Experiment-2

a) **Write a program to demonstrate the operation of timers in 8051 microcontroller to generate the delays using interrupts and without interrupts.**

Definition

A **timer** is a hardware unit inside the microcontroller that counts clock pulses.

- When it counts up to a certain value (overflow), it signals the CPU that a set time has passed.
- In the 8051, timers can also work as **counters** to count external events.

Key points for students

1. **Timer is for measuring time**
    - Just like a kitchen timer rings after a set time, the microcontroller's timer "rings" by setting a flag (TF0 or TF1).
2. **Works on clock pulses**
    - In 8051, the timer gets pulses from the system clock (usually crystal oscillator).
    - Example: With an **11.0592 MHz crystal**, one timer tick happens every **1.085 microseconds**.
3. **Modes of operation in 8051**
    - **Mode 0** → 13-bit timer
    - **Mode 1** → 16-bit timer
    - **Mode 2** → 8-bit auto-reload
    - **Mode 3** → Split timer mode
4. **Two main uses**
    - **Delay generation** (e.g., 5 ms, 1 second)
    - **Event counting** (if set to counter mode)
5. **Two ways to handle timer completion**
    - **Polling**: Keep checking the overflow flag until it's set (CPU is busy waiting).

- **Interrupts**: Let the timer interrupt the CPU when time is up (CPU can do other work meanwhile).

1. Toggling of p1.5 with the delay of 5ms without interrupts using timer 1

```
ORG 0000H
MOV TMOD, #10H   ; Timer1, Mode 1 (16-bit)
AGAIN:
   MOV TL1, #0FFH   ; Load low byte
   MOV TH1, #0EDH   ; Load high byte
   CPL P1.5         ; Toggle P1.5 pin
   SETB TR1         ; Start Timer1
WAIT:
   JNB TF1, WAIT    ; Wait until Timer1 overflows
   CLR TR1          ; Stop Timer1
   CLR TF1          ; Clear overflow flag
   SJMP AGAIN       ; Repeat
END
```

Step-by-Step for Students

1. **ORG 0000H**
   This tells the assembler: start the program at memory address 0.
2. **MOV TMOD, #10H**
   We are using **Timer1** in **Mode 1** (16-bit timer mode).
   - Mode 1 → Timer counts from the loaded value up to 65535 and then overflows.
   - #10H means Timer1 is set, Timer0 is not used.
3. **MOV TL1, #0FFH** & **MOV TH1, #0EDH**
   - TL1 = Low byte, TH1 = High byte of the timer.
   - Together (EDFFH), this is our **starting count**.

- Timer will count from EDFFH to FFFFH → this takes about **5 milliseconds**.

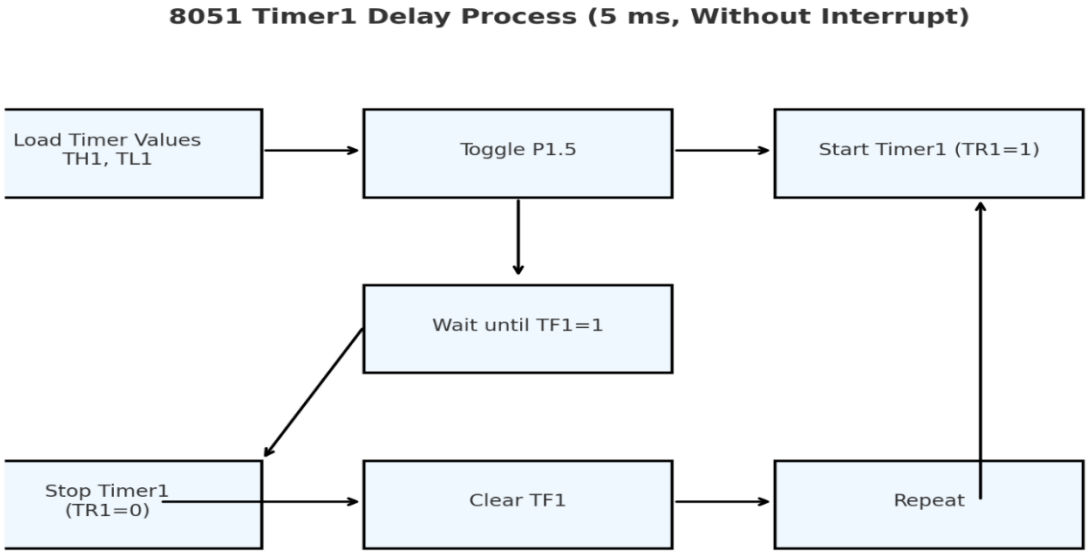
4. **CPL P1.5**
   - CPL = **Complement** (toggle).
   - If P1.5 was HIGH → it becomes LOW, and vice versa.
   - This creates the blinking/toggling effect.
5. **SETB TR1**
   - This starts Timer1.
6. **BACK: JNB TF1, BACK**
   - JNB = Jump if No Bit.
   - We keep checking **TF1** (Timer1 overflow flag).
   - If it is **0**, keep waiting here (busy wait).
   - When timer overflows, TF1 becomes **1**.
7. **CLR TR1**
   - Stop Timer1 after overflow.
8. **CLR TF1**
   - Clear the overflow flag so the next cycle works properly.
9. **SJMP AGAIN**
   - Jump back and do it again forever.

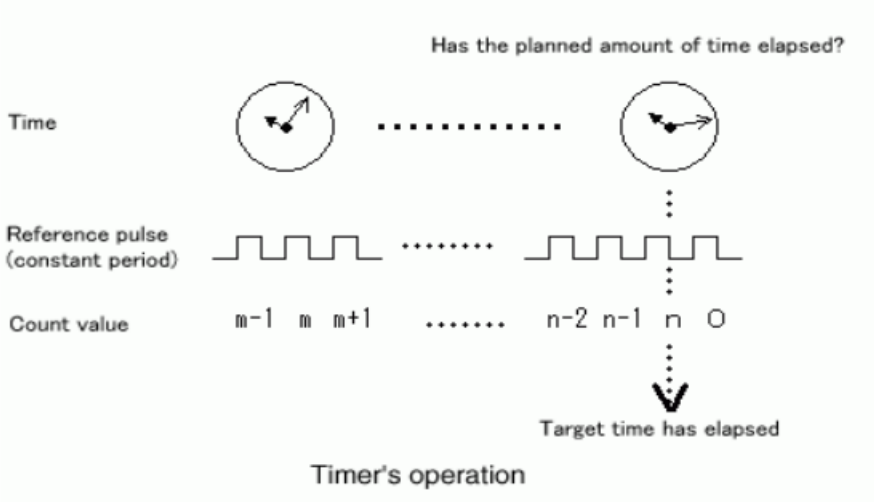
How this makes 5 ms delay

- 8051 with 11.0592 MHz crystal:
  1 machine cycle = 1.085 μs
- Counting from EDFFH to FFFFH = 4609 counts.
- Time = 4609 × 1.085 μs ≈ **5 ms**.

Every loop → Toggle pin → Wait 5 ms → Toggle again → Wait 5 ms → and so on.

- **Polling** wastes CPU time (here we are just waiting).





20250819-1518-51.9600971.mp4

Explanation of the Logic Analyzer Output

1. **Square Wave is Visible**
   - In your Logic Analyzer, you see a waveform with **regular ON/OFF pulses**.
   - That is because CPL P1.5 keeps toggling the pin every time the timer overflows.

- ON = High state, OFF = Low state.

2. **Delay Between Pulses**
   - Each high/low duration is decided by the **Timer1 overflow period**.
   - You loaded TH1 = 0EDH and TL1 = 0FFH.
   - This means the timer doesn't start from 0000H, but from **EDFFH**.
   - Timer counts from EDFFH → FFFFH before overflowing.
   - No. of counts = 65536 – 60927 = **4610 ticks**.
   - If 8051 runs at 12 MHz, then one machine cycle = 1 µs.
     So, delay ≈ 4610 µs = **4.61 ms**.
   - That's why your waveform has **~5 ms ON + 5 ms OFF = ~10 ms total cycle time**.
3. **P1 Line in Logic Analyzer**
   - You attached **P1.5** in Logic Analyzer, so you see toggling.
   - Each vertical line shows a toggle event (HIGH → LOW or LOW → HIGH).
4. **Flat Top and Bottom**
   - The flat parts of the waveform = stable HIGH or LOW output.
   - The transitions = where your CPL P1.5 executed.

2. Toggling of p1.2 with the delay of 5ms with interrupts using timer 0

```
ORG 0000H
LJMP MAIN
ORG 000BH        ; Timer0 ISR address
CPL P1.2         ; Toggle P1.2 whenever Timer0 overflows
MOV TL0, #0FFH   ; Reload Timer0 low byte
MOV TH0, #0EDH   ; Reload Timer0 high byte
RETI
```

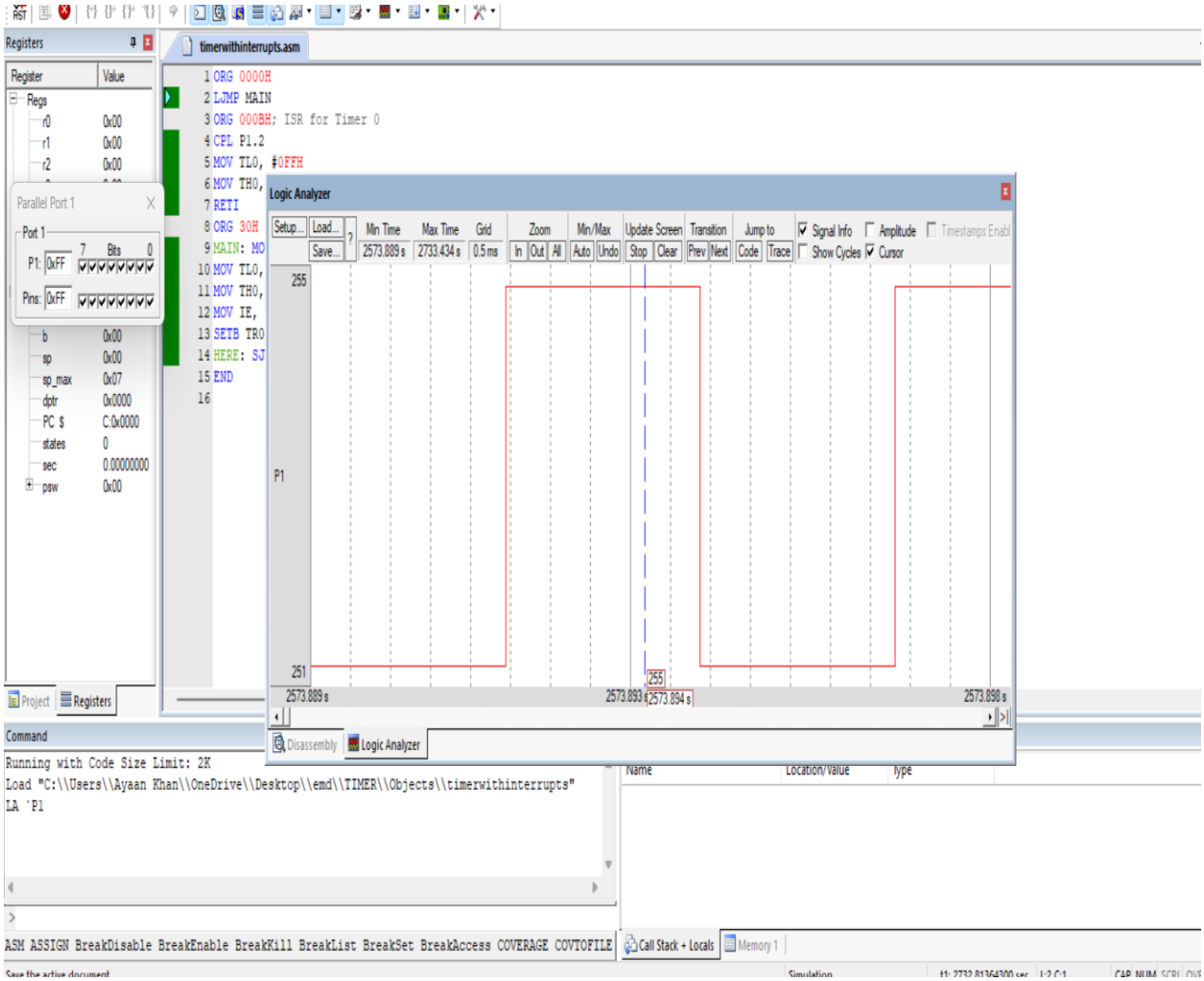
```
ORG 30H
MAIN:
   MOV TMOD, #01h ; Timer0, Mode 1 (16-bit)
   MOV TL0, #00   ; Initial value (low byte)
   MOV TH0, #0DCH ; Initial value (high byte)
   MOV IE, #82H   ; Enable Timer0 interrupt
   SETB TR0       ; Start Timer0

HERE: SJMP HERE    ; Infinite loop (CPU free)
END
```

Step-by-step explanation (very simple)

1. ORG 0000H — Put the first instruction at address 0x0000 (CPU reset address).
2. LJMP MAIN — Jump to the main program so CPU doesn't accidentally run the data/ISR area.
3. ORG 000BH — Place the Timer0 ISR at its fixed vector address 0x000B. The CPU goes here when Timer0 overflows and interrupts are enabled.

   (ISR = Interrupt Service Routine, the special code that runs automatically when an interrupt happens.)
4. CPL P1.2 — Toggle (flip) the pin P1.2. This makes the pin change from HIGH→LOW or LOW→HIGH on every interrupt.
5. MOV TH0, #0EDH / MOV TL0, #0FFH — Reload timer bytes inside ISR so the same time delay repeats after every interrupt.
6. RETI — End of ISR. Return to whatever the CPU was doing; also tell the CPU the interrupt is finished.
7. ORG 0030H — Put the main program code starting at address 0x0030 (keeps vectors at low addresses).
8. SETB P1.2 — Make sure P1.2 starts HIGH (so toggling is predictable).
9. MOV TMOD, #01H — Configure Timer0 in **Mode 1 (16-bit)**. (Lower nibble 0001 = Timer0 Mode1.)

10. MOV TH0, #0EDH & MOV TL0, #0FFH — Load the same 16-bit start value used in ISR so the first delay equals later delays.
11. MOV IE, #82H — Enable interrupts: EA=1 (global enable) and ET0=1 (Timer0 interrupt). Hex 82h = 1000 0010b.
12. SETB TR0 — Start Timer0 counting.
13. HERE: SJMP HERE — Main just loops; the ISR runs in the background on each timer overflow.
14. END — End of program source.



Working Logic

- TMOD = #01h → Timer0, Mode 1 → **16-bit timer**.
- TL0 & TH0 → preload values → timer starts from 0xDC00.
    - Max count of timer = 65536

- Initial value = DC00H = 56320
- Overflow after = 65536 – 56320 = 9216 counts.

👉 If XTAL = 11.0592 MHz:

- 1 machine cycle = 12 / 11.0592 MHz ≈ 1.085 µs
- Delay = 9216 × 1.085 µs ≈ **10 ms** per interrupt.
- Inside ISR: CPL P1.2 → toggles P1.2 on each overflow.
  - So P1.2 is HIGH for 10 ms, LOW for 10 ms → **20 ms total period**.
  - That means frequency ≈ **50 Hz square wave**.

---

3. Logic Analyzer (Right Side of Screenshot)

- The red waveform is **P1.2**.
- It is toggling between **0 and 1** → square wave output.
- Each high pulse = ~10 ms, each low pulse = ~10 ms.
- This matches the calculation above.

So the waveform confirms:
✅ Timer0 overflows → generates interrupt → ISR toggles P1.2.
✅ Square wave of ~50 Hz is observed.

---

4. Registers & Port Window (Left Side of Screenshot)

- **Port 1 shows FF (all high)**, but P1.2 is being toggled, so Logic Analyzer shows it clearly.
- Registers window confirms timer registers (TH0/TL0) are running.
- IE = 82H → Timer0 interrupt enabled.
- TR0 set → Timer0 running.

B. Time delay Generation Using Timers of 8051.

1. Count no. 0f pulses without interrupts using counter 0

```
ORG 0000H
MOV TMOD, #05h
SETB P3.4
MOV TL0, #00H
MOV TH0, #00H
SETB TR0
BACK: MOV A, TL0
MOV P1, A
MOV A, TH0
MOV P2, A
SJMP BACK
END
```

Explanation:

Step-by-Step Explanation

1. Program start

```
ORG 0000H
```

- This tells the assembler: "Start putting my code at memory address 0000H."

---

2. Set Timer/Counter mode

```
MOV TMOD, #05H
```

- TMOD = Timer Mode register.
- #05H means:
  - **Counter 0**
  - **Mode 1** (16-bit mode)
- In Mode 1, Counter 0 counts from 0000H to FFFFH, then overflows.

- **Counter** mode counts **external pulses** on pin **P3.4 (T0)** instead of counting CPU cycles.

---

3. Configure input pin

SETB P3.4

- Sets P3.4 as input (logic high) so pulses can come into **T0**.

---

4. Clear the counter registers

MOV TL0, #00H ; Low byte = 0

MOV TH0, #00H ; High byte = 0

- This makes sure the counter starts from **0**.

---

5. Start the counter

SETB TR0

- TR0 = Timer/Counter 0 Run Control bit.
- Setting it starts counting **external pulses** coming on **P3.4**.

---

6. Main loop – keep reading the count

BACK: MOV A, TL0

MOV P1, A

MOV A, TH0

MOV P2, A

SJMP BACK

- **MOV A, TL0** → Copy low byte of the count into **Accumulator**.
- **MOV P1, A** → Send low byte to **Port 1** (shows on LEDs or debugger).
- **MOV A, TH0** → Copy high byte of the count.
- **MOV P2, A** → Send high byte to **Port 2**.

- **SJMP BACK** → Keep repeating forever.