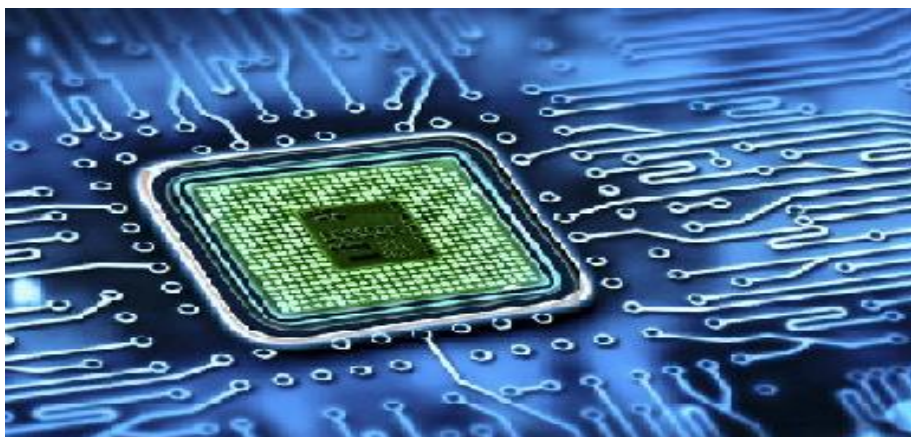


**MUFFAKHAM JAH COLLEGE OF ENGINEERING AND
TECHNOLOGY**
(Affiliated to Osmania University)
Banjara Hills, Hyderabad, Telangana State.



INFORMATION TECHNOLOGY DEPARTMENT
EMBEDDED SYESTEM
LAB MANUAL



TABEL OF CONTENTS

S.No.	Content	Page No.
1.	Institute Vision	I
2.	Institute Mission	I
3.	Department Vision	II
4.	Department Mission	II
5.	PEOs	III
6.	POs	IV
7.	PSOs	IV
8.	Introduction to Embedded Systems laboratory	1
Programs using 8051 Microcontroller		
9.	Program 1: To write a C program to demonstrate LED using 8051 Microcontroller development kit.	21
10.	Program 2: To write a C program to demonstrate Seven Segment using 8051 Microcontroller development kit	24
11.	Program 3: To write a C program to demonstrate Traffic Light Signals using 8051 Microcontroller development kit.	28
12.	Program 4: To write a program for demonstrating Relays and Buzzers using 8051	31
13.	Program 5: To write a program to demonstrate Stepper Motor using 8051 Microcontroller development kit.	35
14.	Program 6: To write a program to demonstrate LCD using 8051 Microcontroller development kit..	39
15.	Program 7: To write a program to demonstrate Keypad using 8051 Microcontroller development kit.	45
16.	Program 8: To Write a program to demonstrate Elevator Controller using 8051 microcontroller development kit.	52
Programs using ARM (RTOS)		
17.	Program 9: Demonstrate the TIMING concept of real time application using RTOS on ARM microcontroller kit.	58
18.	Program 10: Demonstrate the Multi Tasking concept of real time application using RTOS on ARM microcontroller kit.	62
19.	Program 11: Demonstrate the SEMAPHORE concept of real time application using RTOS on ARM microcontroller kit.	66

20.	Program 12: Demonstrate the Message Queues concept of real time application using RTOS on ARM microcontroller kit.	71
21.	Program 13: Demonstrate the Round Robin task scheduling using RTOS on ARM microcontroller kit.	77
22.	Program 14: Demonstrate the Pre-emptive priority based task scheduling using RTOS on ARM microcontroller kit.	82
23.	Program 15: Demonstrate the Priority Inversion based task scheduling using RTOS on ARM microcontroller kit.	87
24.	Program 16: Demonstrate the RS232 serial communication using RTOS on ARM microcontroller kit	90
25.	Annexure – I : OU prescribed programs for ES Laboratory	94

1. Institution Vision

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Institution Mission

- To attain excellence in imparting technical education from the undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs.
- To foster partnership with industry and government agencies through collaborative research and consultancy.
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space.
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents.
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders.
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department Vision

Fostering a bright technological future by enabling the students to function as leaders in software industry and serve as means of transformation to empower society through ITeS.

4. Department Mission

To create an ambience of academic excellence through state of art infrastructure and learner-centric pedagogy leading to employability in multi-disciplinary fields.

5. Program Education Objectives

1. Graduates will demonstrate technical competence and leadership in their chosen fields of employment by identifying, formulating, analyzing and creating efficient IT solutions.
2. Graduates will communicate effectively as individuals or team members and be successful in varied working environment.
3. Graduates will demonstrate lifelong learning through continuing education and professional development.
4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical context.

6. Program Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

7. Programme Specific Outcomes:

1. The IT graduates will work as software engineers for providing solutions to real world problems using structured and object oriented programming languages and open source software.
2. The IT graduates will work as System engineer, Software analyst and Tester for IT and ITes.

EMBEDDED SYESTEM LAB

GENERAL GUIDELINES, PRECAUSIONS AND SAFETY INSTRUCTIONS

1. Sign in the log register as soon as you enter the lab and strictly observe your lab timings.
2. Strictly follow the written and verbal instructions given by the teacher / Lab Instructor. If you do not understand the instructions, the handouts and the procedures, ask the instructor or teacher.
3. Do not work alone! You should be accompanied by your laboratory partner and / or the instructors / teaching assistants all the time.
4. It is mandatory to come to lab in a formal dress and wear your ID cards.
5. Do not wear loose-fitting clothing or jewellery in the lab. Rings and necklaces are usual excellent conductors of electricity.
6. Mobile phones should be switched off in the lab. Keep bags in the bag rack.
7. Keep the labs clean at all times, no food and drinks allowed inside the lab.
8. Intentional misconduct will lead to expulsion from the lab.
9. Do not handle any equipment without reading the safety instructions. Read the handout and procedures in the Lab Manual before starting the experiments.
10. Do your wiring, setup, and a careful circuit checkout before applying power. Do not make circuit changes or perform any wiring when power is on.
11. Avoid contact with energized electrical circuits.
12. Do not insert connectors forcefully into the sockets.
13. Do not try to experiment with the power from the wall plug.
14. Immediately report dangerous or exceptional conditions to the Lab instructor / teacher: Equipment that is not working as expected, wires or connectors are broken, the equipment that smells or “smokes”. If you are not sure what the problem is or what's going on, switch off the Emergency shutdown.
15. Never use damaged instruments, wires or connectors. Hand over these parts to the Lab instructor/Teacher.
16. Be sure of location of fire extinguishers and first aid kits in the laboratory.
17. After completion of Experiment, return the trainer kits, wires, and other components to lab staff. Do not take any item from the lab without permission.
18. Observation book and lab record should be carried to each lab. Readings of current lab experiment are to be entered in observation book and previous lab experiment should be written in Lab record book. Both the books should be corrected by the faculty in each lab.

Introduction to Embedded Systems laboratory

The platform 8051 development kit is intended as a demonstration and evaluation of ATMEL Core 8051 Microcontroller. The kit is a general purpose, low cost and highly expandable micro controller system. It is based on the ATMEL 8051 single chip flash Micro controller.

This controller is a compact, high performance true single board controller. It is perfectly suited for process control. Low voltage values, motor drivers and input switches or sensors can be directly connected to the robust and removable I/O screw terminals.

Laboratory Objective

Upon successful completion of this Lab the student will be able to:

- Apply the design concepts for development of a process and interpret data.
- Demonstrate knowledge of programming environment, compiling, debugging, linking and executing variety of programs.
- Demonstrate documentation and presentation of the algorithms / flowcharts / programs in a record form.
- Validate the process using known input-output parameters.
- Employ analytical and logical skills to solve real world problem and demonstrate oral communication skills.

OVERVIEW OF EMBEDDED SYSTEMS

AT89C51 MICROCONTROLLER

FEATURES

- 80C51 based architecture
- 4-Kbytes of on-chip Reprogrammable Flash Memory
- 128 x 8 RAM
- Two 16-bit Timer/Counters
- Full duplex serial channel
- Boolean processor
- Four 8-bit I/O ports, 32 I/O lines
- Memory addressing capability
 - 64K ROM and 64K RAM
- Power save modes:
 - Idle and power-down
- Six interrupt sources
- Most instructions execute in 0.3 us
- CMOS and TTL compatible
- Maximum speed: 40 MHz @ $V_{cc} = 5V$
- Industrial temperature available
- Packages available:
 - 40-pin DIP
 - 44-pin PLCC
 - 44-pin PQFP

GENERAL DESCRIPTION:

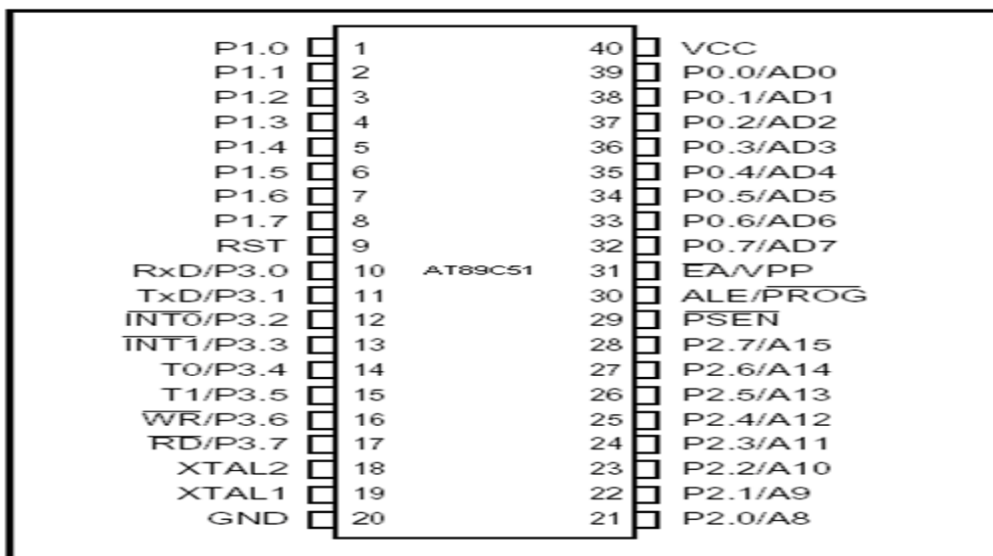
THE MICROCONTROLLER:

A microcontroller is a general purpose device, but that is meant to read data, perform limited calculations on that data and control its environment based on those calculations. The prime use of a microcontroller is to control the operation of a machine using a fixed program that is stored in ROM and that does not change over the lifetime of the system.

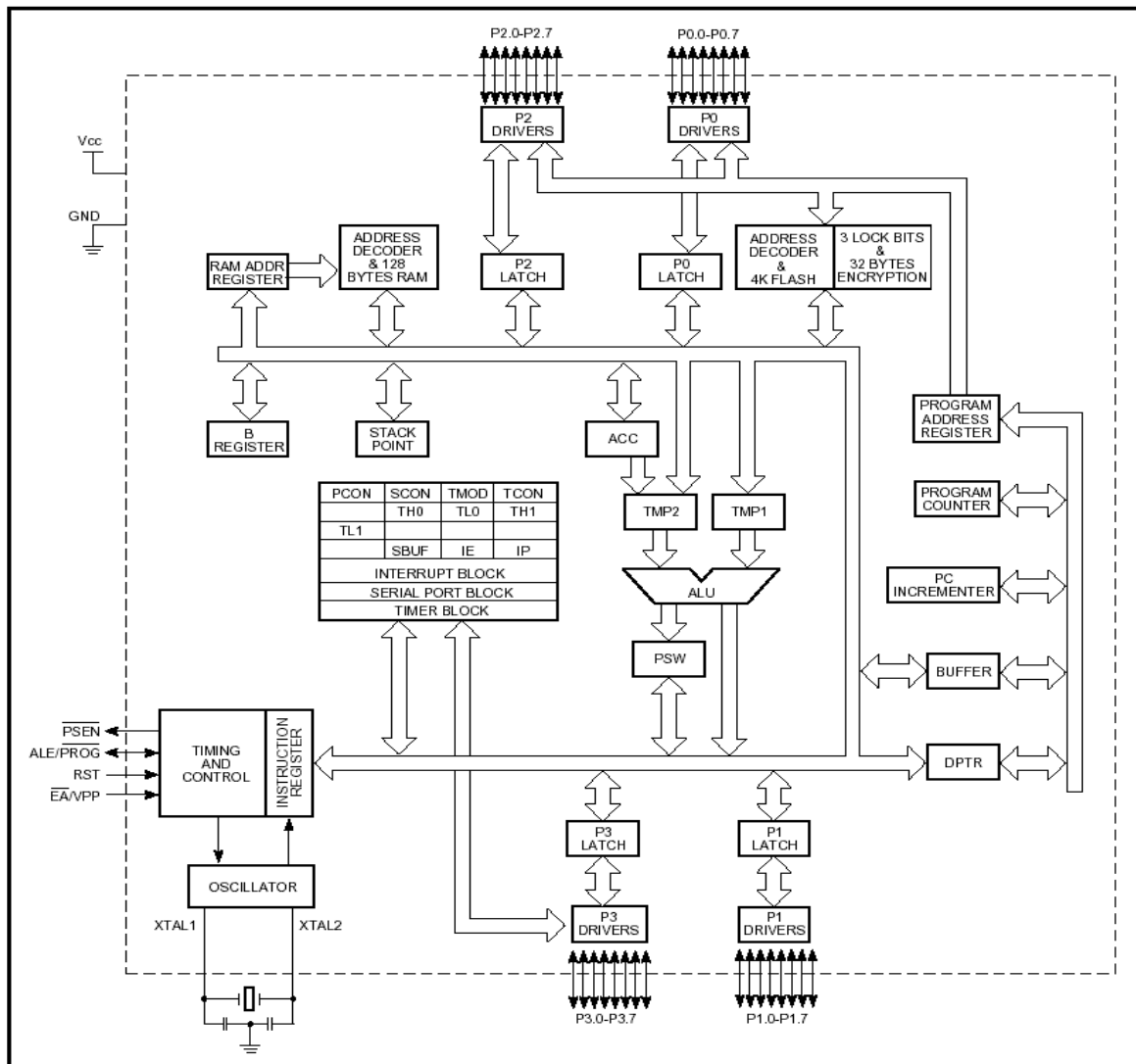
The microcontroller design uses a much more limited set of single and double byte instructions that are used to move data and code from internal memory to the ALU. The microcontroller is concerned with getting data from and to its own pins; the architecture and instruction set are optimized to handle data in bit and byte size.

The AT89C51 is a low-power, high-performance CMOS 8-bit microcontroller with 4k bytes of Flash Programmable and erasable read only memory (EROM). The device is manufactured using Atmel's high-density nonvolatile memory technology and is functionally compatible with the industry-standard 80C51 microcontroller instruction set and pin out. By combining versatile 8-bit CPU with Flash on a monolithic chip, the Atmel's AT89c51 is a powerful microcomputer, which provides a high flexible and cost- effective solution to many embedded control applications.

Pin configuration of AT89c51 Microcontroller



AT89C51 Block Diagram



PIN DESCRIPTION:

VCC-Supply voltage

GND-Ground

Port 0

Port 0 is an 8-bit open drain bi-directional I/O port. As an output port, each pin can sink eight TTL inputs. When 1s are written to port 0 pins, the pins can be used as high impedance inputs.

Port 0 can also be configured to be the multiplexed low order address/data bus during access to external program and data memory. In this mode, P 0 has internal pull-ups. Port 0 also receives the code bytes during Flash programming and outputs the code bytes during program verification. External pull-ups are required during program verification.

Port 1

Port 1 is an 8-bit bi-directional I/O port with internal pull-ups. The port 1 output buffers can sink/source four TTL inputs. When 1s are written to port 1 pins, they are pulled high by the internal pull-ups can be used as inputs. As inputs, Port 1 pins that are externally being pulled low will source current (1) because of the internal pull-ups.

Port 2

Port 2 is an 8-bit bi-directional I/O port with internal pull-ups. The port 2 output buffers can sink/source four TTL inputs. When 1s are written to port 2 pins, they are pulled high by the internal pull-ups can be used as inputs. As inputs, Port 2 pins that are externally being pulled low will source current because of the internal pull-ups.

Port 2 emits the high-order address byte during fetches from external program memory and during access to DPTR. In this application Port 2 uses strong internal pull-ups when emitting 1s. During accesses to external data memory that use 8-bit data address (MOVX@R1), Port 2 emits the contents of the P2 Special Function Register. Port 2 also receives the high-order address bits and some control signals during Flash programming and verification.

Port 3

Port 3 is an 8-bit bi-directional I/O port with internal pull-ups. The port 3 output buffers can sink/source four TTL inputs. When 1s are written to port 3 pins, they are pulled high by the internal pull-ups can be used as inputs. As inputs, Port 3 pins that are externally being pulled low will source current because of the internal pull-ups.

Port 3 also receives some control signals for Flash Programming and verification.

Port pin	Alternate Functions
P3.0	RXD(serial input port)
P3.1	TXD(serial input port)
P3.2	INT0(external interrupt 0)
P3.3	INT1(external interrupt 1)
P3.4	T0(timer 0 external input)
P3.5	T1(timer 1 external input)
P3.6	WR(external data memory write strobe)
P3.7	RD(external data memory read strobe)

RST

Rest input A on this pin for two machine cycles while the oscillator is running resets the device.

ALE/PROG:

Address Latch Enable is an output pulse for latching the low byte of the address during access to external memory. This pin is also the program pulse input (PROG) during Flash programming.

In normal operation ALE is emitted at a constant rate of 1/16 the oscillator frequency and may be used for external timing or clocking purpose. Note, however, that one ALE pulse is skipped during each access to external Data memory.

PSEN

Program Store Enable is the read strobe to external program memory when the AT89c51 is executing code from external program memory PSEN is activated twice each machine cycle, except that two PSEN activations are skipped during each access to external data memory.

EA /VPP

External Access Enable (EA) must be strapped to GND in order to enable the device to fetch code from external program memory locations starting at 0000h up to FFFFH. Note, however, that if lock bit 1 is programmed EA will be internally latched on reset. EA should be strapped to Vcc for internal program executions. This pin also receives the 12-volt programming enable voltage (Vpp) during Flash programming when 12-volt programming is selected.

XTAL1

Input to the inverting oscillator amplifier and input to the internal clock operating circuit.

XTAL 2

Output from the inverting oscillator amplifier.

OPERATING DESCRIPTION

The detail description of the AT89C51 included in this description is:

- Memory Map and Registers
- Timer/Counters
- Interrupt System

MEMORY MAP AND REGISTERS:

Memory

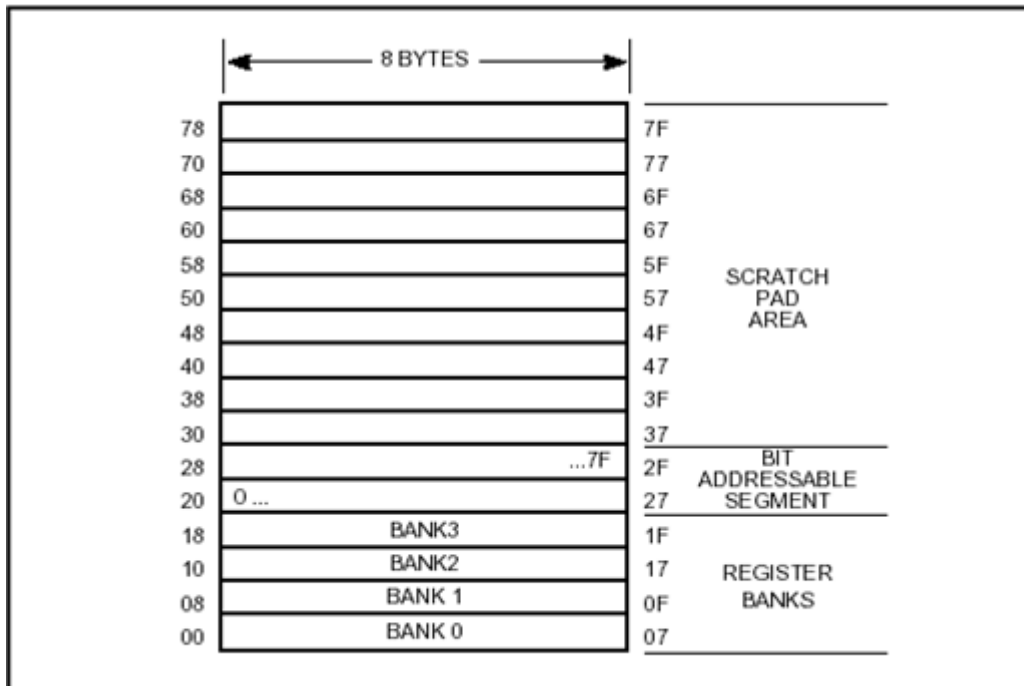
The AT89C51 has separate address spaces for program and data memory. The program and data memory can be up to 64K bytes long. The lower 4K program memory can reside on-chip. The AT89C51 has 128 bytes of on-chip RAM.

The lower 128 bytes can be accessed either by direct addressing or by indirect addressing. The lower 128 bytes of RAM can be divided into 3 segments as listed below

1. **Register Banks 0-3:** locations 00H through 1FH (32 bytes). The device after reset defaults to register bank 0. To use the other register banks, the user must select them in software. Each register bank contains eight 1-byte registers R0-R7. Reset initializes the stack point to location 07H, and is incremented once to start from 08H, which is the first register of the second register bank.

2. **Bit Addressable Area:** 16 bytes have been assigned for this segment 20H-2FH. Each one of the 128 bits of this segment can be directly addressed (0-7FH). Each of the 16 bytes in this segment can also be addressed as a byte.

3. **Scratch Pad Area:** 30H-7FH are available to the user as data RAM. However, if the data pointer has been initialized to this area, enough bytes should be left aside to prevent SP data destruction.



SPECIAL FUNCTION REGISTERS:

The Special Function Registers (SFR's) are located in upper 128 Bytes direct addressing area. The SFR Memory Map in shows that.

Not all of the addresses are occupied. Unoccupied addresses are not implemented on the chip. Read accesses to these addresses in general return random data, and write accesses have no effect. User software should not write 1s to these unimplemented locations, since they may be used in future microcontrollers to invoke new features. In that case, the reset or inactive values of the new bits will always be 0, and their active values will be 1.

The functions of the SFR's are outlined in the following sections.

Accumulator (ACC)

ACC is the Accumulator register. The mnemonics for Accumulator-specific instructions, however, refer to the Accumulator simply as A.

B Register (B)

The B register is used during multiply and divide operations. For other instructions it can be treated as another scratch pad register.

Program Status Word (PSW)

The PSW register contains program status information.

Stack Pointer (SP)

The Stack Pointer Register is eight bits wide. It is incremented before data is stored during PUSH and CALL executions. While the stack may reside anywhere in on chip RAM, the Stack Pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H.

Data Pointer (DPTR)

The Data Pointer consists of a high byte (DPH) and a low byte (DPL). Its function is to hold a 16-bit address. It may be manipulated as a 16-bit register or as two independent 8-bit registers.

Serial Data Buffer (SBUF)

The Serial Data Buffer is actually two separate registers, a transmit buffer and a receive buffer register. When data is moved to SBUF, it goes to the transmit buffer, where it is held for serial transmission. (Moving a byte to SBUF initiates the transmission.) When data is moved from SBUF, it comes from the receive buffer.

Timer Registers

Register pairs (TH0, TL0) and (TH1, TL1) are the 16-bit Counter registers for Timer/Counters 0 and 1, respectively.

Control Registers

Special Function Registers IP, IE, TMOD, TCON, SCON, and PCON contain control and status bits for the interrupt system, the Timer/Counters, and the serial port.

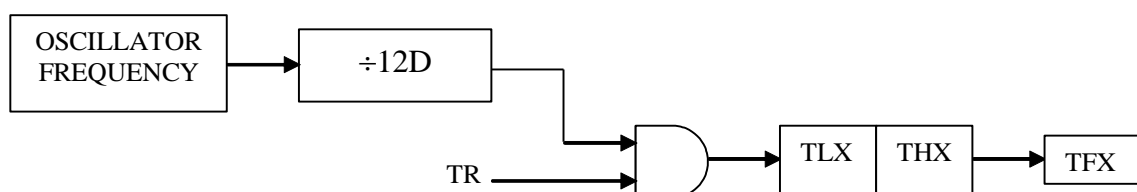
TIMER / COUNTERS:

The IS89C51 has two 16-bit Timer/Counter registers: Timer 0 and Timer 1. All two can be configured to operate either as Timers or event counters. As a Timer, the register is incremented every machine cycle. Thus, the register counts machine cycles. Since a machine cycle consists of 12 oscillator periods, the count rate is 1/12 of the oscillator frequency.

As a Counter, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T0 and T1. The external input is sampled during S5P2 of every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value appears in the register during S3P1 of the cycle following the one in which the transition was detected. Since two machine cycles (24 oscillator periods) are required to recognize a 1-to-0 transition, the maximum count rate is 1/24 of the oscillator frequency. There are no restrictions on the duty cycle of the external input signal, but it should be held for at least one full machine cycle to ensure that a given level is sampled at least once before it changes.

In addition to the Timer or Counter functions, Timer 0 and Timer 1 have four operating modes: 13-bit timer, 16-bit timer, 8-bit auto-reload, split timer.

TIMERS:



SFR'S USED IN TIMERS

The special function registers used in timers are,

- TMOD Register
- TCON Register
- Timer(T0) & timer(T1) Registers

(i) TMOD Register:

TMOD is dedicated solely to the two timers (T0 & T1).

- The timer mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, or 13 bit timer, 8-bit auto reload timer, or two separate timers. Additionally you may configure the timers to only count when an external pin is activated or to count “events” that are indicated on an external pin.
- It can consider as two duplicate 4-bit registers, each of which controls the action of one of the timers.

(ii) TCON Register:

- The timer control SFR is used to configure and modify the way in which the 8051’s two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in TCON SFR.
- These bits are used to configure the way in which the external interrupt flags are activated, which are set when an external interrupt occurs.

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
-----	-----	-----	-----	-----	-----	-----	-----

(iii) TIMER 0 (T0):

- T0 (Timer 0 low/high, address 8A/8C h)

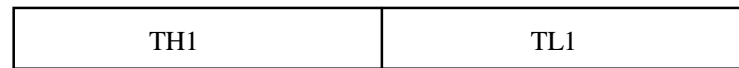
These two SFR’s taken together represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

TH0	TL0
-----	-----

(iv) TIMER 1 (T1):

- T1 (Timer 1 Low/High, address 8B/ 8D h)

These two SFR's, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is Configurable is how and when they increment in value.



The Timer or Counter function is selected by control bits C/T in the Special Function Register TMOD. These two Timer/Counters have four operating modes, which are selected by bit pairs (M1, M0) in TMOD. Modes 0, 1, and 2 are the same for both Timer/Counters, but Mode 3 is different.

The **FOUR** modes are described in the following sections:

Mode 0:

Both Timers in Mode 0 are 8-bit Counters with a divide-by-32 pre scalar. Figure 8 shows the Mode 0 operation as it applies to Timer 1. In this mode, the Timer register is configured as a 13-bit register. As the count rolls over from all 1s to all 0s, it sets the Timer interrupt flag TF1. The counted input is enabled to the Timer when TR1 = 1 and either GATE = 0 or INT1 = 1. Setting GATE = 1 allows the Timer to be controlled by external input INT1, to facilitate pulse width measurements. TR1 is a control bit in the Special Function Register TCON. Gate is in TMOD.

The 13-bit register consists of all eight bits of TH1 and the lower five bits of TL1. The upper three bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the registers.

Mode 0 operation is the same for Timer 0 as for Timer 1, except that TR0, TF0 and INTO replace the corresponding Timer 1 signals. There are two different GATE bits, one for Timer 1 (TMOD.7) and one for Timer 0 (TMOD.3).

Mode 1

Mode 1 is the same as Mode 0, except that the Timer register is run with all 16 bits. The clock is applied to the combined high and low timer registers (TL1/TH1). As clock pulses are received, the timer counts up: 0000H, 0001H, 0002H, etc. An overflow occurs on the FFFFH-to-0000H overflow flag. The timer continues to count. The overflow flag is the TF1 bit in TCON that is read or written by software

Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TL1) with automatic reload, as shown in Figure 10. Overflow from TL1 not only sets TF1, but also reloads TL1 with the contents of TH1, which is preset by software. The reload leaves the TH1 unchanged. Mode 2 operation is the same for Timer/Counter 0.

Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0. Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. The logic for Mode 3 on Timer 0 is shown in Figure 11. TL0 uses the Timer 0 control bits: C/T, GATE, TR0, INT0, and TF0. TH0 is locked into a timer function (counting machine cycles) and over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the Timer 1 interrupt.

Mode 3 is for applications requiring an extra 8-bit timer or counter. With Timer 0 in Mode 3, the AT89C51 can appear to have three Timer/Counters. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3. In this case, Timer 1 can still be used by the serial port as a baud rate generator or in any application not requiring an interrupt.

INTERRUPT SYSTEM

An interrupt is an external or internal event that suspends the operation of micro controller to inform it that a device needs its service. In interrupt method, whenever any device needs its service, the device notifies the micro controller by sending it an interrupt signal. Upon receiving an interrupt signal, the micro controller interrupts whatever it is doing and serves the device. The program associated with interrupt is called as **interrupt service subroutine (ISR)**. Main advantage with interrupts is that the micro controller can serve many devices.

Baud Rate

The baud rate in Mode 0 is fixed as shown in the following equation. Mode 0 Baud Rate = Oscillator Frequency / 12 the baud rate in Mode 2 depends on the value of the SMOD bit in Special Function Register PCON. If SMOD = 0 the baud rate is 1/64 of the oscillator frequency.

If SMOD = 1, the baud rate is 1/32 of the oscillator frequency.

Mode 2 Baud Rate = 2SMODx (Oscillator Frequency)/64.

In the IS89C51, the Timer 1 overflow rate determines the baud rates in Modes 1 and 3.

NUMBER OF INTERRUPTS IN 89C51:

There are basically five interrupts available to the user. Reset is also considered as an interrupt. There are two interrupts for timer, two interrupts for external hardware interrupt and one interrupt for serial communication.

<u>Memory location</u>	<u>Interrupt name</u>
0000H	Reset
0003H	External interrupt 0
000BH	Timer interrupt 0
0013H	External interrupt 1
001BH	Timer interrupt 1
0023H	Serial COM interrupt

Lower the vector, higher the priority. The External Interrupts INT0 and INT1 can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON. The flags that actually generate these interrupts are the IE0 and IE1 bits in TCON. When the service routine is vectored, hardware clears the flag that generated an external interrupt only if the interrupt was transition-activated. If the interrupt was level-activated, then the external requesting source (rather than the on-chip hardware) controls the request flag.

The Timer 0 and Timer 1 Interrupts are generated by TF0 and TF1, which are set by a rollover in their respective Timer/Counter registers (except for Timer 0 in Mode 3). When a timer interrupt is generated, the on-chip hardware clears the flag that is generated.

The Serial Port Interrupt is generated by the logical OR of RI and TI. The service routine normally must determine whether RI or TI generated the interrupt, and the bit must be cleared in software.

All of the bits that generate interrupts can be set or cleared by software, with the same result as though they had been set or cleared by hardware. That is, interrupts can be generated and pending interrupts can be canceled in software.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in Special Function Register IE (interrupt enable) at address 0A8H. There is a global enable/disable bit that is cleared to disable all interrupts or to set the interrupts.

IE (Interrupt enable register):

Steps in enabling an interrupt:

Bit D7 of the IE register must be set to high to allow the rest of register to take effect. If EA=1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA=0, no interrupt will be responded to even if the associated bit in the IE register is high.

Description of each bit in IE register:

D7 bit: Disables all interrupts. If EA =0, no interrupt is acknowledged, if EA=1 each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

- D6 bit: Reserved.
- D5 bit: Enables or disables timer 2 over flow interrupt (in 8052).
- D4 bit: Enables or disables serial port interrupt.
- D3 bit: Enables or disables timer 1 over flow interrupt.
- D2 bit: Enables or disables external interrupt 1.
- D1 bit: Enables or disables timer 0 over flow interrupt.
- D0 bit: Enables or disables external interrupt 0.

Interrupt priority in 89C51:

There is one more SRF to assign priority to the interrupts which is named as interrupt priority (IP). User has given the provision to assign priority to one interrupt. Writing one to that particular bit in the IP register fulfils the task of assigning the priority.

Description of each bit in IP register:

D7 bit: Reserved.

D6 bit: Reserved.

D5 bit: Timer 2 interrupt priority bit (in 8052).

D4 bit: Serial port interrupt priority bit.

D3 bit: Timer 1 interrupt priority bit.

D2 bit: External interrupt 1 priority bit.

D1 bit: Timer 0 interrupt priority bit.

D0 bit: External interrupt 0 priority bit

8051 CORE MICRO CONTROLLER:

General Description

8051 is a high performance microcontroller fabricated using CMOS technology. 8051 is an 8-Bit Micro Controller with 4-Kbytes of Flash memory, 128 Bytes On-chip RAM, 32 programmable I/O Lines, two 16-bit Timers/Counters, 6 Interrupts/2 Priority Levels, UART and an on-chip oscillator and clock circuit. The AT89S52 can be expanded using standard TTL compatible memory.

FEATURES OF KIT

- 8051 based architecture
- 4-Kbytes of on-chip Reprogrammable Flash Memory
- 256 x 8 RAM
- Two 16-bit Timer/Counters
- Full duplex serial channel
- Boolean processor
- Four 8-bit I/O ports, 32 I/O lines
- Memory addressing capability 64K ROM and 64K RAM
- Program memory lock: Lock bits (3)
- Power save modes: Idle and power-down
- Six interrupt sources
- CMOS and TTL compatible
- Maximum speed: 40 MHz @ Vcc = 5V
- Packages available: 40-pin DIP, 44-pin PLCC, 44-pin PQFP

FEATURES OF DEVELOPMENT BOARD

- In-System programming (ISP) facility for supported microcontrollers
- 16 x 2 Character LCD Display
- On board RS-232 compatible serial interface terminated in a 9 pin 'D' female Connector.
- One Temperature sensor (LM35).
- I2C EEPROM (AT 24C16) for storing non-volatile parameters
- I2C Real Time Clock (DS1307) with Lithium battery and SRAM
- I2C 4-Channel (8-Bit) A/D converter
- I2C 1-Channel (8-Bit) D/A Converter
- 4 x 4 Matrix Keypad (optional)

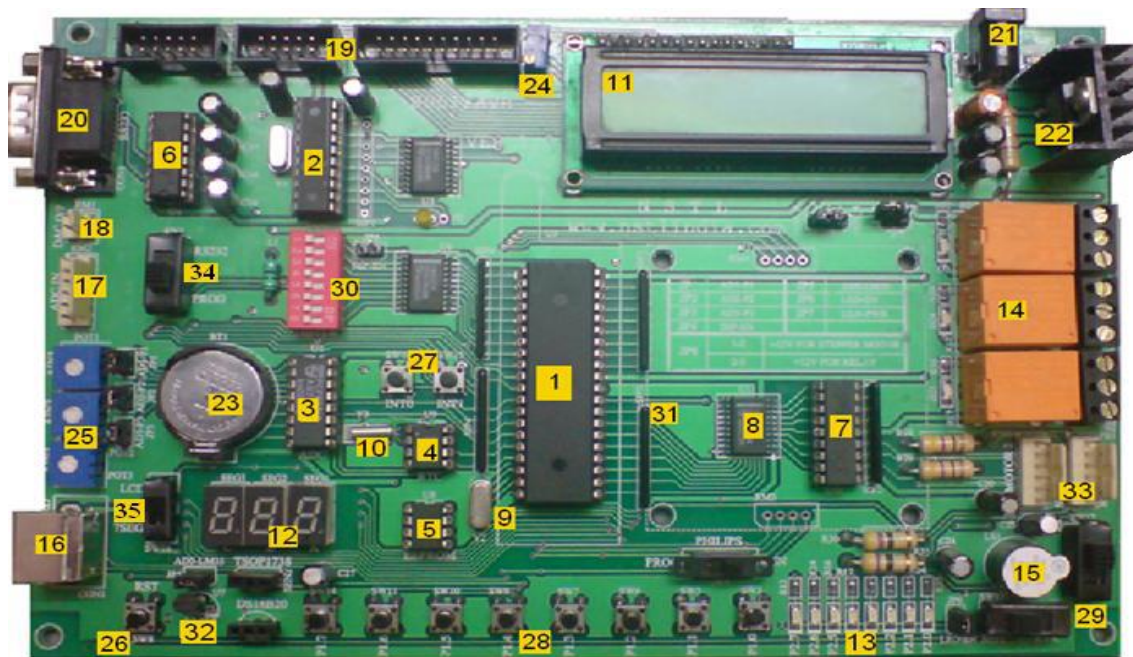
- 8 Push - to - on switches
- Standard AT keyboard Interface
- Two External Interrupts
- Three 7-Segment LED Displays
- 8-high current output pins (500mA) for driving external loads
- Stepper Motor Interfacing
- Supports up to 16 different Micro Controllers of 8051 and AVR Family

ADD ON MODULES

- 4 x 4 KEYPAD
- STEPPER MOTOR
- DS1820 (temperature sensor)
- TSOP1738 (38KHz IR-Receiver)

COMPONENTS

1. AT89S52 (40 PIN DIP)
2. AT89C2051 (ISP)
3. PCF8591 (ADC/DAC)
4. RTC
5. EEPROM
6. MAX 232
7. ULN 2803
8. 74HC573 LATCH (3- ICS)
9. CRYSTAL (11.0592MHZ)
10. CRYSTAL (32.768KHZ)
11. LCD (16X2 PIN CONNECTOR)
12. 7-Segment Display
13. SMD LEDS (8)
14. RELAYS (3)
15. Buzzer
16. PS2 CONNECTOR
17. 6 PIN RELEVANT CONNECTORS
18. 2-PIN RELEVANT CONNECTORS
19. 20 PIN, 10 PIN (2) CONNECTORS
20. DB9 CONNECTOR
21. DC SOCKET
22. 7805 REGULATOR
23. Battery
24. 10K POT (1 for LCD)
25. 10K POTS (3 for ADC)
26. RESET SWITCH (1)
27. INTERRUPT SWITCHES (2)
28. KEYPAD SWITCHES (8)
29. Switches (3)
30. DIP Switches
31. 4.7K RESISTORS (PULL UP RESISTORS)
32. LM-35 (Temperature sensor)
33. STEPPER MOTOR CONNECTORS (2)
34. RS232 or PROG mode switch
35. LCD or 7SEGMENT mode switch



Power supply

A 12V/1A DC power adapter is required to power the 8051 Starter kit. The power output from this adapter is supplied to LM7805 voltage regulator, which gives the constant 5V DC to the starter kit. Capacitors at the output take care of surge current. A few decoupling ceramic capacitors have also been placed around the board.

Address Table:

Device	Address (16-bit)
LCD_EN	0x8000
ADC	0x8400
Key Pad	0x8200
LED's	0x8600
4 x 4 Key Pad	0x8100
Stepper Motor	0x8500
7-Segment Display	0x8C00
DAC	0x8800
LCD Command Write	0x8000
LCD Check	0x8002
LCD Data Write	0x8001

7-Segment LED's Selection:

Device	Address (16-Bit)
7-Segment LED1	0x8C03
7-Segment LED2	0x8C05
7-Segment LED3	0x8C06

ADC PINS

PINS	Address(16-bit)
ADC_START	0x8B00
ADC_ALE	0x8F00

PROGRAM 1

LED

Program Description

In this program we try to glow all seven LED's in different formats (from L → R , R → L , even , odd). Initialize with LED address. For LED's to glow from Right to Left (R → L) one after the other set the counter c=0 and increment the counter by one till the value reaches c=7 and in between call set and clear LED function. LED's glow from Left to Right (L → R) one after the other. Initialize with LED address. For LED's to glow from Right to Left (R → L) one after the other set the counter c=7 and decrement the counter by one till the value reaches c=0 and in between call set and clear LED function.

PROGRAM FOR EXAMPLE OF LED:

```
#include<REGX51.H>
#define LED P2
void delay(unsigned int d);
int main(void)
{
    while(1)
    {
        LED=0x55;
        delay(1000);
        LED=0xAA;
        delay(1000);
    }
}
void delay(unsigned int d)
{
    unsigned int i,j;
    for(i=0; i<d; i++)
        for(j=0; j>101 ;j++);
}
```

PROGRAM TO SHOW L-R & R-L SHIFTING:

```
#include<REGX51.H>
#define LED P2
void delay(unsigned int d);
int main(void)
{
    unsigned int i;
```

```
while(1)
{
    LED=0X01;
    for(i=0;i<7;i++)
    {
        delay(1000);
        LED=LED<<=0X01;
    }
    LED=0X80;
    for(i=0;i<=7;i++)
    {
        delay(1000);
        LED=LED>>=0X01;
    }
}

void delay(unsigned int d)
{
    unsigned i,j;
    for(i=0;i<d;i++)
    for(j=0;j<101;j++);
}
```

Program Validation

Input:

Initialize with LED address. For LED's to glow from Right to Left (R → L) one after the other set the counter c=0 and increment the counter by one till the value reaches c=7.

Output:

LED's to glow from Right to Left (R → L) one after other.

LED EXAMPLE:



L-R7R-L SHIFT:



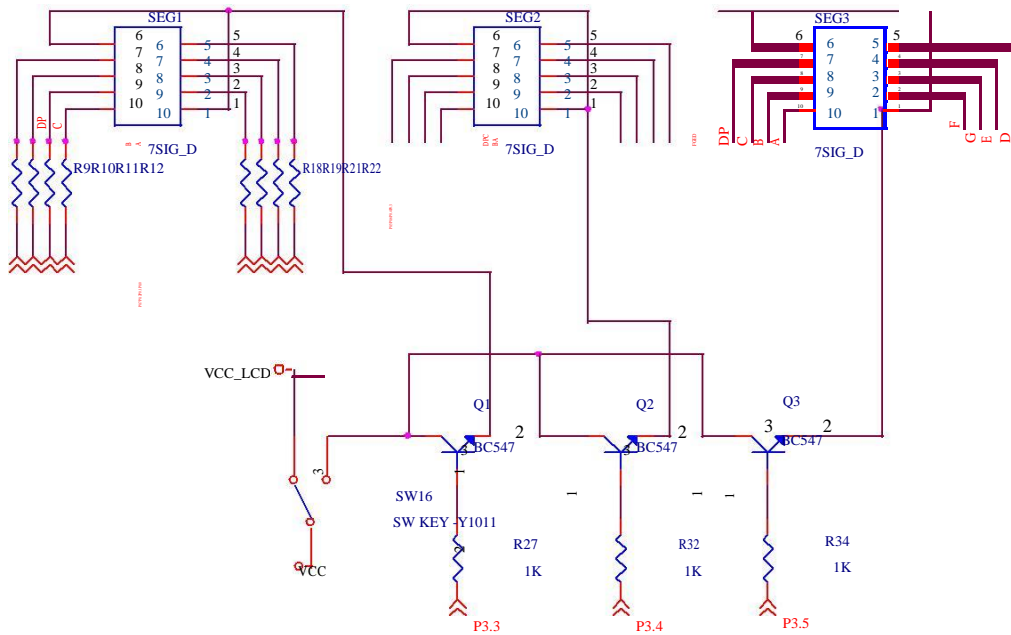
Conclusion:

The C program to demonstrate LED was executed successfully using 8051 Microcontroller development kit.

PROGRAM 2 SEVEN SEGMENT DISPLAY

Three Seven segment display are connected to port0 and the common anode pins of each seven segment are connected to port 3.3, 3.4 and 3.5 respectively. And the same port0 is connected to LCD as well, so to avoid the conflict we have provided the slide switch to select the appropriate display.

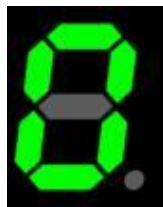
Schematic of Seven Segment Display:



General Description:

A seven segment display, as its name indicates, is composed of seven elements. Individually on or off, they can be combined to produce simplified representations of the Hindu _Arabic numerals. Often the seven segments are arranged in an *oblique*, or italic, arrangement, which aids readability. Each of the numbers 0, 1, 2 and 9 may be represented by two or more different glyphs on seven-segment displays. LED-based 7-segment display showing the 16 hex digits

The seven segments are arranged as a rectangle of two vertical segments on each side with one horizontal segment on the top and bottom. Additionally, the seventh segment bisects the rectangle horizontally. There are also fourteen –segment displays and sixteen segment displays (for full alphanumeric); however, these have mostly been replaced by dot-matrix displays. The segments of a 7-segment display are referred to by the letters A to G, as shown to the right, where the optional DP (decimal point an "eighth segment") is used for the display of non-integer numbers.



It is an image sequence of a "LED" display, which is described technology-wise in the following section. Notice the variation between uppercase and lowercase letters for A–F; this is done to obtain a unique, unambiguous shape for each letter.

ALGORITHM:

STEP1: Configure the Hardware connections of 7-SEGMENT devices

STEP2: Load the data on the port0 and enable 7-Seg1.

STEP3: Load the data on the port0 and enable 7-Seg2.

STEP4: Load the data on the port0 and enable 7-Seg3.

STEP5: End

PROGRAM TO INTERFACE 7 SEGMENT DISPLAY:

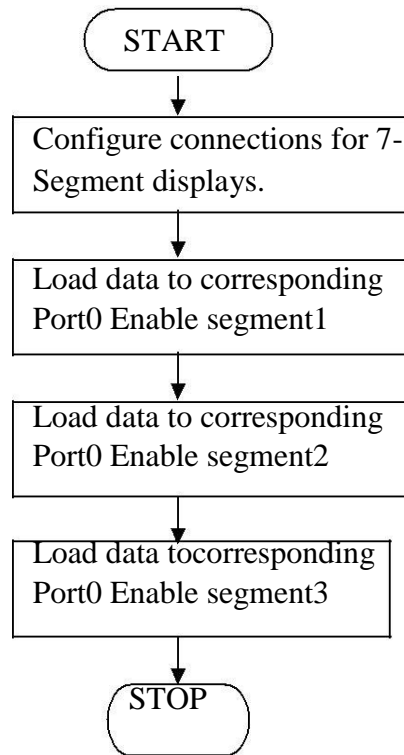
```
#include<REGX51.H>
#define SEG1 {P3_3=0;P3_4=0;P3_5=1;}
#define SEG2 {P3_3=0;P3_4=1;P3_5=0;}
#define SEG3 {P3_3=1;P3_4=0;P3_5=0;}
#define NULL {P3_3=0;P3_4=0;P3_5=0;}

code unsigned char seg[10]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};
int main(void)
{
    unsigned char i,j,k,m;
    unsigned int l;
    for(i=0;i<10;i++)
    {
        for(j=0;j<10;j++)
        {
            for(k=0;k<10;k++)
            {
                for(l=0;l<1000;l++)
                {
                    NULL
                    P0=seg[i];
                    SEG1
                    for(m=0;m<50;m++);

                    NULL
                    P0=seg[j];
                    SEG2
                    for(m=0;m<50;m++);

                    NULL
                    P0=seg[k];
                    SEG3
                    for(m=0;m<50;m++);
                }
            }
        }
    }
}
```

FLOW CHART:



Conclusion:

The C program to demonstrate LED SEVEN SEGMENT DISPLAY was executed successfully using 8051 Microcontroller development kit.

PROGRAM 3 TRAFFIC LIGHT SIGNALS

Program Definition

To write a C program to demonstrate Traffic Light signals using 8051 Microcontroller development kit.

Program Description

We demonstrate traffic signals i.e, all the possible ways where in flow of opposite directions is allowed and also free lefts. Writing hex code for traffic control combinations, we send data as (traffic light hex code) to LED address example led(0x99) then delay for some time , Again send other combination as data to LED

Algorithm:

1. Assign LED address for Microcontroller kit.
2. While true.
3. Write data (traffic light hex code) to LED address.
4. Delay for certain time.
5. Provide other possible ways to LED address.
6. Delay for certain time.
7. Repeat step 5 with different combination.

PROGRAM FOR TRAFFIC CONTROLLER:

```
#include<REGX51.H>
void delay (unsigned int d);
int main(void)
{
    while(1)
    {
        P2=0x54;
        delay(500);
        P2=0x56;
        delay(500);
        P2=0x51;
        delay(500);
        P2=0x59;
```

```

        delay(500);
        P2=0x45;
        delay(500);
        P2=0x65;
        delay(500);
        P2=0x15;
        delay(500);
        P2=0x95;
        delay(500);
    }
}

void delay (unsigned int d)
{
    unsigned int i,j;
    for(i=0;i<d;i++)
        for(j<0;j<101;j++);
}

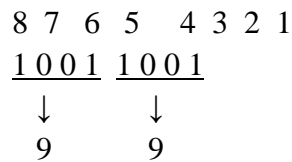
```

Program Validation

Input

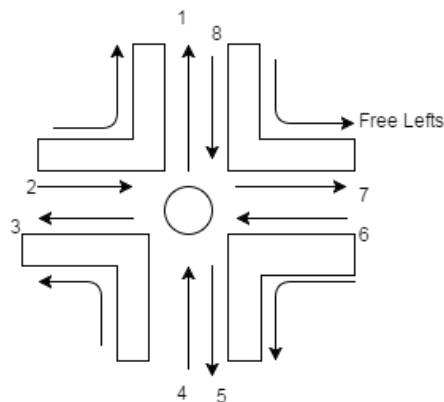
First making the LED's 1,4,5,8 as '1' & others as '0' then converting into decimal

eg: 1 4 5 8



Now, we send data as led(0x99)

Output



Conclusion:

The C program to demonstrate Traffic controller was executed successfully using 8051 Microcontroller development kit.

PROGRAM 4

RELAY AND BUZZER

Program Definition

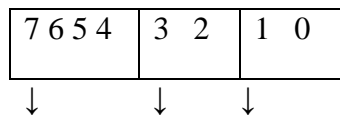
To write a program for demonstrating Relays Buzzer using 8051 development kit.

Program Description

RELAY

We demonstrate glowing of 2 relays one after the other . A relay is a device that responds to a small current or voltage change by activating switches or other devices in an electric circuit. Used for alarming systems.

Ports on MP:



Stepper-Motor Buzzer Relay

Relay 1:

$Prev=Prev | (1 \ll 0)$ (for glowing) , $Prev=Prev \& \sim(1 \ll 0)$ (for clearing)

Relay2 :

$Prev=Prev | (1 \ll 1)$ (for glowing), $Prev=Prev \& \sim(1 \ll 1)$ (for clearing)

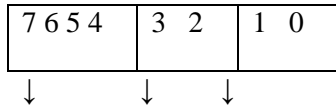
Algorithm

1. Assign LED address for Microcontroller kit
2. While true
3. Set the first relay to 1 which indicates that it is “ON”
4. Delay for certain time.
5. Set the relay to 0 which is “OFF”
6. Delay for certain time.
7. Set the realy2 to ‘1’
8. Delay for certain time.
9. Set the relay2 to ‘0’
10. Delay for certain time.

Program Description Buzzer

We demonstrate buzzing of two Buzzers. A **Buzzer** an electrical device that makes a buzzing noise and is used for signaling.

Ports on MP:



Stepper-Motor Buzzer Relay

For 1st Buzzer:

Set Buzzer: $\text{Prev}=\text{Prev} | (1 \ll 2)$

Reset Buzzer: $\text{Prev}=\text{Prev} \& \sim(1 \ll 2)$

For 2nd Buzzer:

Set Buzzer: $\text{Prev}=\text{Prev} | (1 \ll 3)$

Reset Buzzer: $\text{Prev}=\text{Prev} \& \sim(1 \ll 3)$

Algorithm

1. Assign LED address for Microcontroller kit.
2. While true.
3. Set the first buzzer to 1 which indicates that it is “ON”.
4. Delay for certain time.
5. Set the buzzer to 0 which is “OFF”
6. Delay for certain time.
7. Set the buzzer2 to ‘1’
8. Delay for certain time.
9. Set the buzzer2 to ‘0’
10. Delay for certain time.

PROGRAM TO INTERFACE RELAY AND BUZZER:

```
#include<REGX51.H>
#define RELAY1 P2_4
#define RELAY2 P2_5
#define RELAY3 P2_6
#define BUZZER P2_7
void delay(unsigned int d);
int main(void)
{
    P2=0x00;
    while(1)
    {
        RELAY1=1;
        delay(1000);
        RELAY1=0;
        delay(1000);
        RELAY2=1;
        delay(1000);
        RELAY2=0;
        delay(1000);
    }
}
```



```
        RELAY3=1;
        delay(1000);
        RELAY3=0;
        delay(1000);
        BUZZER=1;
        delay(1000);
        BUZZER=0;
        delay(1000);
    }
}

void delay(unsigned int d)
{
    unsigned int i,j;
    for(i=0;i<d;i++)
        for(j=0;j<101;j++);
}
```

Program Validation

Input for Relay:

Set the first relay to 1 which indicates that it is “ON”
Set the relay to 0 which is “OFF”
Set the relay2 to ‘1’ which indicates that it is “ON”
Set the relay2 to ‘0’ which is “OFF”

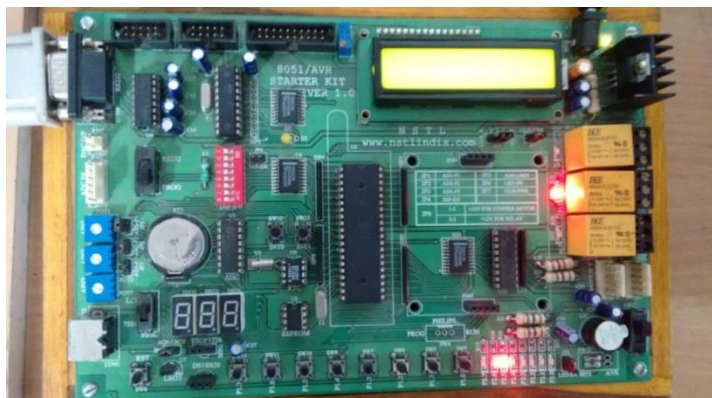
Input for Buzzer:

Set the first Buzzer1 to 1 which indicates that it is “ON”
Set the Buzzer1 to 0 which is “OFF”
Set the Buzzer2 to ‘1’ which indicates that it is “ON”
Set the Buzzer2 to ‘0’ which is “OFF”

Output :

Here two Relays glow one after the other.

RELAYS AND BUZZER:



Conclusion:

The C program to demonstrate Relays & Buzzer was executed successfully using 8051 Microcontroller development kit.

Program 5

STEPPER MOTOR

Program Definition

To write a program to demonstrate Stepper Motor using 8051 development kit.

Basics of Stepper Motor

Of all motors, step motor is the easiest to control. Direction information is very simple and comes down to "left" for logical one on that pin and "right" for logical zero. Motor control is also very simple - every impulse makes the motor operating for one step and if there is no impulse the motor won't start. Pause between impulses can be shorter or longer and it defines revolution rate. This rate cannot be infinite because the motor won't be able to "catch up" with all the impulses.

The key to driving a stepper is realizing how the motor is constructed. A diagram shows the representation of a 4 coil motor, so named because 4 coils are used to cause the revolution of the drive shaft. Each coil must be energized in the correct order for the motor to spin.

The control signals to open and close the switches at the appropriate times in order to spin the motors. The control unit is commonly a computer or programmable interface controller, with software directly generating the outputs needed to control the switches.

Step angle: It is angle through which motor shaft rotates in one step. step angle is different for different motor. Selection of motor according to step angle depends on the application, simply if you require small increments in rotation choose motor having smaller step angle.

No of steps require to rotate one complete rotation = $360 \text{ deg.} / \text{step angle in deg.}$

Steps/second

The relation between RPM and steps per sec. is given by, $\text{steps or impulses /sec.} = (\text{RPM} \times \text{Steps /revolution}) / 60$

Interfacing To 8051

Coil A	Coil B	Coil C	Coil D	Step
0	1	1	0	1
0	0	1	1	2
1	0	0	1	3
1	1	0	0	4

To cause the stepper to rotate, we have to send a pulse to each coil in turn. The 8051 does not have sufficient drive capability on its output to drive each coil, so there are a number of ways to drive a stepper, Stepper motors are usually controlled by transistor or driver IC like ULN2003.

Driving current for each coil is then needed about 60mA at +5V supply. A Darlington transistor array, ULN2003 is used to increase driving capacity of the 8051 chip. Four 4.7k resistors help the 8051 to provide more sourcing current from the +5V supply.

Controlling Stepper Motor With Two Port Pins Only

D0	D0	Coil energized
0	0	AB
0	1	BC
1	0	CD
1	1	DA

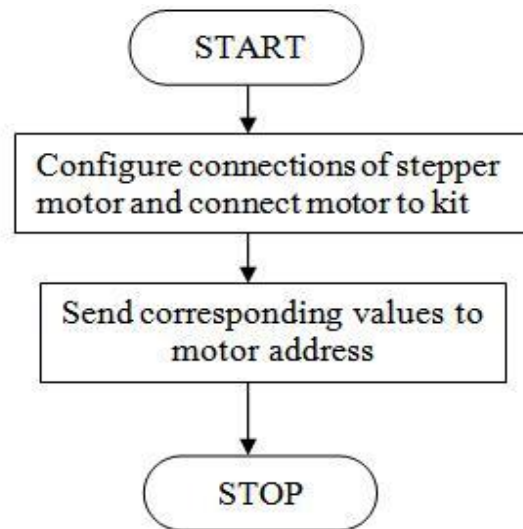
Program Description

Demonstrate “Stepper Motor “ rotating in clockwise and anticlockwise direction. A stepper motor or step motor or stepping motor is a brushless DC electric motor that divides a full rotation into a number of equal steps.

We consider 10H-16
 20H-32
 40H-64
 80H-128

STPPER MOTOR ALGORITHM

- STEP1: Configure the hardware connections of STEPPER Motor.
- STEP2: Connect the STEPPER Motor to J4 connector of 8051 SDK kit.
- STEP3: To run Stepper Motor in FARWORD Direction send 0x01, 0x02, 0x04, 0x08
 sequence of data one at a time to the STEPPER MOTOR ADDRESS
- STEP4: To run Stepper Motor in REVERSE Direction send 0x08, 0x04, 0x02, 0x01
 Sequence of data one at a time to the STEPPER MOTOR ADDRESS
- STEP5: End



PROGRAM FOR STEPPER MOTOR

```
#include<REGX51.H>

void delay(unsigned int d);

int main(void)
{
    while(1)
    {
        P2=0x01;
        delay(1000);
        P2=0x02;
        delay(1000);
        P2=0x04;
        delay(1000);
        P2=0x08;
        delay(1000);
    }
}

void delay(unsigned int d)
{
    unsigned int i,j;
    for(i=0;i<d;i++)
        for(j=0;j<101;j++);
}
```

Program Validation

Input :

1. Stepper(10,1) , Here count=10 and direction=1.
2. Stepper(10,0) , Here count=10 and direction=0.

Output :

1. When count=10 and direction=1 which means the stepper motor rotates for 10 counts in clockwise direction with a delay after each count.
2. Here count=10 and direction=0 which means stepper motor rotates for 10 counts in anticlockwise direction with a delay after each count.

Conclusion:

The program to demonstrate Stepper Motor using 8051 Microcontroller development kit executed successfully.

Program 6

LCD

Program Definition

To write a program to demonstrate LCD using 8051 Microcontroller development kit.

General Description:

The Liquid Crystal Display (LCD) is a low power device (microwatts). Now a days in most applications LCDs are using rather using of LED displays because of its specifications like low power consumption, ability to display numbers and special characters which are difficult to display with other displaying circuits and easy to program. An LCD requires an external or internal light source. Temperature range of LCD is 0°C to 60°C and lifetime is an area of concern, because LCDs can chemically degrade these are manufactured with liquid crystal material (normally organic for LCDs) that will flow like a liquid but whose molecular structure has some properties normally associated with solids.

LCDs are classified as

- Dynamic-scattering LCDs and
- Field-effect LCDs

Field-effect LCDs are normally used in such applications where source of energy is a prime factor (e.g., watches, portable instrumentation etc.). They absorb considerably less power than the light-scattering type. However, the cost for field-effect units is typically higher, and their height is limited to 2 inches. On the other hand, light-scattering units are available up to 8 inches in height. Field-effect LCD is used in the project for displaying the appropriate information.

RS (Command / Data):

This bit is to specify whether received byte is command or data. So that LCD can recognize the operation to be performed based on the bit status.

RS = 0 => Command
RS = 1 => Data

RW (Read / Write):

RW bit is to specify whether controller wants READ from LCD or WRITE to LCD. The READ operation here is just ACK bit to know whether LCD is free or not.

RW = 0 => Write
RW = 1 => Read

EN (Enable LCD):

EN bit is to ENABLE or DISABLE the LCD. When ever controller wants to write some thing into LCD or READ acknowledgment from LCD it needs to enable the LCD.

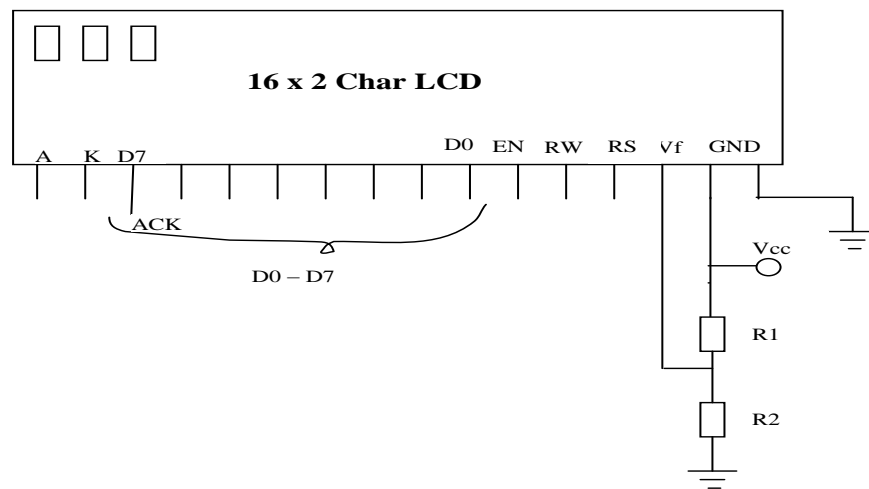
- EN = 0 => High Impedance
- EN = 1 => Low Impedance

ACK (LCD Ready):

ACK bit is to acknowledge the MCU that LCD is free so that it can send new command or data to be stored in its internal Ram locations

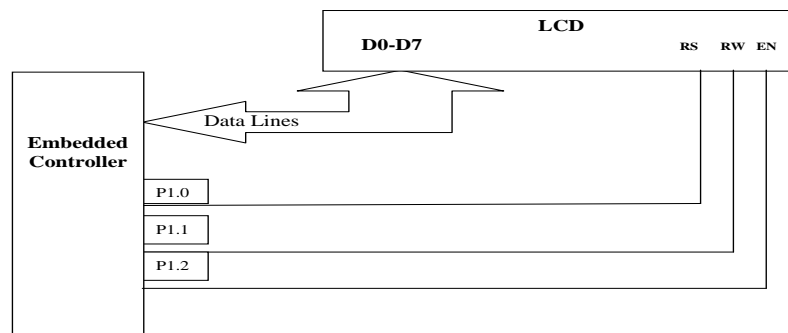
- ACK = 1 => Not ACK
- ACK = 0 => ACK

LCD diagram:



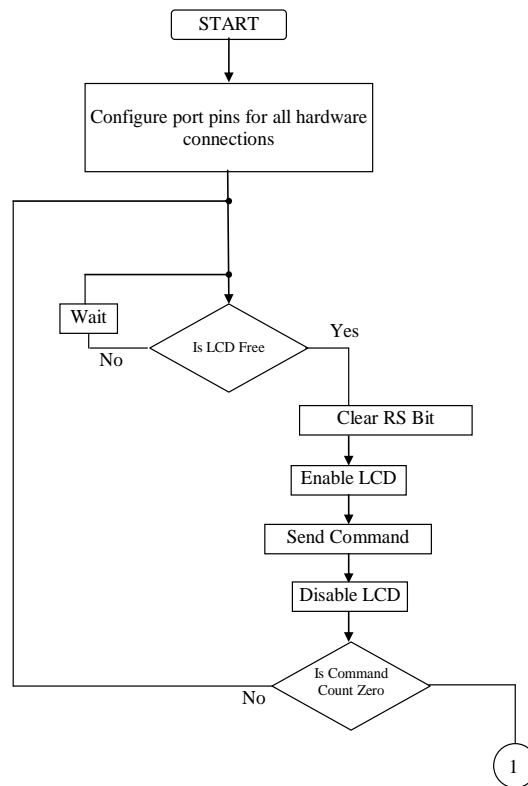
Block Diagram

Hardware connections:



CONTROLLER PINS	LCD PINS	PIN NAME WITH FEATURE
(P1.0)	4	RS (Control Pin)
(P1.1)	5	RW (Control pin)
(P1.2)	6	EN (Control pin)
Port 0	7 to 14	Data Port
40	15 & 2	Vcc
20	16 & 1	Gnd

FLOWCHART:



Program Description

In this program, we try to display characters in LCD display. 16x2 characters can be displayed. Here we need to initialize the LCD first using **lcdinit(void)**. When LCD is initialized ,we store all commands in an array **ledtable[]**. Now put commands on LCD using **put_com(ledtable[i])** then **lcd_check()** is called i.e., it checks LCD value clears all the contents of previous ones using ***lcd_display=0x00**. Then the characters to be displayed are provided using the function call **put_char('char',address)**.

Algorithm

1. Initialize Set of Commands.
2. Initialize LCD with proper set of Commands.
3. Write each Command to LCD Command Write Address.
4. Clear LCD for any prev data.
5. Write Data to LCD Data Write Address.

PROGRAM TO DISPLAY ROLL NO & NAME USING LCD:

```
#include<REGx51.H>
#define LCD P0
#define RS P3_4
#define EN P3_5
void lcdInit(void);
void putComL(unsigned char);
void putCharL(unsigned char);
void putStrL(unsigned char*,unsigned char);
void delay(unsigned int d);
int main(void)
{
    lcdInit();
    delay(1000);
    putStrL("ABDUL HADI",0x01);
    putStrL("032",0xc0);
    while(1);
}
void lcdInit(void)
{
    putComL(0x38);
    putComL(0x0c);
    putComL(0x06);
    putComL(0x01);
    putComL(0x80);
}
void putComL(unsigned char cmd)
{
    RS=0;
    LCD=cmd;
    EN=1;
    delay(100);
    EN=0;
}
void putCharL(unsigned char dat)
{
    RS=1;
    LCD=dat;
    EN=1;
```

```
        delay(100);
        EN=0;
    }
void putStrL(unsigned char *str,unsigned char cmd)
{
    unsigned char i=0;
    putComL(cmd);
    while(*str)
    {
        i++;
        if(i==17)
            putComL(0xc0);
            putCharL(*str++);
    }
}

void delay(unsigned int d)
{
    unsigned int i,j;
    for(i=0;i<d;i++)
        for(j=0;j<101;j++);
}
```

Program Validation

Input: PRATHYUSHA OK

Output:

The string displayed on LCD .



Conclusion:

The program to demonstrate LCD using 8051 microcontroller development kit is executed successful.

PROGRAM 7 KEYPAD

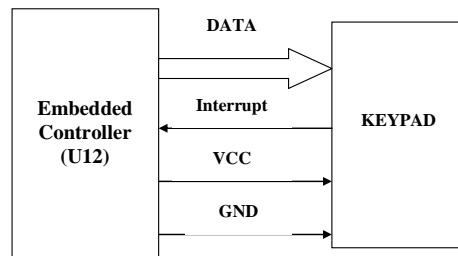
Program Definition

To write a program to demonstrate Keypad using 8051 Microcontroller development kit.
Keypad connector

In this product we have to design the 8 keys keypad directly connecting the 8 keys into 8 pins of Micro Controller. In this board we have to adding the extra feature like Matrix Keypad. In this 8 pins of AT89C51 are connected to the Keypad Connector. Matrix keypads such 4 by 4 can be connected directly to the connector. 5 Volt and Ground power lines are also available on the connector.

These two types of features are only working with any one external interrupt because of every key pressing its generating the interrupt.

Block diagram:



Hardware Connections:

CONTROLLER PINS	KEYPAD	PIN NAME	FEATURE
P0.7 TO P0.2 & P0.1,P0.0	SW1 TO SW6 & 8,9	DATA	DATA
P3.3(INT1)	interrupt	CLK	Interrupt
20		GND	
40		VCC	

Program Description

Keypad is a widely used input device with lots of application in our everyday life. In Toll Gate Indicator Signal Lights the above concept of pressing the key and glowing of LED is used for effective traffic divert.

In this program, we demonstrate keypad wherein when a key is pressed, corresponding LED glows. When a key is pressed, some interrupts is given, which is connected to a port in Microcontroller and so LED glows.

Val=getchar() , val stores the character which is pressed using the function getchar() IE-Interrupt enable, IP-Interrupt Priority, ITI-Interrupt Timer are initialized and **Led(val)** led function is called.

Algorithm

KEYBOARD ALGORITHM

STEP1: Verify the PS2 connector connections which pins are connected to

Microcontroller as data (P1.5) and clock (INT0) pins.

STEP2: Initialize the LCD by passing a set of COMMANDS

STEP3: Initialize the Keyboard by loading proper values in IE, IP registers and set the

ITx (Interrupt Type) bit of corresponding Interrupt

STEP4: Microcontroller waits until press a key. When a key is pressed Interrupts

will occur and key flag is set.

STEP5: Pressed key will generate a scan code and interrupt (key flag is set),

this scan code is compared with the Table what we are mentioned.

COMPARISON WITH CAPS LOCK KEY

STEP6: If the first scan code matches to the CAPS LOCK scan code then cap Flag will complement

STEP7: If the CAPS LOCK is already ON it will OFF (or) If the CAPS LOCK is OFF it will ON

STEP8: Next pressed key scan code will compare with the CAPS lock Table

STEP9: Display the corresponding key in LCD or SERIAL

COMPARISON WITH SHIFT KEY

STEP10: If the **first scan code** matches to the LSHIFT (left shift) or RSHIFT (right shift)

Scan code then set the shift Flag

STEP11: Now compare the second scan code if the **second scan code** is not equal to **8051-SDK**

Release scans code (0xF0) display the corresponding key value from the SHIFT Table

STEP12: if the **second scan code** is **equal** to Release scan code (0xF0) then break the Comparison and wait for press a key.

COMPARISON WITH NORMAL KEY

STEP13: If the **first scan code** not matches with the CAPS LOCK, LSHIFT (left shift)

And RSHIFT (right shift) scan codes then compare the first scan code with the

Unthrift (or) Normal Table values

STEP14: Display the corresponding key in LCD or SERIAL.

STEP15: End

Program:

```
#include<REGX51.H>
#include "lcd.h"
#include "delay.h"
unsigned char keyScan(void);
#define row1 P1_0
#define row2 P1_1
#define row3 P1_2
#define row4 P1_3
#define col1 P1_4
#define col2 P1_5
#define col3 P1_6
#define col4 P1_7

int main (void)
{
    unsigned char key;
    lcdInit();
    putStrL("KEY TEST",0X01);
    putComL(0XC0);

    while(1)
    {
        key=keyScan();
        putCharL(key);
        delay(1000);
    }
}
```

```
unsigned char keyScan(void)
{
    row1=row2=row3=row4=1;
    col1=col2=col3=col4=0;
    while(row1&row2&row3&row4);
    if(!row1)
    {
        col1=1;
        if(row1)
        {
            col1=0;
            while(!row1);
            return('1');
        }
        col1=0;
        col2=1;
        if(row1)
        {
            col2=0;
            while(!row1);
            return('2');
        }
        col2=0;
        col3=1;
        if(row1)
        {
            col3=0;
            while(!row1);
            return('3');
        }
        col3=0;
        col4=1;
        if(row1)
        {
            col4=0;
            while(!row1);
            return('^');
        }
        col4=0;
    }

    else if(!row2)
    {
        col1=1;
        if(row2){
            col1=0;
            while(!row2);
        }
    }
}
```



```
    return('4');
    }
    col1=0;
    col2=1;
    if(row2){
    col2=0;
    while(!row2);
    return('5');
    }
    col2=0;
    col3=1;
    if(row2){
    col3=0;
    while(!row2);
    return('6');
    }
    col3=0;
    col4=1;
    if(row2)
    {
    col4=0;while(!row2);return('V');}
    col4=0;
    }

else if(!row3)
    {
    col1=1;
    if(row3)
    {
    col1=0;
    while(!row3);
    return('7');
    }
    col1=0;
    col2=1;
    if(row3)
    {
    col2=0;
    while(!row3);
    return('8');
    }
    col2=0;
    col3=1;
    if(row3)
    {
    col3=0;
```

```
while(!row3);
return('9');
}
col3=0;
col4=1;
if(row3)
{
col4=0;
while(!row3);
return('M');
}
col4=0;
}
else if(!row4)
{
col1=1;
if(row4)
{
col1=0;
while(!row4);
return('*');
}
col1=0;
col2=1;
if(row4)
{
col2=0;
while(!row4);
return('0');
}
col2=0;
col3=1;
if(row4)
{
col3=0;
while(!row4);
return('#');
}
col3=0;
col4=0;
if(row4)
{
col4=0;
while(!row4);
return('E');
}
}
```

```
col4=0;
}
return 0;
}
```

Program Validation

Input :

key 28 is pressed , val=28 is passed to led()

Output:



LED 2 glows. Similarly for any key pressed its corresponding LED glows .

Conclusion :

Program to demonstrate Keypad using 8051 Microcontroller development kit was successfully executed.

Program 8 ELEVATOR CONTROLLER

Program Definition

To Write a program to demonstrate Elevator Controller using 8051 Microcontroller development kit.

Program Description

First we initialize variable `loc=0X01`, Then put 'loc' value in variable `prev`. The key which is pressed is put into 'loc' using `getchar8()`. Then we check whether 'loc' is greater or less than or equal to 'prev'.

- If (`loc<prev`) Elevator moves downwards .
- When (`loc==prev`) → the LED stops there at that position.
- If (`loc>prev`) Elevator moves upwards.

Algorithm

1. Initialize Elevator to Ground Floor (LED 1 Glows)
2. While true
3. Press Keypad switch to select particular floor
4. Rotate stepper motor clockwise or anti clockwise depending upon the key press
5. Glow the LED of particular floors as stepper motor rotates
6. Repeat steps 3, 4, and

ELEVATOR:

```
#include<REGX51.h>
void delay(unsigned int d);
code unsigned char seg7[10]={0XC0,0XF9,0XA4,0XB0,0X99,0X92,0X82,0XF8,0X80,0X90};
code unsigned char motoru[8]={0X01,0X03,0X02,0X06,0X04,0X0C,0X08,0X09};
code unsigned char motord[8]={0X09,0X08,0X0C,0X04,0X06,0X02,0X03,0X01};

int main (void)
{
  unsigned char loc,pre=0X01,seg=0,i,j;
  loc=pre;
  P3_5=1;
  P2=pre;
  P1=0XFF;
  P0=seg7[seg];
```

```
while(1)
{
P1=0XFF;
while(P1==0XFF);
loc=~P1;
if(loc<pre)
{
while(loc!=pre)
{
pre=pre>>1;
seg--;
for(j=0;j<4;j++)
for(i=0;i<8;i++)
{
P2=motoru[i];
delay(50);
}
P0=seg7[seg];
}
}
else if(loc>pre)
{
while(loc!=pre)
{
pre=pre<<1;
seg++;
for(j=0;j<4;j++)
for(i=0;i<8;i++)
{
P2=motord[i];
delay(50);
}
P0=seg7[seg];
}
}
}
}
void delay(unsigned int d)
{
unsigned int i,j;
for(i=0;i<=d;i++)
for(j=0;j<101;j++);
}
```

ELEVATOR OUTPUT:-



Conclusion:

The C program to demonstrate Elevator controller was executed successfully using 8051 Microcontroller development kit.

ARM

LIST OF LPC2148 PROGRAMS:

Understanding Real time concepts using any RTOS through demonstration of:

- Timing
- Multi-tasking
- Semaphores
- Message queues
- Round Robin Task scheduling
- Preemptive Priority based task scheduling
- Priority inversion
- Signals
- Interrupt service routines

INTRODUCTION:

The LPC2141/2/4/6/8 microcontrollers are based on a 32/16 bit ARM7TDMI-S CPU with real-time emulation and embedded trace support, that combines the microcontroller with embedded high speed flash memory ranging from 32 k B to 512 k B. A 128-bit wide memory interface and unique accelerator architecture enable 32-bit code execution at the maximum clock rate. For critical code size applications, the alternative 16-bit Thumb mode reduces code by more than 30 % with minimal performance penalty.

Due to their tiny size and low power consumption, LPC2141/2/4/6/8 are ideal for applications where miniaturization is a key requirement, such as access control and point-of-sale. A blend of serial communications interfaces ranging from a USB 2.0 Full Speed device, multiple UARTs, SPI, SSP to I2Cs and on-chip SRAM of 8 kB up to 40 kB, make these devices very well suited for communication gateways and protocol converters, soft modems, voice recognition and low end imaging, providing both large buffer size and high processing power. Various 32-bit timers, single or dual IO-bit ADC(s), IO-bit DAC, PWM channels and 45 fast GPIO lines with up to nine edge or level sensitive external interrupt pins make these microcontrollers particularly suitable for industrial control and medical systems.

FEATURES

- 16/32-bit ARM7TDMI-S microcontroller in a tiny LQFP64 package.
- 8 to 40 kB of on-chip static RAM and 32 to 512 kB of on-chip flash program memory. 128 bit wide interface/accelerator enables high speed 60 MHz operation.
- In-System/In-Application Programming (ISP/IAP) via on-chip boot-loader software. Single flash sector or full chip erase in 400 ms and programming of 256 bytes in 1 ms.
- Embedded ICE RT and Embedded Trace interfaces offer real-time debugging with the on-chip Real Monitor software and high speed tracing of instruction execution.
- USB 2.0 Full Speed compliant Device Controller with 2 kB of endpoint RAM.
- In addition, the LPC2146/8 provide 8 kB of on-chip RAM accessible to USB by DMA.
- One or two (LPC2141/2 vs. LPC2144/6/8) 10-bit A/D converters provide a total of 6/14 analog inputs, with conversion times as low as 2.44 μ s per channel.
- Single IO-bit D/A converter provides variable analog output.
- Two 32-bit timers/external event counters (with four captures and four compare Channels each), PWM unit (six outputs) and watchdog.
- Low power real-time clock with independent power and dedicated 32 kHz clock input.
- Multiple serial interfaces including two UARTs (16C550), two Fast I2C-bus (400 k bit/s), SPI and SSP with buffering and variable data length capabilities.
- Vectored interrupt controller with configurable priorities and vector addresses.
- Up to 45 of 5 V tolerant fast general purpose I/O pins in a tiny LQFP64 package.
- Up to nine edge or level sensitive external interrupt pins available.

Program 9 TIMING

9) AIM:

Program to demonstrate Timing in RTOS Description:

Timing systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.). The time slices among tasks are fixed and shared accordingly.

ALGORITHM:

STEP 1: Create 8 different tasks.

STEP 2: Provide switching between different tasks by introducing delays.

STEP 3: Synchronize all tasks.

STEP4: Destroy all the tasks once work is completed.

Program:

Demonstrate the TIMING concept of real time application using RTOS on ARM microcontroller kit

```
#include<RTL.h>
#include<LPC214X.H>
OS_TID id1,id2,id3,id4,id5,id6,id7,id8;
__task void task1(void);
__task void task2(void);
__task void task3(void);
__task void task4(void);
__task void task5(void);
__task void task6(void);
__task void task7(void);
__task void task8(void);
__task void task1(void)
{
unsigned int count=0;
IO1DIR=0X40ff0000;
IO1SET=1<<30;
IO1CLR=0X00ff0000;
id1=os_tsk_self();
os_tsk_prio_self(1);
id2=os_tsk_create(task2,1);
```

```
id3=os_tsk_create(task3,1);
id4=os_tsk_create(task4,1);
id5=os_tsk_create(task5,1);
id6=os_tsk_create(task6,1);
id7=os_tsk_create(task7,1);
id8=os_tsk_create(task8,1);
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(100);
}
}
```

```
__task void task2(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(50);
}
}
```

```
__task void task3(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(70);
}
}
```

```
__task void task4(void)
```

```
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(130);
}
}
```

```
__task void task5(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(110);
}
}
```

```
__task void task6(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(90);
}
}
```

```
__task void task7(void)
{
unsigned int count=0;
while(1)
{
```

```

if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(120);
}
}

```

```

__task void task8(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(20);
}
}

```

```

int main(void)
{
os_sys_init(task1);
}

```

Output: TIMING

The screenshot shows the 'Active Tasks' window in the RTX Kernel environment. It displays a table with the following data:

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
1	task1	1	WAIT_DLY	85			32%
2	task2	1	WAIT_DLY	35			32%
3	task3	1	WAIT_DLY	45			32%
4	task4	1	WAIT_DLY	5			32%
5	task5	1	WAIT_DLY	35			32%
6	task6	1	WAIT_DLY	25			32%
7	task7	1	WAIT_DLY	85			32%
8	task8	1	WAIT_DLY	5			32%
255	os_idle_demon	0	RUNNING				0%

Program 10 MULTI TASKING

10) AIM:

Program to demonstrate Timing in RTOS Description:

Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.). The time slices among tasks are fixed and shared accordingly

ALGORITHM:

STEP I: Create 8 different tasks.

STEP 2: Provide switching between different tasks by introducing delays.

STEP 3: Synchronize all tasks.

STEP 4: Destroy all the tasks once work is completed.

Program:

To Demonstrate the Multi Tasking concept of real time application using RTOS on ARM microcontroller kit

```
#include<RTL.h>
#include<LPC214X.H>
OS_TID id1,id2,id3,id4,id5,id6,id7,id8;
__task void task1(void);
__task void task2(void);
__task void task3(void);
__task void task4(void);
__task void task5(void);
__task void task6(void);
__task void task7(void);
__task void task8(void);
__task void task1(void)
{
unsigned int count=0;
IO1DIR=0X40ff0000;
IO1SET=1<<30;
IO1CLR=0X00ff0000;
id1=os_tsk_self( );
os_tsk_prio_self(1);
id2=os_tsk_create(task2,1);
```

```
id3=os_tsk_create(task3,1);
id4=os_tsk_create(task4,1);
id5=os_tsk_create(task5,1);
id6=os_tsk_create(task6,1);
id7=os_tsk_create(task7,1);
id8=os_tsk_create(task8,1);
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(100);
}
}
```

```
__task void task2(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(50);
}
}
```

```
__task void task3(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(70);
}
}
```

```
__task void task4(void)
{
```

```
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(130);
}
}
```

```
__task void task5(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(110);
}
}
```

```
__task void task6(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(90);
}
}
```

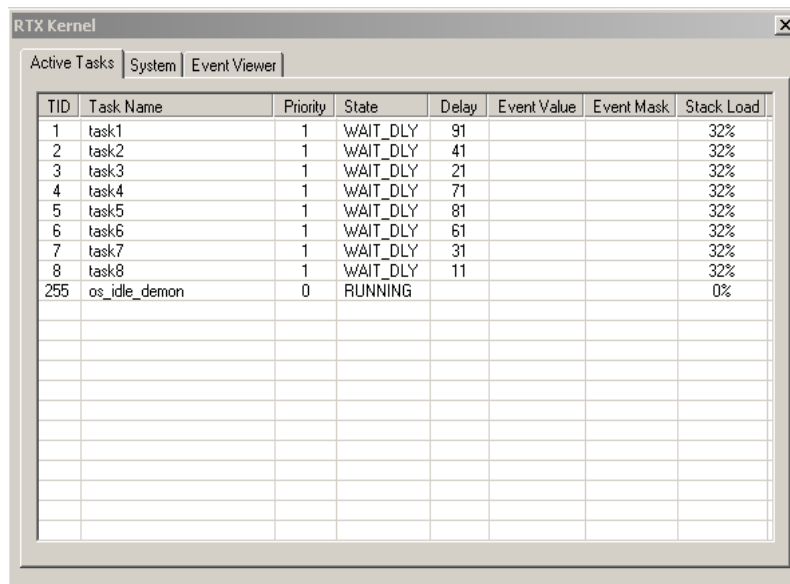
```
__task void task7(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
```



```
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(120);
}
}
```

```
__task void task8(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(20);
}
}
int main(void)
{
os_sys_init(task1);
}
```

Output: Multi Tasking



The screenshot shows the 'Active Tasks' window of the RTX Kernel. It contains a table with the following data:

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
1	task1	1	WAIT_DLY	91			32%
2	task2	1	WAIT_DLY	41			32%
3	task3	1	WAIT_DLY	21			32%
4	task4	1	WAIT_DLY	71			32%
5	task5	1	WAIT_DLY	81			32%
6	task6	1	WAIT_DLY	61			32%
7	task7	1	WAIT_DLY	31			32%
8	task8	1	WAIT_DLY	11			32%
255	os_idle_demon	0	RUNNING				0%

Program 11 SEMAPHORE

11) AIM: Program to demonstrate Semaphores

Description:

When the critical section is longer than a few source code lines or involves lengthy looping, an embedded/real-time algorithm must resort to using mechanisms identical or similar to those available on general-purpose operating systems, such as semaphores and OS-supervised inter process messaging. Such mechanisms involve system calls, and usually invoke the OS's dispatcher code on exit, so they typically take hundreds of CPU instructions to execute, while masking interrupts may take as few as one instruction on some processors. But for longer critical sections, there may be no choice; interrupts cannot be masked for long periods without increasing the system's interrupt latency.

A binary semaphore is either locked or unlocked. When it is locked, tasks must wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. There are several well-known problems with semaphore based designs such as priority inversion and deadlocks.

SEMAPHORE FUNCTIONS:

1) `sem _ acquire` – Acquire .a semaphore

Description

boo! `sem_acquire(resource $sem_identifier)`

`sem_acquire()` blocks (if necessary) until the semaphore can be acquired. A process attempting to acquire a semaphore which it has already acquired will block forever if acquiring the semaphore would cause its maximum number of semaphore to be exceeded.

After processing a request, any semaphores acquired by the process but not explicitly released will be released automatically and a warning will be generated.

Parameters

`sem_identifier`

`sem_identifier` is a semaphore resource, obtained from `sem_get()`.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

2) `sem_get` - Get a semaphore id

Description

Resource `sem_get` (int \$key[, int \$max_acquire [, int \$perm[, int \$auto_release]]])

`sem_get()` returns ,mid that can be used to access the System V semaphore with the given key.

A second call to `sem_get()` for the same key will return a different semaphore identifier, but both identifiers access the same underlying semaphore.

Parameters

key

max_acquire

The number of processes that can acquire the semaphore simultaneously is set to max_acquire (defaults to 1).

Perm

The semaphore permissions.Defaults to 0666. Actually this value is set only if the process finds it is the only process currently attached to the semaphore.

auto release

Specifies if the semaphore should be automatically released on request shutdown.

Return Values

Returns a positive semaphore identifier on success, or **FALSE** on error.

3) `sem_release` - Release a semaphore

Description

boo! `sem_release`(resource \$sem_identifier)

`sem_release()` releases the semaphore if it is currently acquired by the calling process, otherwise a warning is generated.

After releasing the semaphore, `sem_acquire()` may be called to re-acquire it.

Parameters

sem_identifier

A Semaphore resource handle as returned by sem_get().

Return Values

Returns **TRUE** on success or **FALSE** on failure.

4) sem_remove - Remove a semaphore

Description

bool sem_remove (resource \$sem_id identifier)

sem_remove() removes the given semaphore.

After removing the semaphore, it is no more accessible.

Parameters

sem_identifier

A semaphore resource identifier as returned by sem_get().

Return Values

Returns **TRUE** on success or **FALSE** on failure.

ALGORITHM:

Step1: Create 2 different tasks .

STEP 2: Provide switching between different tasks by introducing delays.

STEP 3: Create a semaphore

STEP4: one task acquires semaphore other task waits

STEP5: once first task releases semaphore then it is acquired by another.

STEP6: Destroy all the tasks and semaphore once work is completed.

Program:

Demonstrate the SEMAPHORE concept of real time application using RTOS on ARM microcontroller kit

```
#include<RTL.H>
#include<LPC214X.H>
#include "serial0.h"
extern void Init_Serial(void);
OS_TID tsk1,tsk2;
```

```
OS_SEM semaphore1;

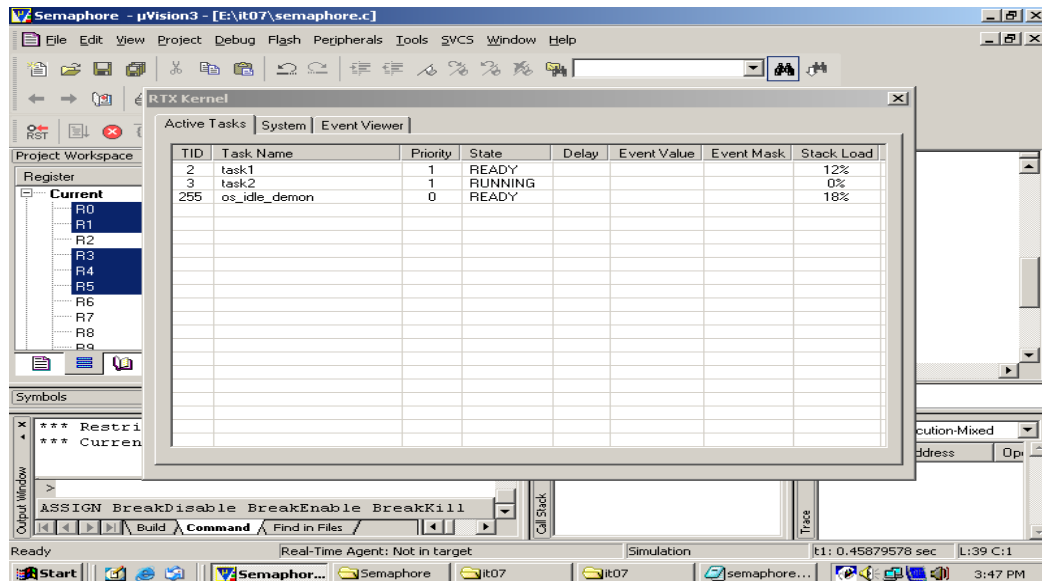
__task void task1(void)
{
OS_RESULT ret;
while(1)
{
os_dly_wait(3);
ret=os_sem_wait(semaphore1,1);
if(ret!=OS_R_TMO)
{
putStrS0("\n\rtask1");
os_sem_send(semaphore1);
}
}
}

__task void task2(void)
{
while(1)
{
os_sem_wait(semaphore1,0xffff);
putStrS0("\n\rtask2");
os_sem_send(semaphore1);
}
}

__task void init(void)
{
InitSerial0(9600);
os_sem_init(semaphore1,1);
tsk1=os_tsk_create(task1,1);
tsk2=os_tsk_create(task2,0);
os_tsk_delete_self();
}
int main(void)
{
os_sys_init(init);

}
}
```

OUTPUT : SEMAPHORE



Program 12 MESSAGE QUEUES

12) AIM:

Program to demonstrate Message Queues.

DESCRIPTION:

Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them.

Most message queues have set limits on the size of data that can be transmitted in a single message. Those that do not have such limits are known as mailboxes.

Many implementations of message queues function internally: within an operating system or within an application. Such queues exist for the purposes of that system only.

1) **msg_get_queue**- Create or attach to a message queue

Description

Resource **msg_get_queue** (int \$key[, int \$perms])

msg_get_queue() returns an id that can be used to access the System V message queue with the given key. The first call creates the message queue with the optional perms. A second call to **msg_get_queueO()** for the same key will return a different message queue identifier, but both identifiers access the same underlying message queue.

Parameters

Key

Message queue numeric ID

Perms

Queue permissions. Default to 0666. If the message queue already exists, the perms will be ignored.

Return Values

Returns a resource handle that can be used to access the System V Message queue

2) **msg_send** - Send a message to a message queue

Description

bool **msg_send** (resource \$queue, int \$msgtype , mixed \$message [, bool \$serialize [, bool \$blocking [, int&\$errorcode]]])

msg_send()sends a message of type msgtype (which MUST be greater than 0) to the message queue specified by queue

Parameters

queue.

msgtype

message

serialize

The optional serialize controls how the message is sent. serialize defaults to **TRUE** which means that the message is serialized using the same mechanism as the session module before being sent to the queue. This allows complex arrays and objects to be sent to other PHP scripts, or if you are using the WDDX serializer, to any WDDX compatible client.

Blocking

If the message is too large to fit in the queue, your script will wait until another process reads messages from the queue and frees enough space for your message to be sent. This is called blocking; you can prevent blocking by setting the optional blocking parameter to **FALSE**, in which case **msg_send()** will immediately return **FALSE** if the message is too big for the queue, and set the optional errorcode to **MSG_EAGAIN**, indicating that you should try to send your message again a little later on.

Errorcode

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Upon successful completion the message queue data structure is updated as follows: msg_lpid is set to the process-ID of the calling process, msg_qnum is incremented by 1 and msg_stime is set to the current time.

3) msg_receive - Receive a message from a message queue

Description

bool **msg_receive**(resource \$queue , int \$desiredmsgtype , int&\$msgtype , int \$maxsize , mixed &\$message [, bool \$unserialize [, int \$flag;s [, int&\$errorcode]]])

`msg_receive()` will receive the first message from the specified queue of the type specified by `desiredmsgtype`.

-Parameters

queue

desired msgtype

If `desired msgtype` is 0, the message from the front of the queue is returned. If `desired msgtype` is greater than 0, then the first message of that type is returned. If `desired msgtype` is less than 0, the first message on the queue with the lowest type less than or equal to the absolute value of `desired msgtype` will be read. If no messages match the criteria, your script will wait until a suitable message arrives on the queue. You can prevent the script from blocking by specifying

MSG_IPC_NOWAIT in the `flags` parameter.

Msgtype

The type of the message that was received will be stored in this parameter.

Maxsize

The maximum size of message to be accepted is specified by the `maxsize`; if the message in the queue is larger than this size the function will fail (unless you set `flags` as described below).

Message

The received message will be stored in `message`, unless there were errors receiving the message.

Unserialize

`unserialize` defaults to **TRUE**; if it is set to **TRUE**, the message is treated as though it was serialized using the same mechanism as the `session` module. The message will be unserialized and then returned to your script. This allows you to easily receive arrays or complex object structures from other PHP scripts, or if you are using the WDDX serializer, from any WDDX compatible source.

If `unserialize` is **FALSE**, the message will be returned as a binary-safe string.

Flags

The optional `flags` allows you to pass flags to the low-level `msg_rcv` system call. It defaults to 0, but you may specify one or more of the following values (by adding or ORing them together).

Flag values for `msg_receive`

'will truncate the message to maxsize and will not signal

Error code

If the function fails, the optional errorcode will be set to the value of the system errno variable.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Upon successful completion the message queue data structure is updated as follows: msg_lrpId is set to the process-ID of the calling process, msg_qnum is decremented by 1 and msg_rtime is set to the current time.

4) msg_remove_queue - Destroy a message queue

Description

bool msg_remove_queue(resource \$queue)

msg_remove_queue() destroys the message queue specified by the queue. Only use this function when all processes have finished working with the message queue and you need to release the system resources held by it.

Parameters

queue

Message queue resource handle

Return Values

Returns TRUE on success or **FALSE** on failure.

ALGORITHM:

STEP 1 : Create two tasks .

STEP 2: one task creates a message queue and sends message into through serial port.

STEP 3: Another receives message from message queue through serial port.

STEP 4: Destroy Message queue

STEP 5: Destroy all tasks.

Demonstrate the Message Queues concept of real time application using RTOS on ARM microcontroller kit.**Program**

```
#include <RTL.h>
#include <LPC214X.H>
#include "serial0.h"
#include "lcd.h"

OS_TID tsk1;
OS_TID tsk2;

unsigned char MSG[16];

os_mbx_declare (MsgBox,16);
_declare_box (mpool,sizeof(MSG),16);

__task void send_task (void);
__task void rec_task (void);

__task void send_task (void) {

    tsk1 = os_tsk_self ();
    tsk2 = os_tsk_create (rec_task, 0);
    os_mbx_init (MsgBox, sizeof(MsgBox));
    os_dly_wait (5);
    while(1)
    {
        putStrS0("\n\rENTER the MSG: ");
        getStrS0(MSG);
        os_mbx_send (MsgBox, MSG, 0xffff);
        putStrS0("\n\rSending MSG...");
        os_dly_wait (100);
    }
    // os_tsk_delete_self ();
}

__task void rec_task (void) {
    unsigned char *rptr;

    while(1)
    {
        os_mbx_wait (MsgBox,(void **)&rptr, 0xffff);
        putStrL("Received MSG: ",0x01);
        putStrL(rptr,0xC0);
    }
}
```

```
_free_box (mpool, rptr);
}
}
int main (void) {
InitSerial0 (9600);
lcdInit();
putCharS0(0x0C);
_init_box (mpool, sizeof(mpool),
          sizeof(MSG));
os_sys_init (send_task);
}
```

Output : MESSAGE QUEUES.



Program 13

ROUND ROBIN

13) AIM:

Program to demonstrate Round-Robin Task Scheduling

DESCRIPTION:

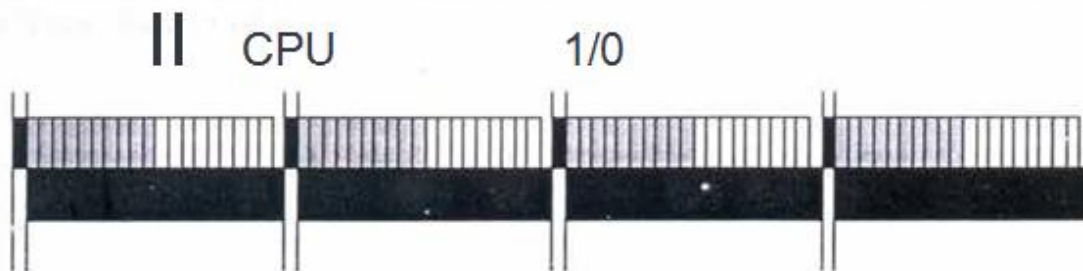
Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.

Round Robin calls for the distribution of the processing time equitably among all processes requesting the processor. Run process for one time slice, then move to back of queue. Each process gets equal share of the CPU. Most systems use some variant of this.

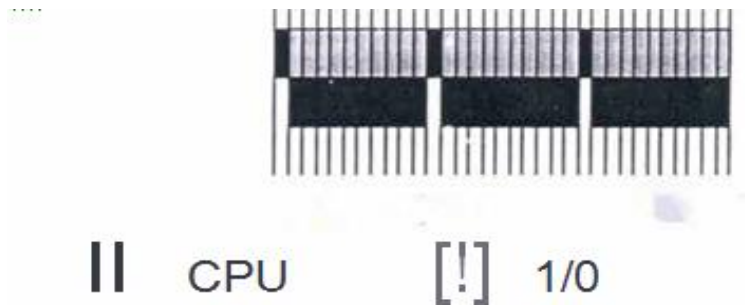
Choosing Time Slice

What happens if the time slices isn't chosen carefully?

For example, consider two processes, one doing 1 ms computation followed by 10 ms I/O, the other doing all computation. Suppose we use 20 ms time slice and round-robin scheduling: I/O process runs at 1/21 speed, I/O devices are only utilized 10/21 of time.



Suppose we use 1 ms time slice: then compute-bound process gets interrupted 9 times unnecessarily before I/O-bound process is runnable

**ALGORITHM:**

Step1: Create 8 different tasks.

STEP 2: Provide switching between different tasks through CPU time slicing.

STEP 3: Synchronize all tasks.

STEP4: Destroy all the tasks once work is completed.

Program:

Demonstrate the Round Robin task scheduling using RTOS on ARM microcontroller kit

```
#include<RTL.h>
#include<LPC214X.H>
OS_TID id1,id2,id3,id4,id5,id6,id7,id8;
__task void task1(void);
__task void task2(void);
__task void task3(void);
__task void task4(void);
__task void task5(void);
__task void task6(void);
__task void task7(void);
__task void task8(void);
__task void task1(void)
{
  unsigned int count=0;
  IO1DIR=0X40ff0000;
  IO1SET=1<<30;
  IO1CLR=0X00ff0000;
  id1=os_tsk_self();
  os_tsk_prio_self(2);
  id2=os_tsk_create(task2,2);
  id3=os_tsk_create(task3,2);
  id4=os_tsk_create(task4,2);
  id5=os_tsk_create(task5,2);
  id6=os_tsk_create(task6,2);
```

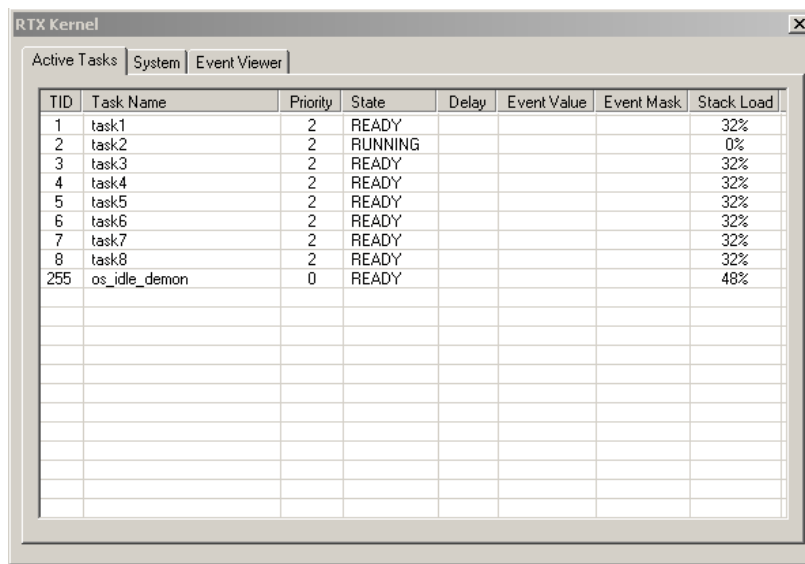
```
id7=os_tsk_create(task7,2);
id8=os_tsk_create(task8,2);
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
}
}
__task void task2(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
}
}
__task void task3(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
}
}
__task void task4(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
}
}
```

```
}
__task void task5(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
  }
}
__task void task6(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
  }
}
__task void task7(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
  }
}
__task void task8(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
```



```
count++;  
}  
}  
int main(void)  
{  
os_sys_init(task1);  
}
```

Output: Round Robin



The screenshot shows the 'Active Tasks' window of the RTX Kernel. It contains a table with the following data:

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
1	task1	2	READY				32%
2	task2	2	RUNNING				0%
3	task3	2	READY				32%
4	task4	2	READY				32%
5	task5	2	READY				32%
6	task6	2	READY				32%
7	task7	2	READY				32%
8	task8	2	READY				32%
255	os_idle_demon	0	READY				48%

Program 14 PRE-EMPTIVE PRIORITY

14) AIM:

Program to demonstrate Preemptive Priority based Task Scheduling

DESCRIPTION:

Run highest-priority processes first, use round-robin among processes of equal priority. Re-insert process in run queue behind all processes of greater or equal priority.

- Allows CPU to be given preferentially to important processes.
- Scheduler adjusts dispatcher priorities to achieve the desired overall priorities for the processes, e.g. one process gets 90% of the CPU.

Comments: In priority scheduling, processes are allocated to the CPU on the basis of an externally assigned priority. The key to the performance of priority scheduling is in

Choosing priorities for the processes.

The O/S assigns a fixed priority rank to every process, and the scheduler arranges the

Processes in the ready queue in order of their priority. Lower priority processes get

Interrupted by incoming higher priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.
- Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.

- Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU time.

ALGORITHM:

STEP 1: Create 8 different tasks.

STEP 2: Assign priority to all the tasks

STEP 3: Provide switching between different tasks through CPU time slicing.

STEP 4: execute all tasks on their order of priority

STEP 5: Destroy all the tasks once work is completed

Demonstrate the Pre-emptive priority based task scheduling using RTOS on ARM microcontroller kit

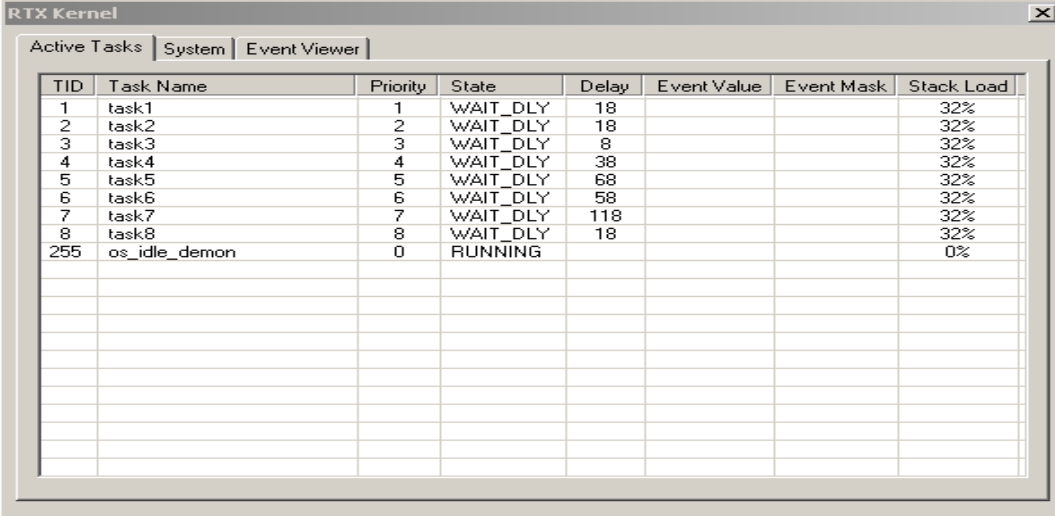
Program:

```
#include<RTL.h>
#include<LPC214X.H>
OS_TID id1,id2,id3,id4,id5,id6,id7,id8;
__task void task1(void);
__task void task2(void);
__task void task3(void);
__task void task4(void);
__task void task5(void);
__task void task6(void);
__task void task7(void);
__task void task8(void);
__task void task1(void)
{
unsigned int count=0;
IO1DIR=0X40ff0000;
IO1SET=1<<30;
IO1CLR=0X00ff0000;
id1=os_tsk_self();
os_tsk_prio_self(1);
id2=os_tsk_create(task2,2);
id3=os_tsk_create(task3,3);
id4=os_tsk_create(task4,4);
id5=os_tsk_create(task5,5);
id6=os_tsk_create(task6,6);
id7=os_tsk_create(task7,7);
id8=os_tsk_create(task8,8);
while(1)
{
```

```
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(100);
}
}
__task void task2(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(50);
}
}
__task void task3(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(70);
}
}
__task void task4(void)
{
unsigned int count=0;
while(1)
{
if(count%2==1)
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(130);
}
}
```

```
}
__task void task5(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
    os_dly_wait(110);
  }
}
__task void task6(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
    os_dly_wait(90);
  }
}
__task void task7(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
    IO1SET=0X00010000;
    else
    IO1CLR=0X00010000;
    count++;
    os_dly_wait(120);
  }
}
__task void task8(void)
{
  unsigned int count=0;
  while(1)
  {
    if(count%2==1)
```

```
IO1SET=0X00010000;
else
IO1CLR=0X00010000;
count++;
os_dly_wait(20);
}
}
int main(void)
{
os_sys_init(task1);
}
```

Output: PRE-EMPTIVE PRIORITY

The screenshot shows the 'RTX Kernel' window with the 'Active Tasks' tab selected. It displays a table with the following data:

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
1	task1	1	WAIT_DLY	18			32%
2	task2	2	WAIT_DLY	18			32%
3	task3	3	WAIT_DLY	8			32%
4	task4	4	WAIT_DLY	38			32%
5	task5	5	WAIT_DLY	68			32%
6	task6	6	WAIT_DLY	58			32%
7	task7	7	WAIT_DLY	118			32%
8	task8	8	WAIT_DLY	18			32%
255	os_idle_demon	0	RUNNING				0%

Program 15 **PRIORITY INVERSION**

15) AIM:

Program to demonstrate Priority Inversion

DESCRIPTION:

In scheduling, priority inversion is the scenario where a low priority task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task has released the resource, effectively "inverting" the relative priorities of the two tasks. If some other medium priority task, one that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both the low priority task and the high priority task.

In some cases, priority inversion can occur without causing immediate harm-the delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high priority task is left starved of the resources, it might lead to a system malfunction or the triggering of pre-defined corrective measures, such as a watch dog timer resetting the entire system. The trouble experienced by the Mars lander "Mars Pathfinder" is a classic example of problems caused by priority inversion in real-time systems.

Priority inversion can also reduce the perceived performance of the system. Low priority tasks usually have a low priority because it is not important for them to finish promptly

(For example, they might be a batch job or another non-interactive activity). Similarly, a high priority task has a high priority because it is more likely to be subject to strict time constraints-it may be providing data to an interactive user, or acting subject to real time response guarantees. Because priority inversion results in the execution of the low priority task blocking the high priority task, it can lead to reduced system responsiveness, or even the violation of response time guarantees.

ALGORITHM:

STEP1: Create 8 different tasks .

STEP 2: Assign priority to all the tasks

STEP 3: Provide switching between different tasks through CPU time slicing.

STEP 4: execute all tasks on their order of priority

STEP 5: Lower priority task will be executed while higher priority waits.

STEP 6: Destroy all the tasks once work is completed.

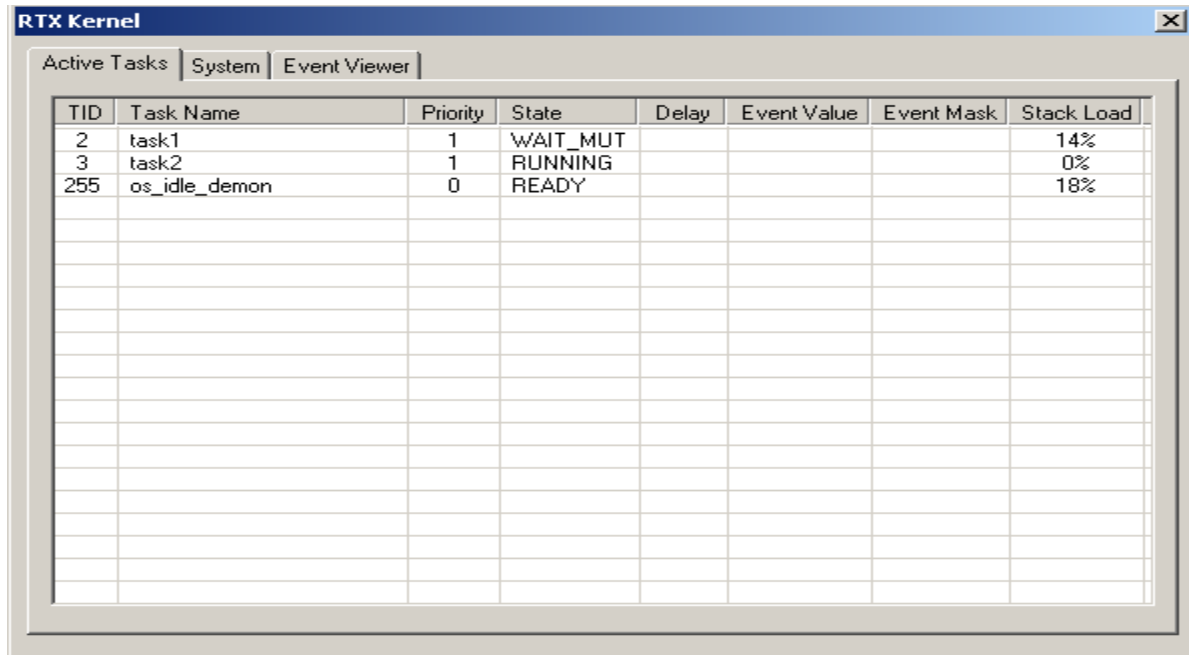
Demonstrate the Priority Inversion based task scheduling using RTOS on ARM microcontroller kit

```
#include<RTL.H>
#include<lpc214X.H>
#include "serial0.h"
extern void Init_Serial(void);
__task void task1(void);
__task void task2(void);
OS_TID tsk1,tsk2;
OS_MUT mutex1;

__task void task1(void){
while(1)
{
putStrS0("\n\r tsk1");
os_dly_wait(100);
os_mut_wait(mutex1,0xffff);
os_mut_release(mutex1);
}}

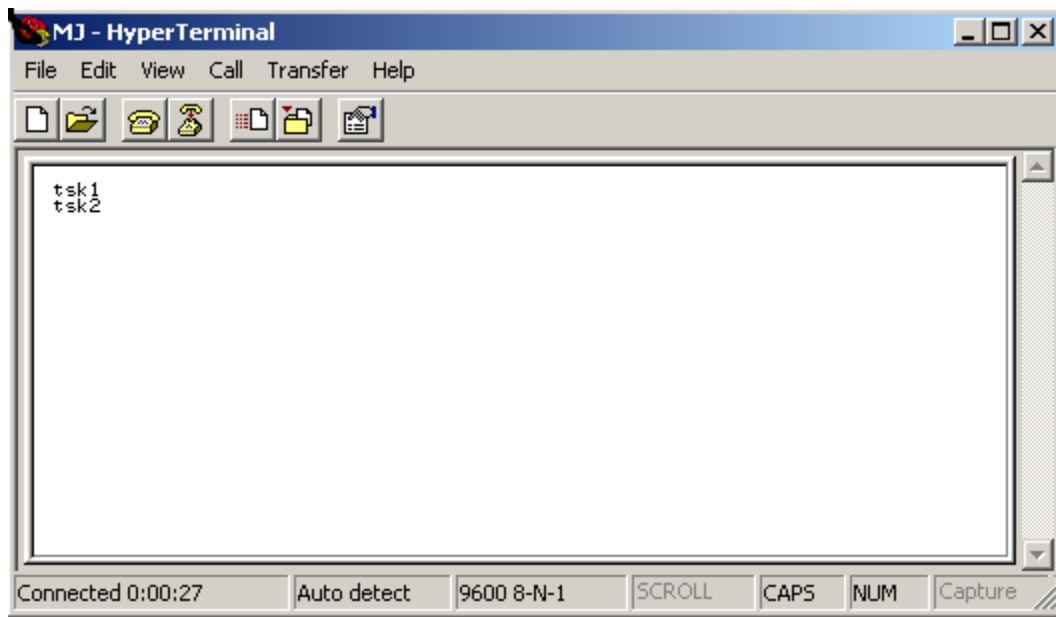
__task void task2(void){
while(1)
{
os_mut_wait(mutex1,0xffff);
putStrS0("\n\r tsk2");
getCharS0();
os_mut_release(mutex1);
}}

__task void init(void)
{
InitSerial0(9600);
os_mut_init(mutex1);
tsk1=os_tsk_create(task1,0);
tsk2=os_tsk_create(task2,1);
os_tsk_delete_self();
}
int main(void)
{
os_sys_init(init);
}
```


Output : PRIORITY INVERSION

The screenshot shows the 'Active Tasks' window in the RTX Kernel. It contains a table with the following data:

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
2	task1	1	WAIT_MUT				14%
3	task2	1	RUNNING				0%
255	os_idle_demon	0	READY				18%



Program 16

COMMUNICATIONS-RS 232

FEATURES:

- Operates from a single 5V Power Supply with 1.0uF Charge-Pump Capacitors
- Operates up to 120 k bit/s
- Two Drivers and Two Receivers
- ± 30 V Input Levels
- Low Supply Current . . . 8 mA Typical

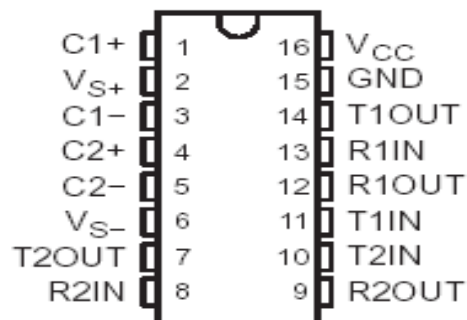
Upgrade with Improved ESD (15kV HBM) and 0.1uF Charge-Pump Capacitors is available With the MAX202.

Applications-- TIA/EIA-232-F, Battery-Powered Systems, Terminals, Modems, and Computers

DESCRIPTION:

The MAX232 is a dual driver/receiver that includes a capacitive voltage generator to supply TIA/EIA-232-F voltage levels from a single 5V supply. Each receiver converts TIA/EIA-232-F inputs to 5V TTL/CMOS levels. These receivers have a typical threshold of 1.3V, a typical hysteresis of 0.5 V, and can accept up to 30V inputs. Each driver converts TTL/CMOS input levels into TIA/EIA-232-F levels.

PIN DIAGRAM OF MAX232



FUNCTION TABLE

EACH DRIVER	
INPUT TIN	OUTPUT TOUT
L	H
H	L

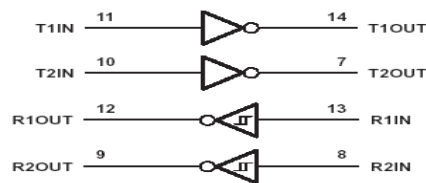
H = high level, L = low level

EACH RECEIVER	
INPUT RIN	OUTPUT ROUT
L	H
H	L

H = high level, L = low level

LOGIC DIAGRAM

(POSITIVE LOGIC)



RECOMMENDED OPERATING CONDITIONS

PARAMETER	MIN	NOR	MAX	UNIT
VCC Supply voltage	4.5	5	5.5	V
VIH High-level input voltage (T1IN,T2IN)	2			V
VIL Low-level input voltage (T1IN, T2IN)			0.8	V
R1IN, R2IN Receiver input voltage		□30		V
TA Operating free-air temperature	0		70	□□□□□C

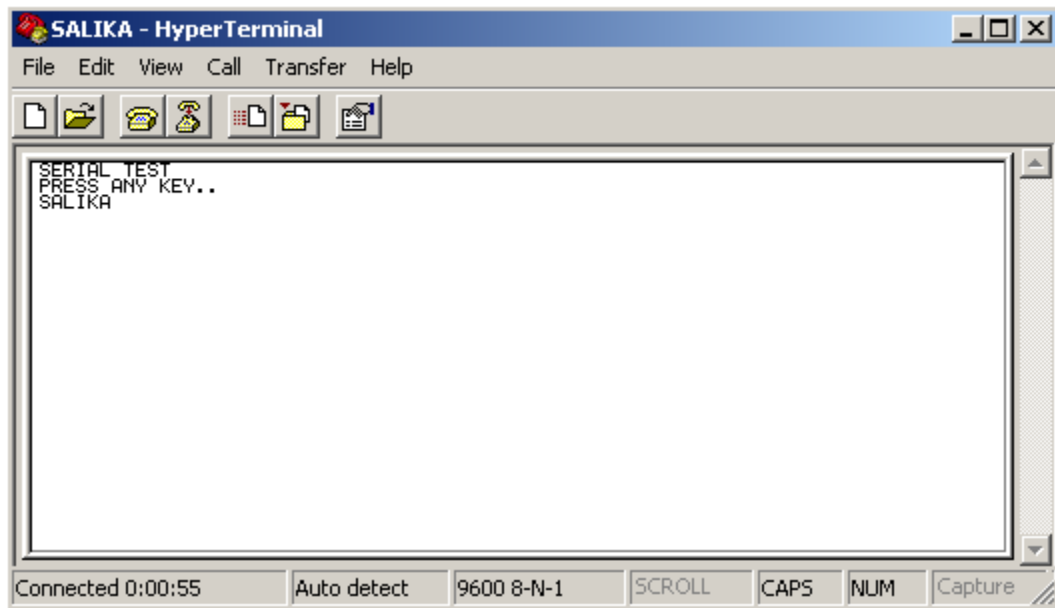
--	--	--

RS232**Program for RS232 using RTOS**

```
#include<REGX51.H>
void serInit(void);
void putChars(unsigned char);
unsigned char getChars(void);
void putStrs(unsigned char *);
int main(void)
{
    unsigned char ch;
    serInit();
    putStrs("SERIAL TEST\n\r");
    putStrs("PRESS ANY KEY.. \n\r");
    while(1)
    {
        ch=getChars();
        putChars(ch);
    }
}
void serInit(void)
{
    SCON=0X50;
    TMOD=0X20;
    TH1=0Xfd;
    TL1=0Xfd;
    TR1=1;
}
void putChars(unsigned char byte)
{
    SBUF=byte;
    while(!TI);
    TI=0;
}
void putStrs(unsigned char *str)
{
    while(*str)
        putChars(*str++);
}
unsigned char getChars(void)
{
    while(!RI);
    RI=0;
```

```
return SBUF;  
}
```

OUTPUT : COMMUNICATIONS-RS 232



Annexure – I**List of programs according to O.U. curriculum****CS 432****EMBEDDED SYSTEMS LAB**

Instruction	3	Periods per week
Duration of University Examination	3	Hours
University Examination	50	Marks
Sessional	25	Marks

1. Use of 8-bit and 32-bit Microcontrollers (such as 8051 Microcontroller, ARM2148 / ARM2378, LPC 2141/42/44/46/48), Microcontroller and C –compiler (Keil, Ride etc.) to:
 - I) Interface Input – Output and other units such as: Relays, LEDs, LCDs, Switches, keypads, Stepper Motors, Sensors, ADCs, Timers.
 - II) Demonstrate Communications: RS232, IIC and CAN protocols.
 - III) Develop Control Applications such as: Temperature controller, Elevator controller, Traffic Controller.

2. Development and Porting of Real time applications on to Target machines such as Intel or other Computers using any RTOS.
 - I) Understanding Real Time Concepts using any RTOS through demonstration of:
 - a) Timing
 - b) Multi-tasking
 - c) Semaphores
 - d) Message Queues
 - e) Round-Robin Task Scheduling
 - f) Preemptive Priority based Task Scheduling
 - g) Priority Inversion
 - h) Signals

 - II) Applications development using any RTOS:
 - a) Any RTOS Booting.
 - b) Application Development under any RTOS.

