Module-5
8051 Programming in C

School of Electronics Engineering (SENSE)
Vellore Institute of Technology
Chennai

Introduction to Keil Software

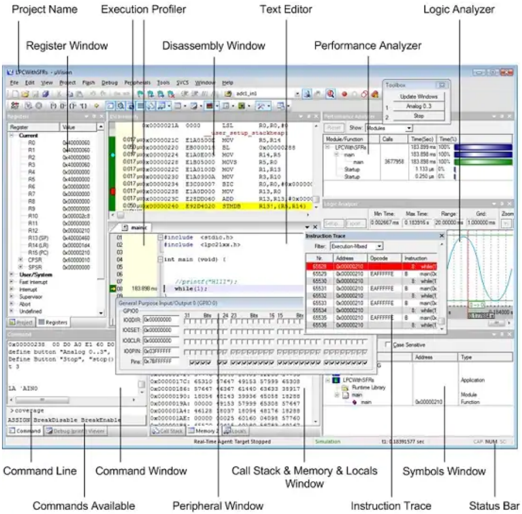


Figure: Keil Integrated Development Environment

Introduction to Keil Software

- Keil is a powerful IDE used for microcontroller development.
- Offers easy-to-use interface for coding, debugging, and simulation.
- Supports a wide range of microcontrollers, including the 8051 series.

```
#include <reg51.h>
sbit LED = P1^0; //Declare a bit for LED connected to pin 1.0

void main() {
    while(1) {
        LED = 0; // Turn on the LED
        // Delay code here
        LED = 1; // Turn off the LED
        // Delay code here
    }
}
```

Setting Up Keil for 8051 Development

- Download and install Keil uVision IDE from the official website.
- Configure the IDE for the 8051 microcontroller series.
- Set up a new project and choose the specific microcontroller model.
- Familiarize yourself with the Project Workspace and its components.

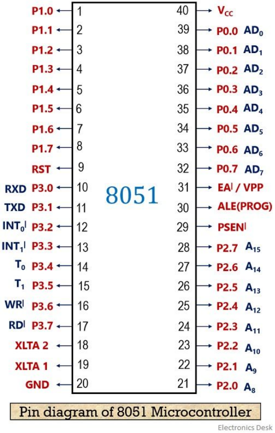
Overview of the 8051 Microcontroller

- The 8051 family has the largest number of diversified(multiple source) suppliers
  1. Intel (original)
  2. Atmel
  3. Philips/Signetics
  4. AMD
  5. Infineon (formerly Siemens)
  6. Matra
  7. Dallas Semiconductor/Maxim

Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

Figure: 8051 Family

Overview of the 8051 Microcontroller



Pin diagram of 8051 Microcontroller

- The 8051 microcontroller is one of the most popular microcontrollers in use today.
- It is an 8-bit processor with 128 bytes of RAM, 4KB of ROM, four parallel I/O ports, and one serial port.

Sample Code to Initialize 8051

```
void main() {
    // Sample code to initialize 8051
    TMOD = 0x01; // Configure timer 0 in mode 1
    TH0 = 0xFC;   // Load the timer value for delay
    TL0 = 0x66;
    TR0 = 1;      // Start the timer
    while(TF0 == 0); // Wait for the timer to overflow
    TR0 = 0;      // Stop the timer
    TF0 = 0;      // Clear the overflow flag
}
```

Data Types

Embedded C Data Types

Data Type	Size in bits	Data range/Usage
Unsigned Char	8	0 to 255
signed Char	8	-128 to +127
Unsigned int	16	0 to 65535
signed int	16	-32,768 to +32,767
Sbit	1	SFR bit addressable only
bit	1	RAM bit addressable only
SFR	8	RAM addresses 80-FF H only

Introduction to Embedded C Data Types

- **Sbit (Single Bit)**:
  - Represents a single bit of an SFR, mainly used for bit manipulation of ports and control registers.
  - Declaration example: `sbit led = P1^0;` // Declares 'led' as a bit on Port 1, Pin 0
  - Usage: Efficient for controlling individual bits like LEDs, switches.
- **Sfr (Special Function Register)**:
  - Represents a Special Function Register which is used for configuring and controlling various peripheral features of the 8051.
  - Declaration example: `sfr P1 = 0x90;` // Declares P1 as an SFR at address 0x90
  - Usage: Directly accessing and manipulating control registers like I/O ports, timers.
- **Bit**:
  - Represents a single bit, used for declaring and manipulating flags or simple binary variables.
  - Declaration example: `bit carryFlag;` // Declares a flag for carry in arithmetic operations
  - Usage: Used for status indicators, condition flags, etc.

Understanding sbit Data Type

```
sbit LED = P1^0;  // Declaring an sbit for LED connected to po

void main() {
    LED = 1; // Turn on the LED
    // Additional code...
}
```

- The sbit data type is used to access individual bits of a port.
- Commonly used for controlling individual peripherals like LEDs, motors, etc.

Understanding sfr Data Type

```
sfr P2 = 0xA0;  // Declaring Port 2 as an sfr at address 0xA0

void main() {
    P2 = 0x55; // Write to Port 2
    // Additional code...
}
```

- The sfr data type represents a full 8-bit special function register.
- Commonly used to interface with microcontroller peripherals.

Understanding bit Data Type

```
bit overflow;  // Declaring a bit variable for overflow

void checkOverflow() {
    overflow = (ACC > 255); // Set overflow based on a conditi
    // Additional logic...
}
```

- The bit data type is used to declare single-bit variables.
- Ideal for flags, status indicators, or simple binary conditions.

Fixed-Width Integer Types (stdint.h)

```
#include <stdint.h>

int8_t a = -128; // 8-bit signed
uint8_t b = 255; // 8-bit unsigned
int16_t c = -32768; // 16-bit signed
uint16_t d = 65535; // 16-bit unsigned
int32_t e = -2147483648; // 32-bit signed
uint32_t f = 4294967295; // 32-bit unsigned
int64_t g = -9223372036854775808; // 64-bit signed
uint64_t h = 18446744073709551615U; // 64-bit unsigned
```

Explanation: Fixed-width integer types provide precise control over integer sizes, essential for hardware programming where memory space is limited.

Using unsigned int to Create a Delay Function

```
#include <reg51.h>

// Delay function using unsigned int
void delay(unsigned int time) {
    unsigned int i, j;
    for(i = 0; i < time; i++)
        for(j = 0; j < 100; j++);
}

void main() {
    while(1) {
        P1 = 0xFF; // Turn ON all LEDs
        delay(100);
        P1 = 0x00; // Turn OFF all LEDs
        delay(100);
    }
}
```

Explanation: This program uses the `unsigned int` data type for the delay function parameters and loop counters to implement a simple delay. This data type ensures that the variables can store the required range of delay values.

Least and Fast Types (stdint.h)

```
#include <stdint.h>

int_least8_t i = 127; // At least 8-bit signed
uint_least8_t j = 255; // At least 8-bit unsigned
int_fast8_t k = 127; // Fastest min 8-bit signed
uint_fast8_t l = 255; // Fastest min 8-bit unsigned
```

Explanation: Least and fast integer types provide flexibility in choosing the smallest or fastest type that meets size requirements, optimizing for memory or speed.

Special Integer Types

```
#include <stdint.h>
#include <stddef.h>

intptr_t ptrToInt = (intptr_t)&a; // Pointer to int
uintptr_t uPtrToInt = (uintptr_t)&b; // Unsigned ptr to int
size_t size = sizeof(a); // Size of variable
ptrdiff_t ptrDiff = (char*)(&a + 1) - (char*)&a; // Pointer di
```

Explanation: These types are used for pointer operations, sizes, and differences, ensuring portability and correctness across different architectures.

Special Integer Types

```
#include <stdint.h>
#include <stddef.h>

void pointerOperations() {
    intptr_t ptrDiff;
    uintptr_t ptrAddress;
    size_t size;
    ptrdiff_t diff;

    // Example operations
    ptrAddress = (uintptr_t)&ptrDiff;
    size = sizeof(ptrDiff);
    diff = (ptrdiff_t)((char*)ptrAddress - (char*)&size);
}
```

Explanation: These types are used for pointer arithmetic and memory sizes, ensuring portability and correctness across different architectures.

Special Purpose Types

```c
volatile int m = 10; // Volatile int
register int n = 20; // Register int

enum state {ON, OFF};
enum state switchState = ON; // Enumeration
```

Explanation: ‘volatile‘ ensures the compiler does not optimize away access, ‘register‘ hints at storing variables in CPU registers for faster access, and ‘enum‘ improves code readability.

Using volatile for Register Access

```
#include <reg51.h>

void main() {
    volatile unsigned char * const pPort1 = &P1;
    *pPort1 = 0xFF;  // Turn ON all LEDs on Port 1
    while(1);
}
```

Explanation: The `volatile` keyword is used for the pointer `pPort1` to ensure that the compiler does not optimize the access to the microcontroller's hardware register `P1`. This is crucial for embedded systems where hardware registers may change independently of the program flow.

Using enum to Enhance Code Readability

```
#include <reg51.h>

// Define states for LED
enum LED_State {LED_OFF, LED_ON};

void setLED(enum LED_State state) {
    P1 = (state == LED_ON) ? 0xFF : 0x00;
}

void main() {
    setLED(LED_ON);  // Turn ON all LEDs
    while(1);
}
```

Explanation: The `enum` data type is used to define `LED_State`, making the code more readable and easier to maintain. It allows the use of `LED_ON` and `LED_OFF` as meaningful constants instead of directly using numbers, which can be less clear.

Delays

Creating Delays: Why They Matter

- Delays are crucial in embedded systems for timing control.
- They can synchronize the microcontroller's operations with external events.
- Example: Creating a precise delay for sensor data reading or actuator control.

Timing Calculations for Delays

- Understanding the clock cycle of the 8051.
- Calculating delay duration based on the oscillator frequency.
- Example: For a 12 MHz clock, one machine cycle = 1 / (12 MHz / 12) = 1 μs.
- Importance of precise delay in real-time applications.

Loop-Based Delays: Writing Efficient Code

```
void delay(unsigned int time) {
    unsigned int i, j;
    for(i = 0; i < time; i++) {
        for(j = 0; j < 1275; j++) {
            // Inner loop for delay
        }
    }
}
```

- Loop-based delays are a simple way to create timing delays.
- The actual delay depends on the number of iterations and the clock speed.

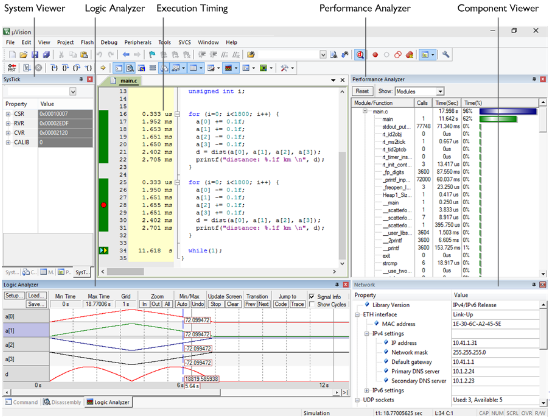
Function-Based Delays: Best Practices

```
void preciseDelay(unsigned int ms) {
    unsigned int i;
    for(i = 0; i < ms; i++) {
        // Call to a calibrated loop function
        calibratedLoop();
    }
}
```

- Function-based delays provide better control and reusability.
- Importance of calibration for accurate timing.
- Best practices: Modular design, calibration against a known time base.

Keil Debugger for Timing Analysis

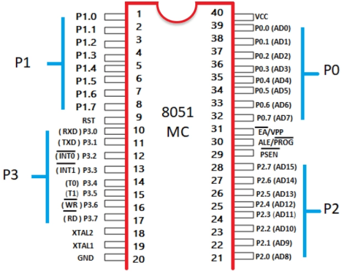
- Using Keil Debugger to analyze and verify delay durations.
- Step-by-step execution to observe timing and behavior of delay functions.
- Tools for measuring execution time and cycle counts.
- Debugging tips: Breakpoints, watch variables, and execution control.



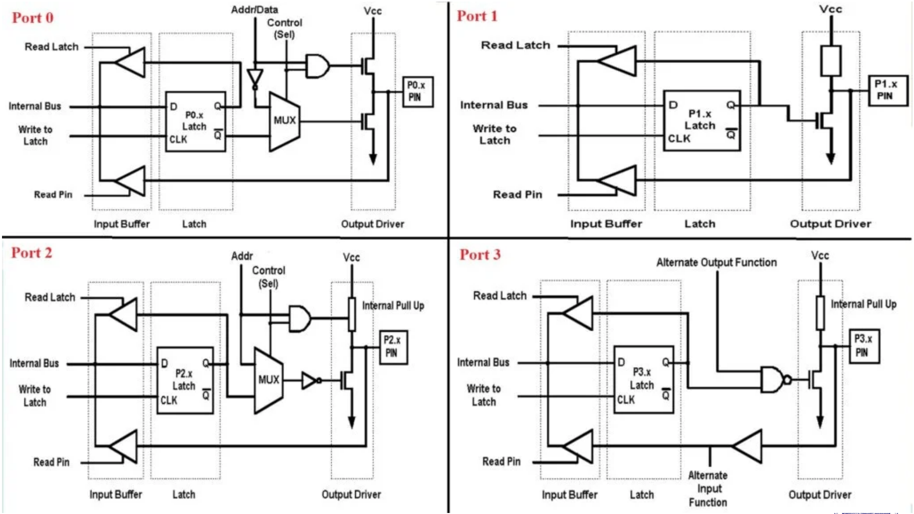
Programming I/O ports

I/O Port Architecture in 8051

- The four 8-bit I/O ports P0, P1, P2 and P3 each uses 8 pins
- Each port's dual role: general-purpose I/O and special functions.
- Bit and byte addressability of ports.
- All the ports upon RESET are configured as output, ready tobe used as input ports operations.



I/O Port Architecture in 8051



Programming I/O Ports: Theoretical Concepts

- Understanding how I/O ports work in the 8051.
- The concept of port latching and tristate buffers.
- Differences between input and output modes.
- Using ports for digital input/output operations.

Byte Addressable I/O Operations: Examples

```
void writeByteToPort(unsigned char data) {
    P1 = data; // Write a byte to Port 1
}

unsigned char readByteFromPort() {
    return P1; // Read a byte from Port 1
}
```

- Demonstration of how to write and read a full byte to/from an I/O port.
- Practical applications in interfacing with external devices.

Bit Addressable I/O Operations: Examples

```
sbit LED = P1^0;  // Declare LED connected to Port 1, Pin 0

void toggleLED() {
    LED = !LED; // Toggle the state of the LED
}
```

- Understanding how to manipulate individual bits of an I/O port.
- Examples include toggling LEDs, reading sensor states.

Practical: Writing to an I/O Port

```c
void initializePort() {
    P2 = 0x00; // Initialize Port 2 to all zeros
    P2 = 0xFF; // Set all bits of Port 2 to high
}
```

- Hands-on exercise: Initializing and writing data to a port.
- This exercise helps understand how data is sent to external peripherals.

Practical: Reading from an I/O Port

```
unsigned char readSensor() {
    return P3; // Read the current state of sensors connected
}
```

- Hands-on exercise: Reading data from a port.
- Application: Interpreting sensor data or user inputs.

Programs on Logical Operations

Using Logical Operations with I/O Ports

```
sbit LED = P1^0;  // LED connected to Port 1, Pin 0

void main() {
    LED = 1; // Turn on LED
    LED = LED & 0xFE; // Clearing bit 0 of Port 1, turning off the LED
    LED = LED | 0x01; // Setting bit 0 of Port 1, turning on the LED
}
```

- Demonstrates using bitwise AND and OR operations with I/O ports.
- Effective for controlling individual device states (e.g., LEDs, motors).

Bitwise AND Operation in Embedded C

```
#include <reg51.h> // Include for 8051 MCU register definitions

void main() {
    unsigned char portValue = P1; // Read current value of Port 1
    unsigned char mask = 0x0F;    // Mask to isolate lower 4 bits
    unsigned char result = portValue & mask; // Perform AND
    P2 = result; // Output result to Port 2
}
```

Explanation: This Embedded C program reads the current value of Port 1 on an 8051 microcontroller, performs a bitwise AND operation with a mask to isolate the lower 4 bits, and outputs the result to Port 2.

Bitwise OR Operation in Embedded C

```
#include <reg51.h> // Include for 8051 MCU register definitions

void main() {
    unsigned char portValue = P1; // Read current value of Port 1
    unsigned char mask = 0xF0;    // Mask to set upper 4 bits
    unsigned char result = portValue | mask; // Perform OR
    P2 = result; // Output result to Port 2
}
```

Explanation: This Embedded C program takes the current value of Port 1 on an 8051 microcontroller, performs a bitwise OR operation with a mask to set the upper 4 bits, and then outputs the result to Port 2.

Data Conversion

Data Conversion Techniques: Overview

- Importance of data conversion in embedded systems.
- Common conversions: Binary to decimal, ASCII to integer, float to string.
- Techniques for efficient data conversion in resource-constrained environments.

Example: ASCII to Integer Conversion

```
unsigned int asciiToInt(char *str) {
    unsigned int result = 0;
    while(*str) {
        result = result * 10 + (*str - '0');
        str++;
    }
    return result;
}
```

- Function to convert a string of ASCII characters to an integer.
- Useful in scenarios where numerical data is received as ASCII (e.g., from sensors or serial communication).

Data Conversion: Float to String

```
char* floatToString(float value, char* buffer, int decimalPlaces) {
    sprintf(buffer, "%.*f", decimalPlaces, value);
    return buffer;
}
```

- Function to convert a floating-point number to a string with specified precision.
- Useful for displaying numerical data on LCDs or sending it over serial connections.

Example: Binary to Decimal Conversion

```
unsigned int binaryToDecimal(unsigned int binary) {
    unsigned int decimal = 0, base = 1;
    while (binary > 0) {
        decimal += (binary % 10) * base;
        binary /= 10;
        base *= 2;
    }
    return decimal;
}
```

- Function to convert a binary number (as an integer) to its decimal equivalent.
- Iteratively processes each digit of the binary number, converting and accumulating the result.

Bit Manipulation: Setting a Bit

```
unsigned char setBit(unsigned char byte, int position) {
    return byte | (1 << position);
}
```

- Function to set (make 1) a specific bit in a byte.
- The position is 0-indexed, where position 0 is the least significant bit.
- Useful for configuring hardware registers or modifying control flags.

Bit Manipulation: Clearing a Bit

```
unsigned char clearBit(unsigned char byte, int position) {
    return byte & ~(1 << position);
}
```

- Function to clear (make 0) a specific bit in a byte.
- Ensures that only the target bit is affected using the bitwise NOT and AND operations.
- Essential for turning off features or flags in register settings.

Data serialization with I/O ports

Introduction to Data Serialization

- Data serialization is the process of converting structured data into a linear sequence of bytes for communication or storage.
- In embedded systems, serialization is crucial for sending and receiving data through I/O ports.
- Serialization protocols vary based on communication interfaces like UART, SPI, or I2C.

Serialization for UART Communication

```
void UART_send(unsigned char data) {
    while (!TXIF);  // Wait for previous transmission to complete
    TXREG = data;   // Load data into the transmission register
}

void UART_sendString(const char *str) {
    while(*str) {
        UART_send(*str++);
    }
}
```

- 'UART_send' is used to serialize and send a single byte.
- 'UART_sendString' serializes and sends a string byte by byte over UART.

Deserialization from I/O Ports

```
unsigned char UART_receive(void) {
    while (!RCIF);  // Wait for data to be received
    return RCREG;   // Read received data from register
}

void UART_receiveString(char *buffer, unsigned int max) {
    unsigned int i = 0;
    do {
        buffer[i] = UART_receive();
    } while (buffer[i++] != '\0' && i < max);
}
```

- 'UART_receive' reads a single byte from the UART receive buffer.
- 'UART_receiveString' reads bytes into a buffer until a null terminator is received or the buffer is full.

Serialization of Complex Data Structures

```
typedef struct {
    int id;
    float value;
    char label[10];
} SensorData;

void serializeSensorData(SensorData *data, unsigned char *buffer) {
    memcpy(buffer, data, sizeof(SensorData));
}

void UART_sendBuffer(unsigned char *buffer, unsigned int size) {
    for (unsigned int i = 0; i < size; ++i) {
        UART_send(buffer[i]);
    }
}
```

- Struct 'SensorData' contains multiple data types.
- 'serializeSensorData' copies the struct into a byte buffer.
- 'UART_sendBuffer' sends the serialized data over UART.

Deserialization of Complex Data Structures

```
void deserializeSensorData(unsigned char *buffer, SensorData *data)
    memcpy(data, buffer, sizeof(SensorData));
}

void UART_receiveBuffer(unsigned char *buffer, unsigned int size) {
    for (unsigned int i = 0; i < size; ++i) {
        buffer[i] = UART_receive();
    }
}
```

- 'deserializeSensorData' reconstructs the struct from a byte buffer.
- 'UART_receiveBuffer' receives serialized data into a buffer over UART.
- Ensure data alignment and struct packing matches on both ends.

Advanced Data Serialization Techniques

```c
typedef struct {
    unsigned int id;
    float temperature;
    char status;
} SensorData;

void serializeSensorData(SensorData *data, unsigned char *buffer) {
    // Implementation assumes little-endian architecture
    memcpy(buffer, data, sizeof(SensorData));
}

void deserializeSensorData(unsigned char *buffer, SensorData *data)
    memcpy(data, buffer, sizeof(SensorData));
}
```

- Serialization for structuring sensor data into a byte stream.
- Use cases: Storing sensor data to EEPROM, sending over UART.

Code Optimization Techniques for 8051

- **Loop Unrolling**:
  - Reducing loop overhead for small, predictable loops.
  - Example: Unrolling a loop for a fixed-size data handling.
- **Efficient Register Usage**:
  - Utilizing registers effectively to reduce memory access.
  - Example: Maximizing the use of the accumulator and register banks.
- **Minimizing Function Calls**:
  - Reducing overhead by using inline functions or macros.
  - Example: Replacing small function calls with inline code.
- **Profiling and Bottleneck Identification**:
  - Analyzing code to find and optimize slow or size-heavy sections.

Keil Project Management and Organization

- **Keil uVision IDE Overview**:
  - Project creation: Setting up microcontroller options, memory settings.
  - File management: Organizing source and header files.
- **Project Structure**:
  - Example: Modular design separating hardware abstraction, business logic, and utility functions.
- **Source Control Integration**:
  - Using version control systems like Git for project tracking and collaboration.
- **Documentation and Maintenance**:
  - Importance of in-code documentation and external documentation for project longevity.

Advanced Debugging in Keil

- **Debugging Peripheral Interactions**:
  - Techniques for monitoring and debugging I/O port operations.
  - Real-time watching of SFRs and peripheral registers.
- **Memory Breakpoints**:
  - Setting breakpoints in memory to trace data changes.
  - Example: Monitoring changes in a buffer during UART communication.
- **Simulator vs. Hardware Debugging**:
  - Comparing the use of simulators to real hardware debugging.
  - Limitations of simulators in replicating real-world scenarios.

Effective Testing with Keil Simulators

- **Simulating External Events**:
  - Techniques to simulate button presses and sensor inputs in Keil.
  - Automating test scenarios for comprehensive coverage.
- **Stress Testing**:
  - Applying load and performance testing to embedded applications.
  - Identifying and resolving timing and resource allocation issues.
- **Test Case Development**:
  - Writing and running unit tests for individual functions and modules.
  - Example: Creating tests for a custom string parsing function.

Troubleshooting Common Keil Issues

- **Compiler and Linker Errors**:
  - Decoding and resolving common error messages.
  - Example: Addressing 'Undefined symbol' errors in linkage.
- **Memory Management Challenges**:
  - Strategies for optimizing RAM and ROM usage.
  - Handling memory overflow and allocation errors.
- **Runtime Errors and Hangs**:
  - Debugging techniques for identifying runtime issues.
  - Example: Diagnosing and fixing an infinite loop condition.

Port Programming Example -1

Programme to toggle all the bits of P1 continuously

Port Programming Example -1

Programme to toggle all the bits of P1 continuously

```
#include<reg51.h>
void main (void)
{
for(;  ;) // repeat forever
 {
    P1=055;  // ox indicates data is in hex
    P1=0XAA;

}
```

Port Programming Example -2

Programme to send values 00 - FF to P1

Port Programming Example -2

Programme to send values 00 - FF to P1

```
#include<reg51.h>
Void main (void)
{
unsigned char z;
for (Z=0;z<=255;z++)
P1=z;
}
```

Port Programming Example -3

Programme to toggle bit D0 of port P1 50,000 times

Port Programming Example -3

Programme to toggle bit D0 of port P1 50,000 times

```
#include<reg51.h>
sbit MYBIT=P1^0;   //sbit is declared out of main program
Void main (void)
{
    unsigned int z;
    for(z=0;z<50000;z++)
    {
        MYBIT=0;
        MYBIT=1;
    }
}
```

Port Programming Example -4

Programme to toggle all the bits of P1 continuously with some delay

Port Programming Example -4

Programme to toggle all the bits of P1 continuously with some delay

```
#include<reg51.h>
void main (void)
{
    unsigned int x;
    for(;   ;) // repeat forever
    {
        P1=055; // ox indicates data is in hex
        for (x=0;x<40,000;x++); //delay size unknown
        P1=0xAA;
        for (x=0;x<40,000;x++);
    }
}
```

Port Programming Example -5

Programme to toggle all the bits of P1 continuously with a 250 ms

Port Programming Example -5

Programme to toggle all the bits of P1 continuously with a 250 ms

```
#include<reg51.h>
void main (void)
{
    while(1)          //repeat forever
    {
    p1=0x55;
    MSDelay(250);
    p1=0xAA;
    MSDelay(250)
    }
}
```

Port Programming Example -6

Programme to get a byte of data from P0. If it is less than 100, send it to P1; otherwise send it to P2

Port Programming Example -6

Programme to get a byte of data from P0. If it is less than 100, send it to P1; otherwise send it to P2

```
#include<reg51.h>
void main (void)
{
    unsigned int mybyte;
    Po=oxFF;
    for (;   ;) // repeat forever
    {
        mybyte = Po; // ox indicates data is in hex
        if(mybyte<100)
            P1=mybyte;//send it to Pi if less than 100
        else
            P2=mybyte; //send it to P2 if more than 100
    }
}
```

Port Programming Example -7

Programme to send hex values for ASCII characters of 0,1,2,3,4,5,A,B,C and D to port P1

Port Programming Example -7

Programme to send hex values for ASCII characters of 0,1,2,3,4,5,A,B,C and D to port P1

```
#include<reg51.h>
void main (void)
{
    unsigned char mynum[] = {'0','1','2','3','4','5','A','B','
    unsigned char z;
    for(z=0;z<10;z++)
    P1=mynum[z];
}
```

Port Programming Example -8

Programme to toggle LED connected to P1

Port Programming Example -8

Programme to toggle LED connected to P1

```
#include<reg51.h>
void main (void)
{
    P1=0;        //clear P1
    LED=0;       //clear LED
    for(;;)      //repeat forever
    {
    P1=LED;
    P1++;        //increment P1
    }
}
```

Port Programming Example -9

Programme to get a byte of data from P1, wait $1/2$ second and then send it to P2

Port Programming Example -9

Programme to get a byte of data from P1, wait $1/2$ second and then send it to P2

```
#include<reg51.h>
void MSDelay(unsigned int);
void main (void)
{
    unsigned char mybyte;
    P1=oxFF;        // make P1 input port
    while(1)
    {
    mybyte=P1;      //get a byte from P1
    MSDelay(500);
    P2=mybyte;      // send it to P2
    }
}
```

Port Programming Example -10

Programme to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2.

Port Programming Example -10

Programme to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2.

```
#include<reg51.h>
sbit mybit=P2^4;          //Use the Px^y format
                          // where x is the port 0,1,2 or 3 and
                          // y is the bit 0-7 of that port
void main(void)
{
while(1)
{
mybit=1;                 // turn on P2.4
mybit=0;                 // turn off P2.4
}
}
```