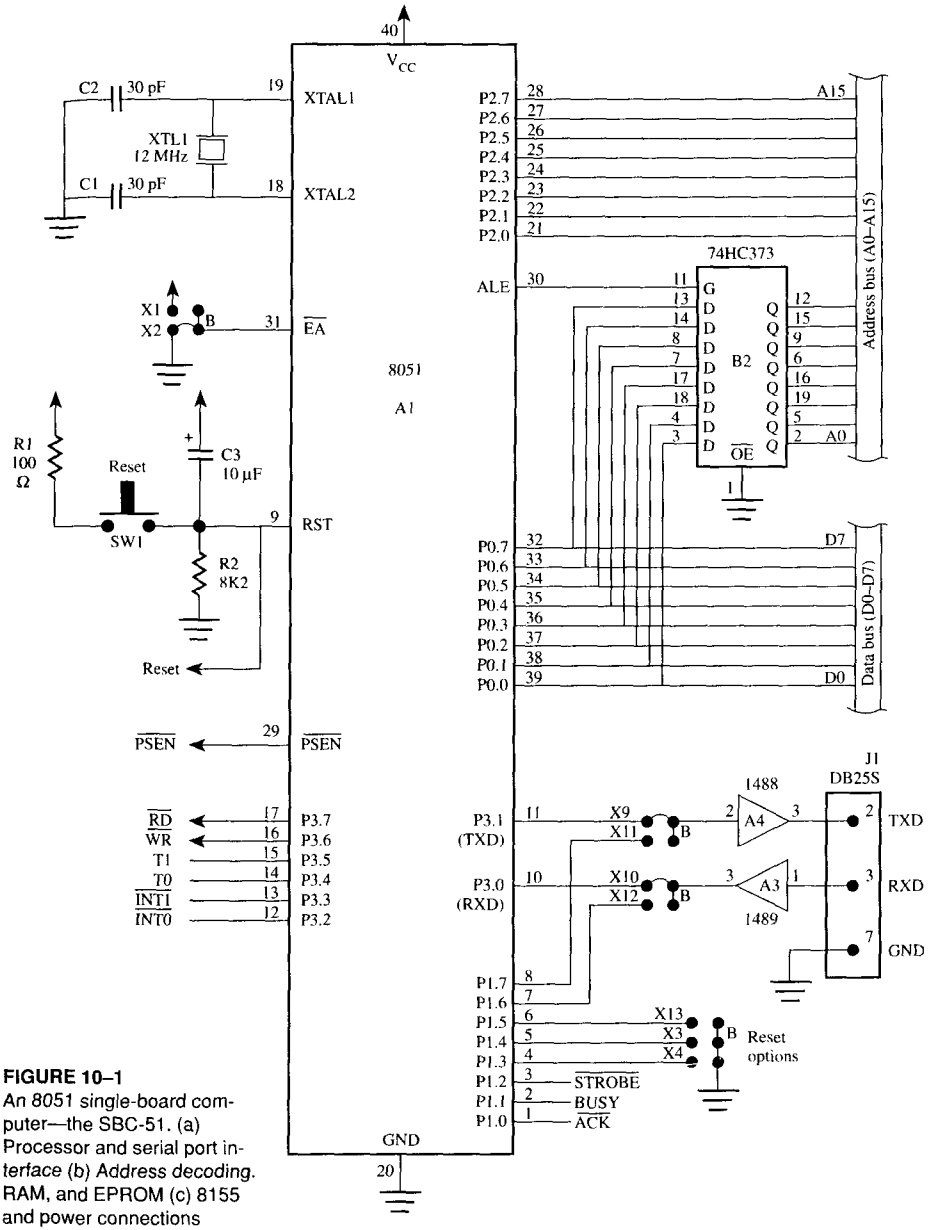


FIGURE 10–1
An 8051 single-board computer—the SBC-51. (a) Processor and serial port interface (b) Address decoding. RAM, and EPROM (c) 8155 and power connections

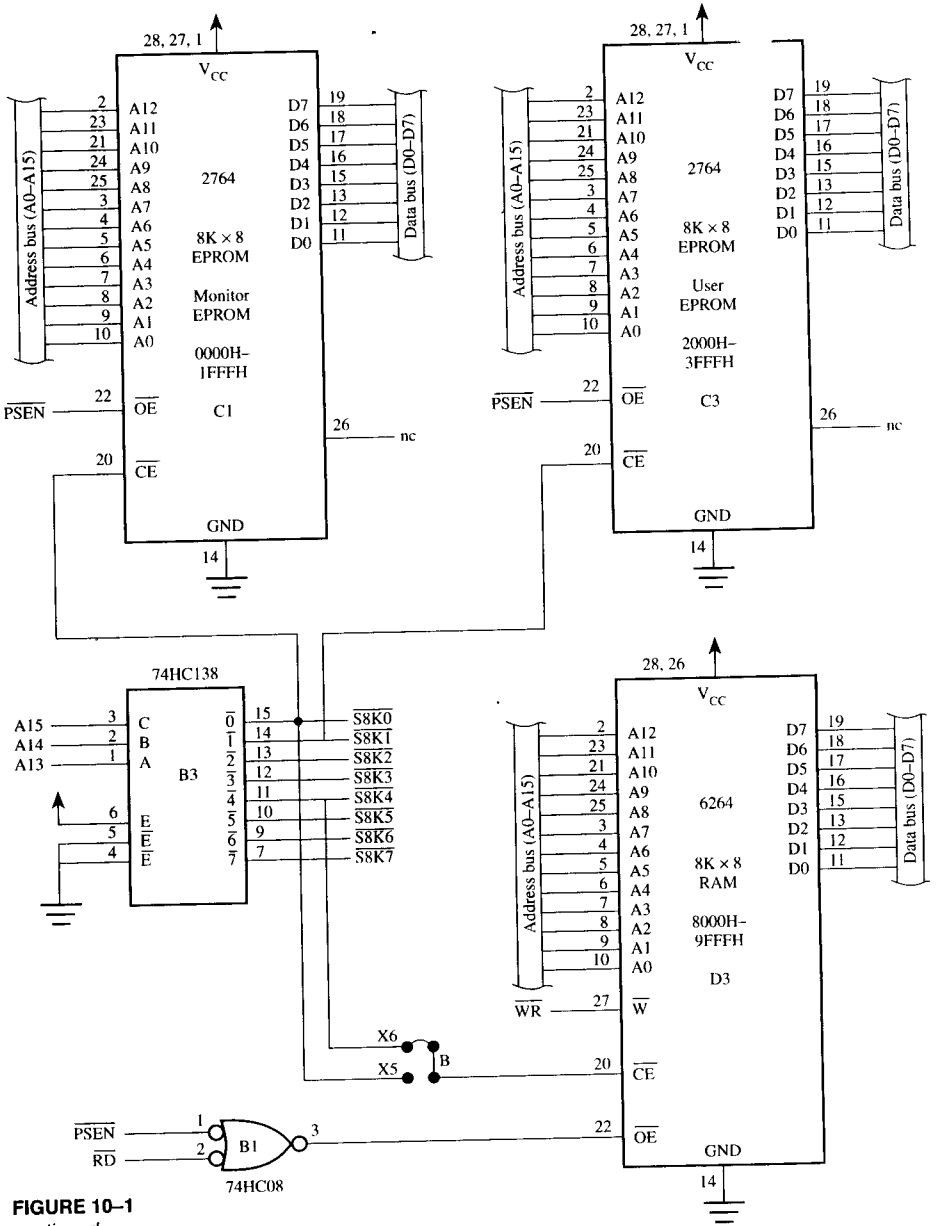
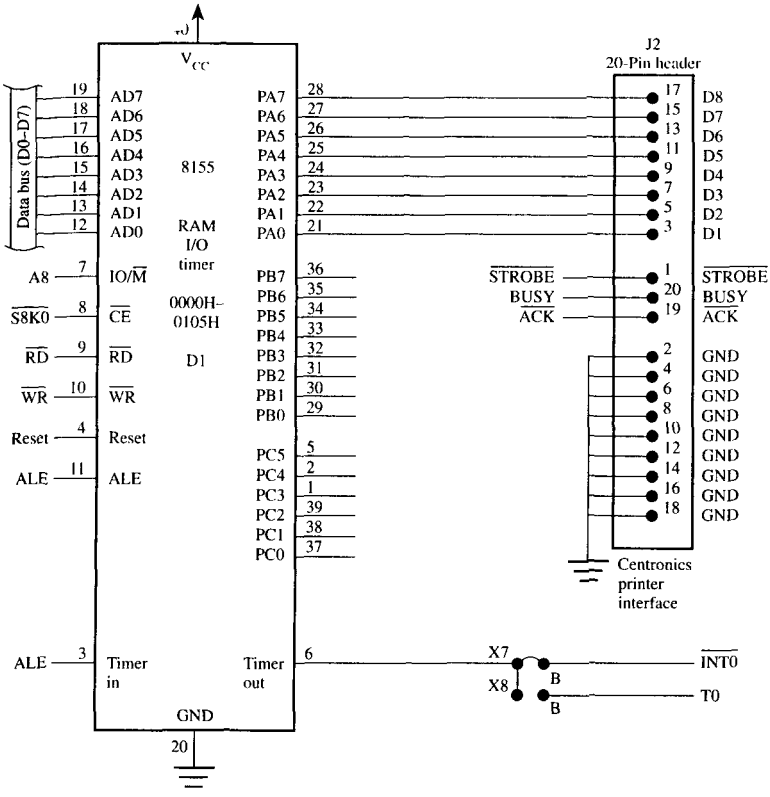


FIGURE 10–1
continued



Unmarked power connections

IC	+5V	GND	+12V	−12V
74HC08	14	7	–	–
74HC138	16	8	–	–
74HC373	20	10	–	–
1488	–	7	14	1
1489	14	7	–	–

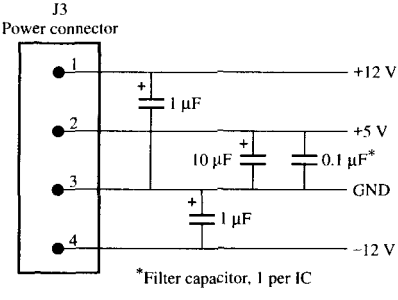


FIGURE 10–1
continued

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 10–1 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 0 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 2.6 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 $\overline{EA}$ (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 74HC138 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 $\overline{S8K0}$ (for "select 8K block 0") through to $\overline{S8K7}$. 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 10–1. The first (labeled "MONITOR EPROM") is selected by $\overline{S8K0}$ and resides in the external code space from address 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 $\overline{S8K1}$ for execution at addresses 2000H to 3FFFH. This IC is intended for user applications and is not needed for basic system op-

eration. Note that both EPROMs are selected only if $\overline{CE}$ (chip enable; pin 20) is active (or low) and $\overline{OE}$ is also active (or low). $\overline{OE}$ is driven by the 80C31's $\overline{PSEN}$ line; thus selection is in the external code space, as expected.

The 6264 8K by 8 RAM IC is selected by $\overline{S8K4}$ (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 2.6.4, Overlapping the External Code and Data Spaces.) This 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 $\overline{S8K0}$, 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 $\overline{RD}$ and $\overline{WR}$.

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/$\overline{M}$ 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
00FFH	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,#0101H          ;DPTR points to 8155 Port A
MOVX A,@DPTR              ;read Port A into Acc
```

To program Port A and Port B as outputs, 1s 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,#00000010B         ;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,A              ;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 10–1. 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 10-2).[1]

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).

10.3 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 for complex applications. Examples include the user interface to microwave ovens or automated banking machines. Figure 10–3 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.

[1]The printed-circuit board version of the SBC-51 is available from URDA, Inc., 1811 Jancey St., Suite #200, Pittsburgh, PA, USA, 15206.

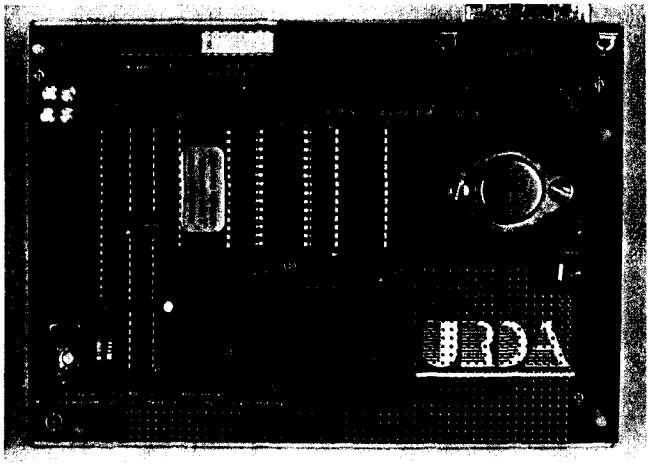
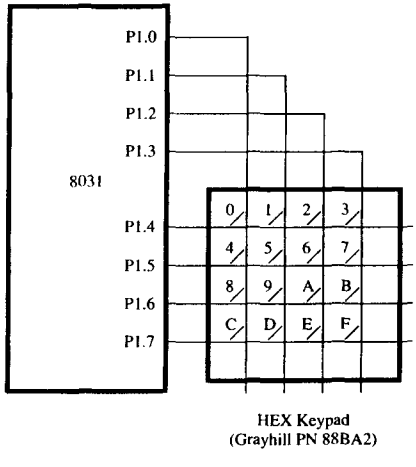


FIGURE 10–2
The printed-circuit board version of the SBC-51. (Courtesy URDA, Inc.)

FIGURE 10–3
Interface to hexadecimal keypad



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 10–4 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 10–4, 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 keypad 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 10–4 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.

10.4 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 3–5.) 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 demonstrate

```
LOC   O'          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 10–4
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 10–4
continued

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 Ω current-limiting resistors and four 2N3904 transistors. Figure 10–5 shows the connections between the 80C51, the MC14499, and the four 7-segment LEDs.

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 10–6. 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 OUT8. 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 10–6 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)
- Definition of symbols (31–38)
- Define storage declarations (lines 40–42)
- Jump table for program and interrupt entry points (lines 44–51)

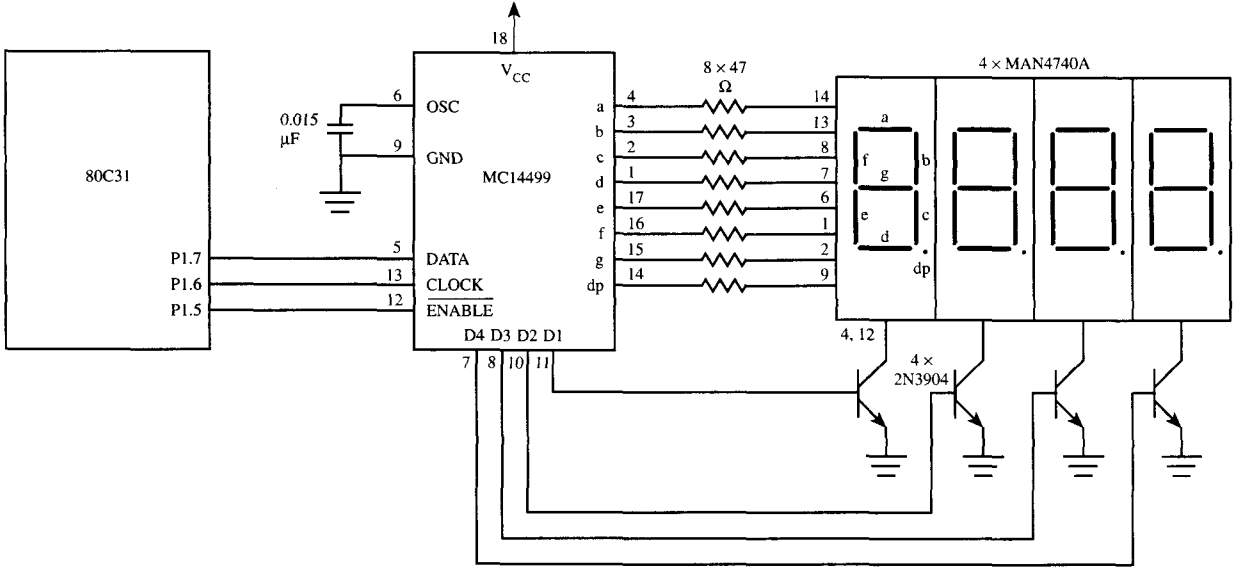


FIGURE 10–5
Interface to MC14499 and four 7-segment LEDs

- Main section (MAIN; lines 56–69)
- External interrupt service routine (EXT0ISR; 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-51'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 10–6) passes control to the label MAIN. All the initialize instructions are contained in lines 56–68. The MAIN section terminates by jumping back to the monitor program.

10.5 LOUDSPEAKER INTERFACE

Figure 10–7 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 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.

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 10–8 (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 $440 \times 2^{12/12} = 880$ Hz. This is the last note in our musical scale. (See Figure 10–8, line 21.) With reference to the bottom note (or "root") in any major scale, the scale in steps is 2, 4, 5, 7, 9, 11, and 12. For example, the note "E" in Figure 10–8 (line 18) is seven steps above the root; thus its frequency is $440 \times 2^{7/12} = 659.26$ 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 timeout (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 10–8, line 14.)

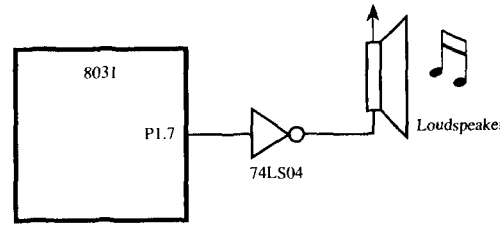
```
LOC                    LINE   SOURCE
                          1   $DEBUG
                          2   $NOPAGING
                          3   $NOSYMBOLS
                          4   ;FILE: MC14499.SRC
                          5   ;*****************************************************
                          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 10–6
Software for MC14499 interface

```
8025 757232          65            MOV     ICOUNT,#50    ;initialize int. counter
8028 D2AF            66            SETB    EA            ;enable interrupts
802A D2A8            67            SETB    EX0           ;enable External 0 int.
802C D288            68            SETB    IT0           ;negative-edge triggered
802E 0200BC          69            LJMP    MON51         ;return to MON51
                     70
                     71    ;*****************************************************
                     72    ; EXTERNAL 0 INTERRUPT SERVICE ROUTINE                *
                     73    ;*****************************************************
8031 D57205          74    EX0ISR: DJNZ    ICOUNT,EXIT   ;on 50th interrupt,
8034 757232          75            MOV     ICOUNT,#50    ; reset counter and
8037 113A            76            ACALL   UPDATE        ; refresh LED display
8039 32              77    EXIT:   RETI
                     78
                     79    ;*****************************************************
                     80    ; UDATE 4-DIGIT LED DISPLAY (EXECUTION TIME = 84 us)  *
                     81    ;                                                     *
                     82    ; ENTER:        Four BCD digits in internal memory    *
                     83    ;               locations DIGITS and DIGITS+1 (MSD in *
                     84    ;               high nibble of DIGITS)                *
                     85    ; EXIT:         MC14499 display updated               *
                     86    ; USES:         P1.5, P1.6, P1.7                      *
                     87    ;               All memory locations and regs intact  *
                     88    ;*****************************************************
803A C0E0            89    UPDATE: PUSH    ACC           ;save Accumulator on stack
803C C295            90            CLR     ENABLE        ;prepare MC14499
803E E570            91            MOV     A,DIGITS      ;get first two digits
8040 114B            92            ACALL   OUT8          ;send two digits
8042 E571            93            MOV     A,DIGITS + 1  ;get second byte
8044 114B            94            ACALL   OUT8          ;send last two digits
8046 D295            95            SETB    ENABLE        ;disable MC14499
8048 D0E0            96            POP     ACC           ;restore ACC from stack
804A 22              97            RET
                     98
                     99    ;*****************************************************
                    100    ; SEND 8 BITS IN ACCUMULATOR TO MC14499 (MSB FIRST)   *
                    101    ;*****************************************************
                    102            USING   0             ;assume reg. bank 0 enabled
804B C007           103    OUT8:   PUSH    AR7           ;save R7 on stack
804D 7F08           104            MOV     R7,#8         ;use R7 as bit counter
804F 33             105    AGAIN:  RLC     A             ;put bit in C flag
8050 9297           106            MOV     DIN,C         ;send it to MC14499
8052 C296           107            CLR     CLOCK         ;3 us low pulse on clock line
8054 00             108            NOP                   ;NOPs needed to stretch pulse
8055 00             109            NOP                   ; (minimum pulse width is
8056 D296           110            SETB    CLOCK         ; is 2 us)
8058 DFF5           111            DJNZ    R7,AGAIN      ;repeat until all 8 bits sent
805A D007           112            POP     AR7           ;restore R7 from stack
805C 22             113            RET
                    114
                    115    ;*****************************************************
                    116    ; UNUSED INTERRUPTS (ERROR; RETURN TO MONITOR PROGRAM)*
                    117    ;*****************************************************
                    118    T0ISR:
                    119    EX1ISR:
                    120    T1ISR:
                    121    SPISR:
805D C2AF           122    T2ISR:  CLR     EA            ;shut off interrupts &
805F 0200BC         123            LJMP    MON51         ; return to MON51
                    124            END
```

FIGURE 10–6
continued

FIGURE 10–7
Interface to a loudspeaker



The software in Figure 10–8 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 10–8 for further details.

10.6 NON-VOLATILE 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 digit in BCD notation).

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 $\overline{\text{STORE}}$ and $\overline{\text{RECALL}}$ 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 10–9.

In this interface example, the $\overline{\text{STORE}}$ and $\overline{\text{RECALL}}$ lines are not used. The various modes of operation are entered by sending the X2444 serial instructions through 8051 port pins.

[1]XICOR, Inc., 851 Buckeye Court, Milpitas, CA 95035

```
                        1     $debug
                        2     $nopaging
                        3     $nosymbols
                        4     ;FILE: SCALE.SRC
                        5     ;*****************************************************
                        6     ;              LOUDSPEAKER INTERFACE EXAMPLE          *
                        7     ;                                                     *
                        8     ; This program plays an A major musical scale using   *
                        9     ; a loudspeaker driven by a inverter through P1.7     *
                       10     ;*****************************************************
                       11     ;                                                     *
                       12     ;  Note  Frequency (Hz)  Period (us)  Period/2 (us)   *
                       13     ;  ----  --------------  -----------  -------------   *
                       14     ;   A        440.00         2273          1136       *
                       15     ;   B        493.88         2025          1012       *
                       16     ;   C#       554.37         1804           902       *
                       17     ;   D        587.33         1703           851       *
                       18     ;   E        659.26         1517           758       *
                       19     ;   F#       739.99         1351           676       *
                       20     ;   G#       830.61         1204           602       *
                       21     ;   A'       880.00         1136           568       *
                       22     ;*****************************************************
00BC                   23     MONITOR  CODE  00BCH       ;MON51 (V12) entry point
3CB0                   24     COUNT    EQU   -50000      ;0.05 seconds per timeout
0005                   25     REPEAT   EQU   5           ;5 x 0.05 = 0.25 seconds/note
                       26
                       27     ;*****************************************************
                       28     ; Note: X3 not installed on SBC-51, therefore         *
                       29     ; interrupts directed to the following jump table     *
                       30     ; beginning at 8000H                                  *
                       31     ;*****************************************************
8000                   32             ORG   8000H       ;RAM entry points for...
8000 028015            33             LJMP  MAIN        ; main program
8003 02806B            34             LJMP  EXT0ISR     ; External 0 interrupt
8006 028025            35             LJMP  T0ISR       ; Timer 0 interrupt
8009 02806B            36             LJMP  EXT1ISR     ; External 1 interrupt
800C 02803A            37             LJMP  T1ISR       ; Timer 1 interrupt
800F 02806B            38             LJMP  SPISR       ; Serial Port interrupt
8012 02806B            39             LJMP  T2ISR       ; Timer 2 interrupt
                       40
                       41     ;*****************************************************
                       42     ; MAIN PROGRAM BEGINS                                 *
                       43     ;*****************************************************
8015 758911            44     MAIN:    MOV   TMOD,#11H   ;both timers 16-bit mode
8018 7F00              45              MOV   R7,#0       ;use R7 as note counter
801A 7E05              46              MOV   R6,#REPEAT  ;use R6 as timeout counter
801C 75A88A            47              MOV   IE,#8AH     ;Timer 0 & 1 interrupts on
801F D28F              48              SETB  TF1         ;force Timer 1 interrupt
8021 D28D              49              SETB  TF0         ;force Timer 0 interrupt
8023 80FE              50              SJMP  $           ;ZzZzZzZz time for a nap
                       51
                       52     ;*****************************************************
                       53     ; TIMER 0 INTERRUPT SERVICE ROUTINE (EVERY 0.05 SEC.) *
                       54     ;*****************************************************
8025 C28C              55     T0ISR:   CLR   TR0                ;stop timer
8027 758C3C            56              MOV   TH0,#HIGH (COUNT)  ;reload
802A 758AB0            57              MOV   TL0,#LOW  (COUNT)
802D DE08              58              DJNZ  R6,EXIT            ;if not 5th int, exit
802F 7E05              59              MOV   R6,#REPEAT         ;if 5th, reset
8031 0F                60              INC   R7                 ;increment note
8032 BF0C02            61              CJNE  R7,#LENGTH,EXIT    ;beyond last note?
8035 7F00              62              MOV   R7,#0              ;yes: reset, A=440 Hz
8037 D28C              63     EXIT:    SETB  TR0                ;no:  start timer, go
8039 32                64              RETI                     ;     back to ZzZzZzZ
```

FIGURE 10–8
Software for loudspeaker interface

```
                        65
                        66      ;*******************************************************
                        67      ; TIMER 1 INTERRUPT SERVICE ROUTINE (PITCH OF NOTES)  *
                        68      ;                                                     *
                        69      ; Note: The output frequencies are slightly off due   *
                        70      ; to the length of this ISR.  Timer reload values    *
                        71      ; need adjusting.                                     *
                        72      ;*******************************************************
803A B297               73      T1ISR:   CPL   P1.7      ;music maestro!
803C C28E               74               CLR   TR1       ;stop timer
803E EF                 75               MOV   A,R7      ;get note counter
803F 23                 76               RL    A         ;multiply (2 bytes/note)
8040 128050             77               CALL  GETBYTE   ;get high-byte of count
8043 F58D               78               MOV   TH1,A     ;put in timer high register
8045 EF                 79               MOV   A,R7      ;get note counter again
8046 23                 80               RL    A         ;align on word boundary
8047 04                 81               INC   A         ;past high-byte (whew!)
8048 128050             82               CALL  GETBYTE   ;get low-byte of count
804B F58B               83               MOV   TL1,A     ;put in timer low register
804D D28E               84               SETB  TR1       ;start timer
804F 32                 85               RETI            ;time for a rest
                        86
                        87      ;*******************************************************
                        88      ; GET A BYTE FROM LOOK-UP OF NOTES IN A MAJOR SCALE   *
                        89      ;*******************************************************
8050 04                 90      GETBYTE: INC   A         ;table look-up subroutine
8051 83                 91               MOVC  A,@A+PC
8052 22                 92               RET
8053 FB90               93      TABLE:   DW    -1136     ;A
8055 FB90               94               DW    -1136     ;A (play again; half note)
8057 FC0C               95               DW    -1012     ;B (quarter note, etc.)
8059 FC7A               96               DW    -902      ;C# - major third
805B FCAD               97               DW    -851      ;D
805D FD0A               98               DW    -758      ;E  - perfect fifth
805F FD5C               99               DW    -676      ;F#
8061 FDA6              100               DW    -602      ;G#
8063 FDC8              101               DW    -568      ;A'
8065 FDC8              102               DW    -568      ;A' (play 4 times; whole note)
8067 FDC8              103               DW    -568
8069 FDC8              104               DW    -568
  000C                 105      LENGTH   EQU   ($ - TABLE) / 2 ;LENGTH = # of notes
                       106
                       107      ;*******************************************************
                       108      ; UNUSED INTERRUPTS - BACK TO MONITOR PROGRAM (ERROR) *
                       109      ;*******************************************************
                       110      EXT0ISR:
                       111      EXT1ISR:
                       112      SPISR:
806B C2AF              113      T2ISR:   CLR    EA       ;shut off interrupts and
806D 0200BC            114               LJMP   MONITOR ; return to MON51
                       115               END
```

FIGURE 10–8
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

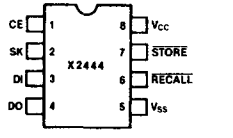
The Xicor X2444 is a serial 256 bit NOVRAM featuring a static RAM configured 16 x 16, overlaid bit for bit with a nonvolatile E^2PROM array. The X2444 is fabricated with the same reliable N-channel floating gate MOS technology used in all Xicor 5V nonvolatile memories.

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^2PROM) is completed in 10 ms or less and a recall operation (E^2PROM 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^2PROM 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

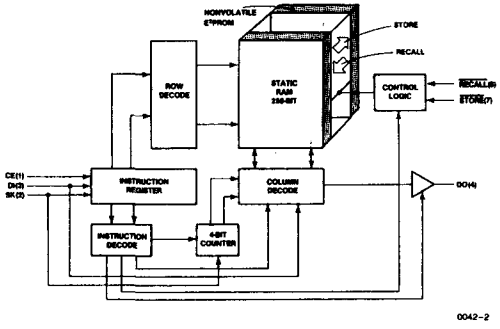


0042-1

PIN NAMES

CE	Chip Enable
SK	Serial Clock
DI	Serial Data In
DO	Serial Data Out
$\overline{RECALL}$	Recall
$\overline{STORE}$	Store
V_{CC}	+5V
V_{SS}	Ground

FUNCTIONAL DIAGRAM



0042-2

May 1987

FIGURE 10–9
Cover page for the X2444 non-volatile RAM data sheet

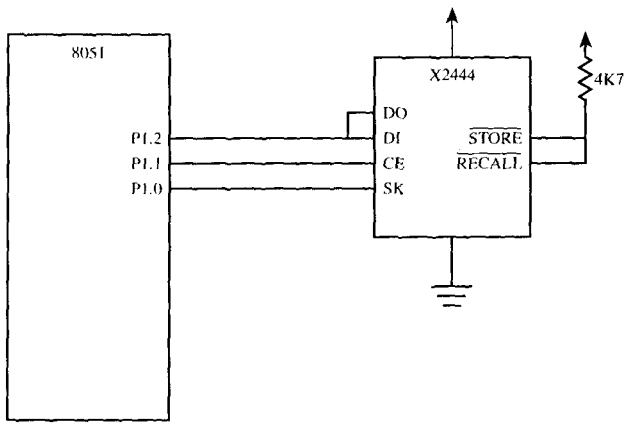


FIGURE 10–10
Interface to X2444 nonvolatile RAM

The interface to the 8051 is shown in Figure 10–10. Only 3 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 opcode 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

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 whenever 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 re-

```
                          1     $DEBUG -
                          2     $NOPAGING
                          3     $NOSYMBOLS
                          4     ;FILE: NVRAM.SRC
                          5     ;*****************************************************
                          6     ;                X2444 INTERFACE EXAMPLE            *
                          7     ;                                                   *
                          8     ; Two subroutines are shown below that SAVE or      *
                          9     ; RECOVER data between a X2444 non-volatile RAM and *
                         10     ; 32 bytes of the 8051's internal RAM.              *
                         11     ;*****************************************************
   0085                  12     RECALL  EQU     85H     ;X2444 recall instruction
   0084                  13     WRITE   EQU     84H     ;X22444 write enable instruction
   0081                  14     STORE   EQU     81H     ;X2444 store instruction
   0082                  15     SLEEP   EQU     82H     ;X2444 sleep istruction
   0083                  16     W_DATA  EQU     83H     ;X2444 write data instruction
   0087                  17     R_DATA  EQU     87H     ;X2444 read data instruction
   00BC                  18     MON51   EQU     00BCH   ;MON51 entry point (V12)
   0020                  19     LENGTH  EQU     32      ;32 bytes saved/restored
   0092                  20     DIN     BIT     P1.2    ;X2444 interface lines
   0091                  21     ENABLE  BIT     P1.1
   0090                  22     CLOCK   BIT     P1.0
                         23
----                     24             DSEG    AT 60H
0060                     25     NVRAM:  DS      LENGTH    ;60H-7FH saved/recovered
                         26
----                     27             CSEG    AT 8000H
8000 110A                28     WX2444: ACALL   SAVE      ;8000H entry point for write
8002 0200BC              29             LJMP    MON51
8005 1149                30     RX2444: ACALL   RECOVER   ;8005H entry point for read
8007 0200BC              31             LJMP    MON51
                         32
                         33     ;*****************************************************
                         34     ; SAVE 8031 RAM LOCATIONS 60H-7FH IN X2444 NVRAM    *
                         35     ;*****************************************************
800A 7860                36     SAVE:   MOV     R0,#NVRAM       ;R0 -> locations to save
800C C291                37             CLR     ENABLE          ;disable X2444
800E 7485                38             MOV     A,#RECALL       ;recall instruction
8010 D291                39             SETB    ENABLE
8012 1184                40             ACALL   W_BYTE
8014 C291                41             CLR     ENABLE
8016 7484                42             MOV     A,#WRITE        ;write enable prepares
8018 D291                43             SETB    ENABLE          ; X2444 to be written to
801A 1184                44             ACALL   W_BYTE
801C C291                45             CLR     ENABLE
801E 7F00                46             MOV     R7,#0           ;R7 = X2444 address
8020 EF                  47     AGAIN:  MOV     A,R7            ;put address in ACC
8021 23                  48             RL      A               ;put in bits 3,4,5,6
8022 23                  49             RL      A
8023 23                  50             RL      A
8024 4483                51             ORL     A,#W_DATA       ;build write instruction
8026 D291                52             SETB    ENABLE
8028 1184                53             ACALL   W_BYTE
802A 7D02                54             MOV     R5,#2
802C E6                  55     LOOP:   MOV     A,@R0           ;get 8051 data
802D 08                  56             INC     R0              ;point to next byte
802E 1184                57             ACALL   W_BYTE          ;sent byte to X2444
8030 DDFA                58             DJNZ    R5,LOOP         ;repeat (send 2nd byte)
8032 C291                59             CLR     ENABLE
8034 0F                  60             INC     R7              ;increment X2444 address
8035 BF10E8              61             CJNE    R7,#16,AGAIN    ;if not finished, again
8038 7481                62             MOV     A,#STORE        ;if finished, copy to EEPROM
803A D291                63             SETB    ENABLE
803C 1184                64             ACALL   W_BYTE
```

FIGURE 10–11
Software for X2444 interface

```
803E C291          65            CLR     ENABLE
8040 7482          66            MOV     A,#SLEEP        ;put X2444 to sleep
8042 D291          67            SETB    ENABLE
8044 1184          68            ACALL   W_BYTE
8046 C291          69            CLR     ENABLE
8048 22            70            RET                     ;DONE!
                   71
                   72    ;*******************************************************
                   73    ; RECOVER 8051 RAM LOCATIONS 60H-7FH FROM X2444 NVRAM *
                   74    ;*******************************************************
8049 7860          75    RECOVER: MOV    R0,#NVRAM
804B C291          76            CLR     ENABLE
804D 7485          77            MOV     A,#RECALL       ;recall instruction
804F D291          78            SETB    ENABLE
8051 1184          79            ACALL   W_BYTE
8053 C291          80            CLR     ENABLE
8055 7F00          81            MOV     R7,#0           ;R7 = X2444 address
8057 EF            82    AGAIN2: MOV     A,R7            ;put address in ACC
8058 23            83            RL      A               ;build read instruction
8059 23            84            RL      A
805A 23            85            RL      A
805B 4487          86            ORL     A,#R_DATA
805D D291          87            SETB    ENABLE
805F 1184          88            ACALL   W_BYTE          ;send read instruction
8061 7D02          89            MOV     R5,#2           ; (+ address)
8063 1178          90    LOOP2:  ACALL   R_BYTE          ;read byte of data
8065 F6            91            MOV     @R0,A           ;put in 8051 RAM
8066 08            92            INC     R0              ;point to next location
8067 DDFA          93            DJNZ    R5,LOOP2
8069 C291          94            CLR     ENABLE
806B 0F            95            INC     R7              ;increment X2444 address
806C BF10E8        96            CJNE    R7,#16,AGAIN2   ;repeat until last
806F 7482          97            MOV     A,#SLEEP        ;put X2444 to sleep
8071 D291          98            SETB    ENABLE
8073 1184          99            ACALL   W_BYTE
8075 C291         100            CLR     ENABLE
8077 22           101            RET                     ;DONE!
                  102
                  103    ;*******************************************************
                  104    ; READ A BYTE OF DATA FROM X2444                      *
                  105    ;*******************************************************
8078 7E08         106    R_BYTE: MOV     R6,#8       ;use R6 as bit counter
807A A292         107    AGAIN3: MOV     C,DIN       ;put X2444 data bit in C
807C 33           108            RLC     A           ;build byte in Accumulator
807D D290         109            SETB    CLOCK       ;toggle clock line (1 us)
807F C290         110            CLR     CLOCK
8081 DEF7         111            DJNZ    R6,AGAIN3   ;if not last bit, do again
8083 22           112            RET
                  113
                  114    ;*******************************************************
                  115    ; WRITE A BYTE OF DATA TO X2444                       *
                  116    ;*******************************************************
8084 7E08         117    W_BYTE: MOV     R6,#8       ;use R6 as bit counter
8086 33           118    AGAIN4: RLC     A           ;put bit to write in C
8087 9292         119            MOV     DIN,C       ;put in X2444 DATA IN line
8089 D290         120            SETB    CLOCK       ;clock bit into X2444
808B C290         121            CLR     CLOCK
808D DEF7         122            DJNZ    R6,AGAIN4   ;if not last bit, do again
808F 22           123            RET
                  124            END
```

FIGURE 10–11
continued

set. 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 10–11.

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 RAM.
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 10–12 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.

10.7 INPUT/OUTPUT EXPANSION

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 10–13.) The additional inputs are sampled 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 10–13), so the next read to DATA IN reads the next bit, and so on.

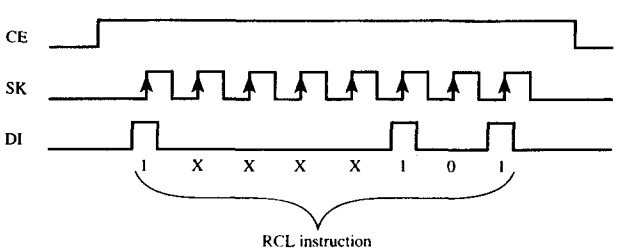


FIGURE 10–12
Timing for the X2444 recall instruction

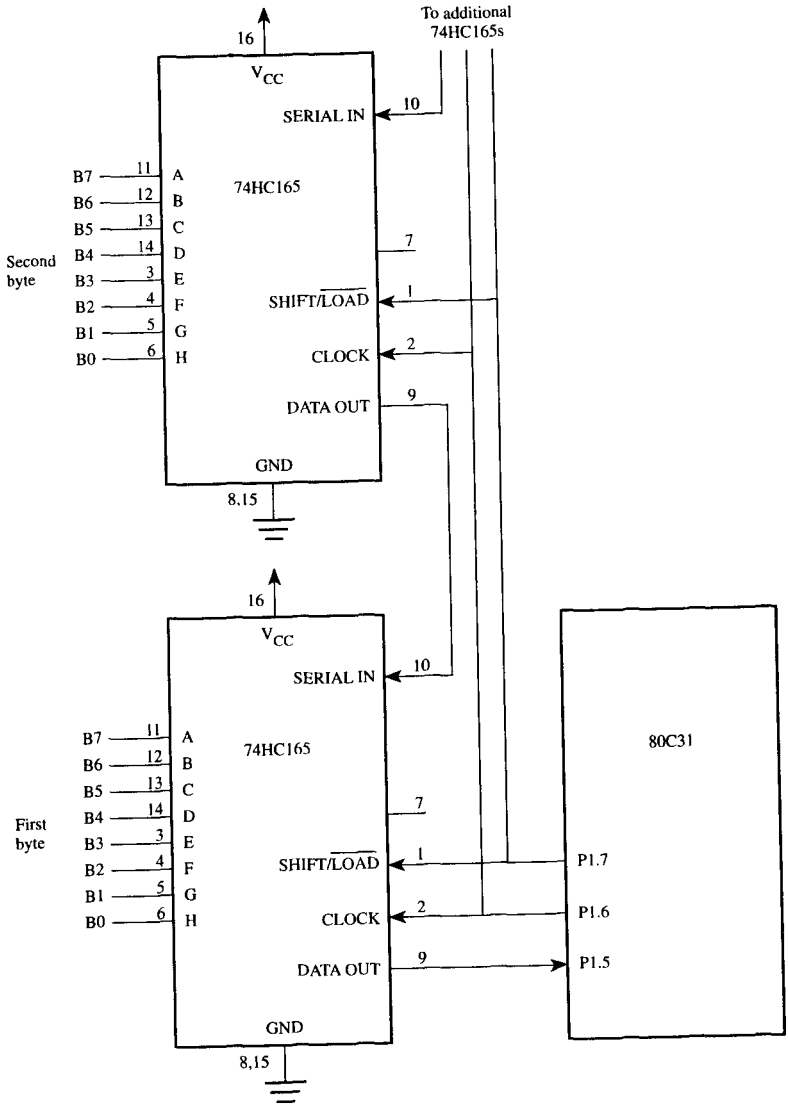


FIGURE 10–13
Interface to two 74HC165s

Design Objective

Write a subroutine that copies the state of the 16 inputs in Figure 10–13 to 8051 internal RAM locations 25H and 26H.

The software to accomplish this is shown in Figure 10–14. 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 is minimal; 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.

10.8 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 10–15). 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 = FFH).

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.

Design Objective

Write a program to generate a sine wave using the DAC interface in Figure 10–15. 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.

```
                        1     $DEBUG
                        2     $NOSYMBOLS
                        3     $NOPAGING
                        4     ;FILE: HC165.SRC
                        5     ;*****************************************************
                        6     ;                74HC165 INTERFACE EXAMPLE          *
                        7     ;                                                   *
                        8     ; The subroutine GET_BYTES below reads multiple (in *
                        9     ; this case two) 74HC165 parallel-in serial-out shift *
                       10     ; registers attached to P1.7 (SHIFT/LOAD), P1.6     *
                       11     ; (CLOCK), and P1.5 (DATA OUT).  The bytes read are *
                       12     ; placed in bit-addressable locations starting at the *
                       13     ; byte address BUFFER.                              *
                       14     ;*****************************************************
  000D                 15     CR       EQU     0DH
  0002                 16     COUNT    EQU     2       ;number of 74HC165s
  0097                 17     SHIFT    BIT     P1.7    ;74HC165 SHIFT/LOAD input
                       18                              ; 1 = shift, 0 = load
  0096                 19     CLOCK    BIT     P1.6    ;74HC165 CLOCK input
  0095                 20     DOUT     BIT     P1.5    ;74HC165 DATA OUT output
  0282                 21     OUTSTR   CODE    0282H   ;subroutines in MON51(V12)
  028D                 22     OUT2HEX  CODE    028DH   ;output byte as two hex char.
  01DE                 23     OUTCHR   CODE    01DEH
                       24
8000                   25              ORG     8000H   ;begin code segment at 8000H
8000 D296              26              SETB    CLOCK   ;set interface lines initially in
8002 D297              27              SETB    SHIFT   ; case not already
8004 D295              28              SETB    DOUT    ;DOUT must be set (input)
                       29
                       30     ;*****************************************************
                       31     ; MAIN LOOP (KEPT SMALL FOR THIS EXAMPLE)           *
                       32     ;*****************************************************
8006 128029            33              CALL    SEND_HELLO_MESSAGE  ;banner message
8009 128011            34     REPEAT:  CALL    GET_BYTES           ;read 74HC165s
800C 128050            35              CALL    DISPLAY_RESULTS     ;show results
800F 80F8              36              JMP     REPEAT              ;loop
                       37
                       38     ;*****************************************************
                       39     ; GET BYTES FROM 74HC165s & PLACE IN INTERNAL RAM    *
                       40     ;                                                   *
                       41     ; Execution time = 112 microseconds (@ 12 MHz).     *
                       42     ; Execution time for N 74HC165s = 6 + (N x 53) us   *
                       43     ;*****************************************************
                       44     GET_BYTES:
8011 7E02              45              MOV     R6,#COUNT   ;use R6 as byte counter
8013 7825              46              MOV     R0,#BUFFER  ;use R0 as pointer to buffer
8015 C297              47              CLR     SHIFT       ;load into 74HC165s by
8017 D297              48              SETB    SHIFT       ; pulsing SHIFT/LOAD low
8019 7F08              49     AGAIN:   MOV     R7,#8       ;use R7 as bit counter
801B A295              50     LOOP:    MOV     C,DOUT      ;get a bit (put it in C)
801D 13                51              RRC     A           ;put in ACC.0 (LSB 1st)
801E C296              52              CLR     CLOCK       ;pulse CLOCK line (shifts
8020 D296              53              SETB    CLOCK       ; bits toward DATA OUT)
8022 DFF7              54              DJNZ    R7,LOOP     ;if not 8th shift, repeat
8024 F6                55              MOV     @R0,A       ;if 8th shift, put in buf.
8025 08                56              INC     R0          ;increment pointer to buf.
8026 DEF1              57              DJNZ    R6,AGAIN    ;get two bytes
8028 22                58              RET
                       59
                       60     ;*****************************************************
                       61     ; SEND HELLO MESSAGE TO CONSOLE (DEBUGGING AID)     *
                       62     ;*****************************************************
                       63     SEND_HELLO_MESSAGE:
8029 908030            64              MOV     DPTR,#BANNER    ;point to hello message
```

FIGURE 10–14
Software for 74HC165 interface

```
802C 120282          65              CALL    OUTSTR          ;send it to console
802F 22              66              RET
8030 2A2A2A20        67      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
                     68
                     69      ;*****************************************************
                     70      ; DISPLAY RESULTS ON CONSOLE (DEBUGGING AID)         *
                     71      ;*****************************************************
                     72      DISPLAY_RESULTS:                ;display bytes
8050 7825            73              MOV     R0,#BUFFER      ;R0 points to bytes
8052 7E02            74              MOV     R6,#COUNT       ;R6 is # of bytes read
8054 E6              75      LOOP2:  MOV     A,@R0           ;get byte
8055 08              76              INC     R0              ;increment pointer
8056 12028D          77              CALL    OUT2HEX         ;output as 2 hex char.
8059 7420            78              MOV     A,#' '          ;separate bytes
805B 1201DE          79              CALL    OUTCHR
805E DEF4            80              DJNZ    R6,LOOP2        ;repeat for each byte
8060 740D            81              MOV     A,#CR           ;begin a new line
8062 1201DE          82              CALL    OUTCHR          ;send CR (LF too!)
8065 22              83              RET
                     84
                     85      ;*****************************************************
                     86      ; CREATE BUFFER IN BIT-ADDRESSABLE INTERNAL RAM       *
                     87      ;*****************************************************
----                 88              DSEG    AT 25H  ;on-chip data segment in
0025                 89      BUFFER: DS      COUNT   ; in bit-addressable space
                     90              END
```

FIGURE 10–14
continued

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 0, 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 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 10–16 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 10–17. The main loop (lines 36–40) does three things: initialize timer 0 to interrupt every 100 μs, turn on interrupts, and sit 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 look-up table using the DPTR and then written to port 1. A constant called STEP is used as the increment through the table. STEP is

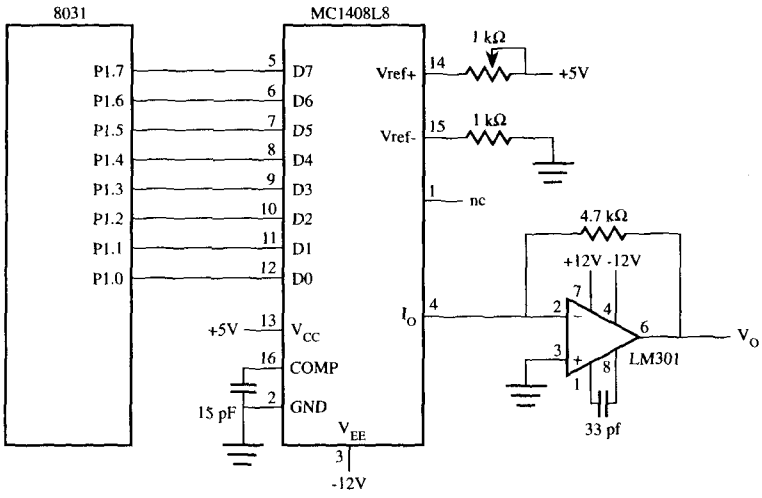


FIGURE 10–15

```
/*****************************************************/
/* 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 10–16

```
                        1     $debug
                        2     $nopaging
                        3     $nosymbols
                        4     ;FILE: DAC.SRC
                        5     ;*****************************************************
                        6     ;                MC1408L8 INTERFACE EXAMPLE         *
                        7     ;                                                   *
                        8     ; This program generates a sine wave using a sine   *
                        9     ; wave look-up table and an interface to a MC1408L8 *
                       10     ; 8-bit digital-to-analog converter.  The program is *
                       11     ; interrupt-driven.                                 *
                       12     ;                                                   *
                       13     ; Data are read from a 1024-entry sine wave table and *
                       14     ; sent to the DAC every 100 us.  Each value sent is *
                       15     ; STEP locations past the previous value sent (with *
                       16     ; wrap around once the end is reached).  The period *
                       17     ; of the sine wave is 100 x (1024 / STEP) us.  For  *
                       18     ; example, if STEP is 20H, the sine wave has a period *
                       19     ; of 100 x (1024 / 32) = 3.2 ms and a frequency of  *
                       20     ; 313 Hz.                                           *
                       21     ;                                                   *
                       22     ; Note:  Initialize STEP in internal location 50H   *
                       23     ; before running the program.                       *
                       24     ;*****************************************************
  00BC                 25     MONITOR  CODE     00BCH       ;MON51 entry (V12)
  0050                 26     STEP     DATA     50H         ;put STEP in internal RAM
                       27
8000                   28               ORG      8000H       ;start at 8000H
8000 028015            29               LJMP     MAIN        ;initialize timer
8003 028037            30               LJMP     EXT0ISR     ;unsused
8006 028022            31               LJMP     T0ISR       ;every 100 us, update DAC
8009 028037            32               LJMP     EXT1ISR     ;unused
800C 028037            33               LJMP     T1ISR       ;unused
800F 028037            34               LJMP     SPISR       ;unused
8012 028037            35               LJMP     T2ISR       ;unused
8015 758902            36     MAIN:     MOV      TMOD,#02H   ;8-bit, auto reload
8018 758C9C            37               MOV      TH0,#-100   ;100 us delay
801B D28C              38               SETB     TR0         ;start timer
801D 75A882            39               MOV      IE,#82H     ;enable timer 0 interrupts
8020 80FE              40               SJMP     $           ;main loop does nothing!
8022 E550              41     T0ISR:    MOV      A,STEP      ;add STEP to DPTR
8024 2582              42               ADD      A,DPL
8026 F582              43               MOV      DPL,A
8028 5002              44               JNC      SKIP
802A 0583              45               INC      DPH
802C 538303            46     SKIP:     ANL      DPH,#03H    ;wrap around, if necessary
802F 438384            47               ORL      DPH,#HIGH(TABLE)
8032 E4                48               CLR      A
8033 93                49               MOVC     A,@A+DPTR   ;get entry
8034 F590              50               MOV      P1,A        ;send it
8036 32                51               RETI
                       52
                       53     EXT0ISR:                      ;unused interrupts
                       54     EXT1ISR:
                       55     T1ISR:
                       56     T2ISR:
8037 C2AF              57     SPISR:    CLR      EA          ;turn off interrupts and
8039 0200BC            58               LJMP     MONITOR     ; return to MON51
                       59
                       60     ;*****************************************************
                       61     ; The following is a sine wave look-up table.  The  *
                       62     ; table contains 1024 entries and is ORGed to begin *
                       63     ; at 8400H to allow easy wrap-around of the DPTR    *
                       64     ; once the end of the table is reached.  The entries *
```

FIGURE 10–17

```
                        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 10–17
continued

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 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.

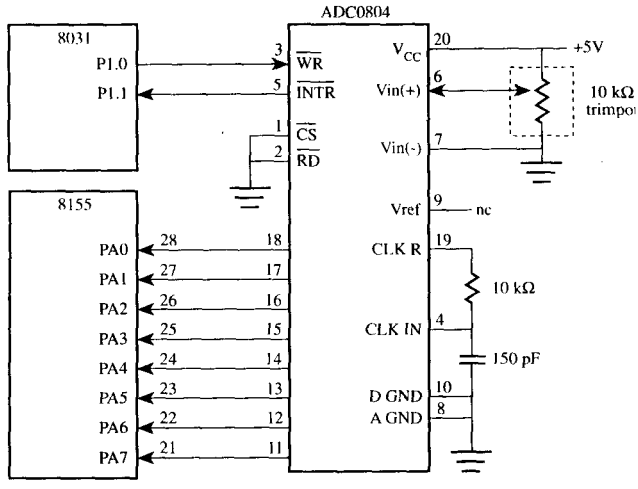
10.9 ANALOG INPUT

Our final design example is an analog input channel. The circuit in Figure 10–18 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 ($\overline{WR}$) and an interrupt output ($\overline{INTR}$). A conversion is started by pulsing $\overline{WR}$ low. When the conversion is complete (100 μs later), the ADC0804 asserts $\overline{INTR}$, making it low. $\overline{INTR}$ is de-asserted (high) on the next 1-to-0 transition of $\overline{WR}$, that initiates the next conversion. $\overline{INTR}$ and $\overline{WR}$ 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 10–18



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 10–19 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 $\overline{WR}$ input. Then, the program sits in a loop waiting for the ADC0804 to finish the conversion and assert $\overline{INTR}$ 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 FFH as the trimpot is adjusted.

The program in Figure 10–19 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.

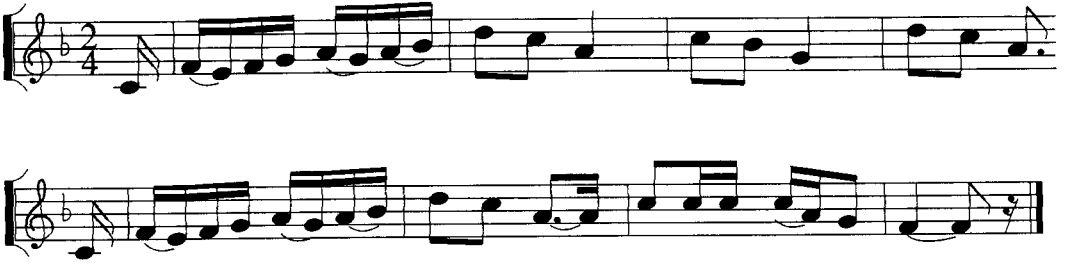


FIGURE 10–20
Musical melody for problem 1

7. In Figure 10–17, 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.