Lab 5: Polling vs. Interrupt-Driven I/O

Simulator Used: EDSIM51DI (8051 Simulator) [1]

1. INTRODUCTION

This laboratory exercise explores two fundamental methods for managing Input/Output (I/O) operations and timing in the 8051 microcontroller: **Polling** and **Interrupt-Driven I/O**. In embedded systems, the choice between these methods significantly impacts CPU efficiency and system responsiveness.

While the previous lab focused on memory architecture, this lab examines how the processor interacts with peripherals—specifically Timer 0—to create precise delays. We will analyze how polling consumes CPU cycles by constantly checking status flags, contrasted with interrupts that allow the CPU to perform other tasks until a peripheral requires service.

2. OBJECTIVES

The primary objectives of this laboratory are:

1. Compare the efficiency of **Polling** versus **Interrupt-driven** execution.
2. Understand the role of the **Timer Flag (TF0)** in a polling loop.
3. Configure **Special Function Registers (SFRs)** for interrupt control (IE, TCON, TMOD)[3].
4. Observe the automatic saving of the **Program Counter (PC)** on the stack during an interrupt[44].
5. Demonstrate real-time background tasking while a peripheral handles timing.

3. THEORY

3.1 Polling Method

In a polling system, the CPU continuously monitors a specific bit (a flag) to see if an event has occurred.

- **Mechanism:** The code enters a tight loop (e.g., `JNB TF0, $`) until the hardware sets the flag.
- **Efficiency:** Very low. The CPU is "busy-waiting" and cannot perform other computations while waiting for the I/O event.
- **Complexity:** Simple to implement but does not scale well for multiple I/O devices.

3.2 Interrupt-Driven Method

Interrupts allow the hardware to signal the CPU only when service is required[5].

- **Mechanism:** The CPU executes a main program. When the timer overflows, the hardware forces a jump to a specific **Interrupt Vector** address[6].
- **Efficiency:** High. The CPU only services the peripheral when necessary, leaving it free for background tasks[77].
- **Stack Usage:** The processor automatically pushes the return address onto the stack before jumping to the **Interrupt Service Routine (ISR)**[888].

4. CODE ANALYSIS

4.1 Polling Implementation (polling.asm)

This program uses Timer 0 in Mode 1 (16-bit) to toggle an LED.

- **Timer Configuration:** MOV TMOD, #01H sets Timer 0 to 16-bit mode.
- **Wait Loop:** The instruction JNB TF0, WAIT_OVERFLOW keeps the CPU trapped in a loop as long as the Timer 0 Overflow Flag (TF0) is zero.
- **Manual Reset:** Once TF0 is set, the software must manually clear the flag (CLR TF0) and stop the timer before restarting the cycle.

4.2 Interrupt Implementation (interrupt-driven.asm)

This program achieves the same LED toggle but uses the 8051's interrupt system.

- **Interrupt Vector:** The code at ORG 000BH (Timer 0 vector) contains an LJMP TIMER0_ISR[9].
- **Enabling Interrupts:** The instructions SETB ET0 (Enable Timer 0 Interrupt) and SETB EA (Enable Global Interrupts) are required to activate the system[10101010].
- **Main Loop Efficiency:** In MAIN_LOOP, the code performs SJMP MAIN_LOOP. In a real-world application, this "dead time" would be replaced with other useful code, as the timing is handled entirely by hardware and the ISR.
- **Automatic Handling:** Unlike polling, the TF0 flag is automatically cleared by hardware when the CPU vectors to the ISR.

5. TIMER CALCULATION (12 MHz Crystal)

To generate a **50ms** delay, we must calculate the initial values for `TH0` and `TL0`.

1. **Machine Cycle**: 12 MHz/12=1 MHz (1μs per tick).
2. **Required Ticks**: 50,000μs/1μs=50,000 ticks.
3. **Initial Value**: 65,536−50,000=15,536.
4. **Hexadecimal**: 15,536=3CB0H.
   - `TH0 = 3CH`
   - `TL0 = B0H`

6. STACK BEHAVIOR SUMMARY

Feature	**Polling (polling.asm)**	**Interrupt-Driven (interrupt-driven.asm)**
Stack Activity	No automatic stack usage.	**PC** (2 bytes) pushed automatically on entry[11].
SP Change	SP remains at reset value ($07H$).	SP increments by 2 during ISR entry[12].
Return Method	SJMP or LJMP back to loop.	RETI instruction pops **PC** and restores execution[13131313].

7. CONCLUSION

This laboratory successfully demonstrated the practical differences between polling and interrupt-driven I/O.

Key Learnings:

1. **CPU Utilization:** Polling is a "blocking" operation that wastes CPU cycles, whereas interrupts enable "non-blocking" I/O management.
2. **Context Saving:** Interrupts rely on the **Stack** to preserve the Program Counter, necessitating proper Stack Pointer initialization (typically above $07H$ or $60H$ to avoid register banks)[14141414].
3. **Hardware Automation:** The 8051 hardware assists in interrupt-driven systems by automatically vectoring to the ISR and clearing flags, reducing the software overhead compared to manual polling.
4. **System Design:** For complex embedded systems with multiple sensors or timing requirements, interrupt-driven architecture is essential for maintaining real-time responsiveness.