The 8051 Microcontroller
Based
Embedded Systems

ABOUT THE AUTHOR

Manish K Patel completed his BE in Electronics and Communication Engineering in 2002 from G H Patel College of Engineering and Technology, Vallabh Vidyanagar, and ME in Electronics and Communication Systems in 2009 from Dharmsinh Desai University, Nadiad. At present, he is working as Assistant Professor in the Department of Electronics and Communication, Faculty of Technology, Dharmsinh Desai University, Nadiad, Gujarat. In his 12 years of teaching experience, he has carried out and guided many projects based on microcontrollers. His area of interest is design and implementation of distributed embedded systems.

The 8051 Microcontroller
Based
Embedded Systems

Manish K Patel

Dharmsinh Desai University, Nadiad
Gujarat

McGraw Hill Education (India) Private Limited
NEW DELHI

McGraw Hill Education Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
P-24, Green Park Extension, New Delhi 110016

The 8051 Microcontroller based Embedded Systems

ISBN (13) : 978-93-329-0125-4
ISBN (10) : 93-329-0125-2

Managing Director: *Kaushik Bellani*

Head—Higher Education (Publishing and Marketing): *Vibha Mahajan*

Senior Publishing Manager (SEM & Tech. Ed.): *Shalini Jha*
Associate Sponsoring Editor: *Smruti Snigdha*
Senior Editorial Researcher: *Amiya Mahapatra*
Manager—Production Systems: *Satinder S Baveja*
Assistant Manager—Editorial Services: *Sohini Mukherjee*
Senior Manager—Production: *P L Pandita*

Assistant General Manager (Marketing)—Higher Education: *Vijay Sarathi*
Assistant Product Manager (SEM & Tech. Ed.): *Tina Jajoriya*
Senior Graphic Designer—Cover: *Meenu Raghav*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Typeset at Print-O-World, 2579, Mandir Lane, Shadipur, New Delhi 110 008, and printed at

Cover Printer :

Contents

Preface

In the past few decades, microcontrollers have changed the way we live, entering almost all aspects of our life. Their production counts are in billions per year. Due to the massive applications of microcontrollers, there is enough space for 8-bit microcontrollers for small- and medium-scale embedded systems in the coming future. Embedded software is used in almost every electronic device today. Therefore, there is a substantial need of skillful programmers and system designers. Moreover, the availability of sufficient hardware resources in recent variants of microcontrollers has led to universal use of high-level languages such as C (except very time-critical applications) in the development of embedded systems.

Target Audience

This book is specifically written for an introductory (first-level) course in the subject, and the contents are class tested to ensure that the treatment is logical and easy to understand for the fresher.

This book can be used by students as a text/reference book for either one-semester or two-semester courses at the undergraduate level, i.e. in B. Tech. Electronics & Communication, Electronics, Computer Science, Information Technology, Instrumentation & Control, Mechatronics, Electrical Engineering, etc. It can also serve as a reference book for Bachelor of Sciences or Master of Sciences in the field of Electronics, Diploma courses, technical training institutes and embedded-system designers.

The text assumes that the readers are familiar with concepts and terminology of digital systems.

Rationale behind Writing this Book

During my interactions with students over several years, undertaking the subject based on the 8051 microcontroller, I found they face several difficulties in studying the subject because of the ***lack of a standalone book*** with easy-to-understand and reader-friendly language, in-depth coverage of topics with balanced treatment of fundamental concepts and practical aspects with applications, proper organization and flow of content. Even I faced similar difficulties while teaching the subject for the first few times. Keeping in mind these issues, I was inspired to write a book that could fulfill the needs of students and could serve as a standalone reference.

The objective of the book is to introduce fundamental hardware, software and architectural aspects of microcontroller-based embedded systems in an elementary and integrated manner and to provide a strong foundation for the development of expertise in designing such systems.

About the Book

This book covers topics the author feels every embedded-system designer must know. The 8051 microcontroller is chosen as the subject as it is the most popular 8-bit microcontroller due to its low cost, easy availability of tools and support, multiple vendors and wide variety of variants, number of companies licensing the core with new peripherals for continuous improvement of their products, the large numbers of 8051 aware engineers (both hardware and software) and reusability of existing 8051 software in the public domain.

The following aspects of embedded-systems design are discussed in the book using the 8051 microcontroller as an illustration.

- **Architecture**
  - Architectural block diagram and Programming model of the 8051
  - Timing diagrams and instruction execution
- **Hardware features and resources**
  - I/O ports, Timers/Counters, Serial data transfer protocols: UART, I2C and SPI, Interrupts
  - System design and Troubleshooting, Introduction to common peripherals and features: CAN, Watchdog timers, PWM, Analog comparators, Multiple DPTRs, ISP and IAP, Comparison and Introduction of MCS 51, MCS 96, MCS 151, MCS 251, AVR ATmega and PIC 18 microcontroller families

- **Programming**
  - Instruction set, Logic/program development steps, Assembly and C language programming with powerful documentation, Debugging and testing the programs with simulation steps and snapshots, Tools for software development
- **Real-world interfacing**
  - Keyboards, Memory (data as well as code memory), EEPROM programming, Display devices: LED, LCD, Seven segment displays, ADCs(On-chip and Off-chip) and DACs (Serial as well as parallel), Real-Time Clock, Stepper motors, dc motors, LM35 temperature sensors, IR sensors, Relays and Opto-couplers, Projects

The presentation of the introduction and background of all concepts is general in nature, and detailed discussion is based on the Intel 8051 microcontrollers and their variants. This approach allows the reader to migrate easily to other microcontroller architectures.

The potential reader can easily understand the importance and need of the book by observing the organization of the topics in the table of contents. For beginners, the theoretical concepts given in the book establish a strong foundation necessary for development of microcontroller-based embedded systems. For experienced readers/professionals working on the projects, it provides detailed coverage of topics and may serve as a practical and reference guide. The book also helps teachers arrange the flow of content best suited for classroom teaching, discussions and presentations.

The book covers a wide variety of latest variants of the 8051 microcontroller that can compete with other microcontroller architectures in the market, a tutorial on the latest software-development tool—IDE (Keil µVision 4.0). It also has complete chapters on advanced serial data-transfer protocols like I2C and SPI, and timing and execution of all types of instructions with the help of data-flow diagrams, introduction to MCS 96, MCS 151, MCS 251, PIC18 and the AVR ATmega family of microcontrollers.

This book can be used to establish the strong background for taking advanced subjects like embedded system design, embedded computer architecture, RTOS and microcontroller architecture.

Chapter Organization

The major theme of the book is 'logical sequencing of the chapters and their topics with proper organization and flow of content to enhance understanding of the entire subject'. The major focus is on concise, to-the-point discussion of topics with **clarity and simplicity**.

The book is divided logically into three parts.

Part 1: Microcontroller Architecture, Programming and Development Tools (Chapters 1 to 12)

Chapter 1 covers the basics of computer systems and microcontrollers. The applications, classification and criterion for selection of microcontrollers for specific applications is discussed. The features of the 8051 family of microcontrollers and comparison with other microcontroller families are given in brief. **Chapter 2** introduces architectural block diagram and programming model of the 8051. **Chapter 3** is dedicated to the tools and program-development process. **Chapters 4 to 8** cover instruction set of the 8051, programming concepts and show how to use the instructions to develop simple programs.

Chapter 9 is devoted to programming examples for array processing, 16-bit arithmetic and code conversions. **Chapter 10** explains timing diagrams and execution of all types of instructions with the help of data-flow diagrams. **Chapter 11** presents the pin diagram of the 8051 and shows how standalone systems using the 8051 can be designed. **Chapter 12** focuses on programming in C language. It shows how the hardware features of the 8051 can be used and controlled by a high-level language.

Part 2: On-chip Peripherals (Chapter 13 to 16)

Chapter 13 discusses in detail the port structure of the 8051. It shows how the ports of 8051 can be used to interface the microcontroller with the external world. **Chapter 14** discusses the need and uses of timers. It covers adequate details of different modes of the timer operation, programming the timers as interval timers as well as event counters. It contains a wide variety of examples in assembly as well as C language related to uses of timers and counters. **Chapter 15** is devoted to serial communications. It discusses different types of communications and need of serial communications. The RS232

standard is introduced, and all modes of UART with programming examples and applications are given in detail. **Chapter 16** discusses the interrupts in detail. The applications and programming of internal and external interrupts are given with sufficient details. Advanced concepts like interrupt priorities, nested interrupts and interrupt latency are introduced.

Part 3: Real-world Interfacing (Chapters 17 to 23)

Chapter 17 covers issues related to designs of keyboards. The major focus is on programming, designing and interfacing different types of keyboards with the 8051. **Chapter 18** is dedicated to display devices like LEDs, 7-segment LEDs and LCDs. It shows methods to program and interface these devices with the 8051. **Chapter 19** focuses on data converters and sensors, i.e. analog-to-digital and digital-to-analog converters, temperature and infrared sensors. The features, programming and interfacing various on-chip and off-chip ADCs, DACs and sensors are discussed in detail. **Chapter 20** discusses the features, design, operation, programming and interfacing circuits of relays, opto-couplers, DC motors and stepper motors. **Chapter 21** is about interfacing external memory to the 8051-based systems. It introduces the types of memories, signals of memory chips and address-decoding methods. The method of on-chip EEPROM programming is also presented. The interfacing, programming and applications of Real-Time Clock chip DS12887 is given in detail. **Chapter 22** covers in detail the advanced serial data-transfer protocols: I2C and SPI. It shows, with numerous examples, how this protocols can be used in real-life applications. **Chapter 23** introduces and compares the features of different variants of the 8051 microcontrollers from various chip manufacturers. It also covers MCS 96, MCS 151, MCS251, PIC18 and AVR ATmega family of microcontrollers.

All theoretical concepts are explained with proper examples and illustrations wherever necessary. Large numbers of programming examples are given, which gives a better insight into the theoretical material and makes the book application-oriented. Multiple examples for the same concept help in clarifying any doubt regarding the concept.

The examples in the book cover adequate details of all aspects of programming and real-world interfacing like logic/program development steps, optimization with respect to execution speed and memory requirements, powerful comments to help understand, upgrade or modify programs, simulation steps, snapshots of outputs, debugging and troubleshooting techniques, and complete hardware interfacing diagrams. The book includes simple projects; each project includes the problem statement, complete schematic diagram, logic/program development steps, assembly and/or C program and suggested modifications. These projects show how the 8051 can be used to in real-life applications.

Each chapter begins with *Learning Objectives* and *Key Terms* that provide an idea about specific outcomes from the chapter. Pictorial illustrations of a majority of fundamental theoretical concepts aid in thorough understanding of the subject.

Objective-type Questions (MCQ), *Review Questions* with *Answers* and *Points to Remember* at the end of the chapters are sufficient to enforce the application of concepts understood in the chapter and will help students prepare themselves for self-test as well as examinations. *Think Boxes* in the text are given at suitable places to highlight miscellaneous concepts and to avoid confusions where students might stumble or get confused.

How to Use the Book

Once the theoretical concepts are understood after reading the topics, the programming examples can be tested in the IDE-μVision 4.0 to have better insight of the topic. Appendix B explains how to use μVision 4.0 to develop, simulate and debug 8051 programs in assembly as well as C language. To support and ease understanding, a stepwise explanation along with screenshots of μVision 4.0 IDE windows is given for sample programs.

Salient Features

- Simple and easy-to-understand language supported with self-explanatory diagrams
- Logical sequencing of topics, concise and to-the-point discussion
- Step by step approach for the software development
- Latest advancements to the field like I2C, SPI, etc., which is not present in any book
- Simulation methods and snapshots of the output for some key examples
- Programming examples in assembly and C languages; all instructions explained through use of examples
- Timing and data-flow diagrams for instruction execution

- Advanced and complex topics like interrupt handling, interrupt latency, lookup tables, timing analysis, stack operations, multiprocessor communications, 8051 enhancements and variants, internal port structure covered with clarity
- Coverage of many variants, peripheral devices, PIC and AVR microcontrollers
- Tutorial of Keil μVision4.0 Integrated development environment
- 6 projects to help students get hands-on experience and improve their designing skills
- Excellent pedagogy:
  - *Learning Objectives* and *Key Terms* at the beginning of each chapter
  - *Points to Remember* at the end of each chapter
  - Discussion Questions within the topics: 25
  - Review Questions with answers: 310
  - Exercise Questions: 410
  - Programming Examples (Assembly and C): 325
  - Objective Questions at the end of each chapter: 301
  - Think Boxes with Answers: 95
  - Illustrations (Figures and Tables): 350

Online Learning Center

The book is supplemented with separate online resources for instructors and students, accessible at *http://www.mhhe.com/patel/mbes*

Online Resources for Instructors

- Complete Solution Manual of the book
- Chapterwise PowerPoint slides
- Additional material on advanced microcontrollers

Online Resources for Students

- *Lab Manual*: A complete Lab Manual containing 14 laboratories with sample references from the book, sample programs followed by laboratory exercises to reinforce the concepts
- Chapterwise Objectives, Key Terms and Points to Remember
- *Projects given in the book*: Each project includes the problem statement, complete schematic diagram, program development, assembly and/or C programs and suggested modifications
- Question papers of different universities with solutions
- *Chapterwise interfacing diagrams*
- Complete designs (Schematic diagram and PCB layout) of the 8051 based hardware boards
- Additional question bank

Acknowledgements

I am highly indebted to Dr Nikhil Kothari, Head, Electronics and Communication Department, Faculty of Technology, who inspired me to write this book and taught me a majority of the concepts covered in the book. I would like to express my sincere thanks to Dr H M Desai, Vice Chancellor, Dharmsinh Desai University, and Prof. D G Panchal, Dean, Faculty of Technology, Dharmsinh Desai University, for providing a creative and challenging atmosphere in the university campus.

I am grateful to my colleagues and friends—Prof. V A Vohra, Prof. D K Rabari, Prof. B P Patel, Prof. R K Dana, Prof. B B Patel, Prof. S S Thavalapill, Dr V M Thumar, Prof. P D Dalal and Prof. H D Patel—for their reviews, suggestions and support during the development of the manuscript. I am thankful to my students for their valuable feedback/reviews of the manuscript and their help in preparing the material of the book. I thank Mr. Nitin Paranjape (MD, Edutech Systems, Vadodara) for providing me the details of hardware boards and tools. I express my acknowledgement to the entire team

at McGraw-Hill Education India, especially, Mr Sourabh Maheshwari and Mr Piyaray Pandita for their support and guidance.

I am grateful to Intel Corporation, NXP Semiconductors and Atmel Corporation for generously allowing me to use the information from their product datasheets. My special thanks to Keil Corporation for allowing me to use snapshots of their IDE μVision 4.0.

I am deeply indebted to my parents; wife, Dr. Devangi; daughter, Ragvi, and family for their constant motivation and understanding.

The following reviewers also deserve a special mention for sending me their feedback and suggestions.

K Venkata Reddy	*Jawaharlal Nehru Technological University (JNTU), Kakinada, Andhra Pradesh*
Padmavathi P	*Malla Reddy Institute of Technology and Sciences, Hyderabad, Andhra Pradesh*
B Bhavani	*Maturi Venkata Subba Rao (MVSR) Engineering College, Hyderabad, Andhra Pradesh*
V Seetha Lakshmi	*Sri Sakthi Institute of Engineering and Technology, Coimbatore, Tamil Nadu*
K R Anil Kumar	*NSS College of Engineering, Palakkad, Kerala*
Suresh Kumar	*Vellamal Engineering College, Surapet , Tamil Nadu*
C A Ghuge	*PES Modern College of Engineering, Pune, Maharashtra*
Rachit K Dana	*Dharmsinh Desai University, Nadiad, Gujarat*
Anil Kumar Sharma	*Abacus Institute of Engineering and Management, Hooghly, West Bengal*
Pinaki R Ghosh	*Adamas Institute of Technology, Kolkata, West Bengal*
Santanu Chattopadhyay	*Indian Institute of Technology (IIT) Kharagpur, West Bengal*
Chanchala Kumari	*National Institute of Technology (NIT) Jamshedpur, Jharkhand*
P K Mukherjee	*Indian Institute of Technology—Banaras Hindu University (IIT-BHU), Varanasi, Uttar Pradesh*
Praveen Malik	*Raj Kumar Goel Institute of Technology, Ghaziabad, Uttar Pradesh*

Feedback Request

I will be grateful to the readers if they have suggestions (feedback/criticism/comments) and can point out errors. These can be sent to *manishpatel_79@yahoo.com*

Manish K Patel

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments from you, all of which can be sent to *info.india@mheducation.com* (kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

1

Introduction to Microcontrollers

Objectives

To discuss and Introduce:

- Basic structure, organization and functions of a computer system along with common terminology of computer literature
- Features and comparison of microprocessors and microcontrollers
- Microcontroller classification with respect to core, memory and instruction set architecture
- Criterion for selection of microcontroller with respect to target product
- Applications of microcontrollers in various fields
- Features, overview and advantages of using the 8051 family microcontrollers
- Variants and enhancements of the 8051
- Concept of embedded systems and their characteristics

Key Terms

- 8051 Family
- Address Bus
- ALU
- Arithmetic/Logic Operations
- Boolean Processor
- Central Processing Unit
- CISC/RISC
- Control Bus
- Data Bus
- Embedded System
- Input/Output (I/O) Unit
- Instructions
- Memory
- Microcoded/Hardwired Design
- Microcomputer
- Microcontroller
- Microprocessor
- Peripherals
- Program/Data Memory
- Von Neumann/Harvard Architectures
- Word Length

A computer is a digital device which understands and operates on binary digits 0 and 1, known as *bits (binary digits)*. The binary digits are processed and stored as voltage levels in the circuits. The computer system needs bit patterns of 0s and 1s to perform any operation. These binary patterns are called *binary instructions* which are recognized and processed (or executed) by a computer to accomplish a task. The designer of the computer decides and implements these bit patterns based on the number and types of operations a computer is required to perform. The most common operations are storing and retrieving binary numbers, arithmetic and logical operations on binary numbers.

1.1 | COMPUTER SYSTEM

A digital computer system typically consists of four major components: the Central Processing Unit (CPU), Memory, Input/ Output (I/O) Unit and System Bus. The simplified arrangement of these components in a computer system is shown in Figure 1.1.

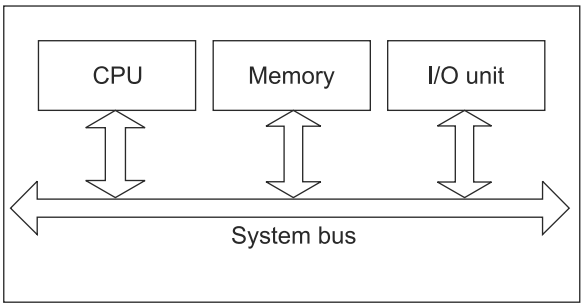


Fig. 1.1 *Basic components of a computer system*

1.1.1 Central Processing Unit

The brain of a computer is the central processing unit. It consists of a group of circuits that determine the operations that the computer can perform. The CPU controls the flow of information among the components of the computer. It also processes the data by performing digital operations on them. The block diagram of the CPU is shown in Figure 1.2.

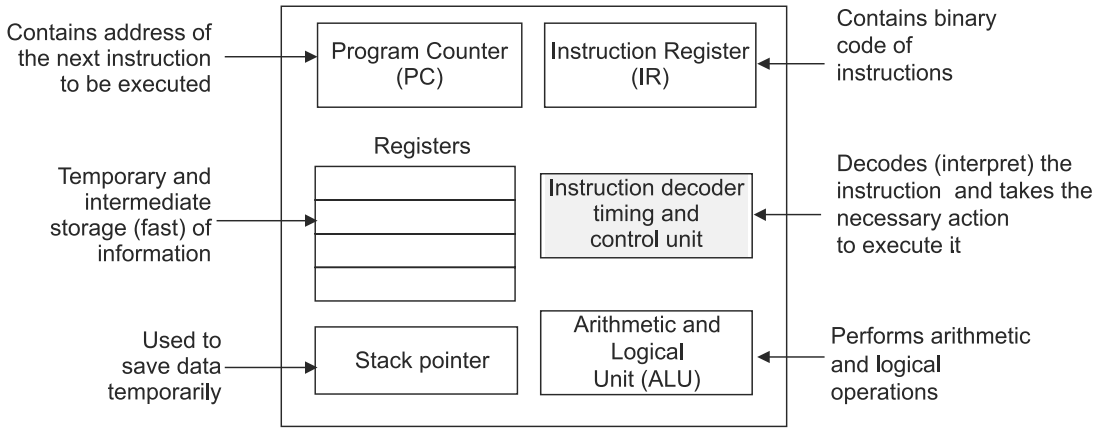


Fig. 1.2 *General block diagram of a CPU (microprocessor)*

CPU Components

All major components as shown in the block diagram of the CPU are discussed briefly in the following section.

(a) Arithmetic and Logic Unit (ALU) The heart of a CPU is the arithmetic and logic unit which is used to perform all arithmetic and logical operations. Addition, subtraction, multiplication and division are arithmetic operations while AND, OR, NOT, EXCLUSIVE-OR and shifting are logical operations.

The operation to be performed by ALU is decided by control signals generated by instruction decoder and timing unit as per the instruction being executed. Two (or more) inputs are usually given to ALU and it will generate the result of the operation. The ALU also updates the status register to indicate the nature of the result.

(b) Registers The registers are used for holding source data and results of operations temporarily. It consist of general-purpose registers and some dedicated registers to perform specific tasks. For example, accumulator register, which is required for most (arithmetic and logical) operations. Since external memory access is slower than access to these registers (because they are part of the CPU), it is preferred to use the registers when several operations must be performed on a set of data. The registers are also referred as *working registers of CPU*.

(c) Instruction Register (IR) The instruction register holds the binary (machine) code of current instruction while it is being decoded and executed*.

(d) Program Counter (PC) The program counter holds the address of the next instruction to be executed, i.e. it points to the instruction that is to be executed next. As the CPU reads the instructions from the memory, the PC is incremented automatically to point to next instruction. The reading of an instruction (or data) from memory is also referred as fetching an instruction. When the execution of current instruction is completed, the address in the PC is placed on the address bus and in response to that the memory places binary code for the next instruction on the data bus, which is then moved in to instruction register. The length of an instruction (in bytes) will be determined when it is decoded and the PC is incremented such that it will point to the next instruction.*

(e) Stack Pointer (SP) The stack is a portion of the memory used by CPU to save register contents and return addresses temporarily for subroutine and interrupt service routine calls. The data is stored in to stack using PUSH instructions and retrieved using POP instructions.

The address of the stack location which was accessed last is kept in the stack pointer.

(f) Instruction Decoder and Control Unit The instruction decoder interprets the instruction present in the instruction register and control unit determines the sequence of operations that should be performed to complete the task specified by an instruction. Control unit also provides timing to all operations. The control unit provides the inputs for the ALU (either from registers or memory), decides the operation to be performed, and makes sure that the result is written to the proper location (register or memory).

1.1.2 Memory

The memory is used to store data and binary instructions. It is normally organized as several modules (chips), where each module contains several memory locations. Each location may contain part or all of the data or instruction. CPU reads (fetches) the instructions from the memory and performs operations (indicated by instructions) on data.

A typical memory module is shown in Figure 1.3. It consists of N memory locations of equal length (usually bytes). Each memory location is assigned unique address (0 to N–1).

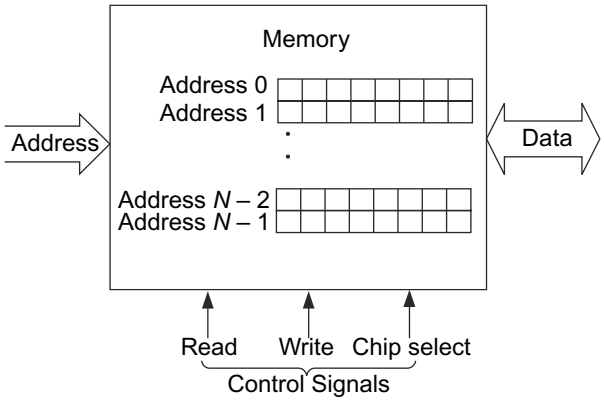


Fig. 1.3 *Generalized memory module*

All memory devices (chips) have common set of input and output signals, like address, data and control signals. Address inputs (address lines) are used to identify one memory location out of N locations. They are designated as A_0 to A_N, where N is always one less than total address lines. The number of address lines determines capacity of the memory devices (In fact, the number of memory locations inside the memory chip will decide how many address lines are required). For example, 10 address lines on a memory chip indicate that there are $2^{10} = 1024$ memory locations. The data lines transfer data to or from memory devices. The control signals are used to enable memory device and to control nature of operation like read or write.

Types of memories and their associated circuits are discussed in detail in Chapter 21.

1.1.3 I/O Unit

Input/output units are used to communicate with the external world. The input unit (devices) sends information to the computer system. Output devices send information from computer system to the external world. Keyboard, mouse, switches, A/D converters are all input devices while video screen, LED, LCD, speaker, D/A converters are all output devices.

* Nowadays, CPUs are available with pipelined architecture where contents of IR changes while execution of current instruction and PC points to next or next to next instruction or even further.

I/O Port, I/O Device and I/O Interfacing Circuits These three terms are used widely in computer hardware literature and are used with blurred distinction between them. Therefore, we need to define them clearly to avoid any confusion.

(a) I/O Port The hardware in a computer that allows information transfer between external world and computer is called I/O port.

These I/O ports are usually eight bits (or multiple of it) wide and thus referred to as a BYTE wide port, i.e. byte wide input port, byte wide output port.

(b) I/O Device (Peripherals) The device that gives information to computer is called input device. For example, keyboard, mouse, joystick, microphone, A/D converters are all input devices.

The device that receives information from computer is called output device. For example, LED, LCD, monitor, printer, speaker, D/A converter are output devices. Memory is both input as well as output device.

(c) I/O Interfacing Circuits The circuits that are used to interconnect (interface) I/O devices with a computer or I/O ports are called I/O interfacing circuits. For example, buffers, latches and voltage level converters are all interfacing circuits. Some I/O devices may also have inbuilt interfacing circuits. The voltage converters are required because different devices may require different operating voltage levels.

THINK BOX 1.1

What are the common input and output devices available in a personal computer?
Keyboard, mouse, CD reader, microphone and joystick are common input devices. LEDs (on the keyboard), monitor (display device), speaker and printer are common output devices. Floppy drive, CD-RW (reader/writer) are input as well as output devices.

1.1.4 System Bus

A group of wires known as a *bus* interconnects the three components of computer systems described above. Physically it is group of 8, 16 or more wires. The system bus provides communication path between CPU, memory and I/O. Three types of buses are required for communications between these three blocks: the address bus, the data bus and the control bus.

Address bus is a group of wires used by CPU to identify specific memory location within a memory chip (also to identify specific memory chip out of many chips present in a computer system) and to identify I/O devices as well. Each memory location or I/O device have a unique address, therefore to access them CPU places their address on to the address bus. Since addresses are always generated and placed by CPU, address bus is unidirectional. Size of the address varies across the systems, usually it is 16 bits (wires), 20 bits or 32 bits wide. The size of the address bus (address lines) determines the maximum amount of locations that can be addressed uniquely, i.e. maximum amount of memory that can be present in a system (more commonly referred as address space). If an address is N bits wide then there are 2^N different addresses (0 to 2^N–1), hence address space is of 2^N bytes*. For example, a 16-bit address bus can address 2^{16} = 65536 bytes of memory.

Data bus transfers data or instructions between CPU and memory or I/O devices. It is bidirectional because data can be transferred in both directions, i.e. from CPU to memory (or output devices) or from memory or input devices to CPU. Usually, it is 8 or 16 bits wide for low end computers and 32 or 64 bits wide for high end computers. Advantage of wider data bus is higher rate of data transfer.

Control bus is used to enable memory and I/O devices to perform read or write operations. It regulates all activities on the bus and specifies timing and direction of the data transfer. Read (RD), write (WR) and memory/I/O (M/I/O) are most common control signals.

Along with these four major components, interfacing circuits, as discussed earlier, are required to interconnect these components. Interfacing circuits coordinates signals on the bus which are generated by various components of the system. Memory interfacing circuit may typically contain a logic circuit to decode the address of a memory location. Buffers and latches are most common devices used between system bus and memory or I/O devices. A detailed block diagram of computer including different buses and interfacing circuits is shown in Figure 1.4.

* A memory location usually contains 8 bits—a byte.

1.1.5 How does the CPU Read Data from the Memory Chip?

To read instruction or data from memory, the CPU places address of the desired memory location on the address bus. Next, it sends the control signal to enable the memory chip. Finally, memory chip places contents of addressed memory location on to data bus, and CPU reads content of the data bus by generating read control signal. For operations like memory write, I/O read, I/O write same sequence of steps will be taken except different control signals are generated for each operation. Thus, all buses are used in coordination to perform any task of data transfer.

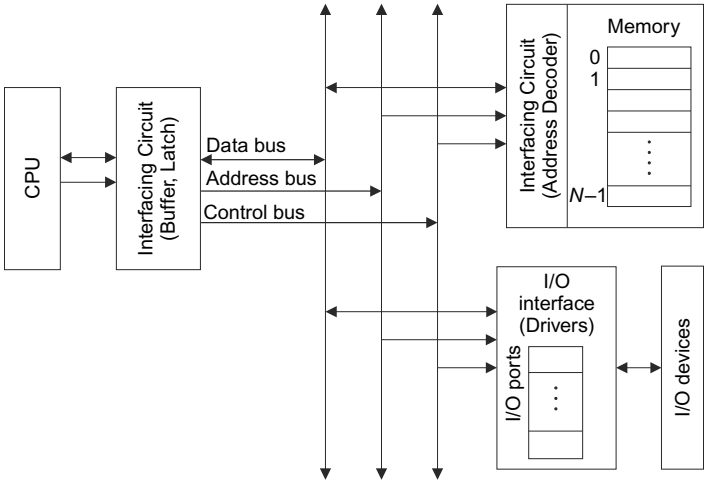


Fig. 1.4 *Computer block diagram with buses and interfacing circuits*

1.1.6 What Can a Computer Do?

A computer performs broadly only two basic types of operations.

1. Data Movement Moving data from one point to another within a system, this includes moving data within circuits inside CPU, storage and retrieval of data to/from memory and data movement to/from I/O devices.

2. Performing Binary Operations Computers perform mainly two types of binary operations. First, logical operations such as AND, OR, NOT, EXCLUSIVE OR, Shifting. These operations provides decision making and control capabilities if used properly. Second, arithmetic operations such as addition, subtraction, multiplication, division, increment and decrement. Any complex operation is realized by sequence of small operations mentioned above.

It is a general misconception that computer spends its major time and efforts in performing arithmetic and logical operations on data. Actually, it spends little time in these operations. Major time is spent in order to locate the desired data items and in moving them within the system, i.e. from memory (or I/O devices) to CPU and vice versa.

1.1.7 How does a Computer Execute Program Instructions?

The process of instruction execution is divided into three cycles.

1. **Instruction Fetch:** Instruction fetch means reading instruction from memory. During this cycle, CPU sends address of the instruction to a memory through address bus. The memory responds by sending instruction byte (or word) to the CPU, where it is held in instruction register.
2. **Instruction Decode:** Decoding means interpreting the instruction and to determine sequence of actions that should be taken to perform the operation specified by an instruction.
3. **Instruction Execute:** In execution cycle, the CPU receives input data either from memory or registers, the result is calculated and finally it is stored back into memory or register as specified in the instruction. All these operations are controlled by control signals generated by a decoder in a predetermined sequence.

Thus, the above three cycles represent overall processing required for a single instruction which is usually referred as instruction cycle. All instructions in a program are executed sequentially by repeating same three cycles. In modern computers, multiple instructions can be fetched, decoded and executed simultaneously.

1.2 MICROPROCESSOR, MICROCOMPUTER AND MICROCONTROLLER

The definitions and brief introduction of these three terms is given in the following section. The term “micro” in the above three terms means small in size, smaller processing time, but it does not mean small processing power.

1.2.1 Microprocessor

The microprocessor is a central processing unit (CPU) built into a single semiconductor chip. The structure of microprocessor is same as CPU discussed in Section 1.1.1 (central processing unit) and as shown in Figure 1.2. The term CPU was used in early days of computers when *CPU* was made using discrete components. Microprocessors are

general-purpose and programmable devices, suitable for many applications like word processing, computing, audio/video/image processing, gaming, database management, monitoring and control of machines. They can be made to perform the different tasks as per user's choice which makes possible to design a system with a great flexibility. It is possible to configure a microprocessor based system as a large system or a small system by adding suitable peripherals.

The microprocessor alone is not sufficient to make a functional system. Additional peripherals and circuitry like memory, input/output devices, decoders, drivers, etc., are required to be connected with it to make a functional system.

The microprocessor is often referred as processor, CPU or Microprocessor Unit (MPU) and these terms are used interchangeably.

1.2.2 Microcomputer

The microcomputer is a small computer built using a microprocessor as a central element. It includes all necessary components required for an application. The I/O devices and memory (types and amount) of a microcomputer are chosen as per the specific application. For example, the personal computers usually contain a keyboard and mouse as input devices. Figure 1.5 shows a typical microcomputer system.

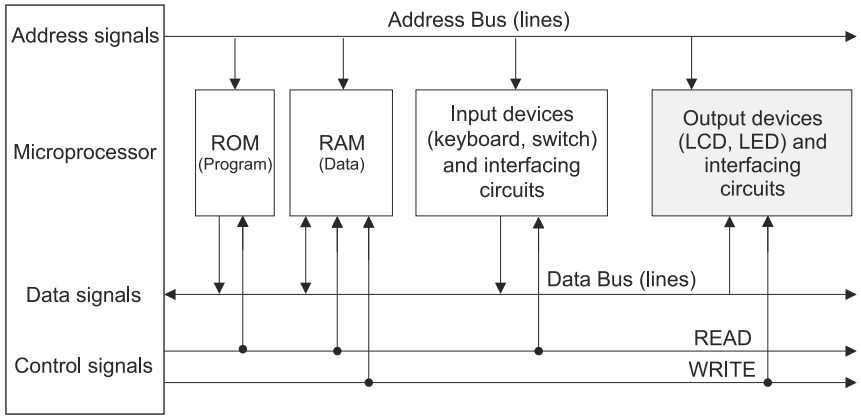


Fig. 1.5 *Microcomputer system built around a microprocessor*

1.2.3 Microcontroller

The microcontroller is an entire computer built into a single semiconductor chip. The term "micro", as mentioned above, means small in size, and the term "controller" means they are normally used to control the machines or gadgets. Hence, by definition, microcontrollers are designed for machine control and/or monitoring applications, rather than direct human interactions.

A microcontroller contains all (or major) components in a single chip to make systems based around them standalone. It includes data and code memory, various on-chip peripherals like timers/counters, serial port, A/D converters, D/A converters, etc., interface controllers, and general purpose I/O ports which allow it to directly interface to external environment. The amount and type of memory, I/O and on-chip peripherals varies across wide range of available microcontrollers. The specific microcontroller is chosen based on the requirements of the end application. The block diagram of a microcontroller is shown in Figure 1.6.

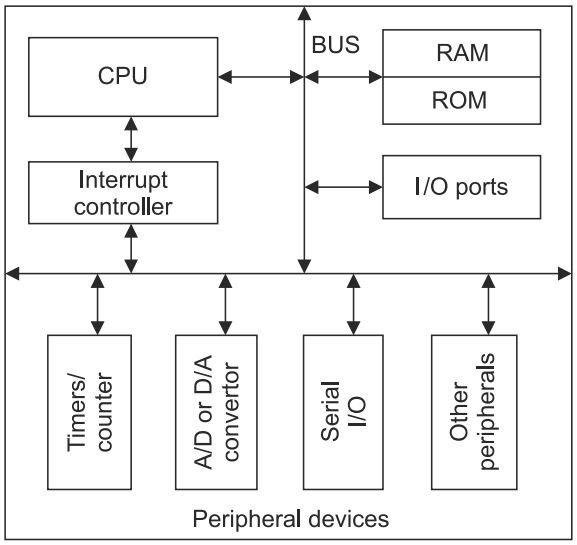


Fig. 1.6 *Block diagram of microcontroller*

Microcontroller based products are generally physically smaller, more reliable and cheaper than microprocessor based products like personal computers. They are most suitable for applications where cost per unit, size and power consumption are very important factors.

The key features of microcontrollers are that they are embedded within an application or a product (often a consumer product) and are usually designed to perform a few specific control oriented operations, for example, microcontrollers are widely used in modern cars where they will perform dedicated tasks, i.e. to regulate the fuel injection in an engine, to regulate brakes of all wheels, or to control the car's air conditioning, or a microcontroller responsible for the air-bag control. Since microcontrollers are usually embedded inside some other device, they are also referred as embedded controllers.

In conclusion, a microcontroller contains all resources within a chip (on-chip) to satisfy need of an average application or project. This integration of all necessary resources will offer following advantages:

1. On-chip peripherals have smaller access time; hence, speed is more.
2. Less number of chips, less wiring, reduced PCB size which in turn reduces development time, manufacturing cost and increases reliability of the system, which are the key factors in the embedded system design.
3. Easy maintenance and future upgradeability

THINK BOX 1.2

Manufacturing cost and product size are the two most important criteria (factors) in a new product designs. How do microcontrollers help achieve the above criteria?

The microcontroller integrates CPU and all other hardware resources like memory, timers, I/O circuits and other peripherals in a single chip. This will require a small PCB for the final product, moreover, integration of major hardware resources in a single chip will reduce design, development and debugging time which will reduce the price of a final product.

1.2.4 Comparison between Microprocessor and Microcontroller

Based upon above discussion the brief comparison between microprocessor and microcontroller is presented in Table 1.1 with respect to their hardware components, instruction set and applications.

1.3 CLASSIFICATION OF MICROCONTROLLERS

The microcontrollers/processors can be classified on the basis of word length, memory architecture, core (CPU) architecture and instruction set architecture.

Table 1.1 *Comparison between microprocessor and microcontroller.*

Microprocessor	Microcontroller
Microprocessor is complete functional CPU, i.e. it contains ALU, registers, stack pointer, program counter, instruction decode and control unit and interrupt processing circuits.	Microcontroller is complete functional microcomputer, i.e. it contains the circuitry of microprocessor and in addition it has built in memory (ROM, RAM), I/O circuits and peripherals necessary for an application.
Microprocessor instruction sets are data processing intensive, means powerful addressing modes and many instructions to move data between memory and CPU to handle large volumes of data.	Microcontrollers have instruction sets that are related to the control of inputs and outputs, means they have many bit handling instructions along with byte processing instructions.
Microprocessor based products are primarily designed to interact with humans and are more flexible to design.	Microcontroller based products are primarily designed to interact with machines; once a system is designed they are less flexible.
Access times for external memory and I/O devices are more, resulting in a slower system.	Access times for on-chip memory and I/O devices are less, resulting in a faster system.
Microprocessor based systems require support devices and are usually bulkier, costly, less reliable and consume more power.	Microcontroller based systems require less external hardware, reducing PCB size and hence are compact, cheaper, more reliable and consume less power.
Software protection is not possible because of the requirement of external code memory.	Software protection is possible because of on-chip code memory.

1.3.1 Word Length: 4 , 8, 16, 32, 64-bit Microcontrollers

The number of bits in a binary pattern that a microcontroller understands and processes at a time is called word length, i.e. number of bits on which ALU can perform operations at a time. Microcontrollers are classified according to their word length, it ranges from 4 or 8 bits for small systems, 16 bits for medium systems and 32 or 64 bits for high-speed large systems.

Bus width, internal bus width, register width and bits are the different terms used interchangeably for the term word length.

The COP400 family from National, and TMS 1000 from Texas Instruments are 4-bit microcontrollers. 8048 family (MCS 48), 8051 family (MCS 51) from Intel, COP 800 family from National, 6805, 6811 from Motorola, PIC16 from Microchip are all 8-bit microcontrollers. 8051XA, 8096 from Intel, 6812 from Motorola are 16-bit microcontrollers. The Intel 251 family, 683XX, ARM 7, 9, 11 based microcontrollers are 32-bit microcontrollers.

Microcontrollers of MCS 51 family are known as 8-bit microcontrollers as they can operate on 8 bits at a time.

THINK BOX 1.3

What is the advantage of wider word length (or bus width)?
Increased data handling and processing capacity (per machine cycle)

1.3.2 Memory Architecture: Von Neumann and Harvard Architectures

This classification of microcontrollers is mainly based on the organization of a program and data memory. The basic characteristics and comparison between these architectures is presented below.

1. Von Neumann Architecture It has a single memory storage to hold both program instructions and data, i.e. common program and data space. The CPU can either read an instruction or data from the memory one at a time (or write data to memory) because instructions and data are accessed using same bus system. The Von Neumann Architecture is named after the mathematician and computer scientist John Von Neumann. The basic organization of memory in this architecture is shown in Figure 1.7.

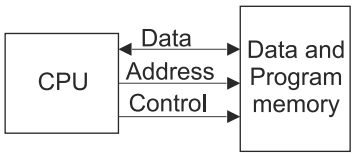


Fig. 1.7 *Von Neumann architecture*

The advantage of Von Neumann architecture is simple design of microcontroller chip because only one memory is to be implemented which in turn reduces required hardware. The disadvantage is slower execution of a program. It is also referred as Princeton architecture as it was developed at Princeton University. Motorola 68HC11 microcontroller is based on Von Neumann architecture.

2. Harvard Architecture It has physically separate memory storage to hold program instructions and data, i.e. separate program and data space. Since it has separate buses to access program and data memory, it is possible to access program memory and data memory simultaneously. The organization of memory and buses in this architecture is shown in Figure 1.8.

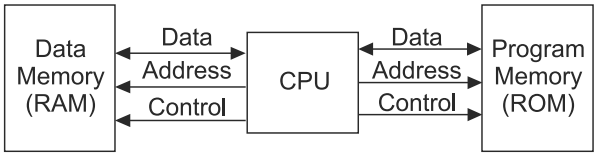


Fig. 1.8 *Harvard architecture*

The advantage of a Harvard architecture microcontroller is that it is faster for a given circuit complexity because it offers greater amount of parallelism. The disadvantage is that it requires more hardware, because two sets of buses and memory blocks are required. MCS 51 (8051 family) and PIC microcontrollers are based on Harvard architecture.

1.3.3 Core Architecture: Microcoded and Hardwired Designs

This classification of microcontrollers is mainly based on design of execution unit (ALU + Instruction decode and control unit). The term *core* is used collectively for all circuitry responsible for execution of the instructions.

1. Hardwired Design A hardwired microcontroller/processor uses the bit patterns of the instructions to select and activate specific circuits (may be unique to each instruction) to execute the instructions. All control signals (or sequence of steps) required to fetch, decode and execute the instructions are generated and controlled by combinatorial logic and state machine circuitry. Hardwired core design is shown in Figure 1.9.

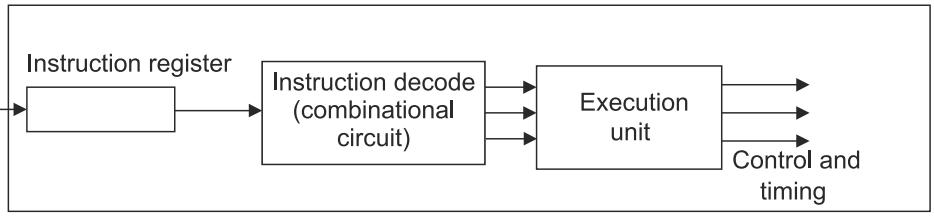


Fig. 1.9 *Hardwired core*

2. Microcoded Design

Microcode is a group of instructions (usually referred as microinstructions) used to implement the instructions of a microcontroller/processor. It resides in a ROM or a programmable logic array (PLA) that is part of the microcontroller chip. The microinstruction is group of bits (stored in ROM) used to represent the sequence of control signals to fetch,

decode and execute the instruction, i.e. control signals (in a sequence) for every instruction are generated using memory. The microinstructions are at an even more detailed level than machine language.

Every instruction has its own microinstruction sequence (microprogram).When an instruction is placed into the instruction register, a few bits of the instruction will identify address of microcode (or effectively instruction routine) and finally the control signals are generated in a sequence to complete a specified operation. The original 8051 is based on microcoded design, which requires 12, 24 or 48 clock cycles for each instruction to execute. The microcoded core is shown in Figure 1.10.

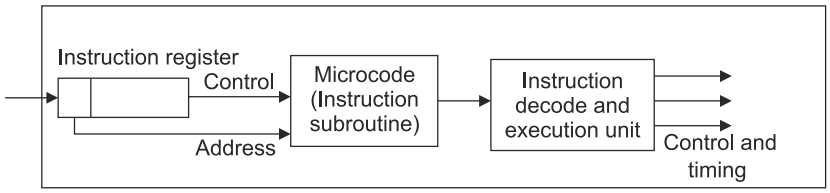


Fig. 1.10 *Microcoded core*

Both designs have advantages as well as disadvantages. A microcoded core is simpler to design, and can be implemented faster. A hardwired microcontroller is more complex to design because instruction decoder will be complex and difficult to debug and is less flexible. The advantage of hardwired design is that it will execute instructions faster.

1.3.4 Instruction Set Architectures: CISC and RISC

This classification is based on capabilities and type of instructions of a microcontroller.

1. CISC: Complex Instruction Set Computer

In the early days of the computer development, most of the program development work was done in assembly language because high-level languages were not developed. Therefore, CPU designers tried to make instructions that will perform as much work as possible, which in turn has led to development of instructions with many addressing modes for arithmetic, logical and data transfer operations. These powerful instructions provide greater flexibility in performing operations. The CISC architecture requires less number of instructions per program at the cost of number of cycles per instruction. The 8051 microcontroller is based on CISC architecture, for example multiply and divide instructions are complex instructions for which operation is performed in hardware.

The common characteristics of CISC architecture are

(a) **Complex hardware:** complex as well as more addressing modes, variable instruction size
(b) Many clock cycles to execute an instruction
(c) High code density—small program size
(d) Complex data types

2. RISC: Reduced Instruction Set Computer

The simple instructions which perform a few operations at a time will provide high performance because of less hardware requirements for instruction decoder. Thus, instructions require very less time to execute. The RISC instructions have few addressing modes supported by all instructions. It reduces the cycles per instruction at the cost of the number of instructions per program. The Microchip PIC microcontrollers are based on RISC architecture.

The common characteristics of RISC architecture are

(a) **Simple hardware**: simple and less addressing modes, fix instruction size
(b) Single clock cycle execution, uniform instruction format
(c) Low code density—larger program size
(d) Few data types in hardware
(e) Emphasis is on software: Compiler design is more complex

THINK BOX 1.4

What are the main architectural differences between microprocessors/ controllers?

- Word size (size of the ALU)
- Addressable memory (memory addressing capacity)
- CPU design (core architecture)
- Instruction execution speed (maximum clock speed supported)
- Instruction set (type of instructions, addressing modes)
- Number of registers

1.4 CHOOSING A MICROCONTROLLER

The microcontrollers are chosen mainly as per requirements of a particular application and a product. The major factors that have to be considered for selecting microcontrollers are discussed briefly in the following section.

(a) Computational Requirements The microcontroller should have enough speed and processing capability to handle all operations of an application in a real time. The speed should be just sufficient to meet the computational needs of an application efficiently. Higher operational speeds than required will unnecessarily increase power consumption.

The microcontroller word length i.e., 8, 16, or 32 bits should match most of the data types to be processed. Hence, it is a major factor in evaluating computational capabilities and suitability of microcontroller for an application.

(b) Hardware Resources Requirement The microcontroller should have sufficient program and data memory to store and execute an application program. An application program should take around 70–80% of the microcontroller's program memory; this will provide space for future upgradations. Microcontroller should have enough I/O pins to provide user interface and connectivity to other modules of an application. It should also have all (or maximum) peripherals like timers/counters, ADC, DAC, serial port, etc., on chip to make product as compact and as reliable as possible.

(c) Power Requirements A microcontroller should have low power consumption. It is critical factor for portable and battery powered products.

(d) Software and Hardware Development Tools and Family Cost and availability of the software development tools like compiler/assembler, debuggers, emulators or Integrated Development Environment (IDE) are important factors to choose a microcontroller. Availability of software libraries and software building blocks will ease the product design. The design team should be well versed with all this tools and family of microcontroller. For example, if design team has experience and expertise in Intel 8051 family, then choosing the 8051 family microcontroller will lead to perfect design of the product. This saves development efforts and reduces implementation time.

The third-party support for all tools is also an important factor to choose a microcontroller.

(e) Cost It is a major factor in selection of a microcontroller. Designer should select cheapest microcontroller that satisfy applications need.

(f) Availability The microcontroller and support chips, if any, should be available easily in enough quantities now and in future throughout life cycle of a product.

(g) Future Upgradeability and Maintenance The microcontroller's ability to upgrade to higher performance or low power versions in future and ease of maintenance of the product also have to be considered.

1.5 APPLICATIONS OF MICROCONTROLLERS

Microcontrollers have changed the way we live in past few decades. They have entered in almost all aspects of our life. Their production counts are in the billions per year, and are integrated into diversified appliances as discussed below.

1. **Household appliances:** Microwave oven, washing machine, coffee machines, refrigerators, digital cameras, alarm clocks, toys, home security systems, remote controllers, exercise machines, sewing machines, air conditioners, etc.
2. **Office and commercial appliances:** Fax machine, photocopier, scanner or printer machine, intercom, computer systems (discussed below), calculators, ATM machines, CCTV camera and surveillance systems, point of sale systems, weighing scales, elevators, lifts, and many products included in household appliances.
3. **Telecommunication:** Telephones, phone answering machines, mobile phones, satellites, etc.
4. **Entertainment and gaming:** Televisions, VCRs, music players, stereo systems, set-top boxes, play stations, video games, musical instruments, etc.
5. **Automotive industry:** Fuel injection, ABS, ignition, power windows and seats, climate control, air bags, brake control, etc.
6. **Industrial automation and manufacturing:** Motor control systems, data acquisition and supervisory systems, industrial robots, electronic metering, etc.
7. **Electronic measurement instruments:** Digital multi-meters, frequency synthesizers and oscilloscopes, logic analyzers, spectrum analyzers, digital thermometers, tachometers, etc.
8. **Biomedical systems:** ECG recorder, blood-cell analyzers, glucose monitor, patient monitoring systems, etc.

9. **Computer systems:** Keyboard controller, CD drive or hard-disk controller, CRT controller, DRAM controller, printer controller, LAN controller, etc.
10. **Military weapons, guidance and positioning systems.**
11. **Aerospace industry.**
    And any automatic or semiautomatic devices around us usually contain microcontrollers.

1.6 HISTORY AND INTRODUCTION TO THE 8051 MICROCONTROLLER FAMILY

Intel introduced the first microprocessor, the 4004, in 1971. This was the beginning of the microprocessor revolution. The 4004 was a 4-bit microprocessor with 640 bytes of memory addressing capacity and 108 KHz clock. It was developed for an electronic calculator. Later, Intel introduced the 8-bit microprocessors, the 8008 and its upgraded version—the 8080. Today, 64 bit (as well as 128 bit) microprocessors with nearly 3 GHz clock and 1TB addressing capacity are available. Till today Intel is the leading company in developing and manufacturing new microprocessors.

In 1976, Intel introduced its first microcontroller, the 8048. It integrated the processer core with program and data memory, two register banks, 1 timer, 27 I/O lines and 2 interrupts. It has 1KB mask ROM as a program memory and 64 bytes of on-chip RAM. It also has external memory support.

In 1980, Intel introduced the 8051. It was an extension to the 8048. The extensions include extra program and data memory, four register banks, multiply and divide instructions, additional timer, UART and five interrupts.

1.7 OVERVIEW OF THE 8051 FAMILY

The 8051 is an 8-bit microcontroller, i.e. 8 bit internal data bus width. It is optimized for 8 bit mathematical and Boolean (single bit) operations. Its family includes 8031, 8032, 8051, 8052, 8751and 8752 microcontrollers. The architecture of the 8051 family of microcontrollers is referred to as the MCS 51 architecture, or simply as MCS51.

1.7.1 Features of the 8051 (MCS 51) Family

The key features of 8051 microcontroller are

1. 8-bit CPU with Boolean processing capabilities
2. 4K bytes on-chip *program memory
3. 128 bytes on-chip data memory
4. 64 Kbytes each program and external data address space
5. 32 bidirectional I/O lines organized as four 8-bit I/O ports
6. Serial port–Full duplex UART
7. Two 16-bit timers/counters
8. Two-level prioritized interrupt structure
9. Direct byte and bit addressability
10. Four register banks
11. Binary or decimal arithmetic support
12. Hardware multiply and divide operations
13. 12 clock cycles per machine cycle

The simplified block diagram of the 8051 showing hardware resources and their organization is shown in Figure 1.11.

The 8051 microcontroller has 4K masked ROM,128 bytes of RAM, two 16-bit timers, 32 I/O pins, one serial port and five sources of interrupts. Apart from 128 bytes internal RAM, the 8051 has various special function registers (SFRs), which controls on-chip peripherals. Programming of the various on-chip peripherals of the 8051 is achieved by loading the appropriate control words into the corresponding SFRs. The 8031/8032 is similar to the 8051, except it does not contain the on-chip ROM, i.e. they are ROMless devices. As a result, external ROM must be added in a system to store a program. While adding external ROM, two 8 bit ports are engaged (as address/data bus), that leaves only two ports for I/O operations. The 8052/32 has extra 128 bytes of RAM and an extra timer. Since 8051 is subset of 8052, all programs

*Some members do not have on-chip ROM and others have 8K on-chip ROM.

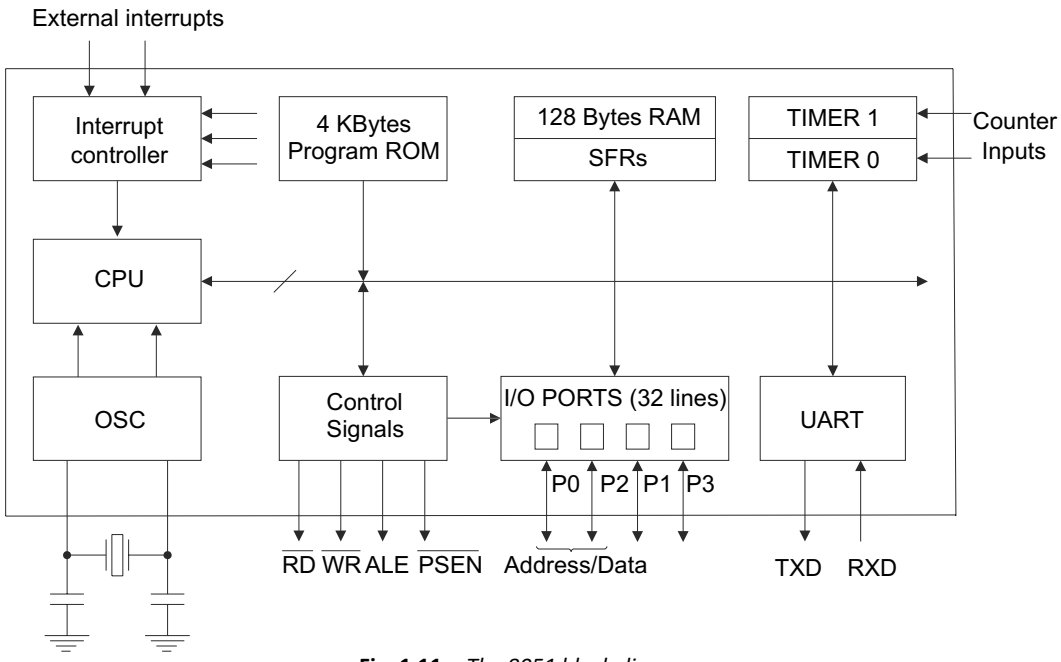


Fig. 1.11 *The 8051 block diagram*

written for 8051 will run on 8052 but reverse may not be always true. Table 1.2 gives the comparison of hardware resources of MCS 51 family of microcontrollers.

Table 1.2 *Comparison of hardware resources of MCS 51 family*

Feature	8031	8051	8751	8032	8052	8752
Program memory	None ROMless	4K ROM	4K EPROM	None ROMless	8K ROM	8K EPROM
Data memory	128 Bytes	128 Bytes	128 RAM	256 Bytes	256 Bytes	256 Bytes
Timers/counters (16-bit)	2	2	2	3	3	3
I/O pins	32	32	32	32	32	32
Serial port	1	1	1	1	1	1
Interrupt sources (Reset not included)	5	5	5	6	6	6

Though the 8051 has 8 bit CPU, its instruction set is also optimized for the Boolean (single bit) operations inherently required in a real world control applications. The Boolean processor provides direct support for bit manipulation. This leads to more efficient programs that need to deal with binary operations frequently used in the machine control applications. Bit addressing can be used for test pin monitoring or user defined program control flags.

The design of MCS 51 family microcontrollers listed in Table 1.2 is based on HMOS (High speed Metal Oxide Semiconductor) technology operating at 12 MHz. CHMOS (complementary HMOS) versions of these family members are also available, they are represented by additional letter 'C' in a part number, for example, 80C51, 87C51. The CHMOS devices have the following advantages:

1. Low power consumption
2. High stability (noise immunity)
3. High speed
4. Support power down and idle modes

1.7.2 8051 Variants and Enhancements

The term "Variants" means the 8051 compatible microcontroller. The 8051 has the widest range of variants amongst all microcontrollers in the market because Intel has licenced other manufacturers to design the microcontrollers based on the 8051 core provided that they should remain code compatible with the original 8051. Today there are around thousand

THINK BOX 1.5

What makes 8051 an 8-bit microcontroller?
The number of bits that ALU can process at a time i.e. size of the ALU. The 8051 has 8 bit ALU.
However there are few exceptions, one of the most popular example is Intel 8088 microprocessor, though it is having 16 bit ALU, it is referred by Intel as an 8 bit microprocessor because it has only 8 bit external data bus interface!

variants of the 8051 manufactured by more than 20 semiconductor companies. The enhancements contain more memory, different on-chip peripherals and higher operating speeds. The enhanced features are under control of additional SFRs. The variants will differ in various aspects like:

1. **Amount of on-chip memory:** RAM and ROM
2. **Type of memory:** RAM—static or dynamic, ROM—Masked ROM, EPROM, EEPROM, Flash, NVRAM
3. **On chip peripherals:** Timers/counters, I/O ports, UART, ADC, DAC, PWM,I2C, SPI, Capture module, Watch dog timer, Real time clock, Analog comparator, USB,CAN, power management module and many application specific peripherals
4. Operating speeds and oscillator sources
5. Operating voltage
6. Power consumption
7. Implementation technology
8. Package type and number of pins

The leading and popular manufacturers of the 8051 variants are Atmel Corporation, Dallas Semiconductors, and Philips (NXP). The 8051 variants from above manufacturers are discussed in detail in Chapter 23. Their product features are briefly discussed below.

(a) Atmel Corporation Flash memory variants, 20/40 pins, varying operating voltages from 2.7 V to 6 V, low cost. These devices can operate from 0 to 24 MHz. Operating with 0 Hz frequency means the contents of RAM are frozen in absence of clock and they do not require refreshing because of SRAM memory. The low frequency operation is preferred when power consumption is an important consideration, for example, for the battery operated devices. The 20 pin members (89C1051, 89C2051) are preferred when small board (PCB) space of final product is desired (provided that the design requirements are fulfilled with only 15 I/O pins). The other important feature is that they have flash ROM that can be erased and programmed electrically.

(b) Dallas Semiconductors NV RAM (battery backed), flash, EPROM variants, program download through serial port eliminating need of EPROM programmer and need of removing chip from the system, this feature is known as In System Programming (ISP), built in real time clocks for few variants, High-speed variants (single clock per machine cycle).

(c) Philips Corporation (NXP semiconductors) Maximum variants, flash memory variants, high integration of built in peripherals, ISP, different packages and pins, low end variants with fewer features, low operational voltage variants, extended I/O, low cost.

1.7.3 Comparison between MCS 51, PIC, AVR and HCS11/12 Families

There are many families of 8-bit microcontrollers from different manufacturers and each family has a wide variety of variants to meet requirements of different applications. Among the various families, the PIC families from Microchip, AVR family from Atmel and HCS 11/12 family from Motorola (Freescale) are most popular. A brief comparison between some of the members from these families is given in Table 1.3.

1.7.4 Advantages of Using 8051 Family of Microcontrollers

The use of 8051 family microcontrollers offers the following advantages:

1. **Availability and support:** Easily and readily available and widely supported, free and commercial third party support is easily available because they are more popular. Hardware and software development tools and training are easily available and are inexpensive. High level language compilers are also available.

Table 1.3 *Comparison between popular 8-bit microcontrollers*

Parameters	MCS 51 (8051)	PIC	AVR	HCS11/12
	89C51	**PIC18F242**	**ATmega32**	**MC68HC912B32**
	89C52	**PIC18F452**	**ATmega64**	**M68HC11K**
	89C54	**dsPIC30F (16 bit)**		
ROM (Flash)	4K	16K	32K	32K
	8K	32K	64K	20K
	16K	Up to 144K		
RAM	128 bytes	768 bytes	2K	1K
	256 bytes	1536 bytes	4K	640 bytes
	256 bytes	8K		
EEPROM	NO	256 bytes	2K	768 bytes
		256 bytes	2K	768 bytes
		4K		
Timers	2(16-bit)	4	3	8
	3(16-bit)	4	4	8
	3(16-bit)	5		
I/O pins	32	34	32	63
		34	53	62
		Up to 54		
Speed up to	33 MHz	40 MHz	16 MHz	25 MHz
		40 MHz	16 MHz	16 MHz
		120 MHz		
Interrupts	5	17	21	24
	6	18	35	22
	6	32		
Power down mode	YES	YES	YES	YES
Watchdog timer	NO	YES	YES	YES
Voltage range	5 V	2 to 5.5 V	4.5 to 5.5 V	6.5 V
				3.0 V to 5.5 V
A to D convertor	NO	10 bit/12 bit	8 CH, 10-bit	8 CH, 10-bit
			8 CH, 10-bit	8 CH, 8-bit
Others	UART	SPI, I2C, PWM In Circuit Debug CAN(dsPIC30F)	JTAG interface, PWM	PWM SPI SCI

2. **Low cost:** High level integration of many peripherals within single chip, only a few external components needed to create a working system.
3. **Effective architecture:** Architecture optimized for the single-bit operations, highly desirable for control applications. Single bit instructions require fewer bytes of code and hence faster execution.
4. **Multiple vendors:** More than 20 manufacturers, more than thousand variants, something for everyone.
5. **Compatibility:** Op-codes are same for all variants, therefore, easy to upgrade to newer variants.
6. **Constant improvement:** Constant improvement in implementation technology and reduction in power consumption.

THINK BOX 1.6

Why are there so many microcontrollers?

These days the microcontrollers are usually designed for specific area of applications. To serve large number of applications (to meet specific requirements of different applications), many microcontrollers with different hardware resource (customized resources) are developed.

1.8 EMBEDDED SYSTEMS

To understand the term "Embedded System", let us first try to see dictionary meaning of the word "Embedded". It means 'hidden inside' or 'fixed' or 'implanted in to bigger system'.

The embedded systems are devices made from combinations of a computer (microcontroller/processor) hardware and software programmed for a fixed and dedicated application(s). The device may be part of (or *implanted* in to) larger system, or specific part of an application or a product (often a consumer product). The computers (microcontrollers/ processors) are *hidden* in the system. It means, the software that controls the application is permanently fixed in to ROM and is not accessible to user of the device. This is the key difference between embedded systems and general purpose computers which can be configured and programmed as per user's choice. The most common examples of embedded systems are washing machines, microwave oven, digital clock, fax machine, cell phones, etc…

The key characteristics of embedded systems are the following:

1. They are designed to perform specific (or limited) tasks.
2. They are tightly constrained with respect to power consumption, size, design, testing and manufacturing costs. These constraints are achieved by selecting microcontroller speed just sufficient to satisfy computational needs, limited memory, and limited peripheral resources to achieve design goal.
3. They guarantee the response to events and completion of tasks within specified time. This is more popularly known as real time operation.

The microcontrollers are most important parts of embedded systems; therefore it is necessary to have knowledge of microcontroller architecture, programming and interfacing with real world to design an embedded system. **This book is written to serve this purpose.**

Embedded Microcontrollers

When all resources (peripherals + memory) required for an application are available within microcontroller chip, it is called an embedded microcontroller. The only additional requirements are power supply, clock and preferably reset circuit.

THINK BOX 1.7

Which architectural techniques are used in newer microprocessor/ controller designs for improved performance?

• Parallel processing • Co-processing • Wider buses • Cache memory • Pipelining

POINTS TO REMEMBER

- The basic operations performed by the computers are storing and retrieving binary numbers, arithmetic and logical operations on the binary numbers.
- The major components of a digital computer system are CPU, memory, I/O unit and system bus.
- The major components of a CPU are ALU, Instruction decode and control unit, registers, program counter, instruction register and interrupt controller.
- The address, data and control buses are required for communication between blocks of a computer system.
- Microcontroller is an entire computer built into a single semiconductor chip.
- Microcontroller-based systems are compact, fast, easy to design and debug, cheaper and consumes less power.
- Microcontrollers/processors can be classified on the basis of Word length, memory architecture, core architecture and instruction set architecture.
- Complex hardware, more addressing modes, variable instruction size and high code density are the key features of the CISC architecture.
- Simple hardware, less addressing modes, fixed instruction size, single clock cycle execution and uniform instruction format are the key features of the RISC architecture.

- The factors affecting microcontroller selection are computational requirements, hardware resources and power requirements, cost, availability, future upgradeability and ease of maintenance.
- The 8051 is an 8-bit microcontroller, i.e. 8-bit internal data bus width optimized for 8-bit math and single bit Boolean operations.
- The variants of 8051 differ in various aspects like on chip memory, on-chip peripherals, operating speeds and oscillator sources, operating voltage and power consumption.

OBJECTIVE QUESTIONS

1. A microcontroller with 64 KB of code memory will require a program counter of,
(a) 8-bit size (b) 16-bit size (c) 64-bit size (d) none of the above

2. An 8-bit microprocessor has essentially,
(a) 8-bit ALU (b) 8-bit Accumulator
(c) most registers are 8-bit registers (d) all of the above

3. In a microcontroller, execution speed of a given program depends upon,
(a) on-chip/off-chip program memory. (b) number of address lines.
(c) size of the program counter (d) clock frequency

4. The microcontroller is useful in systems that have fixed programs for dedicated applications.
(a) True (b) False

5. Microcontrollers usually have,
(a) CPU (b) RAM+ROM (c) peripherals (d) all of the above

6. In RISC architecture, instructions are simpler in order to,
(a) reduce the hardware complexity (b) enhance clock frequency
(c) reduce compiler complexity (d) all of the above

7. CISC machines,
(a) have fewer instructions than RISC machines (b) use more RAM than RISC machines
(c) have medium clock speeds (d) use variable size instructions

8. RISC machines typically,
(a) have high capacity on-chip cache memory (b) have fewer registers than CISC machines
(c) are less reliable than CISC machines (d) execute 1 instruction per clock cycle

9. Harvard architecture has,
(a) separate data and program memory (b) unified cache memory
(c) multiple functional units (d) an on-chip cache

10. What is the difference between the 8031 and the 8051?
(a) the 8031 has no interrupts (b) the 8031 is ROMless
(c) the 8051 is ROM-less (d) the 8051 has 64 bytes more memory

11. A microcontroller is also referred as computer on a chip.
(a) True (b) False

12. An embedded microcontroller usually means,
(a) a microcontroller for the embedded system
(b) microcontroller with embedded processor
(c) a microcontroller with external memories storing the embedded software
(d) microcontroller with all necessary resources available on-chip

13. The basic operations performed by a computer are _____.
(a) arithmetic operations (b) logical operations (c) data movements (d) all of the above

14. Which of the following are output devices?
(a) Keyboard (b) printer (c) display screen (d) scanner

15. _____address lines are required to address 2K X 8 bit memory.
(a) 11 (b) 14 (c) 10 (d) 16

16. _____ can understand the difference between instructions and data.
    (a) input/output device (b) memory
    (c) microprocessor/controller (d) system bus
17. The variants of 8051 microcontroller differ in,
    (a) amount of on-chip memory (b) operating speed
    (c) on-chip peripherals (d) all of the above
18. Embedded systems are tightly constrained with respect to,
    (a) power consumption (b) size (c) cost (d) all of the above
19. Microcontroller based systems are usually,
    (a) compact (b) less power hungry (c) very much expensive (d) slower
20. Signals required for memory devices are,
    (a) address lines (b) control lines (c) data lines (d) all of the above
21. _____ is ROM less variant of the 8051.
    (a) 8031 (b) 8051 (c) 8751 (d) 8052
22. The most common examples of embedded systems are,
    (a) washing machines (b) cell phones (c) digital clock (d) all of the above

Answers to Objective Questions

1. (b)	2. (d)	3. (d)	4. (a)	5. (d)	6. (a), (b)	7. (d)	8. (d)	9. (a)
10. (b)	11. (a)	12. (d)	13. (d)	14. (b), (c)	15. (a)	16. (c)	17 (d)	18. (d)
19. (a), (b)	20. (d)	21. (a)	22. (d)					

REVIEW QUESTIONS WITH ANSWERS

1. List the basic components of a computer system.

A. CPU, memory, I/O unit and system bus.

2. What is the function of an instruction decoder?

A. It interprets the instruction present in an instruction register and determines control signals to be generated to execute an instruction.

3. What information is stored in memory?

A. It stores program instructions, data on which operation are performed and temporary results.

4. List the signals required by memory devices.

A. Address, data and control signals. Control signals also include chip select signal(s).

5. How many memory locations can be addressed by 20 address lines?

A. 2^{20} = 1048576 locations, numbered from 0 to 1048575. It is commonly referred as 1Mega locations.

6. What is meant by a term 'general purpose' used with microprocessors?

A. The user can decide the tasks to be performed by a system as per his/her requirements (of course, within its capability and limitations!)

7. Microcoded and hardwired designs are classified with respect to _____.

A. Core design.

8. What is the key feature of CISC instruction set?

A. Powerful instructions that will do as much work as possible.

9. What is meant by 8-bit CPU?

A. CPU can recognize and process 8 bits at a time.

10. Which register holds the address of the next instruction to be executed?

A. Program counter.

11. Why ROM-less versions of microcontrollers exist?

A. ROM-less versions are used to develop prototype of an application.

12. "Data bus must be bidirectional." Justify the statement.

A. Because it has to write as well as read data to/from memory or peripherals.

13. For a battery-driven product, what is the most important factor in choosing microcontroller and other components?

A. Power consumption

14. List microcontroller-based embedded products attached to the PC.

A. Mouse, keyboard, disk drives, CRT drivers, printers.

15. Which version of the 8051 is suitable for mass production?

A. 8051. It does have 4K masked ROM, which is programmed by manufacturer while production of a chip, which is very much cost effective.

16. What is the size of internal memory in the 8051?

A. 128 bytes of RAM, 21 special function registers (not discussed yet) and 4Kbytes of ROM

17. List the additional hardware components commonly available in a microcontroller.

A. On-chip memory (RAM and ROM), I/O circuits, Timers/Counters, Interrupt and timing circuits, etc.

18. Why is ROM named so?

A. They can only be read by program instructions. They cannot be written by program instructions. Additional hardware support is required to write to them.

19. What is the key feature of a Harvard architecture?

A. It has separate program and data memory and their signals, allowing the parallel access to both of them, which improves the system performance (by providing parallel processing), i.e. fetching of one instruction (from code memory) and execution of the other may be performed simultaneously.

20. How does the size of an on-chip ROM matters?

A. Larger on-chip ROM allows complex and larger programs to be written allowing more (and complex) features supported by the product without having extra memory connected. It may also allow future upgradation easily if there is spare on-chip memory.

EXERCISE

1. Discuss the differences between microprocessors and microcontrollers.
2. List the applications of microcontrollers.
3. List the different on-chip resources available in microcontrollers.
4. Compare hardwired and microcoded designs of a microcontroller. Discuss their advantages and disadvantages.
5. Justify statement “CISC instructions offers higher code density”.
6. How Boolean processing capabilities of the 8051 are most suitable for the control applications?
7. List the features of the 8051 microcontroller. Compare different family members with respect to available on-chip resources. Why do different family members exist?
8. List the various aspects by which microcontroller variants differ.
9. Discuss the importance of having larger on-chip RAM.
10. What are the advantages of using 8051 family of microcontrollers?
11. Discuss in detail factors affecting choice of microcontroller for a given application.
12. List the differences among various 8051 family members.
13. “The microcontroller’s word length, i.e. 8, 16, or 32 bits should match most of the data types to be processed in an application.” Justify.
14. How does Von Neumann architecture simplify design of a system?
15. Explain how RISC architecture improves the performance of a microcontroller?
16. What are the different ways of classifying the microcontrollers?
17. For RISC based microcontrollers, compiler design is more complex. How?
18. What is in-system programming? What is its advantage?
19. “Higher operational speeds of microcontrollers than required will unnecessarily increase the power consumption of a system”. How?
20. Make a list of manufacturers of microcontrollers and list different microcontrollers made by them along with features and on-chip resources.

2

Programming Model and Architecture of the 8051

Objectives

- Create an overview and significance of the architecture and programming model
- Discuss on-chip RAM address space organization in the 8051
- Discuss program memory (ROM) and external RAM address space organization in the 8051
- List the special function registers (SFRs) of the 8051 and their functions
- List and discuss the importance of flags
- Show the significance of the bit addressability
- Introduce the concept of the stack and its use

Key Terms

- Accumulator
- Bit addressability
- Data memory
- Data pointer
- Flags
- Memory Organization
- On-chip/Off-chip Memory
- Peripheral Control Registers
- Peripheral Data Registers
- Program Counter
- Program Status Word
- Program/Code Memory
- R0 to R7
- Register Banks
- Special Function Registers
- Stack
- Stack Pointer
- Status Register

2.1 THE 8051 ARCHITECTURE

The architectural block diagram of the 8051 is shown in Figure 2.1, it shows organization of all hardware components and data-path connections between them. It includes 8-bit ALU along with Boolean processing capabilities, program and data memory, four 8-bit I/O ports, two timers/counters, UART, timing and control circuits and oscillator circuit.

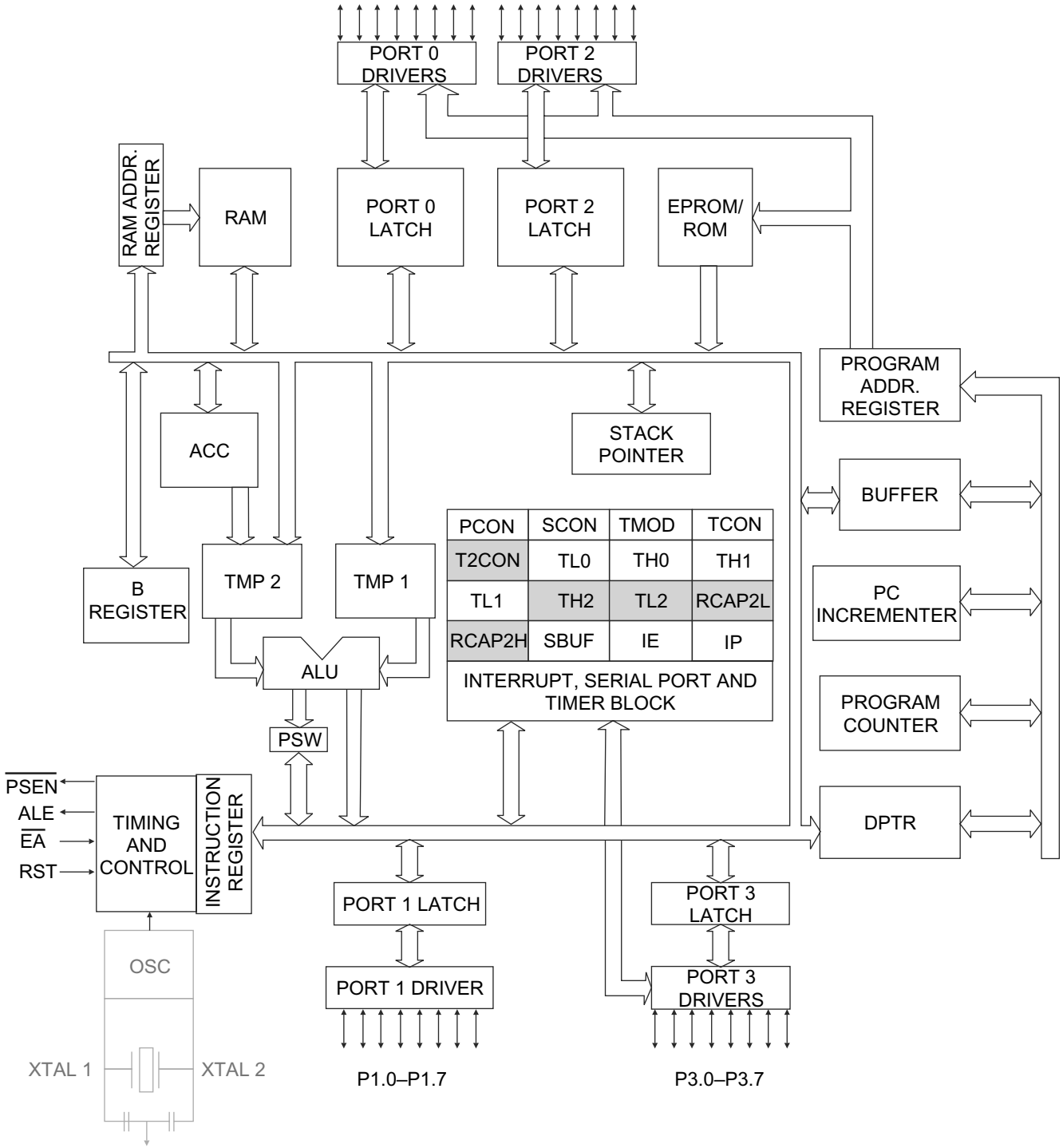


Fig. 2.1 *Block diagram of the 8051 microcontroller*

1. ALU

The Arithmetic and Logic Unit (ALU) performs all arithmetic (addition, subtraction, multiplication and division) and logical (AND, OR, NOT, EXCLUSIVE-OR and rotating) operations on 8-bit data, i.e. the 8051 has 8-bit ALU. The ALU also updates information about the nature of the result in the flag register (PSW).

2. Memory

The 8051 family has separate on-chip program and data memory. The program instructions are stored in a program memory (ROM/EPROM/EEPROM/Flash based on a family member). The amount and type of on-chip program memory is the key factor that differentiate all the members of the family, for example, 80C51 has 4Kbytes of on-chip ROM, whereas 80C52 has 8Kbyte (ROM), 87C51 has 4Kbytes (EPROM) and 87C52 has 8Kbytes(EEPROM) of on-chip program memory. Total program memory (including on-chip ROM) that can be connected with the 8051 is 64Kbytes. Similarly, data memory can be on-chip or off-chip. Internal data memory (RAM) in 80C51 is 128 bytes and in 80C52 is 256 bytes. There are also on-chip RAM locations which are used to program and control various on-chip hardware peripherals and features of the 8051. They are known as *Special Function Registers (SFRs)*. These memories are discussed in detail in the next section.

3. Peripherals

The 8051 has two 16-bit timers (8052 has three timers) that are used for timing and counting applications. It has full duplex serial port (UART) to handle serial data transmission and reception.

4. Timing and Control Unit

This unit generates all timing and control signals necessary for the execution of instructions and synchronizes all internal activities with the clock.

5. Oscillator

The 8051 has an internal (on-chip) oscillator circuit (partial circuit) which generates the clock pulses by which all internal operations are synchronized. The external resonant circuit is connected with this internal on-chip oscillator circuit to make a complete oscillator. Normally quartz crystal is used to make oscillator functional. Typically, 12 MHz (or 11.0592MHz to support standard baud rates for serial port) crystal is used.

All other blocks of the 8051 are discussed in detail in the subsequent chapters.

2.2 | PROGRAMMING MODEL OF THE 8051

Programming model is programmer's view of a microcontroller/processor. It shows only those components (memory or registers) which can be accessed (read/write) by a programmer along with their internal organization. It is a collection of internal registers (and memory locations) that can be used by a programmer to develop any software (i.e. control and application programs) and to use several features of a particular microcontroller. As these registers and memory locations are used by software instructions, it is necessary to have knowledge of the programming model before we start developing any program.

The programming model of the 8051 is shown in Figure 2.2. It contains 8 (or 16) bit registers and memory locations. Each register (or memory location) has an internal 1 byte address with exception of program counter. Some registers are byte as well as bit addressable, i.e. whole byte of data stored in a register can be accessed (read/write) at a time or individual bits can be accessed at a time. The next section provides overview of all components of the programming model.

THINK BOX 2.1

How does a microcontroller's block diagram differ from the programming model?

The block diagram shows hardware architectural components like logic blocks (circuits or devices) used for data handling and processing. It also shows how these blocks are interconnected. It helps in the hardware design of a system.

The programming model shows only those components (memory or registers) which can be accessed (read/write) by a programmer. It helps to learn and develop software for a microcontroller.

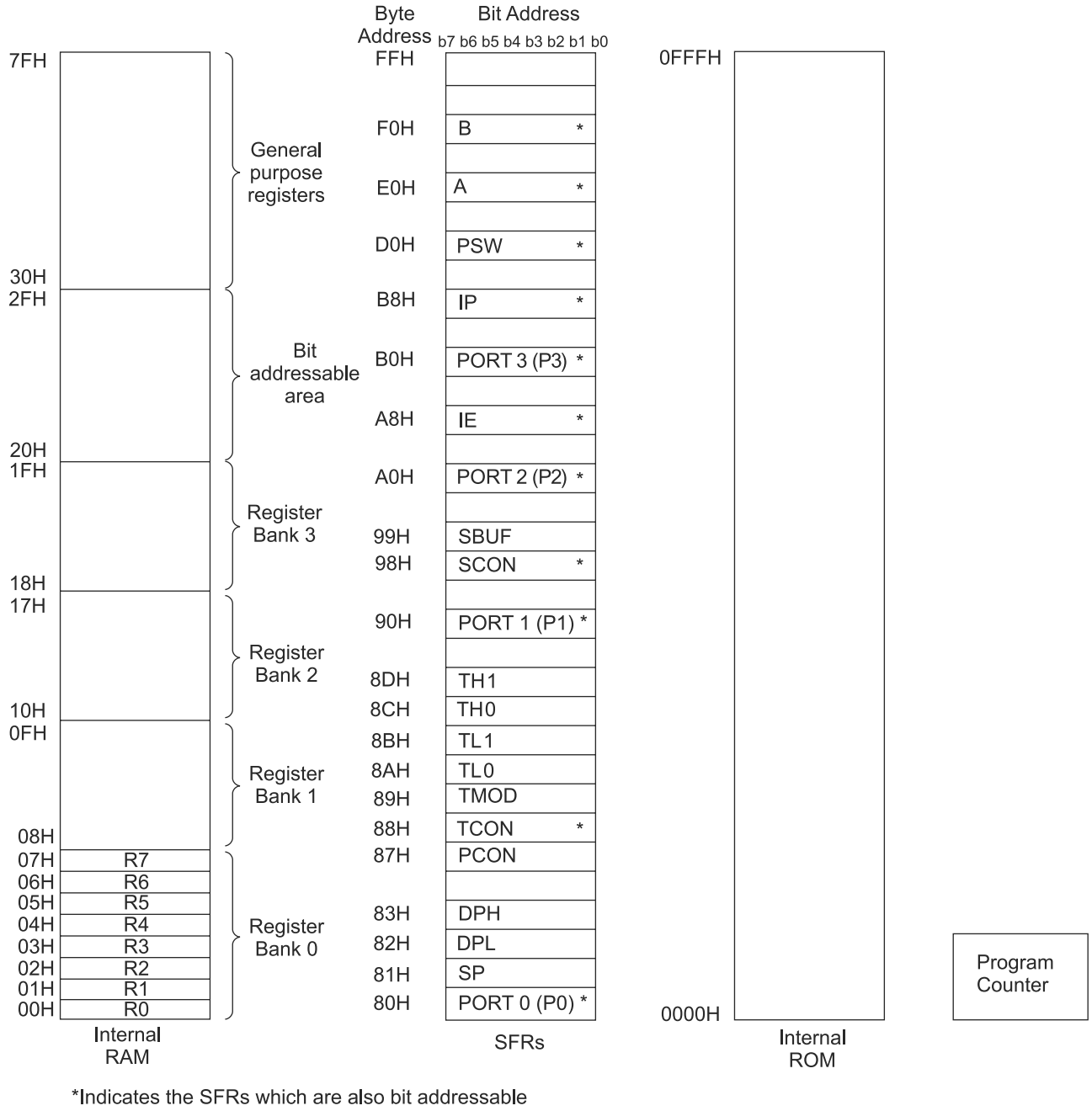


Fig. 2.2 *Programming model of the 8051*

2.3 | ON-CHIP MEMORY ORGANIZATION

The 8051 On-chip memory is organized into three general categories; Special Function Registers, internal RAM and internal ROM as shown in Figure 2.2.

1. Special Function Registers (SFRs)

These registers are used to program and control various on-chip hardware peripherals and features of the 8051. Each SFR has a name which specifies the purpose of the SFR. For example, TCON (Timer Control) register is used to control timer

activities. Note that 128 bytes (80H to FFH) of the SFR address space is available, but only 21 SFRs are defined in the standard 8051. Since SFRs are programmed by instructions, they are RAM locations. SFRs are categorized as follows:

Math registers: A and B
Status register: PSW (Program Status Word)
Program counter: PC
Pointer registers: DPTR (Data Pointer) and SP (Stack Pointer)
Input output port latches: P0, P1, P2, and P3
Peripheral data registers: TL0, TH0, TL1, TH1, and SBUF
Peripheral control registers: IP, IE, TMOD, TCON, SCON, and PCON

2. Internal RAM

The 8051 has 128 bytes of internal RAM. Since it is available on-chip, it is fastest and most flexible in terms of read/write operations. It is used to store temporary data and results. This memory is subdivided into three categories as specified below:

Register Banks: Bank 0, Bank 1, Bank 2 and Bank 3 (00H to 1FH)
Bit Addressable RAM: Memory locations from addresses 20H to 2FH
General Purpose RAM: Memory locations from addresses 30H to 7FH

3. Internal ROM

It is used to store program instructions to be executed by the microcontroller. It may also be used to store permanent data like constants, passwords and lookup tables. The 8051 has 4Kbytes of internal ROM. It is to be noted that different variants of 8051 has different amount and type of on-chip ROM.

2.3.1 Special Function Registers (SFRs)

Brief description and use of all SFRs is discussed below. The purpose of this section is to introduce these registers to a new programmer. Detailed description and programming of each SFR is discussed in the later chapters.

1. Accumulator: A

Accumulator is the most useful and versatile register because it is used in

(a) All arithmetic operations like addition, subtraction, multiplication and division
(b) Majority of logical operations like logical AND, OR, NOT, EX-OR and Rotate
(c) All data transfer between the 8051 and any external memory.

Accumulator register is used to store (collect or accumulate) the result of all arithmetic and majority of logical operations and because of this reason, it is named *accumulator*.

2. B

B is used along with A in multiplication operation to hold one of the operands (either multiplier or multiplicand) and to store higher-order byte of the result. It is also used in division operation to hold divisor and to store remainder of the result. When not used with multiplication or division, it can be used as a general-purpose register where one byte of data may be stored.

3. PSW

Program Status Word is an 8-bit register. It is also referred as flag register or processor status word. Flag is a flip-flop (1-bit storage element) used to store and indicate the nature of result produced by execution of certain instructions. The state of flags (0 or 1) are tested by other instructions (program flow control or branch instructions) to make decisions. PSW structure is explained in Table 2.1.

Table 2.1 *Program status word structure*

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
CY	AC	F0	RS1	RS0	OV	--	P
MSB							LSB

Bit	Symbol	Flag name and description		
7	C (or CY)	Carry; Used in arithmetic, logic and Boolean operations		
6	AC	Auxiliary carry ; useful only for BCD arithmetic		
5	F0	Flag 0; general purpose user flag		
4	RS1	Register bank selection bit 1		
3	RS0	Register bank selection bit 0		
		RS1	RS0	
		0	0	Bank 0
		0	1	Bank 1
		1	0	Bank 2
		1	1	Bank 3
2	0V	Overflow; used in arithmetic operations		
1	--	Reserved; may be used as a general purpose flag		
0	P	Parity; set to 1 if A has odd number of ones, otherwise reset to 0		

CY: Carry Flag It is a carry (or borrow) used in addition and subtraction operations. It is set to 1 when there is carry out from MSB (D7 bit) after an addition (or a borrow into D7 bit during a subtraction). It is also used as the 'Accumulator' for the Boolean operations. It can be directly modified by bit level instructions.

AC: Auxiliary Carry Flag It is a half carry (carry out from bit D3 to D4) used in conventional BCD arithmetic.

F0: Flag 0 It is a general-purpose flag. It can be used as a one-bit memory location to record some event.

RS0 and RS1: Register Bank Select Bits

These bits are used to select the register bank.

OV: Overflow Flag It is set to 1 to indicate that result of signed arithmetic is erroneous (out of range).

P: Parity Flag It indicates the parity of the Accumulator; it is set to 1 if the accumulator register has odd number of ones, otherwise reset to 0, i.e. even parity.

4. Program Counter: PC

Program Counter (PC) is a 16-bit register. It always contains the memory address of the next instruction to be executed, i.e. it points to the instruction that is to be executed next. As the CPU fetches the op-code (instruction byte) from the program memory, the PC is incremented automatically to point to the next instruction. It should be noted that it is not always incremented by one during instruction execution, but it depends on size of the instruction being executed, i.e. if 2-byte instruction is being executed, the PC is incremented by 2.

There is no direct way to modify the PC but it can be modified using jump or call instructions. Same way, there is no direct way to read the value of PC. Since PC is a 16-bit register, the 8051 can access program addresses from 0000H to FFFFH, a total of 64Kbytes of program memory.

Discussion question: What is the value of the PC when the 8051 microcontroller is powered on?

Answer: When the 8051 is powered on, the PC has the value of 0000H in it. This means that it assumes that the byte written at program memory address 0000H is op-code of the first instruction. Therefore in an 8051based system, the first op-code must be written into (burned into) this memory location.

5. Data Pointer: DPTR

DPTR is a 16-bit register. It is used to point to data byte in external data (RAM) or program (ROM) memory. It can be used as a single 16-bit register or can also be accessed as two separate 8-bit registers named DPL and DPH, where DPH means higher byte of the DPTR and DPL is lower byte of the DPTR. DPL and DPH are each assigned a separate address. DPTR does not have single address.

DPTR is under the control of the program, i.e. a programmer can write any value in it at any time as shown in the following instruction.

```
MOV DPTR, #1234H
```

As mentioned earlier, it is used to point to a data byte in (a) External RAM, (b) Internal ROM and (c) External ROM.

6. Stack Pointer: SP

Stack pointer always points to the top of the stack and used to access data from there. It is an 8-bit register. It should be initialized to a defined value (its default value is 07H). While writing a new data byte on the stack, SP is automatically incremented by 1 and data byte is stored at an address SP+1. While retrieving, data will be read from address in SP and then SP is decremented by 1. The data is stored on to the stack using PUSH and CALL instructions and retrieved using POP and RET instructions. Interrupts also use the stack to store the return addresses. The detailed description of the stack and stack pointer is given in later sections of this chapter.

THINK BOX 2.2

How can microcontroller know which memory the DPTR points to?

The value in DPTR register does not imply which memory it is pointing. It is the type of instruction which is using the DPTR as an operand that will decide which memory will be accessed.

For example,

If, DPTR=1000H

```
MOVX A,@DPTR          // will read data from external RAM to A
MOVC A,@A+DPTR        // will read data from external/internal ROM to A
```

7. I/O Port Registers (latches): P0, P1, P2 and P3

The 8051 has four 8-bit ports named as P0, P1, P2 and P3, each can be used as an input or output or both. All ports are byte as well as bit addressable. Each bit corresponds to one of the pin of the microcontroller. P0, P2 and P3 pins have dual functions, but only one function can be used at a time.

8. Peripheral Data Registers: TL0, TH0, TL1, TH1, and SBUF

TL0 (timer 0 lower byte) and TH0 (timer 0 higher byte) together represents a16-bit register for timer 0. The value in this register determines the timing of events controlled by a timer. They are also used as event counters. Similarly, TL1 and TH1 are registers for timer1. SBUF (serial buffer) register is used to hold data to be transmitted or received for serial port.

9. Peripheral Control Registers: IP, IE, TMOD, TCON, SCON, and PCON

IP (interrupt priority) register is used to assign priorities to different interrupt sources. IE (interrupt enable) register is used to enable/disable interrupts. TMOD (timer mode) is used to control behavior, i.e. mode of operation of timers. TCON (timer control) is used to start/stop timers. It also contains the status bits of the timers and status/control bits for the external interrupts. SCON (serial port control) register is used to control the modes of operation of the serial port; it also contains the status bits to indicate completion of data transmission and reception. PCON (power mode control) register is used to select power saving modes of operations, i.e. power down and idle mode. It contains a bit to double the baud rate for serial port and two general-purpose user flags.

2.3.2 Internal RAM

The 8051 microcontroller has a total of 128 bytes of internal RAM. These bytes are assigned addresses 00H to 7FH. These 128 bytes are grouped into three different areas.

1. Register Banks

The first 32 bytes from addresses 00H to 1FH are organized as four banks. Each bank is made up of eight registers named R0 to R7. The four register banks are numbered 0 to 3, i.e. bank0, bank1, bank2 and bank3.The arrangement of register banks along with their addresses is shown in Figure 2.3.

Each bank is assigned different address range, bank 0 address locations 00H to 07H are given names R0 to R7, i.e. 00H is named R0, 01H is named R1 and so on. Similarly, bank1 (08H to 0FH), bank2 (10H to 17H), bank3 (18H to 1FH) locations are given names R0 to R7. Thus, R0 of bank1 is location 08H, and R0 of bank3 is location 18H. This is summarized in Table 2.2.

As per arrangement shown in Figure 2.3, there are four sets of R0 to R7, one set corresponding to each bank. Out of these four banks (set of R0 to R7), only one bank can be accessed at any time. Bits RS0 to RS1 in the PSW (Program

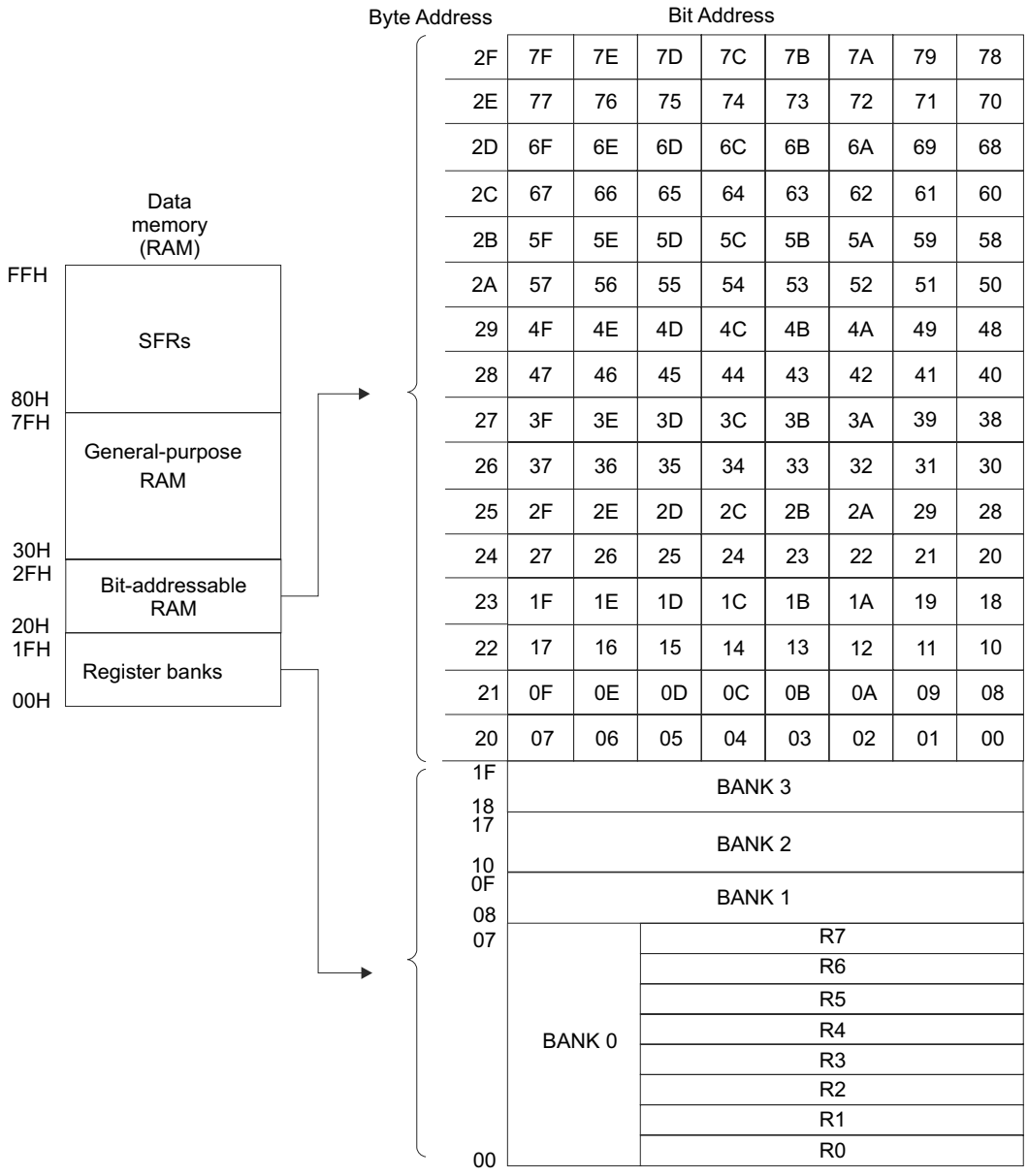


Fig. 2.3 *Internal RAM organization of the 8051*

Status Word) determines which register bank is currently in use. These two bits can be modified at any time by a program to select any one of the bank.

Discussion question Which register bank do we have access to when 8051 is powered up and how it can be changed? How is the selected bank accessed?

Answer Register bank 0. It is the default register bank, since after power on, both bits RS0 and RS1 are initialized with value 0. It will select bank 0. The register bank can be changed by changing bank selection bits RS0 and RS1 bits (D3 and D4 bits) in the PSW. Selecting different bank is also referred as switching of register

Table 2.2 *Register bank address allocations*

Name	Addresses			
	RS1 RS0			
	00	01	10	11
	(Bank 0)	(Bank 1)	(Bank 2)	(Bank 3)
R0	00H	08H	10H	18H
R1	01H	09H	11H	19H
R2	02H	0AH	12H	1AH
R3	03H	0BH	13H	1BH
R4	04H	0CH	14H	1CH
R5	05H	0DH	15H	1DH
R6	06H	0EH	16H	1EH
R7	07H	0FH	17H	1FH

bank. The registers in the selected bank can be accessed either by name (R0–R7) or they may optionally be accessed by their actual addresses.

Example 2.1

Illustrate two different ways to select register bank 1.

Solution:

(i) Using instruction MOV PSW, #000**01**000B
or MOV PSW, #08H

(ii) Second, using bit level instructions

```
CLR PSW.4      // clear RS1 bit
SETB PSW.3     // set RS0 bit
```

The second approach is better because it does not disturb other bits of the PSW register.

RAM addresses 00H to 1FH, when not referred with names, can be used as general purpose RAM. Since only one bank can be accessed at any time, one can raise the question that "What is the advantage of having four register banks?"

When we switch the bank, complete new set of memory locations with names R0 to R7 are made available for use by instructions. So without saving the contents of current bank's R0 to R7, we can directly use a new set of R0 to R7. This will save time to preserve the registers contents when we call a subroutine, i.e. save context switching time.

Discussion question Why is it preferable to refer these RAM locations with their names (R0 to R7) over their addresses?

Answer First, it is easy to refer the locations with names. Secondly, instructions referring these locations with names will requires less bytes as well as less time for execution.

For example, (assume bank 0 is selected)

	Bytes required	Machine cycles required to execute
MOV R0, #10H	2	1
MOV 00, #10H	3	2

Both instructions are doing the same thing, i.e. it loads a number (immediate data) 10H to internal RAM location 00H. In the first instruction, we are referring location 00H with its name R0. It is of 2 bytes and only one machine-cycle execution time. While in second instruction, it is 3 bytes instruction and requires two machine-cycle execution time.

2. Bit Addressable Memory

The 8051 has a bit-addressable area of 16 bytes from byte addresses 20H to 2FH in internal RAM, forming a total of 128 (16 × 8) addressable bits. An addressable bit can be accessed by its bit addresses from 00H to 7FH. Besides these 128 bits, majority of SFRs are also bit addressable. The bit addressable SFRs with address of each bit is shown in Figure 6.1. The instruction that uses this address determines whether a byte or bit is being referenced without confusion. The bit addressable area with address of each bit is shown in Figure 2.3.

Example 2.2

What is the bit address of:

(i) 7th bit of byte address 20H?

(ii) 5th bit of byte address 2FH.

(iii) What are the bit addresses assigned to the byte address 28H?

Solution:

(i) 7th bit of byte address 20H has address 07H, (assuming that bit numbering starts from 0).

(ii) 5th bit of byte address 2FH is 7DH.

(iii) The bit addresses assigned to the byte address 28H is 40H to 47H as shown in Figure 2.4.

Discussion question What is the advantage of having bit-addressable locations?

Answer In many applications, we need to manipulate (to remember or record or change) binary events such as to turn on or off a device, to read status of an input switch. The bit-addressability feature suits perfectly for such applications.

So if such tasks can be handled by only single bit, so why use whole byte? This is the most desirable feature in the machine control applications. Bit addressability helps in developing more readable and efficient programs. A detailed discussion of bit-addressable memory and their use is given in Chapter 6.

3. General-Purpose RAM

Bytes from memory locations 30H to 7FH are used for general-purpose data storage. These 80 locations are widely used by programmers to store temporary data and intermediate results. The advantage of using this memory is that their access is faster compared to other off-chip RAM. This area of memory is also used as a system stack.

(a) The Stack The stack is a section of memory in the internal RAM that is used for temporary storage and retrieval of data (or addresses), while the execution of a program. It is the Last In First Out (LIFO) type memory. This section of memory is accessed by certain instructions or events (like interrupts).

The register used to access contents of the stack is called *stack pointer*. It is an 8-bit register. The power on default value of the SP register is 07H. The stack on the 8051 grows upwards in memory, therefore SP is incremented before data is stored as a result of special instructions (PUSH and CALL). This means that the location 08H is the first location being used for the stack. As data is retrieved from the stack (using POP and RET), the byte is read from the stack and then SP is decremented by one. So in conclusion, the address held in the SP register is the location where the last byte was stored by a stack operation.

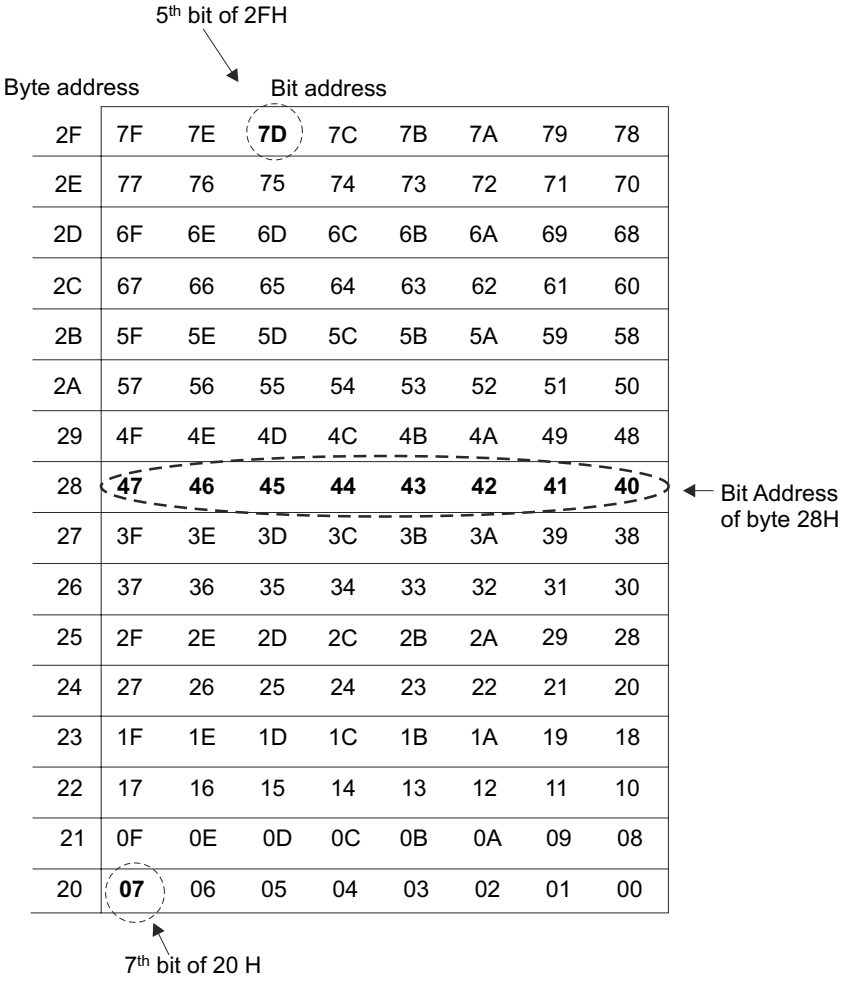


Fig. 2.4 *Bit-addressable memory*

The SP register can be initialized with any address within the internal RAM, i.e. 00H to 7FH. The stack is usually defined (and thus located) at higher addresses in RAM by loading new address into SP before performing any stack operation.

Since stack is growing automatically after each stack operation (data store), the programmer should take care that valuable data in internal RAM (register bank, bit addressable area and general-purpose RAM) is not overwritten and hence lost. Also make sure stack does not grow beyond predefined limits (7FH being highest address).

Discussion Question Explain how the stack is used by the PUSH and POP instructions.

Answer For a PUSH instruction, the stack pointer (SP) is first incremented by 1, and then the data is stored on the stack address where the SP is pointing. For a POP instruction, the data is first retrieved from the stack address where the SP is pointing and then the SP is decremented by 1.

THINK BOX 2.3

Why do you think the size of the stack pointer in the 8051 is only 8 bits?

The 8051 uses internal RAM for the stack (only 128 bytes from 00H to 7FH). 8 bits can address this space.

Example 2.3

Explain how the contents of Accumulator and B registers can be stored and retrieved from the stack.

Solution:

The above task will be accomplished by the following set of instructions.

```
MOV A, #10H          // A = 10H
MOV B, #20H          // B = 20H
MOV SP, #50H         // SP =50 H
PUSH 0E0H            // SP=51H, and (51H) = 10H, SP is incremented by 1 and contents of
                     // Accumulator (address E0H) is stored at memory location 51H
PUSH 0F0H            // SP = 52, (52H) = 20H, SP is again incremented by 1 and contents of
                     // B is stored at memory location 52H
POP 0F0H             // B = (52H)=20H, SP = 51H, data is retrieved in to B from address
                     // 52H and SP is decremented by 1
POP 0E0H             // A= (51H) = 10H, SP = 50H; data is retrieved in to A from address
                     // 51H and SP is again decremented by 1.
```

The reason we used addresses of registers A (E0) and B (F0) instead of using their names is that PUSH and POP instructions support only direct addressing mode which requires addresses instead of names. However, we can write instruction PUSH ACC instead of PUSH 0E0H, as it is supported by the assembler used in the text.

(b) Default Stack and Bank1 As explained earlier, the first memory location used for the stack is 08H (after power ON), which is also R0 of bank1 and stack grows upwards, which means register bank1 and stack are using same memory area. If we want to use register bank1 and 2, we should reallocate another section of RAM to the stack. For example, we can initialize SP with a value 50H. More details of stack and its operation are provided in Chapters 4 and 7.

2.3.3 Internal ROM

The 8051 contains 4Kbytes of internal ROM (on-chip). It occupies address range from 0000H to 0FFFH. Since it is used to store program instructions (code), it is also called *program memory* or *code memory*. Different members of 8051 family contain different amount and type of on-chip ROM. For example, 8051 contains 4Kbytes masked ROM, while 8752 contains 8Kbytes of internal EPROM. This memory is sufficient for small projects and applications. For larger programs, external memory may be connected or family members with higher amount of internal ROM may be used. These days, internal memory up to 64K bytes is available in to variants of the 8051.

Since PC is a 16-bit register, it can address up to 64Kbytes (from 0000H to FFFFH) of program memory. Program (code) addresses above 0FFFH (which is beyond on-chip ROM addresses) will be accessed automatically from external program memory.

We can also connect external ROM if 4K of internal memory is not sufficient for a particular application. Our program may reside partly into internal ROM and partly into external ROM or entire program can be stored into only external memory starting from 0000H. Memory map of ROM, i.e. organization of ROM for the 8051 system is shown in Figure 2.5.

$\overline{EA}$ (external access) pin (pin 31) on the 8051 decides one of the above configuration of ROM used by the 8051. If $\overline{EA}$ pin is connected to V_{CC} (+5 V), the 8051 will access first 4K bytes (0000H to 0FFFH) from internal ROM and any address above 0FFFH will be accessed from external ROM. If $\overline{EA}$ is connected to ground (0 V) then only external memory from 0000H to FFFFH will be accessed by the 8051.

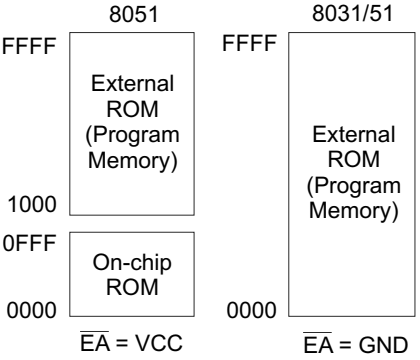


Fig. 2.5 *ROM organization of the 8051*

2.4 EXTERNAL MEMORY ORGANIZATION

There are two parallel 64 kilobytes address spaces; one for the ROM and other for the RAM, i.e. the 8051 can simultaneously address 64 Kbytes of RAM (data memory) as well as 64Kbytes of ROM (program memory) as shown in

Figure 2.6. The data space is accessed using external data movement instructions (MOVX *A, source* or MOVX *destination, A)* and code space is accessed using external code movement instructions (MOVC A, *source*). Both types of instructions uses the DPTR register to point to the actual memory byte.

Program memory is always accessed through PC to fetch the op-code of an instruction.

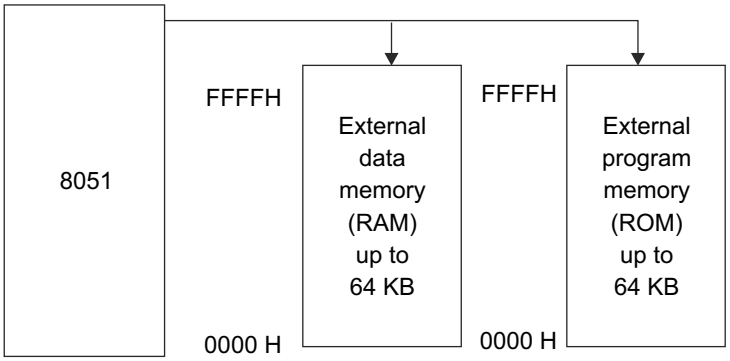


Fig. 2.6 *External program and data memory space for the 8051*

Discussion question How processor will decide which address space (RAM or ROM) to use for a data transfer?

Answer It depends on the type of instruction (opcode) being executed. For ROM access, code ROM movement instructions are used. For example,

MOVC A, @ A+DPTR, or
MOVC A, @A+PC

And, for RAM access, data movement instructions are used. For example,

MOVX A, @DPTR,
MOVX A, @Rp,
MOVX @DPTR, A
MOVX @Rp, A

Note that suffix 'C' is to indicate code memory access instruction and 'X' letter for external data memory access instruction.

When the code ROM movement instructions (MOVC) are executed, the 8051 microcontroller generates the $\overline{\text{PSEN}}$ (program storage enable) signal which can be used to activate and access the ROM chip, whereas the data RAM movement instructions (MOVX) will generate only $\overline{\text{RD}}$ and $\overline{\text{WR}}$ signals which can be used to access data to/from RAM chips. Interfacing of the external code memory and the data memory is described in detail in Chapter 21.

Discussion question How is it possible to store data at address 0000H, even though program instruction is residing at the same address?

Answer The 8051 is designed using Harvard architecture, where the program memory and the data memory are present in different physical address spaces. Therefore, there are two different physical locations having the address 0000H. To access each type of a memory, different instructions are used. The PC is always used to access the program memory while it is never used to access the data memory. MOVC instructions are used to access the program memory while MOVX instructions are used to access the data memory.

POINTS TO REMEMBER

- Programming model is a representation of the resources that can be accessed by a programmer.
- The programming model of the 8051 contains an 8- and 16-bit registers and 8-bit memory locations.
- The 8051 on-chip memory is organized into three general categories, Special Function Registers, internal RAM and internal ROM.
- Special function registers are used to program and control various on-chip hardware peripherals and features of the 8051. They are assigned addresses 80H to FFH in internal RAM area.
- All locations between 80H to FFH are not physically present in the 8051, only 21 locations are occupied by SFRs.
- The 8051 microcontroller has a total of 128 bytes of RAM. These bytes are assigned addresses 00H to 7FH and divided into register banks, bit addressable and general-purpose RAM.
- The 8051 has a bit-addressable area of 16 bytes from byte addresses 20H to 2FH in the internal RAM, forming a total of 128 (16x8) addressable bits.
- An addressable bit can be accessed by its bit address from 00H to 7FH.

- Bit addressability is used to manipulate the binary events, this feature helps in developing more readable and efficient programs.
- The stack is a section of memory locations in the internal RAM that is used for temporary storage and retrieval of the information. It is a Last In First Out (LIFO) type memory.
- The 8051 contains 4Kbytes of internal ROM in the address range 0000H to 0FFFH.
- The 8051 can simultaneously address 64 Kbytes of RAM (data memory) as well as 64Kbytes of ROM (program memory).

OBJECTIVE QUESTIONS

1 The 8051 stack can be defined in,
   (a) internal code memory (b) internal data memory
   (c) external data memory (d) all of the above

2 In the 8051, data can be stored in,
   (a) code memory (b) internal RAM (c) stack memory (d) all of the above

3. The program counter,
   (a) stores the address of the instruction that is currently being executed
   (b) stores the next instruction to be executed
   (c) stores the address of the next instruction to be executed
   (d) stores the instruction that is being currently executed

4. In the 8051, SP is ___ wide register, and may be initialized to point anywhere in the _____.
   (a) 8 byte, on-chip 128 byte RAM (b) 8 bit, on-chip 256 byte RAM
   (c) 16 bit, on-chip 256 byte RAM (d) 8 bit, on-chip 128 byte RAM

5. What is the address range of SFRs in the 8051?
   (a) 00H-77H (b) 40H-80H (c) 80H-7FH (d) 80H-FFH

6. The 8051 can address,
   (a) 64Kbytes of program memory and 64Kbytes of external data memory
   (b) 64Kbytes of program memory or 64Kbytes of external data memory
   (c) 4Kbytes of program memory and 64Kbytes of external data memory
   (d) 4Kbytes of program memory or 64Kbytes of external data memory

7. After power on, the first byte read by the 8051 is,
   (a) opcode (b) operand
   (c) opcode or operand based on an instruction (d) 00h

8. The best place to store rarely used variables in the 8051 is,
   (a) register banks (b) SFRs (c) bit addressable RAM (d) 30H-7FH

9. The 8051 has,
   (a) 128 bits of data memory as bit addressable (b) 16 bytes of data memory as bit addressable
   (c) some of the SFRs as bit addressable (d) all of the above

10. In the 8051, data can be stored in,
   (a) external code memory (b) internal ram
   (c) external data memory (d) all of the above

11. In a microcontroller, the Program counter (PC) always deals with,
   (a) program memory (b) data memory (c) stack memory (d) all of the above

12. Which of the followings is not a bit addressable SFR?
   (a) PSW (b) ACC (c) SP (d) none

13. The special function registers can be referred to by their hex addresses or by their register names.
   (a) True (b) False

14. The internal RAM of the 8051 is,
    (a) 32 bytes (b) 64 bytes (c) 128 bytes (d) 256 bytes
15. When the 8051 is reset and the $\overline{EA}$ line is HIGH, the PC points to the first program instruction in the,
    (a) internal code memory (b) external code memory
    (c) internal data memory (d) external data memory
16. The total external data memory that can be interfaced to the 8051 is,
    (a) 32K (b) 64K (c) 128K (d) 256K
17. RS0 and RS1 are,
    (a) not a part of the PSW as these are not the flags (b) part of the register banks
    (c) PSW.4 and PSW.5, respectively, for selecting the register bank in the PSW (d) PSW.3 and PSW.4, respectively, for selecting the register bank in the PSW
18. Which of the following SFR is bit addressable?
    (a) TCON (b) SP (c) SBUF (d) DPL
19. The 8051 has,
    (a) 4 banks of 4 registers (b) 2 banks of 16 registers
    (c) 8 banks of 4 registers (d) 4 banks of 8 registers
20. Which of the following SFRs of the 8051 is not bit addressable?
    (a) A (b) PSW (c) SBUF (d) P0
21. The 8051 has _____ on-chip program memory.
    (a) 1 Kbytes (b) 2 Kbytes (c) 4 Kbytes (d) 64 Kbytes
22. The stack operations in the 8051 are,
    (a) last in first out (b) first in first out (c) last in last out (d) none of the above
23. The maximum size of the stack in the 8051 can be,
    (a) 128 bytes (b) 256 bytes (c) 64 bytes (d) 64K bytes
24. Which of the following pin of the 8051 is used to select external code memory?
    (a) $\overline{PSEN}$ (b) $\overline{EA}$ (c) ALE (d) $\overline{RD}$
25. LSB of byte address 21H has a bit address____.
    (a) 00H (b) 08H (c) 21H (d) none of the above

Answers to Objective Questions

1. (b)	2. (d)	3. (c)	4. (d)	5. (d)	6. (a)	7. (a)	8. (d)	9. (d)	10. (d)
11. (a)	12. (c)	13. (a)	14. (c)	15. (a)	16. (b)	17. (d)	18. (a)	19. (d)	20. (c)
21. (c)	22. (a)	23. (a)	24. (b)	25. (b)					

REVIEW QUESTIONS WITH ANSWERS

1. In how many distinct categories the address space of the 8051 is divided?

A. Address space of the 8051 is divided into four distinct categories: internal data memory, external data memory, internal program memory, and external program memory.

2. In how many distinct categories is the on-chip memory of the 8051 divided?

A. Special function registers, internal data memory (RAM) and internal program memory (ROM)

3. What is the address range of special function registers and internal RAM?

A. Internal RAM : 00H to 7FH

Special function registers: 80H to FFH (not all locations within this range physically exists on a chip but only 21 locations are occupied by the SFRs)

4. Which byte addresses in the internal RAM are also bit addressable? What address range is assigned to bits of bit addressable internal RAM?

A. Byte addresses 20H to 2FH are also bit addressable. Address range 00H (LSB of 20H) to 7FH (MSB of 2FH) is assigned to bits of the bit addressable RAM.

5. Are all SFRs bit addressable?

A. No. SP, DPL, DPH, TL0, TL1, TH0, TH1, PCON, TMOD, SBUF are not bit addressable.

6. Which register bank is selected by default after the reset? How is a register bank changed?

A. Register bank 0 is selected by default. It can be changed by programming RS1 and RS0 bits in the PSW register as follows.

RS1	RS0	
0	0	Bank 0
0	1	Bank 1
1	0	Bank 2
1	1	Bank 3

7. Access to the on-chip memory is faster compared to off-chip memory. True/false.

A. True.

8. List flags available in the 8051.

A. Carry (CY), Auxiliary Carry (AC), Overflow (O), Parity (P) are math flags, user flags (F0, GF0 and GF1) are general purpose user flags. GF0 and GF1 are available in PCON register.

9. What is the default value of the stack pointer register?

A. SP = 07H.

10. What is the use of SBUF register?

A. SBUF (serial buffer) register is used to hold data to be transmitted or received for the serial port.

EXERCISE

1. Why is the accumulator named so?
2. Where does the PC always point to?
3. Which memory is referred as data memory?
4. Which memory is referred as code memory or program memory?
5. Explain how SP is modified using PUSH and POP instructions.
6. How many ports are available in the 8051? Are all ports bit addressable?
7. Define the term 'programming model'. Draw the programming model of the 8051.Why should we study it?
8. List the data and control registers of the on-chip peripherals of the 8051.
9. Discuss how register banks are useful in context switching.
10. Discuss the stack operation in detail.
11. Can we interface 64Kbytes of both ROM and RAM at the same time with the 8051?
12. Discuss how $\overline{EA}$ is used to select between internal and external program memory.
13. Explain the significance of PSW. What are the applications of Carry and Overflow flags?
14. The special function registers are assigned next 128 locations after the general-purpose data storage and stack. True/False.
15. Can we use program memory to store data? If yes, how?

3

Program Development Process and Tools

Objectives

- Discuss and compare various programming languages
- Introduce the assembly language structure
- Develop a simple assembly language program
- Discuss qualitatively the steps involved in the program execution
- D-iscuss the software and the hardware development tools
- Discuss the significance of the documentation tools
- Show the significance and use of Integrated Development Environment in the system development
- Introduce the common assembler directives
- Explain in detail the program development cycle
- Show how to load a program in to a microcontroller
- Describe the Intel hex file format

Key Terms

- Assembler
- Assembly Language
- Comment
- Compiler
- Debugger
- Documentation
- Download/Burning
- Emulator
- Flowcharts
- Hex File
- High-level Language
- IDE
- Instruction
- Label
- Linker
- Logic Analyzer
- Machine Language
- Mnemonic
- Object File
- Op-codes
- Operands
- Pseudo-codes
- Simulator
- Source Code

The first step in *learning any new language* is to start with most common and simple words and grammar or rules of their usage. Applying the same logic to the learning process of an assembly language, we start with the common terminology and structure of an assembly language and then the complete program development process beginning from program statement to loading a program in to microcontroller memory is discussed. This process is described with the help of a simple program example.

3.1 PROGRAMMING LANGUAGE

Programming means developing a sequence of commands (or instructions) that can be used by computer system to perform a predefined task. It also involves troubleshooting or debugging of instruction sequence to ensure that the desired operation is performed. There are three types (levels) of programming languages based on how closely the language statements are related and resemble to actual operations performed by the microcontroller/ processor. The three types of languages are Machine language, Assembly language and High-level Language.

3.1.1 Machine Language

The microcontroller/processor is a digital device which understands and operates on binary digits 0 and 1, stored as voltage levels in the circuits. The microcontroller needs commands in form of bit patterns (of 0s and 1s) to perform any operation. These binary bit patterns are called *instructions* or *machine codes* which are recognized and processed by a microcontroller to accomplish a task. The most common operations are storing and retrieving binary data, arithmetic and logical operations. Each microcontroller has its own instructions, meanings, and syntax. The designer of the microcontroller decides these bit patterns or instructions based on the number and type of operations it is required to perform. The entire set of the binary instructions is called *instruction set* and it describes and represents machine language of particular microcontroller. Every instruction is represented by a unique bit pattern to distinguish it from another instruction. Since the microcontroller understands only bit patterns, every operation has to be specified only in a machine language.

Single instruction may not be sufficient to accomplish a task; therefore, we need group of instructions written in a particular sequence. These instructions written in a sequence to perform a task are collectively known as a *program* or a *microcontroller program*.

The number of bits in a binary pattern that a microcontroller recognizes and processes at a time is called word, and microcontrollers are classified according to their word length, it ranges from 4 bits for small systems to 64 bits for high-performance computers. Microcontrollers of MCS 51 family are known as 8-bit microcontrollers as they can operate on 8 bits at a time.

3.1.2 Assembly Language

The microcontroller understands only machine language, therefore, we must specify program only in a machine language, however it is not suitable for human use because it is difficult to write and understand instructions using patterns of 0s and 1s. Also, to deal with patterns of 0s and 1s is quite tedious, inconvenient and error inductive for most humans.

For convenience, one can represent the binary instructions in Hexadecimal codes, which is compact form of binary numbers. This will be improvement over dealing directly in binary numbers because instructions can be represented in a compact form and because of that chance of making mistake reduces.

Even if the instructions are written using hexadecimal codes, it is still difficult and time consuming to develop and understand programs. Therefore, manufacturer of microcontroller/ microprocessor assigns unique English-like word (code) to all binary instructions which are referred as *mnemonics*. The mnemonic for an instruction is a word (usually three or four letters) that indicates the operation to be performed by that instruction. The entire group of instructions when represented as mnemonics is called *assembly language of the microcontroller*. It is easy and convenient to develop and understand assembly-language programs. Every microcontroller/processor has different and specific assembly-language, hence, assembly-language program written for one microcontroller will not work on other. Assembly language is also called low-level language because it directly deals with internal hardware of the microcontroller and its statements directly resembles to the operations performed by microcontroller/processor. A computer program known as *assembler* is used to convert assembly-language programs into machine language.

3.1.3 High-Level Language

High-level languages are named so because programmer need not know and worry about the internal details and architecture of the microcontroller/processor to develop the programs, moreover, their statements represent directly the

program logic (algorithm) rather than actual operations performed by a microcontroller, as a result they are also machine independent. For example, BASIC, Pascal, C, C++ and Java are high-level languages. A computer program known as *compiler* is used to convert high-language program into machine-language program.

3.1.4 Comparison between Programming Languages

A brief comparison between the programming languages is given in Table 3.1.

Table 3.1 *Comparison between programming languages*

Sr. No.	Machine Language	Assembly Language	High-level Language
1	The operations are specified by binary bit patterns.	The operations are specified by mnemonics.	Simple statements specify the operations.
2	The program development is difficult, inconvenient and error prone.	The program development is simpler and easier than machine language.	The program development is easiest.
3	It is not user-friendly.	It is less user-friendly.	It is more user-friendly.
4	High code density (Less memory required to store a program)		Low code density (More memory required)
5	Faster execution		Slower execution
6	Both languages are microcontroller dependent and require knowledge of internal details of microcontroller to develop a program.		No or very little knowledge of microcontroller is required

The comparison made in the above table is proved using examples in topic 12.12. 'Performance comparison between assembly and C programs'

As an illustration, a sample program to add two numbers is shown in Figure 3.1 for all three languages for a comparison.

Machine Language (8051)	Assembly Language (8051)	High-level Language (C)

01111000 00001010 (78 0A H)
01110100 00010100 (74 14 H)
00101000 (28 H)
	MOV R0,#10
	MOV A,#20
	ADD A, R0
		char a =10, b =20;
		a = a + b;

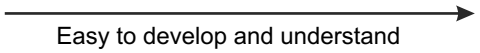


Fig. 3.1 *Sample program in different programming languages*

The comparison of these programming languages is shown in pictorial form in Figure 3.2.

The programming of the 8051 in a high-level language (C) and their advantages are discussed in detail in Chapter 12 (The 8051 programming in C)

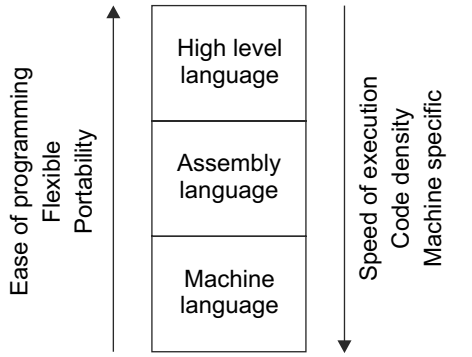


Fig. 3.2 *Comparison of programming languages*

3.1.5 Why Assembly Language?

We need to use and study assembly-language programming because of the following advantages offered by them:

1. The assembly language programs provide direct and accurate control of the microcontroller/processor resources (memory, ports, peripherals and hardware present in a system). The assembly-language programs (for device drivers for peripherals) are efficient in terms of memory requirements and speed of execution.
2. The assembly language codes usually require less memory and they execute faster.
3. It directly allows exploiting the specific features of microcontroller/processor.
4. There is less number of restrictions or rules as compared to the high-level language.

THINK BOX 3.1

Under what circumstances should we write programs in assembly language?
We must write programs in an assembly language when we are writing timing critical code or when the code memory space available to write a program is limited.

3.2 ASSEMBLY LANGUAGE STRUCTURE

An assembly language program statement consists of three fields: Labels, instructions and comments. The arrangement of these three fields in a program is shown below.

[Label:]	Instructions	[//Comments]

Brackets show that the field is optional and may not be present in all statements.

3.2.1 Label

The label assigns a name to a memory location or program statement. It must be followed by the colon symbol ":". While using label as a part of an instruction, the colon symbol should not be used. They are also used to define the symbols. The name of any register or an instruction of a microcontroller cannot be used as a label. For example, DPTR, DPL, MOV or ADD etc. These names are reserved and already defined for use of an assembler program.

One important point that should be considered while choosing the names of labels is that they should be meaningful as they will reduce the need for documentation. Poor names for labels may result in confusion, ambiguity, misunderstanding, mistakes and finally frustration. For example, in a program if two labels used are LAST and FINAL then it may create confusion that which one actually means last! Also, do not use names which have more than one meaning, like label TEMP may be confused between temporary result and temperature.

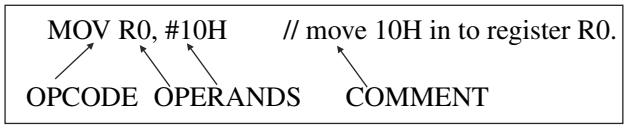
3.2.2 Instructions

Each instruction has two parts; one is operation to be performed called as *op*eration *code* (op-code). For example, ADD, MOV, INC, etc. The second part is data on which operation is to be performed called as *operands*. There are two types of operands:

1. **Source operand(s):** The operands that are input for an operation. The operation may involve one or two operands.
2. **Destination operand(s):** These operands will store the result of an operation.

The operands can be specified in many ways. It may be 8-bit data, 16-bit data, an internal register, a memory location or 8-bit (or 16-bit) address. Some microcontrollers have a few instructions in which operands are specified implicitly, i.e. we need not specify operand in an instruction, it is automatically understood by an instruction.

Above terms can be best explained by the following examples.



In the above instruction, MOV is the op-code (mnemonic), R0 register is destination operand and number #10H is an 8-bit source operand. This instruction will move (load) data 10H in R0 register.

Consider other example,

ADD A, 50H

In the above instruction, ADD is the op-code (mnemonic), an 8-bit memory address 50H is one of the source operand and the register A is source as well as destination operand.

When an operation to be performed is specified in an assembly language (English-like statement), it is referred as mnemonic, while the same thing specified in machine language (binary) is referred as op-code. But, in microcontroller literature these terms (mnemonic and op-code) are used interchangeably.

Instruction Size

The number of bytes required to represent an instruction in a machine language is called instruction size. There are one-, two-, or three-byte instructions in the 8051.

1. **One-byte instructions:** Only one byte is required to represent the op-code as well as operands of an instruction. For example,

Instruction	Machine code	Operation
MOV A, R0	E8	moves contents of R0 in to A

2. **Two-byte instructions:** These instructions occupy two bytes in the machine code.

Instruction	Machine code	Operation
MOV A, #10H	74 10	move number 10H in to A

3. **Three-byte instruction:** These instructions occupy three bytes in the machine code.

Instruction	Machine code	Operation
MOV 20H, #30H	75 20 30	move number 30H in address 20H

3.2.3 Comments

Comments are used to document the program (software) properly and are used to describe the operation performed by an instruction or group of instructions. They make programs more readable and easy to understand. Comments start with a semicolon symbol ";" (or "//" based on assembler) to indicate they are not the operands. The assembler ignores the comments. Even though comments are optional, they should be included in a program as they will assist us in debugging, modifying or adding new features in a program.

Good comments should tell us why an instruction (or group of instructions) is written rather than what an individual instruction is doing in a microcontroller. While writing the comments it is generally assumed that the reader/programmer is familiar with syntax of the language. Following example shows bad way of writing comments because they do not provide any additional information.

```
MOV R0, #10H        // move 10H in to register R0
ADD A, #01H         // add 01H to A
```

Instead of this, comments may be written to explain why we write these instructions, like

```
MOV R0, #10H        // set R0 as loop counter and initialize it with count 10H
ADD A, #01H         // increment A to point to next data element in an array
```

When a subroutine or function is defined, the comments should explain how to use the subroutine, what the input and output parameters are and how the results are returned. Write units of parameters if any, also, specify minimum, maximum or typical values of the parameters. Similar explanation is equally applicable when variables are defined.

Discussion Question What are the fields in an assembly-language instruction (or command)?

Answer The fields in an assembly language programs are,
(a) the labels, (b) the mnemonics (contains op-codes and operands), and (c) the comments.

THINK BOX 3.2

We know both 74H and MOV mean "move" for the 8051. Which of these two is the mnemonic and which is op-code? Why?
MOV is the mnemonic and 74H is the op-code. 74H is the op-code because it represents actual binary instruction. MOV is the mnemonic because it is English-word-like abbreviation.

3.3 ASSEMBLY-LANGUAGE-PROGRAM EXAMPLE

A simple program to illustrate the structure of an assembly-language program is given below.

```
[Label:]   Instructions          [Comments]
           ORG 0000H             // start program at memory location 0000H
           MOV R2, #10H          // load immediate data 10H in to register R2
           MOV R3, #15H          // load immediate data 15H in to register R3
           MOV A, #20H           // load immediate data 20H in to Accumulator
           ADD A, R2             // A = A + R2
           MOV B, A              // save result (A) in to register B
           ADD A, R3             // A = A + R3
           MOV 50H, A            // save result (A) in to memory location 50H
HERE:      SJMP HERE             // wait here indefinitely
           END                   // end of the program
```

Sample program 3.1

3.4 PROGRAM EXECUTION PROCESS

The program execution process for the above program is explained briefly as follows. To understand the steps of the program execution, it is better to have picture of arrangement of machine codes of the program inside the ROM. It is assumed that the program (machine codes) is already loaded into the microcontroller program memory. The simplified arrangement of machine codes along with their assembly language statements are given below for better understanding.

```
ROM Address     Machine codes     Label      Assembly statements
0000            7A 10                        MOV R2, #10H
0002            7B 15                        MOV R3, #15H
0004            74 20                        MOV A, #20H
0006            2A                           ADD A, R2
0007            F5 F0                        MOV B, A
0009            2B                           ADD A, R3
000A            F5 50                        MOV 50H, A
000C            80 FE             HERE :     SJMP HERE
```

The above program will be executed in the following steps.

1. When the 8051 is powered up (or reset), the program counter (PC) has value 0000H in it. Therefore, it starts to fetch first op-code from address 0000H. It will read 7A from the address 0000H. Upon decoding the op-code 7A, it understands that it is op-code for moving immediate value into R2. It expects immediate value to be stored at the next address, therefore it increments PC (to 0001H) and reads a byte (10H) from that address and places 10H into register R2. The first instruction is executed. Now program counter is incremented to 0002H to point to the next instruction.
2. Similar to the first instruction, the op-code 7B is fetched and immediate data byte 15H is placed in to register R3. After completion of this instruction, the PC is incremented to 0004H.
3. Opcode 74 is fetched and in the same way the immediate value 20H is loaded in to A and PC is incremented to point to next instruction, i.e. PC=0006.
4. The 8051 fetches op-code 2A and performs the operation A=A+ R2 and PC is incremented to 0007. Note that PC is incremented only by 1 because it is a one-byte instruction.
5. Same way, all instructions are executed one by one until execution of the last instruction, which in our example keeps microcontroller busy in executing same instruction repeatedly for indefinite time (or until power is removed).

The important point to be understood is that PC is incremented automatically to point to the next instruction in a program.

THINK BOX 3.3

Though microprocessors/controllers understand only binary numbers, why are the hexadecimal numbers used in their literature (during hardware and software development)?

Binary numbers are usually difficult to handle (read/write by humans) because of their length. Hexadecimal numbers are used as shorthand for binary numbers for convenience.

3.5 SOFTWARE AND HARDWARE DEVELOPMENT TOOLS

During the development process of a microcontroller-based system, a host system (PC or laptop) is used because they support the use of the development tools.

The basic development tools used for microcontroller-based system are

1. Design and documentation tools
2. Software development tools
3. Hardware development tools

3.5.1 Design and Documentation Tools

The first step in developing a program is to define clearly the problem to be solved. This will lead to definition of inputs

and outputs of a problem. Without this step, we may write a program that works nicely but may not do the things that we want it to do! The second step is to make a solution plan. Here, what and how operations (tasks and computations) to be performed are determined. Then finally sequence of all operations is decided. Each operation may be further divided into smaller units. This approach of developing a program is called modular design approach. Representation of above operations in a sequence to solve the problem is called *algorithm of the program*. We should first clearly define the algorithm before we start writing actual program instructions. There are two common ways of representing the algorithms: Flowcharts and Pseudo-codes.

1. Flowcharts

A flowchart is a graphical representation of the activities (processes/tasks) to be performed and sequence of steps to be followed to solve a problem. A flowchart facilitates and assists the program development process.

Flowcharts use the graphic symbols to represent different types of program operations. The operation desired is written in the graphic symbol. Figure 3.3 shows some of the common flowchart symbols.

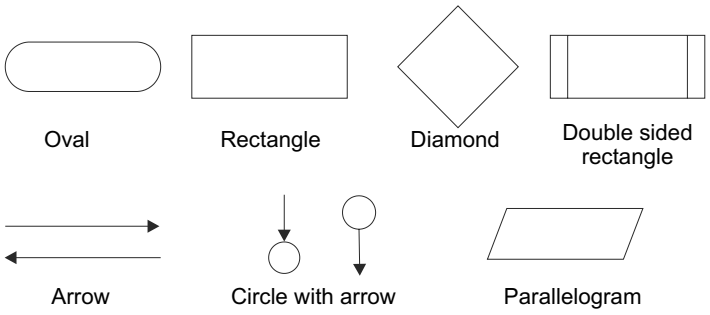


Fig. 3.3 *Common flowchart symbols*

- **Oval:** Used to indicate beginning or end of a program
- **Rectangle:** Represents a process or task
- **Diamond:** Decision making point where program flow may be altered based on the result of a previous process.
- **Double-sided Rectangle:** Shows predefined process or subroutine call, written independently
- **Arrow:** Direction of program execution
- **Circle with Arrow:** Shows continuation or an exit to different column
- **Parallelogram:** Represents input output operation

2. Pseudo-codes

When the algorithm for a problem is to be modified or updated to add new features, it is difficult and cumbersome to maintain (to modify the drawing) the flowcharts as per new changes, i.e. maintenance of flowcharts is difficult, because it requires more space. Moreover, for complex computations, more details are to be presented in each box which is not convenient with flowcharts. Another approach to represent the algorithms is Pseudo-codes which makes maintenance relatively easy and consumes less space. In this, brief description of each action is written in English like statements in a sequence to represent the algorithm.

Any program action could be represented by three basic types of operations.

- **Sequence**: It is series of actions performed one by one.
- **Decision**: It is choosing between two or more alternative operations depending upon some conditions.
- **Repetition**: Repeating series of actions until some condition is fulfilled (or not fulfilled).

Based upon above three types of operations, there are three standard structures to represent all the operations in a program: *sequence, if-else* and *while*.

if, else-if ladder, do-while, repeat- until and *switch* are the other structures derived from these basic structures which are used to make program development more clear and easy. The output of one structure may be connected to any other structure and any structure may be used within another. The structures with their flowcharts and equivalent pseudocode are shown in Figure 3.4.

3.5.2 Software Development Tools

The common software-development tools used for microcontroller based systems are discussed briefly in the following section.

1. Editor

The source files are created using editors. They allow writing assembly language or any high-level language programs using keyboard of the PC. It allows entry, addition, deletion and copying of the program statements.

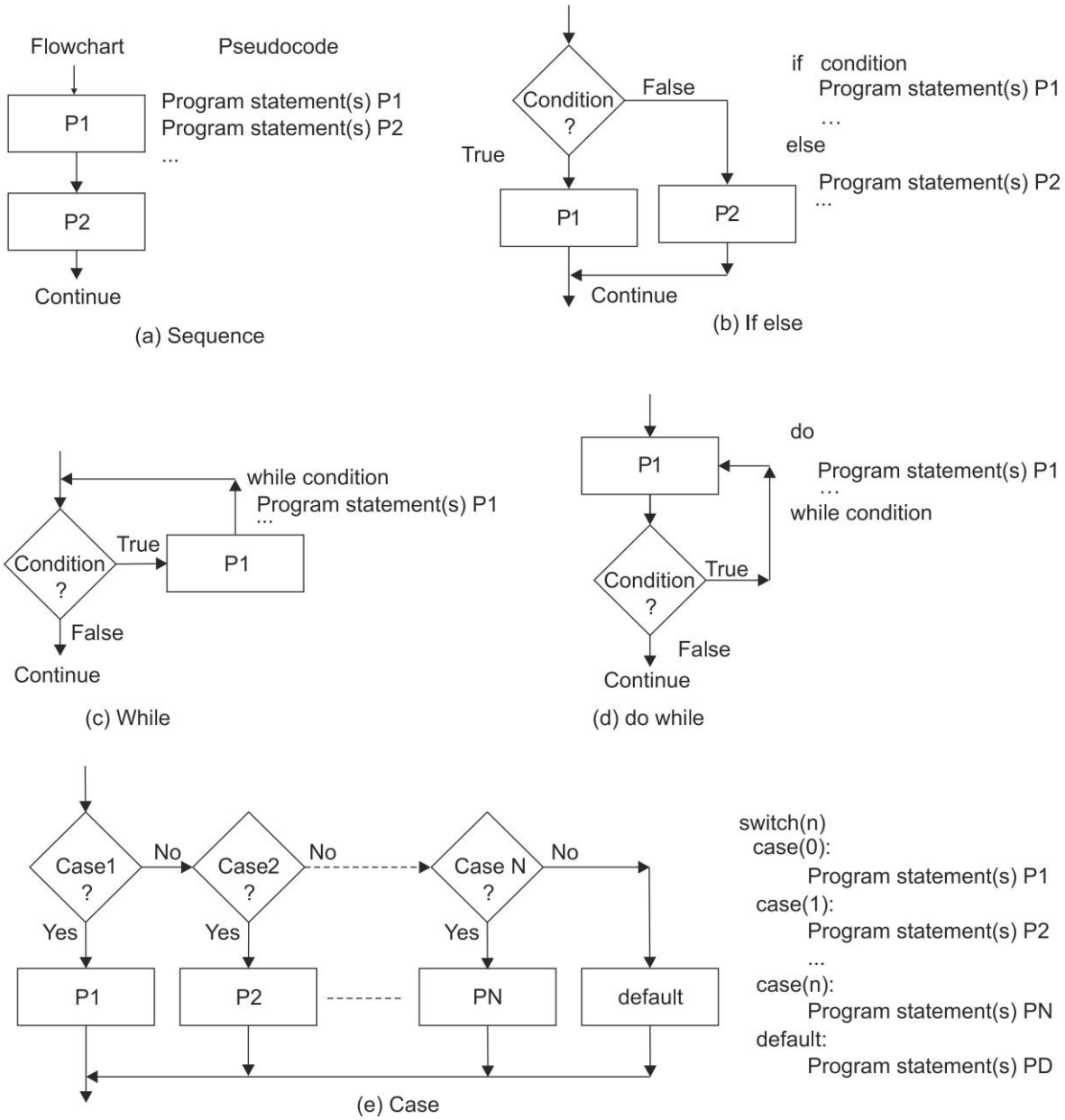


Fig. 3.4 *Common program structures*

2. Assembler

An assembler is a program which converts the assembly language program into corresponding machine language (binary codes). An assembler generates two files, namely the object file and the list file. The object file is assigned an extension '.OBJ' and the list file is assigned extension '.LST.' The object file contains the machine (binary) codes of each instruction along with information about the addresses of them. List file contains the assembly language statements, the machine codes for each instruction and the address (and offset) of each instruction as well as the messages to indicate typing or syntax errors if any. The object file may have unresolved symbols/addresses if the entire program is located in more than one files (as in the case of using library functions), these unresolved addresses are resolved during the linking of all program object files.

Syntax and Logical Errors The assembler is capable of only finding the syntax errors. To check if our program is working correct, we have to run (execute) and test the program, i.e. assemblers cannot find logical errors (commonly referred as bugs). The errors (syntax errors) indicated by the assembler should be corrected using the editor. This edit assemble loop must be repeated until all the errors are removed.

3. Cross Assembler

An assembler that runs on a computer system, which has different microcontroller/processor than the microcontroller/ processor, for which it assembles the source code. For example, an assembler that runs on an Intel x86 machine and assembles source code of the 8051. Usually, most assemblers are *Cross Assemblers*.

4. Compiler

It is a program which converts high-level language programs to machine language. It also indicates the syntax errors in the program, if any. It generates object file corresponding to the target microcontroller.

5. Linker

The linker links more than one object files like object files of the source code files and library object files, if any.

6. Simulator

Simulator is software that functions (pretends) like hardware without having actual hardware. The 8051 microcontroller simulator gives the user an 8051 environment on the host system (PC). It shows all internal registers, entire memory and peripherals on the monitor. It supports breakpoint and single stepping facilities that help the user to debug their program. Since the operation of the microcontroller is simulated, all of the information about the microcontroller operations is directly available to the programmer and hence allows finding the logical errors. It also performs simulation of different peripherals that are used with the microcontroller/ microprocessor in an application. It is a cheaper alternative of the costly 8051 microcontroller kit during application development.

Advantages of using Simulator during Application Development

- Allows simulation of peripherals and I/O devices
- Saves the development time
- Allows checking of software before the hardware is available to the user

THINK BOX 3.4

What are the limitations of simulators?

Since simulation is not a real time execution, actual execution timing and response time of a system may not be exactly simulated because speed of the host system where simulator is running and actual speed of system may not be mapped exactly. Also, some hardware dependent problems may not be simulated.

7. Debugger

Debuggers are similar to simulators except that they execute programs in the real microcontroller/processors. This is done by using a ROM based monitor program in the target board, communicating with a program in the PC through a serial (COM or USB) port. Debugger allows the user to load program into the system memory, execute the program by single stepping and detect the logical errors in the program. It allows the modify register contents, memory locations and rerun the program for different set of data. It also allows using breakpoints. It has all facilities of simulator.

The key feature is that debuggers allow finding logical errors while the program is running on actual hardware. Therefore, it is not a completely software based tool.

THINK BOX 3.5

What is the key difference between debugger and simulator?

The debugger performs all operations in a real microcontroller while the simulator pretends as if it is a microcontroller and hence performs all operations only in software.

3.5.3 Hardware Development Tools

It is a difficult, time-consuming and hectic task to develop a microprocessor based system. Three hardware development tools are used commonly for the development of a microprocessor/microcontroller based system. These tools are used to test the system while program is running on actual hardware. They are in circuit emulator, logic analyzer and target system.

1. Emulator or In Circuit Emulator (ICE)

ICE is used to debug a target system without using the actual microcontroller/processor. It allows the program to be tested on the actual hardware system, with facilities like breakpoints or single stepping. ICEs contain a unit that is connected between the host computer and the board (system) to be tested. A large header and cable assembly connects this unit at the place where the actual microcontroller is to be mounted. The unit emulates the microcontroller in such a way that from the board (actual hardware) point of view, it has an actual microcontroller/processor fitted and from the host computer's point of view, the system is fully controlled allowing a developer to debug and test the program.

2. Logic Analyzer

It is a hardware diagnostic tool used to record and display status of multiple signals like address/data bus, memory read/write, I/O read/write and I/O ports. It can record multiple bus transactions to help the real time debugging. It can simultaneously handle many signals (16, 32, 48, 64 or even more). They allow the user to take samples of signals at any instant. It is more powerful than oscilloscopes because scopes can handle only two or four signals.

3. Target System

It is a hardware board used during development phase and final application is made from it.

3.6 INTEGRATED DEVELOPMENT ENVIRONMENTS (IDE)

IDE is development software which integrates all tools necessary to develop, test and debug the programs for an application into single module. It includes editor, assembler, compiler, linker, simulator, tracer, emulator and logic analyzer. It also supports code burning process. It simulates the hardware components like peripherals, I/O devices and emulators. It supports different families of microcontrollers.

The µvision 4.0 is a popular and user-friendly IDE from Keil Software Inc. designed for Cortex-Mx, ARM7, ARM9, C166, XE166, XC2000 and C51(8051 and all its variants) microcontrollers, which combines project management, make facilities, source code editing, program debugging in one environment. It allows program development in assembly language, high-level language and combination of both in a same program.

Refer Appendix B: Using Keil µvision 4.0 IDE

Appendix B is a complete tutorial based on the µvision 4.0 IDE. It explains how to use µvision 4.0 to develop and test the 8051 programs in an assembly as well as C language. To support and ease the understanding, a step wise explanation along with screenshots of µvision 4.0 IDE windows is given for the sample programs.

Other popular IDEs are Code Composer Studio from Texas Instruments, Code Warrior from Freescale Inc. and MPLAB from Microchip. They are used only for system development based on hardware developed by respective manufacturers.

3.7 ASSEMBLER DIRECTIVES

The assembler directives are the statements that directs the assembler what to do during assembling. They reserve memory space for data, define constants, and tell assembler where to assemble program in a memory. They are also referred as pseudo-instruction statements as they are effective only during the assembly of the program but they do not generate any machine code.

The following are the widely used assembler directives for 8051.

Note: These assembler directives are assembler dependent, thus, their format may vary across different assemblers.

1. Originate-ORG

The ORG directive allows us to place the code and data anywhere in the program address space. The number after the

ORG can be in hex or decimal. If the number is decimal, the assembler will convert it to hex. Its format is 'ORG Address' For example, ORG 0000H will place the instructions (or data) from address 0000H onwards (some assembler use this directive as '.ORG Address')

2. Define Byte-DB

This directive defines the byte-type variable. When DB is used to define data, the numbers can be in decimal, binary, hex, or ASCII formats. For the binary numbers, B is as a suffix. Similarly, H is used after hexadecimal numbers. Irrespective of the type of the byte, the assembler will convert the byte to hex. To indicate ASCII numbers, the characters are placed in quotation marks ('character').The DB is also used to allocate memory in byte sized chunks.

Example 3.1

Specify the addresses of each of the bytes defined by following declarations.

```
ORG 0100H
DB 10
DB 10H
DB 'ABCD'
DB 00011111B
```

Solution:

```
DB 10          // define byte 10 (decimal) and store at address 0100H
DB 10H         // define byte 10 (hex) and store at address 0101H
DB 'ABCD'      // define string 'ABCD' and store at addresses 0102H to 0105H
DB 00011111B   // define byte 00011111(binary) = 1FH and store at address 0106H
```

3. Define Word-DW

This directive defines 16-bit variable. (The meaning of "Word" is microcontroller/processor dependent, for example, for 32-bit controller, word means 32 bits)

4. Equate-EQU

Equates label to the number.

EQU TEMP, #20 will assign name TEMP to data #20.

5. END

End directive tells the assembler to stop assembling.

There are still many other directives like DD, DBIT, DS, PROC, ENDP, LABEL, USING, EXRERN, IF, ELSE, ELSEIF, ENDIF, NAME, PUBLIC, SEGMENT, BSEG, CSEG, BIT, CODE, DATA, EVEN etc. used with different assemblers. Programmer should refer documents or help for the assembler in use.

Note: In Keil documentation, above directives are referred as 'control statements' and the term 'directive' has the other meaning, i.e. it refers to commands that control the source file assembly.

Example 3.2

Write the necessary assembler directives for the following,

(i) Place values 10H, 20H and 30H in three consecutive memory locations starting from code memory address 1000H
(ii) Place the character string " Microcontroller" at the code memory address 2000H onwards
(iii) Access the ports using names P0, P1, P2, P3 instead of their addresses
(iv) Define the constant 0FFH with the name TEMPERATURE

Solution:

(i)
```
ORG 1000H
DB 10H, 20H, 30H
```
(ii)
```
ORG 2000H
DB "Microcontroller"
```

(iii) P0 EQU 80H
P1 EQU 90H
P2 EQU 0A0H
P3 EQU 0B0H
(iv) TEMPERATURE EQU 0FFH

THINK BOX 3.6

How is "Microcontroller" encoded as a null terminated ASCII string?
"4D6963726F636F6E74726F6C6C657200"

3.8 PROGRAM DEVELOPMENT PROCESS

Developing an executable (ready to run on microcontroller) program file from problem statement is the three (or four) step process.

1. Create Source Code

Analyze the problem and type a program in an editor as per syntax of the assembly language. The original program developed by a programmer is called *source code*.

This step involves program logic development and to create source file(s) using an editor. Most common editors are MS DOS EDIT and NOTEPAD as they are available in all Windows operating systems. The source code may be developed in an assembly language or high-level language like C. The assembly language source code has '.s 'or '.src' or '.asm' or '.a' extension depending upon the assembler.

2. Assemble all Source Code Files

Assemble the program using an assembler to generate machine language code (object code). Assemble each source file individually if the program consists of several source code modules. The assembler will produce object file (.obj) and list file (.lst). If the assembler indicates errors then use the editor again for correcting them and again assemble the source file. Repeat this step until removal of all errors. The successful completion of this step will generate '.obj' and '.lst' files. The sample of list (.lst) and object (.obj) file for Sample program 3.1 is given below. The source code written into high-level language is converted into an object file using a compiler.

List file for sample program 3.1

```
000001 0000              ORG 0000H       // start program at memory location 0000H
000002 0000 7A10         MOV R2, #10H    // load immediate data 10H into register R2
000003 0002 7B15         MOV R3, #15H    // load immediate data 15H into register R3
000004 0004 7420         MOV A, #20H     // load immediate data 20H into Accumulator
000005 0006 2A           ADD A, R2       // A = A + R2
000006 0007 F5F0         MOV B, A        // save result (A) into register B
000007 0009 2B           ADD A, R3       // A = A + R3
000008 000A F550         MOV 50H, A      // save result (A) into memory location 50H
000009 000C 80FE   HERE: SJMP HERE       // wait here indefinitely
000010 000E              END             // end of program
```

Object file for sample program 3.1

```
:0E0000007A107B1574202AF5F02BF55080FE47
:00000001FF
```

3. Link all the Object Code Files

If the program consists of several modules, then we use the linker to join them into large object module. The linker program takes object files(s) and produces an executable file with extension '.hex' that can be loaded into microcontroller ROM. (since there is only one source file in given example, object and hex file will be same)

4. Test the Program for Correctness

The program may be checked for correctness using simulator or may be directly burned into ROM of the microcontroller and can be tested with help of In-circuit emulator (ICE). These are powerful tools used for testing and debugging (to find bug or logical error) the program. The µvision 4.0 IDE assemble and link the program modules in a single step. The process of developing executable file is illustrated in Figure 3.5. Note that final executable file may be given any name as per programmer wish.

The complete program-development cycle is illustrated in Figure 3.6.

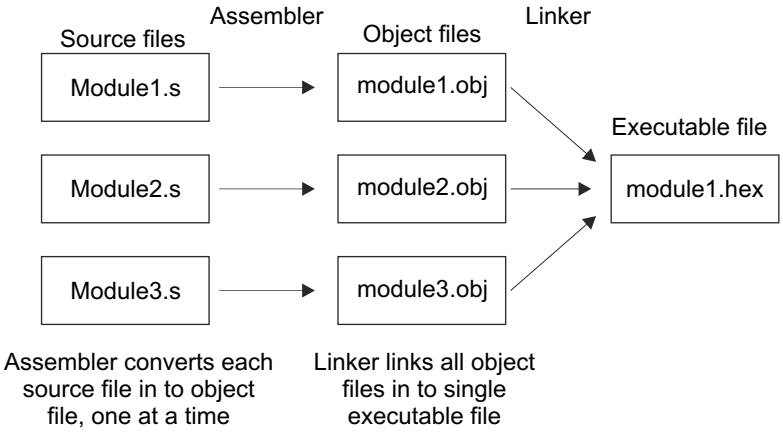


Fig. 3.5 *Simplified process of generating executable file*

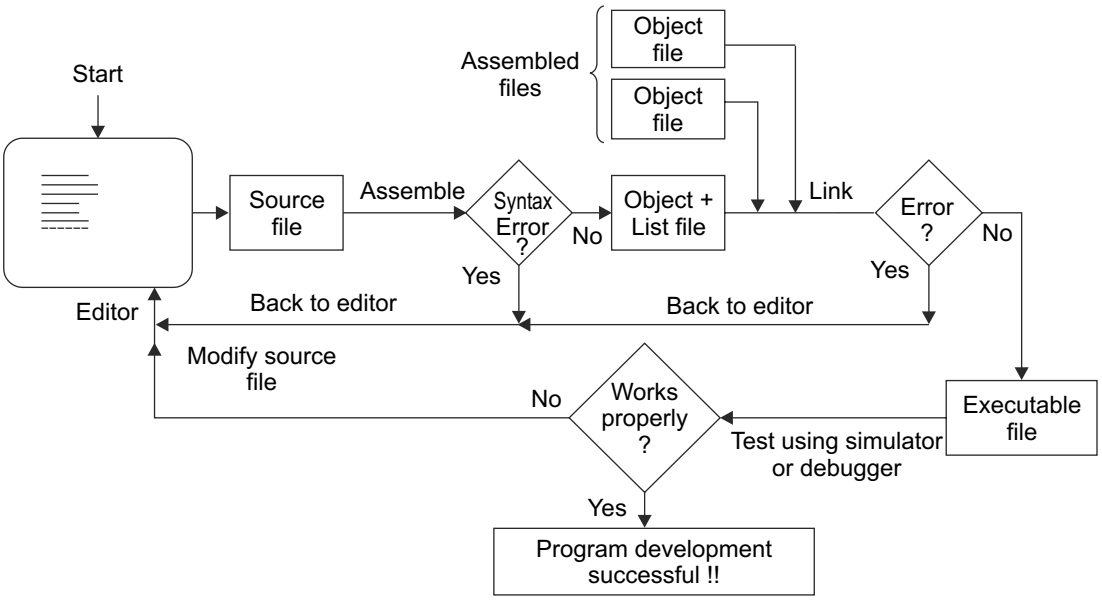


Fig. 3.6 *Program-development cycle*

As shown in the flowchart of the program development cycle, assembler will convert source file into an object file, but if the source file has syntax error or invalid instruction or data, then it will report an error(s) after assembling. Usually, error messages will provide the information about type of error and an experienced programmer will quickly identify the cause of error by reading the error message. The list file will show the place (instruction or line) where the error exist. Once error is identified, correct it using editor and repeat the process until there is no error.

We can also use subroutines or part of a program (that is already tested) in our program by adding corresponding object files to our object file to generate final executable file. This is done by the *linker*. Once an executable file is ready, we should check it to see that whether it works as per our expectations or not. The executable files (usually referred as Hex files in microcontroller literatures) are tested using either simulator or debugger. If it works correctly in a simulator, it can be directly burned into ROM of a microcontroller. But, if program does not give expected output, there is some logical error in the program. (Note that logical errors are not caught up by the assembler.)

The logical errors (usually referred as bugs) can be identified and located by single stepping a program. *Single stepping* is a feature provided by simulator (these days this is also supported by microcontrollers) where one instruction is executed at a time and it waits for a command to go for next instruction. After execution of each instruction, the operands, registers

and memory locations accessed by instruction are checked to verify whether they are updated as per programmers expectations or not, if these locations are not updated correctly after some instruction, there is a problem in that instruction, either a wrong instruction is used, or wrong operands or wrong memory is being accessed by an instruction. Try to see the difference between expected output and actual output, and choose correct instruction and operation, and make corresponding changes in the source file. Now repeat the process of assembling, linking and testing the program on simulator until it gives desired outcome. For a new programmer, the task of finding error may be difficult, but as he practices with small programs frequently, the task will become easy.

To simplify the program development and the testing process use the following guidelines:

(a) Divide the problem into small tasks.
(b) Arrange these tasks in proper sequence to get the desired result.
(This is also referred to as algorithm.)
(c) Represent these sequence of tasks in a flowchart or pseudo-code form.
(d) Write suitable assembly (or high-level) language instructions corresponding to each small task with proper comments.
(e) Combine all instructions to form a program.
(f) Assemble the program and remove syntax errors (if any) and generate object file.
(g) Link other object files (if any) and generate the hex file.
(h) Single step the hex file and find bugs (if any) using the simulator.
(i) Modify the source file as per the bugs.
(j) Repeat above process until desired result is obtained.

Along with these guidelines, reuse the already tested subroutines (if applicable) to speed up the process and always write descriptive comments along with instructions which will help later to modify or upgrade the program. The comments will also help others to understand and use your program.

THINK BOX 3.7

Suppose that we examine the contents of code memory address 100H and found that it contains 74H (0111 0100B). Without any additional information can we say whether it is an op-code or operand? What would be your answer if you had observed same value at code memory address 0000H?

No. it may be op-code or operand.

The value stored at address 0000H must be an op-code because the 8051 expects the op-code at that address. After reset (or power on reset) the PC=0000H, therefore it always expects op-code at reset vector address.

3.9 | LOADING PROGRAM INTO MICROCONTROLLER

Application programs in .HEX format can be loaded in to PROM (or EPROM, EEPROM, Flash memory) of microcontroller using device programmers or In System Programming (ISP).

1. Parallel Programming

Loading program using device programmers is usually referred as parallel programming. The microcontroller chip is inserted into a socket of device programmer and is programmed by transfer of the bytes for each address using software running in the host machine. The device programmers are usually connected to host machine through parallel port, serial port or USB.

2. Serial Programming

The serial programming is more popularly referred as ISP (In System Programming). In this approach, the microcontroller can be programmed using serial data transfer protocol without being removed from the application hardware; the on-chip boot loader provides the interface for the ISP. These chips are programmed using external systems usually PC, which contains a software to handle programming activities. The serial programming is further divided into three types, ISP using UART, SPI and JTAG.

Based on type of on-chip program memory, different methods are used for loading the program into program memory of a microcontroller. These methods are summarized in Table 3.2 .

Table 3.2 *On-chip program memory-loading methods*

Type of on-chip Program Memory	Programming Method	Example
ROM	Programmed during manufacturing; User can not program	805x (or 80C5x), 835x (or 83C5x) chips.
EPROM	**Parallel programming** Plastic package (One Time Programmable): Can be programmed only once. Ceramic windowed package: can be erased using ultraviolet light and reprogrammed	875x or 87C5x chips.
FLASH/ EEPROM	**Parallel programming**	89C5x/89Cx051 chips
	ISP using UART: The programming is done through TXD and RXD pins of the microcontroller. These pins are connected with COM port of the PC. The PC will have appropriate application program to handle programming.	89C51RDx, DS89C4x0 chips. (Remote programming over modem link is possible)
	ISP using Serial Peripheral Interface (SPI): The programming involves signals from SPI interface (MOSI, MISO, CLK) and some other control signals.	AT89Sxx, AT89LPxx, P89LPC9xx chips
	ISP using JTAG: The JTAG Interface allows application programming and debugging of on-chip and off-chip Flash microcontrollers and other devices. These operations are usually handled by application program (IDE: Integrated development environment) running on the PC.	C8051Fxxx (all devices from Silicon Laboratories Inc.), uPSD3xxx (all devices from ST microelectronics)

Refer datasheet of a particular microcontroller to know how to program the program memory.

The process of loading a program into a microcontroller is more commonly known as *downloading a program* or *burning a program into the chip*. The simplified program development and loading process is shown in Figure 3.7.

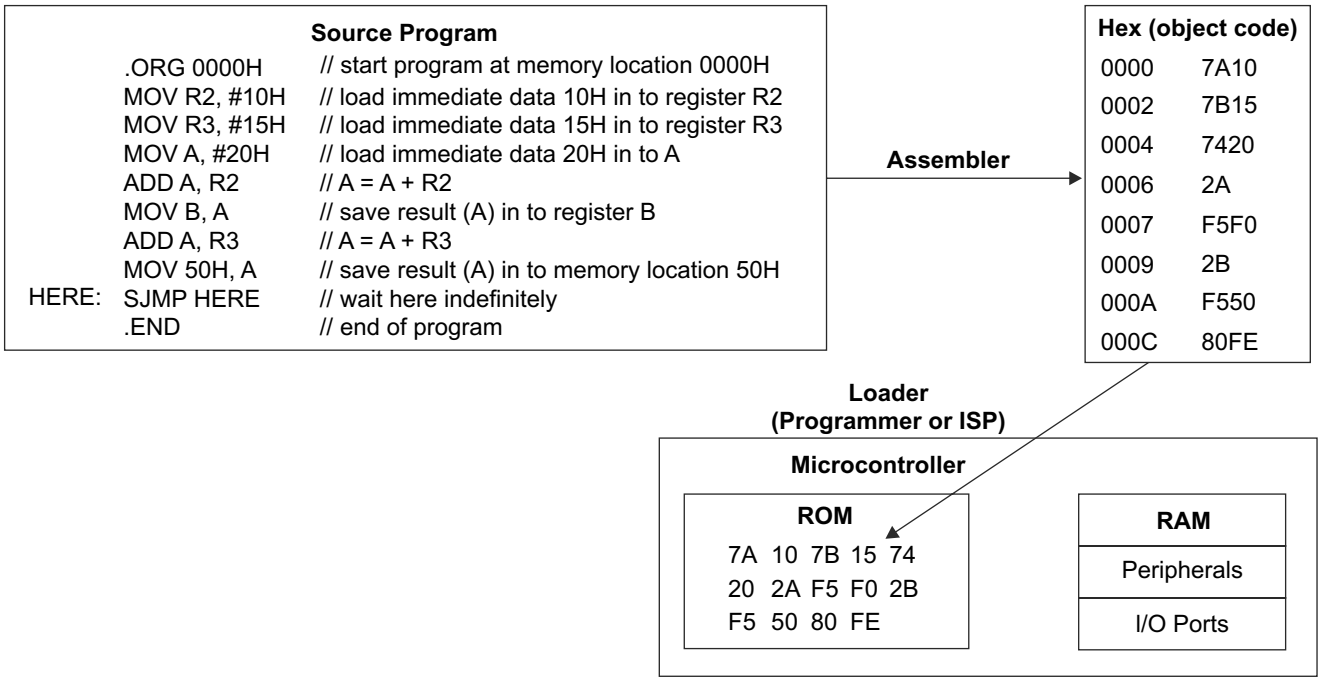


Fig. 3.7 *Simplified program development and loading process*

3.10 | THE INTEL HEX FILE FORMAT

The Intel Hex file format is one of the most popular and commonly used formats for executable files ('.hex' or '.obj' when there is a single source file) to be loaded into microcontroller program memory (ROM). An Intel Hex file is an ASCII file, where each line has the format shown in Figure 3.8.

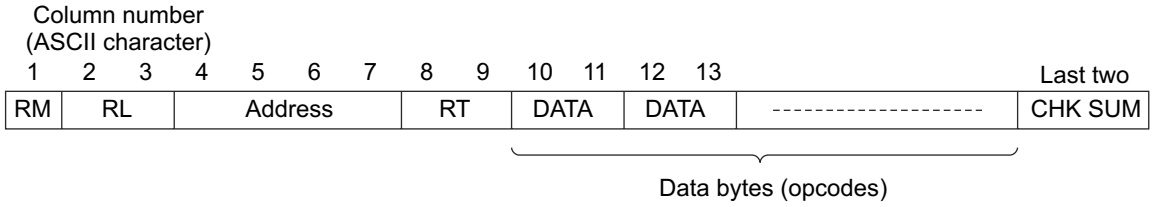


Figure 3.8 *Intel hex file line format*

The fields are explained in Table 3.3.

Table 3.3 *Fields of line of Intel hex files*

Column Number	Field Name	Description
1	Record Marker	It is always a colon ':' to indicate the Intel Hex file.
2-3	Record Length (data length)	It contains the number of data bytes (op-codes) in a line represented as a 2-digit hexadecimal number. This number does not include the check sum bytes or the first 9 characters of the line.
4-7	Address	It contains the starting address into ROM where the data should be loaded. This is a value from 0 to 65535 represented as a 4-digit hexadecimal value.
8-9	Record Type (data type)	It specifies the type of the record for this line. The possible values are: 00 = data, 01 = end of file, 02= extended address.
10-11 and onwards	Data Bytes	These two columns and following columns (in pair of two) are the actual data that will be loaded into the ROM. Note that a line contains up to 16 data bytes.
Last 2 Columns (characters)	Check Sum	The last two character of the line are a check sum for the line. The check sum found by taking the two's complement of the sum of all the data bytes in a line, excluding the check sum bytes and the colon

To understand the format, consider '.hex' file ('.obj') for the Sample program 3.1.

:0E0000007A107B1574202AF5F02BF55080FE47

:00000001FF

Consider the first line,
Record marker: ':' colon in first column
Record Length: 0E (14 bytes of op-codes)
Address: 0000 (the 14 bytes will be stored starting at address 0000)
Record Type: 00(normal data)
Data: 7A,10,7B,15,74,20,2A,F5,F0,2B,F5,50,80,FE
Check sum: 47

Add all the data bytes above (except colon and checksum byte itself)
0E+ 00+00+7A+10+7B+15+74+20+2A+F5+F0+2B+F5+50+80+FE = 6B9
Consider only lower two digits (B9) and neglect upper digit (6)
The two's complement of B9 is 47, which is the checksum for the line.
The second line is *end of file record* (verify it yourself as it is similar to the first line).

POINTS TO REMEMBER

- There are three levels of programming, depending upon how closely the language statements are mapped to actual operations of microcontroller; they are machine language, assembly language and high-level language.
- The number of bits in a binary pattern that the microcontroller recognizes and processes at a time, is called word length. Bus width, internal bus width, register width and bits are the different terms used interchangeably for the word length.

- The assembly-language programs provide direct and precise control of the hardware present in a system. They are more efficient in terms of memory requirements and execution speed.
- Each instruction has two parts, one is operation to be performed called as operation code (op-code) and the second part is data on which operation is to be performed called as operands.
- IDE is development software which integrates all tools necessary to develop, test and debug the programs for an application into single module. It includes editor, assembler, compiler, linker, simulator, tracer, emulator and logic analyzer. It also supports code burning process. It simulates the hardware components like peripherals, I/O devices and emulators. It supports different families of microcontrollers.
- The assembler directives are the statements that directs the assembler what to do during assembling. ORG, DB, EQU, DW, DEFINE and END are most common assembler directives.
- The process of loading a program into a microcontroller is more commonly known as downloading a program or burning a program into a chip.
- The Intel Hex file format is one of the most popular and commonly used formats for executable files to be loaded into microcontroller program memory.

OBJECTIVE QUESTIONS

1. An assembler directive is,
   (a) same as an instruction (b) used to define variables
   (c) used to start a program (d) used to give commands to an assembler
2. A label is used to name a single line of code.
   (a) True (b) False
3. The common extensions of an assembly language program are,
   (a) .s (b) .asm (c) .obj (d) .hex
4. The key features of assembly language are,
   (a) high code density (b) faster execution (c) portability (d) ease of development
5. The key features of high level language are,
   (a) high code density (b) faster execution (c) portability (d) ease of development
6. IDE typically include,
   (a) assembler (b) compiler (c) simulator (d) all of the above
7. DB 'ABCD' declaration will occupy __ bytes.
   (a) 1 (b) 2 (c) 3 (d) 4
8. DB directive is used to declare data bytes of type,
   (a) decimal (b) hexadecimal (c) binary (d) all of the above
9. An assembler converts,
   (a) mnemonics into machine language (b) translates high level language into machine language
   (c) assembly language into machine language (d) none of these
10. In machine language _____.
    (a) programs are written using mnemonics (b) programs are written using hex/binary
    (c) programs are written using ASCII code (d) none of these
11. Programs can be loaded in a microcontroller by,
    (a) parallel programmers (b) ISP using UART
    (c) ISP using SPI (d) All of the above
12. Assembler produces ___ files as output.
    (a) .asm (b) .obj (c) .lst (d) .hex

Answers to Objective Questions

1. (d)
2. (a)
3. (a), (b)
4. (a), (b)
5. (c), (d)
6. (d)
7. (d)
8. (d)
9. (a), (c)
10. (b)
11. (d)
12. (b), (c)

REVIEW QUESTIONS WITH ANSWERS

1. List three levels of programming languages.

A. Machine language, assembly language and high-level language.

2. What are the most common operations performed by a microcontroller/processor?

A. The most common operations are storing and retrieving binary data, arithmetic and logical operations.

3. Define term "word length". What is word length of the 8051?

A. The number of bits in binary pattern that microcontroller recognizes and processes at a time is called word length. Word length of the 8051 is 8 bits.

4. What is an assembler?

A. Assembler is a computer program used to convert assembly-language program into machine language.

5. Which language requires minimum memory for a program?

A. Machine language.

6. Which language is more user-oriented?

A. High level language.

7. What is the size of the 8051 instructions?

A. Depending upon the instruction it will be one, two or three bytes.

8. What are the common ways of representing algorithms?

A. Flowcharts and pseudo-codes.

9. What is an assembler directive? List most common assembler directives.

A. The assembler directives are the statements that directs the assembler what to do during assembling process. DB, DW, ORG, DEFINE, EQU and END are common directives.

10. Define the term "source code". What are common extensions used for assembly language programs?

A. The original program developed by a programmer is called source code. The assembly language source code has '.s 'or '.src' or '.asm' or '.a' extension depending upon assembler.

11. Which files are produced by an assembler?

A. Object file (.obj) and list file (.lst)

12. What are the most common editors available in windows based systems?

A. Edit and Notepad.

13. What is meant by target system?

A. It is hardware board used during development phase and final application is made from it.

14. What are the fields of assembly-language program?

A. Labels, instructions (or mnemonics) and comments

15. List the key characteristics of the high-level language.

A. Ease of programming, flexibility and portability.

EXERCISE

1. Define the program and programming language.
2. Why machine language is not suitable for human use?
3. Compare assembly and machine languages.
4. Justify “machine language has highest code density”.
5. What are the advantages of developing a program in an assembly language?
6. Define the terms op-code and operand.
7. Explain in detail the program execution process in the 8051 using appropriate example.
8. What is a simulator? Discuss advantages of using simulator in the program development process.
9. Differentiate between simulator and debugger.
10. What is IDE? Give examples of some popular IDEs used in embedded system development.
11. What are the functions of a linker?
12. Discuss program development cycle in detail with the help of the flowchart.
13. Discuss different methods of loading a program in to microcontroller chip memory.
14. Differentiate between .obj and .hex files.
15. Which of the following files the DOS EDIT program will produce?
    (a) prog.asm (b) prog.obj (c) prog.exe (d) prog.lst
16. Which of the following files does an 8051 assembler will produce?
    (a) prog.asm (b) prog.obj (c) prog.exe (d) prog.lst
17. The EDITOR produces an ASCII file. True/False.
18. How many bytes are reserved by each of the following declarations?
    (a) DATA1: DB “1234” (b) DATA2: DB “ABC1234”
19. Discuss Intel hex file format in detail.
20. Discuss the importance of software and hardware development tools in the development process of a microcontroller-based products.

4

Addressing Modes and Data Movement Instructions

Objectives

- Introduce the data movement instructions and their significance
- List and classify the addressing modes of the 8051
- Compare the addressing modes along with their uses
- Classify the 8051 addressing space with respect to instruction set
- Discuss notations used for the data of different number systems
- Discuss the operand modifiers # and @
- List the instructions for external data and code memory access
- Describe PUSH, POP and data exchange instructions
- Develop the programs of data movement within any of the address space of the 8051

Key Terms

- Address Space
- Addressing Modes
- Data Memory
- Data Movement
- Direct Addressing Mode
- External Memory Access
- Immediate Addressing Mode
- Indexed Addressing Mode
- Indirect Addressing Mode
- Operand Modifiers :# and @
- Program Addressing Modes
- Program Memory
- Register Addressing
- Register Addressing Mode
- Stack

Introduction to the Instruction Set

Every microcontroller has its own instruction set, the designer of the microcontroller decides these instructions based on the number and type of operations it is required to perform. The instructions are classified according to type of operation performed by them. All instructions can be divided into four categories.

1. Data movement instructions
2. Arithmetic instructions
3. Logical instructions
4. Program flow control instructions

All types of instructions are discussed in detail in subsequent chapters. To start with concepts of assembly language programming, the data movement instructions are discussed in this chapter.

4.1 WHY DATA MOVEMENT INSTRUCTIONS FIRST?

The microcontroller (or any computer system) typically spends maximum time in data movement operations. Therefore, maximum instructions are provided for the data movement operations than for any other type of operation, thus variety of data movement instructions of a microcontroller determines flexibility and ease with which efficient programs can be developed. Moreover, it is easy to understand programming model of any microcontroller with data movement instructions. The 8051 have 30 (27 for 8 bits +1 for 16 bits + 2 for 1 bit) instructions only for data movement out of total 111 instructions. It comes out to be 27%!!!

4.2 ADDRESSING MODES

The way by which source or destination (usually source) operands are specified in an instruction is called *addressing mode*. The instruction may contain one, two or no operand. The data is accessed from source address, processed in the ALU and stored at destination address. The ways by which these data (or address of data) are specified are called addressing modes.

There are essentially four addressing modes used by all microcontrollers (or computer systems), all other modes commonly found in microcontroller/processor literature are types of these four basic addressing modes. They are,

1. Immediate addressing
2. Register addressing
3. Direct addressing
4. Indirect addressing

Yet, there is another way of classifying addressing modes based on whether data is accessed or new instruction is accessed (i.e. instruction execution flow is changed) as a result of instruction execution. They are data addressing and program addressing modes. Data movement, arithmetic and logical instructions use the data addressing modes and program flow control instruction uses the program addressing modes.

The 8051 supports all addressing modes as discussed above, they are summarized in Table 4.1.

Table 4.1 *Addressing modes of the 8051*

Data addressing modes	Program addressing modes
1. Immediate addressing	1. Short (relative) addressing
2. Register addressing	2. Absolute addressing
3. Direct addressing	3. Long (direct) addressing
4. Indirect addressing • Register indirect addressing • Indexed addressing	

Before we start discussion of the addressing modes and type of the instructions, we should know types of memories from where data are accessed by different instructions. The memory of the 8051 may be divided in five types of address spaces. The operands are located in any of the address spaces. The five address spaces are,

1. Bank registers (R0-R7)

2. Accumulator
3. Internal data RAM (00-7F & SFRs)
4. External data RAM (data memory)
5. Program ROM (internal + external ROM)

Note that the Accumulator is considered as a special case even though it is a SFR. In addition with these address spaces, one more place from where data can be obtained is from instruction itself, i.e. in case of immediate addressing mode, data is an integral part of the instruction.

With this basic understanding of the address spaces, let us discuss each addressing mode in detail.

Acronyms used in the Instructions

Acronyms used by Intel in the 8051 instructions are used throughout the book.

data : 8-bit data
data16 : 16-bit data
direct : address in internal RAM or SFRs
Rn : R0 to R7
Ri : R0 or R1

4.2.1 Immediate Addressing Mode

The data (constant) is specified as a part of instruction in a program memory. The data is available immediately as a part of instruction itself, therefore immediate addressing is very fast. However, since the data is fixed, at runtime it is not flexible. The instructions using an immediate operand have an 8-bit or 16-bit number following the op-code. For example,

```
MOV A, #data        // load 8 bit immediate data into Accumulator
MOV A, #55H         // A= 55H

MOV Rn, #data       // load 8 bit immediate data into Rn
MOV R3, #0FFH       // R3= FFH

MOV DPTR, #data16   // load 16-bit number into DPTR
MOV DPTR, #2000H    // DPTR=2000H
```

Note, in instruction 'MOV R3, #0FFH', 0 is used between # and FF to indicate that F is a number and not a letter. This is a requirement for assembler, not a microcontroller. The operation of instruction MOV R0, #10H is shown in Figure 4.1.

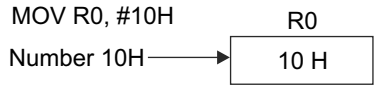


Fig. 4.1 *Immediate addressing mode*

The symbol for the immediate data is pound sign (#). The omission of # sign will result in a valid instruction with other addressing mode but with change in meaning of the instruction. During the execution of the immediate data movement instruction, the program counter is automatically incremented to point to data byte or (bytes for 16-bit number) immediately following the op-code byte; the data byte is copied to the destination. Note that the immediate data byte can not be a destination in an instruction.

- MOV (move) actually means copy data from source to destination.
- The data movement instruction does not affect the flags.

Example 4.1

Write instructions to initialize registers R0, R1 and R2 with values 10H, 20H and 30H respectively.

Solution:

The required operation can be performed by loading immediate values into registers. (Use of immediate addressing mode)

```
MOV R0, #10H        // initialize R0 with immediate value 10H
MOV R1, #20H        // initialize R1 with immediate value 20H
MOV R2, #30H        // initialize R2 with immediate value 30H
```

If bank 0 is selected, same operations can be done by following instructions:

```
MOV 00H, #10H      // initialize memory location 00 (R0) with value 10H
MOV 01H, #20H      // initialize memory location 01 (R1) with value 20H
MOV 02H, #30H      // initialize memory location 02 (R2) with value 30H
```

1. *Notations for Numbers of Different Number Systems*

Microcontroller understands only binary language. Hexadecimal numbers are compact representation of binary numbers so that it becomes easy to handle them. In above examples, the hexadecimal numbers are represented with suffix 'H'. If we want to work with decimal (or octal, ASCII) numbers directly, it is not allowed because they are not understood by the microcontrollers. However, assemblers provide the facility to work with all types of numbers (decimal, octal, ASCII, negative) numbers. The assembler will finally convert all numbers in to hexadecimal before generating machine codes. The format for numbers of different number systems is summarized in Table 4.2 with examples.

Table 4.2 *Notations of numbers of different number systems*

Number type	Suffix	Example	Instruction	Assembled instruction
Binary	B, b	11111010b	MOV A, #10001010b	MOV A, #8AH
		11101011B	MOV A,#11101011B	MOV A, #0EBH
Decimal	D, d	15D	MOV A, #15D	MOV A, #0FH
		25d	MOV A, #25d	MOV A, #19H
		115	MOV A, #115	MOV A, #73H
Octal	O, o,	20O,20o	MOV A, #20O	MOV A, #10H
	Q, q	75Q,75q	MOV A, #75Q	MOV A, #3DH
Hexadecimal	H, h,	25H	MOV A, #25H	MOV A, #25H
	0x (prefix)	FFh	MOV A, #0FFh	MOV A, #0FFH
		0x80	MOV A, #0x80	MOV A, #80H
ASCII	No suffix	A-Z, a-z, 0-9	MOV A, # 'C'	MOV A, #43H
Negative	-	-12, -50	MOV A, #-12	MOV A, #0F4H*

* F4 is 2's complement of decimal 12

> ***Note:*** Numbers without any suffix are decimal numbers

2. *Use of Expressions*

Mathematical or logical expressions may be used in an instruction in place of any numerical value. For example,

MOV A, #10H+20H is assembled as MOV A, #30H and
MOV 10+20, R0 is same as MOV 30, R0 or MOV 1EH, R0

Refer assembler user's guide for more details of use and precedence of expressions.

Example 4.2

How are the following instructions assembled?

(i) MOV A, #20 **(ii) MOV R0, #-1**
(iii) MOV R1, #25H **(iv) MOV R3, #10o**
(v) MOV R1, #15+20 **(vi) MOV R3, #01010101B**
(vii) MOV DPTR, #0x1000 **(viii) MOV DPTR, #1000**

Solution:

(i) MOV A, #0x14 (ii) MOV R0, #0xFF
(iii) MOV R1, #0x25 (iv) MOV R3,#0x08
(v) MOV R1, #0x23 (vi) MOV R3, #0x55
(vii) MOV DPTR, #0x1000 (viii) MOV DPTR, #0x3E8

4.2.2 Register Addressing Mode

In register addressing mode, the operands are specified by register names. Register A and R0 to R7 may be named as a part of the instruction mnemonic. The advantage of register addressing mode is that it occupies only one byte memory,

and is fast because only on-chip registers are accessed, i.e. instruction takes only one machine cycle for execution. For example,

```
MOV A, Rn    // copy contents of register Rn to Accumulator
MOV A, R2    // If R2=10H → A=10H

MOV Rn, A    // copy contents of Accumulator to register Rn
MOV R1, A    // If A=20H → R1=20H
```

The operation of instruction MOV A, R0 is shown in Figure 4.2.

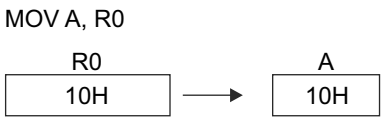


Fig. 4.2 *Register addressing mode*

4.2.3 Direct Addressing Mode

The data is accessed directly from the memory address specified as one of the operand, i.e. one of the operand is an 8-bit address for internal RAM location. Internal RAM includes 128 bytes of RAM from (00H–7FH) and any special function register. It is more flexible compared to immediate and register addressing because the value to be accessed from address may be variable. These are 2-byte instructions (3 bytes when source and destination are both direct addresses). The address refers to either byte location or a specific bit in a bit addressable byte. For example,

```
MOV A, direct        // copy data from (contents of) address direct in to A
MOV A, 10H           // If address 10H contains data 50H i.e (10H) = 50H → A=50H

MOV direct, A        // copy data from A to address direct
MOV 10H, A           // If A=44H → (10H) = 44H

MOV Rn, direct       // copy data from address direct into register Rn
MOV R5, 80H          // If (80H)= FFH → R5 = FFH

MOV direct, Rn       // copy data from Rn to address direct
MOV 50H, R3          // R3=10H → (50H) = 10H

MOV direct, #data    // load 8-bit immediate data in to address direct
MOV 0A0H, #20H       // (A0H) = 20H

MOV direct1, direct2 // copy data from address direct2 to address direct1
MOV 50H, 83H         // If (83H)=10H → (50H) =10H
```

Note that the brackets '()' specify the contents of the address specified in them and may be read as 'the contents of'.

The operation of instruction MOV R0, 15H is shown in Figure 4.3.

The programmer may use numeric address or name for any SFR. For example, following two instructions does the same operation.

```
MOV P0, #55H    // load constant value 55H into port0 latch
MOV 80H, #55H   // load constant value 55H into port0 latch
```

It should be noted that the access to memory locations between 80H to FFH that do not exist physically will result into errors. Refer Figure 2.2 (Programming model of the 8051) for SFR addresses. Bit addressable data movement instructions are discussed in detail in Chapter 6.

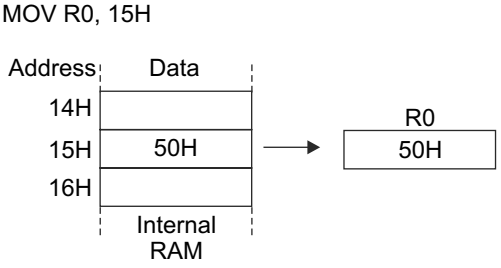


Fig. 4.3 *Direct addressing mode*

THINK BOX 4.1

Can we modify P (parity) bit by writing data directly in to PSW? What will be status of P flag after execution of each of the following instructions?
MOV A, # 07H
MOV PSW, #00H

No. When we write data to the PSW, parity bit remains unchanged because it is affected only by hardware to reflect the parity of A.
After execution of first instruction (MOV A, # 07H), P=1, because A contains odd number of 1's.
After execution of second instruction (MOV PSW, #00H), P remains unaffected (P=1) because its value is affected only by contents of A.

Example 4.3

Illustrate how to load 8-bit data into registers, internal RAM, and moving data between registers and internal RAM.

Solution:

```
MOV   A, #10H      // Load number10H into A register, A=10H (immediate addressing mode)
MOV   R1, A        // Copy contents of A to register R1, R1=10H (Moving data between registers, Register addressing mode)
MOV   R0, A        // R0=10H, copy contents of A to R0 of selected bank (Register addressing mode)
MOV   10H, #20H    // (10H) =20H, load immediate value 20H into internal RAM address 10H
MOV   R1, 10H      // R1= 20H (data present at internal RAM address 10H)
MOV   20H, R1      // (20H) =20H load contents of R1 into internal RAM address 20H
MOV   30H, 10H     // (30H) = (10H) copy contents of internal RAM 10H into 30H
```

Example 4.4

Copy the contents of the DPTR to internal RAM address 10H (DPL) and 11H (DPH)

Solution:

The DPTR is a 16-bit register and internal addresses 10H and 11H are only 8-bit registers, therefore we will move lower byte of DPTR (DPL) into address 10H and higher byte of DPTR (DPH) in to address 11H.

```
MOV 10H, DPL
MOV 11H, DPH
```

4.2.4 Indirect Addressing Mode

The data is specified indirectly in an instruction, i.e. address of the data (rather than data itself) is specified as one of the operand. Here, the register holds the address that contains the data, i.e. the number in the register is treated as a pointer to address. Indirect addressing mode is the most flexible and useful in array operations. There are two types of indirect addressing in the 8051.

1. Register Indirect Addressing Mode

The register indirect addressing uses only register R0 or R1 to hold address of the data in internal RAM, these two registers are also referred to as pointer registers or simply pointers. The symbol @ is used along with R0 or R1 to indicate indirect addressing. For example,

MOV @*Ri*, *#data*	// load constant value in to address contained in *Ri*
MOV @R0, #30H	// If R0=40H, → (40H)=30H
MOV @*Ri, direct*	// copy data form address *direct* to address *Ri*
MOV @R1, 10H	// If (10H)=50H, R1=15H →(15H)=50H
MOV *direct,* @*Ri*	// copy data from address in *Ri* to address *direct*
MOV 10H, @R1	// If R1=50H, (50H)=15H → (10H)=15H
MOV @*Ri,* A	// copy data from A to address in *Ri*
MOV @R0, A	// If A=50H, R0=15H → (15H)=50H
MOV A, @*Ri*	// copy data from address in *Ri* to A
MOV A, @R1	// If R1=50H, (50H)=15H → A=15H

The operation of instruction MOV A, @R1 is shown in Figure 4.4.

Indirect addressing can access internal as well as external memory area. The indirect addresses can be 8 bits or 16 bits. The 8-bit addresses, as mentioned above, are held by R0, R1 or SP only. The 16-bit addresses are held by DPTR, for external memory. External memory data-movement instructions are discussed in Section 4.4.1.

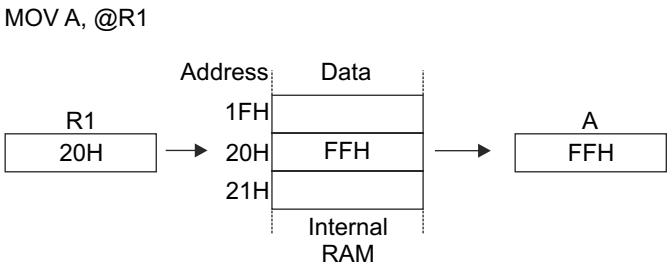


Fig. 4.4 *Indirect addressing mode*

Note: Indirect addressing is used only for accessing RAM locations 00H to 7FH (128 memory locations) and never for accessing SFRs. However, it can access additional 128 RAM locations (from 80H to FFH) available in later versions of the 8051.

Example 4.5

Discuss different methods for moving contents of R2 to R1 (Assume that register bank 0 is selected). Compare all methods with respect to program size and execution speed. Which method is the best?

Solution:

```
(i) Using register addressing
        MOV A, R2                   // 1 byte, 1 machine cycle
        MOV R1, A                   // 1 byte, 1 machine cycle
                                    // Total= 2 bytes, 2 machine cycles (Best method)
(ii) Using register and direct addressing
        MOV A, 02H                  // 2 byte, 1 machine cycle, direct addressing
        MOV 01H, A                  // 2 byte, 1 machine cycle, register addressing
                                    // Total= 4 bytes, 2 machine cycles
(iii) Using direct addressing
        MOV 01H, 02H                // 3 byte, 2 machine cycle, direct addressing
                                    // Total= 3 bytes, 2 machine cycles
(iv) Using register and direct addressing
        MOV 10H, R2                 // 2 byte, 2 machine cycle, register addressing
        MOV R1, 10H                 // 2 byte, 2 machine cycle, direct addressing
                                    // Total= 4 bytes, 4 machine cycles
(v) Using indirect addressing
        MOV R0, #02H                // 2 byte, 1 machine cycle, immediate addressing
        MOV A, @R0                  // 1 byte, 1 machine cycle, indirect addressing
        MOV R1, A                   // 1 byte, 1 machine cycle, register addressing
                                    // Total= 4 bytes, 3 machine cycles
```

2. Indexed Addressing Mode

Two registers are used to form the address of the data. The contents of either DPTR or PC are used as a base address and the A is used as index (or offset) address. The final address is formed by adding these two registers. It results in a forward reference of 0 to 255 bytes from the base address. They are used to access only program memory (internal as well as external.)

Indexed addressing is used to access data tables (lookup tables) from the program memory and implementing jump tables. They are also suitable for multidimensional array operations.

The instructions are,

```
MOVC A, @A+PC       // copy data (or code) byte from program memory address formed by
                    //addition of contents of A and PC into A

MOVC A, @A+DPTR     // copy data (or code) byte from program memory address formed by
                    // addition of contents of A and DPTR into A
```

The mnemonic MOVC means ‘move constant’. The term ‘constant’ indicates that the contents of program memory can only be read but cannot be modified by instructions, they remain constant. As a memory aid the mnemonic MOVC may also be considered as ‘move from code memory’!

The operation of instruction MOVC A, @A+DPTR is shown in Figure 4.5.

Note: For MOVC A, @A+PC, the PC is incremented by 1 (to point to the next instruction) before added to A to form effective address of the code byte. The original data in A is lost and addressed data is placed in the A. The concept of data

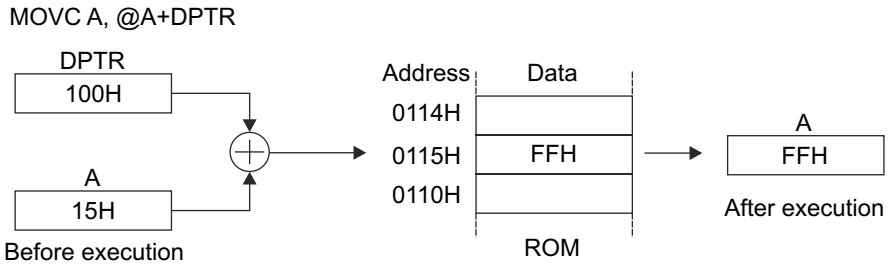


Fig. 4.5 *Indexed addressing mode*

table is discussed in detail in Chapter 8.The use of instruction MOVC A, @A+PC to access look-up tables is illustrated in Example 8.1.

> ***Note:*** Data movement instructions do not change source operands

Example 4.6

Read the contents of the program memory address 50H and copy it into external data memory (RAM) address 50H as well as 100H.

Solution:

Since we have to read data from program memory (ROM), we have to use MOVC' instruction, and to write data in to external RAM, we have to use 'MOVX' instruction.

```
MOV DPTR, #0050H      // initialize DPTR with desired address
MOV A, #00H
MOVC A, @ A+DPTR      // read program memory from effective address
MOVX @DPTR, A         // copy A into external RAM pointed by DPTR (50H)
MOV DPTR, #100H       // point DPTR to 100H
MOVX @DPTR, A         // copy A into external RAM pointed by DPTR (1000H)
```

4.3 OPERAND MODIFIERS: # AND @

As mentioned previously, the 8051 instructions uses symbols # and @ symbols to distinguish addressing mode. The symbol # written before operand indicates that it is an immediate operand while the symbol @ placed before an operand indicates that indirect addressing mode is used.

```
MOV A, #50H      // 50H is immediate operand
MOV A, 50H       // 50H is direct operand
MOV A, R1        // R1 is register operand
MOV A, @R1       // R1 is indirect register operand
```

It is important to note that these two symbols are the potential reasons for the logical errors in the programs, i.e. forgetting them while typing an instruction may result into program with no syntax error but the program will not function as desired. It is difficult and time consuming to find such errors.

Example 4.7

What would be the contents of A and address 10(decimal) after executing the following instructions?

```
MOV R0, #10H
MOV 10H, #20
MOV 35H, #40H
MOV 35, #44H
MOV A, 35
MOV 10, @R0
```

Solution:

A = 44H, (10) = (0AH) = 20d =14H

Example 4.8

The machine code for the instruction MOV A, #10H is 74 10.
Identify the following for the above instruction.
(i) addressing mode (ii) mnemonic (iii) op-code
(iv) source operand (v) destination operand (vi) operation performed

Solution:

(i) addressing mode: Immediate
(ii) mnemonic: MOV
(iii) op-code: 74
(iv) source operand: immediate number 10H
(v) destination operand: A
(vi) operation performed: Immediate number 10H is loaded into A

Summary of the addressing modes and their related memory spaces is given in Table 4.3.

Table 4.3 *Addressing modes and related memory spaces*

Addressing Mode	Memory Space	Operand
Immediate	Program (ROM) memory	Source only
Register	Bank registers (R0 to R7), A, B, CY (Bit)	Source/destination
Direct	Internal RAM, SFRs	Source/destination
Register indirect	Internal/external RAM (@Ri), external RAM @DPTR only)	Source/destination
Index	Program memory (@A +DPTR, @A + PC)	Source only

4.4 EXTERNAL MEMORY DATA MOVEMENTS

The external memory in the 8051 based system may be either data memory (RAM) or program memory (ROM). Instructions that access external memory always use indirect addressing mode. Two different set of instructions are used to transfer data to/from data memory and from program memory.

4.4.1 Data Memory Access

The data memory can be as large as 64 Kbytes. Register R0, R1 and DPTR can be used as a pointer (indirect addressing) to access data bytes from the external RAM. R0 and R1 have limitation to access only address range from 00H to FFH because they are 8-bit register (using 8-bit, we can access 2^8 locations, i.e. 256 bytes – 00H to FFH). DPTR can address any location from 0000H to FFFFH (64 Kbytes, 2^{16} locations). Again to remind, only R0 and R1 can be used with indirect addressing mode.

Mnemonic used for external data memory transfer is MOVX. The letter 'X' in the mnemonic indicates the external data memory. For example, to write data into external RAM, the following instructions are used.

```
MOVX @DPTR, A     // copy contents A into external RAM address contained in DPTR,
                  // If DPTR=1000H and A=10H, → External RAM (1000H)=10H
MOVX @Ri, A       // copy contents A into external RAM address contained in Ri
MOVX @R0, A       // If R0=50H and A=10H, → External RAM (0050H)=10H
```

Similarly, to read data from external RAM, the following instructions are used.

```
MOVX A, @DPTR     // read data from external RAM address pointed by DPTR into A
                  // If DPTR=1000H, (1000H)=20H, → A= 20H
MOVX A, @Ri       // read data from external RAM address pointed by Ri in to A
MOVX A, @R0       // If R0=50H and (50) =10H, → A=10H
```

Here, DPTR or Ri register should be initialized with address of the desired data. Note that all data transfer in this group occurs through Accumulator only, i.e. all external data movements require the A register either as source or destination operand.

Example 4.9

Write instructions to load the value FFH into
(i) Internal RAM addresses 10H to15H
(ii) Port P0 and P1 latch
(iii) External RAM address 1000H

Solution:

(i) One simple way is to load the immediate value directly into the desired addresses as shown below:

```
MOV 10H, #0FFH      // load immediate value FFH into specified addresses directly
MOV 11H, #0FFH
MOV 12H, #0FFH
MOV 13H, #0FFH
MOV 14H, #0FFH
MOV 15H, #0FFH
```

Second way is to load the value into Accumulator and then from A to each address as shown below,

```
MOV A, #0FFH        // First, load immediate value FFH into A and then copy content of A to each address
MOV 10H, A          // Copy contents of A to all addresses from 10H to 15H
MOV 11H, A
MOV 12H, A
MOV 13H, A
MOV 14H, A
MOV 15H, A
```

The advantage of the second method is that it is faster and requires less memory locations to store its machine code. First program requires 12 machine cycles (2 x 6, *two machine cycles for each instruction*) and 18 bytes (3 x 6), while second requires only 7 machine cycles (1 x 7, *only one machine cycle for each instruction*) and 14 bytes (2 x 7).

```
(ii)  MOV P0, #0FFH       // Load value FFH into P0 and P1
      MOV P1, #0FFH
(iii) MOV DPTR, #1000H    // initialize DPTR to point to address 1000H
      MOV A, #0FFH        // load data FFH into A
      MOVX @DPTR, A       // copy contents of A to external RAM address pointed by DPTR (1000H in this example)
```

4.4.2 Program Memory Access

This is the "Read only" type of the data transfer. The data stored in the RAM (either internal or external) are temporary and lost when system is powered down. Sometimes preprogrammed (pre calculated) group of data bytes, i.e. lookup tables, password, strings etc. are needed in some applications. This data must be permanently stored in ROM so that it is always available as and when required. These instructions are also used to read code bytes in some situations.

The instructions to access program memory with examples is already discussed in previous section (see indexed addressing in Section 4.2.4).

Example 4.10

The word 'HI' is burned (written) in the flash ROM address 100H ('H') and 101H ('I'). Copy this word in to internal RAM address 10H and 11H.

Solution:

```
ORG 0000H
MOV DPTR, #100H     // Initialize DPTR to point to ROM address 100H
MOV R0, #10H        // Initialize R0 to point to internal RAM address 10H
CLR A
MOVC A, @A+DPTR     // Read byte from ROM address 100H and place into A
MOV @R0, A          // Copy byte read from ROM to internal RAM address 10H
INC R0              // Increment R0 and DPTR to point to next byte
```

```
INC DPTR
CLR A
MOVC A, @A+DPTR     // Read next byte from ROM address 101H and place into A
MOV @R0, A           // Copy byte to the internal RAM address 11H
ORG 0100H            // Store data in ROM from address 100H onwards
DB 'HI'              // The word to be stored in ROM
```

Note that there are more efficient ways of performing above task, i.e. using loops. The concept of looping is discussed in Chapter 7.

4.5 DATA EXCHANGE

All instructions discussed thus far moves data in only one direction at a time, i.e. from source to destination and an original content of the source operand is not changed.

Exchange instructions as the name suggests, moves data in two directions simultaneously, i.e. from the source to the destination as well as from the destination to the source. All exchanges are only internal to the 8051 and use A register. The examples of exchange instructions are given below.

XCH A, *Rn*	// exchange contents of A and *Rn*
XCH A, R7	// If A= 10H, R7= 30H, → A=30H, R7=10H
XCH A, *direct*	// exchange contents of A and address *direct*
XCH A, 30H	// If A= 10H, (30H)= 20H, → A=20H, (30H)=10H
XCH A, *@Ri*	// exchange contents of A and address in *Ri*
XCH A, @R1	// If A= 10H, R1=30, (30H)= 20H, → A=20H, (30H)=10H
XCHD A, *@Ri*	// exchange lower nibble of A and lower nibble of address in *Ri*
XCHD A, @R1	// If A= 15H, R1=30, (30H)= 20H, → A=10H, (30H)=25H // higher nibbles of both operands are not affected

Example 4.11

Exchange the contents of the PSW and internal RAM address 50H.

Solution:

```
MOV A, PSW    // copy the contents of PSW into A because exchange
              // operation can be done only through A
XCH A, 50H    // exchange A with contents of address 50H
              // now A has contents of address 50H
MOV PSW, A    // move A in to PSW
```

THINK BOX 4.2

Realize the operation performed by XCH A, R2 using MOV instructions.

```
MOV 30H, R2
MOV R2, A
MOV A, 30H
```

4.6 PUSH AND POP INSTRUCTIONS

PUSH and POP are special instructions that are associated with stack operation. Using these instructions, the data is transferred between stack and specified direct address.

The stack is a special memory area in the internal RAM that is used for temporary storage and retrieval of the data, while the execution of a program. It is Last In First Out (LIFO) type memory. The data stored in the stack can be numbers or

address. This section of memory is accessed using PUSH, POP, calls and interrupts. The stack is accessed with the help of stack pointer (SP) register.

SP holds address (of internal RAM) where data from source operand will be saved (pushed) or data to destination address will be retrieved (popped). Stack pointer always points to top of the stack, i.e. last memory address accessed. It should be initialized to a defined value (default value of SP is 07H; refer Topic 7.4 for more details on SP initialization). PUSH instruction saves data on to location pointed by SP, while saving a new data byte, the SP is automatically incremented by 1 and data byte is stored at SP+1 address. POP instruction retrieves data from location pointed by SP, while retrieving, data will be read from address in SP and then SP is decremented by 1. PUSH and POP instructions use SP register indirectly and not specified in an instruction, i.e. these instructions assumes that SP is pointing to top of the stack. These instructions supports only direct addressing mode. The formats of PUSH and POP instructions are given below. Note that a programmer should define the stack at proper place before it can be used; it is done by loading suitable internal RAM address in the SP.

```
PUSH direct     // Increment SP, copy data from address direct to address in SP
POP direct      // copy data from address in SP to address direct, and then decrement SP
```

For example,

```
MOV SP, #50H    // initialize SP to point to address 50H
MOV 35H, #10H   // load data 10H into address 35H
PUSH 35H        // Increment SP to 51H, copy contents of address 35H to address 51H i.e. (51H) = 10H
POP 00H         // copy data from address 51H to address 00H, i.e. (00H) =10H and decrement SP to 50H.
```

Example 4.12

Explain how contents of Accumulator and B registers can be stored and retrieved from the stack.

Solution:

The given task will be accomplished by following set of instructions.

```
MOV SP, #50H    // SP =50 H initialize the stack pointer
MOV A, #10H     // A = 10H
MOV B, #20H     // B = 20H
PUSH ACC        // SP=51H, and (51H) = 10H, SP is incremented by 1 and contents of
                // Accumulator (address E0H) is stored at memory location 51H
PUSH B          // SP = 52, (52H) = 20H, SP is again incremented by 1 and contents of B is stored at memory location 52H
POP B           // B = (52H) = 20H, SP = 51H, data is retrieved into B from address 52H and SP is decremented by 1.
POP ACC         // A= (51H) = 10H, SP = 50H; data is retrieved into A from address 51H and SP is again decremented by 1.
```

Since PUSH and POP instructions support only direct addressing mode, the instruction PUSH ACC will be assembled as PUSH 0E0H and PUSH B instruction as a PUSH 0F0H, i.e. the assembler will replace the names of SFRs with their addresses. The operation of above program is illustrated in Figure 4.6.

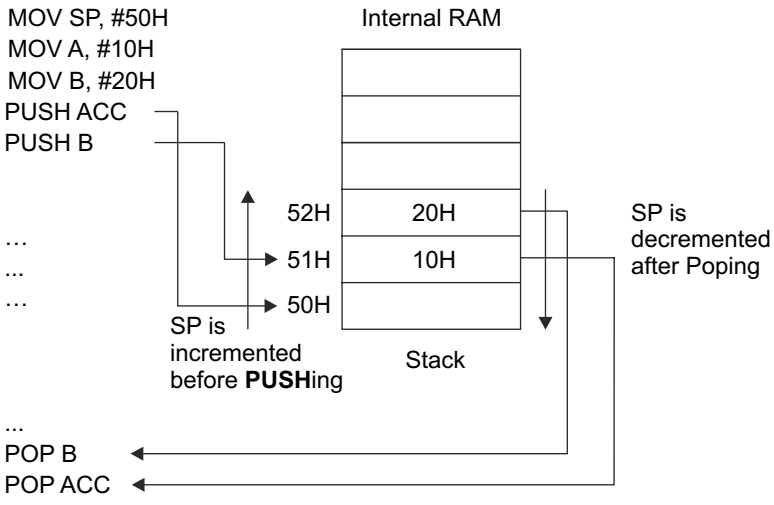


Fig. 4.6 *Stack operation*

Example 4.13

Copy the contents of registers R0 to R7 into internal RAM addresses 40H to 47H respectively using PUSH instructions. Assume bank 0 is selected.

Solution:

Since PUSH instruction automatically increment SP by 1 before saving data on to the stack, we have to initialize SP with 3FH to store data at 40H.

```
MOV SP, #3FH        // Initialize SP with address 3FH
PUSH 00H            // Store R0 (address 00H) at 40H
                    // PUSH will first increment SP to point to 40H and contents of R0 is saved at address40H
PUSH 01H            // Similarly, save R1 to R7 from 41H to 47H
PUSH 02H
PUSH 03H
PUSH 04H
PUSH 05H
PUSH 06H
PUSH 07H
```

Bitwise data transfer instructions are discussed in Chapter 6, and program memory addressing is discussed in detail in Chapter 7.

Example 4.14

State validity of following instructions. Give reason if an instruction is invalid.

(a) MOV R0,#10H	**(b) MOV R0, 10H**	**(c) MOV R2, #256**	**(d) MOV A, @R0**	**(e) MOV A, @R3**
(f) PUSH A	**(g) PUSH 0E0**	**(h) MOV R1, R2**	**(i) MOV 10H, 20H**	**(j) MOV DPTR, A**

Solution:

(a) Valid
(b) Valid
(c) Invalid, R2 is 8-bit register, therefore it can hold maximum value 255 (FFH)
(d) Valid
(e) Invalid, only R0 and R1 can be used for indirect addressing
(f) Invalid, PUSH instructions can be used with only direct addressing.
(g) Valid
(h) Invalid, Register to register data transfer is not allowed
(i) Valid
(j) Invalid, size of both operands should be same

Summary of data movement instructions with examples is given in Table 4.4.

THINK BOX 4.3

Which register of the 8051 can not be accessed using MOV instruction?
PC

POINTS TO REMEMBER

- The microcontrollers typically spend maximum time in data movement operations and usually have maximum instructions for data movement.
- Immediate, register, direct and indirect addressing modes are essentially four addressing modes used by all microcontrollers.

Table 4.4 *Data movement instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
MOV A, **<src>**	A= **<src>**	MOV A, **direct**	MOV A,@**Ri**	MOV A,**Rn**	MOV A,**#data**
		MOV A, **20H**	MOV A,@**R0**	MOV A,**R3**	MOV A,**#10H**
MOV <dest>, **A**	<dest>= **A**	MOV direct, **A**	MOV @Ri,**A**	MOV Rn,**A**	
		MOV 10H, **A**	MOV @R1,**A**	MOV R2,**A**	
MOV <dest>, **<src>**	<dest>= **<src>**	MOV direct, **direct**	MOV direct,@**Ri**	MOV direct,**Rn**	MOV direct,**#data**
		MOV 05H, **12H**	MOV 10H,@**R0**	MOV 50H,**R5**	MOV 50H,**#10H**
MOV DPTR, **#data 16**	DPTR= 16 bit immediate constant				MOV DPTR,**#data16**
					MOV DPTR,**#1000H**
MOVC A, @**A+DPTR**	Read Program memory at (A+DPTR)		MOVC A,@**A+DPTR**		
			MOVC A,@**A+DPTR**		
MOVC A, @**A+PC**	Read Program memory at (A+PC)		MOVC A,@**A+PC**		
			MOVC A,@**A+PC**		
MOVX A, **<dest>**	Read external RAM from <dest>		MOVX A**,@Ri**		
			MOVX A,@**R0**		
			MOVX A,@**DPTR**		
			MOVX A,@**DPTR**		
MOVX <dest>, **A**	Write external RAM at <dest>		MOVX @Ri, **A**		
			MOVX @R0**, A**		
			MOVX @DPTR**, A**		
			MOVX @DPTR**, A**		
PUSH **<src>**	INC SP: MOV "@SP", <src>	PUSH **direct**			
		PUSH **10H**			
POP **<dest>**	MOV <dest>, "@SP": DEC SP	POP **direct**			
		POP **12H**			
XCH A, **<BYTE>**	ACC & <BYTE> exchange data	XCH A, **direct**	XCH A,@**Ri**	XCH A,**Rn**	
		XCH A, **12H**	XCH A,@**R1**	XCH A,**R0**	
XCHD A, @**Rn**	ACC & @Rn exchange low nibbles		XCHD A,@**Ri**		
			XCHD A,@**R1**		

- The data-movement instruction does not affect the flags and actually copy data from source to destination.
- Immediate addressing is fast but at run-time it is least flexible. Register addressing occupies only one-byte memory, and is fast. Direct addressing is more flexible than immediate and register addressing but requires more bytes. Indirect addressing is the most flexible.
- Indirect addressing is used only for accessing RAM locations 00H to 7FH (128 memory locations) and never for accessing SFRs.
- Indexed addressing is used to access data tables (lookup tables) from program memory and implementing jump tables. They are also suitable for multidimensional array operations.
- Instructions that access external memory always use indirect addressing mode and involves use of A register.
- All exchanges are only internal to the 8051 and use A register. Exchange operation moves data in two directions simultaneously.
- PUSH and POP instructions supports only direct addressing mode.

OBJECTIVE QUESTIONS

1. The instructions that copy data from register R0 to register R5 when bank 0 is active are,
   (a) MOV R5, R0 (b) MOV 05, 00 (c) MOV R5, 00 (d) all of the above
2. The addressing mode for an instruction MOV R0, #30H is,
   (a) register addressing mode (b) direct addressing mode
   (c) immediate addressing mode (d) none of the above
3. Determine the incorrect instruction.
   (a) MOVX A,@R1 (b) MOVC A, @A+DPTR (c) MOV @R0, A (d) MOV @DPTR, A
4. MOV A, @ R1 will,
   (a) copy R1 to the accumulator
   (b) copy the accumulator to R1
   (c) copy the contents of internal RAM pointed by R1 to the accumulator
   (d) copy the contents of external RAM pointed by R1 to the accumulator
5. Which of the following instructions will move the number 27H in the accumulator?
   (a) MOV A, P27 (b) MOV A, #27H (c) MOV A, 27H (d) MOV A, @27
6. Which of the following instructions will move the value of port 3 to the register R2?
   (a) MOV P2, R3 (b) MOV R3, P2 (c) MOV 3P, R2 (d) MOV R2, P3
7. The following instruction will copy the contents of A to the address 50H, MOV 50H, A.
   (a) True (b) False
8. Which of the following instructions will copy the contents of RAM whose address is in register R0 to port 3?
   (a) MOV @ P3, R0 (b) MOV @ R0, P3 (c) MOV P3, @ R0 (d) MOV P3, R0
9. Which of the following is invalid instruction/s?
   (a) MOVX R0, @DPTR (b) MOVC A, @A+PC (c) XCH A, 10H (d) XCH A, #10H
10. MOV 10H, 20H is ___byte instruction.
    (a) 1 (b) 2 (c) 3 (d) 4
11. PUSH/POP instructions support only _____ addressing mode.
    (a) Register (b) Immediate (c) Direct (d) Register indirect
12. MOV *destination byte, source byte* instruction has total ___ formats.
    (a) 10 (b) 12 (c) 15 (d) 18
13. The Immediate operand can be,
    (a) source operand only (b) destination operand only
    (c) either source or destination depending upon instruction (d) source and destination operand for some instructions
14. Which of the following instructions is an example for the direct addressing mode?
    (a) MOVA, @R0 (b) MOV R0, #10H (c) MOV 10H, A (d) MOV R5, A
15. Direct addressing mode is used to access,
    (a) internal data memory (b) external data memory
    (c) internal program memory (d) external program memory

16. The addressing mode used by instruction MOVC A, @A+PC is,
    (a) immediate addressing (b) direct addressing
    (c) indirect addressing (d) indexed addressing
17. External data movements can be done using,
    (a) A only (b) A or B (c) any of the port (d) any of the SFR
18. Indirect addressing is used for accessing internal RAM locations,
    (a) 00H to FFH (b) 00H to 7FH (c) 00H to 1FH (d) 20H to 2FH

Answers to Objective Questions

1. (b), (c)	2. (c)	3. (d)	4. (c)	5. (b)	6. (d)	7. (a)	8. (c)	9. (a), (d)
10. (c)	11. (c)	12. (c)	13. (a)	14. (c)	15. (a)	16. (d)	17. (a)	18. (b)

REVIEW QUESTIONS WITH ANSWERS

1. List the address spaces where operands of an instruction may be present.

A. There are five types of address spaces where operands may be present.
   (i) Bank Registers (R0–R7)
   (ii) Accumulator
   (iii) Internal data RAM (00–7F & SFRs)
   (iv) External data RAM (data memory)
   (v) Program ROM (internal + external ROM)

   Apart from these address spaces, one more place from where data can be obtained is from instruction itself, i.e. in case of immediate addressing mode, data is an integral part of the instruction.

2. What is limitation of immediate addressing?

A. In immediate addressing, the data is fixed; therefore at run-time it is not flexible.

3. Size of the immediate data in the 8051instructions is always 8 bits. True/False.

A. False. We can load 16 bit data in to DPTR register.

4. Can we specify data in decimal or ASCII format?

A. No. Microcontrollers understand only binary number system. However, we can use decimal numbers or ASCII characters if they are supported by an assembler. Finally these numbers will be converted into binary by assembler.

5. Which addressing modes can be used with PUSH and POP instructions?

A. Only direct addressing mode can be used with PUSH and POP instructions.

6. Write instruction/s to load value FFH internal RAM address 50H using direct and indirect addressing modes

A. Direct addressing: MOV 50H,#0FFH
   Indirect addressing: MOV R1, #50H
   MOV @R1, #0FFH

7. State the difference between following instructions.

MOV R0, #10
MOV R0, #10H
MOV R0, 10H

A.
```
MOV R0, #10     // Load decimal value 10 (A Hex) in to R0
MOV R0, #10H    // Load hexadecimal value 10H in to R0
MOV R0, 10H     // Move data at internal RAM address 10H to R0
```

8. Write instruction/s to move contents of R0 to R1.

A.
```
MOV A,R0
MOV R1, A
```

9. Can we write instruction MOV R1, R0 for above operation?

A. No, because register to register move instruction is not supported by the 8051.

10. Where is an indexed addressing mode useful?

A. It is used to access look up tables and to implement jump tables.

11. Which address spaces are accessed in direct addressing mode?

A. Internal RAM and SFRs.

12. What is the limitation in using indirect addressing mode?

A. We can use only R0 and R1 to hold indirect data (address of the data)

13. Where is indirect addressing commonly used?

A. It is used commonly used where similar operation is to be performed on different data, i.e. it is more commonly used to implement pointers in loops.

14. Differentiate between mnemonics MOV, MOVX and MOVC.

A. MOV is used to move data within microcontroller RAM, MOVX is used to transfer data between A and external RAM while MOVC is used to read data from code memory (either internal or external) into A .

EXERCISE

1. What is meant by term "addressing mode"? List addressing modes supported by the 8051 with suitable examples.
2. In which address space is the immediate data stored?
3. State addressing mode used in following instructions.
   (a) MOV R0,#10H (b) ADD A, @R0 (c) MOV 50H,@R1 (d) MOVX @DPTR, A
   (e) MOV A, R5 (f) ANL 50H, #10H (g) MOV 50H, 60H (h) ORL 50H, #50H
   (i) MOVC A, @ A+DPTR (j) MOV DPTR, #2500H
4. Differentiate between data and program addressing modes.
5. How are the following instructions assembled?
   (a) MOV A, #20 (b) MOV A, 'C' (c) MOV R0, #-10H
6. What is the use of operand modifiers # and @?
7. Why is the instruction MOV R1, #256 invalid?
8. Find the number of bytes for following instructions.
   (a) MOV R0, A (b) MOV R0, #10H (c) MOV A, #10H
   (d) MOV 50H, #10H (e) MOV 50H, 10H (f) MOV B, #10H
9. Find value of SP and contents of stack after each of the following instructions.
   MOV SP, #50H
   MOV R0, #10H
   MOV R1, #20H
   PUSH 01H
   PUSH 00H
   MOV R0, #05H
   ADD A, R0
   POP 00H
   POP 01H
10. What is meant by stack overflow?
11. Classify the instructions of the 8051 with respect to their functions.
12. Write instruction/s to move the PC into the DPTR and vice versa.
13. Write two instructions for each of the addressing mode of the 8051.
14. Write instruction/s to save R0 of bank 3 on the stack.
15. List the address spaces accessed by each of the addressing mode in the 8051.
16. Indexed addressing mode is type of indirect addressing. Justify
17. Explain operation the MOVC A, @ A+DPTR instruction.
18. What are the advantages of using indirect addressing mode?
19. Can we access SFRs using indirect addressing mode?
20. How external data memory can be accessed?
21. What is the limitation of exchange instruction?
22. What is the maximum size of the stack in the 8051?
23. Write a program to transfer a byte from code memory address 1000H to internal RAM and external RAM address 10H and 1000H respectively.
24. Write a program to transfer a byte from external RAM address 200H to external RAM address 300H.

5

Arithmetic and Logical Instructions

Objectives

- List the arithmetic and logical operations performed by the 8051
- Explain the concept of signed and unsigned numbers and range of these numbers supported by the 8051
- Perform the signed and unsigned addition and subtraction
- Perform the multiplication and division
- Appreciate the role of flags in the arithmetic operations
- Introduce binary coded decimal (BCD) numbers and BCD arithmetic
- Illustrate the use of increment and decrement instructions
- Discuss OR, AND, NOT and EX-OR operations and their applications
- Explain byte as well as bit level logical operations
- List and illustrate the rotate and swap operations performed by the 8051

Key Terms

- Addition
- AND/OR/NOT/EXOR
- Arithmetic Operations
- Auxiliary Carry Flag: AC
- BCD Arithmetic
- Byte/Bit Operations
- Carry Flag: C or CY
- Division
- Logical Operations
- Multiplication
- Overflow Flag: OV
- Rotate
- Signed Arithmetic
- Subtraction
- Unary Operations

The 8051 microcontroller supports basic arithmetic operations such as addition, subtraction, division, multiplication, increment, decrement and logical operations such as AND, OR, NOT, and EX-OR. The unsigned operations with only 8-bit binary integers are supported directly, however, signed and multi-byte operations can be performed with the help of overflow and carry flags. It also supports BCD operations. Thus, arithmetic and logical operations can be used for mathematical calculations and data manipulation.

5.1 | ARITHMETIC INSTRUCTIONS

This group of instructions performs arithmetic operations. The arithmetic operations modify arithmetic flags Carry (CY or C), Overflow (OV) and Auxiliary Carry (AC). These flags (1-bit register) are modified (set/reset) according to the result obtained in an operation. The status of the flags is used as a test condition to make decisions by the program flow control (branch) instructions. A programmer may interpret the flag in more than one way. For example, C flag shows a carry out during addition and indicates borrow during subtraction.

An important point to note is that the flags are stored in the PSW register. Any instruction that modifies byte or bit in the PSW register can change the flags.

First, the arithmetic operations for unsigned numbers are discussed because they are directly supported by the 8051 and in later section, signed operations are discussed. For unsigned numbers, all data bits are used to represent the magnitude of a number. For example, in an 8-bit unsigned number, entire 8 bits are used for magnitude only; therefore, 8-bit number can have any value between 00H to FFH (0 to 255 decimal).

5.1.1 Addition

The 8051 supports half as well as full addition. The half addition is used to add two operands (of 8 bits each) specified in an instruction. While adding two 8-bit numbers, it is possible to get 9-bit result. The 9^{th} bit is the carry and it is stored in the carry (CY or C) flag. The full addition includes carry flag in the addition. The full addition is used for multi-byte operations and it is discussed in the next section.

The mnemonic ADD is used to add two 8-bit operands (half addition). The format of this addition instruction is,

```
ADD A, source      // A=A+ source.
                   // Add contents of A with operand source and place result into A.
```

All addressing modes can be used to specify the *source*. **The register A is always the destination as well as one of the source operand.** The formats of addition instructions for different addressing modes are as follows:

```
ADD A, #data       // add contents of A with immediate data and store result in A
ADD A, #10H        // A= A+10H, If A=05H → A=05H+10H=15H, C=0
ADD A, Rn          // add contents of A with contents of Rn and store result in A
ADD A, R0          // A = A+ R0, If A= F5H, R0=10H → A=F5H+10H=05H, C=1
ADD A, direct      // add A with contents of address direct and store result in A
ADD A, 40H         // A= A + (40H), If A=10H, (40H) =15H → A=10H+15H=25H, C=0
ADD A, @Ri         // add A with contents of address in Ri, store result in A
ADD A, @R0         // A= A+ (R0), If A=20H, R0=10H, (10H) =30H
                   // → A=20H+ (10H) = 20H+30H=50H,C=0
```

After addition, the C flag is set to 1 if there is carry out from MSB (bit position 7), otherwise it is cleared to zero. The AC flag is set to 1, if there is carry out from the lower nibble (bit position 3). The OV flag is set to1 if there is a carry out from bit 7 but not from bit position 6, or if there is a carry out of bit position 6 but not from position 7. The conditions in which the flags are set to 1 are illustrated in Figure 5.1.

The condition in which OV flag is set is given as:

OV = C7 XOR C6.

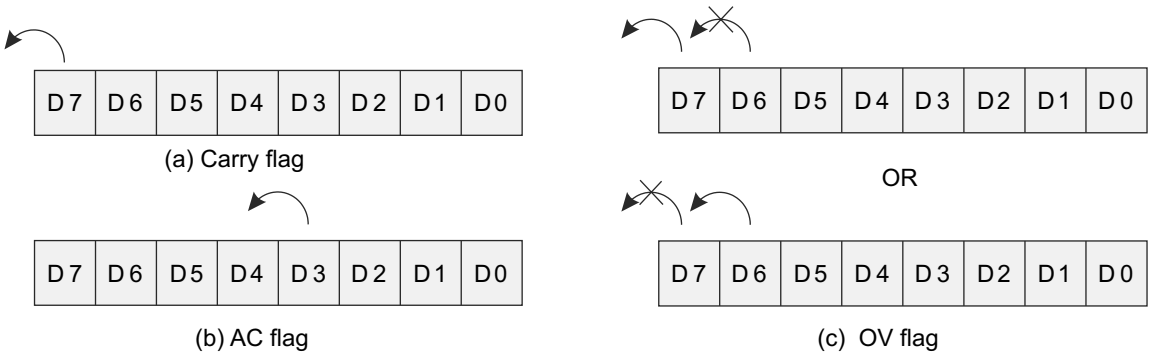


Fig. 5.1 *Conditions that set the flags*

Example 5.1

Assume R0=20H, (20H) =30H, (40H) = FFH, and A=50H. Illustrate the operation of ADD instruction for different addressing modes. Note that (XXH) is read as data at address XXH.

Solution:

Assuming the contents of register/memory locations as given above, before execution of **each** of the following instructions:

```
ADD A, #10H        // A= 50H+10H= 60H, C=0, AC=0
ADD A, R0          // A= 50H+20H= 70H, C=0, AC=0
ADD A, 40H         // A= 50H+FFH= 4FH, C=1, AC=0
ADD A, @R0         // the value present at address 20H is 30H, therefore
                   // A= 50H+30H= 80H, C=0, AC=0
```

Example 5.2

What will be the status of carry (CY) and auxiliary carry (AC) flags after execution of following instructions?

```
MOV A, #97H
ADD A, #80H
```

Solution:

The above two instructions add 97H with 80H and result will be placed in the A. The effect of addition on flags will be as shown below:

```
  97H  =   1001 0111 b
+ 80H  =   1000 0000 b
 117H     1 0001 0111 b
```

As can be seen from the above addition, there will be carry out from bit D7 and no carry from bit D3. Therefore, the carry flag is set, i.e. CY = 1 and auxiliary carry flag is reset, i.e. AC = 0.

Example 5.3

Write a program to add two numbers stored at internal RAM address 10H and 11H and store the result into internal RAM address 20H.

Solution:

Since the numbers to be added are stored in an internal RAM, we can access these numbers using direct addressing as shown in the program.

```
MOV A, 10H         // Read number stored at internal RAM address 10H into A
ADD A, 11H         // Add contents of A with data stored at address 11H and store result in to A
MOV 20H, A         // copy result of addition into internal RAM address 20H
```

Note that addition of two 8-bit numbers may generate 9-bit result (9^{th} bit in carry). The carry should be stored at suitable location when more than two bytes are added. This is illustrated in Example 5.5.

Addition of Multi-Byte Numbers

While dealing with multi-byte numbers (size of numbers is more than 8 bits) addition, we need to consider the propagation of carry from lower bytes to the higher bytes, i.e. we need to perform the full addition. The full addition includes carry

flag in the operation. The instruction ADDC is used for such operations. The suffix C after ADD indicate carry flag is also included in the addition. The operation of ADDC instructions is same as ADD instructions except that it also adds the contents of a carry flag with addition of both the operands. **ADDC means add with carry**. The formats of ADDC instructions for different addressing modes are,

ADDC A, #data	// add contents of A, immediate *data* and carry, store result in A
	// i.e. A = A + data + C
ADDC A, #10H	// A= A+10H+C, If A=05H, C=1→ A=05H+10H+1=16H, C=0
ADDC A, *Rn*	// add contents of A, contents of *Rn* and carry, store result in A
	// i.e. A= A+ *Rn* + C
ADDC A, R0	//A=A+ R0+C, If A=20H,R0=30H,C=0→ A=20H+30H+0=50H, C=0
ADDC A, *direct*	// add A, contents of address *direct* and carry, store result in A
	// i.e. A= A+ (direct) + C
ADDC A, 40H	// A= A + (40H) +C, If A=F0H, (40H) =15H, C=1 →
	// A=F0H+15H+1=06H, C=1
ADDC A, @*Ri*	// add contents A, contents of address in *Ri* and C, store result in A
	// A= A+ (*Ri*) + C
ADDC A, @R0	// A= A+ (R0) + C, If A=20H, R0=10H, (10H) =30H ,C=1
	//→ A=20H+ (10H)+1 = 20H+30H+1=51H,C=0

Multi-byte number addition is illustrated in Example 5.4.

Example 5.4

Write program to add two 16-bit numbers 42E1H and 255CH.

Solution:

The addition of these two 16-bit numbers will be performed in the following manner,

```
    1
  42 E1 H
+ 25 5C H
  68 3D H
```

When lower bytes are added, the result will be 3DH and CY is set (E1+5C=3D, CY=1). The carry is propagated to higher byte, and added to higher bytes, which results in 42 + 25 + 1 = 68H. The program to perform above addition is given below:

```
MOV A, #0E1H        // add lower bytes
ADD A, #5CH
MOV R5, A           // save lower byte result in to R5
MOV A, #42H         // add upper bytes including carry
ADDC A, #25H
MOV R7, A           // save higher byte of result in to R7
```

Usually, the ADD instruction is used for lower byte addition in multi-byte number addition. While ADDC can be used for all the bytes provided that the CY is properly initialized, i.e. for lower byte addition, ADDC can be used if carry is cleared before such addition as shown below:

```
CLR C               // initially carry is cleared
MOV A, #0E1H        // add lower bytes
ADDC A, #5CH
MOV R5, A           // save lower byte result in to R5
MOV A, #42H         // add upper bytes including carry
ADDC A, #25H
MOV R7, A           // save higher byte of result in to R7
```

Note: For this example, carry is not generated after addition of upper bytes; therefore it is not taken care of. Otherwise, it should have been considered as a part of result.

Example 5.5

Write a program to add three numbers stored at internal RAM address 10H, 11H and 12H and store the lower byte of result into internal RAM address 20H and upper bits into 21H.

Solution:

When we add three bytes, the maximum result will require 10 bits (For example, FFH+FFH+FFH=2FDH). The upper two bits of the result must be stored in 21H.

```
CLR C
MOV 21H, #00H       // clear 21H to store upper bits of addition
MOV A, 10H          // add contents of address 10H and 11H
ADD A, 11H
MOV R0, A           // store partial sum temporarily in R0
MOV A, 21H          // add carry (if any) to higher byte of result
ADDC A, #00H
MOV 21H, A          // store back
MOV A, R0           // retrieve partial sum (10H+11H)
ADD A, 12H          // add partial sum with contents of address 12H
MOV 20H, A          // store lower byte of sum in 20H
MOV A, 21H          // add carry (if any) to higher byte of result
ADDC A, #00H
MOV 21H, A          // store upper byte of result in 21H
```

Note: Novice programmers usually **erroneously** consider this type of addition (addition of multiple bytes) as multi-byte numbers addition!!

5.1.2 Subtraction

The mnemonic for subtraction is SUBB, **it means subtract with borrow**. The format of instruction is,

SUBB A, *source* // A= A – *source* – CY
// subtract carry and operand *source* from A and place result into A.

Similar to addition, all four addressing modes can be used for the *source* operand. **The A is always destination as well as one of the source operand.** The formats of subtraction instructions for different addressing modes are as follows:

SUBB A, #data	// subtract immediate *data* and carry from A, store result in A // i.e. A = A – data – C
SUBB A, #10H	// A= A–10H–C, If A=20H, C=1 → A=20H–10H–1=0FH, C=0
SUBB A, *Rn*	// subtract contents of *Rn* and carry from A, store result in A // i.e. A= A– *Rn* – C
SUBB A, R0	//A= A– R0–C, If A=30H, R0=20H,C=0 → A=30H–20H–0=10H, C=0
SUBB A, *direct*	// subtract contents of address *direct* and carry from A, store result in A // i.e. A= A– (direct) – C
SUBB A, 40H	// A= A – (40H) –C, If A=10H, (40H) =15H, C=0 → // A=10H–15H–0=FBH, C=1
SUBB A, @*Ri*	// subtract contents of address in *Ri* and C from A, store result in A // A= A– (Ri) – C
SUBB A, @R0	// A= A– (R0)–C, If A=50H, R0=10H, (10H) =30H, C=1 //→ A=50H– (10H)–1 = 50H–30H–1=1FH, C=0

Subtraction instruction treats carry flag as borrow and always subtract carry flag as a part of operation. In fact, meaning of the mnemonic SUBB is subtract with borrow. Many microcontrollers/processors have two different instructions for subtraction, SUB and SUBB. But in the 8051, we have only SUBB. To perform operation equivalent to SUB using SUBB we have to make CY=0 before execution of an instruction.

After subtraction,
CY=1 if borrow is needed into bit 7, CY=0 otherwise.
AC=1 if borrow is needed into bit 3, AC=0 otherwise.
OV=1 if borrow is needed into bit 7 and not in bit 6 or borrow in to bit 6 and not in bit 7, OV=0 otherwise.

Note: The Accumulator is always destination as well as one of the source operand for Addition and Subtraction.

Example 5.6

Write instructions to subtract 10H from 30H using immediate and register addressing.

Solution:

Using immediate addressing,

```
CLR C
MOV A, #30H        // A=30H
SUBB A, #10H       // A= A– 10H– C= 30H–10H– 0=20H
```

Using register addressing,

```
CLR C
MOV A, #30H        // A=30H
MOV R0, # 10H      // R0= 10H
SUBB A, R0         // A= A–R0–C= 30H–10H– 0 =20H
```

Note that carry flag is cleared before subtraction of 8-bit numbers.

In multi-byte subtraction C is cleared before subtraction of first byte and then included in subsequent higher byte operations.

5.1.3 Signed Arithmetic

All types of data considered so far in this text are assumed to be unsigned numbers, i.e. entire 8-bit operand represents magnitude only, but in real life the number can either be positive or negative. There should be some mechanism to represent and process such positive or negative numbers. The signed numbers are used to serve the purpose. In signed numbers, the most significant bit (MSB) is used to represent the sign (+ or –) and rest of the bits are used for the magnitude as shown in Figure 5.2. For the numbers greater than 8 bits, the leftmost bit of the most significant byte is used to represent the sign. The '0' in the MSB position indicate positive sign while '1' indicates negative sign.

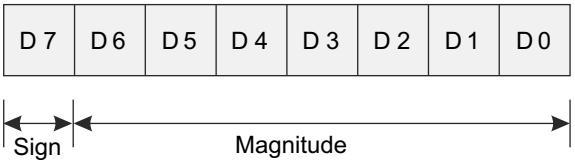


Fig. 5.2 *Signed numbers*

1. Positive Numbers

Bit D7 is '0' for positive numbers, since only seven bits (D0 to D6) are used for magnitude, the range of positive number that can be represented by 8-bit signed number is 0 to +127 as shown in Table 5.1.

If a positive number is larger than +127, a 16-bit or larger sized operand must be used.

Table 5.1 *8-bit positive numbers*

Decimal	Binary	Hex.
0	**0**000 0000	00
+1	**0**000 0001	01
+2	**0**000 0010	02
:	:	:
:	:	:
+126	**0**111 1110	7E
+127	**0**111 1111	7F

2. Negative Numbers

Bit D7 is 1 for negative numbers, however, negative numbers are not represented in true binary form, but it is represented in 2's complement form. Example 5.7 shows how negative numbers are represented.

Example 5.7

Represent –6 and –128 in 8–bit binary using 2's complement representation.

Solution:

The steps involved in this representation

1. Represent the magnitude of the number in 8-bit binary (without sign),

2. Complement each bit and
3. Add one to it.

The given number –6 is represented as,

```
  0000 0110        6 (magnitude) in 8 bit binary
  1111 1001        complement
+        1        add 1
  1111 1010
```

Thus, 11111010 (FAH) is the signed representation (or 2's complement) of – 6.

Similarly, for –128

```
  1000 0000 ——————→ 128 (magnitude),
  0111 1111 ——————→ invert each bit,
+        1 ——————→ add 1
  1000 0000
```

1000 0000 (80H) is the signed representation of –128.

On the contrary, if the 8-bit 2's complement signed number is given; the equivalent decimal number can be found as follows:

If the MSB is 0, the number is positive and other bits represent magnitude of the number. For example, consider 64H (0110 0100), since its MSB is 0, it is a positive number and remaining bits represent 100d, therefore the number is +100d.

If MSB is 1, a number is negative and the magnitude can be found by calculating 2's complement (because negative numbers are already represented in 2's complement) of a number. For example, consider FFH (1111 1111), since its MSB is 1, it is a negative number and magnitude is found from 2's complement of FFH, it is 1H (0000 0001), therefore the number represented by FFH is –1.

The other method is to find equivalent decimal number from the positional weight of each bit of a number. The positional weights for an 8-bit signed number are shown below:

–128	64	32	16	8	4	2	1
MSB							LSB

For example, consider 80H (1000 0000).

The decimal equivalent value of 80H is calculated from positional weights as

1× (–128) + 0×64 + 0×32 + 0×16 + 0×8 +1×4 + 0×2 + 0×1= –128d

Similarly, decimal equivalent for FFH (1111 1111) can be found as

1× (–128) + 1×64 + 1×32 + 1×16 + 1×8 + 1×4 + 1×2 + 1×1= –128+127= –1d

And for 7FH (0111 1111),

0× (–128) + 1×64 + 1×32 + 1×16 + 1×8 +1×4 +1×2 + 1×1 = +127d

From the above examples, we can say that range of byte-sized negative numbers is –1 to –128.

Table 5.2 shows byte-sized signed numbers.

The CY and OV flags are used to handle the unsigned and the signed operations. CY is generally used in the unsigned arithmetic while OV is used in the signed arithmetic.

Table 5.2 *8-bit signed numbers*

Decimal	Binary	Hex.
–128	1000 0000	80
–127	1000 0001	81
:	:	:
:	:	:
–1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+127	0111 1111	7F

THINK BOX 5.1

What are the signed and unsigned decimal representations of an 8-bit number 21H?

Both are 33

Discussion Question For a byte-sized signed number, why is there only 127 positive numbers, while negative numbers are 128?

Answer The number 0 is considered to have positive sign, so we may consider there are 128 (0, +1 to +127) positive numbers.

3. Overflow

A byte-sized signed number may range from –128d (1000 000b) to +127d (0111 1111b). If the result of the operation on signed number exceeds this range, an overflow will occur, which indicates error in a result. The 8051 indicate this error by raising overflow (OV) flag. This problem can be understood by Example 5.8.

Example 5.8

Illustrate the use of overflow flag using suitable example.

Solution:

Consider addition of two signed numbers +90H and +70H.

+90d	5AH	0101 1010 B
+70d	46H	0100 0110 B
+160d	A0H	1010 0000 B
		= A0H OV=1

A0 = –96 decimal

In this example, +90 is added to +70 and the result after addition is – 96 (assuming the signed numbers), which is incorrect, because the expected result is +160, which is larger than what an 8-bit number could represent or an 8-bit register could store (8-bit register could contain only up to +127). The 8051 indicates error by setting OV=1. It is up to the programmer to take care of erroneous result. The interpretation and recovery from an erroneous result is discussed in more detail in next section.

Let us understand signed arithmetic for positive (+ve) and negative (–ve) numbers for addition and subtraction. There are four types of operations in signed arithmetic as shown in Table 5.3.

Table 5.3 *Type of signed operations*

	Operation	Type of signed numbers	Remark
1	Addition	*Unlike* numbers	*Like* numbers means both numbers have same sign, *Unlike* numbers means opposite sign
2	Addition	*Like* numbers	
3	Subtraction	*Unlike* numbers	
4	Subtraction	*Like* numbers	

4. Addition of Unlike Signed Numbers

When *unlike* signed numbers are added then result will be always within a range of –128d to +127d, and the sign of the result will be always correct because the result (either positive or negative) will be smaller than larger of two original numbers. Addition of unlike signed numbers is illustrated in Examples 5.9 to 5.12.

Example 5.9

Add –02d with +30d.

Solution:

	(– 02d) =		1111 1110 B	= FEH
+	(+ 30d) =		0001 1110 B	= 1EH
	+28d	1)	0001 1100 B	1CH
			=+28d	

In the above addition, there is a carry out from bit 7 as well as from bit 6, therefore the overflow (OV) flag is 0, and the result is correct and for this condition, no action should be taken to correct the result.

> ***Note:*** Numbers without any suffix or with suffix 'D' are decimal numbers, and suffix 'B' and 'H' represents binary and hexadecimal numbers respectively.

Example 5.10

Add –128d with +127d.

Solution:

	(–128d) =	1000 0000B	=	80H
+	(+127d) =	0111 1111B	=	7FH
	–1d	1111 1111B		FFH
		= – 1d		

There is no carry out from bit 7 as well as bit 6, so OV=0 and C=0 and for this condition, no action should be taken to correct the sum because it is already correct.

Example 5.11

Add 0d with –1d.

Solution:

	(00d) =	0000 0000B	=	00H
+	(–1d) =	1111 1111B	=	FFH
	–1d	1111 1111B		FFH
		= –1d		

The OV=0, therefore the result is correct.

Example 5.12

Add +1d with –128d.

Solution:

	(+001d) =	0000 0001B	=	01H
+	(–128d) =	1000 0000B	=	80H
	–127d	1000 0001B		81H
		= –127d		

Again, it is seen that the result is correct.

In conclusion, when two unlike signed numbers are added, the result is always a correct signed number by neglecting the carry.

5. *Addition of Like Signed Numbers*

If two positive numbers are added, it is possible that sum may exceed +127d, i.e. overflow may occur. Addition of like signed numbers is illustrated in Examples 5.13 to 5.16.

Example 5.13

Add +40d with +70d.

Solution:

	(+40d) =	0010 1000B	=	28H
+	(+70d) =	0100 0110B	=	46H
	+110d	0110 1110B		6EH
		= +110d		

In the above case, there are no carry from bit 6 and 7 of sum. There, C=0 and OV=0 and the result is correct.

Example 5.14

Add +100d with +70d.

Solution:

	(+100d) =	0110 0100B	=	64H
+	(+070d) =	0100 0110B	=	46H
	+170d (expected)	1010 1010B		AAH
		= –86d (actual result)		

Here, the expected result is +170, which exceeds the maximum positive number (+127) that an 8-bit number can represent (or an 8-bit register can hold). So we get an incorrect result. This is indicated by OV=1, because there is carry out from bit 6 but not from bit 7.

The reason for getting an erroneous result in above example may also be understood this way: Since both the original numbers were positive and therefore the result should have been positive, but we got it as negative because of the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are three ways the result may be interpreted. Though these methods (described next) are unconventional, they would help in instigating a better understanding of the topic.

(a) As we have seen that the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result AAH (**1**010 1010 b). After inverting the sign bit the result will become 2AH (**0**010 1010 b) = +42d. This may be considered as the actual result is 42d higher (because of the + sign) than +128 (if we have some way of representing +128), i.e.

+128d
<u>+042d</u>
+170d

But, we cannot represent +128 using 8-bit number.
Considering it the other way, it may also be seen as +256+ (–86) = +170!

(b) If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,
   (i) If the result is to be used in further arithmetic operations, the number should be re-sized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example, carry is 0 and eight bit result is 10101010, the resultant 16 bits represents 00000000 10101010, which is +170 in 16-bit signed representation.
   (ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result, the carry may be used to determine sign of the expected result. In this example, carry is 0, therefore, the sign of expected result is positive and consider all bits of the result as magnitude only, i.e. 10101010=170, thus, combining the sign and the magnitude, the result would be +170.
   Or simply considering carry as a 9th bit, result would be correct, i.e. 010101010=+170.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get correct sign of the expected result. In the example, the sign bit is 1, after inverting it, we get it as 0 and therefore the sign of the expected result is positive. Now since the sign is positive, treat eight bits of the erroneous result as a magnitude, i.e. 10101010=170, thus, combining the sign and the magnitude, the result would be +170.

Example 5.15

Add –70d with –80d.

Solution:

(–70d) =	1011 1010B	= BAH
+ <u>(–80d)</u> =	<u>1011 0000B</u>	= <u>B0 H</u>
–150d (expected)	1)0110 1010B	6AH
	=+106d (actual result)	

Here, result exceeds –128d, so answer is incorrect. There is carry from bit position 7 and no carry from bit 6, so OV=1 and C=1.

The result is erroneous in above example because both the original numbers were negative and therefore the result should have been negative, but the result we got is positive because of the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are again three ways that the result may be interpreted.

(a) Since the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result 6AH (**0**110 1010 b). After inverting the sign bit, the result will become EAH (**1**110 1010 b) = –22d. This may be considered as the actual result; it is –22 lower (because of the – sign) than –128, i.e.

–128d
+ <u>–022d</u>
–150d

It may also be seen as –256+ (+106) = –150!

(b) If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,
  (i) If the result is to be used in further arithmetic operations, the number should be resized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example carry is 1 and eight bit result is 10101010, the resultant 16 bits represents 11111111 01101010, which is –150 in 16-bit signed representation.
  (ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result,
  The carry may be used to determine sign of the expected result. In this example, carry is 1, therefore the sign of expected result is negative and consider all bits of the result as magnitude after taking its 2's complement, i.e. 2's complement of 01101010 is 10010110 =150, thus, combining the sign and the magnitude, the result would be –150.
  Or simply 101101010= –150 if it is treated as 9- bit signed number.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get correct sign of the expected result. In the example, the sign bit is 0, after inverting it we get it as 1; therefore, the sign of the expected result is negative. Now since the sign is negative, find 2's complement of the erroneous result to get the magnitude, i.e. 2's complement of 01101010, which is 10010110=150, thus, combining the sign and the magnitude the result would be –150.

Example 5.16

Add –20d with –25d.

Solution:

(–20d)	=	1110 1100B	=	ECH
+ (–25d)	=	1110 0111B	=	E7H
–45d		1)1101 0011B		D3H
		= –45d		

Here, C=1 and OV=0. The result is correct.

In conclusion, when two like signed numbers are added, and after addition if OV=0, the result is always a correct signed number by neglecting the carry, otherwise (if OV=1) the result is incorrect.

6. *Subtraction of* Unlike *Signed Numbers*

If two unlike numbers are subtracted, it is possible that result may exceed range of –128d to +127d. The situation is similar to adding the *like* numbers. Subtraction of unlike signed numbers is illustrated in Examples 5.17 and 5.18.

Example 5.17

Subtract +100d from –70d.

Solution:

(–070d)	=	1011 1010B	=	BAH
– (+100d)	=	– 0110 0100B	=	64H
– 170d (expected)		0) 0101 0110B		56H
		= +86d (actual result)		

There is borrow into the bit portion of 6 but not into bit7. OV=1 and C=0. Because OV=1, the result is incorrect.

The result is erroneous in above example because we subtract larger number from smaller one, the result should have been negative, but the result we got is positive because the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are again three ways that the result may be interpreted.

(a) As we have seen that the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result 56H (**0**101 0110 b). After inverting the sign bit, the result will become D6H (**1**101 0110 b) = –42d. This may be considered as the actual result; it is –42 lower (because of the – sign) than –128, i.e.

	–128d
+	–042d
	–170d

It may also be seen as –256+ (+86) = –170!

(b) In a microcontroller, the subtraction is actually performed by adding 2's complement of the subtrahend to the minuend. Because of this, our example may be represented as follows:

(–070d)	=	1011 1010B	= BAH
+ (–100d)	=	+ 1001 1100B	= 9CH
–170d(expected)		1) 0101 0110B	56H
		= +86d (actual result)	

If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,

(i) If the result is to be used in further arithmetic operations, the number should be resized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example, carry is 1 and eight bit result is 01010110, the resultant 16 bits represents 11111111 01010110, which is –170 in 16-bit signed representation.

(ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result, the carry may be used to determine sign of the expected result. In this example, carry is 1, therefore the sign of expected result is negative and consider all bits of the result as magnitude after taking its 2's complement, i.e. 2's complement of 01010110 is 10101010 =170, thus, combining the sign and the magnitude the result would be –170.

Or simply, 101010110 = –170 if it is treated as 9-bit signed number.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get the correct sign of the expected result. In the example, the sign bit is 0, after inverting it we get it as 1; therefore, the sign of the expected result is negative. Now since the sign is negative, find 2's complement of the erroneous result to get the magnitude, i.e. 2's complement of 01010110, which is 10101010=170, thus, combining the sign and the magnitude the result would be –170.

Example 5.18

Subtract –50 from +80.

Solution:

(+80d)	=	0101 0000	=50H
– (–50d)	=	1100 1110	=CEH
+130d (expected)		1) 1000 0010	82H
		= –126d (actual result)	

There is a borrow into bit 7 and no borrow into bit 6. Therefore OV=1 and C=1.The result may be interrupted in a ways similar to above examples.

7. *Subtraction of* Like *Signed Numbers*

The situation here is similar to adding *unlike* numbers. The result will be always correct, i.e. always within the range –128 to +127. So magnitude and sign of the result need not be adjusted.

Subtraction of like signed numbers is illustrated in Examples 5.19 and 5.20.

Example 5.19

Subtract +120d from +101d.

Solution:

(+101d)	=	0110 0101B	= 65H
– (+120d)	=	0111 1000B	= 78H
–19d		1)1110 1101B	EDH
		= –19d	

Here OV=0 and C=1, neglecting carry, the result is correct.

Example 5.20

Subtract –116d from –61H.

Solution:

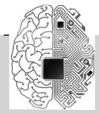
```
(–061d) =      1100 0011B   = C3H
– (–116d) =    1000 1100B   = 8CH
  +55d       0)0011 0111B     37H
                  =+55d
```

Here, OV=0 and C=0. The result is correct.
In conclusion, when two like signed numbers are subtracted, the result is always a correct signed number by neglecting the carry.

From the above discussion, we can make general note that if OV flag is set to 1, the result is incorrect as it is outside the range –128d to +127d.The OV=1 also suggests to complement the sign bit of the result to interpret the result.

8. Recovering a Result from Overflow

If we want to recover the result as well as use it in further arithmetic, the '(i)' part of the second method (to interpret the result) discussed above may be implemented easily in a program because its logic is same for all types of operations. However, if we want only to recover the result, i.e. if the result obtained is the end result and it is not to be used in further arithmetic, the '(ii)' part of the second method '(b)' may be used to implement the program.



THINK BOX 5.2

Show how the erroneous result obtained in signed arithmetic can be recovered and displayed in decimal on the display device (like LCD or monitor).

Overflow flag indicate the erroneous result, i.e. if OV=1, perform the following steps to recover a result.
Use the second method '(b)', ('(ii)' part of that) to interpret the result and write a program as follows:

- If carry is 0, send ASCII of '+' to the display device, treat the 8 bits of the original result as magnitude, convert it to BCD, then into ASCII.
- If carry is 1, send ASCII of '–' to the display device, find 2's complement of the original result and treat it as magnitude.

To avoid an overflow, a programmer has to predict the largest possible result and choose the size of the numbers accordingly, i.e. to support the larger numbers in signed operations one should go for 16-bit signed numbers (or even multi-byte signed numbers as per requirements).

THINK BOX 5.3

Why do we use same logic for addition (or subtraction) of unsigned and signed (2's complement) binary numbers?
Because rules for binary addition (or subtraction) remain same for all type of representations of a binary number. Computers understand only 0s and 1s. It is how we interpret the binary number that makes the difference in the meaning conveyed by them. For example, 1101 represent 11D for unsigned representation and –5D for signed representation. Following example shows how same binary numbers are interpreted in different ways for signed and unsigned representations.

```
  0110  (=6)      0110 (=  6)
+ 1011 (=11)   + 1011 (= –5)
1 0001 (=17)    1 0001 (=  1)
```

THINK BOX 5.4

Can we perform the subtraction operation without SUBB instruction? If yes, How?
Yes. Find two's complement of subtrahend and add it to minuend.

THINK BOX 5.5

How is multi-byte signed arithmetic performed?
Signed numbers are similar to the unsigned numbers except for the most significant bit of a most significant byte; therefore, lower bytes are treated as an unsigned number and overflow is checked only for most significant byte.

5.1.4 Decimal (Binary Coded Decimal—BCD) Arithmetic

Binary representation of decimal digits (0 to 9) is called binary coded decimal. BCD numbers are needed because humans prefer to use decimal number system (digits 0 to 9). Four bits are required to represent the decimal number from 0 to 9 as shown in Table 5.4.

Table 5.4 *BCD codes for decimal digits 0 to 9*

Decimal Digit	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The terms *unpacked* and *packed BCD numbers* are widely used with the BCD numbers. In *unpacked BCD numbers,* the lower four bits of the number are used to represent the decimal digit and the upper four bits are 0. In *packed BCD numbers* two BCD numbers are placed (packed) into a single byte.

Examples of unpacked and packed BCD numbers are shown below.

```
0000 0001 ⟶ unpacked BCD for 1
0000 0101 ⟶ unpacked BCD for 5
   ↓    └──⟶ BCD code of number
Upper nibble
  is zero

0110 1000    packed BCD for 68
1000 0010    packed BCD for 82
  ↓    ↓
Both nibbles are used
```

The 8051 performs all the operations in pure binary. When BCD numbers are used in arithmetic, the result may be non-BCD as shown in Example 5.21.

Example 5.21

Explain the process of BCD addition.

Solution:

Consider addition of two BCD numbers 38 and 24.

```
  38 d     0011 1000 BCD     38H
+ 24 d     0010 0100 BCD     24H
  62 d     0101 1100         5CH
```

In the above addition, when two BCD numbers are added (two operands are assumed to be BCD), the result is 5C, which is non-BCD number because, in the BCD numbers we can use only 0 to 9 (0000 to 1001). The correct result should have been 62d.To correct the result, programmer must adjust result by adding 6 (0110) to the lower digit, i.e.

```
  5C
 +6
  62
```

A similar problem could have occurred in the upper digit of the result while performing addition. For example,

```
  72 d     0111 0010 BCD     72H
+ 41 d     0100 0001 BCD     41H
 113 d     1011 0011 BCD     B3H
```

Again, to get the correct result (72+41=113), we should add 6 to the upper digit, i.e.

```
  B3
+ 6
 113
```

This problem in above example is common while working with the BCD numbers. Therefore, the 8051 has instruction to adjust the BCD result automatically. Instruction DA A is there to correct the BCD addition problem discussed above.

> ***Note:*** Microcontrollers understand only binary numbers, they cannot differentiate between binary and BCD numbers, and it is the programmer who assumes the number to be binary or BCD and treats them accordingly.

DA A Decimal Adjust Accumulator for Addition

DA A works only on the contents of register A, it must be kept in mind that DA A must be used after the addition of BCD numbers and BCD numbers can never have any digit greater than 9. In other words, A to F digits are not allowed. The DA A instruction works only when used after ADD or ADDC instructions and does not give correct answer after SUBB, MUL, DIV or INC operations.

Operation of DA A DA A instruction performs the following operations. After ADD or ADDC instruction,

1. If lower nibble is greater than 9 or if AC=1, add 6 to lower nibble (4 bits)
2. If upper nibble is greater than 9 or if CY=1, add 6 to upper nibble.

The operation of DA A instruction for above-mentioned different conditions is illustrated in Example 5.22.

Example 5.22

Explain with suitable example the operations performed by DA A instruction when
(i) Lower nibble of A is greater than 9 after addition
(ii) AC=1 after addition
(iii) Upper nibble of A is greater than 9 after addition
(iv) C=1 after addition

Solution:

(i) When the lower nibble is greater than 9 after addition,

28 BCD	0010 1000	
+ 12 BCD	0001 0010	
40 BCD	0011 **1010**	3**A**; lower nibble is greater than 9 (invalid BCD)
	+ 0000 0110	add 6 to lower nibble (DA A will do it)
	0100 0000	40 BCD, the desired result

(ii) When AC=1 after addition,

28 BCD	0010 1000	
+ 19 BCD	0001 1010	
47 BCD	0100 0001	41; AC is 1 after addition (invalid result)
	+ 0000 0110	add 6 to lower nibble (DA A will do it)
	0100 0111	47 BCD, the desired result

(iii) When upper nibble is greater than 9,

82 BCD	1000 0010	
+ 21 BCD	0010 0001	
103 BCD	1010 0011	**A**3; upper nibble greater than 9 (invalid BCD)
	+ 0110 0000	add 6 to upper nibble (DA A will do it)
	1 0000 0011	103 BCD, the desired result

(iv) When CY flag is set after addition,

82 BCD	1000 0010	
+ 91 BCD	1001 0001	
173 BCD	**1** 0001 0011	carry flag set after addition (Invalid result)
	+ 0110 0000	add 6 to upper nibble (DA A will do it)
	1 0111 0011	173 BCD, the desired result

It is also possible to have both nibbles as non-BCD after addition as shown.

63 BCD	0110 0011	
+ 88 BCD	1000 1000	
151 BCD	1110 1011	**EB**; both nibbles greater than 9 (invalid BCD)
	+ 0110 0110	add 6 to both nibbles DA A will do it)
	1 0101 0001	151 BCD, the desired result

The AC flag is useful for DA A instruction only. It has no other use to the programmer.

Example 5.23

Illustrate the use of DA A instruction with a suitable example.

Solution:

Consider addition of two BCD numbers 28 and 12.

```
MOV A, #28H          // add 28 and 12
ADD A, #12H          // A= 3AH; non BCD
DA A                 // adjust the result in A to BCD
                     // A= 40
```

Note that the two 8-bit numbers added are assumed to be BCD. The result after addition (3AH) is non-BCD; therefore, to get correct BCD result, the DA A instruction is used.

Example 5.24

Discuss the operation performed by DA A instruction for the following.

```
MOV A, #99H
ADD A, #01H
DA A
```

Solution:

The DA A instruction will add 66H to the result after addition, i.e. 9AH+66H=100H

5.1.5 Multiplication

The 8051 supports 8-bit integer multiplication. Multiplication instruction uses only A and B registers as both source and destination for the operation. The bytes in A and B are assumed to be unsigned. Format of multiply instruction is as follows:

```
MUL AB          // multiply contents A with B; put the lower byte of result in
                // A and higher byte of result in B.
```

Use of OV in Multiplication

The overflow will be set t to 1, if A × B > FFH. **Here, the OV flag does not indicate that an error has occurred**. But, it shows that the result is larger than 8 bits and we need to consider B register for higher byte of the result. Example 5.25 shows the use of multiply instruction.

Discussion question Which flags are affected by the multiply instruction?

Answer The multiply instruction always clears the carry flag to zero and it will set the overflow flag if the result after multiply instruction is greater than 0FFH, else it will clear the overflow flag.

Example 5.25

Multiply the contents of R0 and R1 and place the result into R2 (MSByte) and R3 (LSByte).

Solution:

Let us consider multiplication of largest 8-bit numbers, i.e. assume that R0=FFH, R1=FFH

```
MOV A, R0       // A= FF
MOV B, R1       // B= FF, operands can only be in A and B
MUL AB          // A x B=FE01, B=FE, A=01, OV=1 to indicate that the result is greater than 8 bits
MOV R3, A       // store result LSB into R3
MOV R2, B       // store result MSB into R2
```

Repeat the above program for R0=05H, R1=04H

```
MOV A, R0       // A= 05
MOV B, R1       // B= 04
MUL AB          // A x B=0014, B=00, A=14, OV=0 to indicate that result is only 8 bits
MOV R3, A       // store result LSB into R3
MOV R2, B       // store result MSB into R2; may be ignored for this case
```

Example 5.26

Write a program to find the square of a number stored at internal RAM address 50H. Store the result at address 60H (LSByte) and 61H (MSByte). If the number is AAH, what will be the result and status of OV flag after finding the square of that number?

Solution:

The square of a number is found by multiplying the number with itself. In the 8051, multiply instruction require operands in A and B only; therefore we need to copy the number into A and B registers and then we will use the multiply instruction.

```
MOV A, 50H      // copy the number at address 50H into A
MOV B, A        // copy the same number into B
MUL AB          // find the square by multiplication
MOV 60H, A      // copy the result (LSByte) into address 60H
MOV 61H, B      // copy the result (MSByte) into address 61H
```

If the number is AA, then result will be 70E4H. Since result is greater than FFH the overflow flag will be set, i.e. OV=1 after multiplication.

5.1.6 Division

The 8051 supports 8-bit integer division. The divide instruction uses only A and B registers as both source and destination for the operation. The number in A is divided by number in B. Quotient (result) is placed in A and remainder is placed in B, again both numbers are assumed to be unsigned. Format of divide instruction is as follows,

```
DIV AB      // divide A by B, store the result in A and remainder in B
```

Use of OV in Division The OV flag is set to 1 to show that an attempt to divide by zero has been made, i.e. contents of B=00 before division. The contents of A and B are undefined when division by 0 is attempted.

Discussion Question Which flags are affected by the divide instruction?

Answer The divide instruction always clears the carry flag to zero and it will set the overflow flag if a divisor is 00H, else it will clear the overflow flag.

Example 5.27

Write instructions to divide 95 by 10.

Solution:

```
MOV A, #95      // A=95d = 5FH, number without any suffix is decimal number
MOV B, #10      // B=10d =0AH
DIV AB          // A/B, result is A=9 (quotient), B=5(remainder)
```

> ***Note:*** The original contents of A and B are lost in both multiply and divide instructions.

Example 5.28

Show the status of OV flag and contents of A and B after execution of the following instructions.

```
MOV A, #31H
MOV B, #03H
DIV AB
```

Solution:

The operation performed by the above program is $\frac{31H}{03H} = \frac{49d}{03d}$. Therefore, the result will be, A= 10H (Quotient) and B= 01H (Remainder). The OV=0, because no attempt has been made to divide by zero.

5.1.7 Increment and Decrement

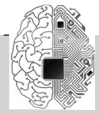
These are the most simple and special instructions for addition and subtraction. They are used to add 1 or subtract 1 from a specified operand. All addressing modes are supported. No flags are affected. These operations provide powerful way to repeat the operations when used with program flow control instructions to, i.e. increment or decrement until the desired result is obtained. The format of these instructions is as follows,

```
INC destination     // add one to the destination operand
DEC destination     // subtract one form the destination operand
```

The formats of increment and decrement instructions for different addressing modes are as follows:

INC A	// Increment the contents of A by 1, A=A+1
INC A	// A=A+1, If A=10H → A=11H
INC *Rn*	// Increment the contents of Rn by 1, *Rn*= *Rn*+1
INC R3	// R3= R3+1, If R3=1AH → R3=1BH
INC *@Ri*	// Increment the contents of address pointed by *Ri* by 1, (*Ri*)= (*Ri*) +1
INC @R0	// (R0)= (R0) +1, If R0=10H, (10H)=55H → (R0)=(10H)=56H
INC *direct*	// Increment the contents of direct address by 1, (direct)=(direct) +1
INC 40H	// (40H)=(40H)+1, If (40H)=1FH → (40H)=20H
INC DPTR	// Increment the contents of DPTR by 1, DPTR=DPTR+1
INC DPTR	// DPTR=DPTR+1, If DPTR=1000H → DPTR=1001H
DEC A	// Decrement the contents of A by 1, A=A–1
DEC A	// A=A–1, If A=10H → A=0FH
DEC *Rn*	// Decrement the contents of Rn by 1, *Rn*= *Rn*–1
DEC R3	// R3= R3–1, If R3=1AH → R3=19H
DEC *@Ri*	// Decrement the contents of address pointed by *Ri* by 1,(*Ri*)= (*Ri*) –1
DEC @R0	// (R0)= (R0) –1, If R0=10H, (10H)=55H → (R0)=(10H)=54H
DEC *direct*	// Decrement the contents of *direct* address by 1 (direct)=(direct) –1
DEC 40H	// (40H)=(40H) –1, If (40H)=1FH → (40H)=1EH

It should be noted that there is no DEC DPTR instruction.



THINK BOX 5.6

What is the difference between following two instructions?

- **INC A**
- **INC ACC**

Both instructions will do the same operation. But, INC A instruction is assembled as a 04 (1-byte instruction) and INC ACC is assembled as 05 E0 (2-byte instruction, direct addressing mode '**I**NC *direct*')

Example 5.29

Illustrate how increment and decrement instructions modify the operands.

Solution:

The following instructions illustrate how increment and decrement instructions modify the operands.

```
MOV A, #0FFH        // A= FFH
MOV R5, #10H        // R5= 10H
MOV R4, #10         // R4= 10d= 0AH
MOV R0, # 20H       // R0= 20H
MOV 20H, A          // (20H)= FFH
INC A               // A= FFH +1=00
DEC R5              // R5= 0FH
DEC R4              // R4= 09
DEC @ R0            // (20H)= FEH
INC 20H             // (20H)= FFH
INC 20H             // (20H)= 00H
DEC 20H             // (20H)=FFH
```

Observe that, when data FF is incremented by 1, it results into 00, i.e. 8-bit data overflow from FF to 00, it is modulo 8-bit operation, But remember that it (INC and DEC) does not affect any flag.

Example 5.30

Find the contents of the operands after execution of each of the following instructions.

```
MOV DPTR, #0FFFFH
DEC DPL
INC DPH
INC DPTR
DEC DPH
INC DPTR
```

Solution:

```
MOV DPTR, #0FFFFH      // DPTR=FFFFH
DEC DPL                // DPL=FEH
INC DPH                // DPH= 00H
INC DPTR               // DPTR=00FFH
DEC DPH                // DPH=FFH
INC DPTR               // DPTR=0000H
```

Note that DPTR overflow from FFFF to 0000.

Example 5.31

Find the contents of the destination operand after execution of each of the following instructions.

```
MOV R5, #10H
INC R5
INC R5
MOV R0, #20H
MOV A, #0FFH
MOV 20H, A
INC @ R0
INC A
MOV 20H, #00H
INC 20H
```

Solution:

```
MOV R5, #10H       // R5= 10H
INC R5             // R5= 11H
INC R5             // R5= 12H
MOV R0, #20H       // R0= 20H
MOV A, #0FFH       // A= FFH
MOV 20H, A         // (20H) = FFH
INC @ R0           // (20H) = 00H
INC A              // A=00H
MOV 20H, #00H      // (20H) = 00H
INC 20H            // (20H) =01H
```

Example 5.32

What should be the initial value of A if the value of A after execution of the following instructions is 00H?

```
MOV R0, A
SUBB A, R0
INC A
```

Solution:

A should have any value between 00H-FFH and CY=1.

To have value 00 after INC A, the value of A before execution of INC A (or after SUBB A, R0) should be FFH. Now, since values of A and R0 are same because of the first instruction, the carry (borrow) flag should be 1 to have value of A=FF after execution of SUBB instruction.

5.2 LOGICAL INSTRUCTIONS

Logic means evaluating the conditions using reasonable thinking and making a decision based on the evaluation. Binary logic deals with data that takes only two discrete values as being true or false (or, yes or no) and operations on such data. There are three basic logical operations: AND, OR and NOT. Many other logical operations are derived from these basic operations.

The 8051 supports logical operations such as AND, OR, EX-OR and NOT. These operations can be performed on a byte or on a single bit at a time. Single-bit operations are useful mostly in machine control applications where we need to monitor and control binary events such as to turn ON or OFF a device, to read status of an input switch.

5.2.1 Byte Operations

The operation specified in an instruction is performed on all 8 bits in a byte. The general format for these instructions are as follows:

ANL *destination, source*	// *destination* =*destination* AND *source*
ORL *destination, source*	// *destination* =*destination* OR *source*
XRL *destination, source*	// *destination* =*destination* EX-OR *source*

Note that these are all bit-wise operations, i.e.

destination bit *Dn* = destination bit *Dn* **OPERATION** source bit *Dn*

All four addressing modes can be used for the *source* operand. The A or a *direct* address in internal RAM can be the *destination*. For example,

1. AND Operation

The format of AND instruction for different addressing modes is given below:

ANL A, *#data*	// bitwise AND operation of A with *data*, A=A AND *data*
ANL A, #10H	// If A= FFH, → A= FFH AND 10H= 10H
ANL A, *Rn*	// bitwise AND operation of A with Rn, A=A AND *Rn*
ANL A, R0	// If A=55H, R0= 4AH, → A=55H AND 4AH= 40H
ANL A, *@Ri*	// bitwise AND operation of A with data pointed by Ri, A=A AND *(Ri)*
ANL A, @R1	// If A= 38H, R1=20H, (20H) =1FH, → A= 38H AND 1FH= 18H
ANL A, *direct*	// bitwise AND operation of A with data in *direct*, A=A AND (*direct*)
ANL A, 20H	// If A= E6H, (20H)=67H, → A=E6H AND 67H= 66H
ANL *direct, #data*	// bitwise AND operation of data in address *direct* with immediate *data* *// (direct) = (direct)* AND *data* (store result in address *direct*)
ANL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH AND E6H= 06H
ANL *direct*, A	// bitwise AND operation of data in address *direct* with A *(direct) = (direct)* AND A
ANL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H AND 0FH= 02H

Example 5.33

Illustrate the use of ANL instruction.

Solution:

```
MOV A, #45H      A= 45H
ANL A, #0EH      A= 45H AND 0EH =04H
```

The operation of ANL instruction is explained below:

```
AND  45 H   01000101
     0EH    00001110
     04H    00000100
```

The AND operation (ANL instruction) is commonly used to mask (set to 0) certain bits of a result as shown in above example (see highlighted bits)

2. OR Operation

The format of OR instruction for different addressing modes is given below,

ORL A, *#data*	// bitwise OR operation of A with *data*, A=A OR *data*
ORL A, #10H	// If A= FFH, → A= FFH OR 10H= FFH
ORL A, *Rn*	// bitwise OR operation of A with *Rn*, A=A OR *Rn*
ORL A, R0	// If A = 55H, R0= 8AH, → A=55H OR 8AH= DFH
ORL A, *@Ri*	// bitwise OR operation of A with data pointed by *Ri*, A=A OR *(Ri)*
ORL A, @R1	// If A= 38H, R1=20H, → (20H) =1FH, A= 38H OR 1FH= 3FH
ORL A, *direct*	// bitwise OR operation of A with data in *direct*, A=A OR (*direct*)
ORL A, 20H	// If A= E6H, (20H) =67H, → A=E6H OR 67H= E7H
ORL *direct, #data*	// bitwise OR operation of data in address *direct* with immediate *data* *// (direct) = (direct)* OR *data* (store result in address *direct*)
ORL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH OR E6H= EFH
ORL *direct*, A	// bitwise OR operation of data in address *direct* with A *// (direct) = (direct)* OR A
ORL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H OR 0FH= 7FH

Example 5.34

Illustrate the use of ORL instructions.

Solution:

```
MOV A, #34H     A= 34H
ORL A, #57H     A= 34H OR 57H =77H
    OR  34 H    00110100
        57 H    01010111
        77 H    01110111
```

The OR operation (ORL instruction) is commonly used to set certain bits of a result to 1 as shown in above example (see highlighted bits).

3. EX-OR Operation

The format of EX-OR instruction for different addressing modes is given below:

XRL A, *#data*	// bitwise X-OR operation of A with *data*, A=A X-OR *data*
XRL A, #10H	// If A= FFH, → A= FFH X-OR 10H= EFH
XRL A, *Rn*	// bitwise X-OR operation of A with Rn, A=A X-OR *Rn*
XRL A, R0	// If A = 55H, R0= 8AH, → A=55H X-OR 4AH= DFH
XRL A, *@Ri*	// bitwise X-OR op. of A with data pointed by Ri, A=A X-OR *(Ri)*
XRL A, @R1	// If A= 38H, R1=20H, (20H) =1FH, → A= 38H X-OR 1FH= 27H
XRL A, *direct*	// bitwise X-OR operation of A with data in *direct*, A=A X-OR (*direct*)
XRL A, 20H	// If A= E6H, (20H) =67H, → A=E6H X-OR 67H= 81H
XRL *direct, #data*	// bitwise X-OR operation of data in address *direct* with *data* *// (direct) = (direct)* X-OR *data* (store result in address *direct*)

XRL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH X-OR E6H= E9H
XRL *direct*, A	// bitwise X-OR operation of data in address *direct* with A // *(direct)* = *(direct)* X-OR A
XRL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H X-OR 0FH= 7DH

Example 5.35

Illustrate the use of XRL instructions.

Solution:

```
MOV A, #0A4H        A= A4H
XRL A, #71H         A= A4H EX-OR 71H =D5H

EX-OR  A4 H         1010 0100
       71 H         0111 0001
       D5 H         1101 0101
```

The EX-OR operation (XRL instruction) is often used to invert certain bits of an operand, i.e. if any bit is EX-ORed with 1, it will be inverted (see highlighted bits). EX-OR operation may also be used to see if two registers (or two bits) have the same value. If two bits of same value are EX-ORed, the result will be always zero. We can use this result along with decision making instructions to take appropriate action.

No flags are affected by byte level logical instructions, expect when destination operand is PSW (direct address).

Example 5.36

Write single instruction for each of the following operat

(i) Clear bits 0,2,3,6 of the A

(ii) Set bits 0,1,5 of the contents of the address 20H

(iii) Complement bits 2,3,4,7 of the A

Make sure that other bits are not disturbed.

Solution:

(i) ANL A, #0B2H
(ii) ORL 20H, #23H
(iii) XRL A, #9CH

4. Logical Operations with Ports

When destination of the logical instruction is port SFR, the latch register (port structure will be discussed in Chapter 13) will be used as both source of data and destination to store the result. In such instructions, the port pins are not read. Consider the following instruction,

```
MOV P1, #0FFH     // port 1 latch =FFH
```

Assume that port P1 pins are connected to base of transistors (each pin with the different transistor) and the above instruction is executed. Since latch contains all 1's, the transistors will be ON and bases of all the transistors will be near to ground level (0.7 volts), therefore P0 pins will be at low (0) level, i.e. port pins will be at logic level 0 even though port latch is at level 1. Now consider instruction,

```
ANL P1, #0F0H
```

In above instruction P0 is initially source of data, so latch of P1 (FFH) is read, and then it will be logically ANDed with immediate value F0H and then result (F0H) will be written back to P0 latch register. For above instruction, if pins were read then result would have been 00 (00 AND 0F) which is incorrect. This issue is discussed in more detail in Section 13.2.

When an instruction uses port as a source, but not as a destination, microcontroller reads port pins (as the source of data) instead of port SFR. For example,

```
ANL A, P1
```

This instruction will AND A with contents of P1 pin (00) for above example, result will be 00 in the Accumulator.

All logical operations discussed till now had two operands, and that's why they are referred as binary operations. There are few instructions which require only one operand. They are referred as *unary operations*.

5.2.2 Unary Operations

Unary operations require single operand thus the operations are performed on a single operand. The source as well as destination for these operations is Accumulator.

1. Clear: The clear instruction is used to clear the contents of Accumulator, the format of this instruction is

```
CLR A          // clear accumulator, A=00H
```

2. Complement (NOT operation): Generally, complement is used to generate 1's complement of the data in an accumulator, i.e. All 0s will be replaced by 1 and 1s will be replaced by 0.

```
CPL A          // complement A, A= Ā
```

Example 5.37

What will be contents of A, after execution of following instructions?

```
MOV A, # 54H
CPL A
```

Solution:

These instructions will find complement of the number 54H.

```
MOV A, # 54H          // A= 54H = 0101 0100 b
CPL A                 // A= ABH = 1010 1011 b
```

The CPL A instruction will complement the contents of A. Therefore, A= ABH after execution of given instructions.

3. Rotate: Rotate operations are useful for monitoring bits of a data byte without using a logical test. The status of bits may be used in decision-making process for certain applications. The rotation can be 1 bit in left or right direction, with or without including carry flag in the rotation. The rotate instruction works only with the Accumulator. Total of nine bits are involved in the rotation operation when carry flag is included and eight bits when carry is not included. Usually, a carry flag is included in an operation when decision making is required because JC (jump if carry) and JNC (jump if no carry) are the instructions which makes a decision based on value of the carry flag. It is also used to perform rotation on multi-byte data. Rotate instructions may also be used to convert parallel data to serial data. The 8051 has four different rotate instructions as described in the following section.

(a) Rotate Accumulator Left by One Bit

```
RL A      // rotate A one bit position to the left, bit D0 to D1, bit
          D1 to D2, …, bit D6 to D7 and bit D7 to D0 as
          illustrated in Figure 5.3.
```

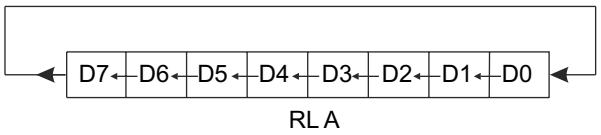


Fig. 5.3 *RL A instruction*

For example,

```
MOV A, #43H       // A= 0100 0011
RL A              // A=1000 0110
RL A              // A=0000 1101
```

(b) Rotate Accumulator Right by One Bit

```
RR A      // rotate A one bit position to the right, bit D0 to D7,
          bit D7 to D6, …, bit D2 to D1 and bit D1 to D0 as
          illustrated in Figure 5.4.
```

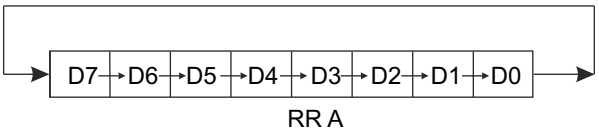


Fig. 5.4 *RR A instruction*

For example,

```
MOV A, #35H       // A= 0011 0101
RR A              // A= 1001 1010
```

(c) Rotate Accumulator Left through Carry by One Bit

```
RLC A     // rotate A one bit position to the left through carry flag, bit D0 to D1, bit D1 to D2, …, bit D6 to D7,
          // bit D7 to CY and CY to D0 as illustrated in Figure 5.5.
```

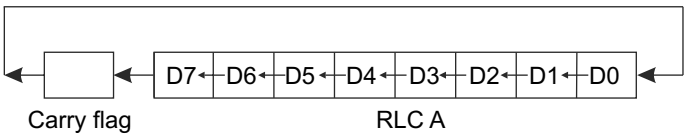


Fig. 5.5 *RLC A instruction*

For example,

```
CLR C              // CY=0
MOV A, #0D6H       // A= 1101 0110, CY=0
RLC A              // A= 1010 1100, CY=1
```

(d) Rotate Accumulator right through Carry by One Bit

RRC A // rotate A one bit position to the right through carry flag, bit D0 to CY, CY to D7, bit D7 to D6, …, bit // D2 to D1 and bit D1 to D0 as illustrated in Figure 5.6.

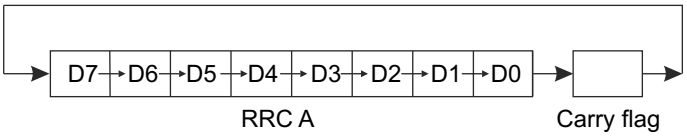


Fig. 5.6 *RRC A instruction*

For example,

```
SETB C             // CY=1
MOV A, #62H        // A= 0110 0010, CY=1
RRC A              // A= 1011 0001, CY=0
```

4. **SWAP:** SWAP A instruction swaps the nibbles of register A, i.e. it interchanges the upper nibble with the lower nibble of A. This operation is equivalent to 4-bit rotation in either left or right direction. It works only with the A register. The operation of swap instruction is illustrated in Figure 5.7.

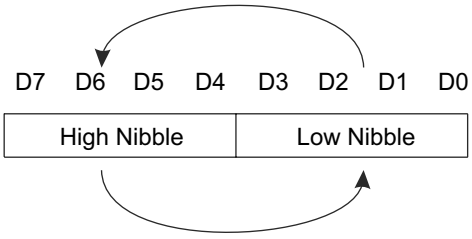


Fig. 5.7 *SWAP instruction*

For example,

```
A=53H (before execution)
SWAP A
A=35H (After execution)
MOV A, #53H      //A=53H
SWAP A           // A=35H
```

THINK BOX 5.7

Realize 'SWAP A' instruction using rotate instructions.
SWAP A can be realized by four RL A (or RR A) instructions.

Example 5.38

If A= 35H and C=1 before execution of the following instructions, write contents of destination operand and C after execution of each instruction.

```
RR A
RR A
RLC A
SWAP A
CPL C
RL A
SWAP A
RRC A
```

Solution:

Initially, A=35H=00110101b, C=1

```
RR A               // A=10011010b, C=1
RR A               // A=01001101b, C=1
```

```
RLC A          // A=10011011b, C=0
SWAP A         // A=10111001b, C=0
CPL C          // A=10111001b, C=1
RL A           // A=01110011b, C=1
SWAP A         // A=00110111b, C=1
RRC A          // A=10011011b, C=1
```

3. Summary of Arithmetic and Logical Instructions

Arithmetic and logical instructions are summarized in Tables 5.5 and 5.6 respectively.

Table 5.5 *Arithmetic instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
ADD A, **<BYTE>**	A= A + **<BYTE>**	ADD A, **direct**	ADD A,**@Ri**	ADD A,**Rn**	ADD A,**#data**
		ADD A, **12H**	ADD A,**@R1**	ADD A,**R4**	ADD A,**#09H**
ADDC A, **<BYTE>**	A= A + **<BYTE>** + C	ADDC A, **direct**	ADDC A,**@Ri**	ADDC A,**Rn**	ADDC A,**#data**
		ADDC A, **12H**	ADDC A,**@R1**	ADDC A,**R4**	ADDC A,**#10H**
SUBB A, **<BYTE>**	A= A – **<BYTE>** – C	SUBB A, **direct**	SUBB A,**@Ri**	SUBB A,**Rn**	SUBB A,**#data**
		SUBB A, **12H**	SUBB A,**@R0**	SUBB A,**R7**	SUBB A,**#25H**
INC A	A= A + 1	Accumulator only			
INC <BYTE>	<BYTE>= <BYTE> + 1	INC **direct**	INC **@Ri**	INC **Rn**	
		INC **12H**	INC **@R1**	INC **R6**	
INC DPTR	DPTR= DPTR + 1	Data Pointer only			
DEC A	A= A – 1	A only			
DEC <BYTE>	<BYTE>= <BYTE> – 1	DEC **direct**	DEC **@Ri**	DEC **Rn**	
		DEC **35H**	DEC **@R0**	DEC **R3**	
MUL AB	B:A= B*A	A & B only			
DIV AB	A= Int [A/B]; B= Mod [A/B]	A & B only			
DA A	Decimal Adjust	A only			

Table 5.6 *Logical instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
ANL A, <BYTE>	A= A AND **<BYTE>**	ANL A, **direct**	ANL A,**@Ri**	ANL A,**Rn**	ANL A,**#data**
		ANL A, **12H**	ANL A,**@R1**	ANL A,**R0**	ANL A,**#10H**
ANL <BYTE>, A	<BYTE>= <BYTE> AND **A**	ANL direct**, A**			
		ANL 10H, **A**			
ANL <BYTE>, #data	<BYTE>= <BYTE> AND **#data**	ANL direct**, #data**			
		ANL 10H, #**20H**			
ORL A, <BYTE>	A= A OR **<BYTE>**	ORL A, **direct**	ORL A,**@Ri**	ORL A,**Rn**	ORL A,**#data**
		ORL A, **12H**	ORL A,**@R0**	ORL A,**R2**	ORL A,**#10H**
ORL <BYTE>, A	<BYTE>= <BYTE> OR **A**	ORL direct, **A**			
		ORL 10H, **A**			
ORL <BYTE>, #data	<BYTE>= <BYTE> OR **#data**	ORL direct, **#data**			
		ORL 10H, #**20H**			

(*contd.*)

(**Table 5.6** *contd.*)

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
XRL A, <BYTE>	A= A XOR **<BYTE>**	XRL A, **direct**	XRL A,@**Ri**	XRL A,**Rn**	XRL A,**#data**
		XRL A, **12H**	XRL A,@**R0**	XRL A,**R2**	XRL A,**#25H**
XRL <BYTE>, A	<BYTE>= <BYTE> XOR **A**	XRL direct, **A**			
		XRL 10H, **A**			
XRL <BYTE>, #data	<BYTE>= <BYTE> XOR **#data**	XRL direct, **#data**			
		XRL 10H, #**20H**			
CLR A	A= 00H	Accumulator only			
CPL A	A= NOT A	Accumulator only			
RL A	Rotate A Left 1 bit	Accumulator only			
RLC A	Rotate A Left 1 bit through Carry	Accumulator only			
RR A	Rotate A Right 1 bit	Accumulator only			
RRC A	Rotate A Right 1 bit through Carry	Accumulator only			
SWAP A	Swap nibbles in A	Accumulator only			

POINTS TO REMEMBER

- The 8051 microcontroller supports basic arithmetic and logical operations such as addition, subtraction, division, multiplication, increment, decrement, AND, OR, NOT and EX-OR.
- The arithmetic operations modify arithmetic flags: carry (CY), overflow (OV) and auxiliary carry (AC).
- The register A is always destination as well as one of the source operand in the addition and subtraction operations. It is always one of the source and destination operand in a multiplication and division.
- Subtraction instruction treats the carry flag as borrow and always subtract carry flag as a part of an operation.
- Negative numbers are not represented in true binary form, but they are represented in 2' complement form.
- The CY and OV flags are there to handle unsigned and signed operations. CY is generally used in unsigned arithmetic while OV is used in signed addition and subtraction.
- Single byte sized signed number may range from –128d (1000 000b) to +127d (0111 1111b).
- In unpacked BCD numbers, the lower four bits of the number are used to represent the decimal digit and the upper four bits are 0. In packed BCD numbers, two BCD numbers are placed (packed) into single byte.
- Microcontrollers understand only binary numbers. They cannot differentiate between binary and BCD numbers; it is the programmer who assumes the number to be binary or BCD and treats them accordingly.
- AC flag is useful for DA A instruction only. It has no other use to the programmer.
- Increment and decrement instructions do not affect the flags.
- The 8051 support byte as well as bit level logical operations such as AND, OR, EX-OR and NOT.
- The AND operation is often used to mask (set to 0) certain bits of a result, the OR operation is often used to set certain bits of a result to 1, while the EX-OR operation is often used to invert certain bits of an operand.
- No flags are affected by byte level logical instructions, except when destination operand is PSW (direct address).
- When destination of the logical instruction is port SFR, the latch register will be used as both source of data and destination to store the result.
- When an instruction uses port as a source, but not as a destination, microcontroller reads port pins (as the source of data) instead of port SFR.
- Rotate operations are useful for monitoring bits of data byte without using a logical test. The status of bits may be used in decision making process for certain applications.

OBJECTIVE QUESTIONS

1. Addition instruction of the 8051 can affect,
   (a) carry flag (b) aux. carry flag (c) overflow flag (d) all of the above
2. The contents of the accumulator after execution of following instructions will be,
   MOV A, #0FH
   ANL A, #2CH
   (a) 11010111 (b) 11011010 (c) 00001100 (d) 00101000
3. Which of the following statements will add the accumulator with register R0?
   (a) ADD @R0, A (b) ADD A, @R0 (c) ADD R0, A (d) ADD A, R0
4. To mask LSB of the A, we must AND it with,
   (a) 7FH (b) 80H (c) FEH (d) FFH
5. To complement the A, we must EX-OR it with,
   (a) 7FH (b) 80H (c) FEH (d) FFH
6. To set MSB of the A, we must OR it with,
   (a) 00H (b) 01H (c) 80H (d) FFH
7. The contents of the accumulator after execution of following instructions will be,
   MOV A, #55H
   ORL A, 01H
   (a) 1B H (b) 55 H (c) 3B H (d) 4B H
8. The following command will rotate the 8 bits of the accumulator one position to the left
   RLC A
   (a) True (b) False
9. The following program will read data of port 1, determine whether bit 2 is high, and if so, send the number FFH to port 2,
   ```
   BACK:  MOV A, P1
          ANL A, #02H
          CJNE A, #02H, BACK
          MOV P2, #0FFH
   ```
   (a) True (b) False
10. DA A instruction adjusts the value in the accumulator resulting from an addition of two BCD numbers only.
    (a) True (b) False
11. For signed operations, -1 is represented in binary as,
    (a) 10000001 (b) 01111111 (c) 10000000 (d) 11111111
12. The contents of A and B after execution of following instructions will be,
    ```
    MOV A, #02H
    MOV B, #04H
    MUL AB
    ```
    (a) A=02H, B= 04H (b) A=08H, B= 04H (c) A=08H, B= 00H (d) A=00H, B= 08H
13. _____is used to indicate error in signed arithmetic operations.
    (a) AC flag (b) OV flag (c) CY flag (d) P flag
14. _____is used to indicate error in unsigned arithmetic operations.
    (a) AC flag (b) OV flag (c) CY flag (d) P flag
15. An alternate instruction for CLR C is,
    (a) CLR PSW.0 (b) CLR PSW.7 (c) CLR PSW.2 (d) CLR PSW.1
16. AC flag is used by,
    (a) arithmetic instructions only (b) logical instructions only
    (c) DA A instruction only (d) all instructions
17. INC instructions affect,
    (a) CY flag (b) AC flag (c) OV flag (d) None of the above
18. A = BAH and CY=0. After execution of instruction SUBB A ,#64H, the status of CY and OV flags will be,
    (a) CY=0, OV=0 (b) CY=0, OV=1 (c) CY=1, OV=0 (d) CY=1, OV=1

19. A = 65H and CY=0. After execution of instruction SUBB A, #78H, the contents of A will be,
    (a) EDH (b) DEH (c) 19H (d) 78H
20. Which of the following instructions require maximum execution time?
    (a) ADD A,@R0 (b) DA A (c) MUL AB (d) DIV AB
21. A = F0H, R0=30H and (30) = AAH. The contents of A after execution of an instruction XCH A, @ R0, will be,
    (a) AAH (b) F0H (c) 30H (d) None of above
22. A = F0H, R1=40H and (40) = 0FH. The contents of A after execution of instruction XCHD A, @ R1, will be,
    (a) 00H (b) F0H (c) 0FH (d) FFH
23. A =FEH and CY=1. The contents of A after execution of instruction RRA will be,
    (a) FCH (b) 7FH (c) F1H (d) FDH
24. If A =D6H and CY=0, the contents of A after execution of instruction RLC A will be,
    (a) ACH (b) CAH (c) ADH (d) DAH
25. If DPTR =20FFH, the contents of DPH after execution of instruction INC DPTR will be,
    (a) 20H (b) 21H (c) 00H (d) FFH

Answers to Objective Questions

1. (d)	2. (c)	3. (d)	4. (c)	5. (d)	6. (c)	7. (b)	8. (b)	9. (a)
10. (a)	11. (d)	12. (c)	13. (b)	14. (c)	15. (b)	16. (c)	17. (d)	18. (b)
19. (a)	20. (c), (d)	21. (a)	22. (d)	23. (b)	24. (a)	25. (b)		

REVIEW QUESTIONS WITH ANSWERS

1. Register A is always destination operand in addition and subtraction instructions. True/False.

A. True.

2. List arithmetic flags of the 8051.

A. Carry (CY), Auxiliary carry (AC), and Overflow (OV).

3. Arithmetic flags are affected only by arithmetic instructions. True/False.

A. False. Arithmetic flags are located in PSW register. Any instruction which can modify PSW will change the flags.

4. Write instruction/s to add numbers 10H and 20H and store the result in to internal RAM address 30H.

A.
```
MOV A,#10H
ADD A, #20H
MOV 30H, A
```

5. What will be the status of CY and AC flags after execution of following instructions?
MOV A, #58H
ADD A, #28H

A. CY=0, AC=1 (carry out from bit D3 to D4).

6. PSW may also be referred as a flag register. True/False.

A. True.

7. What is meant by user flag?

A. A programmer can alter (set or reset) the flag as per the requirements or it can be used to record one-bit event.

8. State the validity of the following instructions.
(a) ADD A, R0 (b) ADD R0, A (c) ADD R1, #05H (d) DEC DPTR

A. (a) Valid.
(b) Invalid, Register A is always destination operand in addition instruction.
(c) Invalid, Register A is always destination operand in addition instruction.
(d) Invalid, there no such instruction in 8051.

9. Discuss the role of overflow flag in a division operation.

A. When an attempt is made to divide some number by zero, the overflow flag will set to 1 to indicate that the result is incorrect (indeterminate).

10. Where should the operands of multiply instruction be stored? Where does it store the result?

A. One of the operand (either multiplicand or multiplier) should be in register A and other in B. The result is stored in B (MSByte) and A (LSByte).

11. State one common application of AND operation.

A. It is commonly used in masking (clear bits to 0).

12. The 8051 has signed multiplication instruction. True/False.

A. False, it has unsigned multiplication instruction.

13. EX-OR operation of number with itself always result in a zero. True/False.

A. True.

14. What is a limitation of the rotate instructions?

A. It works only with register A.

15. Represent decimal number 95 in packed and unpacked BCD format.

A. Packed BCD 1001 0101
Unpacked BCD: 0000 1001, 0000 0101

16. What are the limitations of DA A instruction?

A. It works with only register A and must only be used after ADD or ADDC instruction.

17. Show how the erroneous result obtained in signed arithmetic can be recovered using third method (to interpret the result) and displayed on the display device (like LCD or monitor).

A. Overflow flag indicates the erroneous result.
Write a program as follows:
- Get MSB of the result and complement it.
- If it is 0, send ASCII of '+' to the display device, treat the 8 bits of the original result as magnitude, convert it to BCD, then into ASCII.
- If result of step 1 is 1, send ASCII of '–' to the display device, find 2's complement of the original result and treat it as magnitude.

EXERCISE

1. List the steps of actions taken by DA A instruction with suitable example.
2. How does BCD addition differ from binary addition?
3. Discuss the importance of AC flag.
4. Name different math flags in PSW.
5. If OV=1 after division, what is the cause?
6. Where should the operands of divide instruction be stored? Where does it store the result?
7. Discuss the difference between following two instructions,
   (1) DEC A (2) SUBB A, #01H
8. Discuss common applications of OR and EX-OR operations.
9. Suggest different instruction/s to clear Accumulator.
10. Explain with suitable example how rotate instruction can be use to check whether number is odd or even.
11. Explain with suitable example how rotate instruction can be use to check whether number is positive or negative.
12. What is meant by packed and unpacked BCD numbers?
13. Write a program to multiply two 8-bit numbers stored at internal RAM address 10H and 11H. Store the result at address 12H(MSByte) and 13H(LSByte).
14. Write the instruction/s to perform following operations,
    (a) Mask bit D7 of R2
    (b) Set upper three bits at address 30H
    (c) Exchange the nibbles of R2
15. What will be the contents of register A after execution of each of the following instructions.
    CLR C
    MOV A, #55H
    ORL A, #0F0
    RL A
    RLC A
    RLC A
16. Show how the swap operation is realized using rotate instructions.

6

Bit-Processing Instructions

Objectives

- Identify the bit-addressable address spaces in the 8051
- Discuss the advantages of bit addressability
- Illustrate the bit operations with I/O ports
- Appreciate the use of carry flag in bit processing instructions
- Develop the programs to illustrate the use of bit processing instructions

Key Terms

- Boolean Processor
- Bit Addressability
- Bit Addressable SFR
- Bit-addressable RAM
- Carry Flag: Boolean Accumulator
- Set/Clear
- Complement
- Bit/Byte Address
- Conditional Jumps

In an 8-bit microcontroller, the minimum size of data that can be accessed or processed at a time is one byte. But, in many real-world machine control applications, we may need to manipulate (read/write/modify) only one bit at a time. For example, sensing the state (ON/OFF) of switch, make decision based upon state of the switch, and turn ON or turn OFF the external device. Such operations need to manipulate only required bits of a byte. If we perform these operations by accessing whole bytes, we should manipulate only required bits of byte without disturbing other bits, which demands additional efforts and care from the programmer.

6.1 BIT ADDRESSABILITY

The 8051 simplifies above-mentioned problem by providing unique and powerful feature of single bit addressability and single-bit operations. It contains a complete Boolean (single bit) processor and its instruction set is optimized for the single-bit operations. It supports SET, CLEAR, COMPLEMENT, AND and OR single bit operations. This feature makes the 8051 one of the most obvious choice for industrial machine-control applications.

Bit-processing operations provide the following advantages.

1. Faster execution of a program
2. Less memory required by a program
3. Program listing becomes simple and more readable

Example 6.2 will demonstrate the above advantages of using bit-processing instructions.

THINK BOX 6.1

What wrong may happen if we use following instruction to clear bit 7 of PSW?
MOV PSW, #00H or ANL PSW, #00H?

Other bits (bit 0 to 6) are unnecessarily cleared (disturbed). This may change selected register bank and affect the status of other flags which may result into logical error in a program

6.2 BIT-ADDRESSABLE MEMORIES

The 8051 has bit-addressable memory locations in internal RAM and special function registers. The internal RAM contains 128 addressable bits and majority of SFRs are bit-addressable including all I/O port pins.

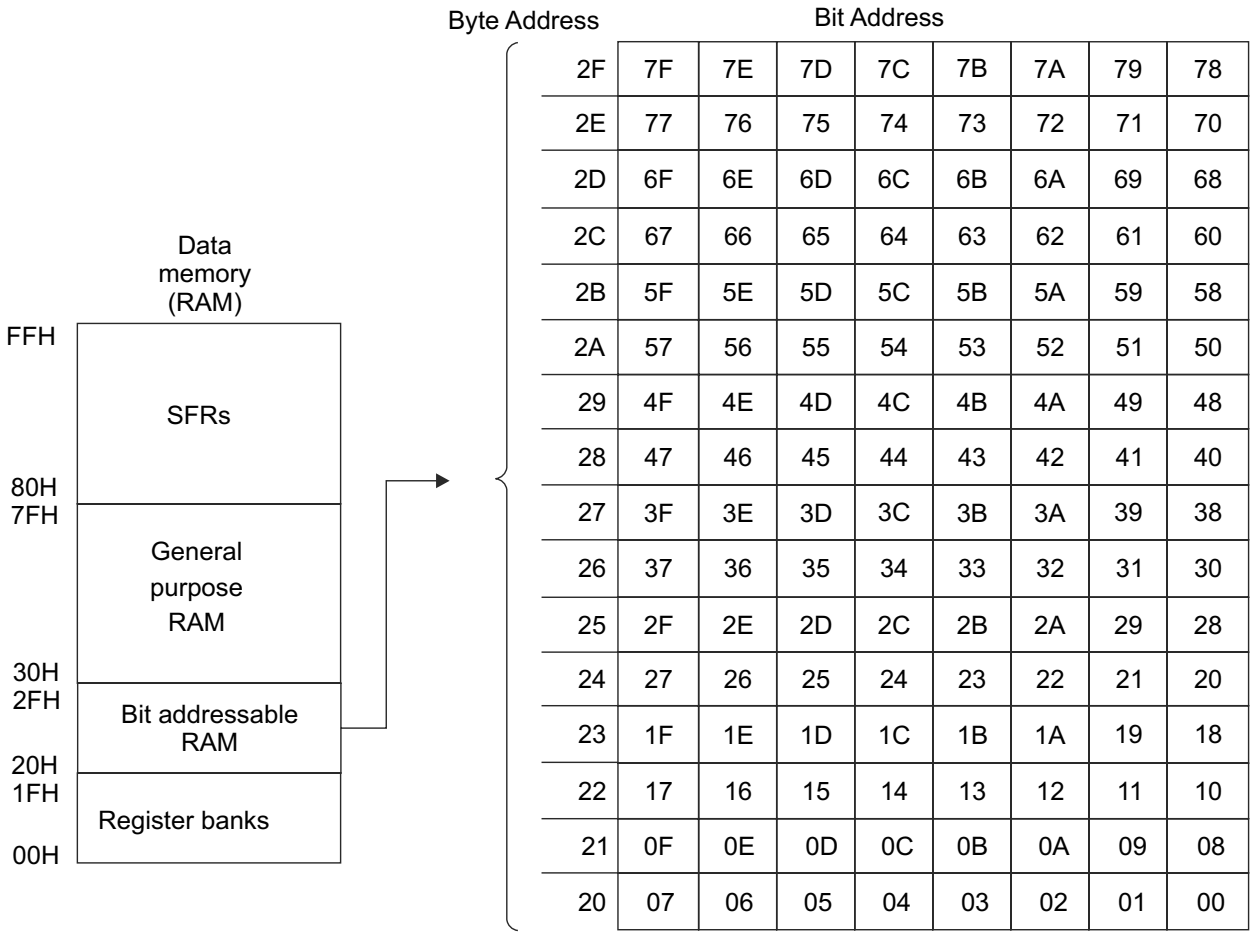
6.2.1 Bit-Addressable Internal RAM

The 8051 has a bit-addressable area of 16 bytes from byte addresses 20H to 2FH in internal RAM, forming a total of 128 (16x8) addressable bits. Addressable bits are assigned bit addresses from 00H to 7FH. The bit-addressable area with their individual bit addresses is shown in Figure 6.1.

As shown in Figure 6.1, internal RAM locations 20H to 2FH are both bit as well as byte addressable. Byte address 20H is assigned bit addresses 00H to 07H (00H address being assigned to least significant bit, 01H address to next higher bit and so on, and finally 07H address to most significant bit); 21H contains bits 08H to 0FH and so on. Hence bit addresses 00H to 7FH belongs to byte address 20H-2FH. The remaining locations in internal RAM must be accessed using their byte addresses only.

The relation between byte address and bit address can be established using the following equation.

$$(\text{Byte address})_H = 20_H + \text{Integer part of } \left[\frac{\text{Bit address (HEX)}}{8}\right] = \left[32\,D + \text{Integer part of } \left[\frac{\text{Bit address (HEX)}}{8}\right]\right] H$$



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

Fig. 6.1 *Internal RAM bit-addressable area*

Discussion Question How would a microcontroller know whether to access a bit or byte?

Answer The instruction (opcode of the instruction) that is used will determine whether a byte or a bit is being referenced without confusion. For example,

MOV A, 20H will access byte address 20H, while MOV C, 20H will access the bit address 20H. These two instructions have different op-codes.

Example 6.1

Which byte addresses do the bit addresses 07H, 24H and 4DH belong to?

Solution:

Bit addresses 07H, 24H and 4DH belongs to byte addresses 20H, 24H and 29H respectively. (See Figure 6.1)

6.2.2 Bit-Addressable Special Function Registers

All ports (P0, P1, P2 and P3), A, B, PSW, IP, IE, ACC, SCON, PCON and TCON are bit-addressable. Byte as well as bit addresses of all SFRs are shown in Figure 6.2.

As shown in Figure 6.2, P0 is assigned bit addresses 80H to 87H, P1 is assigned addresses 90H to 97H, (TCON is given addresses 88H-8FH). It should be noted that each port pin may be treated as a separate single bit port, i.e. each pin may be configured as an input or output independently as illustrated in Figure 6.3. Note that pins P1.0, P1.1, P1.2 and P1.4 are configured as an input and at the same time, all other pins of port 1 are used as an output.

Byte Address	Bit Address b7	b6	b5	b4	b3	b2	b1	b0	SFR name
FFH									
F0H	F7	F6	F5	F4	F3	F2	F1	F0	B
E0H	E7	E6	E5	E4	E3	E2	E1	E0	A
D0H	D7	D6	D5	D4	D3	D2	D1	D0	PSW
B8H	--	--	--	BC	BB	BA	B9	B8	IP
B0H	B7	B6	B5	B4	B3	B2	B1	B0	PORT 3 (P3)
A8H	AF	--	--	AC	AB	AA	A9	A8	IE
A0H	A7	A6	A5	A4	A3	A2	A1	A0	PORT 2 (P2)
99H	*								SBUF
98H	9F	9E	9D	9C	9B	9A	99	98	SCON
90H	97	96	95	94	93	92	91	90	PORT 1 (P1)
8DH	*								TH 1
8CH	*								TH 0
8BH	*								TL 1
8AH	*								TL 0
89H	*								TMOD
88H	8F	8E	8D	8C	8B	8A	89	88	TCON
87H	*								PCON
83H	*								DPH
82H	*								DPL
81H	*								SP
80H	87	86	85	84	83	82	81	80	PORT 0 (P0)

SFRs

* Indicates the SFRs which are not bit addressable

Fig. 6.2 *Bit and byte address of SFRs*

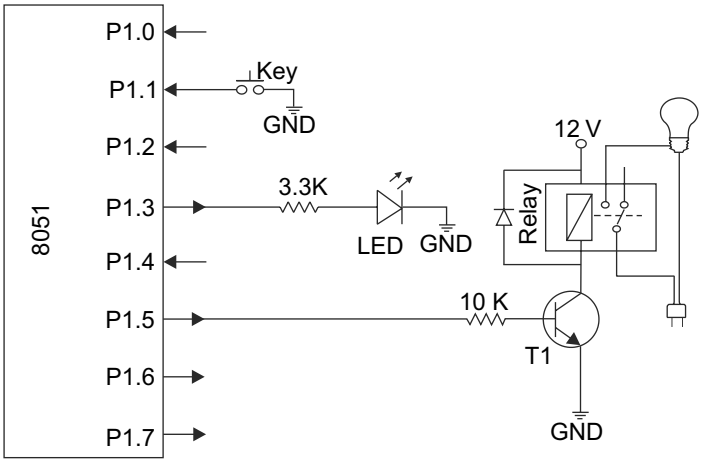


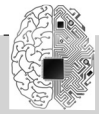
Fig. 6.3 *Independent operation of port pins*

The bits of ports and other SFRs can be accessed either using their addresses or names as given in Table 6.1.

Table 6.1 *SFR bits and their names*

SFR	BIT Name							
	D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
P0	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
P1	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
P2	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
P3	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
ACC	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0
B	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0
PSW	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
TCON	TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
IE	IE.7	*IE.6*	*IE.5*	IE.4	IE.3	IE.2	IE.1	IE.0
IP	*IP.7*	*IP.6*	*IP.5*	IP.4	IP.3	IP.2	IP.1	IP.0

The assembler will replace name with appropriate address while assembling the process, for example, instruction CLR P0.0 will be considered as a CLR 80H.



THINK BOX 6.2

Do you observe any pattern in addresses of bit-addressable SFRs?
Yes. If the lower digit (nibble) of the address is 0 or 8 then only that address is bit-addressable.

THINK BOX 6.3

Do you see any relation between the byte address and bit addresses of SFR?
The address of bit 0 (LSB) of any bit-addressable SFR is same as its byte address.

Discussion Question Can we simultaneously configure different pins of the same port as an input and output? If yes, how?

Solution Yes, we can configure some pins of port as an input and others as an output at the same time because all ports are bit-addressable and each pin may be treated independently using bit processing instructions.

For example, let us configure P1.0 and P1.1 as input pins and P1.2 and 1.3 as output. To configure any port pin as an input, we have to write '1' to corresponding port latch bit, and to configure port pin as an output, there is no special requirement, i.e. we can directly provide output data on port pin by simply writing the data in corresponding port latch bit. (Refer Topic 13.1.1 for more details on configuring port as an input and output). Following instructions are needed to configure P1.0 and P1.1 as input pins,

```
SETB P1.0      // Configure P1.0 as an input pin by writing 1 to P1.0 latch bit
SETB P1.1      // Configure P1.1 as an input pin by writing 1 to P1.10 latch bit
```

THINK BOX 6.4

Why are none of the ROM locations bit-addressable?
ROM usually contains program codes which are byte oriented; therefore they need not be bit-addressable.

6.3 BIT-PROCESSING INSTRUCTIONS

Bit-processing instructions operate on a bit of bit-addressable RAM area and SFRs. The carry flag is a destination for majority of the bit-processing instructions because it can be easily tested and program flow can be altered using decision making instructions (conditional jump instructions). It is equivalent to a single-bit accumulator.

The carry flag in PSW is considered as Boolean Accumulator.

The format of bit processing instructions with examples are given below.

6.3.1 Instruction using Carry

SETB C	// set the carry flag to 1, C=1
CLR C	// clear the carry flag to 0, C=0
CPL C	// complement the carry flag, C = $\overline{C}$
MOV *bit*, C	// copy status of the carry flag to bit address *bit*, (*bit)* = C
MOV 01H, C	// If C=0, → bit address (01H) =0
MOV C, *bit*	// copy status of bit address *bit* to the carry flag, C= *(bit)*
MOV C, 10H	// if bit address (10H) = 0, → C=0
ANL C, *bit*	// AND operation between C and *bit*, C = C AND *bit*
ANL C, 05H	// If C=0 and (05H) =1, → C= 0 AND 1=0
ANL C, */bit*	// AND op. between C and $\overline{bit}$, C = C AND $\overline{bit}$, do not affect status of *bit*
ANL C, /05H	// If C=1 and (05H) =0, → C= 1 AND $\overline{0}$ =1
ORL C, *bit*	// OR operation between C and *bit*, C = C OR *bit*
ORL C, 05H	// If C=0 and (05H) =1, → C= 0 OR 1=1
ORL C, */bit*	// OR op. between C and $\overline{bit}$, C = C OR $\overline{bit}$, do not affect status of *bit*
ORL C, /05H	// If C=0 and (05H) =0, → C= 0 AND $\overline{0}$ =1

THINK BOX 6.5

How can we perform addition between two bits?
Assume *x* and *y* are two bits. The Sum= x EX-OR y = $x\overline{y} + \overline{x}y$ and Carry=xy

6.3.2 Other Instructions

SETB *bit*	// set bit address *bit* to1, *(bit)* =1
SETB 00H	// $(00H)_{Bit}$=1
CLR *bit*	// clear bit address *bit* to 0, *(bit)* =0
CLR 00H	// $(00H)_{Bit}$=0
CPL *bit*	// complement bit address *bit*, *(bit)* = $\overline{(bit)}$
CPL 00H	// If $(00H)_{Bit}$=1 → $(00H)_{Bit}$=0

Example 6.2

Write a part of a program to select register bank 1 using both bit as well as byte-processing instructions. Show that use of bit-processing instructions makes program more efficient in terms of memory requirements and speed of execution.

Solution:

Using byte-processing instructions:

```
MOV PSW, #00001000B        // RS1=0, RS0=1 (3 Bytes, 2 machine cycles)
```

The problem in using the above instruction is that it modifies whole byte, i.e. it disturbs other bits of the PSW register which are not useful in bank selection operation. To preserve the other bits of the PSW, we may use the following two instructions:

```
ORL PSW, # 08H             // set RS0 bit (3 bytes, 2 machine cycles)
ANL PSW, #0EFH             // clear RS1 bit (3 bytes, 2 machine cycles)
                           // Total 6 bytes and 4 machine cycles
```

Using bit-processing instructions,

```
CLRB PSW.4                 // clear RS1 bit (2 bytes, 1 machine cycle)
SETB PSW.3                 // set RS0 bit (2 bytes, 1 machine cycle)
                           // Total 4 bytes and 2 machine cycles
```

It can be clearly seen that bit-processing instructions will make the program more efficient in terms of:

(i) Memory requirements, because it requires only 4 bytes compared to 6 bytes required by byte-processing instructions

(ii) Execution speed, because it requires only 2 machine cycles compared to 4 machine cycles required by byte-processing instructions

Example 6.3

Write instruction(s) to perform following operations:

(i) Read status of port pin P1.0 and store it in to bit D2 of byte address 22H as well as LSB of 2FH.

(ii) Complement the status of pins P1.0, P1.1 and P1.2 without affecting other bits.

(iii) Copy the status of port pin P1.0 to P0.5 as well as P0.6.

(iv) Set bit addresses 10H and 11H to 1.

(v) Set TR1 bit (timer1 run).

Solution:

(i)
```
MOV C, P1.0        //  read  P1.0  in to carry, C= P1.0
MOV 12H, C         // (12H) = C, bit D2 of byte address 22H has bit address 12H
MOV 78H, C         // (78H) = C, LSB of byte address 2FH has bit address 78H
```

Note that when the port bit is used as a source operand, the status of the port pin is read.

(ii)
```
CPL P1.0           // complement  P1.0
CPL P1.1           // complement  P1.1
CPL P1.2           // complement  P1.2
```

Note that when the port bit is used as a destination (or source as well as destination) operand, the status of port latch is modified (or read–modified and written back).

(iii)
```
MOV C, P1.0        // read  P1.0  into carry, C= P1.0
MOV P0.5, C        // send status of carry into P0.5
MOV P0.6, C        // send status of carry into P0.6
```

(iv) SETB 10H
SETB 11H
(v) SETB TR1

Example 6.4

Write a program to read status of the pin P2.1 and send its complement to the pin P1.0.

Solution:

```
SETB P2.1       // configure P2.1 as an input pin
MOV C, P2.1     // read status of pin P2.1 into carry flag
CPL C           // complement the status
MOV P1.0, C     // send complement of the status to the pin P1.0
```

Example 6.5

Show the status of the carry flag and bit address 10H after execution of each of the following instructions.

```
SETB C
CLR 01H
ORL C, /01H
MOV 10H, C
```

Solution:

```
SETB C          // C=1
CLR 01H         // (01H) =0, clear second bit (D1) of byte address 20H
ORL C, /01H     // C  OR  0̄ = 1    OR    1 =1
MOV 10H, C      // (10H) = C=1, bit D0 of byte address 22H
```

Both, bit address 10H and C will be set to 1 after execution of given instructions.

Example 6.6

Find the status of carry flag and other memory locations (involved in instructions) after execution of each of the following instructions.

```
CLR C
MOV 20H, #0FH
MOV C, 00H
ANL C, /07H
ORL C, 01H
CPL 06H
```

Solution:

```
CLR C           // C = 0
MOV 20H, #0FH   // (20H) = 0FH
MOV C, 00H      // C = 1
ANL C, /03H     // C = 0, (03H) =1, bit address 03H is 4th bit of 20H
ORL C, 01H      // C = 1, if (01H) =1, bit address 01H is 2nd bit of 20H
CPL 06H         // (06H) =1, bit address 06H is 7th bit of 20H
```

Example 6.7

Assume that a switch is connected to pin P1.1 and the LED is connected to pin P1.2. Write a program to read the status of switch and send this status to LED.

Solution:

```
SETB P1.1       // configure P1.1 as an input pin
MOV C, P1.1     // read status of pin P1.1 into carry flag
MOV P1.2, C     // send contents of carry flag to P1.2 (LED)
```

The above program fragment can be made more readable by using BIT directive. The BIT directive is used to assign a name to bit-addressable RAM locations. For example, P1.1 pin may be given a name SWITCH and pin P1.2 may be given a name LED using BIT directive to make program more readable and easy to understand. This is shown below.

```
SWITCH   BIT  P1.1     // assign name SWITCH to P1.1
LED      BIT  P1.2     // assign name LED to P1.2

SETB SWITCH            // configure P1.1 as an input pin
MOV C, SWITCH          // read status of pin switch (P1.1) into carry flag
MOV LED, C             // send contents of carry flag to LED (P1.2)
```

Note the program is now easy to understand. Moreover, use of BIT directive also allows easier modification of the program, i.e. we have to change RAM address only once while assigning the name to the address using BIT directive and corresponding change will be reflected in all places where the name is used. For example, if we change the first line of the above program as 'SWITCH BIT P2.1', we will get P1.1 replaced with P2.1 at all the places where the name SWITCH is used. (Refer Section 12.1.2 for more details of BIT directive or BIT data type)

Example 6.8

Realize single bit EX-OR instruction using other bit processing logical instructions.

Solution:

Assume that we want to perform the EX-OR operation between the two bits stored at bit addresses 00H (x) and 01H (y) and store result at address 02H.

We know that the EX-OR operation for two bits x and y is given as x EX-OR y= $x\bar{y} + \bar{x}y$

```
MOV C, 00H
ANL C, /01H          // xȳ
MOV 02H, C           // save partial result xȳ into 02H
MOV C, 01H
ANL C, /00H          // x̄y
ORL C, 02H           // xȳ + x̄y
MOV 02H, C           // save result into 02H
```

6.3.3 Conditional Jump Instructions

These instructions are discussed in more detail in Chapter 7.

```
JNC rel          // jump to address rel if C=0 (jump if no carry)
JC rel           // jump to address rel if C=1 (jump if carry)
JB bit, rel      // jump to address rel if bit =1 (jump if bit)
JNB bit, rel     // jump to address rel if bit =0 (jump if no bit)
JBC bit, rel     // jump to address rel if bit = 1, and then clear bit (jump if bit, then clear)
```

THINK BOX 6.6

An instruction 'MOV C, *bit*' requires 1 machine cycle. What do you think about machine cycles required by an instruction 'MOV *bit*, C'?

2 Machine cycles!

THINK BOX 6.7

Why does there exist bit-processing instructions in a microcontroller, which usually works on the bytes?

In many real-world machine-control applications, we may need to manipulate (read/write/modify) only one bit (or few bits) at a time. Bit processing instructions facilitate these operations easily.

Summary of Bit-Processing Instructions

Bit-processing instructions are summarized in Table 6.2.

Table 6.2 *Bit-processing instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
ANL C, **BIT**	C= C AND **BIT**	ANL C, **bit**			
		ANL C, **10H**			
ANL C, **/BIT**	C= C AND / **BIT**	ANL C, **/bit**			
		ANL A, **/12H**			
ORL C, **BIT**	C= C OR **BIT**	ORL C, **bit**			
		ORL C, **12H**			
ORL C, **/BIT**	C= C OR / **BIT**	ORL C, **/bit**			
		ORL C, **/7FH**			
MOV C, **BIT**	C= **BIT**	MOV C, **bit**			
		MOV C, **20H**			
MOV BIT, **C**	BIT= **C**	MOV bit, **C**			
		MOV 10H, **C**			
CLR **C**	C= **0**	CLR C			
CLR **BIT**	BIT= **0**	CLR **bit**			
		CLR **12H**			
SETB **C**	C= **1**	SETB C			
SETB **BIT**	BIT= **1**	SETB **bit**			
		SETB **10H**			
JC **rel**	Jump if C = 1	JC **rel**			
		JC **HERE**			
JNC **rel**	Jump if C = 0	JNC **rel**			
		JNC **HERE**			
JB BIT, **rel**	Jump if BIT = 1	JB BIT, **rel**			
		JB 12H, **HERE**			
JNB BIT, **rel**	Jump if BIT = 0	JNB BIT, **rel**			
		JNB 10H, **NEXT**			
JBC BIT, **rel**	Jump if BIT = 1; CLR BIT	JBC BIT, **rel**			
		JBC 10H, **NEXT**			
CPL **C**	C = NOT C	CPL C			
CPL **BIT**	BIT = NOT BIT	CPL **bit**			
		CPL **12H**			

POINTS TO REMEMBER

- The 8051 contains a complete Boolean (single bit) processor and its instruction set is optimized for the single-bit operations. It supports SET, CLEAR, COMPLEMENT, AND and OR single bit operations.
- Faster execution, less memory requirements and program readability are the advantages offered by the bit processing instructions.
- The 8051 has a bit-addressable area of 16 bytes from byte addresses 20H to 2FH in internal RAM, forming a total of 128 (16x8) addressable bits. Addressable bits are assigned bit addresses from 00H to 7FH.
- All ports (P0, P1, P2 and P3), A, B, PSW, IP, IE, ACC, SCON, PCON and TCON are bit-addressable.
- The carry flag in PSW is considered as Boolean Accumulator. The carry flag is destination for a majority of bit-processing instructions because it can be easily tested and program flow can be altered using decision making instructions.

OBJECTIVE QUESTIONS

1. Which of the following SFRs is bit-addressable?
   (a) SP (b) P2 (c) TMOD (d) SBUF
2. Bit-addressable locations of the 8051 include
   (a) 16 internal RAM bytes (b) majority of SFRs
   (c) I/O ports (d) all of the above
3. The 8051 has bit-level instructions for
   (a) AND (b) OR (c) complement (d) all
4. Bit-addressable internal RAM has address range of
   (a) 10H to 1FH (b) 20H to 2FH (c) 30H to 3FH (d) 40H to 4FH
5. Which of the following instructions is not a bit-processing instruction?
   (a) CLR C (b) CLR A (c) CPL 10H (d) CLR 10H
6. Which of the following SFRs are bit-addressable?
   (a) P0 (b) DPL (c) B (d) TMOD
7. The 8051 does not have bit-level instructions for
   (a) AND (b) OR (c) Complement (d) EX-OR
8. Which of the following instructions are invalid?
   (a) SETB C (b) SETB B (c) ANL C, #00H (d) MOV C, 10H
9. If (20H)=00H and CY=1 the contents of internal RAM address 20H after execution of following instructions will be,
   ORL C, 00H
   MOV 00H, C
   (a) 00H (b) 01H (c) 20H (d) FFH
10. If (20H) = FFH and CY=1 the contents of internal RAM address 20H after execution of following instructions will be,
    ANL C, 00H
    CPL C
    MOV 00H, C
    (a) 00H (b) 01H (c) FFH (d) FEH

Answers to Objective Questions

1. (b)	3. (d)	5. (b)	7. (d)	9. (b)
2. (d)	4. (b)	6. (a), (c)	8. (b), (c)	10. (d)

REVIEW QUESTIONS WITH ANSWERS

1. List the advantages offered by the bit-processing instructions.

A. Bit-processing instructions makes program more efficient in terms of speed of execution and code density, i.e. faster execution of program, and less memory required by the program. Furthermore, program listing and development becomes simple.

2. What is meant by a Boolean processor?

A. It is a circuit capable of handling single-bit operations. It is equivalent to single bit ALU.

3. All ports of the 8051 are bit-addressable. True/False.

A. True.

4. Which of the following registers are bit-addressable?
A, B, R0, DPL, TL0, TCON

A. A, B and TCON are bit-addressable.

5. In which address space do bit addresses 00H-7FH and 80H-FFH belong?

A. 00H-7FH belongs to internal RAM at byte addresses 20H- 2FH.
80H-FFH belongs to SFRs.

6. Carry for the bit-processing instructions is equivalent to single-bit accumulator. True/False.

A. True.

7. Write instructions to save the status of pin P1.0 at the bit address 10H.

A.
```
MOV C, P1.0          or          CLR 10H
MOV 10H, C                       JB P1.0, HIGH
                                 SJMP NEXT
                        HIGH: SETB 10H
                        NEXT: …
```

8. Write two instructions to clear the carry flag.

A. CLR C or CLR 0D7H

9. State the validity of the following instructions.

(a) SETB C (b) SETB B (c) ANL C, 00H
(d) ANL C, #00H (e) ANL 00H, C (f) MOV C, 10H
(g) MOV C, @R0

A. (a) Valid.
(b) Invalid, SETB is used to set bits, while B is a byte.
(c) Valid.
(d) Invalid, immediate addressing is not supported by bit processing instructions.
(e) Invalid, C is always destination in bit processing AND operations.
(f) Valid.
(g) Invalid, Indirect addressing is not supported by bit processing instructions.

10. Write the instruction to configure port pin P1.0 as an input.

A. SETB P1.0 (by writing 1 to port bit latch).

11. Each pin of a port can be programmed independently as an input or output. True/False.

A. True.

12. All SFRs of the 8051 are bit-addressable. True/False.

A. False.

13. Which byte addresses does the bit addresses 06H and 7FH belong to?

A. 20H and 2FH respectively.

14. What is the address of MSB of port1?

A. 97H.

EXERCISE

1. List all bit-addressable SFRs and write bit addresses assigned to them.
2. Write a program to EX-OR first and second (D0 and D1) bits of Accumulator and save the result in third bit of accumulator.
3. How can the status of C and OV flag be monitored?
4. What is the range of bit addresses assigned to SFRs?
5. Write instructions to toggle pin P1.0 continuously if P2.0 is high, otherwise clear P1.0.
6. How can the status of port pins be monitored?
7. Write a program to generate a rectangular wave of 75% duty cycle on pin P1.0.
8. Show with suitable example that use of bit-processing instructions makes program more efficient in terms of speed of execution and memory requirements.
9. When port bit is used as a destination operand, the status of port latch is modified. True/false.
10. What is common use of ANL C, /*bit* and ORL C, /*bit* instructions?

7

Program-Flow Control Instructions

Objectives

- List and classify the jump instructions
- Compare different unconditional jump instructions along with their range.
- Show how unconditional instructions are coded
- Describe the conditions used for bit and byte-level conditional jump instructions
- Introduce the looping technique
- Illustrate the call and return operations
- Develop the subroutines using call instructions
- Discuss the stack initialization and overflow
- Understand the time delay generation using software

Key Terms

- Call
- Context Saving/Retrieving
- Counter
- Destination Address
- Jump
- Loops
- Page
- Pointer
- Relocatability
- Return
- Return Address
- Stack Overflow
- Stack Pointer
- Subroutine
- Time Delay

The instructions discussed thus far were performing sequential operations, i.e. they were executed one by one in a sequence in the order that they appear in a program memory. Each instruction performs a single and simple operation. Many times we need to change the order of the execution of the instructions to other memory location, based on either certain conditions existing at the time or even without any condition. The program flow control instructions allow the microcontroller to alter the sequence of the program execution. These instructions make the program more flexible and versatile as required by real world applications. These instructions are also referred as *Branch instructions* or simply Jump instructions. The Jump and Call instructions in the 8051 have capability to alter the program flow.

7.1 JUMP INSTRUCTIONS

A jump instruction changes the content of program counter with the new program address usually referred as *destination address* or *target address*. This causes program execution to begin at the destination address. Two types of jumps are supported by the 8051: unconditional and conditional jumps.

7.1.1 Unconditional Jumps

The unconditional jump does not test any condition and jump is always taken. The 8051 supports three types of unconditional jumps: short (relative) jump, absolute jump and long (direct) jump. These jumps differ in range (in terms of bytes) over which the jump can be taken.

1. Short Jump

We know that the PC always contains the address of the next instruction (with respect to instruction that is being executed). Using a short jump, a program may only jump to instructions within 127 bytes in forward direction (+127) or 128 bytes in reverse direction (–128) with respect to contents of the PC (next instruction). A short jump is called *relative jump* because the destination address that is specified in the instruction (and then placed in the program counter) is relative to the address where the jump instruction is written. The advantage offered by relative jump is that it allows relocation, i.e. a program that is written using relative jump instructions can be placed (loaded) anywhere in the entire program space without reassembling. The second advantage is that only 1 byte is required to specify relative address of the destination location which saves the program bytes and increases the speed of execution. The instruction for short jump is SJMP and its format is,

SJMP *rel* // jump to relative destination address *rel*

SJMP is two-byte instruction: the first byte is op-code and second byte is relative address.

Calculating Relative Address from the Actual Destination Address

The relative address is relative to the value of the PC and to calculate relative address, the value of PC is subtracted from actual destination address. Consider the following program given in Figure 7.1.

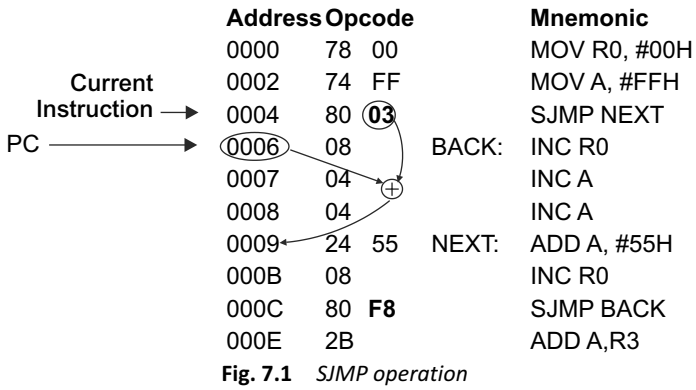


Fig. 7.1 *SJMP operation*

Consider the instruction "SJMP NEXT". It is written at address 0004H, so during the execution of this instruction, PC = 0006H, i.e. the PC points to the next instruction. Now destination instruction is "NEXT: ADD A, #55H" is located at address 0009H, so the relative address of destination instruction is 0009H-0006H = 03H. Therefore, relative address 03H is to be specified as a second byte of the instruction "SJMP NEXT" (see op-codes in the Figure 7.1). Conversely, if relative address is known, we can calculate actual address of the destination instruction. To calculate destination address, the second byte of SJMP instruction is added with the PC. For same example, add relative address into value of PC, i.e. add 03 with 0006H to get 0009H (actual address). The jump described in the above example is *forward jump* because destination address is ahead with respect to jump instruction.

Consider in the above example, the instruction "SJMP BACK". As can be seen, it is a *backward jump* because destination address is at lower address. Here, relative address is specified in 2's complement negative format. While

execution of the instruction "SJMP BACK", the PC = 000E and destination address is 0006H; therefore, displacement is 000EH-0006H=08H. Since displacement 08 is in backward direction, it is specified in 2's complement which is F8H. To get the actual destination address from the relative address, add displacement with value of PC, i.e. 0E+F8=1**06 (**Carry is neglected).

Fortunately, the programmer does not have to calculate the relative address; it is automatically calculated by an assembler, and programmer should use only labels.

The disadvantage of relative addressing is its limited range, i.e. within -128 to 127 bytes with respect to PC. If jump beyond this range is required then a jump can be made to address containing another jump instruction until desired address is reached.

Discussion Question What will be the second byte of the SJMP instruction (relative address) for following program fragment?

```
1000H          SJMP NEXT
…              …
1025H  NEXT:   ADD R1, #05H
```

Answer During the execution of the SJMP instruction, the PC is pointing to the next instruction, i.e. 1002H because SJMP is a two-byte instruction. The relative address will be difference between destination address and PC, therefore, relative address will be 1025H-1002H= 0023H= 23H, Note that only lower byte is considered.

Example 7.1

Assume that 8 switches are connected to port 1 pin and 8 LEDs are connected to port2 pins, write instructions to read status of all switches and send it to LEDs continuously.

Solution:

```
          MOV P1, #0FFH     // configure P1 as input port
REPEAT:   MOV A, P1         // read status of switches
          MOV P2, A         // send status of switches to LEDs
          SJMP REPEAT       // repeat task continuously and unconditionally
```

Example 7.2

Illustrate the common uses of SJMP instruction with suitable example.

Solution:

(i) SJMP is commonly used to repeat a part of a program (or whole program) indefinitely without checking any condition. The structure of such a program is shown below,

```
          …
REPEAT:   …               // repeat following instructions forever
          …
          …
          SJMP REPEAT
          …
```

(ii) SJMP is also used to skip part of the program as shown below,

```
          …
          SJMP SKIP       // continue program execution at label 'SKIP' and skip
          …               // following instructions
          …
          …
SKIP:     …
          …
```

(iii) Third common use of SJMP instruction is to stop program execution. For detailed explanation, refer topic 'How to stop program execution in the 8051?' at the end of Section 7.1.1 in this chapter.

2. Absolute Jump

The instruction for absolute jump is AJMP and its format is,

AJMP *add11* // jump to absolute destination address *add11*

AJMP instruction logically divides entire program memory into 32 pages of 2K Bytes each. The address range of each page is shown in Table 7.1.

Table 7.1 *Address range for pages in 8051*

Page No.	Address range (Hex)	Page No.	Address range (Hex)	Page No.	Address range (Hex)
00	0000- 07FF	0B	5800- 5FFF	16	B000- B7FF
01	0800- 0FFF	0C	6000- 67FF	17	B800- BFFF
02	1000- 17FF	0D	6800- 6FFF	18	C000- C7FF
03	1800- 1FFF	0E	7000- 77FF	19	C800- CFFF
04	2000- 27FF	0F	7800- 7FFF	1A	D000- D7FF
05	2800- 2FFF	10	8000- 87FF	1B	D800- DFFF
06	3000- 37FF	11	8800- 8FFF	1C	E000- E7FF
07	3800- 3FFF	12	9000- 97FF	1D	E800- EFFF
08	4000- 47FF	13	9800- 9FFF	1E	F000- F7FF
09	4800- 4FFF	14	A000- A7FF	1F	F800- FFFF
0A	5000- 57FF	15	A800- AFFF		

As can be seen from the Table, the upper five bits of an address in each page remains constant and represents a page number. For example, consider page 02, its address range is 1000H-17FFH. The upper five bits "**0001 0**000 0000 0000B – **0001 0**111 1111 1111B" represent the page number 02, and throughout the page, these bits are same.

AJMP instruction can jump within a page of 2K. Since upper five bits are same for each page, they need not be specified for destination address. Lower 11 bits hold address within a page and need to be specified. An absolute destination address is formed by taking page number (first five bits) of instruction following the AJMP (page number of address in the PC) and attaching 11 bits of destination address. This can be understood by Example 7.3.

Example 7.3

Show how the destination address in AJMP instruction is specified.

Solution:

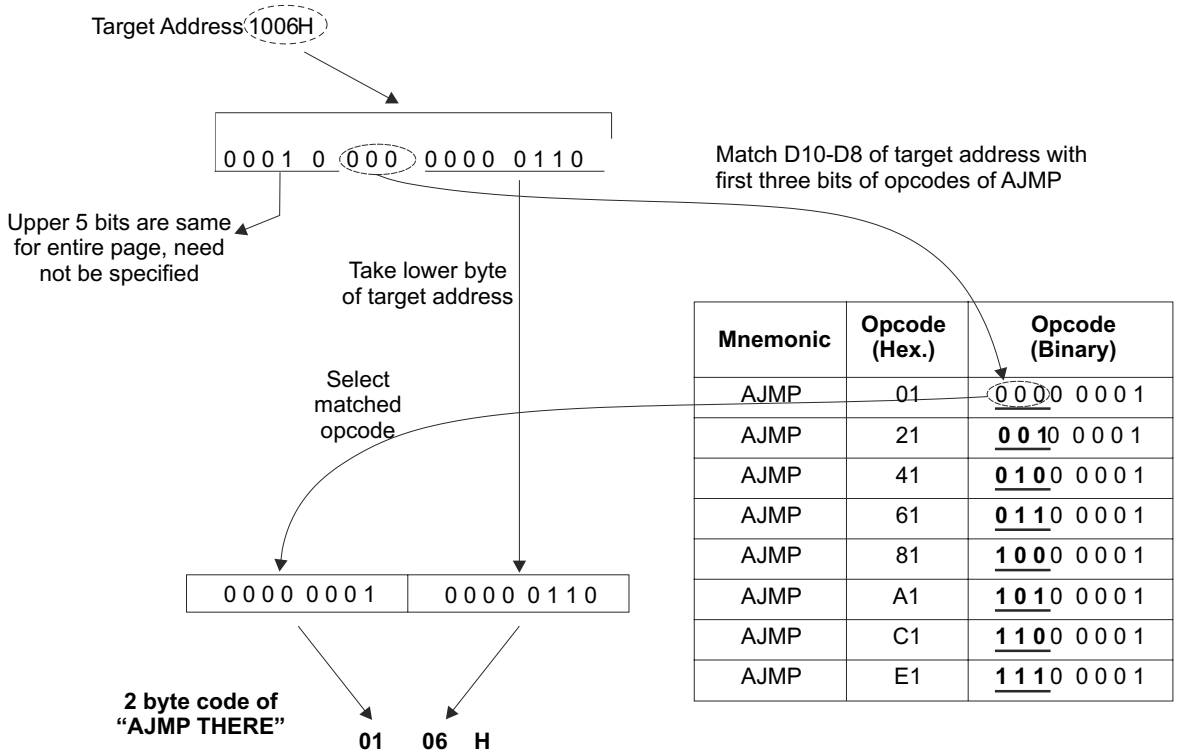
Consider the following program:

Address	Opcode	Mnemonic
1000	78 00	MOV R0,#00H
1002	**01 06**	AJMP THERE
1004	04	INC A
1005	08	INC R0
10**06**	24 55	THERE: ADD A,#55H
1007	08	INC R0

The label "THERE" represents absolute destination address (1006H) for the instruction "AJMP THERE". Since instructions "AJMP THERE" as well as "THERE: ADD A,#55H" belongs to same page (Page 2), we need not to specify upper five bits of destination address, remaining 11 bits are specified as follows.

Observe the target (destination) address, i.e. address 1006H,

The lower 8 bits of destination address (*add11*) are specified directly as a second byte of the AJMP instruction (06 in the example). The remaining 3 bits are specified indirectly by choosing one out of the eight different op-codes of AJMP instruction as shown in Figure 7.2. Match these three bits (bit D10, D9 and D8 of destination address) with the first three bits of the op-codes of the AJMP. Choose the op-code for which these three bits are same. In our example bits, D10-D8 are 000 which matches with the first three bits of op-code 01H (**000**0 0001), therefore choose op-code 01H for AJMP. This way, instruction "AJMP THERE" is coded as "01 06 H".



Mnemonic	Opcode (Hex.)	Opcode (Binary)
AJMP	01	0 0 0 0 0 0 0 1
AJMP	21	0 0 1 0 0 0 0 1
AJMP	41	0 1 0 0 0 0 0 1
AJMP	61	0 1 1 0 0 0 0 1
AJMP	81	1 0 0 0 0 0 0 1
AJMP	A1	1 0 1 0 0 0 0 1
AJMP	C1	1 1 0 0 0 0 0 1
AJMP	E1	1 1 1 0 0 0 0 1

Fig. 7.2 *Opcode generation process for instruction AJMP*

As another example, let us find the op-code for same instruction "AJMP THERE" if the destination address is 1208H.

The upper 5 bits are not required to be specified because they are directly taken from the PC; lower 8 bits are taken from lower byte of the destination address, i.e. 08H. The bits D10-D8 are 010 for the destination address which matches with first three bits of op-code 41H, therefore the two-byte code for instruction "AJMP THERE" is "41 08". If assembler is used to generate the machine codes, this process will be automatically done by an assembler.

When AJMP instruction occurs at page boundary, i.e. at address x7FFH or xFFFH (or x7FEH or xFFEH), the next instruction starts at address x801 or x001 (or x800 or x000), which places destination address on the next page, however, this page change does not cause any problem when there is a forward jump, but causes trouble when the jump is backward. The assembler should report this error while assembling the program, so necessary action (of writing such instructions at other address) can be taken by the programmer to resolve this problem.

The advantage of using AJMP instruction is similar to SJMP, i.e. only two-byte machine code, also jump range is anywhere within 2K page and program is relocatable provided that relocated code begins at the start of a page.

Note: AJMP (and ACALL) are the special instructions for which there are eight op-codes. This is an example of compromise between number of instructions possible for the 8051 and number of bytes required for AJMP (and ACALL) instructions.

THINK BOX 7.1

We know that AJMP and ACALL instructions each have eight op-codes. Can we use any of the op-code any time?
No. It is decided by destination address specified in an instruction.

3. Long Jump

The instruction for long jump is LJMP and its format is,

LJMP *add16* // jump to direct destination address *add16*

LJMP instruction allows a jump to any location within entire program address space, i.e. any where from 0000H to FFFFH. It is a 3-byte instruction, first byte is the op-code for the instruction and the second and third byte represents directly 16-bit destination address. Therefore long jump is also referred as direct jump. Advantage of LJMP instruction is its address range. The disadvantage is that it is three byte instruction and programs using it are not relocatable. LJMP is normally used in larger programs. The jump ranges of all unconditional jumps are illustrated in Figure 7.3.

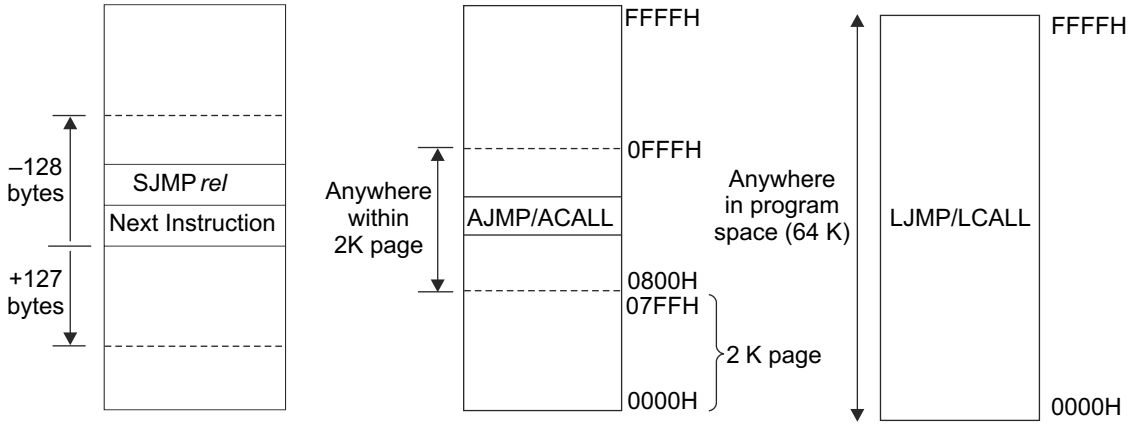


Fig. 7.3 *Range of SJMP, AJMP/ACALL and LJMP/LCALL (not to the scale)*

Discussion Question Compare SJMP, AJMP and LJMP instructions with respect to

(i) Jump range
(ii) Number of bytes of instructions
(iii) Speed of execution
(iv) Relocatability

Answer:

(i) **Jump range**
SJMP: –128 to +127 bytes with respect to PC
AJMP: Anywhere within the current page of 2Kbytes
LJMP: Anywhere within the entire 64Kbytes of program memory

(ii) **Number of bytes of instructions**
SJMP: 2 bytes
AJMP: 2 bytes
LJMP: 3 bytes

(iii) **Speed of execution**
SJMP: 2 machine cycles
AJMP: 2 machine cycles
LJMP: 2 machine cycles

(iv) **Relocatability**
SJMP: Fully relocatable, because relative distance between SJMP instruction and destination address will remain same irrespective of starting address of a program.
AJMP: Relocatable, as long as relocated code begins at the start of a page.
LJMP: Not relocatable, because this instruction uses exact address in the instruction, therefore when program is reloaded at different address, the destination address also changes. So we need to reassemble the code.

Discussion Question When is it preferred to use

(i) SJMP instruction
(ii) LJMP instruction

Answer

(i) SJMP is preferred in a program when distance between jump instruction and destination address is less, i.e. within –128 to +127 bytes, and when available program memory is restricted.
(ii) LJMP is preferred in larger programs where required jump range is more.

How can you stop the Program Execution in the 8051?

The 8051 does not have an instruction to stop the program execution. Therefore, once it is powered on, it always executes some instructions (except for power saving modes).

If a task is completed (or a task have to wait for external event or command), keep the microcontroller busy in an infinite loop using unconditional jump instruction. Jump to the same instruction is commonly used to implement the infinite loop. The SJMP instruction is commonly used for this purpose as shown in the following instruction.

'HERE: SJMP HERE' or 'HERE: AJMP HERE'

These instructions jump to the same instruction forever, which is as good as stopping further program execution.

7.1.2 Conditional Jumps

Conditional jump instructions first test the condition specified in an instruction op-code. If the condition is true then the jump is taken by modifying the value of the PC, otherwise program execution continues to the next sequential instruction. All conditional jumps are short relative jumps. The conditional jumps are categorized as bit and byte jumps.

1. Bit Jumps

Bit jump instructions check the status of addressable bit specified as a part of instruction. The jump is taken to specified relative address if the condition (for specified bit) is satisfied (or true), otherwise program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. These instructions are used to monitor the status of specified bit and to take the decision based on its value. Bit jump instructions are listed below.

JC *rel*	// jump to relative address *rel* if C=1 (Jump if Carry)
JC BACK	// jump to (continue program execution from) label BACK if C=1
JNC *rel*	// jump to relative address *rel* if C=0 (Jump if No Carry)
JNC NEXT	// jump to (continue program execution from) label NEXT if C=0
JB *bit*, *rel*	// jump to relative address *rel* if *bit* =1 (Jump if Bit)
JB 00H, TMP	// jump to label TMP if bit 00H is set (content of 00H is 1)
JNB *bit*, *rel*	// jump to relative address *rel* if *bit* =0 (Jump if No Bit)
JNB 05H, MP	// jump to label MP if bit 05H is clear (content of bit address 05H is 0)
JBC *bit*, *rel*	// jump to relative address *rel* if *bit* = 1, and then clear *bit* (Jump if Bit, then Clear)
JBC 50H, XY	// jump to label XY if bit 50H is set (content of bit address50H is 1), then clear bit 50H

THINK BOX 7.2

How is 'JBC bit, rel' instruction more efficient than other bit jump instructions?

It combines bit test, clear bit and jump operation in a single instruction.

Example 7.4

Illustrate the use of JNC, JC JNB, JB and JBC instructions.

Solution:

(i) Use of JNC

```
        CLR C
        MOV A, #10H
        MOV R0, A
AGAIN:  ADD A, R0
        JNC AGAIN     // repeat addition of A with R0 until carry flag is set
        …
```

(ii) Use of JC

```
                …
                SETB C          // set carry for illustration
                JC AHEAD        // since C=1, continue program execution from label
                …               // AHEAD
                …
AHEAD:          MOV A, P1
                …
```

(iii) Use of JNB and JB

Assume that a pushbutton switch is connected to pin P1.0 and, when a switch is pressed, the logic high is given to the pin. Otherwise, it remains at low logic. Write instructions to toggle the status of pin P2.0 every time the switch is pressed.

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JNB P1.0, WAIT  // wait until switch is pressed
        CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Alternatively, JB can be used as shown below:

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JB P1.0, COMP   // If a switch is pressed, jump to COMP
        SJMP WAIT       // repeat the operation if switch is not pressed
COMP:   CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Assume that a pushbutton switch is connected to pin P1.0. When switch is pressed, logic low is given to the pin; otherwise it remains at high logic. Write instructions to toggle the status of pin P2.0 every time the switch is pressed.

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JB P1.0, WAIT   // wait until switch is pressed
        CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Alternatively, JNB can be used as shown below:

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JNB P1.0, COMP  // If switch is pressed jump to COMP
        SJMP WAIT       // repeat the operation if switch is not pressed
COMP:   CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

(iv) **Use of JBC:** It is illustrated in **Example 16.12** (in the more efficient method part of example).

---

2. Byte Jumps

Byte jump instructions check bytes of data to make a decision whether to jump to destination address or continue to the next instruction. The byte jump instructions are listed below.

```
JZ rel                  // jump to relative address rel if A is 0 (Jump if Zero)
JNZ rel                 // jump to relative address rel if A is not 0(Jump if Not Zero)
CJNE A, direct, rel     // compare A with contents of address direct and jump to relative
                        // address rel if they are not equal, if A is less than contents of address
                        // direct, set carry flag to 1, otherwise clear to 0
                        // CJNE means Compare and Jump if Not Equal)
CJNE A, #data, rel      // compare A with immediate value data and jump to relative address
                        // rel if they are not equal, if A is less than immediate value data, set
                        // carry flag to 1, otherwise clear to 0
```

```
CJNE Rn, #data, rel      // compare Rn with immediate value data and jump to relative address
                         // rel if they are not equal, if Rn is less than immediate value data, set
                         // carry flag to 1, otherwise clear to 0
CJNE @Ri,#data, rel      // compare contents of address in Ri with immediate value data and jump to relative
                         // address rel if they are not equal, if contents of address in Rn is less than immediate
                         // value data, set carry flag to 1, otherwise clear to 0
DJNZ Rn, rel             // decrement register Rn by 1 and jump to relative address rel if
                         // content of Rn is not zero after decrement operation, no flags are affected
                         // DJNZ means Decrement and Jump if Not Zero
DJNZ direct, rel         // decrement contents of address direct by 1 and jump to relative address rel if content of
                         // address direct is not zero after decrement operation, no flags are affected
```

For all the byte jump instructions, the jump is taken to specified relative address if condition is satisfied (true), otherwise program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. These instructions are used to monitor the value of specified byte and to take the decision based on its value. It should be noted that there is no zero flag in the 8051, the instructions JZ and JNZ checks the Accumulator for zero. The DJNZ instruction decrements the specified operand first and then checks operand for zero. The CJNE instruction does not affect any of its operands.

Example 7.5

Write instructions to monitor the status of port 2 continuously until it is 55H.

Solution:

```
        MOV P2, #0FFH          // configure P2 as input
REPEAT: MOV A, P2              // read status of P2 in to A
        CJNE A, #55H, REPEAT   // repeat until P2 status is 55H
        …
```

Example 7.6

(i) Write instructions to increment contents of R5 until it becomes 50H.

(ii) Modify above program fragment to increment R5 until it is equal to contents of address 30H.

Solution:

```
(i)          …
     REPEAT: INC R5                 // increment R5 until its value is 50H
             CJNE R5, #50H, REPEAT
             …
(ii) REPEAT: INC R5                 // increment R5 until its value is equal to content of address 30H
             MOV A, R5
             CJNE A, 30H, REPEAT
             …
```

The jump instructions are widely used in looping, which is discussed in detail in the next topic.

7.2 | LOOPS

In many cases, we have to repeat a single task for several times, and the best way to do this is by looping. The *looping* is a programming technique used to repeat the sequence of instructions several times until certain conditions are met. The repeatability is the key feature and reason for popularity of the microcontroller (or computer) based systems. The looping allows us to develop concise and efficient programs. The most common requirement for loops is to specify loop

count, which determines the number of times a task has to be repeated. The loop count is loaded into some register or memory location before the loop is started. The register or memory location that holds the loop count is usually referred as a counter for that task or a program. After each iteration, a loop counter is decremented by one and a test is made to check if the loop counter is zero, if it is zero then the loop is terminated otherwise the task is repeated. There are two basic types of loops: unconditional and conditional loops. Unconditional loops repeat the task indefinitely without checking any condition until system is reset or power down. Conditional loops repeat the task until certain condition exists.

In the 8051, the instructions DJNZ and CJNE are used to repeat the loop for fixed number of times, while the instructions JZ/JNZ and all bit jump instructions are used to repeat the loop until a flag or bit is set to desired state. The typical structure of loop using the DJNZ instruction is shown below:

```
        MOV R4, #10      // load the count (number of times loop to be repeated)
LOOP:   ...              // begin loop
        ...
        ...
        ...              // end loop
        DJNZ R4, LOOP    // check whether loop is repeated required times, if not repeat it, otherwise, exit from the
                         // loop and continue program execution at next instruction
        ...
```

Let us consider a few simple examples to understand the looping.

Example 7.7

Add 5 to A register ten times.

Solution:

The given task can be performed in many ways; three simple ways are given here.

(i) A counter is initialized with 10 and the counter is decremented after every addition and checked for zero using the DJNZ instruction, if it is not zero then addition is repeated, otherwise the loop has already been repeated 10 times and program execution will come out of the loop and continue to the next task.

```
        CLR A            // clear A
        CLR C            // clear carry flag
        MOV R3, #10      // loop counter R3=10
REPEAT: ADD A, #05       // add 05 to A
        DJNZ R3, REPEAT  // repeat addition operation 10 times
```

(ii) A counter is initialized with 10 and the counter is decremented after every addition using the DEC instruction and compared with zero using the CJNE instruction; if it is not zero then addition is repeated, otherwise program execution will come out of the loop and continue to the next task.

```
        CLR A                  // clear A
        CLR C                  // clear carry flag
        MOV R3, #10            // loop counter R3=10
REPEAT: ADD A, #05             // add 05 to A
        DEC R3                 // decrement the loop count
        CJNE R3, #00, REPEAT   // repeat addition operation 10 times
```

(iii) A counter is initialized with 00 and the counter is incremented after every addition using the INC instruction and compared with 10 using the CJNE instruction; if it is not equal to 10, then addition is repeated, otherwise program execution will come out of the loop and continue to the next task

```
        CLR A                  // clear A
        CLR C                  // clear carry flag
        MOV R3, #00            // loop counter R3=00
REPEAT: ADD A, #05             // add 05 to A
        INC R3                 // decrement the loop count
        CJNE R3, #10, REPEAT   // repeat addition operation 10 times
```

Example 7.8

Write a program to add contents of ten memory locations from 20H onwards.

Solution:

The result of addition may be greater than 8 bits, therefore we need to store result in two 8-bit locations. After each addition carry flag is checked; if it is set, then content of address 41H is incremented which was initialized with value 00H. This will correct the 9th or higher bits of the result.

```
        MOV R2, #0AH        // counter for addition of 10 numbers
        MOV R0, #20H        // initialize pointer to first memory address
        CLR C
        MOV 41H, #00H       // store higher byte of result at 41H
        CLR A               // clear contents of A
NEXT:   ADD A,@R0           // add number pointed by R0 with A
        JC AHEAD
        SJMP SKIP
AHEAD:  INC 41H             // If carry is generated increment contents of
                            // address 41H
SKIP:   INC R0              // point to next number
        DJNZ R2, NEXT       // repeat addition 10 times
        MOV 40H, A          // store LSByte of result at address 40H
```

Nested Loops

By using byte jumps we can repeat the loop for a maximum of 256 times (FFH). If we want to repeat the loop more than 256 times then we have to use a loop inside loop which is referred as *nested loop*. In such a case, we use two (or more) loop counters, refer Example 7.9.

Example 7.9

Read port P0 and send its value on port P2 five hundred times.

Solution:

The number 500 is greater than 256; therefore, we have to use nested loops, each with separate loop count. Let us take 50 as the loop count for outer loop and 10 for inner loop. The total loop count is, therefore, 50 x 10 = 500.

```
        MOV R1, #10         // outer loop count
OUTER:  MOV R2, #50         // inner loop count
INNER:  MOV A, P0           // read the value of P0 and send value to P2
        MOV P2, A
        DJNZ R2, INNER      // inner loop repeat 50 times
        DJNZ R1, OUTER      // outer loop repeat 10 times
```

One important application of loops is in generating time delay using software. Delay generation using software is discussed in detail in Section 7.5 (Time-delay generation using timers is discussed in Chapter 14).

THINK BOX 7.3

Consider the following loop structure.

```
        MOV R2, #COUNT
LOOP:   …
        ..
        DJNZ R2, LOOP
```

What should be the value of COUNT to have maximum iterations of the above loop?

00H. Because DJNZ instruction first decrements the value of specified register (R2 in this case) and then check it for zero.

7.3 | CALLS AND SUBROUTINES

While developing larger programs, many times, we may require it to perform a task (or subtask) repeatedly. Instead of writing group of instructions for this task repeatedly, we can write these instructions as subprograms separately from the main program. This group of instructions is called *subroutine*. The subroutines are used by the main program many times as and when required. When a subroutine is required to be executed, a jump is made to the first instruction of the subroutine. The jump to the subroutine is more commonly referred as a *call*. Upon completion of the subroutine, another jump is made to the calling program (main) to resume the operation. This jump back is referred as *return*. The return is always made to instruction immediately next to the instruction for call.

A subroutine offers following advantages:

1. They simplify program-development process, they allow larger programs to be divided into smaller modules, these modules may be developed independently to speed up the development process; module may contain single subroutine or more than one subroutines.
2. Subroutines may be reused and therefore they save memory space.
3. Modular approach makes debugging and testing of the program easier and therefore saves time and money required for the development.

The subroutines are also referred as *routines* or *procedures*. The only disadvantage of using subroutine is that they reduce the speed of execution of a program because extra time is required in switching between main program and subroutines.

In the 8051, there are two instructions for calling a subroutine, LCALL (long call) and ACALL (absolute call), and RET instruction for return. The formats of these instructions are given below:

LCALL *add16*	// call the subroutine at address *add16* located anywhere in the entire
	// program memory space of 64KB, also save (push) the address of the
	// next instruction following LCALL (return address) on to the stack

LCALL DISPLAY	// call a subroutine DISPLAY, save return address on the stack

ACALL *add11*	// call the subroutine at address *add11* located on same page as the next
	// instruction, also save the address of the next instruction following
	// ACALL (return address) on to the stack

ACALL COMPARE	// call a subroutine COMPARE, save return address on the stack
RET	// return to the calling program by retrieving (pop) the return address from the stack

Before we understand the above instructions, we need to understand the relation of these instructions with the stack because these instructions use the stack.

1. Relation of Calls and Stack

A call (ACALL or LCALL) causes a jump to address where called subroutine is located. After completing a subroutine, the main program (calling program) execution should resume at instruction next to the call instruction. The stack automatically keeps track of where microcontroller is supposed to return after executing the subroutine. When we call a subroutine, the address of the instruction next to the call instruction is automatically saved on to the stack; this address is called return address. At the end of a subroutine, the return (RET) instruction will load return address from the stack to PC to resume the execution of the calling program. The stack pointer (SP) register is used to access the stack. Stack pointer always points to top of the stack, i.e. last memory address accessed.

The process of calling a subroutine and returning from it to resume the operation of the calling program using ACALL instruction, is described in the following steps.

(a) ACALL instruction will save the return address (PC) on the stack using two push operations: lower byte at address SP+1, and higher byte at SP+2. (Stack pointer is automatically incremented before pushing each byte).
(b) Address of subroutine is placed in the PC.
(c) Subroutine is completed.
(d) A RET instruction at the end of the subroutine will retrieve the return address to the PC from the stack using two pop operations. (Stack pointer is automatically decremented after each pop operation).
(e) Calling (main) program resumes its operation from the next instruction after ACALL.

The operation of ACALL instruction is illustrated in Figure 7.4.

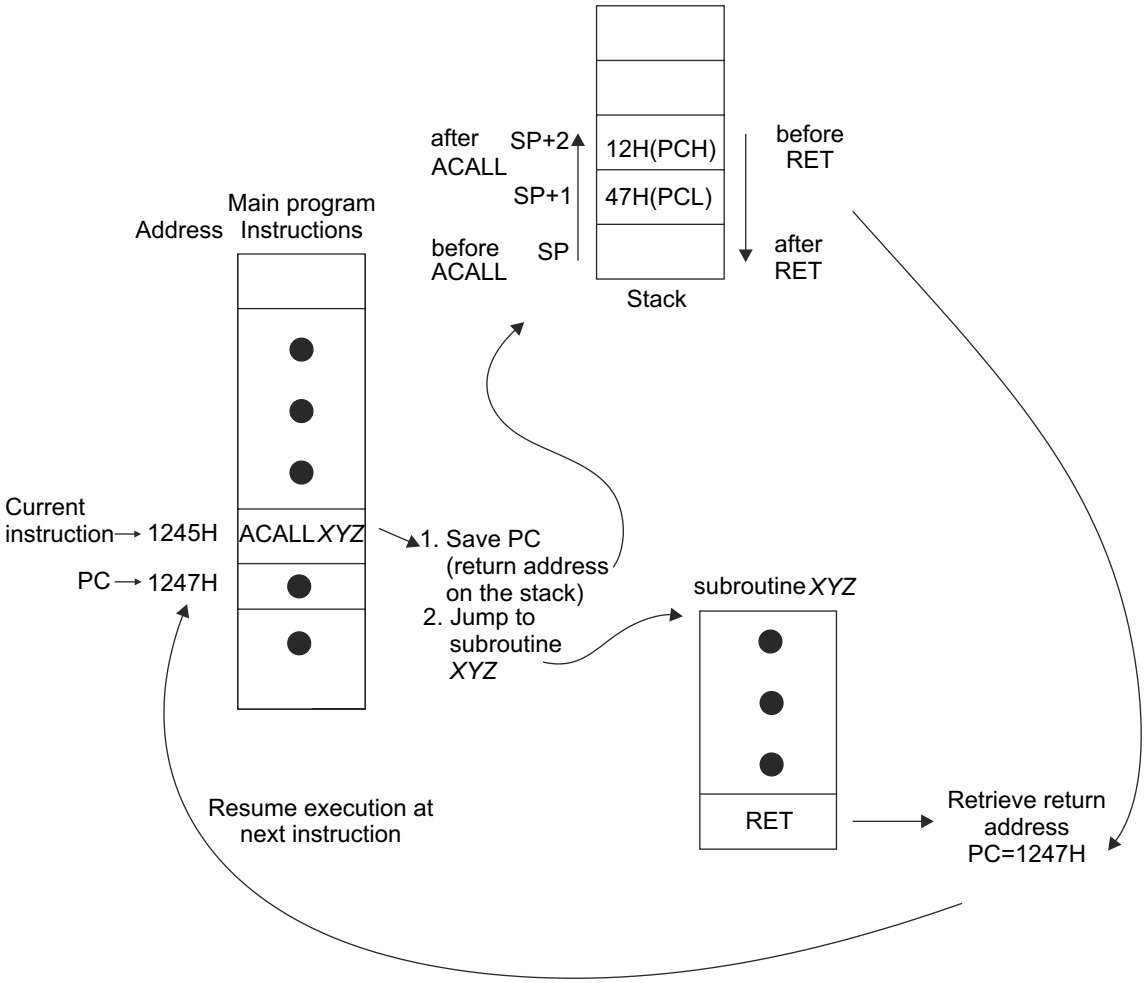


Fig. 7.4 *Operation of ACALL instruction*

The operation of LCALL is exactly same as the ACALL instruction. The only difference it has is that it can call subroutine anywhere in the entire program space of 64Kbytes, while ACALL can call the subroutine within a page of 2Kbytes in which instruction following ACALL is located.

ACALL is a 2-byte instruction, while LCALL is a 3 byte instruction. The way 11-bit address is specified for ACALL instruction is exactly similar to the AJMP instruction (refer Example 7.3)

Example 7.10

Illustrate the use of ACALL instruction to use the subroutine in a program.

Solution:

Assume that we want to write a very simple subroutine to add two numbers stored at addresses 40H and 41H, and place the result at address 42H.

```
MOV SP, #50H            // initialize SP
ACALL ADDITION          // call subroutine ADDITION
…
…
```

```
ADDITION: PUSH ACC          // save the registers (A) used in the subroutine
          PUSH PSW          // save PSW because carry may be affected
          MOV A, 40H        // add numbers
          ADD A, 41H
          MOV 42H, A        // save result at address 42H
          POP PSW           // retrieve the saved registers
          POP ACC
          RET               // return to calling program
```

Note the use of PUSH and POP instructions in above program. They are used to save the registers or memory locations used by a subroutine to perform its internal operations and retrieve the saved contents before returning back to the calling program. In given program, A is saved on stack because it is used by the subroutine, which may contain data of main program and PSW is saved because addition operation may change carry flag. Once all the tasks of subroutine are completed, the saved data is retrieved in same registers. It is done by POPing data in a reverse sequence than a PUSHing because stack is Last In First Out (LIFO) memory.

These extra instructions should always be written to make sure that subroutine does not modify accidently any important data (in register or memory location) used by a main program. Therefore, it is recommended to write subroutines which are transparent to main program, i.e. everything is same before and after a subroutine is called (and executed), this will be very much helpful when a larger program is developed in modules and each module is developed by a different person because the other person should be able to use the subroutine directly without knowing inner details of the subroutine.

Example 7.11

Illustrate the use of LCALL instruction to use subroutine in a program.

Solution:

(i) Assume that we want to send numbers 00H to FFH in increment of one to P2 after every 1 second, assume subroutine 'DELAY' is already available which generates a delay of 1 second.

```
          …
BACK:     MOV P2, A
          LCALL DELAY       // call subroutine 'DELAY'
          INC A
          SJMP BACK
DELAY:    …
          …
          RET               // return to main program
```

(ii) The rectangular wave of 25% duty cycle is to be generated on port pin P1.0. Assume that delay subroutine is available.

```
BACK:  SETB P1.0            // set P1.0 pin, High portion of rectangular wave
       LCALL DELAY          // call DELAY subroutine
       CLR P1.0             // clear P1.0 pin low portion of rectangular wave
       LCALL DELAY          // call DELAY subroutine 3 consecutive times to get
                            // low portion 3 times larger than high time to get 25% duty cycle
       LCALL DELAY
       LCALL DELAY
       SJMP BACK            // repeat operation forever
```

Example 7.12

Write a subroutine to an convert 8-bit binary number stored in the Accumulator into an equivalent BCD number.

Solution:

Refer Example 9.1 for the explanation of conversion process of binary to BCD.

```
            …
            MOV A, #30H       // binary number to be converted
            ACALL BIN2BCD     // call conversion routine
            …
            MOV A, #85H       // other number to be converted
            ACALL BIN2BCD     // call again conversion routine
            SJMP HERE         // skip subroutine
BCD2BIN:    MOV B, #64H       // divisor (100)
            DIV AB            // A=00 quotient, B=30H remainder (these values are for first number –30H)
            MOV R1, A         // store 100's digit in R1 (unpacked BCD digit)
            MOV A, B          // copy remainder in to A for next division
            MOV B, #0AH       // next divisor (10)
            DIV AB            // A=4, B= 8 (these values are for first number –30H)
            MOV R2, A         // 10's digit (unpacked BCD digit)
            MOV R3, B         // 1's digit (unpacked BCD digit)
            RET               // return to main program
HERE:       SJMP HERE         // end of program; loop forever
```

Note that the result obtained by the above program is unpacked BCD digits. We may convert this unpacked digits into packed digits by rotate and OR operations if required.

THINK BOX 7.4

Which op-code is undefined in the 8051?

A5.

The other examples of subroutines are given throughout remainder of the chapters.

2. Cautions while Developing Subroutines

Subroutines allow powerful and efficient way of developing modular programs; however they require more efforts from programmer/system designer during a program development. There are few simple cautions that have to be considered by a programmer; they are listed below.

(a) Always terminate subroutine with RET instruction: The RET instruction will resume operation in a main program. The program execution will not return to the main program when RET instruction is omitted and, therefore, the program will give an erroneous result.

(b) Parameter passing and context saving: Save registers (or memory locations) that are modified (used) by the subroutine/s for its internal operations. It is very common to forget to save the registers that are used by subroutines. This may overwrite or modify the important data in a main program resulting in strange behavior of the program. It is preferred to save all the registers used in the subroutine and PSW just at the beginning of the subroutine (unless application demands to change the specific register), and at the end of a subroutine, retrieve all the saved registers. To speed up the process of context saving (or switching), switch the register bank, which relieves the program from saving the registers R0 to R7. This is one of the major reasons for providing register banks in the 8051.

(c) Context retrieving: Another common error is to save registers onto the stack and then forget to retrieve them all off the stack before exiting the subroutine. An unequal number of save and retrieval of registers will result in return at wrong address. Therefore always make sure that equal number of PUSH and POP instructions are used.

The program of Example 7.12 is rewritten with proper context saving and retrieving in Example 7.13.

Example 7.13

Write a subroutine with context saving and retrieving to convert 8-bit binary number into equivalent BCD number.

Solution:

The binary number to be converted is placed into internal RAM address 10H before calling the subroutine. The three unpacked BCD digits will be stored in R1, R2and R3 of bank 3.

```
            MOV SP, #60H          // initialize the stack pointer at higher address
            …
            …
            MOV 10H, #30H         // binary number to be converted
            ACALL BIN2BCD         // call the conversion routine
            …
            MOV 10H, #85H         // other number to be converted
            ACALL BIN2BCD         // call again conversion routine
            SJMP HERE             // skip subroutine
BCD2BIN:    PUSH ACC              // save A on the stack
            PUSH B                // save B on the stack
            PUSH PSW              // save PSW on the stack
            SETB D3               // switch to register bank 3
            SETB D4
            MOV A, 10H            // get number to be converted
            MOV B, #64H           // divisor (100)
            DIV AB                // A=00 quotient, B=30 H remainder (these values are for first number –30H)
            MOV R1, A             // store 100's digit in R1 (unpacked BCD digit)
            MOV A, B              // copy remainder into A for next division
            MOV B, #0AH           // next divisor (10)
            DIV AB                // A=4. B= 8 (these values are for first number –30H)
            MOV R2, A             // 10's digit (unpacked BCD digit)
            MOV R3, B             // 1's digit (unpacked BCD digit)
            POP PSW               // retrieve PSW (switch back to original register bank)
            POP B                 // retrieve B
            POP ACC               // retrieve A
            RET                   // return to main program
HERE:       SJMP HERE             // end of program; loop forever
```

THINK BOX 7.5

Who do you think should save the registers: a main program (caller) or subroutine (callee) when calling a subroutine (or ISR)?
Both may save the registers. When a caller is saving the registers, it may save the registers which are needed by it or it may save the registers changed by the subroutine, but this requires the knowledge of internal details of subroutine (which is time consuming and difficult). Moreover, this approach will not work with ISRs.

When a subroutine is saving the registers (used by it), it will work for both subroutines and ISRs. In this approach, the subroutine need not have details of the main program and the instructions to save the registers are required to be written only once. Since all registers used in a subroutine are required to be saved, some registers may be unnecessarily saved. Thus, this approach is better and easy.

THINK BOX 7.6

What should be minimum size of the subroutine if it is to be called four times in a program and it is to save the program size?

Assuming that LCALL instruction is used for calling the subroutine. Each time the subroutine is called, 3 bytes are required for calling the subroutine and one byte for return, therefore 13 bytes are occupied if the subroutine is called four times (4x 3 for LCALL + 1 for RET). If the subroutine is of 4 or more bytes (excluding RET), it will occupy 16 bytes (or more) if called 4 times. Therefore minimum size of the subroutine is 4 bytes if is to save the program size for the given condition.

7.4 STACK INITIALIZATION AND OVERFLOW

The reset value of the stack pointer (SP) is 07H, therefore, the first location used for the stack operation is 08H (because SP is automatically incremented by 1 before saving data on the stack). We can use internal RAM addresses 08H to 7FH for the stack, however addresses 08H to 1FH are used by register banks and addresses 20H to 2FH are bit addressable, therefore we should not use this area for the stack (if they are being used by a program). Therefore addresses 30H to 7FH may be used for the stack. The initial portion of this memory area is normally used for storing data, therefore the usual practice is to initialize SP above data area. For example, it may be initialized with address 50H when programmer has reserved the addresses 30H to 50H for storing data.

The 8051 has limited size of the stack, i.e. maximum size of the stack may be 128 bytes (00H to 7FH if these locations are not used for other purpose), therefore make sure that value of SP never exceeds 7FH, otherwise stack will use area reserved for SFRs and overwrite their contents, moreover, all locations between 80H to FFH are not physically present, and data will be lost and program may reach irrecoverable state.

When the value of SP is greater than 7FH, it is referred as *stack overflow* and programmer must make sure that it is always avoided.

THINK BOX 7.7

Internal RAM location with address FFH does not exist physically in the 8051. What will be the contents of A after execution of the instruction MOV A, 0FFH?

Since address FFH does not exist physically, the contents of A are unpredictable.

THINK BOX 7.8

What is the limitation of using only internal RAM as a stack memory in the 8051?

Since internal RAM is only 128 bytes, theoretically maximum size of the stack can only be 128 bytes (practically even less) which is very small.

More care has to be taken by programmer to avoid stack overflow.

THINK BOX 7.9

Can we use SFRs for the stack?

No. Because all the locations between 80H to FFH do not exist physically. Moreover, general data should not be written in the SFRs because it may disturb the operations of the peripheral devices.

7.5 TIME-DELAY GENERATION USING SOFTWARE

Execution of each instruction requires certain number of machine cycles and each machine cycle in the 8051 requires 12 clock cycles (12 oscillator cycles). Time period of a machine cycle depends on frequency of crystal connected to the system (or frequency of external clock signal when on-chip oscillator is not used to generate the clock signal). For example, if 12 MHz crystal is connected to on-chip oscillator, the time period of one clock pulse is 1/12MHz = 0.08333 μs and time period of one machine cycle is 12 × 0.08333 μs = 1 μs. The desired time delay can be generated by wasting the time of microcontroller by executing group of instructions. Before we proceed further to develop the delays using software, let us discuss NOP instruction because it is often used in delay generation.

NOP (No Operation)

NOP is a 1-byte instruction and requires one machine cycle execution time. No operation is performed by this instruction. This instruction only updates the PC to point to the next instruction after execution. It is generally used to waste microcontroller's time in generating time delays using software.

Consider the following instructions:

Instructions	Machine cycles	Execution time (Crystal freq. =12 MHz)
MOV A, R0	1	1 μs
MOV R0, #00H	1	1 μs
NOP	1	1 μs
MOV 10H, 20H	2	2 μs
Total	**5**	**5 μs**

As shown above, the total time required to execute all instructions is 5 μs. Therefore, we can say that these instructions have generated a delay of 5 μs. To generate larger delays, the instructions can be repeated using loops. For example, consider the following instructions:

```
        MOV R0, #0FFH
HERE:   DJNZ, R0, HERE
```

The first instruction requires 1 machine cycle and is executed only once, the second instruction requires 2 machine cycles and it is executed 255 times; therefore, total machine cycles required to complete above two instructions are 1 + (255 × 2) = 511. Assuming crystal frequency equal to 12 MHz, the time required will be 511 μs (0.5 ms approx). To generate even higher delays more instructions can be repeated in a loop or nested loops can be used. It is illustrated in Example 7.14.

Example 7.14

Find the time required to execute (or delay generated by) the following instructions. Assume crystal frequency is 12 MHz.

```
        MOV R0, #249
THERE:  NOP
        NOP
        DJNZ R0, THERE
        NOP
        NOP
        NOP
```

Solution:

The time required by instructions can be calculated as follows.

	Instructions	Execution time (μs)
	MOV R0, #249	1
THERE:	NOP	249 (1 x 249)
	NOP	249 (1 x 249)
	DJNZ R0, THERE	498 (2 x 249)
	NOP	1
	NOP	1
	NOP	1
	Total	1000 μs.

It requires 1000 μs or 1 ms time.

Note that NOP is useful for making adjustments to get exact time delays.

Example 7.15

How much time will be required to execute instructions in Example 7.14 if crystal frequency is 6 MHz?

Solution:

Since crystal frequency is half, it will require double time to execute, i.e. 2000 μs = 2 ms.

Example 7.16

Find time delay generated by the following instructions. Assume crystal frequency is 12 MHz.

```
        MOV R0, # 20
THERE:  MOV R1, # 250
```

```
HERE:  DJNZ R1, HERE
       DJNZ R0, THERE
```

Solution:

It will generate a delay of 1+ 20 x ((1+ (2 x 250)) + 2) =10601 μs.

Example 7.17

Modify the program in Example 7.16 to get an exact delay of 10 ms.

Solution:

```
                MOV R0, # 20
THERE:          MOV R1, # 248
HERE:           DJNZ R1, HERE
                DJNZ R0, THERE
                MOV R0, #09
HERE1:          DJNZ R0, HERE1
```

The delay generated will be exactly 10 ms. Verify using the following expression.

[1+ 20 x ((1+ (2 x 248)) + 2)]+ [1+ (2 x 9)]

Summary of Program-Flow Control Instructions

Unconditional and conditional jumps (byte jumps) are summarized in Tables 7.2 and 7.3 respectively.

Table 7.2 *Unconditional jump instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
SJMP **rel**	Jump to **rel**	SJMP **rel** SJMP **ABC**			
AJMP **addr11**	Jump to **addr11**	AJMP addr11 AJMP **PQR**			
LJMP **addr16**	Jump to **addr16**	LJMP **addr16** LJMP **AGAIN**			
JMP @A + **DPTR**	Jump to **A+ DPTR**		JMP @**A+ DPTR** JMP @**A+ DPTR**		
ACALL **addr11**	Call subroutine at addr11	ACALL **addr11** ACALL **ROUTINE**			
LCALL **addr16**	Call subroutine at addr16	LCALL **addr16** LCALL **AGAIN**			
RET	Return from subroutine	RET			
RETI	Return from interrupt	RETI			
NOP	No operation	NOP			

Table 7.3 *Conditional jump (byte jumps) instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
JZ **rel**	Jump if A= 0	Accumulator only			
JNZ **rel**	Jump if A≠ 0	Accumulator only			
DJNZ <BYTE>, **rel**	Decrement & jump if not zero	DJNZ direct, **rel** DJNZ 10, **NOW**		DJNZ Rn,**rel** DJNZ R0,**NEXT**	
CJNE A, **<BYTE>, rel**	Jump if A≠ **<BYTE>**	CJNE A, **direct, rel** CJNE A, **12, ABC**			CJNE A,**#data,rel** CJNE A,**#10H, AB**
CJNE <BYTE>, **#data, rel**	Jump if <BYTE> ≠ #data		CJNE @ Ri,**#data,rel** CJNE @ R1,**#20H, AB**	CJNE Rn,**#data,rel** CJNE R2,**#10H,NEXT**	

POINTS TO REMEMBER

- The program-flow control instructions make the program more flexible and versatile as required by real-world applications.
- A jump instruction changes the content of program counter with a new program address usually referred as destination address. This causes program execution to begin at destination address.
- The unconditional jump does not test any condition and jump is always taken.
- The 8051 support three types of unconditional jumps: short, absolute and long jump. These jumps differ in range over which the jump can be taken.
- The advantage offered by the relative jump is that it allows relocation and requires only one-byte address to be specified as a part of an instruction op-code. The limitation is that destination address must be within +127 to –128 bytes with respect to the PC.
- AJMP and ACALL are the special instructions for which there are eight op-codes.
- All conditional jumps are short relative jumps.
- There is no zero flag in the 8051, the instructions JZ and JNZ checks the Accumulator for the zero.
- The instruction DJNZ decrements first, then checks for zero and it does not affect any of its operands.
- In the 8051, the instructions DJNZ and CJNE are used to repeat the loop for a fixed number of times, while instruction JZ/JNZ and all bit jump instructions are used to repeat the loop until a flag is set to desired state.
- Use of subroutines (modular approach) makes debugging and testing of a program easier.
- PUSH and POP instructions are used to save the registers or memory locations used by a subroutine to perform its internal operations and retrieve the saved contents before returning back to the calling program.
- The number of PUSH and POP instructions in subroutines should be equal and their sequence must be opposite because the stack is last-in first-out memory.
- The desired time delay can be generated by wasting microcontroller time by executing a group of instructions.

OBJECTIVE QUESTIONS

1. The jump instruction with short absolute address occupies _____ bytes in memory.
   (a) 1 byte (b) 2 bytes (c) 3 bytes (d) 4 bytes
2. The jump instruction with long address occupies _____ bytes in memory.
   (a) 1 byte (b) 2 bytes (c) 3 bytes (d) 4 bytes
3. The addressing mode used in the AJMP instruction is,
   (a) absolute addressing (b) long addressing
   (c) indexed addressing (d) relative addressing
4. The call (ACALL/LCALL) instruction stores the return address,
   (a) on the stack (b) in the stack pointer
   (c) in the program counter (d) in the DPTR register
5. Which of the following registers are modified by the jump instructions?
   (a) A (b) DPTR (c) PC (d) all
6. Which of the following registers are modified by the POP E0 instruction?
   (a) SP (b) A (c) PC (d) all
7. A program flow control instruction always modify,
   (a) PC (b) SP (c) DPTR (d) all
8. An instruction is required to provide a backward jump to a location at offset of 100 bytes and with respect to address of jump instruction, it is preferred to use,
   (a) AJMP (b) SJMP (c) LJMP (d) none
9. The destination address for AMP instruction must be,
   (a) within the same page of 2 Kbytes (b) within the same page of 4 Kbytes
   (c) within the same page of 8 Kbytes (d) within the same page of 16 Kbytes

10. The operation performed by JB bit. rel instruction is,
    (a) if bit=1, PC=PC+*rel* (b) if bit=0, PC=PC+*rel*
    (c) if bit=1, PC=PC+2 (d) if bit=0, PC=PC+3
    (PC is pointing to the next instruction)
11. The operation performed by JNC rel instruction is,
    (a) if c=1, PC=PC+*rel* (b) if c=0, PC=PC+*rel*
    (c) if c=1, PC=PC+2 (d) if c=0, PC=PC+2
    (PC is pointing to the next instruction)
12. JZ rel checks the_____ for decision making.
    (a) result of last operation (b) result of last arithmetic function only
    (c) A (d) zero flag
13. JBC bit, rel instruction may affect,
    (a) CY flag (b) OV flag (c) P flag (d) all of the above
14. CJNE A, 00H, NEXT will always modify,
    (a) Accumulator (b) R0 (c) CY (d) none
15. How many times will the following loop be repeated?
    REPEAT: MOV A, #02H
    JNZ REPEAT
    (a) 0 (b) 1 (c) 2 (d) infinite
16. The following program code will read data from port 1 and copy it to port 2, and it will stop looping when bit 7 of port 2 is set
    REPEAT: MOV A, P1
    MOV P2, A
    JB P2.7, REPEAT
    (a) true (b) false
17. Subroutines are usually written in,
    (a) stack memory (b) code memory (c) internal data memory (d) external data memory
18. PUSH instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) decrement SP by 1 (d) decrement SP by 2
19. ACALL instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) decrement SP by 1 (d) decrement SP by 2
20. LCALL instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) increments SP by 3 (d) do not affect SP

Answers to Objective Questions

1. (b)	2. (c)	3. (a)	4. (a)	5. (c)	6. (d)	7. (a)
8. (b)	9. (a)	10. (a), (d)	11. (b), (c)	12. (c)	13. (d)	14 (d)
15. (d)	16. (b)	17. (b)	18. (a)	19. (b)	20. (b)	

REVIEW QUESTIONS WITH ANSWERS

1. What is meant by destination address?

A. It is an address where the program execution will start after execution of jump or call instruction.

2. What is meant by unconditional jump? List the unconditional jump instructions of the 8051.

A. When program execution is altered by instruction without checking any condition, it is referred as unconditional jump, i.e. the unconditional jump does not test any condition and jump is always taken. SJMP, AJMP and LJMP are the unconditional jump instructions of the 8051.

3. Discuss the advantages offered by relative jump instructions.

A. The program written using relative jumps is relocatable, i.e. a program that is written using relative jumps can be placed (loaded) anywhere in the program address space without reassembling. Second advantage is that only 1 byte is required to specify the relative address of the destination location which saves the program bytes and increases the speed of execution.

4. The backward relative addresses are specified in 2's complement in the 8051. True/False.

A. True.

5. What is special about AJMP and ACALL instructions in the 8051 instruction set?

A. They both have 8 different op-codes, while other instructions have only one op-code.

6. Where is JMP @A+DPTR instruction commonly used?

A. It is commonly used to implement the jump tables.

7. How does the 8051 support JZ and JNZ instructions though it has no zero flag?

A. JZ and JNZ instructions check the contents of Accumulator for zero.

8. Which conditional jump instructions are based on the contents of Accumulator?

A. JZ and JNZ

9. What is meant by mnemonic ACALL and LCALL?

A. ACALL: Absolute call (jump can be taken anywhere in the 2K page)
LCALL: Long call (jump can be taken anywhere in the entire program memory of 64K)

10. How many bytes are required by SJMP, AJMP and LJMP instructions?

A. 2 bytes for SJMP and AJMP, 3 bytes for LJMP instruction.

11. What is meant by relative address?

A. Relative address means how far (in terms of bytes) the destination address is, with respect to address where the jump instruction occurs, i.e. with respect to next instruction after the jump instruction (address of the PC).

12. Upon execution, the jump instructions modify the contents of program counter. True/False.

A. True.

13. What is a loop?

A. The looping is a programming technique used to repeat the sequence of instructions several times until certain conditions are met (or to repeat forever without checking any condition)

14. What are the common requirements of the loops?

A. A counter which specifies the number of times a loop should be repeated and pointers, which will point to different data in each iteration.

15. Stack is a LILO (last in last out) memory. True/False.

A. False, it is LIFO (last in first out) memory.

16. All conditional jump instructions are relative jumps. True/False.

A. True.

17. What is the size of the logical page in the 8051? What is the address range of page 3?

A. Page size is 2Kbytes. Address range of page 3 is 1800H to 1FFFH.

18. List the instructions used to call the subroutines.

A. ACALL and LCALL are instructions used to call the subroutines.

19. "JNC HERE" is a _____ byte instruction.

A. 2.

20. What is the status of specified bit after executing JBC instruction?

A. The status of specified bit is 0.

21. "JNZ HERE" instruction, checks value of ___.

A. Accumulator.

22. What is the key difference between POP and RET instructions?

A. POP retrieves one byte from a stack while RET retrieves two bytes.

EXERCISE

1. Compare execution of ACALL and LCALL instructions.
2. What is meant by return address?
3. Which flag is affected after executing JC instruction?
4. List the advantages and disadvantages of using subroutines.
5. What does the mnemonics CJNE and DJNZ stand for?
6. What is meant by conditional jump? List the conditional jump instructions of the 8051.
7. Show with suitable example how relative address is calculated in the SJMP instruction.
8. What is limitation of the SJMP instruction compared to the AJMP instruction?
9. Show with suitable example how absolute address is calculated in the AJMP instruction.
10. What care has to be taken by a programmer when the AJMP instruction occurs at page boundary?
11. Explain how the ACALL and LCALL modify the contents of the program counter.
12. Define the term *subroutine*. How is it useful?
13. Describe the operation of the RET instruction.
14. Discuss the role of the stack in execution of the ACALL and LCALL instructions.
15. How program resumes its operation after completion of the subroutine?
16. Write a program to place value FFH in to internal RAM addresses 10H to 20H using loop.
17. What is meant by nested loop? Explain with a suitable example.
18. How does the microcontroller know where to return to after executing the subroutine?
19. Find the number of times the following loop is repeated.

```
        MOV R2, #100
THERE:  MOV R3, # 50
HERE:   DJNZ R3, HERE
        DJNZ R2, THERE
```

20. Modify the above instructions to repeat loop for 5000 times.
21. Find the number of times the following loop is repeated.

```
THERE:  MOV R2, #100
        MOV R3, # 50
HERE:   DJNZ R3, HERE
        DJNZ R2, THERE
```

22. Mention the address range of all type of jump instructions.
23. Discuss the advantages and disadvantages of the relative addressing.
24. Compare the SJMP and AJMP instructions.
25. Write a subroutine to generate delay of 1 second. Assume crystal frequency is 12 MHz.
26. Modify above subroutine if crystal frequency is changed to 11.0592 MHz.
27. Numbers of PUSH and POP instructions in a subroutine should be equal. Justify.
28. Discuss how CJNE and DJNZ instructions are executed. Where are they commonly used?
29. Write a subroutine which converts BCD number stored in A to equivalent binary number. Store the result into A.
30. Modify the above subroutine if BCD number is stored at location pointed by R0 and store the result at location pointed by R1.
31. Write a subroutine to count the number of 1's in a byte. Main program reads the byte from port 1. Save the result into top of the stack.
32. Illustrate with a suitable example how parameters can be passed to the subroutine using pointers.
33. Illustrate with a suitable example how parameters can be passed to the subroutine using stack.
34. What is meant by stack overflow? Discuss the consequences of it and how it can be avoided?
35. What is meant by context saving and retrieving?

8

Look-Up Tables and Jump Tables

Objectives

- Discuss the need and advantages of look-up tables
- Show how to implement look-up tables
- Discuss how look-up tables are supported by the 8051
- Develop the programs to illustrate the use of look-up tables
- Show how to implement jump tables
- Develop the program to illustrate the use of jump tables

Key Terms

- 7-segment Codes
- BCD
- Code Conversion
- Faster Evaluation
- Look-up
- Pre-calculated Values

8.1 LOOK-UP TABLES AND THEIR USAGE

In many applications we need to convert one type of number to another type, for example:

1. We have to convert BCD number to 7-segment code if we want to display the BCD number on a 7-segment display.
2. ASCII to BCD conversion may be required for arithmetic operations when numbers are entered through ASCII keyboard. Conversely, BCD to ASCII conversion is required to display the result of arithmetic operation on LCD screen.
3. Binary to BCD conversion and vice versa for arithmetic operations.
4. Binary to Gray number and vice versa to implement algorithm of K-map and tabulation method for simplification of Boolean functions.

For all the above examples, we can use some algorithm for the conversion. Alternatively, we can define a table (array) in the ROM that contains equivalent values in one format corresponding to numbers in the other format in one to one basis, i.e. tables will contain pre-calculated values in the desired format corresponding to each number in the original format. These tables are commonly referred as look-up tables, because for doing conversion we need to "LOOK UP" in the tables.

8.2 FASTER EVALUATION OF FUNCTIONS

Look-up tables are not only used for number conversions, but are also used for many other purposes. For example, consider we want to evaluate the following function,

$$F(x) = x^4 + 2x^3 + x^2 + x$$

for different values of x. If we write a program to evaluate the above function, the microcontroller will have to perform a total of 10 operations (three multiply operations for finding x^4, three multiply operations for $2x^3$, one multiply operation for x^2, and three addition operations for adding all four terms, the overhead of loading values in to A and B registers and storing intermediate results for each operation is neglected for simplicity, otherwise more operations will be required for the actual program).

To execute the above 10 operations, the 8051 requires 31 machine cycles (or 31μs, if crystal frequency is 12 MHz), which in turn reduces the speed of execution of the program.

Let us consider the alternate approach of using a look-up table to solve the same problem. Here, we have to pre-calculate the values of the function for different values of x, i.e.

$F(0) = 0 \quad = 0H$
$F(1) = 5 \quad = 5H$
$F(2) = 38 \quad = 26H$
$F(3) = 147 \quad = 93H$
…

We can use these pre-calculated values to define a look-up table in a memory, let's say at an address 0100H onwards as shown in Table 8.1.

Table 8.1 *Look-up table for $F(x) = x^4 + 2x^3 + x^2 + x$*

Memory address	Value
0100H	0H
0101H	05H
0102H	26H
0103H	93H
…	…

We can access this look-up table as and when required; consider the following instructions,

```
ORG 0000H
MOV DPTR, #0100H      // (2 cycles) load address of look-up table in DPTR
MOV A, #02H           // (1 cycle) value for which expression is evaluated
MOVC A, @A+DPTR       // (2 cycles) access look up table
…
ORG 0100H             // define look-up table at ROM address 0100H
DB 0H, 5H, 26H, 93H …
END
```

The earlier code sequence will evaluate the given function $F(x)$ for the value of x=2 in only five cycles. Therefore, when we want faster evaluation of complex functions, we should use look-up tables.

8.3 MISCELLANEOUS CONVERSIONS

The other situation when we need to use a look-up table is when it is difficult to define a relation (or equation) between two numbers that we want to convert from one to other.

For example, for BCD to 7-segment code conversion, there is no equation (or logic) for performing conversion. Therefore, for this conversion, we need to use look-up tables. So, look-up tables are used to perform complex data conversions and evaluation of functions.

The only disadvantage of using a look-up table is the requirement of extra memory to store values of the look-up table and the programmer has to take extra mental pain to pre-calculate the values in look-up tables, but only once!!!

THINK BOX 8.1

Identify other conversions or evaluations where look-up tables could be useful.

Look-up tables could be useful in evaluating trigonometric (finding sin, cos, etc.), exponential and logarithmic functions. In general, they could be helpful in evaluating any mathematical series.

8.4 THE 8051 AND LOOK-UP TABLES

The 8051 supports implementation of look-up tables by providing the following two instructions.

```
MOVC A, @A+DPTR  // A= (A+DPTR)
MOVC A, @A+PC    // A= (A+PC)
```

For both the instructions, A usually holds the number to be converted, which is usually referred as an *offset*; the DPTR (or PC) holds the starting address of the look-up table, which is also referred as *base address*. Since these two are 16-bit registers, they allow the look-up table to be placed anywhere in the entire program memory. Normally, the PC is used as a base address for small tables that are placed in the body of a program, i.e. the look-up table is placed on the address relatively nearer to the instruction MOVC A, @A+PC, whereas DPTR is used to access data from large tables that are normally placed at end of the program. The use of look-up tables in the 8051 applications is illustrated in the following examples.

Example 8.1

Find the square of a number present in A. Assume the number to be in range 0 to 10d.

Solution:

In this program, a look-up table is defined nearer to the instruction MOVC A, @A+PC and within body of the program.

```
        ORG 0000H
        MOV A, #05H       // find square of 05 H (or place any other number)
        ADD A, #02H       // Look-up table is placed 2 bytes ahead w.r.t. MOVC instruction.
        MOVC A,@A+PC      // get square of value from look-up table
        SJMP DONE         // skip look-up table

        DB 00H            // 0^2
        DB 01H            // 1^2
        DB 04H            // 2^2
        DB 09H            // 3^2
        DB 10H            // 4^2
```

```
        DB 19H          // 5^2
        DB 24H          // 6^2
        DB 31H          // 7^2
        DB 40H          // 8^2
        DB 51H          // 9^2
        DB 64H          // A^2
DONE:   SJMP DONE
        END
```

Note that after execution of the program, A will contain the square of 5. See how the look-up table values are defined directly within the program. This method is used when we do not want to divide program space into code and data spaces and this ***may*** save code space when compared with MOVC A, @A+DPTR because we do not waste any code space between instructions and beginning of the look-up table.

Example 8.2

Write a program to convert BCD number to equivalent 7-segment code.

Solution

Assume common cathode 7-segment display,

```
        ORG 0000H
        MOV DPTR, #0100H    // load address of look-up in DPTR
        MOV A, #06H         // find code for BCD 06
        MOVC A,@A+DPTR      // fetch equivalent 7 segment of 6 code from
                            // look-up table
HERE:   SJMP HERE

        ORG 0100H           // store look-up table at address 100H onwards.
        DB 3FH              // code for 0, considering 'a' segment is connected to LSB
        DB 06H              // code for 1
        DB 5BH              // code for 2
        DB 4FH              // code for 3
        DB 66H              // code for 4
        DB 6DH              // code for 5
        DB 7DH              // code for 6
        DB 07H              // code for 7
        DB 7FH              // code for 8
        DB 6FH              // code for 9
        END
```

For instruction MOVC A, @A+DPTR, the offset register is 8 bit (A). So, size of the look-up table may be up to 256 bytes. To increase the size beyond 256 bytes, DPTR should be changed in increments of 256.

Look-up tables are also used for storing the strings. The following program illustrates how strings are stored into and accessed from the 8051 code memory.

Example 8.3

Store the string "MICROCONTROLLER" in the 8051 code memory address 100H onwards. Write a program to read the string, one character at a time and send it to port P1. Provide delay of 1s before accessing the next character.

Solution:

```
        ORG 0000H
        MOV DPTR, #STRING   // load address of look-up in DPTR
        MOV R3, #0FH        // count for 15 characters
NEXT:   CLR A               //
        MOVC A, @A+DPTR     // fetch character byte from look-up table
        MOV P1, A           // send character on P1
```

```
            ACALL DELAY          // delay of 1s
            INC DPTR             // point to next character
            DJNZ R3, NEXT
HERE:       SJMP HERE

DELAY:      MOV R0, #10          // delay of 1s approx
THERE1:     MOV R1, # 200
THERE:      MOV R2, # 250
HERE1:      DJNZ R2, HERE1
            DJNZ R1, THERE
            DJNZ R0, THERE1
            RET
            ORG 0100H
STRING: DB "MICROCONTROLLER"  // store string at address 100H onwards
            END
```

Example 8.4

Write a program to evaluate sinusoidal function (sin θ) for the angle (θ) 0° to 360° in step of 10°.

Solution:

We need to evaluate the sinusoidal function for many motor control and PWM (pulse width modulation) based applications.

We know that value of sine function varies between –1 to +1 (with fractional values in between). Since there is no direct provision in the 8051 to process fractional numbers, we need to map the range –1 to +1 to 0 to 255, i.e. –1 value is mapped to 0, 0 is mapped to 128, +1 is mapped to 255. By performing this mapping, the 8051 can directly and easily manipulate these values because it is 8-bit microcontroller. The required mapping (scaling) of the values can be done by following formula.

Scaled value =127.5+127.5 sin θ

The scaled values for sine function for angles 0° to 360° in steps of 10° are given in Table 8.2. Note that the scaled values are rounded off to the next higher number.

Table 8.2 *Look-up table for sine function*

Angle (θ)	sin θ	Scaled value 127.5 + 127.5 sin θ		Angle (θ)	sin θ	Scaled value 127.5 +127.5 sin θ
0	0.00	128		190	–0.17	105
10	0.17	150		200	–0.34	84
20	0.34	171		210	–0.50	64
30	0.50	191		220	–0.64	45
40	0.64	209		230	–0.77	30
50	0.77	225		240	–0.87	17
60	0.87	238		250	–0.94	8
70	0.94	247		260	–0.99	2
80	0.98	253		270	–1.00	0
90	1.00	255		280	–0.98	2
100	0.98	253		290	–0.94	8
110	0.94	247		300	–0.86	17
120	0.87	238		310	–0.76	30
130	0.77	225		320	–0.64	46
140	0.64	210		330	–0.50	64
150	0.50	191		340	–0.34	84
160	0.34	171		350	–0.17	106
170	0.17	150		360	0.00	128
180	0.00	128				

The program to evaluate the sine function for angles 0° to 360° in step of 10° and then send the values to port 1 is given below.

```
            ORG 0000H
            MOV DPTR, #LOOK-UP          // address of look-up table
REPEAT:     MOV R1, #36                 // 36 values in look-up table for 0° to 360°
            CLR A
NEXT:       MOV R3, A                   // save A
            MOVC A,@A+DPTR              // fetch value in look-up table
            MOV P1, A                   // send to port 1
            MOV A, R3                   // retrieve A
            INC A                       // next entry in look-up table
            DJNZ R1, NEXT
            SJMP REPEAT                 // repeat cycle forever

LOOK-UP:    DB 128, 150, 171, 191, 209, 225, 238, 247   // look up table
            DB 252, 255, 253, 247, 238, 225, 209, 191
            DB 171,150, 128, 105, 84, 64, 45, 30
            DB 17, 8, 2, 0, 2, 8, 17, 30
            DB 46, 64, 84,106
            END
```

To appreciate the advantage offered by a look-up table in this program, let us consider size and execution time for the program. The size of the program is 52* bytes (16 bytes for program + 36 bytes for look-up table) and execution time for one complete cycle from 0º to 360º is 294 machine cycles. The equivalent program written in the C language using standard library functions is given in Example 12.42, its size is 1454 bytes and it requires 164186 machine cycles to complete one cycle from 0º to 360º!!! Even if the equivalent program is written in assembly language without a look-up table, it would have required comparatively larger size and execution time because we need to evaluate complete sine series (at least first few terms of the series).

To make the look-up table compact, we can use the fact that $\sin \theta = -\sin(\pi + \theta)$, i.e. the values of sin 0° to sin 180° are same as values from sin 180° to sin 360° except for the sign; therefore we need to make the look-up table only up to values of sin 180°, and the same values can be reused with negative sign for sin 180° to sin 360°.

Refer **Examples 19.14 and 19.20** to understand how this program can be useful in generating sine waves using a D/A converter.

*Refer section 12.12 to know how to find size and execution time for a program.

Example 8.5

Write a program to convert a hexadecimal number into its equivalent ASCII number.

Solution:

For simplicity of explanation, we will consider only a one-digit hexadecimal number (00 to 0F). The table of hexadecimal digits and equivalent ASCII numbers is given in Table 8.3.

Table 8.3 *Hexadecimal number to ASCII conversion*

Hexadecimal number	ASCII code (HEX)		Hexadecimal number	ASCII code (HEX)
00	30		08	38
01	31		09	39
02	32		0A	41
03	32		0B	42
04	34		0C	43
05	35		0D	44
06	36		0E	45
07	37		0F	46

The programs for required conversion are given below.

- ***Without using Look-up Table***

```
        ORG 0000H
        MOV A, # HEX_NUMBER   // single digit hex number to be converted
        CJNE A, #09H, NXT     // check if the digit is between 0-9 or A-F
        SJMP NUM
NXT:    JC NUM                // if digit is between 0-9 add 30H
        ADD A, #37H           // if digit is between A-F add 37H
        SJMP HERE
NUM:    ADD A, #30H
HERE:   SJMP HERE             // end
        END
```

- ***With Using Look-up Table***

```
        ORG 0000H
        MOV A, # HEX_NUMBER     // single digit hex number to be converted
        MOV DPTR, #LUT          // starting address of Look-up table
        MOVC A, @A+DPTR         // read equivalent ASCII value from look-up table
HERE:   SJMP HERE
 LUT:   DB 30H, 31H, 32H, 33H, 34H, 35H, 36H, 37H
        DB 38H, 39H, 41H, 42H, 43H, 44H, 45H, 46H
        END
```

The program without look-up table requires 17 bytes and 8/10 machine cycles, while the program with a look-up table requires 24 bytes and 7 machine cycles. Therefore, from the above examples, we can conclude that a look-up table always provides faster execution of a program and sometimes they require more memory.

Refer Examples **17.3, 19.14 and 19.20** for more applications of look-up tables.

8.5 | JUMP TABLES

The 8051 has a special jump instruction which can jump dynamically anywhere in the entire program memory space. Its format is,

```
JMP @A+DPTR          // jump to address formed by adding A with the DPTR
```

This instruction is used to implement jump tables, i.e. to select at runtime one out of many jump addresses depending upon the value of A or DPTR. This makes programs more dynamic and flexible.

Example 8.6

Illustrate the use of JMP @A+DPTR by implementing a jump table.

Solution:

Assume that we want to perform different arithmetic operations as given below based on value of A.

If A=0, call addition subroutine, or If A=1, call subtraction subroutine,

or, If A=2, call multiplication subroutine, or If A=3, call division subroutine

```
                MOV A, #__              // load A with desired value (0, 1, 2, 3)
                MOV   DPTR, # J_TABLE   // load DPTR with address of jump table
                RL A*                   // multiply A by 2
                JMP   @A + DPTR         // select subroutine based on value of A
                …

J_TABLE:        AJMP  ADDITION
                AJMP  SUBTRACTION
```

```
                AJMP    MULTIPLICATION
                AJMP    DIVISION
                …
ADDITION:       …
                …

SUBTRACTION:    …
                …

MULTIPLICATION: …
                …

DIVISION:       …
                …
```

If A = 0, execution proceeds to label ADDITION or if A=3, execution proceeds to label DIVISION (*AJMP instruction is 2-byte instruction)

THINK BOX 8.2

Which decision-making statement of the high-level language does the instruction JMP @A+DPTR represent?
The switch statement.

POINTS TO REMEMBER

- Look-up tables are used commonly for code conversions.
- Look-up tables are used for faster evaluation of complex functions.
- Look-up table is used when it is difficult to define a relation (or equation) between two numbers that we want to convert from one to other.
- The 8051 supports implementation of look-up tables by MOVCA, @A+DPTR and MOVCA, @A+PC instructions.
- The only disadvantage of a look-up table is that it requires more memory.
- Look-up tables are commonly stored in a program memory (ROM).
- Jump tables make programs more dynamic and flexible.
- Jump tables are implemented using JMP @A+DPTR instruction.

OBJECTIVE QUESTIONS

1. Which of the following instructions can be used to access a look-up table?
   (a) MOVX A, @DPTR (b) MOVX A, @R0
   (c) MOVC A, @ A+DPTR (d) MOVX @DPTR, A
2. Look-up tables are generally stored in,
   (a) stack (b) internal RAM (c) external RAM (d) code memory
3. Look-up tables are commonly used in,
   (a) data conversions (b) complex trigonometric evaluations
   (c) complex exponential evaluations (d) all of the above
4. The JMP @A+DPTR instruction is equivalent to,
   (a) for loop (b) switch statement (c) if else statement (d) while loop

5. Jump tables can also be implemented using,
   (a) DJNZ instructions (b) CJNE instructions (c) MOVC instructions (d) None of the above

Answers to Objective Questions

1. (c) 2. (d) 3. (d) 4. (b) 5. (b)

EXERCISE

1. Define the term *look-up table*.
2. List the advantages and disadvantages of the look-up tables.
3. 'Look-up tables are stored in ROM.' Justify.
4. What are the applications of look-up tables?
5. List and explain with a suitable example the instructions of the 8051 used to access the look-up tables.
6. Develop a look-up table for seven-segment codes for common anode configuration.
7. 'Look-up tables require more memory.' Justify.
8. Prove with a suitable example that the use of look-up tables result in a faster evaluation of the function.
9. When is the use of the instruction MOVC A, @A+PC preferred?
10. List the assembler directives used to define the look-up tables.

9

Code Conversions, Array Processing and 16 Bit Arithmetic

Objectives

- Discuss the need of code conversions
- Explain the logic and techniques used for code conversions
- Develop the programs for various code conversions
- Explain the techniques used in an array processing
- Develop the programs for array processing
- Develop the programs for 16-bit arithmetic

Key Terms

- 16-bit Arithmetic
- 1's/2's Complement
- Array Processing
- Ascending/Descending Order
- ASCII
- BCD
- Block Transfer
- Code Conversions
- Counters
- Factorial
- Gray Numbers
- Loops

The microcontroller/processor is a programmable device used in the embedded systems to carry out desired operation/s as per requirements of an application. The operations are realized by developing the algorithms using instructions and features of the microcontroller. The algorithms are implemented using some common programming techniques like looping, counting, look-up tables, subroutines and of course arithmetic/logical data manipulations. The most common real-life operations frequently require code conversions, array processing and multi-byte arithmetic operations.

This chapter contains many real-life programming examples for code conversions, array processing and 16-bit (or multi-byte) arithmetic, using instructions and programming techniques discussed in the previous chapters.

9.1 | CODE CONVERSIONS

Based on an application, different number systems and codes are used in the microcontroller-based systems. For example, when an ASCII keyboard and a CRT monitor (or LCD) is used as an input and output device respectively, the ASCII codes are used by such systems. Seven-segment LEDs require seven-segment codes for all the digits. On the other hand, most of the processing within a microcontroller is performed in a binary system and in some cases, arithmetic operations are performed with the BCD numbers. Therefore, frequently, we require conversion between the ASCII, BCD, and binary number systems based on input and output devices used in a system. This section provides various programming examples for the code conversions.

Example 9.1

Write a program to convert an 8-bit binary number to its equivalent BCD number.

Solution:

For an 8-bit binary number, at most three digits are required to represent its equivalent BCD number. For example,

$FFH_{Binary} = 255_{BCD}$ (three digits)

$64H_{Binary} = 100_{BCD}$

The conversion is achieved by successive division of a binary number by 100, 10, and 1. (In general, it is 10^N, 10^{N-1}, 10^{N-2}, …,10^1.) Note that division by 1 may be skipped.

Consider a binary number 11111111B (FFH).

Divide the given binary number by 100.

FFH/64H = 2 (quotient – first digit)
37H remainder,

Divide remainder from the above operation by 10.

37H/0AH = 5 (quotient – second digit)
5H remainder

Divide remainder from the above operation by 1.

5H/1 = 5 (quotient – third digit)
= 0 remainder

Arrange all the quotients (digits) from left to right and we will get 255, the required BCD number.

Assume that the binary number is first stored in A.

```
ORG 0000H
MOV A, #30H          // binary number to be converted (place any other number in A)
MOV B, #64H          // divisor (100)
DIV AB               // A=00 quotient, B=30 H remainder
MOV R1, A            // store 100's digit in R1 (unpacked BCD digit)
MOV A, B             // copy remainder into A for next division
MOV B, #0AH          // next divisor (10)
DIV AB               // A=4, B= 8
MOV R2, A            // 10's digit (unpacked BCD digit)
MOV R3, B            // 1's digit (unpacked BCD digit)
HERE: SJMP HERE
END
```

The result for given binary number is,
R1= 00
R2= 04
R3= 08

These three unpacked BCD digits can be converted to packed BCD (048).
Note: For a multi-byte binary number, the above operations can be performed in a loop to reduce the size of the program.

Example 9.2

Write a program to convert an 8-bit BCD number to its equivalent binary number.

Solution:

First, each digit should be unpacked. Then, the hundred's (100's) digit is multiplied with 100, the ten's digit is multiplied with 10 and the one's digit is multiplied with 1. Results of all operations (multiplications) are added to get the result in the binary. In general, 10^N's digit is multiplied with 10^N, 10^{N-1}'s digit is multiplied 10^{N-1} and so on up to 10^0's (one's) digit. And all results are added.

For example, consider the two-digit BCD number 29, for simplicity.

29 is unpacked and stored as 02 and 09.

02 is multiplied with 0AH and 09 is multiplied with 1H.

02 x 0AH = 14H, 09 x 1 = 09H

These two results are added

14H + 09H = 1D H= 00011101B which is the binary equivalent of 29 BCD.

(Multiply by 1 operation may be skipped.)

```
        ORG 0000H
        MOV R0, #29H*       // BCD number
        MOV A, R0           //
        ANL A, #0F0H        // mask lower nibble
        SWAP A              // copy upper nibble to lower nibble
        MOV B, #0AH         //
        MUL AB              // multiply 10's digit with 10
        MOV R2, A
        MOV A, R0
        ANL A, #0FH         // mask upper nibble
        ADD A, R2           // add results of multiplications
                            // A will contain the result (A= 1DH for given example)
HERE:   SJMP HERE
        END
```

Note: For a multi-digit BCD number, the above operations can be performed in a loop to reduce program size.

*The BCD number 29 should be written with suffix 'H' (29H), otherwise the assembler will consider it as decimal and automatically convert it to binary!!! The question may arise that if the assembler is doing conversion automatically then do why write a program for conversion!!! The reason is that the automatic conversion by assembler will not be useful when we work with numbers larger than two-digit BCD numbers and the other simple reason is that the above example is to illustrate the conversion process.

Example 9.3

Write a program to convert 2-digit (packed) BCD number into ASCII equivalent numbers.

Solution:

BCD to ASCII conversion is required when we want to display BCD numbers on standard output devices (LCD or monitor of the PC).

The ASCII equivalents of BCD digits are as follows:

BCD digit	Binary	ASCII (HEX)
0	0000	0011 0000 = 30
1	0001	0011 0001 = 31

(Contd.)

(Contd.)

2	0010	0011 0010 = 32
3	0011	0011 0011 = 33
4	0100	0011 0100 = 34
5	0101	0011 0101 = 35
6	0110	0011 0110 = 36
7	0111	0011 0111 = 37
8	1000	0011 1000 = 38
9	1001	0011 1001 = 39

First, each digit should be unpacked and as can be seen from the above table, each unpacked BCD digit is added with 30H (or ORed with 30H) to get an ASCII equivalent.

For example, BCD 25 should be unpacked as 02 and 05 and finally be converted to 32 and 35.

```
        ORG 0000H
        MOV R0, #29H        // BCD number
        MOV A, R0
        ANL A, #0F0H        // mask lower nibble
        SWAP A              // copy upper nibble to lower nibble
        ORL A, #30H         // add 30H with unpacked digit,
        MOV R1, A           // store most significant ASCII byte in to R1
        MOV A, R0
        ANL A, #0FH         // mask upper nibble
        ORL A, #30H         // add 30H with unpacked digit,
        MOV R2, A           // store least significant ASCII byte in to R2
HERE:   SJMP HERE
        END
```

The result is,

R1= 32H, ASCII for 2

R2= 39H, ASCII for 9

Example 9.4

Write a program to convert two bytes in ASCII to the equivalent 2-digit (packed) BCD number.

Solution:

We need to mask the upper nibbles of both ASCII numbers to remove 3 from the upper nibbles and then both unpacked digits are packed.

```
        ORG 0000H
        MOV A, # '3'        // ASCII number corresponding to higher nibble equivalent to 'MOV A, # 33H
        ANL A, #0FH         // mask upper nibble
        SWAP A
        MOV R1, A
        MOV A, # '5'        // ASCII number corresponding to lower nibble
        ANL A, #0FH         // mask upper nibble
        ORL A, R1           // pack both nibbles
HERE:   SJMP HERE
        END
```

The result is A=35H

Note that the ASCII numbers can be directly written in single quotes (''). The assembler will automatically convert them in equivalent binary (HEX) number.

Example 9.5

Write a program to convert an 8-bit binary number into its equivalent ASCII number.

Solution:

This conversion is required when we need to display numbers in decimal number system on standard output devices like the LCD or monitor of a PC.

First, the binary number is converted into a BCD number as in Example 9.1 and then the BCD number is converted into ASCII as discussed in Example 9.3.

Example 9.6

Write a program to convert given 8-bit binary number into its equivalent Gray number.

Solution:

To find the equivalent Gray number, the following operations are to be performed. Copy MSB of the binary number, G7=B7, G6=B7 EX-OR B6, G5= B6 EX-OR B5… G0= B1 EX-OR B0. To perform EX-OR operation between two adjacent bits, the number is copied into other register and shifted to left by one position through carry. Now, the original number and shifted number are EX-ORed (bit wise) with each other and the result is right shifted by one bit through carry.

Assume that 8-bit binary number is present in R0.

```
        ORG 0000H
        MOV A, R0          // binary number
        MOV B, A           // save original number
        RLC A              //EX-OR nearby bits of binary number
        XRL A, B
        RRC A
        MOV R1, A          // store result in R1
HERE:   SJMP HERE
        END
```

The result is,
If R0=07H Binary number
R1= 04H Gray number

Example 9.7

Write a program to convert an 8-bit Gray number into its equivalent binary number.

Solution:

To find the equivalent binary number, the following operations are to be performed.
B7=G7, now this B7 is EX-ORed with G6 to get B6, same way, B6 is EX-ORed with G5 (next Gray digit) to get B5, this process is repeated until B0 is found.
Assume that Gray number is stored in R0.
First, get this number into A and rotate A left by 1 bit without carry i.e. G7 is moved to LSB, and G6 to MSB. Since B7 = G7, B7 is in LSB. To get B6, EX-OR B7 (LSB) with G6 (MSB) of A.
Implement EX-OR operation between MSB and LSB (0E7H and 0E0H respectively) and store result in the C i.e.

C = (0E0H)' (0E7H) + (0E0H) (0E7H)'

(E0 is the bit address of LSB of A, and E7 is the bit address of MSB of A)

Thus, carry will contain derived binary bit B6 which is placed in LSB using rotate left through carry (derived bit B_n is always placed into LSB and EX-ORed with G_{n-1} which is always made available in MSB of A). The above process is repeated 7 times to get the result.

```
        ORG 0000H
        MOV A, R0          // copy gray number in A
        RL A               // G7 is copied to LSB
        MOV R2, #07H       // count for 7 repetitions
REPEAT: MOV C, 0E0H        // perform EX-OR operation between bits Bn and Gn-1
        ANL C , /0E7H
        MOV 10H,C
        MOV C, 0E7H
```

```
        ANL C, /0E0H
        ORL C, 10H
        RLC A              // result will be stored in A
        DJNZ R2, REPEAT    // repeat until B0 is found
HERE:   SJMP HERE
        END
```

The result is,
If R0 = 04H Gray number
A= 07H Binary number

9.2 ARRAY PROCESSING

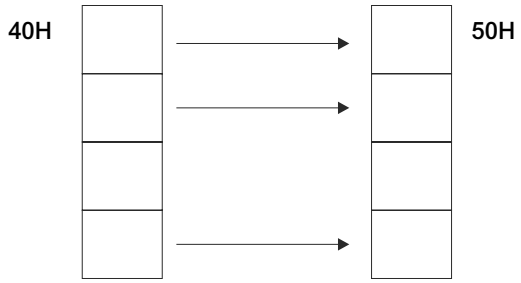
In array processing, usually we have to repeat the similar task on all elements of the array. The looping is commonly used to repeat the task. The looping allows us to develop concise and efficient programs. The most common requirements for the array processing are the loop count and the pointers. The *loop count*, usually referred as a *counter*, determines the number of times a task has to be repeated and pointers point to different elements of an array in each iteration. This section provides various programming examples for the array processing.

Example 9.8

Write a program to copy a block of data (array) from one location to other location in the internal RAM.

Solution:

Assume that the source array starts from address 40H and the destination array at 50H. Size of the array is 10 (0AH) bytes. The operation to be performed is shown as follows.



Here, pointers to source and destination arrays are initialized with starting addresses. Bytes from source to destination array are moved and both pointers are incremented to point to the next byte in the respective arrays. This process is continued till all bytes (10) bytes are transferred.

```
        ORG 0000H
        MOV R0, #40H       // initialize R0 with address of source array (source pointer)
        MOV R1, #50H       // initialize R1 with address of destination array
        MOV R2, #0AH       // initialize R2 with number of bytes in array
NEXT:   MOV A, @R0         // move byte from source array to destination array
        MOV @R1, A
        INC R0             // increment array pointers
        INC R1
        DJNZ R2, NEXT      // repeat until all elements are moved
HERE:   SJMP HERE
        END
```

Example 9.9

Rewrite the above program to copy an array from external RAM to internal RAM.

Solution:

Assume that the source array is stored at the external RAM address 1000H onwards and the destination array at 50H onwards. Size of the array is 10 (0AH) bytes.

```
        ORG 0000H
        MOV DPTR, #1000H    // initialize DPTR with address of source array (source pointer)
        MOV R1, #50H        // initialize R1 with address of destination array
        MOV R2, #0AH        // initialize R2 with number of bytes in array
NEXT:   MOVX A, @DPTR       // move byte from source array to destination array
        MOV @R1, A
        INC DPTR            // increment array pointers
        INC R1              // repeat until all elements are moved
        DJNZ R2, NEXT
HERE:   SJMP HERE
        END
```

Example 9.10

Write an assembly-language program to count positives, negatives, and zeros in the array of signed numbers stored in an external RAM starting at address 1000H. Size of the array is 100 bytes.

Solution:

For signed numbers, MSB indicates the sign of the number. If MSB= 1, the number is negative, and for MSB=0, number is positive. Thus, MSBs of all numbers are observed by moving it to carry flag using rotate left through carry instruction. To determine the number of zeros, the elements are moved into the Accumulator register. The contents of the Accumulator are checked for zero using JZ/JNZ instructions. The elements of the array are accessed one at a time using pointer.

```
ORG 0000H
        CLR PSW.3           // select register bank 0
        CLR PSW.4           //
        MOV R1, #00H        // used to store count of positive numbers
        MOV R2, #00H        // count of negative numbers
        MOV R3, #00H        // count of zeros
        MOV DPTR, #1000H    // pointer to starting address of array
        MOV R4, #64H        // size of array, 64H (100) bytes
NEXT:   MOVX A, @DPTR       //read byte from array
        JZ ZERO             // if number is zero increment count for zero (R3)
        RLC A               // move MSB in to carry
        JC NEG              // if carry set, number is negative
        INC R1              // otherwise number is positive
        SJMP SKIP
NEG:    INC R2
        SJMP SKIP
ZERO:   INC R3
SKIP:   INC DPTR            // point to next array element
        DJNZ R4, NEXT       // check next element of array
HERE:   SJMP HERE           // stop when all elements are checked
        END
```

Example 9.11

Write an assembly-language program to count even and odd numbers in an array of numbers stored in external RAM, starting at the address 1000H. Size of the array is 100 bytes.

Solution:

The number is ODD if the LSB is 1, and EVEN if it is 0. Thus, the LSBs of all numbers are observed by moving it to the carry flag using rotate right through carry instruction.

```
        ORG 0000H
        MOV R1, #00H        // count for odd numbers
        MOV R2, #00H        // count of even numbers
        MOV R4, #64H        // size of array, 64H (100) bytes
        MOV DPTR, #1000H    // pointer to starting address of array
NEXT:   MOVX A, @DPTR       // read byte from array
        RRC A               // move LSB into carry
        JC ODD              // number is odd if MSB is 1
        INC R2              // otherwise even
        SJMP SKIP
ODD:    INC R1
SKIP:   INC DPTR            // point to next array element
        DJNZ R4, NEXT       // check next element of array
HERE:   SJMP HERE           // stop when all elements are checked
        END
```

Example 9.12

Write a program to find the address of a given byte in an array of numbers stored in external RAM starting at address 1000H. Size of the array is 10 bytes.

Solution:

Assume that the byte to be searched is stored at the RAM address 10H.

```
        ORG 0000H
        MOV DPTR, #1000H    // pointer to start of array
        MOV R0, #0AH        //size of the array
BACK:   MOVX A, @DPTR       // read element of array
        CJNE A, 10H, NEXT   // compare array element with number
        MOV R2, DPL         // address of byte is stored in R3, R2
        MOV R3, DPH
        SJMP HERE
NEXT:   INC DPTR            // increment pointer to next element of array
        DJNZ R0, BACK       // repeat until byte found or end of array
HERE:   SJMP HERE           // stop
        END
```

Example 9.13

Modify the above program (in Example 9.12) to find the number of times a given byte occurs in an array.

Solution:

All elements of the array are compared with the given number one by one and count is incremented when they are found to be equal.

```
        ORG 0000H
        MOV R2, #00H        // counter to store result
        MOV DPTR, #1000H    // pointer to start of array
        MOV R0, #0AH        // size of the array
```

```
BACK:   MOVX A, @DPTR       // read element of array
        CJNE A, 10H, NEXT   // compare array element with number
        INC R2
NEXT:   INC DPTR            // increment pointer to next element of array
        DJNZ R0, BACK       // repeat until byte found or end of array
HERE:   SJMP HERE           // stop
        END
```

Example 9.14

Write a program to find the largest number from a given array.

Solution:

Assume that the array is present in internal RAM and starts at address 50H. Size of the array is 0AH (10).

Memory pointer is initialized with the first byte of an array. The largest number will be stored at the internal RAM address 10H. Initially, assuming that first element is the largest number and is read through a pointer and stored in the internal RAM 10H, the memory pointer is incremented to point to the next element. These two numbers are compared and the larger number is placed at the address 10H. This way, each element is compared with the number in the 10H address.

```
            ORG 0000H
            MOV R2, #09H          // repeat operation* = array size-1
            MOV R0, #50H          // initialize R0 as array pointer
            MOV 10H,@R0           // store element of array into RAM address 10H
REPEAT:     INC R0                // point to next element in array
            MOV A,@R0             // read next element
            CJNE A, 10H, AHEAD    // compare two numbers
            SJMP NEXT             // if equal, no action
AHEAD:      JNC EXCHANGE          // if contents of 10H is smaller, exchange with larger
            SJMP NEXT             // if contents of 10H is larger, check with next element
EXCHANGE:   XCH A, 10H
NEXT:       DJNZ R2, REPEAT       // repeat operation for all elements
HERE:       SJMP HERE
            END
```

The result, i.e. the largest number, is stored at address 10H.

Example 9.15

Modify the program of Example 9.14 if we assume the array is stored in the code memory.

Solution:

The array can be defined in code memory using 'DB' directive as shown in a program. For simplicity of program development, it is assumed that a temporary variable (R3, initialized with 0) contains a maximum value. This variable is compared with each element of the array. In each comparison (iteration of loop), the larger of these two values is stored in a variable (R3). Finally, we will have maximum value in R3.

```
ORG 0000H
            MOV R2, #0AH          // repeat operation = array size
            MOV DPTR, #ARRAY      // initialize DPTR as array pointer
            MOV R3, #00H          // temporary variable to hold max number
REPEAT:     CLR A
            MOVC A, @A+DPTR       // read element of array
            MOV 10H, A            // save A
            CLR C
            SUBB A, R3            // compare array element with temp variable (R3)
EXCHANGE:   JC NEXT               // if temp variable (R3) is larger, no action
```

* For 10 elements, only nine comparisons are required.

```
        MOV R3, 10H          // if temp variable (R3) is smaller, store A in R3
NEXT:   INC DPTR             // point to next element in array
        DJNZ R2, REPEAT      // repeat operation for all elements
HERE:   SJMP HERE

ARRAY:  DB 32H, 024H, 45H, 76H, 23H, 39H, 35H, 87H, 21H, 56H
                             // array of 10 numbers in code memory
        END
```

Example 9.16

Write a program to find the smallest number from a given array.

Solution:

In the program of Example 9.14, replace the instruction 'JNC EXCHANGE" with "JC EXCHANGE"

Initially, assuming that the first element is the smallest number and the memory pointer is incremented to point to the next element. These two numbers are compared and smaller number is placed at the address 10H.

Example 9.17

Write a program to arrange a given array of 10 elements in an ascending order.

Solution:

Assume that the array is stored at internal RAM address 50H.

In the first iteration, the array pointer is initialized with the address of the first element, the first element of array is compared with second, smaller of these two will be placed at the first location, this way first element is compared with all elements one by one and thus, the smallest number is brought to the first location of the array. In the next iteration, the pointer is initialized with the address of the second element; now, this element is compared with rest of the array elements and in this way, the next smallest element is brought to the second element. This process is repeated until the pointer is moved across all the elements of the array.

The above steps are implemented using a loop-within-loop structure.

```
        ORG 0000H
        MOV R4, #09H         // count for outer loop
        MOV R0, #50H         // initialize R0 as a array pointer 1
AGAIN2: MOV B, R4            //
        MOV R2, B            // count for inner loop
        MOV A, R0
        INC A
        MOV R1, A            // initialize R1 as a pointer 2
AGAIN1: MOV 10H, @R0         // read elements
        MOV A, @R1
        CJNE A, 10H, AHEAD   // compare and exchange to get smaller number at
                             // location pointed by pointer 1
        SJMP NEXT
AHEAD:  JC EXCHANGE          // if contents of 10H smaller, exchange with larger
        SJMP NEXT
EXCHANGE: MOV @R0, A
        MOV @R1, 10H
NEXT:   INC R1
        DJNZ R2, AGAIN1      // repeat for inner loop
        INC R0               // pointer 1 to next element to get next smallest number
        DJNZ R4, AGAIN2      // repeat for outer loop
HERE:   SJMP HERE
        END
```

Example 9.18

Write a program to arrange a given array of 10 elements in descending order.

Solution:

Replace in the above program (Example 9.17) the instruction 'JC EXCHANGE" with "JNC EXCHANGE".

9.3 | 16-BIT OPERATIONS

The 8051 is an 8-bit microcontroller and supports 8-bit arithmetic and logical operations directly. To perform multi-byte operations, we need to manipulate 8-bit operations. This section provides various programming examples for 16-bit arithmetic and logical operations.

Example 9.19

Write a program to add two 16-bit numbers. Assume the first number is stored at internal RAM address 40H (MSByte) and 41H (LSByte), second number is at 42H (MSByte) and 43H (LSByte). Store the result at 51H (MSByte), 52H (LSByte) and store carry into 50H if any, otherwise clear 50H.

Solution:

The result of addition of two 16-bit numbers may require three bytes (if we store a carry into register).

```
        ORG 0000H
        MOV 50H, #00H       // clear contents address 50H to save carry.
        MOV A, 41H
        ADD A, 43H          // add LSBytes of both numbers
        MOV 52H, A          // store LSBytes of result into 52h
        MOV A, 40H
        ADDC A, 42H         // add MSBytes along with carry if any
        MOV 51H, A          // store MSBytes of result into 51h
        JNC HERE
        INC 50H             // store carry bit in 50H register
HERE:   SJMP HERE           // stop
        END
```

Example 9.20

Modify the above program to subtract two 16-bit numbers.

Solution:

Replace in the above program (Example 9.19) instruction "ADD A, 43H" with following two instructions.

```
        CLR C
        SUBB A, 43H
```

And replace "ADDC A, 42H" with "SUBB A, 42H"

The program for 16-bit subtraction is as follows:

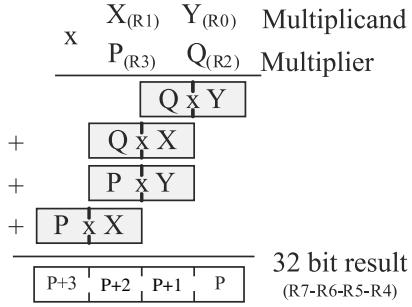
```
        ORG 0000H
        MOV 50H, #00H       // clear contents address 50H to save borrow.
        MOV A, 41H
        CLR C
        SUBB A, 43H         // subtract LSBytes
        MOV 52H, A          // store LSBytes of result into 52h
        MOV A, 40H
        SUBB A, 42H         // subtract MSBytes along with borrow if any
        MOV 51H, A          // store MSBytes of result into 51h
        JNC HERE
        INC 50H             // store borrow bit in 50H register
HERE:   SJMP HERE           // stop
        END
```

Example 9.21

Write a program to multiply two 16-bit numbers. The multiplier is stored in R1-R0 (least significant byte in R0). The multiplier is stored at R3-R2. Store the result at R7-R6-R5-R4.

Solution:

Since we are multiplying two 16-bit numbers, the maximum result can be of 32 bits (4 bytes). For this, we require four 8-bit multiplications (partial products) and the result of partial products are added to get 32-bit result. The program logic is described in the following figure. The register names in the parenthesis indicate where the particular data is stored.




```
        ORG 0000H
        MOV A, R0           // Q*Y
        MOV B, R2
        MUL AB
        MOV R4, A           // store LSByte of Q*Y at P
        MOV R5, B           // store MSByte of Q*Y at P+1
        MOV A, R3           // P*X
        MOV B, R1
        MUL AB
        MOV R6, A           // store LSByte of P*X at P+2
        MOV R7, B           // store MSByte of P*X at P+3
        MOV A, R2           // Q*X
        MOV B, R1
        MUL AB
        ADD A, R5           // add LSByte of Q*X to P+1
        MOV R5, A
        MOV A, B            // add MSByte of Q*X to P+2
        ADDC A, R6
        MOV R6, A
        MOV A, R7           // add carry to P+3 if any
        ADDC A, #00H
        MOV A, R3           // P*Y
        MOV B, R0
        MUL AB
        ADD A, R5           // add LSByte of P*Y to P+1
        MOV R5, A
        MOV A, B            // add MSByte of P*Y to P+2
        ADDC A, R6
        MOV R6, A
        MOV A, R7           // add carry to P+3 if any
        ADDC A, #00H
        MOV R7, A
HERE:   SJMP HERE
        END
```

Example 9.22

Add two 4-digit BCD numbers. Ignore carry after 16 bit (For simplicity).

Solution:

Assume first BCD number is present in DPTR, and MSByte of the second number is present in B and LSByte in A. Store the result in DPTR.

```
        ORG 0000H
        MOV DPTR, #1243H    // four digit BCD number
        MOV A, #56H         // second number in B and A
        MOV B, #78H
        ADD A, DPL          // add lower bytes
        DA A                // adjust A for BCD
        MOV DPL, A
        MOV A, B
        ADDC A, DPH         // add higher bytes
        DA A                // adjust A for BCD
        MOV DPH, A
HERE:   SJMP HERE           // stop
        END
```

The result is 9099 in DPTR.

Example 9.23

Write a program to shift a 16-bit number to the right by one digit.

Solution:

Assume that a 16-bit number is stored at 10H (MSByte) and 11H (LSByte).

```
        ORG 0000H
        MOV A, 11H          // LSByte
        RRC A               // move LSB of LSByte into carry and rotate right
        MOV 11H, A          // store back
        MOV A, 10H          // MSByte
        RRC A               // move LSB of LSByte into MSB of MSByte
        MOV 10H, A
        MOV A, 11H
        MOV ACC.7, C        // move LSB of MSByte into MSB of LSByte
        MOV 11H, A
HERE:   SJMP HERE           // stop
        END
```

9.4 | OTHER PROGRAMS

Example 9.24

Write a program to count the number of 1's and 0's in a byte stored at internal RAM address 50H. Store count for 1's in R0 and that for 0's into R1.

Solution:

All bits (one at a time) of the given number are brought into carry flag using the rotate instruction. Then, the carry flag is checked using JNC or JC instruction, if it is one, the count of 1's will be incremented. The process is repeated for all bits of number, and finally we will have the number of 1's in a byte, then after the number of 0's will be equal to 8 minus number of 1's.

```
        ORG 0000H
        MOV   R0, #00H          // initialize count register for 1's
        MOV   R1, #00H          // initialize count register for 0's
```

```
        MOV   R2, #08H          // counter for eight bits
        MOV   A, 50H            // data byte
NEXT_BIT:RLC  A                 // rotate data bit by bit and check for 1's
        JNC   NEXT
        INC   R0
NEXT:   DJNZ  R2, NEXT_BIT
        MOV   A, #08H           // no. of 0's = 8- no. of 1's
        CLR   C
        SUBB A, R0
        MOV   R1, A             // store count for 0's into R1
HERE:   SJMP HERE               // stop
        END
```

Example 9.25

Modify the above program if the data byte is stored at external RAM address 1000H. Store count for 1's at external RAM location 1001H and that for 0's into 1002H.

Solution:

The MOVX instruction has to be used to access data from external RAM address.

```
        ORG 0000H
        MOV   R0, #00H          // initialize temporary count register for 1's
        MOV   R2, #08H          // counter for eight bits
        MOV   DPTR, #1000H      // pointer to data byte
        MOVX A, @DPTR           // read data byte from external RAM address
L1:     RLC   A
        JNC   L2
        INC   R0
L2:     DJNZ  R2, L1
        MOV   A, #08H
        CLR   C
        SUBB A, R0
        MOV   DPTR, #1002H      // pointer to store 0's
        MOVX @DPTR, A           // store count for 0's
        MOV   DPTR, #1001H      // pointer to store 1's
        MOV A, R0
        MOVX @DPTR, A           // store count for 1's
HERE:   SJMP HERE               // stop
        END
```

Example 9.26

Write a program to find 1's and 2's complement of a given number.

Solution:

Assume that the number is stored at the internal RAM address 10H. Result of 1's complement is stored at the address 11H and that of 2's complement is stored at 12H.

```
        MOV  A, 10H         // load number from RAM address 10H
        CPL  A              // complement all bits of A, i.e. 1's complement
        MOV 11H, A          // store result (1's complement) at address 11H
        ADD  A, #01H        // 2's complement= 1's complement + 1
        MOV 12H, A          // store result (2's complement) at address 12H
```

Note: For 8 bits, the CPL instruction works only with A register.

Therefore, data must be moved to A to perform complement operation.

Example 9.27

Write a program to mask upper nibble of an 8-bit number.

Solution:

Assume that number is in R0 register, store the result in R1.

Masking means 'to hide'. This can be achieved by ANDing the bit with 0, i.e. the result after masking is 0, irrespective of the bit value. The operation of masking is useful when we want to perform operation on only a few bits of a byte. So we mask bits which are not required in an operation. (Sometimes masking is achieved by ORing bit with 1, i.e. the result after masking is 1 irrespective of bit value.)

```
MOV A, R0          // number
ANL A, #0FH        // mask upper nibble by making it 0000
MOV R1, A          // store the result in R1.
```

Example 9.28

Write a program to find factorial of a number.

Solution:

Factorial of a number is calculated by the equation.

$N! = N \times N - 1 \times N - 2 \ldots \times 2 \times 1$

For example,

$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

First, we need to check if the number is 0 or 1. If it is so, the result is 1. (This checking may be skipped.)

The following steps are to be implemented to find the factorial.

Find $N - 1$ and $N \times N - 1$

Assign the above result as

Temp_result = $N \times N - 1$

N (new) = temp_result

$N - 1$(new) = $(N - 1) - 1$

Repeat the above three steps until (N–1) =1

```
        ORG 0000H
        MOV R0, #NUMBER          // number
        MOV A, #00H              // initialize result registers A=LSByte, B=MSByte
        MOV B, #00H
        CJNE R0, #00, AHEAD      // if N=0, N! =1
        SJMP SKIP
        CJNE R1, #00, AHEAD      // if N=1, N! =1
        SJMP SKIP
AHEAD:  MOV A, R0
        MOV R1, A
REPEAT: DEC R1                   // N–1
        MOV B, R1
        MUL AB                   // N x N–1
        CJNE R1, #01H, REPEAT    // repeat until N–1= 1
        SJMP HERE
SKIP:   MOV A, #01
HERE:   SJMP HERE
        END
```

A more effective way of finding factorial is using the recursion, but, to develop programs using recursion in an assembly language is more difficult. Recursion can easily be implemented with the support of a compiler, i.e. in high-level languages. For simplicity, it is not discussed.

Example 9.29

What is the maximum number for which factorial can be found using program of Example 9.28.

Solution:

The above program can be used to find factorial only up to 5!!! For larger numbers, subroutine for 16-bit multiplication has to be written.

POINTS TO REMEMBER

- PC keyboard, CRT monitors and LCD uses ASCII codes.
- Most of the processing within a microcontroller is performed in a binary system.
- The looping allows us to develop concise and efficient programs.
- The most common requirements for the array processing are loop count and pointers.
- The 8051 is an 8-bit microcontroller and supports 8-bit arithmetic and logical operations directly.
- To perform multi-byte operations, we need to manipulate 8-bit operations.

EXERCISE

1. Write an assembly-language program to convert a 16-bit binary number to its equivalent BCD number.
2. Write an assembly-language program to multiply two 16-bit numbers.
3. Write an assembly-language program to find the smallest number from an array of 10 numbers.
4. Write a program to arrange the array in descending order.
5. Write a program to find factorial of a number using stack.
6. Where are the ASCII codes commonly used?
7. Where are the BCD numbers commonly used?
8. Write a program to perform subtraction of two 16-bit numbers.
9. Write a program to divide 16-bit number by 8-bit number.
10. Write a program to find average of elements of array of 10 elements.
11. Write a program to count number of words in paragraph.
12. Write a program to convert four-digit BCD number into binary.
13. Write a program to check if an array of 10 numbers is arranged in descending order or not.
14. Write assembly-language instructions equivalent to the following C statements.

    (a)
    ```
    if      i <=10
            y += 1
    else
            y -= 1
    ```

    (b)
    ```
    if  ( i <10) && ( j >20)
        y += 1
    else
        y – = 1
    ```

15. Write a program to insert a given number in an array (arranged in ascending order) at a position such that the array remains arranged in an ascending order.

10

Timing and Instruction Execution

Objectives

- Discuss the significance and generation of clock pulses
- Classify the instructions with respect to number of bytes and machine cycles
- Show and explain the timing diagrams of various instructions
- Discuss the data and code memory access cycles with timing diagrams
- Illustrate the execution of most common instructions with data flow diagrams

Key Terms

- ALE
- Clock Pulse
- Code Memory Read Cycle
- Crystal Oscillator
- Data Flow Diagram
- Data Memory Read Cycle
- Instruction Execution
- Instruction Register
- Instruction Timing
- Machine Code Bytes
- Machine Cycle
- Op-code Fetch
- PC Incrementer
- Program Address Register
- Program Counter
- RAM Address Register
- State
- System Clock

The microcontrollers/processors are sequential devices; therefore, they require series of pulses for their operation. The instruction execution is timed according to these pulses. In this chapter, we will discuss how these pulses are generated, timings of execution for all types of instructions and data-flow diagrams for the most common instructions.

10.1 THE CLOCK PULSE

The clock is a repetitive sequence of pulses used to synchronize all the internal activities of a microcontroller/ processor. It is the smallest unit of time in which the microcontroller performs a part of the operation. The clock pulses are also referred as a *system clock*. The microcontroller will run from 0 Hz (static operation) to few tens of MHz clock frequency.

An oscillator circuit generates the clock pulses. The 8051 has an in-built crystal oscillator circuit (of course partial circuit) which can be used as a clock source if desired. We need to connect a resonant network (crystal + capacitors) to make it functional. The clock frequency is determined by crystal frequency. We may also use external pulses as the clock signal if required.

10.2 MACHINE CYCLE

It is the time required to completely execute a simple instruction (or partially execute a complex instruction). The number of machine cycles is fixed for a given instruction but varies from one instruction to another. In the 8051, an instruction may require one, two or four machine cycles depending upon the type of instruction.

The machine cycle of the 8051 contains 6 states, numbered S1 to S6; a state is the time required to perform subdivision of basic operation like read, write or decode. Each state requires two clock cycles referred as phase 1 (P1) and phase 2 (P2). Thus, an 8051 machine cycle requires 12 clock cycles (oscillator periods) named as S1P1, S1P2,… to S6P2. The arithmetic and logical operations are completed during Phase 1 and internal data transfers take place during Phase 2.

Nowadays, some variants of the 8051 are capable of executing instructions in a single clock cycle. The machine cycle of the 8051 is shown in Figure 10.1.

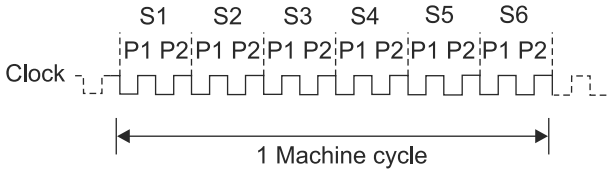


Fig. 10.1 *Machine cycle of 8051*

THINK BOX 10.1

Can we modify the number of clock cycles required in a machine cycle?
No. The manufacturer of a microcontroller chip decides it.

10.3 INSTRUCTIONS TIMING

The instruction set of the 8051 may be classified according to the number of machine cycles required to execute an instruction and number of machine code bytes for an instruction. There are six types of instructions as given below,

1. 1 byte–1 machine cycle instruction
   For example, MOV A, *Rn*
   MOV A, @*R*1
2. 1 byte–2 machine cycle instructions
   INC DPTR
   RET
3. 1 byte–4 machine cycle instructions
   MUL AB
   DIV AB
4. 2 byte–1 machine cycle instructions

MOV A, *direct*
ADD A, *direct*

5. 2 byte–2 machine cycle instructions
MOV *direct*, *Rn*
ACALL *addr11*

6. 3 byte–2 machine cycle instructions
CJNE A, *direct, rel*
LJMP *addr16*

10.3.1 1 Byte–1 Machine Cycle Instructions

The timing diagram for 1 byte–1 machine cycle is shown in Figure 10.2.

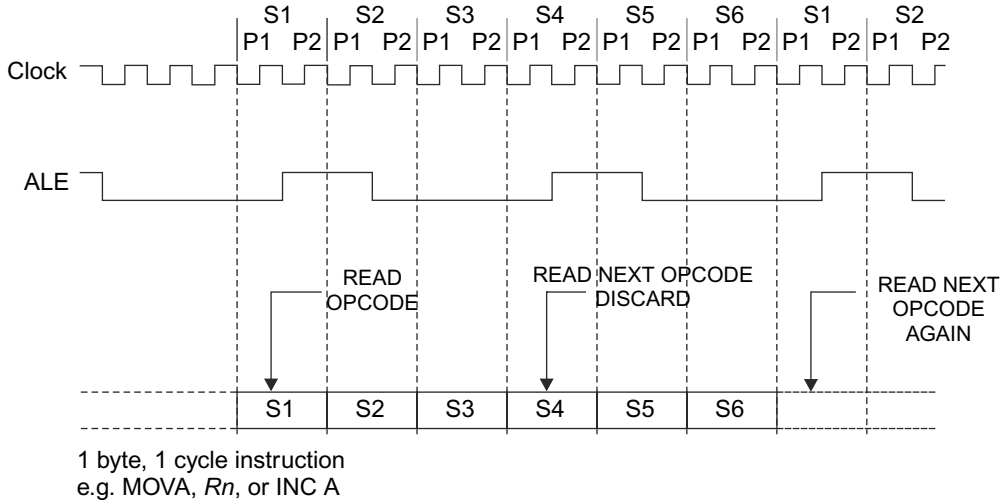


Fig. 10.2 *Timing diagram for 1 byte–1 machine cycle instructions*

Figure 10.2 shows the instruction timing with respect to internal states and phases. Since internal clock signals cannot be accessed directly, the clock (XTAL2 oscillator signal) and the ALE are shown for the reference. ALE is activated twice (except MOVX) during each machine cycle, during S1P2 and S4P2.

Execution of instruction starts at S1P2, when the op-code is fetched and placed into the instruction register. The op-code placed into the instruction register will be decoded (operation to be performed by an instruction is identified) by instruction decoder, and required control signals are generated to execute the instruction. Since there are two program fetches per machine cycle, the program counter is incremented by one and next byte (which will be the next op-code) will be fetched during S4. As the instruction is a 1 byte instruction, second byte will be ignored and program counter is not incremented further. The execution of the instruction completes at the end of S6P2. Therefore PC is effectively incremented by 1 during these instructions.

10.3.2 2 Byte–1 Machine Cycle Instructions

If the instruction is a two-byte instruction, the first byte (op-code) is fetched in similar way as discussed for 1 byte–1 machine cycle instruction. The program counter is incremented and the second byte is read during S4 of the same machine cycle and the program counter is incremented again. Therefore, PC is effectively incremented by 2 during these instructions.

The execution is complete at the end of S6P2 as shown in Figure 10.3.

10.3.3 1 Byte–2 Machine Cycle Instructions

An instruction like INC DPTR takes 1 byte–2 machine cycles. It reads the op-code at S1 and tries to read op-code of next instruction at S4 of the first cycle, and S1 and S4 of the second cycle but discards these three fetches because these instructions require 2 machine cycles to complete the execution, therefore, there is no meaning in fetching the next byte

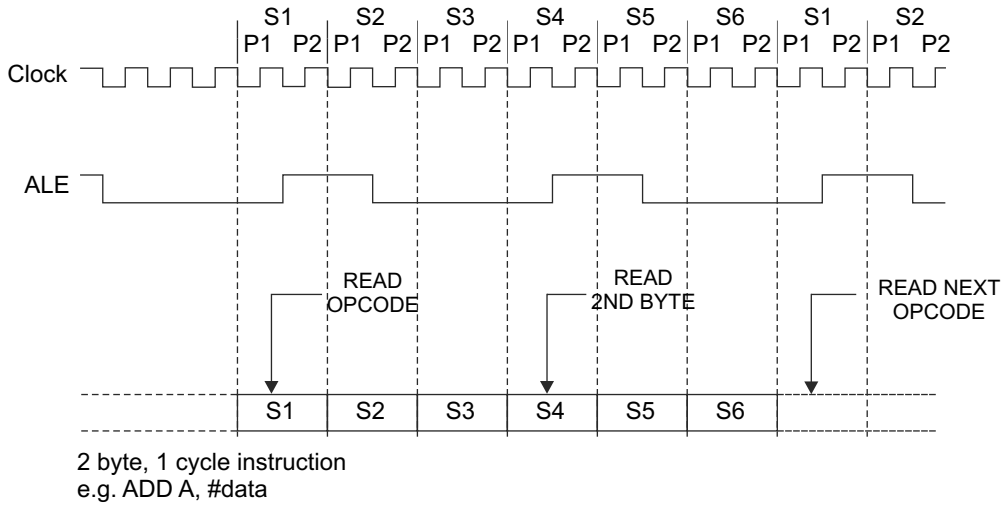


Fig. 10.3 *Timing diagram for 2 byte–1 machine cycle instruction*

during the execution. The execution will be completed at S6P2 of the second machine cycle. The timing diagram for such instructions is shown in Figure 10.4. The PC is effectively incremented by 1 during these instructions.

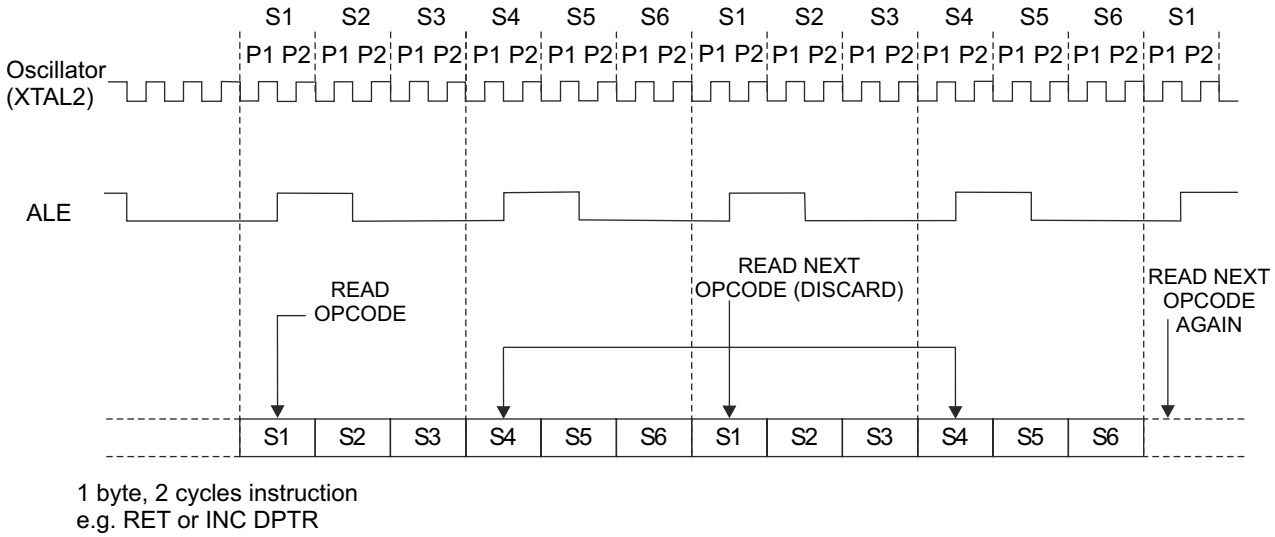


Fig. 10.4 *Timing diagram for 1 byte–2 machine cycle instruction*

10.3.4 2 Byte—2 Machine Cycle Instructions

Now if an instruction is of 2 byte–2 cycles then for that the timing diagram is shown in Figure 10.5. Here, during S1 of the first cycle, the op-code is fetched and the next byte is read during S4 of the first cycle. It reads the next op-code at S1 and S4 of the second cycle but discards it as mentioned above.

10.4 | EXTERNAL MEMORY ACCESS

There may be two types of external memories, the external program memory and the data memory. Program memory is accessed using $\overline{\text{PSEN}}$ (Program Store Enable) as the memory read signal. Data memory is accessed using $\overline{\text{RD}}$ and $\overline{\text{WR}}$ signals.

10.4.1 External RAM (Data Memory) Access

Data memory is accessed using either a 16-bit address (MOVX @DPTR) or an 8-bit address (MOVX @ Ri). The MOVX instruction requires two machine cycles to execute and op-code is not fetched (ALE is not generated) during the second

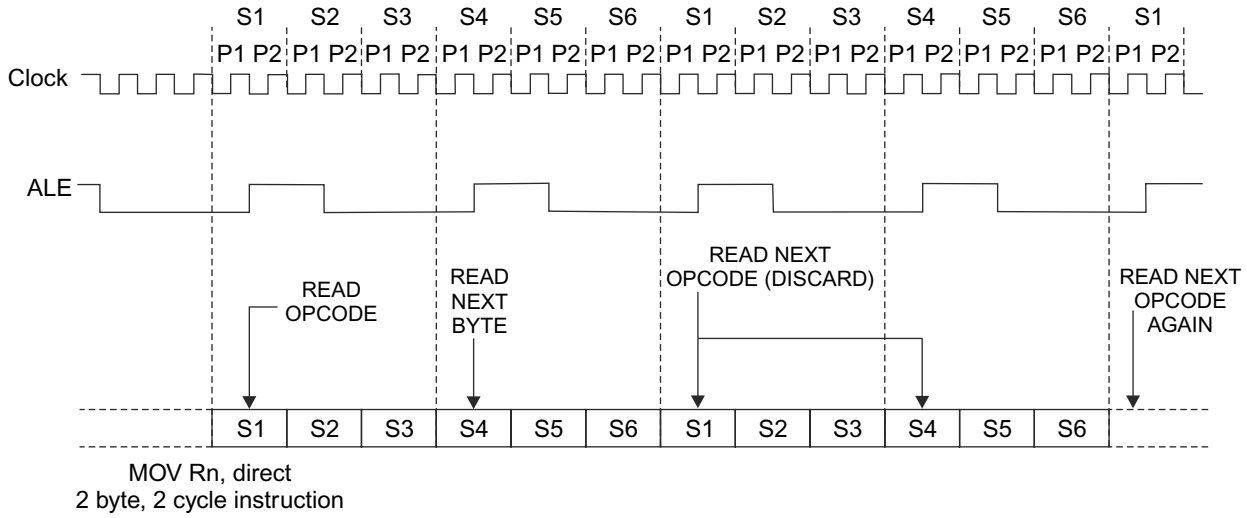


Fig. 10.5 *Timing diagram for 2 byte–2 machine cycle instruction*

cycle of these instructions. This is the only exceptional case when ALE is not generated and thus code memory fetches are skipped. The timing diagram for the MOVX instruction is shown in Figure 10.6.

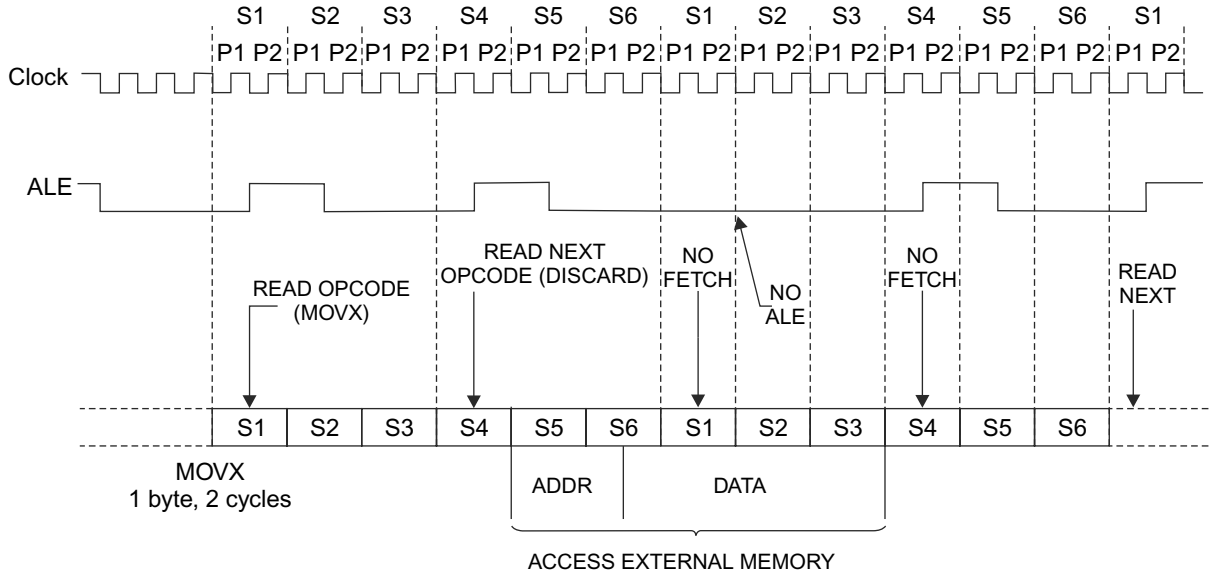


Fig. 10.6 *Timing diagram of the MOVX instruction*

10.4.2 Data Memory Read/Write Cycle

When a 16-bit address is used (MOVX @DPTR), the higher byte of the address is placed on port 2, where it is kept stable for the duration of the read/write cycle. When 8-bit address is used (MOVX @Ri), the contents of the port 2 SFR appear at the port 2 pins during the entire memory cycle. In both cases, port 0 is the time-multiplexed lower order address bus and the data bus. The ALE signal is used to latch the lower address byte into an external latch. The lower address byte becomes valid at the negative edge of ALE. In a read cycle, the incoming byte is accepted at port 0 during S3 of the second cycle. Memory read cycle shown in Figure 10.7.

The write cycle is similar to read cycle. The data byte to be written appears on port 0 just before $\overline{WR}$ is made 0, and held there until $\overline{WR}$ is made 1.

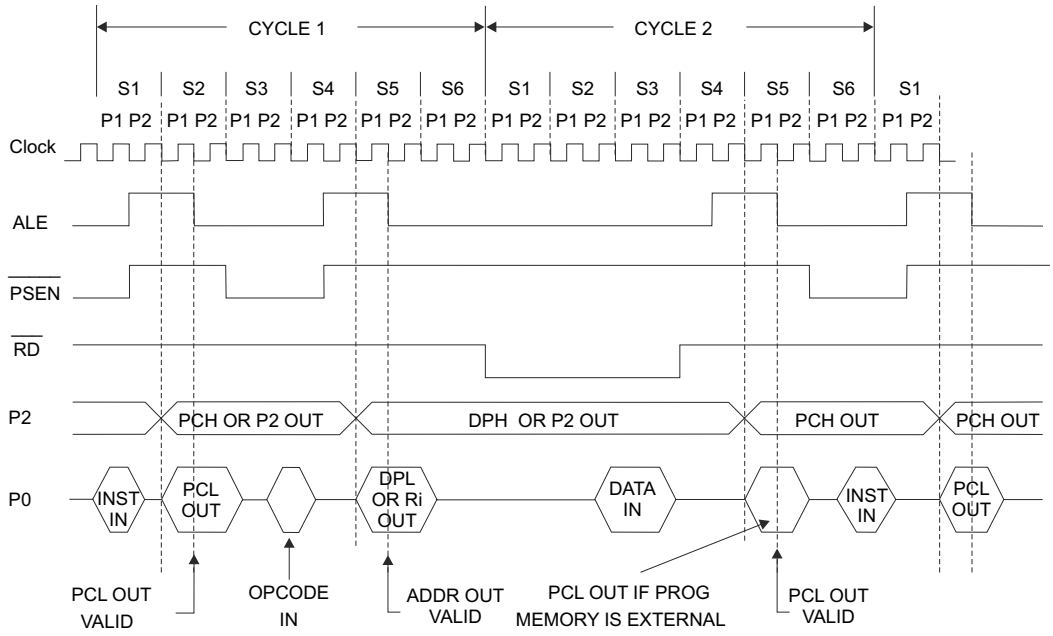


Fig. 10.7 *External data memory read cycle*

10.4.3 External ROM (Code Memory) Access

External code memory is always accessed using a 16-bit address. It will be accessed when $\overline{EA}$ is 0 or when the program memory addresses beyond on-chip memory are being accessed. During the external code memory access, port 2 is dedicated for higher byte of address (higher byte of addresses are provided by high byte of PC) and may not be used for I/O activities. When the program is executed from the internal code memory, $\overline{PSEN}$ is not activated, and addresses are not placed on P0 and P2. While accessing the external code memory, $\overline{PSEN}$ is activated twice every cycle (except MOVX instruction) irrespective of byte fetched for the instruction is required or not. The timing of external code memory access is shown in Figure 10.8.

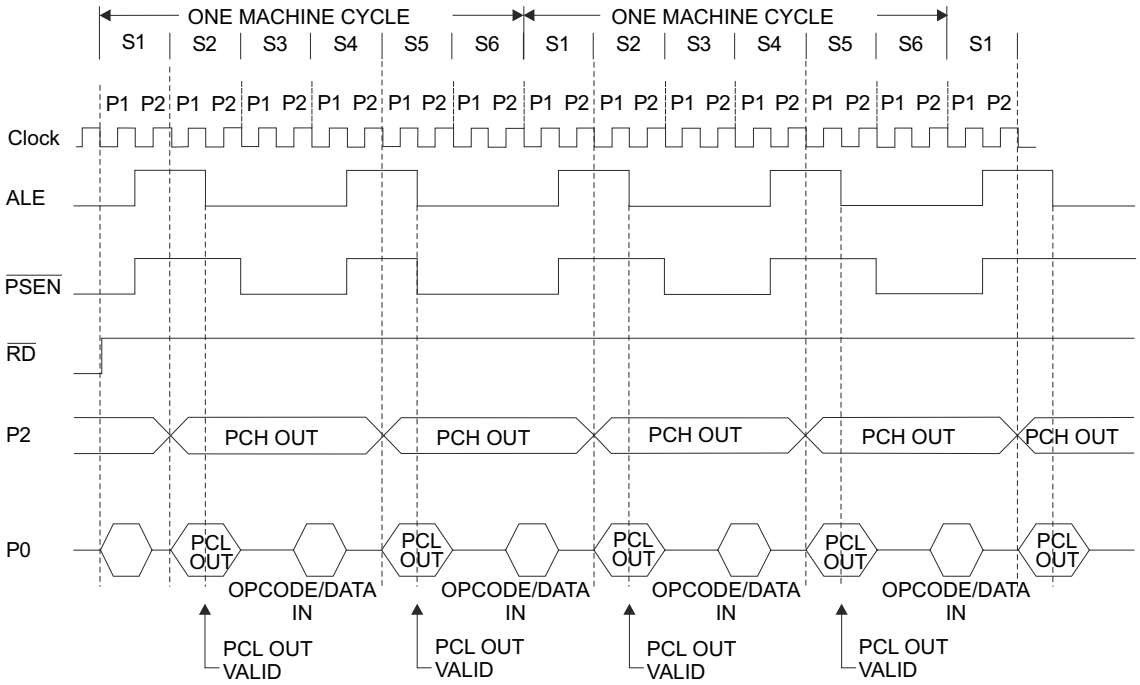
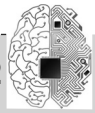


Fig. 10.8 *External code memory access timing*



THINK BOX 10.2

How is the speed of different microprocessors/controllers compared?

The speed is compared by either clock frequency (maximum) or time required to execute an instruction or time required to execute standard programs, usually referred as benchmark programs.

10.5 | 8051 INSTRUCTION EXECUTION

The execution of the 8051 instructions can best be understood by architectural block diagram which shows organization of all hardware components and data path connections between them.

In this section, instruction execution is illustrated by highlighting the active data paths (buses) to show the movement of operands around the various components within the microcontroller. This type of representation of an instruction execution is commonly referred as *data flow diagrams*. Execution of different types of instructions is discussed in the following section.

Note: The following discussion is developed using the general concepts of computer architecture and timing information given in the datasheets. The discussion may be helpful to understand internal working of the microcontroller, it may also be helpful in designing controller/processor core using hardware descriptive language. The steps of operation are given only for more clarity and may not be directly related with machine cycles.

10.5.1 MOV A, *Operand*

The execution of this instruction is illustrated in Figure 10.9.

The execution will proceed in the following logical steps:

1. The contents of the PC (address) are sent to the Program Address Register (PAR). The PAR will hold the address until a byte is fetched from ROM (for duration of memory access time). The PAR is equivalent to the Memory Address Register (MAR) of the memory.

   The PC incrementer increments the PC to point to the current instruction during execution of the previous instruction.
2. The address stored in the PAR is sent to the program memory (ROM) to fetch the op-code.
3. The op-code is fetched from the given address and placed in the instruction register-IR. The op-code present in the IR is given to the instruction decoder (part of Timing and Control block), which will decode the op-code (identify function to be performed by an instruction) and generate the appropriate control signals to execute the instruction.

   If the *operand* is immediate data or direct address, the second byte is fetched from ROM in similar manner (refer timing diagram of 2 byte-1machine cycle instruction in Figure 10.3).
4. If the *operand* is in the internal RAM (as in instructions MOV A, Rn; MOV A,@Ri or MOV A, direct), the decoder will generate the address of internal RAM corresponding to operand and write that address to RAM address register.

   However, if the *operand* is immediate data then the next byte from the code memory is loaded to the accumulator through data bus (not shown).
5. The address in the RAM address register is used to select the appropriate operand from the internal RAM.
6. The selected operand from the internal RAM is transferred to Accumulator through data bus. (The control block will generate signals equivalent to "RAM output enable" and "Load Accumulator" and perform data transfer).

The PC is incremented after fetching each byte of an instruction.

Note that steps 3 and 4 utilize the same path of the internal data bus in a time-multiplexed manner.

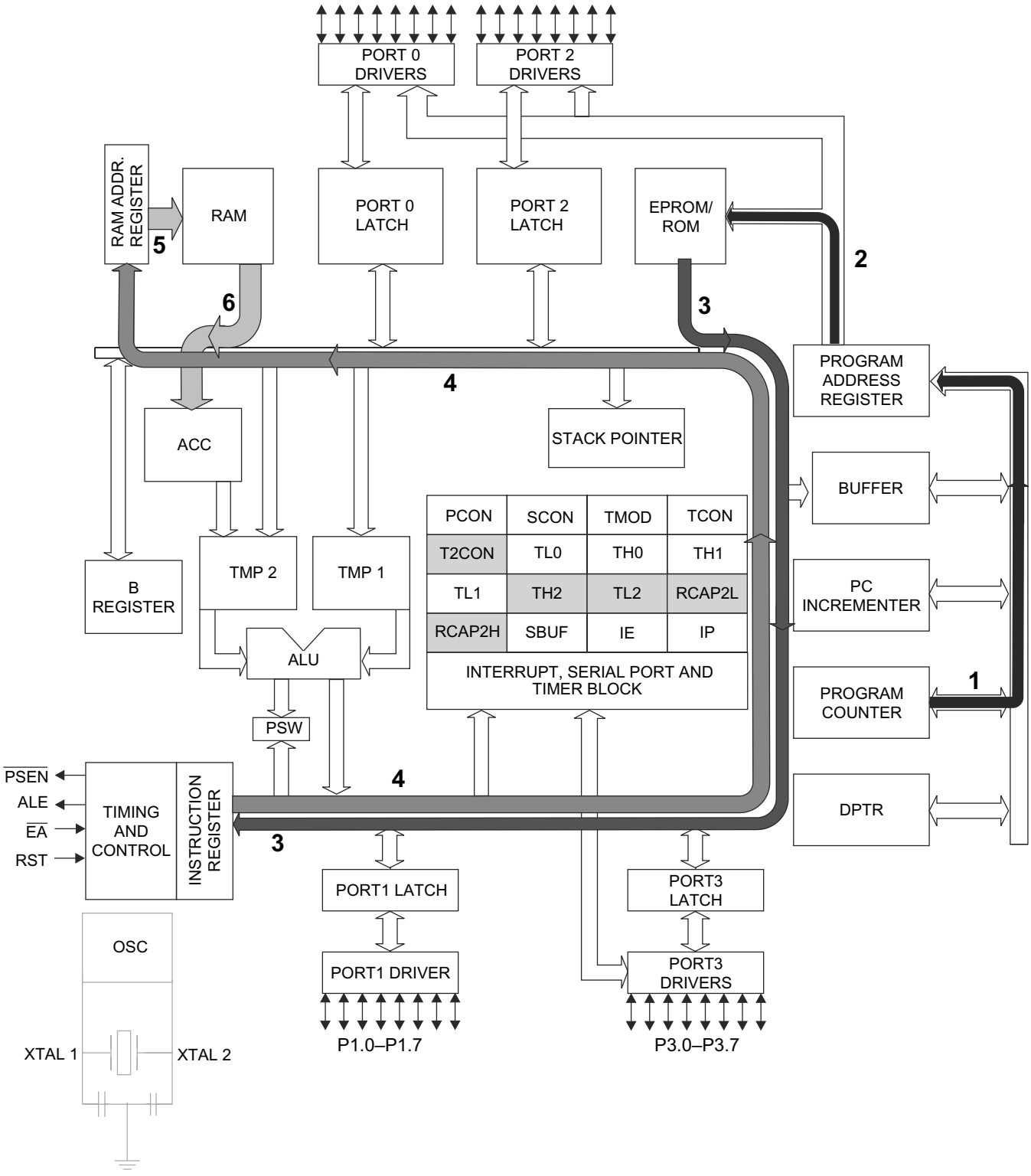


Fig. 10.9 *Execution of MOV A, operand*

10.5.2 ADD A, *Operand*

The execution of this instruction is shown in Figure 10.10.

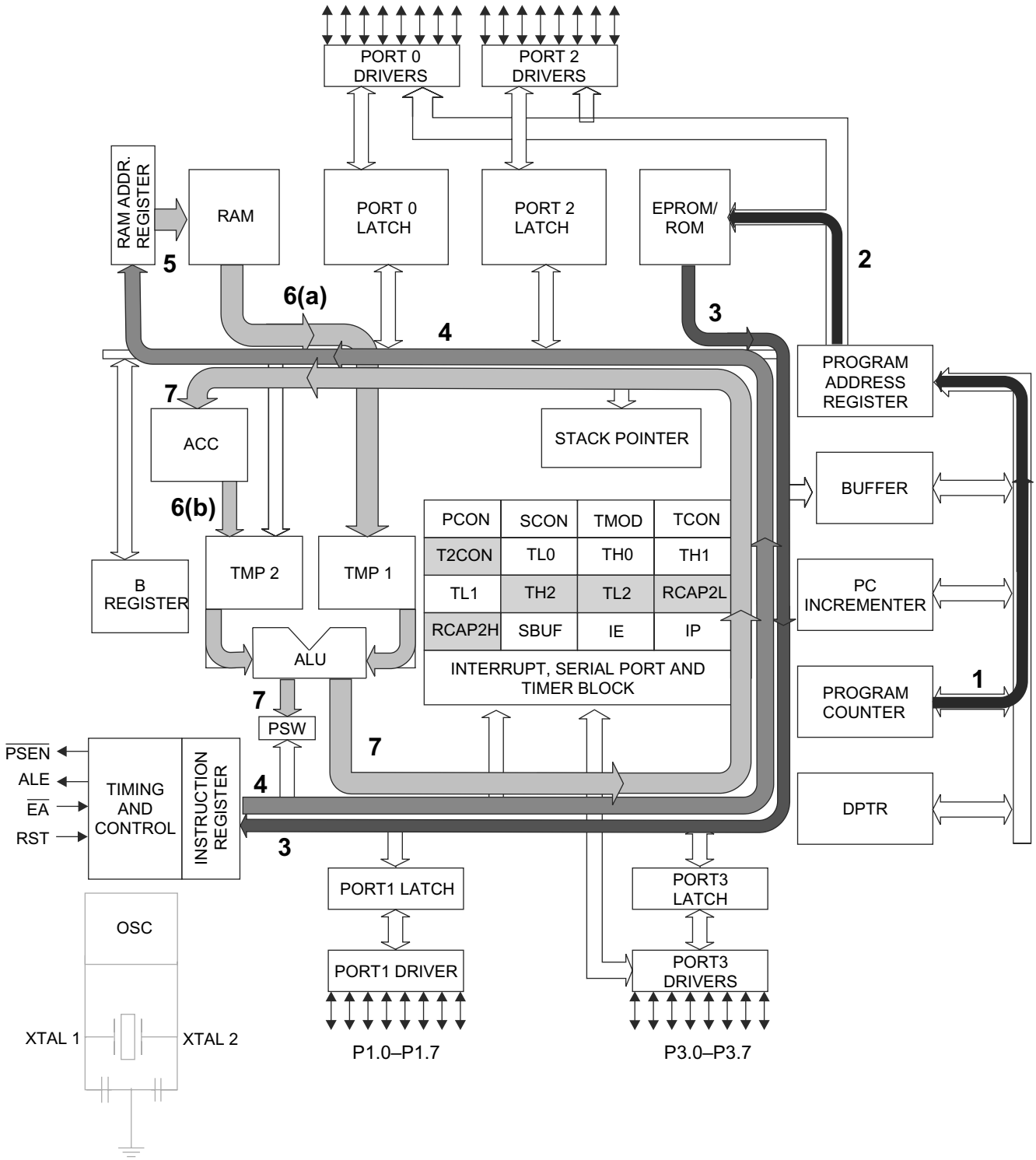


Fig. 10.10 *Execution of ADD A, operand*

The execution will proceed in the following logical steps:

1. The contents of the PC (address) are sent to the Program Address Register (PAR). The PAR will hold the address until a byte is fetched from ROM (for duration of memory access time).
   The PC incrementer increments the PC to point to the current instruction during execution of the previous instruction.
2. The address stored in the PAR is sent to the program memory (ROM) to fetch the op-code.

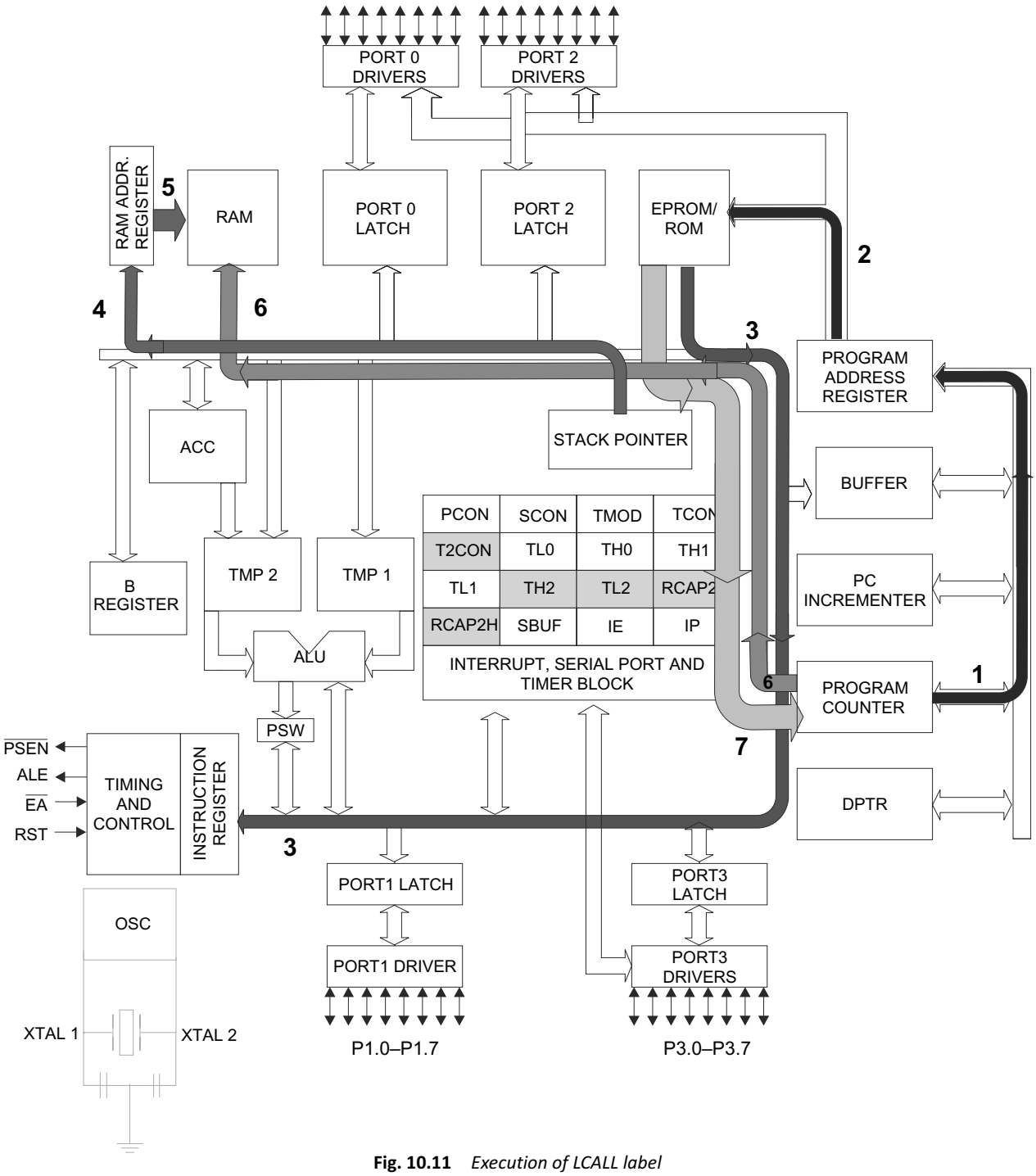


Fig. 10.11 *Execution of LCALL label*

3. The op-code is fetched from the given address and placed in the instruction register-IR. The op-code present in the IR is given to the instruction decoder (part of Timing and Control block), which will decode the op-code (identify function to be performed by instruction) and generate the appropriate control signals to execute the instruction.
   If the *operand* is immediate data or direct address, the second byte is fetched from ROM in similar manner (refer timing diagram of 2 byte—1 machine cycle instruction in Figure 10.3).
4. If the *operand* is in the internal RAM (as in instructions ADD A, Rn; ADD A,@Ri or ADD A, direct), the decoder will generate the address of the internal RAM corresponding to the operand and write that address to the RAM address register.
   However, if the *operand* is immediate data then the next byte from the code memory is read (not shown).
5. The address in the RAM address register is used to select appropriate operand from the internal RAM.
6. (a) The data from the RAM is supplied to the temporary register1.
   (b) The data in the Accumulator is also provided to the temporary register2.
7. In the ALU, the addition operation is performed and the result is provided to the accumulator through the data bus and PSW register is updated as per the result.

The PC is incremented after fetching each byte of an instruction.

Note that steps 3, 4 and 7 utilize same path of internal data bus in a time-multiplexed manner.

10.5.3 LCALL *Label*

The execution of this instruction is shown in Figure 10.11. The execution will proceed in the following logical steps,

1 to 3: These steps are similar for all instructions (see steps for instruction MOV A, *operand* in Section 10.5.1).

4 to 6: The current value of the PC is stored on the stack ($PC_{7\text{-}0}$ at address SP+1 and $PC_{15\text{-}8}$ at SP+2) and the stack pointer is incremented by two. (All steps are not shown in figure)

7: The next two bytes (destination address) from the ROM is fetched (one by one) and placed into the Program Counter (PC) to point to the instruction where the *label* is located.

The new value of the program counter is given to the code memory through the program address register and the next instruction is fetched from the address *label*.

10.5.4 MOVX A, @ DPTR

The execution of this instruction is shown in Figure 10.12. The execution will proceed in the following logical steps:

1 to 3: These steps are similar for all instructions (see steps for instruction MOV A, *operand* in Section 10.5.1).

4. The address of the operand (contents of DPTR) is given to the external RAM through port 0 and port 2.

5. Address latch enable signal (ALE) is generated to capture the address in the external latch, $\overline{RD}$ signals is generated to read the data from external RAM, $\overline{CS}$ is generated using address decoder circuit.

6. From the address in the external RAM, the data is fetched through port 0 and provided to the accumulator through a data bus.

POINTS TO REMEMBER

- The instruction execution is timed according to clock pulses.
- The 8051 has inbuilt crystal oscillator circuit which can be used as a clock source if desired. We need to connect a resonant network to make it functional. The clock frequency is determined by crystal frequency.
- We may also use external pulses as the clock signal.
- The machine cycle of the 8051 contains 6 states, each state requires two clock cycles referred as P1 and P2, thus 8051 machine cycle requires 12 clock cycles.
- ALE is normally activated twice during each machine cycle, during S1P2 and again during S4P2 except MOVX instructions.
- Program memory is accessed using $\overline{PSEN}$ (Program Store Enable) as the memory read signal. Data memory is accessed using $\overline{RD}$ and $\overline{WR}$ signals.

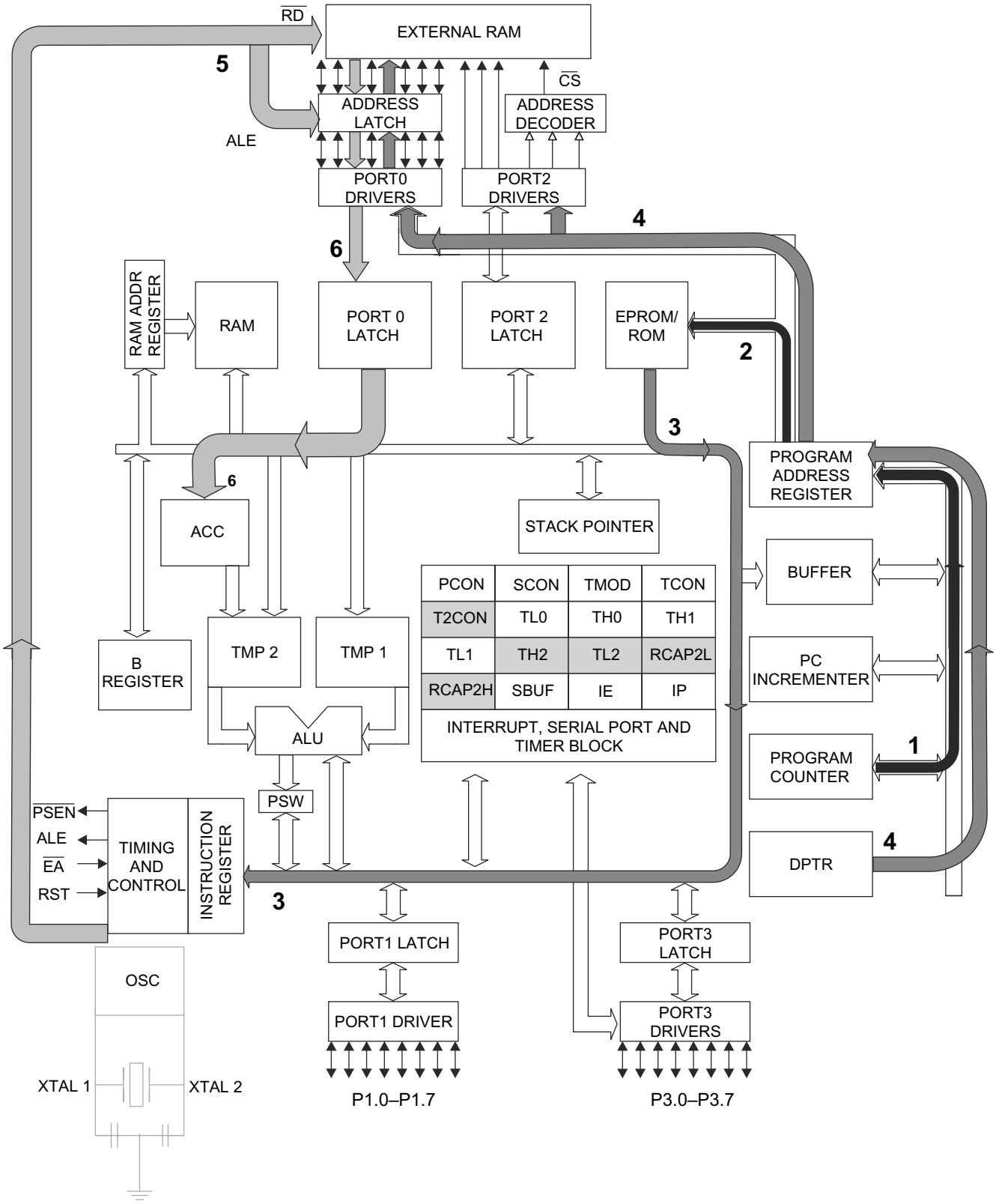


Fig. 10.12 *Execution of MOVX A, @ DPTR*

- When the program is executed from internal code memory, $\overline{\text{PSEN}}$ is not activated, and addresses are not placed on P0 and P2.

OBJECTIVE QUESTIONS

1. We can make a program execution faster, if,
   (a) number of instructions used by the program is reduced
   (b) number of clock cycles used by the program is reduced
   (c) number of bytes used by the program is reduced
   (d) all of the above
2. In a microcontroller, execution speed of a given program depends upon,
   (a) on-chip/off-chip program memory (b) number of address lines
   (c) size of a Program Counter (PC) register (d) clock frequency
3. During execution of the instruction MOV R0, #20 H, the 8051 generates ALE signal,
   (a) one time (b) two times (c) three times (d) twelve times
4. Frequency of the ALE signal is equal to,
   (a) crystal frequency (b) crystal frequency/2
   (c) crystal frequency/6 (d) crystal frequency/12
5. 'CJNE A, *direct, rel'* is _____ instruction.
   (a) 1 byte—2 machine cycle (b) 2 byte—1 machine cycle
   (c) 2 byte—2 machine cycle (d) 3 byte—2 machine cycle
6. 'ACALL *addr11"* is _____ instruction.
   (a) 1 byte—2 machine cycle (b) 2 byte—1 machine cycle
   (c) 2 byte—2 machine cycle (d) 3 byte—2 machine cycle
7. 'ADDA, *direct'* is _____ instruction.
   (a) 1 byte—2 machine cycle (b) 2 byte—1 machine cycle
   (c) 2 byte—2 machine cycle (d) 3 byte—2 machine cycle
8. The OPCODE byte read from the memory is finally delivered to,
   (a) instruction register (b) accumulator
   (c) ALU (d) program counter
9. While reading data from external memory_____ port act as data bus.
   (a) P0 (b) P1 (c) P2 (d) P3
10. While reading data from external memory_____ port provides control signals.
    (a) P0 (b) P1 (c) P2 (d) P3

Answers to Objective Questions

1. (d)	2. (d)	3. (b)	4. (c)	5. (d)
6. (c)	7. (b)	8. (a)	9. (a)	10. (d)

REVIEW QUESTIONS WITH ANSWERS

1. Define the term *clock pulse.*

A. It is a repetitive sequence of pulses used to synchronize all internal activities of a microcontroller/ processor. It is the smallest unit of time in which a microcontroller performs a part of the operation.

2. How is the machine cycle of the 8051 further divided into smaller units?

A. The machine cycle of the 8051 contains 6 states, a state is time required to perform subdivision of basic operations like read, write or decode. Each state requires two clock cycles referred as phase 1 (P1) and phase 2 (P2). Thus, 8051 machine cycle requires 12 clock pulses.

3. How many machine cycles are required by the 8051 instructions?

A. In the 8051, an instruction may require one, two or four machine cycles depending upon the type of instruction.

4. **Write an instruction which requires one byte, one machine cycle.**
   MOV A,R0
5. **What is the time period of a machine cycle for crystal frequency of 12 MHz?**

A. The time period of a clock is 1/12 MHz. Each machine cycle is made from 12 clock periods, therefore time period of a machine cycle is 12/12 MHz= 1μs.

6. **List the instructions in the 8051 that requires 4 machine cycles.**

A. MUL AB and DIV AB are the only two instructions in the 8051 which requires 4 machine cycles.

7. **What is the first operation performed in a machine cycle?**

A. Op-code fetch.

8. **Which signals are used to access external data memory?**

A. $\overline{RD}$ and $\overline{WR}$ signals.

9. **What is the use of MOVX instructions?**

A. They are used to access external data memory.

10. **What is the size of addresses used by MOVX instructions?**

A. Either 8 bits (MOVX @ Ri) or 16 bits (MOVX A, @DPTR).

11. **When a program is executed from internal code memory. PSEN is not activated. True/False.**

A. True.

12. **How does the MOVX instruction execution differ from all other instructions?**

A. The ALE signal is generated only once during second machine cycle of the MOVX instruction, while in all other instructions the ALE is generated twice every machine cycle.

13. **Discuss the role of the Program Address Register (PAR) during instruction execution. Why PC cannot be used for the same operation?**

A. The Program Address Register (PAR) will hold the address of a byte until it is fetched from the ROM. The address must be held constant at least for the memory access time of the ROM. The value of the PC will change during this time therefore address in the PC cannot be directly given to the ROM.

EXERCISE

1. What is meant by data-flow diagram?
2. Define the term machine cycle.
3. List at least two instructions which requires
   (a) 1 byte—2 machine cycles (b) 2 bytes—1 machine cycles
   (c) 2 bytes—2 machine cycles (d) 3 bytes—2 machine cycles
4. ALE signal is generated _____ times in a machine cycle.
5. Draw and discuss timing diagram for instructions having
   (a) 1 byte—2 machine cycles (b) 2 bytes—1 machine cycles
   (c) 2 bytes—machine cycles
6. _____ signal is used to access the external code memory.
7. Draw and explain the timing diagram of the data memory read cycle.
8. Discuss how the data memory write cycle differs from read cycle.
9. Explain the data-flow diagram for the instruction MOV A, R0.
10. Explain with help of the data flow diagram how the instruction MOVX A,@DPTR is executed.
11. Draw data-flow diagram for execution of the ADD A, R1 instruction.
12. List the timing and control signals of the 8051.
13. Discuss the need of the RAM address register.
14. Derive formula to calculate time to execute an instruction.

11

The 8051 Hardware, System Design and Troubleshooting

Objectives

- List the pins of the 8051 along with their alternate functions
- Describe the uses and significance of each pin of the 8051
- Explain the address/data de-multiplexing using ALE and dual role of ports
- Discuss power consumption control modes in the 8051
- Design an 8051 based system
- Discuss the troubleshooting techniques

Key Terms

- 74LS373: Latch
- A15-A8,AD7-AD0
- Address/Data De-multiplexing
- ALE
- External Access: $\overline{EA}$
- Flash Programmer
- Idle Mode
- Lock Bits
- Logical Error (Bug)
- On-chip Oscillator
- Power Control: PCON
- Power Down Mode
- Power On Reset
- Pull-up Resistor
- Troubleshooting

11.1 THE 8051 PIN DIAGRAM

The 8051 family microcontrollers are 40-pin chips available in different packages. The pin diagram of DIP (Dual Inline Package) is shown in Figure 11.1. It has four eight-bit I/O ports, each port pin (except P1) can be used for two different functions as shown in Figure 11.1. However, all functions may not be used at the same time. The function of the pin is determined by an instruction used to access that pin or physical connection to it. Thus, it has effectively 66 (40 normal pins + 24 alternate functions of ports + 2 for programming) pins.

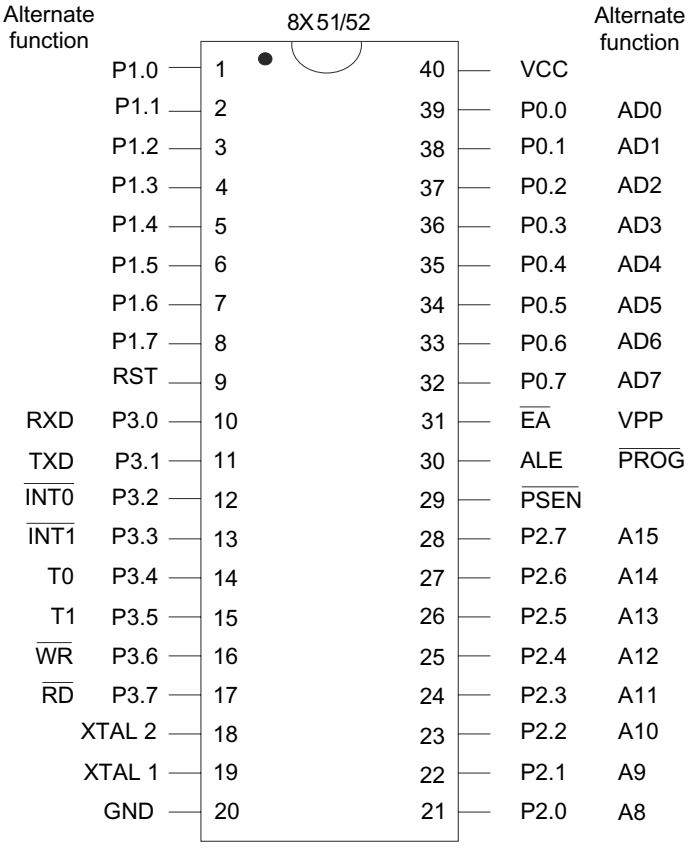


Fig. 11.1 *The 8051 pin diagram*

11.2 THE 8051 PIN DESCRIPTION

V_{CC}: Power Supply Pin The 8051 operates at DC power supply of +5 V with respect to ground. The +5 V is to be connected to the pin V_{CC} (Pin 40). Variation of 10% to 20% in the value of V_{CC} is allowed depending upon the 8051 variant.

GND: Ground Pin 20 is the ground. The supply is connected with respect to the ground pin.

XTAL1 and XTAL2: Oscillator Pins The 8051 has an internal (on-chip) oscillator circuit (partial circuit) which generates the clock pulses which synchronizes all internal operations. The external resonant circuit is connected with this on-chip oscillator circuit to make a complete oscillator. Normally, quartz crystal and capacitors are connected with XTAL1 (Pin 19) and XTAL2 (Pin 18) as shown in Figure 11.2 (a).

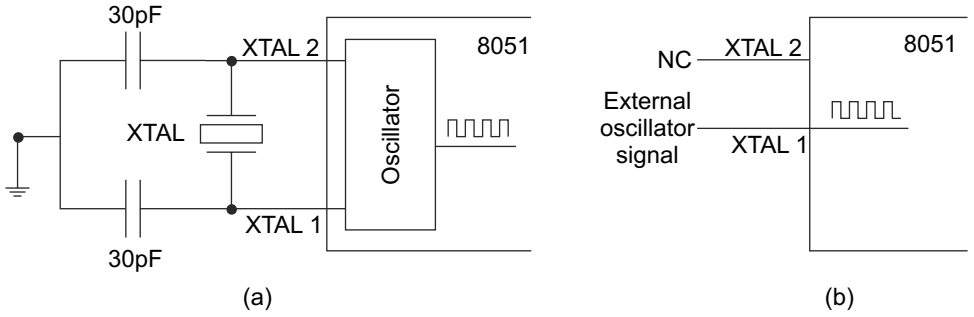


Fig. 11.2 *(a) Crystal connection (b) External clock source connection*

The microcontroller works at a frequency of the crystal oscillator. The variants of the 8051 family operate at different speeds (clock frequency). The maximum speed refers to the maximum crystal frequency that can be connected to XTAL pins. For example, 12 MHz chip should be connected with a maximum 12 MHz crystal. The minimum speed indicates that dynamic RAM is used in a microcontroller (which requires periodic refreshing) and must be operated above minimum speed otherwise data will be lost. Nowadays, the 8051 variants are available, which works at a minimum frequency of 0 Hz, i.e. fully static operation; data in the internal registers will not be lost because of static memories which do not require periodic refreshing.

It is also possible to use a clock signal from the external oscillator. For this configuration, external oscillator signal is connected to XTAL1 pin and XTAL2 pin is left unconnected as shown in Figure 11.2 (b).

THINK BOX 11.1

What is meant by fully static operation? How can this feature be useful in reducing power consumption?

The static RAM is used in the microcontroller, therefore there is no requirement of refreshing the contents of RAM periodically. Because of this, the contents of the RAM are preserved even if clock is freezed (0 Hz), i.e. the clock can be as low as 0 Hz.

Because of this feature, it is not required to operate the microcontroller at certain minimum frequency even when microcontroller has nothing to do except for waiting for some external event. Since power consumption is directly proportional to the clock frequency, the microcontroller can be operated at minimum frequency just sufficient for an application or its clock can be stopped in case of waiting. This way, power consumption can be reduced.

ALE/$\overline{\text{PROG}}$: Address Latch Enable When connecting the 8051 to external memory, port P0 provides both address and data, i.e. address and data are time multiplexed using P0 to save pins of the microcontroller. The ALE (Address Latch Enable, Pin 30) is used for de-multiplexing the address and data. ALE pulse (ALE=1) indicates that the address is present on P0 and is used to enable external latch (normally 74LS373) which will store (and de-multiplex) the address present on P0. Interfacing of external latch and use of ALE is discussed in more detail in next section.

Address/Data Demultiplexing using ALE Ports 0 and 2 are collectively used to provide 16-bit address to external memory. P2 provides upper 8 bits of addresses A_{15} to A_8 and P0 provides lower 8 bits of addresses A_7 to A_0 as well as 8-bit data D_7 to D_0. The P0 is time multiplexed to provide the address and data bus. These multiplexed address and data lines must be separated to access memory. Separation of address and data from P0 is called *address/data demultiplexing*. Figure 11.3 shows simplified memory access timing.

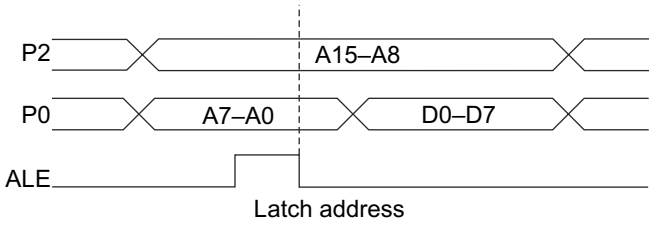


Fig. 11.3 *Simplified memory access timing*

As shown in Figure 11.3, the 8051 first provides upper 8 bits of addresses A_{15} to A_8 on P2 and lower address bits A_7 to A_0 on P0 and along with this, ALE is made high (ALE=1) to indicate P0 contains lower address bits. To extract address, P0 should be connected to a latch chip (typically 74LS373) and ALE should be connected to enable, G (equivalent to chip select) of the latch as shown in Figure 11.4.

ALE=1 will enable 74LS373 and its output will be lower address bits (A_7–A_0), when after some time ALE becomes 0, its contents are latched. The latch retains the address bits until next ALE pulse and P0 will now act as data bus (ALE=0) as shown in Figure11.3. ALE pulse is generated at constant rate of 1/6 oscillator frequency, i.e. twice every

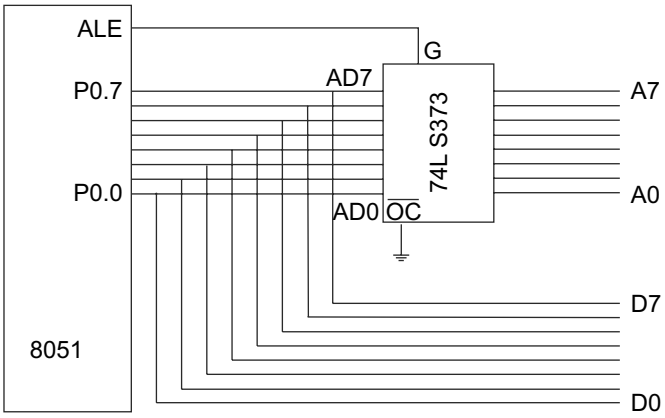


Fig. 11.4 *Address/data de-multiplexing*

machine cycle. It may be used for external timing or clocking purpose (one ALE pulse is skipped during each access to external data memory)

Alternate function of this pin is $\overline{\text{PROG}}$. It is a program input pulse used during on-chip flash memory programming. After reset, the 8051 checks this pin, if it is connected to logic '0' externally, it enters into flash programming mode, otherwise the 8051 continues with normal program execution mode.

RST: Reset Reset (Pin 9) is an active high input. When high pulse is applied to this pin, the microcontroller will reset and terminates all activities and contents of all registers will be lost and default values will be automatically loaded into SFRs as shown in Table 11.1.

Table 11.1 *SFR reset values*

SFR Name	Reset Value
PC	0000H
ACC	00H
B	00H
PSW	00H
SP	07H
DPTR	0000H
P0-P3	FFH
IP	00H
TC0N	00H
TM0D	00H
IE	00H
TH0	00H
TL0	00H
TH1	00H
TL1	00H
SC0N	00H
SBUF	Indeterminate
PC0N	00H

For proper reset operation, RESET signal must be held high at least for two machine cycles. The circuit to generate reset signal is shown in Figure 11.5. When power is applied, the capacitor is effectively short (because high current flows through it to charge it). Therefore, V_{CC} is connected effectively at RST pin, therefore, RST pin remain high for some time (at least for two machine cycles) depending upon RC time constant, this will reset the microcontroller which is also referred as power-on reset. Now, slowly, capacitor will be charged and when it is fully charged, no current will flow through it and therefore lower terminal (-ve terminal) will be effectively connected at ground. If we want to reset the system when application is running, we have to press RESET switch as shown in the Figure 11.5. When switch is pressed, capacitor will be discharged. When the switch is removed, the capacitor will start charging and initially will behave as a short; therefore, V_{CC} will be given to the RST pin, which will reset the microcontroller.

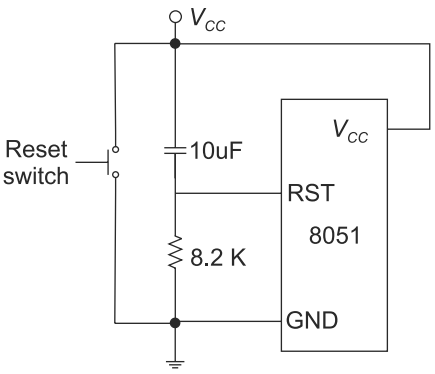


Fig. 11.5 *Power on reset with reset switch*

THINK BOX 11.2

What do internal RAM (00H to 7FH) contains after reset (or power on reset)?

The contents of RAM (00H to 7FH) after reset (or power on reset) are non deterministic and we cannot predict them exactly, moreover these contents may be different each time we reset the device and across the different devices.

$\overline{\text{PSEN}}$: Program Store Enable $\overline{\text{PSEN}}$ (Pin 29) is the active low output control signal used to activate (enable) external ROM/EPROM/EEPROM. Thus, this signal acts as the read strobe (or output enable) to external program memory. $\overline{\text{PSEN}}$ should be connected to $\overline{\text{OE}}$ (Output Enable) pin of ROM chip. Instructions MOVC A, @A+DPTR and MOVC A, @A+PC and op-code fetch from external ROM will activate this signal, however, it is not activated when on-chip ROM is accessed.

$\overline{\text{EA}}$/VPP: External Access Many members of the 8051 family have on-chip ROM. $\overline{\text{EA}}$ (Pin 31) is used for selection of on-chip or off-chip ROM, $\overline{\text{EA}}$=1 will tell the 8051 to read a program code from on-chip (internal) ROM and $\overline{\text{EA}}$ is grounded ($\overline{\text{EA}}$=0) when a program code is totally contained in an external ROM only. However, when $\overline{\text{EA}}$ = 1, any reference to program address which is outside the address range of on-chip memory will automatically access external

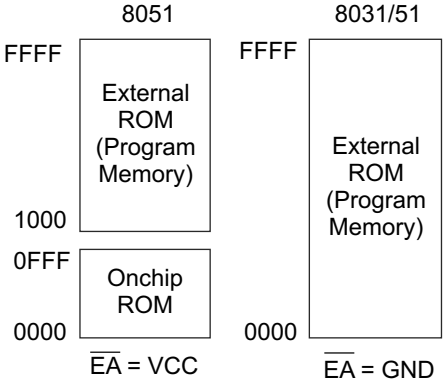


Fig. 11.6 *Program memory selection using $\overline{EA}$*

memory. For example, 89C51 has 4 KB of on-chip program memory (from 0000H to 0FFFH), when $\overline{EA}$ = 1, on-chip program memory addresses from 0000H to 0FFFH will be accessed and any access above address 0FFFH will be made from external memory. Similarly, for 89C52 any access above address 1FFFH will be made from external memory. How $\overline{EA}$ affects the program memory selection is shown in Figure 11.6.

It is to be noted that when $\overline{EA}$=0, all program fetches are from external ROM irrespective of presence of on-chip ROM. Also for ROM-less chip like 8031, $\overline{EA}$ must be grounded. The alternate function of this pin is that it receives 12 V programming enable voltage VPP during flash memory programming for chips that requires 12 V VPP.

Port 0 Port 0 (P0) is an 8-bit open drain bidirectional I/O port; it occupies 8 pins (Pin 32–39). Since all pins are open drain, to use P0 as an input or output, the external pull-up resistor of 10 kΩ (value greater than 1.42 kΩ) must be connected with each pin as shown in Figure 11.7.

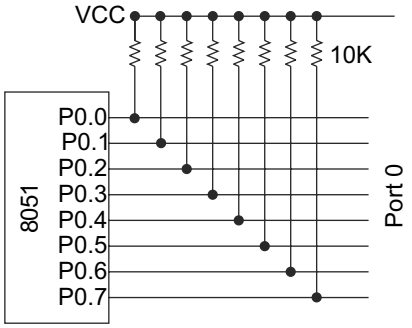


Fig. 11.7 *Port 0 with pull-up resistors.*

In order to configure P0 as an input, the port must be programmed by writing 1 to all bits. P0 also act as multiplexed low order address and data bus during access to external program and data memory. Because of this alternate function, the pins of P0 are also designated as AD_0 to AD_7. In this mode, it has internal pull-up resistors. P0 also receives the code bytes during on-chip flash programming and outputs the code byte during program verification process.

Port 1 Port 1 (P1) is an 8-bit bidirectional I/O port, (Pin 1–8) with internal pull-up resistors. When 1 is written to port 1 latch, it is pulled high by internal pull-ups and can be used as an input. P1 receives low-order address bytes during on-chip flash memory programming and program verification process.

Port 2 Like P1, Port 2 (P2) is also an 8-bit bidirectional I/O port, (Pin 21–28) with internal pull-up resistors. Port 2 also acts as a higher order address bus (A_{15}–A_8) while accessing external program memory as well as access to external data memory that uses 16-bit address (MOVX A, @DPTR or MOVX @DPTR, A)

P2 also receives the high order address bits and control signals during flash programming and verification.

Port 3 As P1 and P2, Port 3 (P3) can act as 8-bit bidirectional I/O port, (Pin 10–17) with internal pull-up resistors. Although it can be used as I/O port, it is more commonly used for alternate functions. P3 provides special signals as external interrupt inputs, transmit (TXD) and receive (RXD) for serial communication, timers external inputs (T0 and T1) and read, write signals. A pin-wise alternate function of P3 is given in Table 11.2.

Table 11.2 *Port 3 alternate functions*

Port Pin	Alternate Function
P3.0 (Pin 10)	RXD (Serial input: Receive Data)
P3.1 (Pin 11)	TXD (Serial output: Transmit Data)
P3.2 (Pin 12)	$\overline{INT0}$ (External interrupt 0)
P3.3 (Pin 13)	$\overline{INT1}$ (External interrupt 1)
P3.4 (Pin 14)	T0 (Timer/Counter 0 external input)
P3.5 (Pin 15)	T1 (Timer/Counter 1 external input)
P3.6 (Pin 16)	$\overline{WR}$ (External Data Memory write strobe)
P3.7 (Pin 17)	$\overline{RD}$ (External Data Memory read strobe)

Detailed explanation and use of TXD and RXD is explained in Chapter 15 (Serial communications) and that for $\overline{INT0}$ and $\overline{INT1}$ is discussed in Chapter 16 (Interrupts). T0 and T1 are explained in Chapter 14 (Timers). Read ($\overline{RD}$) and write ($\overline{WR}$) pins are discussed in Chapter 21 (External memory interfacing).

THINK BOX 11.3

Can you state any advantage of having signals (like $\overline{RD}$, $\overline{WR}$, $\overline{PSEN}$) as active low?

In some logic families like TTL, unconnected input may be considered as logic high. The active low signals make sure that no false action will be taken for such cases.

11.3 | POWER CONSUMPTION CONTROL OF THE 8051

The CHMOS versions of the 8051 has two power saving modes that are activated by programming PCON (Power Control) register. The two modes are idle and power down modes. The details of PCON register is shown in Table 11.3. Note that PCON register is not bit addressable.

Table 11.3 *Power control (PCON) register*

SMOD	--	--	--	GF1	GF0	PDWN	IDLE
MSB							LSB

Bit	Symbol	Description
7	SMOD	Serial baud rate generation mode. When SMOD=1, the baud rate of the UART is doubled.
6	--	
5	--	
4	--	
3	GF1	General-purpose user flag 1. Can be used to store 1 bit information.
2	GF0	General-purpose user flag 0. Can be used to store 1 bit information.
1	PWDN	Power down bit. PWDN =1 will activate power down mode
0	IDLE	Idle mode bit. IDLE =1 will activate idle mode.

1. Idle Mode

The microcontroller will enter into idle mode by setting the IDLE bit to 1. The idle mode stops the program execution and contents of internal RAM are preserved. The oscillator continues to run but the clock is disconnected from the CPU. The timers and serial port operates normally. The microcontroller will come out of idle mode by the activation of any interrupt (or RESET); this will clear the IDLE bit to 0. After completion of interrupt service routine (ISR), the execution is resumed from the instruction next to the instruction which had set the IDLE bit.

2. Power Down Mode

The microcontroller will enter into power-down mode by setting the PWDN bit to 1. The power-down mode stops on-chip oscillator, therefore program execution, timers and serial port operation are also stopped. The contents of the internal RAM are preserved. The only way to come out of power down mode is by applying reset (or power on reset) signal.

Tip for Power Saving

One simple way to save the power consumption is to use crystal frequency just sufficient for an application, operating with higher frequency will unnecessarily increase the power consumption.

THINK BOX 11.4

Why do you think there is a need of power down or idle modes in a microcontroller?
Small (or handheld) embedded systems are usually operated from battery; therefore, lower power consumption is a desired feature. Power down and idle modes can significantly reduce the power consumption of a system. These modes are used when the system should wait for some external or internal event to occur to take some action.

THINK BOX 11.5

What are the other power-saving modes available in newer microcontrollers?
Power Save, Power Down, Idle, Standby, Extended Standby and ADC Noise Reduction.

11.4 DESIGN OF THE 8051 BASED SYSTEM

We will design the simplest system which requires minimum components. It will contain all necessary supporting circuits interfaced in order to make a system functional. The following circuits and components must be present in the simplest system based on the 8051.

1. Reset Circuit

This circuit will generate RESET signal when system is powered up (usually referred as power on reset circuit) or when the RESET key is pressed at any moment. It is required to reset a system while troubleshooting and development of a system (see Figure 11.5).

2. Clock Circuit

The 8051 have partial internal oscillator circuit. To make oscillator circuit functional and complete, we need to connect crystal of desired frequency (refer data sheet for minimum and maximum value for particular variant of the 8051) and two capacitors as shown in Figure 11.8. If external clock source is used, connect the signal as shown in Figure 11.2 (b).

3. Pull-up Resistors (only if port 0 is used as I/O port)

Since the port 0 is open drain port, pull-up resistors are required to be connected with each pin that is used as an input/output. For simple applications, *if port 0 is not used, there is no need to connect pull-up resistors*. Ports 1, 2 and 3 do not require any pull-up resistor since they have internal pull-up resistors (refer Figure 11.7).

4. Demultiplexer

It is a circuit to separate address and data bus and used ***only*** if external memory is interfaced in a system.

Some members of the 8051 family do not have on-chip ROM (like 8031) and other members only have limited on-chip ROM (for example, 89C51, 8751 have 4 Kbytes of on-chip ROM). When these microcontrollers are used for applications which require programs larger than 4 Kbytes, it is required to interface external program ROM. To access data from memory, address and data lines from the port 0 must be demultiplexed. When on-chip memory is sufficient, there is no need to interface external program memory. These days, many variants of the 8051 have up to 64Kbytes of on-chip flash memory.

The circuit diagram of a system with minimum components along with a push button key and LED interfaced for testing and demonstration of working of a system is shown in Figure 11.8. The program which reads the status of a key and sends it to the LED (LED should glow when key is pressed, otherwise off) should be developed and loaded into the microcontroller chip. A Flash programmer device is required to load program into the 89C51. We have used AT89C51 chip because it contains 4Kbytes of on-chip flash ROM (and 87C51 contains 4 Kbytes of EPROM), which can be easily programmed and erased number of times during development and testing of the system. Note that $\overline{EA}$ is connected to V_{CC} to access the program code from internal flash memory.

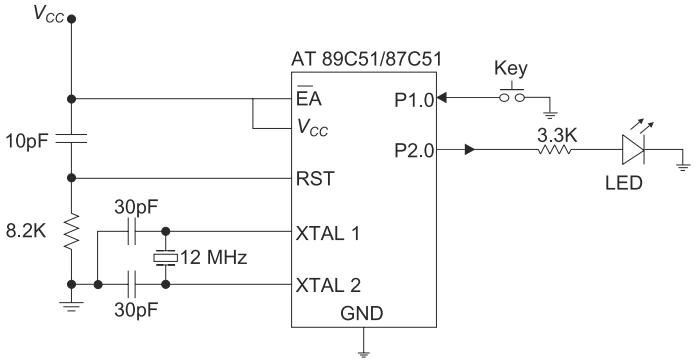


Fig. 11.8 *89C51 based simplest system*

11.5 TROUBLESHOOTING 8051 BASED SYSTEMS

It is very much likely that while developing an application, when we apply power to a complete system (microcontroller with program loaded into it and required components interfaced) nothing would happen or we do not get expected behavior from a system. In such cases, we need to start looking at few common problems as discussed below.

First, check all connections within the system (check connectivity of all wires or tracks using multimeter), then check the power supplied to the various components, specially V_{CC} and ground of all chips present in a system. The supply voltage should be of proper magnitude (refer datasheets of chips if required).

If the supply voltage becomes low after connecting it to the system, there might be a *V*cc to ground short. Next, check the 8051's Reset pin. It should be low when power is applied. The push button that shorts Reset to V_{CC} (to make Reset pin

high, see Figure 11.5) is used to reset the device during development and troubleshooting. Remember that in the 8051, Reset is an active high signal, i.e. a high-voltage level is required to reset the 8051. In many other microcontrollers, reset is active low signal.

Next, the oscillator circuit should be checked. This can be done by connecting XTAL2 pin with an oscilloscope. The clock signal should be roughly a sine wave of approximately 4 to 5 V and the clock frequency should be equal to crystal frequency.

If program is loaded in an on-chip ROM then connect $\overline{EA}$ to V_{CC}, and if only external program memory (ROM) is used then the $\overline{EA}$ pin should be connected to ground.

Conform that the 8051 variant chip (or the external program memory if any) are programmed correctly with the correct (and final) version of a program.

If only on-chip ROM is present in a system, make sure that the program is not accessing any address outside the range of on-chip ROM, also check whether program is trying to access device that is not present in a system.

After checking all the above-mentioned issues related with the hardware; if the system still does not work properly, then we have to go through the program and look for the logical errors (or bugs). The simplest way is to single step the program and check if the program statements produce the correct results as per comments given in the program. To ease the above process of finding logical errors, we should write proper comments while developing a program. The important point to remember is that never use default values while testing a system. For example, do not write '0FFH' to port pins to check 'minimum component 8051 system' functionality because ports will have value 'FFH' even without writing anything to the ports as it is the default value of the port pins after reset. So always use unusual and random values like AAH, 55H, etc.

THINK BOX 11.6

How can the ALE signal be useful to check whether 8051 is operational or not?

In every machine cycle, the 8051 generates two ALE pulses (except external data memory access). This can be checked on oscilloscope to verify whether 8051 is operational or not.

11.6 | PROGRAM MEMORY PROTECTION

Majority of the 8051 variants have feature of Program memory protection. The device contains a lock bit (or bits based on a variant), which once programmed, prevents access to on-chip program memory, i.e. internal ROM cannot be read by any external device. Moreover, the lock bits prevent further programming of the device. Erasing entire on-chip memory removes the effect of lock bits and device can be programmed again.

This feature helps to combat against software piracy. Refer datasheet of the device for the details of lock bits and their programming procedure.

THINK BOX 11.7

What are the advantages of CMOS (technology) based microcontrollers (or devices)?

- Consume less power
- Operates with large variation in temperature and operating voltages
- Higher operating speeds

POINTS TO REMEMBER

- The 8051 microcontrollers are 40-pin chips and has effectively 66 (40 normal pins+24 alternate functions of ports+2 for programming) pins.
- The ALE (Address latch enable, Pin 30) is used for de-multiplexing the address and data.
- Port 0 and 2 are collectively used to provide 16-bit address to external memory. P2 provides upper 8 bits of addresses A_{15} to A_8 and P0 provides lower 8 bits of addresses A_7 to A_0 as well as 8-bit data D_7 to D_0.
- For proper reset operation, the RST signal must be held high at least for two machine cycles.
- $\overline{EA}$ is used for selection of on-chip or off-chip ROM.

- To use P0 as an input or output, the external pull-up resistor must be connected with each pin.
- In order to configure P0 as an input, the port must be programmed by writing 1 to all bits.
- The idle mode stops program execution and contents of internal RAM are preserved. The oscillator continues to run but the clock is disconnected from the CPU. The timers and serial port operates normally.
- The microcontroller will come out of idle mode by the activation of any interrupt.
- The power-down mode stops on-chip oscillator, therefore program execution, timers and serial port operation are also stopped.
- The only way to come out of power-down mode is by applying reset (or power on reset) signal.

OBJECTIVE QUESTIONS

1. Which of the following instructions is used to access code memory?
   (a) MOVC A, @A+DPTR (b) MOVC A, @A+PC (c) MOVX A, @ DPTR (d) all of the above
2. $\overline{PSEN}$ is activated during,
   (a) MOVC A, @A+DPTR (b) MOVC A, @A+PC
   (c) op-code fetch from external program memory (d) all of the above
3. $\overline{EA}$ = 1 indicate,
   (a) only on-chip program memory is accessed
   (b) only off-chip program memory is accessed
   (c) on-chip + off chip (after on-chip range) program memory is accessed
   (d) program memory is not accessed
4. Port ___ needs external pull up resistor when used for I/O operation.
   (a) 0 (b) 1 (c) 2 (d) 3
5. For proper reset operation, RESET signal must be held high at least for __ machine cycles.
   (a) 1 (b) 2 (c) 3 (d) 4
6. The default value of port latches after reset is,
   (a) 00 (b) FFH (c) XX (d) depends on the 8051 variant
7. The alternate function of the port 2 is,
   (a) high order address bus (A15-A8) (c) low order address bus (A7-A0)
   (b) data bus (D7-D0) (d) none of the above
8. In an idle mode,
   (a) program execution is stopped (b) oscillator is stopped
   (c) contents of internal RAM is preserved (d) timers are stopped
9. In a power down mode,
   (a) program execution is stopped (b) oscillator is stopped
   (c) timers are stopped (d) all of the above
10. The frequency of signal at ALE pin is usually
    (a) F_{OSC} (b) $F_{OSC}/12$ (c) $F_{OSC}/6$ (d) $F_{OSC}/2$

Answers to Objective Questions

1. (a), (b) 2. (d) 3. (c) 4. (a) 5. (b) 6. (b) 7. (a)
8. (a), (c) 9. (d) 10. (c)

REVIEW QUESTIONS WITH ANSWERS

1. List the types of packages in which microcontrollers (or other chips) are available.

A. Dual Inline Package (DIP), Shrink Dual Inline Package (SDIP), Heat Dissipating Dual Inline Package(HDIP), Dual Inline Bent from Single inline package(DBS), Single Inline Package (SIL), Small Outline Package(SOP), Shrink Small Outline Package (SSOP), Thin Shrink Small Outline Package (TSSOP), Very Small Outline Package (VSO), Quad Flat Package (QFP), Low Profile Quad Flat Package (LQFP), Shrink Quad Flat Package (SQFP), Thin Quad Flat Package (TQFP), Plastic Leaded Chip Carrier (PLCC).

2. How many pins do microcontroller chips commonly have?

A. Microcontrollers are commonly available in 20, 40, 42, 44, 48, 52, 68, 84 and 100-pin packages.

3. The 8051 has a complete on-chip oscillator. True/False.

A. False. It has a partial on-chip oscillator circuit. External components (crystal and capacitors) are required to be connected to make oscillator functional.

4. What does a minimum speed of a microcontroller indicate?

A. The minimum speed indicates that some internal memories are dynamic (requires periodic refreshing) and must be operated above the minimum speed, otherwise data will be lost.

5. What is the meaning of having a minimum frequency of 0 Hz?

A. Internal registers and memory are static, i.e. data in internal registers will not be lost because of static memories which do not require refreshing.

6. Why are the lower address bus and data bus multiplexed?

A. To save number of pins of the microcontroller.

7. What is the minimum time for which reset signal must be applied?

A. 2 machine cycles.

8. What is the default value of all port latches?

A. All port latches have FFH by default after reset.

9. What are the alternate names assigned to port 0 pins?

A. AD7–AD0.

10. Which port of the 8051 has no dual functions?

A. Port 1.

11. External interrupt 0 input is active high. True/False.

A. False, it is active low.

12. Which pin is used as a timer 0 input?

A. Pin 14 (P3.4).

13. RESET is an active high signal. True/False.

A. True.

14. Port __ requires pull-up resistors when used as an I/O port.

A. 0.

15. When external memory is interfaced with the 8051, only P1 and P3 can be used for I/O operations. True/False.

A. True, because P0 and P2 are used as address/data bus when external memory is interfaced.

16. In which mode is the oscillator frozen?

A. Power-down mode.

EXERCISE

1. $\overline{\text{PSEN}}$ is commonly connected with ___ pin of the ROM.
2. List the conditions when the $\overline{\text{PSEN}}$ is activated.
3. $\overline{\text{PSEN}}$ is not activated when on-chip ROM is accessed. True/false.
4. The 8051 is effectively ___ pin microcontroller.
5. Show with the help of circuit diagram, how lower address bus and data buses are de-multiplexed.
6. Discuss the role of ALE pin.
7. Write the pin number of the following pins: RST, XTAL2 ,XTAL1,PSEN,ALE , EA.
8. What is meant by power on reset? How it can be implemented?
9. Discuss the functions of each port.
10. Discuss the difference between power down and idle modes.
11. How can 8051 come out of power down and idle modes?
12. Discuss the role of $\overline{\text{EA}}$ pin in program memory selection.
13. Discuss in detail alternate functions of port 3.
14. List the components required in a minimum component (cost) 8051 system.
15. Which pins of the 8051 are used for serial communication?
16. Out of two alternate functions of a pin, how microcontroller selects a particular function?
17. What is the frequency of the signal on ALE pin when microcontroller is connected with a crystal of 12 MHz?
18. What should be the minimum value of pull-up resistor connected to port 0?
19. Which SFRs represent power savings modes?

12

The 8051 Programming in C

Objectives

- Discuss the advantages offered by high-level languages
- List the data types for the 8051 supported by the Cx51 compiler
- Appreciate the significance of native word size
- Illustrate the uses of *bit*, *sbit* and *sfr* data types
- Show how to access different address spaces of the 8051 using memory-type specifiers
- Explain how to write the interrupt service routines and select register banks for context switching
- List the operators used in C language
- Learn how to use pointers to access different memory areas of the 8051
- To implement rotate operations in C
- Generate the time delays using C programs
- Discuss the tips to develop efficient programs
- Develop programs in C to exploit the features of the 8051
- Compare performance of assembly and C programs

Key Terms

- bdata
- bit
- Code
- Code Efficiency
- Cx51 Compiler
- data
- idata
- Inline Assembly
- Interrupt
- Machine Independent
- Memory-type Specifier
- Native Word Size
- Operator
- pdata
- Portability
- sbit
- sfr
- Time Delay
- unsigned char
- using
- xdata

Today, the C language is widely used in the development of embedded systems because of programming ease provided by it; it is fairly simple to learn and is microcontroller/processor independent. Moreover, compilers and third party support are readily and easily available. Programming in C (or in any high-level language) offers the following advantages:

1. High-level programs are portable across many hardware architectures, i.e. independent of a microcontroller. The programmer need not be aware of architectural details of specific microcontroller/processor. Therefore, when the microcontroller (target device) changes, only the modules for device drivers and initialization code needs modifications.
2. High-level program development cycle is shorter because of use of functions, standard library functions and modular programming. Library functions are easily and readily available, therefore, they are not required to be developed by a programmer.
3. Use of available device drivers for more common peripherals like timers, UART, I2C, etc., for different systems will save development time and efforts of a programmer. Modular programming approach facilitates reuse of program modules.
4. High-level language program facilitates use of control structures (e.g. while, do-while, break and for) and conditional statements (e.g. if, if-else, else-if and switch- case) to specify program flow by simple statements.
5. Assembly-language code can be inserted in a high-level program, which is referred as inline assembly, allowing direct hardware control as and when required.

In this chapter, we will discuss how to use features of the 8051 in C programs and basics of C programming in general.

12.1 | DATA TYPES FOR THE 8051 SUPPORTED BY CX51 COMPILER

The Cx51 compiler supports all standard C data types as well as additional data types developed specially to facilitate the use of the 8051 features. These data types along with their range of values are summarized in Table 12.1.

Table 12.1 *C data types*

Data type		Bits	Bytes	Range of values
Standard C data types				
signed	char	8	1	–128 to +127
unsigned		8	1	0 to 255
enum		8	1	–128 to +127
		16	2	–32768 to 32767
signed	short	16	2*	–32768 to 32767
unsigned		16	2*	0 to 65535
signed	int	16	2*	–32768 to 32767
unsigned		16	2*	0 to 65535
signed	long	32	4	–2147483648 to 2147483647
unsigned		32	4	0 to 4294967295
float		32	4	+1.175494E-38 to + 3.402823E+38
Additional data types for the 8051				
bit		1	-	0 or 1 (Bit addressable RAM only)
sbit		1	-	0 or 1 (Bit addressable SFR only)
sfr		8	1	0 to 255 (SFRs)
sfr16		16	2	0 to 65535

*It is compiler and/or OS dependent.

12.1.1 Native Word Size: *char*

Every microcontroller/processor has a native word size, i.e. word length of 8, 16 or 32, etc., bits. The microcontroller works most efficiently with a data of native word size. The 8051 and its variants are all 8-bit microcontrollers, therefore they work more efficiently with 8-bit data. Since *char* (or *signed char*; signed type modifier is default type) and *unsigned*

char are 8-bit data types, they are most widely used data types for the 8051. If the value of a variable is greater than 8 bits, always use the smallest possible data type; this will save memory used by a program.

12.1.2 Additional Data Types for the 8051

1. Bit

The 'bit' data type is used to define a one-bit variable. It is defined and declared as follows:

bit *variable_name* ; or
bit *variable_name* = *value*;

where *variable_name* is a name of the bit variable and *value* is value assigned to the bit. For example,

```
bit  output;                  // declare one bit variable output
bit  switch_flag = 0;         // assign value 0 to a variable switch_flag
bit  LED = input;             // assign value of  bit variable input to variable LED
```

All bit variables are stored in a bit addressable area of the 8051, i.e. from 20H- 2FH byte addresses in internal RAM. Since this area has only 16 bytes, a maximum of 128 bit variables may be declared within any one scope.

The limitations applied to the bit variables are that they cannot be declared as a pointer and array of type bits. For example,

```
bit *addr;
bit array [10];     are invalid declarations
```

2. sfr

The 'sfr' data type defines a Special Function Register (SFR) and assigns the name to it. It is declared as follows:

sfr *variable_name* = *address*;

where *variable_name* is the name of the SFR and *address* is the address of the SFR. For example,

```
sfr P0 = 0x80;          // Port 0 (address 80H) is defined as P0 and will be accessed using name P0
sfr P1 = 0x90;          // Port 1 (address 90H) is defined as P1 and will be accessed using name P1
sfr TMOD = 0x89;        // Timer Mode register  (address 89H) is defined as TMOD and
                        // will be accessed using name TMOD
sfr IE = 0xA8;          // Interrupt enable register (address A8H) is defined as IE and will be accessed using name IE
```

Note that any symbolic name may be used in sfr declaration. The variables declared as sfr are assigned values similar to any other C variable, for example,

```
P0 = 0x00;              // 0x00 is written into P0
TMOD= 0x02;             // 0x02 is written into TMOD register
```

3. sbit

The 'sbit' data type defines a bit within a special function register (SFR). It should be noted that bit can be only within bit-addressable SFRs. It is defined and declared as follows:

- sbit *variable_name* = *sfr_name* ^ *bit_position*; or
- sbit *variable_name* = *sfr_address* ^ *bit_position*; or
- sbit *variable_name* = *sbit_address*;

where *sfr_name* is a name of SFR already defined, *sfr_address* is address of a SFR, *bit_position* is position of the bit within the register and *sbit_address* is the bit address. For example,

(a)
```
sfr  PSW = 0xD0;        //define PSW
sbit P = PSW^0;         //parity flag (bit 0 of PSW) is now accessed using name P
sbit AC = PSW^6;        //auxiliary carry flag (bit 6 of PSW) is accessed using name AC
```

(b)
```
sbit P = 0xD0^0;        //parity flag is bit 0 of PSW, accessed using name P
sbit AC = 0xD0^6;       //auxiliary carry flag is  bit 6, accessed using name AC
```

(c)
```
sbit P = 0xD0;          //parity flag P has bit address D0H
sbit AC = 0xD6;         //auxiliary carry flag AC bit address D6H
```

THINK BOX 12.1

What is the difference between data types 'bit' and 'sbit'? Can we interchange their usage?
A 'bit' is used to define a bit in bit-addressable area 20H–2FH while 'sbit' is used to define a bit in bit-addressable SFRs. No, we cannot interchange their usage.

Example 12.1

Illustrate how *sfr* data type is used to access ports of the 8051 in a C program to perform the following operations:
(i) Send 00H and FFH to P2 and P1 respectively.
(ii) Read Port1 and send same to Port 3.
(iii) Read Port1 and send same to Port 2.

Solution:

```
sfr P1   = 0x90;      // Port 1 (address 90H) is defined as P1 and will be accessed using name P1 in a C program
sfr P2   = 0xA0;      // Similarly port 2  and port 3 are defined as P2 and  P3 respectively
sfr P3   = 0xB0;
      void main ( )
        {
                  unsigned char x, y;          // define positive 8-bit variables
                  P2= 00;                      // send 00H to Port 2
                  P1= 255;                     // send 255 (FFH) to Port 1
                  x = P1;                      // read Port1 and store in 8-bit variable x
                  P3= x;                       // send value of x to Port 3
                                               // above two lines effectively reads data from P1 and send to P3
                  P2= P1;                      // read P1 and send its contents on P2.
                                               // value can be read from port and can be directly sent to other port
        }
```

Note the use of data type unsigned char for *x* and *y*. When value of a variable is positive integer and less than or equal to 255, we should prefer to use *unsigned char*. It is the most used data type for the 8051 because it is an 8-bit microcontroller. In general, we should use smallest possible data type to save code memory (to reduce size of a program generated by a compiler).

When the above program is compiled, the compiler will replace addresses of the SFRs in place of their names that were defined using sfr data type. For example, statement P1=255 will be compiled in machine language as 75 90 FF (MOV 90H, #0FFH)

It should be noted that any name can be given to the special function registers. We should assign meaningful name to make programs more readable and easy to understand. For example, Port 1 may be defined (assigned name) SENSORPORT when sensors are connected to it, the statement x = P2 in a program will be written as x= SENSORPORT, which will make it more readable.

Example 12.2

Rewrite above program (Example 12.1) without using sfr declarations for ports.

Solution:

```
#include<reg51.h>
void main ( )
        {
        … // same as above program
        }
```

Note the first line of the program, it is
#include<reg51.h>

The included file reg51.h contains the 'sfr' declarations of all SFRs as well as 'sbit' declarations of bit addressable SFRs. Therefore, ports are directly accessed by their names, i.e. P0, P1, P2 and P3. These symbols are defined in included reg51.h file.

Example 12.3

Can we use any other names for SFRs which are already declared in reg51.h file?

Solution:

Yes, to make program more readable, we can give descriptive names to the already declared SFRs. For example, if Port 1 is connected with 8 switches and we want to continuously monitor and send the status of these switches to Port 2, which is connected with LEDs, we can write a program as follows:

```
sfr  input_switches = 0x90;          // port 1 is given name input_switches
sfr LEDs = 0xA0;                     // port 2 is given name LEDs
      void main ( )
         {
               while (1)             // continuously monitor
               {
               LEDs = input_switches;
               }
         }
```

Note that by using descriptive names, the program becomes more readable. Yet there is another way of using descriptive names. It is illustrated in the Example 12.5.

Example 12.4

Write a C program to send
(i) Values from 00H to FFH to P1 pins in step of 1, i.e. 00,01,02,…FEH,FFH
(ii) Values from 00H to FFH to P2 pins in step of 10, i.e. 00,10,20,…240,250 (00,0AH,14H,…F0H,FAH)

Solution:

```
#include <reg51.h>
void main ( )
{
         unsigned char x, y;                  // define 8-bit unsigned variables.

         for (x=0; x <= 255 ; x++)
               {
               P1 = x;                        // send value of x to P1 pins, send values from 00H to FFH in step of 1
               for (y=0 ; y<=100 ; y++);      // delay generated by software
               }
         for (x=0; x <= 255 ; x= x+10)
               {
               P2 = x;                        // send value of x to P2 pins, send values from 00H to FFH in step of 10
               for (y=0 ; y<=100 ; y++);      // delay generated by software
               }
         }
```

The exact delay generated by the statement 'for (y=0; y<=100; y++);' depends upon the compiler used and microcontroller on which the program is executed. Refer Section 12.10 for more details on how to generate time delays in a C program.

Example 12.5

Illustrate the use of #DEFINE directive and SFR data type.

Solution:

Assume that eight switches are connected to Port 2 and eight LEDs are connected to each pin of Port 1. Read status of each switch and show the status on LEDs as well as send inverted status of switches on P3. Send inverted content of P1 to P0.

```
#include<reg51.h>
```

```
#define LED_PORT P1;          // P1 is accessed using name LED_PORT
# define InPUT_SWITCH P2;     // P2 is accessed using name InPUT_SWITCH
sfr P3 = 0×B0                 // SFR address 0xB0 can be accessed using name P3. (It is only useful when we have not included
                              // reg51.h in our program.)
   void main ( )
   {
        unsigned char i;
        i= InPUT_SWITCH;      // read status of each switch from P2 and
        P1= i;                // *send to LEDs on P1
        P0= ~LED_PORT;        // copy inverted contents of P1 to P0
        P3= ~InPUT_SWITCH;    // send inverted status of switches to P3.
}
```

Note the use of DEFINE directive. It is used to assign meaningful names to registers, which will make the program more readable and easy to understand. SFR data type is used to access any SFR using their names (it is only useful when we have not included reg51.h in our program, otherwise names of all SFRs is defined in included reg51.h file).

***Note:** We cannot use a statement like,

LED_PORT= INPUT_SWITCH

because we cannot use the symbol defined using #DEFINE directive on the left-hand side of an expression.

Example 12.6

Assume that eight LEDs are connected to port P1. Write a C program that sends the count from 00 to FFH continuously (0000 0000 to 1111 1111 in a binary) on the LEDs.

Solution:

```
#include<reg51.h>
#define LED P1                // assign name LED to port P1
    void main ()
    {
    P1=0;                     // initialize P1 with value 0
    for(;;)                   // repeat given task continuously
          {
          LED++;              // increment contents of P1 (Delay may be introduced after this statement)
          }
    }
```

12.1.3 Implementing Infinite Loops in a C Program

Infinite loops are implemented using while (1) or for (;;) statements. The task(s) to be repeated continuously are placed in these loops. The generalized structure for these types of programs is as shown below:

```
      void main ( )
    {
          Initialization, declarations;
          while (1)     // repeat following tasks continuously
          {
          task 1
          …
          task 2
          …
          task n
          }
    }
```

The infinite loop can also be implemented using for (;;) loop as shown below:

```
for (;;)                    // repeat the following tasks continuously
{
task 1
…
task 2
…
task n
}
```

Note that the program execution will never come out of while or for loop. Also, see that semicolon is not used after *while* or *for* statements.

Example 12.7

Show how the assembly language instruction 'HERE: SJMP HERE' is implemented in a C program to stop program execution.

Solution:

The statements while(1); or for(;;); are used to implement 'HERE: SJMP HERE' instruction to stop execution of the program. These statements are written at the end of the program after completion of all tasks as shown below:

```
        void main ( )
{       Initialization, declarations;
        task 1
        …
        …
        task n
        while (1);      // wait here indefinitely or stop program execution
}
```

Refer Section 7.1.1 (How to stop the program execution in the 8051?) for explanation of how the 'HERE: SJMP HERE' instruction effectively stops the program execution.

Example 12.8

Write a C program to send
i) ASCII characters 0, 1, 2, 3, 4,5,6,7, A, B, C, D, E, F, G to Port 1
ii) A string "Microcontroller" to Port 2

Solution:

The given ASCII characters are defined as a string. The hexadecimal values of given characters are sent to Port 1 and the string is sent to port 2, one character at a time. Note that the number of characters to be sent to Port 1 and Port 2 is chosen to be equal for simplicity so that both operations can be performed in a single loop. To define ASCII character in a C program, we have to write ASCII characters in double quotes (" ").

```
#include <reg51.h>
void main (void)
  {
  unsigned char characters[]= "01234567ABCDEFG";
                                        // define given characters as a string
  unsigned char string[] = "Microcontroller";
  unsigned char i,j;                    // temporary variables
    for(i=0; i<15; i++)
    {
    P1=characters[i];                   // send given characters to port 1
    P2= string[i];                      // send string to port 2, one character at a time
    for(j=0; j<100; j++);               // delay
    }
  }
```

Example 12.9

Write an 8051 C program to send values -6,-4,-2, 0,2,4,6 to Port 1.

Solution:

The given numbers can be defined as an array of numbers as shown in a program.
Since negative numbers are also involved, we have to define an array of numbers of type signed char (or char).

```
#include <reg51.h>
void main (void)
        {
        char numbers[]={-6,-4,-2, 0,2,4,6};        // define given values as an array
        unsigned char i,j;                         // temporary variables
              for(i=0; i<7; i++)
              {
              P1=numbers[i];                       // send given numbers to port 1, one at a time
              for(j=0; j<100; j++);                // delay between two consecutive numbers
              }
        }
```

Example 12.10

Write an 8051 C program to toggle the contents of Port P1 40,000 times.

Solution:

Since we have to repeat the given operation 40,000 times, the variable to hold this count should be of type *int* because they can hold values up to 65535.

```
#include <reg51.h>
void main()
   {
   unsigned int i;                  // temporary variable of type int to hold value up to 40,000
   P1=0x00;                         // initialize Port 1 with 00
         for(i=0; i<=40000; i++)    // repeat operation for 40000 times
         {
         P1=~P1;                    // toggle contents of P1
         }
   }
```

Example 12.11

Assuming that 8 LEDs are connected to port 1, write a C program to flash LEDs 100 times.

Solution:

For flashing of LEDs, we need to continuously toggle contents of Port 1, but since microcontrollers execute instructions at a very fast rate, we may not be able to observe the toggling of LEDs. Therefore, we need to introduce delay before toggling the contents of Port 1. In this program, the delay is introduced by calling a function 'DELAY'; once again remember that exact delay generated by a function depends on compiler, microcontroller as well as clock frequency.

```
#include<reg51.h>
void DELAY(void);
void main()
      {
      unsigned char p;
         for ( p=0;p<100;p++)       // repeat following task 100 times
         {
         P1=0xFF;                   // turn ON all LEDs
         DELAY();                   // wait for some time
```

```
        P1=0x00;                // turn OFF all LEDs
        DELAY();                // wait for some time
        }
  }
  void DELAY(void)              // function  to generate delay
        {
        unsigned int  i;
        for(i=0; i<=65535; i++);
        }
```

Example 12.12

Implement the above program using DO-WHILE loop.

Solution:

```
#include<reg51.h>
void DELAY(void);
void main()
  {
  unsigned char p;
        p=0;                                    // initialize temporary variable with 0
        do
              {
                    P1=0xFF;                    // Turn ON all LEDs
                    DELAY();                    // Wait for some time
                    P1=0x00;                    // Turn OFF all LEDs
                    DELAY();                    // Wait for some time
                    p = p+1;
              }
        while ( p<100);                         // repeat above task 100 times
  }
  void DELAY(void)                              // function  to generate delay
              {
              unsigned int  i;
              for(i=0; i<=65535; i++);
              }
```

12.1.4 Bit Addressing in the C Language

All four ports and majority of the SFRs of the 8051 are bit addressable. We can access single bits of these registers without disturbing other bits. This will make programs more efficient. Data types *sbit* and *bit* are used to access single bits at a time.

Example 12.13

Demonstrate how to access individual port pins. Perform the following functions.
(i) Read port pin P1.2 and send the same on P2.5 continuously.
(ii) Toggle P2.2 continuously after some delay.

Solution:

```
#include<reg51.h>
sbit IN_BIT= P1^2;              // P1.2 is accessed using name IN_BIT
sbit OUT_BIT=P2^5;              // P2.5 is accessed using name OUT_BIT
sbit TOGGLE_BIT= P2^2;          // P2.2 is accessed using name TOGGLE_BIT
bit TEMP_BIT;                   // bit data type is used to access bit addressable area
```

```
void main ( )
{
int x;
IN_BIT=1;                          // configure P1.2 as an input
while (1)                          // repeat following steps continuously
        {
        TEMP_BIT= IN_BIT;          // read pin P1.2 and save in bit
                                   // addressable area
        OUT_BIT=TEMP_BIT;          // send stored value at P2.5
        TOGGLE_BIT=0;              // send 0 to P2.2
        for (x=0 ;x <=1000 ; x++ );// delay using software
        TOGGLE_BIT=1;              // send 1 to P2.2 (toggle P2.2)
        for (x=0 ; x<=1000 ; x++); // delay using software
        }
}
```

Example 12.14

Rewrite the above program more efficiently.

Solution:

An alternative short-hand and more efficient way to perform the same operation is given below.

```
#include<reg51.h>
sbit IN_BIT= P1^2;                 // P1.2 is accessed using name IN_BIT
sbit OUT_BIT=P2^5;                 // P2.5 is accessed using name OUT_BIT
sbit TOGGLE_BIT= P2^2;             // P2.2 is accessed using name TOGGLE_BIT

    void main ( )
    {
    int x;
    while (1)                      // repeat following steps continuously
        {
        OUT_BIT= IN_BIT;           // read pin P1.2 and send same on P2.5
        TOGGLE_BIT=~TOGGLE_BIT;    // toggle P2.2
        for (x=0 ; x<=1000 ; x++); // delay using software
        }
    }
```

Example 12.15

Write a C program to toggle MSB of Port 1 forever.

Solution:

```
#include<reg51.h>          // Include header file of 8051
sbit display= P1^7         // Define P1.7 as a display
  void main()
    {
    while (1)              // repeat following tasks forever
        {
        display=1;         // write 1 to Port P1.7
        display=0;         // write 0 to Port P1.7
        }
    }
```

The while loop in the above program can be written more efficiently as follows:

```
    display=1;             // Send 1 to Port P1.7
```

```
while (1)                    // repeat following tasks forever
      {
      display=~ display;     // toggle MSB of P1
      }
```

The delay can be added before toggling operation as per requirement.

Example 12.16

Write a C program to continuously monitor bit P1.7. If it is high, write FFH to P0; otherwise, write 00H to P0.

Solution:

```
#include<reg51.h>
sbit inbit=P1^7;                    // define P1.7 as inbit using sbit data type
   void main()
        {
        inbit =1;                   // configure P1.7 bit as an input
        while (1)                   // continuously monitor P1.7
                {
                if( inbit ==1)      // check if P1.7 is high or not
                      P0=0xFF;      // if P1.7 is high, send FFH to P0
                else
                      P0=0x00;      // or if P1.7 is low, send 00 to P0
                }
        }
```

Example 12.17

Write an 8051 C program to turn on and off LED connected port pin P1.0 30,000 times.

Solution:

```
sbit OUTBIT=0x90;                   //another  way to declare bit P1^0 ( bit address of P1.0 is 90H
   void main()
   {
   unsigned int i;
   OUTBIT=0;                        // initialize P1.0 with 00.
        for(i=0; i<30000; i++)      // loop to repeat operation 30000 times
        {
        OUTBIT =~ OUTBIT ;          // toggle P1.0
        }
   }
```

Delay may be added after togging as per requirements.

Example 12.18

Write an 8051 C program to get the status of bit P0.3, complement it and send it to P0.5 continuously.

Solution:

```
   #include <reg51.h>
   sbit inbit = P0^3;
   sbit outbit = P0^5;
   bit tempbit;
   void main()
   {
   inbit =1;                        // configure pin P0.3 as an input
   while(1)
            {
```

```
            tempbit=inbit;          // read a status of  P0.3
            tempbit=~tempbit        // complement the status of P0.3
            outbit=tempbit;         // send complement of P0.3 to P0.5
            }
   }
```

The while loop in the above program can also be written as,

```
   while(1)
         {
         outbit=~inbit              // read P0.3 , complement it and sent it to P0.5
         }
```

Example 12.19

Assume that switch SW is connected to P1.7 and LED is connected to P1.0.Write a program that monitors switch SW and when it is pressed, it flashes the LED five times.

Solution:

Assume that when switch is pressed logic 0 will be given to P1.7.

```
#include<reg51.h>
void DELAY (void);
sbit SW=P1^7;                           // switch SW is connected to Port 1.7
sbit LED =P1^0;                         // LED is connected to port P1.0
unsigned char i;
unsigned int j;
void main()
{
SW=1;                                   // configure P1.7 as an input
LED =0;                                 // initially LED is off
for (;;)                                // monitor status of switch continuously
   {
   while(SW==0)
            {
         for(i=0; i<10; i++)            // flash LED five times
                  {
                  LED=~LED;             // toggle the status of LED
                  for (j=0; j<65000; j++);  // delay
                  }
            }
   }
}
```

Example 12.20

Write a program to turn on the LED if temperature of furnace goes beyond predefined value.

Solution:

Assume that LED is connected to Pin P1.0 and temperature measurement circuit is available, which will set P1.1 when temperature is greater than predefined value.

```
#include <reg51.h>
sbit TEMPERATURE= P1^1 ;
sbit LED = P1^0 ;
void main ( )
{
```

```
        TEMPERATURE = 1;                    // programming the P1.1 as an input pin
        while (1)
        {
                if (TEMPERATURE == 1)       // checking P1.1 pin
                    LED = 1;                // turn ON LED, if P1.1 is high
                else
                    LED = 0;                // otherwise turn OFF LED
        }
}
```

Example 12.21

Write a program to read all pins of P0. If its value is greater than 50H, then set P1.0 else set P2.0.

Solution:

```
#include < reg51.h>
sbit PORT1_BIT = P1^0;
sbit PORT2_BIT = P2^0;

        void main ( )
        {
        unsigned char i ;
                PORT1_BIT=0;                // clear P1.0 and 2.0
                PORT2_BIT=0;
                i = P0;                     // read pins (data) from port 0
                if (i > 0x50)               // is data > 50H
                PORT1_BIT=1;                // if yes, P1.0 = 1
                else
                PORT2_BIT=1;                // otherwise, P2.0=1
        }
```

Example 12.22

Write an 8051 C program to send the addition of values –25 and 125 to Port P1.

Solution:

```
#include<reg51.h>
void main (void)
{
        signed char i, j;
        i= –25;
        j=125;
        P1=i+j;                 // P1= 100= 0x64
}
```

12.2 | ACCESSING MEMORY AREAS OF THE 8051

The Cx51 compiler provides access to all 8051 memory areas. Variables may be explicitly stored in a specific memory area as per requirement by including a memory-type specifier in the declaration of a variable.

12.2.1 Internal RAM (Data Memory)

A maximum of 256 bytes of internal RAM is available based upon the 8051 variant. The first 128 bytes of this memory area can be accessed directly as well as indirectly. The upper 128 bytes of internal data memory (80H-FFH) can be

accessed only indirectly (in 8052). There is also a 16-byte bit-addressable area starting from 20H to 2FH. Access to internal RAM is fast, but limited in amount. Internal RAM is of three types and accessed using specifiers as discussed below.

(a) data This specifier always stores variables into the first 128 bytes of internal RAM. Variables stored in this area are accessed using direct addressing.

(b) idata This specifier uses all 256 bytes (if it is present in the 8051 variant) of internal RAM. The variables in this area are accessed using indirect addressing which requires more time to access the data than direct addressing.

(c) bdata This specifier refers to the 16 bytes of bit-addressable RAM (20H to 2FH). This allows variables to be declared that may be accessed at the byte as well as bit level.

12.2.2 External RAM (Data Memory)

The 8051 based system can have up to 64K Bytes of external RAM. It may be accessed using following specifiers,

(a) xdata This specifier uses any location within entire 64KByte external data address space.

pdata This memory-type specifier accesses initial 256 bytes of external data memory that is accessed using MOVX @Ri instructions.

12.2.3 Program/Code Memory

Based on the 8051 variant, program memory may be internal, external, or may be a combination of both. It may be accessed using the **code** specifier. The memory area and memory type specifiers are summarized in Table 12.2.

Table 12.2 *Memory type and specifier*

Memory Type	Specifier	Description
Internal RAM	data	First 128 bytes 00H to 7FH
	idata	All 256 bytes
	bdata	Bit addressable area 20H to 2FH;
External RAM	xdata	Entire 64 KBytes
	pdata	First 256 bytes, accessed by MOVX @Rn
Program Memory	code	Entire 64 KBytes

The following statements shows use of memory type specifiers:

```
unsigned char data my_input;              // variable my_input will be stored in internal
                                          // RAM (00H-7FH)
char code string[] = "Microcontroller"; // string will be stored in code memory
long xdata out_array[50];                 // out_array will be stored in external RAM
float idata i,j;                          // i,j will be stored in internal RAM
char bdata switch_status;                 // switch_status will be stored in bit
                                          // addressable RAM (20H-2FH)
```

12.3 BIT ADDRESSABLE VARIABLES

Bit-addressable variables can be addressed as byte or as bits. The variables that are placed in the bit addressable area (20H-2FH) of the internal RAM are bit addressable. The variables declared with the **bdata** memory type are placed into the bit addressable area and variables declared with the **bdata** memory type must be global. The following declarations shows bit-addressable variables.

```
char bdata mydata;       // bit addressable variable mydata
char bdata array[4];     // bit addressable array
```

The variables *mydata* and *array* are bit-addressable. The bits of these variables may be accessed and modified directly using the **sbit** keyword. For example,

```
sbit mybit1 = mydata^1;         // bit 1 of mydata is assigned the name mybit1
sbit mybit3= mydata^3;          // bit 3 of mydata is assigned the name mybit3
sbit array7 = array[0]^7;       // bit 7 of array[0] is assigned the name array7
```

The following example shows how to assign the values to *mydata* and array bits using the above declarations.

```
array7 = 0;             // clear bit 7 in array[0]
mybit1 = 1;             // set bit 1 in mydata
```

12.4 INTERRUPT-SERVICE ROUTINES

The 8051 has five sources of interrupts. The interrupts along with their vector addresses are listed in the Table 12.3.

Table 12.3 *The 8051 interrupts and vector addresses*

Interrupt number	Interrupt	Vector address
0	External interrupt 0	0003H
1	Timer 0 interrupt	000BH
2	External interrupt 1	0013H
3	Timer 1 interrupt	001BH
4	Serial port interrupt	0023H

Interrupt-service routines are specified by the 'interrupt' function attribute. For example, the external interrupt 0 ISR can be defined as,

```
void ext0 (void) interrupt 0
     {
     // body of the interrupt service routine
     }
```

And timer0 ISR can be defined as,

```
void timer0 (void) interrupt 1
     {
         // body of the interrupt service routine
     }
```

The compiler will generate interrupt vector automatically, furthermore the function is terminated by **RETI** instruction.

'using' attribute

Register banks are useful when handling interrupts. For ISR, we can switch to a different register bank to reduce interrupt response time (or reduce interrupt latency). After completing the ISR, we will switch back to the original register bank before returning. This will preserve the contents of the register bank used in the main program without wasting time for saving each register on the stack.

The 'using' function attribute specifies the register bank, a function should use. For example,

```
void timer0(void) interrupt 1  using 2
     {
         // body of the interrupt service routine
     }
```

In above definition, register bank 2 will be used by timer 0 ISR.

THINK BOX 12.2

What can be an argument for the 'using' attribute? What operations are performed by it?

The argument for the using attribute can be 0 to 3. The 'using' attribute does the following operations in the function:

- The current register bank is saved on the stack upon function entry.
- The specified register bank is selected.
- The original register bank is restored before the function is exited.

12.5 OPERATORS

The Cx51 compiler supports all operators supported by ANSI C. The summary of arithmetic, relational, logical and bitwise operators with example and brief explanations is given in tables 12.4 to 12.6.

Table 12.4 *Arithmetic operators*

Operator	Example	Operation
*	i * j	Multiplication
/	i / j	Division
+	i + j	Addition
-	i - j	Subtraction
%	i % j	Modulo; Give the remainder of i divided by j
++	i++	Increment i after using it
--	--i	Decrement i before using it
+	+i	Unary Plus
-	-i	Negation

Table 12.5 *Relational and logical operators*

Operator	Example	Operation
>	i > j	Greater than; True (1) if i is greater than j, otherwise false (0)
>=	i >= j	Greater than or equal to; True (1) if i is greater than or equal to j, otherwise false (0)
<	i < j	Less than; True (1) if i is less than j, otherwise false (0)
!=	i != j	Not equal to; True (1) if i is not equal to j, otherwise false (0)
<=	i < = j	Less than or equal to; True (1) if i is less than or equal to j, otherwise false (0)
==	i == j	Equal to; True (1) if i equals j, otherwise false (0)
ll	i ll j	Logical OR; false (0) if both i and j are 0, otherwise True (1)
&&	i && j	Logical AND; True (1) if i as well as j are 1, otherwise false (0)
!	! i	Logical NOT, True (1) if i is 0, otherwise false (0)

Table 12.6 *Bitwise operators*

Operator	Example	Operation
~	~ i	Bitwise complement (NOT); changes 1 bits to 0 and 0 bits to 1
l	i l j	Bitwise OR of i and j
&	i & j	Bitwise AND of i and j
^	i ^ j	Bitwise XOR of i and j
>>	i >> 2	Right shift; bits in i shifted right 2 bit positions
<<	i << 4	Left shift; bits in i shifted left 4 bit positions

Example 12.23

Write an 8051 C program to toggle all the bits of P1 continuously with some delay using EX-OR operator.

Solution:

When any bit is EX-ORed with 1, we will get the complement of the bit as a result.

```
#include<reg51.h>
        void main ()
        {
        unsigned int i;
        P0=0x5A;                    // initialize P0 with value 5AH
        while (1)
        {
        P0=P0^ 0xFF;                // when any bit is EX-ORed with 1, we will get complement of the bit
        for  (i=0; i<60000; i++);   // delay
        }
        }
```

Example 12.24

Assume that two switches are connected to pins P1.0 and P1.1. Write a program to monitor the status of the switches and perform a task according to the status of switches as given in the following table.

P1.1	P1.0	Task
0	0	send ASCII of P to P0
0	1	send ASCII of Q to P0
1	0	send ASCII of R to P0
1	1	send ASCII of S to P0

Solution:

We will first read status of all pins of Port P1. Since we are interested only in P1.0 and P1.1, we will mask upper six bits (P1.2 to P1.7) using the AND operator.

```
#include<reg51.h>
void main()
{
unsigned char i;
P1= 0xFF;                       // configure P1 as an input port.
i=P1;                           // read status of P1
i=i & 0x03;                     // mask the unused bits, i.e. AND i with 00000011
        switch(i)               // perform task as per status of P1.0 and P1.1
        {
            case(0):            // if status of pins=00, send ASCII of P to Port0
            {
            P0= 'P';
            break;
            }
        case(1):                // if status of pins=01, send ASCII of Q to Port0
            {
            P0= 'Q';
            break;
            }
        case(2):                // if status of pins=10, send ASCII of R to Port0
            {
            P0= 'R';
            break;
            }
        case(3):                // if status of pins=11, send ASCII of S to Port0
            {
            P0= 'S';
            break;
            }
        }
  }
```

Example 12.25

Write a C program to read 1-bit data from ports P1.0 and P1.2. Perform OR operation between these two bits and sends the result to Port P1.7.

Solution:

```
#include <reg51.h>
sbit IN1 P1.0                   // define  pinP1.0 as IN1
sbit IN2 P1.2                   // define  pinP1.2 as IN2
sbit  OUT P1.7                  // define  pinP1.7 as OUT
      void main ()
      {
            IN1=IN2=1;          // configure P1.0 and P1.2 as an input
      while (1)                 // repeat continuously
            {
            OUT=IN1 | IN2;      //OR operation
            }
      }
```

Example 12.26

Write a program to convert an array of 5 packed BCD numbers into array of equivalent ASCII numbers.

Solution:

One BCD number when converted into ASCII will require two bytes. For example, BCD 45 will be represented in ASCII as 34 35.

```
#include <reg51.h>
            void main ( )
            {
            unsigned char i;
            unsigned char BCD[]={(0x59,0x67,0x38,0x86,0x21)} ;
            unsigned char ASCII[10];
            for (i=0 ;i<5;i++)
            {
            ASCII[2*i]= ((BCD[i] & 0xF0)>>4) + 0x30;        // mask lower nibble, shift upper nibble to lower nibble position and add 30H
            ASCII[(2*i)+1]= (BCD[i] & 0x0F) + 0x30;         // mask upper nibble and add 30H
                        }
            }
```

Example 12.27

Write an 8051 C program to convert ASCII digits '8' and '6' to packed BCD and send them on P1.

Solution:

The ASCII codes for 8 and 6 are 38H and 36 respectively, the packed BCD for these two numbers will be 86H. Therefore, to perform this conversion we need to mask upper nibbles of both digits (to get 08 and 06) and then shift left first digit by four nibbles (to get 80) and combine (using OR operator) both digits.

```
#include<reg51.h>
            void main(void)
            {
            unsigned char digit1=  '8';     // ASCII digit 8
            unsigned char digit2 = '6';     // ASCII digit 6
            unsigned char PACKED;
            digit1=digit1 & 0x0F;           // mask upper nibble of 38 to get unpacked  BCD 04
            digit2=digit2 & 0x0F;           // mask upper nibble of 36 to get unpacked  BCD 07
            digit1=digit1<<4;               // shift 08 by four bits to get 80H
            PACKED=digit1 | digit2;         // get packed BCD
            P1= PACKED;                     // send to port 1
            }
```

Example 12.28

Write a C program to convert a binary (Hex) data into its equivalent BCD data.

Solution:

For an 8-bit binary number, at most three digits are required to represent equivalent BCD number. For example,

$FFH_{Binary} = 255_{BCD}$ (three digits)

$64H_{Binary} = 100_{BCD}$

The conversion is achieved by successive division of binary number by 100, 10, and 1. (In general, it is 10^N, 10^{N-1}, 10^{N-2}, 10^1). Note that division by 1 may be skipped.

Consider for binary number 11111111B (FFH),

```
#include<reg51.h>
void main(void)
        {
        unsigned char x, bindata, lowdigit, midddledigit, upperdigit;
        bindata= 0xFF;              // binary number to be converted
        upperdigit = bindata/100;   // divide binary number by 100 to get upper digit
        x= bindata%100;             // get remainder after division by 100
        middledigit= x/10;          // divide remainder by 10 to get middle digit
        lowerdigit= x%10;           // remainder after second division is lower digit
        }
```

12.6 | DATA SERIALIZATION USING PORT PINS

Though the 8051 has dedicated a serial port for the data transmission and reception, we can also transmit and receive data serially using port pins. This can be done by transmitting/receiving one bit at a time. This method has advantage that we can control the sequence data bits (MSB first or LSB first), number of bits and delay between each bit. The disadvantage is that the programmer has to write instructions for transmitting/receiving each bit and the microcontroller will remain busy during that time. The following examples demonstrate the data serialization using port pins.

Example 12.29

Write an 8051 C program to send byte 19H serially one bit at a time through P1.1. Send MSB first.

Solution:

```
#include<reg51.h>
sbit outbit=P1^1;
sbit MSB_A=0xE7;                // define ACC.7 as MSB_A
void main (void)
{
unsigned char sendbyte=0x19;    // define 19H as sendbyte
unsigned char i;
ACC=sendbyte;                   // copy 19H in to accumulator
for(i=0; i<8; i++)              // loop for 8 bits
        {
        outbit=MSB_A;           // send MSB  to  P1.1
        ACC=ACC<<1;             // shift left contents of Accumulator by one bit so that in
                                // next iteration, next bit will be sent to P1.1
        }
}
```

Example 12.30

Modify the above program to send LSB first.

Solution:

```
#include<reg51.h>
sbit outbit=P1^1;
sbit LSB_A=0xE0;                // define ACC.0 as LSB_A
void main (void)
{
unsigned char sendbyte=0x19;    // define 19H as sendbyte
unsigned char i;
ACC=sendbyte;                   // copy 19H into accumulator
for(i=0; i<8; i++)              // loop for 8 bits
        {
        outbit=LSB_A;           // send LSB  to  P1.1
        ACC=ACC>>1;             // shift right contents of Accumulator by one bit so that
                                // in next iteration, next bit will be sent to P1.1
        }
}
```

Example 12.31

Write a C program to receive a byte 35 serially one bit at a time through P2.5. The LSB will be received first.

Solution:

```
#include<reg51.h>
sbit inbit=P2^5;                // define P2.5 as inbit
sbit MSB_A=ACC^7;               // define ACC.7 as MSB_A
void main ()
{
unsigned char i;
inbit =1;                       // configure pin P2.5 as an input
for(i=0; i<8; i++)              // loop to receive 8 bits
        {
        MSB_A=inbit;            // receive bit from P2.5
        ACC=ACC>>1;             // shift Accumulator right by one bit so that after 8
                                // iterations, the bit received first will move into
                                // LSB of Accumulator
        }
}
```

12.7 | ROTATE OPERATIONS IN C

The limitation of shift operation is that the bit which is shifted out from MSB (for shift left operation) or the bit which is shifted out from LSB (for shift right operation) is lost. The rotate operation will move the bit shifted out from MSB into LSB (for rotate left operation) or the bit shifted out from LSB in to MSB (for rotate right operation). Thus, rotate operation treats the data as circular buffer of bits. 1 bit rotation operation for 8-bit data is illustrated in Figure 12.1.

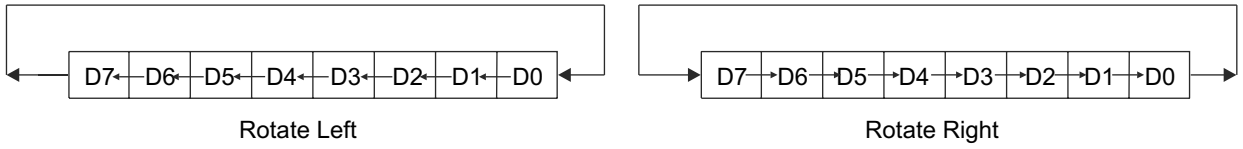


Fig. 12.1 *Rotation of 8-bit data by 1 bit*

12.7.1 Left Rotation

The left rotate operation on *w* bit wide number *n* by *d* bits can be performed in the following steps:

1. Shift left the number *n* by *d* bits ($n << d$)
2. Shift right the original number *n* by *(n – d)* bits $\{n >> (w–d)\}$
3. Perform the bitwise OR operation on the results obtained in steps 1 and 2: $(n << d) | \{n >> (w–d)\}$

Example 12.32

Develop the function to rotate left the 8-bit number by specified number of bits.

Solution:

```
#include<reg51.h>
unsigned char leftrotate(unsigned char, unsigned char);
void main ()
        {
        unsigned char i,k;
        k=0x80;                 // initialize the K with 8-bit number
        i= leftrotate(k,3);     // call the function to rotate left K by 3 bits
        while (1);
        }
unsigned char leftrotate(unsigned char n, unsigned char d)
        {
        unsigned char j;
        j= (n << d)| (n>>(8-d));    // rotate left n by d bits
        return j;
        }
```

12.7.2 Right Rotation

The right rotate operation on *w* bit wide number *n* by *d* bits can be performed in the following steps:

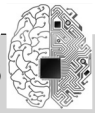
1. Shift right the number *n* by *d* bits ($n >> d$)
2. Shift left the original number *n* by *(n–d)* bits $\{n<<(8–d)\}$
3. Perform the bitwise OR operation on the results obtained in steps 1 and 2: $(n >> d) | \{n<< (w–d)\}$

Example 12.33

Develop the function to rotate right the 8-bit number by specified number of bits.

Solution:

```
#include<reg51.h>
unsigned char rightrotate(unsigned char, unsigned char);
void main ()
        {
        unsigned char i, k;
        k=0x81;                 // initialize k with the 8-bit number
        i= rightrotate(k, 2);   // call the function to rotate right k by 2 bits
        while (1);
        }
unsigned char rightrotate(unsigned char n, unsigned char d)
        {
        unsigned char j;
        j = (n >> d)| (n<< (8-d));   // rotate right n by d bits
        return j;
        }
```



THINK BOX 12.3

How does the rotate operation in C differ from other operations?

Usually the C language statements are compact compared to equivalent assembly-language statements, but for the rotate operations, it is reverse.

12.8 | POINTERS

The Cx51 compiler supports two types of pointers: generic and memory specific. The generic pointer occupies three bytes: the first byte is a selector which indicates the type of memory the pointer points to (either data, idata, xdata, pdata or code). The remaining two bytes are used to hold actual address in the memory address space. For memory types such as data, idata and pdata, only one byte address is required, thus the other byte is not used for actual address but is wasted. To avoid such wastage of space, Cx51 also supports memory-specific pointers, which require less space. Table 12.7 summarizes the bytes required for generic and memory-specific pointers.

Table 12.7 *Pointers supported by the Cx51 compiler*

Pointer Type	Size in Bytes
Generic pointer	3
CODE pointer	2
XDATA pointer	2
PDATA pointer	1
DATA pointer	1
IDATA pointer	1

The advantage of a generic pointer is that it can be used without worrying about actual location of data, but it will result in larger, slower and inefficient code. On the other hand, usage of memory-specific pointers demands little more pain from the programmer, but it will provide smaller, faster and efficient code. The following programs illustrate the use of memory-specific pointers.

Example 12.34

Write a program to transfer a string stored in internal RAM to external RAM at the address 0000H onwards.

Solution:

```
#include <absacc.h>                    // include file to access specific addresses

void main()
{
unsigned int i=0;
char data string[]="MICROCONTROLLER";
                                       // define a string in internal RAM
 char data *str_ptr;                   // pointer to internal RAM location
 str_ptr=string;                       // srt_ptr pointer points to first byte of string
 while (*str_ptr)                      // read until end of string (null character)
                {
                XBYTE[i]=*str_ptr;     // read form string and store to external RAM
                str_ptr++;             // point to next element of string
                i++;                   // next address in external RAM
                 }
while (1);                             // wait indefinitely
 }
```

Note that the XBYTE is the macro used to access the bytes at absolute address in external data memory of the 8051. This macro is defined in 'absacc.h' file.

Example 12.35

Write a program to transfer a string stored in code memory to internal RAM at address 40H onwards.

Solution:

```
#include <absacc.h>                    // include file to access specific addresses

void main()
{
unsigned char i=0x40;                  // starting address where string will be stored
char code string[ ] = "MICROCONTROLLER";
                                       // define string in code memory
char code *str_ptr;                    // pointer to code memory
 str_ptr=string;                       // srt_ptr pointer points to first byte of string
 while (*str_ptr)                      // read until end of string (null character)
        {
        DBYTE[i]=*str_ptr;             // read from string and store to internal RAM
        str_ptr++;                     // point to next element of string
        i++;                           // next address in internal RAM
        }
while (1);                             // wait indefinitely
 }
```

Note that the DBYTE is the macro used to access the bytes at absolute address in internal data memory of the 8051. This macro is defined in 'absacc.h' file.

12.9 POINTERS TO ABSOLUTE ADDRESSES

In the microcontroller (or an embedded system), the addresses of ROM, RAM and peripherals are fixed. If we want to access the data from these fixed addresses, we should use pointers to absolute addresses. The method of initializing a pointer at absolute address is illustrated in the following declaration/definitions.

```
char *ptr = 0x1000 ;      // Declare a generic character pointer to point to address 0x1000
char code *ptr=0x200;     // Declare a character pointer to point to address 0x200 in
                          // code memory
char *ptr2 ;              // Declare a generic character pointer
ptr2 = (char *) 0x1000;   // Initialize pointer ptr2 to point to address 0x1000
char idata *ptr3;         // pointer to internal RAM location
ptr3= (char *) 0x80;      // Initialize pointer ptr3 to point to internal RAM address 0x80
```

Examples 12.36 and 12.37 illustrate the use of absolute pointers.

Example 12.36

Write a program to transfer a string stored in code memory to internal RAM at the address 40H onwards.

Solution:

```
void main()
{
unsigned char i=0;
char code string[]="MICROCONTROLLER";
                                  // define string in code memory
char data *ptr;                   // pointer to internal RAM location
ptr= (char *) 0x40;               // Initialize pointer ptr to point to internal RAM address 0x40
```

```
while(string[i])                    // read until end of string (null character)
        {
        *ptr=string[i];             // read form string and store to internal RAM 40H onwards
        i++;                        // next address in code memory
        ptr++;
        }
        while (1);
}
```

Note that this program is similar to Example 12.35 except the following:
(i) We have not included absacc.h file to access specific addresses directly.
(ii) Pointer to absolute address is used.

Example 12.37

Write a program to transfer a string stored in data memory to port1 (one byte at a time). Provide a delay between each byte.

Solution:

```
void delay (void);                  // declaration of delay routine
void main()
{
unsigned char i=0;
char data string[]="MICROCONTROLLER";
                                    // define string in internal RAM
char idata *P1_ptr;                 // pointer to internal RAM location
 P1_ptr= (char *) 0x90;             // initialize pointer P1_ptr to point to port1 (0x90)
while(string[i])                    // read until end of string (null character)
        {
        *P1_ptr=string[i];          // read form string and send to port 1
        i++;                        // next address in internal RAM
        delay();                    // delay
        }
        while (1);
}

void delay (void)                   // delay function definition
{
unsigned int i;
for (i=0;i<60000; i++);
}
```

Note that idata memory type specifier is used to point to Port 1 SFR.

12.10 | TIME DELAYS IN C

When writing delay routines using various loops in C, we must consider following important issues:

- The variants of the 8051 use different number of clock cycles per machine cycle; therefore, instruction execution speed changes as per number of clocks per machine cycle. The duration of a machine cycle also depends upon the crystal frequency connected to it.
- Different compilers produce different machine codes and, therefore, machine-language instructions generated corresponding to a loop will vary across the compilers, which in turn will require different execution time for same C statements.

Because of the above issues, we may not get exact delay using loops. The programmer must see equivalent assembly code generated by a compiler and he/she should make corresponding changes in a C program (by trial-and-error method) to generate exact delay. The better way to generate exact delay is to either use inline assembly programs or timers available in the 8051.

12.11 | INCREASING THE CODE EFFICIENCY

The 8051 has limited code and data memory (maximum 64Kbytes of code as well as data space). Therefore, we must use them effectively. C programs generally produce larger machine codes which may affect code efficiency in terms of execution speed and memory requirements. Some common tips are discussed in the following section which may help produce more efficient programs.

12.11.1 Variable Size

The microcontroller/processor works most efficiently with data of native word size. The 8051 and its variants are all 8-bit microcontrollers; therefore, they work more efficiently with 8-bit data. Operations that use **char** and **unsigned char** (8-bit data types) are much more efficient than operations that use **int** or **long** types. Handling of larger data types require the use of additional machine-language instructions. By consistently using char or unsigned char whenever possible in a program, we can reduce the size (bytes) of a program.

12.11.2 Use of Unsigned Data Types

The 8051 and its variants does not support direct operations with signed numbers; therefore, compiler will generate larger code to deal with signed data types. Use unsigned data type wherever possible as it will produce smaller machine code.

12.11.3 Use of Bit Variables

When we are dealing with Boolean variables (or flags which can have only two possible values 0 or 1), we should use the bit type instead of an unsigned char. This will help save seven bits, moreover, since bit variables are always stored in internal RAM; therefore, their access will be faster (one machine cycle).

12.11.4 Inline Functions

Use inline functions when functions are called frequently but contains *few bytes of code*. This eliminates the runtime overhead (saving the return address and variables on the stack) related with the function calls and return. Inline functions are examples of how execution speed can be compromised with code size. Inline functions will increase the size of a program that is directly proportional to the number of times the function is called. The program using inline function will run faster, but requires more program memory (ROM).

12.11.5 Use of Internal RAM

Accessing the internal RAM is faster than accessing the external RAM (data memory). Therefore, place most frequently used variables in internal RAM using memory-type specifiers *data, idata and bdata*. Place less frequently used variables in the external RAM using memory-type specifiers *xdata* and *pdata.* However, a programmer should remember that there is a limited amount of internal RAM, and all program variables may not fit into this memory area. In such a case, we must locate some variables in other memory areas.

12.11.6 Inline Assembly/Hand-Coded Assembly

Use assembly language for typical and complex program modules. This will make them as efficient as possible. The compilers produce better machine code than the average programmer; however, for a good programmer there is still an opportunity to make a program more efficient.

12.11.7 Avoid Standard Library Routines

One of the best ways to achieve higher code efficiency (to reduce the size of a program) is to avoid using larger library routines wherever possible. Use of these library functions will produce larger machine code bytes because they are designed to handle all possible cases. Therefore, use inline assembly instructions to handle only specific case.

12.11.8 Use of Intrinsic Functions

Intrinsic functions allow access to specific 8051 instructions to save code space. Most such functions correspond directly to a single assembly-language instruction. For example, _crol_ function corresponds to RL A instruction, _cror_ function correspond to RR A instruction, _nop_ correspond to NOP instruction. The Cx51 compiler provides intrinsic library functions that are defined in the intrins.h file. Refer Cx51 User Guide available in Keil µVision 4.0 help.

12.12 | PERFORMANCE COMPARISON BETWEEN ASSEMBLY AND C PROGRAMS

We have discussed comparison between assembly and high-level (C) language in Section 3.1, Moreover, the advantages of high-level languages are discussed in the beginning of this chapter. The detailed comparison between these languages

is made in this section with the help of a few programming examples. For the sake of clarity, the equivalent programs in both languages are given side by side and comparison is made in the tabular form.

Example 12.38

Write a program to copy a block of data (array) from one location to other location in the internal RAM.

Solution:

This program (assembly) is already discussed in Example 9.8; refer that example for program-development logic and other details. The equivalent programs in both languages are given below along with performance comparison.

Assembly-language program	C language program

ORG 0000H
MOV R0, #40H
MOV R1, #50H
MOV R2, #0AH
NEXT: MOV A,@R0
MOV @R1, A
INC R0
INC R1
DJNZ R2, NEXT
HERE: SJMP HERE
END
	#include <absacc.h>
	void main()
	{
	unsigned char i;
	for (i=0x40; i<0x4A ; i++)
	DBYTE[0x10+i]=DBYTE[i];
	while (1);
	}
	// DBYTE is the macro used to access the bytes at
	// absolute address in internal data memory

Comparison between these two programs is given below:

Comparison criteria	Assembly language program	C language program
Ease of programming	Low (more difficult)	High (easy)
Memory requirements (Bytes)* (Final machine code)	14	18 (main program) +15 (startup file overhead) =33
Execution time (Machine cycles)#	65	123 (main program) +389 (startup file overhead) =512

* The information of number of bytes required can either be calculated manually or can be obtained from list file (easily for assembly programs) and disassembly window and /or output window (for C programs) of Keil µvision 4.0. IDE. Refer Appendix B for use of Keil µvision 4.0. IDE.

The information of number of machine cycles taken for execution can either be calculated manually (for assembly programs) and from 'States' from register window during debugging (for both type of programs) with Keil µvision 4.0. IDE. Refer Appendix B for use of Keil µvision 4.0. IDE

Example 12.39

Write a program to add two 16-bit numbers 42E1H and 255CH.

Solution:

This program (assembly) is already discussed in Example 5.4; refer that example for other details. The performance comparison along with equivalent programs in both languages is given below.

Assembly-language program	C language program

ORG 0000H
MOV A, #0E1H
ADD A, #5CH
MOV R5, A
MOV A, #42H
ADDC A, #25H
MOV R7, A
HERE: SJMP HERE
END
	void main()
	{
	int a, b, c;
	a=0x42E1,b=0x255C;
	c= a + b;
	while (1);
	}

Comparison between these two programs is given below:		
Comparison criteria	**Assembly language program**	**C language program**
Ease of programming	Low (more difficult)	High (easy)
Memory requirements (Bytes) (Final machine code)	12	14(main program) + 15 (startup file overhead) = 29
Execution time (Machine cycles)	8	8 (main program) + 389 (startup file overhead) = 397

Example 12.40

Write a program to find the largest number from given array stored in code memory.

Solution:

This program (assembly) is already discussed in Example 9.15; refer that example for other details. The comparison of programs in both languages is given below.

Assembly-language program	**C language program**

ORG 0000H
MOV R2, #0AH
MOV DPTR, #ARRAY
MOV R3, #00H
REPEAT: CLR A
MOVC A, @A+DPTR
MOV 10H,A
CLR C
SUBB A,R3
EXCHANGE: JC NEXT
MOV R3, 10H
NEXT: INC DPTR
DJNZ R2, REPEAT
HERE: SJMP HERE
ARRAY: DB 32H, 024H, 45H, 76H,
23H, 39H, 35H, 87H, 21H, 56H
	void main()
	{ code unsigned char array [] = {0x32,0x24, 0x45,0x76, 0x23, 0x39, 0x35,
	0x87, 0x21,0x56};
	unsigned char i, j=0;
	for (i=0;i<10;i++)
	{
	if (j<array[i])
	j=array[i];
	}
	while (1);
	}

Comparison between these two programs is given below:		
Comparison criteria	**Assembly language program**	**C language program**
Ease of programming	Low (more difficult)	High (easy)
Development and debug time (Mental pain!)	High	Low
Readability and ease of modification	Less (difficult to understand)	High (easy to understand)
Memory requirements (Bytes) (Final machine code)	32	32(main program) +15 (startup file overhead) =47
Execution time (Machine cycles)	130	147(main program) +389 (startup file overhead) = 536

Example 12.41

Write a program to multiply two 16-bit numbers. The multiplier is stored in R1-R0 (least significant byte in R0). The multiplier is stored at R3-R2. Store result at R7-R6-R5-R4.

Solution:

This program (assembly) is already discussed in Example 9.21; refer that example for other details. The equivalent programs in both languages are given below along with performance comparison.

Assembly-language program	**C language program**

ORG 0000H
MOV A, R0
MOV B, R2
MUL AB
MOV R4, A
MOV R5, B
MOV A, R3
MOV B, R1
MUL AB
MOV R6, A
MOV R7, B
MOV A, R2
MOV B, R1
MUL AB
ADD A , R5
MOV R5, A
MOV A,B
ADDC A, R6
MOV R6, A
MOV A, R7
ADDC A, #00H
MOV A, R3
MOV B, R0
MUL AB
ADD A, R5
MOV R5, A
MOV A,B
ADDC A, R6
MOV R6, A
MOV A, R7
ADDC A, #00H
MOV R7, A
HERE: SJMP HERE
END
	void main()
	{
	unsigned int a, b;
	long c;
	a= 1234, b=5678;
	c= (long) a* b;
	while (1);
	}

Comparison between these two programs is given below:

Comparison criteria	**Assembly language program**	**C language program**
Ease of programming	Low (more difficult)	High (easy)
Development and debug time (Mental pain!)	High	Low
Readability	Less (difficult to understand)	High (easy to understand)
Ease of modification	Less (difficult to modify)	High
Memory requirements (Bytes) (Final machine code)	43	113(main program) +15 (startup file overhead) =128
Execution time (Machine cycles)	51	135(main program) +389 (startup file overhead) = 524

Example 12.42

Write a program to generate sine wave (with use of digital to analog converter).

Solution:

These programs (assembly and C) are discussed in **Interfacing Example 19.14.** Refer that example for other details. The equivalent programs in both languages are given below along with performance comparison.

Assembly-language program

```
        ORG 0000H
        MOV DPTR, #LOOKUP
REPEAT: MOV R1, #18
        CLR A
NEXT:   MOV R3, A
        MOVC A,@A+DPTR
        MOV P2, A
        MOV A, R3
        INC A
        DJNZ R1, NEXT
        CLR A
        MOV R2, #18
        NEXT1:
        MOVC A,@A+DPTR
        CLR C
        MOV R4, A
        MOV A, #0FFH
        SUBB A, R4
        MOV P2, A
        MOV A, R3
        INC A
        DJNZ R2, NEXT1
        SJMP REPEAT
LOOKUP: DB 128, 150, 171, 191, 209, 225, 238, 247
DB 252, 255, 253, 247, 238, 225, 209, 191
DB 171,150
END
```

C language program

```
#include<reg51.h>
void DELAY (void);
void main()
{
unsigned char i,
samples[ ]={128, 150, 171, 191, 209, 225, 238, 247, 252, 255, 253, 247, 238, 225, 209, 191, 171, 150};

while(1)
      {  for ( i=0; i<18; i++)
  {P2=samples[i];
          DELAY();
  for ( i=0; i<18; i++)
{P2 = 255-samples[i]}
            DELAY();
                }
       }
}
void DELAY(void)

       {unsigned int j ;
        for(j=0; j<10000; j++);
        }
```

C program using library function

```
#include<reg51.h>
#include<math.h>
void DELAY (void);
void main()
{
unsigned int i;
while(1)
{  for ( i=0; i<360; i=i+10)
{P2= (1.28 +1.28*sin((3.14/180)* i))* 99.61;
  DELAY();   }
}
}
void DELAY (void)
{unsigned int j;
for(j=0; j<10000; j++);
  }
```

Comparison between these two programs is given below:

Comparison criteria	Assembly program	C program	C program using library function
Ease of programming	Low (Difficult)	High (Easy)	Highest (Easiest)
Readability	Less (difficult to understand)	High (easy to understand)	High (easy to understand)
Ease of modification	Less (difficult to modify)	High	Highest
Memory requirements (Bytes)	50	351+15 =366	1439+ 15=1454
Execution time (Machine cycles) for one cycle from 0º to 360º.	368 machine cycles	971 (excluding delay routine)	164186 (excluding delay routine)

Note that when we use library functions in our programs (sine function in this example), the size of the program would be very large. The reason is that the library functions are designed to handle all types of data (general purpose). Therefore, they are very complex and larger in size. But the advantage offered by them is that they reduce programming efforts.

THINK BOX 12.4

Give the comparison of same code written in assembly and in C.

Assembly-language programs give better performance in terms of memory requirements and execution speed, but require more programming efforts, while high-level (C) language programming provides ease of programming at the expense of performance. Moreover, it should be evident from the above programs that if the target microcontroller (the controller for which we write the programs) changes, the assembly-language programs must be completely modified but C programs require little or no modifications. This means C programs are portable across different microcontrollers.

POINTS TO REMEMBER

- Additional data types for the 8051 are *bit*, *sbit* and *sfr*.
- Since *char* (or *signed char*; signed type modifier is default type) and *unsigned char* are 8-bit data types, they are most widely used data types for the 8051.
- Always use the smallest data type possible to save memory used by a program.
- The 'bit' data type is used to define a one-bit variable. All bit variables are stored in a bit-addressable area.
- The 'sbit' data type defines a bit within a special function register (SFR). It should be noted that bit can be only within bit addressable SFRs.
- Variables may be explicitly stored in a specific memory area as per requirement by including a memory-type specifier in the declaration of a variable.
- *data, idata and bdata* are the type specifiers to access internal RAM, *xdata* and *pdata* are used to access external RAM, while *code* is used to access program memory.
- Interrupt-service routines are specified by the 'interrupt' function attribute.
- The 'using' function attribute specifies the register bank a function should use.

OBJECTIVE QUESTIONS

1 The data type that represents native word size of the 8051 is,
(a) char (b) unsigned char (c) int (d) unsigned int

2. Which of the following declarations is invalid?
(a) bit *addr (b) bit output (c) long xdata out_array[50] (d) char code string[ ] = "Microcontroller"

3. A variable temperature can have any value between 0 and 150, the variable should be declared with data type ___ for efficient use of memory.
(a) char (b) unsigned char (c) int (d) unsigned int

4. A variable temperature can have any value between -50 to +120; the variable should be declared with data type ___.
(a) char (b) unsigned char (c) int (d) unsigned int

5. The 'xdata' memory type specifier is used to store variables in,
(a) first 128 bytes (00H to 7FH) of internal data memory
(b) entire 64 KBytes of external data memory
(c) entire 64 KBytes of program memory
(d) first 256 bytes of external data memory

6. 'pdata' memory type specifier is used to store variables in,
(a) first 128 bytes (00H to 7FH) of internal data memory

(b) entire 64 KBytes of external data memory
(c) entire 64 KBytes of program memory
(d) first 256 bytes of external data memory

7. Which of the following definitions is invalid?
   (a) sbit AC = PSW^6 (b) sbit AC = 0xD0^6 (c) sbit AC = 0xD6 (d) sbit AC = 0xD0:06
8. Which of the following will not help in improving code density of a program?
   (a) Use of native word size (b) Use of inline assembly
   (c) Use of standard library routines (d) Use of unsigned data types
9. Time delay generated by a C program depend on,
   (a) target 8051 variant (b) compiler (c) clock frequency (d) all of the above
10. The range of values that can be stored in a variable of type of 'sfr' is,
    (a) –128 to +127 (b) 0 to 255 (c) –1 to 1 (d) 0 or 1

Answers to Objective Questions

1. (b)	2. (a)	3. (b)	4. (a)	5. (b)
6. (d)	7. (d)	8. (c)	9. (d)	10. (b)

REVIEW QUESTIONS WITH ANSWERS

1. List the key features of a high-level language.

A. a) It is easy to develop, modify and update the programs.
b) It is processor independent and, therefore, portable across many architecture with little modifications.
c) Availability and use of library routines will reduce program development time and efforts.

2. What is meant by native word size? What is native word size of the 8051?

A. Word length of a microcontroller/processor is called native word size. Native word size of the 8051 is 8 bits because it is an 8-bit microcontroller.

3. Discuss the advantages of using variables having size equal to native word size of a microcontroller.

A. Most instructions of a microcontroller can directly process the variables having native word size; therefore, the program will be more efficient in terms of execution speed and memory requirements.

4. Which data type is widely used in the 8051? Why?

A. The unsigned char is the most widely used data type because it is 8-bit data type and native word size of the 8051 is also 8 bits.

5. List the additional data types used for 8051 programming.

A. Bit, sfr, sfr16 and sbit are additional data types.

6. What is meant by declaration *bit status=1*?

A. Assign value 1 to the bit type variable *status*.

7. What is meant by declaration sfr P2 = 0xA0;

A. SFR address A0H is assigned variable name P2.

8. What is meant by inline assembly?

A. When an assembly-language code is inserted in a high-level-language program, it is referred as an inline assembly. It allows direct hardware control.

9. State the validity of declaration *bit *addr*.

A. It is an invalid declaration because bit variables cannot be declared as pointers.

10. Discuss the limitation of the data type *sbit*.

A. The 'sbit' data type defines a bit only within special function registers (SFR).It cannot be used to define bit within bit-addressable area 20H-2FH.

11. What is the advantage of using the smallest possible data type for a variable?

A. It saves memory requirement.

12. Why do we include reg51.h file in a program?

A. The included file reg51.h contains the 'sfr' declarations of all SFRs as well as 'sbit' declarations of bit-addressable SFRs. Therefore, SFRs can be directly accessed by their names.

13. List the memory type specifiers used to access different memories in the 8051.

A. data, idata, bdata, pdata, xdata and code are the memory type specifiers.

14. What operations are automatically performed by the *using* attribute?

a) The current register bank is saved on the stack upon function entry.
b) The specified register bank is selected.
c) The original register bank is restored before the function is exited.

15. What are the disadvantages of using C or a high-level language?

A. a) Programs are not compact, i.e. machine-language program generated by compiling high level language program are larger and requires more memory.
b) Use of generalized library functions may degrade the performance of a program in terms of time (speed of execution).

16. The programmer wants to store string "ABCD" in code memory. How should this string be defined?

A. char **code** string[] = " ABCD"

EXERCISE

1. Discuss the need of using memory-type specifiers in a variable declaration/definition.
2. High-level languages are processor independent. Justify
3. What is meant by declaration *bit SWITCH = input*?
4. State validity of the declaration *bit array [5]*.
5. Show the use of sfr data type with a suitable example.
6. A variable *marks* can have value from 0 to 100. Which of the following declaration is the best? Why?
   char marks
   unsigned marks
   int marks
7. Illustrate the use of #define directive with a suitable example.
8. Illustrate the use of all memory type specifiers with a suitable example.
9. How are the interrupt service routines defined?
10. Discuss the role of *using* attribute in definition of interrupt service routines.
11. List arithmetic and logical operators supported by the Cx51 compiler.
12. Discuss the issues while writing programs for time delay generation.
13. What are the advantages of hand-coded assembly-language instructions?
14. Use of standard library routines usually degrades the code density of a program. Justify.
15. Show that use of unsigned data types will improve the efficiency of the 8051 programs.
16. When should we prefer to use inline functions?
17. Write a function which sets the bit of Port 1 specified as an argument to the function.
18. Write a function which sends binary equivalent of a number passed as an argument to the port 1.
19. Write a C program to convert ASCII string "25" into packed and unpacked BCD numbers.
20. Write a C program to convert binary number into equivalent BCD number.
21. Write a C program to convert BCD number 1234 into ASCII. Store ASCII numbers into internal RAM.
22. How can we define a lookup table in a C program?
23. Write a C program to send contents of Accumulator serially on Pin P1.0 (LSB first)
24. Write interrupt service routine for timer 0 which toggles Pin P1.0 every time when timer 0 overflows.
25. The switch is connected to $\overline{\text{INT0}}$ pin. Write an interrupt-service routine for external interrupt 0 which counts the number of times the switch is pressed and send the count on Port 1.

13

Input/Output Ports

Objectives

- Discuss the significance of I/O ports in a microcontroller
- Explain the functions of each port in the 8051
- Describe the internal structure of each port
- Configure a port as an input or output
- Discuss the difference between reading a port latch and a pin
- List the instructions that reads the port latch and pins
- Discuss the significance of read-modify feature of the 8051
- Develop the programs to handle I/O activities

Key Terms

- Alternate Function
- Bidirectional Data Bus
- Bit addressability
- D Latch
- Higher Order Address Byte
- Input Buffer
- Input/Output Port
- Lower Order Address Byte
- Output Driver
- Port Protection
- Pull-Up Resistor
- Read a Latch
- Read-Modify-Write
- Read a Pin
- Write to Latch

One of the important features of a microcontroller is the number of pins which can be used for connection with the external world. These pins are referred as Input/Output (I/O) pins and a group of such pins are referred as an *I/O port*. The I/O ports are used to transfer data in and out of a microcontroller. In the 8051, each I/O pin has built-in I/O circuits which allows a pin to directly interface with the external circuits. This feature is usually not available in the microprocessors; therefore, we need additional chips with the microprocessors to interface it with the external world.

13.1 | THE 8051 PORTS

The 8051 port structure is extremely versatile and flexible. The device has 32 I/O pins configured as four 8-bit parallel ports named P0, P1, P2 and P3. Each pin can be used as an input or as an output under the software control. These I/O pins can be accessed directly by instructions during the program execution. The I/O ports are memory mapped in the 8051, i.e. they are treated as memory locations.

Out of the 32 I/O pins, 24 pins (P0, P2 and P3) may each be used for two different functions (but only one at a time), providing a total 66 pins (40 normal pins + 24 alternate functions of ports + 2 for programming).

The function performed by a pin at any time depends on which instruction is used to access a pin and what signal is connected to that pin; therefore these factors can be directly controlled by a programmer. The alternate functions of ports are given below.

P0: Low-order address/data bus (AD7–AD0)

P2: High-order address bus (A15–A8)

P3: Each pin has different function as shown in Table 13.1.

Even within a single port, I/O operations may be combined in different ways. Different pins can be configured as an input or output independent of each other or the same pin can be used as an input or output at different times, i.e. all ports are bit-addressable. All ports of the 8051 are eight-pin wide and are bidirectional and each pin consists of a D latch, an Input Buffer and an Output Driver.

Table 13.1 *Port 3 alternate functions*

Port Pin	Alternate Function
P3.0 (Pin 10)	RXD (Serial input: Receive Data)
P3.1 (Pin 11)	TXD (serial output: Transmit Data)
P3.2 (Pin 12)	$\overline{\text{INT0}}$ (External interrupt 0)
P3.3 (Pin 13)	$\overline{\text{INT1}}$ (External interrupt 1)
P3.4 (Pin 14)	T0 (Timer/Counter 0 external input)
P3.5 (Pin 15)	T1 (Timer/Counter 1 external input)
P3.6 (Pin 16)	$\overline{\text{WR}}$ (External Data Memory write strobe)
P3.7 (Pin 17)	$\overline{\text{RD}}$ (External Data Memory read strobe)

The SFR for each port is made of these eight latches, which can be accessed by SFR address for that port; address of each port and corresponding SFR name is shown Table 13.2.

Table 13.2 *Port SFRs and their addresses*

Port	SFR Name	SFR Address	Bit addresses (MSB–LSB)
Port 0	P0	80H	87H–80H
Port 1	P1	90H	97H–90H
Port 2	P2	A0H	A7H–A0H
Port 3	P3	B0H	B7H–B0H

13.1.1 Port 1

Port 1 is a true I/O port because it has no alternate function and therefore it has the simplest structure. The basic operation of Port 1 is discussed first to develop the understanding of some basic concepts like configuring the port as an input or output, reading a port pin and writing to a port pin. The basic discussion of Port 1 is equally applicable to all other ports unless specified explicitly. The simplified structure of Port 1 is given in Figure 13.1.

As mentioned earlier, each port pin has a latch (D latch), input buffers (B1 and B2) and output driver (T1). The data from the internal data bus is written into the D latch when the 'write to latch' signal is activated as a result of appropriate instruction execution. The Q output of the latch is copied into the internal data bus when the 'read latch' signal is activated and the level of a port pin is copied into the internal data bus when the 'read pin' signal is activated. The instructions which activate these signals are discussed in the later section of this chapter.

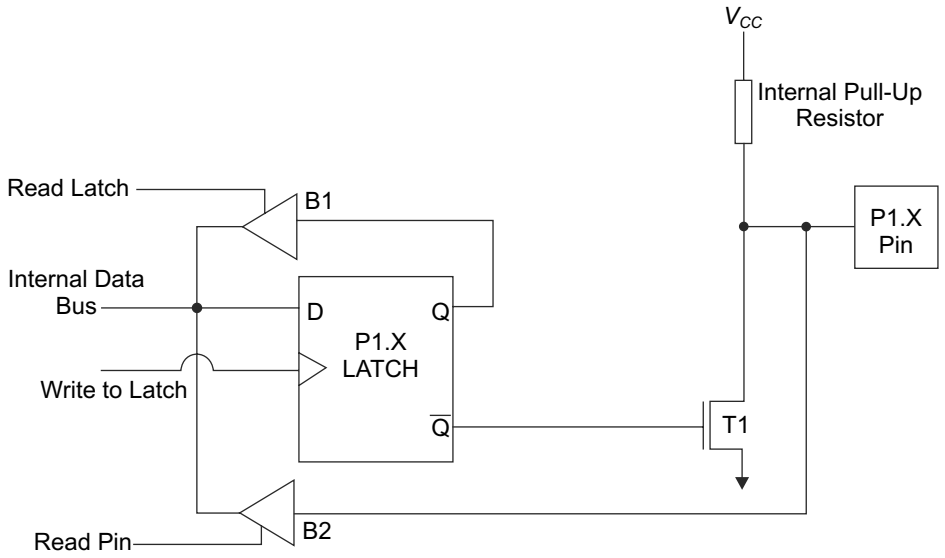


Fig. 13.1 *Port 1 structure*

1. Configuring the Port as an Input

The port pin will be configured as an input when we write '1' to the corresponding bit (latch). The reason for this is explained below. Consider Figure 13.2.

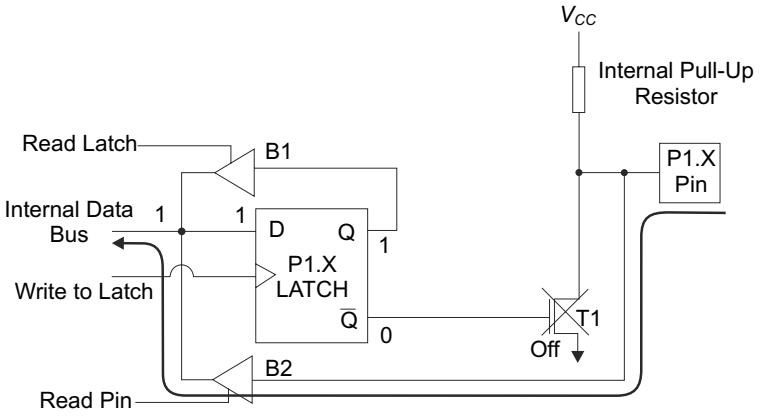


Fig. 13.2 *Reading a port pin*

When '1' is written to a port bit, it is written to the D latch; therefore, 1 will appear at output of the latch, i.e. Q = 1 and $\overline{Q} = 0$. Now $\overline{Q} = 0$ is connected to gate of the transistor (FET) T1, which will turn off T1. It will behave as an open circuit and disconnect the input pin and ground; therefore, the input signal connected to the pin will go to the buffer B2. When we read the input port using an instruction like MOV A, P1, the buffer B2 will be enabled (by 'read pin' signal) and we are actually reading a port pin. Figure 13.2 shows the path for signal flow. The signal level on the pin will be passed to the internal data bus through the buffer B2.

All the port pins are configured as an input after reset because the 8051 writes 1s to all port latches after reset. If a 0 is written by a program to the port latch, it can be reconfigured as an input by writing a '1' to it. Remember that circuits shown in Figure 13.1 and 13.2 are only for one pin. There will be 8 such circuits, one for each pin of the port.

Note*:* Ports P0, P2 and P3 have additional circuitry because they have dual functions; furthermore, the pull-up resistor is internal for ports P1, P2 and P3. We have to connect the external pull-up resistor for P0.

2. Configuring the Port as an Output

Nothing extra has to be done to configure a port as an output—whatever level (1 or 0) is written to a port latch, the same level will directly appear on the port pin. Logic '1' (high) can be written to the port pin by simply writing '1' to the port latch. As shown in Figure 13.3,when '1' is written to the D latch, it will make Q = 1 and $\overline{Q} = 0$, which will turn off T1 and, therefore, '1' will appear on the pin and the pin will source the current; similarly, logic '0' can be written by writing '0' to port latch. When '0' is written, Q = 0 and $\overline{Q} = 1$ which will turn on T1 and the pin will be connected to the ground (0 volt) as shown in Figure 13.4 and it will sink the current. During the execution of an instruction that modifies the value in a port latch, the new value is available to the latch during S6P2 of the last machine cycle of the instruction.

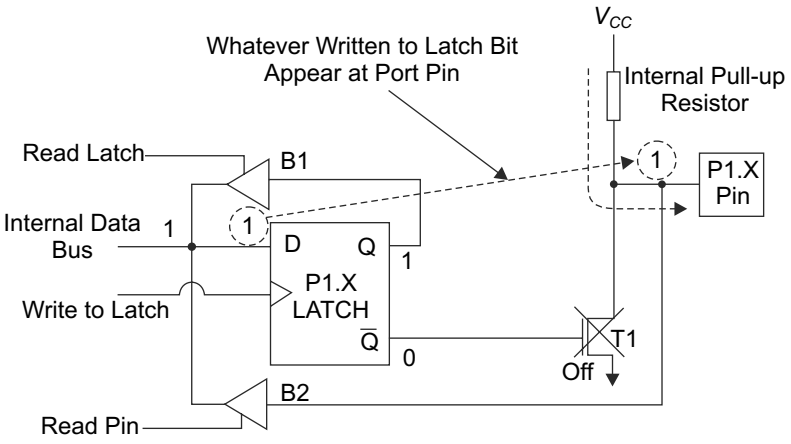


Fig. 13.3 *Writing 1 to the port pin*

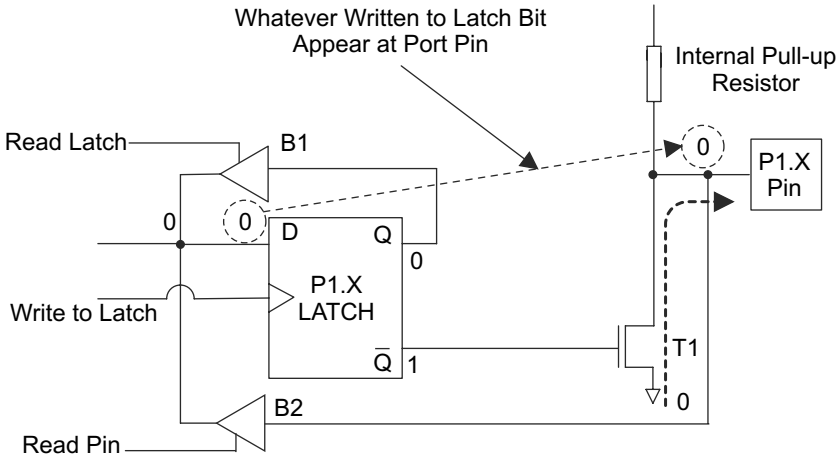


Fig. 13.4 *Writing 0 to the port pin*

Never write '0' to the port that was configured as an input port because it may damage the port when the pin is connected directly to V_{CC} (logic high input). Writing '0' will turn on T1 and it grounds the input pin. This will short the high-level signal (V_{CC}) connected to a pin, and a high current will flow through T1 and may damage it. To avoid such problems, care has to be taken while the connecting input signal to a port pin. One simple way is to use a current-limiting resistor between the input signal and the port pin as shown in Figure 13.5.

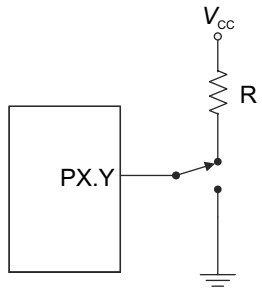


Fig. 13.5 *Port protection using current limiting resistor*

THINK BOX 13.1

What will happen if the attempt to read a port pin is made when corresponding port latch contains 0?
We will read 0, irrespective of the state of signal at the port pin.

13.1.2 Port 0

Port 0 can be used as an input/output or as a bidirectional data bus and low-order address for external memory. The structure of P0 is shown in Figure 13.6.

When P0 is used as an output, writing '0' to pin latches will make Q = 0 and $\overline{Q}$ = 1. The $\overline{Q}$ will be connected to the gate of T1 through multiplexer. (When the port is used as an input/output, internal signal 'Control' will be 0 and connects $\overline{Q}$ to the output of 2-to-1 multiplexer). The $\overline{Q}$ = 1 will turn on T1, thus grounding the port pin and logic 0 will be available to the port pin. The operation is summarized in Figure 13.7.

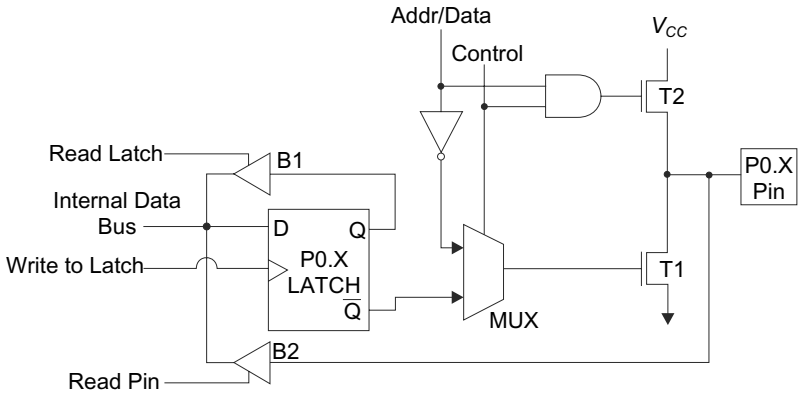


Fig. 13.6 *Port 0 structure*

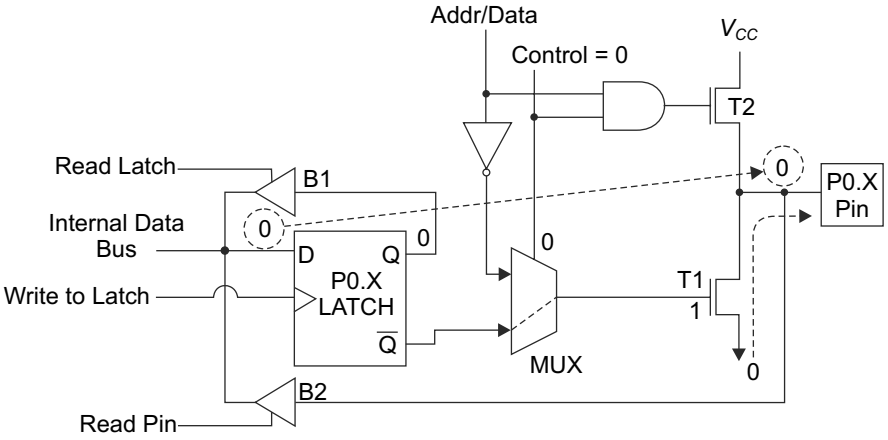


Fig. 13.7 *Writing 0 to Port 0 pin*

While writing '1' to the latches, Q = 1 and $\overline{Q}$ = 0, this will turn off the T1, which 'floats' pin to high impedance state. Therefore, an external pull-up resistor is needed to provide logic '1'. Note that T2 remains off when the pin is used as Input/Output.

Port 0 as an Address/Data Bus

When P0 is used as an address bus (function of the port is decided by instructions used to access the port), the internal 'control' signal connects the address line (indicated as ADDR/DATA in Figure 13.6) to the gate of T1 (lower FET) using the multiplexer. The logic '1'on the address bit will turn on the T2 (upper FET) since 'control' is also '1'when P0 is used as an address bus and at the same time, T1 is turned off, which in turn provides logic '1' on the port pin as shown in Figure 13.8.

When address bit is '0', T1 is turned on and T2 is turned off providing logic '0' at port pin. Once an address is issued and latched into external latch using ALE, the bus behaves as a data bus. Now if data is to be written to the external memory

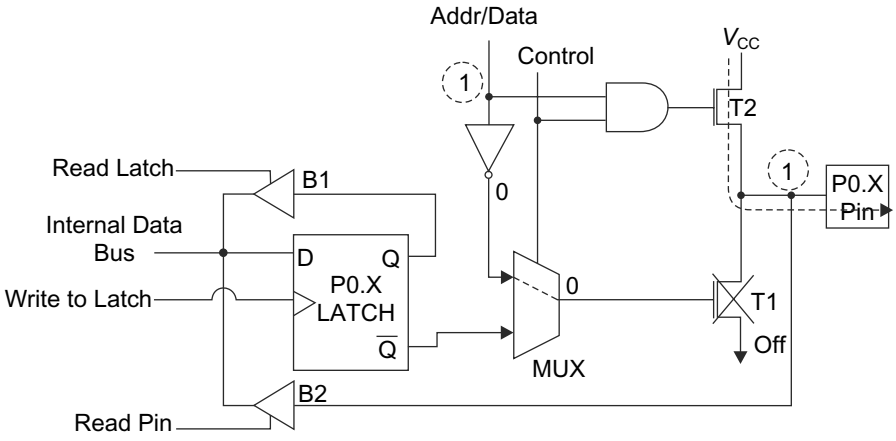
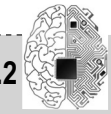


Fig. 13.8 *Port 0 as address bus, Address bit = 1*

(write to port operation), the data will be written to pins exactly in a way as the address is written to the pins as described just above. If the data is to be read from the external memory, the control logic will automatically write '1's to latches to configure port as an input ('control' = 0 and $\overline{Q} = 0$, so both T1 and T2 will be off, 'control' = 0 will connect $\overline{Q}$ of the latch to the gate of T1 through the multiplexer) and the data at the port pin will be transferred to the internal data bus through the buffer B2.

Discussion Question Why do Port 0 pins need external pull-up resistors?

Answer Port 0 needs a pull-up resistor because this port provides an open collector output. With an open collector output, logic 1 cannot be supplied to the pin, when we write 1 to latch bit, the corresponding pin will float (high impedance state). Therefore, to provide logic 1 at the port pin, we need to connect the external pull-up resistor.



THINK BOX 13.2

When P0 is used as address/data bus, how can it generate logic 1 on its pins (corresponding to 1's in address or data) even without using an external pull-up resistor?

Can P0 pins generate logic high without an external pull-up resistor when used as output? Why?

When P0 is used as address/data bus, the FET (T2 in Figure 13.8) is used as pull-up circuit. It is turned on to generate logic 1.

No. Because the FET is inactive when P0 is used as output.

13.1.3 Port 2

Port 2 can be used as Input/Output port **exactly similar to Port 1**. The other use of Port 2 is to provide high-order address byte to access external memory. The structure of Port 2 is shown in Figure 13.9.

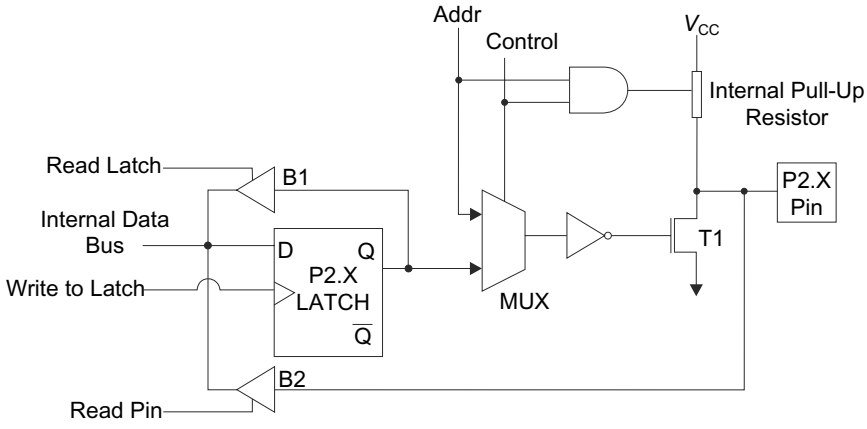


Fig. 13.9 *Port 2 structure*

When it is supplying high-order address signals (A_{15}–A_8), the internal control signals connect address line (indicated as ADDR in Figure 13.9) to the gate of T1 through the inverter. A '0' on the address bit will turn on T1 and provide '0'on the port pin. Similarly, '1' on the address bit will provide '1' on the port pin. Port 2 latches are kept stable for the duration of the entire external memory read/write cycle.

13.1.4 Port 3

Port 3 can also be used as Input/Output port similar to Port 1 and the alternate functions are controlled by various SFRs. Port 3 structure is shown in Figure 13.10.

The alternate functions can only be used if the bit latch in the corresponding port SFR contains a 1, otherwise the port pin is fixed at '0'. As shown in Figure 13.10, if the P3 bit latch contains 1 then output is controlled by the signal labeled 'alternate output function'. The actual P3.X pin level is always available to the pin's alternate input function.

Discussion Question What are the alternate functions of Port 0, 1, 2 and 3?

Answer Port 0 also functions as the lower order 8 bits of the multiplexed address/data bus for external memory access. There is no alternate function for Port 1, for Port 2 also acts as higher-order 8 bits of the address bus for external memory access. Port 3 pins have individual alternate functions. The pins on this port function as external interrupt inputs, serial data input and output, timer/counter inputs and control signals for external memory access (Refer Table 13.1).

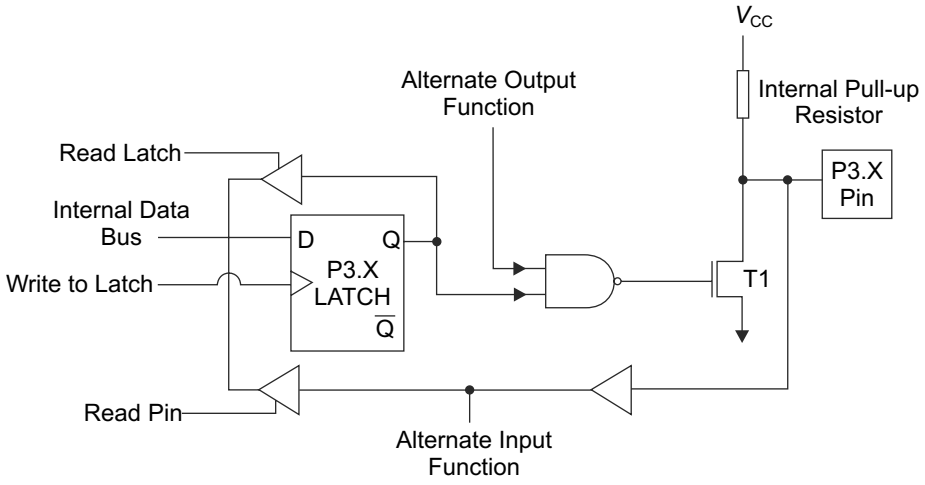
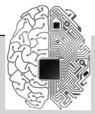


Fig. 13.10 *Port 3 structure*



THINK BOX 13.3

Why P1, P2 and P3 are known as quasi-bidirectional ports while P0 is referred as true bidirectional port?

Since P1, P2 and P3 have fixed internal pull-ups, when used as an input, they will source the current when externally pulled low. P0 pins when used as input, it floats.

Example 13.1

Write a program to continuously toggle the contents of Port 0 and Port 1 after some delay.

Solution:

```
        MOV A, #00H          // Initialize A with 0
BACK:   MOV P0, A            // send the contents of A to P0 and P1
        MOV P1, A
        ACALL DELAY          // delay
        CPL A                // complement (toggle) the contents of A
        MOV P0, A            // send the toggled contents to P0 and P1
```

```
        MOV P1, A
        SJMP BACK          // repeat the process forever
DELAY:  MOV R2, #0FFH      // delay subroutine
HERE:   DJNZ R2, HERE
        RET
```

Example 13.2

Write the program instructions to read contents of Port 0 and send it to Port 1.

Solution:

```
        MOV P0, #0FFH      // configure Port 0 as an input by writing 1s to all bits
        MOV A, P0          // read the contents of P0 into accumulator
        MOV P1, A          // send contents of P0 to P1
```

Example 13.3

Write instructions
(i) To configure pins P1.0, P1.3 and P1.4 as inputs and remaining pins of port 1 as outputs
(ii) To configure P2 as an input
(iii) To output (send) 'ABH' on Port 3

Solution:

To configure a port pin as an input, we need to write 1 to the corresponding port latch. To configure a port pin as an output, nothing special has to be done, i.e. whatever data is written to (output to) port latch is available directly on port pin.

```
(i)     MOV P1, # 19H      // set bits P1.0, P1.3 and P1.4
```

OR the same thing can be done by following bit-level instructions

```
        SETB P1.0          // set the corresponding latch bits to configure pins as an input
        SETB P1.3
        SETB P1.4
(ii)    MOV P2,#0FFH       // configure P2 as an input port
(iii)   MOV P3,#0ABH       // output ABH on Port 3
```

Example 13.4

Write a program to continuously read contents of Port 1; complement it and send it to Port 2.

Solution:

```
        MOV P1, #0FFH      // configure P1 as input
REPEAT: MOV A, P1          // read P1
        CLP A              // complement
        MOV P2, A          // send the complemented contents to P2
        SJMP REPEAT        // repeat the operation continuously
```

The equivalent C program is as given below:

```
#include<reg51.h>
void main ()
{
P1 = 0xFF;          // configure P1 as input
while (1)           // repeat the operation continuously
      P2 = ~P1;     // read P1, complement and send to P2
}
```

Example 13.5

Assume that eight LEDs are connected to Port 1 (one LED to each pin). Write a program to sequentially glow all LEDs one by one (P1.0 to P1.7) with delay between each.

Solution:

Assuming that the LEDs are driven by port pins, i.e. anodes are connected to port pins and cathodes are grounded. We need to write 1 to port pin to glow particular LED.

First, the LED connected to P1.0 should glow, then LED of P1.1, so on and finally LED of P1.7 should glow. This will create a pattern of scrolling LED (glowing) in one direction.

It is assumed that 1-second delay is available (Refer Example 8.3 for the delay of 1s).

```
        ORG 0000H
        MOV A, #01H         // only one bit is 1, i.e. only 1 LED is on at a time
BACK:   MOV P1, A           // send to Port 1
        CALL DELAY          // delay between each glowing LED
        RLA                 // rotate to glow next LED
        SJMP BACK           // repeat the pattern forever
        END
```

Example 13.6

Rewrite a program of Example 13.5 in C using at least two different methods.

Solution:

First method: Eight numbers that should be sent to Port 2 to glow each LED in desired sequence are stored in an array. The array elements are accessed one by one and sent to Port 2.

```
#include<reg51.h>

void main (void)
        {
        unsigned char j, array[] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
        unsigned int i;
                                // array of numbers to glow one LED at a time in a desired sequence
        while (1)               // send sequence continuously
                {
                for ( j=0; j<8; j++)
                        {
                        P2 = array[j];                  // glow one LED at a time
                        for (i=0; i<60000; i++);        // delay
                        }
                }
        }
```

Second method: The temporary variable (initialized with 01H) is rotated in the left direction by one bit on each iteration and sent to Port 2. Refer Section12.7 (Rotate operations in C) for more details on rotation.

```
#include<reg51.h>

unsigned char leftrotate(unsigned char, unsigned char);
void main ()
        {
        unsigned char j,k;
        unsigned int i;
        k = 0x01;                               // initialize temp. variable k with 01
        while (1)                               // send sequence continuously
                {
                for (j=0; j<8; j++)
                        {
                        P2 = leftrotate(k, j);          // rotate number by one bit left in each iteration
                        for ( i=0; i<60000; i++);       // delay
```

```
                }
            }
        }

unsigned char leftrotate(unsigned char n, unsigned char d)
        {
        unsigned char j;
        j= (n << d)| (n>>(8-d));                    // rotate left n by d bits
        return j;
        }
```

Example13.7

Assume that eight LEDs are connected to Port 2. Discuss the different methods to alternately glow all even- and odd-numbered LEDs continuously with delay in between.

Solution:

To glow even-numbered LEDs, the number 'AAH=10101010B' should be sent to P2 and to glow odd numbered LEDs, the number '55H' should be sent to Port 2. Different methods to achieve the required objective is given below.

Assuming that the desired delay routine is available:

(i)

```
BACK:   MOV A, #55H         // number for which odd bits are 1
                            // (D0 being LSB)
        MOV P2, A           // send to Port 2 to glow odd numbered LEDs
        ACALL DELAY         // delay
        MOV A, #0AAH        // number for which even bits are 1
        MOV P2, A           // send to Port 2 to glow even numbered LEDs
        ACALL DELAY         // delay
        SJMP BACK           // repeat continuously
```

(ii)

```
        MOV A, #55H         // number for which odd bits are 1
BACK:   MOV P2, A           // send to Port 2 to glow LEDs
        ACALL DELAY         // delay
        CLP A               // change from 55 to AA and vice versa
        SJMP BACK           // repeat continuously
```

(iii)

```
        MOV A, #55H         // number for which odd bits are 1
BACK:   MOV P2, A           // send to Port 2 to glow LEDs
        ACALL DELAY         // delay
        RRA (or RLA)        // change from 55 to AA and vice versa
        SJMP BACK           // repeat continuously
```

(iv)

```
        MOV A, #55H         // number for which odd bits are 1
BACK:   MOV P2, A           // send to Port 2 to glow LEDs
        ACALL DELAY         // delay
        XRL A,#0FFH         // change from 55 to AA and vice versa
        SJMP BACK           // repeat continuously
```

Example13.8

Rewrite a program of Example 13.7 in C.

Solution:

```
#include<reg51.h>
void main ()
    {
    unsigned int i;
    P2 = 0x55;                          // initialize P2 with 0x55
    while (1)                           // repeat operation continuously
        {
        P2 = ~P2;
        for ( i=0; i<60000; i++);       // delay
        }
    }
```

13.2 READING LATCH VERSUS READING PORT PIN

The instructions that read the latch will read a state of latch, perform arithmetic or logical operation on it and finally write it back to the latch. These are called *Read-Modify-Write instructions*. The advantage of these instructions is that the three operations are performed in a single instruction thus helping in making programs compact and readable. These instructions are listed in Table 13.3.

Table 13.3 *Instructions reading a latch*

Instructions
ANL PX , A
ORL PX , A
XRL PX , A
JBC PX.Y, rel
CPL PX.Y
INC PX
DEC PX
DJNZ PX, rel
MOV PX.Y, C
CLR PX.Y
SETB PX.Y

It can be observed from Table 13.3 that when the destination operand is either a port or a port bit, the instruction reads a latch.

The Read-Modify-Write instructions read the latch (instead of the pin) to prevent a probable misinterpretation of the voltage level of the pin. For example, a port pin may be connected to the base of a transistor as shown in Figure 13.11.

When '1' is written to the port latch, the transistor will be turned on, resulting in the base voltage of 0.7 volts. If the microcontroller reads the port pin in this case, it will read the base voltage of the transistor and interpret it as logic 0 (0.7 is treated as logic level '0') even though port latch contains '1', therefore, reading the latch rather than the pin will read correct value of '1'. The same problem will occur whenever the port pin is heavily loaded.

The instruction which only reads the port, but does not perform any (arithmetic or logical) operation to modify the contents of the port reads the level of a port pin. In general, the instructions that read status of the port pin are one in which port (or port bit) is a source only. These instructions are listed in Table 13.4.

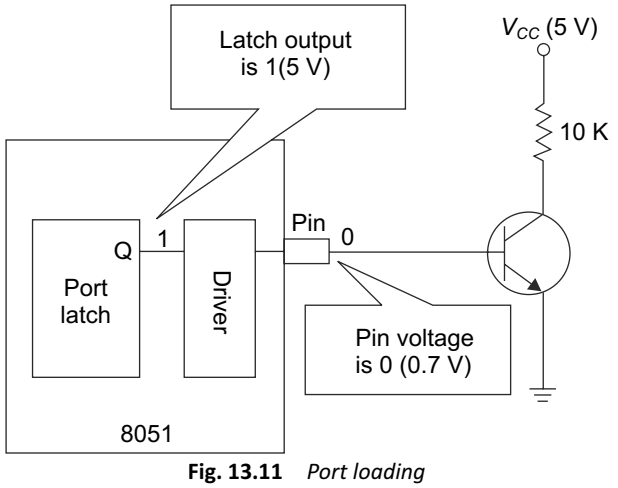


Fig. 13.11 *Port loading*

Table 13.4 *Instructions that reads port pins*

Instructions
MOV A,PX
MOV C,PX.Y
JNB PX.Y, rel
JB PX.Y, rel
CJNE A, PX, rel

Logical Operation with Ports

Logical operations with ports are discussed in detail in Section 5.2.1.

THINK BOX 13.4

What is the direction of Port 0 assumed by an instruction 'JNB P2.0, THERE'?
Input. Because this instruction monitors the status of the pin P2.0.

Example 13.9

Demonstrate how to access individual port pins. Perform the following functions.
(i) Read port pin P1.2 and send same on P2.5 continuously.
(ii) Toggle P2.2 continuously after some delay.

Solution:

```
(i)             ORG 0000H
                SETB P1.2           // configure P1.2 as an input
REPEAT:         MOV C, P1.2         // read P1.2 into carry
                MOV P2.5, C         // send carry to P2.5
                SJMP REPEAT         // repeat continuously
                END
(ii) Assume that delay routine is available.
                ORG 0000H
                CLR C               // initially clear C to send 0 to P2.2
TOGGLE:         MOV P2.2, C         // send contents of C to P2.2
                CPL C               // toggle
                ACALL DELAY         // delay
                SJMP TOGGLE         // repeat continuously
                END
```

Refer Example 12.13 for the equivalent C program.

Example 13.10

Write a program to monitor bit P1.7. If it is high, write FFH to P0; otherwise, write 00H to P0.

Solution:

```
                ORG 0000H
                SETB P1.7           // configure P1.7 as an input
REPEAT:         MOV C, P1.7         // read P1.7 in to C
                JC SET              // if C = 1, jump to send FFH to P0
                MOV P0, #00H        // otherwise (if C=0), send 00H to P0
                SJMP  SKIP          // skip following instruction.
SET:            MOV P0, #0FFH       // send FFH to P0
SKIP:           SJMP REPEAT         // continuously monitor P1.7
                END
```

Refer Example 12.16 for the equivalent C program.

For other programs of port programming, refer Chapter 12, since a majority of the programs of that chapter are about the same.

13.3 PORT CURRENT CAPABILITIES

Port 0 pins as an output can drive up to 8 LS TTL inputs. When it is used as an address/data bus, it uses internal pull-up FET when providing 1's and can source and sink 8 LS TTL inputs.

Port 0 pins can sink maximum 3.2 mA current (I_{OL}). I_{OH} (V_{OH} = 2.4 V) for port 0 is 400 μA

Port 1, 2 and 3 pins can drive (source or sink) up to 4 LS TTL inputs. Port 1, 2 and 3 pins can sink maximum 1.6 mA current (I_{OL}). Port 1, 2 and 3 pins that are externally pulled low will source 500 μA current (I_{IL}) because of the internal pull-ups.

Refer datasheet for a particular 8051 variant for more details.

POINTS TO REMEMBER

- The I/O ports are memory mapped in the 8051, i.e. they are treated as memory locations.
- Port 0 can be used as an input/output and as a bidirectional data bus and low-order address for external memory.
- Port 1 is a true I/O port because it has no alternate function.
- Port 2 can be used as Input/Output port and as a high-order address byte to access external memory.
- Port 3 can be used as Input/Output port or each pin has alternate function.
- The function that a pin performs at a given time depends on what instructions are used to program the pin and what signal is connected to that pin.
- Different pins of the same port can be configured as an input or output, independent of each other or the same pin can be used as an input or output at different times, i.e. all ports are bit addressable.
- Each port pin has a D latch, an input buffer and an output driver.
- The port pin will be configured as an input when we write a '1' to corresponding latch.
- All the port pins are configured as an input after reset.
- When Port 0 is used for I/O operations, external pull-up resistor is needed to provide the logic '1'.
- The advantage of the Read-Modify-Write instruction is that three operations are performed in a single instruction, thus helping in making programs compact and readable.
- When the destination operand, is either a port or a port bit, the instruction reads the latch.
- The instructions which only reads the port, but do not modify the contents of the port usually reads the level of port pin.

OBJECTIVE QUESTIONS

1. The _____does not have alternate function.
   (a) Port 0 (b) Port 1 (c) Port 2 (d) Port 3
2. The _____does not have an internal pull-up resistor.
   (a) Port 0 (b) Port 1 (c) Port 2 (d) Port 3
3. Which pin of Port 3 has an alternative function as $\overline{WR}$ signal for the external data memory?
   (a) P3.7 (b) P3.3 (c) P3.6 (d) P3.1
4. Which pin of Port 3 has an alternative function as $\overline{RD}$ signal for the external data memory?
   (a) P3.0 (b) P3.7 (c) P3.6 (d) P3.1
5. After power on reset, all port latches contain,
   (a) FFH (b) 00H (c) XX (d) FFH or 00H based on the 8051 variant
6. Which of the following instructions read the port pin?
   (a) ANL PX, A (b) INC PX (c) DJNZ PX, rel (d) JNB PX.Y, rel

7. Which of the following instructions read the port latch?
   (a) ANL PX, A (b) INC PX (c) JB PX.Y, rel (d) JNB PX.Y, rel
8. The _____ is used as data bus.
   (a) Port 0 (b) Port 1 (c) Port 2 (d) Port 3
9. The _____ is used as high-order address bus.
   (a) Port 0 (b) Port 1 (c) Port 2 (d) Port 3
10. The following instructions will read data from Port 1 and write it to Port 2, and it will stop looping when Bit 3 of Port 2 is 0,

```
BACK: MOV A, P1
      MOV P2, A
      JB P2.3, BACK
```

   (a) True (b) False
11. The following program will receive data from Port 1, determine whether bit P1.0 is high, and then send the number AAH to Port 3,

```
BACK: MOV A, P1
      ANL A, #01H
      CJNE A, #01H, BACK
      MOV P3, #0AAH
```

   (a) True (b) False
12. The alternate function of Pin 3.4 is,
   (a) timer/counter 0 external input (c) serial input port
   (b) timer/counter 1 external input (d) serial output port

Answers to Objective Questions

1. (b)	2. (a)	3. (c)	4. (b)	5. (a)	6. (d)
7. (a), (b)	8. (a)	9. (c)	10. (a)	11. (a)	12. (a)

REVIEW QUESTIONS WITH ANSWERS

1. What is the default direction of all ports after reset?

A. All ports are configured as input ports after reset. FFH is written into all the port latches.

2. Justify True/False with a reason. "All port pins can be programmed independently for input or output"

A. True. Because all port latches (SFRs) are bit addressable.

3. When using Port 0 as I/O, we need to connect pull-up resistors to port pins. Why?

A. Pins of Port 0 are open drain. Pull-up resistors are used to produce logic 1 when used as output.

4. How many port pins of the 8051 have dual functions?

A. All pins of P0, P2 and P3 have dual functions. Thus, 24 pins of the 8051 have dual functions.

5. How is the function of a port pin selected?

A. It depends on instructions used to access the port pins.

6. What circuits are connected to all port pins?

A. Latch, input buffer and output buffer.

7. How many ports are available to perform the I/O function when external memory is connected?

A. Two ports, P1 and P3.

8. What are the alternate functions of Port 0?

A. P0 is used as multiplexed lower order address and data bus or it can be used as I/O activities.

9. What is the use of ALE pin?

A. It is the Address Latch Enable signal used to de-multiplex lower order address bus and data bus.

10. Why is there no internal pull-up resistor in Port 0 pins?

A. To allow it to multiplex address and data.

11. True or false. Instruction " JB P1.0, NEXT" reads input pin".

A. True.

12. Write two instructions to set P1.0.

A. SETB P1.0 and ORL 80H, #01H.

13. What is the direction of port pin assumed by instruction " JNB P1.1, NEXT".

A. Input.

14. Which port of the 8051 does not have any alternate function?

A. P1.

15. Even when port pin is set to 1, there is much lower voltage than 5 V on it, what may be the cause?

A. The port may be loaded heavily by load like LED.

16. What is the alternate function of port 0?

A. AD0-AD7.

EXERCISE

1. Why internal pull-up (weak pull-up) resistor of P0 cannot be used as pull-up resistor for I/O activities?
2. Why is the pull-up resistor required for P0, when used as an address bus?
3. Why are all the ports except P0 known as quasi bidirectional ports?
4. How is P0 referred as true bidirectional port?
5. How to configure port as an (a) input, and (b) output?
6. What is current sinking and sourcing capacity of each port?
7. Why do we use bit pattern 10101010 (AAH) or 01010101(55H) to test the ports?
8. Discuss the alternate functions of Port 3 pins.
9. How is a lower-order address and data de-multiplexed?
10. While accessing external memory, how is P0 configured as an input while reading data from memory?
11. Explain how logic 0 and 1 are written to ports.
12. What are the reasons for damaging a port? Discuss the different methods to avoid this damage.
13. Explain in detail the Read-Modify-Write feature.
14. While reading a port, some instructions read the port pin while the other reads the port latch. Why?
15. Find the minimum value of a pull-up resistor that can be used for P0.
16. List the instructions which read a port latch.
17. List the instructions which read a port pin.
18. Why is it preferred to use logical OR operation while changing more than one bit at a time?
19. What care has to be taken while moving an immediate value into port latches?
20. How is the port structure of 8051 most suitable for the control applications?
21. Write a program to toggle P1.0 continuously using at least two different methods. [*Hint:* use AA, or CPL instruction]
22. The CPL instruction reads a port latch rather than a port pin. Justify.

14

Timers

Objectives

- Discuss the need and uses of timers in a microcontroller-based system
- Discuss time-delay generation techniques
- Explain how a timer works
- Describe the operating modes of timers of the 8051 and associated registers
- Describe the operation and to configure a timer as an interval timer
- Develop the programs to generate time delay using an interval timer
- Describe the operation and to configure a timer as an event counter
- Develop the programs to count external events

Key Terms

- Auto Reload
- Clock Source
- Event Counter
- Frequency Measurement
- Gate
- Hardware Control
- Initial Count
- Interval Timer
- Overhead
- Pulse-Width Measurement
- Software Control
- TCON
- Timer 0
- Timer 1
- Timer Modes
- Timer Overflow: TF1,TF0
- Timer Registers: TH0,TL0
- Timer Registers: TH1,TL1
- Timer Run:TR1,TR0
- Time Delay
- TMOD

14.1 NEED OF TIMERS

Many timing applications require the generation of accurate time delay between certain events or counting of events happening outside the microcontroller such as counting number of pulses. The subroutine that generates a time delay is commonly required in the programs. The time delay may be generated using software loops that effectively do nothing for a specified time period. Time-delay generation using software is explained in Chapter 7 (Section 7.5). The approach of generating a time delay using software keeps the microcontroller busy and due to that other important tasks may remain unattended and are not executed. To relieve the microcontroller from this burden of generating time delays, the developers of the 8051 have incorporated programmable hardware timer circuits to look after timing and counting activities. Time delays can be generated using hardware timers that count internal clock pulses or they may be programmed as counters which count external events. The basic functions of timers are

1. To measure the time—calculating time elapsed between events. This capability of a timer is deployed in generating time delays as well as waveforms.
2. To count events or to measure frequency or pulse width of an unknown signal.
3. To provide the clock signal to other circuits.

14.2 HOW DOES A TIMER OPERATE?

The microcontroller uses a crystal oscillator to generate clock pulses for its operations. The clock pulses generated by a crystal oscillator are always of fixed and known time period. Counting these clock pulses is as good as measuring the elapsed time period between events. The timer registers are incremented by clock pulses applied to it. When a timer register reaches its maximum value, it overflows on the next clock pulse and indicates this overflow status by setting a flag (referred as timer overflow flag) in its status register or raising an interrupt. This setting of the flag (or interrupt) is the mechanism used by a timer circuit to inform the microcontroller that it has finished its work completely (or partially in some cases), and a necessary action should be taken by the microcontroller as per requirement.

14.3 TIMERS IN THE 8051

The 8051 has two 16-bit timers: Timer 0 and Timer 1, which can be programmed independently. Each timer can be configured either as an interval timer or as an event counter in various operating modes. Timer 0 has two 8-bit registers, namely TL0 (lower byte) and TH0 (higher byte), collectively used as a 16-bit register. Similarly, Timer 1 has TL1 and TH1 registers. These registers can be accessed in the same way as any general-purpose register. The symbols TLX and THX are used at many places in this text to represent timer registers of both the timers. Timer 0 and 1 registers are shown in Figure 14.1.

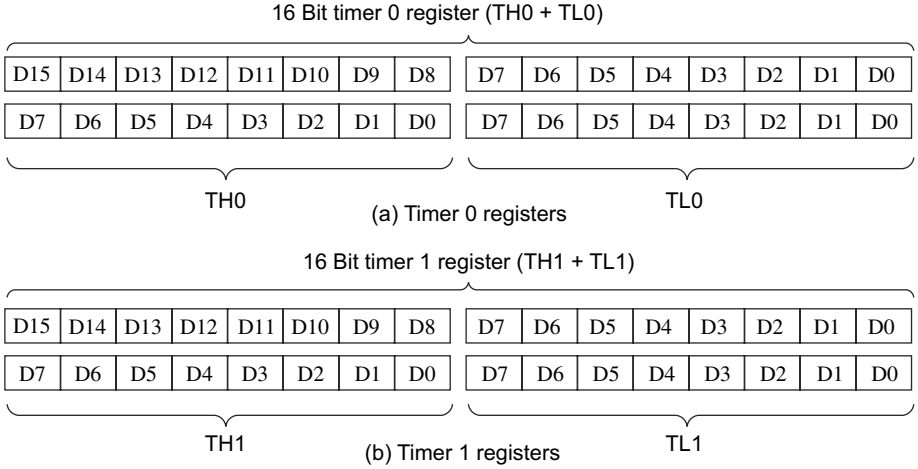


Fig. 14.1 *Timer registers*

The Timer Mode Control (TMOD) register is used to configure both timers into various operating modes and the Timer control (TCON) register is used to control start/stop operations and also has the overflow status flags of both timers in it. The operation of both timers is exactly similar; therefore, any discussion for one is applicable equally to the other, unless specified. The description of both SFRs is given below.

14.3.1 TMOD (Timer Mode Control) Register

TMOD is an 8-bit special function register dedicated to both timers. The lower 4 bits configure Timer 0 and the upper 4 bits configure Timer 1. The bit assignment of the TMOD is shown in Table 14.1.

Table 14.1 *TMOD register*

Timer 1				Timer 0			
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
MSB							LSB

Bit	Symbol	Description		
7/3	Gate	Start/stop control using hardware or software. When Gate = 0, start/stop of timer is controlled only by TR1/0 bits; while Gate = 1, it is controlled by TR 1/0 as well as signal on $\overline{INT1/0}$ pin		
6/2	C/$\overline{T}$	C/$\overline{T}$ = 0 configures the timer as a interval timer (or time-delay generator), C/$\overline{T}$ = 1 will configure the timer as an event counter		
5/1	M1	Mode select bit 1		
4/0	M0	Mode select bit 0		
		M1	M0	
		0	0	Mode 0;13-bit timer
		0	1	Mode 1; 16-bit timer/counter
		1	0	Mode 2; 8-bit auto reload
		1	1	Mode 3; split timer mode, TL0 as 8-bit timer/counter and TH0 as 8-bit timer controlled by control bits of Timer 0 and Timer 1 respectively. Timer 1 operation timer/counter stopped.

1. Gate

The start/stop operation of the timers can be controlled either by software or hardware. In software-controlled operation, start/stop is achieved by timer run bits TR0 and TR1 for Timer 0 and Timer 1 respectively. Timer run bits are located in the TCON register. The instructions "SETB TR0" and "CLR TR0" are used to start and stop Timer 0 respectively. Similarly, "SETB TR1" and "CLR TR1" instructions are used to start and stop Timer 1 respectively. How the Gate bit is used to select either software- or hardware-controlled operation of timers is discussed in the next section.

Software Control of Timers The software control of the timers is illustrated in Figure 14.2 (a). The software control (start/stop of timers) is achieved when the Gate bit is programmed to zero, i.e. Gate = 0. When Gate = 0, the output of the NOT gate is 1, which is applied to the OR gate. Since one of the inputs of the OR gate is 1, its output will be 1 irrespective of the status of the other inputs. The output of the OR gate (= 1) is given to the AND gate. Now one input of the AND gate is 1, therefore output of AND gate is controlled by other input. The other input is TR0/1 bit, therefore output of AND gate is controlled by status of TR0/1 bit, i.e. If TR0/1 = 0, output of AND gate is 0 and if TR0/1 = 1, output of the AND gate is 1. The output of AND gate is used to control the switch which will connect clock signal to the timer stages for their operation. When output of AND gate is 1, the switch is closed and the clock signal is applied to the timer stage and the timer starts its operation. Therefore, effectively we can say that TR0/1 bits only controls the timer operation. See dotted line in Figure 14.2(a). As mentioned earlier, the TR0/1 bit can be set or cleared using SETB (SETB TR0 or SETB TR1) and CLR (CLR TR0 or CLR TR1) instructions.

Hardware Control of Timers The hardware control of timers is illustrated in Figure 14.2 (b). The hardware control of timers is achieved when Gate bit is programmed to one, (i.e.) Gate = 1 and TR0/1 = 1. When Gate = 1, the output of NOT gate is 0, which is applied to OR gate, since one of the input of OR gate is 0, the output is controlled by the second input. The second input to OR gate is given from $\overline{INT0}$ and $\overline{INT1}$ pins for Timer 0 and Timer 1 respectively. When $\overline{INT0/1}$ = 0, the output of OR gate is 0 and when $\overline{INT0/1}$ = 1, the output of OR gate is 1. Since TR0/1 is already programmed to be 1,

the status of $\overline{\text{INT0/1}}$ pin will determine the output of AND gate and finally operation of the timer because output of AND gate is used to connect the clock source to the timers. See dotted line in Figure 14.2(b)

This feature allows the pulse width measurement, i.e. a pulse applied to $\overline{\text{INTx}}$ pin will run the timer as long as it is high, then, the timer register may be read to determine time period of the pulse. It should be noted that the external hardware control is achieved by making corresponding timer runs bits TR 0/1 = 1 as shown in Figure 14.2 (b).

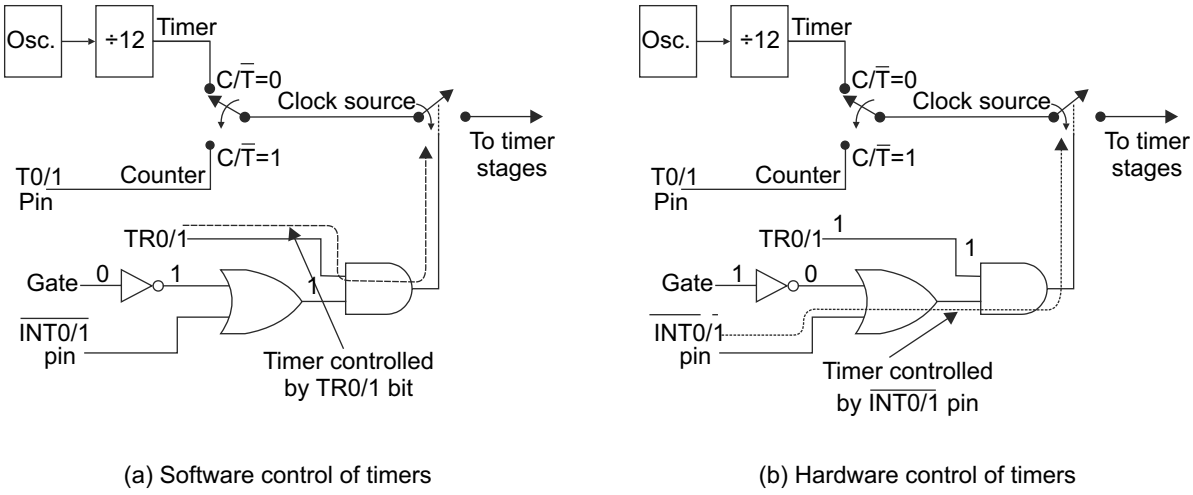


Fig. 14.2 *Software/hardware control of timers*

2. C/$\overline{T}$

The C/ $\overline{\text{T}}$ bit in the TMOD register is used to configure the timer as either interval timer (C/ $\overline{\text{T}}$ = 0) or event counter (C/ $\overline{\text{T}}$ = 1). The interval timer means calculating the time elapsed between events or generating a delay. Event counter means to count the pulses that are generated by external events. The only difference between interval timer and event counter is the clock source used by the timer circuit. All other operations are exactly same for both configurations.

Clock Source for the Timer The selection of the clock source using the C/ $\overline{\text{T}}$ bit is illustrated in Figure 14.3. When C/ $\overline{\text{T}}$ = 0, the selection switch (S in Figure 14.3) will be connected to the point *A*. The point *A* provides internal clock signal generated by a crystal oscillator, therefore, internal clock is used as source of the clock to the timer stage. The internal clock signal is used as a clock source for the interval timer activities; therefore it is also referred as *timer clock*.

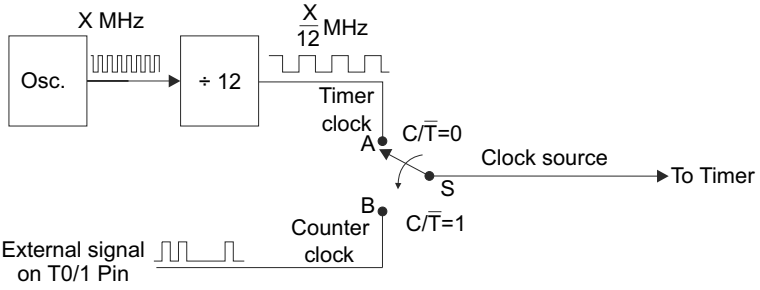


Fig. 14.3 *Clock source for timer stages*

Note that the oscillator signal is always divided by 12 internally and the resultant signal is given to the timer stages, i.e. timer registers are incremented every machine cycle as shown in Figure 14.3.

For a timer clock to reach timer stages, the C/ $\overline{\text{T}}$ bit must be 0 (interval timer operation), bit TRX (TR0 or TR1) must be set to 1(timer run) and the gate bit in TMOD register must be 0 or $\overline{\text{INTx}}$ pin must be 1.We may conclude that the when timer is configured as an interval timer, the timer clock pulses are fed to timer stages when C/ $\overline{\text{T}}$ = 0, timer run bit TRX = 1 AND Gate = 0 or external input pins $\overline{\text{INTx}}$ = 1. The path for clock signal when C/ $\overline{\text{T}}$ = 0 is shown by a dotted line, Figure 14.4 (a).

When C/ $\overline{\text{T}}$ = 1, the selection switch S will be connected to the point *B*. The point *B* is connected to timer input pins T0/1. The clock source for the timer circuits is external pulses on pin T0 (Pin 14) or T1 (Pin 15) of the microcontroller for Timer 0 and Timer 1 respectively. The path for clock signal when C/ $\overline{\text{T}}$ = 1 is shown by the dotted line in Figure 14.4 (b). The other control bits for timer start/stop will remain same as described in the above section.

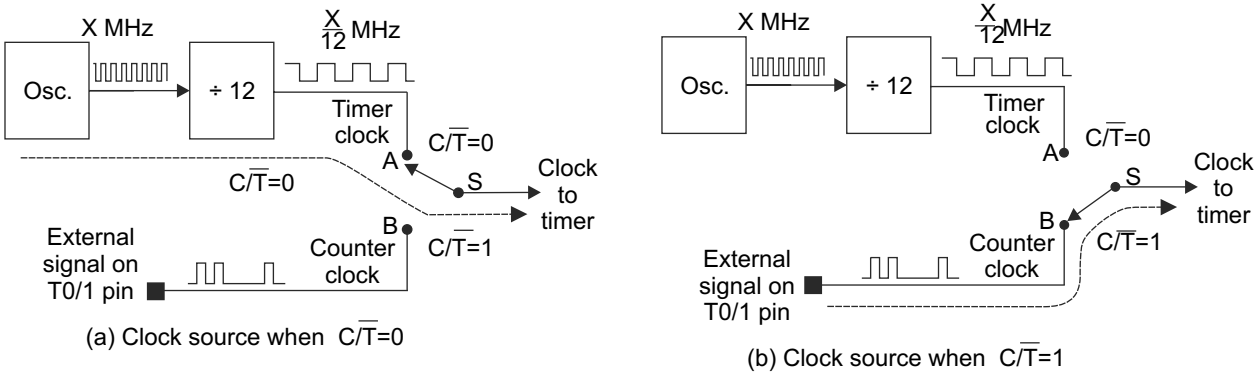


Fig. 14.4 *Clock source selection using C/$\overline{T}$ bit*

3. M1 and M0

M0 and M1 bits select the timer mode. Timers in the 8051 can be operated in the four different modes, (i.e.) Mode 0, 1, 2 or 3. Each mode is discussed in detail in the next section.

Example 14.1

Write instructions to configure,

(i) Timer 1 as an interval timer in Mode 2 and Timer 0 as an interval timer in Mode 0. Start/stop operations of both timers controlled by software.

(ii) Timer 1 as an event counter Mode 1 and Timer 0 as an interval Timer Mode 1. Start/stop operations of both timers controlled by software.

(iii) Timer 1 as an interval timer in Mode 1, Start/Stop operations controlled by software and Timer 0 as an interval timer in Mode 0. Start/stop operations controlled by hardware ($\overline{INT0}$ pin).

Solution:

The format of TMOD is given below:

Timer 1				Timer 0			
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0

(i) The bits of TMOD register are programmed as discussed below:

The start/stop operation should be controlled by software; therefore, both GATE bits (D7 and D3) are programmed as 0. To configure Timer 1 and Timer 0 as the interval timer, both C/$\overline{T}$ bits (bit D6 and D2) should be 0. For Timer 1 in Mode 2, bits M1M0 = 10 and for Timer 2 in Mode 0, M1M0 = 00. Therefore, TMOD register should be programmed as shown below:

Timer 1				Timer 0			
0	0	1	0	0	0	0	0

This binary pattern represents 20H; therefore, the instruction is,

```
MOV TMOD, #20H
```

(ii) For the given requirement, the TMOD register is programmed as explained below.

The start/stop operation should be controlled by software; therefore, both GATE bits (D7 and D3) are programmed as 0. To configure Timer 1 as a counter C/$\overline{T}$ = 1 and Timer 0 as interval timer C/$\overline{T}$ bit (D2) should be 0. For Timer 1 in Mode 1, bits M1M0 = 01 and for Timer 0 in Mode 1, M1M0 = 01. Therefore, TMOD register should be programmed as shown below:

Timer 1				Timer 0			
0	1	0	1	0	0	0	1

Above binary pattern represents 51H, therefore, the instruction is,

```
MOV TMOD, #51H
```

(iii) Similarly, for given requirements, the TMOD register is programmed as shown below:

Timer 1				Timer 0			
0	0	0	1	1	0	0	0

It represents 18H, and the instruction is,

MOV TMOD, #18H

Example 14.2

What will be the frequency of the timer clock if the crystal oscillator frequency is (i) 12 MHz. and (ii) 11.0592 MHz

Solution:

We know that crystal frequency is divided internally by 12 to generate the timer clock; therefore, the timer clock will be $1/12^{th}$ of the crystal frequency as illustrated in Figure 14.5.

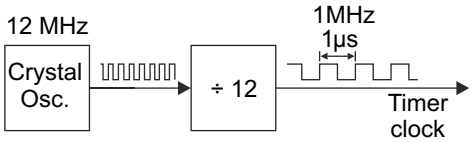


Fig. 14.5 *Timer clock*

(i) For oscillator frequency = 12 MHz

$$\text{Timer clock frequency} = \frac{\text{Crystal frequency}}{12} = \frac{12\text{ MHz}}{12} = 1\text{MHz or time period is 1µs.}$$

(ii) For oscillator frequency = 11.0592 MHz

$$\text{Timer clock frequency} = \frac{\text{Crystal frequency}}{12} = \frac{11.0592\text{ MHz}}{12} = 921.6\text{ KHz or time period is 1.085 µs.}$$

14.3.2 TCON Register

Bit assignment of TCON register is shown and explained in Table 14.2.

Table 14.2 *TCON register*

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
MSB							LSB

Bit	Symbol	Description
7 (TCON.7)	TF1	Timer 1 overflow flag; Set to 1(by hardware) when Timer 1 overflows; Cleared to 0 automatically when controller vectors to interrupt service routine at address 001BH.
6 (TCON.6)	TR1	Timer 1 run control bit; set to 1 by a program to start timer/counter 1; Cleared to 0 to stop Timer/Counter 1.
5 (TCON.5)	TF0	Timer 0 overflow flag; Set to 1 (by hardware) when Timer 0 overflows; Cleared to 0 automatically when controller vectors to interrupt service routine at address 000BH.
4 (TCON.4)	TR0	Timer 0 run control bit; Set to 1 by a program to start timer/counter 0; Cleared to 0 to stop timer/counter 0.
3 (TCON.3)	IE1	External interrupt 1 edge flag; Set by hardware when external interrupt is detected on $\overline{\text{INT1}}$ pin; Cleared by hardware when controller vectors to interrupt service routine at address 0013H only when interrupt is configured as an edge-triggered interrupt (see IT1 bit below).
2 (TCON.2)	IT1	External Interrupt 1 signal type control bit; set to 1 by a program to configure interrupt 1 as a edge triggered (falling edge); cleared to 0 to configure it as level triggered (low level).
1 (TCON.1)	IE0	External interrupt 0 edge flag; Set by hardware when external interrupt is detected on $\overline{\text{INT0}}$ pin; Cleared by hardware when controller vectors to interrupt service routine at address 0003H only when interrupt is configured as an edge-triggered interrupt (see IT0 bit below).
0 (TCON.0)	IT0	External Interrupt 0 signal type control bit; set to 1 by a program to configure interrupt 0 as a edge triggered (falling edge); cleared to 0 to configure it as level triggered (low level).

The timers may be operated in either polling or interrupt mode. In polling mode, the timer overflow flag TF0/1 is polled (continuously monitored) by program instructions and action is taken when this flag is set. The interrupt mode generates the interrupt (timer interrupt) to the program when the timer overflows. The interrupt mode is useful when the

microcontroller's time is important and it has more than one activity to be done, because interrupts permits to execute program (important task) while timing activities continue in the background. In this chapter, we will focus on the polling mode only because of its simplicity. The interrupt mode which is more efficient will be discussed in more detail with examples in Chapter 16 (Interrupts).

14.4 TIMER CIRCUITS AS AN INTERVAL TIMER

14.4.1 Timer Mode 0

The operation of Timer Mode 0 is similar as well as subset of Timer Mode 1, therefore, Timer Mode 1 is described first and then after, Timer Mode 0 is discussed in brief.

14.4.2 Timer Mode 1

In Timer Mode 1, the timer behaves as a 16-bit timer. The timer can be configured in Mode 1 by setting mode bits M1M0 in the TMOD register as "01". Timer registers TLX and THX behaves as eight-bit up counters and collectively they form 16-bit up counter, allowing any value between 0000H to FFFFH to be loaded in these registers. In effect, the timer clock is divided by 256 (maximum) by TLX register and further it is divided by 256 (maximum) by THX register. The operation of Timer Mode 1 is shown in Figure 14.6.

As shown in Figure 14.6, the internal oscillator is the clock source for the timer operation (clock source is selected by $C/\overline{T} = 0$). The oscillator signal is divided internally by 12; therefore, the rate of the timer clock is one pulse per machine cycle. Once the timer is started by making TR0/1 = 1, the timer clock pulses are given to TLX (TL0 for Timer 0 and TL1 for Timer 1) register. The TLX register will work as an 8-bit up counter, i.e. the contents of TLX will be incremented by 1 after every clock pulse. The TLX register will divide (at maximum) the timer clock frequency by 256 based on the initial value present in it (the timer registers can be preloaded with any value as per requirement). The output pulse from TLX is given to THX register, which will further divide (at maximum) the clock pulse by 256. Therefore, THX and TLX collectively behave as a 16-bit up counter. When the count in this 16-bit counter reaches maximum value, i.e. 65535, the contents of the counter will roll back to 0000 and it will set timer overflow flag (TFX) and also generate a timer interrupt.

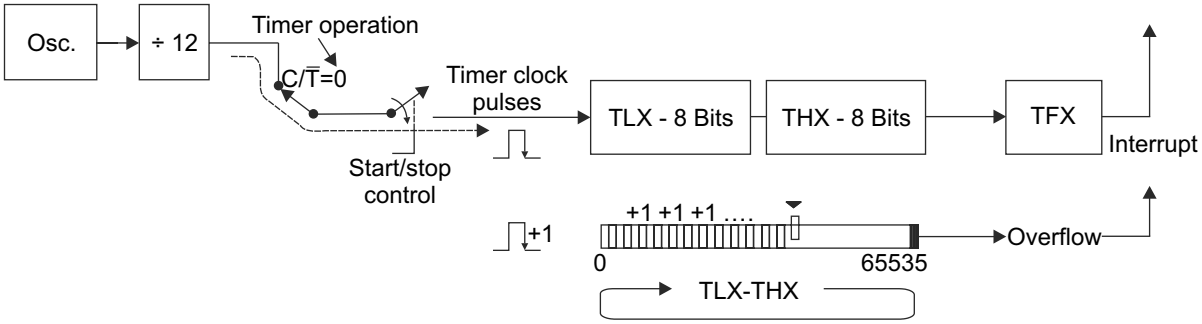


Fig. 14.6 *Timer Mode 1 operation*

1. Operation of the Timer in Mode 1

The exact value to be loaded in TLX and THX registers depend on the amount of time delay required and crystal frequency. It is discussed in the next topic, once the 16-bit value is loaded, the timer can be started by the instruction "SETB TR0" for T0 and SETB TR1 for T1 (or their equivalent instructions; see Example 14.3). Upon starting the timer, it starts to count up at every timer clock pulse, and continues until it reaches the maximum value FFFFH, and on the next pulse, it rolls over to 0000H. This event of rollover from FFFFH to 0000H is popularly referred as *timer overflow*. This timer overflow sets TF (Timer overflow) flag to 1 and interrupt of timer is generated (if enabled) indicating that specified time interval is elapsed or the required time delay is completed.

The timer overflow flag can be monitored using software or it may generate the interrupt when the TF flag is raised to inform the microcontroller core that the timer has completed its work and the required action has to be taken. The timer should be stopped by instruction CLR TRX (or equivalent instructions, see Example 14.3) when it overflows. It should be noted that both timers have their own timer overflow flag, TF0 for timer0 and TF1 for Timer 1.

To repeat the process of time measurement or delay generation, TLX and THX must be reloaded with the original value and TFX flag must be cleared to 0 and timer is started again by setting TRX bit.

Example 14.3

What are the instructions used to start and stop the timers?

Solution:

The timers can be started by programming TR0/1 bit as 1 and stopped by making TR0/1 as 0.

```
For timer T0,
To start,      SETB TR0              or
               SETB TCON.4           or
               ORL TCON, #00010000B (10H)
and to stop,   CLR TR0               or
               CLR TCON.4            or
               ANL TCON, #11101111B (EFH)
Similarly, for timer T1
To start,      SETB TR1              or
               SETB TCON.6           or
               ORL TCON, # 01000000B (40H)
and to stop,   CLR TR1               or
               CLR TCON.6            or
               ANL TCON, #10111111B (BFH)
```

2. Initial Value to be Loaded in Timer Registers

Assume crystal frequency = 12 MHz and we want to generate delay of 10 ms.

We know,

$$\text{Timer clock frequency} = \frac{\text{Crystal frequency}}{12} = \frac{12\text{MHz}}{12} = 1\text{MHz}$$

$$\therefore \text{ Therefore, Timer clock period} = \frac{1}{\text{Timer clock frequency}} = \frac{1}{1\text{MHz}} = 1\mu\text{s}.$$

Also, the 16-bit timer register is incremented by 1 (counts up; because timer registers behave as up counters) every timer clock pulse; therefore, the timer register in our example is incremented every 1μs.

Now, we should find how many timer clock pulses of 1μs are required to make a total interval (delay) of 10 ms. It is calculated by dividing 10 ms by 1μs.

Number of timer clock pulses required to generate a 10 ms delay.

$$= \frac{10\text{ ms}}{1\ \mu\text{s}} = 10000 \text{ pulses}$$

Therefore, once the timer is initialized, after 10000 cycles, it should overflow, i.e. it should roll over from FFFFH (65535) to 0000H, and the value to be loaded in timer registers (count) is,

Count = 65535 – 10000 +1 = 55536 in decimal, or

FFFFH – 2710H + 1 = D8F0 H in hexadecimal.

The extra +1 account for one extra cycle required to roll over from FFFFH to 0000H.

The count D8F0H should be loaded in to TLX and THX registers as THX = D8H and TLX = F0H. To summarize the discussion for this example, if we load timer registers with D8F0H, it will overflow after 10000 timer clock pulses (machine cycles).

In general, if the count = $n_1n_2n_3n_4$H, (four-digit hexadecimal value)

THX = n_1n_2 and TLX = n_3n_4.

The generalized procedure to find count for a specified delay can be summarized as

1. Calculate time period of timer clock pulse.
2. Calculate number of timer clock pulses *N*, required to generate desired time delay by dividing desired time delay by period of timer clock,
3. Count = FFFFH – N (Hex) +1, and
4. Let us say, Count $= n_1n_2n_3n_4$, Load TH and TL as TH $= n_1n_2$ and TL $= n_3n_4$.

Example 14.4

Find the count to be loaded into timer registers to generate a time delay of 500 μs and also write a program for the same using Timer 0 Mode 1. Assume crystal frequency is 12 MHz.

Solution:

First, let us find the count.

$$\text{Timer clock frequency} = \frac{\text{Crystal frequency}}{12} = \frac{12\text{ MHz}}{12} = 1\text{ MHz}$$

$$\text{and, timer clock period} = \frac{1}{\text{Timer clock frequency}} = \frac{1}{1\text{ MHz}} = 1\ \mu\text{s}.$$

$$\text{Number of timer clock pulses required to generate 500 μs delay} = \frac{500\ \mu\text{s}}{1\ \mu\text{s}} = 500\text{ pulses}$$

∴ Therefore, Count = 65535 – 500 +1 = 65036 = FE0CH

∴ Therefore, Load TH0 with FEH and load TL0 with 0CH

The program to generate the required delay is given below:

```
        MOV TMOD, #01H      // configure Timer 0 in Mode 1
        MOV TH0, #0FEH      // load count in timer registers TH0–TL0
        MOV TL0, #0CH
        SETB TR0            // start Timer 0
WAIT:   JNB TF0, WAIT       // wait for time delay of 500 μs
        CLR TR0             // stop timer
        CLR TF0             // clear timer overflow flag
```

3. Square-Wave Generation using Timers

The delay generated by timers can be used to generate square or rectangular waves. The square wave is generated by using timer as an interval timer, and toggling a microcontroller port pin (one or more pins) at repeated fixed intervals of times. The rectangular wave is generated by repeated toggling of the pins after unequal time periods depending upon desired duty cycle of the wave.

Example 14.5

Write a program to generate a square wave of 1 KHz frequency on P2.0.

Assume crystal frequency = 11.0592 MHz.

Solution:

The period of square wave is 1/1 KHz = 1 ms.

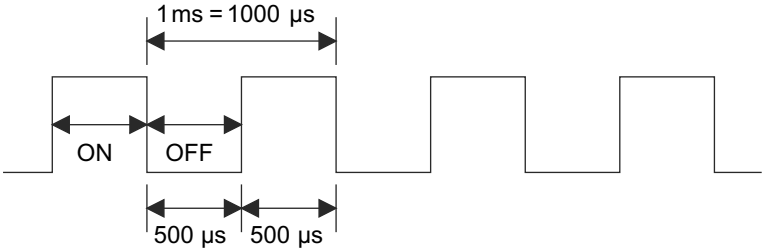


Fig. 14.7 *Illustration for Example 14.5*

Square waves have a duty cycle of 50%, so,
ON period = OFF period = Total Period/2
= 1000 μs/2 = 500 μs
We should generate the time delay of 500 μs and toggle the pin P2.0 every 500 μs to generate square wave of 1 KHz.
The frequency of timer clock = 11.0592 MHz/12 = 0.9215 MHz.
Therefore, the time period of timer clock = 1/0.9215 MHz = 1.085 μs (time period of one machine cycle). Also, the timer register will be incremented by one every 1.085 μs).
Number of timer clock pulses (of 1.085 μs) required to make 500 μs = 500/1.085 = 460.8 = 461 (approx.)
Count to be loaded into TH and TL = 65536 – 461(or 65535 – 461 + 1) = 65075 = FE33H
∴ Therefore, TH1 = 0FEH and TL1 = 33H.

The steps to develop the program to generate the square waves are as follows:
- Configure any one timer as an interval timer using TMOD register.
- Load initial count into timer registers THX–TLX to get desired delay.
- Start the timer.
- Wait until the timer overflows.
- Toggle the port pin on which square wave should be generated.
- Stop the timer and clear the overflow flag and repeat the process continuously

The program to generate the required square wave is given below,

```
        MOV TMOD, #10H      // configure Timer 1 as interval timer and in Mode 1
REPEAT: MOV TL1, #33H       // load count in timer registers TH1-TL1
        MOV TH1, #0FEH
        SETB TR1            // start Timer 1
WAIT:   JNB TF1, WAIT       // wait until timer overflows (wait for 500 μs)
        CLR TR1             // stop timer
        CLR TF1             // clear timer overflow flag
        CPL P2.0            // toggle P2.0 to get square wave
        SJMP REPEAT         // repeat above steps to generate square wave
```

Timing Analysis of the Above Program Let us consider the above program again with number of machine cycles for each instruction.

	Instructions	Machine Cycles
	MOV TMOD, #10H	2
REPEAT:	MOV TL1, #33H	2
	MOV TH1, #0FEH	2
	SETB TR1	1
WAIT:	JNB TF1, WAIT	2
	CLR TR1	1
	CLR TF1	1
	CPL P2.0	1
	SJMP REPEAT	2

Assume that the pin P2.0 is set initially (which is default status after reset).Our goal is to find out actual frequency of the square wave generated on P2.0. We will consider instruction CPL P2.0 (second last instruction) as a reference for doing calculations. To find out actual frequency, we need to find time (number of cycles) elapsed between successive (repetitive) executions of instruction CPL P2.0. It can be calculated as follows.

After completion of instruction CPL P2.0 for the first time, the program will go through the following sequence:

	Instruction sequence	Actual machine cycles taken
	SJMP REPEAT	2
REPEAT:	MOV TL1, #33H	2 jump to REPEAT as a result of above instruction
	MOV TH1, #0FEH	2
	SETB TR1	1

WAIT:	JNB TF1, WAIT	461* (wait until timer overflow flag set)
	CLR TR1	1
	CLR TF1	1
	CPL P2.0	1

Total = 471 cycles

(* 461 is taken for simplicity of explanation, otherwise, it will take 462 cycles because JNB TF1, WAIT instruction requires 2 cycles, so total time for repeated execution must be multiple of 2)

P2.0 is toggled after every 471 machine cycles; therefore, time for one complete cycle of square wave is 2 x 471 = 942 cycles. In seconds, it is 942 x 1.085 μs = 1022.07 μs.

Therefore, actual frequency is 1/1022.07 μs = 978.4 Hz (or 0. 9784 KHz)!!!

As per above calculations, we get an error of 21.6 Hz (1000 – 978.4)

There are two reasons for this error.

First, in every iteration, we need to initialize TH and TL registers, clear TR1 and TF1 and set TR1. So, time is wasted in doing that. This is more popularly known as 'overhead' time, or instructions to perform the above operations repeatedly are *overhead instructions*.

The second reason for getting an error is that the actual count to be loaded in timer registers is 460.8, but we have to round off this value to 461. The solution to minimize the error is to subtract time for overhead instructions from count loaded into timer registers (461– 10 = 451) which will generate a frequency of 999.63 Hz. (the error of 0.33 Hz cannot be eliminated because of rounding of the fractional count).

Simulation Result (In Keil μVision 4.0–IDE) The square-wave output on port pins can be observed in a logic analyzer window. Open the logic analyzer window from **View → Analysis widows → Logic analyzer window**. The snapshot of the output is shown below.

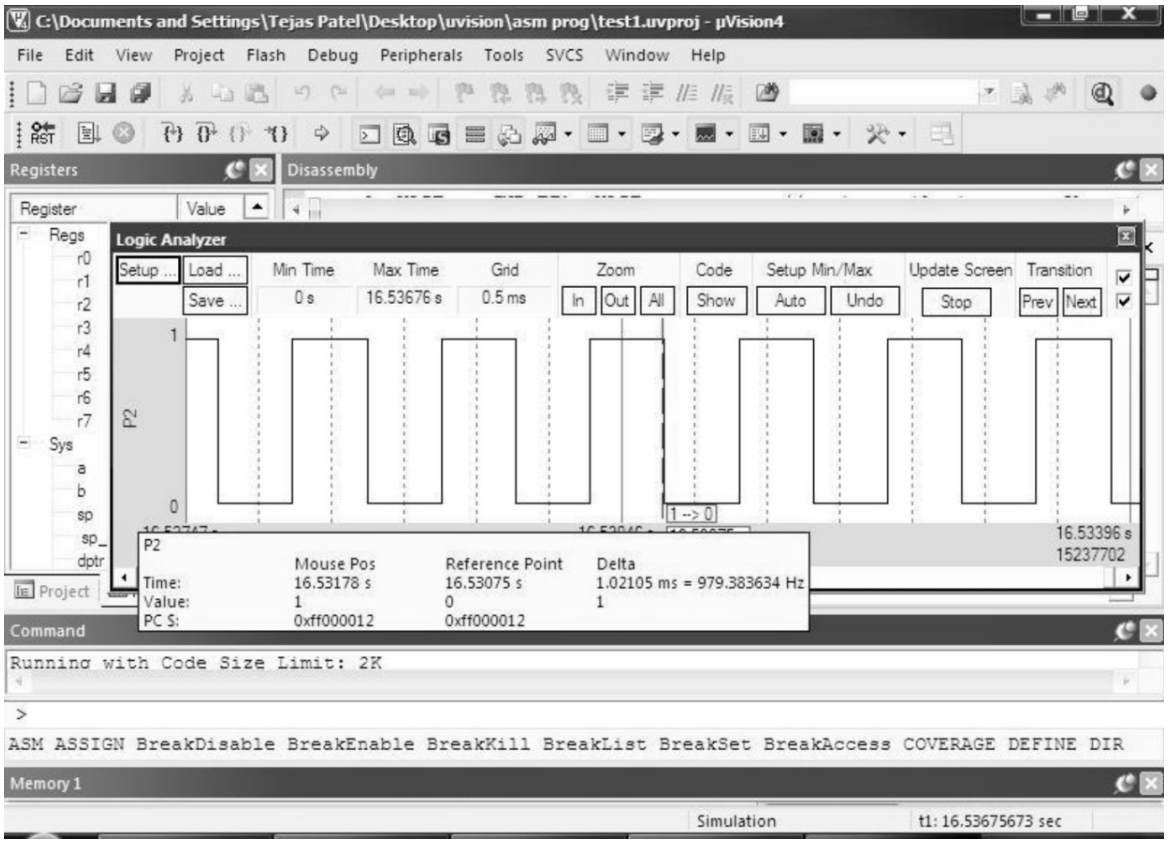


Fig. 14.8 *Program output*

Example 14.6

Rewrite the program of Example 14.5 in the C language.

Solution:

The port pin is toggled after the desired delay. The for (;;) loop is used to repeat the operation of toggling the port pin to generate a square wave. Within for loop, the timer registers are reloaded, the timer is started, and the program will wait until the desired delay is over.

```
# include <reg51.h>
        sbit square_bit  = P2 ^ 0;              // define P2.0 pin as square_bit
        void main ( )
        {
                square_bit  = 1;                // set 2.0 pin to logic '1' (high portion of square wave
                TMOD  = 0x10;                   // Timer 1 configured as timer and in Mode 2
                for (;;)                        // repeat the following statements forever
                                                // to generate square wave
                {
                TH1  = 0xFE;                    // load count in Timer registers TH1–TL1
                TL1  = 0x33;                    //
                TR1 = 1;                        // start Timer1
                while (TF1 ! = 1);              // wait until TF1 overflows
                TR1 = 0;                        // stop timer
                TF1 = 0;                        // clear timer overflow flag.
                square_bit = ~ square_bit;      // toggle p2.0 pin to get square wave
                }
        }
```

THINK BOX 14.1

How can we compensate the effect of overhead instructions while generating periodic pulses?

The time required to execute all overhead instructions must be considered while calculating the initial count to be loaded into timer registers (usually subtracted from the count)

Example 14.7

Write an assembly-language program to generate a square wave on P1.1 with ON time (high time) of 1 ms. Assume the crystal frequency is 11.0592 MHz.

Solution:

Since ON time = 1 ms, the OFF time (low time) of the wave will also be 1 ms; thus, time period of the wave is 2 ms and frequency of square wave is 500 Hz.

The frequency of the timer clock = 11.0592 MHz/12 = 0.9215 MHz

Therefore, the time period of the timer clock (one machine cycle) = 1/0.9215 MHz = 1.085 μs

Now, the number of timer clock pulses (of 1.085 μs) required to make 1 ms = 1000/1.085 = 921.65 = 922

Count to be loaded in to TH and TL = 65536 – 922 = 64614 = FC66H. Therefore, the count to be loaded in the timer registers is, TH1 = 0FCH and TL1 = 66H.

Using Timer 1 as an interval timer in Mode 1, the program will be

```
        MOV TMOD, #10H      // configure Timer 1in Mode 1
REPEAT: MOV TL1, #66H       // load count in TH1–TL1
        MOV TH1, #0FCH
        SETB P1.1           // set P1.1 high to get ON part of
                            // square wave
        SETB TR1            // start Timer 1
```

```
HERE:   JNB TF1, HERE       // wait until timer overflows
        CLR P1.1            // clear P1.1 to get OFF part of square wave
        CLR TR1             // stop timer
        CLR TF1             // clear overflow flag
        SJMP REPEAT         // reload timer registers and repeat
```

Example 14.8

Rewrite the program of Example 14.7 in C using the while loop.

Solution:

```
#include<reg51.h>
sbit square_bit  = P1^1;            // define P1.0 pin as square_bit

        void main()
        {
        TMOD = 0x10;                // configure Timer 1in Mode 1

        while(1)                    // repeat following program forever to generate square wave
                {
                TL1 = 0x66;         // load count in TH1-TL1
                TH1 = 0xFC;
                square_bit  = 1;    // set P1.1 high to get ON part of square wave
                TR1 = 1;            // start timer
                while (TF1! = 1) ; // wait until timer overflows
                TR1 = 0;            // stop timer
                TF1 = 0 ;           // clear timer overflow flag
                square_bit  = 0 ;  // clear P1.1 to get OFF part of square wave
                }
        }
```

Example 14.9

Write a C program to generate a rectangular wave with an ON time of 2 ms and an OFF time of 10 ms on pin P0.0 pin. Assume XTAL = 12 MHz.

Solution:

The frequency of timer clock = 12 MHz/12 = 1 MHz

Therefore, the time period of timer clock (one machine cycle) = 1/1 MHz = 1 μs

The count value for ON time = 65535– (2 ms / 1 μs) + 1
= 65536– 2000 = 63536
= F830H

Similarly,

The count value for OFF time = 65535 – (10 ms/1μs) + 1
= 65536 – 10000 = 55536
= D8F0H

Using Timer 0 in Mode 1, the program is given below.

The separate time delays for ON and OFF time is achieved by setting the port pin high and generating delay for ON time; then clearing the port pin and generation of delay for OFF time. The whole process is repeated using while (1) loop. To make the program more efficient, a function is defined, which will start the timer, wait until it overflows and then stop the timer and clear the timer overflow flag.

```
#include<reg51.h>
sbit rect_bit  = P1^0;                          // define P1.0 pin as rect_bit
```

```
void delay(void) ;                          // declare subroutine delay
        void main()
        {
                while(1)                    // repeat process continuously
                {
                        TMOD = 0X01;        // configure Timer 0 in Mode 1
                        TL0 = 0x30;         // load count in TH0-TL0 to get ON time
                        TH0 = 0xF8;
                        rect_bit  = 1;      // set P0.0 to get ON part of wave
                        delay () ;          // wait until ON time is elapsed
                        TL0 = 0xF0;         // load count in TH0-TL0 to get OFF time
                        TH0 = 0xD8;
                        rect_bit = 0;       // clear P0.0 to get OFF part of wave
                        delay ();           // wait until OFF time is elapsed

                }
        }

        void delay (void)                   // function will wait until overflow flag is set
        {
                        TR0 = 1;            // start Timer 0
                        while (TF0 =  = 0); // wait here until TF0 is set
                        TR0 = 0;            // stop timer
                        TF0 = 0;            // clear timer overflow flag
        }
```

Simulation Result (In Keil µ Vision 4.0–IDE) The square wave-output on port pins can be observed in the logic analyzer window. Open the logic analyzer window from **View → Analysis widows → Logic analyzer window**. The snapshot of the output is shown in Fig. 14.9.

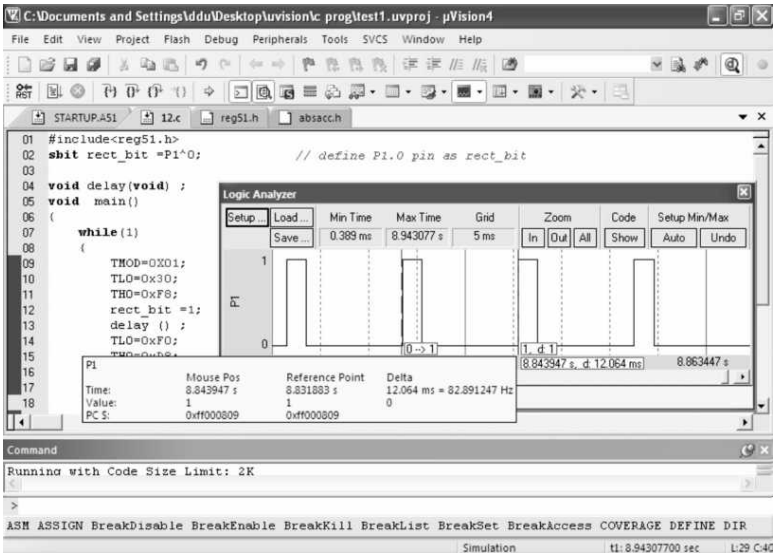


Fig. 14.9 *Program output*

4. *Timer Mode 0*

Timer Mode 0 is similar to Mode 1 except that it is 13-bit mode (8 bits of THX + lower 5 bits of TLX). Timer Mode 0 is kept in the 8051 only to maintain compatibility with the previous version of the microcontroller, the 8048 (MCS 48 family). It is rarely used in the 8051 based systems.

The operation of Timer Mode 0 is illustrated in Figure 14.10. The timer clock pulses are given to the TLX (only lower 5 bits are used in a Mode 0) register. The TLX register will work as a 5-bit up counter, i.e. the contents of TLX will be incremented by 1 after every timer clock pulse. The TLX register will divide (at maximum) the timer clock frequency by 32 based on initial value present in it (the timer registers can be preloaded with any value as per requirement). The output pulse from TLX is given to the THX register, which will further divide (at maximum) the clock pulse by 256. Therefore, THX and TLX (lower 5 bits) collectively behave as a 13-bit up counter. When the count in this 13-bit counter reaches maximum value, i.e. 8191 (1FFFH), the contents of the counter will roll back to 0000 and it will set the timer overflow flag (TFX) and also generate the timer interrupt.

To summarize, the timer in Mode 0 will roll over from maximum 13-bit value, i.e.1FFFH to 0000H and sets TFX flag. The other steps of program development remain same as Mode 1. The operation of Timer Mode 0 is understood by Example 14.10.

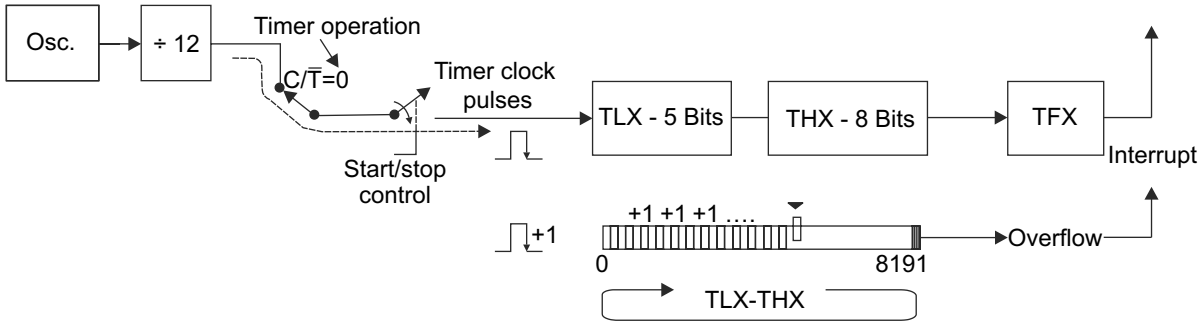


Fig. 14.10 *Timer Mode 0 operation*

Example 14.10

Find delay generated and frequency of signal generated on pin P1.5 by following program. Assume crystal frequency is 12 MHz.

```
        MOV TMOD, #0x00
REPEAT: MOV TL1, #0xF5
        MOV TH1, #0x50
        SETB TR1
WAIT:   JNB TF1, WAIT
        CPL P1.5
        CLR TR1
        CLR TF1
        SJMP REPEAT
```

Solution:

The steps of program development for Mode 0 are the same as Mode 1 except that it is 13-bit mode (we can use only lower 5 bits of TLX register). Refer Example 14.5 for steps to develop the program.

Note the use of immediate numbers used in the program. 0xF5 is same as 0F5H.

In a given program, Timer 1 is configured in Mode 0. It is a 13-bit mode, where only the lower 5 bits of TL1 and all bits of TH1 are used. In the given example, even if TL1 = F5H = 111**10101 b** only lower five bits are used for timing. Therefore, count represented by TL1 = 10101B = 15H and TH1 = 50H. Effectively, the count is 0101 0000 **111**1 0101 = 0101 0000 1 0101 = 0000 1010 0001 0101 = 0A15H = 2581d.

Timer will overflow after 8191– 2581+1 = 5611 timer clock cycles.

The frequency of clock signal given to the timer = 12 MHz/12 = 1 MHz.

The time period of timer clock = 1/1 MHz = 1 µs.

So, time delay generated by the program is 5611 x 1 µs = 5611 µs. (high or low period of square wave).

Therefore, frequency of signal generated on pin P1.5 = 1/ (2 x 5611 µs) = 89.11 Hz.

14.4.3 Timer Mode 2

When we want to generate a time delay repeatedly (as in the case of square-wave generation), we need to reload the timer registers every time they overflow (also, we have to clear the TFX flags and stop the timer every time), this overhead (as explained in Example 14.5) will cause little error in delay generated or programmer have to make extra effort (in terms of time and mental pain) to eliminate the timing error.

Mode 2 offers a simple way to reduce the effect of overhead and automatically reloads the timer register when it overflows. Timer Mode 2 is an 8-bit timer with auto-reload capability. The initial value loaded in THX register is copied into TLX register when the timer is either started or every time TLX overflows from FFH to 00H. The value in THX register remains unchanged.

Operation of Mode 2

Since Mode 2 is an 8-bit mode, any value from 00H to FFH can be initially loaded into the THX register as per required delay. When the timer is started, the value in THX is copied into the TLX register and TLX is incremented by 1 every machine cycle (timer clock pulse) until it reaches maximum value (FFH). Upon the next timer clock pulse, it rolls over from FFH to 00H and sets the timer overflow flag (TFX) to 1 or may also generate an interrupt. The TFX flag can be monitored using software or it may generate an interrupt to indicate that required time period is elapsed. When the timer overflows, it automatically reloads the copy of THX into TLX and continues to increment TLX. To repeat the process of time-delay generation, the only thing that we have to do is clear TFX flag so that it can be monitored to take the required action when it overflows once again. This mode also has applications in setting baud rate in serial communication. The operation of Timer Mode 2 is shown in Figure 14.11.

The process of calculating initial value to be loaded is similar to that of Mode 1. The differences are the following:

1. Calculate number of timer clock pulses *N*, required to generate desired time delay by dividing the desired time delay by period of timer clock, since it is an 8-bit mode, the maximum value of number *N* is 255 only.
2. The 8-bit timer rolls over from FFH to 00H; therefore, to find initial count to be loaded in to THX, use following equation:
   Count = FFH – N + 1 (instead of FFFFH – *N* + 1, see similar equation for Mode 1)

Discussion Question What is the maximum time delay generated by Timer Mode 1?

Answer The time delay that can be generated, depends on two factors:

1. Crystal frequency
2. Initial count loaded in timer registers

Therefore, maximum delay is generated when minimum count is loaded, i.e. it is time period for 65536 timer clock pulses (65536 – 0000). It is illustrated in Example 14.11.

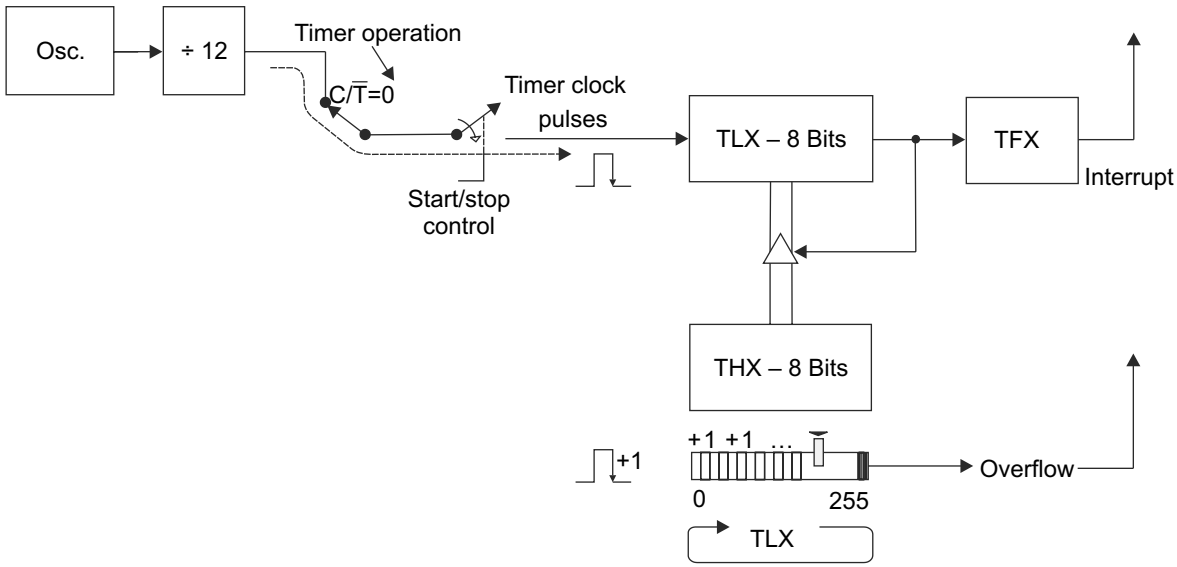


Fig. 14.11 *Timer Mode 2 operation*

Example 14.11

Find the maximum delay generated using Timer Mode 1 and 2. Assume crystal frequency is i) 12 MHz, and ii) 6 MHz.

Solution:

For Timer Mode 1: It is a16-bit mode.

To get the maximum delay, we should load minimum count in the timer registers, i.e. 0000H. So the timer overflow flag will overflow after 65536 cycles. (i) For 12 MHz crystal frequency, time period of one machine cycle (time period of clock signal of timer) is 1μs. Hence, maximum delay that can be generated in Mode 1 is 65536 μs. (ii) For 6 MHz crystal frequency it is 65536 x 2 = 131072 μs.

For Timer Mode 2: It is an 8-bit mode.
To get the maximum delay, we should load minimum count in the timer registers, i.e. 00H. So timer overflow flag will overflow after 256 cycles. (i) For 12 MHz crystal frequency, maximum delay that can be generated in Mode 2 is 256 µs. (ii) For 6 MHz crystal frequency, it is 256 x 2 = 512 µs.

Example 14.12

Write an assembly-language program to generate a square wave of 10 KHz frequency on Pin P1.0 with crystal frequency equal to 24 MHz.

Solution:
Time period of one cycle of 10 KHz = 1/F = 1/10 KHz = 0.1 ms
ON time = OFF time = 0.1 ms/2 = 0.05 ms = 50 µs
The frequency of clock signal given to the timer = 24 MHz/12 = 2 MHz
The time period of timer clock = ½ MHz = 0.5 µs.

The count to be loaded in TH0 is,
count = maximum value of Mode 2 – (pulse period / timer clock period) + 1
= 255 – (50 µs / 0.5 µs) + 1
= 256 – 100
= 156
= 9CH
Let us use Timer 0 in Mode 2.
Therefore, TH0 = 9CHh

```
        MOV TMOD, #02H          // configure Timer 0 in Mode 2
        MOV TH0, #09CH          // load count in TH0
        SETB P1.0               // set P1.0 to get ON time of wave
        SETB TR0                // start Timer 0
HERE:   JNB TF0, HERE           // wait until Timer 0 overflows
        CPL P1.0                // complement pin to get square wave
        CLR TF0                 // clear TF0
        SJMP HERE               // repeat process for ever
```

Note that the count is loaded only in the TH0 register. The contents of TH0 will be automatically copied into TL0 every time when the timer is started or when the timer overflows. Therefore, we need not to reload TL0 for the next cycle (iteration). Also, note that there is no need to stop the timer when it overflows. This will reduce overhead instructions and timing error.

Example 14.13

Write a C program for Example 14.12.

Solution:

```
#include<reg51.h>
sbit square_pin  = P1^0 ;                   // define P1.0 as square_pin
        void main()
                {
                TMOD = 0x02;                // configure Timer 0 in Mode 2
                TH0 = 0x9C;                 // load count in TH0
                square_pin = 1;             // set P1.0 to get ON time of wave
                TR0 = 1;                    // start timer
                while (1)                   // repeat process for ever
                        {
                        while (TF0 =  = 0);         // wait until Timer 0 overflows
                        square_pin  = ~ square_pin ;// toggle pin P1.0 to get square wave
                        TF0 = 0;                    // clear overflow flag
                        }
                }
```

14.4.4 Generating Larger Delays

As seen in Example 14.11, maximum delay that can be generated using timers is very small (only 65536 cycles for Mode 1 and 256 cycles for Mode 2). To generate larger time delays, the simplest solution is to add smaller delays to get larger delay; this can be done by repeating smaller delays using loops. Generation of larger delay is illustrated in Example 14.14.

THINK BOX 14.2

What techniques should be used to create timing intervals (delay) greater than 65536 machine cycles?
16-bit timer (Mode 1) with software loops.

Example 14.14

Write a subroutine to generate delay of 5 seconds using timers. Assume crystal frequency is 12 MHz.

Solution:

A delay of 50 ms is generated using Timer 0 and it is repeated 100 times using a loop.
Timer works with frequency 12 MHz/12 = 1 MHz and time period is 1/1 MHz = 1 μs.
To generate a delay of 50 ms, timer clock cycles required is 50 ms/1 μs = 50000 cycles.
Therefore, count to be loaded is 65536 – 50000 = 15536 = 3CB0 H.
TH0 = 3CH and TL0 = 0B0H.

```
DELAY:      MOV R0, # 100       // 50 ms delay is repeated 100 times to get 5s
            MOV TMOD, #01H      // configure Timer 0 as a timer in Mode 1
REPEAT:     MOV TH0, #0x3C      // load count in timer registers TH0-TL0
            MOV TL0, #0x0B0
            SETB TR0            // start Timer 0
WAIT:       JNB TF0, WAIT       // wait here until timer overflows
            CLR TR0             // stop timer
            CLR TF0             // clear timer overflow flag
            DJNZ R0, REPEAT     // repeat loop 100 times
            RET
```

Note: Overhead due to instructions is neglected. Refer Example 14.5 to understand how to generate exact delay.

Example 14.15

Rewrite the subroutine of Example 14.14 in the C language.

Solution:

```
void delay ()
    {
        unsigned char i;
        for (i = 0; i<100; i++)          // repeat loop 100 times
                                         // 50 ms delay is repeated 100 times to
                                         // generate delay of 5s
        {
        TMOD  = 0x01;                    // configure Timer 0 as a timer in Mode 1
        TL0 = 0xB0;                      // load count in timer registers TH0-TL0
        TH0 = 0x3C;
        TR0 = 1;                         // start timer
        while (TF0! = 1);                // wait for TF0 to roll over
        TR0  = 0;                        // stop Timer 0
        TF0  = 0;                        // clear TF0 flag
        }
    }
```

Example 14.16

Write an assembly-language program to generate a rectangular wave of 5 Hz of 25% duty cycle on pin P1.0.

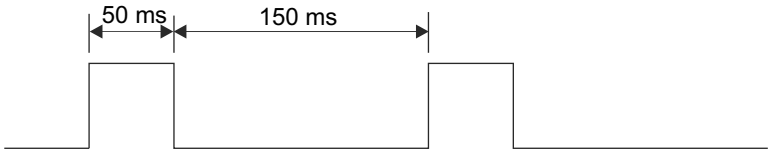


Fig. 14.12 *Illustration for Example 14.16*

Solution:

ON time for 5 Hz signal for 25 % duty cycle is 50 ms and OFF time is 150 ms.

Delay routine of 50 ms (from Example 14.14) can be used for ON time and the same routine can be called three times to generate a delay of 150 ms for OFF time.

```
AGAIN:    SETB P1.0          // set P1.0 = 1 for ON time
          ACALL DELAY        // delay of 50 ms
          CLR P1.0           // clear P1.0 for OFF time
          ACALL DELAY        // delay of 150 ms (3x 50 ms)
          ACALL DELAY
          ACALL DELAY
          SJMP AGAIN         // repeat above steps to generate square wave
                             // continuously
DELAY:    MOV TMOD, #01H     // configure Timer 0 in Mode 1
          MOV TH0, #0x3C     // load count in timer registers
          MOV TL0, #0xB0     // the initialization part may be placed at
                             // beginning of the program
          SETB TR0           // start timer
WAIT:     JNB TF0, WAIT      // wait until timer overflows
          CLR TR0            // stop timer
          CLR TF0            // clear timer overflow flag
          RET                // return to calling program
```

Example 14.17

Write an assembly-language program to generate a square wave with total time period of 2 seconds on Pin P2.7. Use Timer 1 in Mode 1. Assume crystal frequency as 24 MHz.

Solution:

For a total time period of 2 seconds, ON time = OFF time = 1 s. The maximum delay possible with a crystal frequency of 24 MHz, is when both TH and TL is initialized with count 00 and that delay is 32.76 ms. If this delay is repeated 31 times, we will get a delay of 1016 ms ~ 1 second.

```
          SETB P2.7          // initialize pin with high logic
          MOV TMOD, #10H     // configure Timer 1 in Mode 1
AGAIN:    MOV R2, #31        // loop counter to repeat 32.76 ms delay 31 times
                             // (32.76 x 31) to get 1 s delay
BACK:     MOV TL1, #00h      // load count value in TH1-TL1
          MOV TH1, #00h
          SETB TR1           // start timer
HERE:     JNB TF1, HERE      // wait until timer overflows
          CLR TR1            // stop timer
          CLR TF1            // clear timer overflow flag
          DJNZ R2, BACK      // repeat delay of 32.76 ms for 31 times
          CPL P2.7           // toggle pin P2.7 to get square wave
          SJMP AGAIN         // repeat for continuous wave generation
```

Example 14.18

Rewrite the program of Example 14.17 in the C language.

Solution:

The larger delay is achieved by repeating a smaller delay using for loop. Here, the delay of 32.76 ms is repeated 31 times to get a 1 s delay.

```
#include<reg51.h>
sbit toggle_pin = P2^7 ;                    // define P2.7 as toggle_pin
void main()
        {
        unsigned char i ;
        toggle_pin = 1;                     // initialize pin with high logic
        TMOD = 0x10;                        // configure Timer 1 in Mode 1
        while (1)                           // repeat following code forever for
                                            //continuous square wave generation
                {
                for (i = 0; i<31; i++)      // loop to repeat 32.76 ms delay 31 times
                                            // (32.76 x 31) to get 1 s delay
                        {
                        TL1 = 0x00;         // load count value in TH1-TL1
                        TH1 = 0x00;
                        TR1 = 1;            // start timer
                        while (TF1 =  = 0) ;// wait until timer overflows
                        TR1 = 0;            // stop timer
                        TF1 = 0;            // clear overflow flag
                        }
                toggle_pin  = ~ toggle_pin ;// toggle pin P2.7 to get square wave
                }
        }
```

Example 14.19

Write a C program to generate a delay of 500 ms.

Solution:

Assume that crystal frequency = 11.0592 and use Timer 0 in Mode 1. Maximum delay will be generated when TH0 and TL0 are loaded with minimum values, i.e. 00. For 11.0592 MHz crystal frequency, the maximum delay is 71.1 ms (65536 x 1.085). For this example, delay to be generated is too large; therefore, this larger delay is generated by repeating small delay. We will generate a 50 ms delay using the timer and that will be repeated by 10 times to get the desired delay of 500 ms.

```
Count  = 65536 – (50 ms / 1.085 us)
       = 65536 – 46083
       = 19453
       = 4BFDH

delay()
        {
        unsigned char i ;
        TMOD = 0x01;                        // Timer 0, Mode 1
        for (i = 0; i<10; i++)              // repeat 50 ms delay 10 times to get 500 ms
                {
                TL0 = 0xFD;                 // load count in TH0-TL0
                TH0 = 0x4B;                 //
                TR0 = 1;                    // start Timer 0
```

```
        while (TF0! = 1) ;          // wait for TF0 to roll over
        TR0 = 0;                    // stop Timer 0
        TF0 = 0;                    // clear TF0
        }
}
```

Note that only delay routine is given.

14.4.5 Timer Mode 3

In Mode 0, 1 and 2, both timers may operate independently. But if Mode 3 is selected for Timer 0, they cannot operate independently.

Timer 0 in Mode 3 behaves as two completely separate 8-bit timers, TL0 is an 8-bit interval timer/event counter controlled by Timer 0 control bits (C/$\overline{T}$, GATE, TR0, TF0 and $\overline{INT0}$) and TH0 is the only 8-bit interval timer controlled by Timer 1 control bits TR1 and TF1 and, therefore, controls the Timer 1 interrupt. Timer Mode 3 is also referred as *split timer mode*. The control logic for Timer Mode 3 is shown in Figure 14.13.

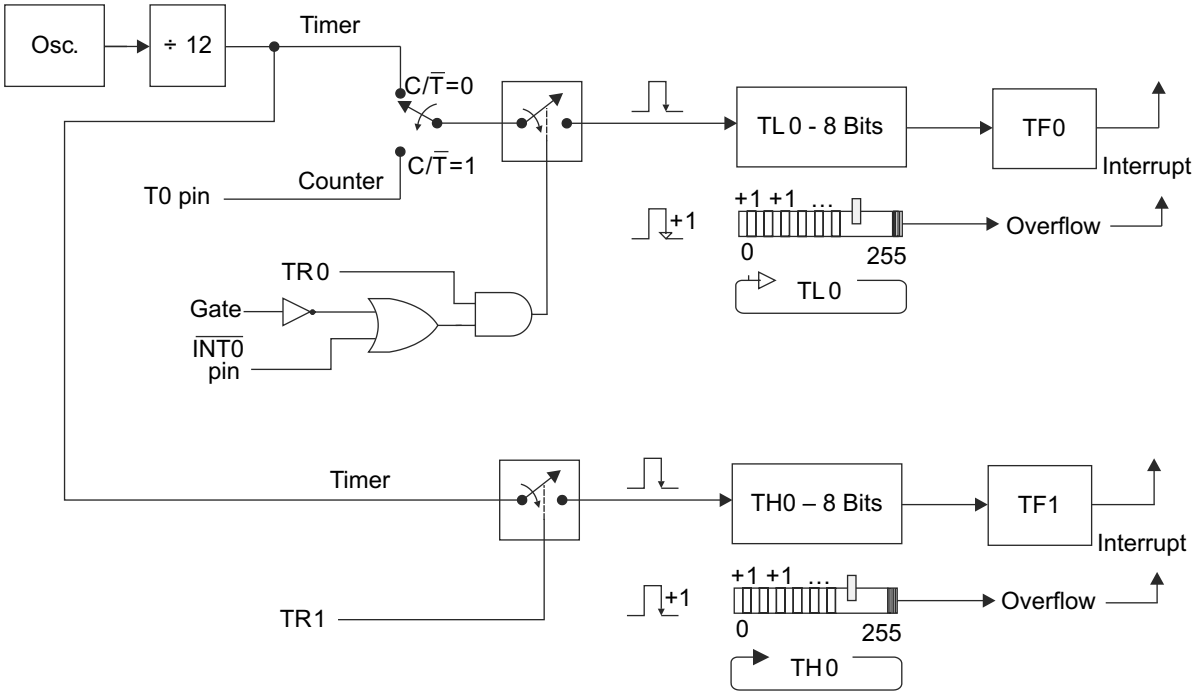


Fig. 14.13 *Timer Mode 3 operation*

While the timer T0 is in Mode 3, the timer T1 may be independently used in Mode 0, 1 and 2 with two major limitations. First, an interrupt cannot be generated by Timer 1 because TF1 overflow flag is used by Timer 0 (TH0); second, the START/STOP of timer cannot be controlled by TR1 flag because it is now used to START/STOP Timer 0 (TH0). Programming Timer 1 in Mode 3 will stop it, so Timer T1 can be used for an application that does not require use of an interrupt for its operation.

14.4.6 Reading the Value of a Timer

Since the 8051 is an 8-bit microcontroller, reading the timer value in 8-bit modes (Mode 2-autoreload or Mode 3-split timer mode) is simple because we need to read only 1 byte value.

But, reading a 13 or 16-bit timer value when the timer is running is complicated, because, while reading a value of TLX, the value of THX may get changed. For example, consider that we wish to read Timer 1 value, assume that TH1 = 00H and TL1 = FFH when the instruction to read TL1 is executed. Now by the time this instruction is executed (it will take at

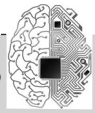
least 1 machine cycle!), TH1 will be 01 (TL1 will roll over from FFH to 00H and TH1 will be incremented from 00H to 01H, i.e. 16-bit value will change from 00FFH to 0100H). Now if we read TH1, we will get 01H. Thus, the 16-bit timer value read by the instructions is 01FFH instead of 00FFH!!! Therefore, to get the correct value of timer count, we have to avoid reading the timer count when TL1 rollover from FFH to 00H. This can be done as follows:

- First, read the higher byte of the timer (THX).
- Then, read the lower byte (TLX).
- Now, read the high byte once again.
- If the high byte read during second time is not the same as that of the first time, repeat the process.

These steps are implemented using the following instructions.

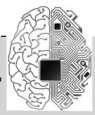
```
REPEAT:   MOV A, TH1
          MOV R2, TL1
          CJNE A, TH1, REPEAT
```

Now, if we want to determine the exact timer value when the instruction MOV A, TH1 was executed for the first time, we need to make adjustments in TL1 and TH1 values that we get from the above code. It is left as an exercise for a reader to develop a program to make these adjustments.



THINK BOX 14.3

Assume that Timer 1 is running and TH1 = 00 and TL1 = FF. What value (16-bit) do we get if we read TH1 first and then TL1?
The 16-bit value will be 0000. Because, when an instruction to read TH1 is completed, the TL1 will overflow from FFH to 00H.



THINK BOX 14.4

What will happen if we load timer registers (16-bit value) when the timer is running?
Sometimes, an incorrect value will be loaded. For example, consider we want to load 00FFH in timer registers. Assume that we load FF first, in the next clock cycle, this value will overflow to 00 and it will increment the higher byte (THX). During the same time, if we load 00 (higher byte) to THX, the 16-bit value loaded is 0000.

Example 14.20

Write a program to generate a square wave of 50 Hz on P1.1 using Timer 1 when Timer 0 is in Mode 3. Assume crystal frequency is 12 MHz.

Solution:

Since we do not have TF1 flag to determine overflow of Timer 1, we have to monitor the value of Timer 1 registers to determine whether required time is elapsed or not. Moreover, we have to use Timer 1 in 16-bit mode (Mode 1) because the count to be loaded for half cycle time (10 ms) is 10000 (2710H). Here, we load Timer 1 registers with initial values 0000H (TH1 = 00H, TL1 = 00H) and we have to continuously monitor timer registers (TH1 and TL1 collectively) until it exceeds the required count (10000 in this example). Once the timer value exceeds the desired count, we will toggle the port pin to generate the square wave and stop Timer 1 by placing it in Mode 3, reload the initial value and restart the timer by placing it in Mode 1.

```
          ORG 0000H              // program starting address
          MOV 20H, #10H          // count for desired delay of 10 ms-LSByte
          MOV 21H, #27H          // count for desired delay of 10 ms-MSByte
          MOV TMOD, #33H         // both timers in Mode 3, Timer 1 operation in
                                 // Mode 3 will be stopped – it will hold the count
BACK:     MOV TH1, #00H          // load initial value in Timer 1 registers
           MOV TL1, #00H
          ANL TMOD, #1FH         // start Timer 1 in Mode 1 when Timer 0 is in Mode 3
```

```
REPEAT:  MOV A, TH1          // read the Timer 1 value, avoid reading count
         MOV R2, TL1         // when TL1 rollover from FFH to 00H
         CJNE A, TH1, REPEAT

         MOV R3, TH1         // compare the 16-bit timer value with the count
         CLR C               // for desired delay
         MOV A, R2
         SUBB A, 20H
         MOV A, R3
         SUBB A, 21H
         JC REPEAT           // monitor timer value until it exceeds count of desired delay
         ORL TMOD, #30H      // stop Timer 1, configuring Timer 1 in Mode 3 will stop it
         CPL P1.1            // complement port pin to get square wave
         SJMP BACK           // repeat the process forever
         END                 // end of program
```

The limitation of the above program is that we will get the timing error because of the overhead of reading a 16-bit timer value, comparing it with desired count and reloading the initial count in the timer registers. The timing error can be minimized by adjusting the count for desired delay, i.e. subtract the number of machine cycles required for the overhead instructions from the desired count. This will counteract the effect of timing error because of overhead instructions!!!

This example illustrates how only Timer 1 can be used without overflow and timer run flag to perform the task when Timer 0 is in Mode 3. The real power of Mode 3 is in using Timer 0 as two 8-bit timers and Timer 1 as 8, 13 or 16-bit times—three independent timing tasks. The use of two timers to perform three independent tasks requires the use of interrupts; therefore, it is illustrated in Chapter 16, Example 16.9.

14.5 | TIMER AS AN EVENT COUNTER

The major difference between interval timer and event counter is source of the clock pulse, when timers are used as an event counters, pin T0/T1 (P3.4/P3.5) are basically used to provide external pulses for Timer 0/1; therefore, external pulses will increment the timer registers TLX and THX. The timer can be configured as an event counter by setting $C/\overline{T} = 1$ in the TMOD register. The other difference between interval timer and event counter is that the counter is normally started with initial value of "0000" so TLX and THX are normally initialized with value 00H. The operation of the event counter is illustrated in Figure 14.14.

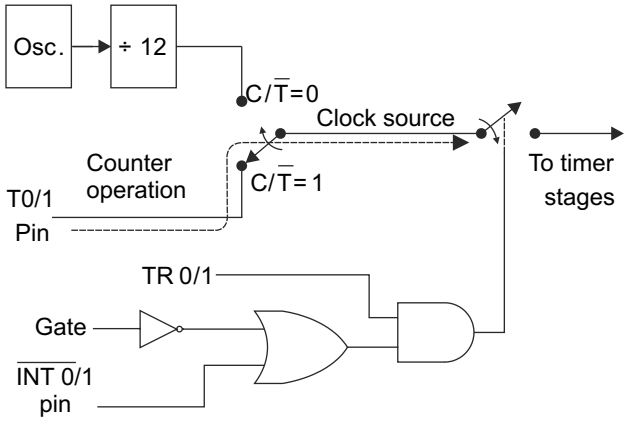


Fig. 14.14 *Event counter operation*

As shown in Figure 14.14, the pulses on the external pin T0 (or T1 for Timer 1) are selected as clock source for the timer operation by setting $C/\overline{T} = 1$ in the TMOD register. The timer registers are normally initialized with 0000 value and the count will increment by 1 when the pulse on pin T0/1 is applied; therefore, the circuit will count the number of pulses (events) applied externally to the timer pins. The count in timer registers at any time will represent the number of pulses applied (or external events occurred) till that time.

Example 14.21

Write a program to count external pulses applied to Timer 1 input T1 (P3.5 pin). Display continuously the count on P2 (LSByte) and P1 (MSByte).

Solution:

The steps to develop a program to count the pulses applied at timer input pins T0/1 are

- Configure any one timer as counter using TMOD register
- Initialize THX-TLX registers with 0000
- Start counting by setting TR0/1 bit, the counter will start counting
- Read timer registers and save value at desired location

```
        ORG 0000H
        MOV TMOD, #50H      // Timer 1 configured in mode1 as a counter
        MOV TL1, #00H       // initial count loaded in TH0-TL0 pair is 0000H
        MOV TH1, #00H
        MOV P2, #00H        // clear P2 and P1
        MOV P1, #00H
        SETB P3.5           // configure T1 pin as an input
        SETB TR1            // start counting
AGAIN:  MOV P2, TL1         // send count on P1, P2 continuously.
        MOV P1, TH1
        SJMP AGAIN          // repeat the process of reading timer registers
        END
```

Simulation Procedure (In Keil μVision 4.0 IDE)

The application of the external pulses can be simulated as follows:

- Free run (or single step) the program.
- Open peripheral windows of Port 1, 2 and 3 from **Peripherals → I/O ports** menu.
- Pin P3.5 is T1 input for Timer 1; therefore, we should apply pulses at this pin.
- Click repetitively on the box of P3.5; this will be considered as an external pulse. Note that the counter registers are incremented on the negative edge of the pulse. A single pulse can be simulated by pair of clicks, i.e. check the box by clicking on it and clear the box by clicking on it again.

Example 14.22

What is the limitation of the program in Example 14.21?

Solution:

Since the maximum value that can be collectively stored by TH1 and TL1 registers is FFFFH, the above program can count up to FFFFH (65535). After that TH1 and TL1 will roll over to 0000H.

Example 14.23

Write a program to count up to 10000000 using T1 (P3.5 pin).

Store the most significant byte (MSByte) of the count in RAM location 50H, middle byte in 51H and LSByte in 52H. Also send MSByte on P3, middle byte on P2 and LSByte on P1.

Solution:

Once the counter reaches the maximum value of FFFFH, the contents of temporary variable (internal RAM address 50H in the example) is incremented. This process is repeated until the temporary variable reaches FFH; this means we are counting 0000 to FFFFH, FF times (FFFFH x FFH).

```
        MOV TMOD, #50H      // Timer 1 configured in mode1 as a counter
        MOV 50H, #00H       // initial count is 000000H
        MOV 51H, #00H
        MOV 52H, #00H
        MOV TL1, #00H       // timer registers are initialized with 0000H
        MOV TH1, #00H
        MOV P3, #00H        // clearP3, P2 and P1
        MOV P2, #00H
        MOV P1, #00H
```

```
            SETB P3.5           // configure T1 pin as an input
            SETB TR1            // start counting
AGAIN:      JB TF1, INCREMENT
CONTINUE:   MOV P1, TL1         // send count on P3, P2, and P1 as well as
            MOV 52, TL1         // store count in 50H, 51H, 52H
            MOV P2, TH1
            MOV 51, TH1
            MOV P3, 50H
            SJMP AGAIN
INCREMENT:  INC 50H             // increment 50 every time counter rolls over
            CLR TF1             // clear timer overflow flag
            SJMP CONTINUE
```

Note: The above program can count from 0 to 16711425 (0 to FEFF01H).

Example 14.24

Modify the program of Example 14.21 to use Timer 1 as a counter in Mode 2.
Display continuously the count on P2. Discuss limitation of the program.

Solution:

```
            MOV TMOD, #60H      // timer mode1 as counter in Mode 2
            MOV TL1, #00H       // initial count is 00H
            MOV TH1, #00H
            MOV P2, #00H        // clear P2
            SETB P3.5           // configure T1 pin as an input
            SETB TR1            // start counting
AGAIN:      MOV P2, TL1         // send count on P2 continuously.
            SJMP AGAIN
```

The limitation is that the above program can count from 0 to 255 only as the timer is used in Mode 2.

14.6 | FREQUENCY MEASUREMENT USING TIMERS

Frequency of the signal is defined as number of cycles per second, and for a digital signal, it can also be defined as number of pulses per second. Therefore, if we count the number of pulses for one second, the count is effectively the frequency of the signal. For frequency measurement, we will use one timer as an interval timer and the other timer as a counter. We will enable the counter for 1 second using the interval timer and after 1 second, we will stop the counter. The count in the counter after 1 second is the frequency of the unknown signal.

THINK BOX 14.5

What is the maximum frequency of input pulse that can be measured reliably using the 8051 timer (counter)?
The input pulses provided at T0 (or T1) are sampled in P2 of State 5 of each machine cycle. The high-to-low transition of input samples will increment the counter by 1. Therefore, to detect the transition (for reliable counting), each high and low level of input pulse must be kept stable for at least one machine cycle. So, minimum time period of one cycle of input pulse should be 2 machine cycles, i.e. 24 clock pulses. So, maximum input pulse frequency that can be reliably counted is equal to oscillator frequency/24.

Example 14.25

Write a program to find frequency of the unknown signal applied at Timer 0 input (T0) pin. Assume crystal frequency is 24 MHz. Display the frequency on Port 1 (lower byte) and Port 2 (higher byte).

Solution:

Since the signal is applied at T0 pin, we need to use Timer 0 as a counter and Timer 1 as an interval timer.

As discussed earlier in Example 14.17, the maximum delay generated by Timer 1 in Mode 1 = 32.76 ms for XTAL = 24 MHz. This delay will be repeated 31 times to get a delay of 1second (approx.).

```
        MOV TMOD, #15H      // configure Timer 1 as interval timer in Mode 1 and Timer 0 as a counter in Mode 1
        SETB P3.4           // configure T0 pin as an input
        MOV TL0, #00h       // Initialize TH0-TL0 for counting the frequency
        MOV TH0, #00h
        MOV R2, #31         // loop counter to repeat 32.76 ms delay 31 times (32.76 x 31) to get 1 s delay
        SETB TR0            // start Timer 0
BACK:   MOV TL1, #00h       // load count value in TH1-TL1
        MOV TH1, #00h
        SETB TR1            // start Timer 1
HERE:   JNB TF1, HERE       // wait until timer1 overflows
        CLR TR1             // stop timer1
        CLR TF1             // clear timer overflow flag1
        DJNZ R2, BACK       // repeat delay of 32.76 ms for 31 times
        CLR TR0             // stop counter after 1 s
        MOV P1, TL0         // display lower byte of frequency at Port 1
        MOV P2, TH0         // display higher byte of frequency at Port 2
```

Example 14.26

Rewrite the program of Example 14.25 in the C language

Solution:

```
#include<reg51.h>
sbit in_signal  = P3^4                          // define P3.4 as in_signal
void main()
{
        unsigned char i ;
        in_signal = 1;                          // configure T0 pin as an input
        TMOD = 0x15;                            // configure Timer 1 as interval timer in Mode 1
                                                // and Timer 0 as a counter in Mode 1
        TL0 = 00;                               // initialize TH0-TL0 for counting the frequency
        TH0 = 00;
        TR0 = 1;                                // start timer 0

                for (i = 0; i<31; i++)          // loop to repeat 32.76 ms delay 31 times
                                                // (32.76 x 31) to get 1 s delay
                        {
                        TL1 = 0x00;             // load count value in TH1-TL1
                        TH1 = 0x00;
                        TR1 = 1;                // start timer
                        while (TF1 =  = 0) ;    // wait until timer1 overflows
                        TR1 = 0;                // stop Timer 1
                        TF1 = 0;                // clear overflow flag
                        }
                TR0 = 0;                        // stop counter after 1s
                P1 = TL0;                       // send lower byte of frequency at Port 1
                P2 = TH0;                       // send lower byte of frequency at Port 1
        }
```

Discussion Question List the applications of timers/counters.

Answer Timers are commonly used for the following applications:

- Square (rectangular) wave generation
- Pulse-width modulation
- Pulse-width measurement
- Pulse generation
- Frequency measurement
- Counting

THINK BOX 14.6

How can we use the timer/counter of the 8051 for angular speed measurement?

We should use an optical or electromechanical sensor which generates one (or more) pulse per revolution with use of signal-conditioning circuit. The pulses can be given to the timer (say T0) input when corresponding timer (Timer 0) is configured as a counter, use other timer (Timer 1) as interval timer (along with loops) to generate a delay of 1 second. Start both timers simultaneously and stop Timer 0 after 1 s when Timer 1 overflows after 1 second. Now the count in Timer 0 registers is the speed of rotation in rotations per minute.

POINTS TO REMEMBER

- The basic functions of the timers are to measure the time (generating time delays), to count the events or to measure frequency of the unknown signal and to provide clock signal to other circuits.
- The 8051 has two 16-bit timers, Timer 0 and Timer 1, which can be programmed independently. Each timer can be configured either as an interval timer or as an event counter in various operating modes.
- The TLX (lower byte) and THX (higher byte) registers are collectively used as a 16-bit register.
- The start/stop operation of the timers can be controlled either by software or hardware.
- The interval timer means calculating time elapsed between events or generating a delay.
- The event counter counts pulses that are generated by external events.
- The only difference between interval timer and event counter is the clock source used by the timer circuit.
- For interval timer, the internal clock signal generated by the crystal oscillator is used as source of the clock.
- For event counter, the clock source for the timer circuits is pulses on pin T0 or T1 of microcontroller for Timer 0 and Timer 1 respectively.
- In the polling mode, timer overflow flag is polled (continuously monitored) and action is taken when this flag is set.
- In the interrupt mode, the interrupt (timer interrupt) is generated when the timer overflows.
- Upon starting the timer, it starts to count up at every timer clock pulse, and continues until it reaches the maximum value FFFFH (for Mode 1), and on next pulse it rolls over to 0000H.
- The Timer Mode 2 is an 8-bit timer with auto-reload capability which reduces the effect of initialization overheads.
- If Mode 3 is selected for Timer 0, T0 and T1 cannot operate independently.

OBJECTIVE QUESTIONS

1. If the 8051 is operated at a crystal frequency of 12 MHz and TMOD = 10H, Timer 1 is operating at a frequency of,
   (a) 12 MHz (b) 1 MHz (c) 921.6 kHz (d) none of the above
2. If the 8051 is operated at a crystal frequency of 12 MHz and TMOD = 40H, Timer 1 is operating at a frequency of,
   (a) 12 MHz (b) 1 MHz (c) 921.6 kHz (d) external signal
3. For a given crystal frequency, minimum time delay generated by Timer 0 is in,
   (a) Mode 0 (b) Mode 1 (c) Mode 2 (d) all of the above
4. Which of the following timer register is bit-addressable?
   (a) TL0 (b) TH1 (c) TCON (d) TMOD
5. Which of the following instructions will load the value 50H into the low byte of Timer 0?
   (a) MOV TH0, #50H (b) MOV TH0, 50H (c) MOV TL0, #50H (d) MOV TL0, 50H

6. The 8051 has _____ 16-bit counter/timers.
   (a) 1 (b) 2 (c) 3 (d) 4
7. The timers in Mode 0 overflow when the register reaches,
   (a) 1FFF (b) FFFF (c) FF (d) none of the above
8. The timers in Mode 1 overflow when the register reaches,
   (a) 1FFF (b) FFFF (c) FF (d) none of the above
9. An alternate function of port pin P 3.4 in the 8051 is,
   (a) Timer 0 input (b) Timer 1 input (c) Interrupt 0 input (d) Interrupt 1 input
10. Both registers TL0 and TL1 are needed to use Timer 0.
    (a) True (b) False
11. If TMOD = 0x80,
    (a) start/stop operation of Timer 1 is controlled by software.
    (b) start/stop operation of Timer 1 is controlled by hardware.
    (c) start/stop operation of Timer 0 is controlled by software.
    (d) start/stop operation of Timer 0 is controlled by hardware.
12. TH0, TH1 SFRs has the addresses,
    (a) 0x8C, 0x8D (b) 0xAD, 0xAB (c) 0x8D, 0x8B (d) 0x8D, 0x8C
13. The instructions MOV TMOD, #30H will
    (a) configure Timer 1 as a timer in Mode 2 (b) configure Timer 1 as a timer in Mode 3
    (c) configure Timer 0 as a timer in Mode 2 (d) configure Timer 0 as a timer in Mode 2
14. When TCON.6 = 1 and TCON.4 = 0,
    (a) Timer 1 is running, Timer 0 is stopped (b) Timer 0 is running, Timer 1 is stopped
    (c) both Timer 0 and 1are running (d) both Timer 0 and 1are stopped
15. Timer 0 in Mode 3 uses,
    (a) only TH1 register (b) only TH0 register (c) only TL0 register (d) TL0 and TH0 as a single register

Answers to Objective Questions

1. (b)	2. (d)	3. (d)	4. (c)	5. (c)	6. (b)	7. (a)	8. (b)
9. (a)	10. (b)	11. (b), (c)	12. (a)	13. (b)	14. (a)	15. (c)	

REVIEW QUESTIONS WITH ANSWERS

1. What are the uses of timers of the 8051?

A. The uses of timers are
- To measure the time: calculate the amount of time elapsed between the events
- Counting the events or to measure unknown frequency of a signal
- Generating baud rates for the serial port

2. List SFRs used to control timer activities.

A. TMOD, TCON, TL0, TH0 (TL1,TH1 for Timer 1)

3. How can a timer be configured as a counter?

A. By setting C/$\overline{T}$ = 1 in TMOD register

4. What is the maximum overflow rate for Timer Mode 1?

A. Crystal frequency/ 12 x 65536.

5. Why is 1 added to the count loaded in the timer register?

A. To compensate for one extra clock cycle that is required to raise timer overflow flag, i.e. TF is raised after timer value is changed from FFFFH to 0000H.

6. What is the key feature of Timer Mode 2?

A. It has to be initialized only once even for repetitive use, i.e. initialization overhead is the minimum for repetitive use.

7. What value is preferred to be loaded into the timer register when used as a counter?

A. 0000H.

8. How can start/stop of a timer be controlled externally?

A. Setting the bit Gate = 1 in TMOD register.

9. What is the source of the clock when the timer is configured as a counter?

A. External pulse on pin T0 (P3.4, pin 14) for Timer 0 and pin T1 (P3.5, pin 15) for Timer 1.

10. What is the difference between timer and counter mode of operation?

A. The source of the clock. For a timer, it is an internal oscillator, and for counter, it is the external pulses on timer input pins.

11. Can we generate time delays without using timers? If yes, how?

A. Yes. By using software delay routines. These routines basically waste the microcontroller time.

12. Can we program the original 8051 to count down?

A. No.

13. Do timers work in different modes when configured as counters?

A. Yes.

14. Write instructions to stop Timer 0.

A. CLR TR0 or CLR TCON.4

15. Do we need to set timer run bit when the timer is controlled by external signal using Gate bit?

A. Yes.

16. True or False. "TMOD is a bit-addressable register"

A. False.

17. What is the advantage of having timer module in the microcontroller chips?

A. They free the microcontrollers from time-keeping activities or at least reduce the burden, which would otherwise have kept microcontrollers busy. This allows the microcontrollers to perform other activities.

EXERCISE

1. What is the advantage of having TCON register as a bit addressable register?
2. Explain the term timer *overflow*.
3. Which SFRs are used for timers/counter operation?
4. What is the use of the TR1 bit in TCON register?
5. What is the use of the TF1 bit in TCON register?
6. What is the use of the C/T bit in TMOD register?
7. Which pins are used to work as counter inputs?
8. What is the minimum amount of delay that can be generated by Timer Mode 0, 1 and 2 if crystal frequency is 12 MHz?
9. How can very large time delays be generated using the timers?
10. How can very large value be counted using the counter?
11. How can timers be used to measure the unknown frequency?
12. Discuss the procedure to find count to be loaded into timer registers to generate a required delay.
13. List actions taken by the 8051 when TF flag is raised.
14. What are the different ways to start/stop timers?
15. Explain using a suitable example how the timing errors because of initialization overhead can be minimized.
16. Find the value to be loaded into TH register for Timer Mode 2 to generate a delay of 50 μs. Assume crystal frequency = 12 MHz.
17. Write equivalent instruction for the following:
    (i) SETB TCON.4 (ii) CLR TCON.7
18. Justify true/ false " Timer 0 and 1 both have their own TF flag".
19. Find rollover values for each timer mode.
20. For the crystal frequency of 11.0592 MHz, find highest as well as lowest frequency of square wave that can be generated using Timer (i) Mode 1, and (ii) Mode 2.
21. What are the applications of Timer Mode 3?
22. Design an 8051 system that can be used to measure motor speed in rpm. [*Hint:* Configure one timer as a counter and the other as a timer to generate a delay of one minute.]
23. What is the maximum frequency that can be measured using timers?

15

Serial Communications

Objectives

- Discuss the need for serial communication
- Compare synchronous and asynchronous serial communications
- Discuss the frame structure of asynchronous serial data transmission
- Explain the data-transfer rate
- Discuss the significance of RS232 serial data transmission standard
- Interface RS232 devices with the 8051
- Discuss the need of UART and its features
- Describe the operating modes of UART and the associated registers
- Illustrate the concept of multiprocessor communications
- Develop the programs to transmit and receive data serially
- Discuss the uses and features of second serial port in the DS89C4X0
- Develop the programs to transmit and receive data for second serial port of DS89C4X0

Key Terms

- Asynchronous Communication
- Baud Rate
- DCE
- DTE
- Frame
- Full-Duplex
- Half-Duplex
- Handshaking
- Mark/Space
- MAX 232/233
- Multiprocessor Communication
- Parity Bit
- PCON
- RS 232
- SBUF
- SCON
- Simplex
- SMOD
- Start Bit
- Stop Bit
- Synchronous Communication
- TI/RI
- Transmit/Receive
- Transmit/Receive Buffer
- TXD/RXD
- UART
- UART Modes: 0, 1, 2, and 3

15.1 NEED FOR THE SERIAL COMMUNICATION

Within a microcontroller, the data is transmitted in parallel form, i.e. more than one bit at a time, typically 8 or 16 bits (or even more bits depending upon size of the data bus). The parallel transmission provides faster data transfer. However, to transmit data over long distance, the parallel data transfer is not preferred because it requires many wires (one wire per bit) which results in higher cost of a system. Therefore, for long distances, data to be transmitted is usually converted from the parallel to serial form, so that it can be transmitted through a single wire (or pair of wires) one bit at a time. The data received in serial form will again be converted back into the parallel form so that it can be easily processed and transferred within a microcontroller system.

There are three types of serial communication systems: simplex, half-duplex and full-duplex.

Simplex A simplex system can transmit data only in one direction, i.e. it is a one-way communication. Radio and television broadcasting, sending data from keyboard to the CPU, sending data to CRT for display are some examples of simplex systems.

Half-duplex The data transfer can take place in both directions between the two systems, but only in one direction at a time. Two-way radio system is a half-duplex system.

Full-duplex System The communication can take place in both the directions simultaneously between two systems; however, separate conductor wires are required for transmission and reception. The telephone line is an example of a full-duplex system. These systems are illustrated in Figure 15.1. TXD in the figure indicate Transmitter Output (serial data output) and RXD is Receiver Input (serial data input). Driver circuits are used to maintain required signal strength through a long-distance serial link (cable).

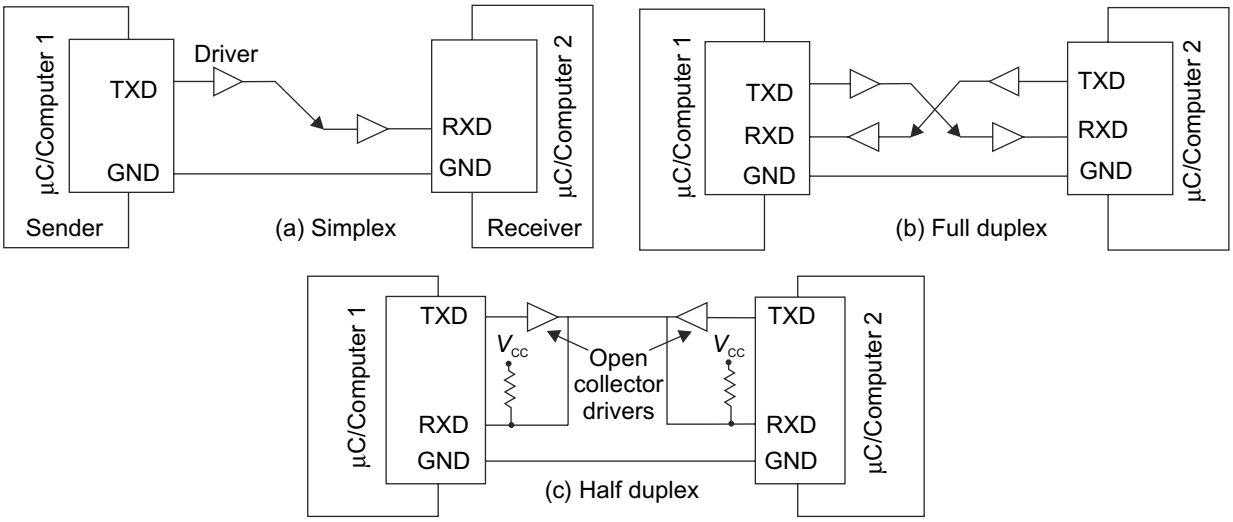


Fig. 15.1 *Simplex, half-duplex and full-duplex systems*

In a simplex system, a transmitter device (microcontroller/microprocessor based system) will transmit serial data through TXD output and a receiver device will receive data through RXD input. Note that in a simplex system, transmitter can only transmit and receiver can only receive the data. In a half-duplex system, both the devices have transmitting and receiving capability, but only one can be utilized at a time, i.e. when System 1 [µC/computer 1 in Figure 15.1(c)] is transmitting the data, System 2 (µC/computer 2) will receive data and vice versa. Note that there is only one link (channel or transmission medium) between the transmitter and receiver. The full-duplex system can perform both the operations simultaneously because they have separate links for transmission and reception.

15.2 SYNCHRONOUS AND ASYNCHRONOUS SERIAL COMMUNICATIONS

Serial communication can be synchronous or asynchronous. In synchronous communication, a common clock is used for both transmitter and receiver. The clock is transmitted separately using a dedicated link (wire), since a common clock is used to maintain synchronization; the messages in this mode are usually very long.

In asynchronous communication, the synchronization is achieved only at the beginning of the transmission and then it is maintained (by transmitter and receiver) till the end of the message. Since there is no synchronization throughout the transmission, the amount of data transmitted is very small, usually one byte (7 or 8 bits). For each new message, the transmitter and receiver are resynchronized at the beginning. These systems are illustrated in Figure 15.2.

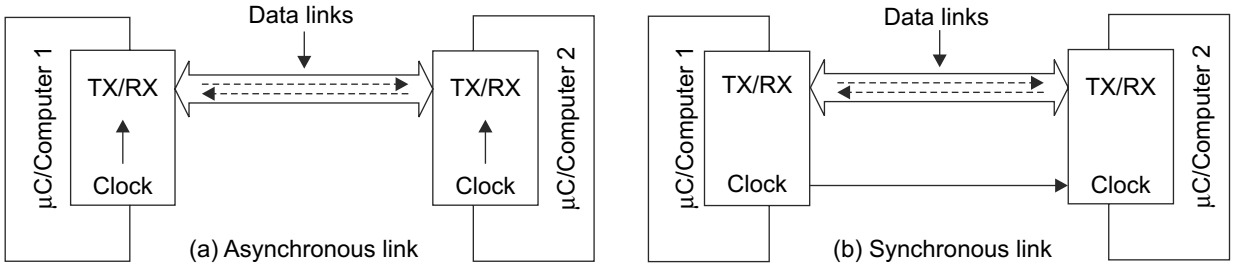


Fig. 15.2 *Asynchronous and synchronous systems*

Note that a dedicated link is used in synchronous communication to achieve synchronization between the transmitter and receiver, the penalty for this is that the system cost will increase. The synchronization in asynchronous systems is achieved through data links, see arrow above clock in Figure 15.2 (a).

15.2.1 Format of Asynchronous Serial Data Frame

A *frame* is a complete and non-divisible group of bits which can be transmitted at a time. It contains actual information (e.g. data) as well as extra bits to maintain synchronization and integrity of the data (start bit, error checking bits, and stop bits). These extra bits are usually referred as *overhead bits*. It has one start bit, data bits, parity bits and stop bits. The device that handles the serial communication can be programmed for data that is 7 or 8 bits wide, no or even or odd parity and number of stop bits as 1 or 2. Figure 15.3 shows the bit-format of a frame for transmitting asynchronous serial data.

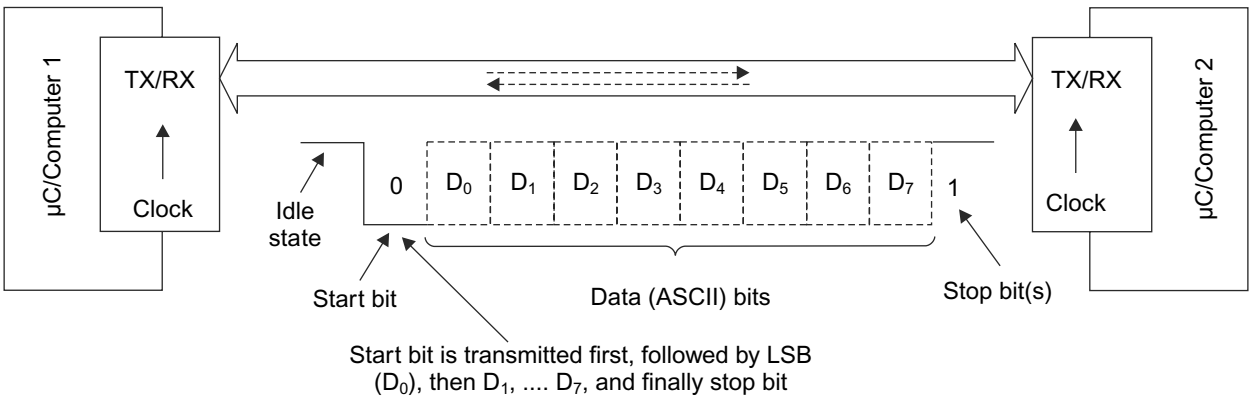


Fig. 15.3 *Frame structure of asynchronous serial data*

As can be seen from Figure 15.3, asynchronous transmission uses byte-oriented format. When there is no data transfer, i.e. when the link is idle, the high level (1) is present on the link, which is referred as *mark* (and the low is referred as *space*). The transmission is started by sending START bit first and the START bit is always low, i.e. the transmitter will drive the serial link to low-logic level to indicate start of the data transfer. This high to low level transition will indicate the receiver to get ready for the data reception. After the time period of a start bit, a data byte is transmitted, one bit at a time (LSB first). Thereafter, STOP bit(s) is sent to indicate the end of transmission of one byte; the STOP bits are always

high. In older systems, the ASCII characters were of 7 bits, now 8 character bits are used because of extended ASCII. In the older systems, two stop bits were used, while in the newer systems, only one stop bit is used. The parity bit may be odd or even. In case of even parity, the number of data bits, including the parity bit has even number of 1s. Also note that transmitter and receiver operates with two independent clocks and they are configured to operate at equal baud rate (see next section). In conclusion, the synchronization is established by START bit at the beginning and STOP bit at the end of each byte.

15.2.2 Rate of Data Transfer

The term *Baud rate* is defined as the rate at which data is being transferred serially, i.e. rate at which signal transitions occur (or frequency of signal transitions).

For example, if the signal makes a transition every 5 ms the baud rate is (1/5 ms) = 200 Baud.

Rate of data transfer is also measured in bits/s. It is the number of bits transmitted per second. The terms *Baud rate* and *bits/s* are not always the same. For example, if only one bit is coded per signal transition then bits/second and Baud rate are same (like BPSK) and when more than one bit is coded per signal transition then both are different. Assume 2 data bits are encoded in a single signal transition (like QPSK) and signal is changing every 5 ms, so Baud rate is 200 Baud as explained above but bits/s is 2 × 200 = 400 bits/s. Hence, the relation between bits/s and Baud rate is given as,

Bits/s = Number of data bits encoded per signal transition × Baud rate

Baud rates of 300, 600, 1200, 2400, 4800, 9600, 19200 are used most commonly.

15.3 RS 232: SERIAL DATA TRANSMISSION STANDARD

Since serial transmission of data is more efficient over long distances, special devices were developed so that the computers could use telephone lines to communicate with distant computers. These devices are referred as Data Communication Equipment or DCE; Modem is an example of DCE. The devices that are sending or receiving digital data are called Data Terminal Equipment (DTE). Computers are DTEs.

To maintain compatibility among all the DCEs manufactured by different manufactures and to establish handshake (co-ordination) signal between DTE and DCE, an interfacing standard called RS232 was developed by the Electronics Industries Association (EIA) in 1960. RS stands for recommended standard, and RS232C is the most popular standard. This standard describes the function of 25 signals as shown in Table 15.1 and handshake mechanism for the serial data transfer. It also defines the signal levels (voltage levels for 0 and 1), impedance levels, rise and fall times, and all other physical-level parameters for these signals. A typical serial link using RS232 is shown in Figure 15.4. The operation and handshaking mechanism for this link will be discussed in the next section.

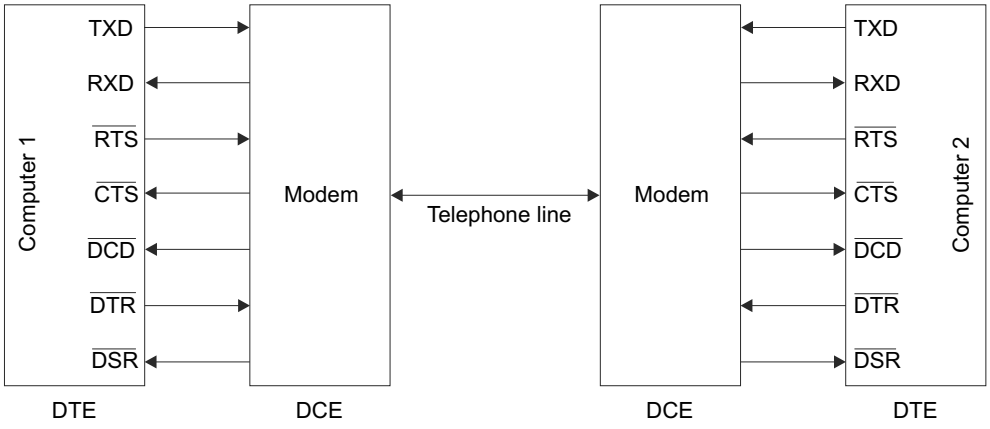


Fig. 15.4 *Serial link using RS232*

15.3.1 Hand-shaking Process between DTE and DCE

After the power is turned ON, DTE asserts the Data Terminal Ready ($\overline{\text{DTR}}$) signal to inform the DEC (modem) that it is ready; see Figure 15.4. When the modem is ready, it asserts the Data Set Ready ($\overline{\text{DSR}}$) signal to the DTE. Once $\overline{\text{DSR}}$

signal has been received, the DTE makes a request to use the data channel by asserting $\overline{\text{RTS}}$ signal to start transmission. Then, the modem at the other end (receiver end) is dialed. This modem (usually in answer mode) replies by sending a signal at a carrier frequency of 2255 Hz. When the modem at the sending terminal receives this signal, it sends Data Carrier Detect ($\overline{\text{DCD}}$) to the DTE, thereafter the modem sends Clear To Send ($\overline{\text{CTS}}$) showing that the channel is ready for transmission. Immediately after receiving $\overline{\text{CTS}}$ signal, the DTE on the transmitter side sends serial data on its TXD output. For reception of data, a similar handshaking process is carried out.

Table 15.1 *RS232 Signals (DB25)*

Pin	Description	Pin	Description
1	Protective ground	14	Secondary transmitted data
2	Transmitted Data (TXD)	15	Transmitted signal element timing
3	Received Data (RXD)	16	Secondary receive data
4	Request to Send ($\overline{\text{RTS}}$)	17	Receive signal element timing
5	Clear to Send ($\overline{\text{CTS}}$)	18	Unassigned
6	Data Set Ready ($\overline{\text{DSR}}$)	19	Secondary receive data
7	Signal Ground (GND)	20	Data Terminal Ready ($\overline{\text{DTR}}$)
8	Data Carrier Detect ($\overline{\text{DCD}}$)	21	Signal Quality Detector
9/10	Reserved for data testing	22	Ring Indicator (RI)
11	Unassigned	23	Data signal rate select
12	Secondary data carrier detect	24	Transmit signal element timing
13	Secondary clear to send	25	Unassigned

All the 25 signals are not used in PC-based serial communication; therefore, the nine signal version (DB 9) was developed. These nine signals are shown in Table 15.2.

RS232C is the most widely used serial I/O interfacing standard. This is used in PCs and many other equipment.

Table 15.2 *RS232 signals for DB9 connector*

Pin	Description
1	Data Carrier Detect ($\overline{\text{DCD}}$)
2	Received Data (RXD)
3	Transmitted Data (TXD)
4	Data Terminal Ready ($\overline{\text{DTR}}$)
5	Signal Ground (GND)
6	Data Set Ready ($\overline{\text{DSR}}$)
7	Request To Send ($\overline{\text{RTS}}$)
8	Clear To Send ($\overline{\text{CTS}}$)
9	Ring Indicator (RI)

THINK BOX 15.1

Which other serial data transmission standards exist? What are their features?
RS422 and RS485 are the other popular serial data transmission standards. They are designed for long distance (up to 4000 ft.) and high speed (100 Kbps for long distance and 1 Mbps for short distance). Since they transmit a signal in the form of differential voltage, they are more immune to noise.

THINK BOX 15.2

What are the types of connectors used for RS232 interface?
DB9 (9-pin D type connector), DB25 (25 pin D type connector) and RJ 45 (8 wire, registered jack-45).

15.3.2 RS232 to TTL Interfacing

Since RS232 standard was developed long before the development of the TTL logic family, its operating voltage level is not compatible with TTL voltage levels. The voltage levels for TTL are 0 V (0 to 0.8 V approx) for low logic and 5 V (2 to 5 V approx) for high logic. The voltage levels of all RS232C signals are as follows. A logic high (or mark) is voltage between –3 V to –15 V under loaded condition (and –25 V at no load). A logic low (or space) is voltage between +3 V to +15 V under loaded condition (+25 V at no load). The voltage ±12 V are commonly used. RS232 is based on negative logic. Typical RS232 voltages are,

High = Mark = – 12 V

Low = Space = +12 V

Because of the voltage differences, if we want to connect a TTL compatible microcontroller to an RS232 system, we must use voltage converters to convert TTL voltage levels to the RS232 voltage levels and vice versa. MAX232 and MAX233 chips are the most popular voltage converts and are also referred as line drivers. Figure 15.5 shows the connection between RS232C DB9 connector (PC side) and the 8051 using MAX232 chip.

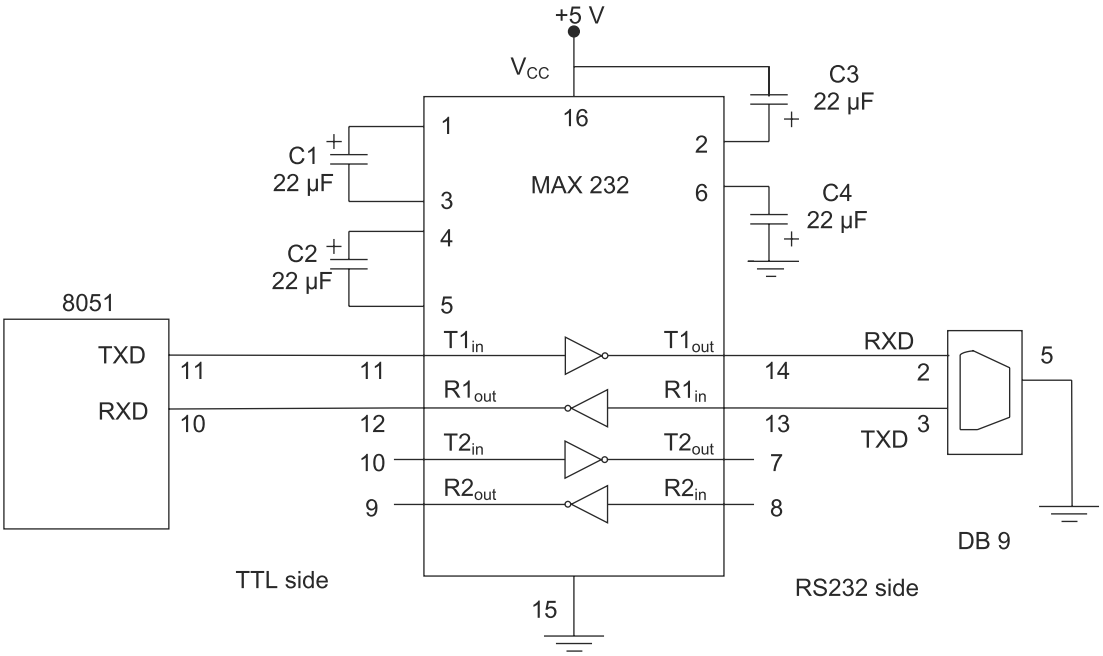


Fig. 15.5 *Interfacing 8051 with RS232 device*

The MAX 232 chip is used as a line driver (voltage converter). It uses +5 V power supply for its operation, which is the same as supply voltage of the 8051, therefore, single supply is required for both the 8051 and MAX 232 chip. (Internally, MAX 232 contains +5 V to +10 V voltage doubler and +10 V to –10 V inverter. It requires four capacitors of typical value 22 µF for this conversion). We know that the 8051 has two pins, TXD (P3.1) and RXD (P3.0), for transmission and reception of serial data, respectively. These pins are TTL compatible; therefore, they require the line drivers to make them RS232C compatible. The MAX 232 has two sets of line drivers for transmitting (T1 and T2) and receiving data (R1 and R2). Only one set is required for one serial communication system. Note that $T1_{in}$ is connected with the TXD pin of the 8051, while $T1_{out}$ is connected with the RXD pin of RS232 side connector, i.e. TXD signal of 8051 is connected (of course after voltage conversion) to RXD signal of RS232 device (PC). Similarly, $R1_{in}$ is connected to the TXD pin of RS232 side connector while $R1_{out}$ is connected with the RXD pin of the microcontroller.

MAX233 is another popular driver chip which will not require external capacitors to be connected, thus, it will save board space, but, it is costlier than MAX232 chip.

15.4 | UART

To interface a microcontroller with the serial data lines, the serial data must be converted in to parallel and vice versa. Serial-in parallel-out and parallel-in serial-out shift registers can be used to perform above conversions. Also, handshaking circuitry is required to synchronize the transmitter and receiver. Many devices are available which contain circuitry required to handle serial communications. Intel 8251 is capable to handle both synchronous as well as asynchronous serial communications. The 8251 is more popularly known as *USART* (*Universal Synchronous Asynchronous Receiver Transmitter*). National Semiconductor INS 8250 is a device capable of handling only asynchronous systems and is commonly referred as UART.

The 8051 has a built-in UART; thus, the 8051 chip is capable of handling asynchronous transmission and reception of serial data. The 8051 has two pins TXD (P3.1) and RXD (P3.0) for transmission and reception of serial data respectively. The UART hardware is more commonly referred as serial port. A typical illustration of serial connection between two microcontrollers is shown in Figure 15.6.

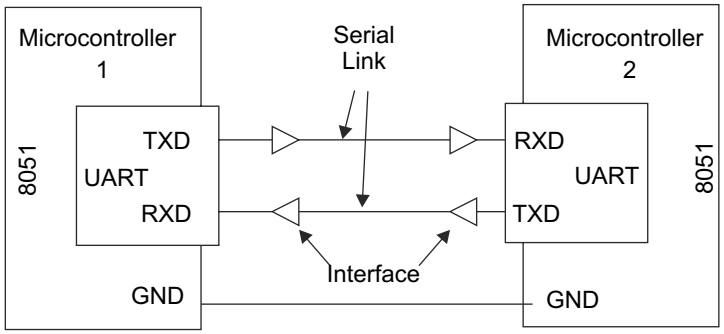


Fig. 15.6 *Serial communication between two microcontrollers*

The serial-data-output pin (TXD–transmit) of one microcontroller is connected to serial-data-input pin (RXD–receive) of the other microcontroller and vice versa. The square box named UART in Figure 15.6 indicates that 8051 has on-chip (within chip) dedicated hardware to handle asynchronous transmission and reception activities. The serial link is a conducting wire and interface circuits are driver circuits to allow long-distance communication. Note that a short-distance serial communication between two 8051 chips do not require any driver circuit.

15.4.1 UART Features

The 8051 UART has the following features:

- Supports full-duplex serial communication, i.e. transmission and reception of data is simultaneously possible
- Transmit and receive buffer registers along with transmitting and receiving shift registers
- Logic for generating the timing signal, i.e. clock
- Status bit showing that data byte has been sent
- Status bit to indicate that data byte has been received

Serial port of the 8051 is controlled by two registers: SBUF and SCON.

15.4.2 SBUF (Serial Data Buffer) Register—One Name–Two Registers

It is an 8-bit register used for the serial data transmission and reception. A data byte must be written to SBUF for transmission through the TXD pin. Same way, it also holds a byte of data received from RXD pin. Though there is only one name (and, therefore, address) given to SBUF register, there are two separate physical registers—*Transmit Buffer* and *Receive Buffer*. These registers are differentiated by a microcontroller as per the operation being performed with them. The operation and connection of these two registers with internal data bus is illustrated in Figure 15.7.

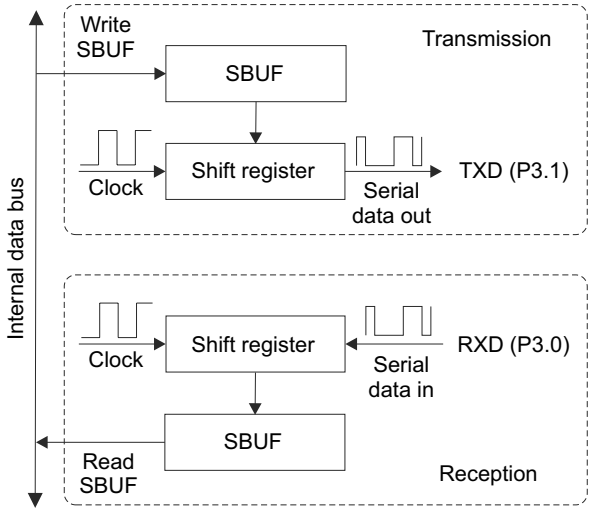


Fig. 15.7 *Two physically separate SBUF registers*

When SBUF is read, the receive buffer is accessed and while transmitting, transmit buffer is accessed. For example, instruction MOV A, SBUF will read received data from receive SBUF register (receiver buffer will receive data from RXD pin), while the instruction MOV SBUF, A will write data to transmit SBUF register (transmitter buffer). The data from the transmitter buffer will be transmitted through the TXD pin.

SCON—Serial Control Register

SCON is the control and status register programmed to configure bits contained in one serial "word", Baud rate and synchronization clock source. It also contains status bits that indicate whether the data is transmitted completely, and any new data is received.

15.4.3 Serial Port Control (SCON) Register

Bit-assignment and description of SCON register is given in Table 15.3.

Table 15.3 *SCON register*

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
MSB							LSB

Bit	Symbol	Description
7	SM0	Serial Port Mode [see table below *]
6	SM1	
5	SM2	Enables multiprocessor I/O in Modes 2 and 3. When set to 1, an interrupt is generated if bit 9 of received data is 1, no interrupt is generated if bit 9 is 0. If Set to 1 for Mode 1, no interrupt will be generated unless a valid stop bit is received. Clear to 0 for Mode 0.
4	REN	Receive Enable if REN = 1
3	TB8	9^{th} data bit to send in 9-bit mode in modes 2 and 3
2	RB8	9^{th} data bit received in modes 2 and 3. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
1	TI	Transmit Interrupt flag (Transmitter Empty Interrupt Flag)—sending finished. Set by hardware at the end of the 8^{th} bit time in Mode 0, or at the beginning of the stop bit in the other modes. It's a signal to the microcontroller that the line is available to transmit a new byte. Must be cleared by software
0	RI	Receive Interrupt Flag–new byte received. Set by hardware at the end of the 8^{th} bit time in Mode 0, or halfway through the stop bit time in the other modes. It signals that a byte is received and should be read quickly prior to being replaced by the new data. Must be cleared by software

Serial Port Mode is selected by the SM0 and SM1 bits:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	8-bit shift register	1/12 the crystal frequency
0	1	1	8-bit UART	Determined by Timer 1
1	0	2	9-bit UART	1/32 the crystal frequency (1/64 the crystal frequency)
1	1	3	9-bit UART	Determined by Timer 1

15.5 MODES OF OPERATION

UART can be configured to operate in one of the four operating modes selected by SM0 and SM1 bits in the SCON register as discussed in the previous section.

15.5.1 Mode 0–8 bit Shift Register Mode

In Mode 0, data are transmitted and received through the RXD pin, and the TXD pin outputs the shift clock and this is used to synchronize data transmission/reception. On transmit, the least significant bit (LSB bit) is sent/received first. The operation of serial port in Mode 0 is shown in Figure 15.8.

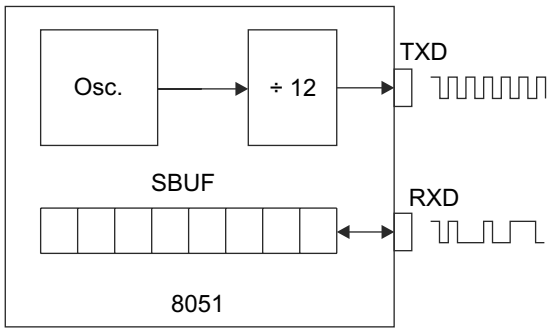


Fig. 15.8 *UART Mode 0 operation*

Transmission Transmission begins by any instruction that uses SBUF as a destination register regardless of the state of TI flag. When the transmission is complete (all 8 bits have been sent), the TI bit of the SCON is set automatically.

Reception Data reception (at RXD pin) begins when the following two conditions are met: bit REN = 1 and RI = 0. The condition RI = 0 is exceptional for Mode 0, all the other modes are enabled to receive when REN = 1 irrespective of RI, because in Mode 0 only, we can control when the reception can occur. When all the 8 bits are received, the RI bit of the SCON register is automatically set to show that one byte is received and also prevents reception of any character until cleared by the program. Timing of Mode 0 is shown in Figure 15.9.

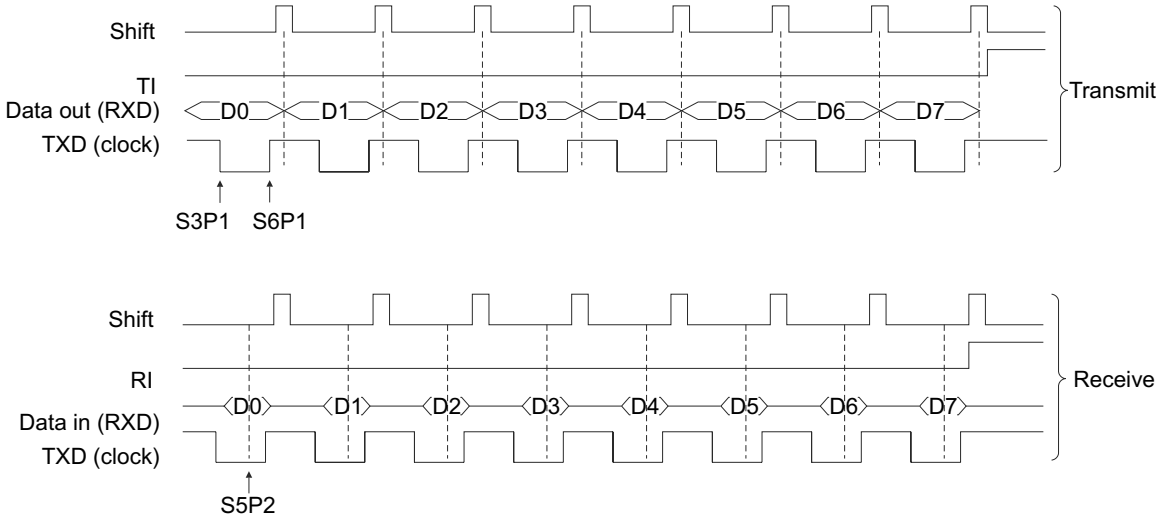


Fig. 15.9 *UART Mode 0 timing*

During the transmission, the data is shifted out from RXD pin, the data changes on trailing edge of the S6P2 state and TI flag is set after transmission of 8 bits. During the reception, the data is sampled on trailing edge of the S5P2 state and shifted into receiver buffer (SBUF) on the leading edge of the TXD clock output (shift clock).

Because there is no overhead (no START and STOP bits or any other bits), this mode is used for high-speed and short-distance transmission.

THINK BOX 15.3

Why are the terms TXD and RXD misleading in UART Mode 0?

In UART Mode 0, the RXD pin is used for both transmission as well as reception (input and output) while the TXD pin serves as a clock.

Example 15.1

Write a program to transmit a data byte '55H' continuously using serial port in Mode 0.

Solution:

For the desired operation, we need to configure serial port into Mode 0 and write the data byte into SBUF register. Once we write the data byte into SBUF, the data transmission is started. Upon completion of transmission, the TI flag will be set, therefore, we need to wait until the TI flag is set and then we need to repeat the operation.

```
        ORG 0000H
        MOV SCON, #00H      // configure serial port in Mode 0
REPEAT: MOV SBUF, #55H      // write data byte in the SBUF for transmission
WAIT:   JNB TI, WAIT        // wait until TI flag is set
        CLR TI              // clear TI before sending the next data byte
        SJMP REPEAT         // repeat the operation
        END
```

Note that data in Mode 0 is transmitted through the RXD pin.

Example 15.2

Write the program of Example 15.1 in the C language.

Solution:

The equivalent C program is given below:

```
#include<reg51.h>
```

```
void main (void)
        {
        SCON = 0x00;                    // configure serial port in Mode 0

              while (1)                 // repeat the operation forever
                    {
                    SBUF = 0x55;        // write data byte in SBUF for transmission
                    while (TI==0);      // wait until data byte is transmitted
                    TI = 0;             // clear TI before sending next byte
                    }
        }
```

Example 15.3

Write a program to receive a data byte using serial port in Mode 0. Send the received byte to Port 1.

Solution:

```
        ORG 0000H
        MOV SCON, #00H      // configure serial port in Mode 0,
        ORL SCON, # 10H     // enable reception
WAIT:   JNB RI, WAIT        // wait until data byte is received
        MOV A, SBUF         // read SBUF
        MOV P1, A           // send received byte on Port 1
        CLR RI              // clear RI before receiving next data byte
HERE:   SJMP HERE
        END
```

Example 15.4

Write the program of Example 15.3 in the C language

Solution:

The equivalent C program is given below:

```
#include<reg51.h>

void main (void)
              {
              SCON = 0x00;          // configure serial port in Mode 0,
              SCON |= 0x10          // enable reception
              while (RI==0);        // wait until data byte is received
              P1=SBUF;              // read SBUF and send byte to P1
              RI = 0;               // clear RI before receiving next byte
              while (1);            // wait indefinitely
              }
```

Note that the reception is enabled and started only when REN = 1 and RI = 0.

Mode 0 may be used to get fast I/O lines (for fast data collection) and the control of multi-point systems using low-cost simple TTL or CMOS shift registers. A block diagram of typical multi-point data collection and control systems is shown in Figure 15.10.

Figure 15.10 illustrates that signals from many sources can be monitored by connecting them to a parallel-to-serial converter and sent to a microcontroller. Upon reception of these data, the microcontroller can take some decisions and issue control signals in serial form and is then given to serial-to-parallel converters. The output of this converter may be used to control different systems. The parallel-to-serial and serial-to-parallel converters are driven by a clock generated at the TXD pin. These converters are enabled one at a time by microcontroller port pins. This is a classical example of I/O port expansion using serial port because using only a few pins of a microcontroller, any number of input and output signals can be connected.

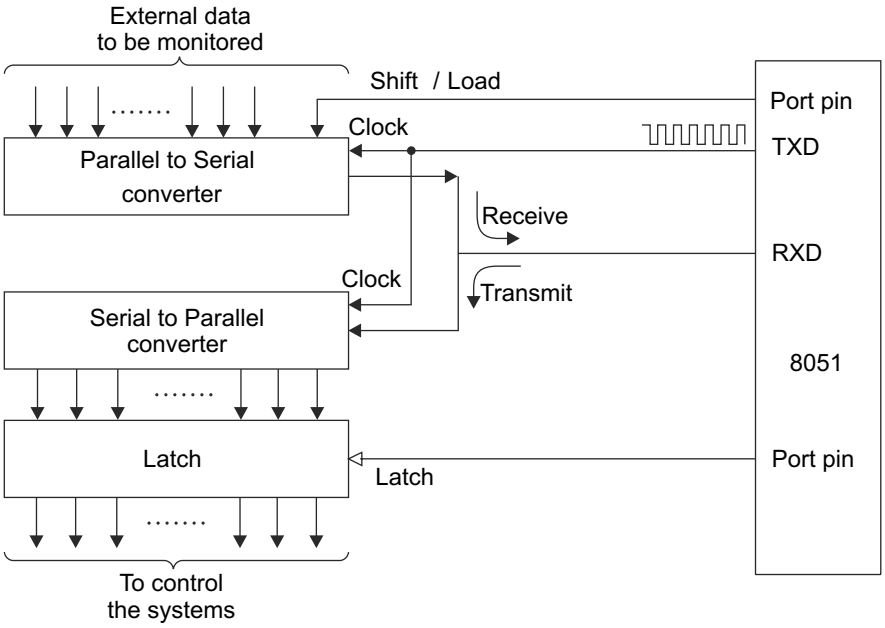


Fig. 15.10 *Simplified diagram of multipoint data collection and control system*

Remember that operation of Mode 0 is not asynchronous. Simultaneous reception and transmission in this mode is not possible. Since the clock rate is very high and transmission (or reception) is completed very quickly, interrupts are not preferred for this mode. The TI and RI flags are usually monitored through polling, i.e. wait for the RI or the TI flag to be set.

Baud Rate for Mode 0

Mode 0 has a fixed baud rate which is 1/12 of the oscillator frequency as shown in Figure 15.11. To run the serial port in this mode, none of the timer/counters need to be set up. Only the SCON register needs to be defined.

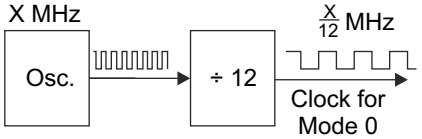


Fig. 15.11 *Clock source for Mode 0*

$$\text{Baud rate} = \frac{\text{Oscillator frequency}}{12}$$

15.5.2 Mode 1—Standard 8-bit UART Mode

UART is designed mainly for this mode and frame format of this mode is compatible with COM port of PCs. This mode transmits 10 bits through TXD pin and receives through RXD pin as follows: a START bit (always 0), 8 data bits (LSB first) and a STOP bit (always 1). The programmer can set its transmission/reception rate using Timer 1. The process of data transmission and reception is illustrated in Figure 15.12.

Transmission

Data transmission begins by writing data to the SBUF register. The START and STOP bits are added by hardware to form a 10-bit frame (Figure 15.12). Then, the 10-bit parallel-to-serial conversion is performed and one bit (LSB first) at a time is transmitted through the TXD pin. Once the complete frame is transmitted, the TI flag is set automatically by the serial port hardware to indicate the end of data transmission. We need to monitor the TI flag to conform that SBUF register is

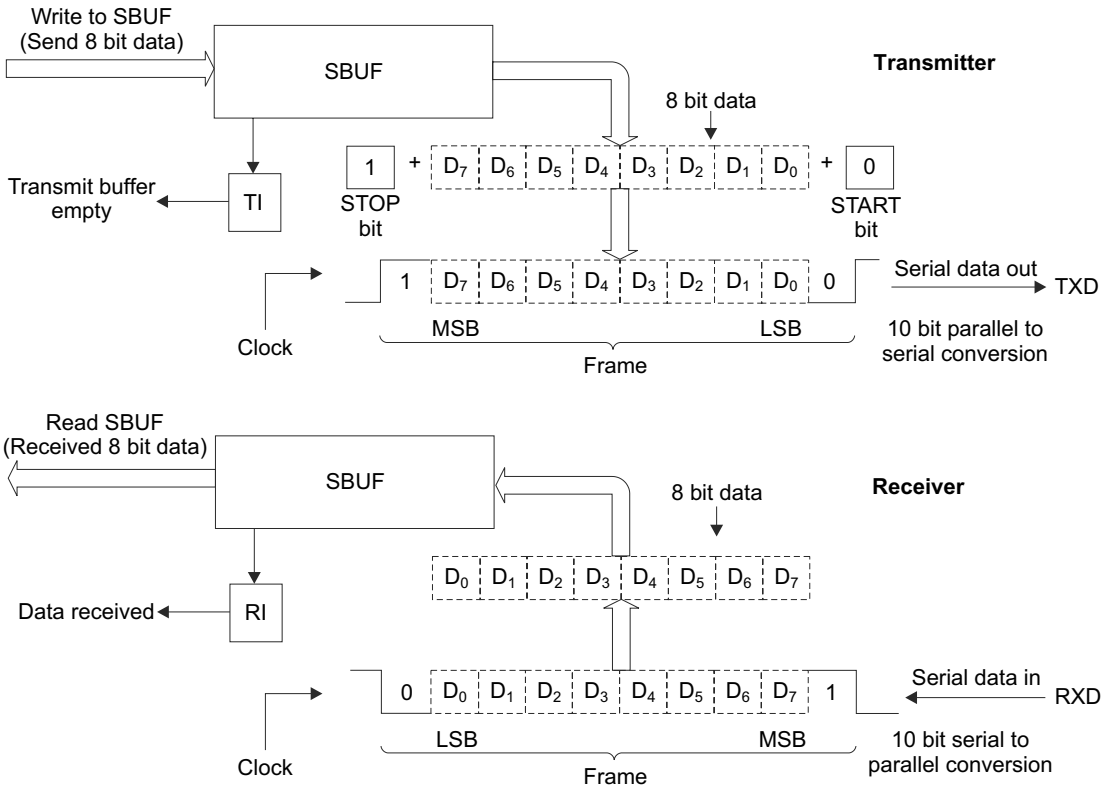


Fig. 15.12 *UART Mode 1 operation*

not overloaded. If the TI flag is set, it implies that last character transmission is completed and now SBUF is empty and the new byte can be written to it to start the next transmission. If a new byte is written to SBUF before TI is raised, the untransmitted part of the previous byte will be lost.

It should be noted that the microcontroller sets the TI flag when it completes byte transfer, whereas it must be cleared by the programmer after the next byte is loaded into SBUF.

Reception

The data reception begins when REN = 1 and high-to-low transition (start bit) is detected on the RXD pin. The received byte is loaded into SBUF register (the START and STOP bits are separated by UART hardware once complete frame is received) and stop bit into RB8 (SCON bit 2) only if the following two conditions are met.

(i) RI = 0, showing that previous data byte is read by the program

(ii) Either SM2 = 0 or stop bit = 1. Normally SM2 = 0 and character will be accepted irrespective of the status of stop bit. A program may check RB8 to ensure that the stop bit is correct, if required.

If these two conditions are not met, the received character is ignored and RI is not set and the receiver circuit waits for the next start bit.

Baud Rate for Mode 1

Timer 1 is used to determine and generate Baud rate for Mode 1. Timer 1 is usually configured in Mode 2 as an auto-reload 8-bit timer. Let us first understand how Timer 1 is used as baud rate generator. We know that the 8051 divides crystal frequency internally by 12 to generate machine-cycle frequency. The unusual value of 11.0592 MHz is used as a crystal frequency to generate standard Baud rates. The reason for using this value will become clearer after the discussion that follows. When crystal (XTAL) frequency is 11.0592 MHz, the machine cycle frequency is 921.6 KHz. The 8051 UART circuitry divides this machine-cycle frequency further by 32*. This division will generate a signal of 28800 Hz. The value 28800 Hz can be divided by an integer to get standard baud rates, for example, 28800/3 = 9600, 28000/6 =

* Machine cycle frequency is divided by 32 when SMOD = 0, and divided by 16 when SMOD = 1, SMOD is a serial baud rate modify bit in PCON special function register, significance of SMOD = 1 is that it can be used to double the baud rates. It is further discussed in Example 15.6.

4800 and 28800/24 = 1200 etc. Thus, 28800 Hz signal is given as input clock to Timer 1 to set the desired standard baud rate. As a matter of fact, Timer 1 will divide 28800 Hz signal by an integer (based on the value loaded into its registers as explained next) to get desired baud rate.

This process is illustrated in Figure 15.13.

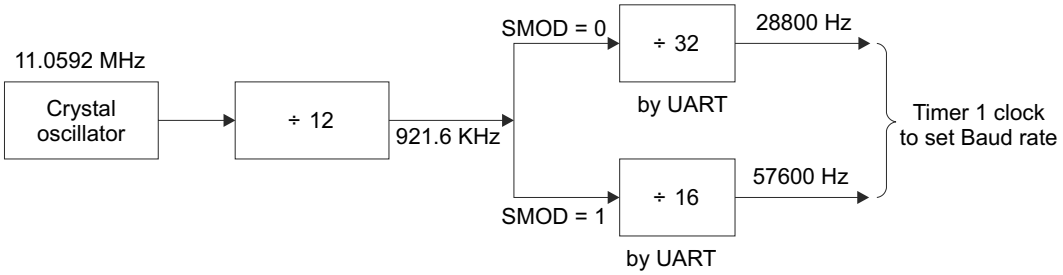


Fig. 15.13 *Timer clock source for Baud-rate generation*

As per the above discussion, Timer 1 clock input frequency is given as

$F_{T1CLK} = F_{OSC} \times 2^{SMOD} / (12 \times 32)$

Now this F_{T1CLK} is further divided by Timer 1 and amount of division is controlled by count loaded into its TH1 register. Hence, baud rate is given as,

$F_{BAUD} = F_{T1CLK} / (256 - TH1)$

$= F_{OSC} \times 2^{SMOD} / 12 \times 32 \times (256 - TH1)$

To get the desired baud rate, appropriate count has to be loaded in TH1 register. It is given from the above equation as,

$TH_1 = 256 - \{(F_{OSC} \times 2^{SMOD}) \div (12 \times 32 \times F_{BAUD})\}$

Example 15.5

What value should be loaded into the TH1 register to get a Baud rate of (i) 9600 bits per second, and (ii) 1200 bps for serial transmission? Assume crystal frequency to be 11.0592 MHz.

Solution:

The value of TH_1 can be found as,

$TH_1 = 256 - \{(F_{OSC} \times 2^{SMOD}) \div (12 \times 32 \times F_{BAUD})\}$

(i) For 9600 bps rate, assuming SMOD = 0

$TH_1 = 256 - \{(11.0592 \times 10^6 \times 1) \div (384 \times 9600)\}$

$= 253 = FDH$

(ii) For 1200 bps rate, assuming SMOD = 0

$TH_1 = 256 - \{(11.0592 \times 10^6 \times 1) \div (384 \times 1200)\}$

$= 232 = E8H$

Program

```
MOV TMOD, #20H      ; initialize Timer 1 in Mode 2
MOV SCON, #4CH      ; initialize serial Mode 1
MOV TH1, #0FDH      ; load count for 9600 bps
```

Example 15.6

Demonstrate the use of SMOD bit to double the Baud rate.

Solution:

Consider Example 15.5. The value FDH is required to be loaded into TH1 to generate baud rate of 9600 bps. For SMOD = 1, baud rate generated by loading the same value into TH1 will be given as,

$F_{BAUD} = F_{OSC} \times 2^{SMOD} / 12 \times 32 \times (256 - TH1)$

= 19200 Hz (bps*)

It can be seen that SMOD = 1 will generate double the Baud rate compared to SMOD = 0 when all other variables in the above equation are same.

* Here, Baud rate and bit rate is same because it is assumed that only one bit is coded per signal transition.

It should be noted that Timer 1 interrupt should be disabled when it is used to generate baud rates. The timer may be configured in either Mode 0, 1 or 2. Table 15.4 shows values to be loaded into the TH1 register for various baud rates for different values of crystal frequencies.

Table 15.4 *TH1 values for various baud rates*

BAUD RATE	F_{OSC} (MHZ)				SMOD BIT
	11.0592	12	16	20	
150	40 H	30 H			0
300	A0 H	98 H	75 H	52 H	0
600	D0 H	CC H	BB H	A9 H	0
1200	E8 H	E6 H	DE H	D5 H	0
2400	F4 H	F3 H	EF H	EA H	0
4800		F3 H	EF H		1
4800	FA H			F5 H	0
9600	FD H				0
9600				F5 H	1
19200	FD H				1

THINK BOX 15.4

How can we generate very low Baud rates for serial communications?

By using Timer 1 in 16-bit mode (Mode 1). But, in this case, there will be overhead instructions because the timer registers must be reinitialized after each overflow, this re-initialization must be done in ISR. The other method is to clock Timer 1 from low-frequency external source at pin T1 (timer as counter).

Software Development to Transmit and Receive Data Serially

The following steps must be taken for serial data transmission:

1. Configure Timer 1 in 8-bit auto-reload mode. (This is the most commonly used configuration.)
2. TH1 is loaded with appropriate value to set the required Baud rate.
3. Configure SCON register to set serial port in Mode 1. REN is set to 1 to enable serial data reception.
4. TR1 is set to start Timer 1 to generate clock at a desired Baud rate.
5. The byte to be transmitted is written to SBUF register.
6. For transmission, the TI flag is monitored to make sure that the byte has been transmitted completely. For reception, the RI flag is monitored to check that byte is received or not. When RI is raised, SBUF contains a received byte. It must be read from SBUF. This monitoring of RI and TI can be done either using polling method or interrupt method.
7. Once TI is set when a byte is transmitted (or RI for reception), it is cleared (TI for transmission or RI for reception) by software so that the next transmission (reception) can be initiated.
8. Repeat steps 5 to 7 for the next byte transmission and steps 6 and 7 for the next byte reception.

The following examples show how the above steps are implemented to develop program for serial data transfer.

Example 15.7

Write an assembly-language program fragment to transmit a letter "A" serially using the serial port at 9600 baud rate.

Solution:

To transmit the data, we need to use Timer 1 to generate a clock signal at the desired Baud rate. It is achieved by the following steps:

- Configure Timer 1 as an interval timer in Mode 2 using TMOD register.
- Load value into TH1 to get desired Baud rate.
- Configure SCON register in Mode 1(8-bit data, 1 stop bit).
- Start Timer 1, this will generate clock at the desired Baud rate.
- Write the character to be transmitted in SBUF register.

  Once the character is written into SBUF register, the transmission of character—one bit at a time will be started.
- Wait until the complete character is transmitted, i.e. wait until TI is set.
- Clear TI flag before transmitting next character, so that it can be monitored again.

The program to perform the above steps is listed below:

```
        MOV TMOD, #20H      // initialize Timer 1, Mode 2(auto-reload)
        MOV TH1, #0FDH      // 9600bps baud rate
        MOV SCON, #50H      // 8-bit data, 1 stop bit, REN enabled
        SETB TR1            // start Timer 1 to generate clock (at baud rate)
        MOV SBUF, #"A"      // letter "A" to be transferred is placed in SBUF
HERE:   JNB TI, HERE        // wait until complete byte is transferred
        CLR TI              // clear TI before sending next byte
```

Example 15.8

Write an assembly-language program to transfer the message "HAPPY " serially, continuously at 9600 bps baud rate, 8-bit data, and 1 stop bit.

Solution:

The initialization part to configure TMOD and SCON and loading the count in TH1 will remain the same as transmitting a single character, the whole string can be transmitted by transmitting one byte at a time and repeating the operation until all characters are transmitted. To make the program more efficient, we will define a subroutine and call it repeatedly for transmitting all the characters. The subroutine will load the character in SBUF register and will wait until complete character (8 bits) are transmitted (TI = 1), and then also clear the TI flag.

```
        ORG 0000H
        MOV TMOD, #20H      // Timer 1configured in Mode 2
        MOV TH1, #0FDH      // set 9600 bps baud rate
        MOV SCON, #50H      // 8-bit data, 1 stop bit, REN enabled
        SETB TR1            // start Timer 1 to generate clock (at baud rate)
REPEAT: MOV A, #"H"         // load "H" into A, and call subroutine that will
        ACALL SEND          // transmit the character
        MOV A, #"A"         // send "A"
        ACALL SEND
        MOV A, #"P"         // send "P"
        ACALL SEND
        MOV A, #"P"         // send "P"
        ACALL SEND
        MOV A, # "Y"        // send "Y"
        ACALL SEND
        MOV A, # " "        // send space
        ACALL SEND
        SJMP REPEAT         // transmit "HAPPY" repeatedly
SEND:   MOV SBUF, A         // serial data transfer subroutine
HERE:   JNB TI, HERE        // wait until the last bit is sent
        CLR TI              // clear TI before sending the next byte
        RET
        END
```

Simulation Result (In Keil µVision 4.0 IDE)

The transmitted data using UART can be observed in serial output windows. Open serial windows from **View→Serial windows→UART#1** menu. The snapshot of the output is shown in Fig. 15.14.

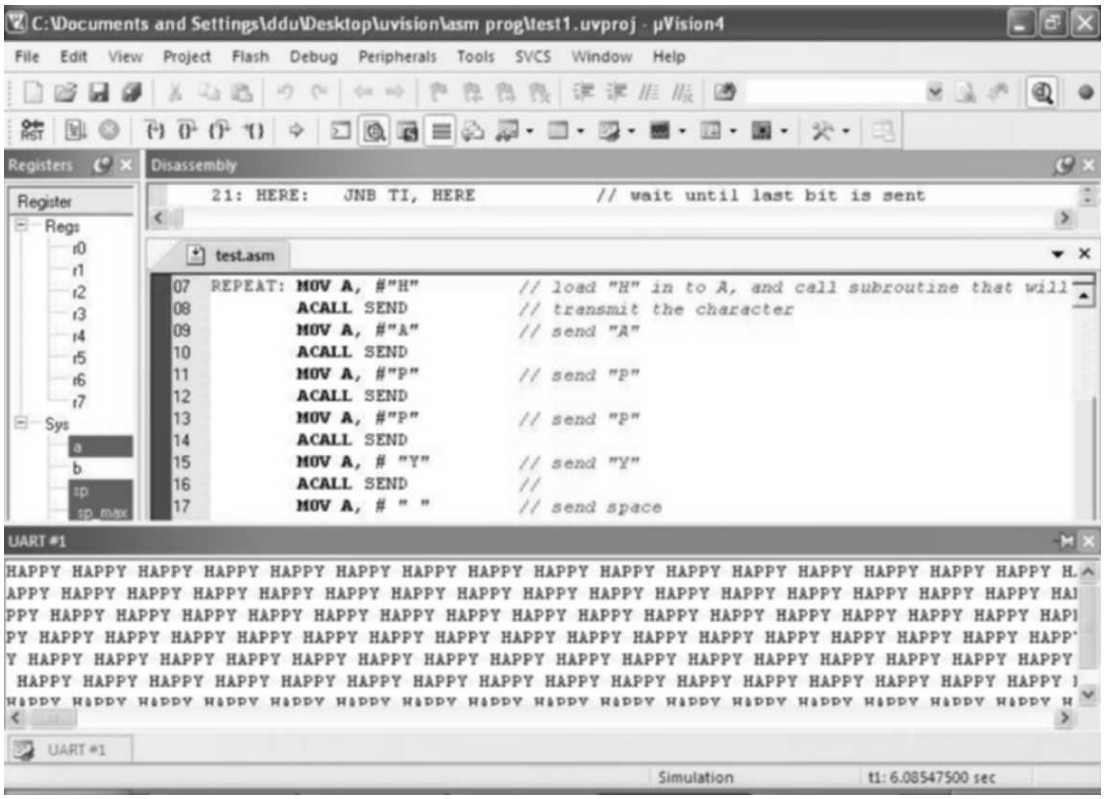


Fig. 15.14 *Output window for Example 15.8*

Example 15.9

Write an assembly-language program to receive bytes serially with baud rate 9600, 8-bit data and 1 stop bit. Simultaneously send received bytes to Port 2.

Solution:

First, we need to enable data reception by setting REN bit in SCON register, the other initialization part of configuring the TMOD and SCON and loading the count in TH1 will remain the same as transmitting the character. Once Timer 1 is started to generate clock at desired rate, the following steps are taken:

- Wait until RI flag is set, RI = 1 indicates that character is received in SBUF.
- Read SBUF and save its contents at desired location.
- Clear RI flag, so that it can be monitored to check reception of next character.
- Wait for reception of next character.

```
        ORG 0000H
        MOV TMOD, #20H          // Timer 1configured in Mode 2
        MOV TH1, #0FDH          // set 9600 bps baud rate
        MOV SCON, #50H          // 8-bit data, 1 stop bit, REN enabled
        SETB TR1                // start Timer 1 to generate clock (at baud rate)
REPEAT: JNB RI, REPEAT          // wait until character byte is received
        MOV A, SBUF             // read and save the received character
        MOV P2, A               // send character to Port 2
        CLR RI                  // get ready to receive next byte
```

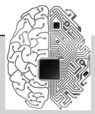
```
SJMP REPEAT                    // go to receive next character
END
```

Simulation Procedure (In Keil μVision 4.0 IDE)

The reception of data byte is simulated using the virtual register SxIN. The steps to simulate reception of data byte are given as follows:

- Free run (or single step) the program.
- When program is waiting to receive a byte, i.e. when it is executing the instruction "REPEAT: JNB RI, REPEAT" the byte can be given to the UART using SxIN virtual register (x represent UART number if there are more than one UARTs in a device. In the case of 8051, there is only one UART, therefore we have to use the name SIN).
- To give data byte 45H, type command SIN = 0x0045 in the command window (in general, the format is SIN = 0x00XX, where XX represents the data byte in hex).
- Execute the command by pressing Enter key. It will simulate the reception of the data byte; the received byte can be read from the SBUF register.

Note that we can give ASCII byte directly to UART using command SIN = 'ASCII code'



THINK BOX 15.5

Why do we need to clear the RI flag once we read the received byte from the SBUF register?

If the RI bit is not cleared, the program will erroneously assume that a new byte is received when it checks the RI flag for the next time. This will cause the program to read the same byte repeatedly until it is not cleared.

Example 15.10

Write a C program to transfer 'A' through 'Z' letters serially at 9600 baud rate continuously. Use 8-bit data and 1 stop bit.

Solution:

The initialization part of control registers will remain the same for transmission as discussed in detail in Example 15.7.

ASCII value of character 'A' is first loaded into temporary variable and the variable is written into SBUF register for transmission, while loop is used to wait until a character is transmitted. The statement 'while (TI==0);' will be repeatedly executed as long as TI is 0, note the semicolon after the statement. When one character is transmitted, the content of temporary variable is incremented to make it the next character in alphabetical order (ASCII value of 'A' is 41H, for 'B' is 42H, and so on). The temporary variable is incremented until it reaches the value of 'Z', after which it is loaded again with 'A'. The whole process is then repeated forever using the while loop (outer while loop using 'while (1) statement).

```
#include<reg51.h>

        void main (void)
        {
                unsigned char ch = 'A';
                TMOD = 0x20;                    // Timer 1configured in Mode 2
                TH1 = 0xFD;                     // set 9600bps baud rate
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR1 = 1;                        // start Timer 1 to generate clock (at baud rate)
                while (1)                       // repeat the task forever
                {
                        SBUF = ch;              // place character value in buffer
                        while (TI==0);          // wait until byte is transmitted
                        TI = 0;                 // clear TI before sending the next byte
                        ch++;
                        if (ch> 'Z')            //  If character was Z, again start with A
                        ch = 'A';
                }
        }
```

Example 15.11

Write a C program to transfer the message "microcontroller" serially at 9600 baud, 8-bit data and 1 stop bit.

Solution:

The message (string) to be transmitted is defined as character array, and a function is defined to transmit one character. The function is called repeatedly using the 'for' loop until all the characters are transmitted. The index is incremented in each iteration of the 'for' loop to point to the next character.

```
#include<reg51.h>
void serialtransmit (unsigned char);

        void main(void)
        {
                unsigned char a[] =  "microcontroller";
                unsigned char i;
                TMOD = 0x20;                    // Timer 1 configured in Mode 2
                TH1 = 0xFD;                     // set 9600bps baud rate
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR1 = 1;                        // start Timer 1 to generate clock
        for (i=0; i<15; i++)                    // transmit string, one character at a time
                {
                        serialtransmit(a[i]);
                }
        }
        void serialtransmit (unsigned char ch)  // serial transmit function
        {
                SBUF = ch;                      // place the character value in buffer
                while (TI==0);                  // wait until transmitted
                TI = 0;                         // clear TI before sending next byte
        }
```

Example 15.12

Assume that the temperature sensor is interfaced with the 8051 through an 8-bit ADC connected on Port 1. Write a C language program to transmit serially the message 'LOW TEMPERATURE' if ADC output is less than 30H, otherwise transmit the message 'HIGH TEMPERATURE'.

Solution:

The two messages are defined as character arrays, since ADC is connected to Port 1, the value of Port 1 is read and compared with 30H. If it is less than 30H, the message 'LOW TEMPERATURE' is sent, otherwise the message 'HIGH TEMPERATURE' is sent.

A function is defined to transmit one character. The function is called repeatedly similar to Example 15.11.

```
#include<reg51.h>

void serialtransmit (unsigned char);

    void main(void)
    {
        unsigned char msg1[] =  "LOW TEMPERATURE";
        unsigned char msg2[] =  "HIGH TEMPERATURE";
        unsigned char i;
        TMOD = 0x20;                            // Timer 1 configured in Mode 2
```

```
        TH1 = 0xFD;                         // set 9600bps baud rate
        SCON = 0x50;                        // 8-bit data, 1 stop bit, REN enabled
        TR1 = 1;                            // start Timer 1 to generate clock
        while (1)                           // monitor temperature continuously.
        {
        if (P1<0x30)
                {
                for (i=0; i<15; i++)        // transmit string, one character at a time
                        {
                        serialtransmit(msg1[i]);    // call function to transmit byte
                        }
                }
        else
                {
                for (i=0; i<16; i++)        // transmit string, one character at a time
                        {
                        serialtransmit(msg2[i]);    //call function to transmit byte
                        }
                }
        }
}
void serialtransmit (unsigned char ch)      // serial transmit function
{
        SBUF= ch;                           // place the character value in buffer
        while (TI==0);                      // wait until transmitted
        TI=0;                               // clear TI before sending next byte
}
```

Example 15.13

Write a program in C to receive bytes of data serially and put them in P0. Set the Baud rate at 2400, 8-bit data and 1 stop bit.

Solution:

```
#include<reg51.h>

void main (void)
{
        unsigned char recbyte;
        TMOD = 0x20;                        // use Timer 1 in 8-bit auto-reload mode
        TH1 = 0xF4;                         // set 2400bps baud rate
        SCON = 0x50;                        // 8-bit data, 1 stop bit, REN enabled
        TR1 = 1;                            // start timer to generate clock
        while (1)                           // repeat task forever
        {
                while ( RI==0);             // wait to receive
                recbyte = SBUF;             // read SBUF and send character value on port 0
                P0 = recbyte;
                RI = 0;                     // clear RI to detect arrival of next byte
        }
}
```

Example 15.14

Read the contents of ports P0, P1 and transfer their contents serially one after the other continuously at Baud rate 4800.

Solution:

The contents of P0 is read first and transmitted, then contents of P1 is read and then transmitted; this process is repeated forever using the SJMP instruction.

```
            MOV TMOD, #20H          // configure Timer 1, 8-bit auto-reload mode
            MOV TH1, #FAH           // set baud rate 4800 bps.
            MOV SCON, #50H          // 8-bit data, 1 stop bit, REN enabled
            MOV P0, #0FFH           // configure P0 as input port
            MOV P1, #0FFH           // configure P1 as input port
            SETB TR1                // start timer to generate clock
REPEAT:     MOV A, P0               // read Port 0
            ACALL TANSMIT           // call subroutine for transmission
            MOV A, P1               // read Port 1
            ACALL TANSMIT           // call subroutine for transmission
            SJMP REPEAT             // repeat task forever
TRANSMIT :  MOV SBUF, A             // load data into the SBUF
WAIT:       JNB TI, WAIT            // wait until character is sent
            CLR TI                  // clear TI before sending the next byte
            RET                     // return to calling program
```

Example 15.15

Rewrite the program of Example 15.14 in the C language.

Solution:

```
#include<reg51.h>
void  transmit ( unsigned char);                    // declare the function to transmit a character
void main()
        {
        unsigned char i ;
        P0 = 0xFF;                                  // configure P0 as an input
        P1 = 0xFF;                                  // configure P1 as an input
        TMOD = 0X20;                                // configure Timer 1 in Mode 2
        TH1 = 0XFA;                                 // set baud rate 4800 bps.
        SCON = 0X50;                                // 8-bit data, 1 stop bit, REN enabled
        TR1 = 1;                                    // start timer to generate clock
        while (1)                                   // repeat continuously
                {
                i = P0;                             // read P0 and call function to transmit a character
                transmit (i) ;
                i = P1 ;                            //  read P1 and call function to transmit a character
                transmit (i) ;
                }
        }
void transmit(unsigned char x)                      // function to transmit one character
        {
        SBUF = x;
        while (TI==0) ;
        TI = 0;
        }
```

Example 15.16

Write a C program to send the messages "Double speed" to the serial port with double Baud rate 56K (28800 x 2). Assume that XTAL = 11.0592 MHz.

Solution:

The Baud rate can be doubled by setting the SMOD bit in the PCON register. Refer Figure 15.13 for more details.

```
#include<reg51.h>
void main (void)
        {
                unsigned char i;
                unsigned char Msg[] = "Double speed";
                TMOD = 0x20;                    // Timer 1 configured in Mode2
                TH1 = 0xFF;                     // set 28800 baud rate
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR1 = 1;                        // start timer to generate clock
                PCON = PCON | 0x80;             // set SMOD bit for double speed of 56K
                        for( i = 0; i <12; i ++)
                            {
                            SBUF = Msg[i];      // place value in buffer
                            while (TI==0);      // wait until byte is transmitted
                            TI = 0;             // clear TI before sending the next byte
                            }
        }
```

Note that the PCON register is not bit-addressable and, therefore, we cannot set SMOD bit by instruction SETB SMOD.

Example 15.17

Write a program to transmit an ASCII character 'Z' continuously with a baud rate of 19200 with a crystal frequency of 11.0592 MHz.

Solution:

The Baud rate of 19200 bps is generated by setting Baud rate as 9600 bps and doubling it by setting the SMOD bit in the PCON register.

```
        MOV A, PCON         // for doubling the baud rate set SMOD bit high (D7 bit of PCON register)
        SETB ACC.7
        MOV PCON, A
        MOV TMOD, #20H      // Timer 1 configured in Mode2
        MOV TH1, #-3        // set baud rate
        MOV SCON, #50H      // 8-bit data, 1 stop bit, REN enabled
        SETB TR1            // start timer to generate clock
        MOV A, #'Z'
BACK:   MOV SBUF, A         // transmit the character
HERE:   JNB TI, HERE        // wait until the character is transmitted
        CLR TI              // clear TI before sending the next byte
        SJMP BACK           // repeat transmission forever
```

Note that the immediate value –3 is loaded into the TH1 register. The assembler will convert -3 into FD (FD is signed number representation, i.e. 2's complement representation of –3).

Example 15.18

Rewrite the program of Example 15.17 in the C language.

Solution:

```
#include<reg51.h>
void main()
        {
        PCON = PCON | 0x80;        // set SMOD bit of PCON to double the baud rate
        TMOD = 0x20;               // Timer 1 Mode2
        TH1 = –3;                  // set baud rate
        SCON = 0x50;               // 8-bit data, 1 stop bit, REN enabled
        TR1 = 1;                   // start timer to generate clock
        while (1)                  // transmit character forever
                {
                SBUF = 'Z';        // transmit the character
                while (TI==0) ;    //  wait until the character is sent
                TI = 0;            // clear TI before sending the next byte
                }
        }
```

15.5.3 Mode 2—Multiprocessor Communication

This mode transmits 11 bits through the TXD pin and receives through the RXD pin as follows: a START bit (always 0), 8 data bits (LSB first), a programmable 9th data bit and a STOP bit (always 1).When transmitting, the 9th data bit is the TB8 bit of the SCON register. When receiving, the 9th data bit is stored into the RB8 bit of the SCON register. The operation of Mode 2 is shown in Figure 15.15.

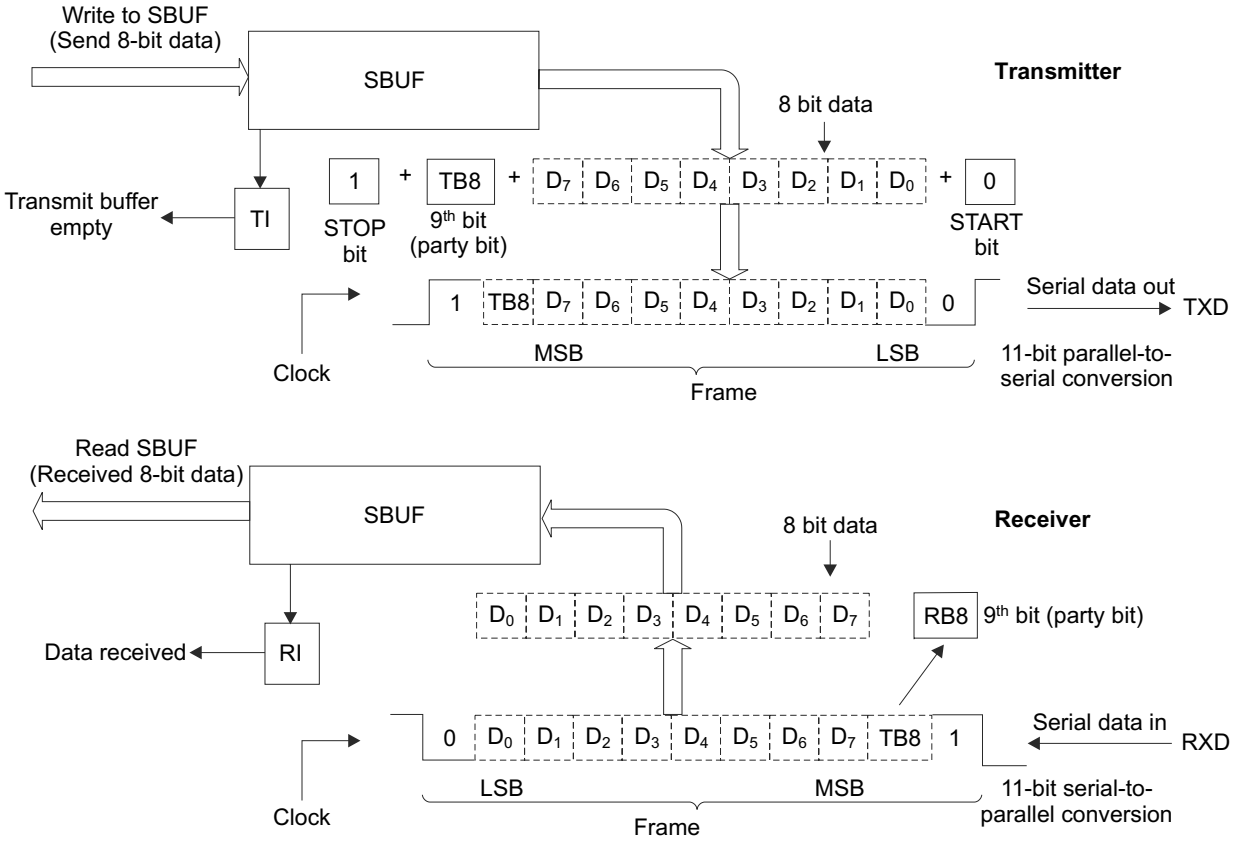


Fig. 15.15 *UART Mode 2 operation*

Transmission

When transmitting, 8 data bits are loaded into the SBUF. The 9th data bit is the value of the TB8 bit in the SCON register. The programmer can set or clear this bit. The START, STOP and TB8 bits are combined by hardware to form an 11-bit frame (Figure 15.15). Then, 11-bit parallel-to-serial conversion is performed and one bit (LSB first) at a time is transmitted through the TXD pin. Once the complete frame is transmitted, the TI flag is set automatically by serial port hardware to indicate the end of data transmission.

Reception

The data reception begins when REN = 1 and high-to-low transition (start bit) is detected on the RXD pin. The received 8 data bits are loaded into SBUF register (the START and STOP bits are separated by UART hardware once the complete frame is received) and 9th data bit into the RB8 (SCON Bit 2) only if the following two conditions are met.

(i) RI = 0, showing that previous data byte has been read by the program, and

(ii) Either SM2 = 0 or received 9th data bit = 1;

If these two conditions are not met, the received character is ignored and RI is not set the and receiver circuit waits for the next start bit.

Multiprocessor Communication

An important application of this mode is in the multiprocessor communication, i.e. communication between two or more microcontrollers. This is done by setting the SM2 bit of the SCON register. After receiving the STOP bit, the serial port interrupt will be generated only if either bit RB8 = 1 (the 9th bit) or SM2 bit is 0.

Consider that there are many microcontrollers connected through a common interface (channel) as shown in Figure 15.16. Each of them will be assigned a unique address. An address byte is differentiated from data byte using 9th data bit. For address, 9th bit =1, and for data byte, this bit is 0. When the microcontroller A (master) wants to transmit data to one (of many) slaves (say, microcontroller C), it first sends the address byte. (All the slaves will have initially SM2 = 1 as shown in Figure 15.16 (a). All the slaves will receive this address byte and it will generate an interrupt in all slaves, the interrupt service routine will check whether it matches their address. Since all the slaves have unique addresses, only one slave (microcontroller C) will match the address. This is how the desired slave is identified, now this slave will clear the SM2 bit of the SCON register and be ready to receive the data bytes sent by the master as shown in Figure 15.16 (b). Other slaves (not addressed) ignore the received data bytes because their SM2 bit is still 1.

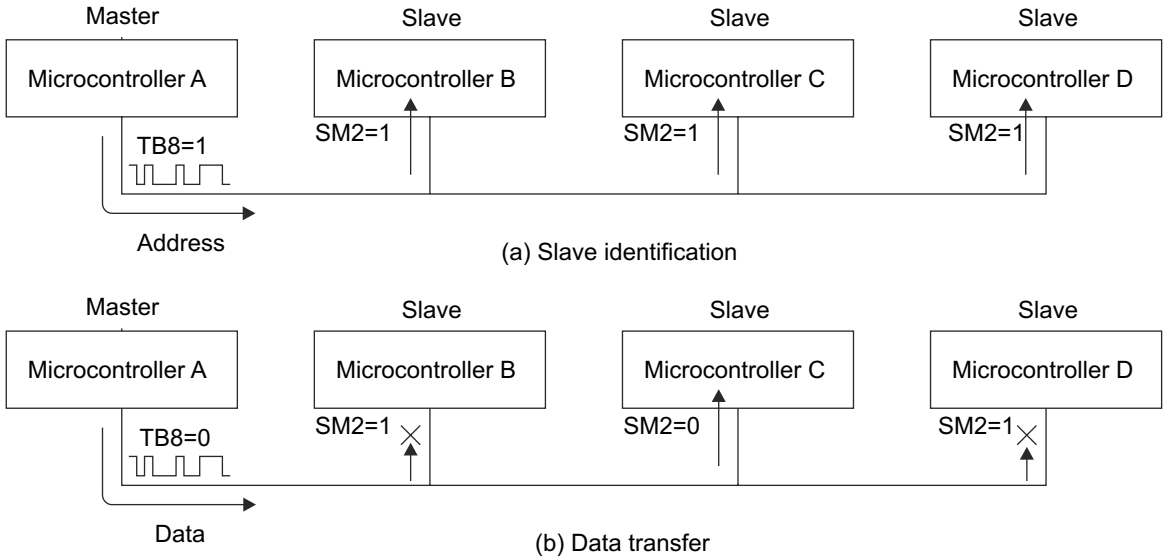


Fig. 15.16 *Multiprocessor communication*

Baud Rate for Mode 2

Mode 2 has a fixed Baud rate which is 1/64 of the oscillator frequency when SMOD = 0, and 1/32 of the oscillator frequency when SMOD = 1. To run the serial port in Mode 2, no timer/counters are needed. Only the SCON register needs to be defined.

Example 15.19

What will be the baud rate for Mode 2 when crystal frequency is 12 MHz?

Solution:

For SMOD = 0, it is 12 MHz/64 = 187.5 Kbps and for SMOD = 1, it is 12 MHz/32 = 375 Kbps.

Example 15.20

Write a program to transmit 9 bits '1 10101010' using the serial port in Mode 2.

Solution:

For the desired operation, we need to configure serial port into Mode 2 and write the lower 8 bits into SBUF register and 9th bit into TB8 bit. Once we write the data byte into SBUF, the data transmission is started. Upon completion of transmission, the TI flag will be set; therefore, we need to wait until TI flag is set and then we need to repeat the operation. Assuming SMOD = 0 and crystal frequency is 12 MHz, the Baud rate will be 187.5 Kbits/s.

```
        ORG 0000H
        MOV SCON, #88H      // configure serial port in Mode 2 and TB8=1
        MOV SBUF, #0AAH     // write 8 bits in SBUF for transmission
WAIT:   JNB TI, WAIT        // wait until TI flag is set
        CLR TI              // clear TI before sending the next data byte
HERE:   SJMP HERE           // wait indefinitely
        END
```

Example 15.21

Rewrite the program of Example 15.20 in the C language.

Solution:

```
#include<reg51.h>

void main (void)
        {
        SCON = 0x88;        // configure serial port in Mode 2, TB8 = 1
        SBUF = 0xAA;        // write 8 bits in SBUF for transmission
        while (TI==0);      // wait until data byte is transmitted
        TI = 0;             // clear TI before sending the next byte
        while (1);          // wait indefinitely
        }
```

15.5.4 Mode 3

Mode 3 is the same as Mode 2 in all respects except the Baud rate. The Baud rate in Mode 3 is variable and generated using Timer 1 similar to Mode 1.

15.6 | SECOND SERIAL PORT IN THE DS89C4X0

The DS89C4x0 family of microcontrollers (high-performance 8051 compatible microcontrollers) from Dallas Semiconductors comes with an additional serial port. This serial port is referred as second serial port and it is functionally similar to the original (first) serial port. It uses a separate set of SFRs and transmit/receive pins for its operation. Both the serial ports can be used simultaneously and may operate at different Baud rates. For the original serial port, Timers 1 or 2 may be used to generate clock signal at desired Baud rate while only Timer 1 can be used for the second serial port.

To avoid the confusion between the two serial ports, the original serial port is referred as Serial Port 0 and all SFRs and pins used by it are assigned suffix '0', while the second serial port is referred as Serial Port 1 and the SFRs and pins used by it are assigned suffix '1'. The names and addresses of SFRs (along with TI and RI flags) and port pins used by each port is summarized in Table 15.5.

Table 15.5 *SFR details of both serial ports in DS89C4x0*

SFR/Pin/ Flag	Original serial port (Serial port 0)	Second serial port (Serial port 1)	Remarks
Transmit	TXD0 (P3.1)	TXD1 (P1.3)	Serial Port 1 uses Port 1 pins for transmit/receive operations.
Receive	RXD0 (P3.0)	RXD1 (P1.2)	
SCON	SCON0 (98H)	SCON1 (C0H)	Serial Port 1 uses addresses for its SFRs which were reserved in the original 8051
SBUF	SBUF0 (99H)	SBUF1(C1H)	
TI	TI0 (SCON0.1 or 99H)	TI1 (SCON1.1 or C1H)	The vector address for Serial Port 0 is 23H while for Serial Port 1 it is 3BH.
RI	RI0 (SCON0.0 or 98H)	RI1 (SCON1.0 or C0H)	

Note that the bit assignment and use of SCON and SBUF registers of both the serial ports is exactly similar except for their addresses. The line driver chip MAX232 or MAX233 can be used to connect these ports to RS232 compatible devices. Since MAX232 and MAX233 have two sets of line drivers, only a single chip is required to connect both the serial ports of DS89C4x0 to the PC or other RS232 device.

The operation of the second serial port is illustrated in the following examples.

Example 15.22

Write an assembly-language program to continuously transmit a letter "M" serially using Serial Port 1 of DS89C430 at 9600 Baud rate. Use Mode 1.

Solution:

This example is almost similar to Example 15.7; refer that example for explanation of program-development steps.

Timer 1 is used by default to generate a clock and set the required Baud rate.

```
        SCON1 EQU  0C0H      // address of SCON1
        SBUF1 EQU  0C1H      // address of SBUF1
        TI1   BIT  0C1H      // bit address of TI1
                             // TI1 EQU 0C1H will also work

        ORG 0000H
        MOV TMOD, #20H       // initialize Timer 1, Mode 2(auto-reload)
        MOV TH1, #0FDH       // 9600 bps Baud rate
        MOV SCON1, #50H      // 8-bit data, 1 stop bit, REN enabled
        SETB TR1             // start timer 1 to generate clock (at baud rate)
REPEAT: MOV SBUF1, #"M"      // letter "M" to be transferred is placed in SBUF
HERE:   JNB TI1, HERE        // wait until complete by is transferred
        CLR TI1              // clear TI before sending the next byte
        SJMP REPEAT
        END
```

The output can be observed in **View → Serial window→ UART#2** window in the simulator. (Keil µVision 4.0 IDE)

Note that we have to define SCON1, SBUF1 and TI1 as shown in the example because they were not available in the original 8051 and therefore older assemblers may not support them.

Example 15.23

Rewrite program of Example 15.22 in C language.

Solution:

```
#include<reg51.h>
sfr SCON1= 0xC0;                    // address of SCON1
sfr SBUF1= 0xC1;                    // address of SBUF1
sbit TI1=0xC1;                      // bit address of TI1

void main (void)
        {
        unsigned char ch = 'M';
        TMOD = 0x20;                // Timer 1configured in Mode 2
        TH1 = 0xFD;                 // set 9600bps baud rate
        SCON1 = 0x50;               // 8-bit data, 1 stop bit, REN enabled
        TR1 = 1;                    // start Timer 1 to generate clock (at baud rate)
        while (1)                   // repeat the task forever
                {
                SBUF1 = ch;         // place character value in buffer
                while (TI1==0);     // wait until byte is transmitted
                TI1 = 0;            // clear TI before sending the next byte

                }
        }
```

Example 15.24

Write a program in assembly language to receive the data byte from Serial Port 0 and send the same data to Serial Port 1 of DS89C430. Do the specified operation continuously. Set Baud rate as 4800 bps.

Solution:

```
                SCON1 EQU  0C0H      // address of SCON1
                SBUF1 EQU  0C1H      // address of SBUF1
                TI1   BIT  0C1H      // bit address of TI1
                                     // TI1 EQU 0C1H will also work
                SCON0 EQU  98H       // address of SCON0
                SBUF0 EQU  99H       // address of SBUF0
                RI0   BIT  98H       // bit address of RI0
                ORG 0000H
                MOV TMOD, #20H       // initialize Timer 1, Mode 2 (auto-reload)
                MOV TH1, #0FAH       // 4800 bps baud rate
                MOV SCON0, #50H      // configure both serial ports for
                MOV SCON1, #50H      // 8-bit data, 1 stop bit, REN enabled
                SETB TR1             // start Timer 1 to generate clock (at baud rate)
REPEAT:         JNB RI0, REPEAT      // wait to receive character at Serial Port 0
                MOV A, SBUF0         // save the received character
                MOV SBUF1, A         // send character to Serial Port 1
HERE:           JNB TI1, HERE        // wait until the complete byte is transferred
```

```
CLR TI1                 // clear TI1 before sending the next byte
CLR RI0                 // get ready to receive the next byte
SJMP REPEAT             // do operation continuously
END
```

Note that we have also defined SCON0, SBUF0 and RI0 in the program. This is required because those registers are defined as SCON, SBUF and RI in the existing assemblers. Had we used the original names of the SFRs (SCON, SBUF and RI) for the Serial Port 0, we could have avoided their definitions in the program. But to avoid the confusion, we use suffix '0'.

The simulation of reception of a byte at Serial Port 0 can be done by command S0IN = 0x00XX, where XX is the byte to be given for reception. See Example 15.9 for more details of SIN.

POINTS TO REMEMBER

- There are three types of serial communication systems: simplex, half-duplex and full-duplex.
- In a synchronous communication, a common clock is used for both transmitter and receiver.
- In an asynchronous communication, the synchronization is achieved only at the beginning of the transmission.
- Baud rate is defined as rate at which signal transition occurs, the terms Baud rate and bits/s are not always the same.
- We must use voltage converters to convert TTL voltage levels to the RS232 voltage levels and vice versa.
- MAX232 and MAX233 chips are the most popular voltage converts and are more commonly referred as line drivers.
- Even though there is only one name and address assigned to the SBUF register, there are two physical registers—Transmit Buffer and Receive Buffer.
- In Mode 0, data are transmitted and received through the RXD pin, and the TXD pin outputs the shift clock and this is used to synchronize data transmission/reception.
- Mode 0 has a fixed Baud rate which is 1/12 of the oscillator frequency.
- The TI flag is set by the microcontroller when it completes byte transfer, whereas it must be cleared by the programmer after the next byte is loaded into SBUF.
- For transmission, the TI flag is monitored to make sure that a byte has been transmitted completely. For reception, the RI flag is monitored to check that byte is received or not.
- Mode 1 transmits 10 bits (1start bit, 8 data bits and 1 stop bit) through TXD pin and receives through RXD pin.
- Timer 1 overflow rate determines Baud rate for Mode 1.
- Mode 2 transmits 11 bits (1 start bit, 8 data bits, programmable 9^{th} data bit and 1 stop bit) through TXD pin and receives through RXD pin.
- Mode 2 has a fixed baud rate which is 1/64 of the oscillator frequency when SMOD = 0, and 1/32 of the oscillator frequency when SMOD = 1.
- Mode 3 is the same as Mode 2 in all respects except that it has a variable Baud rate and controlled by Timer 1 overflow rate.

OBJECTIVE QUESTIONS

1. The baud rate of Mode 0 serial communication is,
   (a) fixed (b) variable (c) 12 Mbps (d) none
2. The Baud rate of Mode 1 serial communication is,
   (a) fixed (b) variable (c) 12 Mbps (d) none
3. Asynchronous transmission always begins with,
   (a)start bit (b) stop bit (c) parity bit (d) sync bit
4. The baud rate of serial transfer using Mode 0 of 8051 running at 6 MHz clock frequency is,
   (a) fixed at 256 Kbps (b) fixed at 512 Kbps
   (c) variable with maximum with 256 Kbps (d) variable with maximum with 512 Kbps

5. An alternate function of Port pin 10 (P3.0) in the 8051 is,
   (a) serial port input: RXD (b) serial port output: TXD
   (c) memory write strobe: $\overline{WR}$ (d) memory read strobe: $\overline{RD}$
6. Which of the following links can be used for either transmission or reception of data?
   (a) Simplex (b) Half-duplex (c) Full-duplex (d) All of the above
7. UART Mode 0 and Mode 1 are,
   (a) both asynchronous (b) both synchronous
   (c) asynchronous and synchronous respectively (d) synchronous and asynchronous respectively
8. When SM2 = 1,
   (a) it facilitates the multiprocessor communication in case of Mode 1 and Mode 3
   (b) it facilitates the multiprocessor communication in case of Mode 2 and Mode 3
   (c) it facilitates the multiprocessor communication in case of Mode 1 only
   (d) it facilitates the multiprocessor communication in case of Mode 0 only
9. If a time period between the serial bits is 0.33 ms, for UART Mode 2, we can transfer in one second,
   (a) 300 characters (b) 275 characters (c) 30 characters (d) 272 characters
10. In RS232,
    (a) 0 means + 3 V to + 25 V (b) 0 means + 5 V to + 25 V
    (c) 1 means + 3 V to + 25 V (d) 1 means + 5 V to + 12 V
11. Match the following:
    (1) TCON (i) arithmetic flags
    (2) SBUF (ii) timer/counter control register
    (3) TMOD (iii) idle and power down bit
    (4) PSW (iv) buffer for TXD and RXD
    (5) PCON (v) timer modes of operation
    (a) 1→ii, 2→iv, 3→v, 4→i, 5→iii
    (b) 1→i, 2→v, 3→iv, 4→iii, 5→ii
    (c) 1→v, 2→iii, 3→ii, 4→iv, 5→i
    (d) 1 → iii, 2 →ii, 3→ i, 4→ v, 5→iv
12. Baud rate for UART Mode 1 is given as,
    (a) $2^{SMOD}/32$ * (Timer 0 over flow rate) (b) $2^{SMOD}/16$ * (Timer 1 over flow rate)
    (c) $2^{SMOD}/16$ * (Timer 0 over flow rate) (d) $2^{SMOD}/32$ * (Timer 1 over flow rate)
13. If number of data bits encoded per signal transition = 4, and baud rate is 600, the data transfer speed in bits per second is,
    (a) 600 bps (b) 1800 bps (c) 1200 bps (d) 2400 bps
14. In which modes, is Baud rate dependent on Timer 1 overflow rate?
    (a) Mode 1 and Mode 3 (b) Mode 2 and Mode 3
    (c) Mode 0 and Mode 2 (d) Mode 0 and Mode 1
15. The 9^{th} data bit is transmitted in Mode 2 using,
    (a) TB8 bit of the SCON register (b) TI bit of SCON register
    (c) SM0 bit of the SCON register (d) none of the above

Answers to Objective Questions

1. (a)	2. (b)	3. (a)	4. (b)	5. (a)	6. (b)	7. (d)
8. (b)	9. (b)	10. (a)	11. (a)	12. (d)	13. (d)	14. (a)
15. (a)						

REVIEW QUESTIONS WITH ANSWERS

1. What are the key features of serial and parallel communications?

A. In serial communication, one bit data is transferred at a time, is slower, uses single wire (or pair of wires) and is suitable for long-distance communications. In parallel communication, multiple bits are transmitted simultaneously on multiple links (wires), faster and suitable for short-distance communications.

2. What is meant by synchronous communication?

A. Transmitter and receiver are synchronized by a common clock signal.

3. What is the advantage and disadvantage of synchronous communications?

A. Advantage is that message length can be larger; disadvantage is that extra link (wire) is required for synchronization, which will increase the system cost.

4. What are the types of serial communications? Give an example of each.

A. Simplex—Sending data to a printer, radio broadcasting; Half duplex—two-way radio system, Full-duplex—Telephone conversation.

5. What are the features of full-duplex systems?

A. Data transmission takes place in both directions simultaneously. Two links are required.

6. How is synchronization achieved in asynchronous communication?

A. Using start and stop bits.

7. "Baud rate and bits/s are not always the same." True or false.

A. True.

8. What are the common Baud rates?

A. 110, 150,300, 600, 1200, 2400, 4800, 9600, 19200.

9. What are the components of a message frame?

A. Data and overhead (start, stop, parity bits)

10. What are the voltage levels used for RS232?

A. –3 V to –15 V for logic level high, typically –12 V.
3 V to 15 V for logic low level low, typically 12 V.

11. Why are line drivers like MAX232 used?

A. To convert the voltages levels of RS232 to TTL levels and vice versa.

12. If two 8051 microcontrollers are connected using serial link, does it require a line driver like MAX232?

A. No. Because both are working at same logic levels (TTL compatible). Line drivers are only used if the two devices operate at different logic levels.

13. Which SFRs in the 8051 are used to control UART operations?

A. Serial Control Register (SCON), Serial Buffer Register (SBUF), TMOD and PCON register.

14. The 8051 has two separate SBUF registers accessed at the same address. Justify true/false with reason.

A. True, one is used to hold the data to be transmitted, usually destination in an instruction, and the other is used to hold received data, usually source in an instruction.

15. If the serial data transmitter program does not seem to send any data, what can be the problem?

A. Check following points in program:

- Check if timer is configured correctly to achieve the desired Baud rate.
- Check if receiver program is set at the same Baud rate.
- Make sure that TI is initialized to 1 when the serial port is first configured. Check if it is monitored using JNB instruction.
- Make sure the right type of cable is used.

16. How is the Baud rate doubled?

A. By setting SMOD = 1 in PCON register.

17. Which bit is used to double the Baud rate?

Bit D7 of PCON register.

EXERCISE

1. In an asynchronous serial data transmission, the synchronization is achieved at the beginning of each message. Justify the statement true/false with reason.
2. What is the significance of using crystal frequency to be 11.0592 MHz?
3. Which SFRs are used for serial data communication?
4. What is the Baud rate in serial Mode 0 for 6 MHz crystal?
5. Which flag is used in serial transmission?
6. List the steps involved in selecting baud rate for serial communications.
7. When are the TI and RI flags raised?
8. What action will be taken by the 8051 when RI or TI is raised?
9. What is the default value of SMOD bit?
10. Which serial mode functions as a shift register?
11. Discuss the significance of RI and TI flags with respect to multi-byte data transfer.
12. Discuss the significance of each bit of the SCON register.
13. Draw the frame structure of serial port Mode 1.
14. Draw the frame structure for Mode 1 for ASCII character M, no parity.
15. Illustrate with an example how Mode 2 can be used for multiprocessor communications.
16. Which timer of the 8051 is used to set the Baud rate? Which mode of the timer is used?
17. “Longer the distance, lesser is the Baud rate” Justify statement with reason.
18. For a crystal frequency of 11.0592 MHz, find the value to be loaded into TH1 for Baud rates of 9600 and 150.
19. Simultaneous transmission and reception of data is not possible in Mode 0. Why?
20. Why is Mode 0 not an asynchronous mode?
21. How is Mode 0 useful to monitor and control many devices?
22. How can the data be transmitted or received in Mode 0?
23. Derive the equation to find the value to be loaded in the TH1 register to set a given Baud rate.
24. Write the steps to develop a program to transmit and receive data serially.
25. Find the Baud rate of serial data transfer when crystal frequency is 16 MHz, and value loaded into the TH1 register is F2H and F7H.
26. How is the interrupt-based serial data reception more efficient?
27. If the serial data receiver program does not seem to receive any data, what can be the problem?
28. If one byte is sent to the 8051, the same byte is received at the receiver repeatedly forever. What can be the problem in a receiver program?

16

Interrupts

Objectives

- Discuss the need and uses of interrupts in the microcontroller-based systems
- Compare the polling and interrupt methods to provide services to the devices
- Discuss the interrupt structure of the 8051
- Introduce the concept of interrupt vector table and interrupt service routines
- Illustrate the interrupt execution process
- Describe the interrupt control structure along with the associated control registers
- Discuss the timer, UART and external interrupts
- Program the timer, UART and external interrupts of the 8051 using interrupts
- Describe the interrupt priority structure and nested interrupts
- Discuss the interrupt blocking conditions in the 8051
- Introduce the concept of interrupt latency and interrupt response timing
- Provide tips and cautions to develop the interrupt service routines
- Develop interrupt-based programs for timers, UART and external interrupts in assembly and C language

Key Terms

- Asynchronous Events
- Blocking Conditions
- Context Saving
- External Interrupts
- Internal Interrupts
- Interrupt Enable (IE) Register
- Interrupt Latency
- Interrupt Priority (IP) Register
- Interrupt Service Routine (ISR)
- Interrupt Vector Table (IVT)
- Main Program
- Nested Interrupts
- Non-Maskable Interrupt
- Polling
- Priority
- Reset
- Response Time
- RETI
- Return Address
- Serial Port Interrupts: TI, RI
- Timer Interrupts: TF1,TF0

16.1 NEED OF INTERRUPTS

An interrupt is a signal generated by an event that causes the controller to stop temporarily its current program-execution activities and perform the task to service that event. Interrupts are used to get controller attention towards important events/activities. They allow the microcontroller system to respond to the asynchronous events while another task is being executed; therefore, they give the illusion of handling many tasks simultaneously. The asynchronous event means we do not know in advance when they will occur.

The real-time systems based on microcontrollers must respond as fast as possible to events generated by peripheral devices present in a system. Interrupts always provide a more efficient and effective way to serve many devices. Also, interrupts allow most efficient utilization of time and resources of the microcontroller.

The interrupt is effectively a hardware-generated call, however, the major difference between call instructions and interrupt is the place where they can occur in a program. The call instructions are executed only from the location where they are placed in a program. While interrupts are generated anywhere and any time in a program, i.e. interrupts are asynchronous events.

16.1.1 How are Interrupts Useful?

There are two ways to determine whether peripheral devices require attention or services of the microcontroller. The first approach, based on software, requires the microcontroller to continuously monitor the status of a peripheral device and provides services when the device requires service. This method is known as *polling*. This method is useful when the processor has to do only one task (or a few) and response time is not an issue.

The problems with the polling method are the following:

1. It wastes microcontroller time by monitoring the device continuously.
2. Priorities cannot be assigned to devices because the status can be monitored one by one for all devices irrespective of their priority and importance.
3. It makes the system slower because high-priority device has to wait for its turn when lower priority devices are being polled and serviced.

To overcome all these problems, the second method, which is based on the hardware, is useful. This method is referred as *interrupt method*. In this method, whenever any device requires attention or services of the microcontroller, it will send an interrupt signal to the microcontroller and in response to that, the microcontroller will stop the current activity and serve the device and thereafter, resume the regular activity. This way, the interrupt method provides advantages like the following:

1. The time of a microcontroller is efficiently used as there is no wastage of time in continuous and unnecessary monitoring.
2. Priorities can be given to different devices as per their importance and the higher priority device may be programmed to interrupt lower priority devices.
3. It makes the system faster because more important activities are handled immediately.
4. More devices can be served (though, only one device at a time).
5. The system can be programmed to ignore any or all devices while handling critical tasks.

The source of interrupt may be internal peripherals like timers, serial port or any external device. When any device generates an interrupt, the microcontroller is forced to call a subroutine program associated with that device; this subroutine is more popularly known as *Interrupt Services Routine* (ISR) or *Interrupt Handler*.

16.2 INTERRUPTS IN THE 8051

Five interrupts are available in the 8051.Three of them are internal interrupts, i.e. they are generated because of internal operation of the 8051. They are Timer 0 (TF0), Timer1 (TF1) and Serial Port (TI or RI) interrupts. The remaining two are external interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$, i.e. they are invoked by external signals given to pins $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$. The bar over INT0 and INT1 indicate that they are active low interrupt inputs. The external interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ are also referred as IE0 and IE1 respectively. For each interrupt source, there is a fixed location in the program memory that contains its

Interrupt Service Routine (ISR). This part of the memory which-stores the ISRs is called the *Interrupt Vector Table* (IVT). It is shown in Table 16.1.

Table 16.1 *Interrupt vector table of the 8051*

Interrupt source	Interrupt vector address	Interrupt type	Interrupt flag clearing
External interrupt 0 ($\overline{\text{INT0}}$)	0003H	External	Auto*
Timer 0 interrupt (TF0)	000BH	Internal	Auto
External interrupt 1 ($\overline{\text{INT1}}$)	0013H	External	Auto*
Timer 1 interrupt (TF1)	001BH	Internal	Auto
Serial port interrupt (TI or RI)	0023H	Internal	By program

* For edge-triggered external interrupts only.

When any interrupt is generated (or asserted), it forces the microcontroller to jump to a fixed address in the vector table. For example, when INT0 is asserted, the microcontroller will automatically jump to the memory address 0003H.Similarly for Timer1 interrupt, it will jump to the address 001BH in the interrupt vector table. Since the program execution is transferred to a fixed location, corresponding to each interrupt source, the interrupts in the 8051 are also referred as vectored interrupts. The interrupt vector table of the 8051 is shown in Figure 16.1.

It can be observed from Figure 16.1 that there are only eight bytes reserved for ISR of each interrupt. If ISR requires 8 or less bytes it can be directly written in Interrupt Vector Table, otherwise LJMP (or AJMP) instruction is written at corresponding address in the vector table to jump to the actual address of the ISR. For example, consider the ISR for INT0 is larger than 8 bytes, and the ISR is written at the address 1000H. So we have to write the LJMP instruction at the memory location 0003H, which in turn will jump to the location 1000H. When the INT0 interrupt is asserted, it will force program execution to start at the address 0003H; at this address, the LJMP instruction is written which in turn diverts the program execution to the address 1000H (see Figure 16.2) and ISR is executed. This approach will increase the interrupt response time.

After completion of ISR, program execution must return back to the interrupted program. This is done by storing the return address (address in the PC when interrupt is asserted) on to the stack before jumping to ISR in an

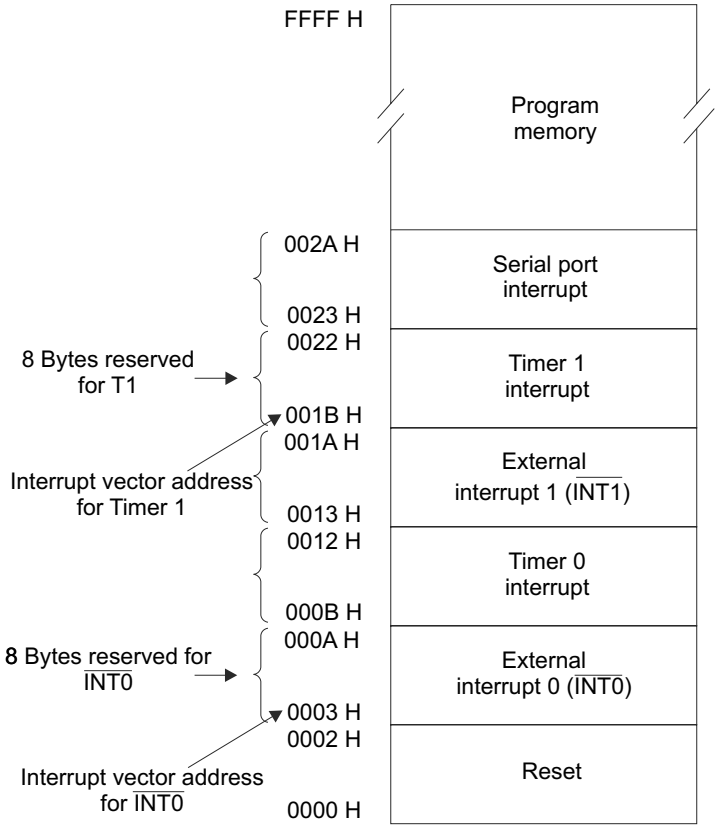


Fig. 16.1 *Interrupt vector table of the 8051*

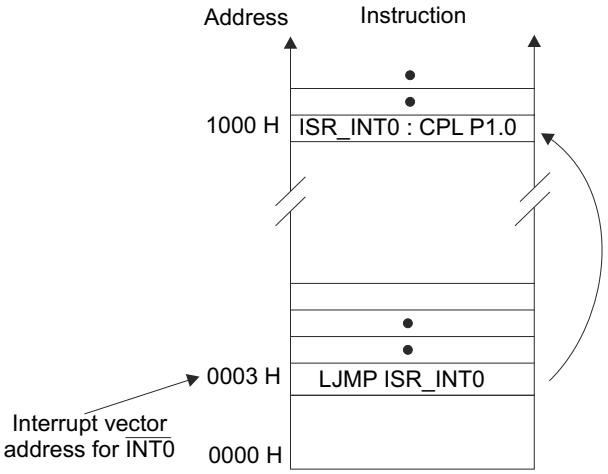


Fig. 16.2 *Jump to interrupt service routine*

interrupt vector table. The PC address will be retrieved back from the stack by RETI instruction usually written at the end of the ISR.

16.2.1 Reset as a Special Interrupt

As a special case, RESET may be considered as the sixth interrupt. When a RESET signal is given, the 8051 will jump to the address 0000H. A major difference between RESET and all other interrupts is that for RESET there is no mechanism to return back to the interrupted program. RESET is somewhat similar to the LJMP instruction while all other interrupts are equivalent to LCALL instruction.

When a RESET signal is given, the microcontroller will stop executing current instruction and immediately jump to the memory location 0000H (even without completing the current instruction as well as saving the return address on to the stack). There are only three bytes of ROM space assigned to the RESET interrupt, i.e. addresses 0000H, 0001H and 0002H. Addresses 0003H onwards belong to the interrupt vector table. Therefore, we usually write LJMP as the first instruction at the address 0000H to skip the interrupt vector table and to jump to the main program (also, AJMP or SJMP may be used if the starting address of the main program is within their range). RESET may also be called non-maskable interrupt because a reset action cannot be stopped by any means. After reset, internal registers will always be loaded automatically with default values as shown in Table 11.1.

16.3 | INTERRUPT HANDLING AND EXECUTION

To allow 8051 to respond to any of the interrupts, they must be enabled by setting interrupt enable flag(s) to 1. The interrupt may be generated by external or internal event. When interrupt is generated, a microcontroller will take the following sequence of steps:

1. It completes execution of current instruction.
2. Saves (pushes) the return address (PC) on to the stack; low byte (PCL) first.
3. The current status of all the Interrupt Enable bits (in IE register) are stored internally, and interrupts of the same or lower priority are disabled (blocked).
4. The PC is loaded with the vector address of the corresponding interrupt source.
5. The interrupt service routine is executed *.
6. At the end of the ISR, the RETI instruction will retrieve the saved status of interrupt enable bits, thus, enabling the same and lower priority interrupts again, and retrieves the return address from the stack into PC and continue to execute from the place it was interrupted.

The process of interrupt execution for the external interrupt 0 (INT0) is illustrated in Figure 16.3.

Assume that external interrupt 0 ($\overline{\text{INT0}}$) is asserted when the instruction 'MOV A, R2' stored at the address 100H is being executed, see Figure 16.3. The microcontroller will complete execution of the ongoing instruction and save the return address (PC) on the stack. Thereafter the status of IE is saved internally and disables interrupts of the same or lower priority. Next, the PC is loaded with vector address (0003H) of $\overline{\text{INT0}}$ and program execution is transferred to the address 0003H. The interrupt service routine is executed and at the end, RETI instruction will restore status of IE, thus re-enabling the interrupts of same or lower priority. The RETI instruction will also retrieve return address into PC and then the execution is resumed to the main program.

THINK BOX 16.1

What steps must be taken by a program to successfully service the received (asserted) interrupt?

To service the interrupt successfully, the program must contain ISRs at corresponding vector addresses of all interrupts (or jump instructions to the ISRs at vector addresses if ISRs require more than 8 bytes). Preferably, save all the registers used in the ISR on the stack to avoid accidental modification of the main program register contents, and perform the operation to service the interrupting device. The ISR program must clear the interrupting condition and reset the interrupt flag if it is not reset automatically. Retrieve the saved registers from stack and come back to the main program using RETI instruction at the end.

*See discussions of individual interrupt source to know how interrupt flags are cleared.

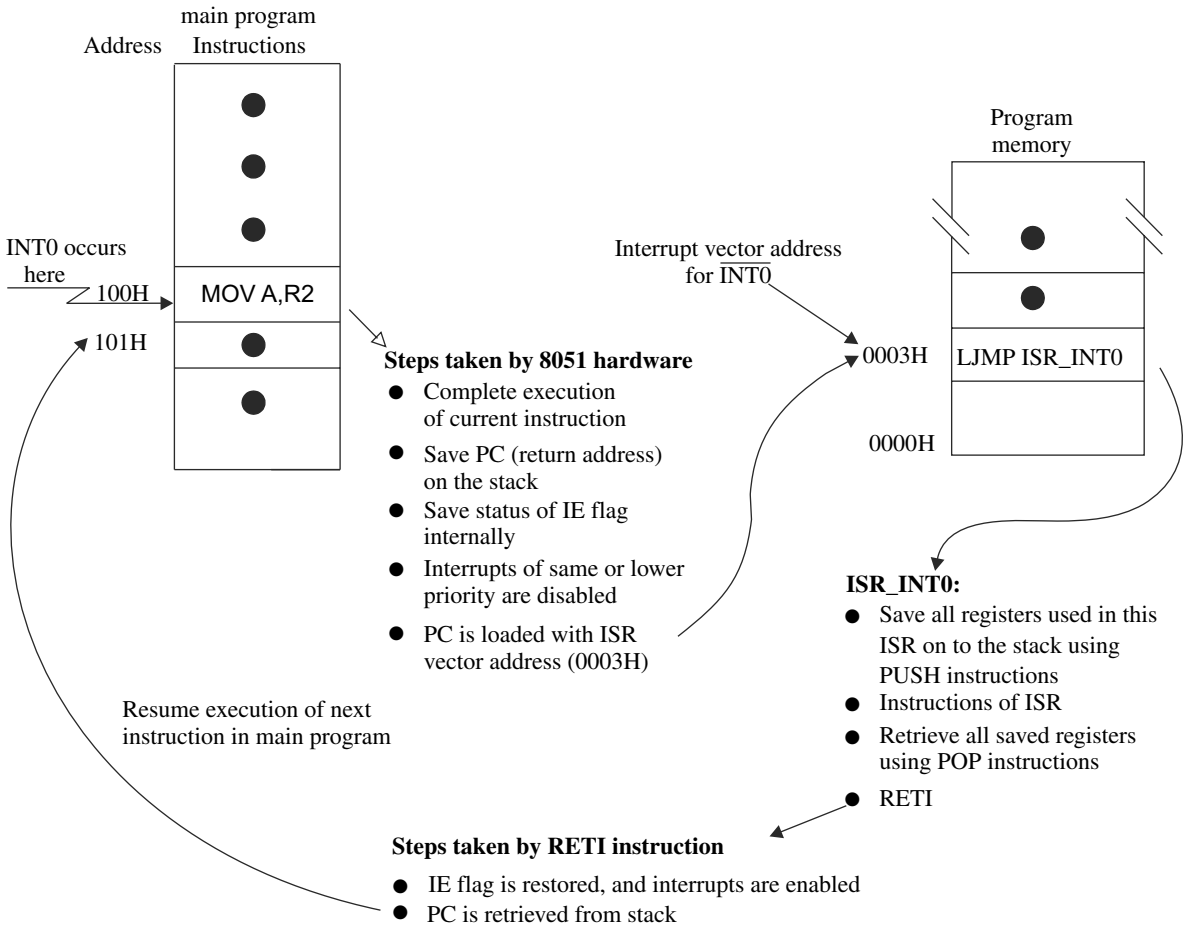


Fig. 16.3 *External interrupt 0 ($\overline{INT0}$) execution process*

16.4 PROGRAMMING THE INTERRUPTS

The 8051 has two registers to control operations of all the five interrupts, the Interrupt Enable (IE) and Interrupt Priority Register (IP). The program can change bits of these registers at any time; therefore, operations of interrupts are fully under control of the program.

16.4.1 Interrupt Enable (IE) Register

The details of IE registers are shown in Table 16.2.

Table 16.2 *Interrupt Enable (IE) register*

EA	--	ET2	ES	ET1	EX1	ET0	EX0
MSB							LSB

Bit	Symbol	Description
7 (IE.7)	EA	Global enable bit; EA = 1 allow each interrupt to be individually enabled or disabled; EA = 0 will disable all interrupts
6 (IE.6)	--	Not implemented, reserved for future use
5 (IE.5)	ET2	Used by later versions of 8051 for Timer 2

(*Contd.*)

(*Contd.*)

Bit	Symbol	Description
4 (IE.4)	ES	ES = 1 enables serial port interrupt, while ES = 0 will disable it
3 (IE.3)	ET1	ET1 = 1 enables Timer 1 overflow interrupt, while ET1 = 0 will disable it
2 (IE.2)	EX1	EX1 = 1 enables external interrupt 1, while EX1 = 0 will disable it
1 (IE.1)	ET0	ET0 = 1 enables Timer 0 overflow interrupt, while ET0 = 0 will disable it
0 (IE.0)	EX0	EX0 = 1 enables external interrupt 0, while EX0 = 0 will disable it

Note that bit 7 of the IE resistor is called EA (Enable All). This must be set to 1 to permit individual interrupts to be enabled by their respective enable bits. This may also be referred as Global Enable bit. After reset (or power on reset), all interrupts are disabled by default; therefore, they should be enabled by a program before the microcontroller can respond to them.

16.4.2 Interrupt Priority (IP) Register

The details of IP registers are shown in Table 16.3.

Table 16.3 *Interrupt Priority (IP) Register*

--	--	PT2	PS	PT1	PX1	PT0	PX0
MSB							LSB

Bit	Symbol	Description
7 (IP.7)	--	Reserved
6 (IP.6)	--	Reserved
5 (IP.5)	PT2	Priority bit for Timer 2 interrupt (8052 only)
4 (IP.4)	PS	Priority bit for serial port interrupt
3 (IP.3)	PT1	Priority bit for Timer 1 interrupt
2(IP.2)	PX1	Priority bit for external interrupt 1
1 (IP.1)	PT0	Priority bit for Timer 0 interrupt
0 (IP.0)	PX0	Priority bit for external interrupt 0
Priority bit = 1 ; high-priority level		
Priority bit = 0 ; low-priority level		

Using the IP register, each interrupt source can be assigned individually either low or high priority level by clearing or setting the corresponding bit. A low-priority interrupt can be interrupted by a high-level priority interrupt, but vice versa is not true.

THINK BOX 16.2

In the 8051 there are natural priorities assigned to each interrupt source (INT0 being the highest and serial interrupt being the lowest priority). Why does there exist an IP register to assign priorities?

Each source may require different priority (other than natural priority) in different applications, this can be done with IP register, i.e. IP register assigns priority to each source dynamically.

Example 16.1

Write instruction to:

(i) enable Timer 1 and external interrupt 1

(ii) disable external interrupt 1

(iii) disable and enable all interrupts using single instructions

Solution:

(i) To enable Timer 1 and external interrupt 1, first EA bit should be 1 (global enable), and EX1 and ET1 bits in IE register should be programmed to 1.

```
MOV IE, #10001100        // enable Timer1 interrupt and external interrupt 1
                         OR
```

Since IE is also a bit-addressable register, individual bits can be programmed to 1 as shown below:

```
SETB IE.7                // set global enable bit
SETB IE.2                // enable external interrupt 1 and timer 1 interrupt
SETB IE.3
```

(ii)
```
MOV IE, #10000000B       // disable external interrupt 1
OR
CLRB IE.2
```

(iii) To disable all interrupts, EA bit should be programmed as 0, irrespective of the status of all other bits in IE registers.

```
MOV IE, #00XXXXXXB       // disable all interrupts
   OR
CLR IE.7
```

To enable all interrupts, EA as well as individual bits of all interrupts should be set to 1.

```
MOV IE, #10011111B       // enable all interrupts
```

For any interrupt to be serviced, the EA bit and the respective enable bit for that interrupt both must be 1.

While performing critical operations, we may want to deactivate a few or all of the interrupts, so that the critical operation can be completed without any disturbance. This can be achieved by programming IE. The controlling of interrupt operation using IE and IP registers is illustrated in Figure 16.4.

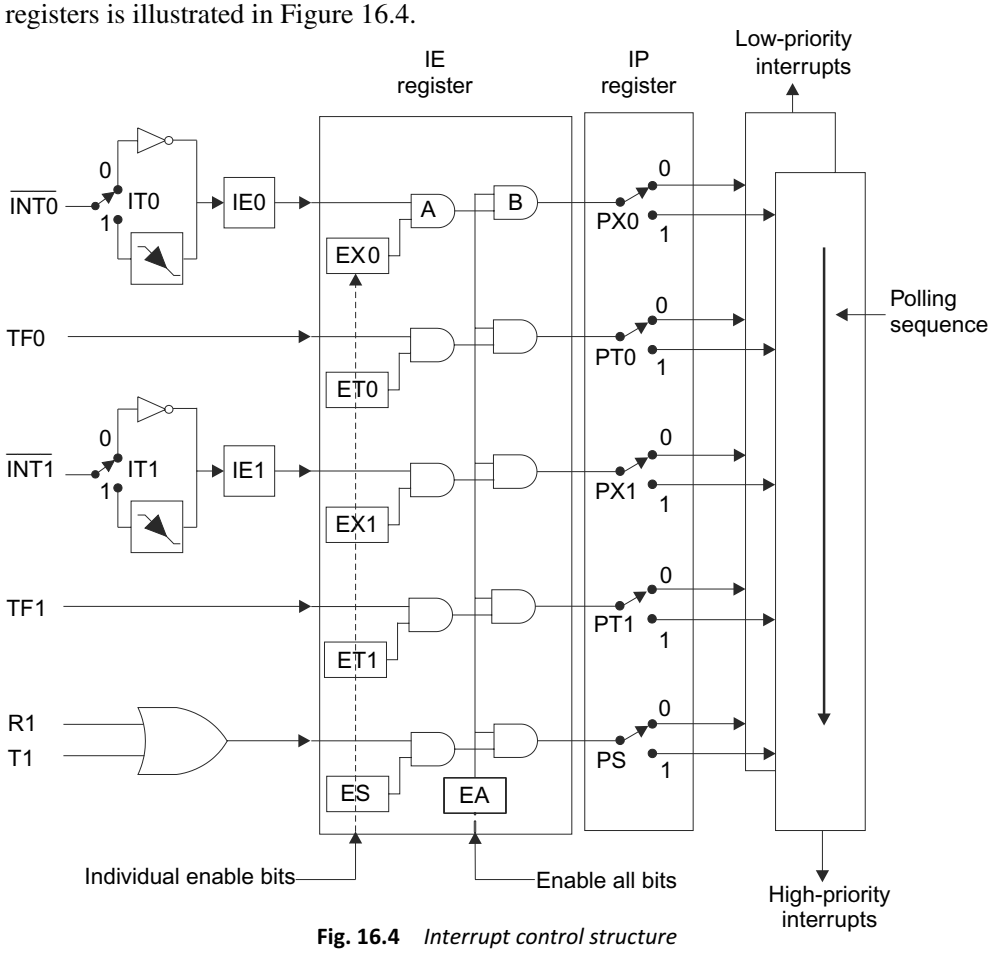


Fig. 16.4 *Interrupt control structure*

As shown in Figure 16.4, to enable any interrupt source, individual enable bit as well as Enable All bit (EA) must be set to 1. If these bits are not set to 1, the AND gates are disabled, thus the interrupt will not be serviced even if it is generated, because interrupt request will not reach to the microcontroller core as the path is blocked by disabled AND gates. For example, assume that external interrupt 0 is asserted, this will set the IE0 flag. To allow this interrupt request to reach the microcontroller core (for sampling and to recognize it), two AND gates designated as *A* and *B* should be enabled. This can be done by setting EX0 and EA bits to 1 because EX0 is connected to one input of gate *A* and EA is connected to gate *B*. If any of these bits are not set then the interrupt request path is blocked. The interrupts can be assigned one of the two priorities, 0 or 1 by programming corresponding bit in IP to 0 or 1. The interrupts which are assigned priority 1 will be polled (and serviced) first in an order shown by the down arrow. Then the interrupts with 0 priority will be polled in the same sequence. Handling of simultaneous interrupt requests is further discussed in topic 16.8.

16.5 | TIMER INTERRUPTS

If timer interrupts are enabled in IE register and timer register overflows, then the corresponding Timer Overflow Flag (TF0 or TF1) will be set to 1, and the microcontroller will jump to the corresponding interrupt vector table (000B for Timer 0 and 001B for Timer 1) and clear the TF0 (or TF1) flag automatically.

16.5.1 Programming of Timer Interrupts

Programming of timer interrupts is illustrated by the following examples.

Example 16.2

Write an assembly-language program to generate square wave of 10 KHz on P2.0 and simultaneously read all pins of P0 and send status of them on to P1 continuously. Assume crystal frequency is 12 MHz.

Solution:

In a given program, we have to perform two different tasks simultaneously, so polling of timer overflow flag cannot be used to generate square wave because it will make the microcontroller busy in monitoring timer overflow flag for majority of the time. Instead of using polling approach, we have to use interrupts to handle such situations more efficiently. We have to write the interrupt service routine for the timer to generate a square wave.

Finding the count for 10 KHz frequency:
Assume Timer 1 is used to generate square wave.
For 10 KHz square wave, ON period = OFF period = 50 μs
Number of timer clock cycles required to generate 50 μs = 50 μs/1 μs = 50
This 50-cycle time delay can be generated by Mode 0, 1, or 2; here, Timer 1 Mode 2 is used.
Therefore, count to be loaded in TH1 = 256 – 50 = 206 = CEH

Follow the steps given below to perform the given task:

- Configure Timer 1 as interval timer in Mode 2.
- Enable Timer 1 interrupt using IE register.
- Load the count in TH1 register for desired delay.
- Start the Timer 1 to generate delay.
- Read continuously P0 and send its contents to P1.
- Define ISR for Timer 1 at its vector address 001BH, the ISR will simply toggle the P2.0 pin every time an interrupt is generated. This will generate a square wave.

```
        ORG 0000H
        LJMP MAIN          // Interrupt Vector Table is bypassed

        ORG 001BH          // ISR of Timer 1
        CPL P2.0           // toggle P2.0 to generate square wave
        RETI               // return to the main program
```

```
        ORG 0100H
MAIN:   MOV TMOD, #20H      // configure Timer 1 as timer in Mode 2
        MOV IE, #88H        // enable Timer 1 interrupts
        MOV TH1, #0CEH      // load count in timer register
        SETB TR1            // start Timer 1
        MOV P0, #0FFH       // configure P0 as input, even though it is by default input after reset, but it will be required
                            // when this program is used as a part of bigger program
AGAIN:  MOV A, P0           // read P0 and send to P1
        MOV P1, A
        SJMP AGAIN          // repeat continuously
        END
```

Note that at the address 0000H, we have the LJMP MAIN instruction. This instruction will jump to the main program after reset and will skip interrupt vector table.

Example 16.3

Rewrite the program of Example 16.2 in C language.

Solution:

The status of Port 0 is sent to Port 1 continuously using the while loop. Whenever Timer 1 interrupt is generated, P2.0 is toggled to generate a square wave.

```
#include <reg51.h>
sbit SWAVE = P2^0;
void timer1 (void) interrupt 3          // ISR for Timer 1
{
SWAVE = ~SWAVE;                         // toggle pin P2.0 to generate square wave
}

void main (void)
    {
        P0=0xFF                         // configure P0 as an input
        TMOD=0x20;                      // initialize Timer 1 in Mode 2
        TH1=0xCE;                       // load count for desired delay
        IE=0x88;                        // enable interrupt for Timer 1
        TR1=1;                          // start Timer 1 to generate delay
    while (1)                           // perform following task forever
        {
        P1=P0                           // read status of P0 and send to P1
         }
    }
```

16.5.2 Simultaneous and Independent use of both the Timers

Both the timers can be programmed independently to perform different tasks. It is illustrated in Example 16.4.

Example 16.4

Write a program to generate square wave of frequencies 500 Hz and 10 KHz on P1.0 and P2.0 respectively. Assume that the crystal frequency is 12 MHz.

Solution:

It is difficult to monitor two timer overflow flags simultaneously such that both timers work independently. Because of the above reason, polling of flags is not preferred. (If polling is used, operation of both timers will not be 'truly' independent, and we will get an error in the output frequency signal, see limitations of polling in topic 16.1). The solution to the above problem is to use interrupts; in this example, we have to

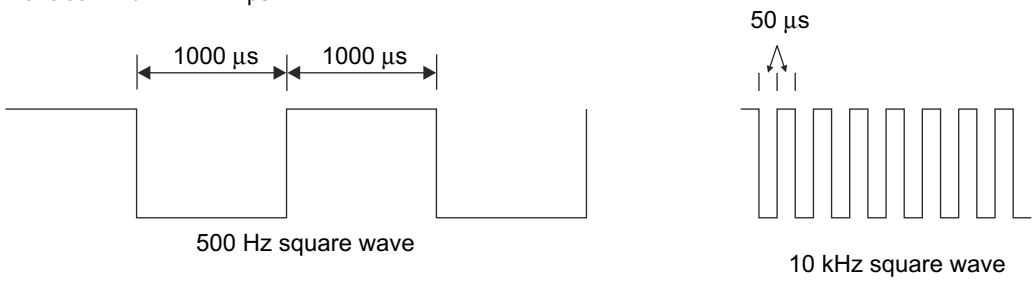
use timer interrupts. Interrupt Service Routines for both Timer 0 and Timer 1 are written on their vector locations. ISRs of both timers will contain overhead instructions (toggling port pins, reloading the count, stop timer, etc.)

Finding the Count for 500 Hz Frequency

Assume that Timer 0 is used to generate this signal.

Timer clock frequency 12 MHz/12 = 1 MHz

Time period of timer clock = 1/1 MHz = 1 μs



For 500 Hz square wave, ON period = OFF period =1000 μs
Number of timer clock cycles required to generate 1000 μs = 1000 μs/1 μs = 1000
This 1000 cycle time delay can be generated by Mode 0 or Mode1, here Timer 0 Mode 1 is used.
Therefore, count to be loaded in TH0, TL0 = 65536 – 1000 = 64536 = FC18H
Therefore, TH0 = FCH, TL0 = 18H

Finding the Count for 10 KHz Frequency

Timer 1 is used to generate this signal.
For 10 KHz square wave, ON period = OFF period = 50 μs
Number of timer clock cycles required to generate 50 μs = 50 μs/1 μs = 50
This 50-cycle time delay can be generated by Mode 0, 1 or 2. Here, Timer 1 Mode 2 is used.
Therefore, count to be loaded in TH1 = 256 – 50 = 206 = CEH
Therefore, TH1 = CEH

The program is developed in the following steps:

- Configure Timer 1 as timer in Mode 2 and Timer 0 in Mode 1.
- Enable Timer 0 and 1 interrupts using IE register.
- Load the count in TH1, TH0-TL0 registers for the desired delay.
- Start both the timers to generate two different delays.
- Wait continuously for timer interrupts to occur.
- Define ISR for Timer 0, the ISR will simply stop timer, toggle the P1.0 pin and reload the count in TH0-TL0 to generate square wave, and finally start Timer 0 again.
- Define ISR for Timer 1, the ISR will simply toggle the P2.0 pin every time the interrupt is generated, this will generate square wave.

```
        ORG 0000H
        LJMP MAIN           // interrupt vector is bypassed by main program

        ORG 000BH           // ISR of Timer 0 at its vector address
        CLR TR0             // stop Timer 0
        CPL P1.0            // toggle P1.0 to generate square wave
        MOV TH0, #0FCH      // reload count in Timer 0
        MOV TL0, #18H
        SETB TR0            // start Timer 0 again
        RETI                // return to the main program

        ORG 001BH           // ISR of Timer 1 at its vector address
        CPL P2.0            // toggle P2.0 to generate square wave
        RETI                // return to the main progrm
```

```
        ORG 0100H           // main program starts at address 100H
MAIN:   MOV TMOD, #21H      // Timer 1 Mode 2, Timer 0 Mode 0
        MOV IE, #8AH        // enable Timer 0 and 1 interrupts
        MOV TH0, #0FCH      // load count in the timer registers
        MOV TL0, #18H
        MOV TH1, #0CEH
        ORL TCON, #50H      // start both timers without affecting other bits equivalent to SETB TR0 and SETB TR1
HERE:   SJMP HERE           // wait until any of the timer overflows
        END
```

Analysis of the Program

First two lines of the program listing are

```
ORG 0000H
LJMP MAIN
```

Whenever we use interrupts (and therefore, ISRs), the area reserved for interrupt vector table [0003H to 0025 (0023 + 2 bytes, at least)] must be bypassed for the main program.

Eight bytes are reserved for each interrupt's ISR. We can directly write ISR in vector table only if the size of the ISR is at most eight bytes including the RETI instruction. Otherwise, the jump instruction (preferably LJMP) should be written to point to the ISR written elsewhere, which will increase the interrupt response time or interrupt latency.

In our program, we have directly written ISR for Timer 0 into the vector table even though its size is 9 bytes because the external interrupt 1 is not used in the program, so, memory reserved for it is used by Timer 0 ISR.

ISR for Timer 0 Mode 1 contains instructions for reloading count, clear TF0, TR0, and restarting timer. These overhead instructions will not be required in Timer 1 Mode 2 because it is on auto-reload mode. Use of the instruction ORL TCON, #50 for starting both timers simultaneously is a more efficient way because the OR operation does not disturb other bits.

Simulation Result (In Keil µVision 4.0 IDE)

The square-wave output on port pins can be observed in the logic analyzer window. Open the logic analyzer window from **View→ Analysis widows → Logic analyzer window**. The snapshot of the output is shown below.

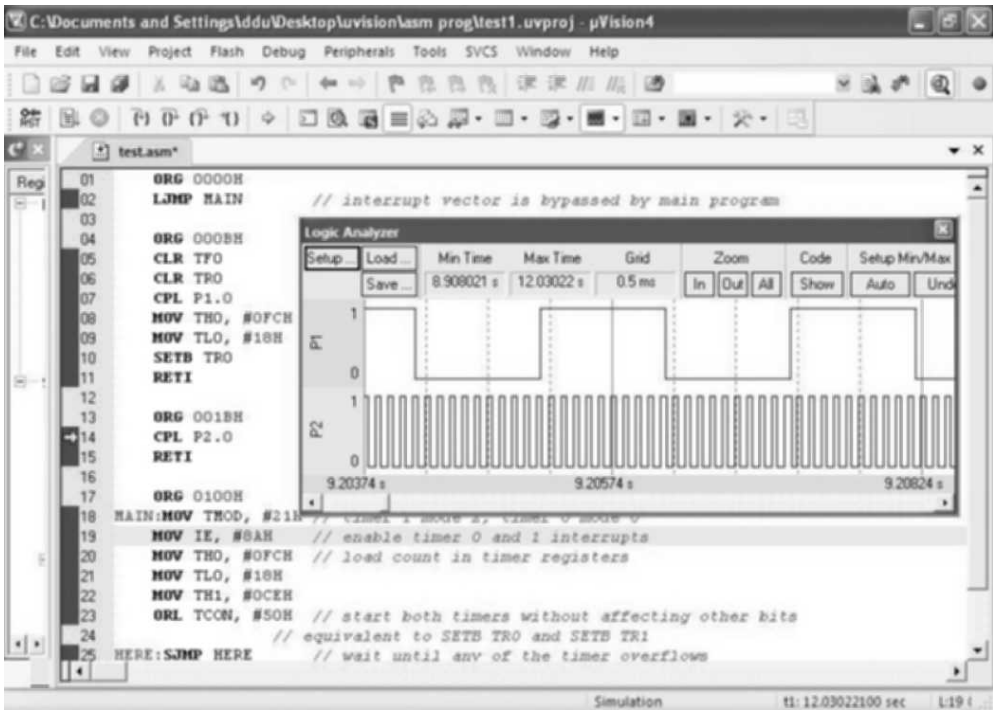


Fig. 16.5 *Output window for Example 16.4*

Example 16.5

Rewrite the program of Example 16.4 in the C language.

Solution:

ISRs are written as a function with the keyword 'interrupt' as shown.

```
# include <reg51.h>
sbit P1_SQUARE = P1^0;
sbit P2_SQUARE = P2^0;

void timer0 (void) interrupt 1                  // ISR for Timer 0
        {
                TR0 = 0;                        // stop timer
                P1_SQUARE = ~ P1_SQUARE;        // toggle P1.0  to get square wave
                TH0 = 0xFC;                     // reload count in TH0-TL0
                TL0 = 0x18;
                TR0 = 1     ;                   // start Timer 0
        }
void timer1 (void) interrupt 3                  // ISR for Timer 1

        {
                P2_SQUARE = ~ P2_SQUARE;        // toggle P2.0 to get square wave
        }
void main ()
        {
                TMOD = 0x21;                    // Timer 1 Mode 2, Timer 0 Mode 1
                IE = 0x8A;                      // enable timer 0 and 1 interrupts
                TH0 = 0xFC;                     // load count in timer registers
                TL0 = 0x18;
                TH1 = 0xCE;
                TCON = TCON | 0x50;             // start both timers
                while (1);                      // loop here for ever
        }
```

Example 16.6

Write an assembly-language program that continuously reads the status of P2.0 pin and sends it to P0.0 and simultaneously generate the square wave of 2 KHz on pin P1.0. Assume the crystal frequency to be 11.0592 MHz.

Solution:

Timer 0 is used in auto-reload mode, to generate square wave of 2 KHz we have to toggle pin P1.0 every 250 μs.
Timer clock frequency = 11.0592 x 10^6 /12 = 921.6 KHz
Time period of timer clock cycle = 1.085 μs
No. of cycles required to generate 250 μs = 250/1.085≈ 230
Therefore count to be loaded in TH0 = 256 –230 = 26 = 1AH

The steps of program development are similar to that of Example 16.2 except the following step.
Read status of P2.0 and send it to P0.0 instead of reading all port pins.

```
        ORG 0000H
        LJMP MAIN           // avoid using Interrupt Vector Table for main program
        ORG 000BH           // ISR for Timer 0 interrupt
        CPL P1.0            // complement P1.0 bit to generate square wave
        RETI                // return from ISR to the main program
```

```
        ORG 0030H           // start main program after interrupt vector table
MAIN:   MOV TMOD, #02H      // initialize Timer 0 in Mode2
        SETB  P2.0          // configure P2.0 as input
        MOV TH0, #1AH       // load timer count
        MOV IE, #82H        // enable Timer 0 interrupt
        SETB TR0            // start Timer0
BACK:   MOV C, P2.0         // read status of P2.0
        MOV P0.0, C         // send it on P0.0
        SJMP BACK           // repeat continuously
        END
```

Example 16.7

Rewrite the program of Example 16.6 in the C language.

Solution:

The status of pin P2.0 is sent to pin P0.0 continuously using the while loop. Whenever Timer 0 interrupt is generated, P1.0 is toggled to generate a square wave.

```
#include <reg51.h>
sbit ibit = P2^0;
sbit obit = P0^0;
sbit SWAVE = P1^0;

void timer0 (void) interrupt 1          // ISR for Timer 0
{
SWAVE = ~SWAVE;                         // toggle pin p1.0 to generate square wave
}
void main (void)
    {
        ibit = 1;                       // make P2.0 input
        TMOD = 0x02;                    // initialize Timer 0 in Mode2
        TH0 = 0x1A;                     // load count
        IE = 0x82;                      // enable interrupt for Timer 0
        TR0 = 1;                        // start Timer 0
    while (1)
        {
        obit = ibit;                    // read P2.0 and send to P0.0
        }
    }
```

Example 16.8

Write a C program using interrupts to perform two different tasks simultaneously.

(i) Count the number of 1 Hz pulses applied to Timer input pin T1 and display it on P2(LSByte of count) and P0 (MSbyte of count)

(ii) Generate a square wave of 2 KHz on P1.0.

Solution:

The counting of pulses on the T1 pin can be done by configuring Timer 1 as a 16-bit counter and configure the T1 pin (P3.5) as an input, the Timer 1 registers are initialized with value 0000. Once Timer 1 is started, it will increment Timer 1 register contents after every input pulse. The values of Timer 1 registers are continuously sent to P0 and P2. The square wave is generated by using Timer 0 as an interval timer in Mode 2. The ISR of Timer 0 will toggle P1.0 every time a Timer 0 interrupt is generated.

```
#include<reg51.h>
```

```
sbit SWAVE = P1^0;                  // define P1.0 as SWAVE
sbit COUNTER_IN = P3^5;             // define P3.5 as COUNTER_IN
void timer0 (void) interrupt 1      // Timer 0 ISR
        {
        SWAVE = ~SWAVE;             // toggle pin p1.0 to generate square wave
        }
void main (void)
{       TL1 = 0x00;                 // initialize counter with value 0000H
        TH1 = 0x00;
        TMOD = 0x52;                // Timer 1 as counter in Mode1 and Timer 0 as timer in Mode 2
        TH0 = 0x1A;                 // count for 2 kHz square wave (See Example 16.6)
        IE = 0x82;                  // enable Timer0 interrupt
        COUNTER_IN = 1;             // configure Timer 1 input pin as an input
        TR0 = 1;                    // start timers for counting and delay generation
        TR1 = 1;
        while (1)                   //  repeat task forever
                {
                P2 = TL1;           // display count on P0 and P2
                P0 = TH1;
                }
}
```

Example 16.9

Write a program to generate square wave of frequencies 2 KHz, 4 KHz and 100 Hz on P1.0, P1.1 and P1.2 respectively. Assume that the crystal frequency is 12 MHz.

Solution:

Since we require three independent timing tasks, we will use Timer 0 in Mode 3 (it will be used to generate 2 KHz and 4 KHz signals) and Timer 1 in Mode 1 to generate a 100 Hz signal. Since all the three tasks should be independent, we should use interrupts: 2 KHz and 4 KHz signals are generated using interrupts, and 100 Hz signal is generated using polling method.

Finding the Count for 2 KHz and 4 KHz Frequency

Timer 0 is used in Mode 3 to generate these signals.
TL0 is used to generate a 2 KHz signal.
For a 2 KHz square wave, ON period = OFF period = 250 μs
Number of timer clock cycles required to generate 250 μs = 250 μs/1 μs = 250
Therefore, count to be loaded in TL0 = 256 – 250 = 06
Therefore, TL0 = 06H
Similarly, TH0 is used to generate a 4 KHz signal.
For a 4 KHz square wave, ON period = OFF period = 125 μs
Number of timer clock cycles required to generate 125 μs = 125 μs/1 μs = 125
Therefore, count to be loaded in TL0 = 256 - 125 = 131 = 83H
Therefore, TH0 = 83H

The Count for 100 Hz Frequency

Timer 1 in Mode 1 is used for this signal.
For 100 Hz square wave, ON period = OFF period = 5000 μs
Number of timer clock cycles required to generate 5000 μs = 5000 μs/1 μs = 5000
Therefore, count for desired delay is 5000 = 1388H (Note that we do not subtract this count from 65536 because when Timer 0 is in Mode 3, we do not have TF1 to determine overflow of Timer 1.

```
        ORG 0000H
        LJMP MAIN               // interrupt vector is bypassed by the main program
```

```
            ORG 000BH           // ISR of Timer 0 at its vector address
            CLR TF0             // clear Timer TL0 overflow flag
            CLR TR0             // stop Timer TL0
            CPL P1.0            // toggle P1.0 to generate square wave
            MOV TL0, #06H       // reload count in Timer 0
            SETB TR0            // start timer TL0 again
            RETI                // return to the main program

            ORG 001BH           // ISR of Timer 1 at its vector address
            CLR TF1             // clear Timer TH0 overflow flag
            CLR TR1             // stop Timer TH0
            CPL P1.1            // toggle P1.1 to generate square wave
            MOV TH0, #083H      // reload count in Timer TH0
            SETB TR1            // start Timer TH0 again
            RETI                // return to the main program

            ORG 0100H           // main program starts at address 100H
MAIN:       MOV IE, #8AH        // enable timer interrupts
            MOV TL0, #06H       // load count for Timer TL0
            MOV TH0,#83H        // load count for Timer TH0
            MOV 20H, #88H       // count for desired delay of 5000 µs -LSByte
            MOV 21H, #13H       // count for desired delay of 5000 µs -MSByte
            MOV TMOD, #33H      // both timers in Mode 3, Timer 1 operation in
                                // Mode 3 will be stopped - it will hold the count
BACK:       MOV TH1, #00H       // load initial value in Timer 1 registers
            MOV TL1, #00H
            ANL TMOD, #1FH      // start Timer 1 in Mode 1 when Timer 0 is in
                                // Mode 3
            ORL TCON, #50H      // start TL0 and TH0 timers without affecting the other
                                // bits equivalent to SETB TR0 and SETB TR1

REPEAT:     MOV A, TH1          // read the Timer 1 value, avoid reading count
            MOV R2, TL1         // when TL1 rollover from FFH to 00H
            CJNE A, TH1, REPEAT
            MOV R3, TH1         // compare the 16-bit timer value with the count
            CLR C               // for desired delay
            MOV A, R2
            SUBB A, 20H
            MOV A, R3
            SUBB A, 21H
            JC REPEAT           // monitor timer value until it exceeds count of desired delay
            ORL TMOD, #30H      // stop Timer 1, configuring Timer 1 in Mode 3
                                // will stop it
            CPL P1.2            // complement port pin to get square wave
            SJMP BACK           // repeat the process forever
            END                 // end of program
```

The limitation of this program is that we will get a timing error because of the overhead instructions.

16.6 | EXTERNAL INTERRUPTS

The pins $\overline{\text{INT0}}$ (P3.2, pin 12) and $\overline{\text{INT1}}$ (P3.3, pin 13) are used as external interrupt input pins. When these pins are activated (made low), the 8051 is interrupted and jumps to interrupt vector table of the corresponding interrupt. The external interrupts can be configured to be either level-triggered or transition (edge) triggered interrupts.

16.6.1 Level-Triggered External Interrupts

Level-triggered interrupt will be activated by providing low level on the interrupt pins. This is a default mode after reset. Writing 0 to ITX bits (in TCON register) configures external interrupt in level-triggered mode. Interrupts in this mode are not latched, i.e. external source that generates the interrupt; controls directly the IEX interrupt flag (in TCON register). The low level must be removed before execution of the last instruction (RETI) of the ISR, otherwise same low level will be misinterpreted as another interrupt. The system designer should take care for this issue. The low level should be maintained until the start of the execution of ISR.

THINK BOX 16.3

How can the unwanted multiple processing of the same low level-triggered interrupt be avoided?

Set the interrupting signal high before leaving the ISR. This can be done using a flip-flop. Connect the interrupting signal at the input of flip-flop and output of flip-flop to $\overline{\text{INTX}}$ pin. Connect a port pin of the 8051 to PRESET input of the flip-flop. Just before the RETI instruction gives positive pulse to the PRESET pin (negative pulse if PRESET is active low), this will make $\overline{\text{INTX}}$ pin high.

The other method is using software delay. Place a delay before the RETI instruction. The delay should be larger than the time for which interrupting signal remains low.

16.6.2 Transition (Edge) Triggered External Interrupts

Edge-triggered interrupt will be activated by providing high-to-low level transition on interrupt pins $\overline{\text{INTX}}$. Writing 1 to ITX bits (in TCON register) configures an external interrupt in the edge-triggered mode. High-to-low level transition on $\overline{\text{INTX}}$ pin will set IEX flag (in TCON) and will interrupt the microcontroller and automatically jump to vector address of the corresponding interrupt source and then the ISR is executed. It should be noted that IEX flag is automatically cleared by hardware to 0 when program execution is vectored to ISR.

At the end of the ISR, execution of RETI will resume the program execution at the place where it was interrupted. Since, during execution of an ISR, all interrupts of the same and low priority are blocked, any further interrupt (level transition) on the same pin $\overline{\text{INTX}}$ is ignored until execution of the RETI instruction.

The generation of transition-triggered external interrupt 1 is illustrated in Figure 16.6.

Figure 16.6 shows a simple circuit in which the voltage on pin $\overline{\text{INT1}}$ (P3.3, Pin 13) is normally 5 V. Pressing the switch will cause a high-to-low level transition as the voltage changes from 5 V to 0 V; therefore, the interrupt is generated and the controller is vectored to the address 0013H. If the switch is pressed and held then the logic 0 level is held on P3.3; this will not have any effect on interrupt operation because the interrupt will only be generated if there is high-to-low level transition (key bouncing must be considered here, refer topic 17.1.1 for more details of bouncing).

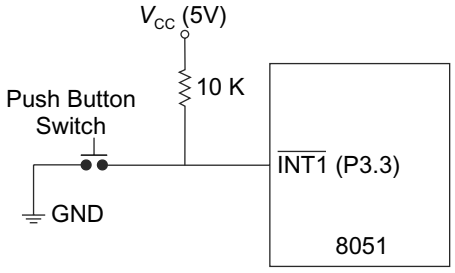


Fig. 16.6 *Generation of external interrupt 1 ($\overline{\text{INT1}}$)*

Example 16.10

Write an assembly-language program to count the number of times an external interrupt 1($\overline{\text{INT1}}$) occurred (see Figure 16.6). Also send the count on Port 2.

Solution:

We will use one register to hold the count and initialize the count with 00, the count will be incremented every time the program enters into ISR. This will effectively count number of times an external interrupt 1 is asserted. The ISR will also send content of this counter register to Port2.

```
        ORG 0000H
        LJMP MAIN          // avoid using Interrupt Vector Table for main program
        ORG 0013H          // ISR of external interrupt 1, INT1
        INC R0             // increment the count every time ISR is entered
        MOV P2, R0         // send count on P2
        RETI               // return to the main program

        ORG 0030H          // start main program after Interrupt Vector Table
MAIN:   MOV R0, #00H       // counter to count no. of times interrupt is occurred
        MOV IE, #84H       // enable external interrupt 1
        SETB TCON.2        // configure ext. interrupt 1 as edge-triggered
HERE:   SJMP HERE          // wait for interrupt to occur
                           END
```

Note: We need to add delay in ISR before returning to the main program. Otherwise the same interrupt will be processed again because the time required to execute ISR is very less compared to the time required to press and release the key. (Refer topic 17.1.1).

Simulation Procedure (In Keil µVision 4.0)

The application of an external interrupt (pulse) can be simulated as follows:

- Free run (or single step) the program.
- When the program is waiting for external interrupt to occur, i.e. it is executing instruction 'HERE: SJMP HERE', the external interrupt may be applied.
- Open peripheral windows of Port 2 and 3 from **Peripherals→ I/O ports** menu.
- Pin P3.3 is an external interrupt input pin ("INT1"); therefore, we should apply interrupt pulses at this pin.
- Click repetitively on the box of P3.3. This will be considered as an external interrupt pulse. Note that the edge-triggered interrupts are generated on a negative edge of the pulse. A single pulse can be simulated by a pair of clicks, i.e. check the box by clicking on it and clear the box by clicking on it again.

Example 16.11

Rewrite the program of Example 16.10 in the C language.

Solution:

```
#include<reg51.h>
        void ext1 (void) interrupt 2        // external interrupt 1 ISR
                {
                count ++;                   // increment the count every time ISR is entered
                P2 = count;                 // send count on P2
                }

        void main (void)
                {
                unsigned char count = 0;    // counter to count no. of times interrupt is
                                            // occurred
                IE = 0x84;                  // enable external interrupt 1
                TCON| = 0x04;               // configure ext. interrupt 1 as edge triggered
                while (1);                  //  wait for interrupt to occur
                }
```

16.6.3 Pulse Generation using External Interrupt

The circuit of Figure 16.6 can be modified to generate a pulse of any amplitude and width. It is shown in Figure 16.7.

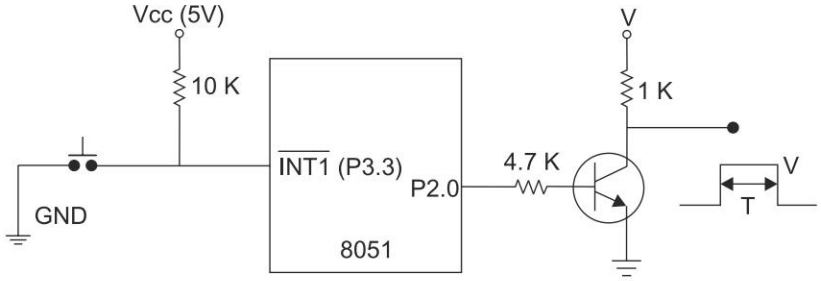


Fig. 16.7 *Pulse generation using external interrupt*

When the pushbutton switch is pressed, it will ground $\overline{\text{INT1}}$ pin and an interrupt will be generated (in real life, instead of pressing pushbutton, the interrupt may be triggered by some external event) and execution will be vectored to ISR of $\overline{\text{INT1}}$, which will contain pulse generation subroutine. (Clear P2.1, wait for a predefined delay corresponding to the desired pulse width and set P2.1. Note that P2.1 = 0 is required for output pulse to be high because of inverter action of the transistor circuit.) For a very small delay, use maximum oscillator frequency. The smallest delay can be the time required to execute two instructions (SETB P2.0 and CLRB P2.0) plus interrupt latency of the 8051. For a very large delay, use minimum oscillator frequency and/or generate maximum delay using software or timers.

The amplitude of pulse will depend upon the supply voltage (V) connected to the transistor inverter.

The program to generate a pulse of desired width is given below.

```
            ORG 0000H
            LJMP MAIN          // avoid using Interrupt Vector Table for main program
            ORG 0013H          // ISR of external interrupt 1, INT1
            CLR P2.0           // low to high part (rising edge) of pulse
                               // opposite logic is used because of inverter action
            LCALL DELAY        // delay for desired width (assume Delay routine is available)
            SETB P2.0          // high to low part (falling edge) of pulse
            RETI               // return to the main program
            ORG 0030H          // start main program after Interrupt Vector Table
MAIN:       MOV IE, #84H       // enable external interrupt 1
            SETB TCON.2        // configure ext. interrupt 1 as edge-triggered
HERE:       SJMP HERE          // wait for interrupt to occur
            END
```

16.6.4 Sampling of Edge-Triggered Interrupts

Since interrupts are sampled every machine cycle (12 oscillator periods), to recognize edge-triggered interrupt, the $\overline{\text{INTX}}$ pin must be kept high for at least 1 machine cycle and then low for at least 1 cycle as shown in Figure 16.8.

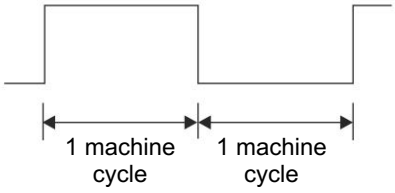


Fig. 16.8 *Minimum time period for detection of edge-triggered interrupt*

THINK BOX 16.4

How can we connect active high (high-level) interrupt signals with $\overline{\text{INTX}}$ pins?

Use NOT gate between interrupt signal and $\overline{\text{INTX}}$ pin.

16.7 | SERIAL PORT INTERRUPTS

The 8051 serial port has transmitter and receiver subsystems. A serial port interrupt is generated when either Transmit Interrupt flag (TI) or Receive Interrupt flag (RI) is set to 1. The transmit and receive interrupts are combined (ORed) to generate a common serial port interrupt as shown in Figure 16.4. The transmit interrupt is generated when transmission of a byte written to SBUF register is completed, i.e. when the stop bit is being transferred. It indicates that transmit buffer is empty and the next byte can be placed in it to begin the next byte transmission; because of this, TI is also known as Transmitter Empty Interrupt (TEI) flag. The receive interrupt is generated when a byte or character is received in the receive buffer, i.e. when the stop bit is being received. It indicates that a data byte is received and should be read quickly prior to being replaced by a new data byte.

As there are two different sources [either Transmitter (TI) or Receiver (RI)] for a common serial port interrupt, these flags are not cleared automatically when microcontroller vectors to ISR, but ISR should check whether TI or RI is the source of the interrupt, and clear the interrupting flag using software instruction. This is somewhat the opposite with the timer and edge-triggered external interrupts, where interrupt flags are automatically cleared when the microcontroller is vectored to ISR. The use of TI and RI flags in transmitting and receiving character byte(s) is illustrated in the following examples.

Example 16.12

Write a program that reads incoming data from serial port (RXD pin) at Baud rate 9600 and sends it to Port 1. Assume the crystal frequency to be 11.0592 MHz.

Solution:

The following operations are performed for the given task:

- Configure Timer 1 as an interval timer in Mode 2 using TMOD register.
- Load value into TH1 to get the desired Baud rate.
- Configure SCON register in Mode 1(8-bit data, 1 stop bit).
- Start Timer 1, this will generate clock at the desired Baud rate
- Wait until RI flag is set, RI = 1 indicates that character is received in SBUF.

Define ISR for serial port interrupt, which will perform the following operations:

- Check whether RI or TI is the cause of interrupt generation.
- If it was TI, clear it and return to the main program.
- Otherwises read SBUF and send its contents at Port 1.
- Clear RI flag, so that it can be monitored to check reception of the next character and return to the main program.
- Wait for the reception of the next character.

```
        ORG 0000H
        LJMP MAIN       // avoid using Interrupt Vector Table for main program
        ORG 0023H       // serial port ISR at vector address 0023
        JNB RI, SKIP    // if RI is low, go to skip (same interrupt is also generated when TI is set)
        MOV A, SBUF     // otherwise, read received serial data from SBUF register
        MOV P1, A       // send it to Port1
        CLR RI          // clear RI to enable reception of next character
        RETI            // return to the main program
SKIP:   CLR TI          // clear TI
        RETI            // return to the main program

        ORG 100H
MAIN:   MOV TMOD, #20H  // initialize Timer1 in Mode 2
        MOV TH1, #0FDH  // load count to get 9600 baud rate
        MOV SCON, #50H  // serial port in Mode 1
        MOV IE, #90H    // enable serial interrupt
        SETB TR1        // start Timer1 to generate clock at baud rate
HERE:   SJMP HERE       // wait here until a character is received
        END
```

The More Efficient Method

The more efficient way to write the same ISR is to use JBC instruction instead of JNB instruction. The use of JBC instruction saves the use of CLR RI (extra CLR instruction) and is, therefore, more efficient. The same ISR can be rewritten using JBC instruction as follows:

```
        ORG 0023H       // serial port ISR at vector address 0023
        JBC RI, SKIP    // if RI is set (the serial interrupt is generated because of reception of a character), clear RI (to
                        // enable reception of next character) and jump (proceed) to read SBUF
        CLR TI          // otherwise serial interrupt is generated because of completion of transmission of character, clear TI to
                        // enable transmission of the next character
        RETI            // return to the main program
SKIP:   MOV A, SBUF     // read received serial data from SBUF register
        MOV P1, A       // send it to Port1
        RETI            // return to the main program
```

Note that the second method requires one less instruction (CLR RI) because this is performed by the JBC instruction itself!

Simulation Procedure (In Keil µVision 4.0 IDE)

The reception of a data byte is simulated using the virtual register SxIN. The steps to simulate reception of a data byte are given as follows:

- Free run (or single step) the program.
- When the program is waiting to receive a byte, i.e. when it is executing the instruction "HERE: SJMP HERE", the byte can be given to the UART using SxIN virtual register (x represents UART number if there are more than one UARTs in a device; in case of 8051 there is only one UART, therefore, we have to use the name SIN).
- To give data byte 45H, type command SIN = 0x0045 in the command window. (In general, the format is SIN = 0x00XX, where XX represents the data byte in hex.)
- Execute the command by pressing Enter key. It will simulate the reception of data byte; the received byte can be read from the SBUF register.

Note that we can give the ASCII byte directly to UART using command SIN = 'ASCII code'.

Example 16.13

Rewrite the program of Example 16.12 in the C language.

Solution:

```
#include<reg51.h>
void serial (void) interrupt 4                  // serial port  ISR
        {
        if (RI !=1)                             // If RI = 0, TI has generated interrupt
                TI = 0;                         // clear TI
                else                            // If R1 = 1, perform the following task
                        {
                        P1 = SBUF;              // read SBUF and send to P1
                        RI = 0;                 // clear RI to detect reception of the next byte
                        }
                }

        void main (void)
        {
                TMOD = 0x20;                    // use Timer 1 in 8-bit auto-reload mode
                TH1 = 0xFD;                     // set 9600bps baud rate
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR1 = 1;                        // start timer to generate clock
                IE = 90H;                       // enable serial interrupt

                while (1);                      // wait here until character is received
        }
```

Example 16.14

Write a program that continuously reads Port 2 and sends it to Port 0, and simultaneously also reads incoming data from serial port (RXD pin) at baud rate 9600 and sends it to Port 1. Assume the crystal frequency to be 11.0592 MHz.

Solution:

```
        ORG 0000H
        LJMP MAIN           // avoid using Interrupt Vector Table for main program
        ORG 0023H
        JNB RI, SKIP        // if RI is low, go to skip (same interrupt is also generated when TI is set)
        PUSH ACC            // save A on the stack
        MOV A, SBUF         // otherwise, read received serial data from SBUF register
        MOV P1, A           // send it to Port1
        CLR RI              // clear TI
        POP ACC             // retrieve A from stack
        RETI                // return to the main program
SKIP:   CLR TI              // clear TI
        RETI                // return to the main program

        ORG 100H
MAIN:   MOV P2, #0FFH       // configure P2 as an input port
        MOV TMOD, #20H      // initialize Timer1 in Mode 2
        MOV TH1, #0FDH      // load count to get 9600 baud rate
        MOV SCON, #50H
        MOV IE, #90H        // enable serial interrupt
        SETB TR1            // start Timer 1
BACK:   MOV A, P2           // read data from Port 2
        MOV P0, A           // send it to Port 0
        SJMP BACK           // repeat continuously
        END
```

Example 16.15

Write a C program that simultaneously performs the following operations continuously:
(i) Reads data from P1.0 and sends it to P2.0, (ii) Generate square wave of 2.5 kHz on pin P2.1
(iii) Sending numbers '0' to '9' to the serial port repeatedly
Assume the crystal frequency to be 11.0592 MHz.

Solution:

Timer 0 is used in Mode 2 to generate square wave.

```
#include<reg51.h>
sbit ibit = P1^0;
sbit obit = P2^0;
sbit SWAVE = P2^1;
unsigned char ch = '0';

void timer0 (void) interrupt 1          // ISR for Timer 0 to generate square wave
{
SWAVE = ~SWAVE;
}
void serial (void) interrupt 4          // ISR to transmit data on serial port
{
if (TI==1)                              // send 0 to 9 continuously
        {
        ch++;                           // next character
```

```
                if (ch > '9')                   // send 0 after sending 9
                ch = '0';
                TI = 0;                         // clear TI before sending the next number
                SBUF = ch;                      // place updated character ch into SBUF
                }
else
                {
                RI = 0;                         // clear RI
                }
}

void main (void)                                // main program
{
                ibit = 1;                       // configure P1.0 as input
                TH1= 0xFD;                      // 9600 baud rate
                TMOD = 0x22;                    // initialize both timers in Mode 2
                TH0 = 72;                       // load count for 2.5 kHz (47H)
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR0 = 1;                        // start Timer 0
                TR1 = 1;                        // start Timer 1 to generate clock
                IE = 0x92;                      // enable Timer 0 and serial interrupts
                SBUF = ch;                      // place character ch into SBUF
                while (1)
                {
                obit = ibit;                    // read P1.0 and sends it to P2.0
                }
}
```

Simulation Result (In Keil µVision 4.0 IDE)

The transmitted data using UART can be observed in the serial output window. Open serial windows from **View → Serial windows→ UART#1** menu. The square-wave output on port pins can be observed in the logic analyzer window. Open the logic analyzer window from **View → Analysis widows → Logic analyzer window**. The snapshot of the output is shown as follows.

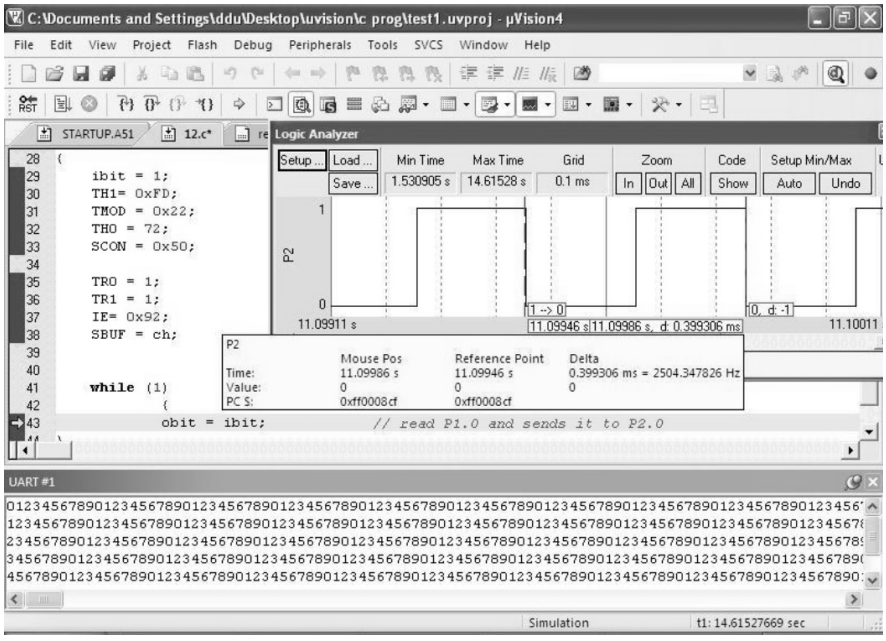


Fig. 16.9 *Output window for Example 16.5*

Example 16.16

Write a C program that simultaneously performs the following operations continuously:
(i) Reads data from P1.0 and sends it to P2.0
(ii) Generate square wave of 2.5 KHz on pin P2.1
(iii) Receive data serially and send it to Port 0
Assume the crystal frequency to be 11.0592 MHz

Solution:

Timer 0 is used in Mode 2 to generate a square wave.

```
#include<reg51.h>
sbit ibit = P1^0;
sbit obit = P2^0;
sbit SWAVE = P2^1;

void timer0 (void) interrupt 1                  // ISR for Timer 0 to generate square wave
{
SWAVE = ~SWAVE;
}
void serial (void) interrupt 4                  // ISR to receive data from serial port
{
        if (TI==1)
                {
                TI = 0;                         // clear interrupt (TI)
                }
        else
                {
                P0=SBUF;                        // put value on Port 0
                RI=0;                           // clear RI to be ready to receive next byte
                }
}

void main (void)                                // main program
{
        ibit = 1;                               // configure P1.0 as input
        TH1= 0xFD;                              // 9600 baud rate
        TMOD = 0x22;                            // configure both timers in Mode 2
        TH0 = 72;                               // load count for 2.5 KHz (47H)
        SCON = 0x50;                            // 8-bit data, 1 stop bit, REN enabled
        TR0 = 1;                                // start Timer 0 to generate square wave
        TR1 = 1;                                // start Timer 1 to generate clock
        IE= 0x92;                               // enable Timer 0 and serial interrupts
        while (1)                               // repeat task continuously
        {
        obit = ibit;                            // read P1.0 and sends it to P2.0
        }
}
```

16.8 | INTERRUPT PRIORITIES

When two or more interrupts occur simultaneously, there will be a dilemma as to which of these should be responded first? To resolve this problem, priorities are given to all the interrupts. In the 8051, there is two-tier priority structure. In the first tier, there are two levels of priorities, 'high' or 'low' which may be assigned using the IP register as discussed earlier. Writing '1' to the corresponding bit in IP will assign higher priority and '0' will assign lower priority. When the interrupts of different levels are generated simultaneously, the interrupt with 'high' priority is serviced first.

Furthermore, within each level, if two or more interrupts occur at the same time, then the second-tier priority is used. In the second tier, they will be serviced as per natural priority of the 8051 as shown in Table 16. 4.

Table 16.4 *Natural interrupt priorities*

Interrupt Source	Priority
External Interrupt 0 ($\overline{\text{INT0}}$)	Highest
Timer Interrupt 0 (TF0)	↓
External Interrupt 1($\overline{\text{INT1}}$)	↓
Timer Interrupt 1(TF1)	↓
Serial Port Interrupt (RI or TI)	Lowest

These are the priorities assigned upon reset and indicate the internal polling sequence of the interrupts by the 8051. It should be noted that second-tier priority (priority within level) is used only (to decide priority) when two or more interrupt requests of the same priority level are asserted simultaneously.

Example 16.17

Assign highest priority to Timer 0 and lowest priority to $\overline{\text{INT0}}$.

Solution:

```
MOV IP, #00011110
```

This instruction will assign priority 'high' to Timer 0 (TF0), external interrupt 1 (IE 1), Timer 1 (TF1), and serial port (TI or RI) interrupts and among these 'high' priority level interrupts, Timer 0 (TF0) has the highest priority (only for this example!) and then $\overline{\text{INT1}}$, TF1 and serial port interrupt in decreasing order. Since external interrupt 0 ($\overline{\text{INT0}}$) is assigned 'low' priority it will be given the lowest priority.

Example 16.18

Assign highest priority to $\overline{\text{INT1}}$ and lowest priority to Timer 0 interrupt.

Solution:

```
MOV IP, #00011100
```

This instruction will assign priority 'high' to external interrupt 1 (IE 1), Timer 1 (TF1), and serial port (TI or RI) interrupts and among these 'high' priority level interrupts, $\overline{\text{INT1}}$ has the highest priority and then the TF1 and serial port interrupt in decreasing order. Since timer interrupt 0 is assigned 'low' priority, it will be given the lowest priority.

16.9 | NESTED AND MULTIPLE INTERRUPTS

What will be the response of the 8051 when the ISR of one interrupt is in progress and another interrupt is generated?

Here, a low-priority interrupt can be interrupted by high-priority interrupt but not by another low or same-priority interrupt. They will be serviced only after completion of high-priority interrupts. Example 16.19 will help clarify the microcontroller behaviour (and response) to multiple interrupts.

Example 16.19

Discuss for the following situations the sequence in which the interrupts are serviced. Assume that the external interrupts are configured as edge-triggered.

(a) Assume that external interrupt 1 ($\overline{\text{INT1}}$) ISR is being executed and external interrupt 0 ($\overline{\text{INT0}}$) is asserted. Consider IP = 00H (reset value).
(b) Assume that external interrupt 1 ($\overline{\text{INT1}}$) ISR is being executed and external interrupt 0 ($\overline{\text{INT0}}$) is asserted. Consider IP = 01H
(c) Assume that Timer 0 (TF0) ISR is being executed and external interrupt 0 ($\overline{\text{INT0}}$) is asserted. If IP = 00H (reset value).
(d) Assume that Timer 0 (TF0) ISR is being executed and external interrupt 0 ($\overline{\text{INT0}}$) is asserted. If IP = 02H.
(e) Assume that Timer 0 (TF0) ISR is being executed and external interrupt 0 ($\overline{\text{INT0}}$) is asserted. If IP = 01H.
(f) Assume that external interrupts 0 ($\overline{\text{INT0}}$) as well as ($\overline{\text{INT1}}$) are asserted simultaneously. IP = 00H (reset value).
(g) Assume that external interrupts 0 ($\overline{\text{INT0}}$) as well as ($\overline{\text{INT1}}$) are asserted simultaneously. IP = 04H.
(h) Assume that external interrupt 0 ($\overline{\text{INT0}}$) ISR is being executed and the same interrupt is asserted again.
(i) Assume that Timer 0 (TF0) ISR is being executed and both the external interrupts are asserted simultaneously. If IP = 1DH.

Solution:

(a) ISR of external interrupt 1 will be completed first, and then ISR of the external interrupt 0 will be serviced. Even if external interrupt 0 is having a higher natural priority, it cannot interrupt the ISR of external interrupt 1 because both the interrupts are assigned the same priority (0) using IP register and we know the fact that interrupt of same or lower priority cannot interrupt an ongoing ISR. The natural priority order is ***only*** used when two or more interrupts (of same priority assigned using IP) are asserted simultaneously. This is the most common mistake made by novice (or even experienced) programmers while working with multiple interrupts.

(b) ISR of external interrupt 1 is interrupted and the controller will service the ISR of external interrupt 0 because it is assigned higher priority (1) using IP register. Upon completion of ISR of external interrupt 0, the execution of ISR of external interrupt 1 is resumed.

(c) ISR of Timer 0 will be completed first, and then ISR of the external interrupt 0 will be serviced. See explanation of part (a) of this question.

(d) ISR of Timer 0 will be completed first, and then ISR of the external interrupt 0 will be serviced because Timer 0 is assigned higher priority (1) using IP.

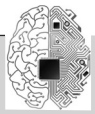
(e) ISR of Timer 0 is interrupted and the controller will service the ISR of external interrupt 0 because it is assigned higher priority (1) using IP register. Upon completion of ISR of external interrupt 0, the execution of ISR of Timer 0 is resumed.

(f) Since both the interrupts are assigned same priority (0) using IP, and they are asserted simultaneously, the natural priority will be considered, i.e. ISR of external interrupt 0 is serviced first and then ISR of external interrupt 1 is serviced. [Since both the interrupts are assumed to be edge-triggered, they will be remembered (latched by IE0 and IE1 bits).]

(g) Since the external interrupt 1 is assigned higher priority (1) using IP, it will be serviced first and then the external interrupt 0 is serviced.

(h) Interrupt on the same pin is ignored.

(i) ISR of Timer 0 is interrupted and ISR of external interrupt 0 is serviced first and then ISR of external interrupt 1 is serviced, and then ISR of Timer 0 is resumed.



THINK BOX 16.5

Priorities are used to determine which interrupt source will be serviced first when two or more interrupts are generated simultaneously. Where else do you think are priorities useful?
Priorities are also used to determine whether an interrupt source can interrupt the active ISR, i.e. used in nested interrupts. Usually, interrupts with high priority may interrupt the ISR with lower priority.

16.10 | BLOCKING CONDITIONS

In the 8051, interrupt flags are sampled at S5P2 of each machine cycle and samples are polled in the next machine cycle. If one of these flags was set during S5P2 of the previous cycle, it will generate an interrupt and the controller will jump to the corresponding vector address provided this interrupt is not blocked by any of the following conditions.

1. An ISR of higher or equal priority is in progress.
2. The current cycle (polling of sample) is not the final cycle of the instruction being executed; this will ensure that the instruction in progress will be completed before vectoring (jumping) to any ISR.

3. The instruction being executed is RETI or any write to the IE or IP register; this will execute at least one more instruction before any interrupt is vectored to ISR. This condition guarantees that changes of the interrupt status can be observed by the interrupt controller.

The 8051 does not latch (remember) an external level-triggered interrupt request that is blocked by the above conditions; so, an interrupt source must keep their signal active until the interrupt is handled. The system designer is responsible to handle such situations.

16.11 | INTERRUPT LATENCY

Latency is the time elapsed between generation of an interrupt and execution of the first instruction of the ISR. The interrupt response timing for the 8051 is shown in Figure 16.10.

If an interrupt request is generated, it will be latched in S5P2 of that machine cycle and interrupt flags status is polled in the next machine cycle (M2 in Figure 16.10). If interrupt service is not blocked by any of the three conditions discussed above, the microcontroller will be vectored to ISR (it will take 2 cycles, M3 and M4); therefore, a minimum 3.25 cycles will be required to respond to the interrupt.

If an interrupt of higher priority level is generated before S5P2 of machine cycle M3, microcontroller will be vectored to ISR of high-priority interrupt during M5 and M6 cycles without executing any of the instructions of lower priority ISR.

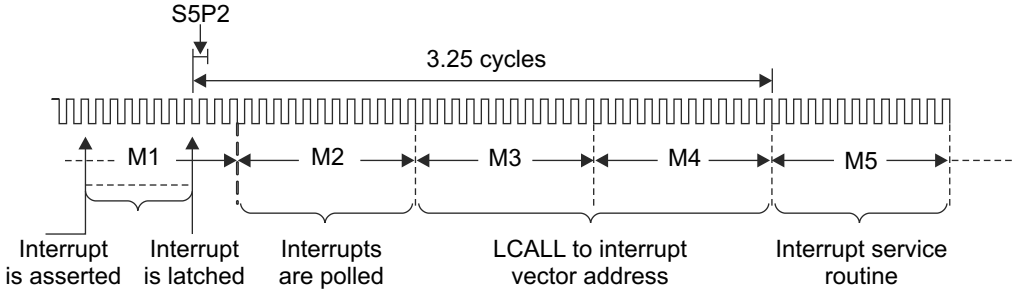


Fig. 16.10 *Interrupt response timing*

A longer time may be required if the interrupt request is blocked by one of the three conditions. If the instruction being executed is not in its final cycle, then the maximum additional wait time is 3 cycles (because longest instruction is of 4 cycles) and if instruction is RETI (or write to IE or IP register), the maximum additional wait time may be 5 cycles (1 cycle to complete instruction being executed (RETI) + 4 cycles (corresponding to the longest instruction, i.e. MUL or DIV) to complete one more instruction. The worst-case response occurs when interrupt (of high level) is asserted just before RETI instruction (of low level ISR) and is followed by multiply instruction. This worst-case response time is shown in Figure 16.11.

As shown in Figure 16.11, assume that an interrupt occurs when interrupt of high priority is asserted just after start of S5P2 state and before RETI instruction of low-priority ISR and is followed by a multiply instruction. Since interrupts are

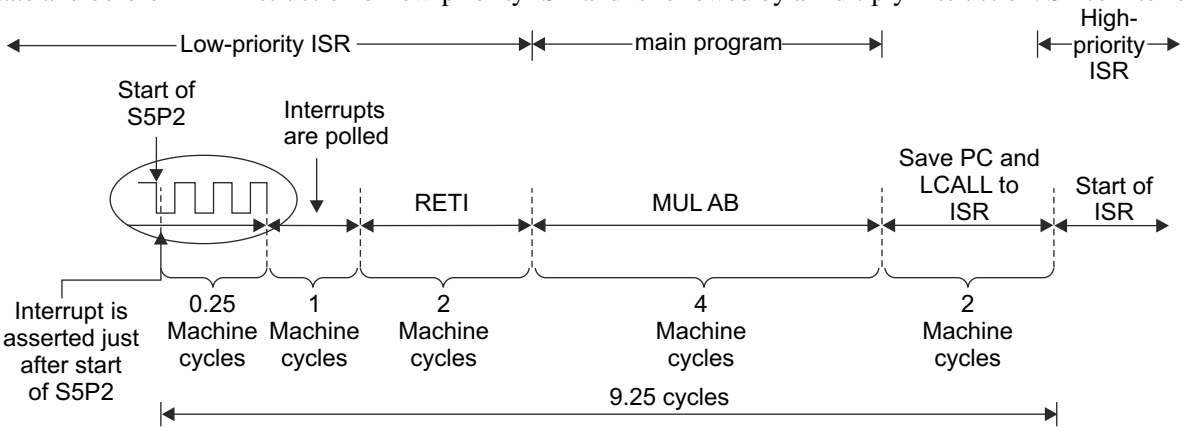


Fig. 16.11 *Worst-case interrupt response time*

polled in the next cycle after they are sampled, it will take 9 machine cycles (1 cycle for polling + 2 cycles for RETI + 4 cycles for multiply or divide instruction + 2 cycles to save PC and vector to ISR) to start ISR. The additional 0.25 cycles is because S5P2 occurs 0.25 cycles before end of machine cycle (9^{th} state out of total of 12 states of machine cycles). Thus, in a single interrupt system, the latency is more than 3.25 cycles and less than 9.25 cycles.

16.12 GENERATING INTERRUPTS USING INSTRUCTIONS

The interrupts can be generated in the software by setting corresponding bit in TCON or SCON register. For example, instruction SETB TF0 will generate Timer 0 interrupt and force the microcontroller to vector to the address 000BH. Similarly, external interrupt ISRs can be tested by setting IE0 or IE1 bits by instructions. This feature will help to test ISRs without generating the actual interrupts and without any external hardware.

16.13 CAUTIONS WHILE DEVELOPING INTERRUPT SERVICE ROUTINES

Interrupts provide a powerful and efficient way of developing programs; however, they require more efforts from programmer/system designer during program development. A small mistake may require huge amount of debugging time because it is difficult to identify the bugs in interrupt service routines. There are few simple cautions that have to be considered by a programmer; they are listed below:

1. Always terminate ISR with RETI instruction: interrupt service routines are always terminated with the RETI instruction. It is quite common to use the RET instruction (even though the programmer knows the difference between these two instructions). The RET instruction will only resume operation in the main program, but, it will not restore (re-enable) status of equal or lower priority interrupts which were disabled (by the 8051 when ISR was entered). This will stop responding to the interrupts (of equal or lower priority) any more. Always make sure ISR ends with the RETI instruction.
2. **Context saving:** Save registers (or memory locations) that are modified (used) by ISR for its internal operations. It is very common to forget to save the registers that are used by the ISR. This may overwrite or modify the important data in the main program resulting in erroneous behaviour of the program. It is preferred to save all the registers used in the ISR as well as PSW just at the beginning of the ISR (unless the application demands to change the specific register), and at the end of ISR, retrieve all the saved registers. This approach is better compared to saving all the registers used in the main program, because the ISR developer may not know the details of the main program and when the interrupt will occur!

   To speed up the process of context saving (or switching), switch the register bank, which relieves the program from saving the registers R0 to R7. This is one of the major reasons for providing register banks in the 8051.
3. **Context retrieving:** Another common error is to save registers onto the stack and then forget to retrieve them all from the stack before exiting the interrupt service routine. An unequal number of save and retrieval of registers will result in return at the wrong address. Therefore, always make sure that an equal number of PUSH and POP instructions are used.

16.14 DILEMMA: USE INTERRUPT OR POLLING?

Table 16.5 briefly gives guidelines about the situations in which the interrupts and polling are preferred.

Table 16.5 *Use of interrupt or polling*

Interrupts are preferred when…	Polling is preferred when…
Faster response time is desired.	Response time is not an issue.
Many tasks are to be handled (either in main program or other ISRs).	Few the tasks are to be handled.
Events are generated by hardware.	When operator is human.
When average time between occurrence of interrupts is large compared to the interrupt latency.	When average time between the occurrence of interrupts is nearly equal to the interrupt latency.

And finally, interrupts are normally used to combat against emergency and unexpected conditions.

16.15 | PROJECT: FULL-DUPLEX SYSTEM

Problem Statement

Design a full-duplex system for serial communication between two 89C51 based systems (boards). The status of input switches connected to one microcontroller should be sent continuously to LEDs connected on another microcontroller and vice versa.

Solution:

The full-duplex system will transfer data in both the directions simultaneously by using two serial links, one for transmission and the other for reception. The block diagram of a desired system is shown in Figure 16.12.

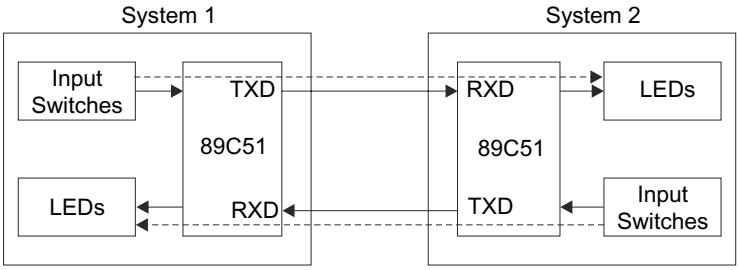


Fig. 16.12 *Block diagram of 89C51 based full-duplex system*

The desired operation of simultaneous data transfer in both the directions can be achieved by (i) continuously monitoring the status of input switches (the discussion is equally applicable to both systems) and then transmit the status serially, and (ii) reading received data and sending them to LEDs; these operations are handled by interrupt service routine of serial port. If TI interrupt is generated, the status of P2 is read and transmitted through TXD and if RI flag is generated, contents of SBUF are read and sent to LEDs. The circuit diagram of a full-duplex system is shown in Figure 16.13.

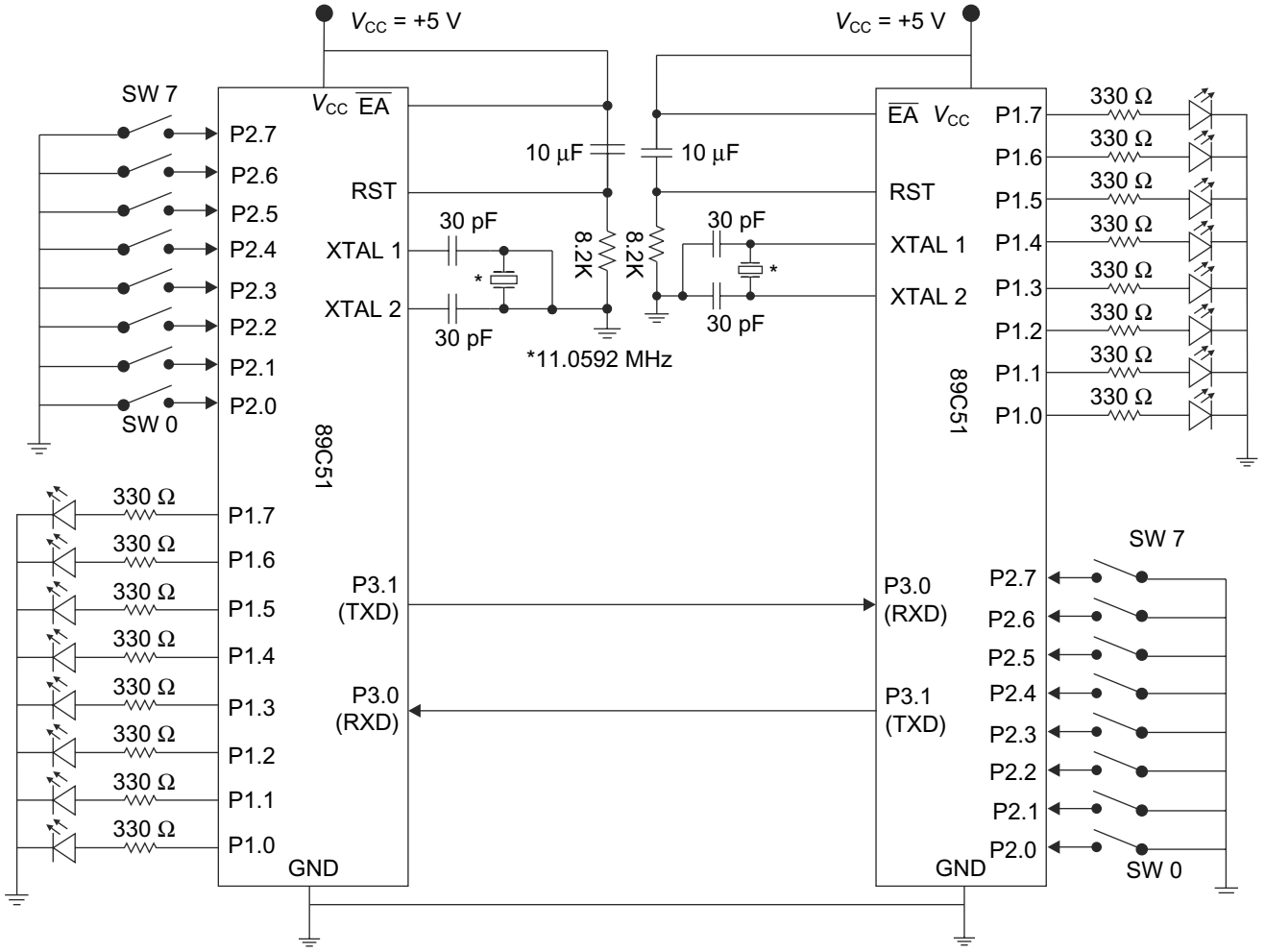


Fig. 16.13 *Full-duplex communication between two 89C51 based systems*

For both the microcontrollers, eight switches are connected with Port 2; eight LEDs are connected with Port 1. The TXD pin of one microcontroller is connected with RXD pin of another microcontroller and vice versa.

Program Development

Since both systems (89C51 boards) are doing the same operations, the program discussed in the following section must be loaded into both systems.

The steps to develop the program are the following:

- Configure P2 as an input because switches are connected to it.
- Load value into TH1 to set required baud rate.
- Configure Timer 1 in Mode 2 to generate clock signal at desired Baud rate.
- Configure SCON register for Mode 1 and reception enabled.
- Enable serial port interrupt and start Timer 1 to generate clock signal.
- Wait for serial interrupt to occur.

The ISR for serial port should perform the following tasks:

- If TI = 1, read status of switches and load into SBUF for transmission.

Clear TI, clear TI before sending the next byte.

- If RI = 1, read SBUF register and send the contents to LEDs (Port 1).

Clear RI, clear RI before receiving the next byte.

```
#include<reg51.h>
        void serial (void)interrupt 4                   // ISR to transmit and receive data on serial port
        {
                if (TI==1)
                        {
                        SBUF = P2;              // read status of switches and transmit it
                        TI = 0;                 // clear interrupt
                        }
                else
                        {
                        P1 = SBUF;              // send received value on LED (port 1)
                        RI = 0;                 // clear RI to be ready to receive the next byte
                        }
        }

        void main (void)                                // main program
        {
                P2 = 0xFF;                      // configure P2 as input port
                TH1 = 0xFD;                     // 9600 Baud rate
                TMOD = 0x20;                    // configure Timer 1 in Mode 2
                SCON = 0x50;                    // 8-bit data, 1 stop bit, REN enabled
                TR1 = 1;                        // start Timer 1 to generate clock
                IE = 0x90;                      // enable serial interrupts
                SBUF = P2;                      // read status of switches and transmit it
                while (1);                      // wait for serial interrupt
        }
```

Suggested Modification

Transmit the status of switches from System 1 to the PC (through MAX232 driver) and observe the received data at the PC in the hyper-terminal window. Send a byte from PC and display it on LEDs of System 1.

POINTS TO REMEMBER

- Interrupts provide a more efficient and effective way to serve many devices and allow the most efficient utilization of time and resources of the microcontroller.
- The interrupts are asynchronous events.
- Polling wastes microcontroller time in continuous monitoring of the device and, therefore, results in a slower system.
- Five interrupts are available in the 8051; 2 timers, 1 serial port and 2 external interrupts.
- There are eight bytes reserved for the ISR of each interrupt in the interrupt vector table.
- The reset may be considered as a special interrupt with exception that there is no mechanism to return back to interrupted program.
- The Interrupt Enable (IE) and Interrupt Priority (IP) register are the two registers to control operations of all five interrupts.
- A low-priority interrupt can be interrupted by high-level priority interrupt but vice versa is not true.
- The external interrupts can be activated in two different configurations, the level triggered and transition (edge) triggered interrupts.
- External interrupts in level-triggered mode are not latched. The external source that generates the interrupt controls directly the IEX interrupt flag and the low level must be removed before the end of the ISR.
- For edge-triggered external interrupts, IEX flag is automatically cleared by hardware to 0 when program execution is vectored to ISR.
- The transmit and receive interrupts are combined (ORed) to generate a common serial port interrupt.
- The Transmit Interrupt (TI) is generated when the transmission of the byte written to SBUF register is completed. TI is also known as transmitter empty interrupt flag.
- The Receive Interrupt (RI) is generated when a byte or character is received in the receive buffer.
- Since there are two different sources (TI or RI) for a common serial port interrupt, these flags are not cleared automatically but the ISR should clear the interrupting source flag using a software instruction.
- In the 8051, interrupt flags are sampled at S5P2 of each machine cycle and samples are polled in the next machine cycle.
- The 8051 does not remember any interrupt source; therefore, interrupt sources must keep their signals active until an interrupt is handled.
- Interrupt latency is the time elapsed between generation of interrupt and execution of the first instruction of ISR.
- In a single interrupt system, the latency is more than 3.25 cycles and less than 9.25 cycles.
- Interrupt service routines are always terminated with the RETI instruction.

OBJECTIVE QUESTIONS

1. Number of bytes reserved for each interrupt in the interrupt vector table is,
   (a) 4 (b) 8 (c) 16 (d) none of the above
2. UART (serial port) interrupt will be vectored to the address,
   (a) 0000H (b) 0003H (c) 0013H (d) 0023H
3. In 8051, an external interrupt 0 has vector address _____ and interrupt is asserted if _____.
   (a) 000BH, a high-to-low transition on pin INT1
   (b) 001BH, a low-to-high transition on the pin INT1
   (c) 0003H, a high-to-low transition on the pin INT0
   (d) 0023H, a low-to-high transition on the pin INT1

4. For level-triggered external interrupt, once the ISR is started, the source of the interrupt has to be disabled by,
   (a) the ISR (b) the microcontroller
   (c) the external device (d) any of above
5. Serial port interrupt has vector address _____ and interrupt is asserted if _____.
   (a) 0013H, either TI or RI flag is set (b) 0023H, either TI or RI flag is reset
   (c) 0013H, either TI or RI flag is reset (d) 0023H, either TI or RI flag is set
6. If the oscillator frequency of the 8051 based system is 6 MHz, it will require at least,
   (a) 3 μs (b) 2 μs (c) 8 μs (d) 6.5 μs
   to initiate the interrupt service routine after receiving the interrupt request.
7. The 8051 can serve _____ interrupt sources.
   (a) 3 (b) 4 (c) 5 (d) 6
8. The 8051 has _____external interrupts.
   (a) three (b) two (c) five (d) none of the above
9. Which memory address is assigned to Timer 0 as a vector address?
   (a) 0003H (b) 000BH (c) 0013H (d) 0023H
10. Reset signal acts as,
    (a) non-maskable vectored interrupt (b) non-maskable, non-vectored interrupt
    (c) maskable vectored interrupt (d) maskable, non-vectored interrupt
11. If IP.4 = 10H,
    (a) serial port interrupt has the highest priority
    (b) Timer T1 interrupt has the highest priority
    (c) serial port interrupt has the lowest priority
    (d) Timer T1 interrupt has the lowest priority
12. If IP = 00H,
    (a) INT1 has the highest priority
    (b) INT0 has the highest priority and serial interface has the lowest priority
    (c) Timer 0 has the highest priority
    (d) there is no priority, interrupt processes in the order of its occurrence
13. Which is the lowest priority interrupt in the 8051?
    (a) External interrupt 0 (b) Timer 1 interrupt
    (c) Serial port interrupts (d) External interrupt 1
14. Serial port interrupt is generated, if _____ bits are set.
    (a) IE (b) RI, IE (c) IP, TI (d) RI, TI
15. In 8051 ___interrupt has the highest natural priority.
    (a) IE1 (b) TF0 (c) IE0 (d) TF1
16. The EA bit disables,
    (a) all maskable interrupts (b) only timer interrupts
    (c) only external interrupts (d) only serial interrupts

Answers to Objective Questions

1. (b)	2. (d)	3. (c)	4. (c)	5. (d)	6. (d)	7. (c)	8. (b)
9. (b)	10. (a)	11. (a)	12. (b)	13. (c)	14. (d)	15. (c)	16. (a)

REVIEW QUESTIONS WITH ANSWERS

1. Which SFRs are used to control and serve the interrupts?

A. IE, IP and TCON.

2. What is the key advantage of using interrupts?

A. They relieve the microcontrollers from waiting for events to occur, thus it saves controllers time.

3. How many interrupts do we have in the 8051?

A. Five. External interrupt 0 and 1 ($\overline{INT0}$ and $\overline{INT1}$), timer overflow 0 and 1(TF0 and TF1) and serial port interrupt (RI or TI).

4. EA = 1 in IE register enables all interrupt. True or false?

A. False. Individual interrupt bit must be enabled along with EA.

5. What is the default status of all interrupts after reset?

A. All interrupts are disabled after reset.

6. There is single interrupt in interrupt vector table assigned to both TI and RI. True or False?

A. True.

7. What are the activation levels of external hardware interrupts?

A. External interrupts can be configured to be level-triggered or edge-triggered.

8. Upon activation of an interrupt, the 8051 immediately jumps to interrupt vector table. Justify true/false with reason.

A. False, the 8051 first completes current instruction and saves the return address to the stack.

9. What are the addresses reserved in the interrupt vector table for external interrupts?

A. For $\overline{INT0}$, 8 bytes from address 0003H and for $\overline{INT1}$, 8 bytes from address 0013H are reserved.

10. Does an interrupt provide multitasking in the 8051?

A. The 8051 can execute only one instruction at a time. However, they can provide the illusion of multitasking because once they are configured, they respond to certain conditions automatically.

11. If interrupt service executes only once, what can be the problem?

A. The programmer might have forgotten RETI instruction at the end of ISR, or using RET in place of RETI will cause the problem.

12. Which register is used to assign priorities in the 8051? Is it bit-addressable?

A. Interrupt Priority (IP) register. Yes, it is bit-addressable.

13. Why do we write the LJMP instruction at the address 0000H in interrupt based programs?

A. To skip interrupt vector table.

14. How many memory locations are reserved for each interrupt in interrupt vector table?

A. 8 bytes for each interrupt.

15. What should be done when the length of ISR is greater than 8 bytes?

A. Jump should be taken to suitable address in the program memory.

16. What is the disadvantage of having ISR length more than 8 bytes?

A. Interrupt response will be slower because an extra jump from IVT has to be taken to a suitable address to accommodate the larger ISR.

17. Can we change the natural priorities of interrupts? How?

A. Yes. Using IP register.

18. Which interrupt has the highest priority?

A. External interrupt 0.

EXERCISE

1. Define the terms interrupt, interrupt service routine and interrupt vector table.
2. Compare interrupt subroutines with a normal subroutine.
3. List the steps to enable an interrupt.
4. Which SFRs are used for interrupts?
5. What logic must be written to corresponding bit in IE register to enable an interrupt?

6. How does an external interrupt differ from internal interrupt?
7. Discuss the importance of stack pointer with respect to interrupt service routines.
8. Define the term interrupt latency.
9. Compare polling with interrupt method with respect to efficient handling of activities.
10. How are the register banks useful to reduce interrupt response time?
11. What actions will be taken by a microcontroller in response to external interrupt?
12. Discuss the difference between RET and RETI instructions.
13. Compare edge-triggered and level-triggered interrupts.
14. Can we replace RETI with RET instruction? Why?
15. What is the minimum pulse duration to detect edge-triggered interrupts?
16. Discuss the role of TCON.0 and TCON.2 in execution of external interrupt 0.
17. What is nested interrupt?
18. Write a short note on interrupt priority.
19. Make a table of memory addresses assigned to each interrupt in the interrupt vector table.
20. What happens when high-priority interrupt is asserted while the 8051 is serving lower priority interrupt? Also discuss the reverse case.
21. Explain how the interrupts are prioritized.
22. What is meant by "high-level" and "low-level" interrupts?
23. When would we use level-triggered interrupts?
24. Can we consider RESET as an interrupt? How it is different from other interrupts?
25. Assume that external interrupt 1 ($\overline{INT1}$) ISR is being executed and external interrupt 0 ($\overline{INT0}$) is asserted. Consider IP = 1FH. What will be the sequence in which interrupts are served?
26. Write a single instruction to enable all the interrupts.
27. How many bytes are reserved in program memory for RESET?
28. When are TI and RI raised? How are they cleared?
29. Timer interrupts must be cleared by software once serviced. Justify true/false with reason.

17

Interfacing Keyboards

Objectives

- Discuss the types, operation and design aspects of keyboards
- Discuss the key debouncing techniques using hardware and software
- Explain the algorithms for key-code generation for simple and matrix keyboards
- Design and interface keyboards with the 8051
- Develop programs for key identification and key-code generation

Key Terms

- ASCII Keyboards
- Debounce Delay
- Key Bouncing
- Key Closure Identification
- Key-Code Generation
- Key Debouncing
- Key Identification
- Key Press/Release
- Matrix Keyboard
- Multiple Key Press
- Pushbutton Switch
- Simple Keyboard

Keyboards are interfaces between humans and computer systems. They are used for giving commands to control the system; also, the data is usually given through them. The keyboards may be as simple as a single "pushbutton" switch used for start/stop operation or they may be fairly complex as ASCII keyboards used in the personal computers.

17.1 | KEYBOARD DESIGN CONSIDERATIONS

Keyboards are operated by humans and should be designed to tolerate mischief, rough use or intentional misuse by the user. The keyboard should work properly even in one of the following unusual conditions.

1. Multiple key press or release simultaneously
2. Key pressed for longer time
3. Fast key press and release
4. Combination of any of the above conditions

These problems may be solved by a combination of hardware and software. The designer should try to resolve the above issues by software as much as possible to reduce the cost of a system. Interfacing keyboards with the microcontroller is a classic example of hardware-software co-design.

Software for keyboards may be based on the polling of keys or interrupt-based. The choice is made by the system developer based on an application. Polling is used when the system has to wait for a command from the keyboard to perform any task. Interrupts are used when the controller has to serve many operations.

The keyboards are essentially a collection of switches (keys); therefore, before we discuss keyboard designs, we should consider mechanical properties of switches.

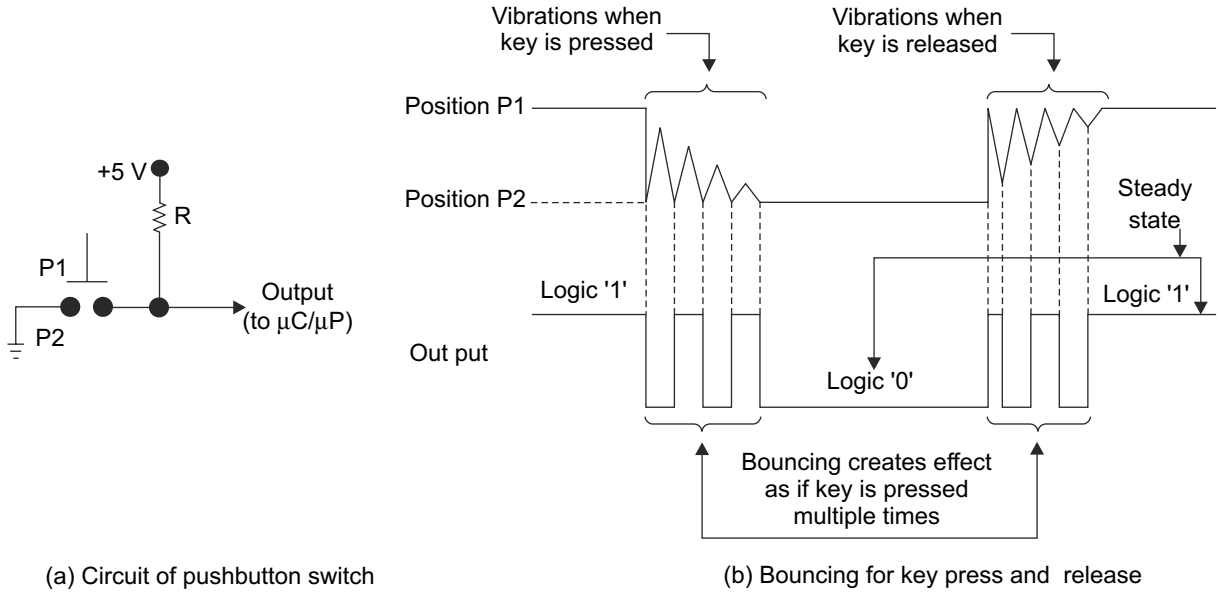


Fig. 17.1 *Key-bouncing mechanism*

17.1.1 Mechanical Properties of the Switches

The keyboards that are interfaced with a microcontroller/processor are usually made from pushbutton switches. When this pushbutton key is pressed or released, the metal contact momentarily bounces (vibrates) before making steady contact; this is commonly referred as *key bouncing*. Because of bouncing, a key makes and breaks contact many times even for a single-time key press as shown in Figure 17.1; therefore, it is necessary that the bouncing of the key should not be considered (read) as multiple-time key press and the techniques to eliminate this problem is known as *key debouncing*. There are two approaches to implement key debouncing: one is by using hardware and another is by using software. The software approach is preferred many times because it will reduce the cost of a system.

When a key is pressed (moving from P1 to P2), it will make and break contact with the point P2 many times because of the vibrations and this will generate many transitions of logic levels 0 and 1 (pulse train) at the output of the circuit. This signal is given as input to the microcontroller; the therefore, microcontroller will erroneously consider this bouncing as multiple-time key press. The microcontroller can detect these fast transitions because they work at a very high speed.

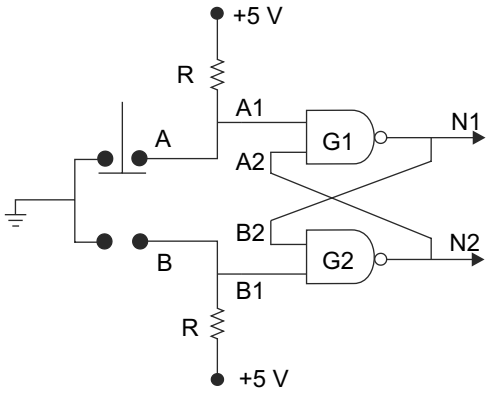


Fig. 17.2 *Hardware debouncing circuit*

17.1.2 Key Debouncing using Hardware

Figure 17.2 shows a debouncing circuit; it consists of one flip-flop. As per the connections, the state (output) of NAND gates don't change when the key is released from the point A. The outputs changes only when the key is connected at point B. When the key is connected at A, A1 is low and output N1 becomes high (if any input of NAND gate is low, its output is 1); this makes B2 high (1), since B1 is previously high, output N2 becomes low (0), which in turn makes A2 low. When the key is pressed, it is disconnected from *A*, as a result A1 will become high, but since A2 is already low, output N1 does not change. When the key reaches *B* (connected with *B*) then, only the output of gates change. Hence, when key changes the contacts (from *A* to *B*) during transition, the output does not change; this eliminates the problem of key bouncing.

THINK BOX 17.1

Is the bouncing time dependent on pull-up resistor? (See Figure 17.2.)
No. It depends on mechanical properties of a switch.

17.1.3 Key Debouncing using Software

In this approach, when a key press is detected, the microcontroller waits until the key reaches steady state and thereafter, the status of the key is checked again. The bouncing period is usually around 10 ms. So in the software approach, once a key press is detected, the microcontroller will wait for 10 ms (normally using delay subroutine). The delay is normally referred as the *debounce delay*.

17.2 KEYBOARD CONFIGURATIONS

The switches of keyboards may be connected directly to I/O pins or placed in the matrix with rows and columns. Based upon the fashion in which keys are connected, there are two configurations of keyboards: simple keyboards (one-dimensional) and matrix keyboards (two-dimensional). The hardware and software design of both types are discussed in the following sections.

17.2.1 Simple Keyboard Configuration (Using I/O Pins directly)

When the keys to be connected are less, i.e. around 10, each key may be directly connected with port pins as shown in Figure 17.3.

Eight keys are connected with Port 2 of the 8051. When any key is pressed, the corresponding port pin is grounded, for example if Key 0 is pressed, pin P2.0 is grounded. When keys are not pressed, the port pins are connected with V_{CC} (logic high) through resistors. The key identification can be done by monitoring port status. This process is discussed in detail in the following sections.

Advantages of a Simple Keyboard

- Easy keyboard design
- Easy software development
- Faster key identification

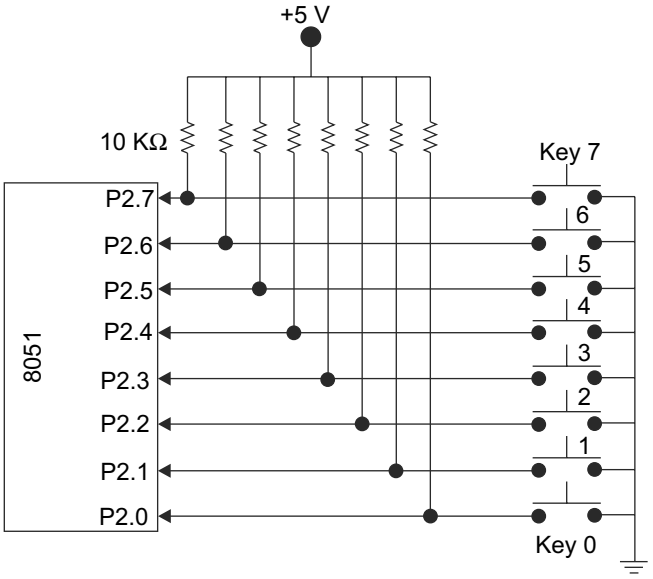


Fig. 17.3 *Simple keyboard*

Disadvantages of Simple Keyboard

In a simple keyboard, the number of port pins required increases in direct proportion to the number of keys connected. (*N* port pins will be occupied by *N* keys).When number of keys increases, this technique will occupy more port pins, which probably leaves no more pins free for other operations.

Key-Press Detection and the Code Generation

The process for generating code for a pressed key requires the following operations.

- Identify the key closure.
- Debounce the key.
- Identify the key and generate appropriate code like hexadecimal or ASCII code for the key pressed.

These steps are explained using a flowchart shown in Figure 17.4 for the simple 8-key keyboard shown in Figure 17.3.

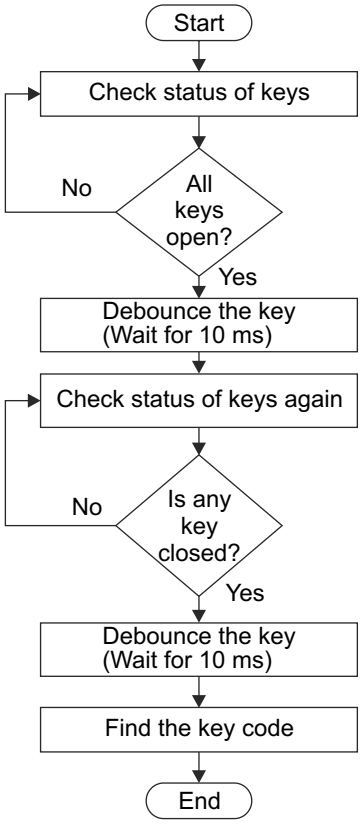


Fig. 17.4 *Algorithm for key identification and code generation*

Algorithm

- Check the status of all the keys (by reading a port) to determine whether a previously pressed key is released or not. If a key is not released, wait until it is released. Once a key is released, wait for debounce delay time. Now all the keys are released (open).
- Now, continuously check the status of all keys until any key press is found. Once any key is pressed, wait for debounce delay.
- Read status of all the keys and identify the key pressed and generate the code. (The status of the port pin for a pressed key will be 0 and all other pins will be 1; therefore, the status of the port will be read and the key will be identified by software.)

In Figure 17.3, 8 keys are connected to separate pins of Port 2. Each pin of the port gives the status of the key connected to that port pin. When a key is open, the port pin is at Logic HIGH and when a key is pressed, port pin is at Logic 0. The process begins by checking whether the previous key has been released. This will eliminate the problem of multiple reading of the same key and when the key is pressed for a long time. The key press and release are both debounced by waiting for 10 ms. Now the microcontroller waits for a key press and once key press is found, after waiting for debouncing delay, the binary code for a pressed key is found usually by setting a counter.

Program for a Simple Keyboard

Example 17.1 demonstrates the subroutine for key identification for keyboard configuration of Figure 17.3 with respect to the flowchart of Figure 17.4.

Example 17.1

Write a program subroutine to identify the key pressed for configuration of Figure 17.3.

Solution:

The steps to develop the program are shown in the flowchart of Figure 17.4.

Subroutine KEY is given to get key code with key debounce.

```
KEY:        MOV R2, #00H              // store code of pressed key in R2
            MOV R3, #08H              // counter for 8 keys
NO_REL:     MOV A, P2                 // read key status
            CJNE A, #0FFH, NO_REL     // wait until all the keys are released
            ACALL DBOUN               // delay for debouncing (when a previous key is released)
WAIT:       MOV A, P2                 // read key status
            CJNE A, #0FFH, IDENTIFY   // if a key pressed, go to identify key
            SJMP WAIT                 // otherwise wait until the key is pressed
IDENTIFY:   ACALL DBOUN               // debouncing delay (when a key is pressed)
            MOV A, P2                 // read the keys again for identification
AGAIN:      RRC A                     // identify the key by moving port bits into
            JC NEXTKEY                // carry flag, C = 0 indicates key pressed on that pin
            SJMP FOUND                // C = 0 indicates the key identified
NEXTKEY:    INC R2                    // code for key pressed
            DJNZ R3, AGAIN            // check the next key
            MOV R2, #0FFH             // error
FOUND:      RET
DBOUN:      MOV R6, #10               // debounce delay for 10ms (Xtal = 12MHz)
THR2:       MOV R7, #250
THR1:       NOP
            NOP
            DJNZ R7, THR1
            DJNZ R6, THR2
            RET
```

The above subroutine gives the binary equivalent value of the pressed key into R2 register. Note that when all the eight pins are checked and key pressed is not identified then the program stores FF in R2 to indicate an error. This may happen because of noise spike. The noise spike may generate an effect as if any key is pressed but after debounce delay, when the key status is read again for key identification, the effect of noise spike may not be there. This condition of erroneous key press is indicated by FF in R2. (This type of error occurs rarely.)

The other approach to handle the above problem is to compare the status of keys for the following two instants:

(i) When key press is detected, and

(ii) Just after debounce delay corresponding to key-press detection. If these two statuses are same then a valid key was pressed and we may proceed for key identification, otherwise key press detection was because of the noise and the program has to wait until a valid key is pressed.

Example 17.2

Rewrite the program of Example 17.1 in the C language.

Solution:

```
#include<reg51.h>

void delay(void) ;                // declare function delay
unsigned char key(void);          // declare function key

void main()
{
```

```
        unsigned char keycode;
        keycode = key();          // call function for getting the keycode
        while (1);
}

unsigned char key(void)
{
        unsigned char i, j, k, m;
        i = 00;                   // store the code of pressed key in variable i

        while (P2 != 0xFF);       // read key status and wait until all the keys are released
        delay ();                 // delay for debouncing (when previous key is released)
        while (P2==0xFF);         // wait until a key is pressed
        delay ();                 // delay for debouncing (when previous key is pressed)
                                  // key identification follows
        k=P2;                     // store the key status into a temp. variable
        for (j=0; j<8; j++)
              {
              m = k >> 1| k<< 7;                  // rotate right the status of P2(keys) by one bit
              k = m;                              // store rotated value back in k
              if (( m & 0x80) == 0)               // check MSB; if it is 0, key pressed is detected
              goto found;                         // and come out of the loop
              i++;                                // if MSB is not 0, key is not detected
              }                                   // repeat the operation
        i = 0xff;                                 // error
        found: return i;
}
void delay (void)
{       int a;
        for (a=0; a<10000; a++);
}
```

Key Identification using the Hardware Technique

The interfacing diagram for key identification using hardware technique is shown in Figure 17.5.

When all the keys are open, the output of AND gate (INT) remains high. When any key is pressed, the output of AND gate becomes low; this output may be connected to external interrupt pin (for example, $\overline{\text{INT0}}$). The interrupt pin may be used to interrupt the microcontroller for key identification.

Advantage of Hardware Technique

Here, the microcontroller will be relieved from continuously monitoring the key status and, therefore, there is no wastage of time of a microcontroller. The microcontroller can perform other activities when no key is pressed and when any of the keys is pressed then interrupt will be generated and interrupt service routine will perform the operation of key identification and code generation. Then, the normal program execution is resumed; this way, the microcontroller time is utilized more efficiently.

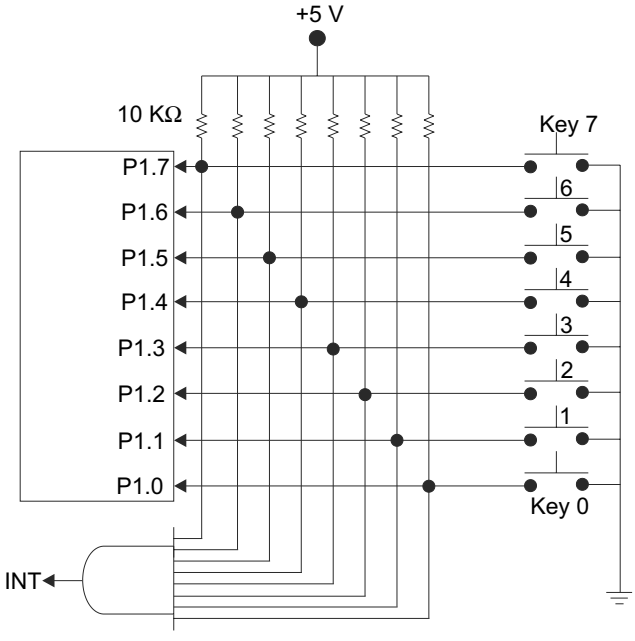


Fig. 17.5 *Keyboard using interrupt*

17.2.2 Matrix Keyboard Configuration

To save the number of port pins, the keys are arranged in a matrix form. The keyboard will have *N* rows and *M* columns. The number of port pins occupied to interface $N \times M$ keyboard is only $N + M$. This configuration is known as *matrix keyboard*. Figure 17.6 (a) shows 16 keys arranged in four rows and four columns. This arrangement is popularly referred as a 4×4 matrix

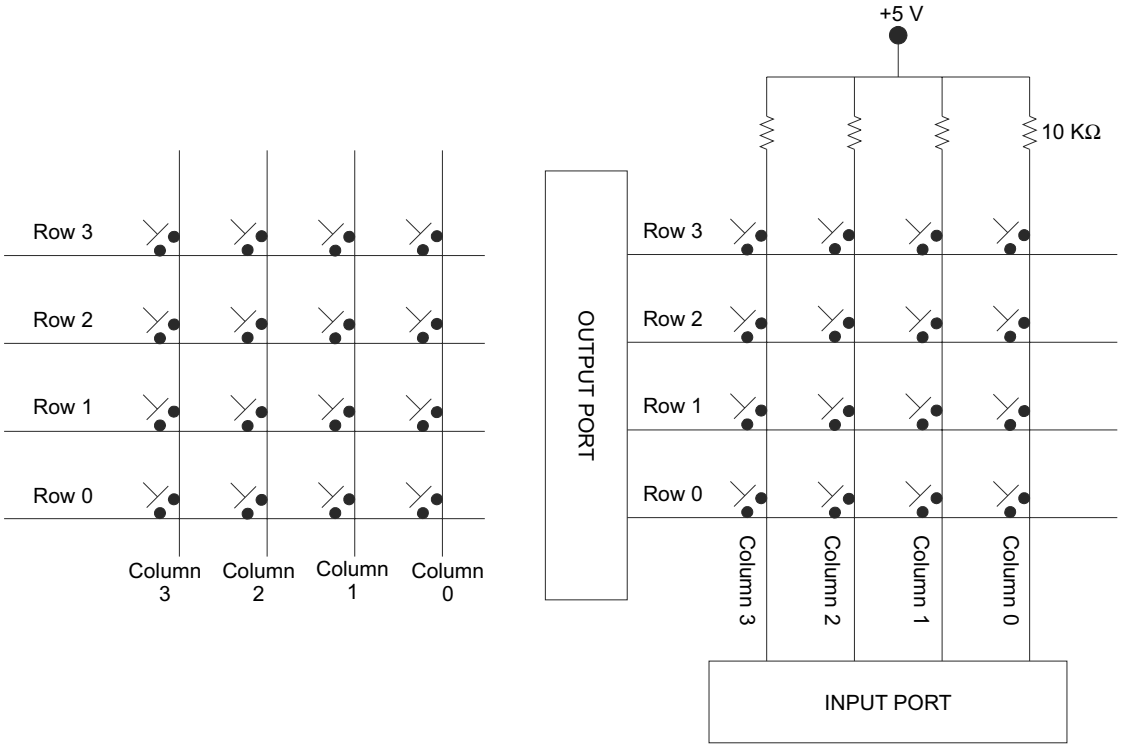


Fig. 17.6 *Matrix keyboard*

keyboard. Note that it requires only 4 + 4 port pins but a simple keyboard configuration would have required 16 port pins. The keys shown in Figure 17.6 are pushbutton keys.

When the keys are open, rows and columns are not connected (do not have any connection), and when any key is pressed, it connects the corresponding row and column. Rows are usually connected at the output port of the microcontroller/ processor (rows are configured as outputs) and columns are connected with the input port of the microcontroller (columns are configured as inputs) as shown in Figure 17.6 (b). Initially, all the rows are grounded (low logic level). Now, if any key is pressed, it makes the corresponding column low, otherwise the column will remain high. The interfacing of the 4×4 matrix keyboard with the 8051 is shown in Figure 17.7. The key identification process is discussed in detail in the next section.

Key-Code Generation

The steps to identify the pressed key and to generate the key codes are given below:

(a) Check for previously pressed key release (see Figure 17.7).

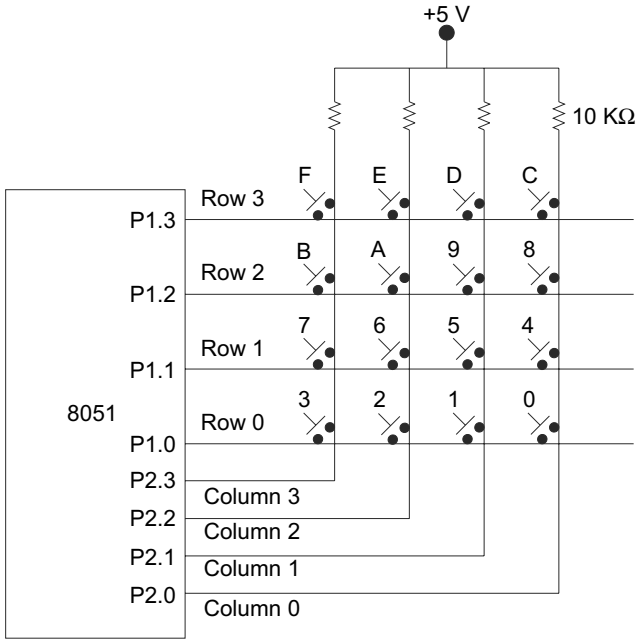


Fig. 17.7 *4 × 4 matrix keyboard interfaced with the 8051*

Check whether all the keys are released or not. It is done by grounding all the rows (by sending zeros). Now read the columns. High status of all columns indicate all the keys are open, otherwise the previous key is not released; therefore, wait until all the keys are released. This will eliminate the problem of multiple reading of the same key and when a key is pressed for a long time. When a previously pressed key is released, debounce the key release.

(b) Identify valid key closure.

Read columns and again check their status; if all the columns are high, no key is pressed and wait until a key is pressed. Low on any of the column indicates a key is pressed. Once a key press is detected, debounce the key press by waiting for 10 ms and read the columns again.

(c) Identify the key pressed and generate the code for key.

There are two methods for key identification and code generation, one is using the counters and the other method is using look-up tables. They are discussed in subsequent sections of the chapter.

Key Identification and Key Code Generation using Counters

The interfacing Example 17.3 demonstrates the program for key identification using counters.

Example 17.3

Design a system which contains a 16-key matrix keyboard and 8 LEDs interfaced with 89C51. Develop a program to detect the key press (key closure) and key identification. The binary code of the pressed key should be displayed on LEDs.

Solution:

The 16 keys are configured as 4 × 4 matrix keyboard. The pins P1.0 to P1.3 are configured as output and are connected to the rows. The pins P2.0 to P2.3 are configured as inputs and are connected with the columns. Eight LEDs are connected with 8 pins of Port 3. The complete interfacing diagram of a required system is shown in Figure 17.8.

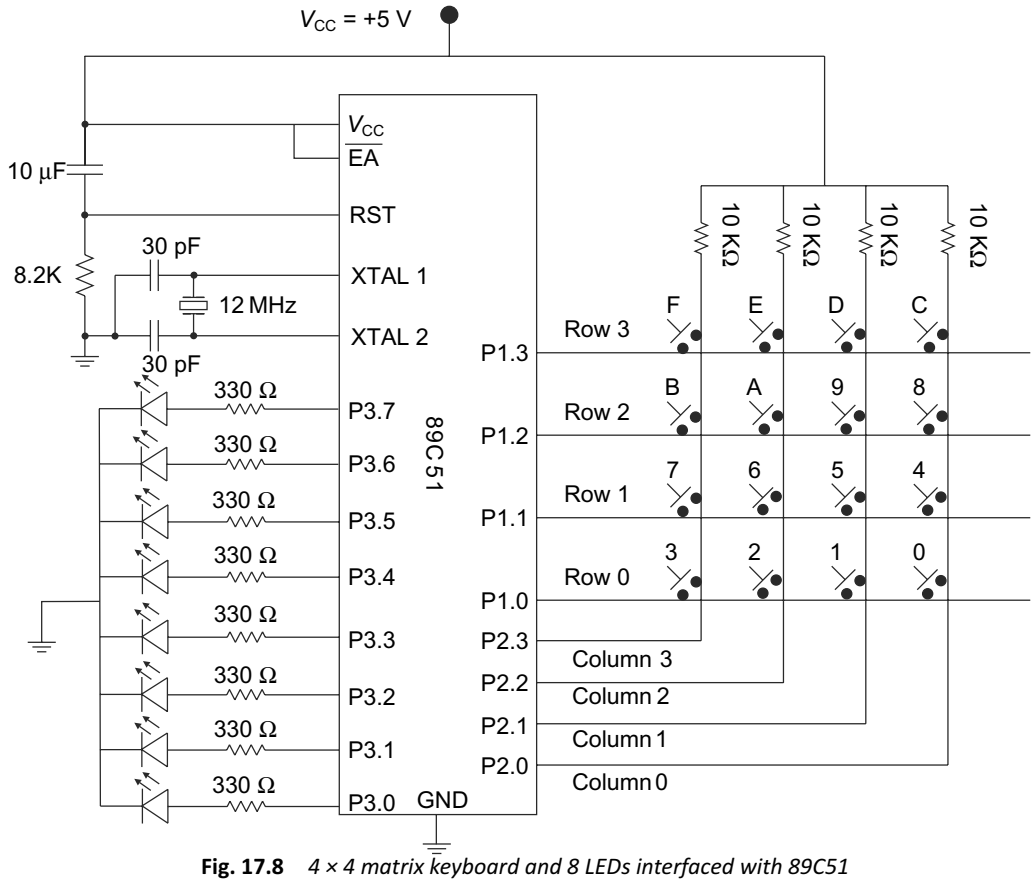


Fig. 17.8 *4 × 4 matrix keyboard and 8 LEDs interfaced with 89C51*

Port pins P1.0 to P1.3 are connected with Row 0 to Row 3 respectively, and P2.0 to P2.3 are connected with Column 0 to Column 3 respectively. The key which connects Row 0 and Column 0 will have a binary code 0. Row 0 and Column 1 will have the code as 1 and so on, as shown in Figure 17.8. The microcontroller AT89C51 is chosen because it has 4Kbytes of on-chip flash memory, easily available at a low cost. Note that $\overline{EA}$ is connected to V_{CC} because the program will be stored in on-chip memory. Power ON reset circuit is also included and crystal of 12 MHz is connected.

The binary codes of the 16 keys are from 0 to F and to display these codes, only 4 LEDs are required, but 8 LEDs are interfaced here because the program may be modified in future to display ASCII codes of these keys.

The program for key-code generation using counters is discussed here.

The steps in program development to detect a key press (key closure) and key identification are the following:

- Check for previously pressed key release.
- Wait until the previously pressed key is released.
- Once the previously pressed key is released, wait for the debounce delay.
- Wait until the pressed new key is pressed.
- Once any key is pressed, wait for the debounce delay.
- Ground one row at a time and check each column for zero.
- Set two loop counters, one for row and the other for column identification of the pressed key.
- The outer loop grounds one row at a time and the inner loop checks each column for zero.
- For each row, the inner loop is repeated four times (number of columns).
- For every column check, the counter is incremented.
- For four rows, the inner loop is repeated sixteen times and the counter is incremented from 0 to F.

```
        ORG 0000H
START:  MOV R0, #00             // binary code for the pressed key will be stored in R0
        MOV P2, #0FFH           // configure P2 as I/P port
        MOV P1, #00H            // ground all the rows
NO_REL: MOV A, P2
        ANL A, #0FH             // mask the upper nibble which is not used for keyboard
        CJNE A, #0FH, NO_REL
                                // if all the keys are not high previous key is not released
        LCALL DBOUN             // debounce for the key release
WAIT:   MOV A, P2               // check for any key press and wait until key is pressed
        ANL A, #0FH
        CJNE A, #0FH, K_IDEN    // key identify
        SJMP WAIT
K_IDEN: LCALL DBOUN
        MOV R4, #7FH            // only one row is made 0 at a time
        MOV R2, #04             // row counter
        MOV A, R4
NXT_ROW: RL A
        MOV R4, A               // save data to ground the next row
        MOV P1, A               // ground one row
        MOV A, P2
        ANL A, #0FH             // mask the upper nibble
        MOV R3, #04             // column counter
NXT_COLM: RRC A                 // move A0 bit in carry
        JNC KY_FND
        INC R0
        DJNZ R3, NXT_COLM
        MOV A, R4
        DJNZ R2, NXT_ROW
        SJMP WAIT               // no key closure found, go back and check again
KY_FND: MOV A, R0               // hex code of key is in R0, store it in A
```

```
            SJMP CONTINUE
DBOUN:      MOV R6, #10          // debounce delay for 10ms (Xtal=12MHz)
THR2:       MOV R7, #250
THR1:       NOP
            NOP
            DJNZ R7, THR1
            DJNZ R6, THR2
            RET
CONTINUE:   MOV P3, A            // send binary code for the pressed key on Port 3
            SJMP START           // go for detecting the next key press and identification
```

Key Identification and Key-Code Generation using the Look-Up Table

Example 17.4 demonstrates the program for key identification using a look-up table.

Example 17.4

Modify the program of interfacing Example 17.3 to generate the ASCII equivalent code for a pressed key.

Solution:

We have to create a look-up table for ASCII equivalent numbers for binary 0 to F. The binary code to ASCII code conversion is done by accessing the look-up table.

Write the following instructions at 'CONTINUE' label in the program of interfacing Example 17.3.

```
CONTINUE:   MOV DPTR, #300H      // address of the look-up table
            MOV A, R0            // binary code of the key is in R0
            MOVC A,@A+DPTR       // access the corresponding ASCII code in A
            SJMP START

            ORG 300H             // look-up table
            DB 30H,31H,32H,33H,34H,35H,36H,37H,
            DB 38H,39H,41H,42H,43H,44H,45H,46H
```

THINK BOX 17.2

There is a microcontroller embedded in the ASCII keyboards. What are the operations that the microcontroller software should perform?

The software should detect whether any key is pressed (or previously pressed key is released). After this, it has to wait for debouncing, and then has to identify which key is pressed and finally send the ASCII code of the pressed key to the motherboard.

POINTS TO REMEMBER

- When a key is pressed or released, the metal contact momentarily bounces (vibrates) before making steady contact; this is commonly referred as key bouncing.
- The software approach for debouncing is preferred because it will reduce the cost of the system.
- The process for key-code generation involves operations like identify the key closure, debounce the key and identify the key and generate appropriate code like binary (hexadecimal) or ASCII code for key pressed.
- For a simple keyboard configuration, N port pins will be occupied by N keys.
- The number of port pins occupied to interface a $N \times M$ matrix keyboard is only $N + M$.
- There are two methods for key identification and code generation; one is using the counters and the other method is using look-up tables.

OBJECTIVE QUESTIONS

1. The advantage/s of software debouncing compared to hardware debouncing is,
   (a) faster operation (b) reduce the component count
   (c) simplify the keyboard manufacturing (d) all of the above
2. The optimum configuration to connect 20 keys with the microcontroller is,
   (a) 20-key simple keyboard (b) 5 × 4 matrix keyboard
   (c) 4 × 5 matrix keyboard (d) all are equally efficient
3. Debounce delay should be called when,
   (a) a key is pressed (b) a key is released
   (c) either a key is pressed or released (d) multiple keys are pressed
4. The number of port pins occupied by a 5 × 5 matrix keyboard is,
   (a) 5 (b) 10 (c) 11 (d) 25
5. The human factors that need to be considered while developing software for key-code generation are,
   (a) multiple key press or release simultaneously
   (b) key pressed for longer time
   (c) fast key press and release
   (d) all of the above

Answers to Objective Questions

1. (b), (c) 2. (b), (c) 3. (c) 4. (b) 5. (d)

REVIEW QUESTIONS WITH ANSWERS

1. What are the methods to achieve debouncing?

A. Using hardware and software.

2. List the human factors that must be considered while developing keyboard software.

A. Multiple key press, rapid key press and key held for longer time.

3. What is the advantage of a matrix keyboard?

A. Less port pins are required to interface more keys

4. What is the advantage of a simple keyboard?

A. Simpler software development

5. Can we generate the ASCII codes of pressed keys? How?

A. Yes, by using look-up tables.

EXERCISE

1. List the products in which the keyboards are used.
2. List the various types of keys available in the market. Discuss their applications and characteristics.
3. Compare the hardware and software methods of key debouncing.
4. What is meant by an invalid key closure?
5. How can the interrupts be used to develop efficient key-detection programs?

18

Interfacing Display Devices: LED, Seven-Segment Display and LCD

Objectives

- Discuss the operations of common display devices like LEDs, seven segment display and LCDs
- Discuss the segment and digit multiplexing for seven-segment display modules
- List the pins of typical LCD and the function of each pin
- Develop and list the command codes for operation of the LCD
- Software and power-on initialization process for the LCD
- 4-bit and 8-bit modes of operation of the LCD
- Interface LEDs, seven-segment display and LCDs with the 8051
- Develop programs to display data on LED, seven-segment display and LCD

Key Terms

- 4/8 bit LCD Operation
- 7447/7448
- Buffer
- Busy Flag
- Common Anode Module
- Common Cathode Module
- Digit/Segment Multiplexing
- LCD Command/Data Register
- LCD Commands
- LCD Initialization
- Segment Driver
- Sink/Source Current

The microcontroller sends the information to the external world through an output port to the output devices. The output devices receive the result of operations or commands (triggers or response to some operations) from the microcontroller. The output devices are used either to display information or to perform (or control) the process or they may generate other control signals. The common display devices are LEDs, seven-segment displays, LCDs and CRT screens. Other output devices that are used to perform the operations are printers, motors, relays and data converters. This chapter describes the operation and interfacing of display devices like LEDs, seven-segment displays and LCD with the 8051.

18.1 | LIGHT EMITTING DIODES

The Light Emitting Diode (LED) is the simplest display element. LEDs are commonly used as indicator lights in majority of the electronic and other devices like televisions, audio/video systems, printers, washing machines, disk drives, control panels, etc. They are used to indicate the status of the device like powered on, running, waiting, error, etc.

LEDs are available in wide varieties of shapes and colours to support a wide variety of applications. The colour of an LED is determined by the semiconductor material used for it. The most common colours are green, red yellow, orange, blue and white. The infrared LEDs are also easily available.

When the anode of an LED is made positive with respect to the cathode, it will be forward-biased and will emit the light. The specifications of an LED are its forward voltage V_F and forward current I_F.

The typical values of V_F are from 2 to 3 V and that of I_F are from 10 to 20 mA, depending upon the type and size of the LED. The LEDs are normally operated around I_F. Three different techniques of connecting LEDs with the 8051 are shown in Figure 18.1.

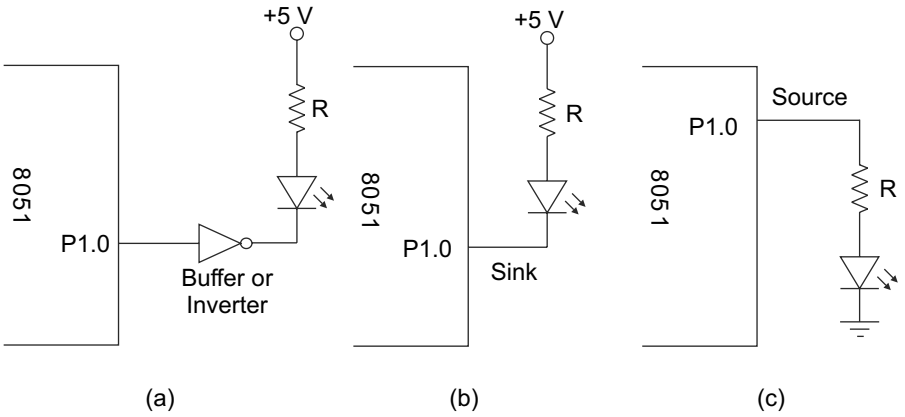


Fig. 18.1 *LED interfacing with 8051*

As shown in Figure 18.1 (a), the pin P1.0 is connected to a buffer (inverter). When P1.0 is high, output of an inverter is low causing the current to flow through the LED and it will glow. When P1.0 is low, output of an inverter is high, and no current flows through the LED and it will stop emitting light. The inverter in the circuit acts as a current buffer and prevents the port pins from current loading, the current flows between the inverter and the 5 V supply.

The resistor R will limit the current through the LED and protects it from damage. The value of R for a LED having V_F =2 V and I_F = 10 mA can be calculated as follows:

$$R = \frac{5 - V_F}{I_F} = \frac{5 - 2}{10 \times 10 - 3} = 300\,\Omega \approx 330\,\Omega$$

The LED may be directly connected to a port pin as shown in Figure 18.1 (b). Under the normal conditions, the port pin is made high by writing '1' to it by using SETB instruction, the LED will not glow because cathode is also high (in fact, after reset, the port pins are pulled high by internal pull-up resistors). The LED will glow when the pin is made low, where it sinks the current, a current limiting resistor should be connected as shown to protect the LED as well as port circuits (refer topic 13.1.1 for more details). The port pin will source the current when the port pin is high for configuration as shown in Figure 18.1 (c). This will require current in mili-amperes from the port pin.

Interfacing Example 18.1

Interface eight LEDs and eight pushbutton switches to Port 3 and Port 2 of 89C51 respectively. Write a program to monitor the status of all switches and display it on the corresponding LEDs, i.e. LED 0 should glow when switch 0 is pressed and so on.

Solution:

The complete interfacing of 8 LEDs and 8 switches is shown in Figure 18.2. Normally, all the pins of Port 2 are high. (Logic level of port pins are pulled high by internal pull-up resistors when 1 is written to pin latch to configure it as an input.) When any switch is pressed, the corresponding port pin is grounded. The LED will glow when the corresponding port pin is made high as cathodes of all LEDs are grounded.

Note that each LED is connected with its own resistor. We cannot use a single resistor at the cathode side because all the LEDs do not have the same characteristics, so one LED will draw more current than the others and may be damaged. After that, the remaining LEDs will get higher current and get damaged consequently. The other reason is that the intensity of the LEDs will be inversely proportional to the number of LEDs glowing at a time, i.e. when lesser LEDs are glowing, they will glow with more intensity and vice versa.

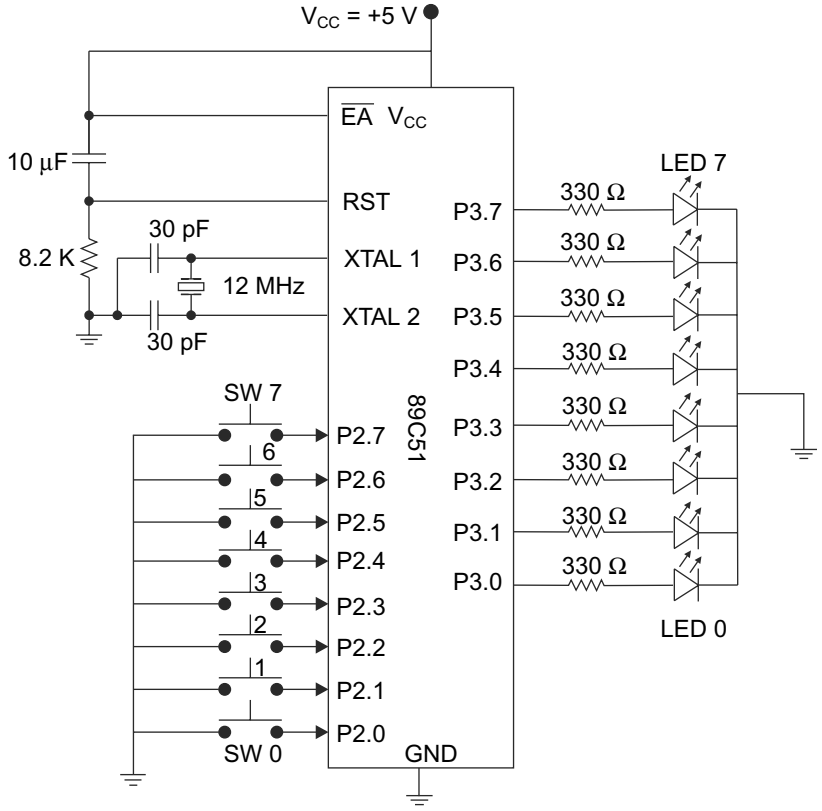


Fig. 18.2 *Interfacing 8 LEDs and 8 pushbutton switches with 89C51*

To display the status of switches on LEDs, the program should read the status of Port 2 and send it on Port 3 continuously.

```
        ORG 0000H
REPEAT: MOV A, P2H      // read the status of switches
        MOV P3, A       // send value to the LEDs
        SJMP REPEAT     // continuously monitor the status and display on LEDs
        END
```

Example 18.2

For the circuit of interfacing Example 18.1 (Figure 18.2), write a program to glow LEDS one by one in a sequence continuously.

Solution:

Only one pin is made high at a time by sending a data byte, which contains only one bit as '1', for example, 00000001. After some delay, the contents of the data byte is rotated in left (or right) direction and sent to the port; this will glow the next LED and the process is repeated.

```
            ORG 0000H
            MOV A, #01H       // only one bit is high, so one LED will glow at a time
REPEAT:     MOV P3, A         // send value to the port
            ACALL DELAY       // wait for delay
            RL A              // rotate to glow the next LED
            SJMP REPEAT       // repeat process forever
DELAY:      MOV R0, #10       // delay of 1s approx, 12 MHz crystal
THERE1:     MOV R1, # 200
THERE:      MOV R2, # 250
HERE:       DJNZ R2, HERE
            DJNZ R1, THERE
            DJNZ R0, THERE1
            RET
           END
```

Example 18.3

Rewrite the program of Example 18.2 in the C language.

Solution:

Program can be written in the C language as,

```
#include<reg51.h>
        void main ()
                {
                unsigned char i, j;
                unsigned int k;
                while (1)                                   // send pattern continuously
                        {
                        j = 1;                              // only one bit is high, so one LED will glow at a time
                        for (i=0; i<8; i++)
                                {
                                P3 =  j;                    // send the data byte to LED port
                                for ( k=0; k<60000; k++); // delay
                                j = j<<1;                   // shift data byte left by one bit
                                }
                        }
                }
```

Applications of LEDs

Few common applications of LEDs are listed below:

- Status indicators on all sorts of equipment
- Traffic lights and signals, motorcycle, bicycle and car brake lights (or rear light clusters)
- Remote controls, such as for TVs and VCRs (infrared LEDs)

THINK BOX 18.1

List different types of LEDs available in the market.

LEDs are classified according to the following criteria

1. **Light colour:** Red, orange, yellow, white, green, blue, multicolor LEDs.
2. **Outer surface shape:** Round LED, square, rectangular, etc.
3. **Diameter:** 2 mm, 4 mm, 5 mm, 10 mm, etc.
4. **Intensity (operating current):** Standard brightness (intensity <10 mcd), high brightness (intensity = 10 mcd to 100 mcd), ultra-high brightness (intensity >100 mcd).
5. **Directivity:** Highly directive (viewing angle: 5° to 20°), Standard (20° to 40°), Scattering (40° to 100°)

- Message displays at airports, bus and railway stations
- Fiber-optic communications
- Movement sensors (for example, optical mice)
- Decoration and lighting

18.2 SEVEN-SEGMENT DISPLAY

A seven-segment display is used to display the decimal (or hexadecimal) numbers. They consist of a group of seven LEDs (rectangular), they also have LED for a dot point (decimal point), therefore, they contain eight LEDs in a module which are arranged as shown in Figure 18.3.

The LEDs are assigned names as *a* to *h* as shown. Seven-segment displays are used for displaying the numeric information in the electronic meters, digital clocks, and other electronic devices. There are two types of seven-segment display modules: (i) common cathode: where all the LEDs in the module have common cathode and, (ii) common anode: where all the LEDs have common anode. These configurations are shown in Figure 18.4. Note that resistors shown are not a part of the module, but they are connected externally.

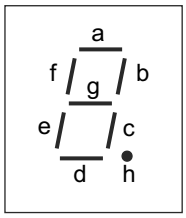


Fig. 18.3 *Seven-segment module*

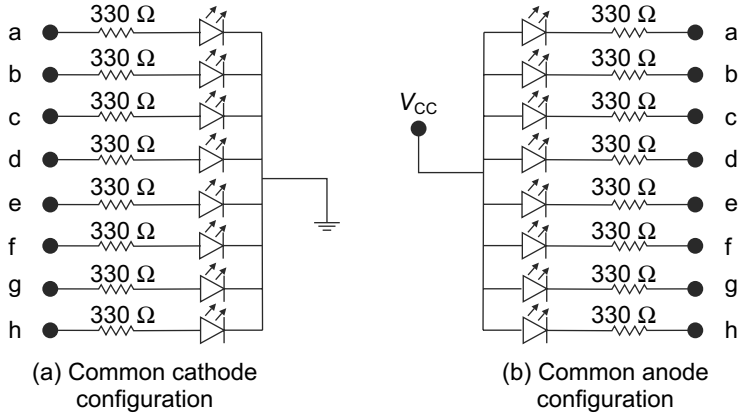


Fig. 18.4 *Common cathode and anode configurations*

18.2.1 Segment Multiplexing within one Seven-Segment Display

The current drawn by the seven-segment module is around 15 × 8 = 120 mA. This higher current requirement can be reduced by multiplexing segments within a display module. In this approach, all the segments are activated one by one and one segment at a time. Hence, the current requirement is reduced to 15 mA. This process is repeated fast enough to create an illusion of simultaneous activation of all the segments. The limitation of this method is that only few seven segments can be interfaced with a microcontroller because refreshing each segment within each display module will keep the microcontroller busy in looking after only displays.

18.2.2 Digit Multiplexing

When two or more digits are required for an application, we need to use digit multiplexing. In this approach, only one display module (digit) is activated at a time and all the modules are refreshed one by one at a fast rate so that all the display modules appear to be simultaneously active. Figure 18.5 shows interfacing of four seven-segment modules with the 8051.

IC7448 is BCD to seven-segment code converter and digit driver (provides required drive current for a display module). It provides an active high output which can be directly used for common cathode seven segment (or common anode seven segment, if used through transistor driver, i.e. inverter). Four port pins of the microcontroller are connected as an input to 7448 and its output are connected to all seven-segment modules in parallel. To display the digit, BCD code for a digit is sent to 7448 from port pins (lower 4 bits of Port 1 in Figure 18.5) and to select a particular digit (module), the common pin of the corresponding module is grounded by making the corresponding pin of the port high (Port 2 in Figure 18.5). IC7447 provides active low outputs which can be directly used with common anode seven-segment modules.

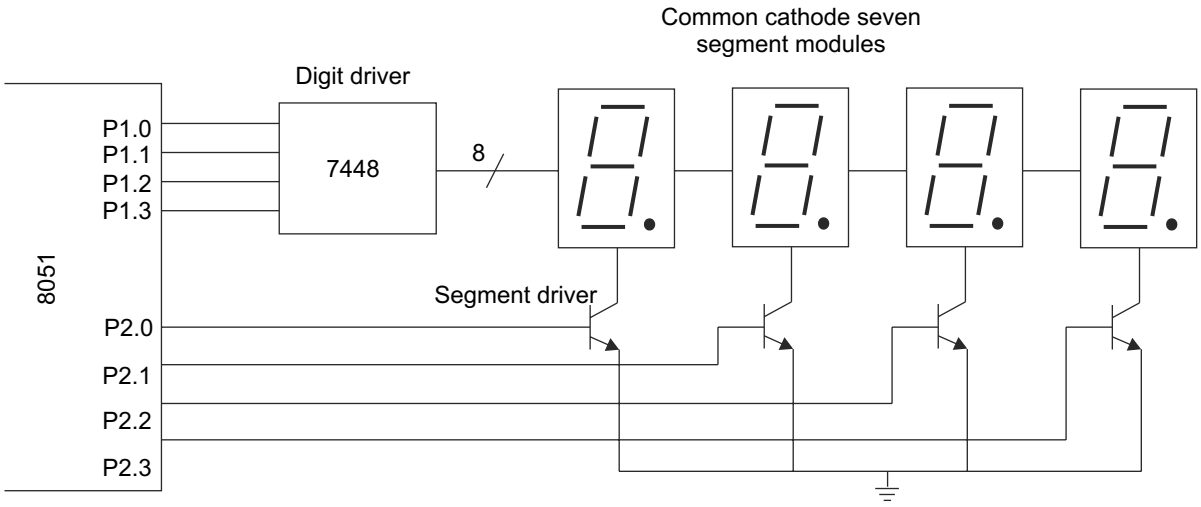


Fig. 18.5 *Interfacing four seven-segment common cathode modules with 8051*

Example 18.4

Assume that one common cathode seven-segment module is connected through 7448 with Port 1 and the common pin is permanently grounded. Write a program to display numbers from 0 to 9 repeatedly in a sequence on seven-segment modules. Provide the delay between two numbers.

Solution:

```
REPEAT:   MOV A, #00H          // display 0 first
NEXT:     MOV P1, A            // send BCD code to 7448
          MOV R5,#200          // delay
THERE2:   MOV R6,#255
THERE1:   NOP
          NOP
          NOP
          NOP
          DJNZ R6,THERE1
          DJNZ R5,THERE2
          INC A
          CJNE A, #0AH, NEXT   // display up to 9 only
          SJMP REPEAT          // repeat the sequence
```

Example 18.5

Rewrite the program of Example 18.4 in the C language.

Solution:

Program can be written in the C language as follows:

```
#include<reg51.h>

    void main ()
        {
        unsigned char i;
        unsigned int k;
        while (1)                          // send 0-9 continuously
                {
                for (i=0; i<=9; i++)
```

```
                    {
                    P1= i;                      // send number to seven-segment
                    for ( k=0; k<60000; k++);   // delay
                    }
                }
        }
```

Example 18.6

Modify the program of Example 18.4 if Port 1 is directly connected with a seven-segment module (P1.0 is connected with segment 'a' and P1.7 with segment 'h').

Solution:

The direct connection of the 8051 with a seven-segment module is shown in Figure 18.6.

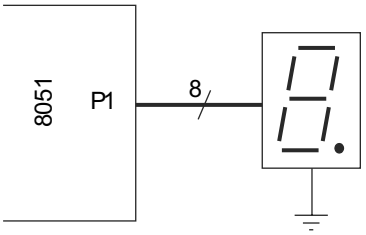


Fig. 18.6 *Direct connection of the 8051 with seven-segment module*

Since the seven-segment module is connected directly with port pins, we have to send seven-segment codes for 0 to 9 from the microcontroller. A look-up table is used to store seven-segment codes of numbers 0 to 9. To find the seven segment code for a particular number for common cathode module, first determine which segments should glow, and write '1' for that segment, write '0' for all other segments. For example, to display '1', segments b and c should be made ON by sending 1 to them; therefore, the code for '1' is 06H (h g f e d c b a = 0000 0110).

```
            ORG 0000H
REPEAT:     MOV A, #00H             // display 0 first
            MOV DPTR, #100H         // address of lookup table
NEXT:       MOV R2, A               // save A
            MOVC A, @ A+DPTR        // get code from lookup table
            MOV P1, A               // send BCD code to display module
            MOV A, R2               // retrieve value of A
            MOV R5,#200             // delay
THERE2:     MOV R6,#255
THERE1:     NOP
            NOP
            NOP
            NOP
            DJNZ R6,THERE1
            DJNZ R5,THERE2
            INC A
            CJNE A, #0AH, NEXT      // display up to 9 only
            SJMP REPEAT             // repeat the sequence
            ORG 100H                // lookup table
            DB                      3FH,06H,5BH,4FH,66H,6DH,7DH,07H,7FH,6FH
```

Example 18.7

Rewrite the program of Example 18.6 in the C language.

Solution:

The program can be written in the C language as follows:

The seven-segment codes of numbers 0 to 9 are stored in an array; the elements of array are accessed one by one and are displayed on the seven-segment module.

```
#include<reg51.h>

        void main ()
            {
unsigned char sevenseg_codes[]={0x3f,0x06,0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f};
```

```
unsigned char i;
unsigned int k;
while (1)                                   // send 0-9 continuously
        {
        for (i=0; i<=9; i++)
                {
                P1 = sevenseg_codes[i];     // send code to seven-segment
                for ( k=0; k<60000; k++);   // delay
                }
        }
}
```

THINK BOX 18.2

List the other commonly used BCD to seven-segment decoder ICs.
4511/14511: BCD to seven-segment latch/decoder/driver
4543: BCD to seven-segment latch/decoder/driver

18.3 | LIQUID CRYSTAL DISPLAY (LCD)

The LED (including multisegment LEDS) are useful in indicating that an application is running, is connected or is waiting, etc. They do not display all the ASCII characters, special characters and graphics. An LCD can display all the above characters easily. The interfacing (hardware + software) is relatively easy because refreshing of all the characters is done automatically by refreshing the controller circuit present in the LCD modules. They are commonly available in 16 × 1, 20 × 1, 20 × 2, 20 × 4 and 40 × 2 sizes (the first number indicates characters in a line and second number is number of lines in a display module).

Advantages of using LCD as a Display Device

- LCDs provide a better user interface as they can display ASCII messages
- Lower power consumption
- Display information is updated at a high speed because it has inbuilt refresh controller

18.3.1 Pin Description for LCD

LCD modules usually have 14 pins. These pins are described in Table 18.1.

Table 18.1 *LCD pin description*

Pin No.	Name	Function	Description
1	V_{SS}	Power	GND
2	V_{DD}	Power	+ 5 V
3	V_{EE}	Contrast adjust	0 – 5 V
4	RS	Register select	Signal to select data or command register of the LCD. RS = 0 select command register (for write); Busy flag, address counter (for read) RS = 1 select data register (for write)
5	R/W	Read / Write	Signal to read/write data from/to LCD. RW = 0 Write to LCD RW=1 read from LCD
6	E	Enable (Strobe)	High to low pulse is applied to this pin to enable LCD to accept (latch) data/ command present on its data lines D0-D7
7-14	D0-D7	Data lines D0 (Pin-7-LSB), D7 (Pin-14-MSB)	Bidirectional data lines used to send data/command to LCD; or read LCD internal registers, D7 is also used as a busy flag, D4 -D7 are used in the 4 bit operation

There are three control signals and 8 (or 4 for 4-bit mode) data lines. E (Enable) signal will inform the LCD module that the microcontroller is sending the data. RS (Register Select) signal selects the command or data register of the LCD, when RS = 0, data sent by microcontroller will be treated as a command to the LCD module, and RS = 1 indicates that data is a text to be displayed on the LCD module. R/W (Read/Write) indicates read or write operation to be performed, R/W = 0, data is written to the LCD and R/W = 1, indicates that the data is being read from the LCD module. A high-to-low transition on the E pin is required to latch and then execute the command given through data lines and the time required to execute the commands depend upon the value of crystal frequency used in the LCD module.

18.3.2 LCD Commands

Table 18.2 shows a list of commands (instructions) recognized by the LCD controller with description of how each command is formed using the different combinations of LCD signals along with typical execution time of each command.

The LCD commands are used to select various display functions like cursor positioning, blinking, character size, cursor shift, data format (4-bit or 8-bit). For example, to set entry mode as shift cursor left, make I/D = 0 and S = 0. Therefore, the command will be 04H and this command will be given to LCD through data lines D7-D0. Refer Table 18.2 to understand how the various commands are formed.

18.3.3 Initialization of the LCD using the Internal Reset Circuit

An Internal Reset Circuit (IRC) will automatically initialize the LCD when it is powered and V_{CC} reaches the full value (4.5 V) within 10 ms. The following commands/instructions are executed during the initialization process. The Busy Flag (BF) remains high (busy state) until the process is completed. The LCD will remain in busy state (BF = 1) for 10 ms after V_{CC} rises to 4.5 V.

- Display Clear
- Function set:
  DL = 1: 8-bit interface
  N = 0 : Messages are displayed in one line
  F = 0 : 5 × 7 dots—font size of character
- Display ON/OFF control:
  D = 0 : Display OFF
  C = 0 : Cursor is OFF
  B = 0 : Blink OFF
- Entry Mode Set:
  I/D = 1: Displayed addresses are automatically incremented by 1
  S = 0 : Display shift off

If the internal power supply does not reach 4.5 V within 10 ms, the display will not operate normally. In this case, the display can be initialized through the software.

18.3.4 Software Initialization of the LCD

Though software initialization is not mandatory, it is recommended that this procedure always be followed. When the internal power supply reset timings are not met then the display must be initialized by the steps shown in Figure 18.7.

18.3.5 LCD Timing

Development of a program for the LCD can be better understood by LCD timing. An LCD timing diagram is shown in Figure 18.8.

As shown in the timing diagram, when a command is to be given to the LCD, RS should be made 0 to select the command register. Then, R/W is made 0 for writing to LCD. Write command /data on the port to which the data bus of the LCD is connected. Apply High to the Low transition on the enable pin which will latch contents present on data bus into the LCD internal registers and start the execution of command.

18.3.6 Modes of operation

- 8-bit Mode
- 4-bit Mode

Table 18.2 *LCD commands*

Command	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Command Code (Hex)	Execution Time (Max.) f_{cp} = 250 kHz
Clear display	0	0	0	0	0	0	0	0	0	1	01	1.64 ms
Cursor home	0	0	0	0	0	0	0	0	1	x	02	1.64 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	04–Shift cursor left	40 µs
											06–Shift cursor right	
											05–Shift display right	
											07–Shift display left	
Display on/off control	0	0	0	0	0	0	1	D	U	B	08–Display off, Cursor off	40 µs
											0A–Display off, Cursor on	
											0C–Display on, Cursor off	
											0E–Display on, Cursor blink off	
											0F–Display on, Cursor blink	
Cursor/Display Shift	0	0	0	0	0	1	D/C	R/L	x	x	10–Shift cursor left	40 µs
											14–Shift cursor right	
											18–Shift display left	
											1C–Shift display right	
Function set	0	0	0	0	1	DL	N	F	x	x	28–2line,5X7matrix,4 line	40 µs
											38–2line,5X7matrix,8 line	40 µs
Set CGRAM address	0	0	0	1	CGRAM address							40 µs
Set DDRAM address	0	0	1	DDRAM address							80– Set cursor at beginning of line 1	40 µs
Read "BUSY" flag (BF)	0	1	BF	DDRAM address								40 µs
Write to CGRAM or DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0		40 µs
Read from CGRAM or DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0		40 µs

I/D 1 = Increment (by 1)	0 = Decrement (by 1)	R/L 1 = Shift right	0 = Shift left
S 1 = Display shift on	0 = Display shift off	DL 1 = 8-bit interface	0 = 4-bit interface
D 1 = Display on	0 = Display off	N1= Display in two lines	0 = Display in one line
U 1 = Cursor on	0 = Cursor off	F 1 = Character format 5 × 10 dots	0 = 5×7 dots
B 1 = Cursor blink on	0 = Cursor blink off	D/C 1 = Display shift	0 = Cursor shift

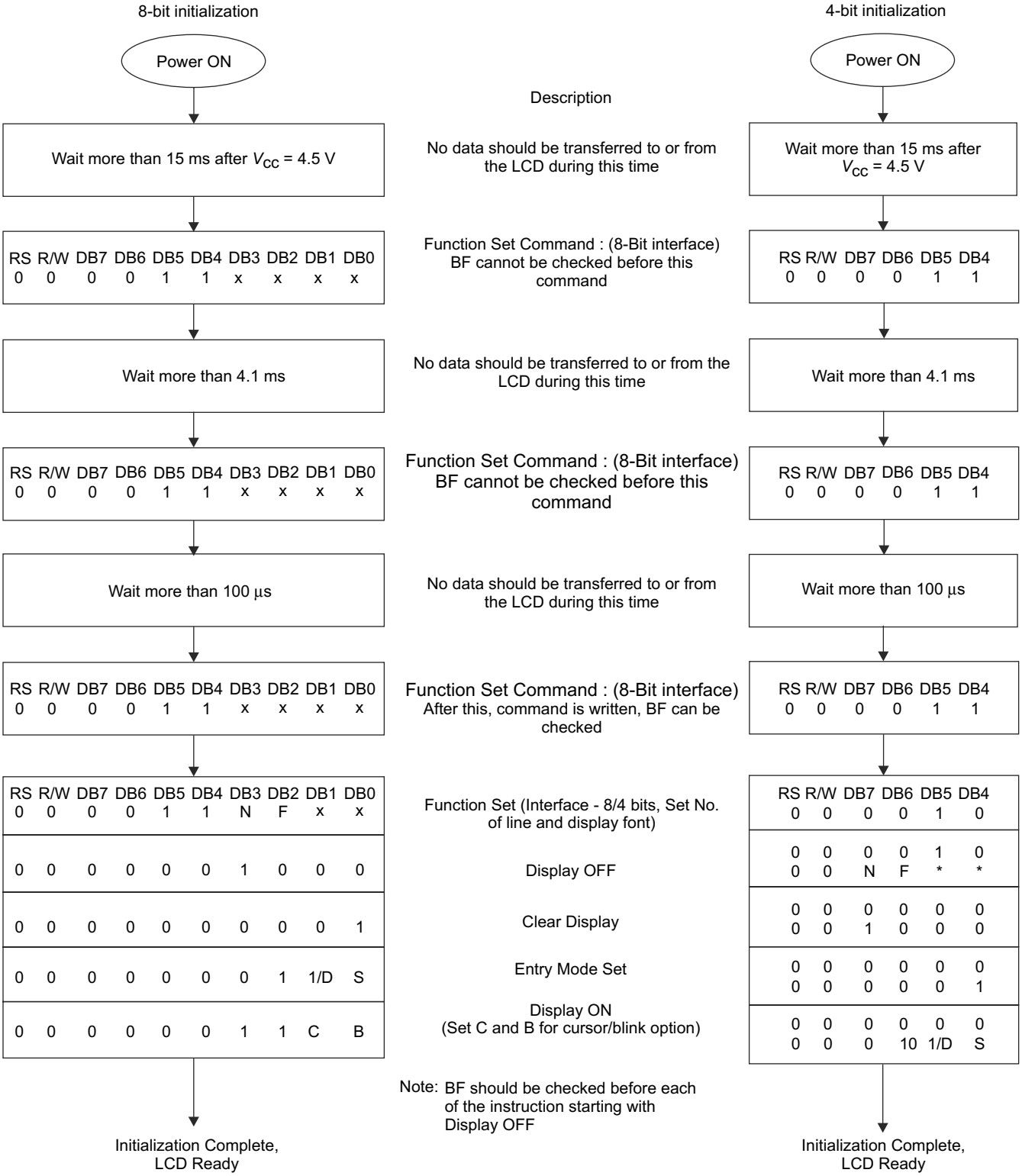


Fig. 18.7 *8-bit and 4-bit software initialization of LCD*

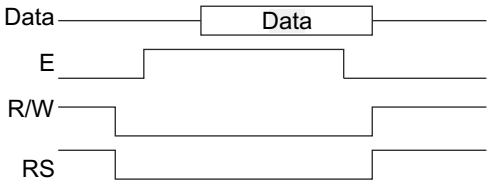


Fig. 18.8 *LCD timing*

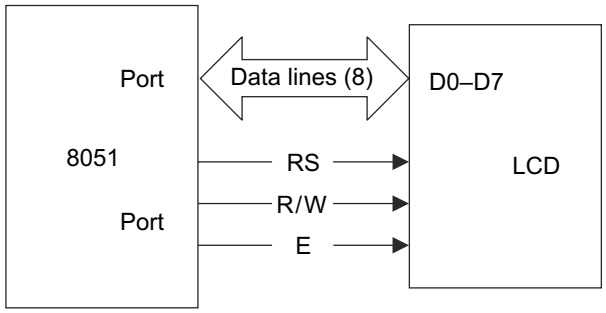


Fig. 18.9 *Simplified connections of LCD with 8051*

8-bit Mode

In this mode, all the eight data line pins (D7-D0) are used for giving data/commands. Figure 18.9 shows the simplified connections of the LCD module with the 8051 microcontroller.

Any port of the 8051 is connected with the data lines of LCD. (Remember that data lines of the LCD need not be always connected to the data bus of the 8051). Three pins from other port are connected with the control signals: Register Select, Read/Write and Enable.

The LCD can accept and execute different commands as shown in Table 18.2. When it is executing the commands, i.e. performing internal activities, data should not be sent to it otherwise the data will be overwritten. To indicate that it is busy in doing internal activities, it makes the bit D7 high (busy flag). So, a program must monitor D7 before giving the next command or data to the LCD. To avoid monitoring the busy flag, one can put reasonably larger delay between the two consecutive commands/data writes to make sure that the LCD has finished internal operation and is not busy. But this approach is not efficient and wastes the time. The only advantages of this approach is that it saves one I/O pin of microcontroller because R/$\overline{W}$ can be permanently grounded.

Interfacing Example 18.8

Interface 16 x 2 LCD module with the 89C51 and develop a program to display message 'HI' at the beginning of the first line.

Solution:

The complete interfacing diagram of the LCD with the 89C51 is shown in Figure 18.10. Note that the LCD is connected in an 8-bit mode. Port 2 of the microcontroller is connected with data lines of the LCD (P2.0 to P2.7 with D0 to D7 respectively). RS, R/W and E signals are connected with P1.0, P1.1 and P1.2 respectively. A potentiometer (10 KΩ) is connected between V_{CC} and ground to set the contrast of the display.

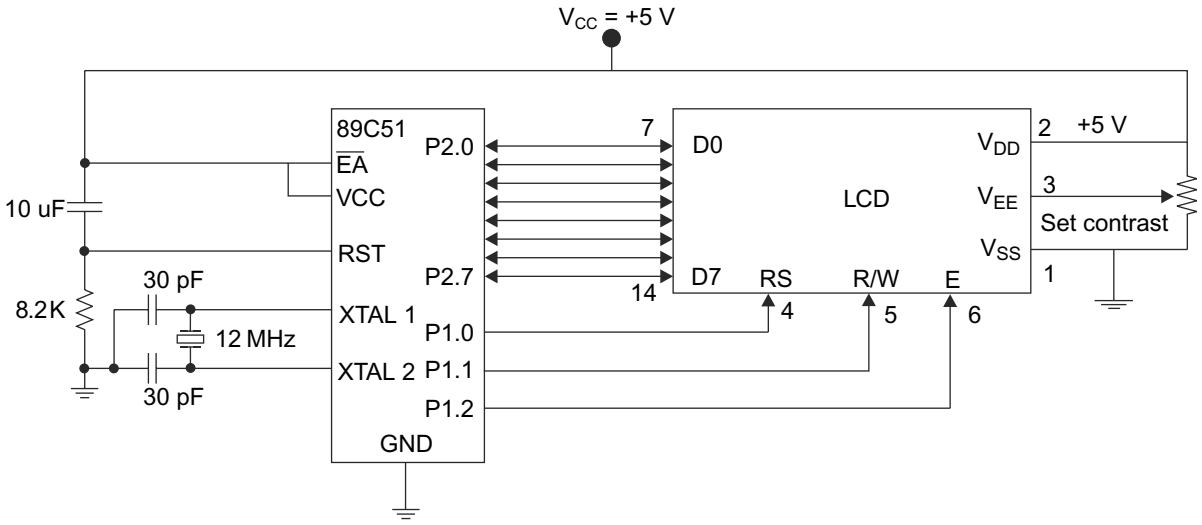


Fig. 18.10 *8-bit mode LCD interfacing with the 8051*

Steps for program development to display the data on the LCD are given as follows

- Initialize the LCD by sending a set of commands to it,

The commands are,

- Configure data bus as 4-bit or 8-bit mode
- Select character font, (i.e.) dots/character
- Configure display and cursor type, i.e. display ON or not, cursor blinking or not
- Configure display and cursor movement, i.e. left shift or right shift or shift off
- Configure display position
- Clear display

(The commands are sent by making RS = 0 and R/W = 0, and high to low pulse on E pin, a delay should be provided between two commands to ensure that the LCD has executed the previous command.)

- Send ASCII values of the characters to be displayed one character at a time with delay between them.
  (Data characters are sent to the LCD by making RS = 1 and R/W=0, and high to low pulse on E pin, with a delay between two consecutive data characters).

To make the program more efficient, two subroutines are defined, COMMAND for sending command and DISPLAY for sending data. The corresponding routines are called when data or command is to be given to the LCD. Note that the delay routine is a part of both the routines to wait before issuing the next command or data.

```
//P2.0-P2.7 are connected to LCD data pins D0-D7
//P1.0 is connected to RS pin of LCD
//P1.1 is connected to R/W pin of LCD
//P1.2 is connected to E pin of LCD
            ORG 0000H
            LCALL WAIT          // initialization of LCD by software
            LCALL WAIT          // this part of program is not mandatory but
            MOV A, #38H         // recommended to use because it will
            LCALL COMMAND       // guarantee proper initialization even when
            LCALL WAIT          // power supply reset timings are not met
            MOV A, #38H
            LCALL COMMAND
            LCALL WAIT
            MOV A, #38H
            LCALL COMMAND       // initialization complete

            MOV A, #38H         // initialize LCD, 8-bit interface, 5X7 dots/character
            LCALL COMMAND       // send command to LCD
            MOV A, #0FH         // display on, cursor on with blinking
            LCALL COMMAND       // send command to LCD
            MOV A, #06          // shift cursor right
            LCALL COMMAND       // send command to LCD
            MOV A, #01H         // clear LCD screen and memory
            LCALL COMMAND       // send command to LCD
            MOV A, #80H         // set cursor at line 1, first position
            LCALL COMMAND       // send command to LCD
            MOV A, #'H'         // H to be displayed
            LCALL DISPLAY       // send data to LCD for display
            MOV A, #'I'         // I to be displayed
            LCALL DISPLAY       // send data to LCD for display
HERE:       SJMP HERE           // wait indefinitely
COMMAND:                        // command write subroutine
            MOV P2, A           // place command on P1
            CLR P1.0            // RS = 0 for command
```

```
        CLR P1.1        // R/W = 0 for write operation
        SETB P1.2       // E = 1 for high pulse
        LCALL WAIT      // wait for some time
        CLR P1.2        // E = 0 for H-to-L pulse
        LCALL WAIT      // wait for LCD to complete the given command
        RET
DISPLAY:                // data write subroutine
        MOV P2, A       // send data to port 1
        SETB P1.0       // RS = 1 for data
        CLR P1.1        // R/W = 0 for write operation
        SETB P1.2       // E = 1 for high pulse
        LCALL WAIT      // wait for some time
        CLR P1.2        // E = 0 for H-to-L pulse
        LCALL WAIT      // wait for LCD to write the given data
        RET
WAIT:   MOV R6, #30H    // delay subroutine
THERE:  MOV R5, #0FFH   //
HERE1:  DJNZ R5, HERE1  //
        DJNZ R6, THERE
        RET
```

Note. The delay routine 'WAIT' is placed inside the routines 'DISPLAY' and 'COMMAND' to make the program more compact, otherwise the delay routine could also have been placed in a main program before issuing every new command (or data).

Example 18.9

Rewrite the program of Interfacing Example 18.8 in the C language.

Solution:

```
#include<reg51.h>
sbit RS=P1^0 ;
sbit RW=P1^1 ;
sbit E=P1^2 ;

void COMMAND (unsigned char);
void DATADISPLAY (unsigned char);
void DELAY (void);

void main()
{
        DELAY();                // initialization of LCD by the software
        DELAY();                // this part of the program may be skipped
        COMMAND(0x38) ;
        DELAY();
        COMMAND(0x38) ;
        DELAY();
        COMMAND(0x38) ;
        DELAY();

        COMMAND(0x38) ;         // LCD command for LCD 2 lines 5*7 matrix
        COMMAND(0x0F) ;         // display on, cursor on with blinking
        COMMAND(0x06) ;         // shift cursor right
        COMMAND(0x01) ;         // clear display
        COMMAND(0x80) ;         // cursor at line 1, position 0
        DATADISPLAY('H') ;      // send data to LCD
        DATADISPLAY('I') ;
```

```
}
void COMMAND(unsigned char cmd)
        {
        P2 = cmd ;              // send command
        RS = 0;                 // RS=0 for command
        RW = 0;                 // R/W=0 for write command
        E = 1;                  // high to low pulse for write
        DELAY();
        E = 0;
        DELAY();                // wait before giving the next command
        }
void DATADISPLAY(unsigned char data)
        {
        P2 = data;              // send data
        RS = 1;                 // RS=1 for data
        RW = 0;                 // R/W=0 for write command
        E = 1;                  // high to low pulse for write
        DELAY();
        E = 0;
        DELAY();                // wait before giving the next command
        }
void DELAY(void)
        {
        unsigned int i ;
        for(i=0; i<7000; i++);
        }
```

Example 18.10

Modify the program of Interfacing Example 18.8 to display HI in second line, beginning at the 5th location.

Solution:

Replace MOV A, #80H (set cursor at Line 1, first position) with MOV A, #0C5H (set cursor at Line 2, fifth position)

In a LCD, we can display data at any location. It is selected by 'SET DDRAM ADDRESS' command. Its format is as shown below:

Command	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
Set DDRAM address	0	0	1	A	A	A	A	A	A	A

Where AAAAAAA = 0000000 to 0100111 for line 1 and 1000000 to 1100111 for line 2.

The upper address can go up to 0100111 for a 40-character wide LCD while, it will go up to 0010011 for a 20-character wide LCD. The starting and ending addresses for several LCDs is shown in Table 18.3.

Table 18.3 *Addresses for the characters of LCDs*

Type	Line	Start address (Left most character) to end address
16 × 2	Line 1	80 81 82 and so on up to 8F
	Line 2	C0 C1 C2 and so on up to CF
20 × 1	Line 1	80 81 82 and so on up to 93
20 × 4	Line 1	80 81 82 and so on up to 93
	Line 2	C0 C1 C2 and so on up to D3
	Line 3	94 95 96 and so on up to A7
	Line 4	D4 D5 D6 and so on up to E7

Importance of Monitoring Busy Flag

D7 (pin 14) of the LCD module acts as a busy flag when read (RS=0, R/W=1). In Example 18.8, we have placed a long delay at the end of COMMAND and DISPLAY routines (between two consecutive commands or data write operations) to make sure that LCD has completed the previous operation. This approach wastes much more time and makes the program slower because many times our program will waste time (wait) unnecessarily even when the previous operation might have been finished. The better approach is to monitor the busy flag to decide whether the LCD is ready for the next operation or not. Example 18.8 can be modified to save the time. Instead of waiting for a long time before issuing the new commands, simply monitor the busy flag using the "WAIT: JNB P2.7, WAIT" instruction. This instruction will be executed repeatedly until the LCD is no longer busy (completed the previous command).

Example 18.11

Modify program of Interfacing Example 18.8 to monitor the busy flag and to decide whether the LCD is ready for the next operation or not.

Solution:

The initialization by software is not done in this program.

```
                ORG 0000H
                MOV A, #38H       // Initialize LCD, 8-bit interface, 5×7 dots/character
                LCALL COMMAND     // send command to LCD
                MOV A, #0FH       // display on, cursor on with blinking
                LCALL COMMAND     // send command to LCD
                MOV A, #06        // shift cursor right
                LCALL COMMAND     // send command to LCD
                MOV A, #01H       // clear LCD screen and the memory
                LCALL COMMAND     // send command to LCD
                MOV A, #80H       // set cursor at line 1, first position
                LCALL COMMAND     // send command to LCD
                MOV A, #'H'       // H to be displayed
                LCALL DISPLAY     // send data to LCD for display
                MOV A, #'I'       // I to be displayed
                LCALL DISPLAY     // send data to LCD for display
HERE:           SJMP HERE         // wait indefinitely
                                  // command write subroutine
COMMAND:        ACALL READY       // check busy flag
                MOV P2, A         // place command on P1
                CLR P1.0          // RS = 0 for command
                CLR P1.1          // R/W = 0 for write operation
                SETB P1.2         // E = 1 for high pulse
                LCALL WAIT        // wait for some time
                CLR P1.2          // E = 0 for H-to-L pulse
                RET
                                  // data write subroutine
DISPLAY:        ACALL READY       // check busy flag
                MOV P2, A         // send data to port 1
                SETB P1.0         // RS = 1 for data
                CLR P1.1          // R/W = 0 for write operation
                SETB P1.2         // E = 1 for high pulse
                LCALL WAIT        // wait for some time
                CLR P1.2          // E = 0 for H-to-L pulse
                RET
```

```
READY:    SETB P2.7          // configure P2.7 as input
          CLR P1.0           // RS = 0 for command
          SETB P1.1          // RW = 1 for reading
WAIT:     CLR P1.2           // E = 1 for high pulse (see the following loop)
          ACALL DEALY
          SETB P1.2          // high to low pulse on E
          JNB P2.7, WAIT     // wait until busy flag is 0
          RET
```

Example 18.12

Rewrite the program of Example 18.11 in the C language.

Solution:

The corresponding C language program using the busy flag is,

```
#include<Reg51.h>
sbit RS=P1^0 ;
sbit RW=P1^1 ;
sbit E=P1^2 ;
sbit BUSY= P2^7;
void READY (void);
void COMMAND (unsigned char);
void DATADISPLAY (unsigned char);
void DELAY (void);

void main()
{       COMMAND(0x38) ;          // LCD command for LCD 2 lines 5*7 matrix
        COMMAND(0x0F) ;          // display on, cursor on with blinking
        COMMAND(0x06) ;          // shift cursor right
        COMMAND(0x01) ;          // clear display
        COMMAND(0x80) ;          // cursor at line 1, position 0
        DATADISPLAY('H') ;       // send data to LCD
        DATADISPLAY('I') ;
}
void COMMAND(unsigned char cmd)
        {
        READY();                 // check busy flag
        P2 = cmd ;               // send command
        RS = 0;                  // RS = 0 for command
        RW = 0;                  // R/W = 0 for write command
        E = 1;                   // high to low pulse for write
        DELAY();
        E = 0;
        }
void DATADISPLAY(unsigned char data1)
        {
        READY();                 // check busy flag
        P2 = data1;              // send data
        RS = 1;                  // RS=1 for data
        RW = 0;                  // R/W=0 for write command
        E = 1;                   // high to low pulse for write
        DELAY();
        E = 0;
        }
```

```
void READY (void)
        {
        BUSY = 1;                   // configure P2.7 as input
        RS = 0;                     // RS=0 for command
        RW = 1;                     // R/W=1 for read command
        while (BUSY==0)
                {
                E = 1;              // high to low pulse for read
                DELAY();
                E = 0;
                }

        }
void DELAY(void)
        {
        unsigned int i ;
        for(i=0; i<7000; i++);
        }
```

4-bit Mode

When many devices are to be interfaced with the microcontroller, more I/O pins should be spared for other devices. In 4-bit mode, only the top 4 bits (D7-D4) are used for issuing data/command as shown in Figure 18.11. A byte is sent by consecutively sending two nibbles.

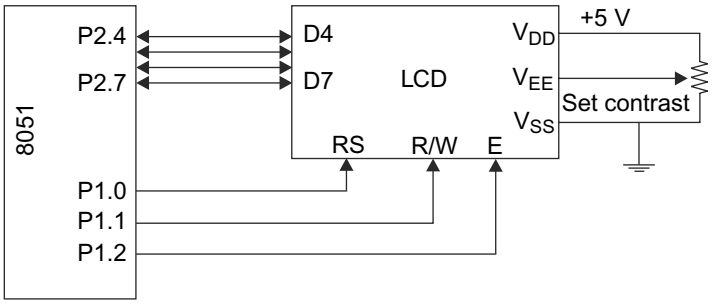


Fig. 18.11 *4-bit mode LCD interfacing with 8051*

18.4 | PROJECT: SIMPLE BURGLAR ALARM SYSTEM

Problem Statement Design a simple burglar alarm system that monitors the status of windows/doors. When an unauthorized person opens the windows/doors, the system should sound alarm and display the number of opened windows/door on a seven-segment display.

Solution A simple burglar alarm system can be designed using an 8051 microcontroller to demonstrate the use of I/O ports. The circuit of Interfacing Example 18.1 given in Figure 18.2 can be modified to realize the system. It is assumed that the authorized person will deactivate the system before opening the windows/doors. Figure 18.12 shows the diagram for the burglar alarm system.

The four switches are connected to port pins P2.0 to P2.3. Each switch is connected to a window or a door; the switches are normally closed and will be open when the corresponding window or door is opened (illegally). The wiring of these switches should have buffers connected with them for longer distance. The buffers are not shown for simplicity.

In normal condition (window/door closed), the logic level at port pins P2.0 to P2.3 is 'low'. When any one of these switches (window/door) is opened, the corresponding logic level at the port pin will be 'high' (because the port pins are pulled high by internal weak pull-up resistors).

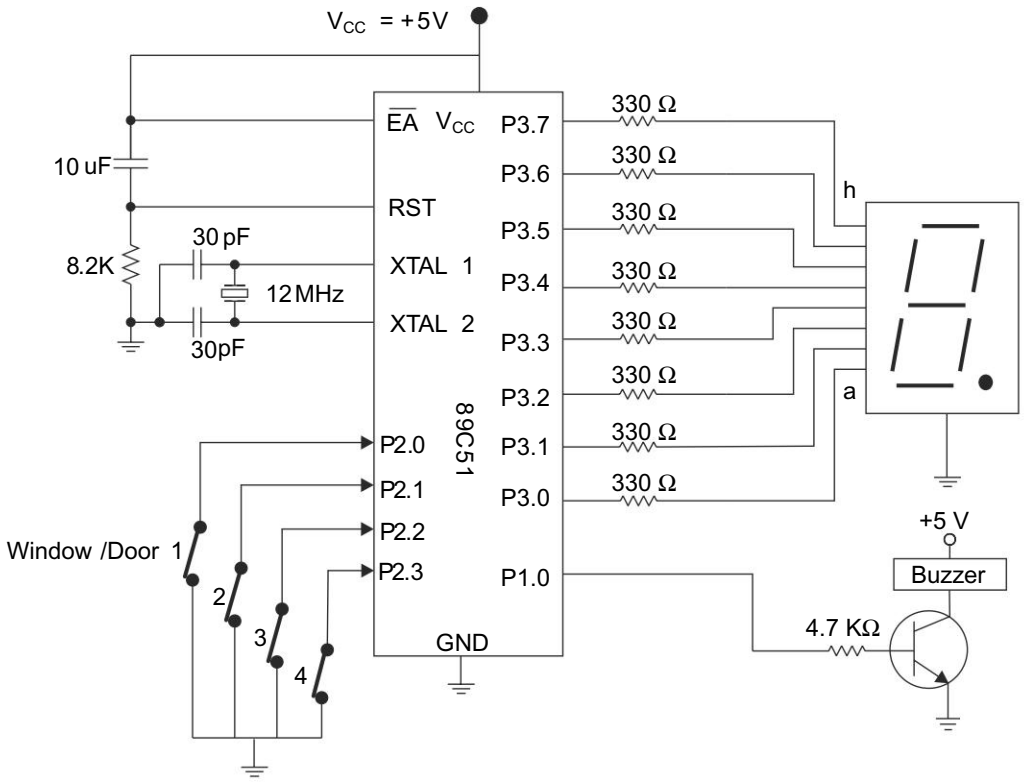


Fig. 18.12 *Simple burglar alarm system*

The system output consists of a seven-segment display and a buzzer. The common cathode seven-segment display is connected with Port 3 pins with segment 'a' connected with P3.0 and segment 'h' connected with P3.7. The number of opened windows/doors will be displayed on a seven-segment display. The buzzer is connected with Pin P1.0 through buffer. The buzzer will be made ON to sound the alarm when any of the windows/doors is opened.

Program Development

The program will continuously monitor Port 2. After reading Port 2 pin status, the upper four bits are masked because they are not used in the system. The masked result is compared with '0'. If it is '0', it indicates that all the switches are closed; therefore, no action should be taken and the program should continue monitoring Port 2. If the result after masking is not '0', it indicates that one of the windows/doors is open, the buzzer should be made ON first, and the window/door number should be displayed on the seven-segment display. This can be done by iteratively rotating the result of masking through carry flag until carry flag is 1. The counter can be incremented in each iteration to determine opened window/door number. Once it is determined, the corresponding seven-segment code is sent to the seven-segment module through Port 3. To simplify the program, only one window/door is displayed. In the case of multiple opened windows/doors, the program may be modified to display the corresponding numbers one by one.

The assembly-language program for the burglar alarm system is given below.

```
        ORG 0000H
        MOV R2, #00H            // counter for window door number
        MOV DPTR, #0100H        // address of look up table for 7 seg. codes
        CLR P1.0                // buzzer is off normally
        MOV P3, #00H            // seven-segment display is off normally
        MOV P2, #0FFH           // configure Port 2 as input
BACK:   MOV A, P2               // read status of all switches
        ANL A, #0FH             // mask upper for bits of Port 2 as they are not used
        JZ BACK                 // if all the switches are closed, keep on monitoring their
```

```
                              // status, otherwise activate buzzer to sound alarm and
                              // display number of opened window/door on 7seg.display
      SETB P1.0               // turn ON buzzer
NXT:  RRC A                   // rotate contents of A to determine number of opened window
      JC DISPLAY
      INC R2                  // increment window/door counter
      SJMP NXT
      DISPLAY:
      MOV A, R2               // place opened window number in accumulator for display
      MOVC A, @A+DPTR         // access the corresponding code from look-up table
      MOV P3, A               // display code on seven-segment module
HERE: SJMP HERE

      ORG 100H
      DB 06H, 5BH, 4FH, 66H   // codes for 1 to 4
      END
```

The equivalent C program is given below.

```
     #include<reg51.h>
     sbit buzzer=P1^0;

void main ()
     {
     unsigned char sevenseg_codes[]={0x06,0x5b, 0x4f, 0x66,};// codes for 1 to 4
     unsigned char i;
     buzzer=0;                // buzzer is off normally
     P3=0x00;                 // seven-segment display is off normally
     P2=0xFF                  // configure Port 2 as input
     do ()
        {
        i=P2                  // read status of switches
        i &= 0x0F             // mask upper for bits of Port 2 as they are not used
        }
     while (i==0)             // if all the switches are closed, keep on monitoring their
                              // status otherwise activate buzzer to sound alarm and
                              // display the number of opened window/door on 7seg. display

buzzer = 1;                   // turn ON buzzer

switch (i)
     {
     case (1):                // if switch 1 is opened, display 1
          {
          P3 = sevenseg_codes[0];
          break;
          }
     case (2):                // if switch 2 is opened, display 2
          {
          P3= sevenseg_codes[1];
          break;
          }
     case (4):                // if switch 3 is opened, display 3
```

```
          {
          P3 = sevenseg_codes[2];
          break;
          }
     case (8):                // if switch 4 is opened, display 4
          {
          P3 = sevenseg_codes[3];
          break;
          }
     }
while (1);
     }
```

Suggested Modifications

- Modify the system to accommodate 20 windows/doors.
- Connect the buffers with the switches to support longer wiring length.
- Display numbers of all opened windows/doors one by one continuously with a delay of 1 second.

THINK BOX 18.3

Which other sensors can be used in the place of switch (for this project)?
LDR, infrared sensors, motion detectors, touch sensors (or any tactile sensor)

POINTS TO REMEMBER

- LEDs are used to indicate the status of the device like powered on, running, waiting, error, etc.
- There are two types of seven-segment display modules, common cathode and common anode.
- IC7448 is BCD to seven-segment code converter and digit driver used for common cathode seven-segment display and IC7447 is used for common anode seven-segment display.
- Though software initialization of LCD is not mandatory, it is recommended that this procedure always be followed.
- D7 (pin 14) of the LCD module acts as a busy flag when read.
- In an 8-bit mode, all the eight data line pins (D7-D0) are used for giving data/commands, while in a 4-bit mode, only the top 4 bits (D7-D4) are used for issuing data/commands.

OBJECTIVE QUESTIONS

1. Interfacing LCD with 8051 will require at least _____ data lines along with the _____ signals.
   (a) 6, RS, RW (b) 5, RW, EN (c) 8, RS, EN, RW (d) 4, RS, EN, R/W
2. While sending a command code to the LCD, Status of RS, E and R/W are,
   (a) 0, High to Low pulse, 1 (b) 0, Low to High pulse, 1
   (c) 0, High to Low Pulse, 0 (d) all of the above
3. Which line will instruct the LCD that the microcontroller is sending data?
   (a) DB0 (b) RW (c) RS (d) EN
4. ___ pulse on the E pin will provide command code in the LCD.
   (a) High to Low pulse (b) Low to High pulse
5. 'Clear display' is a _____and its value is ____H.
   (a) data, 0f (b) data, 01 (c) command code, 0f (d) command code, 01
6. E pin is an _____ and R/W pin is an _____ pin for the LCD.
   (a) input, input (b) input, output (c) output, input (d) output, output

7. 'Entry mode set' command for LCD can shift,
   (a) cursor left (b) cursor right (c) display right (d) all of the above
8. The address of the leftmost character of Line 2 of 16 × 2 LCD is,
   (a) 80H (b) C0H (c) 94H (d) D4H
9. The address of the rightmost character of Line 2 of 20 × 4 LCD is,
   (a) 93H (b) D3H (c) A7H (d) E7H
10. Seven-segment code for number 5 for the common cathode display is,
    (a) 3FH (b) 5BH (c) 66H (d) 6DH

Answers to Objective Questions

1. (d) 2. (c) 3. (d) 4. (a) 5. (d) 6. (a) 7. (d)
8. (b) 9. (b) 10. (d)

REVIEW QUESTIONS WITH ANSWERS

1. LCD is an ASCII device. Justify true or false with reason.

A. True, because, it has to be given an ASCII code for a character to be displayed.

2. What are the key features of the LCD?

A. LCD can display numbers, characters and graphics. It has an inbuilt refreshing controller.

3. How can the command be given to the LCD?

A. By making RS = 0 and R/W = 0.

4. How can LCD be read?

A. By setting R/W =1.

5. LCD will not accept any command when a busy flag is set. Justify true or false with reason.

A. True. Busy flag is set to indicate that the LCD is busy in doing its internal activities, hence it cannot accept any command.

6. What are the addresses for cursor for 20 X 2 LCD?

A. For Line 1, 80H to 93H and for Line 2, C0H to D3H.

7. What is meant by 4-bit mode of LCD?

A. Only four data bits (D4 to D7) are used for data transfer between the microcontroller and LCD.

8. Under what condition is LCD properly initialized by the internal reset circuit?

A. When supply voltage reaches 4.5 V within 10 ms.

9. Why is the software initialization of LCD preferred?

A. It guarantees proper initialization irrespective of the power supply reset timing.

EXERCISE

1. Compare LCDs with LEDs.
2. What is the status of RS, R/W when sending a command code to the LCD?
3. What is the status of RS, R/W when sending a data to the LCD?
4. What is a busy flag? How can it be monitored?
5. List the cursor addresses for 20 x 4 and 40 x 7 LCD.
6. Compare 8-bit and 4-bit modes of LCD operation.
7. Write steps to initialize the LCD.
8. List instructions executed by the LCD during internal initialization.

19

Interfacing ADC, DAC and Sensors

Objectives

- Discuss the need for analog-to-digital and digital-to-analog converters
- List the common methods of conversion and parameters for the ADCs and DACs
- List the common ADC and DAC chips with their specifications
- List the pins of ADC 080X and function of each pin
- Describe handshaking process between ADC and a microcontroller
- Interface the ADC and DAC chips with the 8051
- Understand the usage of on-chip ADC
- Interface the temperature and IR sensors with the 8051
- Develop the programs for data conversions for ADCs, DACs and temperature sensors

Key Terms

- ADC 080X
- AGND/DGND
- Conversion Speed
- DAC 0808
- DAC AD557
- Differential Input
- End of Conversion
- Flash Converter
- Handshaking
- *N*-bit ADC/DAC
- R-2R
- Resolution
- Sine Wave Generation
- Start a Conversion
- Successive Approximation

19.1 ANALOG TO DIGITAL CONVERTERS

19.1.1 Need for Analog-to-Digital Converters

In our daily life, everything with which we deal and experience is analog in nature. For example, audio, video, temperature, pressure, velocity, humidity, voltage or any measurable quantity, is usually in an analog form. If we want to interface (and thus, process) an analog signal with the microcontroller, we must convert that analog signal to digital signal and this conversion is performed by circuits called analog-to-digital convertors, and usually they are referred as ADC (Analog-to-Digital Convertor).

Nowadays, a majority of signal processing is done using digital systems because of their efficient, economical, and reliable operation. The majority of real-world signals, when converted to electrical signals using sensors will remain analog in nature, but the digital systems such as microcontrollers/processors use a binary system of zeros and ones. Therefore, if we want to process the analog signals using digital systems (to take advantages of digital systems), we need to convert the analog signals into digital signals using an analog-to-digital converter.

Before we discuss how to interface an ADC with a microcontroller, we first take a look at the various methods of analog-to-digital conversion.

19.1.2 Methods of Conversion

There are many methods for converting analog signals to digital form. Each method has its own advantages and disadvantages but the choice of ADC depends mostly on an application and other factors like conversion speed, accuracy, hardware cost, and the stability.

The most common methods used for these conversions are as follows:

1. Flash (parallel) converter
2. Successive Approximation (SAR) Converter
3. Ramp (counter) Converter
4. Single/dual Slope (integrating) Converter

Yet there are many other converters like pipelined, sigma delta and hybrid ADCs. The brief comparison between the common ADCs is given in Table 19.1.

Table 19.1 *Comparison of different ADC types*

Type	Speed	Resolution	Complexity and cost	Accuracy	Application
Flash	Fastest	Poor	Highest	Good	Fast signals
SAR	High	High	Moderate	High	Communication and Instrumentation
Counter/dual slope	Slow	Good	Moderate	Good	Instrumentation

19.1.3 ADC Parameters

Analog-to-digital converter parameters help in selecting an ADC device for a particular application because these parameters (specifications) will decide the performance of the final product in which ADC is being used.

Resolution The resolution specifies the smallest input voltage change that can be detected by the converter, i.e. the change in input that causes the digital output to change by 1. The number of bits in a digital output determines resolution of the ADC. The ADCs are specified as 8, 10, 12, 16 or 24-bit ADC. For example, 8-bit ADC has $2^8 = 256$ levels and $2^8 - 1 = 255$ steps. Its resolution is 1/255, i.e. one part in 255.

Conversion Time It is the time taken by the ADC to produce a valid binary output after the application of 'Start the conversion' command.

Linearity It is the measure of how straight the transfer function is. The ADC is linear if the resolution is constant throughout the conversion range.

Accuracy It is the measure of how close the actual output is to the ideal value.

Range It is the difference between the maximum and the minimum voltage (or current) that can be applied to ADC as an input.

Other parameters are Effective Number Of Bits (ENOB), quantization error, aperture time, acquisition time and monotonicity.

It should be noted that the sampling frequency of the analog signal should be at least double compared to the bandwidth (maximum frequency) of analog signal. The analog input samples must be held constant using sample-and-hold circuits when the conversion is being performed. For slow-varying analog signals (like temperature), sample-and-hold circuits are not required because conversion time of ADC are very less compared to the rate of change of magnitude of analog signal. Many enhanced 8051 family members have in-built (on-chip) ADCs. First, we will discuss interfacing of some common ADC chips with the 8051(External ADCs) and then the on-chip ADC will be discussed.

Example 19.1

What is the resolution of (i) 8, (ii) 10, (iii) 16, (iv) 24 bit ADC for input analog voltage range 0 to 5 volts.

Solution:

(i) **8 bits:** The resolution is $1/(2^8–1) = 1/255$, i.e. 1 part in 255.

For input range of 0 to 5 V, the resolution is $V_{MAX}/(2^8–1) = 5\text{ V}/255 = 19.60\text{ mV}$

(ii) **10 bits:** The resolution is $1/(2^{10}–1) = 1/1023$, i.e. 1 part in 1023.

For input range of 0 to 5 V, the resolution is $V_{MAX}/(2^{10}–1) = 5\text{ V}/1023 = 4.88\text{ mV}$

(iii) **16 bits:** The resolution is $1/(2^{16}–1) = 1/65535$, i.e. 1 part in 65535.

For input range 0 to 5 V, the resolution is $V_{MAX}/(2^{16}–1) = 5\text{V}/65535 = 76.29\ \mu\text{V}$

(iv) **24 bits:** The resolution is $1/(2^{24}–1) = 1/16777215$, i.e. 1 part in 16777215.

For input range of 0 to 5 V, the resolution is $V_{MAX}/(2^{24}–1) = 5\text{V}/16777215 = 0.298\ \mu\text{V}$

Discussion Question Discuss the factors affecting selection of the ADC chip.

Answer The selection of the ADC for a particular application is done based on the specifications discussed in the above section. But, the specifications are selected after evaluating the following points.

- **Rate of change of analog input signal:** This will help decide the sampling rate and, therefore, conversion time required.
- **Accuracy of the conversion required:** Based on this, resolution of the ADC can be decided.
- **Range of analog input signal:** Based on this, we can decide whether a particular chip can be used or not.
- **Clock and voltage requirements of ADC chip:** ADC chips with inbuilt clock generator and operating with single V_{CC} are preferred.
- **Number of analog input signals to be processed:** This will help select the chip which supports and processes sufficient number of analog signals.

19.1.4 Common ADC Chips

The ADC080x series from National Semiconductors uses successive approximation method for conversion and is TTL compatible to be interfaced with microcontrollers/processors. Among all the members, the difference is only in the accuracy of the output. The commonly used ADC chips with a brief description are listed in Table 19.2.

Table 19.2 *Commonly used ADC chips*

Name	Description	Manufacturer
ADC0801	8-bit ADC, 100 µs conversion time, ± 0.25 LSB adjusted error	National Semiconductors
ADC0802/*03*	8-bit ADC, 100 µs conversion time, ± 0.5 LSB unadjusted/*adjusted* error	National Semiconductors
ADC0804/05	8-bit ADC, 100 µs conversion time, ± 1 LSB unadjusted error	National Semiconductors
ADC0808/*09*	8-bit, 8 channel 100 µs conversion time, ± 0.5/*1* LSB unadjusted error	National Semiconductor
AD571	10-Bit, A/D Converter, Complete with Reference and Clock	Analog Devices
MAX1204/*02*	5V, 8-Channel, Serial, 10/*12*Bit ADC with 3 V Digital Interface	Maxim
MAX195	16-Bit, Self-Calibrating, 10 us Sampling ADC	Maxim

19.1.5 ADC 0801/02/03/04/05 Chips

The ADC0801/02/03/04/05 chips are functionally the same except for the accuracy (refer the data sheet for more details). They are all 8-bit analog-to-digital converters. Their resolution is 1 part in 255 ($1/2^8$–1), i.e. it can detect minimum change of 1 part if V_{REF} is divided into 255 parts (or levels). The conversion time is minimum –100 μs and it depends on the clock signal applied to CLK R and CLK IN pins. The features of these chips are

- 8-bit successive approximation ADC
- Conversion time of 100 μs
- On-chip clock generator
- Operates on single +5 V power supply
- TTL compatible output
- Zero adjustments not required

The pin diagram of these ADC chips is shown in Figure 19.1. Note that all these chips have the same pin configurations.

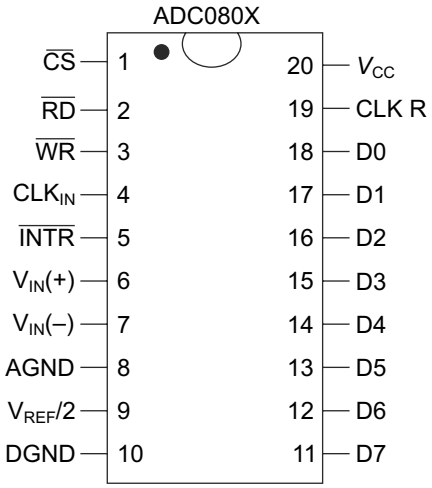


Fig. 19.1 *Pin diagram of ADC080X chips*

The pin description is given in Table 19.3.

Table 19.3 *Pin description of ADC 080X chip*

Pin No.	Pin Name	Direction	Description
1	$\overline{CS}$	Input	Active low chip select signal, made low to activate ADC chip.
2	$\overline{RD}$	Input	Active low, used to read converted digital data from the ADC chip. High to low transition (pulse) is applied to $\overline{RD}$ to read the data from data output pins.
3	$\overline{WR}$	Input	Active low, used to inform ADC chip to *start the conversion.*
4	CLK IN	Input	To use an internal clock generator, these pins are connected with R and C as shown in Figure 19.2. Clock frequency is given as f = 1/1.1RC. When the external clock source is used, it must be connected to CLK IN pin.
19	CLK R	Input	
5	$\overline{INTR}$	Output	Active low indicates that conversion is complete.
6	V_{in} (+)	Input	These pins collectively provide analog differential input voltage. $V_{in} = V_{in}$ (+) – V_{in} (–). The V_{in} (–) is normally grounded for simple applications.
7	V_{in}(–)	Input	
20	V_{cc}	Input	+5V power supply to the chip, also used as a reference voltage when $V_{REF}/2$ pin is open.
9	$V_{REF}/2$	Input	Used to set input voltage range (to set resolution) other than 0–5V, i.e. may be connected to 2 V or 0.5 V for input range 0–4 V (resolution = 4/255 = 15.68 mV) or 0–1V (resolution = 1/255 = 3.92 mV) respectively.
11-18	D_7–D_0	Outputs	Digital data output pins. D_7 MSB
8	AGND	Input	Analog and digital grounds are connected to ground of V_{in} and ground of V_{cc} respectively for isolation of V_{in} from switching transients caused by D_0–D_7.
10	DGND	Input	

Analog Inputs

ADC 080X chips have two analog input pins, V_{IN} (+) and V_{IN} (–). These two inputs are the differential inputs to the operational amplifier (internal). $V_{IN} = V_{IN}$ (+) – V_{IN} (–). The V_{IN} (–) is normally grounded for the simple applications. The differential inputs reject common mode noise.

Analog Input Voltage Range

$V_{REF}/2$ pin is used to set input voltage range. When this pin is open, the analog input voltage is in the range 0 to 5 V. Usually, input voltage range is 0 to 2 × $V_{REF}/2$. Note that the maximum differential input should not exceed 5 V. The relation of V_{IN} range with V_{REF} is shown in Table 19.4.

Table 19.4 *Analog input range selection using $V_{REF}/2$*

$V_{REF}/2$ (Volts)	V_{IN} range (Volts)	Step size (resolution)(mV)
Open (not connected)	0 to 5	5/255 = 19.60
2.0	0 to 4	4/255 = 15.68
1.28	0 to 2.56	2.56/255 = 10.03
0.5	0 to 1	1/255 = 3.92

Digital Output

Digital output will be available at pins D_7–D_0 (D_7 is MSB). These pins are tri-state buffered and the output will be available only when $\overline{CS}$ and $\overline{RD}$ are made low. The digital output is given as,

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}}$$

Digital output is digital output in decimal and V_{IN} is the differential input voltage and step size is resolution for corresponding $V_{REF}/2$.

Clock Source

The ADC080X chips have the internal clock generator (RC oscillator). Resistor and capacitor are connected to CLK R and CLK pin. The clock frequency is given as f = 1/1.1RC. The clock frequency should not exceed 1460 kHz.

Example 19.2

What will be the digital output for (i) V_{IN} = 1.28 V, (ii) V_{IN} = 2 V, and (iii) V_{IN} = 5 V. $V_{REF/2}$ pin is open. V_{IN} = [V_{in} (+) – V_{in} (–)]. Assume V_{in} (–) is 0.

Solution:

When $V_{REF}/2$ is open, analog input voltage range is 0 to 5 volts and the step size is 5/255. And we know that the digital

$$\text{output} = \frac{V_{IN}}{\text{Step size}}$$

(i) V_{IN} = 1.28 V

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{1.28}{(5/255)} = \frac{1.28}{19.60\,\text{mV}} = 65.28 \approx 65_D = 01000001_B$$

(ii) V_{IN} = 2 V

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{2}{(5/255)} = \frac{2}{19.60\,\text{mV}} = 102_D = 01100110_B$$

(iii) V_{IN} = 5 V

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{5}{(5/255)} = \frac{5}{19.60\,\text{mV}} = 255_D = 11111111_B$$

Example 19.3

Repeat Example 19.2 if $V_{REF}/2$ is 2 V.

Solution:

When $V_{REF}/2$ = 2 V, analog input voltage range is 0 to 4 volts and the step size is 4/255. And we know digital output = $\frac{V_{IN}}{\text{Step size}}$.

(i) V_{IN} = 1.28 V,

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{1.28}{(4/255)} = \frac{1.28}{15.68\,\text{mV}} = 81.60 \approx 81_D = 01010001_B$$

Note that we cannot approximate 81.60 to 82 because 0.6 corresponds to 9.41 mV which is lesser than the step size (15.68 mV), therefore, it is not detected by ADC.

(ii) V_{IN} = 2 V,

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{2}{(4/255)} = \frac{2}{15.68\,\text{mV}} = 127.5_D \approx 127 = 01111111_B$$

(iii) V_{IN} = 5 V,

We cannot apply 5 V because the input voltage range is 0 to 4 V.

THINK BOX 19.1

How can we connect the external clock source to ADC080X chips?
Connect the external clock source directly to CLK IN pin and keep CLK R pin open.

Interfacing Example 19.4

Interface ADC0804 chips with the 89C51 and write a program to take 10 samples of analog signal connected at input of the ADC. Take the sample every 1 second and store them at internal RAM addresses.

Solution:

The interfacing diagram of the ADC 080X with the 8051 is shown in Figure 19.2. Power-on reset and clock circuit of the 89C51 is not shown for the simplicity. Port 2 of the microcontroller is configured as an input and connected with digital output (D_7–D_0) pins of ADC. Control signals are connected with Port 1 pins as shown in Figure 19.2. V_{IN} (–) of ADC is grounded; therefore, the effective analog input voltage is the voltage applied at *V*in (+) pin. Here, input voltage is applied through the potentiometer (by varying the value of resistor, the input voltage can be varied between 0 to 5 V). In real life, input voltage is applied from the transducer and signal conditioning circuit. Self-clocking is (internal oscillator) used by connecting R and C as shown in Figure 19.2, for the values used, clock frequency is 606 kHz and the conversion time is 110 µs. Note that $V_{REF}/2$ pin is open; therefore, analog input range is 0 to 5 V.

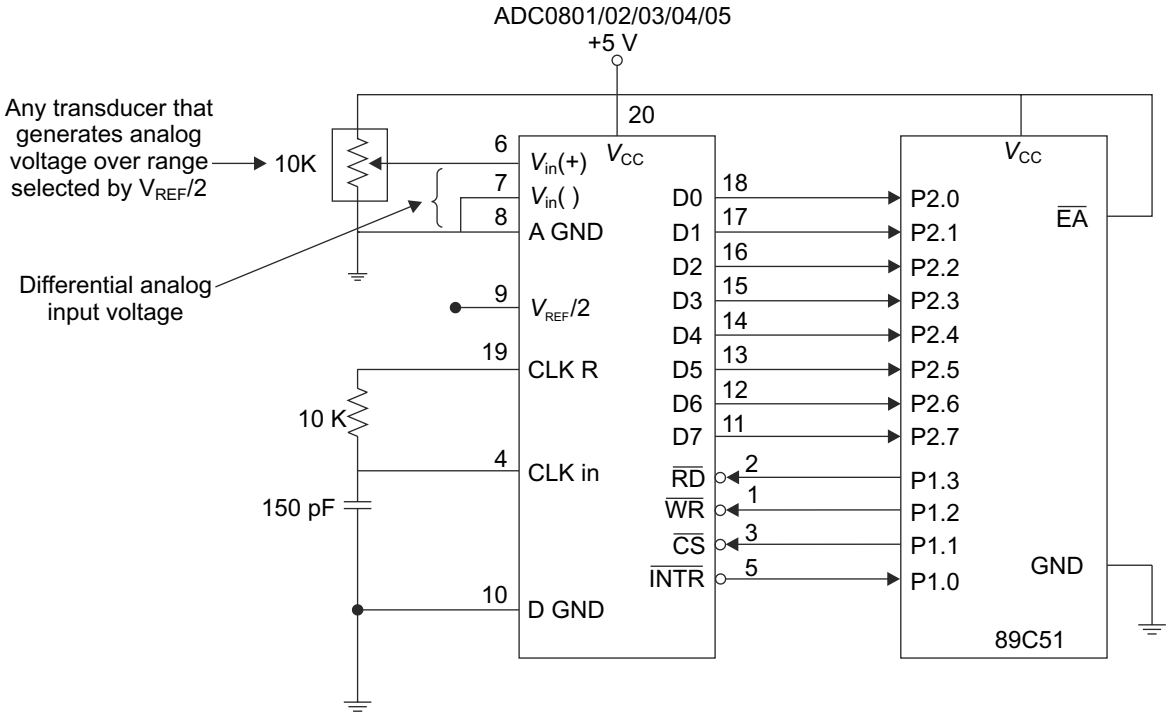


Fig. 19.2 *Interfacing ADC 080X with 8051*

19.1.6 Handshaking Process between the Microcontroller and ADC 0804 Chip

The steps to develop the program of data conversion is understood by the following steps:

- The conversion is going to start by these steps: make $\overline{CS}$ and $\overline{WR}$ simultaneously low, the ADC will remain in a reset state until the $\overline{CS}$ and $\overline{WR}$ inputs remain low. Conversion will start when one (or both) of these inputs make a low to high transition.
- Monitor the 'end of conversion' ($\overline{INTR}$) pin until it becomes 0; when it is 0, it indicates that the conversion is completed.
- To read data from the ADC, make $\overline{CS}$ and $\overline{RD}$ low, this will output the data on D_7–D_0 pins, read this data using appropriate instruction (read port to which the digital output is connected). When $\overline{RD}$ is made low, $\overline{INTR}$ will become high automatically. Remember that the port should be configured as an input.

The program will store 10 samples of data at internal RAM address 40H onwards.

```
            ORG 0000H
            MOV R0, #40H        // pointer to store samples
            MOV R2, #0AH        // counter for 10 samples
            SETB P1.0           // configure P1.0 as input for connecting INTR
            MOV P2, #0FFH       // configure P2 as input for reading data
            CLR P1.1            // make CS low
NEXT:       CLR P1.2            // make WR low
            SETB P1.2           // WR = 1, low-to-high pulse for starting ADC
HERE:       JB P1.0, HERE       // wait until the end of conversion
            CLR P1.3            // make RD low to read conversion result
            MOV A, P2           // read data from ADC
            MOV @R0, A          // store the sample at 40H onwards
            INC R0              // increment the pointer
            SETB P1.3           // make RD high before taking the next sample
            LCALL DELAY         // delay of 1 second
            DJNZ R2, NEXT       // repeat until 10 samples are taken
HERE1: SJMP HERE1

DELAY:      MOV R5, #10         // delay for 1s (Xtal = 12 MHz)
THR3:       MOV R6, #100
THR2:       MOV R7, #250
THR1:       NOP
            NOP
            DJNZ R7, THR1
            DJNZ R6, THR2
            DJNZ R5, THR3
            RET
            END
```

Example 19.5

Write the program of Interfacing Example 19.4 in the C language.

Solution:

```
#include<reg51.h>
sbit INTR = P1^0 ;
sbit CS = P1^1 ;
sbit WR = P1^2;
sbit RD = P1^3;
void DELAY (void);
void main()
        {
        unsigned char i;
        unsigned char samples[10];

        P2 = 0xFF;              // configure P2 as an input for reading data
        INTR = 1;               // configure P1.0 as input for connecting INTR
        CS = 0;                 // make CS low, enable ADC chip
        for ( i = 0; i<10; i++) // get 10 samples
        {
        WR = 0;                 // low-to-high pulse at WR will start conversion
        WR = 1;
```

```
        while (INTR = = 1);        // wait until the end of conversion
        RD = 0;                    // make RD low to read conversion result
        samples[i] = P2;           // read sample
        RD = 1;                    // make RD high before taking next sample
        DELAY();                   // delay of 1s
        }
    }
void DELAY(void)                   // delay routine of 1s approx
    {
    unsigned int j ;
    for(j = 0; j<50000; j++);
    for(j = 0; j<50000; j++);
    }
```

19.1.7 ADC 0808/0809 Chips

These chips are similar to ADC 0801/02… chips discussed just above. The difference is that they are 8-channel ADCs, i.e. 8 analog inputs can be monitored using the same chip, but only one input at a time. The functional diagram of ADC0808/09 is given in Figure 19.3.

The START signal is similar to $\overline{WR}$ of the ADC0801 except that it is active high. The channel input signals C, B and A_{LSB} are used to select one out of the 8 input channels, OE is similar to $\overline{RD}$ and EOC is similar to $\overline{INTR}$ of the ADC0801. ALE signal is used to latch the address (C, B, A) into the chip to select the input channel. Note that the clock signal has to be supplied externally. The resolution (or range of input signal) of ADC is selected by V_{REF} (+) (in fact, it is decided by V_{REF} (+) – V_{REF} (–), but V_{REF} (–) is usually grounded). The relation of Vin range with Vref is shown in Table 19.5. Note that the relation is different from 080x ADCs.

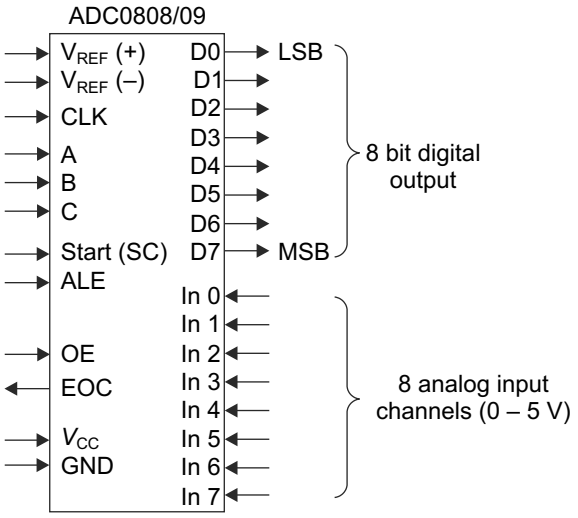


Fig. 19.3 *Functional diagram of ADC0808/09*

Table 19.5 *Analog input range selection using V_{ref}*

V_{REF} (+) (V) , when V_{REF}(–) = 0	V_{in} range (V)	Step size (resolution) (mV)
Open (not connected)	0 to 5	5/255 = 19.60
4.0	0 to 4	4/255 = 15.68
2.56	0 to 2.56	2.56/255 = 10.03
2.0	0 to 2	2/255 = 7.84
1	0 to 1	1/255 = 3.92

Example 19.6

What will be the digital output of 0808/09 for (i) V_{IN} = 1.28 V, (ii) V_{IN} = 2 V, (iii) V_{IN} = 5 V. V_{REF} (+) = 2 V and V_{REF} (–) = 0 V.

Solution:

When V_{REF} (+) = 2 V and V_{REF} (–) = 0 V, analog input voltage range is 0 to 2 volts and the step size is 2/255. And we know digital

$$\text{output} = \frac{V_{IN}}{\text{Step size}}$$

(i) V_{IN} = 1.28 V,

$$\text{Digital output} = \frac{V_{IN}}{\text{Step size}} = \frac{1.28}{(2/255)} = \frac{1.28}{7.84\,\text{mV}} = 163.2 \approx 163_D = 10100011_B$$

(ii) V_{IN} = 2 V,

Digital output = $\frac{V_{IN}}{\text{Step size}} = \frac{2}{(2/255)} = \frac{2}{7.84\,\text{mV}} = 255_D = 11111111_B$

(iii) V_{IN} = 5 V,

We cannot apply input greater than 2 V for given case.

Example 19.7

ADC 0808 is interfaced with the 8051. Write a C program to perform A/D conversion for all the channels one by one. Store the result into an array.

Solution:

We will define the function ADCONVERT which will accept the channel number as input parameter and perform the conversion operation. The channel numbers (one by one) are given to this function using for loop. Assume that digital output is connected to port 2.

Steps to develop the program for A to D conversion are

- Select input channel number by providing A, B and C
- Latch this channel number (address) by making ALE = 1
- Apply low-to-high pulse to start the conversion (SC = 1)
- Wait until the conversion is complete
- Enable output of ADC chip by applying low-to-high pulse on OE pin
- Repeat this process for all the input channels

```
#include<reg51.h>
sbit ADDR_A = P1^0;              // A pin of ADC is connected with P1.0
sbit ADDR_B = P1^1;              // B pin of ADC is connected with P1.1
sbit ADDR_C = P1^2;              // C pin of ADC is connected with P1.2
sbit ALE = P1^3;                 // ALE pin of ADC is connected with P1.3
sbit OE = P1^4;                  // OE pin of ADC is connected with P1.4
sbit SC = P1^5;                  // SC pin of ADC is connected with P1.5
sbit EOC = P1^6;                 // EOC pin of ADC is connected with P1.6
unsigned char ADCONVERT (unsigned char);
void delay (void);
void main()
        {

        unsigned char i, digital_sample[8];
        P2 = 0xFF;               // configure P2 as input as it is connected with D7-D0 of ADC
        EOC = 1;                 // configure EOC as an input
        ALE = 0;                 // clear ALE
        OE = 0;                  // clear OE
        SC = 0;                  // clear SC

        for (i = 0 ; i<8 ; i++)
        {
        digital_sample[i] = ADCONVERT(i);          // call convert function and store samples
        }
        }

unsigned char ADCONVERT (unsigned char j)
{       unsigned char result;
                                 // select input channel
```

```
        ADDR_A = (j & 01);        // LSB of the channel number
        ADDR_B = (j & 02)>>1;     // 2nd bit of the channel number
        ADDR_C = (j & 04) >>2;    // MSB 3rd bit of the channel number
        delay();
        ALE = 1;                  // latch the channel address
        delay();
        SC = 1;                   // start the conversion
        delay();
        ALE = 0;
        SC = 0;
        while (EOC ! = 0);        // wait until the end of conversion
        while ( EOC ! = 1);       // wait for EOC to become high again
        OE = 1;                   // enable output of ADC
        delay();
        result = P2;              // read result from P2
        OE = 0;
        return result;            // return the conversion result
}

void delay()
{       unsigned char k;
        for (k = 0 ; k<25 ;k++);
}
```

19.1.8 Serial ADC Chips

The ADC chips discussed thus far are parallel devices, i.e. they provide output in parallel form—all bits at a time. The disadvantages of these devices are,

- As the resolution (number of digital output bits) increases, total number of pins of a chip also increases, which will result in increase in the size of the final product.
- More number of digital output bits will also consume more number of pins of the microcontroller/processor which probably leaves no more pins free for interfacing other hardware.

To overcome these problems, serial ADCs are developed which have single Data output pin irrespective of the resolution of ADC. In a serial ADC, all the output bits are sent out serially one bit at a time.

Serial ADC Chip MAX1112/MAX1113

The MAX1112/MAX1113 are serial ADC chips from Maxim Corporation, the features of these chips are

- 8-bit serial (SPI based) ADC with 8 channel analog inputs
- Internal track/hold, voltage reference, clock, and serial interface
- They operate from a single power supply (+4.5 V to +5.5 V)
- Consume 135 μA current when sampling at rate up to 50 Ksps
- Successive approximation based ADC
- Software configurable unipolar/bipolar and single-ended/differential operation

The only difference between a MAX1112 and MAX1113 is that the MAX1112 has 8 channels of analog inputs while MAX1113 has 4 channels of analog inputs. (As a result, MAX1112 is 20 pin chip and MAX1113 is 16 pin chip). The designer should select a suitable chip as per the application requirements. These chips have a single D_{OUT} pin on which the data is serially available after conversation. The pin-diagram of these pins is shown in Figure 19.4.

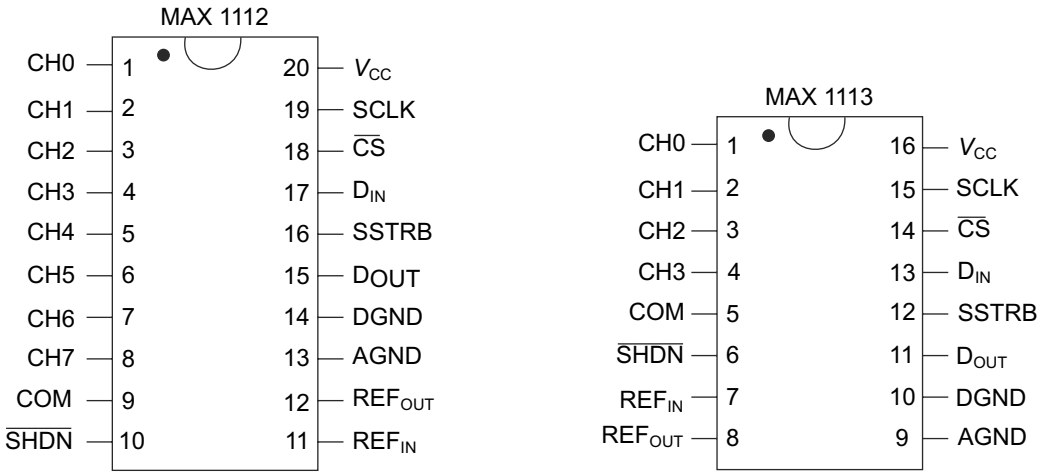


Fig. 19.4 *Pin-diagram of Serial ADC chips MAX1112/MAX1113*

Pin Description of MAX1112/1113

Table 19.6 gives the pin description of serial ADC chips MAX 1112/MAX1113.

Table 19.6 *Pin description of MAX1112/1113*

Pin		Name	Function
MAX1112	MAX1113		
1-4	1-4	CH0-CH3	Analog input channels. For single-ended mode, each input can be used as an analog input, where COM pin is used as a reference. In the differential mode, we have four sets (two sets for 1113) of differential inputs. The pairs are CH0-CH1, CH2-CH3 and so on. The channel is selected by receiving the control byte through D_{IN} pin.
5-8	–	CH4-CH7	
9	5	COM	Ground for analog inputs in a single-ended mode.
10	6	$\overline{SHDN}$	Normally floats. If low, ADC is shut down to save power (10 μA current); otherwise, the chip is fully operational.
11	7	REF_{IN}	Reference input voltage. It decides the step size. Connected to REF_{OUT} to use the internal reference.
12	8	REF_{OUT}	Internal reference generator output. A 1 μF bypass capacitor is connected between this and AGND.
13	9	AGND	Analog ground.
14	10	DGND	Digital ground.
15	11	D_{OUT}	Serial data output. Data is clocked out one bit at a time on H-to-L (falling) edge of SCLK.
16	12	SSTRB	Serial strobe output. In the internal clock mode, it becomes low when conversion begins and high when the conversion is complete. In the external clock mode, SSTRB remains high for two clock periods before the MSB is shifted out. High impedance when CS is high (for external clock mode only).
17	13	D_{IN}	Serial data input. Data is clocked in at SCLK positive (L-to-H) edge.
18	14	CS	Chip select (active low). Used to enable the chip, data is clocked into D_{IN} only if CS is low. When it is high, D_{OUT} is high impedance.
19	15	SCLK	Serial clock input is used to bring the data out and send in the control byte (one bit at a time). In the external clock mode, the conversion speed is set by this. (Duty cycle must be 45% to 55%).
20	16	V_{DD}	Positive supply voltage, +4.5 V to +5.5 V.

Control Byte of MAX1112/MAX1113

The control byte is used to perform the following operations:

- Select input analog channel
- Select unipolar or bipolar result mode
- Select single-ended or differential input mode
- Select internal or external clock mode
- Fully-operational or power-down mode

Table 19.7 shows the control byte format. The control byte is sent to MAX1112 serially one bit at a time (MSB first) via the D_{IN} pin with the help of SCLK. MSB is always high which indicates the start of the control byte.

Table 19.7 *Control byte of MAX 1112/1113*

D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
START	SEL2	SEL1	SEL0	UNI/BIP	SGL/DIF	PD1	PD0
MSB							LSB

BIT	NAME	DESCRIPTION
7	START	START = 1 after CS goes low indicates start of the control byte and is sent first
6	SEL2	Select the input channel to be used for the conversation See Table 19.8 for values of this bit for all channels
5	SEL1	
4	SEL0	
3	UNI/BIP	1 = uni polar output mode, output is between 00- FFH
		0 = bipolar output mode, output is in 2's complement
2	SGL/DIF	1 = single-ended input mode, inputs are single-ended with COM as reference
		0 = differential input mode, the voltage difference between the two channels is considered as input (the pairs are CH0-CH1, CH2-CH3 and so on)
1	PD1	1 = fully operational
		0 = power-down to save power
0	PD0	1 = external clock mode, the conversion speed is decided by SCLK.
		0 = internal clock mode, SSTRB pin goes high to signal the end of conversion

Table 19.8 *Analog input channel selection*

SEL2	SEL1	SEL0	Analog channel selected
0	0	0	CH0
0	0	1	CH1
0	1	0	CH2
0	1	1	CH3
1	0	0	CH4
1	0	1	CH5
1	1	0	CH6
1	1	1	CH7

Sending Control Byte to MAX 1112/MAX1113

The timing diagram of sending the control byte to ADC chip is shown in Figure 19.5.

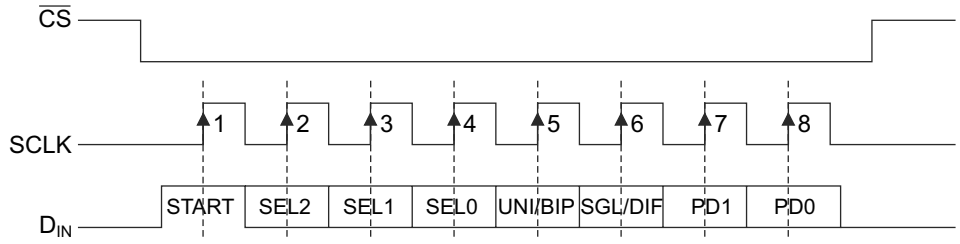


Fig. 19.5 *Timing diagram for sending the control byte to MAX1112/1113*

The timing of sending control byte to ADC chips is explained in the following steps. Configure the command word as per the requirement and to develop a program, follow the steps given below.

- Load the counter variable with 8 to send 8 bits serially.
- Load accumulator with required control byte.
- Select ADC chip by clearing Chip Select signal ($\overline{CS} = 0$).
- Clear carry flag and move MSB of the accumulator into carry flag using RLC instruction.
- Send content of the carry flag to D_{IN} pin through a port pin.
- Give positive pulse (low to high) to SCLK pin, it will send START bit (see dotted lines in the timing diagram as shown in Figure 19.5).
- Rotate contents of Accumulator, decrement the counter and repeat the above steps (step 4 to 6) for sending all the 8 bits.

When the last bit of the control byte (PD0) is sent into ADC chip, the conversation starts and SSTRB goes low.

Reading Digital Output Data (Result) from the Serial ADC Chip

When analog-to-digital conversion is complete, SSTRB goes high. Now the 8-bit digital data can be brought out of the MAX1112/MAX1113 from D_{OUT} pin using SCLK. Each negative (high to low) pulse to the SCLK pin gives one bit of the digital data (MSB first) from D_{OUT} pin. Note that MAX1112 gives MSB out at the second negative pulse of SCLK after SSTRB goes high. Therefore, we have to send 9 SCLK pulses to get the complete data byte. This process is illustrated in Figure 19.6.

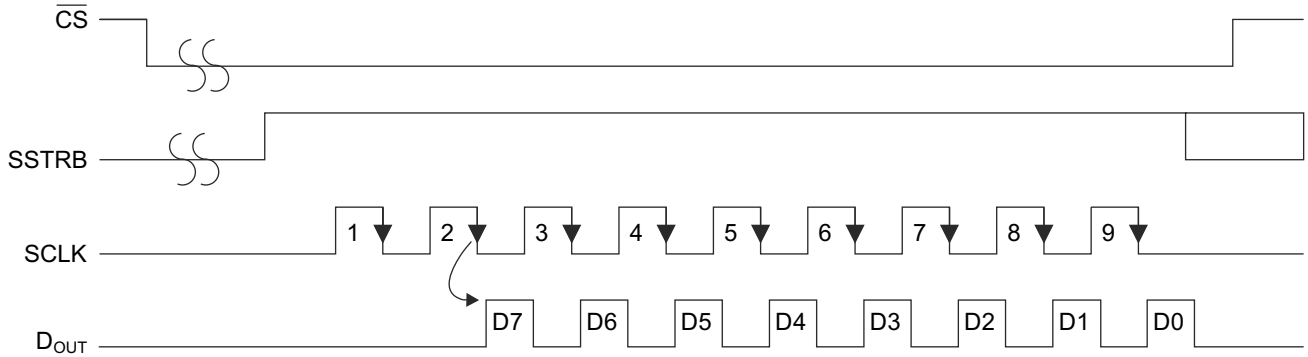


Fig. 19.6 *Timing diagram for reading digital output byte from MAX1112/MAX1113*

To develop a program to read a data byte from ADC chip, follow the steps given below:

- Wait for the completion of conversion by monitoring SSTRB pin (SSTRB = 1 indicates end of the conversion).
- Load the counter variable with 8 to read 8 bits serially.
- Give a negative pulse (high to low) to SCLK pin, it will place MSB of digital data on D_{OUT} pin (see curved arrow in the Figure 19.6).

- Read data from D_{OUT} pin to carry flag through the port pin.
- Shift accumulator to left through the carry using RLC instruction
- Decrement the counter and repeat the process (step 3 to 5) until the counter becomes 0.

Note that one extra negative pulse has to be given to read the data byte.

The above steps are effectively implementing SPI protocol using software, i.e. it is example of 'Bit Banging' a SPI protocol. (Refer Chapter 22, topic 22.6 for more details of SPI)

Interfacing Example 19.8

Interface the MAX1112 with the 89C51. What will be the control word for converting the input at CH4 and single-ended, unipolar mode with internal clock and fully operational mode?

Solution:

To select CH4,

Bits SEL2 = 1, SEL1 = 0 and SEL0 = 0;

For single-ended and unipolar mode,

Bits UNI/BIP = 1, SGL/DIF = 1

And for using the internal clock and fully operational mode,

Bit PD1 = 1, PD0 = 0

The START bit must be 1,

Therefore, command word will be CEH.

The interfacing of MAX1112 with the 89C51 is shown in Figure 19.7. As shown in the figure, SHDN pin is permanently connected to V_{cc}, so the ADC is not in shut-down mode, $\overline{CS}$ is connected to P2.3, the control word has to be supplied through P2.0 because it is connected with D_{IN} of the ADC and the digital data is read through P2.1 as it is connected to D_{OUT}. End conversion (SSTRB) is monitored through P2.4. The analog input is given to CH4 through a potentiometer; any voltage between 0 to 5 V can be given as analog input by setting the knob of potentiometer. In practice, an input voltage is applied from the transducer and the signal conditioning circuit.

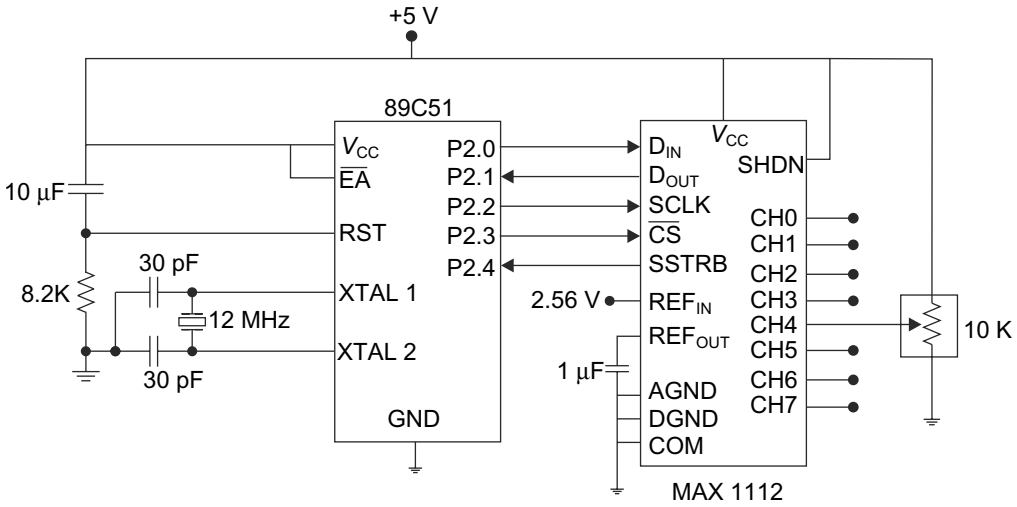


Fig. 19.7 *Interfacing of the MAX1112 with the 89C51*

Example 19.9

Write a program to send the control byte into MAX1112 for the circuit shown in Figure 19.7. Read the result after the conversion and send it to Port 1.

Solution:

```
// following part of the program will send control byte to ADC
ORG 0000H
```

```
        DATAIN BIT P2.0         // assign name DATAIN to Pin P2.0
        DATAOUT BIT P2.1        // assign name DATAOUT to Pin P2.1
        SCLK BIT P2.2           // assign name SCLK to Pin P2.2
        CS BIT P2.3             // assign name CS to Pin P2.3
        STROB BIT P2.4          // assign name STROB to Pin 2.4

        MOV A, #0CEH            // select CH4, single ended, unipolar mode
        MOV R2, #08             // load count for 8 bits
        CLR CS                  // CS = 0, enable the ADC chip
        CLR C                   // clear carry flag, bits are sent through carry
NEXT:   RLC A                   // place MSB into carry
        CLR SCLK                // low SCLK Low-to-High pulse
        MOV DATAIN, C           // send bit to Din
        ACALL WAIT              // wait
        SETB SCLK               // Low-to-High pulse to latch data in ADC
        ACALL WAIT              // delay
        DJNZ R2, NEXT           // repeat the operation for all 8 bits
        SETB CS                 // CS = 1
        CLR SCLK                // SCLK = 0 during conversation

        // following part of the program will read the digital data from ADC

        SETB STROB              // configure STROB as an input
        SETB DATAOUT            // configure DATAOUT from ADC as an input
HERE1:  JNB STROB, HERE1        // wait until the end of conversion
        SETB SCLK               // high-to-low pulse (extra pulse)
        ACALL WAIT
        CLR SCLK
        ACALL WAIT
        CLR A                   // data will be stored in A
        MOV R2, #08             // counter to read 8 bits
NEXT1:  SETB SCLK               // high-to-low pulse will provide data bit at Dout
        ACALL WAIT
        CLR SCLK
        ACALL WAIT
        MOV C, DATAOUT          //read data bit into carry flag
        RLC A                   //move the bit in A
        DJNZ R2, NEXT1          //repeat for all the 8 bits
        SETB CS                 //CS = 1
        MOV P1, A               //send digital data to Port 1
HERE2:  SJMP HERE2

WAIT:   MOV R3, #0FFH           //delay subroutine
HERE:   DJNZ R3, HERE
        RET
        END
```

Example 19.10

Rewrite the program of Example 19.9 in the C language.

Solution:

Assume the delay routine 'wait' is available.

```
#include<reg51.h>
sbit DATAIN = P2^0;             // define Pin P2.0 as DATAIN
```

```
sbit DATAOUT = P2^1;          // define Pin P2.1 as DATAOUT
sbit SCLK = p2^2;             // define Pin P2.2 as SCLK
sbit CS = P2^3;               // define Pin P2.3 as CS
sbit STROB = P2^4;            // define Pin 2.4 as STROB
sbit MSB_A = ACC^7;           // define MSB of Accumulator as MSB_A
sbit LSB_A = ACC^0;           // define LSB of Accumulator as LSB_A

void main(void)
        {
        unsigned char controlbyte = 0xCE;          // select CH4, single-ended, unipolar mode
        unsigned char i;
        ACC = controlbyte;                         // write control byte into A
        CS = 0;                                    // CS = 0, enable the ADC chip
                for(i = 0; i<8; i++)               // loop to send 8 bits of control byte
                        {
                        SCLK = 0;
                        DATAIN = MSB_A;            // send A.7 (MSB) of A to Din pin of ADC
                        wait();
                        SCLK = 1;                  // Low-to-High pulse to latch data in ADC
                        wait();
                        ACC = ACC<<1;              // place the next bit to be sent into MSB of A
                        }
        DATAOUT = 1;                               // configure Dout as input from ADC
        STROB = 1;                                 // configure STROB as an input from ADC
        CS = 0;                                    // select the ADC chip
        while (STROB = = 0);                       // wait until the end of conversion
        SCLK = 1;                                  // high-to-low pulse (extra pulse)
        wait();
        SCLK = 0;
        wait();
                for(i = 0; i<8; i++)               // loop to receive 8 bits result
                        {
                        SCLK = 1;                  // high-to-low pulse will provide data bit at Dout
                        wait();
                        SCLK = 0;
                        wait();
                        LSB_A = DATAOUT;           // read data bit from DOUT of ADC into LSB of A
                        ACC = ACC << 1;            //shift left contents of A
                        }
        CS = 1;
        P0 = ACC;                                  //send the result of conversion on P0
        }
```

Common Serial ADC Chips

The list of common serial ADC chips along with their parameters is given in Table 19.9.

Table 19.9 *Serial ADC chips*

ADC chip No.	Manufacturer	Bits	Conversion Time	Package and Pins
AD677	Analog Devices	16	10 μs	16-DIP 28-SOIC

Contd.

ADC chip No.	Manufacturer	Bits	Conversion Time	Package and Pins
AD7893	Analog Devices	12	6 µs	8-DIP, SOIC
AD7898	Analog Devices	12	3.3 µs	8-SOIC
MAX186 MAX188	Maxim	12	10 µs	20-DIP, SO, SSOP
MAX187 MAX189	Maxim	12	8.5 µs	8-PDIP 16-SO
ADC0831/ 832/834	Texas Instruments	8	32 µs	8-14- 20- DIP
TLC1549C, TLC1549I, TLC1549M	Texas Instruments	10	21 µs	8-D, JG, P 20-FK

19.1.9 On-chip ADCs

The new enhanced versions of the 8051 usually have ADC(s) built on the chip. For example, AT87C5111/12, P87LPC768/69, P89LPC933/34/35 chips have on-chip multichannel ADC(s). We will discuss on-chip ADC of P87LPC768 microcontroller.

P89LPC768

The P87LPC768 is a 20-pin enhanced 80C51 microcontroller designed for low pin count applications demanding a high degree of integration and low-cost solutions. It takes 6 clocks per machine cycle, i.e. it executes instructions at double the speed than 8051 when operating at the same clock frequency. It has the following features:

- 4Kbytes of EPROM as a code memory
- 4-channel, 8-bit ADC
- 4-channel, 10-bit PWM
- Two analog comparators
- On-chip RC oscillator
- LED drive capability (20 mA) on all port pins
- 2.7 V to 6.0 V operating range
- Two 16-bit counters, UART, I2C, oscillator fail detect, 8 key-pad interrupt inputs, four interrupt-priority levels, idle and power-down modes

Refer the datasheet for more details.

The operation of on-chip ADC of P89LPC768 is controlled by special function register ADCON and result of the conversion is stored in DAC0 register. The bit assignment with a brief description of ADCON register is given in Table 19.10.

Table 19.10 *ADCON register*

ENADC	–	–	ADCI	ADCS	RCCLK	AADR1	AADR0
MSB							LSB

Bit	Symbol	Description
7	ENADC	The ADC is enabled by setting this bit (ENADC = 1). It must be set 10 µs before a conversion is started.
6	–	Reserved for future use. Should not be set to 1.
5	–	Reserved for future use. Should not be set to 1.
4	ADCI	Conversion complete flag. This flag is set when an A to D conversion is completed. This bit will generate an hardware interrupt, if enabled. This bit must be cleared by the user program.

Contd.

3	ADCS	Start conversion. Setting this bit starts the conversion of the selected ADC input. It remains set while the conversion is in progress and is cleared automatically when the conversion is finished. When ADCS = 1 or ADCI = 1, new start commands are ignored.
2	RCCLK	If RCCLK = 0, the CPU clock is used as the ADC clock. When RCCLK = 1, the internal RC oscillator is used as the clock source.
1	AADR1	These two bits are used to select the analog input channel for the conversion. The status 00, 01, 10 and 11 of these two bits selects AD0 (P0.3), AD1 (P0.4), AD2 (P0.5) and AD3 (P0.6) inputs respectively for the conversion.
0	AADR0	

Note that P89LPC768 has a very limited number of pins. Therefore, the ADC power supply and references are shared with the microcontrollers power pins V_{DD} and V_{SS}. The A/D converter operates down to a V_{DD} supply of 3.0 V. The programming of on-chip ADC is illustrated in Example 19.11.

Example 19.11

Write a program to convert analog inputs applied at AD1 as well as AD2 inputs of P89LPC768 microcontroller. Store the results at internal RAM memory addresses 20H and 21H.

Solution:

Assume that the CPU clock is used as a clock source for conversion (RCCLK = 0) and polling of ADCI flag is done to monitor the end of the conversion. Assuming that the crystal frequency is 12 MHz, the conversion time is 15.5 μs

The assembly-language program is given below:

```
        ORG 0000H
        MOV PT0AD, #78H     //disable digital inputs on ADC input pins without affecting other pins
        ANL P0M2, #87H      //disable digital outputs on ADC input pins without affecting other pins
        ORL P0M1, #78H      // disable digital outputs on ADC input pins without affecting other pins
        MOV ADCON, #81H     // enable ADC and select AD1 (P0.4) input for the conversion and
                            // CPU clock as a clock source for conversion
        NOP                 // allow time for stabilization (at least 10 μs)
        NOP
        SETB ADCS           // start the conversion
WAIT:   JNB ADCI, WAIT      // wait until the conversion is complete
        MOV 20H, DAC0       // store result at internal RAM address 20H
        CLR ADCI            // clear ADCI flag
        MOV ADCON, #82H     // select AD2 (P0.5) input for the conversion
        SETB ADCS           // start the conversion
WAIT1:  JNB ADCI, WAIT1     // wait until the conversion is complete
        MOV 21H, DAC0       // store result at internal RAM address 21H
        CLR ADCI            // clear ADCI flag
HERE:   SJMP HERE
        END
```

Example 19.12

Rewrite the program of Example 19.11 in the C language.

Solution:

The corresponding C program is given below

```
#include<reg768.h>
#include <absacc.h>             // include file for DBYTE

void main()
{
```

```
PT0AD = 0x78;              // disable digital inputs on ADC input pins without affecting other pins
P0M2 & = 0x87;             // disable digital outputs on ADC input pins without affecting other pins
P0M1| = 0x78;              // disable digital outputs on ADC input pins without affecting other pins
ADCON = 0x81;              // enable ADC and select AD1 (P0.4) input for the conversion and CPU
                           // clock as a clock source for conversion
ADCS = 1;                  // start the conversion
while (ADCI = = 0);        // wait until the conversion is complete
DBYTE[0x20] = DAC0;        // store result at internal RAM address 20H
ADCI = 0;                  // clear ADCI flag
ADCON = 0x82;              // select AD2 (P0.5) input for the conversion
ADCS = 1;                  // start the conversion
while (ADCI = = 0);        // wait until the conversion is complete
DBYTE[0x21] = DAC0;        // store result at internal RAM address 21H
ADCI = 0;                  // clear ADCI flag
while(1);                  // end
}
```

19.1.10 Applications of ADCs

ADCs are used wherever an analog signal has to be processed, stored or transmitted in digital form. They form one of the most important components of an embedded system. Only a few applications are listed below:

- Mobile phones
- TV tuner cards, software defined radios
- Digital storage oscilloscopes
- Sound, music and video recording
- Data logging remote digital signal processing
- Data acquisition systems
- Industrial process control
- All systems that process the analog signals (like temperature, pressure, velocity, humidity, voltage) using digital processing

THINK BOX 19.2

Make a list of 8051-based microcontrollers having on-chip ADC.

DS87C550:10 bit-8 channel (Dallas Semiconductors), ADμC812:12 bit-8 channel, ADμC816:16 bit-2 channel, ADμC824:12 bit and 16 bit ADC (Analog Devices), AT89C5132:10 bit-2 channel, AT89C51AC3:10 bit-8 channel (Atmel), P80C592:10 bit-8 channel, P89LPC9103:8 bit-4 channel (NXP), C8051F00x:12 bit-8 channel, C8051F041:12 bit-12 channel (Silicon Laboratories).

19.2 | DIGITAL-TO-ANALOG CONVERTERS

The Digital-to-Analog Converter (DAC) is a device used to convert the digital signal to analog signal. The processed digital signal must be converted back to an equivalent analog voltage or current signal before they can be given back to the real-world application. The simple DAC is made from an op-amp and either binary weighted resistors or R-2R resistor circuits. The problem with the binary weighted DAC is that it requires binary weighted resistor values (X Ω, 2X Ω, 4X Ω, 8X Ω…) which may not be readily available, especially if the number of digital inputs are more (usually greater than four); therefore they are not precise. R-2R DACs are more popular and precise because it requires only two values of resistors.

19.2.1 DAC Parameters

Resolution It is the smallest apparent change in the output, i.e. the change in the output that is caused when the digital input changes by 1. The number of bits of digital input determines the resolution of the DAC. The DACs are specified

as 8, 10, 12 or 16-bit DAC. For example, 8-bit DAC has $2^8 = 256$ levels and $2^8 - 1 = 255$ steps in output. Its resolution is 1/255, i.e. one part in 255

Settling Time When a digital input is applied, the analog output oscillates for some time before giving a final value, the time required to settle the output voltage within ± 0.5 LSB of the final value is referred as *settling time*.

Other parameters like linearity, accuracy, range are similar to analog to digital converter chips.

19.2.2 Common DAC Chips

The commonly used ADC chips with a brief description are listed in Table 19.11.

Table 19.11 *Common DAC chips*

DAC	Description	Manufacturer
AD557	8-Bit DAC, 0 to 2.56 V O/P, single supply operation	Analog Devices
DAC8043	12-Bit serial input DAC converter	Analog Devices
MAX 505/506	Quad 8-bit DAC, single or dual supply operation	Maxim
MAX509/510	Quad 8-bit serial DAC, single or dual supply operation	Maxim
MAX520/521	Quad/Octal, 2-wire serial 8-bit DACs	Maxim
TLC5615	10-Bit serial DAC, single supply operation	Texas Instruments
DAC0808	8-Bit DAC	National Semiconductor

19.2.3 DAC AD557 Chip

The AD557 is an 8-bit DAC which provides analog output in the form of voltage. It has data input latches to support a direct interface with microcontrollers/processors. It operates using single supply voltage of +5 V. Its output voltage range is in between 0 to 2.56 V. No external components are required to interface it with the microcontroller/processor.

The output voltage is given by,

$$V_{out} = \frac{V_{in}\text{ (decimal equivalent)}}{255} \times 2.56$$

Therefore, resolution of DAC AD557 is,

$$V_{out} = \frac{1}{255} \times 2.56 = 10.039 \text{ mV}$$

It is a change in output voltage caused by 1-bit change in binary input.

Example 19.13

What will be the output of DAC AD557 when digital input is
(i) 1000 0000B, and (ii) 1111 1111B

Solution:

(i) Input 1000 0000 is equal to 128 decimal

Therefore,

$$V_{out} = \frac{V_{in}\text{ (decimal equivalent)}}{255} \text{ x } 2.56 = \frac{128}{255} \text{ x } 2.56 = 1.285 \text{ V.}$$

(ii) Input 1111 1111 is equal to 255 decimal

Therefore,

$$V_{out} = \frac{V_{in}\text{ (decimal equivalent)}}{255} \text{ x } 2.56 = \frac{255}{255} \text{ x } 2.56 = 2.56 \text{ V.}$$

Interfacing Example 19.14

Interface the DAC AD557 with the 89C51 and write a program to generate a sine wave.

Solution:

The interfacing of the AD557 with the 89C51 is shown in Figure 19.8.

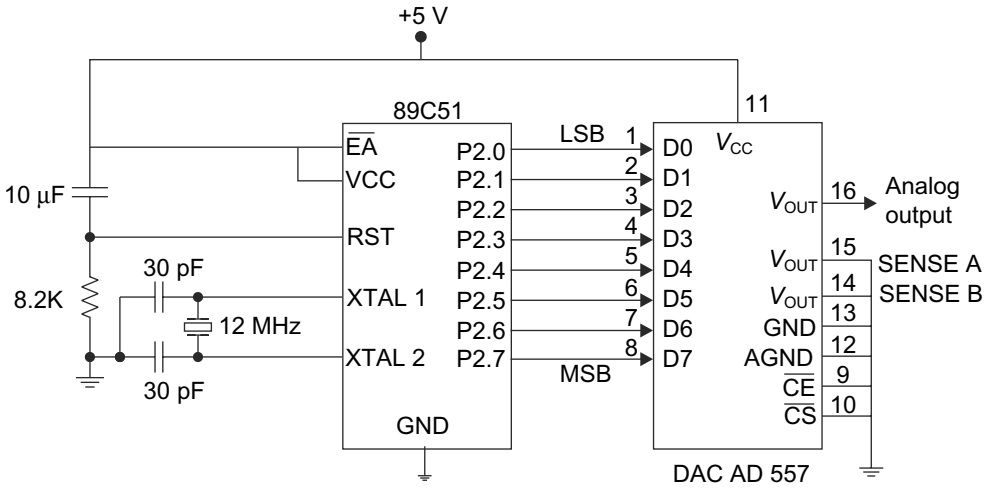


Fig. 19.8 *Interfacing DAC AD557 with 8051*

The AD557 has data input latches and these latches are controlled by Chip Enable ($\overline{CE}$) and Chip Select ($\overline{CS}$) inputs. $\overline{CE}$ and $\overline{CS}$ are internally "NORed"; therefore, the latches accepts and sends input data to the DAC section when both $\overline{CE}$ and $\overline{CS}$ are "0". When any (or both) of these two go to Logic "1," the input data is latched into the registers and held until both the signals are returned to "0." Note that $\overline{CS}$ and $\overline{CE}$ should be made low to perform the conversion. Digital input pins (D7-D0) is connected with Port 2. V_{OUT} SENSE A and V_{OUT} SENSE B are used to change the effective gain of the output buffer and in effect, the output voltage scale can be changed. (Refer datasheet of DAC AD557.)

Program to Generate Sine Wave using DAC AD557

It is preferred to use a look-up table to store magnitudes for sine function for the different values of angles. We know that sin θ = –sin (π + θ), i.e. the values of sin 0° to sin 180° are the same as values from sin 180° to sin 360° except for the sign; therefore, we need to make look up table only up to values of sin180°, the same values can be reused with a negative sign for sin 180° to sin 360°. The values in the lookup table are calculated as shown in Table 19.12.

Table 19.12 *Look-up table for sine wave*

Angle (θ)	Sin θ	Scaled value V_{OUT} = 1.28 V + 1.28 sin θ	Value for DAC = V_{OUT} × 99.61
0	0.00	1.28	128
10	0.17	1.50	150
20	0.34	1.72	171
30	0.50	1.92	191
40	0.64	2.10	209
50	0.77	2.26	225
60	0.87	2.39	238
70	0.94	2.48	247
80	0.98	2.54	253
90	1.00	2.56	255
100	0.98	2.54	253

Contd.

Contd.

110	0.94	2.48	247
120	0.87	2.39	238
130	0.77	2.26	225
140	0.64	2.10	209
150	0.50	1.92	191
160	0.34	1.72	171
170	0.17	1.50	150
180	0.00	1.28	128

We have taken V_{OUT} = 1.28 V + 1.28 sin θ to ensure that only the positive values are sent to DAC. Here, –1 to +1 values of sine function is mapped to 0 to 2.56 V because full-scale output voltage is +2.56 V. The full-scale output voltage is reached in 255 steps of 10 mV. Hence, input value (steps) required to generate 1 volt output is calculated as 255 steps/2.56 V = 99.61. Therefore, the value sent to DAC input to get output voltage is 99.61×V_{OUT} .The values for sin 180° to sin 360° can be calculated by subtracting the corresponding value of look-up table from 255. For example, to find the value of sin 190°, we will subtract the corresponding value of sin (190–180)° = sin 10° from 255, i.e. sin190° = 255–150 = 105.

Program

```
        ORG 0000H
        MOV DPTR, #LOOKUP       // address of look-up table
REPEAT: MOV R1, #18             // 18 values in look-up table for 0° to 180°
        CLR A
NEXT:   MOV R3, A               // save A
        MOVC A,@A+DPTR          // fetch the value in look-up table
        MOV P2, A               // send to DAC
        MOV A, R3               // retrieve A
        INC A                   // next entry in the look-up table
        DJNZ R1, NEXT
        CLR A
        MOV R2, #18
                                // negative cycle 180° to 360°
NEXT1:  MOV R3, A               // save A
        MOVC A,@A+DPTR          // fetch the value in look-up table
        CLR C                   // clear carry before subtraction
        MOV R4, A               // find the negative value
        MOV A, #0FFH
        SUBB A, R4
        MOV P2, A               // send to DAC
        MOV A, R3
        INC A                   // next entry in the look-up table
        DJNZ R2, NEXT1
        SJMP REPEAT             // repeat the cycle forever

LOOKUP: DB 128, 150, 171, 191, 209, 225, 238, 247    // look-up table
        DB 252, 255, 253, 247, 238, 225, 209, 191
        DB 171,150
        END
```

Note: The delay can be placed between two samples, but for simplicity, it is not placed in the program.

The snapshot of values sent to Port 2 is shown in Figure 19.9. Note that the output shown in the logic analyzer window is not the output of DAC, but it is showing analog equivalent values sent to Port 2 for performing conversion.

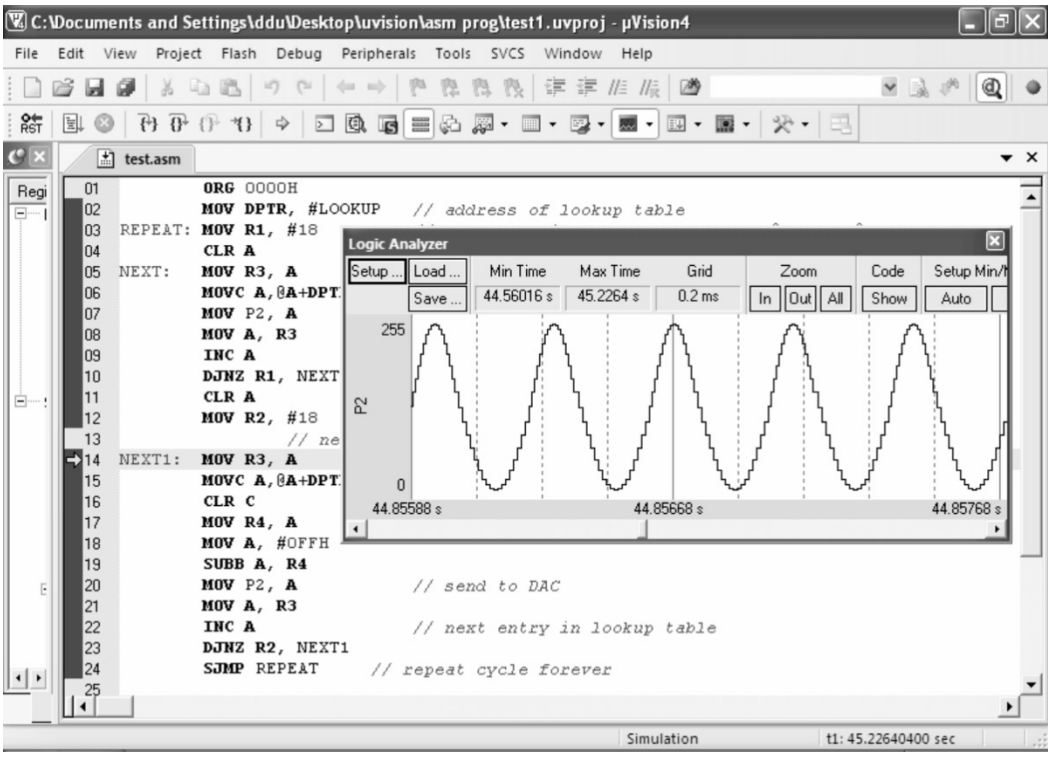


Fig. 19.9 *Sine wave generated using DAC*

Example 19.15

Rewrite the program of Example 19.14 in the C language.

Solution:

The program can be written in the C language as follows:

```
#include<reg51.h>
void DELAY (void);
void main()
        {
        unsigned char i, samples[] = {128, 150, 171, 191, 209, 225, 238, 247, 252, 255, 253, 247,
                                      238, 225, 209, 191, 171, 150};
                                // sine samples
        while(1)                // continuously generate sin wave
              {
              for ( i = 0; i<18; i++)                   // sine wave from 0º to 180º
                        {
                        P2 = samples[i];                // send samples (digital value) to DAC
                        DELAY();                        // delay (can be varied for different frequency)
                        }
              for ( i = 0; i<18; i++)                   // sine wave from 180º to 360º
                        {
                        P2 = 255-samples[i];            // send samples (digital value) to DAC
                        DELAY();                        // delay (can be varied for different frequency)
                        }
              }
        }
```

```
void DELAY(void)                                    // delay routine of 10 ms approx
        {
        unsigned int j ;
        for(j = 0; j<10000; j++);
        }
```

More values can be calculated in the look-up table for a more accurate sine wave. A more compact look-up table may be made because the values of 0° to 90° are repeated in all the four quadrants with change in the sign in different quadrants. But it will require a complex and bigger program.

Discussion Question What is the frequency of the sine wave generated by the program of Example 19.14? How can this frequency be varied?

Answer The frequency of the sine wave depends on the number of samples S in one cycle and time period T between the two consecutive samples (sampling period, or rate at which these samples are given to DAC). The time period of one cycle of sine wave is $S \times T$; therefore, the frequency will be $1/(S \times T)$.

The frequency of sine wave can be changed by varying S or T, usually S is kept constant, and therefore, the frequency can be changed by changing the rate at which the samples (look- up table values) are given to DAC. The maximum frequency is limited by the clock frequency of the microcontroller.

19.2.4 DAC 0808 Chip

It is an 8-bit DAC chip based on R/2R method of conversion. The output is compatible with TTL and CMOS logic. It provides 256 (2^8) discrete current levels, i.e. the output of DAC 0808 is current and, therefore, it is necessary to convert this into voltage. It requires 2 mA reference current for a full-scale input and two power supplies. The pin diagram of DAC0808 is shown in Figure 19.10.

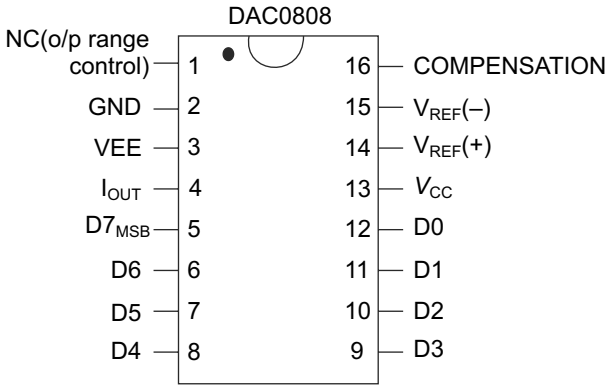


Fig. 19.10 *Pin diagram of DAC 0808*

Interfacing Example 19.16

Interface the DAC 0808 with the 89C51 and write a program to generate a sawtooth wave.

Solution:

The interfacing of the DAC 0808 with the 89C51 is shown in Figure 19.11.

The analog output is current I_{OUT} and will be converted to voltage by connecting the op-amp based current to voltage converter to I_{out} pin. The op-amp also prevents any loading between I_{OUT} and the connected load.

The analog output current I_{OUT} is given as,

$$I_{REF} = \frac{V_{REF}}{R}\left(\frac{D7}{2}+\frac{D6}{4}+\frac{D5}{8}+\frac{D4}{16}+\frac{D3}{32}+\frac{D2}{64}+\frac{D1}{128}+\frac{D0}{256}\right) = I_{REF}\left(\frac{D7}{2}+\frac{D6}{4}+\frac{D5}{8}+\frac{D4}{16}+\frac{D3}{32}+\frac{D2}{64}+\frac{D1}{128}+\frac{D0}{256}\right)$$

I_{REF} can be set by changing the voltage and resistor connected with pin 14(V_{REF} (+)). It requires a reference current of 2 mA for a full scale input. It can be set to 2 mA when the voltage and resistor at pin 14 are +5 V and 2.5 kΩ (5 V/2.5 kΩ). When all the inputs D7-D0 are 1 s, then $I_{OUT\ MAX}$ = 1.992 mA and the output voltage of the current to voltage converter (op-amp circuit) is $V_{OUT} = I_{OUT} \times 5$ K, therefore $V_{OUT\ MAX}$ = 9.961 V ≈10 V. Thus, the output voltage range is 0–10 V (0 V when all digital inputs are 0 and 10 V when all the inputs are 1). This output voltage range may be changed by changing the value of R_f (feedback resistor in current to voltage converter), for example, if R_f = 2.5 kΩ, the output voltage range is 0–5 V.

Operation of DAC0808

The operation of DAC 0808 is very simple. It is only required to give digital input at pins D7-D0 of DAC, i.e. an 8-bit digital input is loaded onto input lines D7-D0 of the DAC0808, and the analog output will be available till the digital input is held constant. An 8-bit digital data may vary

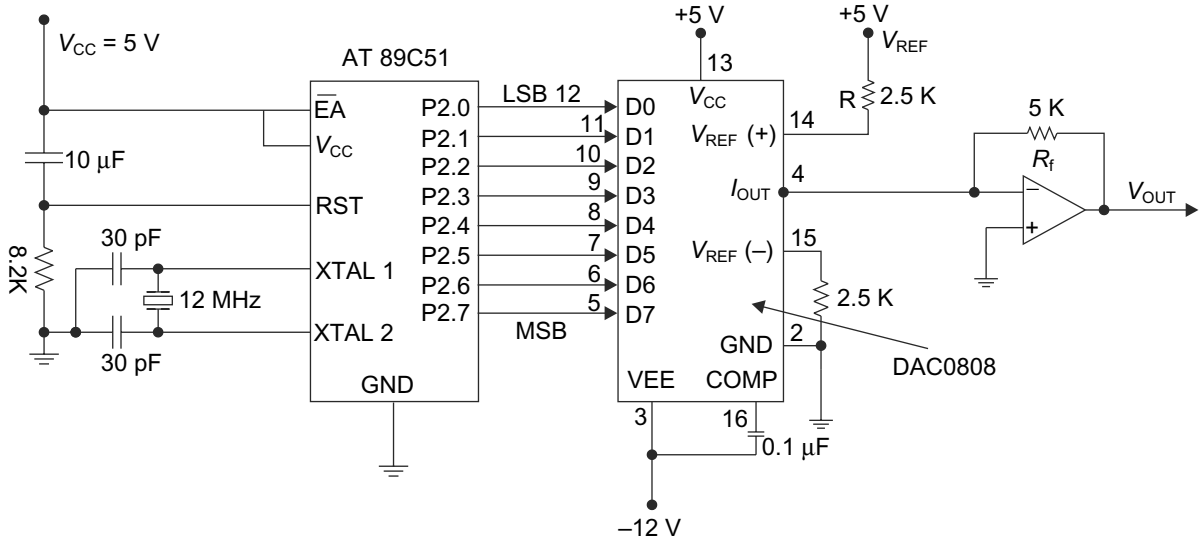


Fig. 19.11 *Interfacing DAC 0808 with 8051*

from 00000000 to 11111111 (0 to 255). The DAC0808 converts this 8-bit data into analog current I_{OUT}. The Op-amp circuit then converts this current into analog voltages. The resulting voltage is equal to V_{OUT}.

The program to generate staircase saw tooth (or ramp) wave using DAC 0808 is given below.

To generate a staircase ramp waveform, we will initialize one register (A) with 0, and send the contents to DAC for conversion, the value of the register is incremented and again the data is given to DAC; this process is repeated continuously so that the value of the register will go from 0 to 255 and because of that, the output will vary from 0 to 10 V.

```
            ORG 0000H
            CLR A
REPEAT:     MOV P2, A           // send data to DAC
            INC A               // increment to get saw tooth wave
            ACALL DELAY         // wait at least for conversion time
            SJMP REPEAT

DELAY:      MOV R6, #10         // delay for 10ms (Xtal = 12 MHz)
THR2:       MOV R7, #250        // the delay of 10ms is only given for demonstration,
                                // it should be at least greater than settling time
THR1:       NOP
            NOP
            DJNZ R7, THR1
            DJNZ R6, THR2
            RET

            END
```

Example 19.17

Rewrite the program of Example 19.16 in the C language.

Solution:

Program can be written in the C language as follows:

```
#include<reg51.h>

void DELAY (void);
```

```
void main()
        {
        unsigned char i;
                while(1)                                // continuously generate sawtooth wave
                {
                for ( i = 0; i<256; i++)
                        {
                        P2 = i;                         // send digital value to DAC
                        DELAY();                        // delay of 10 ms
                        }
                }
        }
void DELAY(void)                                        // delay routine of 10 ms approx
        {
        unsigned int j ;
        for(j = 0; j<10000; j++);
        }
```

Discussion Question Where are these staircase ramp waves useful?

Answer They are usually used for curve tracers where this ramp voltage is used to deflect electron beam on the x–axis at a constant rate.

Example 19.18

For the circuit of Figure 19.11, what will be the full-scale output (or output voltage range) if R_f = 2.0 kΩ?

Solution:

We know I_{REF} = 2 mA, and for full scale output, all the digital inputs must be 1.

$\therefore$ $I_{OUT} = I_{OUT\ MAX}$ = 1.992 mA

$V_{OUT} = I_{OUT}$ x 2 K, therefore $V_{OUT\ MAX}$ = 3.984 $V \approx$ 4 V, i.e. output range is 0-4 V.

Example 19.19

What digital input is required to be given to the circuit of Figure 19.11 if we want the analog output as (i) 1 V, (ii) 1.28 V, and (iii) 5 V?

Solution:

For Figure 19.11, V_{REF} = 5 V, R = 2.5 kΩ, R_f = 5 kΩ

When all the inputs D7-D0 are 1 s then $I_{OUT\ MAX}$ = 1.992 mA and the output voltage of current-to-voltage converter (op-amp circuit) is $V_{OUT} = I_{OUT}$ x 5 K,

Therefore $V_{OUT\ MAX}$ = 9.961 $V \approx$10 V.

Therefore, voltage resolution = 9.961/255 = 0.0390627 V.

(i) For output = 1 V, digital input (decimal) = 1/0.0390627 = $25.59_D = 25_D = 00011001_B$

(ii) For output = 1.28 V, digital input (decimal) = 1.28/0.0390627 = $32.767_D = 32_D = 00100000_B$

(iii) For output = 5 V, digital input (decimal) = 5/ 0.0390627 = $127.99_D = 127_D = 01111111_B$

Example 19.20

Make a look-up table similar to Interfacing Example 19.14 to generate sine wave using DAC 0808 that has a full-scale output voltage of 10 V.

Solution:

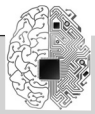
The look-up values and their calculations are shown in Table 19.13.

Table 19.13 *Look-up table values for sine wave (V_F = 10 V)*

Angle (θ)	sin θ	Scaled value V_{OUT} = 5 V + 5 sin θ	Value for DAC V_{OUT} × 25.5
0	0.00	5.00	128
10	0.17	5.87	150
20	0.34	6.71	171
30	0.50	7.50	191
40	0.64	8.21	209
50	0.77	8.83	225
60	0.87	9.33	238
70	0.94	9.70	247
80	0.98	9.92	253
90	1.00	10.00	255
100	0.98	9.92	253
110	0.94	9.70	247
120	0.87	9.33	238
130	0.77	8.83	225
140	0.64	8.21	209
150	0.50	7.50	191
160	0.34	6.71	171
170	0.17	5.87	150
180	0.00	5.00	128

We have taken V_{OUT} = 5 V + 5sin θ to ensure that only positive values are sent to DAC. Here, –1 to +1 values of sine function is mapped to 0 to 10 V because full-scale output voltage is +10 V. Hence, 255 steps/10 V = 25.5 steps per volt. The values for sin 180° to sin 360° can be calculated by subtracting the corresponding value of a look-up table from 255. For example, to find value of sin 190°, we will subtract the corresponding value of sin (190–180)° = sin 10° from 255, (i.e.) sin190° = 255–150 = 105.

More values can be calculated in the look-up table for a more accurate sine wave.



THINK BOX 19.3

Can we operate DAC0808 in bipolar range? If yes, how?

Yes. (Bipolar range means it can generate positive as well as negative output analog output.) Connect resistor of 5 kΩ between V_{REF} and I_{OUT} pin. This will subtract 1 mA (V_{REF} /5 kΩ) current from the current generated by input signal. This will give output range of –5 *V* to +5 V. For example, when all inputs are 1:

$V_{OUT\,MAX}$ = ($I_{OUT\,MAX}$ – (V_{REF} /5 kΩ)) R_F = (1.992 mA – 1 mA) 5 kΩ = 4.96 V

Similarly, when all the inputs are 0, the output would be –5 V.

19.2.5 Applications of DACs

Applications of DACs are listed briefly as below:

- CD players, digital music players, and PC sound cards (because, audio signals are usually stored in a digital form (CD or memory cards) and in order to play it on the speakers, they must be converted into an analog signal
- Analog video monitors and digital video players with analog outputs
- Mobile phones
- Digitally controlled potentiometers
- Data acquisition and process control systems
- Programmable current sources and voltage sources
- VCO tuning

THINK BOX 19.4

Make a list of 8051 based microcontrollers having on-chip DAC.

ADμC812:12 bit- 2 channel, ADμC816:16-bit, ADμC824:12-bit (Analog Devices), P89LPC9103:8-bit 4-channel (NXP), C8051F00x:12 bit 2-channel, C8051F041:12 bit- 2 channel (Silicon Laboratories).

19.3 | TEMPERATURE SENSOR: LM35

The LM35 is an integrated circuit temperature sensor from National Semiconductors. It combines the sensor and electronic circuit into a single IC package and does not require any external calibration. Its output voltage is linearly proportional to temperature (centigrade). It outputs 10 mV for each degree of centigrade temperature.

19.3.1 LM35 Specifications

The specifications of LM35 are

- Temperature range of –55 ºC to +150 ºC
- Linear output scale factor of +10.0 mV/ ºC
- Operating voltage from +4 V to +30 V
- Accuracy of 0.5 ºC (at +25 ºC)
- Less than 80 μA current drain
- Low self-heating (–0.08 ºC) in still air
- Low output impedance of 0.1 Ω for 1 mA load

LM35 is a three-terminal temperature sensor available in TO–46 and TO–92 packages (also available in the other packages). Three terminals are V_s (supply voltage), V_{OUT} (output voltage) and GND (ground). It can be used in two configurations as shown in Figure19.12.

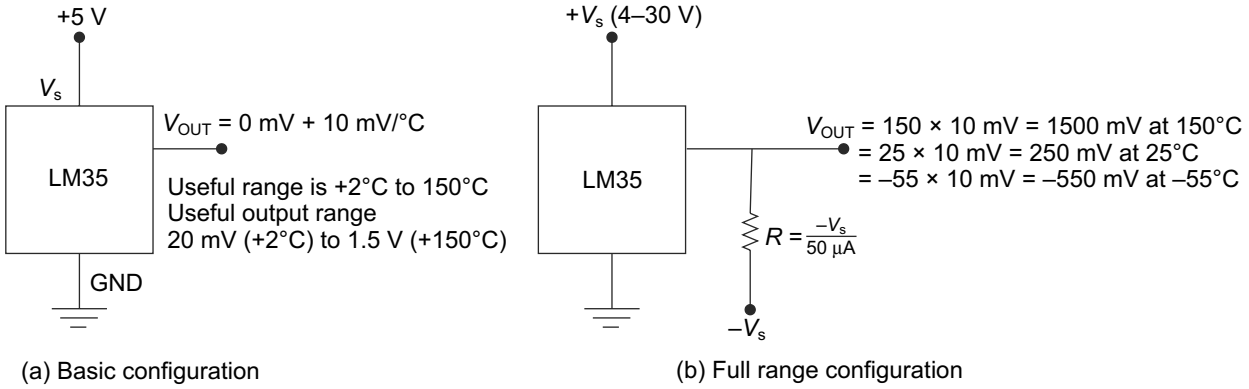


Fig. 19.12 *Configurations of LM35*

The basic configuration is used for temperature range of 2ºC to 150ºC only. The corresponding output voltage is from 20 mV (2 × 10 mV) to 1500 mV (150 × 10 mV). The advantage of this configuration is that it requires only one power supply. The full-range configuration can be used throughout the full range, i.e. –55ºC to +150ºC. The output voltage is directly calibrated for temperature, i.e. the output is adjusted to 0 V for 0ºC, 1.5 V for +150ºC and –550 mV for –55ºC. The only problem with this configuration is that it requires dual power supplies and one external resistor (the resistor value should be $R = -V_s/50\ \mu A$).

The output voltage of LM35 can be amplified to give the voltage range needed for a particular application. For example, for 0–150ºC temperature range, LM35 will generate output voltage from 0 to 150 × 10 mV (1500 mV = 1.5 V). This 0–1.5 V must be converted to 0–5 V for giving it to ADC. Therefore, we need to amplify this signal by (5 – 0/1.5 – 0, it is output range/ input range) = 3.33. The circuit is shown in Figure 19.13.

The other option to connect LM35 directly to ADC (for example, ADC0804) without using an amplifier is by changing the input voltage range using $V_{REF}/2$ input. If $V_{REF}/2$ is connected to 0.75 V, the input range of ADC will become 0 to 1.5 V which will produce a full-scale output. The resolution ADC for this case will be 1.5/256 = 5.8 mV.

The output voltage of LM35 can be directly (or through an amplifier as discussed above) given to ADC, which will give us the digital equivalent of temperature. Interfacing of LM35 is illustrated in the project on temperature monitoring system at the end of this chapter.

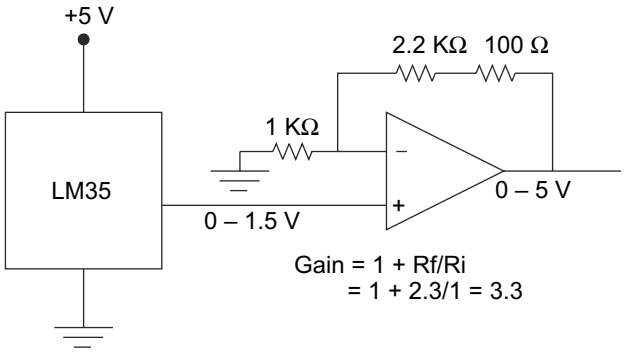


Fig. 19.13 *LM35 connection with Amplifier*

The LM35 series of temperature sensors include LM35A, LM35, LM35C and LM35D. All these sensors are used for different temperature ranges and they have a different accuracy. Exact sensor should be selected as per the application requirement. Refer datasheets of these components for more details.

19.3.2 Common Temperature Sensors

LM34 temperature sensors are similar to LM35 except that they provide output voltage proportional to the Fahrenheit temperature. Some other temperature sensors along with their parameters are listed in Table 19.14.

Table 19.14 *List of common temperature sensors*

Temp. sensor	Accuracy (°C)	Temperature range (°C)	Supply voltage	Package
AD590	0.5	–55 to +150	+4 V to +30 V	FP-2, SOIC-8, TO-52
ADT7310	0.05	–55 to +10	+2.7 V to +5.5 V	SOIC-8
TMP35	1	+10 to +125	+2.7 V to +5.5 V	SOIC-8, SOT23-5, TO-92
TMP36	1	–40 to +125	+2.7 V to +5.5 V	SOIC-8, SOT23-5, TO-92
TMP37	2	+5 to +100	+2.7 V to +5.5 V	SOIC-8, SOT23-5, TO-92
MCP9501/02/03	1	–40 to +125	+ 2.7 V to 5.5 V	5/SOT-23
TC622	1	–40 to +125	+4.5 V to +18 V	5/TO-220 8/PDIP 8/SOIC
LM84	1	0 to 125	+3 V to +3.6 V	16SSOP

19.3.3 Applications of Temperature Sensors

The common applications of temperature sensors are listed below:

- Personal computers, servers and other PC peripherals (CPU thermal management)
- General-purpose temperature monitoring, entertainment systems, office equipment
- Temperature monitor for power supply, power systems, battery management
- Air conditioning system, Industrial process control and environmental control systems
- Thermal protection and fire alarms

THINK BOX 19.5

Make a list of 8051 based microcontrollers having on-chip temperature sensors.

ADμC816, ADμC824, ADμC836 (Analog Devices), C8051F00x:12 bit–2 channel, C8051F000 to F063, C8051F1xx, Si100x (Silicon Laboratories), CC2430F128, CC2430F32, CC2431 (Texas Instruments).

19.4 INFRARED (IR) SENSORS

Infrared (IR) light (radiation) has a wavelength longer than that of the visible light, measured from the edge of visible red light at 0.7 micrometres up to 300 micrometres. Since IR radiation has a frequency below our eye sensitivity, it cannot be seen. Infrared light can be easily generated and detected; moreover, it does not suffer from electromagnetic interference. Therefore, it is widely used in short-distance communication and control circuits, especially in a variety of remote-controlled systems like TVs, VCRs, home and car audio systems and line-follower robots.

There are two types of IR sensors. One provides analog output; they are exactly similar to photodiodes and phototransistors except for the frequency for which they are sensitive. They produce very small output in terms of μA or mA when IR light falls on them; thus, additional amplifier circuits are required to make them useful for controlling applications. The other type of IR sensors have built-in circuits which provides a binary output, which is usually compatible with microcontroller voltage levels (TTL or CMOS) and, therefore, they can be directly connected with the microcontrollers. The second type of IR sensors are discussed briefly in this section. TSOP17xx series of IR sensors, usually referred as IR receivers, are widely used in remote-controlled applications.

19.4.1 TSOP 17xx IR Receivers

The TSOP 17xx series of IR receivers are capable of receiving only pulsed IR light (pulse code modulated) of carrier frequency F_O (from 30 kHz to 56 kHz, depending upon the member of a series) and cannot receive other frequencies. It generates an original modulating signal at the output that were sent by IR transmitter. It has three pins, namely 1-GND, 2-Supply, 3-Output. It operates at a supply voltage from –0.3 V to + 6 V and draws a current of 3 mA. The block diagram of TSOP17xx receivers is shown in Figure19.14.

The front end of this module has a PIN photodiode and the input signal from the remote transmitter is passed into an Automatic Gain Control (AGC) stage from which the signal passes into a bandpass filter and finally into a demodulator. The demodulated output drives an output *NPN* transistor. Note that it provides active low output, i.e. when PCM light from the transmitter falls on the receiver, low logic is generated at the output and in absence of any signal from the transmitter, the output is at high logic.

The features of TSOP 17xx are

- Photodetector and preamplifier and PCM filter are included in same package
- TTL and CMOS compatible output
- Immunity against ambient light and low power consumption
- Data reception rate up to 2400 bits per second.
- Supports all major transmission codes like RC5, RC6, R2000, NEC, Toshiba Micom format, Sharp Code, and Sony Format (SIRCS)

The members of the 17xx series along with their carrier frequency (frequency they can sense) are listed in Table 19.15.

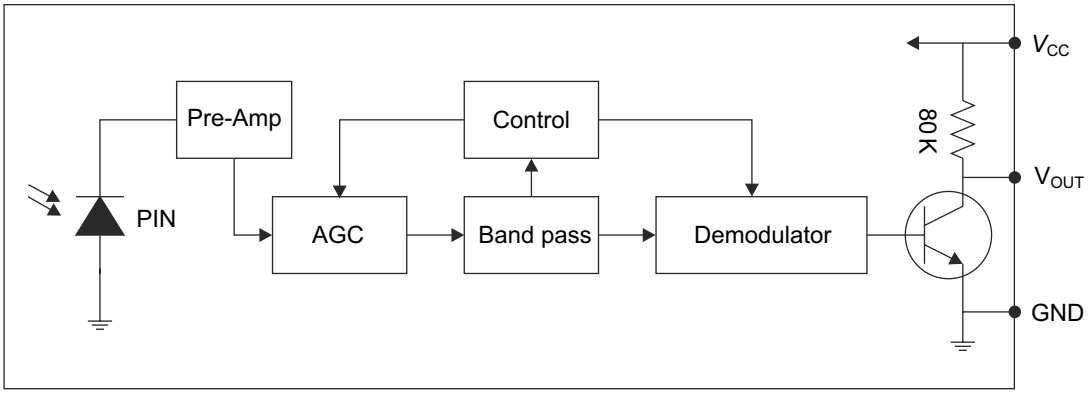


Fig. 19.14 *Block diagram of TSOP 17xx IR receivers*

Table 19.15 *TSOP 17xx members*

Member	Carrier frequency (F_O)
TSOP 1730	30 KHz
TSOP 1733	33 KHz
TSOP 1736	36 KHz
TSOP 1737	36.7 KHz
TSOP 1738	38 KHz
TSOP 1740	40 KHz
TSOP 1756	56 KHz

Note that all the receivers require that the transmitter should transmit the PCM signal at corresponding carrier frequency only, for example, for the successful reception using TSOP 1736; the transmitter must transmit the signal with a carrier frequency of 36 KHz. Moreover, the signals from tungsten bulbs, sunlight, fluorescent lamps will cause disturbances in reception; therefore, these IR receivers must be placed suitably to avoid these signals if possible.

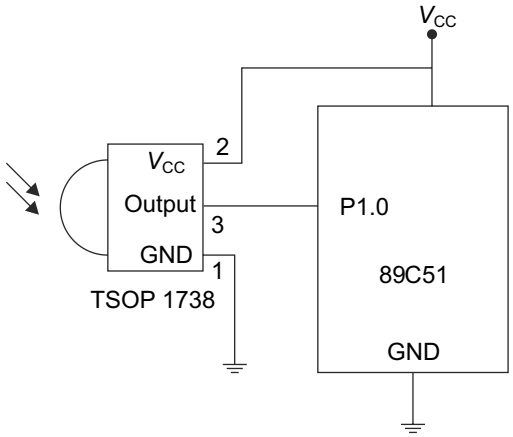


Fig. 19.15 *Connection of TSOP 1738 IR Receiver with 89C51*

19.4.2 Interfacing of TSOP 17xx with the 8051

Since output of TSOP 17xx IR receivers are directly TTL compatible, it can be directly connected to any port pin of the microcontroller. One typical connection with the microcontroller is shown in Figure 19.15. When no IR signal is falling on the receiver, its output is high, when the IR signal falls on the receiver, the output will be low, thus the output signal from the IR receiver can be monitored (using either polling or interrupt method) by a microcontroller to take suitable action based on the received commands or data. The microcontroller can be programmed to receive any of the transmission codes and take a suitable action based on the application requirement. Please refer the details of code format used in an application for bit timings and code format to program the microcontroller to receive and decode the codes.

19.4.3 Applications of IR Sensors

Few common applications of the IR sensors are listed below:

- Entertainment equipment control (remote controls for TVs, CD players, audio systems)
- Industrial process control/inspection and temperature monitoring
- Security systems (to detect warm body intrusion)
- The Infrared thermometer (noncontact type)
- Medical and military applications for imaging

PROJECT: TEMPERATURE MONITORING SYSTEM

Problem Statement

Design a temperature monitoring system using the 89C51 microcontroller which measures and displays the temperature on the LCD. The temperature range to be measured is from 10°C to 150°C. A buzzer should sound if the temperature goes above 100°C. Discuss the steps to develop the program and write a program to perform the given task.

Solution

We will use LM35 as a temperature sensor. It provides output voltage in direct proportion to the temperature. LM35 output voltage increases by 10 mV for each °C increase in its temperature. We will use LM35 in basic configuration shown in Figure 19.12.

This configuration is used for a temperature range from 2°C to 150°C only. The corresponding output voltage is from 20 mV (2 × 10 mV) to 1500 mV (150 × 10 mV). Our temperature range is within this range; therefore, this configuration is used. For analog-to-digital conversion, the ADC 0804 is used. Since LM35 gives 10 mV increase in the output voltage per degree centigrade, we should configure resolution of ADC0804 to be 10 mV. This can be done by connecting $V_{ref}/2$ of the ADC0804 to 1.28 V; this will lead to input voltage range of 0 to 2.56 for full-scale output. For every change of 10 mV at the input of the ADC, the digital output will be incremented by 1. Therefore, for every change of 1ºC temperature, output of ADC will change by 1. The different values of output of ADC corresponding to temperatures are shown in Table 19.16.

Table 19.16 *V_{out} of ADC0804 for different temperatures*

Temperature (ºC)	V_{in} for ADC (mV)	V_{out} (binary)
10	100	0000 1010
20	200	0001 0100
50	500	0011 0010
100	1000	0110 0100

The interfacing circuit for the given problem is shown in Figure 19.16. It includes the interfacing of LCD, ADC0804, LM35 and a buzzer with the 89C51. Note that the power-on reset circuit is not shown for simplicity.

The output of LM35 is given to V_{IN} (+) input of ADC. V_{IN} (–) is grounded; therefore, the effective analog input is equal to voltage at V_{IN} (+) pin. Data output pins D_7–D_0 of ADC are connected to Port 1 and data pins D_7–D_0 of LCD are connected

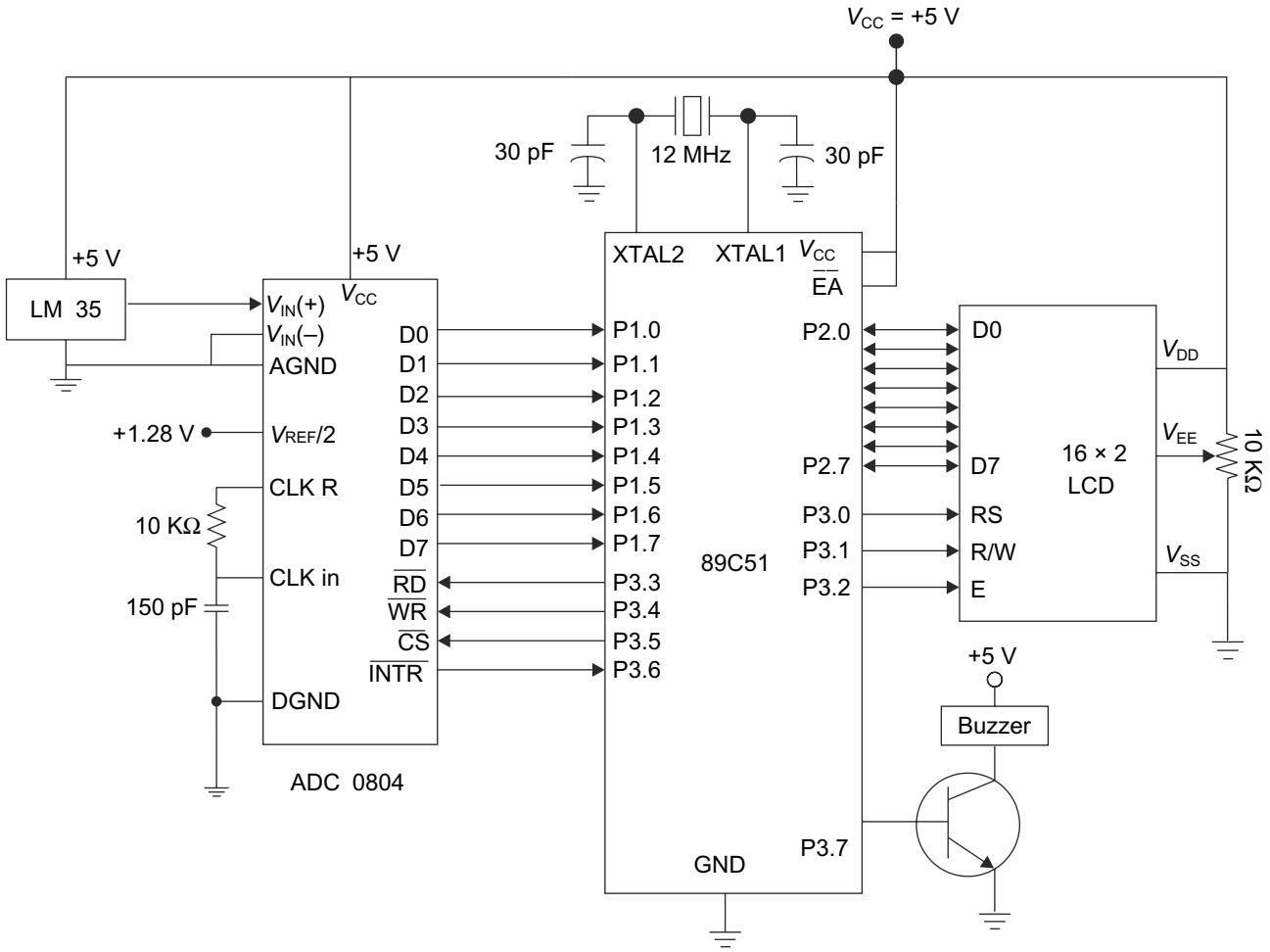


Fig. 19.16 *Temperature monitoring system*

to Port 2. The pins of Port 3 are used as control signals of ADC and LCD. P3.7 pin is connected to the buzzer. V_{REF} /2 of ADC chip is connected to 1.28 V to get the resolution of 10 mV.

Steps to develop a program for temperature monitoring systems are

- Configure port connected with D7-D0 of ADC as inputs.
- Initialize LCD for 8-bit mode and 5x7 character size.
- Start analog to digital conversion by making $\overline{WR}$ = 0.
- Wait for the end of conversion by monitoring $\overline{INTR}$ pin.
- Read digital output from ADC by reading Port 1.
- The output of ADC is directly calibrated to give temperature (in binary).
- Compare temperature with 100 (64H).
  If temperature is greater than 100, make buzzer ON by making P3.7 = 1.
  Otherwise, keep the buzzer as OFF.
- Convert temperature in binary into ASCII.
- Display ASCII value of temperature on LCD.
- Start the analog to digital conversion again and repeat the process continuously.

Main Program

```
          CLR P3.7              // make buzzer OFF
          MOV P1, #0FFH         // configure Port 1 as an input port
REPEAT:   LCALL AD_CONVERSION   // call A to D conversion subroutine
          LCALL COMPARISON      // if temperature > 100 °C, make the buzzer ON
                                // otherwise, keep the buzzer OFF
          ACALL BIN_ASCII       // convert the temperature in binary to ASCII
          ACALL DISP_LCD        // display the temperature on LCD
          SJMP REPEAT           // repeat the process of monitoring continuously
                                // Analog-to-digital conversion subroutine
AD_CONVERSION:
          SETB P3.6             // configure P3.6 as an input to monitor the end of
                                // conversion (INTR pin)
          CLR P3.5              // make CS low for selecting chip
          CLR P3.4              // make WR low
          SETB P3.4             // WR = 1, low-to-high pulse for starting ADC
HERE:     JB P3.6, HERE         // wait until the end of conversion
          CLR P3.3              // make RD low to read the conversion result
          MOV A, P1             // read the data from ADC
          MOV 40H, A            // store the result at address 40H
          SETB P3.3             // make RD high before taking the next sample
          RET                   // return to the main program
                                // Temperature comparison routine
COMPARISON:
          CLR C
          MOV A, 40H            // get temperature from address 40H
          SUBB A, #100          // compare the temperature with 100 °C
          JNC B_ON              // if temp > 100, make the buzzer ON
          CLR P3.7              // otherwise, make the buzzer OFF
          SJMP OVER
B_ON:     SETB P3.7
OVER:     RET                   // return to the main program
                                // Binary to ASCII conversion subroutine

BIN_ASCII:
          MOV A, 40H            // get the temperature in binary from address 40H
```

```
          MOV B, #64H          // divisor (100)
          DIV AB               // A has 100's digit (unpacked BCD digit)
          ORL A, #30H          // convert BCD digit into ASCII
          MOV 50H, A           // store 100's ASCII digit at address 50H
          MOV A, B             // copy remainder into A for next division
          MOV B, #0AH          // next divisor (10)
          DIV AB               // A has 10's digit (unpacked BCD digit)
          ORL A, #30H          // convert BCD digit into ASCII
          MOV 51H, A           // store 10's ASCII digit at address 51H
          MOV A, B             // B has 1's digit (unpacked BCD digit)
          ORL A, #30H          // convert BCD digit into ASCII
          MOV 52H, A           // store 1's ASCII digit at address 52H
          RET                  // return to the main program
                               // LCD display subroutine
DISP_LCD:
          MOV A, #38H          // initialize LCD, 8-bit interface, 5X7 //dots/character
          LCALL COMMAND        // send command to LCD
          MOV A, #0FH          // display on, cursor on with blinking
          LCALL COMMAND        // send command to LCD
          MOV A, #06           // shift cursor right
          LCALL COMMAND        // send command to LCD
          MOV A, #01H          // clear LCD screen and memory
          LCALL COMMAND        // send command to LCD
          MOV A, #86H          // set cursor at line 1, 6th position
          LCALL COMMAND        // send command to LCD
          MOV R2, #03H         // counter for three digits
          MOV R0, #50H         // ASCII digits are stored 50H onwards
          MOV A,@R0            // read ASCII digit
NEXT:     LCALL DISPLAY        // send data to LCD for display
          INC R0               // point to next ASCII digit
          DJNZ R2, NEXT
          MOV A, # "C"         // display C for centigrade
          LCALL DISPLAY
          RET

                               // Command write subroutine
COMMAND:  ACALL READY          // check busy flag
          MOV P2, A            // place the command on P1
          CLR P3.0             // RS = 0 for command
          CLR P3.1             // R/W = 0 for write operation
          SETB P3.2            // E = 1 for high pulse
          LCALL WAIT           // wait for some time
          CLR P3.2             // E = 0 for H-to-L pulse
          RET

                               // data write subroutine
DISPLAY:  ACALL READY          // check busy flag
          MOV P2, A            // send data to Port 1
          SETB P3.0            // RS = 1 for data
          CLR P3.1             // R/W = 0 for write operation
          SETB P3.2            // E = 1 for high pulse
          LCALL WAIT           // wait for some time
          CLR P3.2             // E = 0 for H-to-L pulse
          RET
```

```
READY:    SETB P2.7              // configure P2.7 as input
          CLR P3.0               // RS = 0 for command
          SETB P1.1              // RW = 1 for reading
WAIT:     CLR P3.2               // E = 1 for high pulse (see the following loop)
          ACALL DEALY
          SETB P3.2              // high-to-low pulse on E
          JNB P2.7, WAIT         // wait until the busy flag is 0
          RET
```

Suggested Modification

Modify the system to make it 8-channel data acquisition system. The system should monitor 8 different analog signals. During the normal operation, the system should send the data (digital equivalent) from all the sensors (one by one) to the PC through serial port. The program in the PC should store samples for each sensor in different arrays. When any of the analog input exceeds a predefined limit, the system will turn the buzzer ON and displays the name of the source which has exceeded the specified limit.

[**Hint:** Use an 8 channel ADC like ADC0808/09 for monitoring 8 analog inputs. Refer Figure 15.5 for details of how to interface the 8051 to COM port (RS 232 side) of the PC]

PROJECT: FUNCTION GENERATOR

Problem Statement

Design a simple function generator using the 89C51 and DAC, which can generate a sine wave, sawtooth (ramp wave) or square wave. The function generator should have a frequency knob and waveform output selection switches.

Solution

The frequency of output signal (any of the three different waves) should vary as we rotate the frequency control knob, i.e. when the knob is at fully anti-clockwise position, the frequency should be minimum, and as we rotate the knob in the clockwise direction, the frequency should increase until we reach the extreme position. This type of frequency control is achieved using potentiometer when voltage V_{max} is connected across it. The potentiometer will give us 0 V to V_{MAX} volts (0 V–5 V for our design) when we rotate the knob.

Three input switches connected with the microcontroller port pins selects the type of wave that is to be generated. The block diagram of the desired function generator is shown in Figure 19.17.

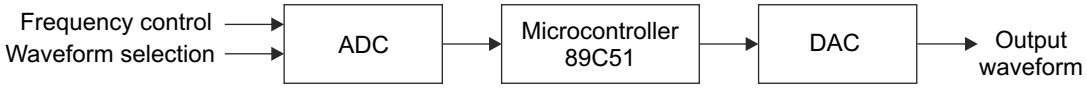


Fig. 19.17 *Block diagram of a function generator*

The signal from the potentiometer (analog signal) should be given to ADC. The output of ADC can be used to control the frequency of the output waveform. The output frequency should change from minimum to maximum when the frequency control knob is varied. The ADC0804 is used for this operation. The DAC is required to generate different waves. The complete diagram of the frequency generator is shown in Figure 19.18. Note that the power-on reset circuit is not shown for the simplicity.

The potentiometer of 10 KΩ is used as a frequency control knob. Supply voltage (+5 V) is connected across it; therefore, it can give 0 to 5 V as input to ADC depending upon the position of knob. $V_{REF}/2$ of ADC is kept open to select analog input range as 0 to 5 V. Port 1 pins are connected with the data pins of ADC and Port 3 pins (P3.4 to P3.6) are used as control signals of the ADC as shown in the Figure. The pin P3.3 is used as an external interrupt 1 input. Port pin P3.0, P3.1 and P3.2 are connected with the switches to select sine wave, sawtooth wave and square wave generation respectively. Port 2 is connected with the DAC data lines. The output of DAC is given from current to the voltage converter which will output the desired wave.

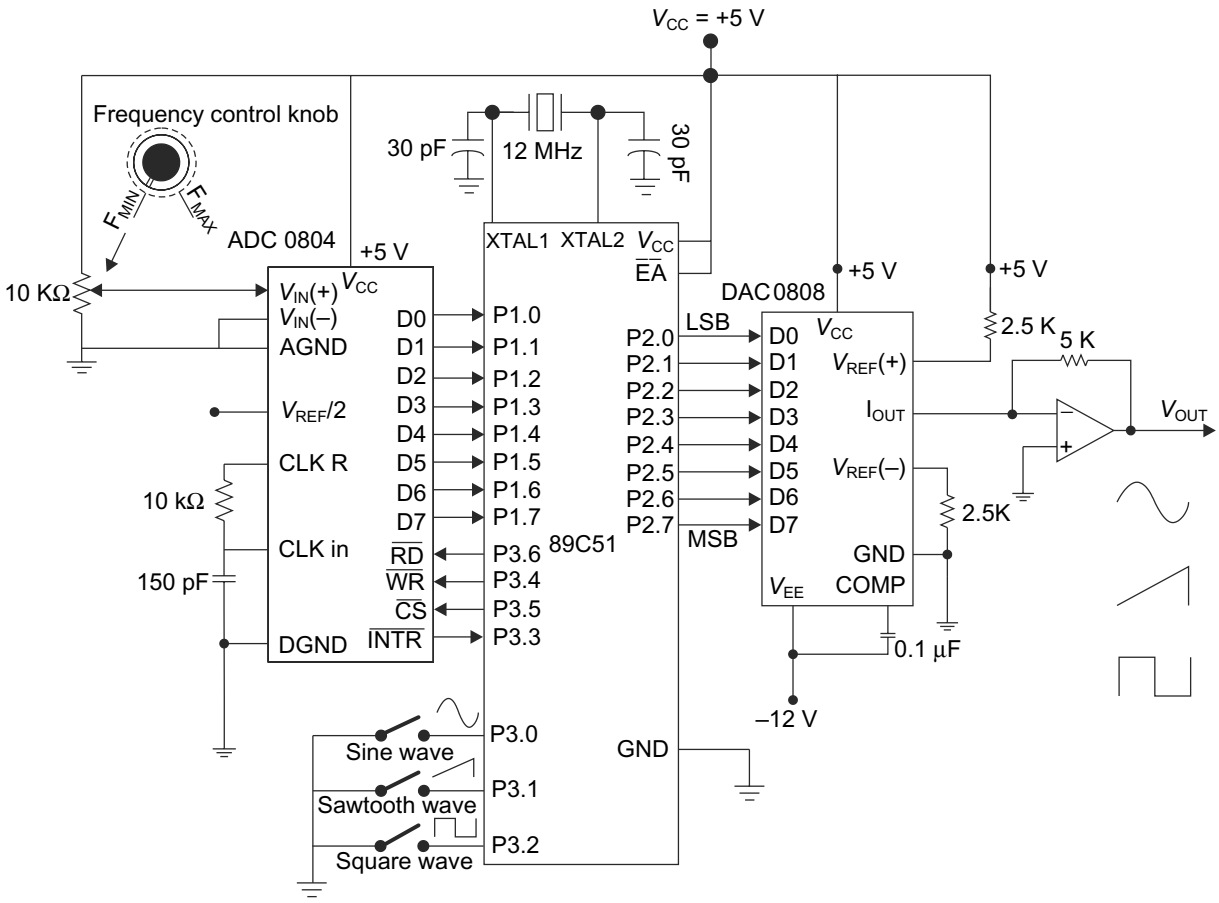


Fig. 19.18 *Simple function generator using 89C51*

Waveform-Generation Technique

First, the status of three switches is monitored to select the output waveform. The waveforms are generated using look-up table technique. In this technique, the sample of desired wave shape is stored in a look-up table (or in an array for high-level language). These samples are fed into DAC and the desired waveform is generated at the output waveform.

Frequency Control

The frequency of the desired waveform can be changed by changing the rate at which the samples are given to the DAC. Now, for our function generator, the frequency is controlled by a variable knob as discussed above. The potentiometer of 10 KΩ is used as the frequency-control knob. Supply voltage (+5 V) is connected across it; therefore, it will give 0 to 5 V as input to ADC depending upon the position of the knob. The output of the ADC is used to control the time delay between two consecutive samples that are given to the DAC; this will control the time required to give all samples of one cycle to the DAC and finally the frequency of output waveform will change according to this time delay.

The value of the ADC output is passed as a parameter to the delay function. The output of the ADC is first subtracted first from 255 and then the result is passed to delay function because when ADC output is 00, we require minimum frequency of the signal. Therefore, 00 is subtracted from 255 and the result (255 – 00 = 255) is passed as a parameter to the delay function. This maximum value will generate the maximum delay between the two consecutive samples and, therefore, we will generate an output waveform with a minimum frequency. As ADC output is increased, the value passed to delay function will decrease and the frequency will increase.

Program Development

Three arrays are defined to store sample values of different waves. Based on the position of input (waveform select) switches, one array is passed to a function which continuously reads elements of the array (one by one) and gives it to DAC to generate the corresponding waveform. The time delay between accessing the two samples (hence frequency) is

controlled by DELAY function. The position of the frequency knob is continuously monitored by ADC, and an equivalent digital output is passed as a parameter to the DELAY function. End of the conversion from ADC is monitored through external interrupt 1; therefore, the microcontroller can generate waveforms continuously without wasting time in polling end of the conversion pin.

The steps to write the program are the following:

- Define three arrays, each containing samples of sine, saw tooth and square wave.
- Configure P1 and pin P3.3 as an input.
- Enable external interrupt 1 and configure it as an edge-triggered interrupt.
- Monitor position of the frequency knob by starting ADC conversion.
- Read status of waveform select switches.
- Based on the status of these switches, pass the corresponding array to function which generates a waveform by giving array elements to DAC with a delay between two successive <something is missing>.

The interrupt will be generated when A to D conversion is completed; the ISR should perform the following operations:

- Read output of ADC and manipulate it, (255- output).
- Place the manipulated value in global variable AD_result.
- Return back to the main program.

The DELAY function will generate a delay as per the value of the variable AD_result (output of ADC).

```
#include<reg51.h>
sbit SIN = P3^0;              // switch connected to P3.0 is used to generate sin wave
sbit SAWTOOTH = P3^1;         // switch connected to P3.1 is used to generate sawtooth wave
sbit SQUARE = P3^2;           // switch connected to P3.2 is used to generate square wave
sbit INTR = P3^3;             // end of conversion signal is connected as ext interrupt 1 input
sbit CS = P3^5;               // chip select signal of ADC
sbit WR1 = P3^4;              // start the conversion signal of ADC
sbit RD1 = P3^6;              // read the signal of ADC

void wave_generate (char []);// function to generate the desired wave
void DELAY (unsigned char);  // delay subroutine
unsigned char AD_result;     // variable to store ADC output
void main()
{
unsigned char WAVE_SELECT;
unsigned char sin_samples[]   = {128, 150, 171, 191, 209, 225, 238, 247, 252, 255, 253, 247,
                                 238, 225, 209, 191, 171, 150, 127, 105, 84, 64, 46, 30, 17,
                                 8, 2, 0, 2, 8, 17, 30, 46, 64, 84, 105};
                                // samples of sine wave
unsigned char sawtooth_samples[] = { 0, 7,1, 4, 21, 28, 35, 42, 50, 57, 64, 71, 78, 85, 92,
                                 99, 106, 113, 120, 127, 135, 142, 149, 156, 163, 170, 177,
                                 184, 191, 198, 205, 212, 220, 227, 234, 241, 248};
                                // samples of saw tooth wave
unsigned char square_samples[] = {255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
                                 255, 255, 255, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                 0, 0, 0, 0, 0, 0, 0, 0};
                                // samples of square wave.
     INTR = 1;                  // configure P3.3 as external interrupt 1 input
     CS = 0;                    // make CS low for selecting ADC chip
     IE = 0x84;                 // enable external interrupt 1
     TCON | = 0x04;             // configure external interrupt 1 as edge-triggered
     P1 = 0xFF;                 // configure P0 as an input port to get ADC result
```

```
while (1)
     {
     WR1 = 0;                  // make WR low
     WR1 = 1;                  // WR = 1, low-to-high pulse for starting ADC
     WAVE_SELECT = P3;         // read status of wave select switches
     WAVE_SELECT & = 0x07;     // mask upper 5 bits of P3 as switches are connected only
                               // with lower three pins

     switch(WAVE_SELECT)       // perform the task as per status of P1.0 and P1.1
        {
        case(6):               // if status of switches = 110, generate sine wave
               {
               wave_generate (sin_samples);
               break;
               }
        case(5):               // if status of switches = 101, generate sawtooth wave
               {
               wave_generate (sawtooth_samples);
               break;
               }
        case(4):               // if status of switches = 011, generate square wave
             {
             wave_generate (square_samples);
             break;
             }
        }
     }
}

void wave_generate (char samples[])

     {
     unsigned char i;
     for ( i = 0; i<36; i++)   // each wave has 36 samples
           {
           P2 = samples[i];    // send samples (digital value) to DAC
           DELAY(AD_result);   // time-delay depends on ADC output or Frequency
                               // knob position
           }
     }

void DELAY(unsigned char k)    // delay routine
     {
     unsigned int j ;
     for(j = 0; j<50 * k; j++);
     }
void ext1 (void) interrupt 2   // external interrupt 1 ISR
{
     RD1 = 0;                  // make RD low to read the conversion result
     AD_result = P1;           // read data from ADC
     AD_result = 255 -AD_result;
     RD1 = 1;                  // make RD high before taking the next sample
}
```

Simulation Output

The snapshot of the output when Pin 3.0 = 0 (generate sine wave) is given in Figure 19.19. Note that the output in the logic analyzer is not the output of DAC but it is the analog equivalent of the digital value sent to Port 2.

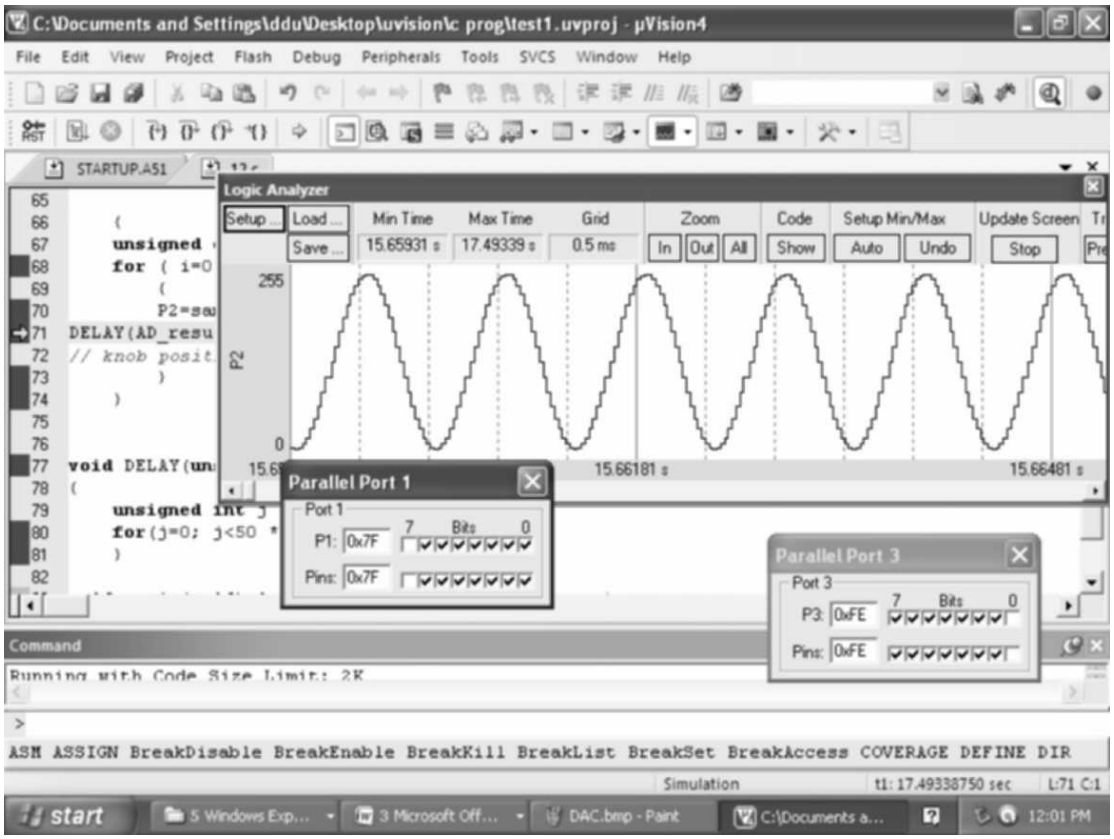


Fig. 19.19 *Output of function generator*

Suggested Modifications

- Add an amplitude control knob in the system.
- Generate one more waveform, triangular wave, and add a fourth input switch.
- Increase the number of samples for all waveforms to get a better output.
- Write a more efficient program with respect to memory usage. [**Hint:** use few samples and get others from them, for example, full cycle of sine wave can be generated from samples of only quarter cycle.]

Exercise Find the minimum and maximum frequency of the waveform generated by the function generator and modify the program to change the frequency range.

POINTS TO REMEMBER

- Today, a majority of signal processing is done in the digital systems because of their increasingly efficient, reliable and economical operation.
- The number of bits in digital output determines resolution of the ADC. The ADCs are specified as 8, 10, 12, 16 or 24-bit ADC.
- The ADC0801/02/03/04/05 chips are common 8-bit ADCs.
- The ADC0808/09 chips are 8-channel ADCs i.e. 8 analog inputs can be monitored using the same chip, but only one input at a time.

- R-2R DACs are more popular and precise because they require only two values of resistors.
- The AD557 is an 8-bit DAC that provides analog in the form of voltage, including output amplifier. It operates using single supply voltage of +5 V. Its output voltage range is 0 to 2.56 V.
- The DAC0808 is an 8-bit DAC chip based on R/2R method of conversion.
- LM35 is a three-terminal temperature sensor which outputs the voltage which is linearly proportional to the temperature (centigrade). It outputs 10 mV for each degree of centigrade temperature.

OBJECTIVE QUESTIONS

1. The ADC0804 has _____ resolution.
   (a) 4-bit (b) 8-bit (c) 10-bit (d) 12-bit
2. The end-of-conversion on the ADC0804 is indicated by,
   (a) WR pin (b) CS pin (c) INTR pin (d) $V_{ref}/2$ pin
3. The start-conversion command is given to _____pin of the ADC0804.
   (a) WR (b) CS (c) INTR (d) $V_{ref}/2$
4. Resolution of ADC is defined as,
   (a) 1/ (2N – 1) (b) 2N–1 (c) 1/ (2^N–1) (d) 2^N–1
5. DAC0808 and AD557 are ___bit DACs.
   (a) 16 (b) 4 (c) 8 (d) 2
6. MAX1112 has __ analog input channels.
   (a) 1 (b) 2 (c) 4 (d) 8
7. The output of DAC 0808 is usually connected to,
   (a) current-to-voltage converter (b) amplifier
   (c) attenuator circuit (d) integrator
8. Which of the following value is sent to 0–10 V DAC to generate 7.5 V?
   (a) 192 (b) 238 (c) 128 (d) 255
9. The number of analog input channels for on-chip ADC of P89LPC768 is,
   (a) 1 (b) 4 (c) 8 (d) 16
10. Which bit ADCON register of P89LPC768 indicates that the A-to-D conversion is complete?
    (a) ENADC (b) ADCI (c) ADCS (d) AADR1
11. Which pin of ADC 0808 indicates that the conversion is complete?
    (a) EOC (b) SC (d) OE (e) ALE
12. Which of the following DAC provides the best resolution?
    (a) 4-bit (b) 8-bit (c) 10-bit (d) 12-bit
13. If input voltage range of an ADC0808 converter is 0 to +4 V then the step size is,
    (a) 19.53 mV (b) 2.5 V (c) 15.68 mV (d) 256 mV
14. For ADC MAX1112, we should apply ___ clock pulses to get a byte of digital data.
    (a) 1 (b) 2 (c) 8 (d) 9
15. Temperature range of LM35 is,
    (a) –55 ºC to +150 ºC (b) –50 ºC to +150 ºC
    (c) –55 ºC to +130 ºC (d) 0 ºC to +150 ºC

Answers to Objective Questions

1. (b) 2. (c) 3. (a) 4. (c) 5. (c) 6. (d) 7. (a)
8. (a) 9. (b) 10. (b) 11. (a) 12. (d) 13. (c) 14. (d)
15. (a)

REVIEW QUESTIONS WITH ANSWERS

1. What is meant by resolution of the ADC?

A. Smallest change in the input signal that is reflected in the digital output or the smallest change in input that can be measured by the ADC.

2. What are the key characteristics of the ADC?

A. Resolution, i.e. the number of bits and conversion time.

3. Which type of ADC has the minimum conversion time?

A. Flash ADC.

4. What is the use of WR pin in ADC0804?

A. It is used to inform the ADC chip to start the conversion process.

5. What are the types of DAC?

A. R-2R and ladder DAC.

6. To get a full-scale output voltage, what input should be given to the DAC?

A. All digital input pins must be at logic level high.

7. How many discrete analog voltage levels are provided by a 12-bit DAC?

2^{12} = 4096 discrete voltage levels.

8. Why it is necessary to separate the digital ground from analog ground in a typical ADC?

A. To isolate the digital and analog circuits because digital grounds are noisier than the analog grounds and because of the switching noise generated in digital chips when they change the states. For digital lines, this is not a problem if it does not cross a logic threshold. But for an analog signal, the noise is added directly to the signal.

9. Give the status of CS and WR in order to start a conversation for the ADC0804.

A. CS = 0, WR = L-to-H.

10. Which pin of the ADC0804 indicates end of conversation?

A. INTR.

11. The output of DAC0808 is in _____ (current/voltage).

A. Current.

EXERCISE

1. Discuss handshaking process between ADC chip and a microcontroller for performing a conversion.
2. Explain with suitable example how an analog input voltage range for ADC can be changed?
3. How can the conversion time of ADC be compromised with its resolution?
4. List the different types of ADC architectures.
5. List the different ADC chips available in the market with their specifications.
6. What is meant by a serial ADC?
7. Why does ADC0804 have separate analog and digital ground?
8. Write a program to generate sawtooth and triangular waves using DAC.
9. List the applications of ADCs and DACs.

20

Interfacing Relays, Opto-Couplers, Stepper and DC Motors

Objectives

- Describe the construction, operation and types of relays, opto-couplers, stepper and DC motors
- List the parameters of relays
- Discuss the common driver circuits for relays, stepper and DC motors
- Discuss the techniques of direction and the speed control of stepper and DC motors
- Interfacing of relays, opto-couplers and stepper motors with the 8051
- Develop the programs to control DC and stepper motors

Key Terms

- Bipolar Stepper Motor
- Common Terminal
- Direction Control
- Driver Circuit
- Electromechanical Relay
- Free Wheeling Diode
- Half/Full Step Sequence
- Isolation
- Normally Closed: NC
- Normally Open: NO
- PM Stepper Motor
- Position Control
- Solid State Relay
- Speed Control
- Stator/Rotor
- Step Angle
- Unipolar Stepper Motor
- Wave Drive

Microcontrollers are mainly used in the control applications where they control the circuits in the external world which may be working at different voltage or power levels. Therefore, microcontrollers must be electrically isolated from external circuits, otherwise higher operating voltages in external circuits will damage the microcontroller. Relays and opto-couplers are commonly used to isolate the circuits operating at different power levels. This chapter describes operation and interfacing of relays, opto-couplers and additionally, stepper and DC motors.

20.1 | RELAYS

A relay is an electrical switch that is controlled by a small current or voltage changes caused by the electrical devices in a system. Relays are used to control (turn on/off) high power loads like motors, heaters and bulbs, etc., using low-power controlling circuits like microcontroller/processor systems. Generally, they are used to electrically isolate the two systems operating with different power levels. Two types of relays are available: Electromechanical Relay (EMR) or Solid-State Relay (SSR).

An electromechanical relay consists of an electromagnet, a metallic switch and a spring. When an electromagnet is energized, the electromagnetic field is generated which opens (or closes) the switch contact, which is normally held in one position by a spring. The metallic switch is used to operate with high power loads. Solid-state relays are completely made from semiconductor material. They employ transistor switch to electronically control the high-power loads. Different types of relays are shown in Figure 20.1.

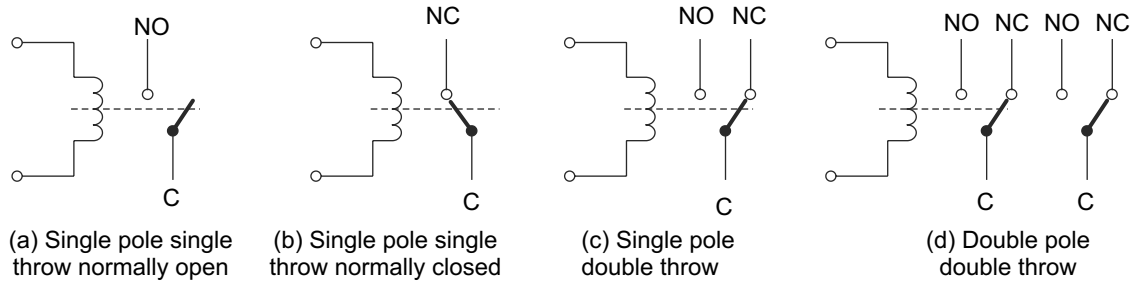


Fig. 20.1 *Different types of electromechanical relay*

Relays have two terminals (leads) at the input side and generally three terminals at the output side. The two terminals at the input side are used to energize the relay coil. The output side terminals are Common (C), Normally Closed (NC) and Normally Open (NO). NC is connected with a common lead when the coil is not excited, and NO is connected with a common lead when the coil is excited. There may be more than one pair of such terminals to provide control over the multiple loads.

20.1.1 Relay Operation

Consider relay shown in Figure 20.1(a), it is a Single-Pole, Single-Throw-Normally Open (SPST-NO) type relay, when the coil is energized (current is flowing when voltage is applied across it; therefore, the coil will behave as an electromagnet, the contact will be closed. Therefore, the common (C) and normally open (NO) terminals are connected. The relay of Figure 20.1(b) is Single-Pole Single-Throw Normally Closed (SPST-NC) and when the coil is energized, C and NC will be disconnected and this open connection will switch off the load connected with this terminal. The relay shown in Figure 20.1(c) is Single-Pole Double-Throw (SPDT) and when the coil is energized, C will be connected to NO and disconnected from NC. Therefore, any load connected to NC will be switched off and the load connected with NO will be switched on. The relay of Figure 20.1(d) has multiple pairs of terminals which can be used to control multiple loads at the same time and it is referred as Double-Pole Double-Throw (DPDT) type relay.

One of the useful types of the relay is latching relay. It does not require a continuous coil current to maintain its present state. When a short-duration pulse is applied across the relay terminals, the state of the relay will be changed and the relay will remain in the new state indefinitely until another pulse is applied.

20.1.2 Relay Driver Circuits and Interfacing

Based on the type of a relay, the voltage and current required to drive a relay coil will vary from few volts (around 5 V) to few tens of volts (around 30 V) and currents up to 20 mA. The microcontroller pins cannot provide the sufficient currents

to drive the relay coils; therefore, we need to use the driver circuit between a microcontroller pin and a relay. The driver circuit may be based on transistor, open collector buffers, driver ICs such as ULN 2003 (or other member of ULN 2xxx family) or a FET. The type of a driver circuit to be used depends on the type of relay used. These driver circuits and their interfacing with the 8051 are shown in Figure 20.2.

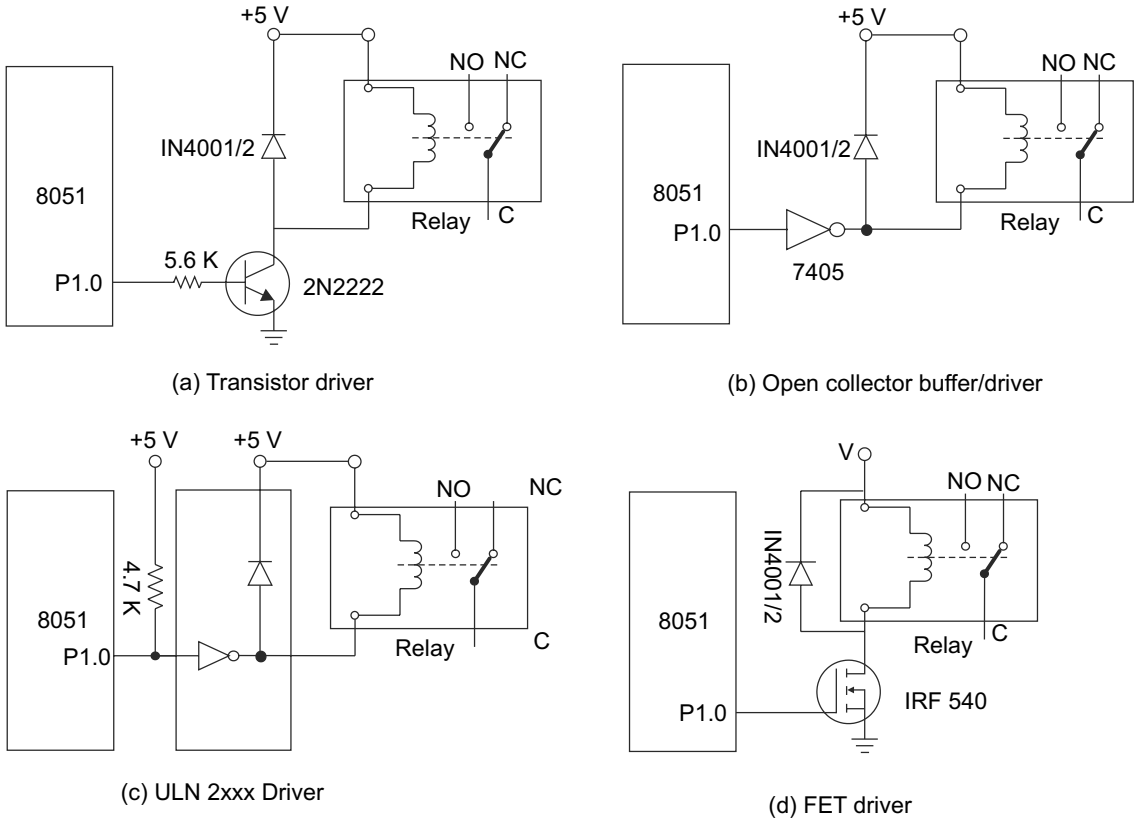


Fig. 20.2 *Relay driver circuits*

Driver Circuit Operation

For the transistor driver of Figure 20.2(a), when the control input (from microcontroller pin P1.0) is high, the transistor will conduct (goes into saturation) and current flow through relay coil (coil is energized) and relay terminal C will be disconnected from NC and connected to NO. The diode (usually referred as a freewheeling diode) is connected in parallel to the relay coil to prevent the coil from high voltage induced by (as a self-induction) a sudden stop of current flowing through the coil. The purpose of this diode is to "cut off" the induced high voltage and to protect the driving circuit of the relay. When the coil is deactivated (when control input changes from 1 to 0), there will be a large *dI/dt* in the coil, which will produce a large back emf of the magnitude from 50 V to 200 V depending upon the relay and driver circuit; this voltage may damage the driver circuit. The freewheeling diode will bypass (suppress) this voltage by providing a short circuit across the coil. (Diodes of the IN400x family have maximum voltage ratings from 50 V for IN4001 to 1000 V for IN4007, appropriate diode should be used). The driver of Figure 20.2 (b) will energize the relay coil when the control input is 1 and sink the current through coil. The driver of Figure 20.2(c) uses driver IC (ULN2xxx series) which has inbuilt freewheeling diodes. These driver chips can provide currents up to 1.5 A (for example, ULN2002 – 0.6 A, ULN2003 – 1.5 A) and can be easily driven from TTL 5 V control signal. The FET drivers [Figure 20.2 (d)] are used for the coils which require high currents (above 500 mA, for example, FET IRF540 can sink currents up to 28 A). Note that the dc motors have coils similar to relays; therefore, these driver circuits may also be used to drive the dc motors.

Note that I_{OL} (sink current) of the chosen driver circuit must be greater than the current sufficient to drive a relay.

Interfacing of the relay with the 8051 is illustrated in Figure 20.3; in this figure the relay is used to control the bulb (instead of bulb, any other load may be connected).

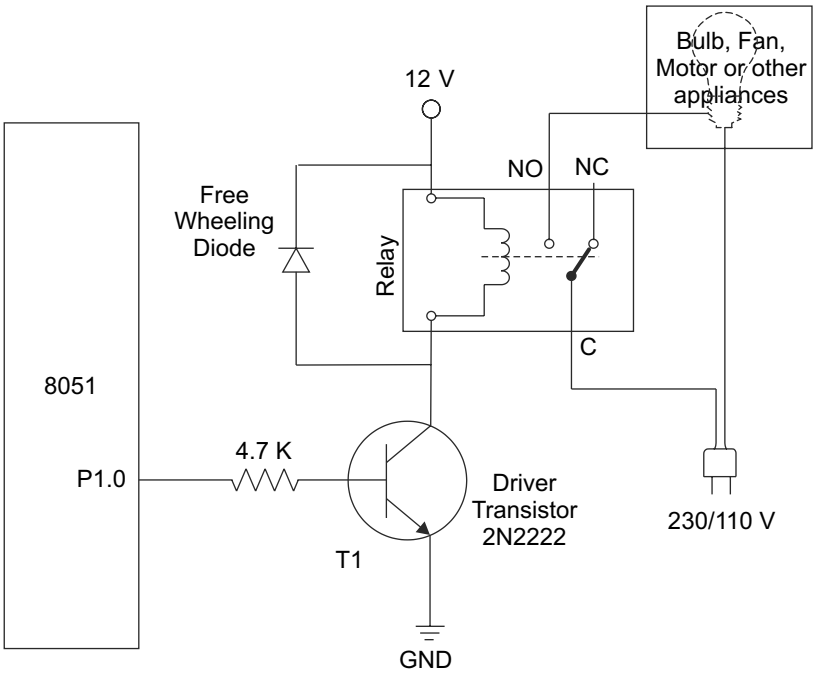


Fig. 20.3 *Relay driving a bulb*

As discussed in the above section, when the control input (P1.0) is high, the transistor will conduct and current will flow through the relay coil (relay is activated), this will change the position of the common (C) terminal to NO terminal, which will complete high-voltage circuit and 230 V (or 110 V) is applied across the bulb, and bulb will glow. A freewheeling diode, as mentioned earlier, will protect the driver transistor from damage. Note that any other load like motor, fan, heater, etc., can be used in the place of a bulb in the circuit of Figure 20.3.

Advantages of using a Relay

The advantages of relays are that any short circuit or fault on high-power side will have no effect on the controlling side; hence, the controlling circuits are protected from the damage because of isolation between them. Furthermore, the load circuit can be ac or dc.

Drawbacks of Electromechanical Relays

Drawbacks of electromechanical relays are a low switching frequency due to mechanical components, and a lower life because of wear of the mechanical parts.

The advantages of SSR are higher switching frequency and longer life because no mechanical components are involved, higher reliability, quieter and no arcing.

20.1.3 Parameters of Relays

Input Coil Side (Low Power, Controlling Side)

- **Pick-up voltage:** It is the minimum coil voltage above which the relay activation is assured.
- **Dropout voltage:** It is the maximum coil voltage below which the relay deactivation is assured
- Coil resistance
- AC or DC excitation

Output Switch Side (High Power, Load Side)

- Maximum current (contact current), voltage (contact voltage), power
- ON resistance, OFF resistance
- Turn-on time, turn-off time

The list of relays along with their parameters is given in Table 20.1.

Table 20.1 *Relays and their parameters*

Part No.	Contact type	Coil voltage	Coil resistance (Ω)	Contact current	Contact voltage	Pins
2077394 (J V-3S-KT)	SPST-NO	3 V_{dc}	45	5*A*	250 V_{ac} / 30 V_{dc}	4
1860070 (HE3621*A*0500)	SPST-NO	5 V_{dc}	500	1*A*	200 V_{dc}	4
2077378 (FBR161NED006)	SPST	6 V_{dc}	100	10*A*	120 V_{ac} / 30 V_{dc}	5
843155 (JS1-5 V)	SPDT	5 V_{dc}	69.4	10*A*/ 5*A*	250 V_{ac}/100 V_{dc}	5
2081650 (EC2-5NJ)	DPDT	5 V_{dc}	178	2*A*	220 V_{ac}/220 V_{dc}	8
2133499 (S3-12 V)	DPDT	12 V_{dc}	720	4*A*	250 V_{ac} / 30 V_{dc}	12

20.2 | OPTO-COUPLER

Opto-couplers are devices used to electrically isolate the dangerous high voltage/current load circuits being controlled by low-power controlling circuits (made from microcontrollers). They are also referred as *opto-isolators*. They have some features of transistors and relays, they are used in solid-state relays. The electromagnet of a relay is replaced by pair of LED and photosensitive element like phototransistor or photothyristor; it is used as a switching element instead of a metallic switch and spring. The principle is that the opto-coupler uses an optical path to transfer a signal between components (devices) of circuits operating at different power levels and this way, there is no electrical connection between the circuits, which isolates them electrically. Since there is no mechanical component, it can operate at a high switching frequency of the transistor. The LED and phototransistor must be separately powered to get the perfect isolation. Interfacing of an opto-coupler with the 8051 is illustrated in Figure 20.4.

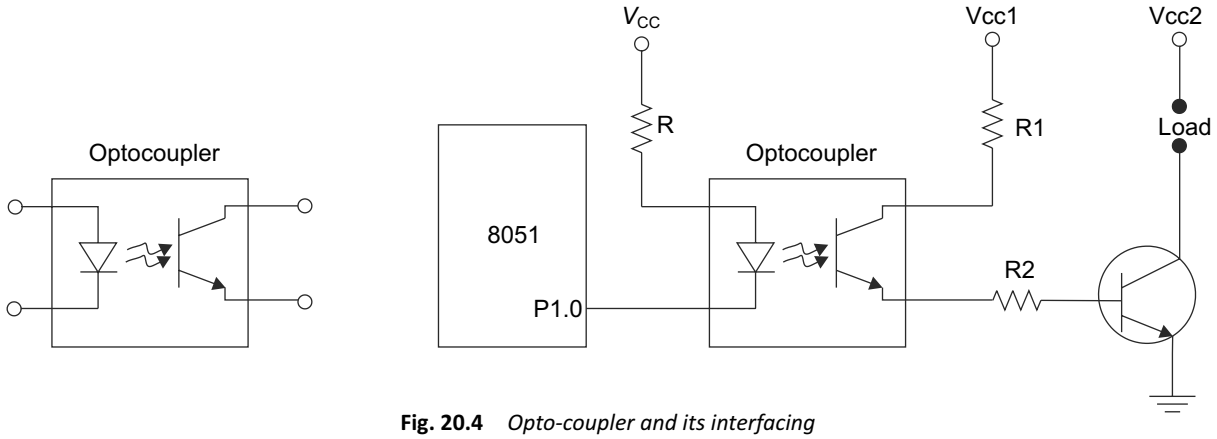


Fig. 20.4 *Opto-coupler and its interfacing*

20.2.1 Opto-Coupler Operation

An opto-coupler has an LED and phototransistor pair (or pairs) separated by an air gap. The LED will act as a transmitter and the phototransistor will act as a receiver. The control input is applied at the input of the LED. When the control input is high, the current flows through the LED, it transmits a light through the gap and the phototransistor will conduct and produces a signal proportional to the signal applied to the LED. This way, a signal is transmitted without any direct physical connection between the components and thus, it will provide the isolation. Opto-couplers are available in IC package form which contains one or more opto-couplers.

The circuit shown in Figure 20.4 shows that a simple transistor driver circuit is required between the load and opto-coupler. This circuit can be used for the load requiring low currents (around 500 mA). For loads which require higher currents, the other driver is needed between the opto-coupler output and the load as shown in Figure 20.17.

Some of the popular opto-couplers with their specifications are listed in Table 20.2.

Table 20.2 *List of common opto-couplers*

Opto-coupler	Typical Current Transfer Ratio (CTR)	Isolation voltage
IL74/ILD74/ILQ74(Single/dual/quad channel) Phototransistor opto-couplers (TTL compatible)	35%	5300 V_{rms}
4N25/4N26/4N27/4N28 single channel opto-coupler with base connection	>20/20/10/10%	5000 V_{rms}
PC817/PC827/PC837/PC847 (Single/dual/triple/quad channel)	50% minimum	5000 V_{rms}
MCT2/MCT2E Phototransistor opto-couplers (TTL compatible)	High	1500/3550 V_{rms}

20.2.2 Applications of Opto-Couplers

The common applications of opto-couplers are listed briefly as follows:

- ac mains detection and power supply feedback applications and power supply regulators
- SMPS
- Medical, industrial, automotive equipment
- Logic ground isolation, network bus isolation and power supply isolation
- Signal transmission between circuits of different potentials and impedances

20.3 | STEPPER MOTORS

A stepper motor, as the name suggests, rotates in discrete angular steps. Each step is initiated in response to the application of a current pulse. The specific number of pulses will rotate the motor to a specific known angle; hence, the motor's position and movement can be controlled accurately without any feedback mechanism (an open-loop controller) because each pulse can be controlled by a microcontroller. Therefore, they are well suited for position-control applications like robotic arms (or any pick-and-place equipment), cutting machines, automatic assembly lines, printers, plotters, disk drives and image scanners.

Three types of stepper motors are available: permanent-magnet, variable-reluctance and hybrid stepper motors, but only the permanent-magnet stepper motor is discussed in more detail in the following section.

20.3.1 Permanent-Magnet Stepper Motors

Permanent-magnet stepper motors are made from two parts: a rotor and a stator. The rotor is gear-shaped and permanently magnetized and the stator has two coil pairs (also called phases) wound on several teeth (poles) of a stator. The coil wound on a tooth is effectively an electromagnet. The simplified operation of a stepper motor is shown in Figure 20.5.

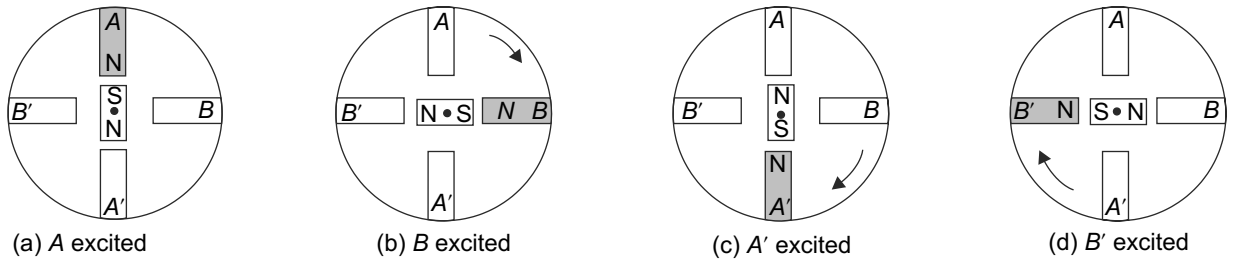


Fig. 20.5 *Simplified operation of stepper motor*

When Coil *A* is energized, the rotor is attracted towards it and the rotor aligns itself with Coil *A* and stops. Next, Coil *B* is energized, which attracts the rotor towards it, thus the rotor moves from Coil *A* to Coil *B*. If we successively energize coils *A*, *B*, *A*', *B*', as shown in Figure 20.5, the rotor is turned clockwise and completes one revolution. The sequence is repeated for the continuous rotation of the motor. The minimum amount of rotation in a single step (pulse) is called step-angle and depends on the number of teeth of both the stator and rotor. The stepper motor step angle can be as small as 0.72° or as large as 90°. The common step angles are 1.8°, 2.0°, 5.0°, 7.5° and 15°.

There are two types of PM stepper motors based on the arrangement of stator winding: unipolar and bipolar stepper motors.

20.3.2 Unipolar and Bipolar Stepper Motors

In a unipolar motor, each coil has a centre tap lead as shown in Figure 20.6, and has a total of six leads (or five when both the common leads are shorted). The bipolar motor is similar to the unipolar motor except that it does not have a center tap connection and has a total of four leads.

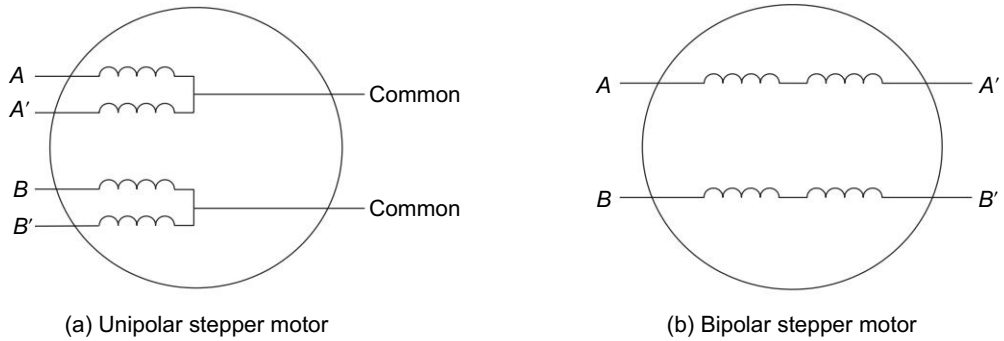


Fig. 20.6 *Unipolar and bipolar stepper motor*

The operation of a unipolar stepper motor is shown in Figure 20.7 and described in the next paragraph. As shown in the figure, a common lead is connected to supply voltage (Logic 1) and the other ends are connected to the ground (Logic

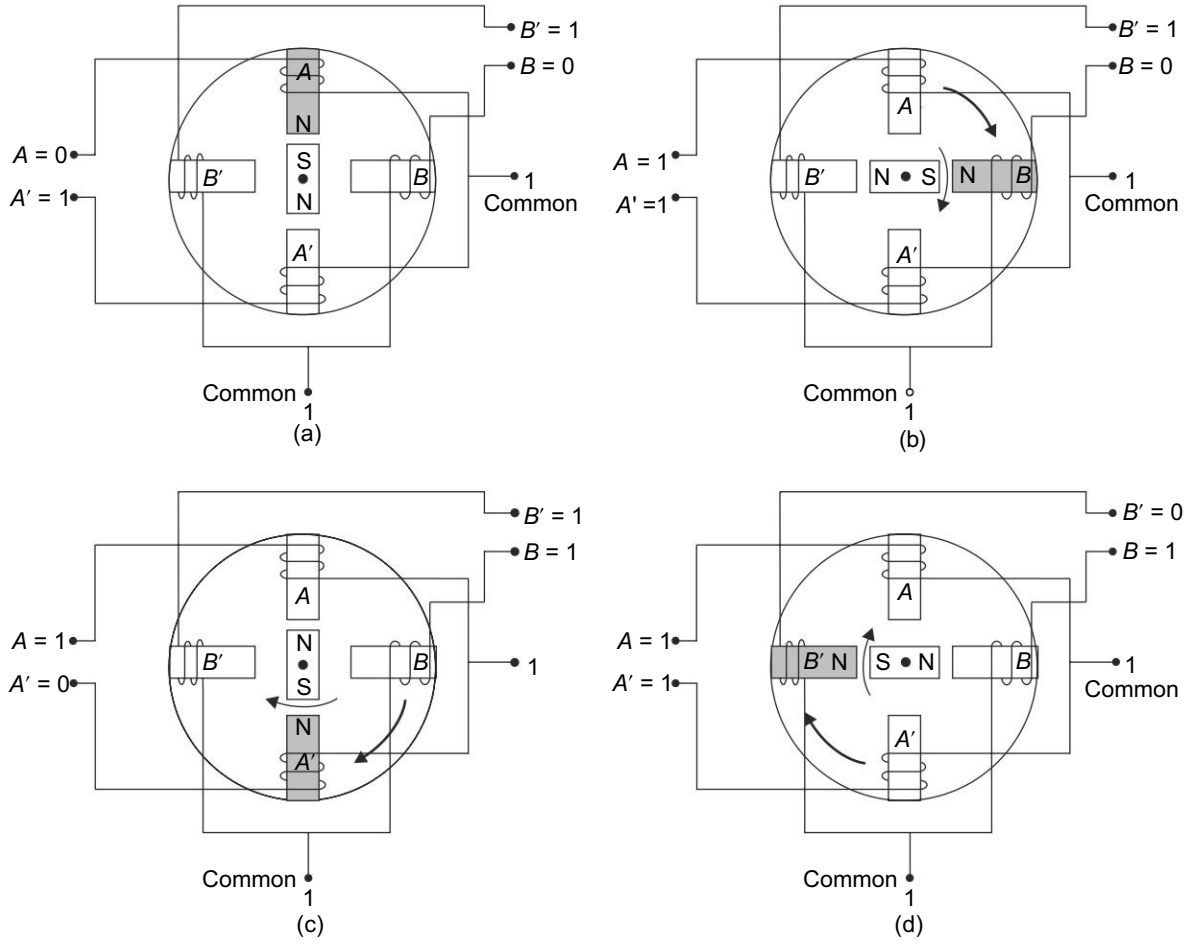


Fig. 20.7 *Operation of unipolar stepper motor*

0) to energize the coil. Only a single half of the coil is energized at a time, hence they produce less torque. The current always flows in only one direction through each coil (phase); therefore, the name is unipolar. The advantage is that they require simpler driver circuits which reduce the system cost and complexity.

Initially, phase *A* is energized by connecting lead *A* to ground, i.e. *A* = 0 (note that to allow the current to flow through coil lead, *A* should be connected to ground when the common lead is at logic 1- supply voltage) and all other phases are connected to the supply voltage, i.e. *A*' = *B* = *B*' = 1.The rotor is attracted towards phase *A*. Next, Phase *B* is energized by making lead *B* = 0, and *A* = *A*' = *B*' = 1. Now, the rotor is attracted towards phase *B*. Same way Phase *A*' and *B*' are energized successively and the motor rotates in a clockwise direction. The sequence of logic levels to be applied to all phases to the rotate motor is summarized in Table 20.3.

Table 20.3 *Logic levels and sequence to rotate unipolar motors (wave drive)*

Step	Phase *A*	Phase *B*	Phase *A*'	Phase *B*'
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

In the above sequence of activating phases *A*, *B*, *A*', *B*', only one phase at a time is more popularly known as a *wave drive* 4-step sequence.

If we energize two phases at a time and follow the sequence as *A* + *B* (both *A* and *B* phases are energized simultaneously), *B* + *A*', *A*' + *B*', *B*' + *A*,…, the motor will rotate in the clockwise direction in a similar manner as discussed above, but the advantage is that motor will produce almost double torque. The disadvantage is that this will consume more power because the two coils are drawing the current simultaneously. This sequence is known as *full-step sequence*. It is summarized in Table 20.4 (a).

Microcontrollers are not capable of driving the coils of a stepper motor directly because of the higher current requirements of the motors. Therefore, the microcontrollers are interfaced with stepper motors through driver circuits. Driver circuit may be made from a discrete transistor and diode pairs or it may be a driver chip.

Table 20.4(a) *Full-step sequence*

Step	Phase *A*	Phase *B*	Phase *A*'	Phase *B*'
1	0	0	1	1
2	1	0	0	1
3	1	1	0	0
4	0	1	1	0

Table 20.4(b) *Inverted full-step sequence when interfaced through the driver*

Step	Phase *A*	Phase *B*	Phase *A*'	Phase *B*'
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

The common characteristic of both types of drivers is that they invert the input signal. Therefore, the inverted sequence should be generated by a microcontroller as shown in Table 20.4 (b). The inverted sequence is required in all types of sequences when they are applied through drivers.

Yet, there is another sequence. If we energize phases *A*, *A* + *B*, *B*, *B* + *A*', *A*', *A*' + *B*', *B*', *B*' + *A*'… successively, it will halve the angular rotation (step angle) between each step and hence doubles the steps per revolution, i.e. increases the resolution of the rotation. This sequence is called *half-step sequence*. The half-step sequence is summarized in Table 20.4 (c). The resolution of a stepper motor is defined as the number of pulses required to complete one revolution. The other popular sequence is a micro-sequence but it is not discussed.

Table 20.4(c) *Half-step sequence*

Step	Phase *A*	Phase *B*	Phase *A′*	Phase *B′*
1	0	1	1	1
2	0	0	1	1
3	1	0	1	1
4	1	0	0	1
5	1	1	0	1
6	1	1	0	0
7	1	1	1	0
8	0	1	1	0

The bipolar stepper motors are rotated by sequence, $A - A'$ (A is connected to logic '1' and A' is with '0'), $B - B'$ ($B = 1$, $B' = 0$), $A' - A$, $B' - B$… (Note that the bipolar motors do not have a common terminal). The advantage of a bipolar motor is that they produce more torque because all the phases are utilized simultaneously. The disadvantage is that they require more complicated driver circuits because polarities of both the ends of coils are changed simultaneously. It is usually achieved by an H-bridge arrangement. Refer topic 20.4.3 and Figure 20.11 for circuit and operation of H-bridge.

20.3.3 Direction and Speed Control

The direction of rotation can be changed by reversing the sequences discussed in the above section. It is true for all types of sequences.

The speed of rotation depends on the frequency of pulses given by a microcontroller and type of sequence, i.e. half or full sequence, construction of a motor (number of rotor and stator teeth) and of course on the load. For example, if the stator has 4 poles and the rotor has 5 teeth, the motor will rotate 18° (360°/4 × 5) per step. 1.8° is the most common step-angle. The list of some stepper motors with their parameters is given in Table 20.5.

Table 20.5 *Stepper motors with their specification*

Part No.	Type	Rated voltage (V_{dc})	Step angle (°)	Phase resistance (Ω)	Current (mA)	Holding torque (g-cm)
237825	Unipolar	5	7.5	10	500	110
237607	Unipolar	5	1.8	5	1000	3976
224022	Unipolar	12	1.8	75	160	900
237490	Bipolar	5	1.8	5	1000	1800
237472	Bipolar	12	1.8	30	400	2100

20.3.4 Interfacing with the 8051

Interfacing of the stepper motor with the 8051 is discussed in Interfacing Example 20.1.

Interfacing Example 20.1

Interface a unipolar stepper motor with the 8051 using a suitable driver circuit and write a program to rotate the stepper motor in clockwise direction using full-step sequence.

Solution:

Interfacing of a unipolar stepper motor with the 8051 is shown in Figure 20.8. The leads *A*, *A*', *B* and *B*' are connected with the microcontroller pins through a driver circuit. The sequences discussed above are generated using a microcontroller and given to the stepper motor. The operation of the circuit is further explained with help of the program.

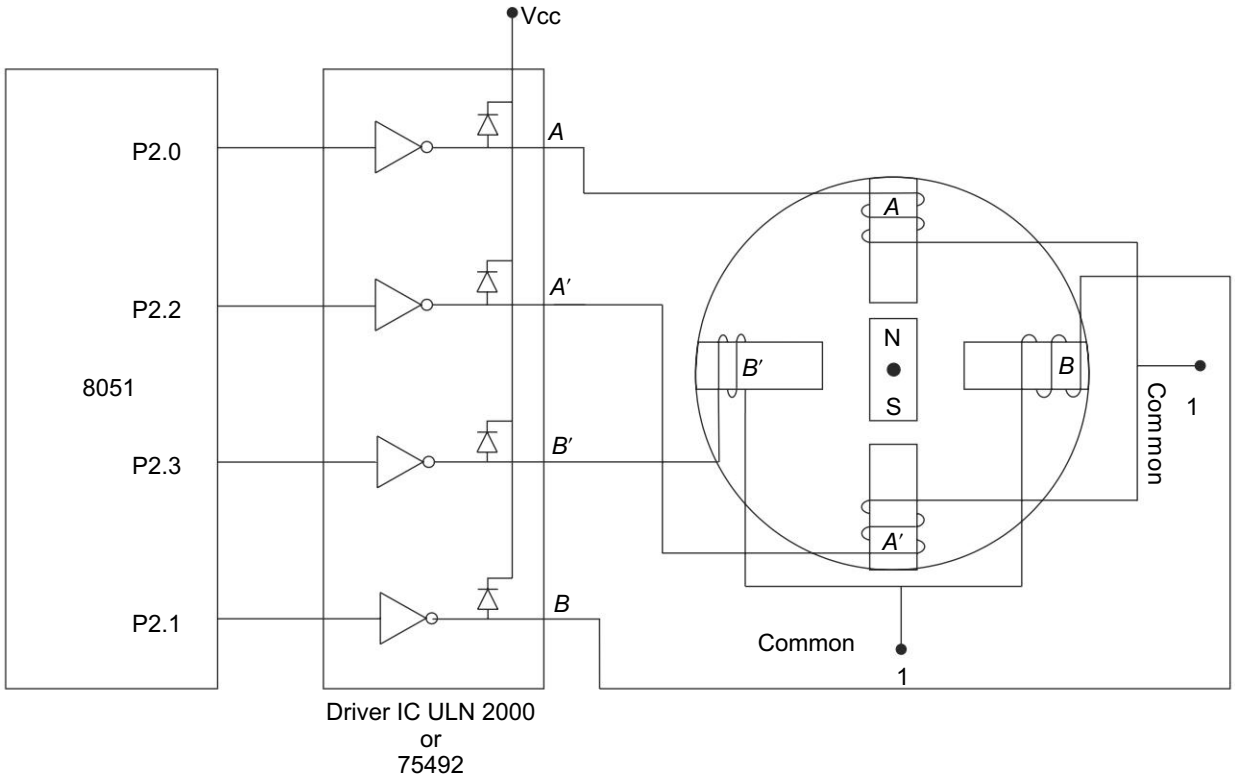


Fig. 20.8 *Interfacing of stepper motor with the 8051*

Circuit Operation

The four leads of stator winding (*A*, *A*′, *B*, *B*′) are connected with the microcontroller port pins P2.0 to P2.3 through a driver circuit. The common terminals are connected with V_{cc}. The driver circuit used here is ULN2000 because it has 7 Darlington drivers with internal free-wheeling diodes for protection as discussed in the above section. (We may also use a discrete transistor driver but it will require the extra diodes to be interfaced, which will unnecessarily increase circuit complexity.)

When we make any port pin high, the driver output is low, and the current will flow through the corresponding coil. For example, when P2.0 is made low, the current will flow from the common terminal (V_{cc}) to the winding *A* to the output *A* of the driver circuit. This way Coil *A* is energized, similarly any other coil can be energized by making the corresponding port pin high.

Program-development steps to rotate the stepper motors are the following:

- To generate a full-step sequence, the port pins are loaded with any one of the four values. The values are CC, 66, 33, 99 (these are inverted values because of inverting the nature of a driver circuit).
  To generate wave drive sequence, the four values are 88, 44, 22 and 11 (inverted).
  To generate half-step sequence, the eight values are 08, 0C, 04, 06, 02, 03, 01 and 09.
- To rotate the motor in a clockwise direction, rotate the initial value loaded in port pins in the right direction using RR A instruction.
  To rotate the motor in anti-clockwise direction, rotate the initial value in left direction using RL A instruction.
  Note that half-step sequence will not be generated simply by rotation. Therefore, we need to store these values in a look-up table and access these values one after the other.
- Repeat the above step continuously for rotation of the motor.

The program to rotate a stepper motor in clockwise direction using full-step sequence is given below.

In Figure 20.8, a driver circuit is used to interface the microcontroller with a stepper motor, and as discussed above, drivers usually invert the inputs. Therefore, we need to use inverted full-step sequence as shown in Table 20.4 (b) (inverted full-step sequence when interfaced through the driver).

```
        ORG 0000H
        MO V A, #0CCH       // full-step sequence, the first entry in table is 1100 = C.
                            // here, CC is used for the continuous rotation
```

```
REPEAT:  MO V P2, A          // send sequence to motor
         RR A                // next step of the sequence to rotate in clockwise rotation
         LCALL DELAY         // wait before going to the next step
         SJMP REPEAT         // next step
DELAY:   MO V R1, # 60H      // delay, it can be varied to change speed of rotation
THERE:   MO V R2, #0FFH
HERE:    DJNZ R2, HERE
         DJNZ R1, THERE
         RET
         END
```

Example 20.2

Rewrite the program of Interfacing Example 20.1 in the C language.

Solution:

```
#include<reg51.h>

    void main (void)
        {
        unsigned char i, j,a[ ] = {0xCC,0x66,0x33,0x99};
                                    // full-step sequence,
                                    // array is used to store steps of sequence
        while (1)                   // send the sequence continuously
            {
            for (j = 0; j< = 3; j++)
                {       P2 = a[j];
                        for ( i = 0; i<20; i++);  // delay
                }
            }
        }
```

Example 20.3

Modify the program of Interfacing Example 20.1 to rotate the motor in anti-clockwise direction.

Solution:

Replace instruction RR A with RL A in the program of Example 20.1. This will send the reverse sequence steps to the stepper motor and, therefore, it will rotate in the reverse direction.

```
         ORG 0000H
         MO V A, #0CCH       // full-step sequence, the first entry in table is 1011 = C.
                             // here, CC is used for continuous rotation.
REPEAT:  MO V P2, A          // send the sequence to motor
         RL A                // next step of sequence to rotate in anti-clock-wise rotation.
         LCALL DELAY         // wait before going to the next step
         SJMP REPEAT         // next step
DELAY:   MO V R1, # 60H      // delay, it can be varied to change the speed of rotation
THERE:   MO V R2, #0FFH
HERE:    DJNZ R2, HERE
         DJNZ R1, THERE
         RET
         END
```

Example 20.4

Rewrite the program of Interfacing Example 20.3 in the C language.

Solution:

```
#include<reg51.h>

        void main (void)
                {
                unsigned char i, j,a[ ] = {0xCC,0x99, 0x33, 0x66};
                                // full-step sequence, array is used to store steps of sequence
                while (1)       // send sequence continuously
                        {
                        for (j = 0; j< = 3; j++)
                              {     P2 = a[j];
                                    for ( i = 0; i<20; i++);// delay
                              }
                        }
                }
```

Example 20.5

Modify the program of Example 20.1 to drive a stepper motor using a wave-drive sequence.

Solution:

For the given modification, only the step sequence is required to be modified. In a wave drive, only one phase is active at a time (see Table 20.3). Therefore, in the first step, 1000 (note that it is inverted) is to be applied to the phases. We use 88 here instead of 8 because we want continuous rotation.

Replace the instruction MO V A, #0CCH with MO V A, #88H in the program of Example 20.1.

The program is listed below.

```
        ORG 0000H
        MO V A, #88H        // wave drive sequence, the first entry in table is 1000 = 8
                            // here 88 is used for continuous rotation.
REPEAT: MO V P2, A          // send the sequence to motor
        RR A                // next step of sequence to rotate in clockwise rotation.
        LCALL DELAY         // wait before going to the next step
        SJMP REPEAT         // next step
DELAY:  MO V R1, # 60H      // delay, it can be varied to change the speed of rotation
THERE:  MO V R2, #0FFH
HERE:   DJNZ R2, HERE
        DJNZ R1, THERE
        RET
        END
```

Example 20.6

Rewrite the program of Interfacing Example 20.5 in the C language.

Solution:

```
#include<reg51.h>

        void main (void)
                {
                unsigned char i, j,a[ ] = {0x88,0x44,0x22,0x11};
```

```
                                        // wave drive sequence,
                                        // array is used to store steps of sequence
while (1)                               // send sequence continuously
        {
        for (j = 0; j< = 3; j++)
                {       P2 = a[j];
                        for ( i = 0; i<20; i++);                    // delay
                }
        }
}
```

Example 20.7

For a circuit of Figure 20.8, write a C program to rotate the stepper motor in a clockwise direction using a half-step sequence.

Solution:

Refer Table 20.4 (c) for half-step sequence.

```
#include<reg51.h>

    void main (void)
        {
        unsigned char i, j,a[] = {0x08, 0x0C,0x4, 0x06,0x02, 0x03,0x01,09};
                                        // half-step sequence, array is used to store steps of sequence
        while (1)                       // send sequence continuously
                {
                for (j = 0; j<8; j++)
                        {       P2 = a[j];
                                for ( i = 0; i<20; i++);                    // delay
                        }
                }
        }
```

Note that sequence is inverted.

THINK BOX 20.1

Why can we not connect the motors directly to the microcontroller pins?

Usually, the current required by the motor is larger than what a microcontroller pin can provide; therefore, we need to connect interfacing circuit (driver) between them.

20.3.5 Rotation of Motor for Specified Angle

For position-control applications, the rotation of a motor should be stopped after it has moved certain angle (or certain number of revolutions). This can be achieved by applying a fixed number of pulses to the motor. The number of pulses required for the desired angular rotation is given as,

$$\text{No. of pulses} = \frac{\text{Desired angular rotation}}{\text{Step angle}}$$

Example 20.8

Write a program to rotate a stepper motor by 80° in the clockwise direction using a full-step sequence. The motor has a step angle of 2°.

Solution:

$$\text{No. of pulses required} = \frac{\text{Desired angular rotation}}{\text{Step angle}} = \frac{80^\circ}{20^\circ} = 40.$$

The motor should be given only 40 pulses to rotate it by 80°.

```
        ORG 0000H
        MO V A, #0CCH       // full-step sequence, the first entry in table is 1100 = C.
                            // here CC is used for continuous rotation.
        MO V R3, #40        // counter for 40 pulses
REPEAT: MO V P2, A          // send sequence to motor
        RR A                // next step of sequence to rotate in clock-wise rotation
        LCALL DELAY         // wait before going to the next step
        DJNZ R3, REPEAT     // next step
HERE1:  SJMP HERE1          // stop after 40 pulses

DELAY:  MO V R1, # 60H      // delay, it can be varied to change the speed of rotation
THERE:  MO V R2, #0FFH
HERE:   DJNZ R2, HERE
        DJNZ R1, THERE
        RET
        END
```

20.3.6 Applications of Stepper Motors

The common applications of stepper motors are listed below:

- Dot matrix printers
- Disk drives, floppy drive, CD drives, plotters, image scanners
- Robotics (robotic arms or any pick and place equipment)
- Industrial controls, automation and automatic assembly lines
- Cutting machines
- Low-speed high-torque applications

20.4 | DC MOTORS

A DC motor consists of a stator and a rotor. The stator consists of a frame, field system and brushes. The frame (or yoke) is the outermost metal part of the motor and forms a part of the magnetic circuit. The field system consists of field poles and field windings; the winding is placed over the poles and arranged such that when the current is passed to it, alternate poles are magnetized with N and S polarity which will produce flux (magnetic field) in the air gap. The rotor consists of an armature and winding on it. When armature windings are connected to a DC supply, the current-carrying armature conductors produce their own field; this field tries to come in line with the magnetic field generated by field winding, thus electromagnetic torque is developed on an armature and the motor starts rotation. The torque and the speed of DC motor can be accurately controlled.

The DC motors use DC voltage (direct current) to achieve the rotary motion. They have two terminals, positive and negative, with which to control the speed and direction of the rotation. Connecting DC power supply to these terminals rotates the motor in one direction and reversing the polarity of the power supply reverses the direction of rotation. Unlike the stepper motor where the number of steps can be controlled, the DC motor rotates continuously and only the speed can be controlled. The speed of the DC motor is measured in Revolutions Per Minute (RPM).

20.4.1 Analog Speed Control

The speed of the DC motor depends on the supply voltage and the load connected to it. The speed increases with the supply voltage but, we cannot exceed the supply voltage beyond the rated voltage. The speed of the DC motor is highest at no load and as the load is increased, the speed is decreased. Overloading the DC motor will damage it because of the excessive heat generated due to the high current consumption at higher loads. Certain minimum supply must be applied to overcome the inertia of motor and make the motor to turn.

20.4.2 Digital Speed Control

Speed of the dc motor can be controlled by a digital PWM signal. The PWM signal will turn the motor on and off at a very fast rate. If the PWM signal is fast enough, the motor will be turned off before it reaches the maximum speed, and will be turned on again before speed reduces to zero. So, on an average, the motor will rotate at a speed that is proportional to the duty cycle of the PWM signal.

20.4.3 Direction Control

If the polarity of the supply voltage is reversed, the direction of rotation is also reversed. The effect of reversing the polarity of the supply voltage is illustrated in Figure 20.9. This type of permanent connections is suitable when the application requires fixed direction of rotation.

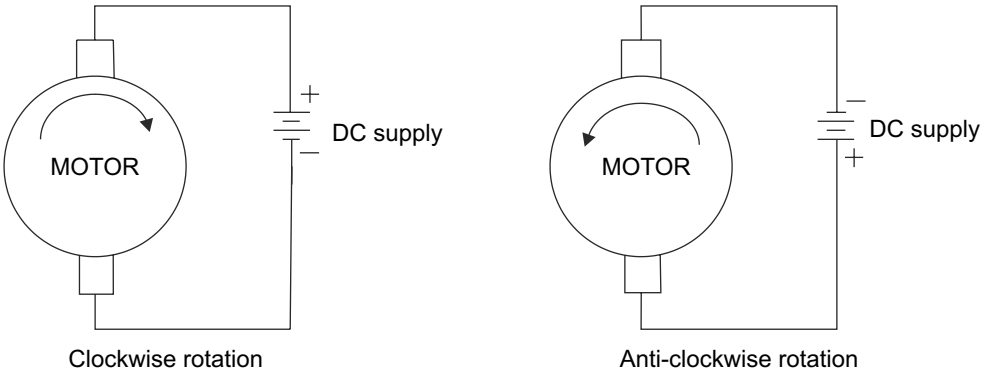


Fig. 20.9 *Direction control of the DC motors*

In some applications, bidirectional control is required. It is achieved using H-bridges; they contain four switches. By changing the state (ON or OFF) of the switches, the direction of current in the motor windings can be changed and, therefore, direction can be changed. The operation of an H-bridge is illustrated in Figure 20.10.

As shown in the figure, when all the switches are open, no current will flow through the motor and it will be off. When only SW1 and SW4 are closed (shorted) the current will flow through the motor and it will be rotated in a clockwise direction. Similarly, when SW2 and SW3 are closed, the motor will rotate in an anticlockwise direction. When all the switches are closed, it will short the power supply. The practical motor drive circuit for bidirectional control of DC motor is shown in Figure 20.11.

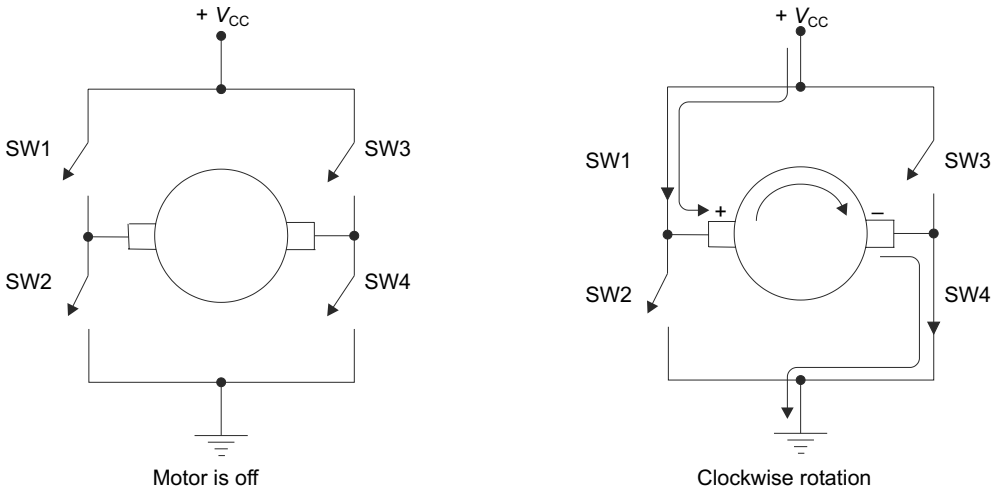


Fig. 20.10 *(Contd.)*

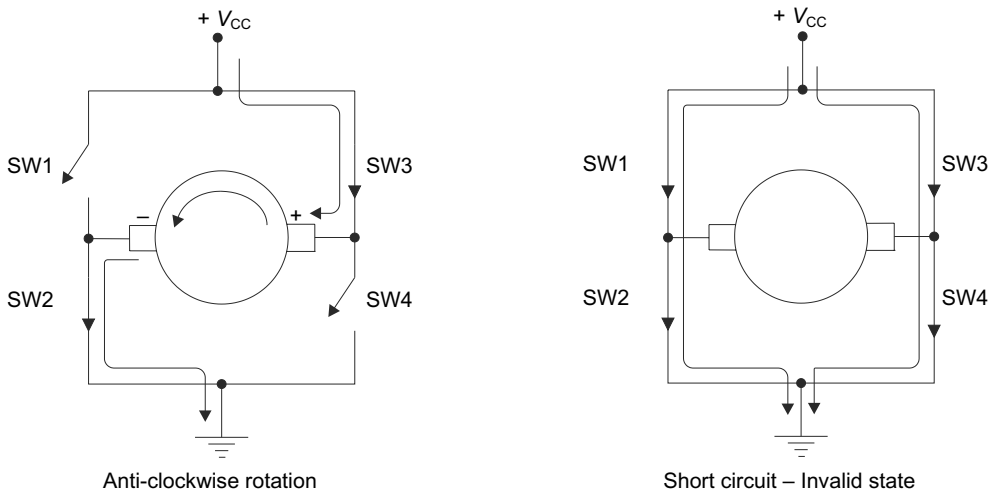


Fig. 20.10 *Bidirectional control of the DC motor using H bridge*

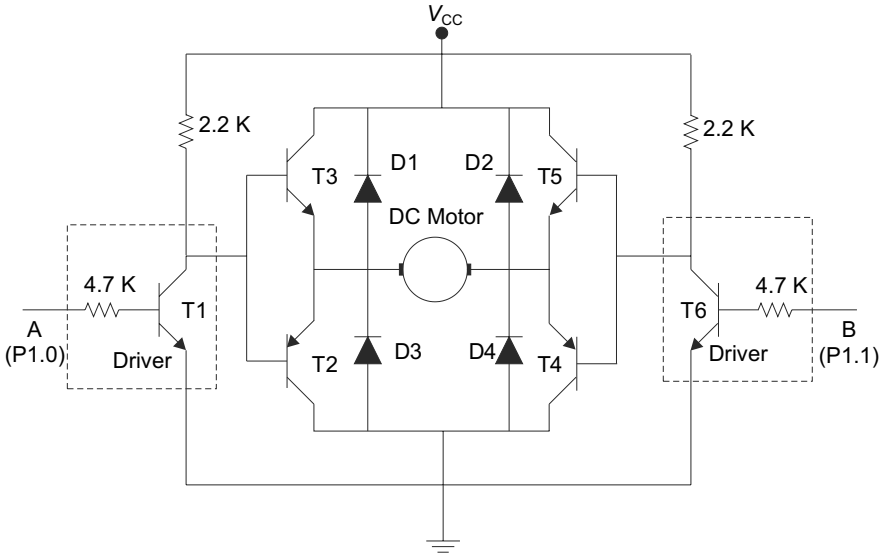


Fig. 20.11 *DC motor drive circuit for directional control*

As already mentioned in the above section, the direction of the DC motor is controlled using four switch (four transistors) circuit known as *H-bridge*. The H-bridge (or full-bridge) consists of two pairs of push-pull drivers (T2-T3 and T4-T5 in Figure 20.11), called half-bridges and with the motor (or other load) connected between them. Two digital signals *A* (port pin P1.0) and *B* (P1.1) control the direction of the rotation. If $A = 0$ and $B = 1$, the transistors T3 and T4 will conduct and the current will flow in path V_{cc} – T3 – Motor – T4 – Ground, and the motor will rotate in a clockwise direction. When $A = 1$ and $B = 0$, the transistors T2 and T5 will conduct and the current will flow in path V_{cc} – T5 – Motor – T2 – Ground, and the motor will rotate in an anti-clockwise direction. Four diodes (known as freewheeling diodes) protect the transistors from a voltage that is generated when the motor is suddenly turned off.

Digital speed control is easily possible by setting $B = 0$ and putting a PWM signal on *A*.

Interfacing Example 20.9

Assume that switch SW is connected to the pin P2.0 and P1.0 is connected to the input of half-bridge (Point *A* in Figure 20.11) and P1.1 is connected to the input of another half-bridge (Point *B*). Write a program to monitor the status of SW, if SW = 1, rotate the dc motor in a clockwise direction, or if SW = 0, reverse the direction of the motor.

Solution:

To rotate the motor in the clockwise direction, *A* should be 1(P1.0 = 1) and *B* should be 0 (P1.1 = 0) and to reverse the direction, *A* should be 0(P1.0 = 0) and *B* should be 1 (P1.1 = 1)

```
        SETB P2.0              // configure P2.0 as input because switch is connected with it
REPEAT: JB P2.0, CLKWISE       // if SW = 1, make P1.1 = 0, P1.0 = 1
        MO V P1, #00000010B    // otherwise, make P1.1 = 1, P1.0 = 0 to rotate  motor in anti-clockwise direction
        SJMP REPEAT            // repeat the operation
CLKWISE: MOV P1, #00000001B    // rotate motor in clockwise direction
        SJMP REPEAT            // repeat the operation
```

Example 20.10

Rewrite the program of Interfacing Example 20.9 in the C language.

Solution:

```
#include<reg51.h>
        sbit SW = P2^0;

        void main()
                {
                        SW = 1;                 // configure P2.0 as input because the switch is connected with it
                while(1)                        // continuously monitor status of SW
                        {
                        if (SW = = 1)           // if SW = 1, make P1.1 = 0, P1.0 = 1
                          P1 = 1;               // to rotate motor in clockwise direction
                        else                    // otherwise, make P1.1 = 1, P1.0 = 0
                         P1 = 2;                // to motor in anti-clockwise direction
                        }
                }
```

20.4.4 Pulse-Width Modulation (PWM)

In pulse-width modulation, the duty cycle of digital pulse train used to drive the motor is varied (modulated) and this will change the average DC voltage (and hence power of a pulse train is varied), which will effectively change the speed of the dc motor. Duty cycle is ratio of ON time to total time of a pulse. The effect of changing the duty cycle of digital pulse on the average voltage is illustrated in Figure 20.12.

As can be seen from Figure 20.12, when the duty cycle is less, the average DC voltage is also less and as duty cycle is increased, average dc voltage also increases, thus by changing duty cycle (pulse width), we can change the applied power to the motor, which will change the speed of the DC motor. Note that the total time period of the pulse remains constant, and this total time period is chosen such that the inertia of the rotor will smooth out voltage (or power) fluctuations.

20.4.5 DC Motor Driver Circuits

The driver circuit may be designed from a transistor, MOSFET or operational amplifier as shown in Figure 20.13. The control input shown in the figure may be DC voltage from DC voltage source or PWM signal (usually generated by the microcontrollers).

The transistor driver as shown in Figure 20.13(a) is suitable for small motors but it is not suitable for the larger motors because large power dissipation will occur in the transistor itself. Note that the control signal cannot be directly given from the microcontroller but a small signal driver stage is required in between. The freewheeling diodes protect the transistor (or any other switching device) from overvoltage that is generated when the motor is suddenly turned off. The MOSFET (power MOSFET, for example, IRF540, IRFH3707) drivers have high input impedance, high switching speeds and allow larger currents [Figure 20.13(b)]. Op-Amp drivers (power Op-Amp) driver circuits can provide large currents around 10 A with a voltage around 30 V (Figure 20.13(c)) (see Figure 20.2 for other driver circuits). The list of DC motors with their parameters is given in Table 20.6.

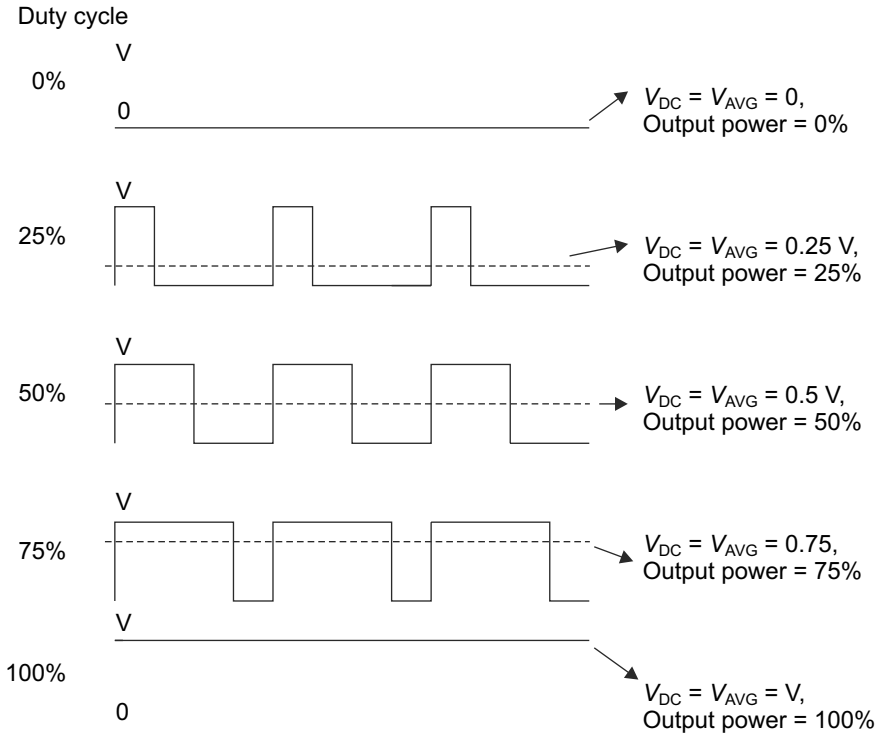


Fig. 20.12 *Effect of duty cycle on average DC voltage*

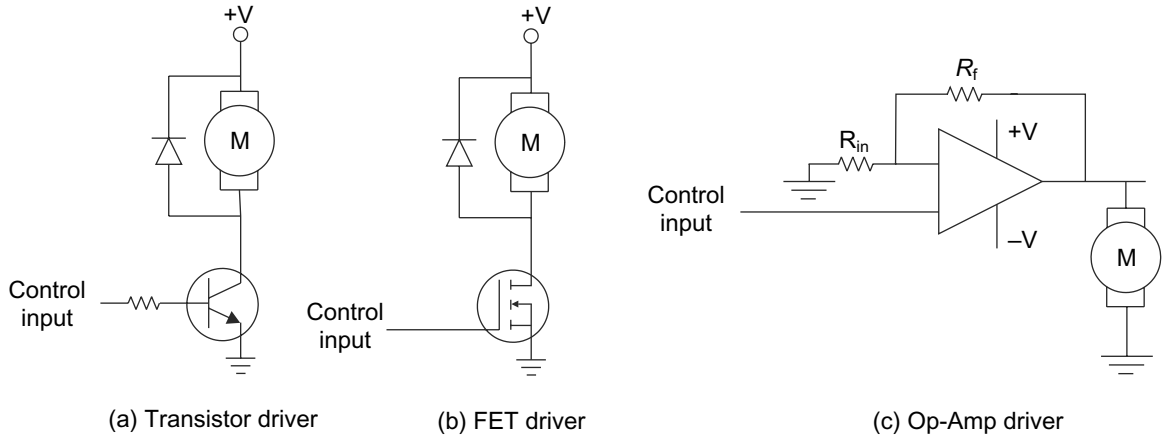


Fig. 20.13 *DC motor driver circuits*

Table 20.6 *DC motors with their parameters*

Part No.	Operating Voltage (V_{dc})	Current (mA)	Speed (rpm)	Torque (g-cm)
238511	6	80	2100	11
231829	6	280	4260	18
2095533	9	370	6320	32
1955669	12	250	6600	34
174693	12	15	3000	85
232040	12	1200	12150	62

20.4.6 Interfacing DC Motors with the 8051

When the motors are suddenly turned off or when there is a sudden change of current in the windings, a back emf (high voltage spike) is generated which may damage switching element or control circuit including a microcontroller. These spikes are referred as switching transients. To protect the microcontrollers from these switching transients, microcontrollers are isolated from high power circuits of motor drivers. The simple way to obtain isolation is to use an opto-coupler between the microcontroller-based control circuit and motor-driver circuits. The operation of opto-coupler is already given in topic 20.2 of this chapter. The interfacing circuit is shown in Figure 20.14.

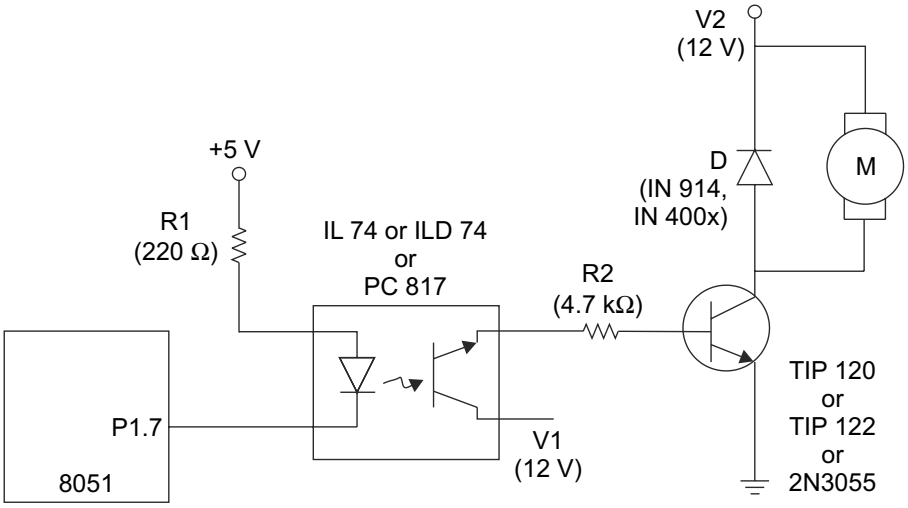


Fig. 20.14 *Interfacing DC motor with 8051 through opto-coupler*

The circuit uses opto-coupler which isolates the motor and the 8051. Note that separate power supplies are used for the microcontroller and motor driver circuit, this will also allow use of high-power DC motors (high-power loads). For the circuit shown in Figure 20.14, the port pin P1.7 is used to control the switching of the DC motor. When P1.7 is made low, LED in the opto-coupler will glow and the phototransistor will conduct, and the transistor base current will flow, the transistor is switched ON, and the motor will rotate. When P1.7 is high, the motor is switched OFF. Remember, for the circuit shown in Figure 20.14, the motor can rotate only in one direction.

Interfacing Example 20.11

Assume that switch SW (ON-OFF) is connected to port P2.0 for the circuit shown in Figure 20.14. Write a program to monitor the status of the switch. If it is low, apply 25% DC power, otherwise, apply 50% DC power to the motor using PWM technique.

Solution:

The desired operation can be achieved using the following steps:
The status of SW is monitored using the JB/JNB instruction.
If switch SW = 1 (P2.0 = 1),
    Turn ON motor by making P1.7 = 0 (see description of Figure 20.14)
    Apply delay1 (wait),
    Turnoff motor by making P1.7 = 1
    Apply delay2; it will be three times delay1,
Else,
    Turn ON motor by making P1.7 = 0
    Apply delay3 (wait), it will be two times the delay 1
    Turnoff motor by making P1.7 = 1
    Apply delay3 (wait), it will be two times the delay 1

    Repeat the above process continuously.

```
        ORG   0000H
        SETB  P2.0          // configure P2.0 as input because switch is connected with it
        SETB  P1.7          // initially turn OFF motor
REPEAT: JB P2.0, DUTY50
        CLR   P1.7          // turn ON motor (high portion of pulse)
        MO V  R2, #25       // 25 % ON time
        ACALL DELAY
        SETB  P1.7          // turn OFF motor (low portion of pulse)
        MO V  R2, #75       // 75 % OFF time
        ACALL DELAY
        SJMP  REPEAT        // repeat the task of monitoring switch
DUTY50: CLR   P1.7          // turn ON motor (high portion of pulse)
        MO V R2, #50        // 50 % ON time
        ACALL DELAY
        SETB  P1.7          // turn OFF motor (low portion of pulse)
        MO V  R2, #50       // 50 % ON time
        ACALL DELAY
        SJMP  REPEAT        // repeat the task of monitoring switch
DELAY:
HERE1:  MO V  R3, #255
HERE:   DJNZ  R3, HERE
        DJNZ  R2, HERE1
        RET
        END
```

Example 20.12

Rewrite the program of Interfacing Example 20.11 in the C language.

Solution:

```
#include<reg51.h>
sbit SW = P2^0;                 // define P2.0 as SW
sbit MOTOR_CTRL = P1^7;         // define P1.7 as MOTOR_CTRL
void delay (unsigned int);      // delay subroutine

void main()
{
        SW = 1;                 // configure P2.0 as input because switch is
                                // connected with it

        MOTOR_CTRL = 1;         // initially turn OFF motor
        while (1)               // repeat the task of monitoring switch continuously
        {
         if (SW = = 0)          // if SW = 0, give 25% power to DC motor
                {
                MOTOR_CTRL = 0; // turn ON motor (high portion of pulse)
                delay(25);      // 25 % ON time
                MOTOR_CTRL = 1; // turn OFF motor (low portion of pulse)
                delay(75);      // 75% OFF time
                }
        else                    // when SW = 1 give 50% power to DC motor
                {
```

```
                    MOTOR_CTRL = 0;      // turn ON motor (high portion of pulse)
                    delay(50);           // 50 % ON time
                    MOTOR_CTRL = 1;      // turn OFF motor (low portion of pulse)
                    delay(50);           // 50 % OFF time
                    }
            }
}
void delay(unsigned int duty)            // delay subroutine
            {
            unsigned char i;
            for(i = 0; i< (255 *duty; i++);
            }
```

Lab exercise Find out the frequency (or time period) of PWM waveform generated by the above program (Example 20.12).

PROJECT: DC MOTOR-SPEED-CONTROL SYSTEM

Problem Statement

Design an 8051 (89C51) based system to control the speed of a DC motor. The system should have the speed-control knob to change the speed of the DC motor.

Solution:

The speed of the motor should vary as we rotate the speed control knob, i.e. when the knob is at a fully anti-clockwise position, the speed of the motor should be 0. As we rotate the knob in clockwise direction; the speed should increase until we reach the extreme position. This type of control is achieved using the potentiometer when the voltage V_{MAX} is connected across it. The potentiometer will give us 0 V to V_{MAX} volts (5 V for our design) when we rotate the knob.

The signal from the potentiometer (analog signal) should be given to the ADC. The output of the ADC can be used to control the duty cycle of the PWM signal with the help of timers of the 8051. The PWM signal duty cycle should change from 0 to 100% when the speed-control knob is varied. The ADC0804 is used for this operation. The block diagram of the system is shown in Figure 20.15.

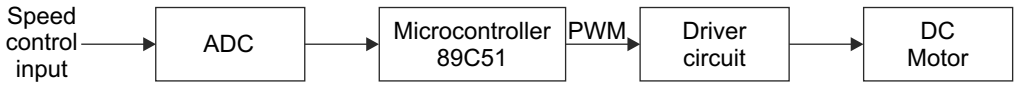


Fig. 20.15 *Block diagram of motor speed control system*

PWM Generation

To generate a required PWM signal, we have to use both timers of the 8051. One timer (timer 0) will be programmed in Mode 2 (for simplicity) to overflow (generate interrupt) at a fixed interval of time. This time should be chosen equal to the total time period (T_{ON} +T_{OFF}) of a required PWM signal. The other timer, (Timer 1), is used to control the duty cycle of the PWM. When Timer 0 overflows, stop Timer 0 and we should read the output of the ADC, load this value into Timer 1 registers and reload the count in Timer 0 again and start both the timers. When Timer 1 overflows, the motor is turned ON until the end of the PWM cycle (when Timer 0 overflows). This way, when the speed-control knob is at the minimum position, the analog voltage given to the ADC is 0 V and therefore, the digital output will be 00H, thus Count 00 is loaded in Timer 1, which will take the maximum time to overflow, and after that motor is turned ON for the minimum time (for the value of 00H input, when Timer 1 overflows, Timer 0 will also overflow, and the motor is not turned ON at all; therefore, duty cycle is 0%.). When the speed-control knob is at the maximum position, the analog voltage given to the ADC is +5 V, the digital output will be FFH and thus count FFH is loaded in Timer 1, which will take only one machine cycle to overflow and then the motor is made ON for the maximum time; therefore, the duty cycle will be almost 100% and motor will rotate at the maximum speed. (Alternatively, output of the ADC will be subtracted from FFH and the result

is loaded into Timer 1 register, motor is turned ON, start both the timers, and turn OFF the motor when Timer 1 overflows.)

The PWM wave form generated by this method is illustrated in Figure 20.16.

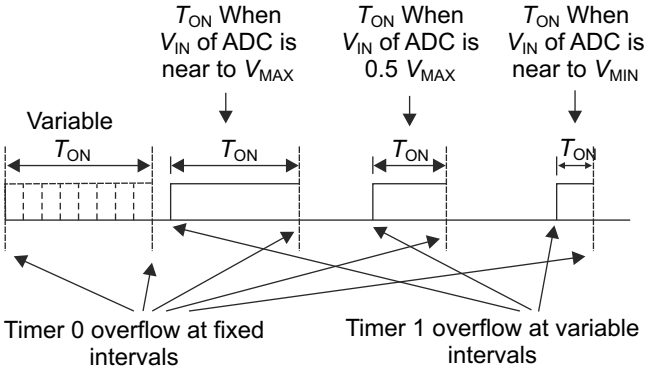


Fig. 20.16 *Generation of PWM signal by speed control input*

The complete circuit diagram of the motor-speed-control system is shown in Figure 20.17.

The potentiometer of 10 KΩ is used as the speed-control knob. Supply voltage (+5 V) is connected across it; therefore, it can give 0 to 5 V as an input to the ADC depending upon the position of the knob. $V_{REF}/2$ of ADC is kept open to select the analog input range as 0 to 5 V. Port 1 pins are connected with the data pins of the ADC and Port 3 pins are used as control signals of the ADC as shown in Figure 20.17. Port pin 2.0 is used to turn the motor ON or OFF. When P2.0 = 0, the transistor 2N2222 will be off and 12 V will be given to the gate of the MOSFET; therefore, the MOSFET will conduct and motor will be turned ON. Similarly, when P2.1 = 1, motor is turned OFF.

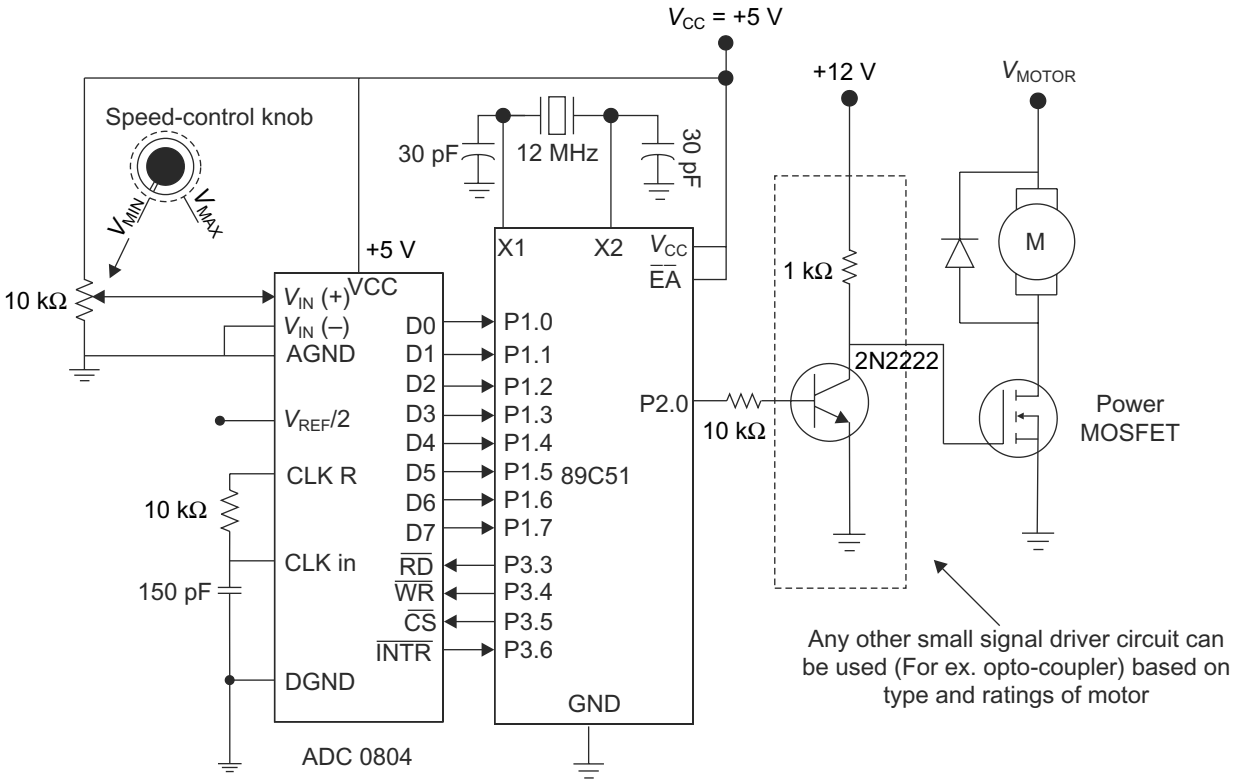


Fig. 20.17 *Motor-speed control system*

Note that P2.0 cannot be directly connected to the MOSFET gate because the voltage needed to drive power MOSFETs is around 10 V but the microcontroller 8051 can output only 5 V at port pins. (There are MOSFETs which can be driven with 5 V; for such cases, the microcontroller can be directly connected to the gate of the MOSFET.)

The driver circuit is selected based on the following parameters:

- Motor ratings (operating voltage, current ratings)
- Input requirements of a switching component (transistor, MOSFET, Op-Amp)
- Degree of isolation required between a control circuit and a load circuit

The steps to develop the program are:

- Configure both timers in Mode 2.
- Load TH0 with FFH, (to get the maximum period for PWM cycle).
- Start Timer 0.
- Monitor ADC input continuously in the main program.
- When Timer 0 overflows, execution will go to Timer 0 ISR.
- Timer 0 ISR will perform the following operations:
  - Turn OFF motor if it is made on by Timer 1 ISR
  - Stop Timer 0 (for synchronization)
  - Read ADC output and load it into TH1 register
  - Start both the timers again
  - Return to the main program
- When Timer 1 overflows, execution will go to Timer 0 ISR.
- Timer 1 ISR should perform the following operations:
  - Turn on motor
  - Stop Timer 1 (for synchronization)
  - Return to the main program

The assembly-language program is given below.

```
        ORG 0000H
        LJMP MAIN           // interrupt vector is bypassed by main program

        ORG 000BH           // ISR of Timer 0 at its vector address
        LJMP T0_ISR         // jump to Timer 0 ISR

        ORG 001BH           // ISR of Timer 1 at its vector address
        CLR P2.0            // turn ON motor (see Figure 20.17)
        CLR TR1             // stop Timer 1
        RETI                // return to the main program

        ORG 0100H           // main program start at address 100H
MAIN:   MOV TMOD, #22H      // both timers in Mode 2
        MOV IE, #8AH        // enable Timer 0 and 1 interrupts
        MOV TH0, #00H       // load count in Timer 0 register for width of PWM
        SETB TR0            // start Timer 0
        MOV P1, #0FFH       // configure P0 as input port
        SETB P3.6           // configure P3.6 as input to monitor the end of
                            // conversion (INTR pin)
        CLR P3.5            // make CS low for selecting ADC chip
AGAIN:  CLR P3.4            // make WR low
        SETB P3.4           // WR = 1, low-to-high pulse for starting ADC
HERE:   JB P3.6, HERE       // wait until the end of conversion
        CLR P3.3            // make RD low to read conversion result
        MOV A, P1           // read data from ADC
        MOV R4, A           // store result in register R4
        SETB P3.3           // make RD high before taking next sample
        SJMP AGAIN          // continuously monitor ADC result

T0_ISR: SETB P2.0           // turn OFF motor (see Figure 20.17)
        ANL TCON,#0AFH      // stop both timers, equivalent to CLR TR0
                            // and CLR TR1
        MOV TH1, R4         // read result of ADC and load into TH1
```

```
        ORL TCON, #50H      // start both the timers without affecting other bits
                            // equivalent to SETB TR0 and SETB TR1
        RETI
        END
```

Note that the above program works properly with ADC output values between 0FH and F1H, for extreme lower values (less than 0FH) and extreme high values (greater than F1H), timers will not be synchronized because of interrupt latency. This problem occurs because PWM is generated using the software.
This dictates the need of PWM generation using dedicated hardware like PWM module.

The above program can be written in the C language as

```
# include <reg51.h>
sbit MOTOR_CONTROL = P2^0;
sbit INTR = P3^6;
sbit CS = P3^5;
sbit WR1 = P3^4;
sbit RD1 = P3^3;

unsigned char AD_result;

void timer0 (void) interrupt 1      // ISR for Timer 0
      {
            MOTOR_CONTROL = 1;      // turn OFF motor (see Figure 20.17)
            TCON & = 0xAF;          // stop both the timers, equivalent to CLR TR0
                                    // and CLR TR1
            TH1 = AD_result;        // read result of ADC and load into TH1
            TCON | = 0x50;          // start both the timers without affecting other bits
                                    // equivalent to SETB TR0 and SETB TR1
            }

void timer1 (void) interrupt 3      // ISR for Timer 1

      {
      MOTOR_CONTROL = 0;            // turn ON motor (see Figure 20.17)
            TR1 = 0;                // stop Timer 1
      }

void main ()
      {
            TMOD = 0x22;            // both timers in Mode 2
            IE = 0x8A;              // enable Timer 0 and 1 interrupts
            TH0 = 0x00;             // load count in Timer 0 register for width of PWM
            TR0 = 1;                // start Timer 0
            P1 = 0xFF;              // configure P0 as input port
            INTR = 1;               // configure P3.6 as input to monitor the end of
                                    // conversion (INTR pin)
            CS = 0;                 // make CS low for selecting ADC chip
            while (1)               // continuously monitor ADC result
                  {
                  WR1 = 0;              // make WR low
                  WR1 = 1;              // WR = 1, low-to-high pulse for starting ADC
                  while (INTR = = 1);   // wait until the end of conversion
                  RD1 = 0;              // make RD low to read the conversion result
                  AD_result = P1        // read data from ADC
                  RD1 = 1;              // make RD high before taking the next sample
                  }
      }
```

Suggested Modifications

- Add direction control feature in the system.
- Use opto-coupler driver circuit.

PROJECT: AUTOMATIC STREET LIGHT CONTROL SYSTEM

Problem Statement

Design a simple 89C51 based system which will automatically switch ON a bulb (of street light) at night and switch OFF the bulb during daytime.

Solution For the desired operation, we should have a light-sensing component. LDR (Light Dependent Resistor) is the most common light-sensing element. LDRs resistance varies from mega ohms for darkness to a few hundred ohms for bright light. LDR can be used in voltage comparator circuits which produces high (or low) output when the light intensity falls below a certain level.

Atmel 89C2051 have an inbuilt precision analog comparator circuit (Op-Amp based). Pin P1.0 is +ve input and pin P1.1 is –ve input of a comparator, pin P3.6 is the output of the comparator that can also be accessed through the software (similar to any other port latch). The advantage of using the internal analog comparator is that it saves PCB space (because of component reduction) and the designer is relieved from the comparator circuit design). Moreover, 89C2051 is low cost and only 20-pin chip which further reduces the size and cost of final PCB (board).

Figure 20.18 shows the complete interfacing diagram for a desired system. The Op-Amp is shown using dashed lines and within the 89C2051 block to show that it is internal to the 89C2051. Note that the power-on reset and crystal connections are not shown for simplicity.

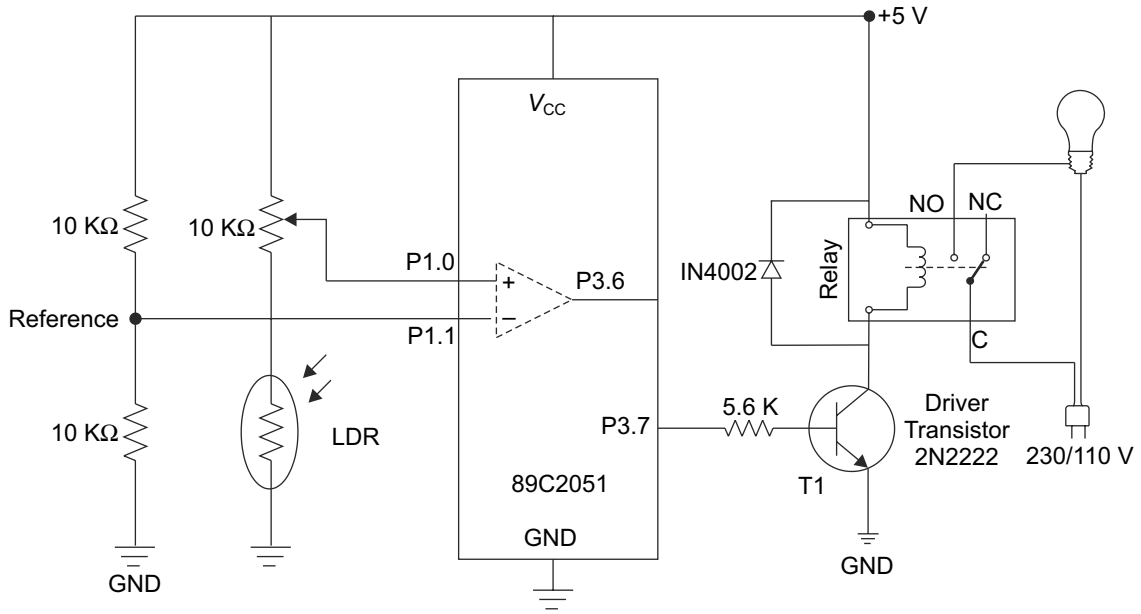


Fig. 20.18 *Automatic street light control system*

Circuit Operation

Two voltage-divider networks are used as an input to the comparator. The reference voltage is fixed to 2.5 V approx. and given to inverting terminal of the Op-Amp. At night time (or for very low light conditions), the resistance of LDR is very high (MΩ); therefore, the maximum voltage is dropped across it (nearer to 5 V) and thus, the voltage applied at the non-inverting terminal of the Op-Amp is higher than the inverting terminal, hence output is high (1 V) at the port pin P3.6 and corresponding latch. During the daytime (or for higher light conditions), the resistance of LDR is very low; therefore, less voltage will drop across it. Thus, the voltage applied at the non-inverting terminal is less than that of the inverting

terminal; hence output of the comparator is low (0 V) at the port pin P3.6. The status of P3.6 latch can be monitored using the instruction JB (or JNB). If P3.6 is high, we will set P3.7, which will turn ON the driver transistor, which in turn will energize the relay and the bulb will be turned ON. During daytime, P3.6 = 0, will clear P3.7 and the relay will be de-energized and the bulb will be turned OFF. A potentiometer is used in series with LDR to set the level of light at which the circuit should turn ON or turn OFF the bulb. Note that the reference voltage should be stable for proper operation of the system. The professional version of the circuit may introduce hysteresis loop in the comparator part to have better stability of the circuit output!!

The steps to develop program for the system are the following:

- Read the status of comparator output (P3.6).
- If P3.6 = 1, turn on the light.
- Else, turn off the light.
- Repeat the above steps continuously.

```
          ORG 0000H
REPEAT:   JB P3.6, TURN_ON    //if comparator output is high (at night) turn ON relay
          CLR P3.7            //else, comparator output is low (at day time), turn OFF
          SJMP REPEAT         //relay
TURN_ON:  SETB P3.7           //turn ON relay
          SJMP REPEAT         //monitor the light intensity continuously.
          END
```

If the microcontroller is required to perform the other tasks (for other applications), the status of P3.6 can be monitored periodically.

Note that the same system may also be designed without using a microcontroller, but the intention of using a microcontroller in this application is to demonstrate how the microcontroller can be used for light-sensing applications as well as how they can be used to control high-power circuits.

THINK BOX 20.2

When should we prefer to use 20-pin microcontrollers like 89C2051 or 89C4051 in a system?

When PCB space available for the product (or size of the product) is small, we should use low pin-count microcontrollers, provided that the need of I/O lines are 15 or less and the program size is less than or equal to 2Kbytes (4Kbytes for 89C4951).

POINTS TO REMEMBER

- Relays and opto-couplers are used to electrically isolate the two systems operating with different power levels.
- Relays have generally three leads, Common (C), Normally Closed (NC), Normally Open (NO) along with the two leads for excitation.
- A freewheeling diode is connected in parallel to the coil to prevent appearance of high voltage of self-induction caused by a sudden stop of the current flow through the coil.
- Drawbacks of the electromechanical relays are a low switching frequency and lower life because of wear of the mechanical parts.
- Since there is no mechanical component, the opto-coupler can operate at a high switching frequency.
- The LED and phototransistor in an opto-coupler must be separately powered to get perfect isolation.
- Stepper motors are well suited for the position control applications.
- In unipolar stepper motors, the current always flows in only one direction through each phase; therefore, the name unipolar.

- The advantage of bipolar motor is that they produce more torque because all the phases are utilized simultaneously. The disadvantage is that they require more complicated driver circuits because polarities of both the ends of coils are changed simultaneously.
- The direction of the rotation of the stepper motor can be changed by reversing the sequence of pulses.
- The speed of the rotation of stepper motor depends on the frequency of pulses given by a microcontroller.

OBJECTIVE QUESTIONS

1. When a relay is energized,
   (a) NC and common leads are disconnected (b) NO and common leads are connected
   (c) NO and NC are connected (d) NO and common leads are disconnected
2. The step angle of the stepper motor depends on,
   (a) teeth in the rotor
   (b) poles in the stator
   (c) teeth in the rotor as well as poles in the stator
   (d) none of the above
3. The direction of the rotation of a stepper motor depends upon,
   (a) voltage polarity (b) pulse sequence (c) current direction (d) none of the above
4. The speed of rotation of stepper motor depends upon,
   (a) frequency of input pulses (b) polarity of the pulses
   (c) input voltage of pulses (d) all of the above
5. Generally, the full-step angle of a stepper motor is taken as,
   (a) 3.6° (b) 1.8° (c) 0.9° (d) none of the above
6. If the stepper motor has four electromagnets as stator and five teeth in its rotor then the angle of rotation for the full step would be,
   (a) 72° (b) 18° (c) 9° (d) none of the above
7. Speed of the dc motor depends on,
   (a) applied voltage (b) load connected (c) duty cycle of PWM (d) all of the above
8. A stepper motor has a step angle of 2°, the number of pulses required to rotate the motor by 360° is,
   (a) 1 (b) 4 (c) 180 (d) 360
9. The isolation voltage for IL74 is,
   (a) 5000 V_{rms} (b) 5300 V_{rms} (c) 2500 V_{rms} (d) 1500 V_{rms}
10. ILD74 is___ channel phototransistor base opto-couler.
    (a) 1 (b) 2 (c) 4 (d) 8

Answers to Objective Questions

1. (b)	2. (c)	3. (b)	4. (a)	5. (b)
6. (b)	7. (d)	8. (c)	9. (b)	10. (b)

REVIEW QUESTIONS WITH ANSWERS

1. Why are relays required to be used in the circuits?

A. To isolate the two circuits operating at different power levels. This will provide protection to controlling circuits.

2. Why are electromechanical relays and solid-state relays named so?

A. An electromechanical relay is a combination of electric (coil) and mechanical (switch and spring) components. Solid-state relays are completely made from semiconductor materials.

3. What are the advantages of SSR over EMR?

A. High switching frequency, more reliable, more life.

4. Why is the driver circuit required to drive a relay coil?

A. Microcontrollers are not capable of sourcing sufficient currents to drive the coil.

5. Why do electromechanical relays have lower switching frequency?

A. Because of inertia of the mechanical components involved.

6. Do all the relays have NC lead?

A. No. Some may have only NO lead.

7. What is a freewheeling diode? Why is it used?

A. It is connected in anti-parallel to the coil to prevent self-induced high voltage caused by a sudden stop of current flow through the coil. They protect driver circuits.

8. What is the advantage of using stepper motors?

A. Rotor position or angular rotation can be controlled without any feedback. This reduces the complexity of a system.

9. How are stepper motors most suitable for position-control applications?

A. Rotation is precisely controlled by the number of pulses applied, and the number of pulses is controlled by a microcontroller. Hence, angular rotation can be controlled accurately by the microcontrollers, which is the basic requirement of the position control applications.

10. Can we use the unipolar motor as a bipolar motor? What about reverse?

A. Yes, do not use the common leads. The bipolar motors do not have a centre tap connection, so they cannot be used as a unipolar motor.

11. Why are unipolar and bipolar stepper motors named so?

A. In unipolar motor phase, the current flows only in one direction, while in the case of bipolar motors, current flows in both the directions.

12. What are the advantage and disadvantages of half-stepping?

A. The advantage is it increases the resolution. The disadvantage is that it produces less torque.

13. Why is the inverted sequence generated by microcontrollers rather than the normal sequence?

A. Because driver circuits normally inverts the input signals.

14. If a stepper motor takes 90 steps to make one revolution, what is the step angle for this motor?

A. 4 degrees.

15. How do we change a dc motor's rotation direction?

A. By reversing the polarity of supply voltage.

EXERCISE

1. What is an H-Bridge? Why is it used?
2. Write a program in the assembly language to rotate a unipolar motor in clockwise direction using half-step sequence. What modification is required in a program to rotate the motor in a reverse direction? Refer Figure 20.8.
3. Rewrite the above programs in the C language.
4. Discuss the significance of providing delay between two steps to rotate a motor.
5. Design a driver circuit for relay coil which requires a current of 20 mA and a voltage between 3.5 V to 5.5 V.
6. What are the factors affecting the step angle of a stepper motor?
7. List the advantages, disadvantages and applications of stepper motors.
8. Compare all types of sequences to drive the stepper motors with respect to torque produced.
9. Draw interfacing diagram to connect the microcontroller 8051 with bipolar PM stepper motor through the driver circuit made from discrete transistor and diode pairs. Also explain operation of all the steps of rotation using neat diagrams. Assume that the two switches S1 and S2 are connected to port pins P1.0 and P1.1. Write a program in assembly language to rotate the motor in a clockwise direction when switch S1 is pressed and in anticlockwise direction when S2 is pressed. The motor should be in idle condition when none of the switches is pressed.

21

Interfacing External Memory and Real-Time Clock

Objectives

- Discuss the input and output signals of the memory chips
- List and discuss the types of semiconductor memory devices
- Discuss the address-decoding techniques
- List and discuss the signals of the 8051 used in memory interfacing
- Explain the code and data memory interfacing with the 8051
- Show how the program memory can be used as data memory and vice versa
- Illustrate how ROM or RAM can be used as data as well as program memory
- Discuss the need of real-time clocks
- Introduce DS12887 RTC chip and discuss the function of each pin of a chip
- Interface the DS12887 RTC chip with the 8051
- Develop the programs for setting time, calendar and alarms

Key Terms

- Address Decoding
- Address Latch Enable: ALE
- Address Lines
- Address Range
- Chip Select/Enable: $\overline{CS}$, $\overline{CE}$
- Control Signals
- Data Lines
- Decoders
- DRAM
- DS12887
- EEPROM
- EPROM
- External Access: $\overline{EA}$
- Flash Memory
- Latch
- Memory Map
- Non-Volatile Memory: ROM
- NV-RAM
- On-Chip Memory
- Oscillator Control
- Output Enable: $\overline{OE}$
- Periodic Interrupts
- Program Store Enable: $\overline{PSEN}$
- PROM: OTP
- Read/Write: $\overline{RD}$, $\overline{WR}$
- Real-Time Clock
- SRAM
- Time, Calendar and Alarms
- Update Cycle
- Volatile Memory: RAM

Every microcontroller/processor based system has a memory subsystem, either on-chip or off-chip. The memories are broadly classified into two categories: Volatile and Non-volatile memory. The volatile memory, as the name suggests, loses its contents when the power is removed; it is usually referred as Random Access Memory (RAM) or Read/Write memory. It contains temporary data. Non-volatile memory retains its contents even when the power is removed; it is referred as Read Only Memory (ROM). It contains the programs (system software) and permanent data. This chapter explains how to interface both types of memories to the 8051 family of microcontrollers. The principles discussed in this chapter allow virtually any microcontroller/processor to be interfaced to any memory system.

THINK BOX 21.1

Why is the RAM named so?

During the early days of computer development, there were two types of volatile memory: First, which could only be accessed sequentially and the second, where any address can be accessed directly. The second type of memory was referred as 'Random Access Memory'; to reveal the fact that any 'random' address could be accessed at any time. The former type of memory is not commonly used any more, but the term RAM has remained in use till date.

21.1 | MEMORY INTERFACING AND ITS NEED

Memory Interfacing

To connect the memory chips with a microcontroller using logic circuits such that the microcontroller can communicate (read/write) with them.

When the available on-chip RAM and ROM are not sufficient, we need to add an external memory to the system. For example, when the programs are developed in a high-level language, the program size may exceed the available on-chip program memory and we need to add ROM externally or when the 8051 microcontroller is used in a data acquisition system, internal RAM will not be sufficient and external RAM is added. In general, for the larger software, we require to interface extra memory. However, it should be noted that these days, versions of the 8051 are available which contain up to 64 Kbytes of on-chip program memory. I have included this topic in the text because a system designer must be aware of all the aspects of system design including fundamental concepts of memory interfacing and all types of memories used in the embedded systems.

21.2 | MEMORY CHIPS

All the memory devices have a common set of input and output signals like address, data and control signals. A typical memory module with these signals is shown in Figure 21.1.

21.2.1 Address Signals

Address-input signals (address lines) are used to identify one memory location out of *N* locations. Address inputs are usually labeled from A_0 (LSB of address) to A_N, where *N* depends on the total memory locations in a memory module, i.e. memory capacity. For example, a memory chip that has 16 address pins has its address signals (pins) labeled from A_0 to A_{15} (note that the first address pin is labeled from 0, not from 1). The number of address pins required for a memory module (chip) is decided by the number of memory locations present within it. The group of address lines is referred more commonly as *address bus*.

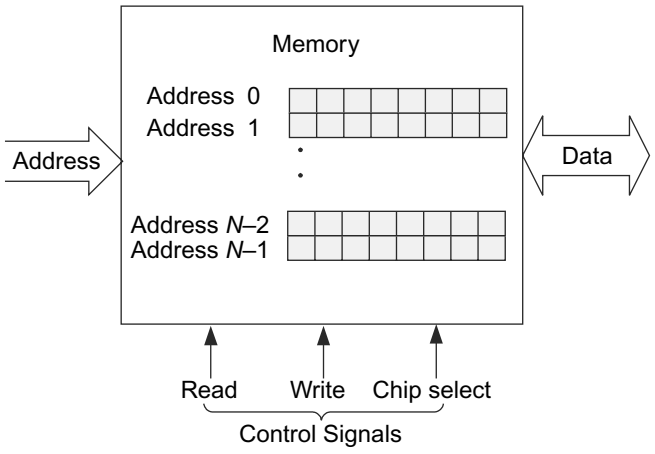


Fig. 21.1 *Simplified memory module*

If there are *N* address lines, then there are 2^N different addresses (0 to 2^N-1) in a memory chip, for example, 16-bit address lines indicates that there is 2^{16} = 65536 bytes* of memory having addresses from 0 to 65535, i.e. 0000H to FFFFH. Conversely, 1 Kbyte memory device has 10 (1K = 1024 = 2^{10}) address lines. Memory chips are usually referred by their capacity in kilobytes or megabytes. The more common memory chips have capacity between 1 Kbytes to 256 Mbytes.

21.2.2 Data Signals

The data signals (or data lines) are used to transfer the data between memory and the CPU. They are the lines through which the data are written to (or read from) the memory chip. Since the data transfer can take place in both the directions, they are bidirectional. Data pins on memory devices are usually labeled D_0 to D_7 for an 8-bit memory chip. An 8-bit wide memory device is usually referred as a byte wide memory. Though many memory chips are 8 bits wide, not all memory needs to be 8 bits wide. A memory chip with 4K memory locations and 8 bits in each location is usually represented as a 4K X 8 memory chip. The group of data lines is referred more commonly as *data bus.*

21.2.3 Control Signals

Each memory chip has one or more input signals that select or enable the memory chip. These signals are usually referred as a Chip Enable ($\overline{CE}$) or Chip Select ($\overline{CS}$). If there is a bar over these signals, as mentioned, it indicates that they are active low signals and the memory chip can be read or written to when this signal is at logic level 0. If more than one chip select signals are present on a chip, all must be activated to enable the chip to access the data.

All memory chips have other control inputs which control the direction of flow of data. A ROM usually has only one such control input referred as Output Enable $\overline{OE}$, which brings the data on the data pins from internal memory cells. A RAM has one or two control inputs referred as R/ $\overline{W}$, $\overline{RD}$ and $\overline{WR}$ or $\overline{WE}$ (or $\overline{W}$) and $\overline{OE}$. This signal selects a read or a write operation. R/ $\overline{W}$ = 0 or $\overline{WE}$ = 0 selects a write and R/$\overline{W}$ = 1 or $\overline{OE}$ = 0 selects a read operation.

21.3 | SEMICONDUCTOR MEMORY DEVICES

Before we discuss how to interface memory with the microcontroller, it is essential to briefly understand the operation of memory components. In this section, classification and function of semiconductor memories is discussed. RAM can be static or dynamic, and nonvolatile memory is available as ROM, PROM, EPROM, EEPROM, FLASH, and NVRAM. All types of semiconductor memories are shown in Figure 21.2.

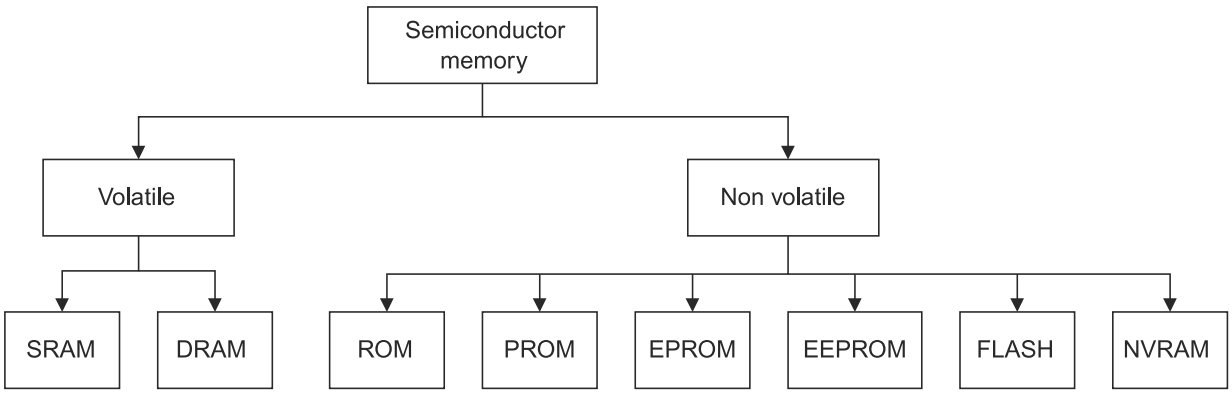


Fig. 21.2 *Semiconductor memories*

21.3.1 Volatile Memory

These memories lose their contents when the power is removed; they are used to store temporary data/results while processing. They can be written directly by the program operations (instructions). Static RAM and dynamic RAM are types of a volatile memory.

* A memory location usually contains 8 bits—a byte.

SRAM: Static RAM

A SRAM chip is made from an array of cells, where each cell can store one bit of information. A cell is made from flip-flop. The flip-flops consists of six (or four) transistors. The basic block of a SRAM cell is shown in Figure 21.3 (a). Since these devices require only power supply to retain the data, they are also referred as static memory. Address line is used to select the cell while the data bit is written to (or read from) the cell using a data bit line. A large number of such cells is arranged in matrix form to get bigger storage capacity. The 61xx family of memory chips is SRAM, where xx in the chip number represent capacity in Kbits, for example, 6116 has 16 Kbits (organized as 2K × 8) bits.

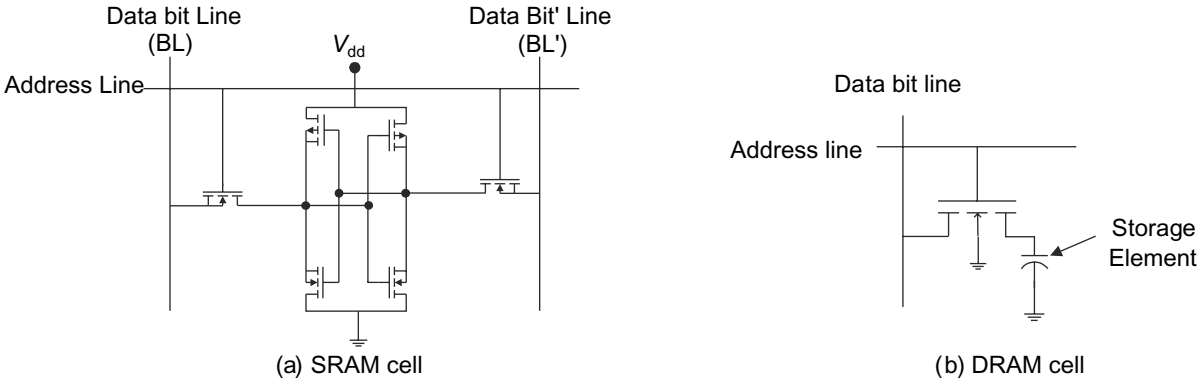


Fig. 21.3 *SRAM and DRAM memory cells*

DRAM: Dynamic RAM

In DRAM cell, one bit of information is stored using a capacitor as shown in Figure 21.3 (b). Along with the capacitor, one transistor is used to enable the cell for read/write operations. The less number of components required to implement a storage element saves space (silicon area) to implement a DRAM cell; therefore, DRAM has a much higher storage density compared to SRAM (for equal silicon area). It is around four times denser than SRAM for the same chip size (and cost). To store a logic high (1), the cell is selected by an address line (which will select the transistor) and then voltage is applied at the data bit line, which will charge the capacitor, this charged capacitor represents logic high. Similarly, to store a logic low (0), the cell is selected and the capacitor is discharged and to retrieve the stored information, select the cell and read the status of capacitor, i.e. charged capacitor represents 1 and discharged capacitor represents 0.

The disadvantage of DRAM is that the capacitor retains data for a very small time (usually 2 or 4 ms). After every 2 or 4 ms, the contents of the DRAM must be rewritten (usually referred as refreshing) because the capacitors lose their charges. Refreshing circuits are implemented within DRAM chip to preserve the data for a longer time. Another disadvantage of DRAM is that they are slower compared to SRAM. Because of the larger capacity, DRAMs require more number of address pins. To save the number of address of pins, they are multiplexed with respect to rows and columns. The 41xx family of memory chips is DRAM, where xx in the chip number represents the capacity in Kbits, for example, 4164 has 64Kbits (64K × 1) bits.

21.3.2 Nonvolatile Memory

As opposed to RAM, nonvolatile memories preserve the data even if the power is removed. They are used to store programs and permanent data. The disadvantage of these memories is that writing data into them is much slower as they require external writing hardware (programmers) or special support on a chip for this purpose. The nonvolatile memory is available in many forms today as discussed below.

ROM

Read Only Memories (ROMs) are the most basic types of nonvolatile semiconductor memories. As the name suggests, the program instructions cannot write to a ROM. The ROM is written (programmed) during its fabrication by the manufacturer. We have to give data to be written to ROM to the chip manufacturer, where a specific chip is fabricated containing our data.

Mask ROM is the common type of ROM made from a matrix of memory cells. A mask (chip fabrication terminology) is used to create the connections between rows and column as per the data to be programmed. The basic ROM cell is shown in Figure 21.4.

The ROM is used in the final production version when all the program code has been fully tested. They are economical for mass production.

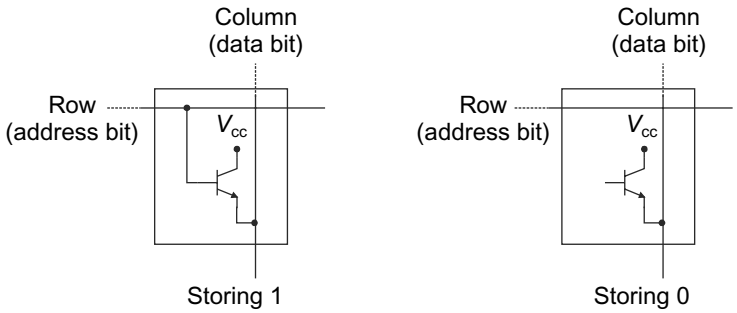


Fig. 21.4 *ROM cell*

PROM: Programmable ROM

PROMs are also made from a matrix of memory cells; each cell contains a programmable link (silicon fuse). In an unprogrammed PROM chip (a new chip), all the fuses are connected as shown in Figure 21.5 and, therefore, each cell stores Logic 1. To program logic 0 in a cell, the cell is selected and high current pulse is applied which will burn (damage) the fuse; this will give the output 0 when a cell is selected. The advantage is that they can be programmed by the user, but once programmed, it cannot be erased. Because of this, they are also referred as *One Time Programmable (OTP)* memory. PROMs are not commonly used today. The PROM cell is shown in Figure 21.5.

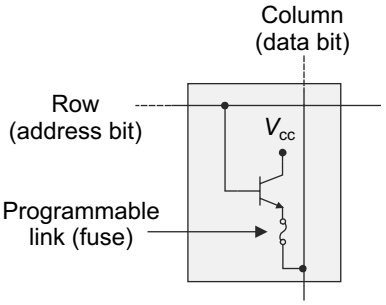


Fig. 21.5 *PROM cell*

EPROM: Erasable Programmable ROM

The key feature of EPROM is that it is user-programmable and erasable. The contents of the memory can be erased from the memory by exposing the memory chip to the ultraviolet radiation for a shorter period of time. Therefore, it can be used many times. They are made using FETs and the data is programmed by charging (injecting electrons) the gate of this FET through the process called *Avalanche Injection*.

By exposing the EPROM chip to ultraviolet light for about 20 minutes, the gates are discharged and the EPROM is erased. To support the erasure process, the EPROMs have a smaller glass window in their package, through which UV light can fall on the chip. Once the chip is programmed, this window is covered by a lightproof seal. The problem with this memory is that the programming and erasing is complex because a special programming voltage is required, which is usually higher than the operating voltage, and to erase it, the UV light source is required, and this process is time consuming as well. The 27xx family of chips are EPROMs, where xx in the chip number represents capacity in Kbits, for example, 2732 has 32Kbits (4K × 8) bits.

EEPROM: Electrically Erasable and Programmable ROM

The problems of EPROMs are removed in the EEPROM while all the advantages are maintained, i.e. separate voltage is not required for programming and no UV light source is required for erasing. EEPROM are similar to EPROM, except that they are erased electrically (using voltage). Each byte in the EEPROM can be individually programmed as well as erased. They are commonly used during the development process. They are more commonly referred as E^2PROM (e squared PROM). The 28xx family of chips is EEPROMs, where xx in the chip number represents capacity in Kbits, for example, 2864 has 62Kbits capacity (8K × 8) bits.

Flash

Flash memories are similar to EEPROM. The only difference is that the erasing is not possible for each byte (address) individually, but only for larger blocks or the entire memory, i.e. an entire chip can be erased 'in a flash', and hence, they are named so. Since the bytes cannot be programmed individually, the internal circuit is simple, which makes these memories cheaper. The 28Fxx family of chips is flash memory chips.

NVRAM: Nonvolatile RAM

It uses the good properties of volatile as well as nonvolatile memories; they have SRAM which is backed up by a small internal battery. Normally, the external power is used for the circuit operations, while in the absence of external power, the internal battery controls the operations of the memory, this way SRAM contents are preserved. DS1220 (2K×8), DS1225 (8K × 8) are NVRAM chips.

21.4 MEMORY MAP AND ADDRESS DECODING

In a system, there may be many memory chips, each with a unique address range. The assignment of addresses to the memory locations in the various memory chips present in the system is referred as *memory map* of the system. The microcontroller issues a specific address on its address bus to read (or write) data from memory chip. The process of selecting the correct memory location from the correct memory chip is known as *address decoding*.

When we interface a memory chip with a microcontroller, the number of address lines of a memory chip and a microcontroller need not be always the same, for example, when 8Kbyte memory is interfaced, it will have 13 address lines, while the 8051 have 16 address lines. The 8051 microcontroller always issues the 16-bit address, so this 16-bit address should be understood by the memory chip which has only 13 address lines. The memory address decoding circuit will map these 16-bit addresses to 13-bit addresses so that the proper data transfer can take place between microcontroller and memory chip. The address decoder will resolve the problem of address-line mismatch. Moreover, more than one memory chip may be present in a system at a time, and only 1 chip (as per the address issued by the microcontroller) should be selected (enabled) for data transfer and all the other should be disabled. This selection of proper memory chip is also done by the address decoder circuit. Moreover, without the address decoder, only 1 memory chip up to 64 Kbyte can be connected in a system (8051 based) which will be always selected irrespective of the address issued by a microcontroller. To summarize, the address decoder will

- Resolve the problem of the number of address lines mismatch between memory chip and a microcontroller
- Select the appropriate memory chip out of the many chips to locate required data byte
- Allow more than one memory chips to be connected in a system

There are two simple ways to implement the address-decoding circuits. They can be implemented using either,

1. Discrete logic gates or
2. Decoders.

Yet, there is another relatively complex approach to design the decoding circuits; it uses PLDs (Programmable Logic Devices). It is generally not used with the 8051 and, therefore, not discussed.

THINK BOX 21.2

Why do we represent 65536 bytes as 64K rather than 65K?
Because 64 = 2^6, 64K (64000) is considered as the round number closest to 65536.

21.4.1 Signals used in Memory Interfacing

Before we start designing the address decoder circuits, we should have a clear understanding of signals involved in the memory interfacing from both microcontroller as well as the memory chip; therefore, it is discussed first.

21.4.2 The 8051 and the Corresponding Memory Chip Signals

Address Lines (A_{15}–A_0)

The 8051 has 16 address lines. Port 0 (A_7–A_0) and Port 2(A_{15}–A_8) provides these signals. Depending upon the size of the memory chip, the lower address lines of the microcontroller are connected directly to the address lines of a memory chip, for example, if size of the memory chip is 1 Kbyte, it will have 10 address lines A_9–A_0. Therefore, the lower 10 address lines (A_9–A_0) of the microcontroller is connected with the corresponding address lines of the memory chip. Remaining upper address lines (A_{15}– A_{10}) are used as an input to the address decoder to generate chip select signal for the memory chip.

Data Lines (D_7 – D_0)

Port 0 provides time multiplexed data and lower 8 address signals. The data lines of the microcontroller are directly connected to the data lines of a memory chip.

It is assumed that the data lines and lower 8 address lines are demultiplexed before being connected to the memory chip. See topic 11.2 (subtopic: Address/Data demultiplexing using ALE) and Figure 11.4 for a detailed description of how to demultiplex address and data lines.

$\overline{\text{PSEN}}$ Program Store Enable ($\overline{\text{PSEN}}$) is used to activate (enable) the external ROM chips. Thus, this signal acts as read strobe (or output enable) to the external program memory. It should be connected to $\overline{\text{OE}}$ (Output Enable) pin of the ROM chip. It is activated while reading only the external program memory. When there is only a single ROM chip on a system, $\overline{\text{PSEN}}$ may also be connected to $\overline{\text{CS}}$ (Chip Select) signal to simplify the interfacing circuit. Note that in a special case, $\overline{\text{PSEN}}$ may be connected to $\overline{\text{OE}}$ of RAM chip, when RAM is used as a code memory.

$\overline{\text{RD}}$ The Read signal is activated by the microcontroller to read data from the external RAM. It is connected with $\overline{\text{OE}}$ of the RAM chip. It is activated by the instructions like MOVX A, @DPTR or MOVX A, *@Ri* which reads external RAM. However, $\overline{\text{RD}}$ may be connected to $\overline{\text{OE}}$ of ROM chip, when ROM is used as a data memory.

$\overline{\text{WR}}$ The Write signal is activated while writing data to a RAM. It is usually connected with $\overline{\text{WE}}$ (Write Enable) pin of the RAM chip. Instructions MOVX *@Ri*, A or MOVX @DPTR, A will activate this signal.

$\overline{\text{EA}}$ It is used to select on-chip or off-chip memory. Refer topic 11.2 for a detailed description of this pin.

The connection of the 8051 signals with the corresponding signals of the memory chips is summarized in Table 21.1.

Table 21.1 *Connections between 8051 and the memory chips*

The signals of 8051		to be connected with	
		RAM chip	ROM chip
$\overline{\text{RD}}$		$\overline{\text{OE}}$	$\overline{\text{OE}}$ (only when used as data memory)
$\overline{\text{WR}}$		$\overline{\text{WE}}$	–
$\overline{\text{PSEN}}$		$\overline{\text{OE}}$ (only when used as code memory)	$\overline{\text{OE}}$, may also be connected to $\overline{\text{CS}}$ when only one chip is in the system
Address lines	Lower (A_n– A_0)	A_n– A_0 ; n depends on the size of memory chip	A_n –A_0 ; *n* depends on the size of memory chip
	Higher (A_{15} – A_{n+1})	To address decoder to generate $\overline{\text{CS}}$ or $\overline{\text{CE}}$	To address decoder to generate $\overline{\text{CS}}$ or $\overline{\text{CE}}$
D_7 – D_0		D_7 – D_0	D_7 – D_0

Example 21.1

What will be the values on the address pins of the microcontroller if the data from external RAM address 5700H is to be accessed?

Solution:

To access data from the external RAM address 5700 H, address pins A_{15} – A_8 will be 0101 0111 (57H) while the address pins A_7 – A_0 will be 0000000 (00H).

21.4.3 Address Decoder using Logic Gates

The memory chip containing the desired address is selected using a simple combinational circuit using gates. The address decoding using the logic gates is best explained by Example 21.2.

Example 21.2

Interface two 4K x 8 bits RAM memory chips consequently from address 0000H onwards. (Note: 4K x 8 bits RAM is taken only for the illustration).

Solution:

To address 4 Kbytes of memory, 12 address lines (2^{12} = 4096) will be required. The address range for both the chips will be calculated as follows:

Address lines															
A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
The address range of chip 1 – 0000H to 0FFFH															
0	***0***	***0***	***0***	0	0	0	0	0	0	0	0	0	0	0	0
0	***0***	***0***	***0***	1	1	1	1	1	1	1	1	1	1	1	1
The address range of chip 2 – 1000H to 1FFFH															
0	***0***	***0***	***1***	0	0	0	0	0	0	0	0	0	0	0	0
0	***0***	***0***	***1***	1	1	1	1	1	1	1	1	1	1	1	1

12 address lines (A_{11}–A_0) will be connected to the address lines of both the memory chips to locate 1 byte out of 4096 bytes. The remaining four address lines (A_{15}–A_{12}) remains constant throughout the address range of each chip and therefore should be used to select a chip, i.e. to generate Chip Select (usually $\overline{CS}$) signal for each chip. The $\overline{CS}$ for the first chip should be activated ($\overline{CS1} = 0$) when A_{15}–A_{12} are 0000H (see the table of address range of each chip). In the same way, $\overline{CS}$ for the second chip should be activated ($\overline{CS2} = 0$) when A_{15}–A_{12} are 0001. It can be done as shown in Figure 21.6.

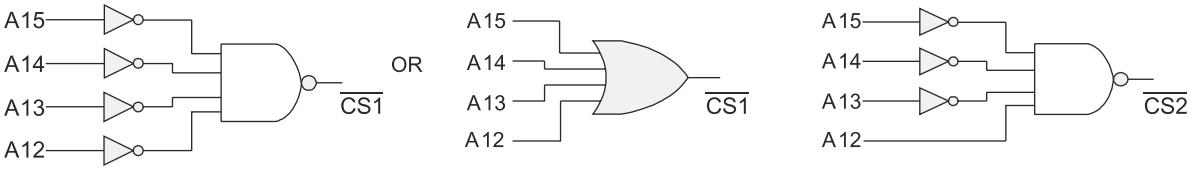


Fig. 21.6 *Generation of Chip select signals using gates*

Complete interfacing can be done as shown in Figure 21.7 (address and data lines are demultiplexed).

Note that the data lines and address lines (A_{11}–A_0) of a microcontroller are connected with data lines and address lines (A_{11}–A_0) of both the memory chips simultaneously, but based on the desired address, only one chip will be activated, therefore, data will be transferred to/from the correct chip without any confusion and bus contention. The chip which is not selected will behave as if it is not physically connected.

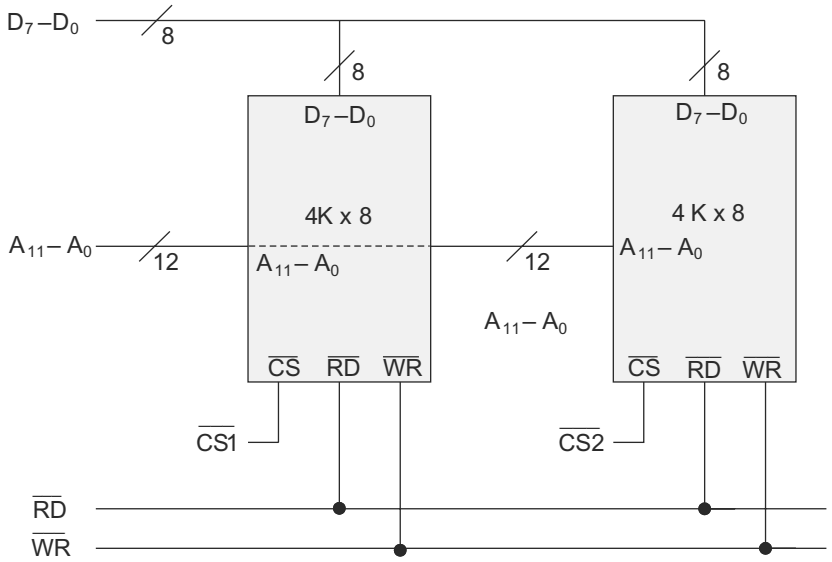


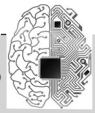
Fig. 21.7 *Interfacing two 4K × 8 bits RAM memory chips*

Memory Read and Write Operations

To read the data from the memory chip, the corresponding address is placed on the address bus. The lower address lines are connected to the address pins of the memory chip and higher address lines are connected to the address decoder to select the appropriate chip as discussed in the above section. The selected chip will get data from the specified location and to read that data, the microcontroller will generate $\overline{RD}$ signal and data will be transferred to the microcontroller through the data bus. For writing the data, address of the memory location is placed on the address bus and thereafter data to be written is placed on the data bus and $\overline{WR}$ signal is generated which will write the data present on the data bus into the desired memory location. (Note that lower address and data bus has to be demultiplexed).

21.4.4 Address Decoder using Decoder Chip

A decoder is a combinational circuit that selects (or activate) one output line depending upon the binary code at the input. *n* inputs are used to activate one out of *m* ($m = 2^n$) outputs. The 8205 and 74LS138 are 3 to 8 decoders. The 74LS138 has three input lines and eight active low output lines. It has three enable signals, one active high (G1) and two active low ($\overline{G2A}$ and $\overline{G2B}$), all the three signals must be enabled for its operation, i.e. G1 = 1, $\overline{G2A} = \overline{G2B} = 0$.



THINK BOX 21.3

What address range corresponds to 1001 XXXX XXXX XXXX?
9000H to 9FFFH

The 74LS138 can be used to connect up to eight memory chips in a system. The use of 74LS138 is illustrated in Example 21.3.

Example 21.3

Interface three 8K × 8-bit memory chips consecutively from the address 2000H onwards.

Solution:

The 8 K byte memory chip requires 13 address lines (A_{12}–A_0), since the starting address of the first chip is 2000H, the address range of each chip is calculated as follows:

Address lines															
A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
The address range of chip 1 – 2000H to 3FFFH															
0	***0***	***1***	0	0	0	0	0	0	0	0	0	0	0	0	0
0	***0***	***1***	1	1	1	1	1	1	1	1	1	1	1	1	1
The address range of chip 2 – 4000H to 5FFFH															
0	***1***	***0***	0	0	0	0	0	0	0	0	0	0	0	0	0
0	***1***	***0***	1	1	1	1	1	1	1	1	1	1	1	1	1
The address range of chip 3 – 6000H to 7FFFH															
0	***1***	***1***	0	0	0	0	0	0	0	0	0	0	0	0	0
0	***1***	***1***	1	1	1	1	1	1	1	1	1	1	1	1	1

The lower 13 address lines (A_{12}–A_0) of microcontroller are connected with address lines of each memory chip and higher address lines (A_{15}–A_{13}) are given to the decoder input to select the appropriate chip. Complete interfacing is shown in Figure 21.8.

Note that a decoder chip is always selected, i.e. G1 = V_{CC}, $\overline{G2A}$ = $\overline{G2B}$ = GND. Outputs Y1, Y2 and Y3 are connected to $\overline{CS}$ of each memory chip. D_7 – D_0 of the microcontroller is connected to D_7 – D_0 of all the memory chips.

Note that Chip 1 is enabled when the address is between 2000H to 3FFFH ($A_{15}A_{14}A_{13}$ = 001; therefore Y1 is activated, Y1 = 0). Chip 2 is selected for address range 4000H to 5FFFH ($A_{15}A_{14}A_{13}$ = 010) and Chip 3 is selected for address range 6000H to 7FFFH ($A_{15}A_{14}A_{13}$ = 011).

Note: V_{CC} and Gnd connections of memory chips are not shown throughout the chapter for simplicity.

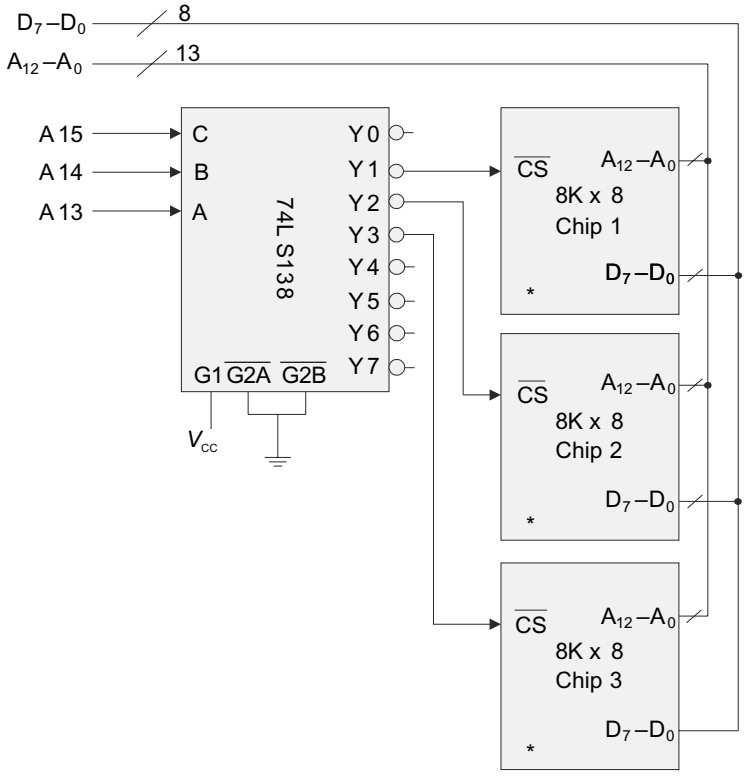


Fig. 21.8 *Interface three 8K x 8 bit chips*

*Exact difference between interfacing RAM and ROM is discussed in detail in the next section.

Example 21.4

Find the address range that is decoded by Y_0 and Y_7 outputs in Example 21.3.

Solution:

Y_0 will decode the address range from 0000H to 1FFFH and Y_7 will decode the range E000H to FFFFH.

It should be noted that the address range within a memory chip of any 8Kbyte is from 0000H to 1FFFh. But this address range can be mapped at any range of the microcontroller address space like 0000 to 1FFF or 2000 to 3FFF or 4000 to 5FFF or 6000 to 7FFF and so on as shown in Figure 21.9.

Broadly, there are two types of address decoding:

1. **Full decoding or absolute decoding:** This method uses all the address lines. The advantage of this method is that each memory location has a unique address. It is recommended to use this method.
2. **Partial decoding:** This method does not use all the address lines; therefore, each memory location will have multiple addresses. The only advantage of this method is that it reduces the cost of the decoding circuit.

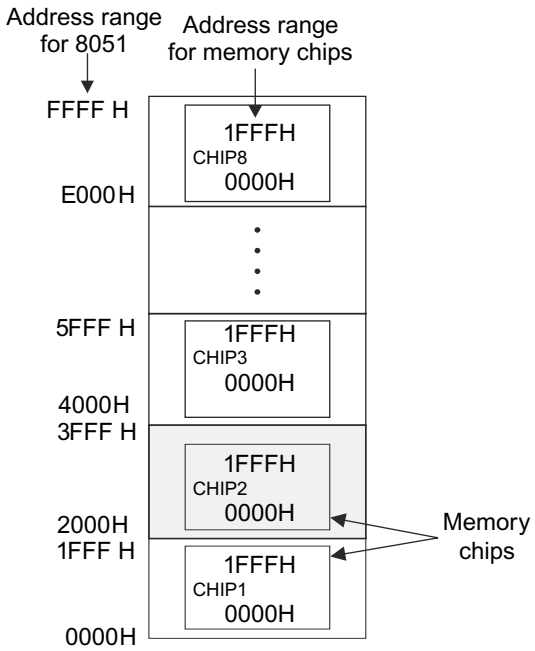


Fig. 21.9 *Mapping of 8K memory chips on the 8051 address space*

21.5 PROGRAM/CODE MEMORY INTERFACING

The 8051 has 16 address lines; therefore, it has 64Kbytes of program memory address space. As discussed in topic 21.4.2, $\overline{PSEN}$ of the microcontroller is normally connected with $\overline{OE}$ of the program memory (ROM) to access code bytes. $\overline{PSEN}$ is generated either when the program memory is accessed by 'MOVC' instructions or during the program byte fetches. It may also be connected with $\overline{CE}$ (Chip Enable) of the memory chip if only one ROM chip is present in a system. Interfacing of 16 Kbyte ROM is shown in Figure 21.10. Since the chip capacity is 16K bytes, the address range will be from 0000H to 3FFFH. It will require 14 (2^{14} = 16K) address lines.

$\overline{CE}$ should be generated using the address-decoder circuit when there is more than one ROM chips present in a system. It is illustrated in Example 21.5.

Note that A_{14} and A_{15} are not connected. These pins are not used and left unconnected, because there is no requirement of the address decoder since there is only one ROM chip connected at address 0000H to 3FFFH. The disadvantage of this is that each memory location will have four addresses. $\overline{EA}$ is connected to the ground because only external code memory is used. 74LS373 is an 8-bit latch used to de-multiplex the lower address and data bus. ALE signal is used to enable the latch when the address is present on the lower address/data (AD_7–AD_0) bus.

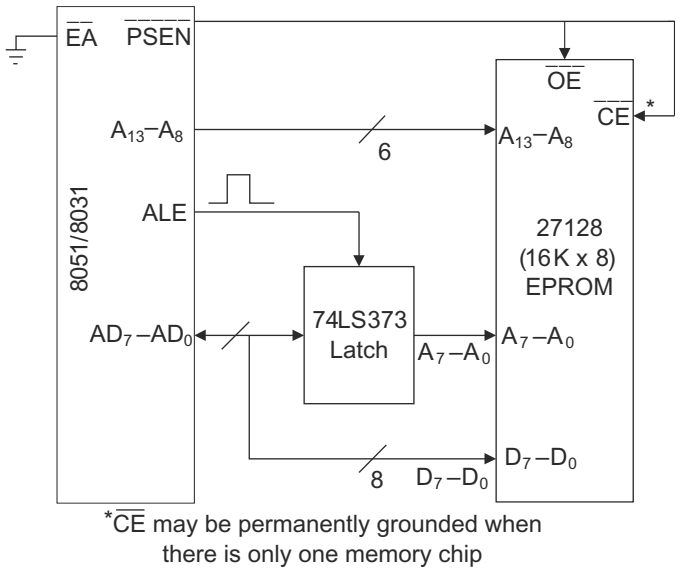


Fig. 21.10 *Interfacing of 16K x 8 EPROM with the 8051*

Example 21.5

Interface two 4K bytes ROM (2732) chips from address 0000H onwards.

Solution:

Since each chip capacity is 4K bytes, the address range for the first will be from 0000H to 0FFFH and for the second chip, it will be 1000H to 1FFFH. It will require 12 (2^{12} = 4K) address lines. The interfacing diagram is shown in Figure 21.11.

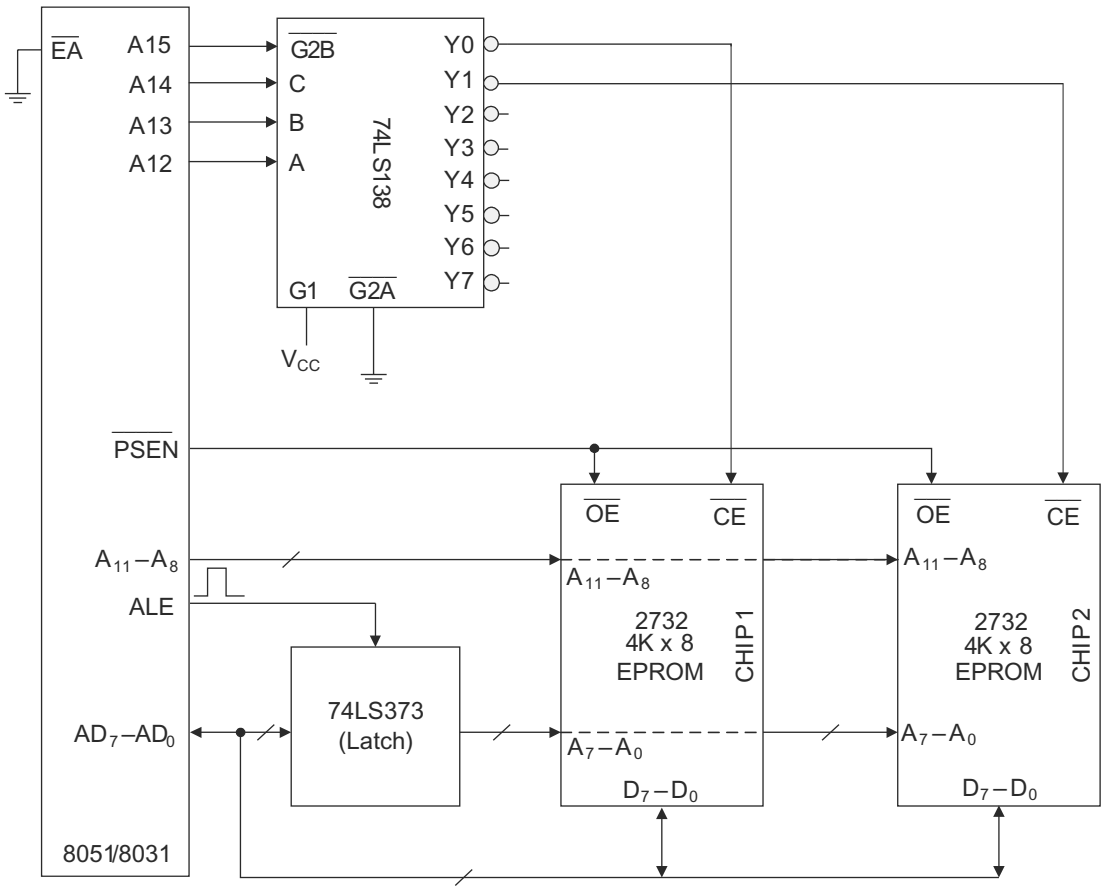


Fig. 21.11 *Interfacing of program memory -Two 4K x 8 EPROM chips*

Note that the data lines and address lines (A_{11}– A_0) of the microcontroller are connected with data lines and address lines (A_{11}–A_0) of both the memory chips simultaneously, but based on the desired address, only one chip will be activated and, therefore, data will be transferred from the correct chip without any confusion and bus contention. $\overline{CE}$ of Chip 1 and Chip 2 are generated by Y_0 and Y_1 of the decoder respectively. The decoder is enabled only when A_{15} = 0 because $\overline{G2B}$ is connected with A_{15} of the microcontroller. $\overline{OE}$ is connected to $\overline{PSEN}$.

Example 21.6

For a memory interfacing circuit shown in Figure 21.11, write a C program to read 10 bytes of code memory (EPROM) from address 1000H onwards. Store the bytes in the internal RAM from the address 20H onwards.

```
#include <reg51.h>
#include <absacc.h>                                    // include file for CBYTE and DBYTE

void main (void)
        {
```

```
unsigned char i;
unsigned int j;
i  = 0x20;                          // starting internal RAM address 20 H, from where data will be stored
for ( j = 0x1000; j<0x100A; j++)    // read the data from address 1000H to 1009H
      {
      DBYTE [i]  = CBYTE[j];        // read the data from code memory address 'j' and
                                    // store at internal RAM address 'i'
      i++;                          // Increment internal RAM address
      }
}
```

Note that the CBYTE and DBYTE are the macros used to access the bytes at absolute address in the code and internal data memory of the 8051 respectively. These macros are defined in 'absacc.h' file.

21.6 | DATA/RAM MEMORY INTERFACING

The 8051 has 64 Kbyte data-memory space. It is accessed using 'MOVX' instructions. As discussed in topic 21.4.2, $\overline{RD}$ and $\overline{WR}$ signals from the 8051 are used to access the RAM.

Example 21.7

Interface two 16 Kbytes RAM chips with the 8051. Connect the first chip starting at 4000H onwards and the second chip C000H onwards.

Solution:

The 16K Byte chip requires 14 address lines (A_{13}–A_0). The address range for Chip 1 is 4000H to 7FFFH and for the second chip, it is C000H to FFFFH. Interfacing is shown in Figure 21.12.

The chip1 is selected when $A_{15} = 0$, $A_{14} = 1$.

The chip2 is selected when $A_{15} = A_{14} = 1$.

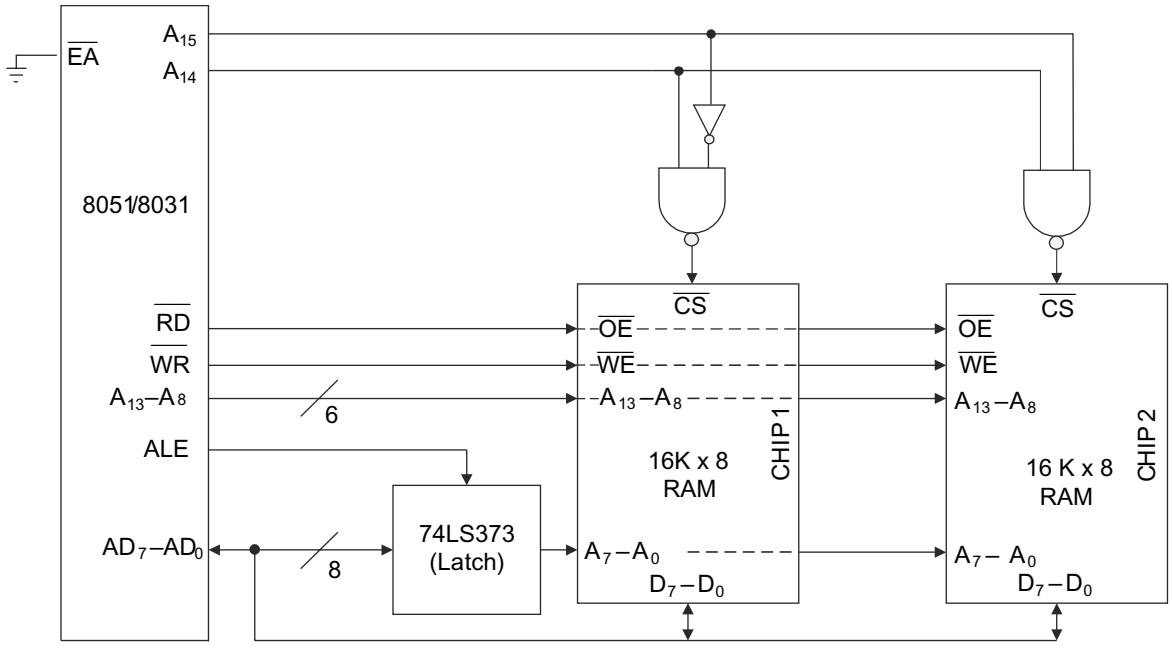


Fig. 21.12 *Interfacing of data memory—Two 16 x 8 RAM chips*

Example 21.8

For a memory interfacing circuit shown in Figure 21.12, write a program to store ASCII codes for numbers 0 to 9 in the external RAM from the address 4000H onwards.

Solution:

The data byte to/from external memory is written/read using 'MOVX' instructions. The starting address of the memory is stored in the DPTR register and it is incremented for every memory access. The ASCII codes for 0 to 9 are 30H (48d) to 39H(57d).

```
        ORG 0000H
        MOV DPTR, #4000H        // initialize DPTR to point to address 4000H
        MOV R2, #10             // counter for 10 numbers (0 to 9)
        MOV A, #30H             // load data 30H into A (ASCII code for 0)
NEXT:   MOVX @DPTR, A           // copy contents of A to external RAM address pointed by DPTR (4000H in this example)
        INC A                   // ASCII of the next number
        INC DPTR                // point to the next address
        DJNZ R2, NEXT           // go for storing the next number
HERE:   SJMP    HERE
        END
```

Example 21.9

Rewrite the program of Example 21.8 in the C language.

Solution:

The corresponding C language program is:

```
#include <reg51.h>
#include <absacc.h>                 // include file for XBYTE
void main (void)
    {
    unsigned char i,
    unsigned int j;
    i = 0x30;                       // ASCII for 0
    for ( j = 0x4000; j<0x400A; j++)    // write data from address 4000H to 4009H
        {
        XBYTE [j]  = i;             // write ASCII of 0 to 9
        i++;                        // ASCII value of next number
        }
}
```

Note that the XBYTE is the macros used to access the bytes at an absolute address in external data memory of the 8051.

Example 21.10

Write a C program to read 10 bytes of data from external RAM from location 2000H and output the same on Port P2.

Solution:

```
#include <reg51.h>
#include <absacc.h>                 // include file for XBYTE
void main (void)
    {
    unsigned int i;
    for (i = 2000; i<200A; i++)     // read data from 2000 to 200A
    P2 = XYBTE [i];                 // read data byte from RAM and send it to P2
    }
```

Refer topics 4.4 and 9.2 for more assembly language programs related to the external data movements.

21.7 DATA MEMORY USING ROM

So far, we considered data memory as RAM only, i.e. RAM is used to store the data. But there are times when we need to store data into ROM, for example, look-up tables are examples of data and are required to be stored in ROM because they have to remain permanently in the memory. Examples of look-up tables are table of 7-segment codes for BCD numbers, array of data samples for sine wave to be generated by DAC, passwords (strings), step sequences of stepper motors, or any group of pre-calculated values. The use of external ROM as data memory is shown in Example 21.11.

Example 21.11

Interface 4Kbyte of ROM (2732-EPROM) as data memory starting at the memory location 0000H.

Solution:

The 4K Byte chip requires 12 address lines (A_{11}–A_0). The point to be noted is that the microcontroller should treat this ROM as a RAM and hence, the $\overline{RD}$ is connected to the $\overline{OE}$ to access data from the ROM as shown in Figure 21.13 (note that $\overline{PSEN}$ will not be used to access the memory). Instructions that are used to access RAM, i.e. MOVX A, @DPTR or MOVX A, @*Ri* should be used.

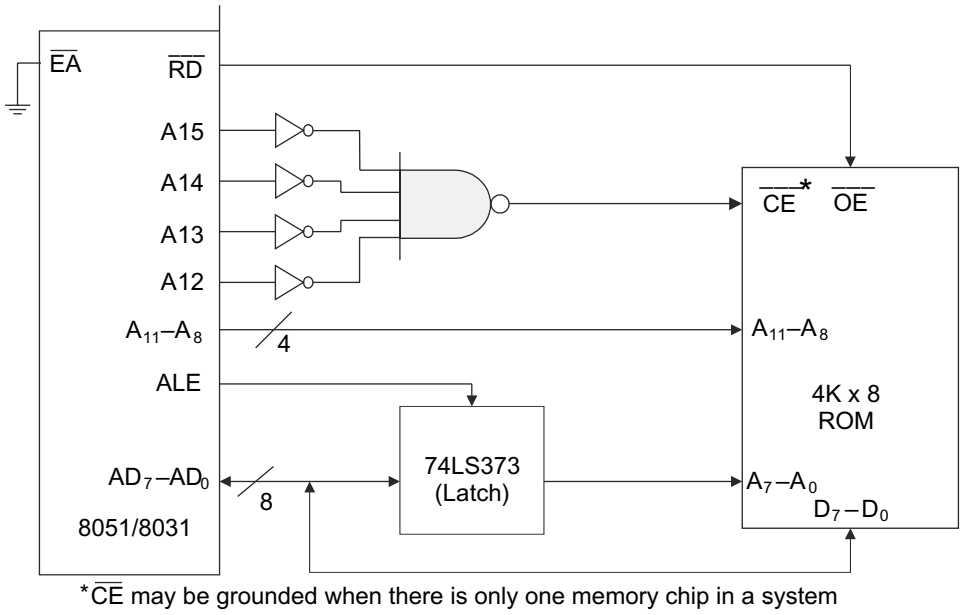


Fig. 21.13 *ROM as a data memory*

'ROM as a data memory' means ROM chip is connected in the address space of RAM by using the control signal used to read data from RAM, i.e. the data from ROM (instead of RAM) will be read, when MOVX A,@DPTR instructions are executed. Remember that while using ROM as data memory we should not attempt to write data into ROM.

21.8 ROM AS DATA AS WELL AS PROGRAM MEMORY

Some part of ROM may be used to store the program while the other part may be used to store the data. We know that $\overline{PSEN}$ is used to access the program and $\overline{RD}$ is used to read the data space. So $\overline{OE}$ of ROM must be activated for both code and data access. So $\overline{RD}$ and $\overline{PSEN}$ should be ANDed to get $\overline{OE}$ as shown in Figure 21.14.

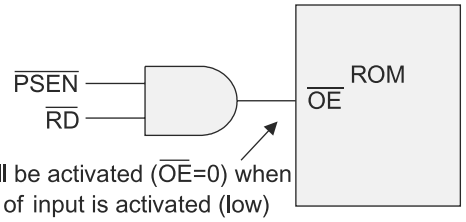


Fig. 21.14 *ROM as program as well as data memory*

21.9 RAM AS DATA AS WELL AS PROGRAM MEMORY

This case is similar to ROM as data as well as program memory. The $\overline{PSEN}$ and $\overline{RD}$ signals are used in the same way. The $\overline{WR}$ should be connected to $\overline{WE}$ of the memory chip. The RAM memory is read as data or code memory and written as only data memory. The program may be executed from RAM by accessing it as program memory; this will be helpful in debugging. It is shown in Figure 21.15.

We can also connect ROM and RAM both with 8051. This will be illustrated in Example 21.12.

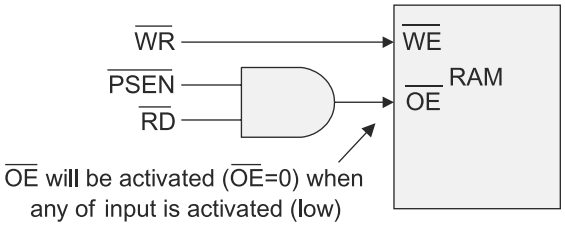


Fig. 21.15 *RAM as data as well as program memory*

Example 21.12

Interface 8 Kbytes of SRAM (6264) and 8 Kbytes of EEPROM (2864) with the 8051 both starting at the address 0000H.

Solution:

The 8051 has two parallel address spaces for RAM and ROM ranging from 0000H to FFFFH; therefore, we can connect both the RAM and ROM at the same address range. Different control signals are used to access data from both the memories.

The 8 KByte chip requires 13 address lines (A_{12}–A_0). RAM signals $\overline{WE}$ and $\overline{OE}$ are connected with $\overline{WR}$ and $\overline{RD}$ of the 8051 respectively and ROM signals $\overline{OE}$ and $\overline{CE}$ are connected with $\overline{PSEN}$ of the 8051as shown in Figure 21.16.

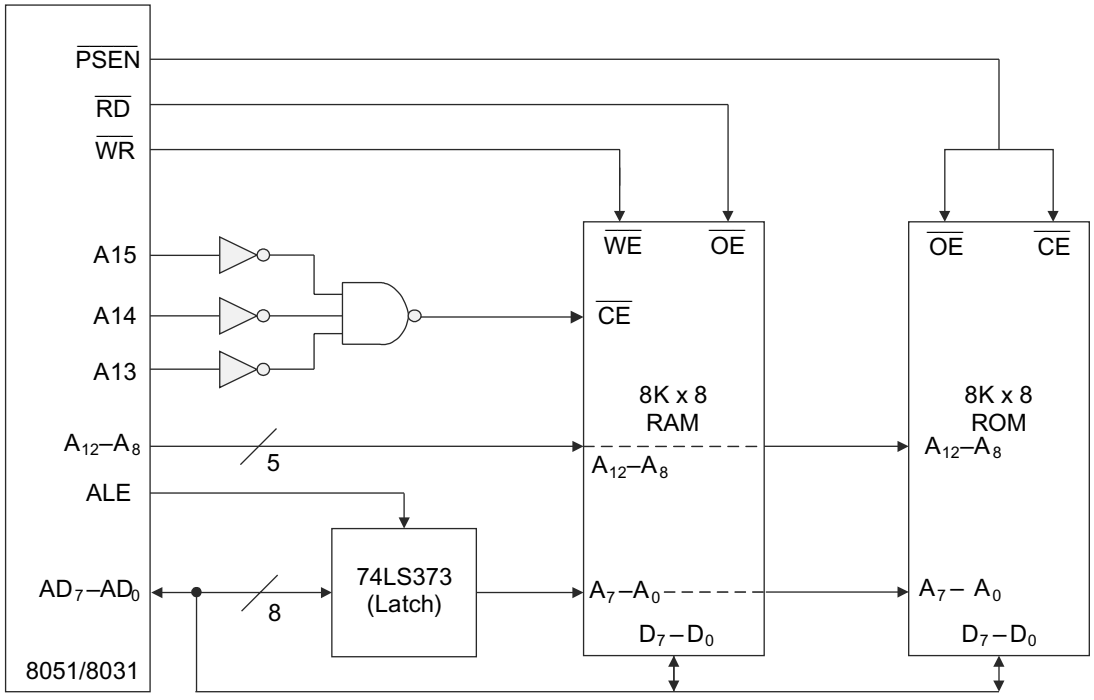


Fig. 21.16 *Interfacing RAM as well as ROM with 8051*

21.10 ON-CHIP EEPROM PROGRAMMING IN AT89S8253

The AT89S8253 has 12K bytes of on-chip In System Programmable flash as well as 2K bytes of EEPROM. The 12Kbytes of flash is used as code memory and 2Kbytes of EEPROM is used as a data memory. The 12Kbyte of flash is placed in the code memory address space from the address 0000H to 2FFFH. The 2Kbytes of data EEPEROM is placed in the data memory address space from 0000H to 07FFH. This data EEPROM can be programmed during runtime using program instructions, i.e. microcontroller instructions can program this EEPROM. This feature allows storing (or updating) some

important data in real time without affecting other bytes. Moreover, the data will remain stored in the memory (because EEPROM is nonvolatile memory) even if the power is removed. This feature has got many applications, for example, in remote sensing applications, the device can gather some important data in EEPROM which may be used later for the analysis.

The data EEPROM in AT89S8253 has auto-erase capability at byte level, i.e. we need not erase the EEROM byte before writing to it. This EEPROM is byte readable, byte or page writeable. In page mode, 32 bytes can be written to or erased using a single write cycle. The lower 5 bits of page address will vary from 00000 to 11111 and the remaining address bits will remain constant. For example, addresses 0000H to 001F belong to the same page. Note that in the page mode, addresses of all memory locations to be written must belong to the same page.

The read/write/erase operation of data EEPROM can be done with the help of MOVX instructions and memory control special function register-EECON. It is referred as data EEPROM control register. The bit configuration and description of EECON register is shown in Table 21.2.

Table 21.2 *EECON register*

–	–	EELD	EEMWE	EEMEN	DPS	RDY/BSY	WRTINH
MSB							LSB

Bit	Symbol	Description
7	–	–
6	–	–
5	EELD	EEPROM Load Enable bit. Used for page write mode. A MOVX instruction writing into EEPROM will not start write cycle if EELD = 1, but it will just load data into the volatile data buffer of the EEPROM. Before the last MOVX, clear this bit to program all the bytes previously loaded on the same page of the address given by the last MOVX instruction.
4	EEMWE	EEPROM Write Enable bit. Set this bit to 1 before starting the write operation to EEPROM with the MOVX instruction. The program should clear this bit to 0 after write operation is completed.
3	EEMEN	On-chip EEPROM Access Enable. If EEMEN = 1, the MOVX instruction will access the on-chip EEPROM instead of external data memory if the address used is less than 2K. When EEMEN = 0 or the address used is $\geq$ 2K, MOVX accesses external data memory.
2	DPS	Data Pointer Register Select. DPS = 0 selects the first bank of data pointer register, DP0, and DPS = 1 selects the second bank, DP1.
1	RDY/$\overline{\text{BSY}}$	RDY/$\overline{\text{BSY}}$ (Ready/Busy) flag for the data EEPROM. This is a read-only bit which is cleared by the hardware during the write cycle of the on-chip EEPROM. It is set by the hardware when the write operation is completed.
0	$\overline{\text{WRTINH}}$	WRTINH (Write Inhibit) is a READ-ONLY bit which is cleared by hardware when the VCC is too low for the write cycle of the on-chip EEPROM to be executed. When this bit is cleared, an ongoing write cycle will be aborted or a new write cycle will not start.

Steps to Write (Program) a Byte in Data EEPROM

- Set EEMEN bit, (EEMEN = 1) in EECON register to enable the access to data EEPROM.
- Set EEMWE bit, (EEMWE = 1) in EECON register to enable EEPROM for writing into it.
- Load the address of memory that is to be written (programmed) into DPTR register (Make sure that the selected DPTR (by DPS bit in EECON register) is used.
- Load desired data byte into the Accumulator register.
- Load the byte to desired address using instruction MOVX @DPTR, A.
- Monitor RDY/$\overline{\text{BSY}}$ bit, and wait until RDY/$\overline{\text{BSY}}$ = 1 to make sure that the write operation is completed.
- Clear EEMWE bit, (EEMWE = 0) after the write operation is completed.
- Clear EEMEN bit (EEMEN = 0) to access the off-chip data memory, if desired.

Steps to Write a Page (Maximum 32 Bytes) in Data EEPROM

- Set EEMEN bit, (EEMEN = 1) in EECON register to enable the access to data EEPROM.
- Set EEMWE bit, (EEMWE = 1) in EECON register to enable EEPROM for writing into it.
- Set EELD bit, (EELD = 1) in EECON register to enable the page load mode.
- Load all the desired locations on page with suitable data using MOVX instructions.
- Clear EELD bit, (EELD = 0) before writing last MOVX instruction for a given page.
- so that after the last MOVX instruction all bytes in the page are written simultaneously.
- Monitor RDY/$\overline{\text{BSY}}$ bit, and wait until RDY/$\overline{\text{BSY}}$ = 1 to make sure that the write operation is completed.
- Clear EEMWE bit, (EEMWE = 0) after write operation is completed.
- Clear EEMEN bit (EEMEN = 0) to access the off-chip data memory, if desired.

Steps to Read a Byte(s) from Data EEPROM

- Set EEMEN bit, (EEMEN = 1) in EECON register to enable the access to data EEPROM.
- Load the address of memory that is to be read into DPTR register (Make sure that the selected DPTR (by DPS bit in EECON register) is used.
- Read the byte from desired address using instruction MOVX A, @DPTR.
- Repeat the above two steps until all the desired data bytes are read from data EEPROM.
- Clear EEMEN bit (EEMEN = 0) to access off-chip data memory, if desired.

The following programming examples illustrate the process of EEPROM programming and reading.

Example 21.13

Write a program to write (burn or program) 10 data bytes stored at internal RAM address 40H onwards to the data EEPROM address 0000H onwards.

Solution:

The program can be written in two ways, using the byte write mode or page write mode.

(i) Program using byte-write mode is given below:

```
        EECON EQU 96H
        ORG 0000H
        MOV R2, #10          // counter to write 10 data bytes,
        MOV R0, #40H         // load R0 with address of the first data byte to be written
        ORL EECON, #08H      // EEMEN = 1, enable access to EEPROM
        ORL EECON, #10H      // EEMWE = 1, enable EEPROM for writing
        MOV DPTR, #0000H     // load DPTR with address of first location of EEPROM
NXT:    MOV A,@R0            // place the data byte to be written into Accumulator
        MOVX @DPTR, A        // program data byte into EEPROM
WAIT:   MOV A, EECON         // move EECON in A to check RDY/BSY bit
        ANL A, #02H          // mask all the bits except RDY/BSY
        JZ WAIT              // if RDY/BSY = 0, wait until the completion of write operation
        INC DPTR             // point to the next EEPROM location
        INC R0               // point to the next data byte
        DJNZ R2, NXT         // repeat operation for 10 bytes
        ANL EECON, # 0E7     // clear EEMEN and EEMWE bits to prevent further access to EEPROM
        END
```

(ii) Program using page-write mode is given below:

```
        EECON EQU 96H
        ORG 0000H
        MOV R2, #09          // counter to write 9 data bytes, the 10th byte will be written
                             // outside the loop after clearing EELD bit to start page write operation
        MOV R0, #40H         // load R0 with address of first data byte to be written
```

```
        ORL EECON, #08H     // EEMEN = 1, enable access to EEPROM
        ORL EECON, #10H     // EEMWE = 1, enable EEPROM for writing
        ORL EECON,#20H      // EELD = 1, to enable page mode
        MOV DPTR, #0000H    // load DPTR with address of first location of EEPROM
NXT:    MOV A,@R0           // place the data byte to be written into Accumulator
        MOVX @DPTR, A       // program data byte into EEPROM buffer
        INC DPTR            // point to the next EEPROM location
        INC R0              // point to the next data byte
        DJNZ R2, NXT        // repeat operation for 9bytes
        ANL EECON, # 0DFH   // clear EELD before last MOVX instruction
        MOVX A,@R0          // place last data byte to be written into Accumulator
        MOVX @DPTR, A       // write last byte in EEPROM
WAIT:   MOV A, EECON        // move EECON in A to check RDY/BSY bit
        ANL A, #02H         // mask all bits except RDY/BSY
        JZ WAIT             // if RDY/BSY = 0, wait until completion of write operation
        ANL EECON, # 0E7    // clear EEMEN and EEMWE bits to prevent further access to EEPROM
        END
```

Note that the program based on pagewrite mode in this example will perform the same operation 10 times faster than the byte-write mode.

Example 21.14

Write a program to read 10 bytes from on-chip EEPROM address 0000H onwards and store these data in the internal RAM in the address 20H onwards.

Solution:

```
        EECON EQU 96H
        ORG 0000H
        MOV R2, #10         // counter to read 10 data bytes,
        MOV R0, #20H        // load R0 with destination
        ORL EECON, #08H     // EEMEN = 1, enable access to EEPROM
        MOV DPTR, #0000H    // load DPTR with address of first location of EEPROM
NXT:    MOVX A,@DPTR        // read the byte from EEPROM
        MOV @R0, A          // store data in internal RAM address 20H onwards
        INC DPTR            // point to next EEPROM location
        INC R0              // point to next data byte
        DJNZ R2, NXT        // repeat operation for 10 bytes
        ANL EECON, #0E7H    // clear EEMEN and EEMWE bits to prevent further access to EEPROM
        END
```

The on-chip data EEPROM of microcontrollers from different manufacturers require different methods for their programming, for example, on-chip data EEPROM of LPC93x microcontrollers is programmed through DEEADR, DEECON and DEEDAT special function registers, moreover they do not have an auto-erase capability, i.e. we need to erase a memory location before we can write to it. They do not use MOVX instructions to access the EEPROM. Refer datasheet of a particular microcontroller to know how to program on-chip data EEPROM.

21.11 | REAL-TIME CLOCK

Real-Time Clock (RTC) is a peripheral module (available either as a separate chip or on-chip peripheral) used to keep time (real time), alarm and calendar information. It provides seconds, minutes, hours, day of the month, month, year, day of the week, and day of the year. This information can be used to perform the timing operations such as time stamping of an event, generating alarms at a specified time and generating very long time delays. Usually RTC modules are powered separately and are isolated from the rest of the circuitry, allowing them to operate even in the absence of main power

supply and because of this reason the RTC devices are named so. The DS12887 is a popular RTC chip manufactured by Dallas Semiconductors. In this section, programming and interfacing of DS12887 with the 8051 is discussed.

21.11.1 DS12887: Real-time Clock Chip

Main Features of the DS12887

- Keeps the information of seconds, minutes, hours, day of the week, date, month, and year (with leapyear adjustments up to the year 2100)
- Contains the lithium battery which retains chip contents for 10 years in the absence of external power supply
- Binary/BCD formats of time, calendar and alarm
- 12/ 24 hour clock modes (with AM/PM indications in 12-hour mode)
- 14 bytes for clock and control registers and 114 bytes for general-purpose RAM
- Programmable square wave output signal

Refer datasheet of DS12887 for more details and other features.

The pin diagram of DS12887 chip is shown in Figure 21.17. The pin description of DS12887 is given in Table 21.3.

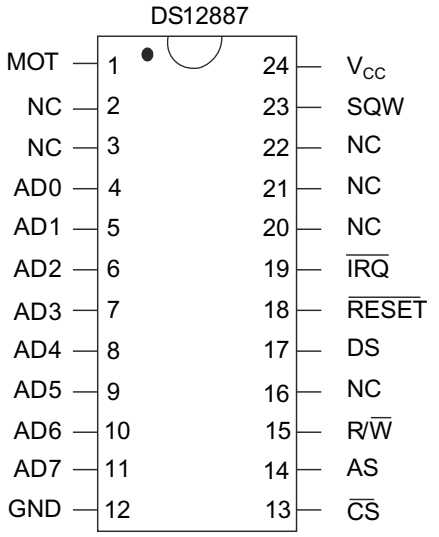


Fig. 21.17 *Pin diagram of RTC chip DS12887*

Table 21.3 *Pin description of DS12887*

Pin	Description
V_{CC}	+5 V power supply. When it falls below a 3 V, the external supply is disconnected, and internal lithium battery automatically provides power. For its value less than 4.25 V, the read/write operations are suspended but the timing activities are continued. When V_{CC} becomes greater than 4.25 V, the chips can be accessed only after 200 ms.
GND	Ground.
AD_0–AD_7	Multiplexed address/data bus. The address (provided by the microcontroller) is latched within 12887 at the trailing edge of the AS signal.
AS	Address strobe. Used to de-multiplex address/data. It is equivalent to ALE of the 8051.
MOT	Used to select between Intel and Motorola bus timing formats. Connect to ground (or unconnected) for Intel timing and to Vcc for Motorola timing.
SQW	Square Wave Output. Square waves of 15 different frequencies can be generated by DS12887. The frequency can be selected by programming register A. The square wave output can be turned on/off by SQWE bit in Register B.
DS	Data Strobe or Read Input. When MOT = 0, the DS pin works as Read ($\overline{RD}$). It is equivalent to $\overline{OE}$ of a memory chips.
R/$\overline{W}$	When MOT = 0, the R/$\overline{W}$ is an active low signal which works as write. It is equivalent to $\overline{WE}$ of a memory chip.
$\overline{CS}$	Chip Select. It is an active low signal used to access the chip during read and write operations. For V_{CC} < 4.25 V, this pin is ignored, i.e. access to the chip is inhibited irrespective of the status of chip select pin. This will save the internal data in absence of external power supply.
$\overline{IRQ}$	Interrupt Request Output. The interrupt enable bits in Register B must be set to high to use IRQ. This signal may be used to interrupt the microcontroller.
$\overline{RESET}$	This clears PEI, AIE, UF, IRQF, PF, AF, UIE, SQWE bits. It has no effect on time, calendar and RAM contents. It is usually connected to Vcc.

21.11.2 Address Map

The DS12887 contains 128 bytes of RAM (addresses 00-7FH). First ten bytes (00-09H) contains time, calendar and alarm information. The next four bytes (0A-0D) are the control and status registers. Remaining 114 bytes (0E-7FH) can be used as general purpose RAM. Table 21.4 shows detailed address map of the DS12887.

21.11.3 Interfacing DS12887 with the 8051

The interfacing of RTC chip DS12887 with the 8051 is shown in Figure 21.18. Since DS12887 has multiplexed 8-bit address/data bus ($AD_{7\text{-}0}$), these pins are directly connected with the $AD_{7\text{-}0}$ (lower address and data bus, i.e. P0) of the 8051.

Table 21.4 *Address map of DS12887*

ADDRESS	FUNCTION	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	RANGE *
00H	Seconds	0	10 Seconds			Seconds				00–59 (00–3BH)
01H	Seconds alarm	0	10 Seconds			Seconds				00–59 (00–3BH)
02H	Minutes	0	10 Minutes			Minutes				00–59 (00–3BH)
03H	Minutes alarm	0	10 Minutes			Minutes				00–59 (00–3BH)
04H	Hours	AM/PM	0	0	10 Hours	Hours				AM:1-12, PM:81-92# 00–23(00–17H)
		0		10 Hours						
05H	Hours alarm	AM/PM	0	0	10 Hours	Hours				AM:1-12, PM:81-92# 00–23(00–17H)
		0		10 Hours						
06H	Day	0	0	0	0	0	Day (1 = Sunday)			01–07 (01–07H)
07H	Date	0	0	10 Date		Date				01–31 (01–1FH)
08H	Month	0	0	0	10 Months	Month				01–12 (01–1CH)
09H	Year	10 Years				Year				00–99 (01–63H)
0AH	Register A	UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0	—
0BH	Register B	SET	PIE	AIE	UIE	SQWE	DM	24/12	DS	—
0CH	Register C	IRQF	PF	AF	UF	0	0	0	0	—
0DH	Register D	VRT	0	0	0	0	0	0	0	—
0EH–7FH	RAM	X	X	X	X	X	X	X	X	—

* When DM (bit 2 in Register B) = 0, the RANGE is in decimal (BCD) and for DM = 1, RANGE is in binary. The values in binary should be directly loaded in the corresponding RAM location, for example, to load 3BH in a seconds location (address 00H), write 3BH directly into address 00H.

AM: 01-1CH, PM: 81-8CH for binary mode.

The Chip Select signal can be generated using address decoder circuit; the input to the address decoder circuit will be high order address bus. (Refer topic 21.4 for details of address decoding). For simplicity, the Chip Select ($\overline{CS}$) signal is permanently grounded; therefore, the RTC chip is always selected. The $\overline{RD}$ and $\overline{WR}$ of the 8051 is connected with DS and R/$\overline{W}$ of the DS12887, therefore the DS12887 is mapped on the external data memory space, i.e. addresses 00-7FH of the DS12887 is seen as an external data memory connected from 00H to 7FH.

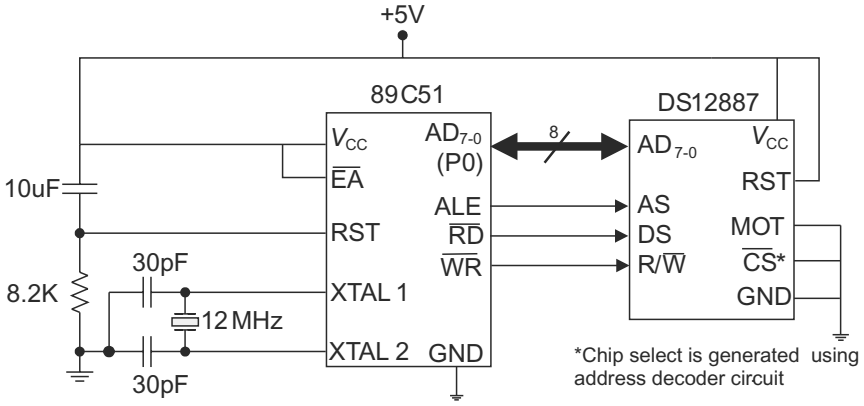


Fig. 21.18 *Interfacing RTC chip DS12887 with the 8051*

The ALE signal of the 8051 is connected directly with the AS signal of the DS12887 to demultiplex address and data. Note that there is no need of external latch circuit (like 74373) to de-multiplex address and data because DS12887 has an internal latch. Since only 8 address bits (A_{7-0}) are used in our example, the contents of the DS12887 can be accessed using instructions MOVX A, @Ri or MOVX @Ri, A. (If we use A_{15-8} to generate chip select signal then the DS12887 can be accessed using instructions MOVX A,@DPTR or MOVX @DPTR,A).

21.11.4 Programming the DS12887

Oscillator Control Bits

To save the internal lithium battery, the new DS12887 chips have the internal oscillator turned off. To use the DS12887 for timing activities, first, we have to turn on the oscillator. Writing 010 to bits D_6 D_5 D_4 (DV_2 DV_1 DV_0 bits,) in Register A (address 0AH, refer Table 21.4) will turn on the internal oscillator.

Setting the Time and Date

As per the datasheet of the DS12887, the following points must be considered while programming the chip.

- The SET bit (bit 7 of Register B) must be programmed to logic 1 before we modify (or initialize) the time, calendar and alarm registers. This will prevent the updates of these registers by RTC chip hardware when a user program is modifying them. Once these registers are initialized, the SET bit must be cleared so that they can be updated by RTC chip hardware.
- The time, calendar and alarm registers must use the same data mode at a time. (either BCD or binary). This data mode can be selected by DM bit (bit 2 of the Register B, refer Table 21.4). DM = 0 means BCD mode while DM = 1 selects binary mode.
- 12/24 hour mode can be selected by 24/12 bit (bit 1 in Register B, refer Table 21.4). 24/12 bit = 0 selects 12-hour mode and 24/12 = 1 selects 24-hour mode. The 24/12 bit cannot be modified without reinitializing the hour registers. When the 12 hour mode is selected, the MSB of the hour's byte represents AM when it is 0, while it represents PM when it is 1.
- The DSE (Daylight Savings Enable) = 1 (bit 0 in Register B) will enable the two special updates: on the first Sunday in April, the time increments to 3:00:00 a.m. from 1:59:59 a.m., and on the last Sunday in October when the time changes to 1:00:00 a.m. from 1:59:59 a.m.. DM = 0 will not enable any updates.
- The time, calendar, and alarm registers are always accessible because they are double buffered.
- When V_{CC} is applied to the chip and reaches the level of greater than 4.25 V, the chip can be accessed only after 200 ms.

Setting the time and date in DS12887 is explained in the Example 21.15.

Example 21.15

For the interfacing circuit shown in Figure 21.18, set the time as 10:20:40 a.m. in 12-hour BCD mode and daylight saving. Set the date as 17th January 2014 and the day as Friday.

Solution:

As discussed earlier, the RAM of DS12887 in Figure 21.18 is mapped to the external data memory address space (addresses 00–7FH). Since $\overline{CS}$ is grounded, only 8-bit addresses are used to access RAM of DS12887. This can be done using MOVX A, @Ri or MOVX @Ri, A. Assume that the delay subroutine of 200 ms is available.

```
                        // turn on the oscillator for the first time
LCALL DELAY             // wait for 200 ms after power on.
MOV R1, #10             // initialize R1 as a pointer to Register A (address 10 or 0AH)
MOV A, #20H             // configure the command word to turn on oscillator by making bits
                        // D6D5D4 = 010
MOVX @R1, A             // turn on the oscillator by sending command to Register A
                        // configure the time mode
MOV R1, #11             // initialize R1 as a pointer to Register B (address 11 or 0BH)
MOV A, #81H             // configure the command word for 12 h BCD mode with daylight
                        // savings, SET bit = 1 to prevent updates during initialization
MOVX @R1, A             // send the above command to Register B
                        // set the time
MOV R1, #00             // initialize R1 as a pointer to seconds register (address 00)
MOV A, #40H             // load seconds value = 40H into A
MOVX @R1, A             // send seconds value to seconds register (00H)
MOV R1, #02             // initialize R1 as a pointer to the minutes register (address 02)
MOV A, #20H             // load minutes value = 20H into A
MOVX @R1, A             // send the minutes value to minutes register (02H)
```

```
MOV R1, #04          // initialize R1 as a pointer to hours register (address 04)
MOV A, #10H          // load hours value  = 10H into A, Bit7  = 0 for AM
MOVX @R1, A          // send hours value to the hours register (04H)

                     // set the Day and DATE
MOV R1, #06          // initialize R1 as a pointer to day register (address 06)
MOV A, #06H          // load day value  = 06H (Friday, day of the week) into A
MOVX @R1, A          // send day value to the day register (06H)
MOV R1, #07          // initialize R1 as a pointer to the date register (address 07)
MOV A, #17H          // load date value  = 17H (day of the month) into A
MOVX @R1, A          // send date value to the date register (07H)
MOV R1, #08          // initialize R1 as a pointer to the month register (address 08)
MOV A, #01H          // load month value  = 01H (January) into A
MOVX @R1, A          // send month value to the month register (08H)
MOV R1, #09          // initialize R1 as a pointer to the year register (address 09)
MOV A, #14H          // load year value  = 14H (2014) into A
MOVX @R1, A          // send year value to the year register (09H)

                     // allow update of time and calendar
MOV R1, #11          // initialize R1 as a pointer to Register B (address 11 or 0BH)
MOV A, #01H          // SET bit  = 0 to allow update of the time and calendar by RTC
                     // don't disturb other bits (12 hr BCD mode, daylight saving)
MOVX @R1, A          // send the above command to Register B
```

Example 21.16

Rewrite the program of Example 21.15 in the C language. (For the interfacing circuit shown in Figure 21.18, set the time as 10:20:40 a.m. in 12-hour BCD mode and daylight saving. Set the date as 17th January 2014 and the day as Friday.)

Solution:

Assume that the delay routine of 200ms is available

```
#include <reg51.h>
#include <absacc.h>          // include file for XBYTE

void main (void)
    {
                             // turn on the oscillator for the first time
    delay();                 // wait for 200 ms after power on.
    XBYTE[10] = 20H          // configure the command word to turn on oscillator by making bits
                             // D6D5D4 = 010 and write to Register A

                             // configure the time mode
    XBYTE[11] = 81H          // configure command word for 12 hr BCD mode with daylight
                             // savings, SET bit  = 1 to prevent the updates during initialization
                             // and write the above command to Register B

                             // set the time
    XBYTE[00] = 40H          // set the seconds value  = 40H into seconds register
    XBYTE[02] = 20H          // set the minutes value  = 20H into minutes register
    XBYTE[04] = 10H          // set the hours value  = 10H into hours register, bit 7 = 0 for AM

                             // set the Day and DATE
    XBYTE[06] = 06H          // set day value  = 06H (Friday) into day of the week register
    XBYTE[07] = 17H          // set date value  = 17H into day of the month register
```

```
XBYTE[08] = 01H          // set month value  = 01H (January) into month register
XBYTE[09] = 14H          // set year value  = 14H (2014) into year register

                         // allow the update of time and calendar
XBYTE[11] = 01H          // SET bit  = 0 to allow the update of time and calendar by RTC
                         // don't disturb other bits (12 hr BCD mode, daylight saving)
```

Note that the XBYTE is the macro used to access the bytes at absolute address in external data memory of the 8051. This macro is defined in the 'absacc.h' file.

Reading the Time and Date

Since the DS12887 is mapped on external data memory space at addresses 00–7FH, the internal RAM of the DS12887 can be read as if it is RAM chip. The contents of the DS12887 RAM can be read using instruction MOVX A, @Ri. Example 21.17 shows how to read time and date.

Example 21.17

For an interfacing circuit shown in Figure 21.18, write a program to read the time and date from DS12887 chip and display it on LCD.

Solution:

Assume that

- The RTC chip is already in use, i.e. the oscillator is turned on and time and date are already set earlier.
- The RTC is programmed in the BCD data mode and to display BCD data on the LCD, a routine 'DISPLAY' is available (Refer Example 18.8 for this). Also, the 'DISPLAY' routine contains a routine for BCD to ASCII conversion (Refer Example 9.3 for BCD to ASCII conversion). Thus, the DISPLAY routine converts a BCD byte into two ASCII bytes and sends both the bytes on the LCD.

```
                         // read and display time in HH:MM:SS format
MOV R1,#04H              // initialize R1 as a pointer to hours register (address 04)
MOVX A, @R1              // read hours into A
ACALL DISPLAY            // display hours on LCD
MOV A, # ':'             // display ':'
ACALL DISP1              // routine to display ASCII character in A
MOV R1, #02H             // initialize R1 as a pointer to the minute register (address 02)
MOVX A, @R1              // read minutes into A
ACALL DISPLAY            // display minutes on LCD
MOV A, # ':'             // display ':'
ACALL DISP1              // routine to display ASCII character in A
MOV R1,#00H              // initialize R1 as a pointer to seconds register (address 00)
MOVX A, @R1              // read seconds into A
ACALL DISPLAY            // display seconds on LCD

                         // display following on the next line in the LCD

                         // read and display the date in DD:MM:YYYY format
MOV R1,#07H              // initialize R1 as a pointer to the date register (address 07)
MOVX A, @R1              // read date into A
ACALL DISPLAY            // display date on LCD
MOV A, # ':'             // display ':'
ACALL DISP1              // routine to display ASCII character in A
MOV R1,#08H              // initialize R1 as a pointer to the month register (address 08)
MOVX A, @R1              // read month into A
ACALL DISPLAY            // display month on LCD
MOV A, # ':'             // display ':'
ACALL DISP1              // routine to display ASCII character in A
MOV A, # '2'             // display '2' for 2014
```

```
            ACALL DISP1          // routine to display ASCII character in A
            MOV A, # '0'         // display '0' for 2014
            ACALL DISP1          // routine to display ASCII character in A
            MOV R1,#09H          // initialize R1 as a pointer to the year register (address 09)
            MOVX A, @R1          // read year into A
            ACALL DISPLAY        // display year on LCD
DISPLAY:    ACALL BCD2BIN        // subroutine for DISPLAY
            …
            …
            …
DISP1:      …
```

21.11.5 Square Wave Output

The DS12887 can be programmed to generate the square wave of different frequencies at the SQW pin. The frequency of the square wave is selected by bits $RS_3RS_2RS_1RS_0$ (bits D_4 to D_0) of Register A as shown in Table 21.5. The square wave can be used to sound a buzzer for alarm applications. The generation of the square wave is illustrated in Example 21.18.

Table 21.5 *Frequency and periodic interrupt rate selection using Register A (bits D_4 to D_0)*

RS3	RS2	RS1	RS0	Periodic interrupt rate	Frequency		RS3	RS2	RS1	RS0	Periodic interrupt rate	Frequency
0	0	0	0	None	None		1	0	0	0	3.90625 ms	256 Hz
0	0	0	1	3.90625 ms	256 Hz		1	0	0	1	7.8125 ms	128 Hz
0	0	1	0	7.8125 ms	128 Hz		1	0	1	0	15.625 ms	64 Hz
0	0	1	1	122.070 ms	8.192 KHz		1	0	1	1	31.25 ms	32 Hz
0	1	0	0	244.141 ms	4.096 KHz		1	1	0	0	62.5 ms	16 Hz
0	1	0	1	488.281 ms	2.048 KHz		1	1	0	1	125 ms	8 Hz
0	1	1	0	976.5625 ms	1.024 KHz		1	1	1	0	250 ms	4 Hz
0	1	1	1	1.953125 ms	512 Hz		1	1	1	1	500 ms	2 Hz

Example 21.18

Write a program to generate the square wave of 8.192 KHz on SQW pin of DS12887.

Solution:

The frequency of the square wave can be generated using $RS_3RS_2RS_1RS_0$ (bits D_4 to D_0) of Register A. For frequency of 8.192 KHz, $RS_3RS_2RS_1RS_0$ = 0011. To generate the square wave, SQWE (Square Wave Enable, bit 3 of Register B) bit must be programmed to 1.

```
MOV R1, #10          // initialize R1 as a pointer to A register (address 10 or 0AH)
MOV A, #23H          // select frequency and turn on the oscillator
MOVX @R1, A          // send the command to Register A
MOV R1, #11          // initialize R1 as a pointer to B register (address 11 or 0BH)
MOVX A,@R1           // read Register B into A
ORL A, #08H          // SQWE = 1 to enable square wave generation, don't disturb other bits
MOVX @R1, A          // start the square wave generation
```

21.11.6 Alarms

The DS12887 can be used to generate the alarms. The alarm can be generated once-per-day, once-per-hour, once-per-minute and once-per-second. For once-per-day alarm, we need to write the desired time into the respective alarm locations (address 01: seconds alarm, address 03: minute alarm, address 05: hours alarm), when actual time (RTC time, hh:mm:ss)

and the time set by the alarm locations match, the AF bit (bit 5 in Register C) is set to high (AF = 1) to indicate the alarm signal. This bit can either be monitored by polling method or can be programmed to generate the interrupt on the IRQ pin of the DS12887. The IRQ pin can be used for alarm interrupt only if AIE bit (bit 5 in Register B) is high.

For once per hour alarm, we have to set the two most significant bits (bit 6 and 7) of hour alarm location to 1. (When bits 6 and 7 are set to 1, the byte is referred as "don't care" code; any value between C0 to FF can be used as "don't care" code). For once per minute alarm, we have to write "don't care" code to minute alarm as well as the hour alarm locations (address 03 and 05). Similarly, for once-per-second alarm, we have to write "don't care" code to hour alarm, minute alarm and seconds alarm locations (addresses 01, 03 and 05). The program to use the alarm feature is illustrated in Example 21.19.

Example 21.19

Design a simple system using DS12887 to generate an alarm using a buzzer. Configure an alarm in once-per-day mode and set the alarm time as 05:30:00 a.m. The buzzer should sound for 10 seconds when the alarm is generated every day at a specified time.

Solution:

The simplified circuit for the required system is shown in Figure 21.19. The IRQ pin of the DS12887 is connected to the external interrupt 1 pin of the 8051. The square-wave signal from SQW pin of the DS12887 is used to sound the buzzer. Set the desired alarm time in the alarm at respective locations. When RTC time and alarm time matches, the IRQ will be generated, this signal is used as an external interrupt to the 8051. The Interrupt service routine should generate the square wave on SQW pin for 10 seconds and should return to the main program.

Assume that the RTC is already in use, (i.e.) oscillator is ON and the time and calendar is already programmed earlier. Also, routine of 10 second delay is available.

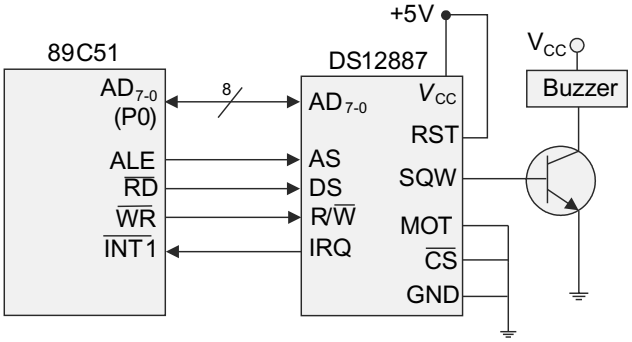


Fig. 21.19 *Alarm system using DS12887*

```
        ORG 0000H
        LJMP MAIN           //skip the Interrupt Vector Table

        ORG 0013H           // ISR for external interrupt 1 to generate square wave for 10 s
        MOV R1, #10         // initialize R1 as a pointer to A register (address 10 or 0AH)
        MOV A, #23H         // select the frequency and turn on the oscillator
        MOVX @R1, A         // send the command to Register A
        MOV R1, #11         // initialize R1 as a pointer to B register (address 11 or 0BH)
        MOVX A,@R1          // read Register B into A
        ORL A, #08H         // SQWE = 1 to enable square wave generation, don't disturb // other bits
        MOVX @R1, A         // start the square wave generation
        LCALL DELAY_10S     // delay of 10 seconds
        MOVX A,@R1          // read Register B into A
        ANL A, #0F7H        // SQWE = 0 to stop the square wave generation
        MOVX @R1, A         // stop the square wave generation
        RETI                // return to the main program

        ORG 100H            // main program
                            // configure the time mode
MAIN:   MOV R1, #11         // initialize R1 as a pointer to Register B (address 11 or 0BH)
        MOV A, #81H         // configure command word for 12 hr BCD mode with daylight
                            // savings, SET bit  = 1 to prevent updates during initialization
        MOVX @R1, A         // send the above command to Register B
```

```
                    // set the time
        MOV R1, #01     // initialize R1 as a pointer to seconds alarm register (address 01H)
        MOV A, #00H     // load seconds alarm value  = 00H into A
        MOVX @R1, A     // send seconds alarm value to seconds alarm register (01H)
        MOV R1, #03     // initialize R1 as a pointer to minutes alarm register (address 03H)
        MOV A, #30H     // load minutes alarm value  = 30H into A
        MOVX @R1, A     // send minutes alarm value to the minutes alarm register (03H)
        MOV R1, #05     // initialize R1 as a pointer to hours alarm register (address 05H)
        MOV A, #05H     // load hours alarm value  = 05H into A, Bit7 = 0 for AM
        MOVX @R1, A     // send hours alarm value to the hours alarm register (05H)
                    // set AIE bit to enable the alarm interrupt
         MOV R1, #11    // initialize R1 as a pointer to B register (address 11 or 0BH)
        MOVX A,@R1      // read Register B into A
        ORL A, #20H     // AIE = 1 to enable alarm interrupt, don't disturb other bits
        ANL A, #7FH     // SET = 0 to allow the update
        MOVX @R1, A     // load above command in B register
HERE:   SJMP HERE       // wait here for the alarm interrupt to occur.
```

21.11.7 Periodic Interrupts

Similar to alarm signals (once per day,…, once per second), we can also generate IRQ signal at the rate (interval) that can be selected by $RS_3RS_2RS_1RS_0$ (bits D_4 to D_0) of Register A as shown in Table 21.5. To generate the IRQ signal using periodic interrupt, we have to set PIE bit (bit 6 of Register B). When this interrupt is generated the PF bit (bit 6 in Register C) is set, this will set IRQF bit and that in turn will generate IRQ. This feature can be used to generate the interrupts once every 500 ms to once every 122 μs. (refer Table 21.5). The following program fragment configures DS12887 to generate IRQ every 500 ms.

```
MOV R1, #10       // initialize R1 as a pointer to A register (address 10 or 0AH)
MOV A, #2FH       // select the periodic interrupt interval as 500 ms
MOVX @R1,A        // send the command to Register A
MOV R1, #11       // initialize R1 as a pointer to B register (address 11 or 0BH)
MOVX A,@R1        // read Register B into A
ORL A, #40H       // PIE = 1 to enable periodic int generation, don't disturb other bits
MOVX @R1, A       // start the periodic interrupt generation.
```

21.11.8 Update Cycles

The DS12887 performs an update cycle (to update time and calendar) every second irrespective of the SET bit in Register B. This allows RTC to maintain the time accurately irrespective of reading or writing the time and calendar registers. When the SET bit = 1, the user copy of the time, calendar, and alarm bytes is frozen and will not be updated. (The time, calendar and alarm bytes are double buffered). If we read the time (or calendar) during an update cycle, we may not get the exact time (this is very less probable).The above problem can be avoided by the following methods.

UIP (Update In Progress) bit in Register A indicates whether the update cycle is in progress or not. This bit becomes high once per second and once it becomes high, the update cycle starts 244 μs later. This means if we read this bit, and it is 0, we have a minimum of 244 μs before the actual update takes place. Therefore, the program should read/write the time and calendar during this time only.

The UF (Update-ended Interrupt Flag) bit (bit 4 in Register C) is set after each update cycle (every second). If UIE bit (bit 4 in Register B) = 1, UF = 1 will set IRQF bit to 1, which will generate the IRQ. The generation of IRQ indicates that there is around 999 ms to read the valid time. UF is cleared by reading Register C.

21.11.9 Interrupt Sources

As discussed in the above topic, there are three sources of interrupts: Alarm Interrupt (AF bit), Periodic Interrupt (PF bit) and Update-Eend Interrupt (UF bit). To enable these interrupts sources to generate the IRQ, bits AIE, PIE and UIE must be set to 1 respectively. The programmer should select the appropriate source to generate the IRQ by programming these bits.

IRQ (or IRQF flag) = (PF · PIE) + (AF · AIE) + (UF · UIE)

POINTS TO REMEMBER

- ROM usually contains the programs (system software) and permanent system data, while RAM stores the temporary data.
- Address input signals (address lines) are used to identify one memory location out of N locations.
- Each memory chip has one or more inputs like Chip Select ($\overline{CS}$) or Chip Enable ($\overline{CE}$) that selects or enables the memory chip.
- A RAM has one or two control inputs referred as R/$\overline{W}$ or $\overline{WE}$ (or $\overline{W}$) and $\overline{OE}$.
- DRAM is slower compared to SRAM because a refreshing is required in DRAM to preserve the data.
- DRAM has a higher packing density.
- The process of selecting the correct memory location from the correct memory chip is known as address decoding.
- Address lines (A15-A0), data lines (D7-D0), $\overline{PSEN}$, $\overline{ED}$, $\overline{WR}$ and $\overline{WA}$ signals of the 8051 are used for memory interfacing.
- $\overline{PSEN}$ of the 8051 is normally connected with $\overline{OE}$ of the program memory to access the code bytes.
- $\overline{RD}$ and $\overline{WR}$ signals from the 8051 are used to access the RAM.
- The SET bit (bit 7 of Register B) must be programmed to logic 1 before we modify (or initialize) the time, calendar and alarm registers. Once these registers are initialized, the SET bit must be cleared so that they can be updated by RTC chip hardware.
- The time, calendar and alarm registers must use the same data mode at a time.
- The time, calendar, and alarm registers are always accessible because they are double buffered.
- When V_{CC} is applied to the chip and reaches the level of greater than 4.25 V, the chip can be accessed only after 200 ms.
- The alarm in DS12887 can be generated once per day, once per hour, once per minute and once per second.
- There are three sources of interrupts in DS12887: Alarm Interrupt (AF bit), Periodic Interrupt (PF bit) and Update-End interrupt (UF bit).

OBJECTIVE QUESTIONS

1. The total amount of external code memory that can be interfaced with the 8051 is,
   (a) 32 K (b) 64 K (c) 128 K (d) 256 K
2. The I/O ports that are used as address and data bus for external memory are,
   (a) Ports 1 and 2 (b) Ports 1 and 3 (c) Ports 0 and 2 (d) Ports 0 and 3
3. Select the odd one:
   (a) $\overline{RD}$ (b) $\overline{WR}$ (c) $\overline{PSEN}$ (d) $\overline{OE}$
4. $\overline{CS}$ for the memory chips can be generated with the help of,
   (a) decoder (b) PAL logic (c) discrete gates (d) all of the above
5. The number of address lines for a 4096 × 8 memory chip is,
   (a) 14 (b) 12 (c) 10 (d) 8
6. Which of the following memory must be refreshed periodically?
   (a) EPROM (b) DRAM (c) SRAM (d) NV-RAM
7. Which of the following signals must be used in fetching data from external data ROM?
   (a) $\overline{RD}$ (b) $\overline{WR}$ (c) $\overline{PSEN}$ (d) both (a) and (c)
8. To interface the external EPROM memory, it is necessary to de-multiplex the address/data lines of the 8051.
   (a) True (b) False
9. Which of the following signals must be used in fetching data from the external data RAM?
   (a) $\overline{RD}$ (b) $\overline{WR}$ (c) $\overline{PSEN}$ (d) both (a) and (b)
10. Which type of memory has the highest packing density?
    (a) ROM (b) SRAM (C) DRAM (d) Flash

11. $\overline{\text{PSEN}}$ of the 8051 is normally connected with_____ signal of program memory to access code bytes.
    (a) $\overline{\text{CS}}$ (b) $\overline{\text{WE}}$ (c) R/$\overline{\text{W}}$ (d) $\overline{\text{OE}}$
12. IRQ in DS12887 can be generated by,
    (a) AF bit (b) PF bit (c) UF bit (d) all of the above

Answers to Objective Questions

1. (b)	2. (c)	3. (d)	4. (d)	5. (b)
6. (b)	7. (a)	8. (a)	9. (d)	10. (c)
11. (d)	12. (d)			

REVIEW QUESTIONS WITH ANSWERS

1. Which signal is used to inform the 8051 that program is contained within on-chip ROM?

A. $\overline{\text{EA}}$.

2. What are the signals required by the memory devices?

A. Address, data and control signals like chip select, read, write.

3. The given memory chip has 10 address lines and 8 data pins. What is the capacity of chip?

A. The memory chip has 2^{10} = 1024 = 1K locations; each location can hold 8 bits of data because it has 8 data pins. Thus, it has 1K x 8 bits or 1 KBytes.

4. What is the advantage of masked ROM? Where are they used?

A. Their cost per unit is very low when produced in bulk. They are used in a final product when the design of a product is completely verified.

5. Why the static memory is named so?

A. Only DC power has to be applied to retain the data; no other action is required, hence the name static.

6. ROM cannot be used to store program data. True or false.

A. False, they can be used to store permanent data.

7. When are masked ROMs written?

A. Masked ROMs are programmed during the manufacturing process.

8. List the different methods for memory address decoding.

A. Address decoding using discrete logic gates, decoders and using programmable logic.

9. What are the types of address decoders?

A. Linear and absolute address decoders.

10. Which pins of the 8051 are used as control signals for memory chips?

A. $\overline{\text{PSEN}}$, $\overline{\text{WR}}$ and $\overline{\text{RD}}$.

11. In the 8051, when $\overline{\text{EA}}$ = 1, any memory access above address FFFH reads from the external ROM. True or False.

A. True.

12. $\overline{\text{PSEN}}$ is not activated during on-chip memory access. True or False.

A. True.

13. How can ROM be used to store data?

A. $\overline{\text{RD}}$ signal is used to select ROM and MOVX instructions are used to access the stored data.

14. What is the difference between $\overline{\text{PSEN}}$ and $\overline{\text{RD}}$?

A. $\overline{\text{PSEN}}$ is generally used to access the code memory while $\overline{\text{RD}}$ is used to access the data memory.

15. Can we use single ROM chip to store code as well as data?

A. Yes.

16. DRAMs must be refreshed periodically. True or False.

A. True.

17. What is the maximum amount of data memory that can be connected to the 8051?

A. 64 Kbytes, however more memory can be connected using extra port pins.

18. In the 8051, there can be two separate data memories from address 00H to 7FH. Justify true/false with reason.

A. True. Internal RAM as well as external RAM for address 00H to 7FH.

19. The 8051 can address both 64 K Program memory and 64K data memory in the same system. True or False.

A. True.

20. What should be the ending address of 4 KB RAM?

A. 0FFFH.

21. What does OTP stand for?

A. One time programmable.

22. ROM is accessed with _____ signal from 8031.

A. $\overline{\text{PSEN}}$

23. RAM is accessed with _____ or _____ signals from 8031.

A. $\overline{\text{RD}}$ and $\overline{\text{WR}}$

24. How can we generate interrupts of intervals less than 1 s in a DS12887?

A. Using periodic interrupts.

EXERCISE

1. Define the term “memory access time”.
2. Define the term “memory map”.
3. “The more address pins, the more memory capacity”. Under what condition is the statement true?
4. SRAM cell has a larger size compared to DRAM cell. Justify.
5. “ROMs do not have write pin.” Justify.
6. Why are RAMs named so?
7. What are the disadvantages of EPROM compared to EEPROM?
8. Why is the memory address decoding required?
9. Compare absolute address decoders with linear decoders with respect to circuit complexity.
10. Discuss the use of $\overline{\text{PSEN}}$ to access external code memory.
11. Differentiate between MOVC and MOVX instructions.
12. Why is the Flash memory named so?
13. Compare EEPROM and Flash memory with respect to the speed of operation.
14. Why are dynamic RAMs slower? Discuss the benefits of using DRAMs.
15. Explain the operation and construction of DRAM cell.
16. If the starting address of 8Kbyte RAM chip is 1000H. What will be the ending address?
17. If a memory chip is decoded from addresses 8000H to 8FFFH, what is the size of a memory chip?
18. Discuss the operation of NV RAM.
19. What are the advantages of using NVRAM?
20. Write a program to transfer 10H bytes of data from external data ROM to external Data RAM. Assume suitable starting addresses.
21. Write a program to transfer 20Hbytes of data from the internal RAM to external RAM. The starting address of both the memories is 0020H.
22. Write a program to generate the square wave of 1.024 kHz on the SQW pin of DS12887.
23. Write a program to configure an alarm in once per day mode and set the alarm time as 8:15:00 a.m.

22

I2C and SPI Protocols

Objectives

- Introduce the I2C and SPI protocols
- Discuss the signals involved in I2C and SPI protocols
- Discuss the registers involved in I2C and SPI device programming
- Discuss the steps to transmit and receive data using I2C and SPI protocols
- Introduce the P89C66X microcontrollers and its I2C interface
- Introduce the AT89S8252/53 microcontrollers and its SPI interface
- Interface the serial EEPROM with I2C bus
- Interface the two devices using SPI bus
- Develop the programs for data transfer using I2C and SPI devices
- Discuss the applications of I2Cand SPI devices

Key Terms

- Acknowledge
- Arbitration
- Clock Stretching
- Master
- MISO
- MOSI
- SCL
- SCLK
- SDA
- Serial Bus
- Serial EEPROM
- Slave
- START
- STOP
- Wire-AND

22.1 | INTER INTEGRATED CIRCUIT (IIC OR I^2C)

Inter Integrated Circuit, usually referred as I squared C bus, was originally developed by Philips as a control bus for short-distance communication between a small number of devices (microcontrollers and peripherals) on a single board. Now it is a well-recognized standard throughout the semiconductor industry and is used by a majority of chip manufacturers in their devices, i.e. the devices have incorporated I2C interface module (hardware interfacing circuit) to support I2C protocol.

I2C is a synchronous serial bus that uses the two wires, SDA (Serial Data) and SCL (Serial Clock), for communication (half-duplex) between master and slave devices. SDA line is used for bidirectional data transfer and SCL is used to synchronize the data transfer between master and slave device. Since only two wires (two pins of the device) are used for the communication, the size and power consumption of a system is greatly reduced. The master is usually the microcontroller and initiates the data transfer, and the slaves are the other microcontrollers or peripheral devices. The I^2C protocol uses bus-arbitration mechanism allowing the multimaster configuration; this allows only one device to be master at a time. The I^2C bus supports 7-bit as well as 10-bit addressing schemes. All the I2C devices are designed to be able to communicate together on the same two wire bus. I2C is also referred as Two-Wire Serial Interface (TWI).

I2C Bus Features

1. Only two wires are required: a Serial Data Line (SDA) and a Serial Clock Line (SCL).
2. All the devices connected to the bus are individually addressable by software.
3. It is a multimaster bus (no central master) with bus arbitration and collision detection capabilities.
4. Byte-oriented, bidirectional serial data transfers available with one of the three speed modes dependent on the device and clock speeds,
   - Standard mode (transmission speeds up to 100 Kbit/s)
   - Fast mode (up to 400 Kbit/s)
   - High-speed mode (up to 3.4 Mbit/s)
5. Serial clock synchronization allows the devices with different bit rates to communicate.
6. The number of devices that can be connected to the I2C bus is limited only by the maximum bus capacitance of 400 pF.

22.2 | I2C BUS HARDWARE CONFIGURATION

The I^2C bus with peripherals connected to it is illustrated in Figure 22.1.

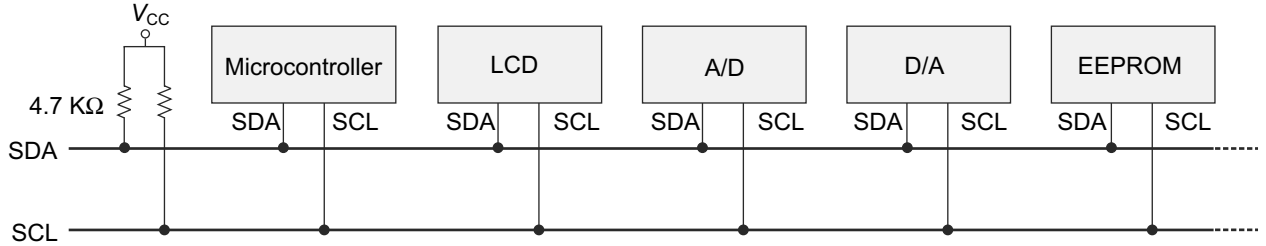


Fig. 22.1 *Basic configuration of the IIC bus*

The two lines, SDA and SCL, of all the I2C devices are open drain (or open collector) and require a pull-up resistor (2 KΩ to 10 KΩ, typically 4.7 KΩ) for each line. The common pull-up for each line gives wired-AND connections between all the devices connected to the bus, i.e. if one or more devices pull the line to the low logic level, the line will be low irrespective of the state of the other devices.

22.2.1 V_{CC} or V_{DD}

Because I2C devices implemented with different technologies (like CMOS, NMOS or bipolar technology) can be interfaced with the I2C bus, the logic levels for LOW (0) and HIGH (1) state are not the same for all, rather it depends on

the corresponding V_{CC} or V_{DD} of the device. Logic level LOW is between 0 to 30% of V_{DD} (V_{IL} is $0.3V_{DD}$) and logic level HIGH is between 70% of V_{DD} to V_{DD} (V_{IH} is 0.7 V_{DD}). The exceptions to this specification are some older devices which have V_{IL} = 1.5 V and V_{IH} = 3.0 V.

THINK BOX 22.1

How to Interface the I2C devices with different logic levels on the same bus?

Because the SDA and SCL signals (pins) are open drain, high-voltage level of the bus is determined by the V_{CC} or V_{DD} (voltage connected to the pull-up resistor). If the I2C devices with different voltage levels are required (for example, master with V_{DD} of 1.8 V, I2C bus at 5 V and a slave devices at 3.3 V), voltage level converters are required to be used between them.

22.2.2 I2C Devices

Before we start the discussion of I2C devices, we need to be familiar with the I2C bus terminology. The common terms are summarized as follows:

Transmitter: The device which sends data on the I2C bus.

Receiver: The device which receives data from the I2Cbus.

Master: The device which begins as well as terminates the data transfer and generates the clock.

Slave: The device which is controlled by a master, i.e. which receives the clock and the data.

Multimaster: The system in which more than one master can exist and may try to control the bus simultaneously without corrupting the data.

The I^2C bus supports 7-bit as well as 10-bit addressing schemes of slaves. With a 7-bit addressing, 112 devices can be connected with the I2C bus (remaining addresses are reserved). Each device is assigned a unique address. The devices connected to the I2C bus are usually referred as *nodes*. In I2C, the master as well as slave can receive or transmit data; therefore, there are four modes of operations. They are Master Transmit, Master Receive, Slave Transmit, and Slave Receive. Remember that each device can operate in more than one mode at different times, but only in one mode at a time.

Discussion Question: Illustrate how an I2C node can operate in different modes.

Answer: When the microcontroller is connected to a serial ADC (or E^2PROM) through an I2C bus, the microcontroller will be in Master Transmit mode to write the data (control byte) into ADC (or E^2PROM), the ADC (or E^2PROM) will be in a Slave Receive mode. When the microcontroller reads the data from the ADC (or E^2PROM), it is in Master Receive mode and ADC (or E^2PROM) is in Slave Transmit mode.

22.3 I2C PROTOCOL

22.3.1 START and STOP Conditions

In the I2C protocol, each transmission begins with a START condition and is terminated by a STOP condition. The START and STOP conditions are always generated by the master. The START and STOP conditions are represented by 'S' and 'P' respectively.

The START condition is generated by a negative (high to low) transition on the SDA line when the SCL is high. The STOP condition is generated by a positive (low to high) transition on the SDA line when the SCL is high. The START and STOP conditions are illustrated in Figure 22.2.

The bus is considered busy by the other devices between START and STOP conditions, and the other master will not try to take the control of the bus during this time. Once data transfer has been started (after generation of START condition), if a master wants to start a new transfer with the other slave (or even the same slave) without generating STOP condition, it generates a new START condition before the STOP condition, this new START condition is referred as REPEATED START condition. It is represented by 'Sr'.

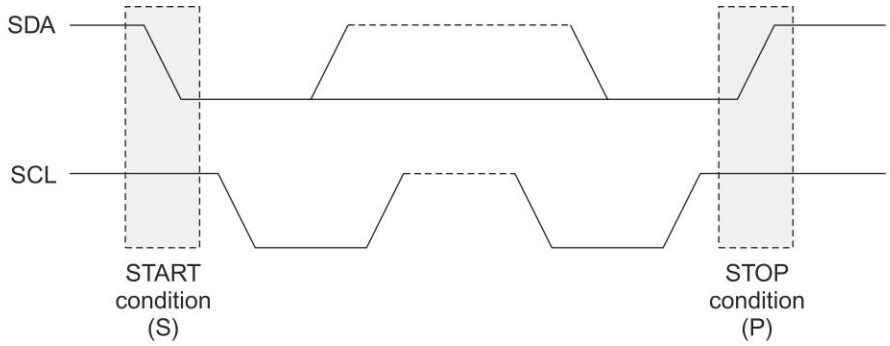


Fig. 22.2 *START and STOP conditions*

22.3.2 Data Validity

In I2C protocol, the data on the SDA line must be stable when SCLK is high. The data (HIGH or LOW state of the SDA line) can change only when the SCL line is LOW. Note that the SDA line is used to transmit data bytes as well as address bytes. The only exceptions to this rule are START and STOP conditions. One clock pulse is required to transmit for each data bit. The valid data states are shown in Figure 22.3.

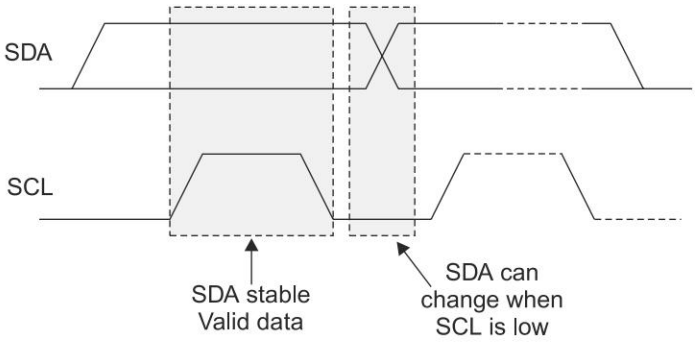


Fig. 22.3 *Valid data states on I2C bus*

THINK BOX 22.2

Why START and STOP conditions do not follow data validity rules?

Since START and STOP conditions must be different from the data (bits of address or data), they do not follow the data validity rules.

22.3.3 Data Transfer Operations on I2C Bus

All the data transfer operations are either write or read operations. During the write operations, the master first transmits the address of the slave and thereafter, transmits the data (master-transmit mode) and the slave receives address as well as data (slave-receive mode) and during read operations the master first transmits the address of the slave, thereafter it receives the data (master-receive mode) and the slave transmits the data (slave-transmit mode). The first byte (8 bits) is placed on the SDA line by the transmitter, and each byte is followed by an Acknowledge bit from the receiver. Therefore, each byte transfer lasts for 9 bits. The number of bytes that can be transmitted between START-STOP condition are unlimited. All the data or address bytes are transferred with the Most Significant Bit (MSB) first. The clock is always generated by the master irrespective of whether it is in transmit or receive mode.

22.3.4 Write Operation

The master starts all the data-transfer operations. Data write operation (transmission) in I2C is started by a START condition, followed by a slave address byte (SLA + R/W), one or more data bytes and each byte is followed by an acknowledge bit, the acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent, finally the transmission is finished by a STOP condition. Figure 22.4 shows a data-write (master-transmit or slave-receive) operation.

The slave address is used to address (select) a specific slave device connected on the bus. The first 7 bits of the slave address byte are 7-bit slave address and eighth bit is READ/WRITE ($R/\overline{W}$) control bit, this byte is referred as (SLA + R/W). If $R/\overline{W} = 0$ (SLA + W), the master will write (transmit) the next data byte to the slave, and if $R/\overline{W} = 1$ (SLA+R)

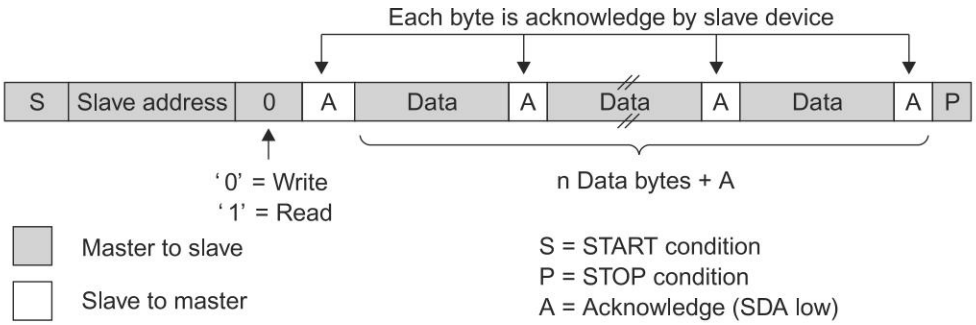


Fig. 22.4 *Data write operation—master-transmit mode*

master will read (receive) the next data byte from the slave. The addressed slave will send Acknowledge bit (A) in the ninth clock (on SCL). The Acknowledge bit is defined as follows: the transmitter releases the SDA line during the 9th clock pulse, and receiver will pull the SDA line LOW (SDA = 0).

After receiving the Acknowledge (A) bit from the slave, the master will transmit one or more data bytes, each followed by A bit from the slave. When the master has sent all data bytes, it will transmit STOP condition to end the data transfer, or a REPEATED START condition to start a new transfer.

If the slave cannot receive address or data because of some reason, SDA remains HIGH during the ninth clock pulse, this is defined as the Not Acknowledge ($\overline{A}$) condition. Then, the master will generate either a STOP condition to end the transfer, or a repeated START condition to start a new transfer. There are five reasons when Not Acknowledge condition is generated. They are:

- When the addressed slave is not present on the bus
- When the slave is busy with doing high-priority activity, for example executing ISR
- When the slave cannot understand command given to it
- When the slave can no more receive or transmit data
- Master receiver signals the end of transfer using $\overline{A}$

Discussion Question: When does the master need to generate a repeated START (Sr) condition?

Answer: In the following conditions, the master will gentare an Sr condition.

1. While communicating with one slave, if the master wants to change the direction of data transfer. For example, if the master is initially writing data to the slave, and after that if the master wants to read from the same slave before releasing the bus.
2. If the next data byte to be read (or written) from the slave is not from the next consecutive address. For example, if the master wants to write one data byte at the address 50H of EEPROM, and the next byte at 60H.
3. When the communication with one slave is completed, and if the master wants to start communication with the other slave.

Example 22.1

Illustrate how the master writes two data bytes 1110 0011 and 0110 0010 to a slave with the address 1010000.

Solution:

The master will write two bytes to the slave in the following steps:

- The master will generate a START condition.
- The master transmits the slave address byte (10100000, SLA+W) on the SDA line (MSB first). The first 7 bits for slave address and 8th bit to indicate the master will write the next byte to the slave.
- The slave pulls SDA line low to generate the Acknowledge (A) condition to indicate that it has successfully received the byte and is ready to receive the next byte.
- After receiving A, the master will transmit the first byte 1110 0011 on the SDA line (MSB first).
- The slave will again generate A to indicate that it is ready to receive the next byte.
- After receiving A, the master will transmit the second byte 0110 0010 on the SDA line (MSB first).

- The slave will again generate A to indicate that it is ready to receive the next byte.
- After receiving A, since the required bytes are transmitted, the master does not want to transfer any more bytes; therefore, it will terminate the transfer by generating a STOP condition.

This operation is illustrated in Figure 22.5.

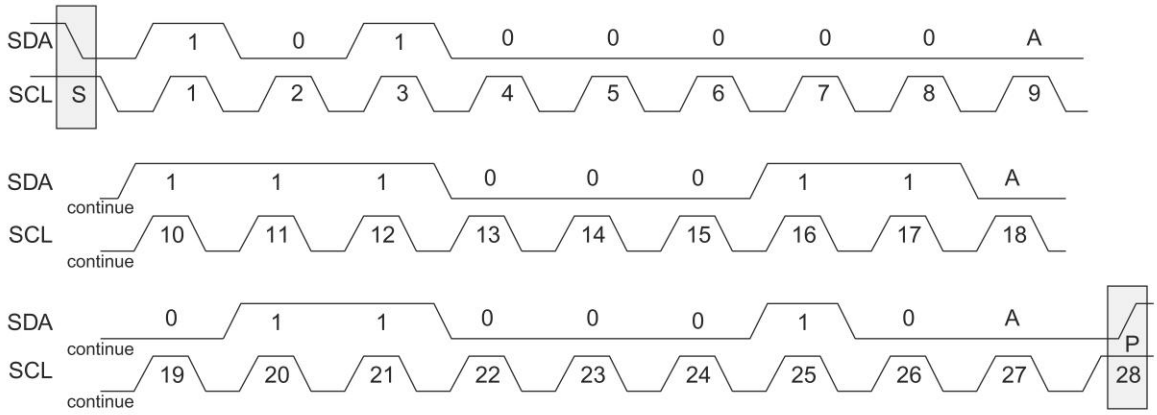


Fig. 22.5 *SDA and SCL status to write bytes 1110 0011 and 0110 0010 to slave 1010000*

22.3.5 Read Operation

The read operation is similar to the write operation except for the direction of the data transfer and the READ/WRITE control bit.

The read operation is summarized in the following steps.

1. Master generates START condition.
2. Master transmits slave address byte (SLA+R) on the SDA line (MSB first). First 7 bits for slave address and the 8^{th} bit to indicate the master will receive the next byte from the slave.
3. The slave pulls SDA line low to generate Acknowledge (A) condition to indicate that it has successfully received the address byte and it will transmit the next byte.
4. After receiving A, the master will become the receiver, and the slave will become the transmitter.
5. Now, the slave transmits a byte on the SDA line, and the master will receive the same.
6. After reception of the byte, the master will generate Acknowledge (A) condition.
7. Steps 5 and 6 are repeated until all the bytes are received.
8. When the master does not want to receive any more bytes, it will generate Not acknowledge condition and thereafter, the master will terminate the transfer by generating STOP condition.

Figure 22.6 shows a data read (master-receive or slave-transmit) operation.

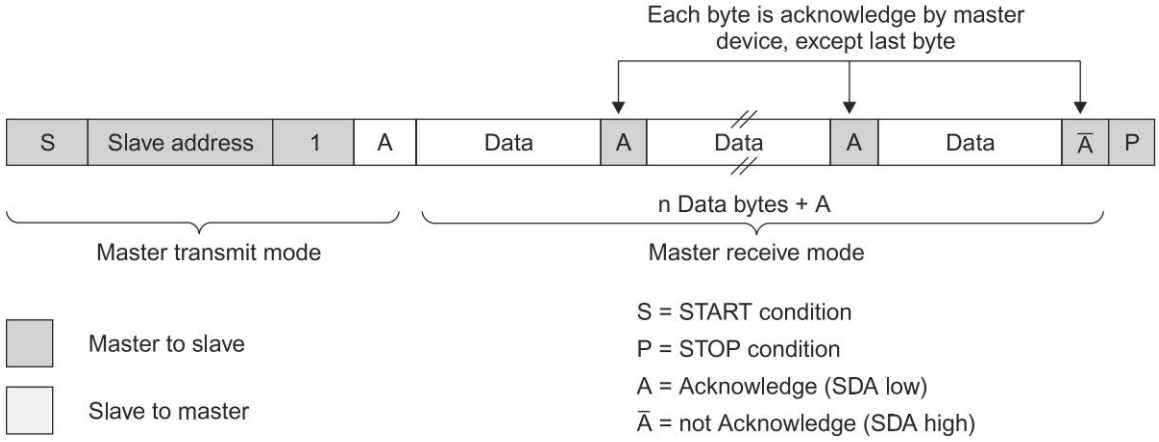


Fig. 22.6 *Data read operation—master-receive mode*

22.3.6 Arbitration and the Clock Synchronization

The I^2C protocol uses the bus-arbitration mechanism which allows the multimaster configuration. Arbitration mechanism allows only one master at a time to take control of the bus.

A master may start a transfer only if the bus is free; therefore, each master will wait until the current transmission is completed and then starts to get the control of the bus. Now, if two or more masters begin a transmission simultaneously, for such situations, arbitration is required to determine which master will have control of the bus and complete its transmission. Arbitration takes place bit by bit. During every bit, each master checks the level of SDA line, and compares it with what it has sent. If these two levels do not match (a master tries to send a HIGH, but detects that the SDA level is LOW), that master will lose the arbitration and, therefore, stops the transmission by switching off its SDA output driver, and will switch to the slave mode. The winning master will continue the data transfer. The arbitration process is shown in Figure 22.7.

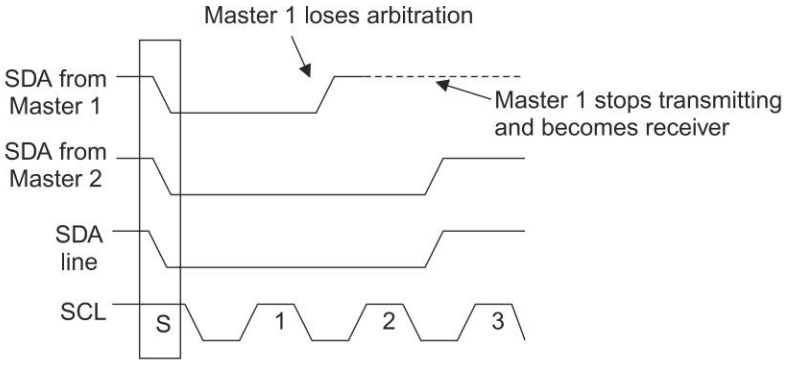


Fig. 22.7 *Arbitration mechanism*

When Master 1 sends a High logic level, Master 2 is sending logic Low; therefore, the SDA line status will be low. Master 1 detects this low logic on the SDA line; therefore, it will lose the arbitration and stop to send the data through SDA line and turn to become a receiver. Note that the arbitration process may continue for many bits, in fact two (or more) masters can finish the whole transmission without error if the data to be transmitted are exactly the same. During the arbitration, no data is lost.

The arbitration process takes place with respect to SCL line. But if the two masters have different SCL time periods then they need to be synchronized because the process requires only one reference clock. Since SCL lines of all the devices are wire-ANDed, the resultant clock (synchronized clock) is generated with its LOW period decided by the master with the longest LOW period, and its HIGH period decided by the master with the shortest HIGH period.

THINK BOX 22.3

What happens if two masters attempt to send the same data to the same address at the same time?

Since both masters want to send the same bits (address + data), the data transfer will be completed without any error. Arbitration is continued throughout the complete data transfer.

22.3.7 Clock Stretching

Clock stretching is the feature used to control the rate of data transfer, i.e. for flow control. If a slave is not ready to transmit or receive any more data until it completes the high-priority function like servicing an ISR, it will extend the clock period by pulling the SCL line to low level after transmitting (or receiving) a byte (or a bit in bit-oriented transfer) of data. The low level on SCL line forces the master in wait state because the master cannot raise the clock to high logic because devices are wire-ANDed on the I2C bus. When the slave is free and ready to transfer the next byte, it will release the SCL line. The operation of clock stretching is illustrated in Figure 22.8.

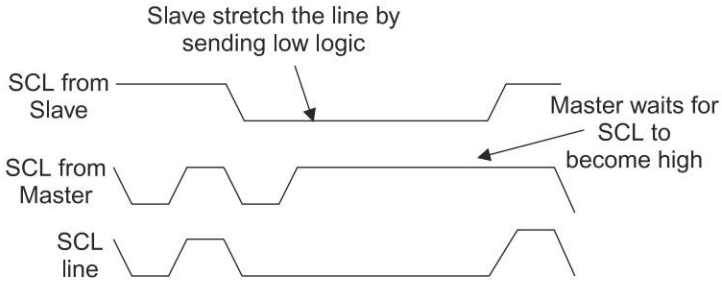


Fig. 22.8 *Clock stretching*

It should be noted that clock-stretching capability is not available with most slave devices because they do not have the SCL driver.

22.3.8 Burst Read/Write Modes

Burst mode is an efficient way of writing (or reading) data to (or from) consecutive locations of the slave. In this mode, the address of the first location is issued by the master, followed by the data for that location. After that, the data bytes are written (or read) to (or from) the successive memory locations. The I2C device (for example EEPROM) automatically increments the address after each data-byte transfer. This process is continued until the data transfer is completed, i.e. STOP condition is issued by the master.

The process of burst writes and read operations for two bytes are illustrated in Figure 22.9. Figure 22.9 (a) shows two data bytes 10H and 11H are written at location 20H onwards in a slave with the address 1010000 (= 0101 0000 = 50H) and Figure 22.9 (b) shows that how two bytes are read from the starting location 20H of a slave with the address 1010000.

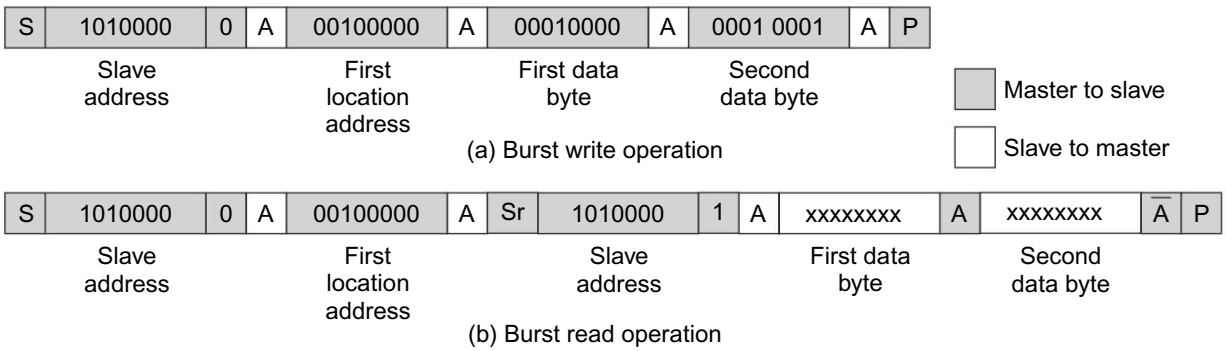


Fig. 22.9 *Burst write and read operations*

22.4 DRIVING THE I2C BUS

There are three ways to drive the I2C bus:

1. **Using a microcontroller having on-chip I2C interface module**: There are two types of on-chip I2C interface modules:
   (i) *Bit-oriented* : The microcontroller is interrupted after every bit transmission.
   (ii) *Byte-oriented* : The microcontroller can be interrupted after every byte transmission.
2. **Using a microcontroller without I2C interface module**: The I2C protocol can be emulated bit-by-bit using the software and any two bidirectional open drain port pins. It is referred as *bit banging*.
3. **Using the microcontroller without I2C interface module and parallel to I2C bus controller chip.**

In this chapter, we will discuss the interfacing of I2C devices with microcontroller having an on-chip I2C interface module. The I2C interface of P89C66x series of microcontrollers is discussed. P89C66x microcontrollers are members of the 80C51 family with enhanced speed compared to the 80C51 microcontroller and have features like:

- 80C51 CPU
- The P89C660/662/664 contains 16KB/32KB/64KB on-chip flash program memory and 512bytes/1KB/2KB RAM
- In-System Programming (ISP) and In-Application Programming (IAP) capability
- 6 clocks/machine cycle (can also be programmed for 12 clocks/machine cycle)
- Speed up to 20 MHz (6 clocks/machine cycle operation) and 33MHz (12 clocks/machine cycle operation) and fully static operation
- Eight interrupts and four interrupt priority levels
- On-chip peripherals like enhanced UART, three 16-bit timers, PCA interface and I2C interface module
- Power control modes**:** Idle and Power-down mode
- Programmable clock out

- Second DPTR register
- Asynchronous port reset
- Low EMI (inhibit ALE)

These microcontrollers are available in 44-pin Plastic Leaded Chip Carrier (PLCC) package and a 44-pin Low Quad Fat Pack (LQFP) package. Out of the 44 pins, only 40 pins are used and 4 pins have no connections.

(Refer datasheets for detailed descriptions of these features.)

22.4.1 I2C Interface Module of P89C66x Microcontrollers

The P89C66x has four I2C SFRs; they are discussed in brief as given below. Refer the datasheets of P89C66x microcontrollers for a detailed explanation.

S1CON: Serial 1 Control (I2C control) Register

This register is read or written by the CPU. It controls the operation of I2C interface module. The bit assignment of S1CON with a brief description is shown in Table 22.1.

Table 22.1 *S1CON register*

CR2	ENSI	STA	STO	SI	AA	CR1	CR0
MSB							LSB

Bit	Symbol	Description
S1CON.7	CR2	Used to define clock speed (see Table 22.2)
S1CON.6	ENSI	Enable serial 1(I2C) interface, ENSI = 1 will enable the I2C interface module, ENSI = 0 will disable the I2C interface module
S1CON.5	STA	STA = 1 will generate a start condition
S1CON.4	STO	STO = 1 will generate a stop condition
S1CON.3	SI	Serial interrupt, SI is set when any of the 25 possible I2C states is entered. If the EA and ES1 bits (of Interrupt Enable register IEN0) are also set, a serial interrupt is requested and the microcontroller is vectored to ISR
S1CON.2	AA	Assert Acknowledge, when AA = 1, an acknowledge (low level on SDA) will be sent by master during the acknowledge clock pulse on the SCL line
S1CON.1	CR1	Used to define the clock speed (see Table 22.2)
S1CON.0	CR0	Used to define the clock speed (see Table 22.2)

The SI bit and the STO bits are affected by the hardware. The SI bit is set by hardware after completion of the current task and the STO bit is cleared by the hardware when the STOP condition is detected on the bus. When SI = 1, the low period of the serial clock on the SCL line is stretched to give sufficient time to software to decide and perform the next operation and during this time, the serial transfer is suspended. When software has completed its task, SI must be cleared by software. Once SI = 0, I2C interface module will start the next operation. Note that all the accesses (address, status and data registers) are done before clearing this flag.

Table 22.2 *I2C clock-speed selection*

6 clocks/machine cycle operation			
CR2	CR1	CR0	I2C bit rate (kHz) = F_{OSC} divided by
0	0	0	128
0	0	1	112
0	1	0	96
0	1	1	80
1	0	0	480
1	0	1	60
1	1	0	30
1	1	1	48 × (256-reload value in Timer 1)

Setting STA bit causes the I2C interface module to enter in the master mode and to generate a START condition if the bus is free, otherwise, the master will wait until the bus is free and then generate a START condition. The STA bit also

generates a repeated START condition when the device is already in the master mode. Setting AA bit to high enables the generation of Acknowledge (low-level on SDA) when the device is in slave or receiver mode.

S1STA: Serial 1 Status (I2C Status) Register

SC4	SC3	SC2	SC1	SC0	0	0	0
MSB							LSB

S1STA is a read-only register, SC4–SC0 shows the status of I2C interface module and the bus, these bits are used to determine whether the last operation was completed successfully or not, and helps decide the next operation. There are 26 possible I2C status codes. When the code is F8H, there is no relevant information available and the SI bit is not set. All other 25 status codes correspond to defined I2C states. SI = 1 indicates that any one of these states is entered. Lower three bits are always zero.

S1DAT: Serial 1 Data (I2C Data) Register

S1DAT register contains the data to be transmitted or the data received. The CPU can read and write to this register (MSB first) while it is not in the process of shifting a byte. Thus, this register should only be accessed when the SI bit is set.

S1ADR: Serial 1 Address (I2C Address) Register

7-bit slave address	GC
MSB	LSB

The first 7 bits represent a 7-bit slave address to which the device will respond when configured as a slave. It is only used to configure the I2C device in a slave mode. In the master mode, this register is not used. When GC = 1, (General Call bit) the general call address (00H) is recognized by the device, otherwise it is ignored.

Discussion Question: Differentiate the terms I2C bus, I2C protocol, I2C interface module and I2C device.

Answer:

I2C Bus: Two common wires (SDA and SCL) to which all the other devices are connected. These wires are used to transfer the data between the devices.

I2C Protocol: Set of rules that govern the data transfer between devices through an I2C bus. The I2C protocol may be implemented in a hardware or software.

I2C Interface Module: The circuitry which implements the I2C protocol in the hardware. I2C interface module is also referred as I2C module or I2C interface.

I2C Device: The device which has an on-chip I2C interface module.

22.4.2 Programming I2C Interface of P89C66x

In this section, programming of the I2C interface in Assembly and C language is discussed. The programs discussed in this section are based on the assumption that there is only one master present on the bus. The programming examples are based on the polling of SI flag and without checking the status register and does not use the I2C interrupts.

Using I2C Interface as a Master

To configure the I2C device in a master mode, we should initialize I2C module, generate a START condition, transmit or receive data and finally a STOP condition.

Initialization of I2C Module

Follow the steps given below to initialize the I2C module:

1. Configure I2C bit rate using CR2-CR0 bits in S1CON register.
2. Enable I2C module by setting ENSI bit (ENSI = 1) in S1CON register.

Generate START Condition

To start the data transmission, we must generate START condition, it is done in the following steps:

1. Set STA bit in S1CON register.
2. Clear SI bit. (If it is already set by the previous operation, it is advised to always clear this bit.)
3. Monitor the SI bit, wait until SI = 1 to make sure that START condition is generated and transmitted completely.

Transmit Data

After generating a START condition successfully, to transmit the data, follow the steps given below.

1. Write data (SLA + W or SLA + R) to S1DAT register.
2. Clear SI bit.
3. Monitor SI bit, wait until SI = 1 to make sure that the data is transmitted completely.
4. If SLA + W (slave address + write) is transmitted, we should write the next byte(s) to S1DAT register and repeat steps 2 and 3 until all the bytes are transmitted.

Read Data

After transmitting SLA + R (slave address + Read), follow the steps given below.

1. Set AA bit to send Acknowledge bit when byte is received (optional).
2. Clear SI bit.
3. Monitor SI bit, wait until SI = 1 to make sure that the data byte is received.
4. Read SADAT register and save at a proper location for future use.
5. Repeat steps 1, 2, 3 and 4 until all the bytes are received.

Generate a STOP Condition

When all the data bytes are transmitted/received, we must generate STOP condition to terminate the data transfer and to release the bus, it is done in the following steps:

1. Set ST0 bit in S1CON register.
2. Clear SI bit.
3. Monitor SI bit, wait until SI = 1 to make sure that STOP condition is generated and transmitted completely.

Example 22.2

Write a program to write 0010 0101(25H) to a slave with the address 1010001.

Solution:

```
        S1CON EQU 0D8H          // S1CON has address D8
        S1DAT EQU 0DAH          // S1DAT has address DAH
        SI BIT 0DBH             // SI has address DBH
        STA BIT 0DDH            // STA has address DDH
        STO BIT 0DCH            // STO has address DCH

        ORG 0000H
        MOV S1CON, #44H         // enable I2C module, set clock speed = FOSC/128,
                                // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
        SETB STA                // set STA to generate a START condition
HERE1:  JNB SI, HERE1           // wait until SI = 1 to conform that START condition is generated successfully
        CLR STA                 // clear START bit, do not generate repeated start
        MOV S1DAT, #0A2H        // send slave address + W to indicate master will write next byte
        CLR SI                  // clear SI
HERE2:  JNB SI, HERE2           // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        MOV S1DAT, #25H         // send data 25H
        CLR SI                  // clear SI
HERE3:  JNB SI, HERE3           // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        SETB STO                // set STO to generate STOP condition
HERE:   SJMP HERE               // loop forever
        END
```

Example 22.3

Rewrite program of Example 22.2 in the C language.

Solution:

The corresponding C language program is given below.

```
#include <reg66x.h>          // include file for P89C66X microcontrollers
void main()
        {
        S1CON = 0x44;        // enable I2C module, set clock speed = FOSC/128,
                             // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
        STA = 1;             // set STA to generate START condition
        while (!SI);         // wait until SI = 1 to conform START condition is generated successfully
        STA = 0;             // clear START bit, do not generate repeated start
        S1DAT = 0xA2;        // send slave address + W to indicate that master will write next byte
        SI = 0;              // clear SI bit
        while (!SI);         // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        S1DAT = 0x25;        // send data 25H
        SI = 0;              // clear SI bit
        while (!SI);         // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        STO = 1;             // set STO to generate STOP condition
        while (1);           // loop forever
        }
```

Example 22.4

Write a program to read a byte from a slave with the address 1010001.

Solution:

```
        S1CON EQU 0D8H       // S1CON has address D8
        S1DAT EQU 0DAH       // S1DAT has address DAH
        SI BIT 0DBH          // SI has address DBH
        STA BIT 0DDH         // STA has address DDH
        STO BIT 0DCH         // STO has address DCH

        ORG 0000H
        MOV S1CON, #44H      // enable I2C module, set clock speed = FOSC/128,
                             // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
        SETB STA             // set STA to generate START condition
HERE1:  JNB SI, HERE1        // wait until SI = 1 to conform START condition is
                             // generated successfully
        CLR STA              // clear START bit, do not generate repeated start
        MOV S1DAT, #0A3H     // send slave address + R to indicate that master will read the next byte
        CLR SI               // clear SI
HERE2:  JNB SI, HERE2        // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        CLR SI               // clear SI
HERE3:  JNB SI, HERE3        // wait until SI = 1 to conform byte is received in S1DAT
        MOV R1, S1DAT        // read a byte from S1DAT and store in R1
        SETB STO             // set STO to generate STOP condition
HERE:   SJMP HERE            // loop forever
        END
```

Example 22.5

Rewrite program of Example 22.4 in the C language.

Solution:

The corresponding C language program is given below.

```
#include <reg66x.h>              // include file for P89C66X microcontrollers
void main()
        {
        unsigned char i;
        S1CON = 0x44;            // enable I2C module, set clock speed = FOSC/128,
                                 // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
        STA = 1;                 // set STA to generate START condition
        while (!SI);             // wait until SI = 1 to conform START condition is generated successfully
        STA = 0;                 // clear START bit, do not generate repeated start
        S1DAT = 0xA3;            // send slave address + R to indicate master will read next byte
        SI = 0;                  // clear SI bit
        while (!SI);             // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        SI = 0;                  // clear SI bit
        while (!SI);             // wait until SI = 1 to conform byte is received in S1DAT
        i = S1DAT;               // read a byte from S1DAT and store in variable i
        STO = 1;                 // set STO to generate STOP condition
        while (1);               // loop forever
        }
```

22.4.3 Interfacing PCF8594C-2 Serial EEPROM

PCF8594 is a 512 byte (512 × 8 bit or 4Kbits) of serial EEPROM with I2C interface. It can be written by the microcontroller and the information retained even after the power is turned off. It is very useful in applications where the data has to be preserved even after the power is turned off. It has the following features:

- Read/write through I2C interface
- 4 Kbits nonvolatile storage organized as 512 × 8 bits
- Single V_{CC} from 2.5 V to 6 V and on-chip voltage multiplier
- Byte-write mode and 8 byte page-write mode
- Sequential as well as random read capability
- 1,000,000 Erase/Write (E/W) cycles
- 10 years data retention time

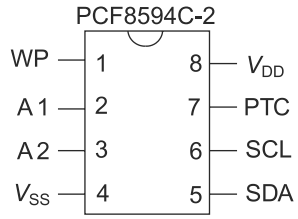


Fig. 22.10 *PCF8594C-2 Pin diagram*

It is in an 8-pin chip available in DIP8 and SO8 package as shown in Figure 22.10. Pin description is of PCF8594C-2 is given in Table 22.3.

Table 22.3 *Pin description of PCF8594C-2*

Pin	Symbol	Description
1	WP	Write Protection input, when WP = 0, write access is allowed to all the 512 bytes When WP = 1, we can write to only lower 256 bytes and upper 256 bytes are write protected
2	A1	Hardware selectable Address input 1
3	A2	Hardware selectable Address input 2
4	V_{SS}	Ground
5	SDA	Serial data line
6	SCL	Serial clock line
7	PTC	Programming time control input, should be connected to $V_{DD,}$ or left unconnected
8	V_{DD}	Supply voltage (2.5 V to 6 V)

Addressing of PCF8594C-2

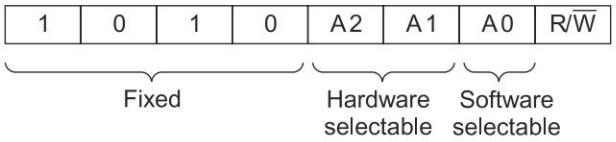


Fig. 22.11 *Format of PCF8594 address byte*

The format of address byte that is to be sent by master device to PCF8594C-2 is shown in Figure 22.11.

The first four bits of the address are fixed and cannot be changed. A2 and A1 bits are hardware selectable, i.e. they can be connected to 1 (V_{DD}) or 0 (V_{SS}). Using these two pins, maximum of four PCF8594 chips can be simultaneously connected to the bus, allowing total 2Kbytes of EEPROM connected with the bus. A0 bit is software selectable. When A0 = 0, the memory chip will access lower 256 bytes (00H – FFH) while A0 = 1 will force the memory chip to access upper 256 bytes (100H – 1FFH).

Address of the internal memory is calculated by A0 sent in the device address byte + second address byte sent by a master. For example, when A0 = 0, and second address byte sent by the master is 10H, it will access the internal address 010H, and when A0 = 1, and the second address sent by the master is 10H, it will access the internal address 110H.

Write Operation

PCF8594 supports byte or page (8 bytes) write operations. It requires a second address byte (First address byte—SLA + W is a device address). This will select one of the 256 bytes from memory depending upon the status of A0 bit sent by master as discussed above. After receiving the two bytes of the address, it will store the third byte at a specified address, the address will be incremented automatically and the next received bytes are stored at consecutive addresses. In a page mode, it can receive 8 bytes in a single transfer (2 address bytes + 6 data bytes). After each byte received, it will respond with Acknowledge automatically. The data transfer should be terminated by the master after the 8th byte with a STOP condition.

Read Operation

Read operations are similar to the write operations with the exception that the LSB of the slave address is 1, i.e. the first address byte is SLA + R. There are three types of read operations: Current Address Read, Random Read, and Sequential Read.

Note that the address counter within a chip will auto increment from FFH to 00H, not FFH to100H. If we want to write the data at the address 100H after writing to FFH, we should send again two address bytes with A0 = 1.

The operation of other EEPROMs from PCF85xx series is the same except for data storage capacity. (For example, PCF8582-256 × 8, PCF8598-1024 × 8)

24xxx and AT24Cxxxx Serial EEPROMs

Other popular serial EEPROM chips are 24xxx series EEPROMs. They are I2C compatible and have the maximum clock frequencies ranging from 100 kHz to 1 MHz and are available with a capacity of 128 bits to 512 Kbits (64 K × 8). AT24Cxxxx EEPROM chips are available from 1Kbyte to 1 MByte storage capacity. The operation and internal organization of these EEPROM is identical to PCF8594 except for the data storage capacity.

Example 22.6

Design a board which interfaces P89C662 microcontroller with PCF8594 EEPROM, through a I2C bus. Write a program (i) to write a string "HAPPY" in the EEPROM at address 050 onwards. (ii) read a string written at 050H and send to Port 2.

Solution:

The interfacing diagram of P89C662 with PCF8594 EEPROM through I2C bus is shown in Figure 22.12.

As shown in Figure 22.12, A1 and A2 pins of EEPROM are grounded, therefore, it will have a slave address 1010 000 (when A0 = 0) or 1010 001(when A0 = 1). WP pin is grounded; therefore, the microcontroller can always write at any address. SCL and SDA pins of both the devices are connected with each other and these two lines are pulled up using 4.7 KΩ resistors.

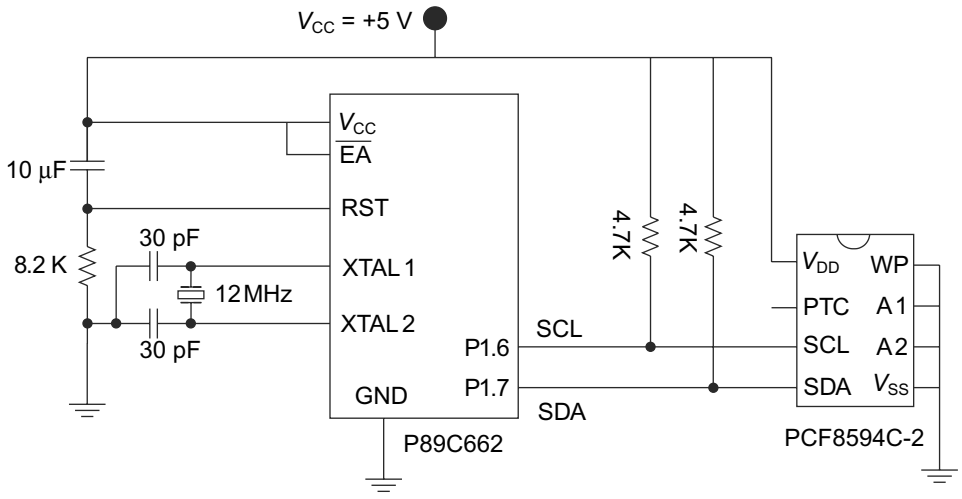


Fig. 22.12 *Interfacing PCF8594 with P89C662 through I2C bus*

The program to write the string "HAPPY" in the EEPROM at address 050 onwards is written below.

```
        S1CON EQU 0D8H          // S1CON has address D8
        S1DAT EQU 0DAH          // S1DAT has address DAH
        SI BIT 0DBH             // SI has address DBH
        STA BIT 0DDH            // STA has address DDH
        STO BIT 0DCH            // STO has address DCH

        ORG 0000H
        MOV DPTR, #0100H        // initialize DPTR with stating address of string
        MOV R2, #05H            // counter to write 5 characters
        MOV S1CON, #44H         // enable I2C module, set clock speed = FOSC/128,
                                // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
        SETB STA                // set STA to generate START condition
HERE1:  JNB SI, HERE1           // wait until SI = 1 to conform START condition is generated successfully
        CLR STA                 // clear START bit, do not generate repeated start
        MOV S1DAT, #0A0H        // send slave address + W to indicate that master will write the next byte
        CLR SI                  // clear SI
HERE2:  JNB SI, HERE2           // wait until SI = 1 to conform that byte in S1DAT is transmitted successfully
        MOV S1DAT, #50H         // send second address byte to write the string at address 050H onwards
        CLR SI                  // clear SI
HERE3:  JNB SI, HERE3           // wait until SI = 1 to conform second address byte is transmitted successfully
NEXT:   CLR A
        MOVC A, @A+DPTR         // read string character
MOV     S1DAT, A                // write string character in S1DAT
        CLR SI                  // clear SI
HERE4:  JNB SI, HERE4           // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
        INC DPTR                // point to the next character in string
        DJNZ R2, NEXT           // transmit 5 characters one by one
        SETB STO                // set STO to generate STOP condition
HERE:   SJMP HERE               // loop forever
        ORG 100H
        STRING DB "HAPPY"
        END
```

Note: The Program to read a string from 050 is given in Second part of example 22.7.

Example 22.7

Rewrite the program of Interfacing Example 22.6 in C Language.

Solution:

The corresponding C program is given below:

```
#include <reg66x.h>
void main()
    {
    unsigned char i, string [] = "HAPPY";
    S1CON = 0x44;              // enable I2C module, set clock speed = F_OSC/128,
                               // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
    STA = 1;                   // set STA to generate START condition
    while (!SI);               // wait until SI = 1 to conform START condition is
                               // generated successfully
    STA = 0;                   // clear START bit, do not generate repeated start
    S1DAT = 0xA0;              // send slave address + W to indicate that master will write next byte
    SI = 0;                    // clear SI bit
    while (!SI);               // wait until SI = 1 to conform byte in S1DAT is
                               // transmitted successfully
    S1DAT = 0x50;              // send second address byte to write string at address 050H onwards
    SI = 0;                    // clear SI bit
    while (!SI);               // wait until SI = 1 to conform second address byte in S1DAT is transmitted successfully
    for (i = 0; i<5;i++)
            {
            S1DAT = string[i]   // write string character in S1DAT
            SI = 0;            // clear SI bit
            while (!SI);       // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
            }
    STO = 1;                   // set STO to generate STOP condition
    while (1);                 // loop forever
    }
```

The program to read a string from the address 050 of the EEPROM is written below.

```
#include <reg66x.h>
void main()
    {
    unsigned char i, j;
    S1CON = 0x44;              // enable I2C module, set clock speed = F_OSC/128,
                               // CR2 = CR1 = CR0 = 0, AA = 1, clear SI
    STA = 1;                   // set STA to generate START condition
    while (!SI);               // wait until SI = 1 to conform START condition is generated successfully
    STA = 0;                   // clear START bit, do not generate repeated start
    S1DAT = 0xA0;              // send slave address + W to indicate that the master will write next byte
    SI = 0;                    // clear SI bit
    while (!SI);               // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
    S1DAT = 0x50;              // send second address byte to read string from address 050H onwards
    SI = 0;                    // clear SI bit
    while (!SI);               // wait until SI = 1 to conform second address byte in S1DAT is
                               // transmitted successfully
    STA = 1;                   // set STA to generate repeated START condition
    while (!SI);               // wait until SI = 1 to conform repeated START condition is generated successfully
    STA = 0;                   // clear START bit
```

```
S1DAT = 0xA1;          // send slave address + R to indicate that master will read next byte
SI = 0;                // clear SI bit
while (!SI);           // wait until SI = 1 to conform byte in S1DAT is transmitted successfully
for (i = 0; i<5; i++)  // read five characters of string
        {
        SI = 0;        // clear SI bit
        while (!SI);   // wait until SI = 1 to conform byte is received in S1DAT
        j = S1DAT;     // read a byte from S1DAT and store in variable j
        P2 = j;        // send received byte to Port 2
        }
STO = 1;               // set STO to generate STOP condition
while (1);             // loop forever
}
```

Using I2C Device as a Slave

To configure I2C device in the slave mode, we should initialize I2C module, monitor the bus condition, transmit or receive the data requested by master device

Initialization of I2C Module

Follow the steps given below to initialize I2C module in slave mode:

1. Write slave address (device own address) in S1ADR register.
2. Enable I2C module by setting ENSI bit (ENSI = 1) in S1CON register.
3. Set AA bit in S1CON register to generate the Acknowledge bit.

Monitor the Bus Condition

Once the I2C module is initialized in a slave mode, the device should monitor the bus to detect whether it is addressed by a master device. When the device is addressed by the master, it responds by generating Acknowledge signal on the bus and sets SI bit. The slave device should monitor the SI bit to decide that it is addressed.

Transmit Data

Once the slave device is addressed for read, follow the steps given below:

1. Write data byte in S1DAT register.
2. Clear SI bit.
3. Monitor SI bit, wait until SI = 1 to make sure that the data is transmitted completely.
4. Repeat steps 1, 2 and 3 until STOP or repeated condition is detected on bus.

Read Data

Once the slave device is addressed for write, follow the steps given below:

1. Set AA bit to send Acknowledge bit when the byte is received (optional)
2. Clear SI bit
3. Monitor SI bit, wait until SI = 1 to make sure that data byte is received
4. Read SADAT register and save at proper location for future use
5. Repeat steps 1, 2, 3 and 4 until STOP or repeated condition is detected on the bus

22.5 | I2C DEVICES

Today I^2C interface is available on-chip in many devices as listed below.

- Microcontrollers (Majority newer variants of 8051, PIC and AVR, ARM, MSPs, MIPs, Power PC, etc.)
- ADCs (MAX 1036/37, MCP3021/22, PCF 8591)
- DACs (MAX521, TDA 8442/44)
- EEPROM/RAM (24Cxxxx, PCF 85xx , SDA5xx, M24Cxx)
- Real-time clocks (PCF8573, PCF 8583)

- Audio/video processors
- LED/LCD drivers
- Temperature sensors
- DTMF generators
- Bus extension/controllers
- General-purpose I/O drivers

22.6 | SERIAL PERIPHERAL INTERFACE

Serial Peripheral Interface (SPI) is a synchronous serial bus standard developed by Motorola capable to transfer the data up to a speed of 10 Mbit/s. SPI is a full-duplex bus, i.e. data is simultaneously transmitted and received. SPI devices communicate in a master/slave configuration where the master device begins the data transfer. Multiple slave devices can be connected with the master with individual slave select (chip select) lines. SPI is also referred as a *four-wire* serial bus or synchronous serial interface SSI. SPI interface is incorporated into many devices such as ADCs, DACs, and EEPROMs from many semiconductor manufacturers.

The SPI bus defines four signals or lines.

MOSI (Master-Out Slave-In): It is a data line on which the data is sent from a master to slave.

MISO (Master-In Slave-Out): It is a data line on which the data is sent from a slave to master.

SCLK (Serial Clock): 50 % duty cycle clock generated by a master to synchronize the data transfer.

$\overline{SS}$ (Slave Select): This signal is generated by a master and used to activate the slave device. It is generally used to initiate and terminate the data transfer. $\overline{SS}$ is optional and may also be implemented using the port pin.

Note that the names of these four signals may vary for a particular device.

The data is transmitted in blocks of 8 bits. It is used for short distance communications. SPI interface is illustrated in Figure 22.13.

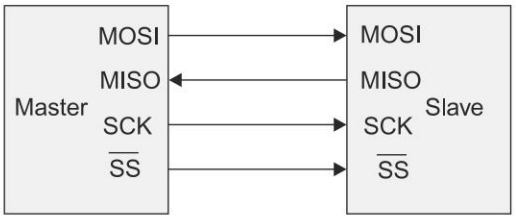


Fig. 22.13 *SPI interface*

The master is usually a microcontroller and the slave can be either microcontroller or any other device.

22.6.1 SPI Operation

The SPI data transfer involves one master device and one or more slave devices. If a single slave is used, the $\overline{SS}$ pin of the slave may be connected to logic low, however, there are some slave devices which require high-to-low transition on this pin to start their operation. For multiple slave devices connected with SPI bus, individual $\overline{SS}$ signal is required from the master for each slave device. Only one slave should be selected at a time to communicate with the master.

SPI data transfer involves two shift registers, one in a master and other in a slave device. The master generates the clock signal for these shift registers. The connection of the master and slave with internal connections of the shift registers and clock is shown in Figure 22.14.

As shown in Figure 22.14, the MOSI pin of master is connected to MOSI pin of the slave and MISO pin of the slave is connected with MISO pin of master. This way, the two shift registers form a ring in which the data flows from master to slave and at the same time, other data flows from the slave to master. The master provides the common clock to both shift registers to synchronize the data transfer. To initiate a data transfer, the master has to configure the clock frequency less than or equal to the maximum frequency the slave device can support.

During each clock cycle, the following operations are performed simultaneously:

- The master transmits (shift out) a bit on the MOSI line (MSB first), the slave receives it (shift in) from the same line.
- The slave transmits a bit on the MISO line, the master receives it from that same line.

In SPI interface, the size of the shift registers is usually 8 bits; therefore, after 8 clock pulses, the contents of the two shift registers are exchanged. Then, each device reads a new data byte received, and does something with it, like storing in the

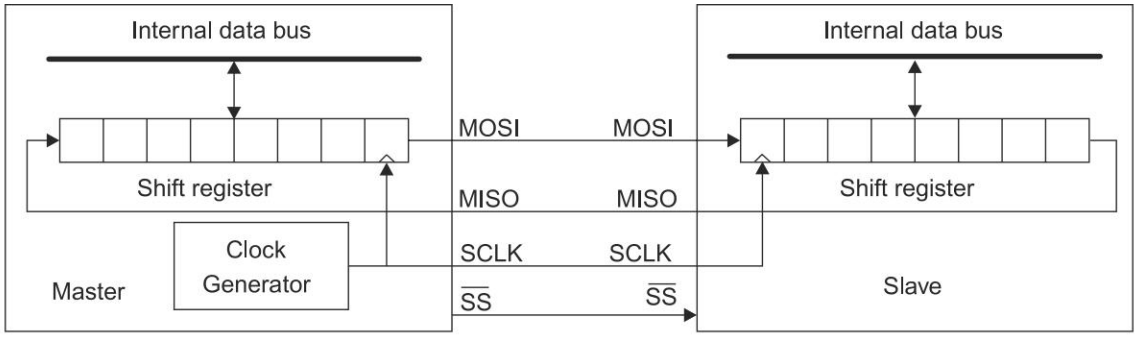


Fig. 22.14 *SPI Internal Block Diagram*

memory or takes some action based on the received byte. Note that when the master transmits a byte, it must receive a byte from slave, even if the received byte may not be useful.

For transmission of a byte, the master writes the byte in its shift register and generates 8 clock pulses. After 8 clock pulses, the byte will be transmitted to the slave shift register. If there are more bytes to transmit, the shift registers are loaded with the new bytes and the process is repeated. For the reception of a byte (by the master), the slave will place the byte in its shift register, and after 8 clock pulses, the data will be received in the master shift register. During the entire data transfer, the $\overline{SS}$ must remain low.

For some slave devices like EEPROMs, the master has to first transmit the commands (for example, for AT250xx SPI serial EEPROMs, the commands are write enable, reset write enable, read/write status register, read data, write data) and address of the location from where to access the data. Then the actual data transfer will be started.

22.6.2 Clock Polarity and Phase in SPI Device

In addition to setting the clock frequency, the master and slave must be configured to operate with the same clock polarity (usually referred as COPL) and clock phase (CPHA) with respect to the data.

The timing diagram to show the status of the clock and data lines for all the combinations of CPOL and CPHA is shown in Figure 22.15.

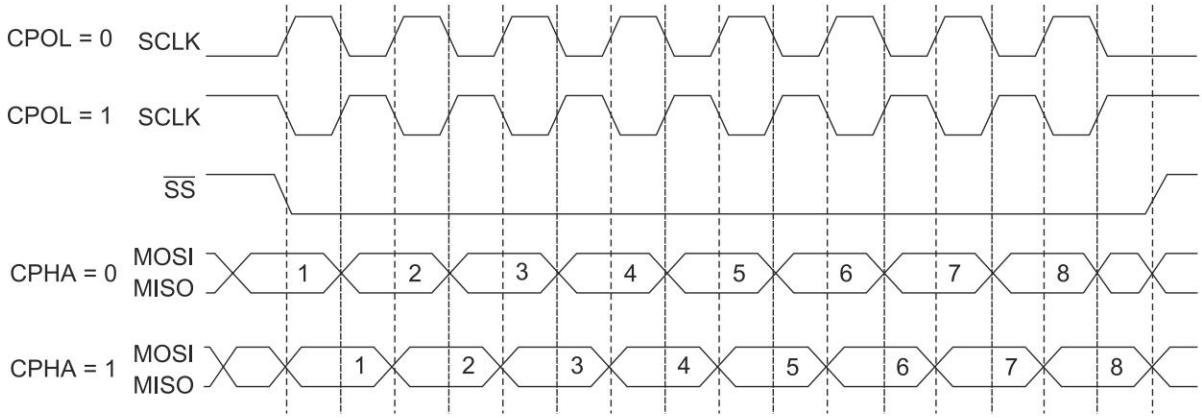


Fig. 22.15 *Clock phase and polarity*

When CPOL = 0, the idle value of the clock is zero.

- For CPHA = 0, the data is sampled (received) on the rising edge (low-to-high transition) of the clock and the data is transmitted on a falling edge (high-to-low transition).
- For CPHA = 1, data is sampled on the falling edge of the clock and the data is transmitted on a rising edge.

When CPOL = 1, the idle value of the clock is one.

- For CPHA = 0, the data is sampled on the falling edge of the clock and the data is transmitted on a rising edge.
- For CPHA = 1, the data is sampled on the rising edge of the clock and the data is transmitted on a falling edge.

The combinations of polarity and phases of the clock are usually referred as *modes* which are commonly numbered as shown in Table 22.4.

Table 22.4 *Modes in SPI*

CPOL	CPHA	Mode
0	0	0
0	1	1
1	0	2
1	1	3

22.6.3 SPI Bus Configurations

There are two SPI bus configurations. They are independent-slave and daisy-chain configurations. In an independent- slave SPI configuration, each slave is selected by a separate select line, and MOSI and MISO lines of each slave are connected together. Since the MISO pins of all the slaves are tied together, they must be tri-state pins and only one slave should be selected at a time. This is the commonly used configuration. In a daisy-chain, the output (MISO) of the first device is connected with input (MOSI) of the next slave, and so on; finally output (MISO) of the last slave is connected with input (MISO) of the master. This configuration requires common select line. Note that this configuration may not be supported by all the SPI devices. These configurations are shown in Figure 22.16.

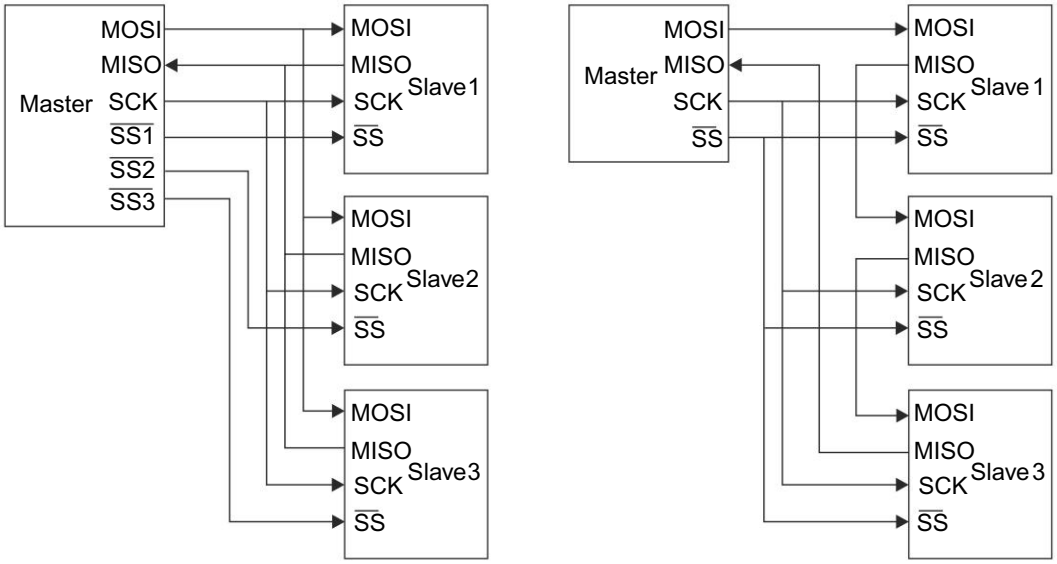


Fig. 22.16 *SPI bus configurations*

The SPI interface module of AT89S825x microcontroller is discussed in the next section.

22.7 | AT89S825X

Atmel's AT89S825x (52/53) microcontrollers are variants of the 80C51 family with the additional features like:

- AT89S8252/53 has 8Kbytes/12K Bytes of on-chip In-System Programmable (ISP) Flash Program Memory (SPI Interface for program downloading)
- 2K Bytes EEPROM Data Memory (Endurance of 100,000 write/erase cycles)
- Fully Static Operation: 0 Hz to 24 MHz (in x1 and x2 Modes)
- Nine-interrupt and four-interrupt priority levels
- Internal Power-on Reset and dual data pointers
- On-chip peripherals like enhanced UART, three 16-bit timers, Enhanced SPI interface module, watchdog timer
- Programmable and Fuseable x2 clock option and Power-off flag
- 2.7 V to 5.5 V operating voltage range

(Refer datasheet of AT89S825x for more details on these features.)

22.7.1 SPI Interface Module of AT89S825x Microcontrollers

The AT89S825x has three SPI SFRs; they are discussed in brief as given below. Refer datasheets of AT89S825x microcontrollers for a detailed explanation.

SPCR: SPI Control Register

It controls the operation of the SPI interface module. The bit assignment of SPCR with a brief description is shown in Table 22.5.

Table 22.5 *SPCR register*

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
MSB							LSB

Bit	Symbol	Description
7	SPIE	SPI Interrupt Enable. SPIE = 1 enables SPI interrupt when the ES bit in the IE register is set (ES = 1), SPIE = 0 disables SPI interrupt.
6	SPE	SPI Enable. SPI = 1 enables the SPI module and connects SS, MOSI, MISO and SCK to pins P1.4, P1.5, P1.6, and P1.7. SPI = 0 disables the SPI channel.
5	DORD	Data Order selection. When DORD = 1, LSB is transmitted first, DORD = 0 MSB is transmitted first during the data transfer
4	MSTR	Master/Slave Select. MSTR = 1 configures the device in master mode. MSTR = 0 configures the device in slave mode.
3	CPOL	Clock Polarity. When CPOL = 1, SCK is high when idle. When CPOL = 0, SCK of the master device is low idle. Refer Figure 22.15
2	CPHA	Clock Phase. The CPHA bit along with the CPOL bit controls the clock and the data relationship between master and slave. Refer Figure 22.15
1	SPR1	SPI clock rate select. These two bits select the SCLK rate of the master device. These bits have no effect on slave. SPR1 SPR0 SCK 0 0 = fosc/4 (fosc/2 in x2 mode) 0 1 = fosc/16 (fosc/8 in x2 mode) 1 0 = fosc/64 (fosc/32 in x2 mode) 1 1 = fosc/128 (fosc/64 in x2 mode)
0	SPR0	

$\overline{SS}$ Pin

When AT89S8253 is configured as a master, SS pin (P1.4) is ignored and can be used for general-purpose input or output operations. However, in the slave mode, SS must be driven low externally to activate the device. When SS is driven high, the slave's SPI module is deactivated.

SPSR—SPI Status Register

The bit assignment of SPSR with a brief description is shown in Table 22.6.

Table 22.6 *SPSR Register*

SPIF	WCOL	LDEN	-	-	-	DISSO	ENH
MSB							LSB

Bit	Symbol	Description
7	SPIF	SPI Interrupt Flag. When a byte transfer is complete, the hardware will set SPIF bit, an interrupt is generated if ES = 1. The SPIF bit is cleared automatically by reading the SPI status register followed by reading/writing the SPI data register.
6	WCOL	If ENH = 0, this bit is write-collision flag. This bit is set if the SPI data register is written during a data transfer. This bit is cleared by reading the SPI status register followed by reading/writing the SPI data register. IF ENH = 1, this bit works in an enhanced mode as Tx buffer full. Writing when WCOL = 1 in an enhanced mode will overwrite the data already present in the Tx Buffer. In this mode, this bit is cleared when the write buffer has been unloaded into the serial shift register.

Contd.

Contd.

5	LDEN	Load enable for the Tx buffer in an enhanced mode. If ENH = 1, it is safe to load the Tx Buffer when LDEN = 1 and WCOL = 0. It is high during bits 0 - 3 and is low during bits 4 – 7 of the SPI byte transmission time frame.
4	–	
3	–	
2	–	
1	DISSO	Disable Slave Output bit. If set, this bit causes the MISO pin to be tri-stated, therefore more than one slave device can share the same interface. Normally, the first byte sent by the master could be the slave address and only the selected slave should clear its DISSO bit.
0	ENH	If clear, SPI is in the normal mode. If set, SPI is in enhanced mode with write double buffering. The Tx buffer shares the same address with the SPDR register.

SPDR – SPI Data Register

The SPDR Register is a read/write register. To write into the SPI shift register to transmit the data, data byte must be written to SPDR, and to read the received data, we must read from SPDR. Writing to SPDR starts the data transmission if SPI module is enabled. We should not write to SPDR until the last byte is transmitted completely, otherwise a collision will happen.

Interfacing Example 22.8

Design a system in which two AT89S8253 microcontrollers are connected with each other through SPI bus, one as a master and the other as a slave. Write a program for master as well as slave in Mode 0 (CPOL = CPHA = 0) to (i) send a byte from master to slave, and (ii) send a string " MASTER" from the master to slave and simultaneously send a string "SLAVES" from the slave to master.

Solution:

The interfacing diagram of two AT89S8253 microcontrollers through SPI bus is shown in Figure 22.17.

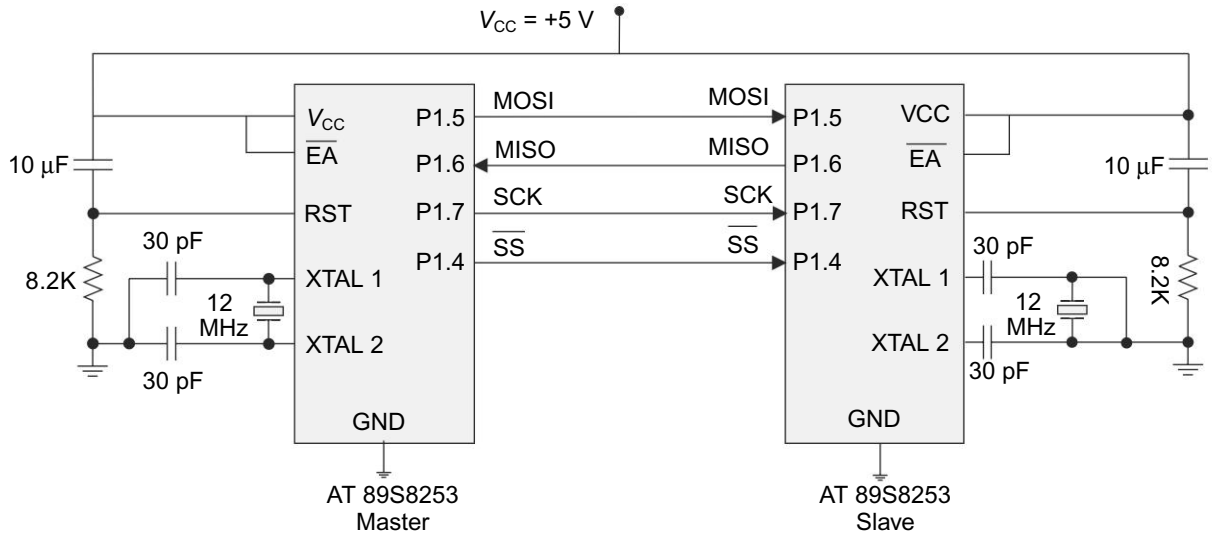


Fig. 22.17 *Connections of two microcontrollers through SPI bus*

Note that the V_{CC} of both microcontrollers may be different. For some microcontrollers, the $\overline{SS}$ pin of the master should be driven high externally or configured as input to detect mode faults. In P896xx microcontrollers, if the $\overline{SS}$ pin of the master is driven low by an external signal, the MSTR bit is cleared and it becomes a slave. Refer datasheet of the SPI device for more details of the $\overline{SS}$ pin.

Program-development Steps (For Master)

1. Configure SPCR register as follows:
   - SPIE = 0, polling of SPIF flag is done to determine the completion of data transmission/reception
   - SPE = 0, all the other bits of SPCR must be set as per the requirement before enabling the SPI module

- DORD = 0, MSB is transmitted first
- MSTR = 1, select the device as a master
- CPOL = CPHA = 0, Mode 0
- SPR1 = SPR0 = 0; clock frequency = F_{OSC}/4 (Assuming X2 bit in CLKREG register is 0)

(Note that the clock phase, polarity and frequency may be varied as per the application requirement.)

2. Enable SPI module by setting SPE (SPE = 1) in the SPCR register.
3. In the master mode, $\overline{SS}$ is ignored and may be used for general I/O; connect this pin (P1.4) or any other free port pin to $\overline{SS}$ of slave to select the slave device. Clear the corresponding port pin to 0 to select the slave device (for this example P1.4 is used).
4. Write the byte to be transmitted into SPDR register.
5. Wait until SPIF = 1 to make sure that the byte is transmitted.
6. When the master is transmitting a byte to the slave, the slave also sends a byte to the master; the received byte will be available in SPDR register, read SPDR register and save to some memory location if desired.
7. For multi-byte transmission, repeat steps 4 to 6 until all the bytes are transmitted.
8. Set the pin used to select slave device(P1.4) to 1 to indicate the end of data transfer.

Program-development Steps (For slave)

1. Configure SPCR register in a similar manner as master except MSTR bit clear MSTR bit to 0 to configure the device as slave.
2. Configure $\overline{SS}$ pin as input so that the master can select the slave device.
3. Since the slave also transmits the byte to master, when the master sends the byte to slave, write a byte into SPDR register if desired.
4. Wait until SPIF = 1 to make sure that byte is received.
5. Read received byte from SPDR and save to some memory location if desired.
6. Repeat steps 3 to 5 until the device is selected by the $\overline{SS}$ pin.

The program for master to transmit a byte is given below:

```
#include<reg8253.h>                  // include file for AT8958253 microcontroller
sbit SS = P1^4;                      // P1.4 is used to select the slave device
void main ( )
        {
        SPCR = 0x10;                 // device as a master, configure clock polarity, phase and frequency
        SPCR | = 0x40;               // enable SPI module
        SS = 0;                      // select the slave device
        SPDR = 0x44;                 // write data byte to be transmitted into SPDR
        while ((SPSR &0x80) = = 0);  // wait until SPIF = 1 to conform that byte is transmitted
                                     // successfully
        }
```

The program for a slave to receive a byte is given below,

```
#include<reg8253.h>                  // include file for AT8958253 microcontroller
void main ( )
        {
        unsigned char i;
        SPCR = 0x00;                 // device as a slave,
        SPCR | = 0x40;               // enable SPI module
        while ((SPSR &0x80) = = 0);  // wait until SPIF = 1 to conform that byte is transmitted successfully
        i = SPDR                     // read a received byte from SPDR and save in memory
        }
```

The program of the master device to transmit a string “MASTER” to slave and receive a string from the slave is given below:

```
#include<reg8253.h>
sbit SS = P1^4;  // P1.4 is used to select the slave device
void main ( )
```

```
{
unsigned char i, transmit_string [] = "MASTER"
unsigned char receive_string [7];
SPCR = 0x10;                    // device as a master, configure clock polarity, phase and frequency
SPCR | = 0x40;                  // enable SPI module
SS = 0;                         // select the slave device
for (i = 0; i<6; i++)           // send (as well as receive ) all characters of string one by one
        {
        SPDR = transmit_string [i];   // write the data byte to be transmitted into SPDR
        while ((SPSR &0x80) = = 0);   // wait until SPIF = 1 to conform that byte is
                                      // transmitted successfully
        receive_string[i] = SPDR;     // read a received string from slave
        }
}
```

The program of a slave device to transmit a string "SLAVES" to master and receive a string from the master is given below:

```
#include<reg8253.h>
void main ( )
        {
        unsigned char i, transmit_string1 [] = "SLAVES"
        unsigned char receive_string1 [7];
        unsigned char i;
        SPCR = 0x00;                    // device as a slave,
        SPCR | = 0x40;                  // enable SPI module
        for (i = 0; i<6; i++)           // receive (as well as transmit) all characters of string one by one
                {
                SPDR = transmit_string1 [i];   // write data byte to be transmitted into SPDR
                while ((SPSR &0x80) = = 0);    // wait until SPIF = 1 to conform that byte is
                                               // received successfully
                receive_string1[i] = SPDR;     // read a received string from master
                }
        }
```

22.7.2 Interfacing MAX512/13 with SPI Bus

MAX512/13 is a triple voltage output 8-bit DACs with SPI interface. The features of these DACs are:

- Three 8-bit DACs (DAC A, DAC B, DAC C)
- Operate from a single +5 V (MAX512) or +3 V (MAX513) Supply, or from bipolar supplies
- Unipolar or bipolar voltage outputs
- Operates up to 5 MHz frequency and compatible with SPI,QSPI and Microwire bus
- Low Power Consumption (1mA operating current, <1μA shutdown current)
- Available in 14-Pin SO/DIP Packages
- Reset Pin and Software Reset

The pin diagram of MAX512/13 is shown in Figure 22.18.

Pin description of MAX512/13 is given in Table 22.7.

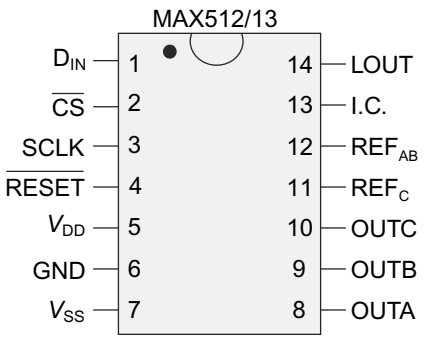


Fig. 22.18 *Pin diagram of MAX512/13*

Table 22.7 *Pin description of MAX 512/13*

Pin	Symbol	Description
1	D_{IN}	Serial digital data Input. It gives data to internal 16-bit shift register.
2	$\overline{CS}$	Chip Select. Enables data at DIN pin to be shifted into the 16-bit shift register. Program commands are executed at the rising edge of CS.
3	SCLK	Serial Clock Input. Data is clocked in on the rising edge of SCLK.
4	$\overline{RESET}$	Asynchronous Reset Input. Clears all the internal registers to their default values (FFH for DAC A and B registers). All other registers are cleared to 0.
5	V_{DD}	Positive power supply (2.7 V to 5.5 V).
6	GND	Ground
7	V_{SS}	Negative power supply 0V or (–1.5 V to –5.5 V). Connect to GND for single supply operation.
8	OUTA	DAC A Output voltage (Buffered)
9	OUTB	DAC B Output voltage (Buffered)
10	OUTC	DAC C Output voltage (Unbuffered)
11	REFC	DAC C reference voltage
12	REFAB	DAC A/B reference voltage
13	I.C.	Internally connected. Keep open
14	LOUT	Logic output (latched)

Serial-Input Data Format and Control Codes for MAX512/13

MAX512/13 requires 16-bit input word (8 bits control + 8 bits data). The format of 16 bit command is shown below.

Control byte								Data byte							
Q2	Q1	SC	SB	SA	LC	LB	LA	D7	D6	D5	D4	D3	D2	D1	D0
Loaded first													Loaded last		

Data is shifted in (clocked) starting with Q2 followed by the remaining control bits and the data byte. The LSB of the data byte (D0) is the last bit clocked into the shift register. Q2 is the uncommitted bit and can be set or reset (don't care). Q1 = 0, will reset LOUT and Q1 = 1 will set LOUT. SC, SB and SA (shut-down bits) are used to shut down the DAC C, B and A respectively, they are active high bits, i.e. Sx = 1 will shut down the corresponding DAC. LC, LB and LA (load bits) are used to select the latch of DAC C, B and A where the next data byte will be stored. They are also active high, i.e. Lx = 1 will load the data in the corresponding latch for conversion. For example, 16-bit word 31E5H (0011 0001 1110 0101) will clear LOUT, shut down DAC B and C, and load 8-bit data E5H in the latch of DAC A for conversion.

22.7.3 Interfacing MAX512/13 with AT89S8253

Interfacing MAX512/13 with AT89S8253 is illustrated in the following example.

Interfacing Example 22.9

Interface MAX512/13 with AT89S8253 and write a program for master (AT89S8253) to generate a sawtooth wave at the output of DAC.

Solution:

The interfacing diagram of MAX512/13 (slave) with AT89S8253 (master) through SPI bus is shown in Figure 22.19.

The MOSI pin of the microcontroller is connected with D_{IN} pin of DAC, the digital data bits are received through this pin. The DAC is selected using P1.1 pin; note that it is not the $\overline{SS}$ pin of the master. The P1.1 is used to select the slave to show that any port pin of the master can be used to select the slave. REF_{AB} and REF_C are connected to V_{CC} to get a full-scale output of 5 V. RESET pins are connected with V_{CC} because

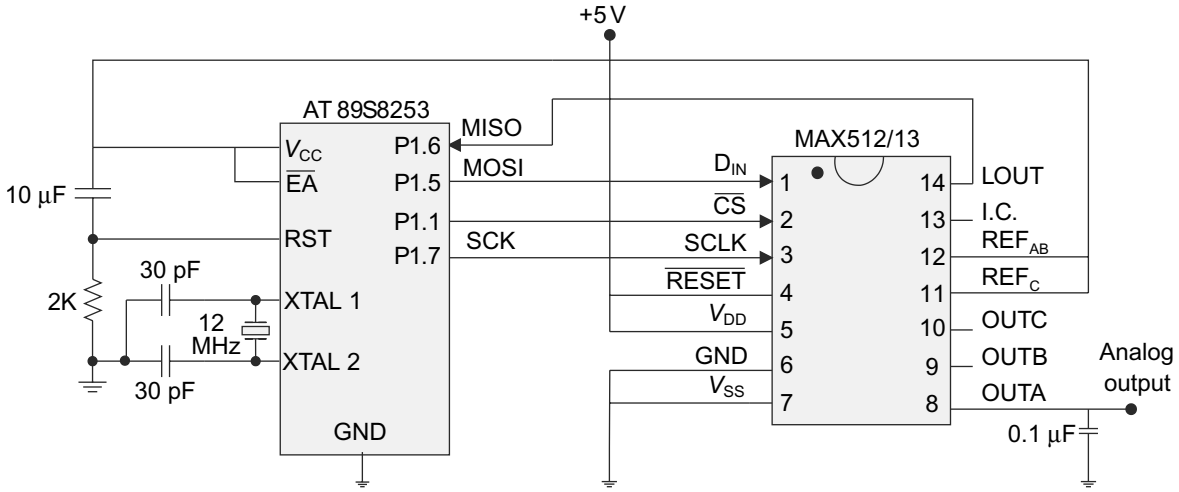


Fig. 22.19 *DAC MAX512/13 interfacing through SPI bus*

we do not want to asynchronously clear the internal latches of DACs. It may be controlled by any port pin of the master if desired. The LOUT pin of DAC is connected with MISO pin of the master, the data received by master through MISO pin is the 16-bit word sent by the master in an earlier cycle. This can be used by the master for auxiliary control, if desired.

Note that the commands in MAX512/13 are executed at the positive edge of CS signal.

Program Development

The steps for the program to generate a sawtooth wave are given below.

1. Configure AT89S8253 in the master mode in SPCR register.
2. Configure the command word to shut down DACs B and C, and load the data in DAC A latch.
3. Initially, the data byte is initialized with minimum value (0).
4. Select the slave device by making P1.1 = 0.
5. Load the command word in SPDR register.
6. Wait until SPIF = 1 to ensure that the command byte is transmitted completely.
7. Load the data byte in SPDR register.
8. Wait until SPIF = 1 to ensure that the data byte is transmitted completely.
9. Increment the data byte (to get sawtooth wave).
10. Make CS of slave by making P1.1 = 1, during the positive edge of CS, MAX512/13 will execute the commands given to it.
11. Repeat steps 4 to 10 continuously to get sawtooth wave continuously.

The corresponding program is given below.

```
#include<reg8253.h>
sbit SS = P1^1;                          // P1.1 is used to select the slave device
void main ( )
        {
unsigned char i = 0;                     // initialize digital data byte with 0
SPCR = 0x10;                             // device as a master, configure clock polarity, phase and frequency
SPCR | = 0x40;                           // enable SPI module
        while (1);                       // generate saw-tooth wave continuously
        {
        SS = 0;                          // select the slave device (enable MAX512/13)
        SPDR = 0x30;                     // write the command byte to be transmitted into SPDR
                                         // only DAC A is operational
        while ((SPSR &0x80) = = 0);      // wait until SPIF = 1 to conform that command byte is
                                         // transmitted successfully
        SPDR = i;                        // write the data byte to be transmitted into SPDR
```

```
        while ((SPSR &0x80) = = 0);     // wait until SPIF = 1 to conform that data byte is
                                        // transmitted successfully
        SS = 1;                         // positive edge at CS will start executing the command in
                                        // DAC
        i++;                            // increment the digital byte to generate saw-tooth wave
        }
}
```

Note that the above program can be modified to generate a sine wave by using the look-up table technique discussed earlier.

22.8 | SPI DEVICES

Today, SPI interface is available on-chip in many devices as listed below.

- Microcontrollers (Majority newer variants of the 8051, PIC and AVR, ARM, MSPs, MIPs, Power PC, etc.)
- ADCs (AD7715, AD7811, MAX144/45, MCP3001/02)
- DACs
- EEPROM/RAM (AT25xxx, NM25C020/40, 25AA040, 25LC040)
- Real time clocks/timers
- Audio/video processors
- MMC or SD cards

22.9 | COMPARISON BETWEEN I2C AND SPI PROTOCOLS

A brief comparison between I2C and SPI protocol and their features is given in Table 22.8.

Table 22.8 *Comparison between I2C and SPI protocols*

Parameter	I2C	SPI
Devices in network	Multiple master, multiple slaves	Single master, multiple slave
Signals	Two : SDA and SCL	Four: MOSI, MISO, SCLK, $\overline{SS}$ (actually 3+ n lines, n based on number of slaves)
Bus length	Limited by 400 pF bus capacitance	Between the boards
Communication type	Half-duplex, MSB is first transmitted	Full-duplex, (MSB or LSB is first transmitted)
Operating voltage	1.8 V to 5.5 V, based on device	1.8 V to 5.5 V, based on device
Addressing scheme	7-bit or 10-bit slave addressing	No specific addressing scheme
Data rate	Standard : up to 100Kbps Fast: up to 400Kbps High speed: up to 3.4 Mbps	Up to 10Mbps, based on device
Message size	8 bits	Flexible, based on device

Note: The programming and interfacing of I2C and SPI modules will remain logically the same for any microcontroller. The only thing that changes across the various microcontrollers is SFR names (and its bit names) corresponding to these modules. Also, the clock-speed selection options may vary.

POINTS TO REMEMBER

- I2C is a synchronous serial protocol supporting half-duplex communication.
- All the I2C devices have open collector (drain) SDA and SCL lines.
- In I2C protocol, the master always initiates a transfer, generates the clock and terminates the data transfer.
- The I^2C bus supports 7-bit as well as 10-bit addressing schemes (7-bit addressing is more common).

- In I2C protocol, the data on the SDA line must be stable when SCLK is high.
- Last bit of the slave address bye (read/ write) is used to indicate the direction of data transfer.
- During the arbitration process, the clock signal having the longest low period is considered as a reference.
- The I2C device (for example, EEPROM) usually increments (internally) the address after each byte transfer until STOP condition is detected.
- The P89C66x microcontroller has four I2C SFRs, S1CON, S1DAT, S1STA and S1ADR.
- SPI is a synchronous serial protocol supporting full-duplex communication.
- The SPI bus specifies four signals, MOSI,MISO,SCLK and $\overline{SS}$.
- SPI data transfer involves the two shift registers, one in the master and other in the slave device.
- In SPI protocol, the master and the slave must be configured to operate with the same clock polarity and phase with respect to the data.
- The AT89S825X microcontrollers have three SPI SFRs, SPCR,SPDR and SPSR.

OBJECTIVE QUESTIONS

1. In I2C protocol, the clock is generated by,
   (a) master (b) slave (c) master or slave based on the direction of data transfer
   (d) master or slave depends upon the device
2. The flow control in I2C protocol is achieved by,
   (a) clock stretching (b) clock synchronization (c) arbitration (d) all of the above
3. START condition on I2C bus is indicated by,
   (a) low-to-high pulse on SDA line while SCL pin is high
   (b) high-to-low pulse on SDA line while SCL pin is high
   (c) low-to-high pulse on SDA line while SCL pin is low
   (d) high-to-low pulse on SDA line while SCL pin is low
4. STOP condition on I2C bus is indicated by,
   (a) low-to-high pulse on SDA line while SCL pin is high
   (b) high-to-low pulse on SDA line while SCL pin is high
   (c) low-to-high pulse on SDA line while SCL pin is low
   (d) high to low pulse on SDA line while SCL pin is low
5. Arbitration is lost by a device which,
   (a) sends low logic first (b) sends high logic first
   (c) has slower speed (d) has faster speed
6. Generation of Acknowledge is controlled by,
   (a) STA bit (b) ENSI bit (c) AA bit (d) CR0 bit
7. In the fast mode, I2C bus supports data rate up to,
   (a) 100 Kbps (b) 400 Kbps (c) 3.4 Mbps (d) 1 Gbps
8. The value of pull-up resistors connected to SDA and SCL lines must be,
   (a) less than 2 KΩ (b) 2 KΩ to 10 KΩ (c) 10 KΩ to 100 KΩ (d) greater than 100 KΩ
9. In I2C protocol, each byte transfer lasts for,
   (a) 7 bits (b) 8 bits (c) 9 bits (d) 10 bits
10. Data register should be accessed when,
    (a) SI = 1 (b) SI = 0 (c) AA = 1 (d) AA = 0
11. In SPI protocol, the clock is generated by,
    (a) master (b) slave (c) master or slave based on the direction of data transfer
    (d) master or slave depends upon the device
12. SPI supports,

(a) asynchronous full-duplex data transfer (b) asynchronous half-duplex data transfer
(c) synchronous full-duplex data transfer (d) synchronous half-duplex data transfer

13. CPOL = 0, CPHA = 1, represents,
(a) Mode 0 (b) Mode 1 (c) Mode 2 (d) Mode 3
14. When CPOL = 0, the idle state of clock is,
(a) high (b) low (c) programmable (d) depend on a device
15. SPI flag is cleared by,
(a) software after SPDR is written (b) hardware after SPDR is read
(c) hardware after reading SPSR followed by reading/writing SPDR
(d) software after reading SPSR followed by reading/writing SPDR
16. SPI is _____duplex and I2C is _____ duplex system.
(a) half, half (b) half, full (c) full, half (d) full, full
17. In which device/s I2C is commonly available?
(a) memory chips (b) DAC (c) ADC (d) all of the above

Answers to Objective Questions

1. (a)	2. (a)	3. (b)	4. (a)	5. (b)
6. (c)	7. (b)	8. (b)	9. (c)	10. (a)
11. (a)	12. (c)	13. (b)	14. (b)	15. (c)
16. (c)	17. (d)			

REVIEW QUESTIONS WITH ANSWERS

1. START condition is generated immediately when STA bit is set. True/False. Justify.

A. False. If the bus is not free, the device will wait until it is released by the other master device; thereafter, the START condition is generated.

2. Which bit is monitored to conform that the byte transmission is completed?

A. SI bit in S1CON (for P89C66X).

3. What is the limit on the number of bytes that can be transmitted in between pair of START and STOP conditions?

A. There is no limit.

4. I2C lines require external pull-up resistors. True/False. Justify.

A. True. Because SDA and SCL pins of I2C devices are open collector (drain).

5. In I2C protocol, each bit of data is followed by an Acknowledge bit. True/False.

A. False. Each byte is followed by an Acknowledge bit.

6. In SPI protocol, 1 byte is transferred per clock cycle. True/False. Justify.

A. False. Only one bit is transmitted per clock cycle.

7. What is the direction of SCLK pin for master and slave device?

A. For the master, SCLK is output and for a slave it is input.

8. Master usually utilizes only three pins of SPI module. True/False. Justify.

A. True. In the master mode, SS pin is not used by SPI module and may be used for general-purpose input/output pin. Note that this is not true for all the devices, it is device dependent. Refer datasheet of SPI device.

9. SPI devices always transfer data in a group of 8 bits. True/False. Justify.

A. False. Usually it is 8 bits, but not limited to 8 bits. SPI is flexible in the choice of bits transferred.

10. Why are MISO pins of slave devices usually tri-state pins?

A. To avoid the bus contention when multiple slaves are connected with MISO pin of master (in independent slave select).

11. Once SPI device is configured as master, can it work as a slave?

A. Yes, but it is device dependent. Some SPI devices (for example, P89C66X microcontrollers) when configured as masters, if SS pin is driven low externally, MSTR or the equivalent bit will be cleared and device will work as a slave.

12. What are the alternate naming conventions for SPI pins?

A. MOSI has alternate names like, SIMO, DI, SDI, DIN, SI.
MISO has alternate names like, SOMI, DO, SDO, DOUT, SO.
SCLK has alternate names like SCK, CLK.
SS has alternate names like nCS, CS, nSS, STE.
Note that the different names are used by different manufacturers for their products.

13. We need not configure the clock rate of an SPI slave device. True/False. Justify.

A. True. Master always provides the clock to the slave.

EXERCISE

1. How is I2C interrupt enabled in P89C662?
2. Which pins of P89C662 microcontroller are used as I2C pins?
3. List the conditions when I2C master device will generate the repeated start condition.
4. Write the interrupt service routine to transmit a byte on I2C bus.
5. What is meant by clock stretching?
6. Discuss how wire-AND connection of all SDA and SCL lines help in bus arbitration.
7. Discuss the arbitration process with the timing diagram.
8. List the conditions when Not Acknowledge is generated.
9. How is SPI interrupt enabled in AT9S8253?
10. How can we configure the SPI device to operate in Mode 3?
11. Write steps to transmit multiple bytes on the I2C bus.
12. Which pins of AT9S8253 microcontrollers are used as SPI pins?
13. List the microcontrollers having I2C module.
14. List the microcontrollers having SPI module.
15. What are the features of AT9S8253 and P89C662?
16. List the devices that support I2C and SPI protocols.
17. Discuss the significance of SPIF, WCOL, DISSO bits of SPSR register.
18. What is the significance of X2 bit of CLKREG register of AT89S8253?
19. How is the clock rate of the SPI device configured?
20. How can more than one slave be connected on the SPI bus?
21. Write interrupt service routine used to transmit a byte on the SPI bus.
22. Write steps to transmit/receive multiple bytes on the SPI bus.
23. Compare I2C and SPI protocols.

23

The 8051 Variants, AVR and PIC Microcontrollers

Objectives

- Discuss the enhanced features of the 8052 microcontroller
- List the 8051 variants and their features manufactured by Atmel, NXP, Dallas semiconductor and Silicon Laboratories
- List and discuss the uses and features of the common on-chip peripherals like CAN, watchdog timer, ADC, DAC, PWM, analog comparator
- Discuss the features of MCS151/251 and MCS 96 family microcontrollers
- Introduce AVR and PIC microcontrollers

Key Terms

- 8032/8052
- 8051 Variants
- Analog Comparator
- AVR Microcontrollers
- Capture Mode
- Controller Area Network
- High-speed Microcontroller
- IAP/ISP
- Inter Integrated Circuit
- MCS 151/251
- MCS 96
- PIC Microcontrollers
- Pulse-width Modulator
- Timer 2
- Watch Dog Timer

The term '*variants*' in this chapter means the 8051 compatible microcontroller. The 8051 has the widest range of variants among all the microcontrollers in the market because Intel has licensed other manufacturers to design the microcontrollers based on the 8051 core, provided that they should remain code-compatible with the original 8051. Today, there are around thousand variants of the 8051 manufactured by more than 20 semiconductor companies. The enhancements contain more memory, different on-chip peripherals and higher operating speeds. The enhanced features are under the control of additional SFRs.

23.1 | THE 8051 ENHANCEMENTS

8032/8052 is an enhanced version of the 8051. They have the following enhancements over the original 8051.

23.1.1 Additional 128 Bytes of On-Chip RAM

There are additional 128 bytes of on-chip RAM from the address 80H to FFH. They have the same address range as SFRs in the 8051. To avoid the conflict in accessing the additional RAM and SFRs, the additional RAM can only be accessed through indirect addressing. Remember that SFRs cannot be accessed using indirect addressing in the 8051 and 8052. The memory map for the internal RAM for the 8052 is shown in Figure 23.1.

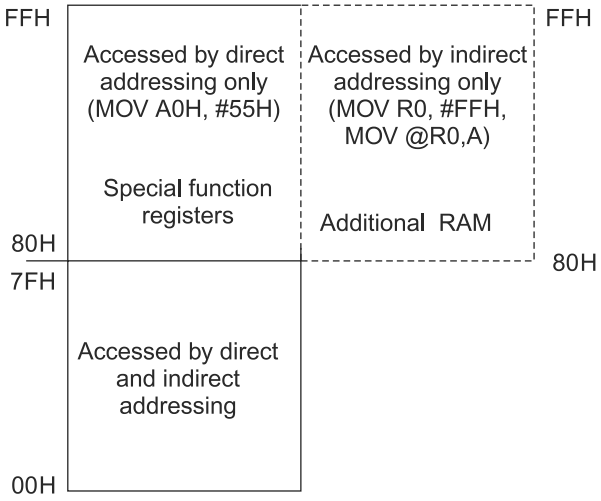


Fig. 23.1 *On-chip RAM of the 8052*

23.1.2 Timer 2

Timer 2 (T2) is a 16-bit timer. It may be programmed as an interval timer as well as an event counter similar to Timer 0 and Timer 1. It may also be used as a clock generator for the UART. This additional clock generator allows the transmitter and receiver to operate at different frequencies.

Timer 2 can also be programmed in 'capture' mode. In this mode, the current value of the timer registers is stored (captured) in its capture registers when there is a high-to-low transition at external input pin T2EX (P1.1 pin). T2 has overflow flag TF2, and additional flag T2EX, which can be set by an external signal at T2EX. T2 may be programmed to generate an interrupt (sixth source of the interrupt). It is controlled and programmed by the five additional SFRs.

In addition to these two enhancements, the 8052 has 8 KBytes of on-chip program memory.

23.1.3 Maximum Clock Speed

Different variants of the 8051/52 can be operated with the clock frequency up to 12, 16, 20, 24, 33 and 40 MHz. The minimum clock speed of 0 Hz indicates the static operation, i.e. the microcontroller has static RAM. (Refer data sheets for the exact value for specific part).

Clocks/Machine Cycle

The 8051 requires 12 clocks per machine cycle. New high-speed and accelerated variants of the 8051 operate at 6, 4, 2 or 1 clocks/machine cycle. For example, P87C768 requires 6 clocks/machine cycle; P89LPC92x requires 2 clocks/machine cycle; DS5000 requires 4 clocks/machine cycle and DS89C420/30/40/50 requires 1clock/machine cycle.

23.1.4 Program Memory Identification

There are no rules defined for naming the 8051 variants. The following points may be helpful to identify the type of program memory. The second digit indicates normally the type of on-chip program memory.

- 0 for mask ROM –8**0**51
- 7 for EPROM/OTP – 8**7**51
- 9 for FLASH – 8**9**C51

The third digit is 3 for ROM-less versions — 80**3**1/80**3**2

23.2 | 8051 VARIANTS FROM NXP (PHILIPS)

The major manufacturer of the 8051 variants is NXP (around 50 variants). With the basic 8051 core, the variants have high-capacity on-chip program memory (up to 32 K/64 K), with peripherals like I^2C bus, 8/10 bit ADCs, CAN Bus, Capture and Compare registers, WDT, PWM, IAP, ISP, additional timer/counter. NXP has few high-speed variants, for example, P89C54X2 microcontroller requires only 6 clocks per machine cycle and P89LPC916/917 requires two to four clocks per machine cycles. Table 23.1 shows some of the variants from NXP.

Table 23.1 *8051 variants from NXP*

	89C51RB2xx	89C51RC2xx/ 89C51RD2xx	89C662/ 89C664	89LPC916/ 89LPC917	89LPC980/89LPC985
On-chip program memory (Flash)	16 Kbytes	32/64 Kbytes	32/64 Kbytes	2 Kbytes	4 K/8 Kbytes
On-chip RAM	256 +256	256+256/768	1/2 KBytes	256 bytes	256/512 bytes
I/O pins	32	32	32	14	26
Timer/Counters	3	3	3	2	7
UART	1	1	1	1	1
Interrupts#	7	7	8	14/13	13
Other features	WDT, ISP, 2 DPTRs, IAP, PWM	WDT, ISP, 2 DPTRs, IAP, PWM	WDT, 2 DPTRs, IAP, ISP, I2C, PWM	WDT, 2 DPTRs, SPI, IAP, ISP, ADC/DAC	WDT, 2 DPTRs, SPI, IAP, ISP, ADC, PWM, RTC, I2C, Analog comparators

23.3 | 8051 VARIANTS FROM ATMEL CORPORATION

Atmel Corporation is another major manufacturer of the 8051 variants. It has introduced flash-memory-based variants at a low cost. The devices have peripherals like ISP, WDT and SPI. The variants are available in 20/40 pins and varying operating voltages from 2.7 to 6 V. Table 23.2 shows some of the variants from Atmel.

Table 23.2 *8051 variants from Atmel*

	89C51/89LV51*	89C52/89LV52*	89C2051 (20 pin)	89C1051 (20 pin)	89S8252/89S8253	89S53
On-chip program memory(Flash)	4 KBytes	8 KBytes	2 Kbytes	1 KBytes	8/12 KBytes	12 KBytes
On-chip RAM	128 bytes	256 bytes	128 bytes	64 bytes	256 bytes	256 bytes
On-chip EEPROM	–	–	–	–	2 KBytes	–
I/O Pins	32	32	15	15	32	32
Timer/Counters	2	3	2	1	3	3
UART	1	1	1	–	1	1
Interrupt Sources#	6	8	6	3	9	9
Lock Bits	3	3	2	2	3	3
Others			Analog comparator		2 DPTR, SPI, WDT	2 DPTR, SPI, WDT

* 89LV51/52 are low-voltage devices, i.e. operating voltage range is 2.7 V to 6 V.
#Sources may have same Vector sources like TI & RI. (Refer datasheets for more details)

23.4 | 8051 VARIANTS FROM DALLAS SEMICONDUCTOR

Dallas has redesigned the 8051 architecture (hardwired in place of micro-coded) and introduced high-speed microcontrollers (HSMs). All the instructions are executed in a single clock cycle (4 clock cycles in some variants) which requires 12 clock cycles in a traditional 8051. Moreover, the devices have additional serial port, WDT, RTC second data pointer, IAP, ISP and NV RAM (battery backed). Table 23.3 shows some of the variants from Dallas Semiconductor.

Table 23.3 *8051 variants from Dallas Semiconductor*

	DS89C430	DS89C450	DS5000	DS87C550	DS8C7520
On-chip program memory	16 Kbyte Flash	64 Kbyte Flash	8 Kbyte NV-RAM	8Kbyte EPROM	16Kbyte EPROM
On-chip RAM	256 Bytes + 1 Kbyte	256 Bytes + 1 Kbyte	128 Bytes	256 Bytes + 1 Kbyte	256 Bytes + 1 Kbyte
I/O pins	32	32	32	55	32
Timer/counters	3	3	2	3	3
UART	2	2	1	2	2
Interrupt sources	14	14	6	15	14
Other features	2 DPTRs, IAP, ISP, WDT	2 DPTRs, IAP, ISP, WDT	IAP, ISP	2 DPTRs, WDT, 10bit ADC (8ch), 8bit PWM (4ch)	2 DPTRs, WDT

23.5 | 8051 VARIANTS FROM SILICON LABORATORIES

Silicon Laboratories' (SiLab) 8-bit mixed signal microcontrollers utilize Silicon Labs' proprietary CIP 51 hardwired microcontroller core. The CIP 51 is fully compatible with the MCS-51 instruction-set and has pipelined architecture which greatly improves its instruction throughput. It executes most instructions (around 70%) in one or two system clock cycles. These devices deliver up to 100 MIPS (Million Instructions Per Second) peak throughput. They have an on-board JTAG debug capability which supports read/write access to memory/registers, breakpoints, watch points, single stepping and free running commands. Table 23.4 shows some of the 8051 variants from Silicon Laboratories.

Table 23.4 *8051 variants from Silicon Laboratories*

	C8051F020 (20 MIPS)	C8051F060 (25 MIPS)	C8051F122* (100 MIPS)
On-chip program memory (Flash)	64 K FLASH	64K FLASH	128K
On-chip RAM	256 bytes + 4 K	256 bytes +4 K	256 bytes +8 K
I/O pins	64	59	64
Timer/counters	5	5	5
UART	2	2	2
Other features	SPI, I2C, 12 and 8 bit ADCs (8ch), on-chip temperature sensor, 12 bit DAC (2ch), PCA	WDT, PCA, SPI, I2C, CAN 2.0B, 16 bit ADC (2ch), 10 bit ADC (8ch), 12 bit DAC (2ch), 3 Analog Comparators, On-Chip Temperature sensor	Capture/Compare, SPI, I2C, RTC, 10 and 8 bit ADCs (8ch), 12 bit DAC (2ch), On-Chip Temperature sensor

*C8051F120/21/22/23/30/31/32/33 and C8051F360/61/62/63/64/65 are the fastest microcontrollers (100 MIPS).

23.6 | COMMON ON-CHIP PERIPHERALS

23.6.1 Watchdog Timer

A Watchdog Timer (WDT) is a peripheral used to monitor the execution of the system software. It resets the microcontroller when program execution is erroneous, i.e. the microcontroller is no longer executing correct and expected sequence of programmed instructions. WDT is a down counter. Once enabled, it starts counting down from some initial value and when the count reaches zero, it resets the microcontroller, thus, reinitializing the system. The system programmer selects the initial value and the system software periodically reloads the value in WDT registers. If the count reaches zero, it is assumed that program is malfunctioning and the system is reset and recovers the system. This way WDT implements automatic recovery mechanism in a system.

WDT is useful when the system is operated in a noisy environment where interference may cause the system to malfunction, for example, the program hangs due to a fault in the interfaced circuit or due to an exception condition. It is also very much helpful when it is difficult to reach the system to reset it manually, like satellites which are operated remotely, and when the system is operating in hazardous environments. The operation of WDT is illustrated in Figure 23.2.

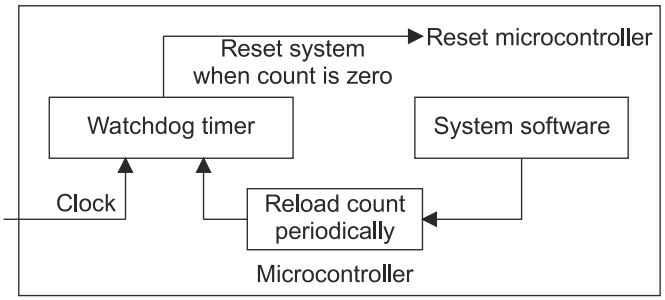


Fig. 23.2 *Watchdog timer operation*

23.6.2 Controller Area Network (CAN)

Controller Area Network (CAN) is an asynchronous serial bus used mainly in real time and distributed embedded applications. It was developed by Robert Bosch Corporation mainly for the automobile applications and now also is widely used in the industrial automation. The CAN bus has two wires, CANH and CANL, usually terminated with resistors at both ends, it can transfer data up to 1 Mbit/s.

The CAN bus has the following features:

- Real-time operation (priorities can be assigned to messages)
- Secure communications (high level of error detection—15-bit CRC messages)
- Faulty devices can disconnect themselves
- Configuration flexibility because of bus arbitration
- High degree of immunity to electromagnetic interference (differential link)
- Easy to add new nodes in the existing network
- Low latency time

23.6.3 Analog Comparator

Analog comparator is an op-amp comparator which compares the unknown analog voltage with reference analog voltage, the output of comparator will be 1 when the unknown voltage is greater than the reference voltage. Refer 'Project: Automatic Street Light Control system' at the end of Chapter 20 for details and application of on-chip analog comparators.

23.6.4 Pulse-width Modulator

A pulse-width modulation can generate the digital signals of varying frequency and duty cycle. A pulse-width Modulator (PWM) changes duty cycle (width) of a pulse from 0 to 100%, thus changing average (DC) value of the pulse from 0 to 100%. It is an effective digital-to-analog conversion technique used commonly in DC motor speed-control applications. Refer topic 20.4.4 and 'Project: DC Motor Speed Control System' at the end of Chapter 20 for more details of PWM and its application to control the speed of the motor.

23.6.5 ADC and DAC

8/10/12/16 bit ADC and DAC are usually available on-chip in many 8051 variants. 1/4/8 channels ADC are commonly available. Refer topic 19.1.9 for more details and programming of on-chip ADC.

23.6.6 Real-Time Clock (RTC)

RTC provides the information of date and time that can be used for many applications. It provides seconds, minutes, hours, day of month, month, year, day of week, and day of the year. Usually, RTC modules are powered separately and are isolated from the rest of the chip, allowing them to operate while the main chip power has been removed. Refer topic 21.11 for more details and programming of RTC.

23.6.7 Other Peripherals and Features

The other common on-chip peripherals are Programmable Counter Array (PCA). The PCA module allows the timing of duration of an event. This module allows getting the current state of some registers which constantly changes its value. It also allows to trigger an external event when a predetermined amount of time has elapsed. Ethernet and USB controllers are also available in the newer variants; these are the protocols for high-speed serial communication. Some of the common features that are available in new variants of the 8051 are discussed as follows.

Multiple DPTRs

Multiple data pointers provide faster and flexible ways to access the data memory. Note that only one DPTR is active at a time. We can switch between two DPTRs using a particular control bit in some SFR. Refer the datasheet of the particular microcontroller in use to find the details of control bit and SFR used to select the DPTR in use. For example, P89C66x, AT89S825x, P89C51RDxxxx, P89LPC92x microcontroller chips have dual DPTRs.

Some variants of the 8051 like C509 (C500 family from Siemens) have 8 data pointers. Refer the C500 microcontroller family datasheet or user's manual for details of how to access the multiple data pointers and advantages offered by them.

Four Levels of Interrupt Priorities

The 8051 has only two levels of interrupt priorities that can be assigned using the IP register. New enhanced variants of the 8051 have four levels of interrupts. These microcontrollers have an extra register for assigning the priorities usually known as IPH register. The priority of an interrupt source can be determined collectively by IPH and IP register. For example, IPH.x = 0, IP.x = 0 will assign the lowest priority (Level 0) to the selected interrupt source and IPH.x = 1, IP.x = 1 will assign the highest priority (Level 3) to the selected interrupt source. The priority scheme for servicing the interrupts is the same as that for the 8051, except there are four interrupt levels. This provides more powerful and flexible interrupt priority assignment compared to the 8051. For example, P89C66x and P89C51RDxxxx microcontrollers have four levels of interrupt priorities.

In-System Programming (ISP)

In-System Programming (ISP) feature allows the microcontroller code memory to be programmed without being removed from the application hardware. The pre-programmed on-chip boot loader provides the interface for the ISP. These chips are programmed using external systems, usually PC, which contains the software to handle programming activities. The programming (usually serial) is further divided into three types, ISP using UART, SPI and JTAG. Refer topic 3.9, 'Loading Program in to Microcontroller' for brief discussion of the types of ISPs. The ISP feature facilitates the remote programming of the microcontroller through the serial link (either using UART or SPI). Refer the datasheet of the respective microcontroller for details of how to perform In-system programming, i.e. programming algorithm and signals involved in the process. ISP is also referred as *serial programming*.

In-Application Programming (IAP)

In-Application Programming (IAP) allows the code memory (Flash) to be programmed by the user application program, i.e. the flash will be programmed during runtime by a set of program instructions. It is performed by calling the low-level subroutines through a common entry point in the on-chip Boot ROM. The Boot ROM contains permanent subroutines (programmed by the manufacturer) that handles the operations of flash programming, the examples of operations are erase block, program byte, verify byte, program security lock bit, etc. The boot ROM is usually separate from the flash memory. The user program calls these subroutines with appropriate parameters to accomplish the desired operation. Note that the user program should be executed from a different memory area than the area being programmed. This feature has many applications, for example, in remote sensing applications, the device can gather some important data in flash which may be used later for the analysis. It can also be used to monitor and store the system parameters which can be later used for diagnosis in case of system failure. This will also allow downloading a new code portion during runtime.

Brown-out-detect, JTAG, on-chip debug, multiple UARTs, low-voltage operation, various power-down modes are the features available in various variants.

23.7 | MCS 151/251 MICROCONTROLLERS

Intel MCS 151/251 microcontrollers are enhancements to the 8051. They have instructions for 16/32 bit data transfer, up to 16 MB RAM + ROM address space, stack up to 64 Kbytes and the instruction executes in two or four clock cycles. MCS 151/251 is based on hardwired architecture with instruction pipelining for enhanced performance. A detailed comparison between MCS 51, MCS 151 and MCS 251 is given in Table 23.5.

Table 23.5 *Comparison between MCS 51, MCS 151 and MCS 251*

Feature	MCS 51	MCS 151	MCS 251
Clocks/Machine cycle	Minimum 12	Minimum 2	Minimum 2
Pipelining	No (Sequential execution unit)	Yes (Pipelined execution unit)	3 Stage (Pipelined execution unit)
PC	16-bit	16-bit	24-bit
Address space (code + data)	64 K+64 K	64 K + 64 K	16MB for both
Interrupts	6	8	9
Interrupt priority levels	2	4	4
I/O pins	32	32	32
Registers	A, R0-R7	A, R0-R7	256 byte register file (16-8 bit, 16-16 bit, 10-32 bit)
Operand size	8-bit	8-bit	8,16 and limited 32 bit
Internal code fetch	8-bit	16-bit	16-bit
Max. stack size	128 Bytes	256 Bytes	64 K
Onchip code memory	4 K	8K(SA)/16K(SB)	16KB(SB)
Page mode	Not supported	Supported	Supported
Wait states	Can not be inserted	Can be inserted	Can be inserted
Instruction set	MCS 51	MCS 51 compatible	Enhanced MCS 51
Performance w.r.t. 8051	1	5 times	Up to 15 times
Others		WDT, PCA	WDT, PCA, Register to register operations, Extended addressing modes, Larger bit addressable space

23.8 | MCS 96 MICROCONTROLLERS

The MCS 96 family members are 16-bit microcontrollers. They are designed to handle high-speed calculations and fast I/O operations efficiently. They have a dedicated I/O subsystem and 16-bit register file based ALU (RALU) with 16-bit multiply and divide capabilities. The MCS 96 family is also referred as 80196. They support bit, byte, word, double-word (unsigned 32 bit), long (signed 32 bit), operations. A brief comparison between MCS 51 and MCS 96 family is given in Table 23.6.

Table 23.6 *Comparison between MCS 51 and MCS 96 families*

Feature	MCS 51 (8 x 51)	MCS 96 (8xC196xx)
CPU	8-bit	16-bit
PC	16-bit	16-bit
Address space (code + data)	64 K + 64 K(Harvard arch.)	64 K for both (Princeton arch.)
Interrupts	6	20
I/O pins	32	40 to 64
Registers	A, R0-R7 (A based ALU)	256 bytes register file (Register file based ALU)
Timers	2 (16 bits)	2 (16 bits)
UART	1	1(dedicated baud rate generator)
On-chip RAM	128 Bytes	Up to 1.5 K (min 232 bytes)
On-chip code memory	4 K	8/16/32/48/56 K
Others		WDT, PWM, 8 channel 10-bit ADC, HOLD/HLDA protocol

23.9 | AVR MICROCONTROLLERS

Alf-Egil Bogen and Vegard Wollan at the Norwegian Institute of Technology originally designed the key architecture of AVR microcontrollers and then it was acquired and further developed by Atmel.

The AVR microcontrollers use Harvard architecture and are designed with RISC philosophy. They are 8-bit microcontrollers (except AVR32). Based on its design philosophy, AVR ***may*** stand for Advanced Virtual RISC. The AVR was one of the first microcontroller families to use on-chip flash as a program memory.

The AVR microcontrollers have the following advantages and features:

- High performance; True RISC architecture
  - High code density
  - C compiler optimized RISC architecture/instruction set
  - Single cycle execution for most instructions
  - One MIPS per MHz up to 20 MHz
  - 32 general-purpose registers
  - Modified Harvard architecture having separate buses for code and data memory
- Low power consumption
  - picoPower™ technology
  - 1.8 to 5.5 V operation
  - Fast wake-up from sleep modes
  - Software controlled frequency
- High integration and scalability
  - Wide range of on-chip peripherals/interfaces
  - Large device range
  - Variety of pin counts
  - Pin/feature compatible families
  - Low system cost
- In-system development
  - In-system programming
  - On-chip debugging
  - In-system verification
- Availability of development tools
  - AVR studio (free)
  - WINAVR C compiler
  - On-chip debuggers
  - Third-party support

The AVR microcontrollers are classified into many families like:

- Classic AVR
- Mega AVR
- Tiny AVR
- Xmega AVR
- Application specific AVR
- AVR with FPGA (FPSLIC)
- AVR32 (32-bit AVR)

Each family has its own features and applications, for example, Tiny AVR microcontrollers have less memory, small size (pin count) and limited peripherals and they are suitable for simpler applications. The Mega AVR has a fairly large amount of memory (up to 256 KB), large pin count (28–100 pins), higher number of peripherals and it is suitable for moderate to complex applications. Application-specific AVRs have special feature and capabilities like LCD controller,

USB controller, CAN controller, Ethernet controller, advanced PWM, Zigbee, automotive and battery management. All AVRs, except AVR32, are 8-bit microcontrollers.

A brief comparison between some members from these families is given in Table 23.7. Since there are many members in each family, the table gives a general comparison between the three subfamilies for each category. The parameters are given in a range instead of a specific value.

Table 23.7 *Comparison between AVR families of microcontrollers*

Parameters	ATtiny	ATmega	ATxmega	AT32UC3A
	ATtiny2x ATtiny4x ATtiny8x	ATmega12x ATmega16x ATmega32x/ATmega64x	ATxmega128xx ATxmega192xx/ATxmega256xx ATxmega64xx	AT32UC3Axxxx AT32UC3Bxxxx AT32UC3Cxxxx
ROM	2 Kbytes	128 Kbytes	128 Kbytes	64–256 Kbytes
	4 Kbytes	16 Kbytes	192 Kbytes/256 Kbytes	64–512 Kbytes
	8 Kbytes	32 KB/64 KB	64 Kbytes	64–512 Kbytes
RAM	128 Bytes	4–16 Kbytes	8 Kbytes	32–128 Kbytes
	256 Bytes	0.5–2.1 Kbytes	16 K /16 Kbytes	16–96 Kbytes
	512 Bytes	1–2.5 Kbytes /4–8 Kbytes	4–8 Kbytes	20–68 Kbytes
EEPROM	128 Bytes	4096 Bytes	2048 Bytes	0 Bytes
	0–256 Bytes	512 Bytes	2048 Bytes/4096 Bytes	0 Bytes
	64–512Bytes	1024 Bytes/2048 Bytes	2048 Bytes	0 Bytes
Timers	2	3–6	02–08	6–10
	2	2–4	5–7/5–7	10
	2	2–4/2–6	2–8	3–6
I/O Pins	6–28	44–100	44–100	100–144
	6–32	32–64	64/64	48–64
	8–32	32–100/32–100	64–100	64–144
CPU Speed (MIPS)	4–20 MHz	16–20 MHz	32 MHz	66 MHz
	8–20 MHz	16–20 MHz	32 MHz/32 MHz	60 MHz
	12–20 MHz	16–20 MHz/16–20 MHz	32 MHz	66 MHz
BOR,POR,WDT	WDT	WDT	WDT	WDT
	WDT	WDT	WDT	WDT
	WDT	WDT/WDT	WDT	WDT
Supply Voltage Range	1.8 V–5.5 V	1.8 V–5.5 V	1.6 to 3.6	3.0–3.6 or (1.65–1.95 + 3.0–3.6)
	1.8 V–5.5 V	1.8 V–5.5 V	1.6 to 3.6	3.0–3.6 or (1.65–1.95 + 3.0–3.6)
	1.8 V–5.5 V	1.8 V–5.5 V	1.6 to 3.6	3.0 to 3.6 or 4.5 to 5.5
A to D Convertor (channels)	0–8	8–16	8–16	8
	4–12	8–12	16/16	6–8
	4–28	8–12/8–16	8–16	11–16
Others	UART/SPI/TWI/RTC	UART/SPI/TWI/RTC	UART/SPI/TWI/RTC	UART/SPI/TWI/RTC
PWM	3–6	6–15	6–24	0–13
	2–6	4–10	18–22/18—22	0–13
	2–9	4–10/4–15	6–24	14–20

23.9.1 AVR ATmega Family

This section focuses on AVR ATmega microcontroller family. The brief discussion of the programming model, features and peripherals of ATmega16 family is presented, but it does not cover the specific details of the device. The purpose of this section is to introduce and make a reader familiar with AVR ATmega microcontrollers.

There are many members in the AVR ATmega family of microcontrollers; the basic difference between all of them is in the available on-chip memory, I/O port pins, peripherals, operating frequency and the total number of pins. For example, ATmega8/16/32 have 8K/16K/32K bytes of on-chip flash memory, 1K/1K/2K bytes of on-chip data RAM, 23/32/32 I/O pins and total 32/44/44(TQFP package) pins.

The features of ATmega16 microcontrollers are listed below:

- High performance, low-power 8-bit microcontroller
- 131 Instructions—single-cycle execution for most instructions
- 32 × 8 general-purpose registers
- Fully static operation
- Up to 16 MIPS throughput at 16 MHz
- 16 Kbytes of In-System Self-Programmable flash with true read-while-write operation
- 512 Bytes data EEPROM and 1KByte internal SRAM
- JTAG (IEEE std. 1149.1 Compliant) Interface
- Three timers/counters
- Four PWM channels
- 8-channel, 10-bit ADC
- Byte-oriented two-wire serial interface
- Programmable serial USART
- Master/Slave SPI serial interface
- Programmable watchdog timer with separate on-chip oscillator
- On-chip analog comparator
- Power-on reset and programmable brown-out detection
- Internal calibrated RC oscillator
- Six sleep modes: Idle, ADC noise reduction, power-save, power-down, standby and extended standby
- 32 programmable I/O lines
- Available in 40-pin PDIP, 44-lead TQFP, and 44-pad MLF

23.9.2 Programming Model of ATmega16 Family of Microcontrollers

The programming model of ATmega16 consists mainly of three parts: Registers, Data Memory and Program Memory. Registers contain 32 general-purpose registers named as R0 to R31 (also referred as fast access register file), status register, program counter and stack pointer. Data memory contains SFRs (I/O registers), general-purpose RAM and data EEPROM. The fast access register file is a part of data memory. In-System Programmable Flash is a program memory. The programming model of (oversimplified) ATmega16 family is shown in Figure 23.3. The description of each of these registers is given in the next section.

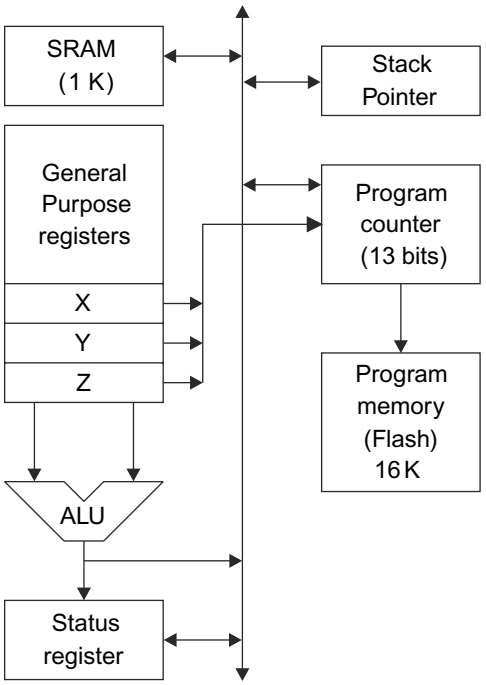


Fig. 23.3 *Simplified programming model of AVR (ATmega 16) microcontrollers*

General-Purpose Registers: R0–R31

R0–R31 are the general-purpose working registers used to store the data temporarily. These registers are similar to the Accumulator in the 8051. They are collectively referred as *register file*. In major ALU operations, the source operands are supplied from the register file and the result is stored back in them. The register file has a single-cycle access time; therefore majority of

the instructions involving ALU operations are executed in a single machine cycle. Because of this, the register file is also referred as *fast-access register file*.

These registers are part of the data memory and they are mapped at the first 32 locations of data memory. They are also assigned data memory addresses $00H to $1FH (R0 to R31) as shown in Figure 23.4.

The registers R26 to R31 can be also used as three 16-bit registers. These 16-bit registers are referred as the X, Y and Z registers. The mapping of these three registers on R26 to R31 is shown in Figure 23.5.

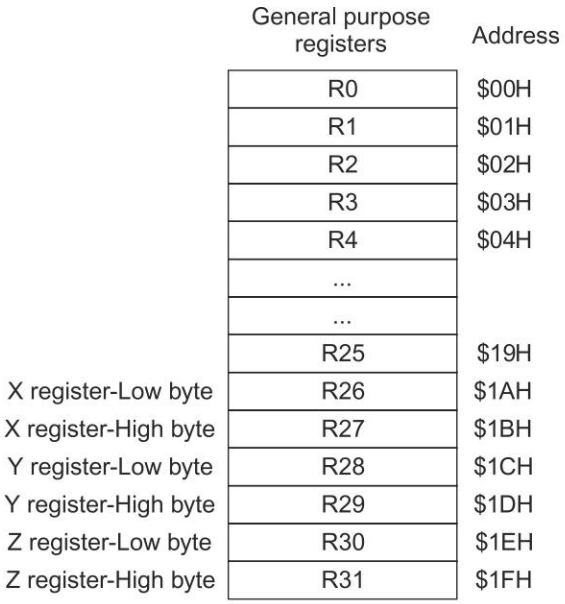


Fig. 23.4 *General-purpose working registers of AVR microcontrollers*

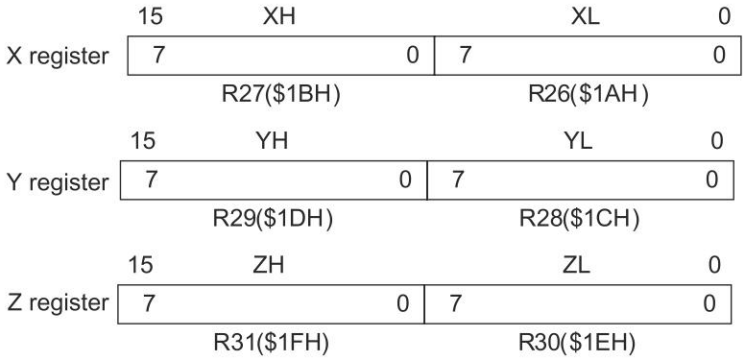


Fig. 23.5 *Mapping of X, Y and Z registers into the register file*

The X, Y and Z registers can be used as pointers for the code and data memory. They provide flexible and efficient ways of memory addressing.

STATUS Register

After an arithmetic or logical operation, the Status register is updated to reflect the nature of the result of the operation. Since it contains flags, it is also referred as Flag register. It is an 8-bit register. The structure of status register is shown and explained below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C

Bit 7 : I -Global Interrupt Enable Flag

Allow each interrupt to be individually enabled or disabled.

1 = Interrupts are enabled, along with this, individual interrupts enable bits must be set

0 = Disable all interrupts

Bit 6 : T- Bit Copy Storage Flag

The Bit load/store instructions use the T-bit as source or destination. A bit from a register (R0-R31) can be moved to T by the Bit Store (BST) instruction, or a bit in T can be moved to a Register by the Bit Load (BLD) instruction.

Bit 5 : H-Half Carry Flag (Equivalent to auxiliary carry in the 8051 and it is used in BCD operations)

1 = A carry out from the 4th low-order bit of the result

0 = No carry out from the 4th low-order bit of the result

Bit 4 : S-Sign Flag.

It is always EX-OR of N and V flags. It helps to know the sign of the real result and the result not corrupted by overflow.

Bit 3 : V-Overflow Flag (It is used for signed arithmetic)

1 = Overflow occurred in signed arithmetic operation

0 = No overflow occurred

Bit 2 : N-Negative Flag

It indicates the status of MSB of the result, i.e. whether the result is positive or negative and it is used in signed operations.

1 = Result is negative (MSB of the result)

0 = Result is positive (MSB of the result)

Bit 1 : Zero Flag

1 = The result of an arithmetic or logic operation is zero

0 = The result of an arithmetic or logic operation is not zero

Bit 0 : Carry or Borrow flag

1 = A carry out from the MSB of the result

0 = No carry out from the MSB of the result

Program Counter

The Program Counter (PC) always contains the address of the next instruction to be executed. Each code memory location in AVR microcontrollers is 2 bytes wide (16 bits). The 16 Kbytes code memory of ATmega16 is organized as 8 K × 2 bytes; therefore, its program counter is 13 bits wide (2^{13} = 8 K memory locations, each of two bytes). Based on the family member, PC in AVR microcontroller can be up to 22 bits.

Stack Pointer

The stack pointer is 16-bit register organized as two 8-bit registers: SPH (higher byte) and SPL (lower byte). After power-up, the SP is initialized by default with 0000, which is also address of the R0. Therefore, we should initialize SP with a proper value at the beginning of the program. The stack in the AVR microcontroller grows downwards; therefore, SP should be initialized with higher RAM address.

Data Memory

The AVRs have data RAM space of 64 Kbytes. The amount of on-chip RAM varies across different family members, for example, ATmega16 has 1 Kbyte on-chip SRAM. The data RAM space is made from three major components: general-purpose registers: R0-R31 (register file), SFRs (I/O memory) and general-purpose RAM. The organization of data memory of ATmega16 is shown in Figure 23.6.

The general-purpose registers (register file) is available in all the AVR family members; they have addresses $00H to $1FH (R0–R31). The SFRs and general-purpose RAM vary across different family members, for example, ATmega16 has 64 SFRs (I/O registers). They have data memory addresses $20H to $5FH (these locations can also be accessed with I/O addresses $00H to $3FH). It has 1K byte of general-purpose on-chip RAM, the address range is from $0060H to $045FH. The family members with more than 32 I/O pins also have an extended SFR memory.

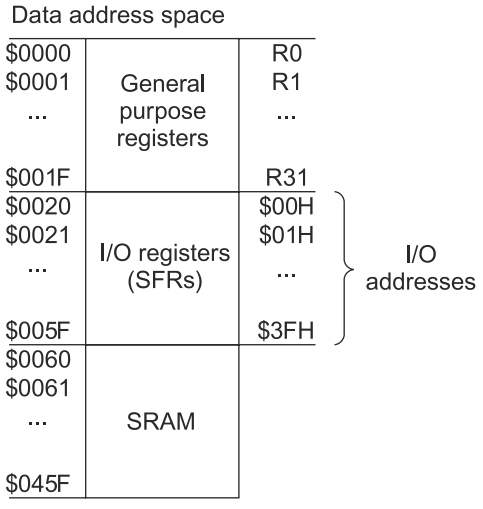


Fig. 23.6 *Data memory organization in ATmega16*

Data EEPROM

The ATmega16 equipped with 512 bytes of EEPROM. It can be used as data memory. This data EEPROM can be programmed during runtime using program instructions. It is used to store data permanently. The

EEPROM is organized as a separate data space and has addresses $000H to $1FFH. The EEPROM is accessed (read/write) through three SFRs: EEDR (EEPROM data register), EEARH-EEARL (EEPROM address registers high-low) and EECR (EEPROM control register).

Program Memory

The AVR has a program memory space of 8Mbytes (4M X 2Bytes) with address range of $000000H to $3FFFFFH. The amount of on-chip program memory varies across the different family members, for example, ATmega16 has 16 Kbytes of on-chip In-System Reprogrammable Flash memory as a program memory organized as 8K X 2 bytes with address range of $0000 to $1FFFH.

I/O Ports and Peripherals

The AVR microcontrollers have 1 to 11 eight-bit I/O ports (3 to 86 I/O pins) based on the family member. ATmega16 has 4 eight-bit I/O ports (Port A, B C and D, 32 I/O pins).The AVR microcontrollers have common peripherals like timers, ADC, UART, SPI, TWI, RTC, and PWM. The number (or number of channels) of peripherals varies across the family members. It also has features like Power-On Reset (POR), Internal RC oscillator, Brown-Out Reset (BOR), interrupts with two-level priority scheme, Watchdog Timer (WDT), sleep modes, code protection and In-System Self-Programmable flash with true read-while-write operation.

23.10 | PIC® MICROCONTROLLERS

Peripheral Interface Controller (PIC®) is a family of microcontrollers developed by Microchip Technology. The PIC® microcontrollers use the Harvard architecture and are designed with RISC philosophy. The PIC® microcontrollers are very much popular due to their low cost, wide and easy availability, large number of family members to meet the specific requirements of different applications, powerful documentation with a large number of application notes, availability of free or inexpensive software and hardware development tools.

The PIC® Architecture has the following advantages:

- Small Instruction Set (around 35 instructions for the low-end microcontrollers to about 80 instructions for the high-end microcontrollers)
- Based on the RISC Architecture
- On-chip Oscillator with programmable speeds
- Low cost
- Wide range of on-chip peripherals/interfaces, i.e. A/D, D/A, Timers, I^2C, SPI, USB, UART, PWM, LIN, CAN, PSP, WDT, POR, BOR, Comparators and Ethernet (not all the peripherals/interfaces in all devices)
- Fully static design, i.e. oscillator clock may be stopped at any time and may be restored back

PIC® microcontrollers are classified in two major categories: 8-bit and 16-bit microcontrollers where each category is further divided into product families as shown in Table 23.8.

Table 23.8 *PIC® microcontroller families*

8-bit MCU Product Family	16-bit MCU Product Family
PIC10	PIC24F
PIC12	PIC24H
PIC14	dsPIC30
PIC16	dsPIC33
PIC18	

Note: The microcontrollers from the PIC10 to PIC14 families are considered low-end microcontrollers. The microcontrollers in the PIC16 to PIC18 families are considered mid-level microcontrollers and PIC 24 to PIC33 families (16-bit PICs®) are considered high-end microcontrollers.

A brief comparison between some of the members from these families is given in the Table 23.9. Since there are hundreds of members in each family, the table gives a general comparison between three subfamilies for each category. The parameters are given in a range instead of specific value. The smaller value is for low-end PIC® and higher value is for high-end PIC® for that sub-family.

Table 23.9 *Comparison between PIC® microcontroller families*

Parameters	PIC16	PIC18	PIC24E	dsPIC30	dsPIC33
	PIC16F1xxx PIC16F6xx PIC16LF190x	PIC18F1xx0 PIC18F44xx PIC18F87xxx	PIC24EP32xx20x PIC24EP64xx20x PIC24EP512Gx8xx	dsPIC30F20xx dsPIC30Fxxxx dsPIC30F601XA	dsPIC33FJ06GSx0x dsPIC33FJxxxxx0x dsPIC33EP64MCx0x
ROM	3.5K–28K 1.7K–7K 3.5–14K	4K–8K 16K–24K 128K	32K 64K 512K	12K 24K–66K 132K–144K	6K 16K–32K 64K
RAM	128–2048Bytes 64–256Bytes 128–512Bytes	256 Bytes 768–2048 Bytes 3862–4096 Bytes	4K 8K 53K	512–1024Bytes 1024–2048Bytes 6144–8192Bytes	256–1024Bytes 1024–4096Bytes 8K
EEPROM	0–256Bytes 128–256 Byte 0	128–256 Bytes 0 –256 Bytes 0 –1024 Bytes	0 0 0	0–1024Bytes 1024 Bytes 2048–4096Bytes	0 0 0
Timers	2–6 (8–bit), 1– 3(16–bit) 1–2 (8–bit), 1(16–bit) 1 (8–bit), 1(16–bit)	1 (8–bit), 3 (16–bit) 1 (8–bit), 2– 3(16–bit) 0–3 (8–bit), 1–5 (16–bit)	5 (16–bit), 2(32–bit) 5 (16–bit), 2(32–bit) 5 (16–bit), 2(32–bit)	3 (16–bit), 1(32–bit) 3–5 (16–bit), 1–2(32–bit) 5 (16–bit), 2(32–bit)	2 (16–bit) 3 (16–bit), 1(32–bit) 5 (16–bit), 2(32–bit)
I/O Pins	12–53 12–18 25–36	16 34–36 51–69	21–35 21–53 21–122	12–35 12–68 52–68	18–21 15–35 21–53
CPU Speed in MIPS	5–12 5 5	10 10–16 10–16	70 70 70	30 30 30	40 16–70 30–40
BOR, POR, WDT	N, N, N N, N, N N, N, N	Y, N, Y Y, N, Y Y, N, Y	Y, Y, Y Y, Y, Y Y, Y, Y	Y, Y, Y Y, Y, Y Y, Y, Y	Y, Y, Y Y, Y, Y Y, Y, Y
Supply Voltage Range	1.8 V–5.5 V 2 V–5.5 V 1.8 V–3.6 V	2 V–5.5 V 2 V–5.5 V 2 V–3.6 V	3 V–3.6 V 3 V–3.6 V 3 V–3.6 V	2.5 V–5.5 V 2.5 V–5.5 V 2.5 V–5.5 V	3 V–3.6 V 3 V–3.6 V 3 V–3.6 V
A to D Convertor	1 1 1	1 (4–7 Ch) 1 (9–11 Ch) 1(2–24 Ch)	1 (6–9 Ch) 1 (6–16 Ch) 2 (32 Ch)	1 (6–10 Ch) 1 (6–10 Ch) 1 (16 Ch)	1 (6–10 Ch) 1 (6–16Ch) 1 (6–24Ch)
DMA Channel	0 0 0	0 0 0	4 4 15	0 0 0	0 0–4 4
Others	UASRT, SPI, I2C,	ICSP, ICD–DEBUG, PWM, Comparators, LIN	ICSP, ICD–DEBUG, PWM, Comparators, UART, SPI, I2C, LIN, IrDA	ICSP, ICD–DEBUG, PWM, Comparators, UART, SPI, I2C, CAN, LIN, IrDA	ICSP, ICD–DEBUG, PWM, Comparators, UART, SPI, I2C, CAN, JTAG Interface, LIN, QEI, IrDA

23.10.1 PIC18 Family

This section focuses on the PIC18 microcontroller family. A brief discussion of the programming model, architecture, peripherals and features of PIC18FXX2 family is presented, but it does not cover the specific details of each device in the family. The purpose of this section is to introduce and make a reader familiar with the mid-level PIC microcontrollers.

There are many members in the PIC18F family of microcontrollers; the basic difference between all of them is in the available on-chip memory, I/O port pins, peripherals, operating frequency and total number of pins. For example PIC18F242/252/442 have 16K/32K/16K bytes of on-chip flash memory, 768/1536/768 bytes of on-chip RAM, 3/3/5 I/O ports and total 28/28/40(or 44) pins.

The features of PIC18FXX2 family are listed below:

- Total 77 instructions and C compiler optimized RISC architecture/instruction set
- Up to 10 MIPs operation
- 16-bit wide instructions, 8-bit wide data path
- 8 × 8 single-cycle hardware multiplier
- High current sink/source capacity (25 mA/25 mA)
- Three external interrupt pins
- Four timer modules
- Two Capture/Compare/PWM (CCP) modules
- Master Synchronous Serial Port (MSSP) module (3-wire SP I and I^2C Master/slave mode)
- USART and PSP modules
- 10-bit analog-to-digital converter module
- Programmable low-voltage detect and brown-out reset
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Programmable code protection and power-saving SLEEP mode
- Selectable oscillator options
- In-Circuit Serial Programming and ICD via two pins
- Fully static design
- Wide operating voltage range (2.0 V to 5.5 V)
- Low power consumption

23.10.2 Programming Model of PIC18 family of Microcontrollers

As discussed in Chapter 2, the programming model is the user's view of a microcontroller, and provides a representation of the internal architecture. It is a collection of internal registers (and memory locations) that can be used by a programmer to develop any software (i.e. control and application programs) and to use several features of a particular microcontroller. The programming model of PIC18F is divided broadly into three parts: Registers, Data Memory and Program Memory. The registers include Working Register (WREG—equivalent to Accumulator in the 8051), Status Register (STATUS), Bank Select Register (BSR), File Select Registers (FSR), PC, Table pointer, Stack pointer, multiplication registers (PRODH and PRODL). The data memory includes SFRs for peripherals and general-purpose registers and program memory to store the program instructions. The programming model of PIC18F family is shown in Figure 23.7. The description of each of these registers is given in the next section.

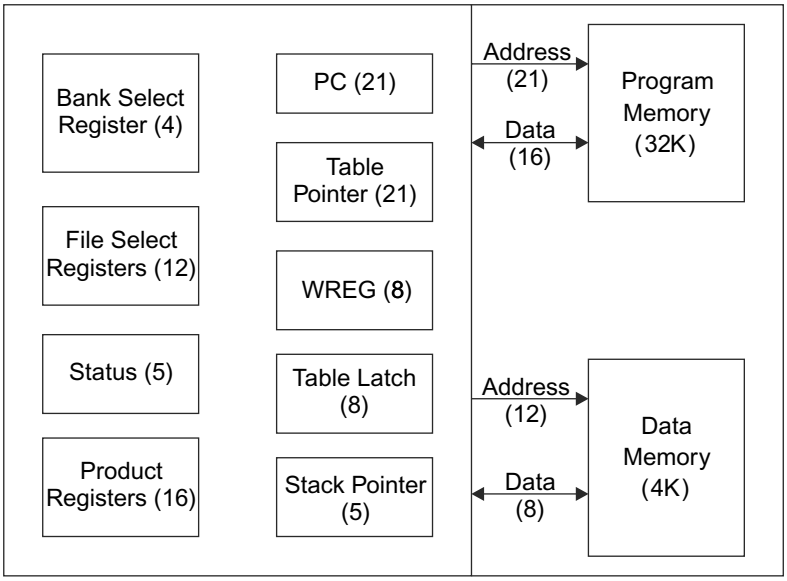


Fig. 23.7 *Simplified programming model of PIC18 family microcontrollers*

Working Register (WREG)

The working register is an 8-bit register similar to the Accumulator in the other microprocessors/ controllers. It is used to hold one of the source operand for arithmetic and logical instructions. Based on the instruction, it may also store the result of the operation. One of the important features of this microcontroller is that the result of arithmetic/logical operation can be saved in either WREG or data memory registers (File

register) based on the value of destination bit in an instruction. (Refer PIC18 MCU family reference manual for details of instruction set).

Bank Select Register (BSR)

Since PIC18 devices have 12-bit address bus for data memory, it can have a maximum of 4 Kbytes (4096 bytes) of data memory. The entire data memory is divided into 16 banks (Bank 0 to Bank F), each containing 256 bytes. The lower 4 bits of the Bank Select Register (BSR < 3:0 >) are used to select which bank will be accessed by an instruction. The upper 4 bits for the BSR are reserved (always zero).

File Select Registers (FSR)

File Select Registers are used in indirect addressing mode. An FSR register is used as a pointer to the data memory location that is to be read or written, i.e. it holds the address of the data memory. There are three file select registers, FSR0, FSR1 and FSR2. To address the entire data memory space of 4096 bytes, these registers are 12-bit wide. The 12 bits of the addresses are represented by a combination of two 8-bit registers. These registers are represented as FSRnH and FSRnL, i.e. FSRn is composed of registers FSRnH : FSRnL as shown below.

FSR0 = FSR0H : FSR0L (FSR0H < 3:0 > : FSRL0 < 7:0 >)
FSR1 = FSR1H : FSR1L (FSR1H < 3:0 > : FSRL1 < 7:0 >)
FSR2 = FSR2H : FSR2L (FSR2H < 3:0 > : FSRL2 < 7:0 >)

Each FSR register has an INDF register and four addresses associated with it. Based on INDFn address selected in an instruction, the FSRnH:FSRnL registers are modified in a different manner, for example the address in the FSRn may be auto incremented before or after the data transfer, or auto decremented before or after the data transfer, or the value in the WREG register is added as an offset to FSRn.

STATUS Register

The STATUS register contains five arithmetic flags, namely Negative, Overflow, Zero, Digital Carry and Carry. These flags contain the arithmetic status of the ALU, i.e. the nature of the result produced by execution of an instruction. Other instructions can test these flags and make decisions based on the flag states. The structure of status register is shown and explained below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	–	N	OV	Z	DC	C

Bit 7 : Unimplemented

Bit 6 : Unimplemented

Bit 5 : Unimplemented

Bit 4 : Negative Bit

It indicates the status of MSB of the result, i.e. whether the result is positive or negative and it is used in signed operations.
1 = Result is negative (MSB of the result)
0 = Result is positive (MSB of the result)

Bit 3 : Overflow Bit

It is used for signed arithmetic.
1 = Overflow occurred in signed arithmetic operation.
0 = No overflow occurred.

Bit 2 : Zero Bit

1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

Bit 1 : Digit Carry or Borrow Bit

Equivalent to auxiliary carry in the 8051 and it is used in BCD operations.

1 = A carry out from the 4th low order bit of the result
0 = No carry out from the 4th low order bit of the result

Bit 0 : Carry or Borrow Bit

1 = A carry out from the MSB of the result
0 = No carry out from the MSB of the result

Table Pointer

The table pointer register is used to move the data bytes between the program memory and the data memory. These operations are referred as table-read and table-write. The Table-read operation reads the data from program memory and copies it into the data memory and Table-write operation writes the data to program memory from data memory.

The Table Pointer (TBLPTR) can address any byte within the program memory. The TBLPTR is made from three registers: Table Pointer Upper Byte, Table Pointer High Byte and Table Pointer Low Byte (TBLPTRU: TBLPTRH: TBLPTRL).

Program Counter

The Program Counter (PC) always contains the address of the next instruction to be executed, i.e. it points to the instruction that is to be executed next or it points to the instruction that is currently being fetched. The program counter is a 21-bit wide register; therefore, the addressing capacity of PIC18 family is 2^{21} bytes, i.e. 2 MB. As the CPU fetches the program instructions from the program memory, the PC is incremented automatically to point to the next instruction.

The PC is comprised of three 8-bit registers—PCL, PCH and PCU. The PCL is the lower byte of a 21-bit address, i.e. it contains the PC < 7:0 > bits, PCH is higher (middle) byte of the PC and contains the PC < 15:8 > bits and PCU is the upper byte of PC and contains the PC < 20:16 > bits. PCH and PCU are not directly readable or writable. The PCH and PCU registers are updated through the PCLATH and PCLATU registers respectively.

Stack and Stack Pointer

The stack in PIC18 microcontrollers contains 31 registers, where each register is 21-bit wide. These registers are used for temporarily holding the program memory addresses during the program execution. These 31 registers are separate from the program and data memory and are referred as hardware stack. To address these 31 registers, the PIC18 microcontrollers use 5 bits of stack pointer registers (STKPTR < 4:0 >).

The PC is pushed on the stack when a CALL (all types) instruction is executed. The PC value is retrieved from the stack by RETURN (all types) instruction. The CALL instruction will first increment the stack pointer and the contents of the PC are written to the stack register pointed by the stack pointer. The RETURN instruction will retrieve the contents from the stack register pointed by stack pointer into PC and then the stack pointer is decremented.

Special Function Registers (SFRs)

The Special Function Registers (SFRs) are used to control the operations of the CPU and peripheral devices. They are used to configure or initialize the peripheral modules for the desired operation. SFRs are part of the data memory and are implemented as static RAM. There are two categories of SFRs, those related with CPU operations and those related with the peripheral devices. For example, stack pointer and program counter are SFRs related with CPU operations and Timer and ADC control registers are SFRs related with peripherals. The SFRs are assigned the addresses between F80H to FFFH in a data memory. A list of these SFRs along with their addresses can be found from the datasheet of the device.

Data Memory

Since PIC18 devices have 12-bit address bus for data memory, it can have a maximum 4Kbytes (4096 bytes) of data memory; thus, the address space of the data memory ranges from the address 000H to FFFH. It has 8-bit data bus and is implemented as static RAM. The data memory includes SFRs for peripherals and general-purpose registers (GPRs). The SFRs are used to control the operations of CPU and peripheral devices, while GPRs are used for temporary data storage during user's program execution.

Figure 23.8 shows the data memory organization for the PIC18FX42 devices.

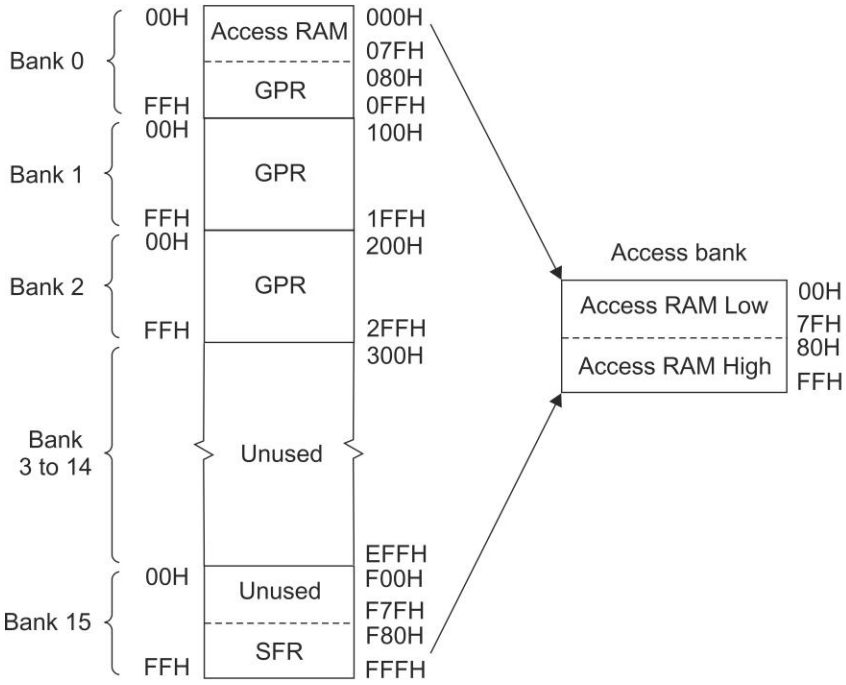


Fig. 23.8 *Data memory organization for the PIC18FX42 devices*

The entire data memory is divided into 16 banks (Bank 0 to Bank F), each containing 256 bytes. The lower 4 bits of the Bank Select Register (BSR < 3:0 >) are used to select which bank will be accessed by an instruction. The data memory can be accessed either directly using BSR or indirectly using FSR or through Access Bank.

As shown in Figure 23.8, the Access bank consists of data memory locations 00H to 7FH (GPRs) of bank 0 and 80H to FFH (SFRs) of bank FH. The Access Bank allows the access to a commonly used data memory location irrespective of the BSR value, thus, providing faster access (in single cycle) to these data memory locations.

Program Memory

Program memory is used to store the program instructions. The PIC18 family devices have 21-bit wide program counter, therefore, the addressing capacity of this family is 2^{21} bytes, i.e. 2 MB. The program memory ranges from address 000000H to 1FFFFFH. In the PIC18C family, program memory is implemented using EPROM, while in PIC18F; it is implemented using flash memory. The PIC18F252/452 has 32 Kbytes of flash memory, while the PIC18F242/442 has 16 Kbytes of flash. Figure 23.9 shows the program memory organization for the PIC18FX42 devices.

Since majority of the instructions (73 out of 77) are 16-bit wide (word), the program memory has 16-bit data bus. This allows fetching of a 16-bit instruction in a single cycle. Reading a location which is not physically implemented in the program memory map will give all '0's (a NOP instruction). The addresses 0000H, 0008H and 0018H are RESET vector, the higher priority interrupt vector and lower priority interrupt vector respectively.

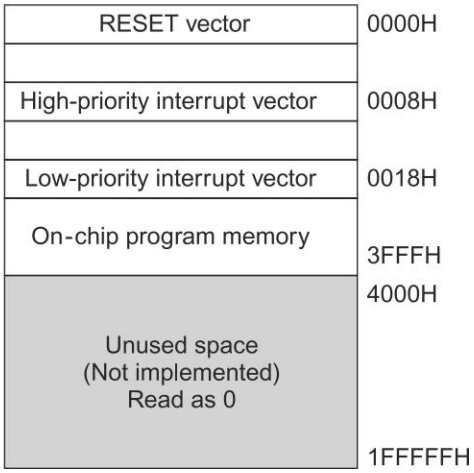


Fig. 23.9 *Program memory organization for the PIC18FX42 devices*

Data EEPROM Memory

This data EEPROM can be programmed during runtime using program instructions, i.e. microcontroller instructions can program this EEPROM over the entire V_{DD} range. This feature allows storing (or updating) some important data in real-time without affecting other bytes, moreover the data will remain stored in memory (because EEPROMs is nonvolatile memory)

even if the power is removed. The data EEPROM is not a part of the data memory and it is indirectly accessed through the SFRs. The PIC18F family has 256 bytes to 1024 bytes of data EEPROM.

I/O Ports

There are either three or five I/O ports available based on a device. For example, PIC18F242/252 has three ports namely A, B and C, and PIC18F442/452 has five ports: A, B, C, D and E. Most pins of the I/O ports are multiplexed with an alternate function for the peripheral devices. Each port has three SFRs associated with it for its operation. They are TRIS, PORT and LAT registers. TRIS is used to configure the direction of the port, i.e. input or output, PORT is used to read the port and LAT is used when the port is to be used as a bidirectional port. Each SFR for all the ports are mapped in the data memory.

Peripheral Devices

The PIC18Fxxx family has 4 timers, 2 CCP (Capture/Compare/Pulse width modulation), serial communication modules (Master Synchronous Serial Port and addressable USART), 5/8 channel 10-bit A/D converters, parallel slave port (only for PIC18F442/452).

It also has features like Power-On Reset (POR), Power-Up Timer, Oscillator Start Up Timer, Brown-Out Reset (BOR), interrupts with two-level priority scheme, Watchdog Timer (WDT), sleep mode, programmable low voltage detect, code protection and In-circuit Serial Programming (ICSP) and oscillator selection (8 different modes).

POINTS TO REMEMBER

- The 8052 has additional 128 bytes of on-chip RAM from address 80H to FFH and can only be accessed through indirect addressing.
- Timer 2 allows the transmitter and receiver to operate at different frequencies.
- Timer 2 can also be programmed in 'capture' mode where the current value of timer registers is stored in its capture registers when there is a high-to-low transition at external input pin T2EX (P1.1 pin).
- The minimum clock speed of 0 Hz indicates the static operation, i.e. microcontroller has static RAM.
- I^2C is a synchronous bus that uses two wires SDA (Serial Data) and SCL (Serial Clock) for short distance communication between microcontroller(s) and peripherals.
- A Watchdog Timer (WDT) is a peripheral used to monitor the execution of the system software.
- Watchdog timers are useful when the system is operated in a noisy environment, operated remotely, and when the system is operating in hazardous environments.
- Controller Area Network (CAN) is an asynchronous serial bus used mainly in real time and in distributed embedded applications like automobiles and industrial automation.
- SPI is full-duplex bus, i.e. the data is simultaneously transmitted and received.

OBJECTIVE QUESTIONS

1. What is the function of a watchdog timer?
   (a) The watchdog timer is an external timer that resets the system if the software fails to operate properly.
   (b) The watchdog timer is an internal timer that sets the system if the software fails to operate properly.
   (c) The watchdog timer is an internal timer that resets the system if the software fails to operate properly.
   (d) None of the above.
2. DS89c420 is really an 8052 chip.
   (a) True (b) False
3. DS89C420 has _____bytes of on-chip ROM and _____bytes of RAM.
   (a) 16 K and 256 (b) 0 K and 256 (c) 8 K and 128 (d) 64 K and 256
4. In DS89C4x0, the number of clock pulses per machine cycle are,
   (a) 1 (b) 12 (c) 6 (d) 4
5. 8052 has ___ 16-bit timers.
   (a) 1 (b) 2 (c) 3 (d) 4

6. SPI is _____ wire and CAN is _____ wire communication system.
   (a) 4, 2 (b) 2, 2 (c) 2, 4 (d) 4, 4
7. Which of the following is ROM-less microcontroller chip?
   (a) 8051 (b) 8751 (c) 89C51 (d) 8031
8. Which of the following peripherals are using asynchronous data-transfer scheme?
   (a) CAN (b) SPI (c) I^2C (d) UART
9. DS89C420/30/40/50 require __clocks/machine cycle.
   (a) 1 (b) 2 (c) 6 (d) 12
10. The AT89S8253 has __ bytes of on-chip code memory.
    (a) 4 K (b) 8 K (c) 12 K (d) 16 K

Answers to Objective Questions

1. (c) 2. (a) 3. (a) 4. (a) 5. (c)
6. (a) 7. (d) 8. (a), (d) 9. (a) 10. (c)

REVIEW QUESTIONS WITH ANSWERS

1. How does the 8052/8032 differ from the 8051?

A. The 8052/8032 has additional 128 bytes of internal RAM, additional timer (Timer2) and 8Kbytes of on-chip program memory (ROM).

2. How are the additional 128 bytes of internal RAM in the 8052 accessed?

A. Additional internal RAM is accessed using only indirect addressing.

3. How does the 8052 differentiate between SFRs and additional RAM?

A. SFRs are accessed using direct addressing and additional RAM is accessed using indirect addressing.

4. I^2C is a synchronous bus. True/False.

A. True.

5. What are the key features of I^2C bus?

A. Simplicity. It requires only two wires for communication. Any device can be added or removed to the bus easily by simply connecting them to these two wires.

6. What is the use of watch dog timer?

A. Watch dog timer resets the microcontroller when the program execution is erroneous, i.e. the microcontroller is no longer executing correct and expected sequence of programmed instructions.

7. Which SFR is used to program the data EEPROM in the AT89S8253?

A. EECON.

8. RTC provides the information of date and time. True/False.

A. True.

9. What is meant by the 8051 variant?

A. It is a chip which has the same instruction set as that of 8051, but may have some extra on-chip memory, peripherals and features.

EXERCISE

1. Discuss the features of Timer 2.
2. List the hardware resources of 8052, 8032, 8751 and 8752.
3. List the common 8051 variants made by Atmel along with their hardware resources.
4. List the common peripherals available in the different 8051 variants.
5. What is the bit size of commonly available on-chip ADCs?
6. Discuss the terms ISP and IAP.
7. In what type of environments are thewatch dog timers useful?
8. Explain the operation of watchdog timer.
9. List the features of CAN bus.
10. What is PWM? Where is it useful?
11. How is the MCS 151 family of microcontrollers superior to the 8051?
12. Discuss the features of ATmega16 family of microcontrollers.
13. Discuss the features of PIC18 family of microcontrollers.

Appendix A

The 8051 Instruction Set Summary

ACALL *ADDR11* (ACALL *LABEL*)

Function Absolute call (anywhere within page of 2 K)

It calls the subroutine located at the specified address within 2 Kbyte page with respect to the PC. The instruction automatically saves the address of the next instruction (address of the instruction itself + 2) onto the stack (low byte first) and increments the stack pointer by two. The new value of PC (destination address) is obtained by combining the five bits of the PC (PC_{15} through PC_{11}), op-code bits 7 through 5, and the second byte of the instruction. The program execution is then transferred to the new value of PC.

Since three bits of op-code are decided by A_{10} toA_8 of the destination address, this instruction has 8 op-codes. Refer topic 7.3 for more details.

Flags Affected None

ACALL addr 11 or Label **// SP = SP + 1;(SP) = PCL; SP = SP + 1; (SP) = PCH; $PC_{10\text{-}0}$ = addr 11**

Bytes: 2 **Cycles:** 2

Example Assume that SP is initialized with the value 50H. The instruction ACALL DELAY is written at the program memory address 0100H, label DELAY is at location 0250H as shown below. Consider the following instruction,

```
00FF            …
0100            ACALL DELAY
0102            …
                …
                …
0250  DELAY:    …
```

After execution of instruction ACALL DELAY, the contents of registers/memory locations will be changed as follows,

```
PC = 0250H      // program execution will start at the destination address
(51H) = 02H     // return address, i.e. address of the next instruction is saved
(52H) = 10H     // on the stack, lower byte first
SP = 52H        // SP is incremented by 2 because 2 bytes are saved on stack
```

ADD A, *SOURCE OPERAND*

Function Add source operand byte with Accumulator.

It adds the specified source operand byte with the Accumulator, and stores the result in the Accumulator. The carry is set if there is a carry out from Bit 7, otherwise it is cleared. The auxiliary carry flag is set if there is a carry out from Bit 3, otherwise it is cleared. When adding unsigned numbers, the carry flag indicates an overflow occurred, i.e. the result is greater than $FF_{H.}$

The OV is set if there is a carry out from Bit 7 but not from Bit 6, or, a carry out from Bit 6 but not from Bit 7 otherwise, it is cleared. When adding signed numbers, OV indicates an overflow occurred, i.e. the result is outside the signed number range -128 to + 127 and the result is wrong.

Flags Affected C, OV, AC

The source operand can be specified by any of the four addressing modes: Immediate, Register, Direct or Register Indirect.

ADD A, #data	**// A = A + data**	**Bytes: 2**	**Cycles: 1**
ADD A, Rn	**// A = A + Rn**	**Bytes: 1**	**Cycles: 1**
ADD A, direct	**// A = A + direct**	**Bytes: 2**	**Cycles: 1**
ADD A, @Ri	**// A = A + (Ri)**	**Bytes: 1**	**Cycles: 1**

Example

Unsigned Addition Assuming that all the numbers in the following instructions are unsigned numbers.

```
MOV A, #10H          // A = 10H
ADD A, #20H          // A = 30H, C = 0, AC = 0

MOV R3, #80H         // R3 = 80H
MOV A, #97H          // A = 97H
ADD A, R3            // A = 17H, C = 1, AC = 0

MOV 40H, #88H        // (40H) = 88H
MOV A, #0A9H         // A = A9H
ADD A, 40H           // A = 31H, C = 1, AC = 1

MOV R0, #10H         // R0 = 10H
MOV 10H, #20H        // (10H) = 20H
MOV A, #30H          // A = 30H
ADD A, @R0           // A = 50H, C = 0, AC = 0

ADD R2, A            // invalid, A must be the destination for all ADD instructions
ADD A,@R3            // invalid, only R0 and R1 can be used for indirect addressing
```

Signed Addition Assuming all the numbers in the following instructions are signed numbers.

```
MOV A, #0AH  or MOV A,#10 or MOV A,#+10  // A = 0AH
ADD A, # + 5                  // A = 0FH, C = 0, OV = 0

MOV A, #100  or MOV A, #64H   // A = 64H
MOV R0, #50  or MOV R0,#32H   // R0 = 32H
ADD A, R0                     // A = 96H, C = 0, OV = 1(result is
                              // outside the range of 8-bit signed
                              // numbers

MOV A, #-10  or MOV A, #0F6H  // A = F6H (2's complement of 10)
ADD A, #-5   or ADD A, #0FBH  // A = F1H = -15 C = 1, OV = 0
```

ADDC A, *SOURCE OPERAND*

Function Add the source operand byte and Carry flag with the Accumulator. (Full addition)

It adds the specified source operand byte as well as Carry flag with the Accumulator, and stores the result in the Accumulator. The C, AC and OV flags are affected exactly similar to the ADD instruction discussed above.

Flags Affected C, OV, AC

The source operand can be specified by any of the four addressing modes: Immediate, Register, Direct or Register Indirect. While dealing with the multi-byte addition, we need to consider the propagation of carry from lower bytes to the higher byte, i.e. we need to perform a full addition. The instruction ADDC is used for such operations. The suffix C after ADD indicates carry flag is included in the addition.

ADDC A, #data	**// A = A + data + C**	**Bytes: 2**	**Cycles: 1**
ADDC A, *Rn*	**// A = A + Rn + C**	**Bytes: 1**	**Cycles: 1**
ADDC A, *direct*	**// A = A + (direct) + C**	**Bytes: 2**	**Cycles: 1**
ADDC A, @Ri	**// A = A + (Ri) + C**	**Bytes: 1**	**Cycles: 1**

Example Consider two 16-bit numbers 5292H and 1570H.

The addition of these two 16-bit numbers will be performed in the following manner:

```
    1
  52 92 H
+ 15 70 H
  68 02 H
```

The program to perform the above addition using ADDC instruction is given below:

```
CLR C                 // C = 0; initially carry is cleared
MOV A, #092H          // A = 92H
ADDC A, #70H          // A = 92H + 79H + 0 = 02H, C = 1
MOV R2, A             // R2 = 02H, save lower byte result into R2
MOV A, #52H           // A = 52H,
ADDC A, #15H          // A = 52H + 15H + 1 = 68H, C = 0
MOV R3, A             // R3 = 68, save higher byte of result into R3
```

AJMP *ADDR11* (AJMP *LABEL*)

Function Absolute jump (anywhere within a page of 2 K)

It transfers the program execution to the specified address within 2 Kbyte page with respect to the PC. The new value of PC (destination address) is obtained by combining the five bits of the PC (PC_{15} through PC_{11}), op-code bits 7 through 5, and the second byte of the instruction. Since three bits of op-code are decided by A_{10} toA_8 of the destination address, this instruction has 8 op-codes. Refer topic 7.1.1 for more details.

Flags Affected None

AJMP addr11	**/ $PC_{10\text{-}0}$ = addr11**	**Bytes: 2**	**Cycles: 2**

Example The instruction AJMP SKIP is written at the program memory address 0100H, label SKIP is at the location 0250H as shown below. Consider the following instruction:

```
00FF                  …
0100                  AJMP SKIP
0102                  …
                      …
                      …
0250    SKIP:         …
                      …
```

After execution of instruction AJMP SKIP, the contents of the PC will be changed as follows:

```
PC = 0250H            // program execution will start at the destination address
```

ANL *DESTINATION BYTE, SOURCE BYTE*

Function Bitwise AND operation between the source and destination byte.

It performs the bitwise logical AND operation between the source and destination operands and stores the result in the destination operand. The Accumulator or a direct address in the internal RAM can be the destination. When the destination is the Accumulator, the source can use immediate register, direct or register indirect addressing, and when the destination is a direct address, the source can be the immediate data or Accumulator.

Flags Affected None (Flags may be affected when the destination operand is PSW(direct address))

The format of the instruction for each addressing mode is given as follows,

ANL A, *#data*	**// A = A AND *data***	**Bytes: 2**	**Cycles: 1**
ANL A, *Rn*	**// A = A AND *Rn***	**Bytes: 1**	**Cycles: 1**
ANL A, *@Ri*	**// A = A AND *(Ri)***	**Bytes: 1**	**Cycles: 1**
ANL A, *direct*	**// A = A AND (*direct*)**	**Bytes: 2**	**Cycles: 1**
ANL *direct, #data*	**// *(direct)* = *(direct)* AND *data***	**Bytes: 3**	**Cycles: 2**
ANL *direct*, A	**// *(direct)* = *(direct)* AND A**	**Bytes: 2**	**Cycles: 1**

Example

```
MOV A, #0FFH          // A = FFH
ANL A, #10H           // A = FFH AND 10H = 10H

MOV R0, #4AH          // R0 = 4AH
MOV A, #55H           // A = 55H
ANL A, R0             // A = 55H AND 4AH = 40H

MOV 10H, #72H         // (10H) = 72H
MOV A, #0FH           // A = 0FH
ANL 10H, A            // (10H) = 72H AND 0FH = 02H
```

The AND operation (ANL instruction) is often used to mask (set to 0) certain bits data byte as shown below:

```
      55 H   0101 0101
AND   0EH    0000 1110
      04H    0000 0100
```

Note: When the destination operand is a port address, the original port data will be read from the port latch, not the port pins.

ANL C, *SOURCE OPERAND BIT*

Function Logical AND operation between Carry and source operand bit

It performs the logical AND operation between the Carry flag and source operand bit (Boolean), stores the result in Carry flag. If the source bit is a 0 then the carry flag is cleared, otherwise, carry flag remains unchanged.

A slash (/) preceding the source operand bit indicates that the complement of the bit is used as the source value, but the source bit is not affected. No other flags are affected. Only direct addressing is allowed for the source operand.

Flags Affected C

The format of the instruction for each addressing mode are given below:

ANL C, bit	**// C = C AND (bit)**	**Bytes: 2**	**Cycles: 2**
ANL C, /bit	**// C = C AND ($\overline{bit}$)**	**Bytes: 2**	**Cycles: 2**

Example

```
MOV A, #0FFH      // A = FFH
CLR C             // C = 0
ANL C, ACC.0      // C = C AND ACC.0 = 0 AND 1 = 0

MOV A, #00H       // A = 00H
SETB C            // C = 1
ANL C, /ACC.0     // C = C AND ACC.0 = 1 AND 1 = 1
```

CJNE *DESTINATION BYTE, SOURCE BYTE, REL ADDRESS*

Function Compare and jump if not equal.

It compares the magnitudes of the destination byte and source byte (first two operands) and jumps to the specified relative address if their values are not equal; otherwise the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. The carry flag is set if the destination byte (unsigned) is less than the source byte, else the carry flag is cleared. Neither operand is affected.

Flags Affected *C*

Four addressing mode combinations are possible, the destination byte may be Accumulator and it can be compared with the direct addressed byte or the immediate data byte. In the other format, destination byte may be indirect RAM location or register, and they can be compared with an immediate data byte. The format of the instruction for each addressing mode is given below,

CJNE A, *direct*, *rel*	**// If A≠(direct), PC = PC + rel**	**Bytes: 3**	**Cycles: 2**
CJNE A, *#data*, *rel*	**// If A≠data, PC = PC + rel**	**Bytes: 3**	**Cycles: 2**
CJNE *Rn*, *#data*, *rel*	**// If Rn≠data, PC = PC + rel**	**Bytes: 3**	**Cycles: 2**
CJNE @*Ri*,*#data*, *rel*	**// If (Ri)≠data, PC = PC + rel**	**Bytes: 3**	**Cycles: 2**

Example The following instructions monitors the status of Port 2 continuously until it is 50H:

```
          MOV P2, #0FFH           // configure P2 as input
REPEAT:   MOV A, P2               // read status of P2 into A
          CJNE A, #50H, REPEAT    // repeat until P2 status is 50H
```

The following instructions increment R5 until it is equal to the contents of the address 30H:

```
REPEAT:   INC R5                  // increment R5 until its value is equal to content
                                  // of address 30H
          MOV A, R5
          CJNE A, 30H, REPEAT
```

The following instructions will set all pins of Port 1 if contents of A are less than 75H, set all pins of P2 if A = 75, otherwise it will clear all pins of Port 1.

```
          CJNE A, #75H, AHEAD
          MOV P2, #0FFH
          SJMP EXIT
AHEAD:    JC SETP1
          MOV P1, #00H
          SJMP EXIT
SETP1:    MOV P1, #0FFH
EXIT:     …
```

CLR A

Function Clear Accumulator

It clears the Accumulator. All bits are cleared to 0, i.e. A = 00.

Flags Affected None

CLR A	**// A = 0**	**Bytes: 1**	**Cycles: 1**

Example

```
CLR A                 // A = 0
MOV R3, A             // R3 = 0
MOV 10H, A            // (10H) = 0
MOV P2, A             // P2 = 0
```

CLR *BIT*

Function Clear bit

It clears the specified bit (bit = 0). This instruction can operate on carry flag or any bit-addressable location.

Flags Affected None (Flags are affected directly if the bit address of the flag is specified)

CLR C	**// C = 0**	**Bytes: 1**	**Cycles: 1**
CLR bit	**// bit = 0**	**Bytes: 2**	**Cycles: 1**

Example

```
CLR P1.0              // P1.0 = 0
CLR C                 // C = 0
CLR 00H               // bit address location 00H = 0, i.e. LSB of byte 20H is cleared
```

CPL A

Function Complement Accumulator

It logically complements each bit of the Accumulator (one's complement), i.e. 0s are changed to 1s and 1s are changed to 0s.

Flags Affected None

CPL A	**// A = $\overline{A}$**	**Bytes: 1**	**Cycles: 1**

Example Assume A = 55H before execution of the following instruction:

```
CPL A                   // A = AAH
```

CPL *BIT*

Function Complement bit

It complements the specified bit (bit = $\overline{bit}$). This instruction can operate on the carry flag or any bit-addressable location.

Flags Affected None (Flags are affected directly if the bit address of the flag is specified)

CPL C	**// C = $\overline{C}$**	**Bytes: 1**	**Cycles: 1**
CPL bit	**// bit = $\overline{bit}$**	**Bytes: 2**	**Cycles: 1**

Example

If C = 0 before execution of the following instruction,

```
CPL C                   // C = 1
```

If P1.1 = 0 before execution of the following instruction,

```
CPL P1.1                // P1.1 = 1
```

Note: When bit address of the port pin is specified, the original port pin status will be read from the port latch, not the port pins.

DA A

Function Decimal-adjust Accumulator after addition

It is used after addition of BCD numbers (packed BCD) to convert (adjust) the result into BCD form.

ADD or ADDC instruction might have been used to perform the addition. The result is adjusted in the following conditions.

(i) Lower nibble of A is greater than 9 after addition
(ii) AC = 1 after addition
(iii) Upper nibble of A is greater than 9 after addition
(iv) C = 1 after addition

Flags Affected C

DA A	**// adjust the result to BCD**	**Bytes: 1**	**Cycles: 1**

Example

(i) When lower nibble is greater than 9 after addition:

```
    28 BCD    0010 1000
+   12 BCD    0001 0010
    40 BCD    0011 1010     3A; lower nibble is greater than 9(Invalid BCD)
            + 0000 0110     add 6 to the lower nibble (DA A will do it)
              0100 0000     40 BCD, desired result
MOV A, #28H             // A = 28H
ADD A, #12H             // A = 3AH (Invalid BCD)
DA A                    // A = 40 (Desired BCD result)
```

(ii) When AC = 1 after addition:

```
    28 BCD    0010 1000
+   19 BCD    0001 1010
    47 BCD    0100 0001     41; AC is 1 after addition(Invalid result)
            + 0000 0110     add 6 to the lower nibble (DA A will do it)
              0100 0111     47 BCD, desired result
```

```
MOV A, #28H          // A = 28H
ADD A, #19H          // A = 41H (Invalid result)
DA A                 // A = 47H (Desired BCD result)
```

(iii) When upper nibble is greater than 9:

```
    82 BCD    1000 0010
+   21 BCD    0010 0001
   103 BCD    1010 0011     A3; upper nibble greater than 9(Invalid BCD)
            + 0110 0000     add 6 to the upper nibble (DA A will do it)
           1) 0000 0011     103 BCD, desired result
```

```
MOV A, #82H          // A = 82H
ADD A, #21H          // A = A3H (Invalid BCD)
DA A                 // A = 03, C = 1 (Desired BCD result)
```

(iv) When CY flag is set after addition:

```
    82 BCD    1000 0010
+   91 BCD    1001 0001
  173 BCD    10001 0011     carry flag set after addition (Invalid result)
           +  0110 0000     add 6 to the upper nibble (DA A will do it)
            1 0111 0011     173 BCD, desired result
```

```
MOV A, #82H          // A = 82H
ADD A, #91H          // A = 13H, C = 1(Invalid result)
DA A                 // A = 73H, C = 1 (Desired BCD result)
```

(V) It is also possible to have both the nibbles as non-BCD after addition as shown:

```
    63 BCD    0110 0011
+   88 BCD    1000 1000
  151 BCD    1110 1011     EB; both nibbles greater than 9(invalid BCD)
           +  0110 0110     add 6 to both the nibbles (DA A will do it)
            1 0101 0001     151 BCD, desired result
```

```
MOV A, #63H          // A = 63H
ADD A, #88H          // A = EBH (Invalid BCD)
DA A                 // A = 51, C = 1 (Desired BCD result)
```

Note: This instruction performs the BCD conversion by adding 00 H, 06H, 60H, or 66H to the Accumulator, depending on the contents of Accumulator, C or AC flag. Remember it does not simply convert a hexadecimal number in the Accumulator to BCD number. DA A does not apply to decimal subtraction.

DEC *BYTE*

Function Decrement the specified byte

It decrements the destination byte by 1. (Internally, it subtracts 1 from the specified destination byte.) The original value of 00H when decremented using this instruction, underflows to 0FFH.

Flags Affected None

The operand can be specified by any of the three addressing modes: Register (including Accumulator as a special case), Direct or Register Indirect. The format of the instruction for each addressing mode is given below.

DEC A	**// A = A–1**	**Bytes: 1**	**Cycles: 1**
DEC *Rn*	**// *Rn* = *Rn*–1**	**Bytes: 1**	**Cycles: 1**
DEC @*Ri*	**// (*Ri*) = (*Ri*) –1**	**Bytes: 1**	**Cycles: 1**
DEC *direct*	**// (direct) = (direct) –1**	**Bytes: 2**	**Cycles: 1**

Example

```
MOV R5, #10H         // R5 = 10H
DEC R5               // R5 = 0FH
DEC R5               // RF = 0EH
```

```
MOV R0, #20H          // R0 = 20H
MOV A, #0FFH          // A = FFH
MOV 20H, A            // (20H) = FFH
DEC @ R0              // (20H) = FEH
DEC A                 // A = FEH

MOV 20H, #00H         // (20H) = 00H
DEC 20H               // (20H) = FFH
```

DIV AB

Function Divide Accumulator by B

It divides the contents of the Accumulator by the contents of Register B. The contents of A and B registers are assumed to be unsigned numbers. After the division, the Accumulator contains quotient of the result and B contains the remainder of the result. The carry flag is always cleared. The OV flag will be set to 1 if an attempt to divide by 0 has been made, i.e. if B = 0 and DIV AB instruction is executed, otherwise OV is also cleared. The contents of A and B are undefined for divide by zero operation.

Flags Affected C(always cleared) , OV **Bytes:** 1 **Cycles:** 4

DIV AB // $A_{(Quotient)}B_{(Remainder)} = A/B$

Example

```
MOV A, #0FAH          // A = FAH
MOV B, #12H           // B = 12H
DIV AB                // A = 0DH, B = 10H, C = 0, OV = 0

MOV A, #80H           // A = 80H
MOV B, #00H           // B = 12H
DIV AB                // A = Undefined, B = Undefined, C = 0, OV = 1
```

DJNZ *BYTE, REL ADDRESS*

Function Decrement and jump if not zero

It decrements the specified byte first, and jumps to the specified relative address if the result (after decrement) is not zero; otherwise, program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. Note that it is a relative jump.

Flags Affected None

Two addressing-mode combinations are possible, the destination byte may be register or direct address. The format of the instruction for each addressing mode is given below.

DJNZ Rn, rel **// Rn = Rn-1; If Rn ≠ 0, PC = PC + rel** **Bytes:2** **Cycles: 2**
DJNZ direct, rel **// (direct) = (direct)-1; if (direct) ≠ 0, PC = PC + rel** **Bytes:3** **Cycles: 2**

The DJNZ instruction is used to repeat the loop for a fixed number of times. The typical structure of the loop using DJNZ instruction is shown below. It reads the value of P2 and sends it to P1 10 times.

```
       MOV R4, #10        // load the count (number of times loop to be repeated)
LOOP:  MOV A, P2          // A = P2begin loop
       MOV P1, A          // P1 = A

       DJNZ R4, LOOP      // check whether the loop is repeated required times, If not
                          // repeat it, otherwise, exit from the loop and continue
                          // program execution at the next instruction
       ...
```

INC *BYTE*

Function Increment the specified byte

It increments the destination byte by 1. (Internally, it adds 1 to the specified destination byte.) The original value of FFH when incremented using this instruction, overflows to 00H.

Flags Affected None

The operand can be specified by any of the three addressing modes: Register (including Accumulator as a special case), Direct or Register Indirect. The format of the instruction for each addressing mode is given below:

INC A	**// A = A + 1**	**Bytes: 1**	**Cycles: 1**
INC Rn	**// Rn = Rn + 1**	**Bytes: 1**	**Cycles: 1**
INC @Ri	**// (Ri) = (Ri) + 1**	**Bytes: 1**	**Cycles: 1**
INC direct	**// (direct) = (direct) + 1**	**Bytes: 2**	**Cycles: 1**
INC DPTR	**// DPTR = DPTR + 1**	**Bytes: 1**	**Cycles: 2**

Example

```
MOV R5, #10H          // R5 = 10H
INC R5                // R5 = 11H
INC R5                // R5 = 12H

MOV R0, #20H          // R0 = 20H
MOV A, #0FFH          // A = FFH
MOV 20H, A            // (20H) = FFH
INC @ R0              // (20H) = 00H
INC A                 // A = 00H

MOV 20H, #00H         // (20H) = 00H
INC 20H               // (20H) = 01H
```

INC DPTR

Function Increment the DPTR

It increments the DPTR by 1. The original value of FFFFH when incremented using this instruction, overflows to 0000H. Note that it is a modulo-16 operation, i.e. overflow of DPL (lower byte) from 0FFH to 00H increments the DPH (higher byte).

Flags Affected None

The format of the instruction is given below,

INC DPTR	**// DPTR = DPTR + 1**	**Bytes: 1**	**Cycles: 2**

Example

```
MOV DPTR, #10FFH          // DPTR = 10FFH
INC DPTR                  // DPTR = 1100H
INC DPTR                  // DPTR = 1101H
```

JB *BIT, REL*

Function Jump if bit set

It jumps to the relative address if specified bit is set (bit = 1), otherwise the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. The specified bit is not affected. This instruction is used to monitor the status of the specified bit and to take the decision based on its value.

Flags Affected None

The format of the instruction is given below:

JB bit, rel	**// If bit = 1, PC = PC + rel**	**Bytes: 3**	**Cycles: 2**

Example Assume that a pushbutton switch is connected to the pin P1.0. When the switch is pressed, logic low is given to the pin; otherwise, it remains at high logic. Following instructions toggle the status of the pin P2.0 every time the switch is pressed.

```
       SETB P1.0          // configure pin P1.0 as an input
WAIT:  JB P1.0, WAIT      // wait until the switch is pressed
       CPL P2.0           // complement P2.0
       SJMP WAIT          // repeat the operation
```

JBC *BIT, REL*

Function Jump if Bit is set and Clear bit

It jumps to the relative address if the specified bit is set (bit = 1) and also clears the bit to zero; otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. This instruction is used to monitor the status of the specified bit and to take the decision based on its value.

Flags Affected None (Flags are affected directly if bit address of the flag is specified)

The format of the instruction is given below:

JBC bit, rel **// If bit = 1, PC = PC + rel, bit = 0** **Bytes: 3** **Cycles: 2**

Example Assume that A = 21H (0010 0001) before the execution of following instructions:

```
JBC ACC.7, NEXT
JBC ACC.0, AHEAD
```

The above instructions cause program execution to continue at the label AHEAD and modifies the Accumulator to 20H (0010 0000).

Note: When the bit address of the port pin is specified, the original port pin status will be read from the port latch, not the port pins.

JC *REL*

Function Jump if Carry is set

It jumps to the relative address if Carry flag is set (C = 1); otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. This instruction is used to monitor the status of carry flag and to take the decision based on its value.

Flags Affected None

JC rel **// If C = 1, PC = PC + rel** **Bytes: 2** **Cycles: 2**

Example

```
       SETB C              // C = 1
       JC AHEAD            // since C = 1, continue the program execution to label
       …                   // AHEAD
       …
AHEAD: MOV A, P1
       …
```

JMP *@A + DPTR*

Function Jump indirect

It jumps to the address formed by the addition of Accumulator and DPTR (modulo 16 operation). Neither the Accumulator nor the DPTR is affected. No flags are affected. It is used to implement jump tables. The jump tables are used to dynamically jump to the different addresses based on values of certain variables.

Flags Affected None

JMP @A + DPTR **// PC = A + DPTR** **Bytes: 1** **Cycles: 2**

Example Jump table is implemented as shown below:

```
          MOV DPTR, # J_TABLE
          JMP @A + DPTR
          …
J_TABLE:  AJMP L1
          AJMP L2
          AJMP L3
          AJMP L4
```

If A = 0, execution proceeds to label L1 or if A = 6, the execution proceeds to label L4 (AJMP instruction is a 2-byte instruction)

JNB *BIT, REL*

Function Jump if bit not set (Jump if No Bit)

It jumps to the relative address if specified bit is cleared (bit = 0); otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. The specified bit is not affected. This instruction is used to monitor the status of the specified bit and to take the decision based on its value.

Flags Affected None

The format of the instruction is given below:

JNB bit, rel **// If bit = 0, PC = PC + rel** **Bytes: 3** **Cycles: 2**

Assume that a pushbutton switch is connected to the pin P1.0. When the switch is pressed, logic high is given to the pin; otherwise, it remains at low logic. The following instructions toggle the status of the pin P2.0 every time the switch is pressed.

```
      SETB P1.0          // configure pin P1.0 as an input
WAIT: JNB P1.0, WAIT     // wait until the switch is pressed
      CPL P2.0           // complement P2.0
      SJMP WAIT          // repeat the operation
```

JNC *REL*

Function Jump if Carry not set (Jump if No Carry)

It jumps to the relative address if Carry flag is not set (C = 0); otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. This instruction is used to monitor the status of carry flag and to take the decision based on its value.

Flags Affected None

JNC rel **// If C = 0, PC = PC + rel** **Bytes: 2** **Cycles: 2**

Example

```
        CLR C              // C = 0
        JNC AHEAD          // since C = 0, continue the program execution to label
        …                  // AHEAD
        …
AHEAD:  MOV A, P1
        …
```

JNZ *REL*

Function Jump if Accumulator is not zero (Jump if Not Zero)

It jumps to the relative address if the content of Accumulator is not zero (A≠0); otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. This instruction is used to monitor the contents of Accumulator and to take the decision based on its value.

Flags Affected None

JNZ rel **// if A≠0, PC = PC + rel** **Bytes: 2** **Cycles: 2**

Example

```
i)      CLR A              // A = 0
        JNZ AHEAD          // since A = 0, continue program execution to the next
        …                  // instruction
        …
AHEAD:  MOV A, P1
        …

ii)     MOV A, #10H        // A = 10H
        JNZ AHEAD          // since A≠0, continue the program execution to label
        …                  // AHEAD
        …
AHEAD:  MOV A, P1
        …
```

JZ *REL*

Function Jump if Accumulator is zero (Jump if Zero)

It jumps to the relative address if the content of Accumulator is zero (A = 0); otherwise, the program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. This instruction is used to monitor the contents of Accumulator and to take the decision based on its value.

Flags Affected None

JZ rel **// if A = 0, PC = PC + rel** **Bytes: 2** **Cycles: 2**

Example

```
i)      CLR A              // A = 0
        JZ AHEAD           // since A = 0, continue the program execution to label
        …                  // AHEAD
        …
AHEAD:  MOV A, P1
        …

ii)     MOV A, #10H        // A = 10H
        JZ AHEAD           // since A≠0, continue the program execution to next
        …                  // instruction
        …
AHEAD:  MOV A, P1
        …
```

LCALL *ADDR16* (LCALL *LABEL*)

Function Long call (anywhere within entire 64 K program memory)

It calls the subroutine located at the specified address within the entire 64 K program memory. The instruction automatically saves the address of the next instruction (address of instruction itself + 3) onto the stack (low byte first) and increments the stack pointer by two. The new value of PC (destination address) is obtained by loading the second and third bytes of the LCALL instruction into the PC lower byte and PC higher byte respectively. The program execution is then transferred to the new value of PC. Refer topic 7.3 for details.

Flags Affected None

LCALL addr 16 or Label **// SP = SP + 1;(SP) = PCL; SP = SP + 1; (SP) = PCH; PC = addr 16**

Bytes: 3 **Cycles:** 2

Example Assume that SP is initialized with the value 50H. The instruction LCALL DELAY is written at the program memory address 0100H, label DELAY is at location 0250H as shown below. After executing the following instruction,

```
00FF                …
0100                LCALL DELAY
0103                …
                    …
                    …
0250   DELAY:       …
                    …
```

After execution of instruction LCALL DELAY, the contents of the registers/memory locations will be changed as follows:

```
PC = 0250H          // program execution will start at the destination address
(51H) = 03H         // return address, i.e. address of the next instruction is saved
(52H) = 10H         // on the stack, lower byte first
SP = 52H            // SP is incremented by 2 because 2 bytes are saved on stack
```

LJMP *ADDR16*

Function Long Jump (anywhere within entire 64K program memory)

It transfers the program execution to the specified address anywhere within the entire 64K program memory. The new value of PC (destination address) is obtained by loading the second and third bytes of the LJMP instruction into PC lower byte and PC higher byte respectively. The program execution is then transferred to the new value of PC. Refer topic 7.1.1 for more details.

Flags Affected None **Bytes:** 3 **Cycles:** 2

Example The instruction LJMP SKIP is written at the program memory address 0100H, label SKIP is at location 0250H as shown below. After executing the following instruction:

```
00FF                …
0100                LJMP SKIP
0103                …
                    …
                    …
0250   SKIP:        …
                    …
```

After execution of instruction LJMP SKIP, the contents of the PC will be changed as follows:

```
PC = 0250H          // program execution will start at the destination address
```

MOV *DESTINATION BYTE, SOURCE BYTE*

Function Move (copy) source byte to destination byte

It moves (copies) the contents of the source byte into a specified destination byte. The source byte is not affected. This is the most flexible instruction. Fifteen combinations of source and destination are possible.

Flags Affected None

(i) Accumulator is the destination byte (operand)

MOV A, #data	**// A = data**	**Bytes: 2**	**Cycles: 1**
MOV A, Rn	**// A = Rn**	**Bytes: 1**	**Cycles: 1**
MOV A, direct	**// A = (direct)**	**Bytes: 2**	**Cycles: 1**
MOV A, @Ri	**// A = (Ri)**	**Bytes: 1**	**Cycles: 1**

Example Assume R2 = 20H, (30H) = 20H and R0 = 30H before execution of the following instructions:

```
MOV A, #10H         // A = 10H
MOV A, R2           // A = R2 = 20H
MOV A, 30H          // A = (30H) = 20H
MOV A, @R0          // A = (R0) = (30H) = 20H
MOV A, @R2          // Invalid, only R0 and R1 can be used for indirect addressing.
```

(ii) Rn is the destination byte (operand)

MOV Rn, #data	**// Rn = data**	**Bytes: 2**	**Cycles: 1**
MOV Rn, Rn	**// Invalid**		
MOV Rn, A	**// Rn = A**	**Bytes: 1**	**Cycles: 1**
MOV Rn, direct	**// Rn = (direct)**	**Bytes: 2**	**Cycles: 2**
MOV Rn, @Ri	**// Invalid**		

Example Assume A = 40H, (30H) = 20H before execution of the following instructions:

```
MOV R1, #50H        // R1 = 50H
MOV R2, A           // R2 = A = 40H
MOV R4, 30H         // R4 = (30H) = 20H
```

(iii) Direct address is the destination byte (operand)

MOV direct, #data	**// (direct) = data**	**Bytes: 3**	**Cycles: 2**
MOV direct, Rn	**// (direct) = Rn**	**Bytes: 2**	**Cycles: 2**
MOV direct*1*, direct*2*	**// (direct1) = (direct2)**	**Bytes: 3**	**Cycles: 2**

MOV direct, @Ri	**// (direct) = (Ri)**	**Bytes: 2**	**Cycles: 2**
MOV direct, A	**// (direct) = A**	**Bytes: 2**	**Cycles: 1**

Example Assume A = 40H, (30H) = 20H, (40H) = FFH and R1 = 30H before the execution of each of the following instructions:

```
MOV 20H, #0ABH      // (20H) = ABH
MOV 10H, R1         // (10H) = R1 = 30H
MOV 50H, 40H        // (50H) = (40H) = FFH
MOV 60H, @R1        // (60H) = (R1) = 20H
MOV 80H, A          // (80H) = A = 40H
```

(iv) Indirect address is the destination byte (operand)

MOV @Ri, #data	**// (Ri) = data**	**Bytes: 2**	**Cycles: 1**
MOV @Ri, A	**// (Ri) = A**	**Bytes: 1**	**Cycles: 1**
MOV @Ri, direct	**// (Ri) = (direct)**	**Bytes: 2**	**Cycles: 2**

Example Assume A = 40H, (30H) = 20H, R0 = 40H and R1 = 50H before the execution of following instructions:

```
MOV @R0, #55H       // (R0) = (40H) = 55H
MOV @R1, A          // (R1) = (50H) = A = 40H
MOV @R1, 30H        // (R1) = (50H) = (30H) = 20H
```

MOV *DESTINATION BIT, SOURCE BIT*

Function Move bit data

It moves (copies) the source bit (Boolean variable) into the specified destination bit. For this instruction, one of the operands must be the carry flag and the other may be any directly addressable bit.

Flags Affected None (Flags are affected directly if bit address of the flag is specified)

MOV C, bit	**// C = bit**	**Bytes: 2**	**Cycles: 1**
MOV bit, C	**// bit = C**	**Bytes: 2**	**Cycles: 2**

Example

```
CLR C               // C = 0
MOV P2.0, C         // P2.0 = C = 0
MOV P1, #55H        // P1 = 55H
MOV C, P1.0         // C = 1
MOV P2.2, C         // P2.2 = C = 1
```

MOV DPTR, *# DATA16*

Function Load Data Pointer with a 16-bit value

It moves the 16-bit data (only immediate) into the Data Pointer (DPTR) register. The 16-bit data is placed into the second and third bytes of the instruction. This is the only instruction in 8051 which moves 16 bits of data at a time.

Flags Affected None

MOV DPTR, #data16	**// DPTR = data16**	**Bytes: 3**	**Cycles: 2**

Example

```
MOV DPTR, #1000H    //DPTR = 1000H or DPH = 10H, DPL = 00H
```

MOVC A, @A + BASE REGISTER

Function Move code byte in to A

It copies the data (or code) byte from the program memory. The address of the byte is formed by addition of contents of A and 16-bit base register, which may be either the DPTR or the PC.

When the base address is PC, the PC is incremented to the address of the next instruction before being added with the Accumulator. The base registers are not affected. Modulo–16 addition is performed while the addition of A and base register. These instructions are used to access the data from look-up tables. Refer topic 8.4 for details.

Flags Affected None

MOVC A, @A + DPTR	**// A = $(A + DPTR)_{code}$**	**Bytes: 1**	**Cycles: 2**
MOVC A, @A + PC	**// A = $(A + PC)_{code}$**	**Bytes: 1**	**Cycles: 2**

Example

(i) for MOVC A, @A + DPTR

```
ORG 0000H
MOV DPTR, #1000H      // load the address of look-up table in DPTR
MOV A, #02H           // value for which square is to be found
MOVC A, @A + DPTR     // access look-up table, place the result in A
…
ORG 1000H             // define lookup table at ROM address 0100H
DB 0H, 1H, 4H, 9H, 10H,19H,24H,31H,40H,51H,64H
```

(ii) for MOVC A, @A + PC

```
ORG 0000H
MOV A, #02H           // value for which square is to be found
ADD A, #02H           // look-up table is placed 2 bytes ahead w.r.t. MOVC instruction.
MOVC A, @A + PC       // access look-up table, place result in A
SJMP DONE             // skip lookup table
                      // define lookup within the program
DB 0H, 1H, 4H, 9H, 10H, 19H, 24H, 31H, 40H, 51H, 64H
DONE: SJMP DONE
```

The above programs will find the square of number present in A. The result will be available in A after execution of MOVC A, @A + DPTR or MOVC A, @A + PC instruction. The square is found by accessing the corresponding entry from the look-up table defined in the program memory. The second method is used when we do not want to divide the program space into code and data spaces.

MOVX *DESTINATION BYTE, SOURCE BYTE*

Function Move to/from external data memory

It transfers the data between the Accumulator and a byte from external data memory. The suffix 'X' in the mnemonic indicates the external data memory. The address of the data byte can be specified by 8-bit or 16-bit indirect address. The 8-bit address can be specified either by R0 or R1. Using these instructions, we can access only 256 bytes and it is usually used to access I/O ports. The 16-bit address is specified by the DPTR register and any byte from the entire 64 K data memory can be accessed.

Flags Affected None

MOVX A, @DPTR	**// A = $(DPTR)_{Ext\ RAM}$**	**Bytes: 1**	**Cycles: 2**
MOVX @DPTR, A	**// $(DPTR)_{Ext\ RAM}$ = A**	**Bytes: 1**	**Cycles: 2**
MOVX A, @Ri	**// A = $(Ri)_{Ext\ RAM}$**	**Bytes: 1**	**Cycles: 2**
MOVX @Ri, A	**// $(Ri)_{Ext\ RAM}$ = A**	**Bytes: 1**	**Cycles: 2**

Example Assume DPTR = 0100H, $(0100)_{Ext\ RAM}$ = 20H, R0 = 30H and $(30H)_{Ext\ RAM}$ = 35H before execution of the following instructions:

```
MOVX A,@DPTR          // A = 20H
MOVX A, @R0           // A = 35H
```

Assume A = 55H, DPTR = 0100H and R1 = 80H before execution of the following instructions:

```
MOVX @DPTR, A         // (0100)Ext RAM = 55H
MOVX @R1, A           // (80H) Ext RAM = 55H
```

MUL AB

Function Multiply

It multiplies the contents of Accumulator with B and places the lower byte of the result in Accumulator and higher byte of the result in B. The bytes in Accumulator and B are assumed to be unsigned. The overflow will be set *t* to 1, if A × B > FFH. Here, OV flag does not mean that an error has occurred. But, it signals that the result is larger than 8 bits and we need to consider B register for higher byte of the result.

Flags Affected C (always cleared), OV

MUL AB	**// $B_{(MSB)}A_{(LSB)}$ = A x B**	**Bytes: 1**	**Cycles: 4**

Example

```
MOV A, #07H          // A = 07H
MOV B, #08H          // B = 08H
MUL AB               // A = 38H = 56, B = 00H C = 0, OV = 0

MOV A, #0FFH         // A = FFH
MOV B, #0FFH         // B = FFH
MUL AB               // A = 01H, B = FEH C = 0, OV = 1
```

NOP

Function No operation

It performs no operation and the execution continues to the next instruction. It is commonly used in generating time delays to waste the machine cycles.

Flags Affected None

NOP	// No operation	**Bytes:** 1	**Cycles:** 1

Example The following instructions will generate a positive pulse of 4 machine cycles on P1.0:

```
SETB P1.0
NOP
NOP
NOP
CLR P1.0
```

Note that CLR instruction also requires one machine cycle. Therefore, the time duration of pulse is 4 cycles (3 cycles of NOPs + 1 cycle of CLR P1.0)

ORL *DESTINATION BYTE, SOURCE BYTE*

Function Bitwise OR operation between source and destination byte.

It performs the bitwise logical OR operation between the source and destination operands and stores the result in the destination operand. The Accumulator or a direct address in the internal RAM can be the destination. When the destination is the Accumulator, the source can use the immediate register, direct or register indirect addressing, and when the destination is a direct address, the source can be the immediate data or Accumulator.

Flags Affected None [Flags may be affected when destination operand is PSW (direct address)]

The format of the instruction for each addressing mode is given below:

ORL A, #data	**// A = A OR data**	**Bytes: 2**	**Cycles: 1**
ORL A, Rn	**// A = A OR Rn**	**Bytes: 1**	**Cycles: 1**
ORL A, @Ri	**// A = A OR (Ri)**	**Bytes: 1**	**Cycles: 1**
ORL A, direct	**// A = A OR (direct)**	**Bytes: 2**	**Cycles: 1**
ORL direct, #data	**// (direct) = (direct) OR data**	**Bytes: 3**	**Cycles: 2**
ORL direct, A	**// (direct) = (direct) OR A**	**Bytes: 2**	**Cycles: 1**

Example

```
MOV A, #0FEH          // A = FEH
ORL A, #10H           // A = FEH OR 10H = FEH

MOV R0, #4AH          // R0 = 4AH
MOV A, #55H           // A = 55H
ORL A, R0             // A = 55H OR 4AH = 5FH

MOV 10H, #72H         // (10H) = 72H
MOV A, #0FH           // A = 0FH
ORL 10H, A            // (10H) = 72H OR 0FH = 7FH
```

The OR operation (ORL instruction) is often used to set certain bits of a result to 1 as shown in the example below (see highlighted bits).

```
      34H   0011 0100
OR    57H   0101 0111
      77H   0111 0111
```

Note: When the destination operand is a port address, the original port data will be read from the port latch, not the port pins.

ORL C, *SOURCE OPERAND BIT*

Function Logical OR operation between Carry and source operand bit

It performs the logical OR operation between the Carry flag and source operand bit (Boolean) and stores the result in the Carry flag. If the source bit is a 1 then the carry flag is set; otherwise, the carry flag remains unchanged.

A slash (/) preceding the source operand bit indicates that the complement of the bit is used as the source value, but the source bit is not affected. No other flags are affected. Only direct addressing is allowed for the source operand.

Flags Affected C

The format of the instruction for each addressing mode are given below:

ORL C, bit	**// C = C OR (bit)**	**Bytes: 2**	**Cycles: 2**
ORL C, /bit	**// C = C OR ($\overline{\text{bit}}$)**	**Bytes: 2**	**Cycles: 2**

Example

```
MOV A, #0FFH          // A = FFH
CLR C                 // C = 0
ORL C, ACC.0          // C = C OR ACC.0 = 0 OR 1 = 1

MOV A, #0FFH          // A = FFH
CLR C                 // C = 0
ORL C, /ACC.0         // C = C OR ACC.0 = 0 OR 0 = 0
```

POP *DIRECT*

Function Pop from the stack

It reads (copies) a byte from the stack (internal RAM address pointed by the SP) and the value read is then transferred to the specified direct address. The SP is decremented by one after reading a byte.

Flags Affected None (Flags will be affected if the specified direct addess is PSW)

POP direct	**// (direct) = (SP); SP = SP-1**	**Bytes: 2**	**Cycles: 2**

Example Assume (45H) = 20H, (44H) = 10H and SP = 45H before execution of the following instructions:

```
POP 80                // P0 (80H) = 20H, SP = 44H
POP E0                // A(E0H) = 10H, SP = 43H
```

PUSH *DIRECT*

Function Push onto the stack

Description The Stack Pointer is first incremented by one. The contents specified the direct address is then copied onto the stack (internal RAM address pointed by the SP).

Flags Affected None

PUSH direct **// SP = SP + 1; (SP) = (direct)** **Bytes: 2** **Cycles: 2**

Example Assume B = 20H, A = 15H and SP = 45H before execution of the following instructions:

```
PUSH F0              // SP = 46H, (46H) = (F0H) = 20H
PUSH E0              // SP = 47H, (47H) = (E0H) = 15H
```

RET

Function Return from subroutine

It is used to return to the main program from the subroutine which was called by ACALL or LCALL instruction. It pops (retrieves) two bytes from the top of the stack (return address) into the PC and the SP is decremented by two. The program execution continues at the return address, usually the instruction immediately following an ACALL or LCALL.

Flags Affected None

RET **// PCH = (SP); SP = SP-1; PCL = (SP); SP = SP-1** **Bytes: 1** **Cycles: 2**

Example Assume (45H) = 10H, (44H) = 20H and SP = 45H before execution of the following instructions:

```
RET  // PC = 1020H, SP = 43H. Program execution continues at address 1020H.
```

RETI

Function Return from the interrupt service routine

It is used to return to the main program from the Interrupt Service Routine (ISR). It pops (retrieves) two bytes from the top of stack (return address) into the PC and SP is decremented by two. It also retrieves the saved status of interrupt enable bits, thus, enabling the same and lower priority interrupts again, i.e. restores the interrupt logic. The program execution continues at the return address, which is usually the next instruction after the last instruction executed before entering into ISR, i.e. instruction at which the interrupt request was detected.

Flags Affected None

RETI **// PCH = (SP); SP = SP-1; PCL = (SP); SP = SP-1** **Bytes: 1** **Cycles: 2**

Example Assume (45H) = 10H, (44H) = 20H and SP = 45H before execution of the following instructions,

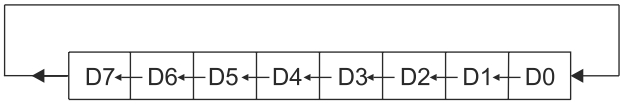
```
RETI      // PC = 1020H, SP = 43H.
```

Program execution continues at address 1020H and also // restores the interrupt logic.

RL A

Function Rotate accumulator left

It rotates the eight bits of Accumulator one bit position to the left, bit D0 to D1, bit D1 to D2, …, bit D6 to D7 and bit D7 to D0 as illustrated in the figure given below:



RL A

Flags Affected None

RL A **// ACC.0 = ACC.7, ACC.7 = ACC.6… ACC.1 = ACC.7** **Bytes: 1** **Cycles: 1**

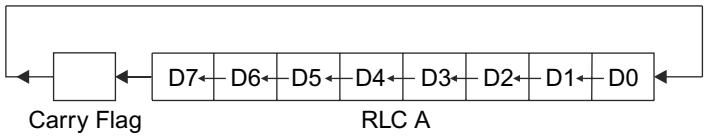
Example

```
MOV A, #43H          // A = 0100 0011
RL A                 // A = 1000 0110
RL A                 // A = 0000 1101
```

RLC A

Function Rotate accumulator left through carry

It rotates eight bits of the Accumulator one bit position to the left through Carry flag, bit D0 to D1, bit D1 to D2, …, bit D6 to D7, bit D7 to CY and CY to D0 as illustrated in the figure given below:



Flags Affected C

RLC A **// C = ACC.7,ACC.7 = ACC.6,… ACC.0 = C** **Bytes: 1** **Cycles: 1**

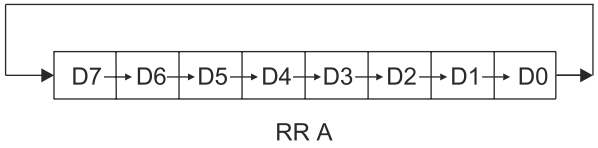
Example

```
CLR C                // CY = 0
MOV A, #0D6H         // A = 1101 0110, CY = 0
RLC A                // A = 1010 1100, CY = 1
```

RR A

Function Rotate accumulator right

It rotates the eight bits of Accumulator one bit position to the right, bit D0 to D7, bit D7 to D6, …, bit D2 to D1 and bit D1 to D0 as illustrated in the figure given below:



Flags Affected None

RR A **// ACC.7 = ACC.0, ACC.6 = ACC.7… ACC.0 = ACC.1** **Bytes: 1** **Cycles: 1**

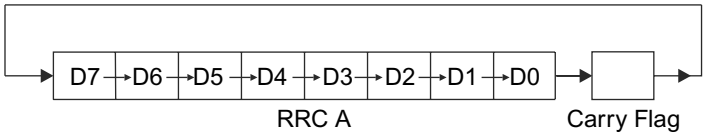
Example

```
MOV A, #35H          // A = 0011 0101
RR A                 // A = 1001 1010
```

RRC A

Function Rotate accumulator right through carry

It rotates the eight bits of Accumulator one bit position to the right through carry flag, bit D0 to CY, CY to D7, bit D7 to D6, …, bit D2 to D1 and bit D1 to D0 as illustrated in the figure given below:



Flags Affected C

RRC A **// C = ACC.0, ACC.7 = C,… ACC.0 = ACC.1** **Bytes: 1** **Cycles: 1**

Example

```
SETB C               // CY = 1
MOV A, #62H          // A = 0110 0010, CY = 1
RRC A                // A = 1011 0001, CY = 0
```

SETB *BIT*

Function Set bit

It sets the specified bit (bit = 1). This instruction can operate on carry flag or any bit-addressable location.

Flags Affected None (Flags are affected directly if the bit address of the flag is specified)

SETB C	**// C = 1**	**Bytes: 1**	**Cycles: 1**
SETB bit	**// bit = 1**	**Bytes: 2**	**Cycles: 1**

Example

```
SETB P1.0          // P1.0 = 1
SETB C             // C = 1
SETB 00H           // bit address location 00H = 1, i.e. LSB of byte 20H is set
```

SJMP *REL*

Function Short jump

It transfers the program execution to the specified relative address with respect to PC, i.e. the range of destination address is from –128 bytes to + 127 bytes from the address of the next instruction. The destination address is calculated by adding the relative address with the PC. Refer topic 7.1.1 for details.

Flags Affected None

SJMP rel	**// PC = PC + rel**	**Bytes: 2**	**Cycles: 2**

Example The instruction SJMP SKIP is written at the program memory address 0100H, label SKIP is at the location 0150H as shown below. The program executies the following instruction:

```
00FF               …
0100               SJMP SKIP
0102               …
                   …
                   …
0150  SKIP:        …
```

After execution of the instruction SJMP SKIP, the contents of PC will be changed as follows:

```
PC = 0150H         // program execution will start at the destination address
```

SUBB A, *SOURCE OPERAND BYTE*

Function Subtract with borrow

It subtracts the specified source operand byte as well as Carry flag from the Accumulator, and stores the result in the Accumulator. It sets the carry (borrow) flag if a borrow is needed for bit 7, otherwise clears it. (If C was already set before executing a SUBB instruction, this means that a borrow was needed for the previous step in a multi-byte subtraction). AC is set if a borrow is needed for bit 3; otherwise clears it. OV is set if a borrow is needed into bit 7, but not in bit 6, or into bit 6, but not in bit 6. When subtracting the signed numbers, OV indicates an overflow occurred, i.e. result is outside the signed number range –128 to + 127 and result is wrong.

Note that there is no SUB instruction in the 8051. To perform the SUB operation, clear the carry flag (C = 0) and use the SUBB instruction.

Flags Affected C, OV, AC

The source operand can be specified by any of the four addressing modes: Immediate, Register, Direct or Register Indirect.

SUBB A, #data	**// A = A – data – C**	**Bytes: 2**	**Cycles: 1**
SUBB A, Rn	**// A = A – Rn – C**	**Bytes: 1**	**Cycles: 1**
SUBB A, direct	**// A = A – direct – C**	**Bytes: 2**	**Cycles: 1**
SUBB A, @Ri	**// A = A – (Ri) – C**	**Bytes: 1**	**Cycles: 1**

Example

Unsigned Subtraction Assuming that all the numbers in the following instructions are unsigned numbers.

```
CLR C
MOV A, #30H          // A = 30H
SUBB A, #10H         // A = A-10H = 30H-10H = 20H

CLR C
MOV A, #30H          // A = 30H
MOV R0, # 10H        // R0 = 10H
SUBB A, R0           // A = A-R0 = 30H-10H = 20H
```

Signed Subtraction Assuming that all the numbers in the following instructions are signed numbers.

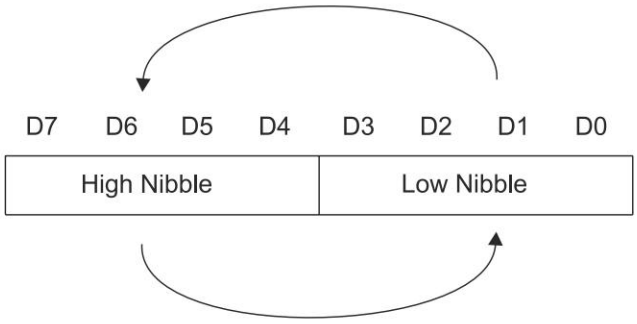
```
CLR C                                  // C = 0
MOV A, #BAH    or MOV A, #-70H         // A = BAH
SUBB A, #64H                           // A = 56H, C = 0, OV = 1(Incorrect result)

CLR C          // C = 0
MOV A, #65H    or MOV A, # + 101       // A = 65H
SUBB A, #78H   or SUBB A, # + 120      // A = EDH = -19H, C = 1, OV = 0 (Correct result)
```

SWAP A

Function Swap nibbles within Accumulator

It swaps the nibbles of the Accumulator, i.e. it interchanges the upper nibble with lower nibble. This operation is equivalent to 4-bit rotation in either left or right direction. The operation of the swap instruction is illustrated in the figure given below:



Flags Affected None

SWAP A	**// $A_{LN} = A_{HN}$; $A_{HN} = A_{LN}$**	**Bytes: 1**	**Cycles: 1**

Example

```
MOV A, #57H          // A = 57H
SWAP A               // A = 75H
```

XCH A, *SOURCE OPERAND*

Function Exchange Accumulator with source operand

It exchanges the contents of the Accumulator and the specified source operand byte.

Flags Affected None

The source operand can be specified by any of the three addressing modes: Register, Direct or Register Indirect.

XCH A, Rn	**// A = Rn; Rn = A**	**Bytes: 1**	**Cycles: 1**
XCH A, direct	**// A = (direct); (direct) = A**	**Bytes: 1**	**Cycles: 1**
XCH A, @Ri	**// A = (Ri); (Ri) = A**	**Bytes: 1**	**Cycles: 1**

Example

```
MOV A, #35H          // A = 35H
MOV R2, #70H         // R2 = 70H
XCH A, R2            // A = 70H, R2 = 35H
```

```
MOV A, #50H          // A = 50H
MOV 40H, #70H        // (40H) = 70H
XCH A, 40H           // A = 70H, (40H) = 50H

MOV A, #35H          // A = 35H
MOV R0, #10H         // R0 = 10H
MOV 10H, #20H        // (10H) = 20H
XCH A, @R0           // A = 20H, (10H) = 35H
```

XCHD A, *@RI*

Function Exchange digits (nibbles)

It exchanges only lower nibble of the Accumulator and the lower nibble of internal RAM byte pointed by Ri. The higher nibbles of both the operands are not affected.

Flags Affected None

XCHD A, @Ri **// $A_{LN} = (Ri)_{LN}; (Ri)_{LN} = A_{LN}$** **Bytes: 1** **Cycles: 1**

Example

```
MOV A, #35H          // A = 35H
MOV R0, #10H         // R0 = 10H
MOV 10H, #20H        // (10H) = 20H
XCHD A, @R0          // A = 30H, (10H) = 25H
```

XRL *DESTINATION BYTE, SOURCE BYTE*

Function Bitwise EX-OR operation between the source and destination byte.

It performs the bitwise logical EX-OR operation between the source and destination operands and stores the result in the destination operand. The Accumulator or a direct address in internal RAM can be destination. When the destination is the Accumulator, the source can use the immediate register, direct or register indirect addressing, and when the destination is a direct address, the source can be the immediate data or Accumulator.

Flags Affected None (Flags may be affected when destination operand is PSW(direct address))

The format of the instruction for each addressing mode is given below:

XRL A, #data	**// A = A EX-OR data**	**Bytes: 2**	**Cycles: 1**
XRL A, Rn	**// A = A EX-OR Rn**	**Bytes: 1**	**Cycles: 1**
XRL A, @Ri	**// A = A EX-OR (Ri)**	**Bytes: 1**	**Cycles: 1**
XRL A, direct	**// A = A EX-OR (direct)**	**Bytes: 2**	**Cycles: 1**
XRL direct, #data	**// (direct) = (direct) EX-OR data**	**Bytes: 3**	**Cycles: 2**
XRL direct, A	**// (direct) = (direct) EX-OR A**	**Bytes: 2**	**Cycles: 1**

Example

```
MOV A, #0FEH         // A = FEH
XRL A, #10H          // A = FEH EX-OR 10H = EEH

MOV R0, #4AH         // R0 = 4AH
MOV A, #55H          // A = 55H
XRL A, R0            // A = 55H EX-OR 4AH = 1FH

MOV 10H, #72H        // (10H) = 72H
MOV A, #0FFH         // A = FFH
XRL 10H, A           // (10H) = 72H EX-OR FFH = 8DH
```

The EX-OR operation (XRL instruction) is often used to invert certain bits of an operand, i.e. if any bit is EX-ORed with 1, it will be inverted (see highlighted bits). EX-OR operation may also be used to see if the two registers (or two bits) have the same value. If two bits of the same value are EX-ORed, the result will be always zero.

```
        A4H   1010 0100
EX-OR   71H   0111 0001
        D5H   1101 0101
```

Note: When destination operand is a port address, the original port data will be read from the port latch, not the port pins.

Appendix B

Using Keil μVision 4.0 IDE

Integrated Development Environment (IDE) is the development software which integrates all the tools necessary to develop, test and debug the programs for an application. It includes the editor, assembler, compiler, linker, simulator, tracer, emulator and logic analyzer. It can be used for programming in both assembly and C language. It also supports the Flash burning process.

A **µVision 4.0** (microvision 4.0) is popular and a user-friendly IDE from Keil software Inc. designed for Cortex-Mx, ARM7, ARM9, C166, XE166, XC2000 and C51(8051 and all its variants) microcontrollers, which combines project management, make facilities, source-code editing and program debugging in one environment. The µvision integrates all the necessary program-development tools in a single application that provides a seamless embedded project-development environment. It includes Ax51 macro assembler (Ax51 means A51or A251 or AX51 assembler, capital X indicates extended versions like NXP 80151MX, Dallas 390, etc.); Lx51 linker/locater, Libx51 library manager, OHx51 object-to-hex converter and Cx51 ANSI C compiler. The Cx51 compiler and the Linker/Locator provide the optimum 8051 architecture support.

CREATING A SAMPLE APPLICATION PROGRAM USING KEIL μVISION 4.0 IDE

This tutorial describes the basics of µVision and shows how to use the user interface to create a sample program in assembly ('.S') as well in 'C' language. To support and ease the understanding, a stepwise explanation along with the screenshots of µVision 4.0 IDE windows is given for a sample program. The sample programs are chosen such that the maximum features of IDE will be explored while testing them.

Note: All the snapshots given in the chapter are captured from evaluation version of µVision 4.0 IDE. The evaluation- version support limited program size up to only 2 Kbytes.

DEVELOPING AND TESTING A PROGRAM IN ASSEMBLY LANGUAGE USING KEIL µVISION 4.0 IDE

Opening an Application

For opening an application, select **Start → Keil µVision 4.0**. The screen (environment) will be opened as shown in Figure B.1. The screen has three major windows as shown in Figure B.1.

Project Window It shows the source files and groups, register window, functions window and book window.

Workspace or Editor Window It shows all the opened source files, disassembly window and performance information. Source codes are edited in this window.

Output Window It shows the program-build progress information (assembling, linking, or compiling, errors and warnings.) It also shows the command, memory, call stack and local variable windows.

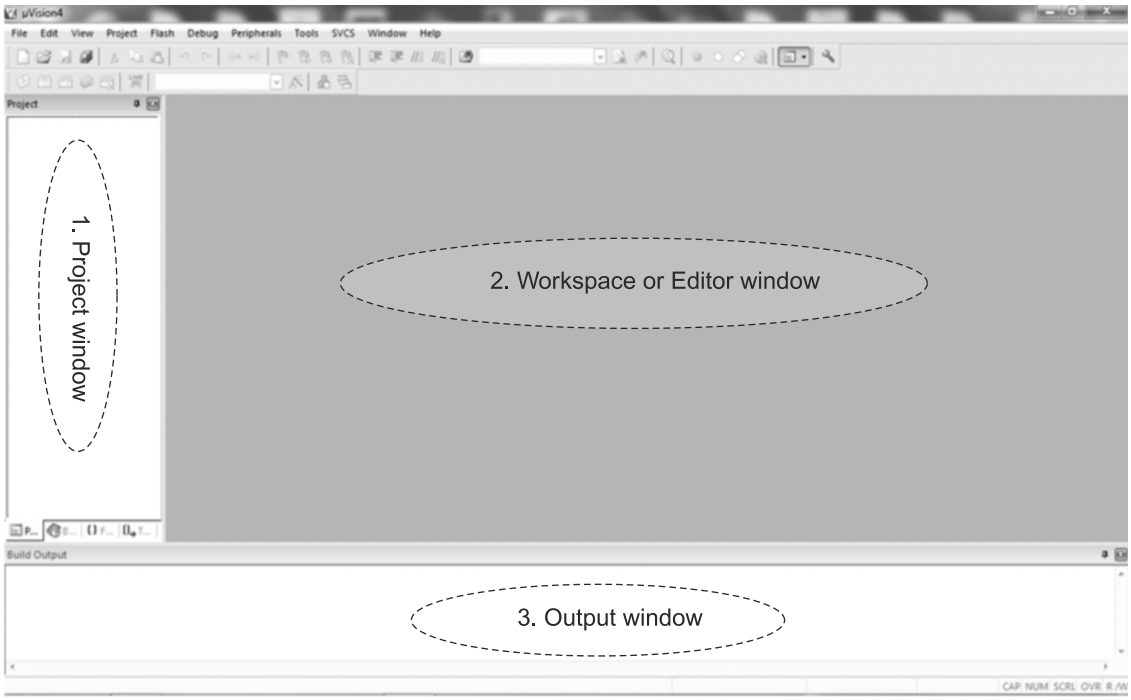


Fig. B.1 *Default µVision 4.0 Window*

Creating a new Project

To create a new project, select **Project → New µVision project** from the µVision top menu bar as shown in Figure B.2**.** This will open a "Create New Project" window dialog that asks you for the new project name as shown in Figure. B.3.

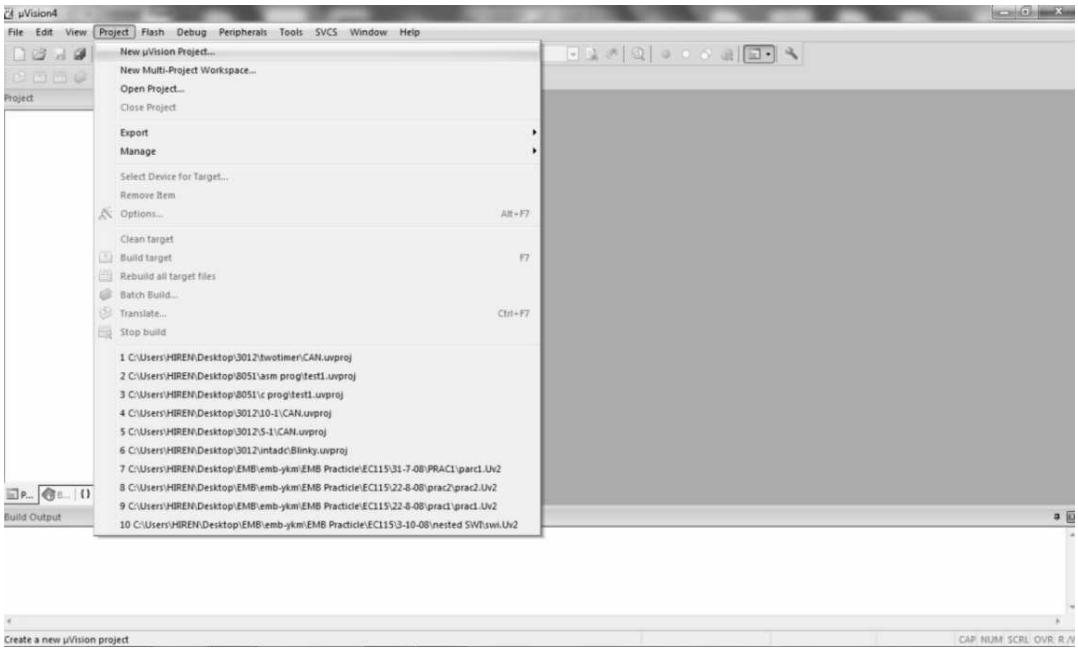


Fig. B.2 *Creating a new project*

In this window, browse to the folder where you want to store all the project files, or create the new folder in desired disk drive) see Figure B.3. Type a desired project name in the **File name** box and **Save** it. This will create a project file with the given name in desired folder. µVision4 automatically assigns the extension **.uvproj** to the project. It is advisable to use a separate folder for each project.

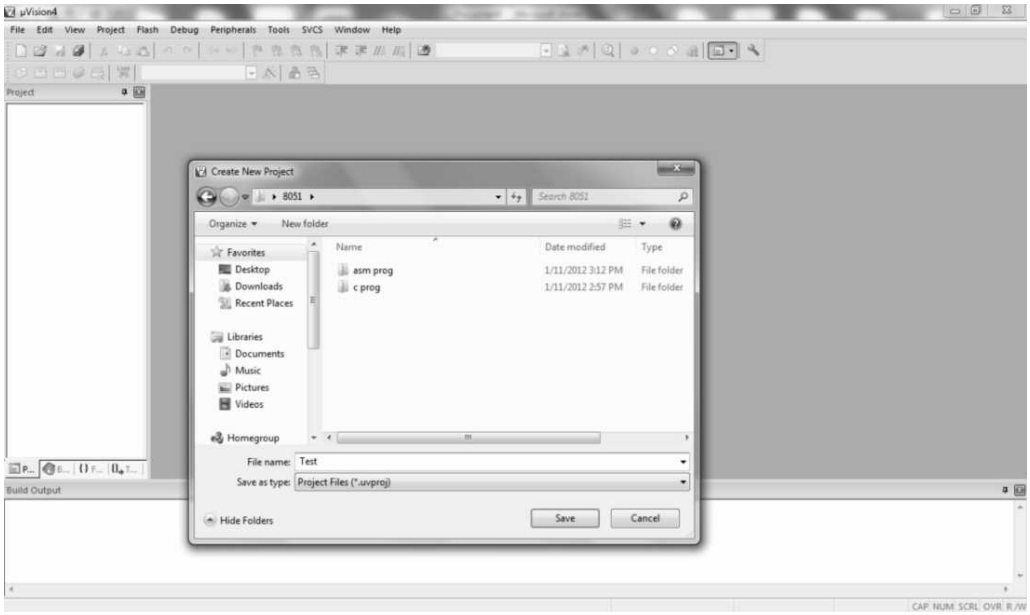


Fig. B.3 *Assigning a project name and location to store it.*

Selecting a Device (Microcontroller)

Whenever a new project is created, it is necessary to select a microcontroller (referred as target device) for which the project is developed. The **Select Device** dialog box (appear after above step) shows the µVision device database. Select **Atmel→AT89C51** (or other device as per requirement) as shown in Figure B.4.

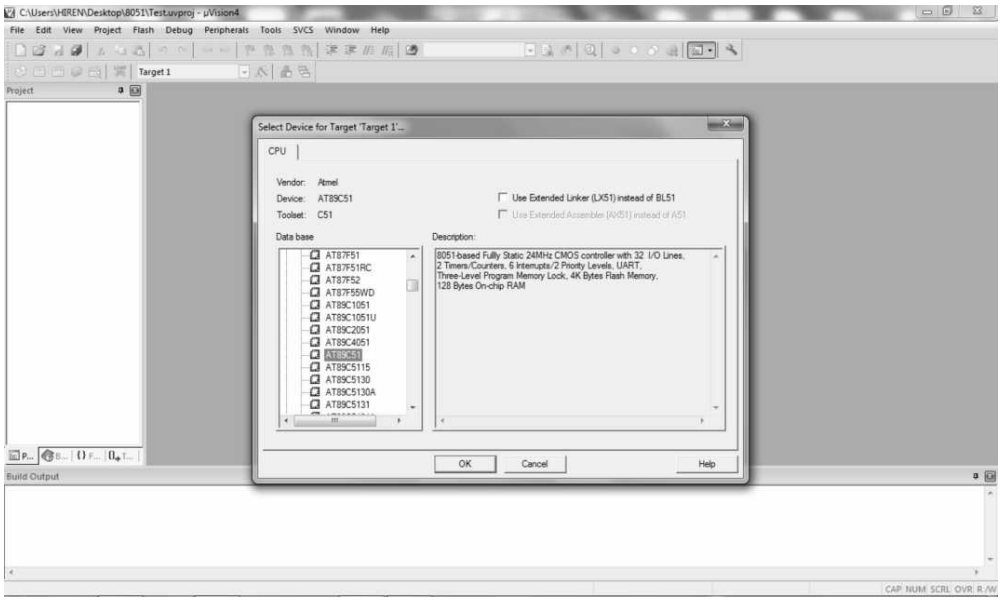


Fig. B.4 *Selecting the target device*

After the device selection, there will be one notification, asking for adding 8051startup code or not as shown in Figure B.5. Select **No.**

Note that if you want to make the program in assembly (.s) language then it is not needed to add this startup file. This file is needed only when you want to write the program in C language.

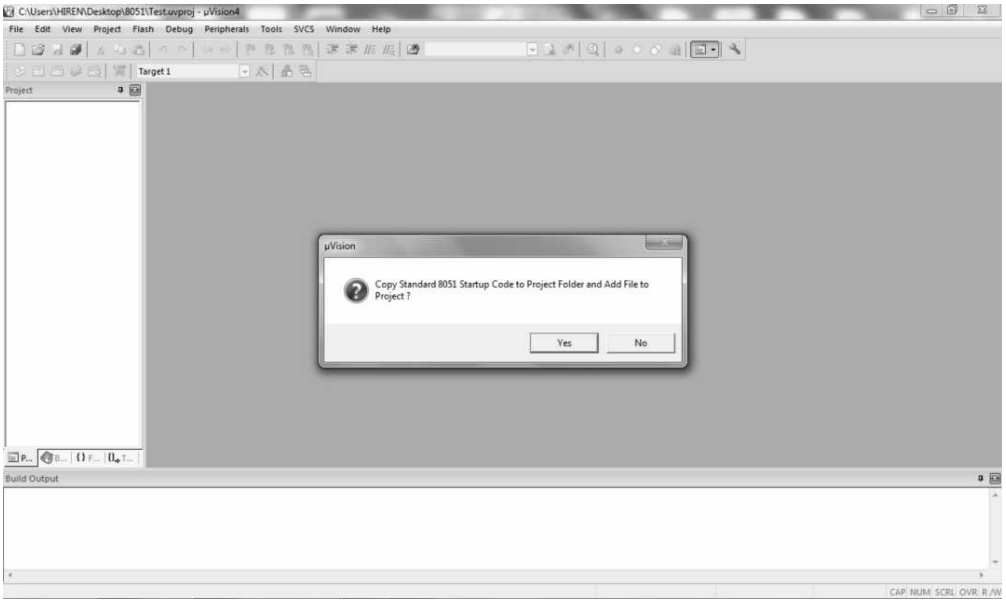


Fig. B.5 *Adding startup file to the project*

Now, the user has one project ready for the application build up.

Set Target Options

To set options for the target microcontroller, select **Project → Options for target 'Target1…'** or **(Alt + F7).** The window as shown in Figure B.6 appears. All the hardware parameters for the selected microcontroller device can be set in this window. Set the crystal frequency and other parameters as per the target board.

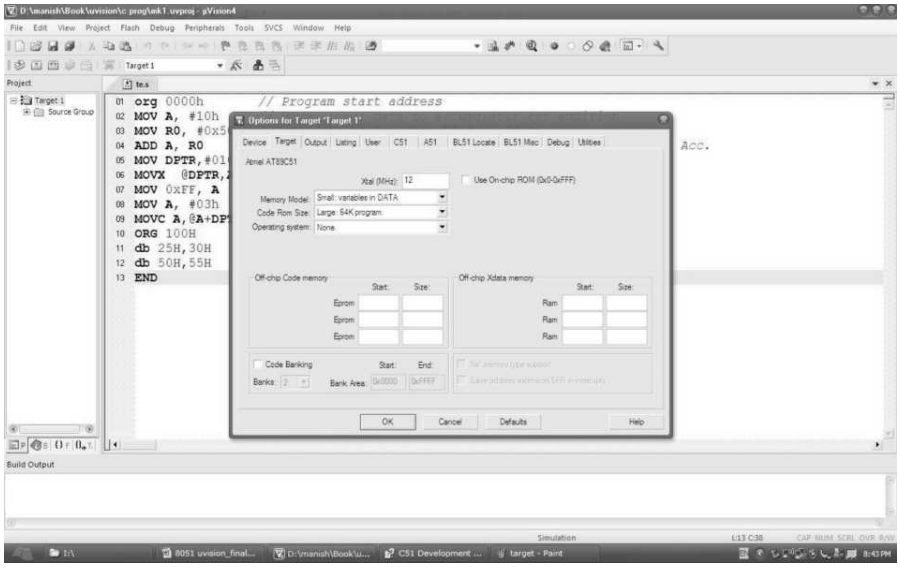


Fig. B.6 *Options for the target device*

Creating Source Code file/s and adding to the Project

For writing (editing) the source program, go to **File → New** or press **ctrl + n**. One blank file will be opened, type the program as per the application requirement. The sample program used for demonstration is listed below.

Sample Assembly-Language (.s) Program

```
ORG 0000h            ; program start address
MOV A, #10h          ; move (load) data to accumulator for addition
MOV R0, #20h         ; move data to Register R0 for addition
ADD A, R0            ; add content of Reg. R0 to Acc. and store result into Acc.
MOV DPTR, #0100h     ; move 16-bit address to DPTR register
MOVX @DPTR, A        ; move(store) data to the external memory
MOV 0xFF, A*         ; move data to internal memory (data memory) (see note below)
MOV A, #03h          ; load pointer (offset ) value in ACC.
MOVC A, @A+DPTR      ; load data from code memory (dptr +Acc) to Acc.
ORG 100H             ; store data at address 0x100h
db 25H, 30H          ; store data 0x25 at 0x100,and 0x30 to 0x0101
db 50H, 55H          ; store data 0x50at 0x102,and 0x55 to 0x0103
END                  ; end of the program
```

*Note that the instruction '**MOV 0xFF, A'** tries to access an address (0xFF) which physically does not exist on the 8051 microcontroller. The instruction is included in the program to emphasize that the assembler will not report any error for such a case. It is the responsibility of a programmer to take care to avoid accessing physically non-existent addresses.

Type the program and save the file in the folder where the project is stored. See Figure B.7. The source program file is saved with the name **te.s** (.s or .src or .asm or .a51 extension for assembly-language programs). Now it is necessary to add this file in your project.

For adding the source file to the project, go to the Project window and right click on '**Source Group 1'** and select **Add files to Group...** as shown in Figure B.8 and add the source program file.

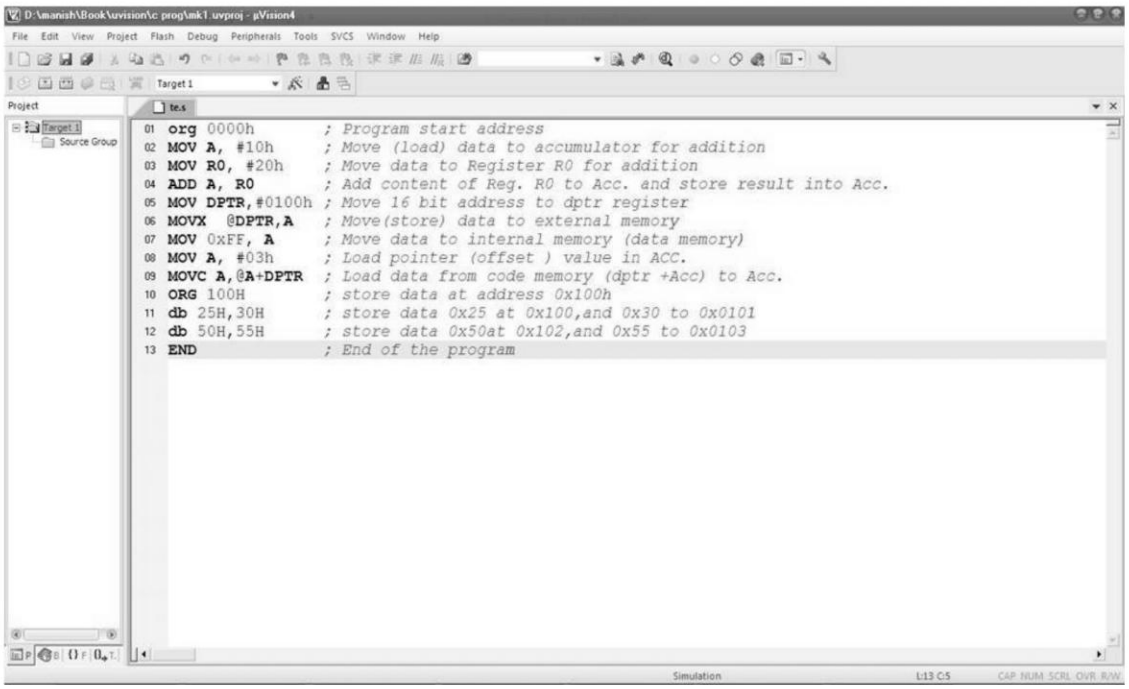


Fig. B.7 *Editing the source program file*

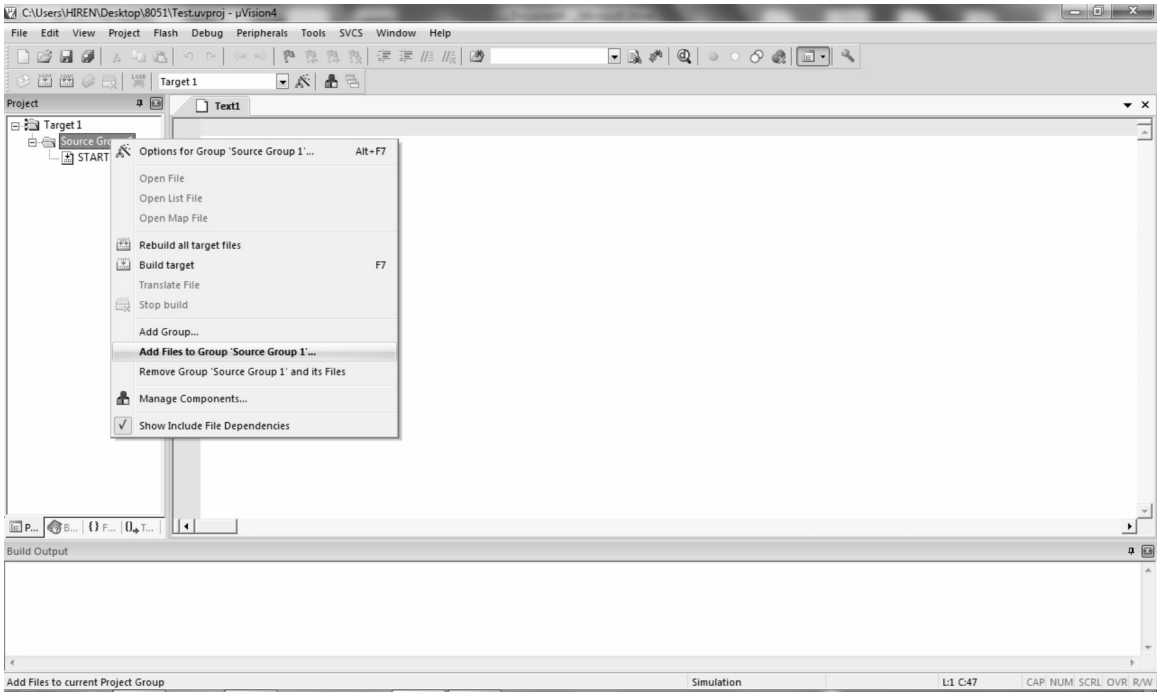


Fig. B.8 *Adding source file to the project*

Building the Project

Building involves assembling of each source file (compiling for .C files) and linking all the object files created by the assembler or compiler and to generate the final executable file. To build the program, go to **Project Build target (or press F7).** This will give a list of errors and warning if any in Build output window as shown in Figure B.9. Correct the errors and warnings in a source file(s) and repeat the process of building until you get '0' Errors and '0' Warnings.

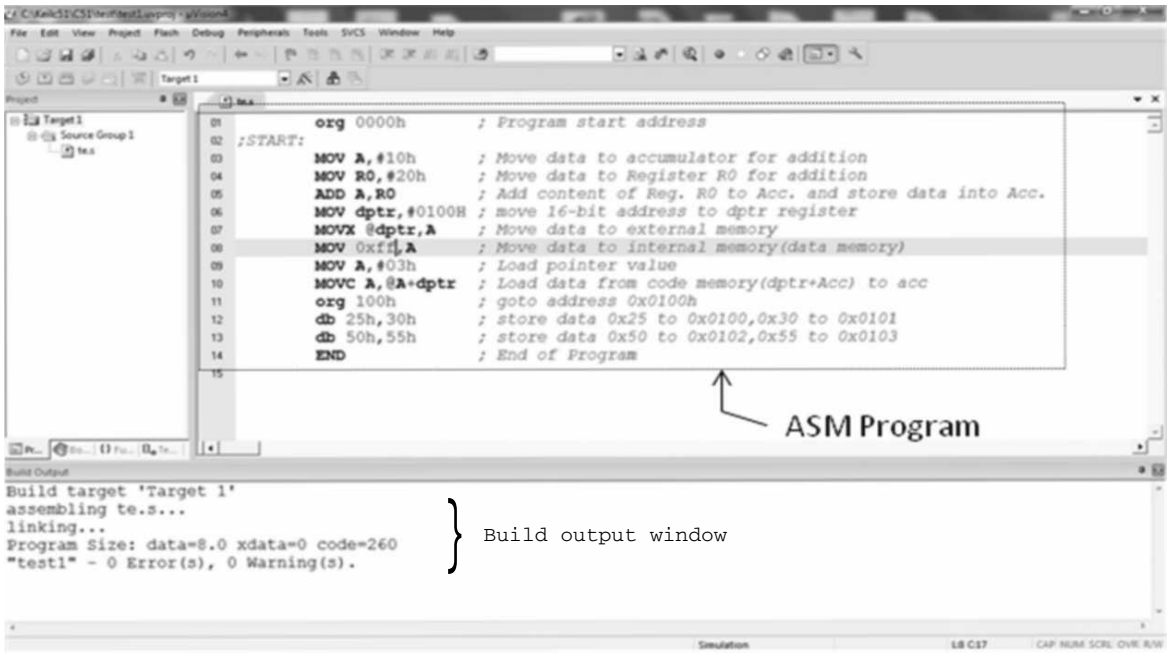


Fig. B.9 *Assembling (Building) the project*

Testing and Debugging the Program

Debugging of the program can be done in one of the two modes, either *simulator* or *target debugger*. The process is the same in both the modes except that in the *target debugger* mode, the debugger is connected with the actual hardware and allows debugging directly from the hardware. The process of debugging is explained for the simulator mode for simplicity. To debug the program proceed with the following steps.

To start the debug session, go to **Debug→Start/Stop Debug Session** (or press **ctrl+F5**) as shown in Figure B.10. For initial testing of the program, usually *single stepping* is used; therefore, the remaining process is explained with respect to the single stepping of the program.

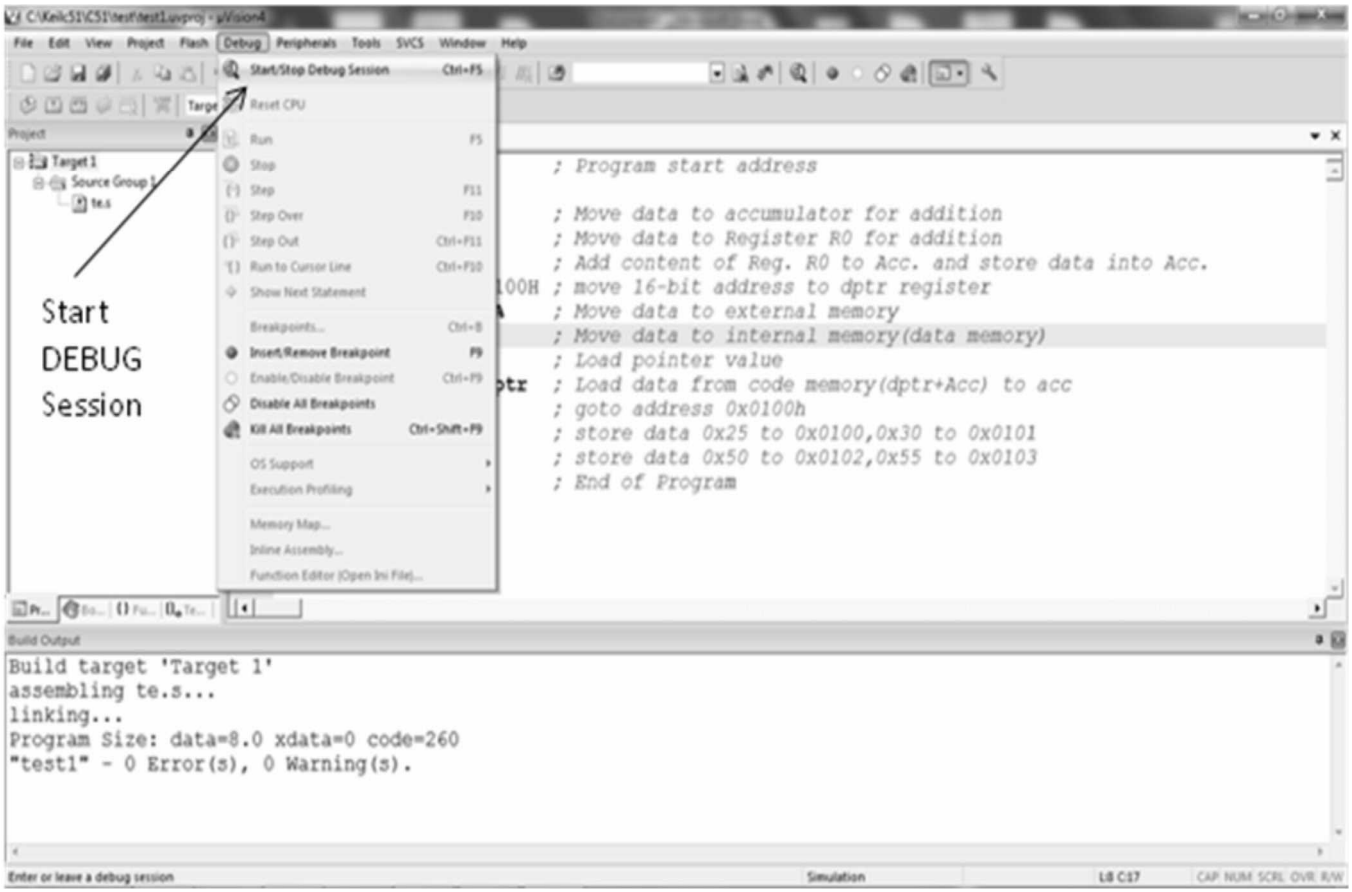


Fig. B.10 *Starting debug session*

Go to **Debug→Step (or press F11)** for single stepping,

As shown in Figure B.11, when debug mode is stared, two windows are opened, one of them is of source program window and another one is of disassembly window. Disassembly window is widely useful when the program is written in 'C language; it will give a corresponding assembly program of the 'C' program.

The arrow is pointing to the instruction to be executed (MOV A, #10H, the first instruction in our example). Now click on **STEP** (or press **F11**) for executing the instruction pointed by the arrow.

After the above step, first instruction is executed and its result is updated (A = 10H in the example) in the project window (registers) as shown in Figure B.12 and the arrow will now point to the next instruction.

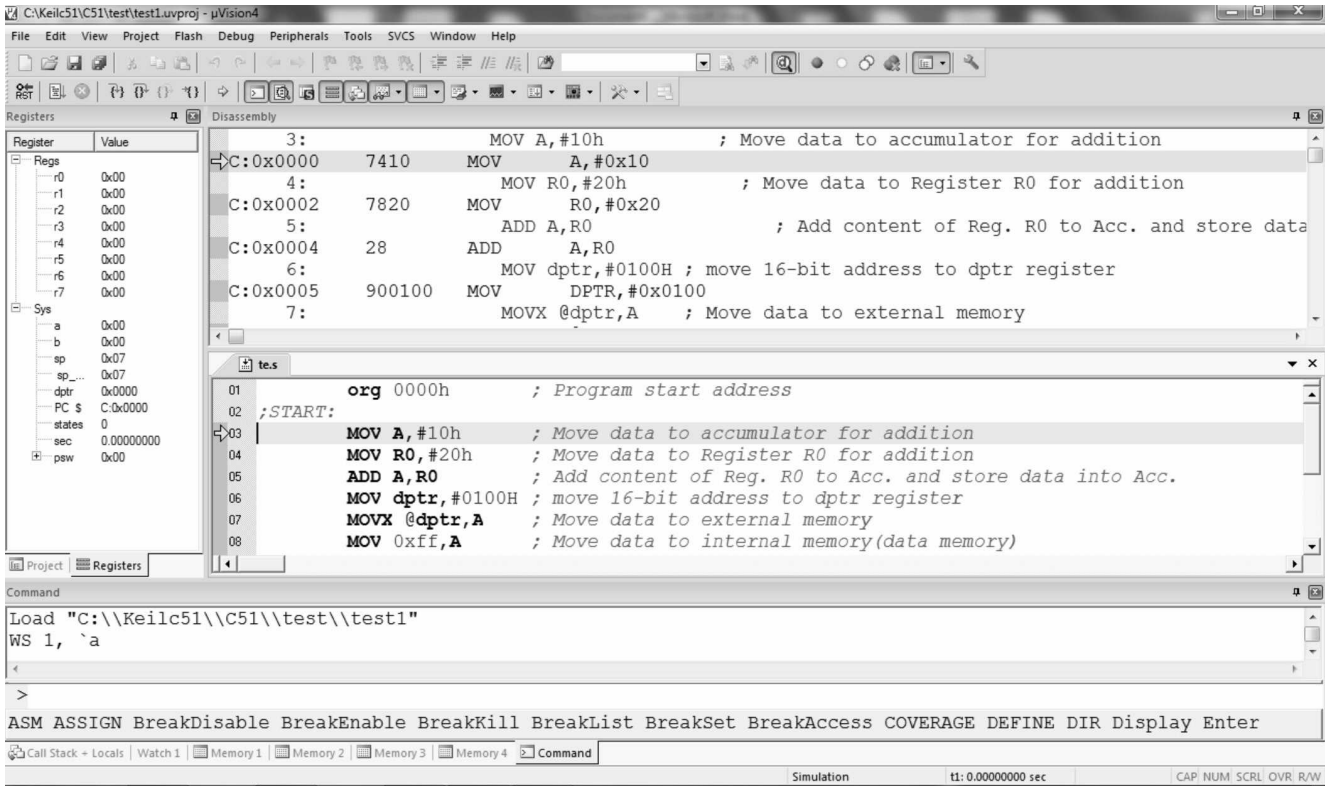


Fig. B.11 *Execution of the first instruction (MOV A, #10H)*

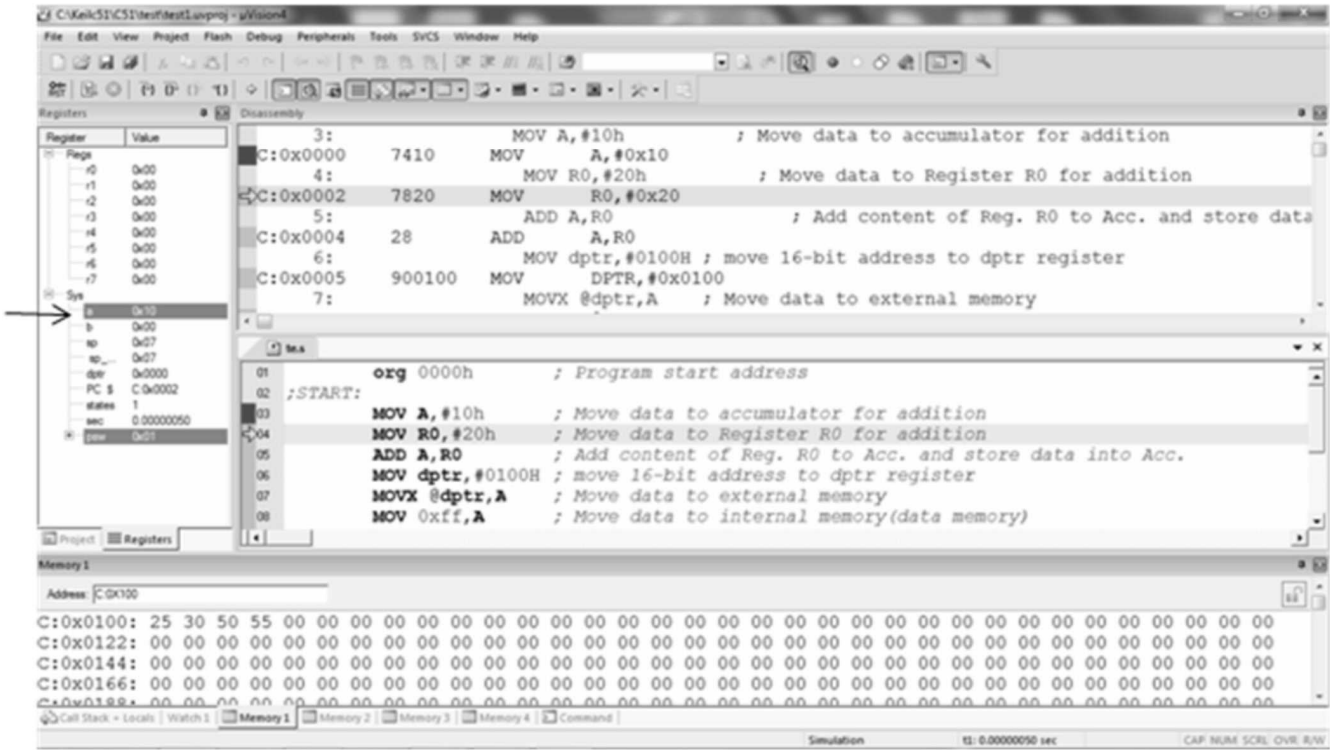


Fig. B.12 *After the execution of first instruction (MOV A, #10H)*

Following a similar process, after execution of the second instruction, 20H is stored in Register R0 as shown in Figure B.13.

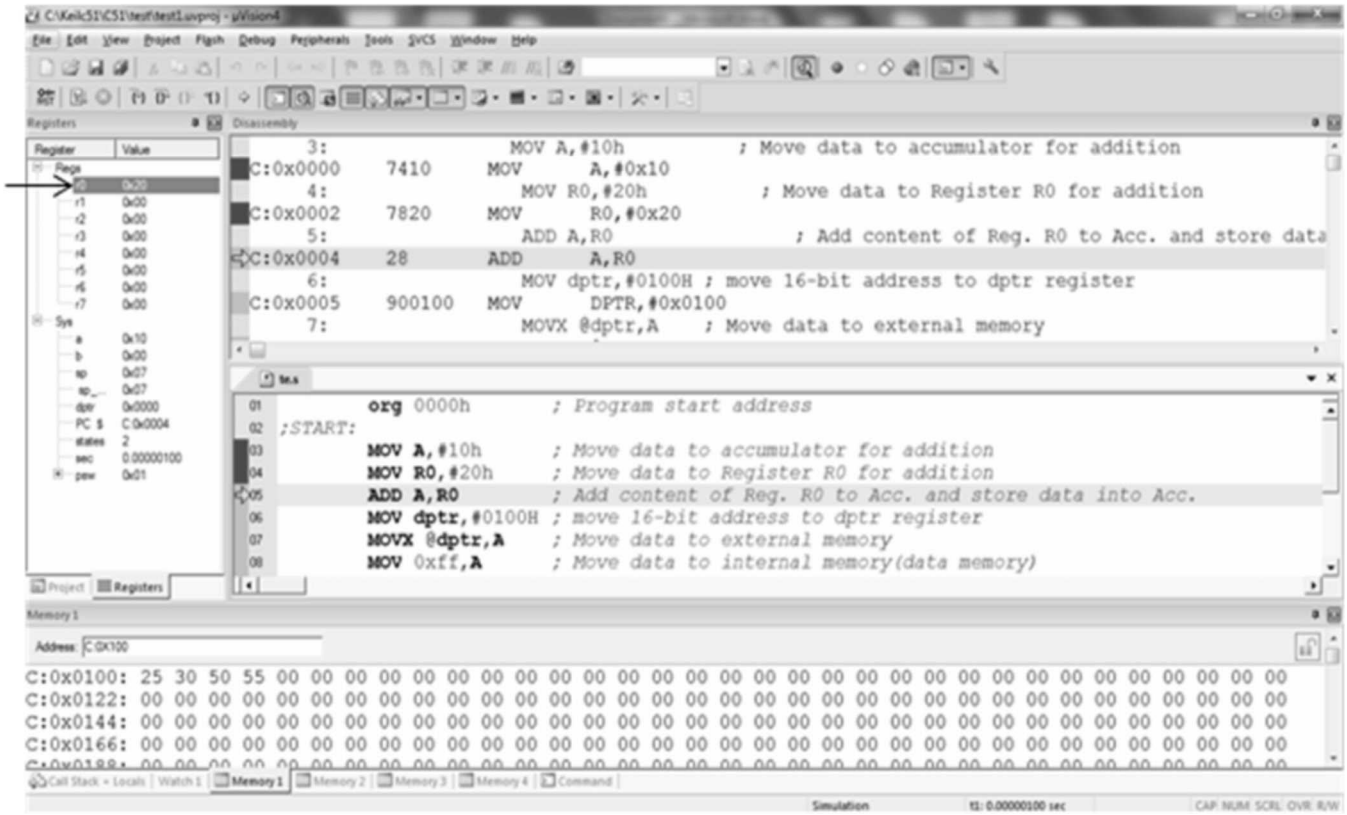


Fig. B.13 *Execution of the second instruction (MOV R0, #20h)*

To Observe Memory (data or program memory) Contents

Now for viewing the external data memory space, go to the memory window (**View→Memory Windows→memory1**) and in the Address box, type the address that you want to observe. Type **x: 0x0100** (x: address, x stands for external RAM). It will show the addresses and their contents from x: 0x0100 and so on as shown in Figure B.14. Note that the value at location 0x0100 is 00. (The address x: 0100 is chosen because the sample program will modify its contents after the execution of instruction MOVX @DPTR, A).

Similarly, to observe the contents of internal RAM type **d: address,** for example, d: 0xFF (See note below) will show the contents of the address FFH as shown in Figure B.15 and for the code memory, type the command **c: address** as shown in Figure B.16. (**Note:** though the internal RAM address 0xFF does not exist physically, the simulator will show and modify its contents as per the instruction; once again, it is reminded that it is the responsibility of a programmer to avoid such accesses!)

In this way, all the instructions can be executed, one at a time and effect of the instruction can be observed in the respective memory window. Once the single stepping is satisfactorily completed, one can go for free running (**Debug → Run** or by pressing **F5**) the program.

As explained above, any simple program can be executed by using µVision 4.0 IDE. When the simulation is satisfactorily completed, the microcontroller is programmed with the .hex version of the program. A ".hex" file is created only when **Create HEX File** is enabled in the dialog **Options for Target – Output**. For downloading the .hex file, refer **Flash** in top menu bar.

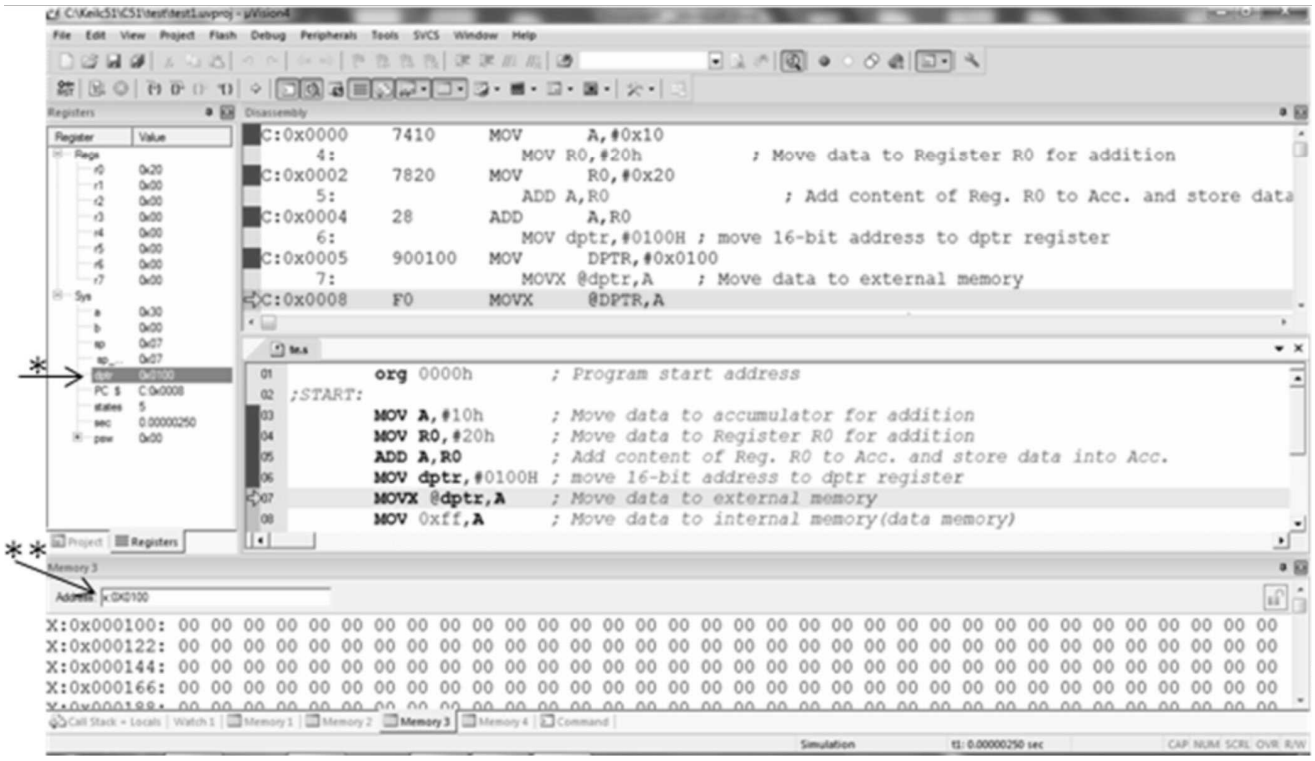


Fig. B.14 *To observe the contents of external data memory*

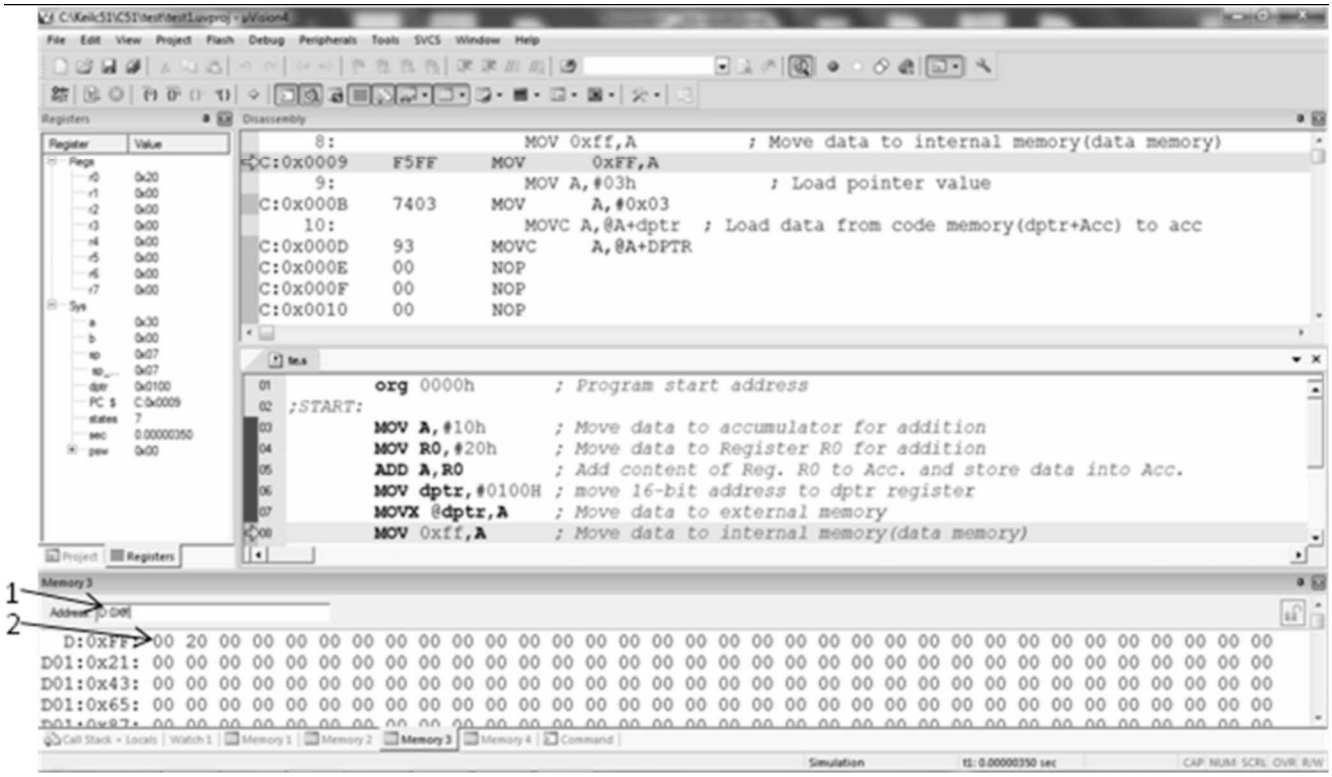


Fig. B.15 *To observe the contents of internal RAM*

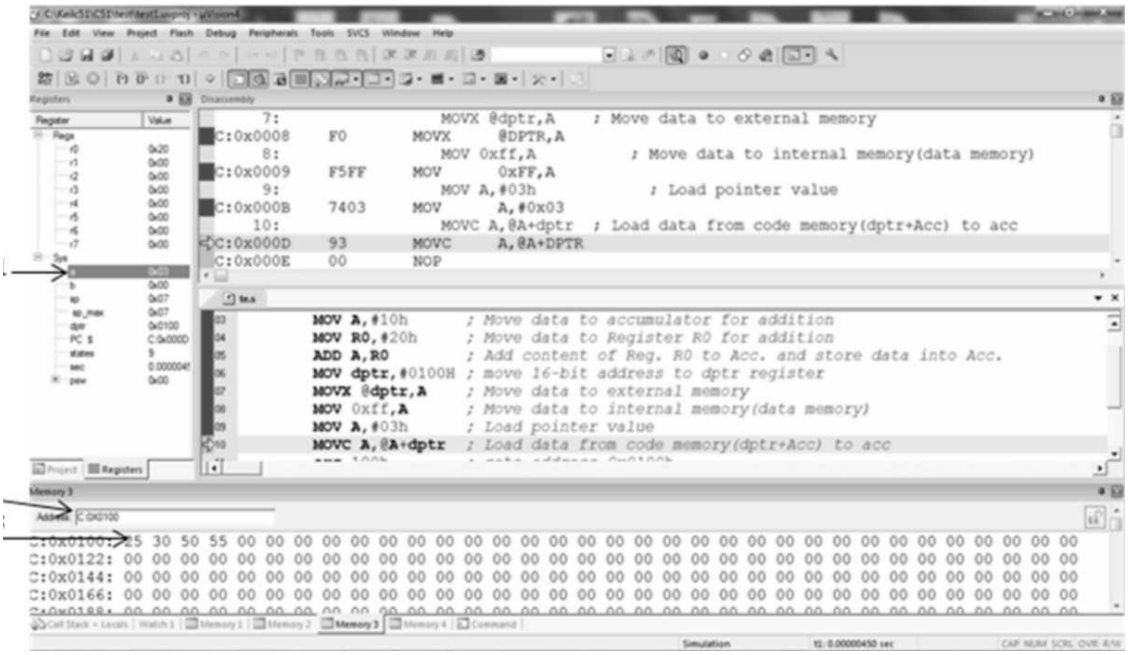


Fig. B.16 *To observe the contents of code memory*

Accessing the ports and viewing peripheral windows is explained in the next sample program written in the C language.

Developing and Testing the Program in C Language

Use of C language enables portable and efficient application programming for the embedded systems. "reg51.h" is needed to be included when the user wants to write a program in the 'C' language, which contains address declarations for registers/SFRs of the 8051 microcontroller.

Sample Program:

```
#include<reg51.h>
//sfr P0=0x80;
#define testp0 P0
sbit LED = P2^1;
bit a;
unsigned char portread;
void main()
     {      P0 = 0x00;
            P1 = 0x00;
            P2 = 0x00;
            P3 = 0xff;
            LED = 1;
            P1 = 0xf0;
            Portread = P3;
            testp0 = 0x0f;
            RS1 = 0x00;
            RS0 = 0x01;
            TMOD = 0x20;
            A = 1;
            LED =0;
     }
```

The initial steps of creating a new project, Select the device, Set target options, Creating source code file/s and adding to the project and Building the project are the same as that explained in the above section for assembly-language programs with two exceptions. First, after a device is selected, select **Yes** in the notification asking for adding 8051 startup code as shown in Figure B.5. Second, save the C program file with extension .c.

Figure B.17 shows the sample C program as a part of the project developed in µVision 4.0.

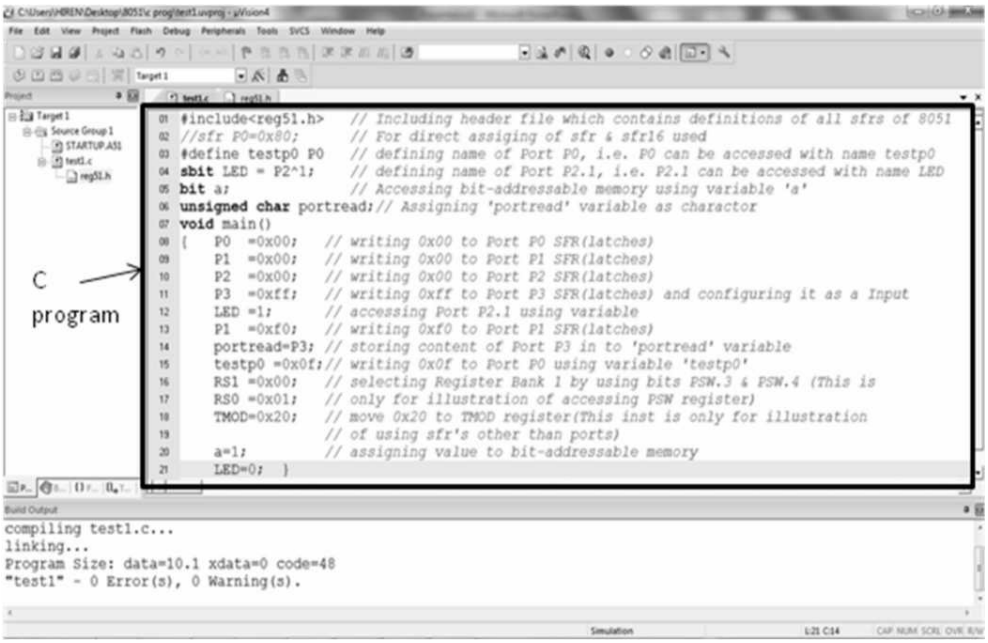


Fig. B.17 *Program written in C language*

As shown in Figure B.17, reg51.h is added in the program file which consists of the definition of addresses of all SFRs of 89C51. Figure B.18 (a) and (b) shows the contents of reg51.h. They show the address assigned to each port as well as all other SFRs and bitwise address allocation in each register (SCON in the screen shot) respectively.

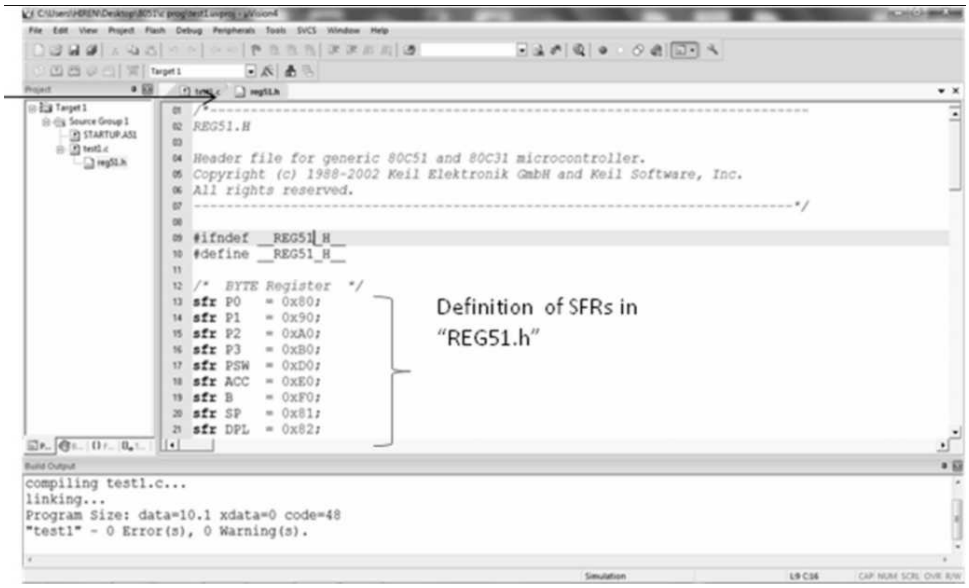


Fig. B.18(a) *Address definitions of SFR registers in reg51.h*

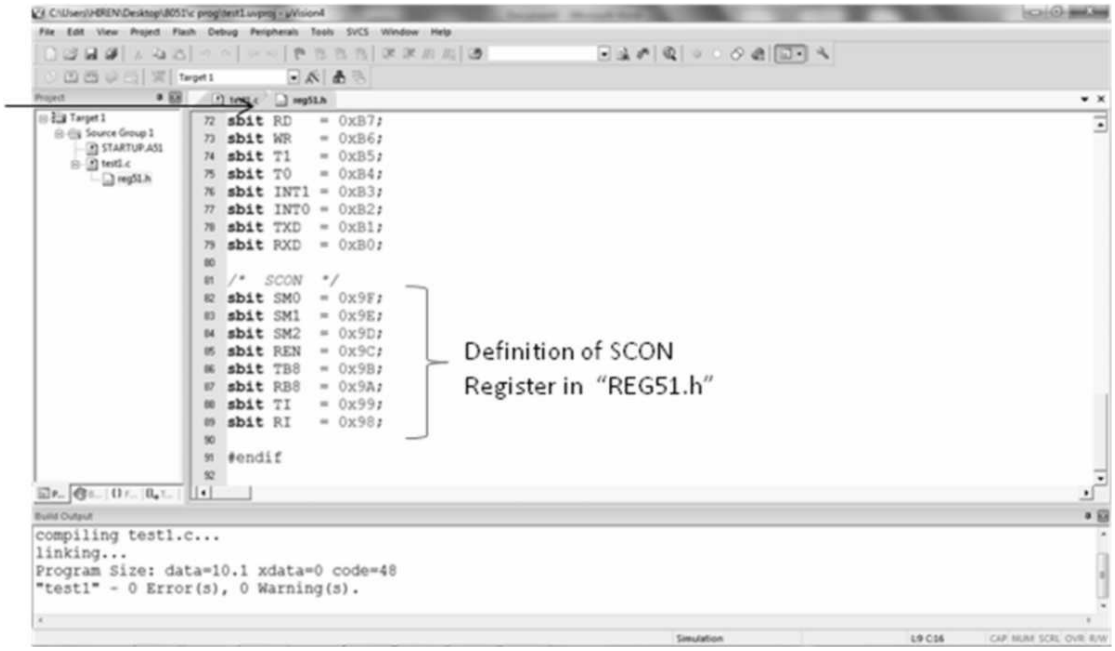


Fig. B.18(b) *Bit-wise allocation of addresses for each register (SCON shown for reference)*

Details of special data types used for 8051 and basics of C programming are given in detail in Chapter 12 (The 8051 programming in C).

Testing and Debugging the Program

Debugging is also similar to that discussed for the assembly-language program.

To start the debug session, go to **Debug→Start/Stop Debug session** (or **ctrl+F5)** as shown in Figure B.19.

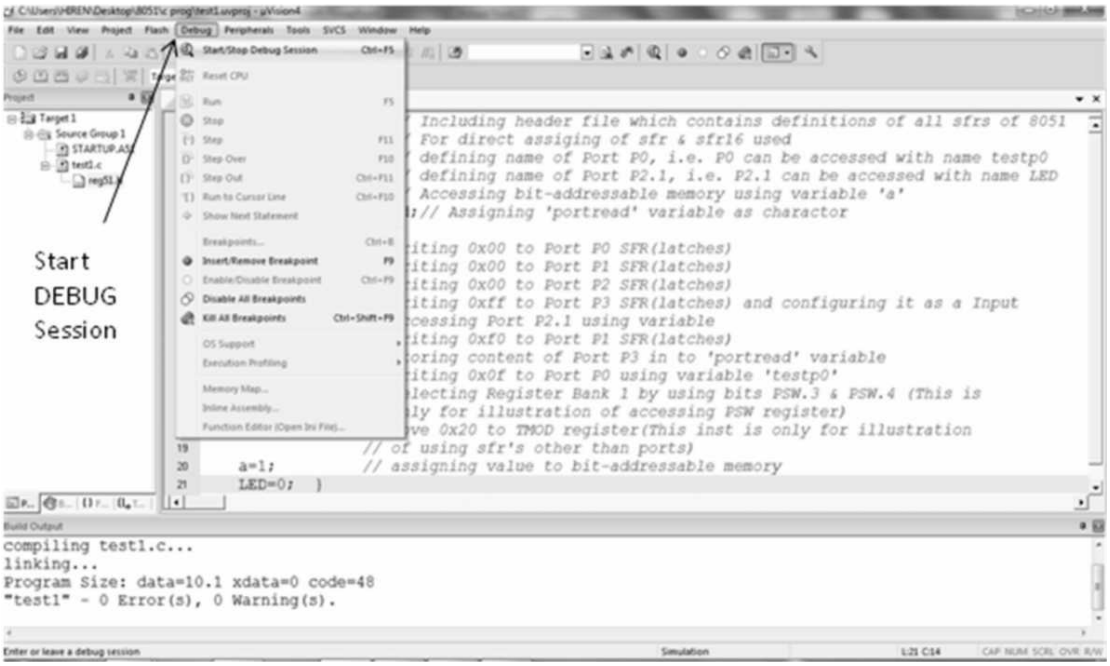


Fig. B.19 *Debug session*

Go to **Debug→Step (or press F11)** for single stepping.

In the DEBUG mode, it is possible to check the values in the all the registers, memory locations as discussed in the above section.

Moreover, SFRs of ports, timers and other peripherals are displayed in the form of peripheral windows. The peripheral window shows the peripheral settings and allows changing these settings. For selecting such peripherals, go to **Peripherals** and select the peripheral which you want to observe. Figure B.20 shows how to select the peripheral window and peripheral windows for Port 0, 1 and 2. Note that the default values (after reset) of ports are FFH.

Fig. B.20 *DEBUG mode peripherals windows view*

Single step all the statements of program in a similar manner until the end of the program. Figure B.21 shows screenshot after execution of the first statement 'P0= 0x00" and Figure B.22 shows the screenshot after execution of the last statement 'LED=0'.

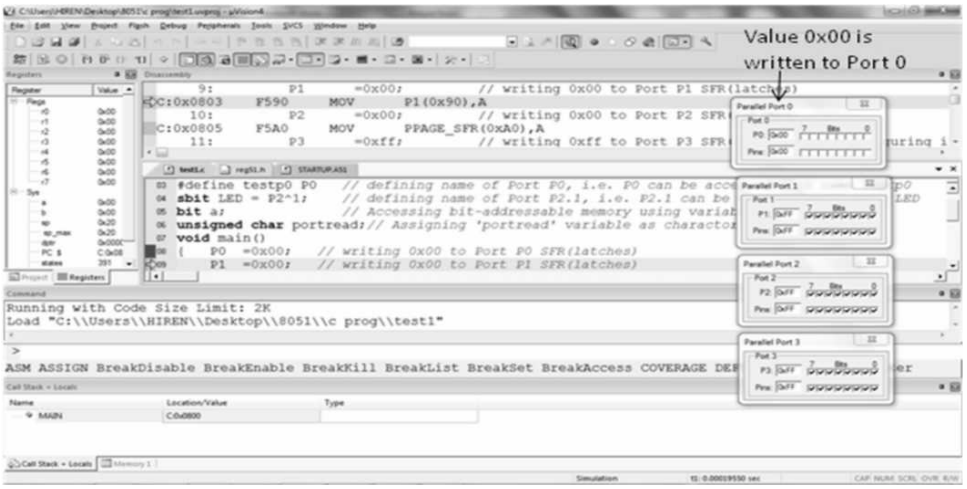


Fig. B.21 *Execution of first statement (P0=0x00)*

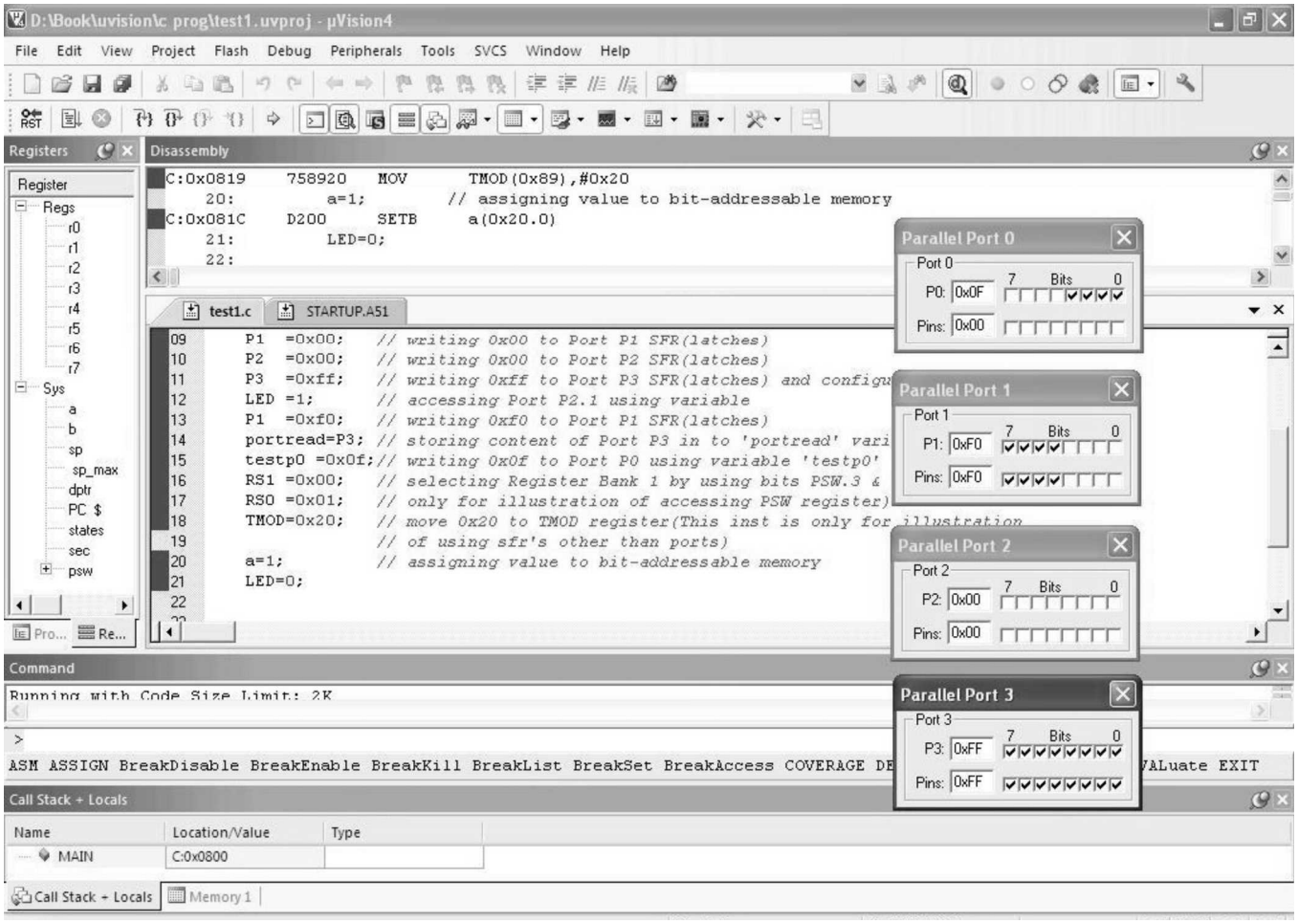


Fig. B.22 *Execution of the last statement (LED=0)*

Other Options for Debugging a Program

Till now, we have seen only the process of single stepping a program. The other advanced options to debug a program are given briefly as follows:

Run: Debug → Run (or F5) : Free run a program, or continue execution up to the breakpoint.

Stop: Debug → Start/Stop the debug session (Ctrl+ F5) : Stops program execution

Step Over: Debug → Step Over (or F10): Executes a function in a single step.

Step Out: Debug → Step Out (Ctrl+ F11): Completes the current function and then stop

Run to Cursor Line: Debug → Run to Cursor Line (Ctrl+ F10): Execute up to cursor line

Insert/Remove Break Point: Debug → Insert/Remove Break Point (F9): Insert/Remove break point, during free-running a program, execution continues up to the breakpoint.

Reset CPU: Debug → Reset CPU: Bring the CPU to the RESET state (PC=0000, and SFRs= default values)

The information on Open source softwares (Compilers and Assemblers) is available at following links:

http://sdcc.sourceforge.net

http://www.eclipse.org/downloads

Appendix C

APPENDIX C (I)

Instructions Arranged Functionally

Acronyms used in the instructions		
data	:	8 bit data
data16	:	16 bit data
direct	:	address in internal RAM or SFRs
Rn	:	R0 to R7
Ri	:	R0 or R1
( )	:	Contents of

ARITHMETIC INSTRUCTIONS

Instruction		Operation	Flags	Bytes	Machine cycles
ADD	A, Rn	A = A+Rn	C OV AC	1	1
ADD	A, direct	A = A+(direct)	C OV AC	2	1
ADD	A, @Ri	A = A+ (Ri)	C OV AC	1	1
ADD	A, #data	A = A+ data	C OV AC	2	1
ADDC	A, Rn	A = A+Rn+C	C OV AC	1	1
ADDC	A, direct	A = A+(direct)+C	C OV AC	2	1
ADDC	A, @Ri	A = A+ (Ri)+C	C OV AC	1	1
ADDC	A, #data	A = A+ data+C	C OV AC	2	1
SUBB	A, Rn	A = A– Rn-C	C OV AC	1	1
SUBB	A, direct	A = A–(direct)-C	C OV AC	2	1
SUBB	A, @Ri	A = A– (Ri)-C	C OV AC	1	1
SUBB	A, #data	A = A– data-C	C OV AC	2	1
INC	A	A = A+1		1	1
INC	Rn	Rn = Rn+1		1	1
INC	direct	(direct) = (direct)+1		2	1
INC	@Ri	(Rn) = (Rn)+1		1	1
DEC	A	A = A–1		1	1
DEC	Rn	Rn = Rn–1		1	1
DEC	direct	(direct) = (direct)–1		2	1
DEC	@Ri	(Ri) = (Ri)–1		1	1
INC	DPTR	DPTR = DPTR+1		1	2
MUL	AB	$B_{(MSB)}A_{(LSB)} = A \times B$	C=0 OV	1	4
DIV	AB	$A_{(Quotient)}B_{(Remainder)} = A/B$	C=0 OV	1	4
DA	A	$A_{BCD} = A_{binary}$	C	1	1

LOGICAL INSTRUCTIONS

Instruction		Operation	Flags	Bytes	Machine cycles
ANL	A, Rn	A = A AND Rn		1	1
ANL	A, direct	A = A AND (direct)		2	1
ANL	A, @Ri	A = A AND (Ri)		1	1
ANL	A, #data	A = A AND data		2	1
ANL	direct, A	(direct) = (direct) AND A	#	2	1
ANL	direct, #data	(direct) = (direct) AND data	#	3	2
ORL	A, Rn	A = A OR Rn		1	1
ORL	A, direct	A = A OR (direct)		2	1
ORL	A, @Ri	A = A OR (Ri)		1	1
ORL	A, #data	A = A OR data		2	1
ORL	direct, A	(direct) = (direct) OR A	#	2	1
ORL	direct, #data	(direct) = (direct) OR data	#	3	2
XRL	A, Rn	A = A XOR Rn		1	1
XRL	A, direct	A = A XOR (direct)		2	1
XRL	A, @Ri	A = A XOR (Ri)		1	1
XRL	A, #data	A = A XOR data		2	1
XRL	direct, A	(direct) = (direct) XOR A	#	2	1
XRL	direct, #data	(direct) = (direct) XOR data	#	3	2
CLR	A	A = 0		1	1
CPL	A	A = A’		1	1
RL	A	A0 = A7; A7 = A6; … A1 = A7		1	1
RLC	A	C = A7; A7 = A6; … A0 = C	C	1	1
RR	A	A7 = A0; A6 = A7; … A0 = A1		1	1
RRC	A	C = A0; A7 = C; … A0 = A1	C	1	1
SWAP	A	$A_{LN} = A_{HN}$; $A_{HN} = A_{LN}$		1	1

DATA MOVEMENT INSTRUCTIONS

Instruction		Operation	Flags	Bytes	Machine cycles
MOV	A, Rn	A = Rn		1	1
MOV	A, direct	A = (direct)		2	1
MOV	A, @Ri	A = (Ri)		1	1
MOV	A, #data	A = data		2	1
MOV	Rn, A	Rn = A		1	1
MOV	Rn, direct	Rn = (direct)		2	2
MOV	Rn, #data	Rn = data		2	1
MOV	direct, A	(direct) = A		2	1
MOV	direct, Rn	(direct) = Rn		2	2
MOV	direct1, direct2	(direct1) = (direct2)		3	2

Instruction		Operation	Flags	Bytes	Machine cycles
MOV	direct, @Ri	(direct) = (Ri)		2	2
MOV	direct, #data	(direct) = data		3	2
MOV	@Ri, A	(Ri) = A		1	1
MOV	@Ri, direct	(Ri) = (direct)		2	2
MOV	@Ri, #data	(Ri) = data		2	1
MOV	DPTR, #data16	DPTR = data16		3	2
MOVC	A, @A+DPTR	$A = (A+DPTR)_{code}$		1	2
MOVC	A, @A+PC	$A = (A+PC^{\blacktriangle})_{code}$		1	2
MOVX	A, @Ri	$A = (Ri)_{Ext\ RAM}$		1	2
MOVX	A, @DPTR	$A = (DPTR)_{Ext\ RAM}$		1	2
MOVX	@Ri, A	$(Ri)_{Ext\ RAM} = A$		1	2
MOVX	@DPTR, A	$(DPTR)_{Ext\ RAM} = A$		1	2
PUSH	direct	SP = SP+1; (SP) = (direct)		2	2
POP	direct	(direct) = (SP); SP = SP-1	#	2	2
XCH	A, Rn	A = Rn; Rn = A		1	1
XCH	A, direct	A = (direct);(direct) = A	#	2	1
XCH	A, @Ri	A = (Ri);(Ri) = A		1	1
XCHD	A, @Ri	$A_{LN} = (Ri)_{LN}$;$(Ri)_{LN} = A_{LN}$		1	1

BIT-PROCESSING INSTRUCTIONS

Instruction		Operation	Flags	Bytes	Machine cycles
CLR	C	C = 0	C = 0	1	1
CLR	bit	bit = 0	*	2	1
SETB	C	C = 1	C = 1	1	1
SETB	bit	bit = 1	*	2	1
CPL	C	C = C'	C	1	1
CPL	bit	Bit = bit'	*	2	1
ANL	C, bit	C = C AND bit	C	2	2
ANL	C, /bit	C = C AND /bit	C	2	2
ORL	C, bit	C = C OR bit	C	2	2
ORL	C, /bit	C = C OR /bit	C	2	2
MOV	C, bit	C = bit	C	2	1
MOV	bit, C	bit = C	*	2	2
JC	rel	if C = 1, $PC = PC^{\blacktriangle}+rel$		2	2
JNC	rel	if C = 0, $PC = PC^{\blacktriangle}+rel$		2	2
JB	bit, rel	if bit = 1, $PC = PC^{\blacktriangle}+rel$		3	2
JNB	bit, rel	if bit = 0, $PC = PC^{\blacktriangle}+rel$		3	2
JBC	bit, rel	if bit = 1, $PC = PC^{\blacktriangle}+rel$; bit=0	*	3	2

$^{\blacktriangle}$PC is having address of the next instruction, (i.e.) PC is pointing to the next instruction.

* Flags are affected directly if the bit address of the flag is specified.

Flags are affected if the address of PSW is specified.

PROGRAM FLOW CONTROL INSTRUCTIONS (CALL AND JUMPS)

Instruction		Operation	Flags	Bytes	Machine cycles
ACALL	addr11	SP = SP+1;(SP) = PCL▲; SP = SP+1; (SP) = PCH▲; PC_{10-0} = addr 11		2	2
LCALL	addr16	SP = SP+1;(SP) = PCL▲; SP = SP+1; (SP) = PCH▲; PC = addr 16		3	2
RET		PCH = (SP); SP = SP-1; PCL = (SP);SP = SP-1		1	2
RETI		PCH = (SP); SP = SP-1; PCL= (SP);SP = SP-1		1	2
AJMP	addr11	PC_{10-0} = addr11		2	2
LJMP	addr16	PC = addr16		3	2
SJMP	rel	PC = PC▲+rel		2	2
JMP	@A+DPTR	PC = A+DPTR		1	2
JZ	rel	if A = 0, PC = PC▲+rel		2	2
JNZ	rel	if A ≠ 0, PC = PC▲+rel		2	2
CJNE	A,direct, rel	If A ≠ (direct), PC = PC▲+rel	C	3	2
CJNE	A,#data, rel	If A ≠ data, PC = PC▲+rel	C	3	2
CJNE	Rn,#data, rel	If Rn ≠ data, PC = PC▲+rel	C	3	2
CJNE	@Ri, #data, rel	If (Ri) ≠ data, PC = PC▲+rel	C	3	2
DJNZ	Rn, rel	Rn = Rn-1; If Rn ≠ 0, PC▲ = PC+rel		2	2
DJNZ	direct, rel	(direct) = (direct)-1; if (direct)≠0, PC = PC▲+rel		3	2
NOP		No operation		1	1

▲PC is having the address of the next instruction, i.e. PC is pointing to the next instruction.

APPENDIX C (II) INSTRUCTIONS ARRANGED ALPHABETICALLY

Instruction		Operation	Flags	Bytes	Machine cycles
ACALL	addr11	SP = SP+1;(SP) = PCL▲; SP = SP+1; (SP) = PCH▲; PC_{10-0} = addr 11		2	2
ADD	A, Rn	A = A+Rn	C OV AC	1	1
ADD	A, direct	A = A+(direct)	C OV AC	2	1
ADD	A, @Ri	A = A+ (Ri)	C OV AC	1	1
ADD	A, #data	A = A+ data	C OV AC	2	1
ADDC	A, Rn	A = A+Rn+C	C OV AC	1	1
ADDC	A, direct	A = A+(direct)+C	C OV AC	2	1
ADDC	A, @Ri	A = A+ (Ri)+C	C OV AC	1	1
ADDC	A, #data	A = A+ data+C	C OV AC	2	1
AJMP	addr11	PC_{10-0} = addr11		2	2
ANL	A, Rn	A = A AND Rn		1	1
ANL	A, direct	A = A AND (direct)		2	1

Instruction		Operation	Flags	Bytes	Machine cycles
ANL	A, @Ri	A = A AND (Ri)		1	1
ANL	A, #data	A = A AND data		2	1
ANL	direct, A	(direct) = (direct) AND A		2	1
ANL	direct, #data	(direct) = (direct) AND data		3	2
ANL	C, bit	C = C AND bit	C	2	2
ANL	C, /bit	C = C AND /bit	C	2	2
CJNE	A, direct, rel	If A ≠ (direct), PC = PC▲+rel	C	3	2
CJNE	A, #data, rel	If A ≠ data, PC = PC▲+rel	C	3	2
CJNE	Rn, #data, rel	If Rn ≠ data, PC = PC▲+rel	C	3	2
CJNE	@Ri, #data, rel	If (Ri) ≠ data, PC = PC▲+rel	C	3	2
CLR	A	A = 0		1	1
CLR	C	C = 0	C=0	1	1
CLR	bit	bit = 0		2	1
CPL	A	A = A’		1	1
CPL	C	C = C’	C	1	1
CPL	bit	bit = bit’		2	1
DA	A	$A_{BCD} = A_{binary}$	C	1	1
DEC	A	A = A-1		1	1
DEC	Rn	Rn = Rn-1		1	1
DEC	direct	(direct) = (direct)-1		2	1
DEC	@Ri	(Ri) = (Ri)-1		1	1
DIV	AB	$A_{(Quotient)}B_{(Remainder)} = A/B$	C=0 OV	1	4
DJNZ	Rn, rel	Rn = Rn-1; If Rn ≠ 0, PC = PC▲+rel		2	2
DJNZ	direct, rel	(direct) = (direct)-1; if (direct) ≠ 0, PC = PC▲+rel		3	2
INC	A	A = A+1		1	1
INC	Rn	Rn = Rn+1		1	1
INC	direct	(direct) = (direct)+1		2	1
INC	@Ri	(Rn) = (Rn)+1		1	1
INC	DPTR	DPTR = DPTR+1		1	2
JB	bit, rel	if bit = 1, PC = PC▲+rel		3	2
JBC	bit, rel	if bit = 1, PC = PC▲+rel; bit = 0		3	2
JC	rel	if C = 1, PC = PC▲+rel		2	2
JMP	@A+DPTR	PC = (A+DPTR)		1	2
JNB	bit, rel	if bit = 0, PC = PC▲+rel		3	2
JNC	rel	if C = 0, PC = PC▲+rel		2	2
JNZ	rel	if A ≠ 0, PC = PC▲+rel		2	2
JZ	rel	if A = 0, PC = PC▲+rel		2	2
LCALL	addr16	SP = SP+1 → (SP) = PCL▲ → SP = SP+1 → (SP) = PCH → PC = addr16		3	2
LJMP	addr16	PC = addr16		3	2
MOV	A, Rn	A = Rn		1	1
MOV	A, direct	A = (direct)		2	1
MOV	A, @Ri	A = (Ri)		1	1
MOV	A, #data	A = data		2	1
MOV	Rn, A	Rn = A		1	1

Instruction		Operation	Flags	Bytes	Machine cycles
MOV	Rn, direct	Rn = (direct)		2	2
MOV	Rn, #data	Rn = data		2	1
MOV	direct, A	(direct) = A		2	1
MOV	direct, Rn	(direct) = Rn		2	2
MOV	direct1, direct2	(direct1) = (direct2)		3	2
MOV	direct, @Ri	(direct) = (Ri)		2	2
MOV	direct, #data	(direct) = data		3	2
MOV	@Ri, A	(Ri) = A		1	1
MOV	@Ri, direct	(Ri) = (direct)		2	2
MOV	@Ri, #data	(Ri) = data		2	1
MOV	DPTR, #data16	DPTR = data16		3	2
MOV	C, bit	C = bit	C	2	1
MOV	bit, C	bit = C		2	2
MOVC	A,@ A+DPTR	$A = (A+DPTR)_{code}$		1	2
MOVC	A, @A+PC	$A = (A+PC^{\blacktriangle})_{code}$		1	2
MOVX	A, @Ri	$A = (Ri)_{Ext\ RAM}$		1	2
MOVX	A, @DPTR	$A = (DPTR)_{Ext\ RAM}$		1	2
MOVX	@Ri, A	$(Ri)_{Ext\ RAM} = A$		1	2
MOVX	@DPTR, A	$(DPTR)_{Ext\ RAM} = A$		1	2
MUL	AB	$B_{(MSB)}A_{(LSB)} = A \times B$	C=0 OV	1	4
NOP		No operation		1	1
ORL	A, Rn	A = A OR Rn		1	1
ORL	A, direct	A = A OR (direct)		2	1
ORL	A, @Ri	A = A OR (Ri)		1	1
ORL	A, #data	A = A OR data		2	1
ORL	direct, A	(direct) = (direct) OR A		2	1
ORL	direct, #data	(direct) = (direct) OR data		3	2
ORL	C, bit	C = C OR bit	C	2	2
ORL	C, /bit	C = C OR /bit	C	2	2
POP	direct	(direct) = (SP); SP = SP-1		2	2
PUSH	direct	SP = SP+1 → (SP) = (direct)		2	2
RET		PCH = (SP)→SP = SP-1→PCL = (SP)→SP = SP-1		1	2
RETI		PCH = (SP)→SP = SP-1→PCL = (SP)→SP = SP-1		1	2
RL	A	A0 = A7; A7 = A6; … A1 = A7		1	1
RLC	A	C = A7; A7 = A6; … A0 = C	C	1	1
RR	A	A7 = A0; A6 = A7; … A0 = A1		1	1
RRC	A	C = A0; A7 = C; … A0 = A1	C	1	1
SETB	C	C = 1	C=1	1	1
SETB	bit	bit = 1		2	1
SJMP	rel	$PC = PC^{\blacktriangle}+rel$		2	2
SUBB	A, Rn	A = A- Rn-C	C OV AC	1	1
SUBB	A, direct	A = A-(direct)-C	C OV AC	2	1

Instruction		Operation	Flags	Bytes	Machine cycles
SUBB	A, @Ri	A = A- (Ri)-C	C OV AC	1	1
SUBB	A, #data	A = A- data-C	C OV AC	2	1
SWAP	A	$A_{LN} \leftrightarrow A_{HN}$		1	1
XCH	A, Rn	A↔ Rn		1	1
XCH	A, direct	A↔ (direct)		2	1
XCH	A, @Ri	A↔(Ri)		1	1
XCHD	A, @Ri	$A_{LN} \leftrightarrow (RI)_{LN}$		1	1
XRL	A, Rn	A = A XOR Rn		1	1
XRL	A, direct	A = A XOR (direct)		2	1
XRL	A, @Ri	A = A XOR (Ri)		1	1
XRL	A, #data	A = A XOR data		2	1
XRL	direct, A	(direct) = (direct) XOR A		2	1
XRL	direct, #data	(direct) = (direct) XOR data		3	2

▲PC is having address of the next instruction, i.e. PC is pointing to the next instruction.

APPENDIX C (III) HEXADECIMAL CODES OF 8051 INSTRUCTIONS

Hex Code	Instruction		Bytes
00	NOP		1
01	AJMP	rel	2
02	LJMP	rel	3
03	RR	A	1
04	INC	A	1
05	INC	direct	2
06	INC	@R0	1
07	INC	@R1	1
08	INC	R0	1
09	INC	R1	1
0A	INC	R2	1
0B	INC	R3	1
0C	INC	R4	1
0D	INC	R5	1
0E	INC	R6	1
0F	INC	R7	1
10	JBC	bit, rel	3
11	ACALL	rel	2
12	LCALL	rel	3
13	RRC	A	1
14	DEC	A	1
15	DEC	direct	2
16	DEC	@R0	1
17	DEC	@R1	1
18	DEC	R0	1
19	DEC	R1	1
1A	DEC	R2	1
1B	DEC	R3	1
1C	DEC	R4	1
1D	DEC	R5	1
1E	DEC	R6	1

Hex Code	Instruction		Bytes
1F	DEC	R7	1
20	JB	bit, rel	3
21	AJMP	rel	2
22	RET		1
23	RL	A	1
24	ADD	A, #data	2
25	ADD	A, direct	2
26	ADD	A, @R0	1
27	ADD	A, @R1	1
28	ADD	A, R0	1
29	ADD	A, R1	1
2A	ADD	A, R2	1
2B	ADD	A, R3	1
2C	ADD	A, R4	1
2D	ADD	A, R5	1
2E	ADD	A, R6	1
2F	ADD	A, R7	1
30	JNB	bit, rel	3
31	ACALL	rel	2
32	RETI		1
33	RLC	A	1
34	ADDC	A, #data	2
35	ADDC	A, direct	2
36	ADDC	A, @R0	1
37	ADDC	A, @R1	1
38	ADDC	A, R0	1
39	ADDC	A, R1	1
3A	ADDC	A, R2	1
3B	ADDC	A, R3	1
3C	ADDC	A, R4	1
3D	ADDC	A, R5	1

Hex Code	Instruction		Bytes
3E	ADDC	A, R6	1
3F	ADDC	A, R7	1
40	JC	rel	2
41	AJMP	rel	2
42	ORL	direct, A	2
43	ORL	direct, #data	3
44	ORL	A, #data	2
45	ORL	A, direct	2
46	ORL	A, @R0	1
47	ORL	A, @R1	1
48	ORL	A, R0	1
49	ORL	A, R1	1
4A	ORL	A, R2	1
4B	ORL	A, R3	1
4C	ORL	A, R4	1
4D	ORL	A, R5	1
4E	ORL	A, R6	1
4F	ORL	A, R7	1
50	JNC	rel	2
51	ACALL	rel	2
52	ANL	direct, A	2
53	ANL	direct, #data	3
54	ANL	A, #data	2
55	ANL	A, direct	2
56	ANL	A, @R0	1
57	ANL	A, @R1	1
58	ANL	A, R0	1
59	ANL	A, R1	1
5A	ANL	A, R2	1
5B	ANL	A, R3	1
5C	ANL	A, R4	1
5D	ANL	A, R5	1
5E	ANL	A, R6	1
5F	ANL	A, R7	1
60	JZ	rel	2
61	AJMP	rel	2
62	XRL	direct, A	2
63	XRL	direct, #data	3
64	XRL	A, #data	2
65	XRL	A, direct	2
66	XRL	A, @R0	1
67	XRL	A, @R1	1
68	XRL	A, R0	1
69	XRL	A, R1	1
6A	XRL	A, R2	1
6B	XRL	A, R3	1
6C	XRL	A, R4	1
6D	XRL	A, R5	1
6E	XRL	A, R6	1
6F	XRL	A, R7	1

Hex Code	Instruction		Bytes
70	JNZ	rel	2
71	ACALL	rel	2
72	ORL	C, bit	2
73	JMP	@A+DPTR	1
74	MOV	A, #data	2
75	MOV	direct, #data	3
76	MOV	@R0, #data	2
77	MOV	@R1, #data	2
78	MOV	R0, #data	2
79	MOV	R1, #data	2
7A	MOV	R2, #data	2
7B	MOV	R3, #data	2
7C	MOV	R4, #data	2
7D	MOV	R5, #data	2
7E	MOV	R6, #data	2
7F	MOV	R7, #data	2
80	SJMP	rel	2
81	AJMP	rel	2
82	ANL	C, bit	2
83	MOVC	A, @A+PC	1
84	DIV	AB	1
85	MOV	direct, direct	3
86	MOV	direct, @R0	2
87	MOV	direct, @R1	2
88	MOV	direct, R0	2
89	MOV	direct, R1	2
8A	MOV	direct, R2	2
8B	MOV	direct, R3	2
8C	MOV	direct, R4	2
8D	MOV	direct, R5	2
8E	MOV	direct, R6	2
8F	MOV	direct, R7	2
90	MOV	DPTR, #data	3
91	ACALL	rel	2
92	MOV	bit, C	2
93	MOVC	A, @A+DPTR	1
94	SUBB	A, #data	2
95	SUBB	A, direct	2
96	SUBB	A, @R0	1
97	SUBB	A, @R1	1
98	SUBB	A, R0	1
99	SUBB	A, R1	1
9A	SUBB	A, R2	1
9B	SUBB	A, R3	1
9C	SUBB	A, R4	1
9D	SUBB	A, R5	1
9E	SUBB	A, R6	1
9F	SUBB	A, R7	1
A0	ORL	C, /bit	2
A1	AJMP	rel	2

Hex Code	Instruction		Bytes
A2	MOV	C, bit	2
A3	INC	DPTR	1
A4	MUL	AB	1
A5	reserved		
A6	MOV	@R0, direct	2
A7	MOV	@R1, direct	2
A8	MOV	R0, direct	2
A9	MOV	R1, direct	2
AA	MOV	R2, direct	2
AB	MOV	R3, direct	2
AC	MOV	R4, direct	2
AD	MOV	R5, direct	2
AE	MOV	R6, direct	2
AF	MOV	R7, direct	2
B0	ANL	C, /bit	2
B1	ACALL	rel	2
B2	CPL	bit	2
B3	CPL	C	1
B4	CJNE	A, #data, rel	3
B5	CJNE	A, direct, rel	3
B6	CJNE	@R0, #data, rel	3
B7	CJNE	@R1, #data, rel	3
B8	CJNE	R0, #data, rel	3
B9	CJNE	R1, #data, rel	3
BA	CJNE	R2, #data, rel	3
BB	CJNE	R3, #data, rel	3
BC	CJNE	R4, #data, rel	3
BD	CJNE	R5, #data, rel	3
BE	CJNE	R6, #data, rel	3
BF	CJNE	R7, #data, rel	3
C0	PUSH	direct	2
C1	AJMP	rel	2
C2	CLR	bit	2
C3	CLR	C	1
C4	SWAP	A	1
C5	XCH	A, direct	2
C6	XCH	A, @R0	1
C7	XCH	A, @R1	1
C8	XCH	A, R0	1
C9	XCH	A, R1	1
CA	XCH	A, R2	1
CB	XCH	A, R3	1
CC	XCH	A, R4	1
CD	XCH	A, R5	1
CE	XCH	A, R6	1
CF	XCH	A, R7	1
D0	POP	direct	2

Hex Code	Instruction		Bytes
D1	ACALL	rel	2
D2	SETB	bit	2
D3	SETB	C	1
D4	DA	A	1
D5	DJNZ	direct, rel	3
D6	XCHD	A, @R0	1
D7	XCHD	A, @R1	1
D8	DJNZ	R0, rel	2
D9	DJNZ	R1, rel	2
DA	DJNZ	R2, rel	2
DB	DJNZ	R3, rel	2
DC	DJNZ	R4, rel	2
DD	DJNZ	R5, rel	2
DE	DJNZ	R6, rel	2
DF	DJNZ	R7, rel	2
E0	MOVX	A, @DPTR	1
E1	AJMP	rel	2
E2	MOVX	A, @R0	1
E3	MOVX	A, @R1	1
E4	CLR	A	1
E5	MOV	A, direct	2
E6	MOV	A, @R0	1
E7	MOV	A, @R1	1
E8	MOV	A, R0	1
E9	MOV	A, R1	1
EA	MOV	A, R2	1
EB	MOV	A, R3	1
EC	MOV	A, R4	1
ED	MOV	A, R5	1
EE	MOV	A, R6	1
EF	MOV	A, R7	1
F0	MOVX	@DPTR, A	1
F1	ACALL	rel	2
F2	MOVX	@R0, A	1
F3	MOVX	@R1, A	1
F4	CPL	A	1
F5	MOV	direct, A	2
F6	MOV	@R0, A	1
F7	MOV	@R1, A	1
F8	MOV	R0, A	1
F9	MOV	R1, A	1
FA	MOV	R2, A	1
FB	MOV	R3, A	1
FC	MOV	R4, A	1
FD	MOV	R5, A	1
FE	MOV	R6, A	1
FF	MOV	R7, A	1

Appendix D

ASCII Codes

ASCII Control Characters (Character code 0-31)		
DEC	HEX	Code
00	00	NUL
01	01	SOH
02	02	STX
03	03	ETX
04	04	EOT
05	05	ENQ
06	06	ACK
07	07	BEL
08	08	BS
09	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	SI
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

ASCII Printable Characters (Character code 32-127)								
DEC	HEX	Symbol	DEC	HEX	Symbol	DEC	HEX	Symbol
32	20	Space	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	DEL

The PC Extended ASCII Codes (Character code 128-255)												
DEC	**HEX**	**Symbol**	**DEC**	**HEX**	**Symbol**	**DEC**	**HEX**	**Symbol**	**DEC**	**HEX**	**Symbol**
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E		190	BE	╛	222	DE	▐	254	FE	■
159	9F	*f*	191	BF	┐	223	DF	▀	255	FF	

Appendix E

Special Function Registers Quick View

SFRs Byte and Bit Addresses

SFR	Byte Address	Bit addresses								Function
		D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
P0	80H	87	86	85	84	83	82	81	80	Port 0
SP	81H									Stack Pointer
DPL	82H									Data Pointer Lower byte
DPH	83H									Data Pointer Higher byte
PCON	87H									Power Control register
TCON	88H	8F	8E	8D	8C	8B	8A	89	88	Timer Control register
TMOD	89H									Timer Mode register
TL0	8AH									Timer 0 Lower byte
TL1	8BH									Timer 1 Lower byte
TH0	8CH									Timer 0 Higher byte
TH1	8DH									Timer 1 Higher byte
P1	90H	97	96	95	94	93	92	91	90	Port 1
SCON	98H	9F	9E	9D	9C	9B	9A	99	98	Serial port Control register
SBUF	99H									Serial port data Buffer
P2	A0H	A7	A6	A5	A4	A3	A2	A1	A0	Port 2
IE	A8H	AF	--	--	AC	AB	AA	A9	A8	Interrupt Enable register
P3	B0H	B7	B6	B5	B4	B3	B2	B1	B0	Port 3
IP	B8H	--	--	--	BC	BB	BA	B9	B8	Interrupt Priority register
PSW	D0H	D7	D6	D5	D4	D3	D2	D1	D0	Program Status Word
ACC	E0H	E7	F6	E5	E4	E3	E2	E1	E0	Accumulator
B	F0H	F7	F6	F5	F4	F3	F2	F1	F0	B register

PSW: Program Status Word (Address: D0H)

Bit address	D7	D6	D5	D4	D3	D2	D1	D0
Bit name	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
Symbol	CY	AC	F0	RS1	RS0	OV	--	P
	MSB							LSB

Bit	Symbol	Flag name and description		
7	C (or CY)	Carry; Used in arithmetic, logic and Boolean operations		
6	AC	Auxiliary carry ; useful only for BCD arithmetic		
5	F0	Flag 0; general-purpose user flag		
4	RS1	Register Bank Select bit 1		
3	RS0	Register Bank Select bit 0		
		RS1	RS0	
		0	0	Bank 0
		0	1	Bank 1
		1	0	Bank 2
		1	1	Bank 3
2	0V	Overflow; used in arithmetic operations		
1	--	Reserved; may be used as a general-purpose flag		
0	P	Parity; set to 1 if A has odd number of ones, otherwise reset to 0		

TMOD: Timer Mode Control Register (Address: 89H)

Timer 1				Timer 0			
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
MSB							LSB

Bit	Symbol	Description		
7/3	Gate	Start (or stop) Control using hardware or software. When Gate = 0, start (or stop) of the timer is controlled only by TR1/TR0 bits, while Gate = 1, it is controlled by TR1/TR0 as well as signal on $\overline{INT1}/\overline{INT0}$ pin		
6/2	C/$\overline{T}$	C/$\overline{T}$ = 0 configures the timer as an interval timer (or time delay generator), C/$\overline{T}$ = 1 will configure the timer as event counter		
5/1	M1	Mode select bit 1		
4/0	M0	Mode select bit 0		
		M1	M0	
		0	0	Mode 0;13-bit timer
		0	1	Mode 1; 16-bit timer/counter
		1	0	Mode 2; 8-bit auto reload
		1	1	Mode 3; split timer mode, TL0 as 8-bit timer/counter and TH0 as 8-bit timer controlled by control bits of Timer 0 and Timer 1 respectively. Timer 1 operation timer/counter stopped.

TCON: Timer Control Register (Address: 88H)

Bit address	8F	8E	8D	8C	8B	8A	89	88
Bit name	TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
Symbol	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
	MSB							LSB

Bit	Symbol	Description
7	TF1	Timer 1 overflow flag; Set by the hardware when Timer/counter 1 overflows; Cleared by the hardware when the controller vectors to interrupt service routine at the address 001BH
6	TR1	Timer 1 run control bit; set to 1 by a program to start Timer/counter 1; Cleared to 0 to stop Timer/Counter 1
5	TF0	Timer 0 overflow flag; Set by the hardware when the Timer/counter 0 overflows; Cleared by the hardware when the controller vectors to interrupt service routine at the address 000BH
4	TR0	Timer 0 run control bit; Set to 1 by a program to start Timer/counter 0; Cleared to 0 to stop Timer/Counter 0
3	IE1	External interrupt 1 edge flag; Set by the hardware when external interrupt is detected on $\overline{\text{INT1}}$ pin; Cleared by the hardware when the controller vectors to interrupt service routine at address 0013H only when interrupt is configured as an edge-triggered interrupt (see IT1 bit below)
2	IT1	External Interrupt 1 signal type control bit; set to 1 by a program to configure interrupt 1 as an edge-triggered (falling edge); cleared to 0 to configure it as level-triggered (low level)
1	IE0	External interrupt 0 edge flag; Set by the hardware when external interrupt is detected on $\overline{\text{INT0}}$ pin; Cleared by the hardware when the controller vectors to interrupt service routine at address 0003H only when interrupt is configured as an edge triggered interrupt (see IT0 bit below)
0	IT0	External Interrupt 0 signal type control bit; set to 1 by a program to configure interrupt 0 as an edge-triggered (falling edge); cleared to 0 to configure it as level-triggered (low level)

SCON: Serial Port Control Register (Address: 98H)

Bit address	9F	9E	9D	9C	9B	9A	99	98
Bit name	SCON.7	SCON.6	SCON.5	SCON.4	SCON.3	SCON.2	SCON.1	SCON.0
Symbol	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
	MSB							LSB

Bit	Symbol	Description
7	SM0	Serial Port Mode [see table below *]
6	SM1	
5	SM2	Enables multiprocessor I/O in Mode 2 and 3. When set to 1, an interrupt is generated if bit 9 of received data is 1, no interrupt is generated if bit 9 is 0. If Set to 1 for Mode 1, no interrupt will be generated unless a valid stop bit is received. Clear to 0 for Mode 0
4	REN	Receive enable if REN=1
3	TB8	9^{th} data bit to send in 9 bit mode in Modes 2 and 3
2	RB8	9^{th} data bit received in Modes 2 & 3. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode0, RB8 is not used
1	TI	Transmit Interrupt flag (Transmitter empty Interrupt Flag) – sending finished. Set by the hardware at the end of the 8^{th} bit time in Mode 0, or at the beginning of the stop bit in the other modes. It's a signal to the microcontroller that the line is available to transmit a new byte. Must be cleared by software
0	RI	Receive Interrupt Flag – new byte received. Set by the hardware at the end of the 8^{th} bit time in Mode 0, or halfway through the stop bit time in the other modes. It signals that a byte is received and should be read quickly prior to being replaced by a new data. Must be cleared by the software

*Serial port mode is selected by the SM0 and SM1 bits:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	8-bit Shift, Register	1/12 the crystal frequency
0	1	1	8-bit UART	Determined by Timer 1
1	0	2	9-bit UART	1/32 the crystal frequency (1/64 the crystal frequency)
1	1	3	9-bit UART	Determined by Timer 1

IE: Interrupt Enable Register (Address: A8H)

Bit address	AF	--	AD	AC	AB	AA	A9	A8
Bit name	IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
Symbol	EA	--	ET2	ES	ET1	EX1	ET0	EX0
	MSB							LSB

Bit	Symbol	Description
7	EA	Global Enable bit; EA=1 allows each interrupt to be individually enabled or disabled; EA= 0 will disable all the interrupts
6	--	Not implemented, reserved for future use
5	ET2	Used by later versions of 8051 for Timer 2
4	ES	ES= 1 enables serial port interrupt, while ES=0 will disable it
3	ET1	ET1= 1 enables Timer 1 overflow interrupt, while ET1=0 will disable it
2	EX1	EX1= 1 enables external interrupt 1, while EX1=0 will disable it
1	ET0	ET0= 1 enables Timer 0 overflow interrupt, while ET0=0 will disable it
0	EX0	EX0= 1 enables external interrupt 0, while EX0=0 will disable it

IP: Interrupt Priority Register (Address: B8H)

Bit address	--	--	BD	BC	BB	BA	B9	B8
Bit name	IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
Symbol	--	--	PT2	PS	PT1	PX1	PT0	PX0
	MSB							LSB

Bit	Symbol	Description
7	--	Reserved
6	--	Reserved
5	PT2	Priority bit for Timer 2 interrupt (8052 only)
4	PS	Priority bit for serial port interrupt
3	PT1	Priority bit for Timer 1 interrupt
2	PX1	Priority bit for external interrupt 1
1	PT0	Priority bit for Timer 0 interrupt
0	PX0	Priority bit for External interrupt 0
Priority bit = 1 ; high-priority level		
Priority bit = 0 ; low-priority level		

PCON: Power Control Register (Address: 87H)

SMOD	--	--	--	GF1	GF0	PDWN	IDLE
MSB							LSB

Bit	Symbol	Description
7	SMOD	Serial baud rate generation mode. When SMOD=1, the baud rate of the UART is doubled.
6	--	
5	--	
4	--	
3	GF1	General-purpose user flag 1. Can be used to store 1 bit information.
2	GF0	General-purpose user flag 0. Can be used to store 1 bit information.
1	PWDN	Power-down bit. PWDN =1 will activate the power-down mode
0	IDLE	Idle mode bit. IDLE =1 will activate the idle mode.

Index