"Microcontroller Fundamentals: 8051 Basic I/O and Timers"

Module 1: Introduction to the 8051 Microcontroller and Development Environment

1.1 What is a Microcontroller?

A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. Unlike a general-purpose microprocessor (like those in computers), which are designed for broad applications, microcontrollers are optimized for specific control tasks, often in real-time. They integrate a processor core, memory (both program and data), and programmable input/output peripherals on a single chip. This self-contained nature makes them ideal for small, cost-effective embedded applications.

1.2 Introduction to the 8051 Microcontroller

The 8051 is an 8-bit microcontroller family developed by Intel in 1980. Despite its age, it remains widely used for educational purposes and in various industrial applications due to its simplicity, robust instruction set, and vast community support. It is based on the Harvard architecture, meaning it has separate memory spaces for program and data. The 8051's key features include:

- 8-bit CPU: Processes data in 8-bit chunks.
- On-chip Flash/ROM: For program storage (typically 4KB, 8KB, or more depending on the derivative).
- On-chip RAM: For data storage (typically 128 bytes or 256 bytes).
- Four 8-bit I/O Ports: General-purpose input/output pins (P0, P1, P2, P3).
- Two 16-bit Timers/Counters: For generating delays, measuring time, and counting external events (Timer 0 and Timer 1).
- Full Duplex Serial Port: For communication with other devices (UART).
- Interrupt Controller: For handling external and internal events.
- On-chip Clock Oscillator: Provides the timing for the microcontroller's operations.

1.3 8051 Architecture Overview

The 8051 architecture can be understood by examining its core components:

- CPU (Central Processing Unit): The brain of the 8051, responsible for executing instructions. It contains the Arithmetic Logic Unit (ALU), Program Counter (PC), Data Pointer (DPTR), and various registers.
- Program Memory (ROM/Flash): Stores the user program (instructions). The 8051 fetches instructions from this memory.
- Data Memory (RAM): Used for temporary data storage, variables, and the stack. The 8051's RAM is divided into several areas:
    - Register Banks: Four banks of 8 registers (R0-R7) each, providing fast access to frequently used data.
    - Bit-Addressable RAM: A specific area of RAM where individual bits can be accessed and manipulated.

    - General Purpose RAM: For general data storage.
    - Special Function Registers (SFRs): These registers control and monitor the operation of the 8051's internal peripherals (e.g., I/O ports, timers, serial port). Each SFR has a unique address.
- I/O Ports: Four 8-bit bidirectional ports (P0, P1, P2, P3) that can be configured as inputs or outputs.
- Timers/Counters: Two 16-bit timers/counters (Timer 0 and Timer 1) used for precise timing, delay generation, and event counting.
- Serial Port (UART): Enables serial communication with other devices.
- Interrupt Control: Manages and prioritizes interrupt requests from various sources.
- Bus Control: Manages the flow of data between the CPU, memory, and peripherals.
- Oscillator and Clock Circuit: Provides the necessary clock pulses for the 8051's internal operations. The crystal oscillator frequency determines the instruction cycle time.

1.4 8051 Development Board

An 8051 development board typically includes:

- 8051 Microcontroller Chip: The core component.
- Power Supply: For providing the necessary voltage to the board.
- Crystal Oscillator: Provides the clock signal.
- Reset Circuit: For resetting the microcontroller.
- LEDs: For visual output and testing I/O.
- Push Buttons/Switches: For input and testing I/O.
- Seven-Segment Displays/LCD Interface: For displaying numerical or textual information.
- Serial Communication Interface: For connecting to a computer (e.g., via USB-to-UART converter).
- Programming Interface: For downloading programs to the microcontroller.
- Headers/Jumpers: For connecting external components and configuring the board.

1.5 Keil uVision IDE

Keil uVision is a widely used Integrated Development Environment (IDE) for 8051 microcontroller development. It provides a complete software development environment including:

- Text Editor: For writing C or assembly code.
- C Compiler (Keil C51): Translates C code into machine-executable instructions.
- Assembler: Translates assembly code into machine-executable instructions.
- Linker/Locator: Combines compiled object files and assigns memory addresses.
- Debugger: Allows step-by-step execution of code, viewing register contents, and memory inspection to identify and fix errors.

- **Simulator: Simulates the 8051 microcontroller's behavior without requiring physical hardware, enabling testing and debugging of programs.**

Procedure for using Keil uVision (Basic Steps):

1. **Create a New Project: Start a new project and select the target microcontroller (e.g., Atmel 89C51, NXP P89V51RD2).**
2. **Add Source Files: Create new C files and add them to the project.**
3. **Write Code: Write your C program in the editor.**
4. **Configure Project Options: Set the crystal frequency, memory models, and output format.**
5. **Build Target: Compile, assemble, and link the project to generate a HEX file (the executable program for the microcontroller).**
6. **Start Debug/Simulator: Initiate the debugger or simulator to test your program.**

Module 2: Basic I/O Operations

2.1 8051 I/O Ports

The 8051 microcontroller has four 8-bit I/O ports: P0, P1, P2, and P3. Each port consists of 8 pins, which can be individually configured as either input or output.

- **Port 0 (P0): A true open-drain bidirectional port. It requires external pull-up resistors when used as an output in non-multiplexed mode. It also functions as the multiplexed address/data bus for external memory interfacing.**
- **Port 1 (P1): A true bidirectional I/O port with internal pull-ups.**
- **Port 2 (P2): A true bidirectional I/O port with internal pull-ups. It also functions as the high-order address bus for external memory interfacing.**
- **Port 3 (P3): A true bidirectional I/O port with internal pull-ups. Many pins on P3 have alternate special functions (e.g., serial communication, external interrupts, timer inputs).**

2.2 Controlling I/O Ports in C

In C programming for the 8051, I/O ports are typically accessed as SFRs. Each port is represented by a register, and individual bits within the port can be accessed directly.

Port Addressing:

The 8051 architecture provides direct bit addressing for P0, P1, P2, and P3, making it easy to manipulate individual pins.

For example, to access Port 1, you would use the `P1` SFR. To access a specific pin, say P1.0, you can use `P1_0`.

Output Operations (Blinking LEDs):

To make an LED blink, we need to turn it ON and OFF with a delay in between. A logic LOW (0V) typically turns an LED ON (current flows from VCC through the LED to the port pin configured as output LOW), while a logic HIGH (VCC) turns it OFF (no potential difference). However, the exact behavior depends on how the LED is connected (common anode or common cathode). For common anode configuration, a LOW turns ON the LED.

Example: Blinking an LED connected to P1.0

```c
#include <reg51.h> // Include header for 8051 SFRs

void delay_ms(unsigned int ms); // Function prototype for delay

void main() {
    while (1) { // Infinite loop
        P1_0 = 0; // Turn LED ON (assuming common anode, or active low)
        delay_ms(500); // Wait for 500 milliseconds
        P1_0 = 1; // Turn LED OFF
        delay_ms(500); // Wait for 500 milliseconds
    }
}

// Simple delay function (will be improved with timers later)
void delay_ms(unsigned int ms) {
    unsigned int i, j;
    for (i = 0; i < ms; i++) {
        for (j = 0; j < 120; j++); // Adjust this value for accurate delay based on crystal
frequency
    }
}
```

Numerical Explanation of `delay_ms`:

The `delay_ms` function uses nested `for` loops to create a delay. The inner loop `for (j = 0; j < 120; j++);` executes 120 times. The outer loop `for (i = 0; i < ms; i++)` executes `ms` times. The value `120` is an empirical value that needs to be tuned based on the 8051's crystal frequency.

Let's assume an 8051 with a 12 MHz crystal.

- One machine cycle for a standard 8051 with a 12 MHz crystal is 12 clock cycles.
- Time for one machine cycle = 12 / 12 MHz = 1 microsecond (μs).
- The inner loop `for (j = 0; j < 120; j++);` might take approximately 2-3 machine cycles per iteration (depending on compiler optimization and instruction set). Let's assume 2 machine cycles per iteration for simplicity.
- So, `120 * 2` machine cycles = `240` machine cycles.

- Total time for inner loop = 240 μs.
- For a 1ms delay (1000 μs), we would need 1000 / 240 iterations of the inner loop in theory. The 120 value is thus adjusted through trial and error or precise calculation to achieve the desired delay. This "software delay" is inefficient and not very accurate; hence, we will introduce timers for better delays.

Input Operations (Reading Switch Inputs):

To read a switch, the port pin connected to the switch must be configured as an input. When a switch is pressed, it typically pulls the pin to a logic LOW (0V) if connected to ground, or HIGH (VCC) if connected to VCC through a pull-up resistor.

Example: Reading a switch connected to P1.1 and controlling an LED on P1.0

```c
#include <reg51.h> // Include header for 8051 SFRs

sbit LED = P1^0; // Define LED as P1.0
sbit SWITCH = P1^1; // Define SWITCH as P1.1

void main() {
  // Configure P1.1 as input (default for P1 with internal pull-ups)
  // No explicit configuration needed for P1 as input due to internal pull-ups

  while (1) { // Infinite loop
    if (SWITCH == 0) { // If switch is pressed (assuming active low switch)
      LED = 0; // Turn LED ON
    } else {
      LED = 1; // Turn LED OFF
    }
  }
}
```

2.3 Pull-up Resistors

For input pins, especially those on Port 0, external pull-up resistors are often necessary. When a pin is configured as input, its internal MOSFET is turned off, effectively making it high-impedance. Without a pull-up resistor, the pin's voltage can "float," leading to unpredictable readings. A pull-up resistor connects the input pin to VCC, ensuring a default HIGH state when the switch is open. When the switch is closed, it pulls the pin to LOW. Ports P1, P2, and P3 have internal pull-up resistors, simplifying their use as inputs.

2.4 Bit Manipulation and Logical Operations

The 8051's instruction set is rich in bit-level operations, which are very useful for controlling individual pins or flags. In C, these operations are typically performed using logical operators:

- Bitwise AND (`&`): Used to clear specific bits or check if a bit is set.
  - Example: `P1 = P1 & 0xFE;` // Clears P1.0 (sets it to 0), leaves other bits unchanged. `0xFE` in binary is `11111110`.
- Bitwise OR (`|`): Used to set specific bits.
  - Example: `P1 = P1 | 0x01;` // Sets P1.0 (sets it to 1), leaves other bits unchanged. `0x01` in binary is `00000001`.
- Bitwise XOR (`^`): Used to toggle specific bits.
  - Example: `P1 = P1 ^ 0x01;` // Toggles P1.0.
- Bitwise NOT (`~`): Used to invert all bits.
  - Example: `P1 = ~P1;` // Inverts all bits of Port 1.
- Left Shift (`<<`): Multiplies by powers of 2.
  - Example: `data = 1 << 3;` // `data` becomes 8 (00001000b).
- Right Shift (`>>`): Divides by powers of 2.
  - Example: `data = 8 >> 1;` // `data` becomes 4 (00000100b).

For individual bit access, the `sbit` keyword in Keil C is very convenient:

```
C

sbit my_led = P1^0; // Declares 'my_led' as an alias for P1.0
my_led = 0; // Directly controls P1.0
```

Module 3: Timer Programming

3.1 Introduction to Timers

The 8051 has two 16-bit timers/counters, Timer 0 and Timer 1. These timers can be used for:

- Generating Delays: Creating precise time intervals.
- Event Counting: Counting external pulses or events.
- Baud Rate Generation: For the serial communication port.

Each timer consists of two 8-bit registers:

- Timer 0: `TH0` (Timer High 0) and `TL0` (Timer Low 0)
- Timer 1: `TH1` (Timer High 1) and `TL1` (Timer Low 1)

These register pairs form a 16-bit timer. When configured as a timer, they increment at a rate determined by the system clock. When configured as a counter, they increment on external pin transitions.

3.2 Timer Control Registers (TMOD and TCON)

Two Special Function Registers (SFRs) are crucial for controlling the timers:

- TMOD (Timer Mode Register): This 8-bit register defines the operating mode for Timer 0 and Timer 1. It is not bit-addressable.

B	Name	Description
7	GATE1	Timer 1 Gate control (1: Gated by INT1 pin, 0: Not gated)
6	C/T1	Timer 1 Counter/Timer select (1: Counter, 0: Timer)
5	M1_1	Timer 1 Mode bit 1
4	M0_1	Timer 1 Mode bit 0
3	GATE0	Timer 0 Gate control (1: Gated by INT0 pin, 0: Not gated)

2	C/T 0	Timer 0 Counter/Timer select (1: Counter, 0: Timer)
1	M1_ 0	Timer 0 Mode bit 1
0	M0_ 0	Timer 0 Mode bit 0

- Timer Modes (M1, M0 bits):
  - Mode 0 (13-bit Timer): `M1=0, M0=0`. Uses TLx as an 8-bit counter and THx as a 5-bit counter. Not commonly used.
  - Mode 1 (16-bit Timer): `M1=0, M0=1`. Uses TLx and THx as a 16-bit counter. The timer increments from `0000H` to `FFFFH`, then overflows, setting the TFx flag. This is the most commonly used mode for generating delays.
  - Mode 2 (8-bit Auto-Reload Timer): `M1=1, M0=0`. Uses TLx as an 8-bit counter and THx to hold the reload value. When TLx overflows from `FFH` to `00H`, it sets the TFx flag and is automatically reloaded with the value in THx. Useful for generating precise periodic events.
  - Mode 3 (Split Timer Mode): `M1=1, M0=1`. Timer 0 splits into two 8-bit timers (TL0 and TH0). Timer 1 stops counting in this mode. Less commonly used.
- TCON (Timer Control Register): This 8-bit register contains the run control bits and overflow flags for the timers, and also interrupt-related flags. It is bit-addressable.

B	Name	Description
7	TF 1	Timer 1 Overflow Flag (set when Timer 1 overflows, cleared by hardware when CPU vectors to ISR)

6	TR1	Timer 1 Run Control Bit (1: Timer 1 ON, 0: Timer 1 OFF)
5	TF0	Timer 0 Overflow Flag (set when Timer 0 overflows, cleared by hardware when CPU vectors to ISR)
4	TR0	Timer 0 Run Control Bit (1: Timer 0 ON, 0: Timer 0 OFF)
3	IE1	External Interrupt 1 Edge Flag (set by hardware when INT1 external interrupt edge detected)
2	IT1	External Interrupt 1 Type Select (1: Edge-triggered, 0: Level-triggered)
1	IE0	External Interrupt 0 Edge Flag (set by hardware when INT0 external interrupt edge detected)
0	IT0	External Interrupt 0 Type Select (1: Edge-triggered, 0: Level-triggered)

3.3 Calculating Timer Values for Delays (Mode 1)

In Mode 1 (16-bit timer), the timer increments by 1 for every machine cycle. A standard 8051 with a 12 MHz crystal has a machine cycle time of 1 microsecond (μs).

This means the timer increments 1,000,000 times per second.

To generate a specific delay, we need to load the timer with an initial value such that it overflows after the desired time.

The maximum count for a 16-bit timer is 65536 (from 0 to 65535).

Formula for Initial Timer Value (Mode 1, 16-bit Timer):

Initial Value = 65536 - (Desired Delay in microseconds)

This formula assumes a 12 MHz crystal frequency where one machine cycle is 1 microsecond. If the crystal frequency is different, the `Desired Delay` needs to be scaled by `(Crystal Frequency in MHz / 12)`.

For a 12 MHz crystal:

Initial Value = 65536 - Desired Delay (in µs)

Example: Generate a 1 ms (1000 µs) delay using Timer 0, Mode 1 with a 12 MHz crystal.

Initial Value = 65536 - 1000 = 64536

This 16-bit value (64536) needs to be loaded into TH0 and TL0.

To convert 64536 to hexadecimal: 64536_10=FA00_16

So, `TH0 = 0xFA` and `TL0 = 0x00`.

Procedure for Delay Generation using Timers (Polling Method - Mode 1):

1. Configure TMOD: Set the desired timer mode (e.g., Mode 1 for Timer 0: `TMOD = 0x01`).
2. Load Timer Registers: Load `THx` and `TLx` with the calculated initial value.
3. Start Timer: Set the `TRx` bit in `TCON` to 1 (e.g., `TR0 = 1`).
4. Wait for Overflow: Monitor the `TFx` flag in `TCON`. Wait until `TFx` becomes 1.
5. Stop Timer: Clear the `TRx` bit (e.g., `TR0 = 0`).
6. Clear Flag: Clear the `TFx` flag (e.g., `TF0 = 0`). This is crucial, especially if you're not using interrupts, as the flag needs to be ready for the next overflow.

Example: Blinking an LED using Timer 0 for 500 ms delay

C

```c
#include <reg51.h>

sbit LED = P1^0; // Define LED as P1.0

void timer_delay_ms(unsigned int ms);

void main() {
  while (1) {
        LED = 0; // Turn LED ON
        timer_delay_ms(500); // 500 ms delay
        LED = 1; // Turn LED OFF
        timer_delay_ms(500); // 500 ms delay
    }
}

void timer_delay_ms(unsigned int ms) {
    unsigned int i;
    for (i = 0; i < ms; i++) {
        // For a 12MHz crystal, 1 machine cycle = 1 us.
        // To get 1ms delay, we need 1000 machine cycles.
        // Initial value = 65536 - 1000 = 64536 (0xFA00)
        TMOD = 0x01; // Timer 0, Mode 1 (16-bit timer)
        TH0 = 0xFA;  // Load high byte
        TL0 = 0x00;  // Load low byte
        TR0 = 1;     // Start Timer 0
        while (TF0 == 0); // Wait for overflow flag to set
        TR0 = 0;     // Stop Timer 0
        TF0 = 0;     // Clear Timer 0 overflow flag
    }
}
```

3.4 Generating Square Waves using Timers (Mode 2 - Auto Reload)

Mode 2 (8-bit auto-reload timer) is ideal for generating periodic events, like square waves. In this mode, `TLx` counts up, and when it overflows (from `FFH` to `00H`), the value from `THx` is automatically loaded into `TLx`. This makes it easy to generate a constant time interval.

Formula for Initial Timer Value (Mode 2, 8-bit Auto-Reload Timer):

The `THx` value determines the count. If `TLx` counts from `N` to `FFH` and overflows, the number of counts is `256 - N`.

$THx = 256 - (DesiredNumberofCounts)$

Where:

- **`Desired Number of Counts` is the number of machine cycles for one half-period of the square wave.**

Example: Generate a square wave with a 1 kHz frequency (period = 1 ms). Each half-period is 0.5 ms (500 μs).

- **Number of machine cycles for 0.5 ms = 500.**
- **`TH0 = 256 - 500` (This won't work directly since 256 - 500 is negative).**

This indicates that an 8-bit timer cannot directly provide a 500 μs delay with a 12 MHz crystal (as its maximum count is 256 μs). We would need to use a smaller crystal frequency or a different approach (e.g., using a 16-bit timer and toggling the pin within the interrupt service routine, or using a software loop in combination with the 8-bit auto-reload timer).

Let's adjust the example for a frequency that an 8-bit timer can handle. Suppose we want a square wave with a period of 256 μs (approx. 3.9 kHz). Each half-period is 128 μs.

- Desired number of counts for one half-period = 128.
- `TH0 = 256 - 128 = 128 (0x80H)`

Procedure for Square Wave Generation using Timers (Mode 2 with Interrupts):

1. Configure TMOD: Set the desired timer mode (e.g., Mode 2 for Timer 0: `TMOD = 0x02`).
2. Load THx: Load `THx` with the calculated reload value.
3. Enable Timer Interrupts:
    - Set `ETx` bit in `IE` (Interrupt Enable) register to 1 (e.g., `ET0 = 1` for Timer 0 interrupt).
    - Set `EA` bit in `IE` register to 1 to enable global interrupts (`EA = 1`).
4. Start Timer: Set the `TRx` bit in `TCON` to 1 (e.g., `TR0 = 1`).
5. Write Interrupt Service Routine (ISR): Create an ISR for the timer. This routine will be executed every time the timer overflows. Inside the ISR, toggle the output pin. The `TFx` flag is automatically cleared by hardware when the ISR is entered.

Example: Generating a square wave on P1.0 using Timer 0, Mode 2 (12 MHz crystal, target 3.9 kHz square wave)

```c
#include <reg51.h>

sbit SQUARE_WAVE_OUT = P1^0; // Define output pin

void timer0_ISR(void) interrupt 1 { // Timer 0 Interrupt Vector (interrupt 1)
  SQUARE_WAVE_OUT = ~SQUARE_WAVE_OUT; // Toggle the output pin
  // TF0 is automatically cleared by hardware when entering ISR
  // TH0 does not need to be reloaded as it's in auto-reload mode
}

void main() {
  SQUARE_WAVE_OUT = 0; // Initialize output to LOW

  TMOD = 0x02; // Timer 0, Mode 2 (8-bit auto-reload)
  TH0 = 0x80;  // Load TH0 for 128 counts (256 - 128 = 128) -> 128 us half-period
```

```
    TL0 = 0x00;  // Initial value of TL0 (can be anything, it will reload from TH0 on first
overflow)

    ET0 = 1;     // Enable Timer 0 interrupt
    EA = 1;      // Enable global interrupts

    TR0 = 1;     // Start Timer 0

    while (1) {
        // Main loop can perform other tasks
    }
}
```

Numerical Explanation for Square Wave Example:

- Crystal Frequency = 12 MHz
- Machine Cycle Time = 12 / 12 MHz = 1 µs
- `TH0 = 0x80` (decimal 128). In Mode 2, `TL0` counts from `TH0` up to `FFH`, then overflows.
- Number of counts per overflow = `256 - 128 = 128` counts.
- Time for one overflow = `128 counts * 1 µs/count = 128 µs`.
- Since the ISR toggles the pin every time the timer overflows, one full cycle of the square wave will take two overflows.
- Period of square wave = `128 µs (HIGH) + 128 µs (LOW) = 256 µs`.
- Frequency of square wave = `1 / (256 µs) = 1 / (256 * 10^-6 s) ≈ 3906.25 Hz ≈ 3.9 kHz`.

3.5 Timer Interrupts

Using timer interrupts is a more efficient way to handle time-based events compared to the polling method (where the CPU continuously checks the `TFx` flag). When a timer overflows and its interrupt is enabled, the CPU automatically jumps to a predefined Interrupt Service Routine (ISR), executes the code within it, and then returns to the main program. This allows the main program to perform other tasks while the timer runs in the background.

Interrupt Related SFRs:

- IE (Interrupt Enable) Register: Controls which interrupts are enabled or disabled.
  - `EA` (Enable All): Global interrupt enable/disable bit. Must be set to 1 for any interrupt to occur.
  - `ET0`, `ET1`: Enable Timer 0/1 interrupts.
  - `EX0`, `EX1`: Enable External Interrupt 0/1.
  - `ES`: Enable Serial Port interrupt.
- IP (Interrupt Priority) Register: Defines the priority of interrupts (not covered in detail here, but useful for complex systems).

Interrupt Priority:

The 8051 has a fixed interrupt priority scheme if the IP register is not modified. Generally, external interrupts have higher priority than timer interrupts.

Structure of an ISR in Keil C:

```c
void timer_isr_name(void) interrupt <interrupt_vector_number> {
   // Your code to be executed when the interrupt occurs
}
```

- interrupt <interrupt_vector_number>: Specifies the interrupt vector.
  - 0: External Interrupt 0 (INT0)
  - 1: Timer 0
  - 2: External Interrupt 1 (INT1)
  - 3: Timer 1
  - 4: Serial Port

Module 4: Simulation and Debugging with Keil uVision

4.1 The Importance of Simulation

Simulation is a crucial step in embedded systems development. It allows developers to:

- Test Code without Hardware: Develop and test programs even before the physical hardware is available.
- Identify Logic Errors: Debug software logic in a controlled environment.
- Verify Timing and Performance: Check if delays are accurate and if the system responds as expected.
- Observe Internal States: Monitor register values, memory contents, and I/O port states.
- Faster Development Cycle: Rapidly iterate and test changes.

4.2 Keil uVision Debugger Features

The Keil uVision debugger provides powerful tools for program analysis:

- Source Code Window: Displays your C or assembly code, allowing you to set breakpoints.
- Disassembly Window: Shows the machine code generated from your source, useful for understanding low-level execution.
- Registers Window: Displays the current values of all 8051 SFRs (e.g., ACC, B, PSW, TMOD, TCON, P0, P1, P2, P3, TH0, TL0, etc.). This is essential for verifying I/O and timer operations.

- Memory Window: Allows you to view and modify the contents of program memory, data memory, and external memory.
- Watch Window: Enables you to monitor the values of specific variables or expressions during execution.
- Call Stack Window: Shows the sequence of function calls.
- Peripherals Window: Provides graphical interfaces to simulate the behavior of 8051 peripherals like I/O ports, timers, and the serial port. You can visually toggle bits on simulated ports, or see the timer counting.
- Breakpoints: You can set breakpoints at specific lines of code. When the program execution reaches a breakpoint, it pauses, allowing you to inspect the system state.
- Step-by-Step Execution:
    - Step Over: Executes a function call as a single step without going into the function's code.
    - Step Into: Steps into a function call, allowing you to debug the function's internal logic.
    - Step Out: Continues execution until the current function returns.
    - Run: Executes the program continuously until a breakpoint is hit or the program finishes (in a simulator, it might run indefinitely in an infinite loop).

4.3 Simulating I/O Operations in Keil uVision

1. Start Debug Session: After building your project, go to `Debug -> Start/Stop Debug Session`.
2. Open Peripherals Window: Navigate to `Peripherals -> I/O Ports -> Port 1` (or any other port you are using).
3. Observe Port State: As your program executes, you will see the virtual pins in the I/O Ports window change state (High/Low) corresponding to your code (e.g., an LED blinking).
4. Simulate Input: You can click on the virtual pins in the I/O Ports window to change their state, simulating a switch press or external input. For example, if you have a switch connected to P1.1, you can click on P1.1 in the peripheral window to toggle its state and observe how your program reacts.

4.4 Simulating Timer Operations in Keil uVision

1. Start Debug Session: `Debug -> Start/Stop Debug Session`.
2. Open Peripherals Window: Navigate to `Peripherals -> Timers -> Timer 0` (or Timer 1).
3. Observe Timer Registers: You can see the `TH0` and `TL0` (or `TH1`, `TL1`) registers counting in real-time.
4. Monitor Flags: The `TCON` and `TMOD` registers are also visible in the SFRs window, allowing you to monitor the `TFx` flags and `TRx` bits.
5. Verify Delays: By observing the time displayed in the debugger status bar and the toggling of your output pin, you can verify if your generated delays and square wave frequencies are accurate.

4.5 Debugging Techniques

- **Breakpoints: Set breakpoints at key points in your code (e.g., before and after a delay, where an I/O operation occurs, or inside an ISR) to pause execution and examine register values and variable states.**
- **Single Stepping: Use `Step Over` or `Step Into` to execute your code one line at a time, closely observing the changes in SFRs and memory.**
- **Watch Window: Add important variables (like `ms` in `timer_delay_ms`) or SFRs (like `TMOD`, `TCON`, `TH0`, `TL0`) to the Watch window to see their values change dynamically.**
- **Memory Window: Inspect specific memory locations if you are dealing with arrays or larger data structures.**
- **Logic Analyzer: Keil uVision sometimes offers a basic logic analyzer feature (though a dedicated external logic analyzer is better for complex timing), which can graphically display the waveforms of chosen I/O pins, confirming square wave generation.**

Deliverables:

- **C Code: All the C programs developed during the course for basic I/O (blinking LEDs, reading switch inputs) and timer programming (generating delays, creating square waves).**
- **Simulation Results: Screenshots or descriptive notes of the Keil uVision simulator showing:**
  - **LED blinking correctly.**
  - **LED responding to simulated switch input.**
  - **Timer registers counting up.**
  - **Output pin toggling for square wave generation, indicating the approximate frequency.**
- **Output on Hardware (Optional/If Available): If an 8051 development board is available, a demonstration (e.g., video, photo) of the programs running on the actual hardware, confirming the functionality.**
