XMUT202 Digital Electronics

8051 Assembly Languages

Week 13 Lecture 1

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Today's topic

- Polling
- Polling vs. interrupt.
- Interrupt mechanism
- Interrupt organisation.
- 8051 interrupts.
- Interrupt vector table in 8051.
- Interrupt priority.
- Examples of interrupts (e.g. reset, timer, and external)

Polling

Polling involves the main program loop constantly checking the status of a Peripheral, such as a flag or a pin, until an event happens.

For example, if you want to know whether a timer has reached a specific count, the program repeatedly reads the Timer 0 flag (TF0) register.

How does it work in the 8051?

Typically, use instructions like `RC R7, TF0` (Read the TF0 register into register R7) and then a conditional jump (`CJNE`) to determine if the flag is set. If it's set, you'd execute the code associated with that event.

Polling

Example (Timer 0):

Imagine you want to use Timer 0 to generate an interrupt when it reaches 64 counts. Polling would mean the main loop **always** checks the flag. This consumes CPU time regardless of whether the timer has reached its count.

Drawbacks: Polling is inefficient because the CPU is constantly busy checking status, which can lead to high CPU utilisation and slow down the system.

Polling

The program waits until Timer 0 overflows.

```
MOV TMOD, #01H      ; Timer0 Mode1 (16-bit timer)

MOV TH0, #0FCH      ; Load high byte
MOV TL0, #018H      ; Load low byte

SETB TR0            ; Start Timer0

WAIT: JNB TF0, WAIT ; Keep checking TF0 flag

CLR TR0             ; Stop timer
CLR TF0             ; Clear overflow flag
```

Polling

The program waits until Timer 0 overflows.

```
MOV TMOD, #01H      ; Timer0 Mode1 (16-bit timer)
```

1. Configure Timer 0

   This sets:
   - Timer 0
   - Mode 1 (16-bit timer)

TMOD register: 0000 0001B
Mode 1 = 16-bit timer.

Polling

The program waits until Timer 0 overflows.

```
MOV TH0, #0FCH      ; Load high byte
MOV TL0, #018H      ; Load low byte
```

2. Load Initial Timer Value

These values are loaded into:

- TH0 = high byte
- TL0 = low byte

Timer begins counting from:

FC18H

up to:

FFFFH

Then overflow occurs.

Polling

The program waits until Timer 0 overflows.

```
SETB TR0              ; Start Timer0
```

3. Start Timer

TR0 = Timer Run control bit.

Setting it to 1 starts Timer 0.

Polling

The program waits until Timer 0 overflows.

```
WAIT: JNB TF0, WAIT ; Keep checking TF0 flag
```

4. Polling Process
This is the polling instruction.
Meaning:
Jump back to WAIT if TF0 = 0

The CPU continuously checks the TF0 flag.

Remember what TF0 is?
TF0 = Timer 0 Overflow Flag
•TF0 = 0 → timer still counting
•TF0 = 1 → timer overflow happened
So the processor keeps looping until overflow occurs.

Polling

The program waits until Timer 0 overflows.

```
CLR TR0              ; Stop timer
CLR TF0              ; Clear overflow flag
```

Polling

polling an input pin instead of a timer flag.

```
WAIT: JB P1.0, WAIT
MOV A, #55H
```

Continuously check pin P1.0If P1.0 = 1, keep waiting
When P1.0 becomes 0, execute next instruction

Polling

in 8051, polling very commonly uses:
•JB → Jump if Bit = 1
•JNB → Jump if Bit = 0
because polling usually means:
repeatedly checking a single flag bit or pin.

Instruction	Meaning
JB bit,label	Jump if bit = 1
JNB bit,label	Jump if bit = 0

Most status indicators in 8051 are single bits, such as:
•TF0 (Timer overflow flag)
•RI (Receive interrupt flag)
•TI (Transmit interrupt flag)
•P1.0 (input pin)
So instructions that directly test bits are very useful.

Polling

Other Instructions Sometimes Used in Polling

Although JB/JNB are most common, polling can also use:

Instruction	Usage
JZ	Jump if accumulator = 0
JNZ	Jump if accumulator ≠ 0
CJNE	Compare and jump if not equal
DJNZ	Decrement and jump if not zero

Polling

Example Using JNZ

```
MOV A, P1
WAIT: JNZ WAIT
```

- Continuously read P1 again and again
- Wait until all buttons are pressed (00H)

Polling vs. Interrupt

Interrupts provide a much more efficient way to handle events. Instead of the program constantly checking, the 8051 monitors for a specific signal (usually a change in a flag register). When the signal occurs, the 8051 immediately suspends its current task and jumps to a special routine called an Interrupt Service Routine (ISR).

Example (Timer 0): configure Timer 0 to generate an interrupt on overflow. When the timer reaches 64 counts, the T0F flag is set. The 8051 automatically jumps to the Timer 0 ISR, which resets the T0F flag and executes the code to handle the timer period.

Advantages: Interrupts are much more efficient because the CPU only responds when an event occurs. This reduces CPU utilisation and improves overall system performance.

Polling vs. Interrupt

Key Differences

Polling: The main loop always checks the status. This consumes CPU time even when the device isn't actively changing its state.

Interrupts: The 8051 only responds when the interrupt signal occurs. The CPU can continue executing other tasks while waiting for the interrupt.

Why Interrupts are *Generally Better* in the 8051:

The 8051 has limited processing power. Polling would quickly become a bottleneck, especially with multiple peripherals. Interrupts allow the 8051 to efficiently handle multiple events without constantly checking each device individually.

Blocking Code

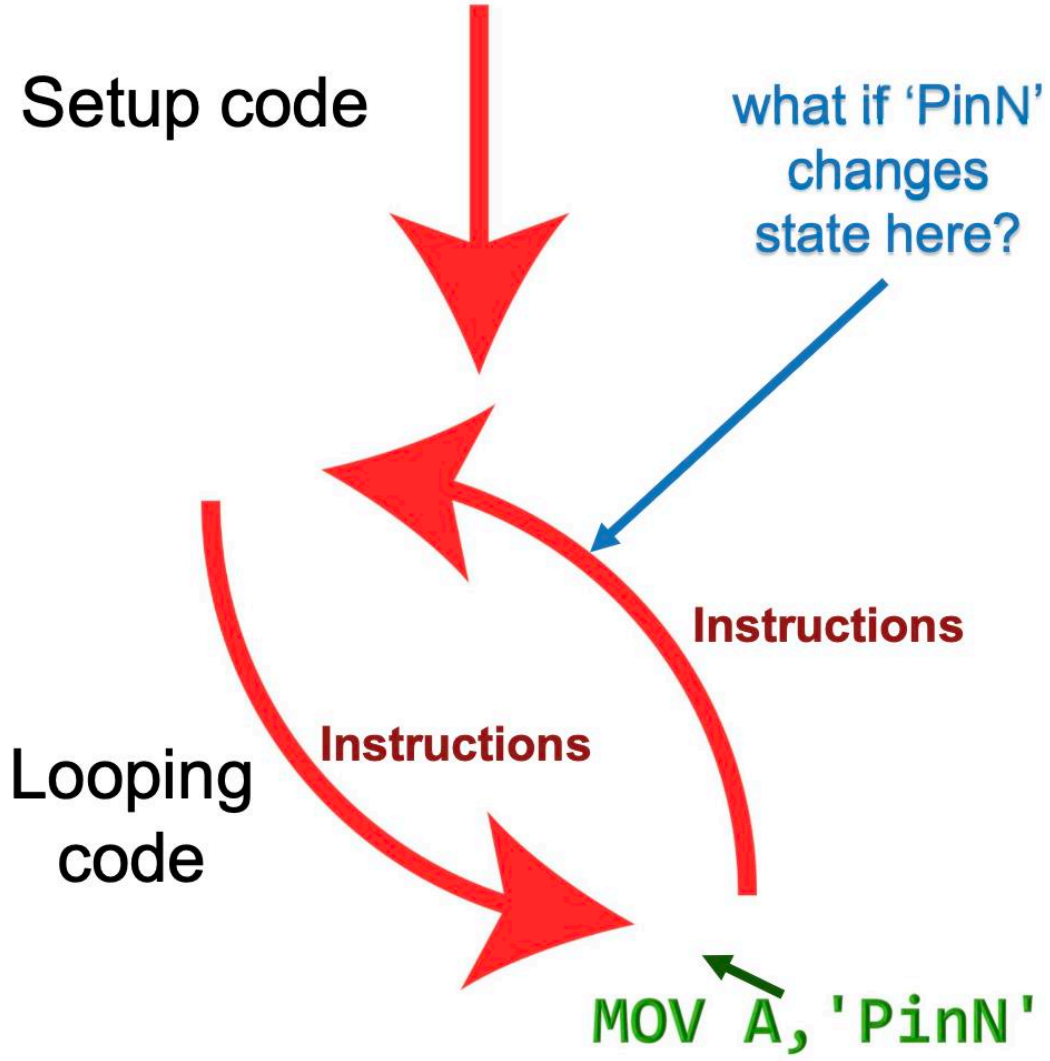
Blocking code refers to a sequence of instructions that halts the execution of the main program until a specific event occurs. Essentially, the processor sits idle, waiting for something to happen.

The 8051, like many microcontrollers, doesn't have a built-in mechanism for truly "blocking" a process in the same way a high-level language might.
However, you can simulate blocking behaviour using **polling**.

Example:
Imagine you want to read data from a serial port. If the serial port isn't ready to send or receive data, the 8051 will wait (block) until data is available.

Problems with Blocking: Excessive blocking can severely degrade system performance. If the event you're waiting for is infrequent, the processor spends much of its time idle, wasting resources. It can also lead to missed deadlines if the event takes longer than expected.

Blocking Code



- What if 'PinN' changes during one of the blocks of instructions?
- And what if it's *very* important that we not miss 'PinN' changing state?
- We might solve this by configuring an interrupt to call some code and leave the pre- defined instructions whenever the pin changes.

Blocking Code

PinN changes during one of the blocks of instructions

This describes a situation where a peripheral's output pin (PinN) changes state while a sequence of instructions is being executed. This can cause problems if the code isn't designed to handle these unexpected changes.

Why? The 8051's registers and memory locations are accessed in a specific order.
If a peripheral changes its output pin during the middle of a read or write operation, it can corrupt data, cause unpredictable behaviour, or even crash the program.

Example: Let's say you're reading a value from a port register (e.g., Port 2) while a button connected to Port 2 is being pressed. A button press could cause Port 2 to change state during the read operation, resulting in an incorrect value being read.

Blocking Code

Solutions

The best solution is to use interrupts. Interrupts allow the 8051 to respond to the button press immediately without waiting for the current instruction to complete.

Careful Code Design

If you must use polling, you need to design your code to be robust against unexpected changes. This might involve:

- Short, fast loops.
- Careful timing considerations.
- Using techniques like "flushing" registers to ensure data is written correctly before a read. (This is more advanced and requires a deep understanding of the 8051 architecture.)

Polling vs. Interrupt

Feature	Polling	Interrupts
Definition	Repeatedly checking device status	Signal halting a program for a task
Best Suited For	Small systems, few devices	Larger systems, multiple devices
Processor Usage	High – Constant checking	Low – Only when triggered
System Performance	Can degrade – Increased load	Improves – Reduced load
Efficiency	Less efficient	More efficient

Example

This example will demonstrate the concepts of interrupts, polling, and external input, illustrating the potential issues we've been discussing.

Project: Simple Digital Thermometer with 8051

Hardware:

- 8051 Microcontroller
- LM35 Temperature Sensor (Analog-to-Digital Converter - ADC)
- LCD Display (e.g., 16x2 LCD)
- Push Button (to reset the display)

Example

Software (Conceptual Outline):

1. Initialisation:

- Configure the 8051's clock.
- Initialise the LCD display.
- Configure the ADC to read the sensor.
- Enable the ADC.
- Enable the LCD.
- Disable interrupts initially.

2. Main Loop (Polling - Initially):

- Read the ADC value from the LM35 sensor.
- Convert the ADC value to a temperature reading (using the LM35's datasheet).
- Display the temperature on the LCD.
- Brief delay (e.g., 10ms) to avoid overwhelming the LCD.

3. Interrupt Service Routine (ISR) – Button Press:

- When the reset button is pressed, the interrupt triggers.
- Reset the ADC.
- Disable the LCD.
- Re-enable interrupts.
- Reset the temperature display on the LCD.

Example

How does it relate to Our Concepts?

Polling: The `MAIN_LOOP` continuously polls the ADC to get the temperature reading.

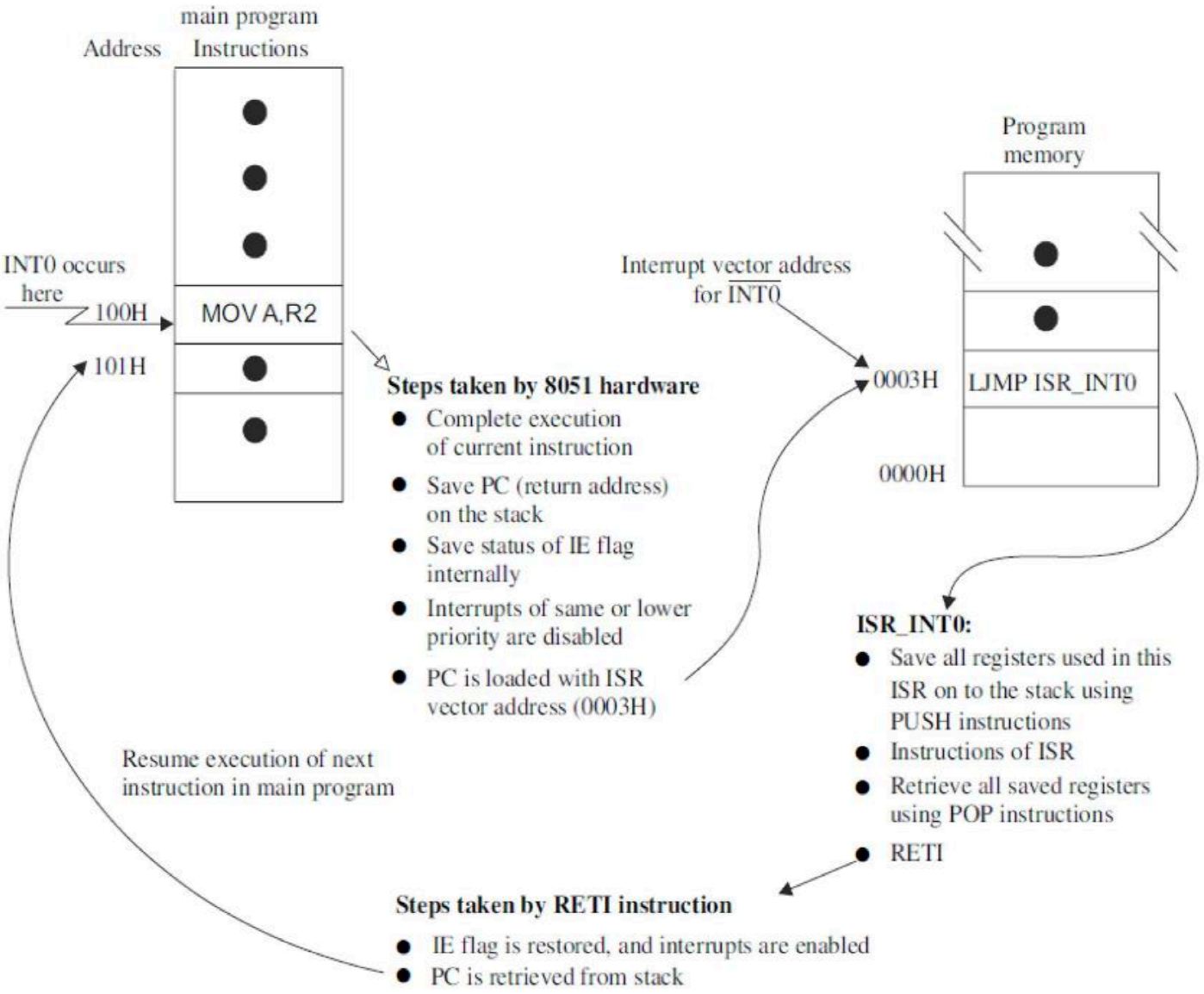
Interrupts: The `ISR_BUTTON` handles the reset button press, demonstrating how an interrupt can quickly disable the polling loop and reset the display.

Potential Issues:

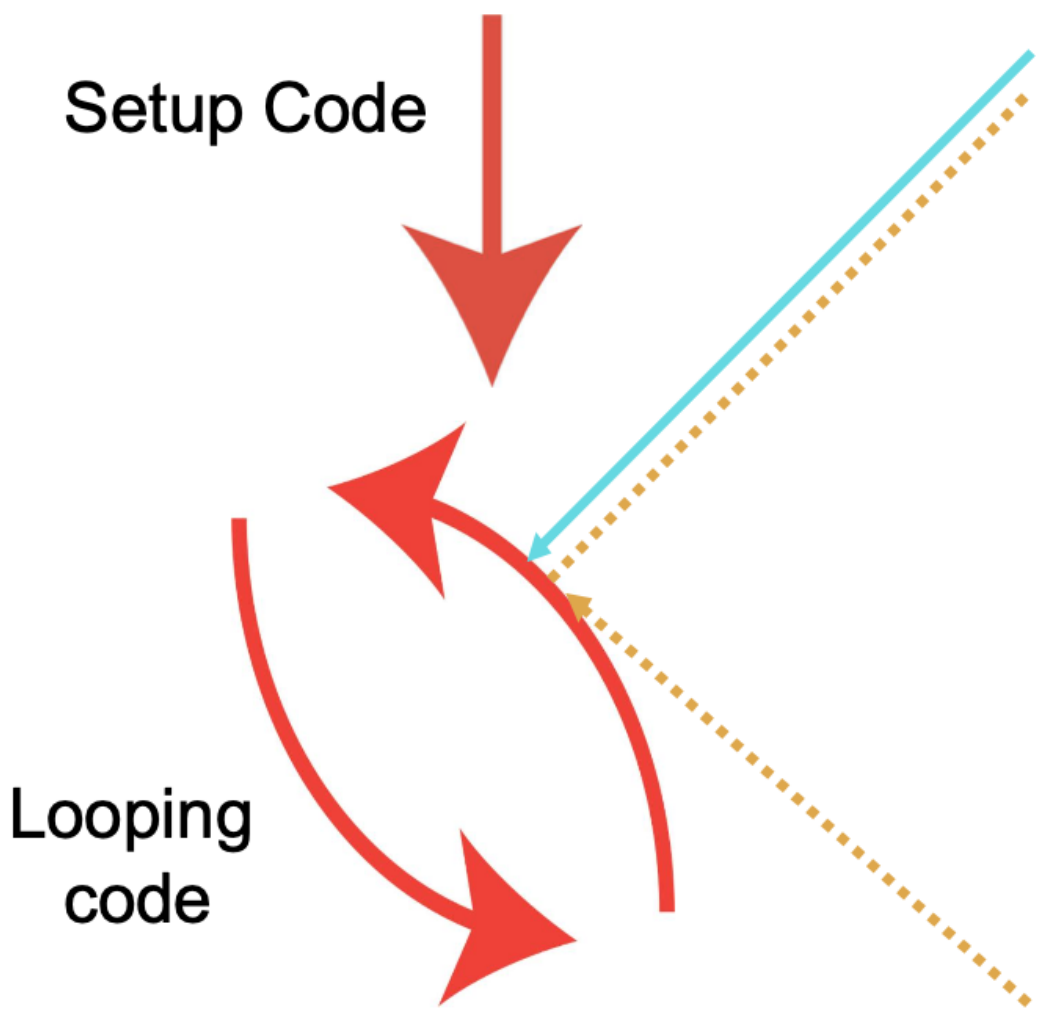
If the ADC reading takes a long time, the main loop could block, and the LCD might not update correctly. The button-press interrupt is crucial to prevent this.

Interrupt

- **How an interrupt works in the 8051 microcontroller**, specifically for **External Interrupt 0 (INT0)**.



Interrupt-driven code



'PinN' changes state here

- Interrupt flag raised
- Complete current instruction
- Save program counter address of next inst.
- Load interrupt vector into program counter
- Service interrupt (ISR = interrupt service routine), AKA 'handler'
- Return to loop (restore program counter), RETI

Interrupt Mechanism

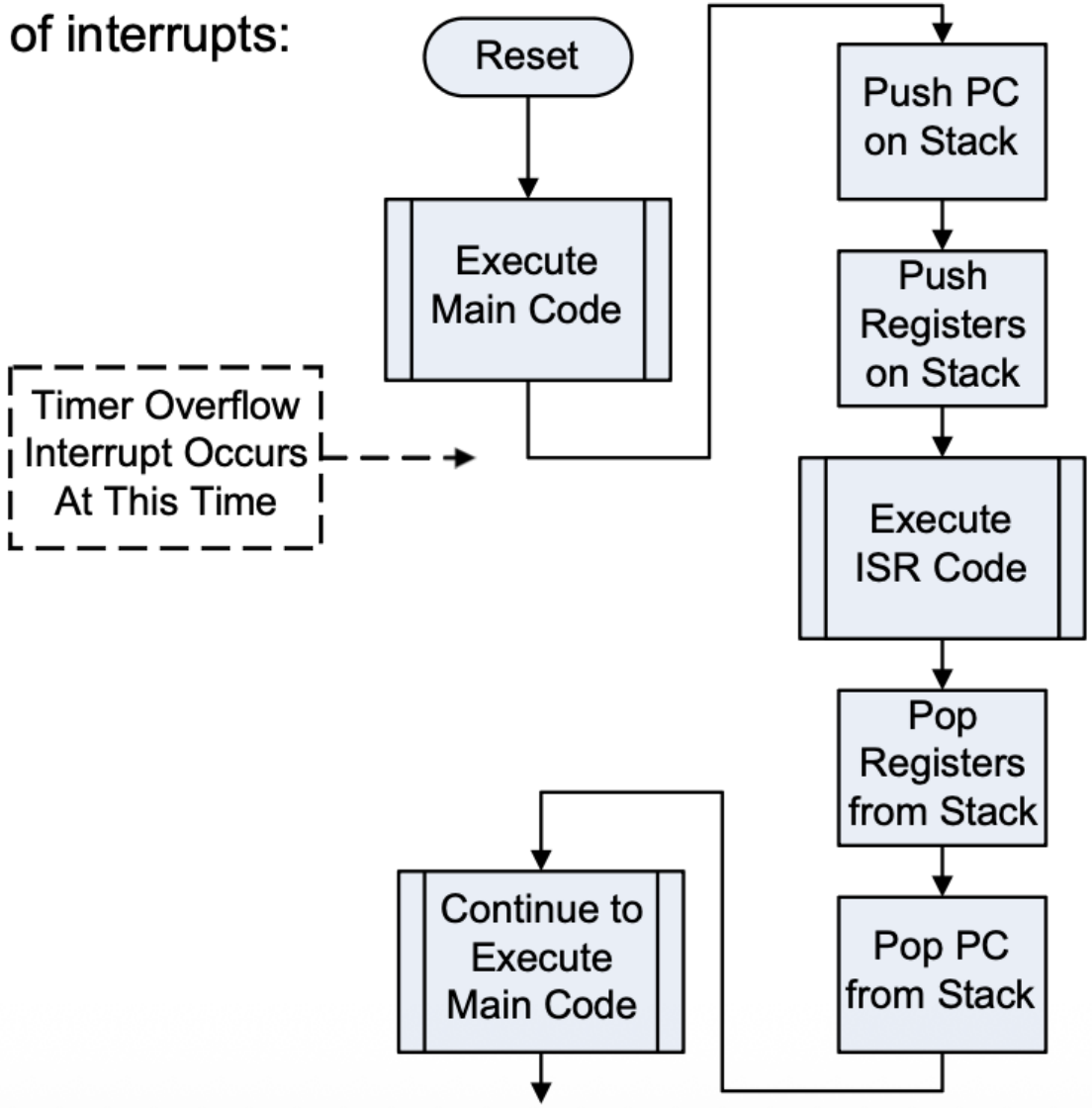
- An interrupt is the occurrence of a condition that causes a temporary suspension of a program while the condition is serviced by another (sub) program
- Interrupts are important because they allow a system to respond asynchronously to an event and deal with the event while in the middle of performing another task.
- An **interrupt-driven system** gives the illusion of doing many things simultaneously
- The (sub) program that deals with an interrupt is called an **interrupt service routine (ISR) or interrupt handler.**

Interrupt Mechanism

- The ISR executes in response to the interrupt and generally performs an input or output operation to a device
- When an interrupt occurs, the main program temporarily suspends execution and branches to the ISR
- The ISR executes, performs the desired operation, and terminates with a “return from interrupt” (RETI) instruction.
- The RETI instruction is different from the normal “RET” instruction.

Interrupt Mechanism

Mechanism of interrupts:



Interrupt Organisation

All interrupts are disabled after a system reset and enabled individually by software

- Interrupts allow special routines to be run when the microcontroller enters certain states.
- We'll discuss the 8051-specific states below, but interrupts are often associated with microcontrollers' hardware peripherals
- When some hardware peripheral (timer, serial port, external I/O port, etc.) changes state, main program flow can be set to be interrupted to handle this change in a timely manner.

Interrupt Organisation

- Interrupts disrupt the running of normal code.
- It is important to follow best practices with interrupts to keep from disrupting normally-running code:
- Keep interrupt service routines short.
- Try to avoid calling subroutines within your interrupt service routine.
- Ask what might happen if an interrupt is itself interrupted…
  - Interrupts are handled in order of a defined priority.
  - Consider interrupt priority: which interrupt is most important?
- If particularly time-sensitive code is used (e.g., bit-banging emulation of a serial port), consider disabling interrupts during this routine.

Interrupt Organisation

- Re-enable after this routine is finished.
- The ISR executes in response to the interrupt and generally performs an input or output operation on a device
- When an interrupt occurs, the main program temporarily suspends execution and branches to the ISR
- The ISR executes, performs the desired operation, and terminates with a “return from interrupt” (RETI) instruction
- The RETI instruction is different from the normal “RET” instruction

8051 Interrupts

The original 8051 has four interrupt types: *Reset, Timer, External, and Serial*

1. Reset: When reset , PC is loaded with ROM address 0x0000. Unlike other interrupts, prior addresses aren't loaded to the stack, etc.

2. Timer Interrupts: one interrupt for Timer 0 and one for Timer 1.

Timer 0 interrupt: PC is vectored to 0x000B

Timer 1 interrupt: PC is vectored to 0x001B

8051 Interrupts

3. External Interrupts: Interrupts that are called in response to external signals.

External interrupt 0 - INT0 (AKA EXT1): PC vectored to ROM 0x0003.

External interrupt 1 - INT1 (AKA EXT2): PC vectored to ROM 0x0013.

4. Serial Interrupt: An interrupt that is called in response to serial events.

Serial interrupt: PC is vectored to ROM 0x0023.

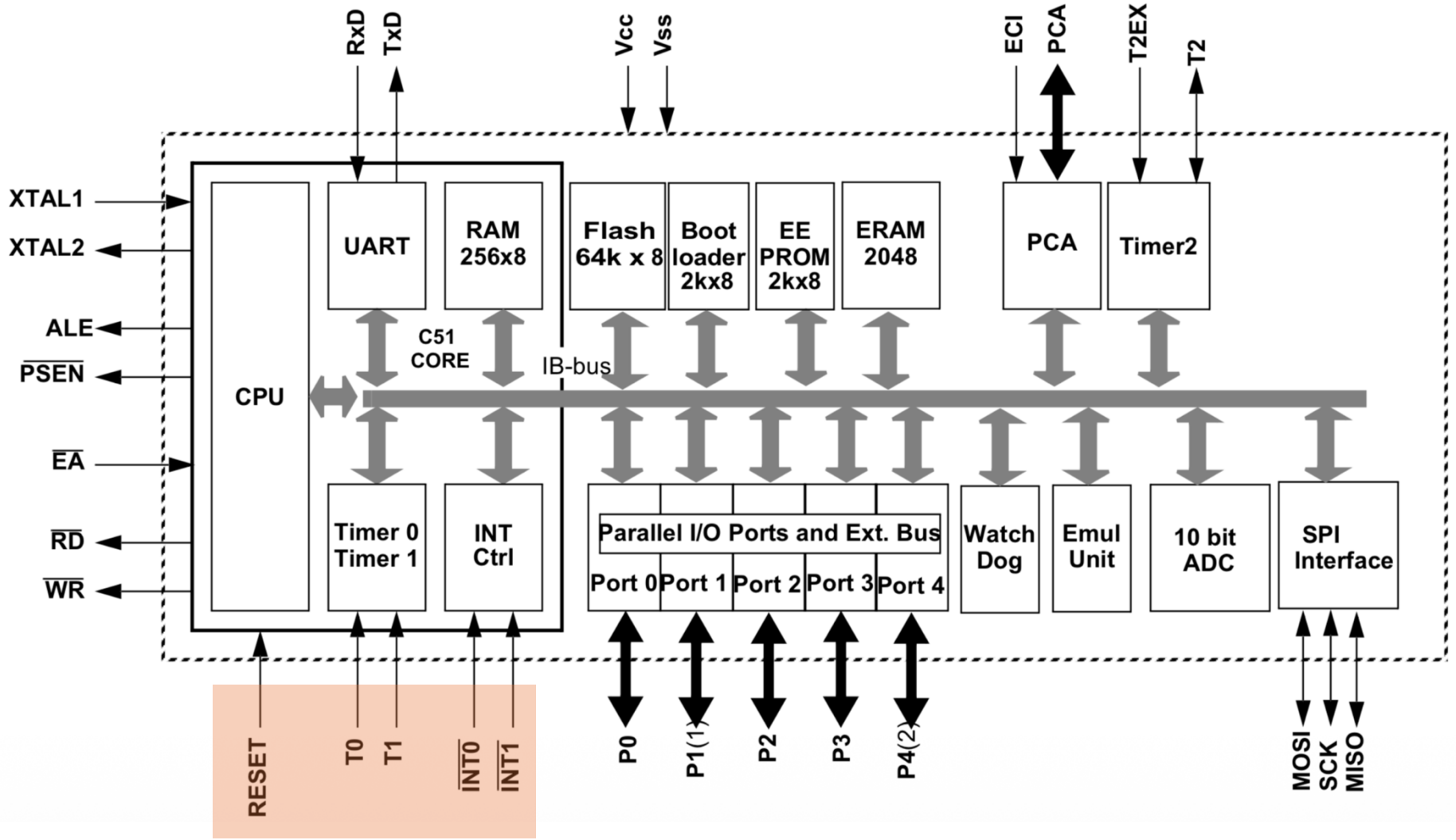
The C8051F020x has additional interrupts; we'll look at these later.

If the ISR is short, it can fit in the 8 bytes allocated to the interrupts.

If the ISR is longer, the vector address holds an LJMP instruction that points to another memory location, allowing for a longer ISR.

8051 Interrupts

In 8051, these interrupts of RESET, TIMER (Timer 0 and Timer 1), and EXTERNALS (INT0 and INT1) are stored in the microcontroller architecture.



Interrupt Vector Table

Details of type of interrupts in 8051.

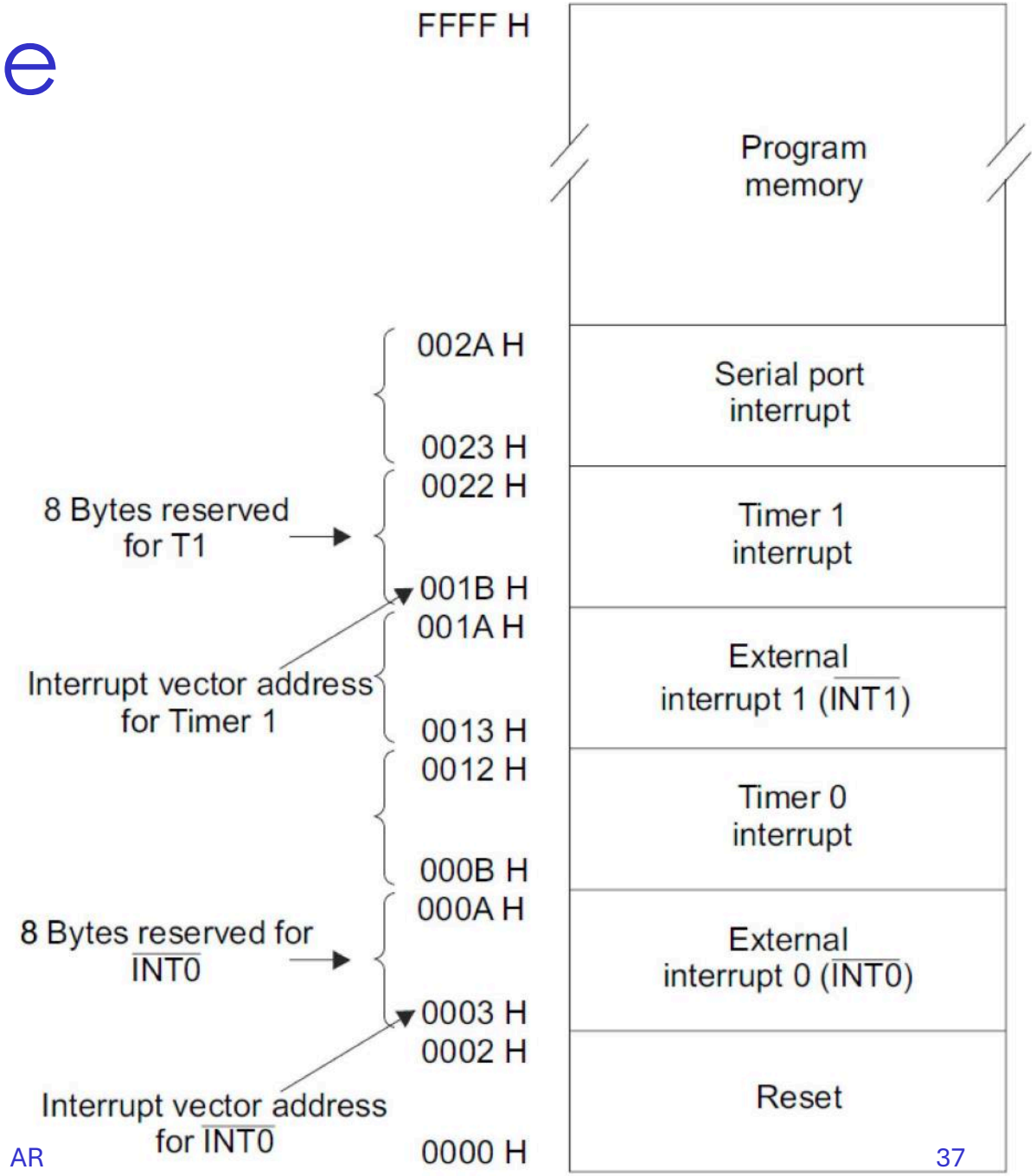
Interrupt	ROM ADDRESS (for start of ISR)	INTERRUPT PIN	Notes
RESET	0x0000H	9	Note only 3 bytes of ROM between this and INT0. See example, next slide. Interrupt flag auto-clears.
INT0	0x0003H	P3.2	Interrupt flag auto-clears.
Timer0	0x000BH	N/A	Interrupt flag auto-clears
INT1	0x0013H	P3.3	Interrupt flag auto-clears
Timer1	0x001BH	N/A	Interrupt flag auto-clears
Serial Port	0x0023H	N/A	Software must clear this interrupt flag.

Interrupt Vector Table

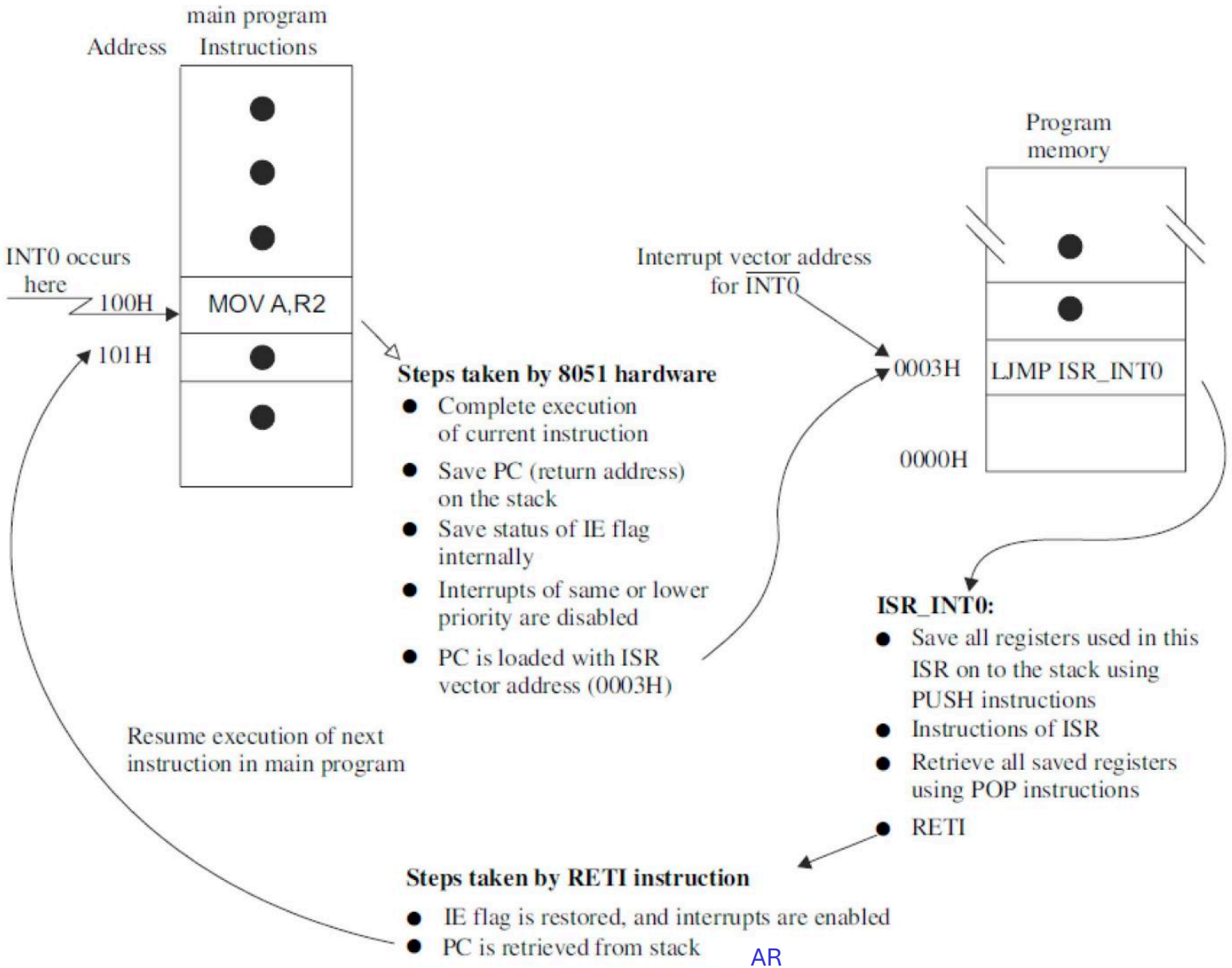
Location of interrupts in

8051 microcontroller:

- Reset: 00H – 02H
- External interrupt 0 (INT0): 03H – 0AH
- Timer 0: 0BH – 012H
- External interrupt 1, (INT1): 13H – 1AH
- Timer 1: 1BH – 22H
- Serial port: 22H – 23H



Interrupt Vector Table



Example 1: Reset

- Note the low ROM addresses for the interrupt vector table addresses.
- Also, note how Reset has only three bytes of ROM (0x0000-0x0002).
- If we're not careful, our PC will just step through the other ISR's and execute them. Instead, we must quickly LJMP past these ISR addresses into an unused memory space.
- We can use the ORG directive to tell our assembler to assemble the code to make sure that we bypass this vector table.

```
ORG 0          ;This is our RESET address.
LJMP BODY      ;We jump past the interrupt vector table addresses

ORG 30H        ;We use org directive to associate the following
               ; instructions with addresses past 30H

BODY:
;Our non-ISR code goes here, since we set ORG past our ISR
;vector table addresses
```

Reset.asm

The IE (IEN0) Register

IE stands for Interrupt Enable. IEN0 is the 8-bit register that controls which interrupt sources are enabled for the 8051 microcontroller. It's located at address `0FF4H` in the 8051's memory map.

Bit Mapping: Each bit in the IE register corresponds to a specific interrupt source.

Bit	Interrupt Source	Address(es) Triggering the Interrupt
0	Timer 0 Overflow	Timer 0 Control Register (TCON)
1	Serial Port 1 (SP1)	Serial Port 1 Control Register (SCON)
2	Timer 1 Overflow	Timer 1 Control Register (TCON)
3	Serial Port 2 (SP2)	Serial Port 2 Control Register (SCON)
4	External Interrupt 0	External Interrupt 0 Control Register (IE0)
5	External Interrupt 1	External Interrupt 1 Control Register (IE1)
6	Timer 1 Overflow	Timer 1 Control Register (TCON)
7	Timer 0 Overflow	Timer 0 Control Register (TCON)

The IE (IEN0) Register: How does it Work

1. Enabling Interrupts: When a bit in the IE register is set to ‘1’, it enables the corresponding interrupt source. This means that the 8051 will respond to events triggered by that source.

2. Interrupt Trigger: When an interrupt source (e.g., Timer 0 reaching its maximum count) occurs,

it sets a flag in its associated control register (e.g., T0F in the Timer 0 Control Register).

3. Interrupt Vectoring: The 8051’s program counter (PC) is temporarily overwritten with the interrupt vector address for the enabled interrupt. The interrupt vector address is the starting address of the Interrupt Service Routine (ISR) for that specific interrupt.

4. ISR Execution: The 8051 jumps to the ISR, executes the code in the ISR, and then returns to the main program.

5. Disabling Interrupts: When the ISR finishes, the 8051 typically clears the flag in the control register that triggered the interrupt. The 8051 can also *disable* interrupts entirely by setting the

IE bit to ‘0’. This is done to prevent unexpected interrupts from interfering with critical code

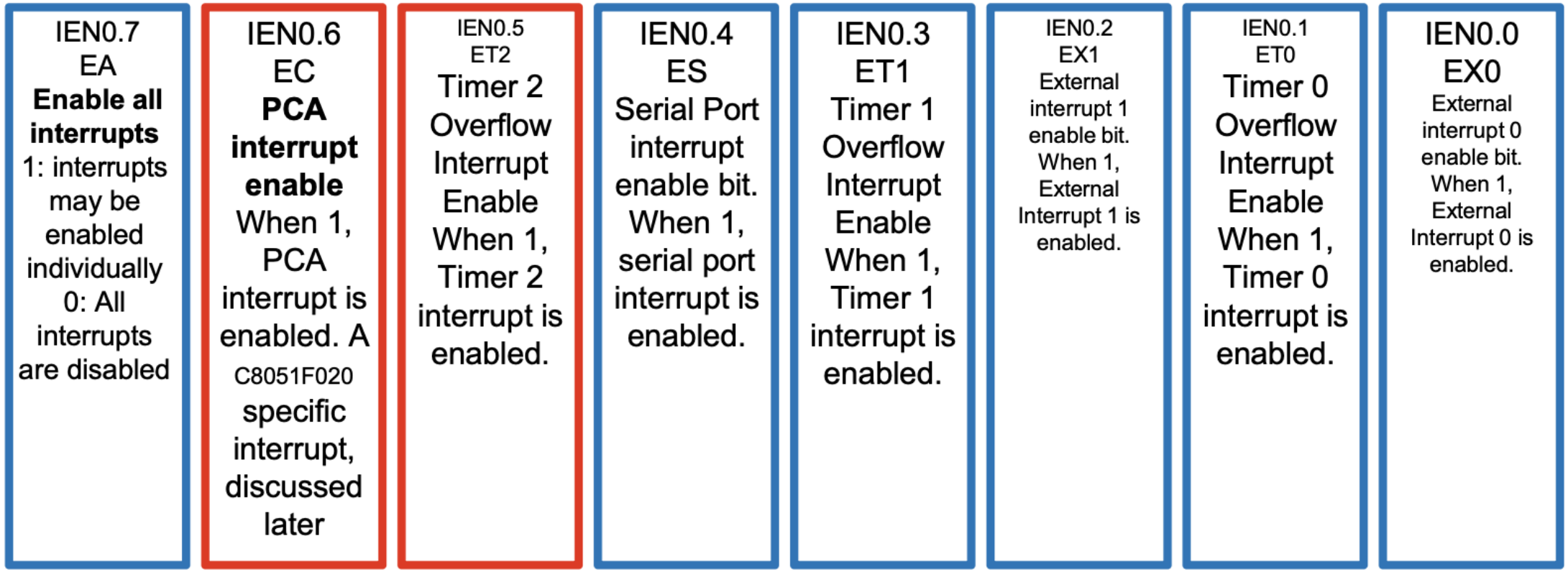
execution.

Note: The IE register is often controlled by the IEN bit in the Control Register (RC7). Setting the IEN bit enables all interrupts, while clearing it disables all interrupts.

The IE (IEN0) Register: How does it Work

The special function register that allows for interrupt control is the IE register. IE = interrupt enable.

- Allows for bit-by-bit control of each individual interrupt as well as all interrupts at once.
- Original 8051's have a single IE register; the C8051F020x has two.
- We'll focus on the first one, called IEN0 on the C8051F020x.



Interrupt Enable Registers

IEN0 (S:A8h)
Interrupt Enable Register

7	6	5	4	3	2	1	0
EA	EC	ET2	ES	ET1	EX1	ET0	EX0

Bit Number	Bit Mnemonic	Description
7	EA	**Enable All Interrupt bit** Clear to disable all interrupts. Set to enable all interrupts. If EA=1, each interrupt source is individually enabled or disabled by setting or clearing its interrupt enable bit.
6	EC	**PCA Interrupt Enable** Clear to disable the PCA interrupt. Set to enable the PCA interrupt.
5	ET2	**Timer 2 Overflow Interrupt Enable bit** Clear to disable Timer 2 overflow interrupt. Set to enable Timer 2 overflow interrupt.
4	ES	**Serial Port Enable bit** Clear to disable serial port interrupt. Set to enable serial port interrupt.
3	ET1	**Timer 1 Overflow Interrupt Enable bit** Clear to disable timer 1 overflow interrupt. Set to enable timer 1 overflow interrupt.
2	EX1	**External Interrupt 1 Enable bit** Clear to disable external interrupt 1. Set to enable external interrupt 1.
1	ET0	**Timer 0 Overflow Interrupt Enable bit** Clear to disable timer 0 overflow interrupt. Set to enable timer 0 overflow interrupt.
0	EX0	**External Interrupt 0 Enable bit** Clear to disable external interrupt 0. Set to enable external interrupt 0.

Reset Value = 0000 0000b
bit addressable

Interrupt Priority

- If we're implementing an interrupt-rich application, we need to consider what happens if multiple interrupts arrive at the same time (interrupt during ISR).
- Older 8051's had less flexible interrupt priority control.
- For more on original 8051 priority, see https://what-when-how.com/8051-microcontroller/interrupt-priority-in-the-805152/

- The C8051F020x has a much more modern interrupt priority configuration capability, which we'll focus upon.
- Upon boot-up, the interrupts are given default priority hierarchy.
- We can change this priority hierarchy by manipulating two SFR's.
- IPL0 and IPH0 registers.

Interrupt Priority

Decreasing Priority

By default, the interrupt priority is:

1. External interrupt 0 (INT0)
2. Timer 0 (TF0)
3. External interrupt 1 (INT1)
4. Timer 1 (TF1)
5. Programmable counter array (CF)
6. Serial UART (RI or TI)
7. Timer 2 (TF2)
8. ADC *(discussed later)* (ADCI)
9. SPI

Setting Interrupt Priority

Each interrupt has two bits that can be used to set its priority.

- These bits are in the IPH0 and IPL0 registers.
- These two bits give four possible levels of interrupt priority.

If two interrupts are configured with the same priority level, then they are handled in order of their ranking in the default priority level scheme (prev. slide).

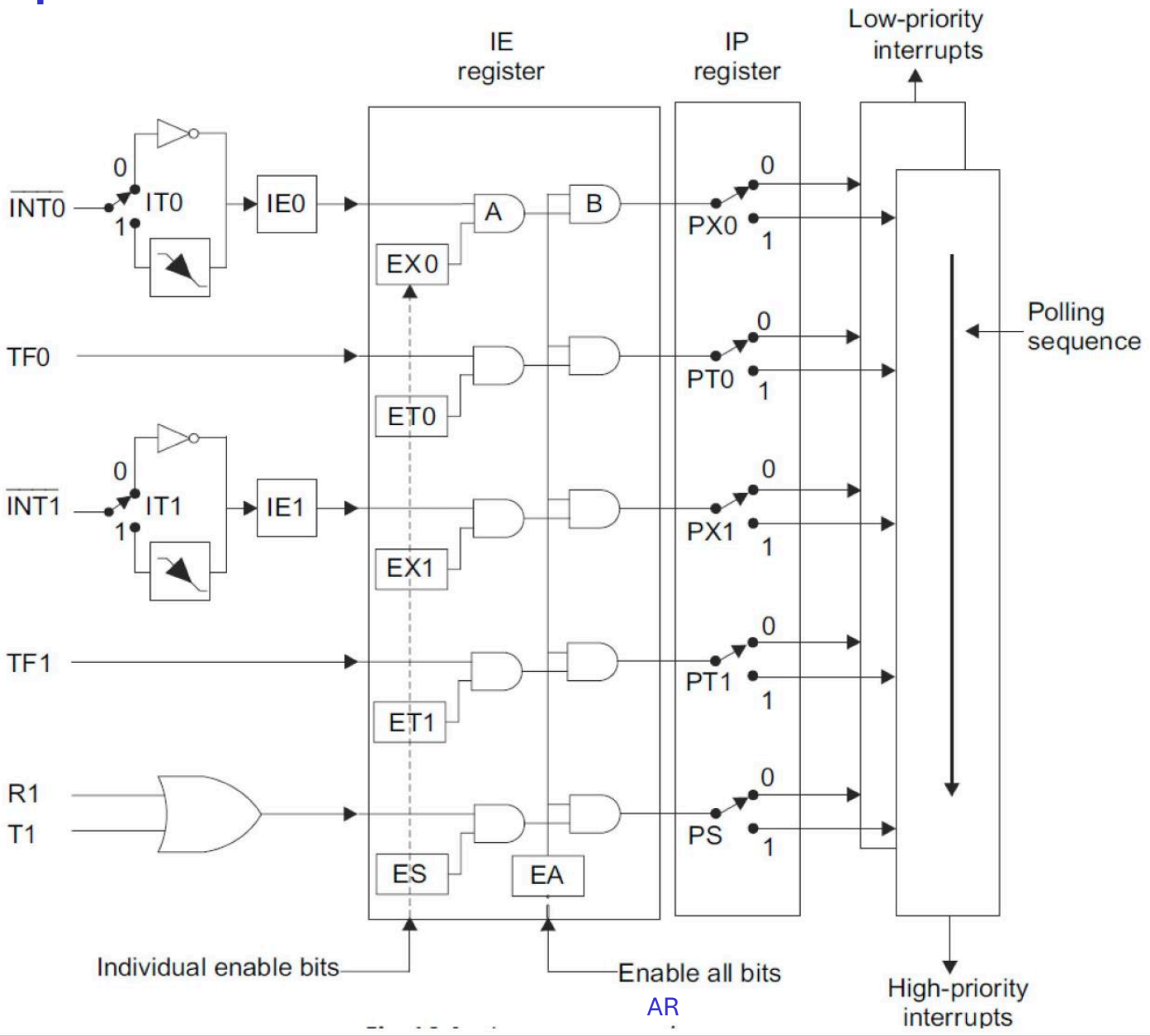
IPH.x	IPL.x	Priority level (0: lowest, 3: highest)
0	0	0
0	1	1
1	0	2
1	1	3

The IPL and IPH Registers

IPH0	IPL0
7. (Reserved)	7. (Reserved)
6. PPCH (PCA Int)	6. PPCH (PCA Int)
5. PT2H (Timer2)	5. PT2 (Timer2)
4. PSH (Serial port)	4. PS (Serial port)
3 PT1H (Timer 1)	3. PT1 (Timer 1)
2. PX1H (INT1)	2. PX1 (INT1)
1. PT0H (Timer 0)	1. PT0 (Timer 0)
0. PX0H (INT0)	0. PX0 (INT0)

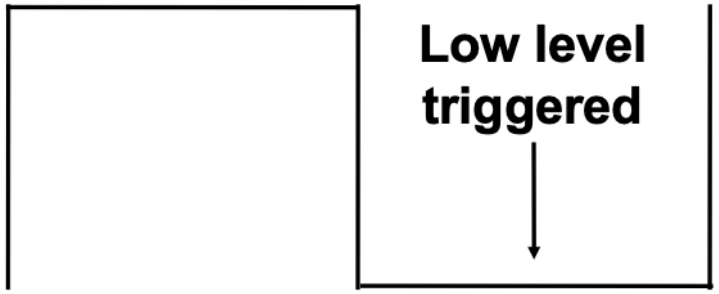
IPH.x	IPL.x	Priority level (0: lowest, 3: highest)
0	0	0
0	1	1
1	0	2
1	1	3

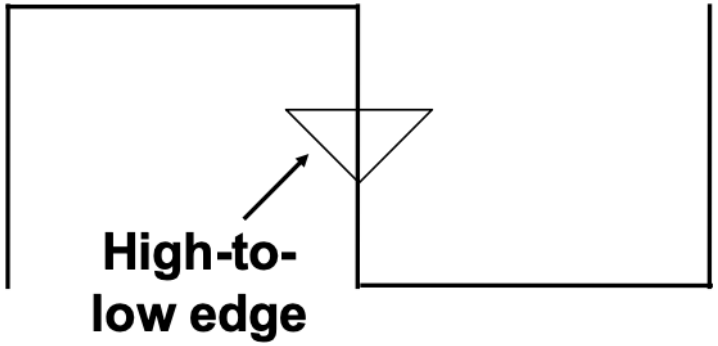
Interrupt Control Structure



External Interrupt

In addition to responding to a timer, external pin activations can be configured to interrupt program flow.

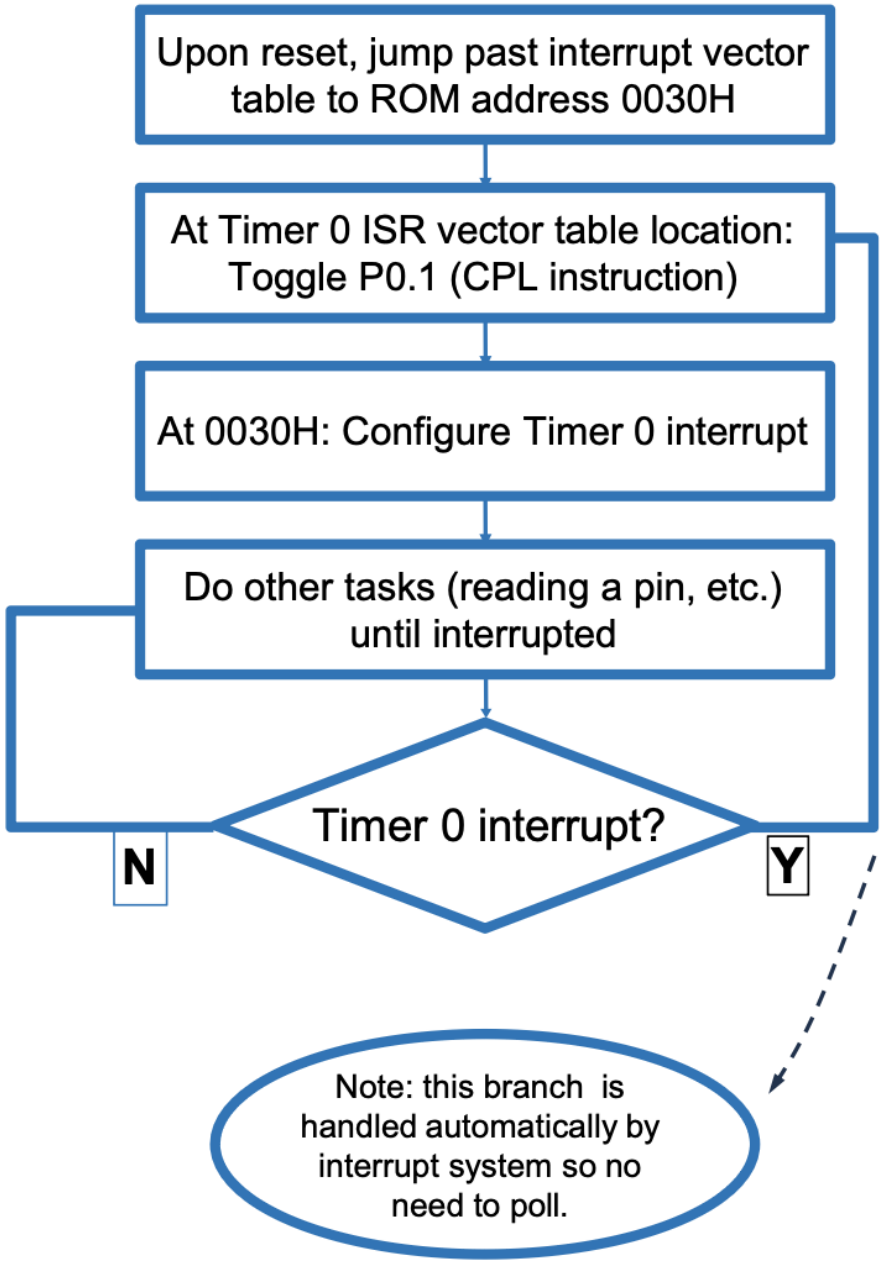




External Interrupt

Goal: replicate the "EXAMPLE SQUARE WAVE" code from a previous slide, replacing the polling approach with an interrupt-driven approach.

- Advantages: we can free up the CPU from having to monitor the TFx flag.
- For this example, we'll keep the ISR short so it will fit within the 8 Bytes of ROM allocated by the Interrupt Vector Table.



Example 2: Timer Interrupt

```
;We'll be using the interrupt address space, so we need to make
;sure that our main program bypasses this.

ORG 0000H
LJMP MAIN                       ;Jump past the interrupt vector table.

;—-ISR for Timer 0. Will generate a square wave at Port 0.1
ORG 000BH                       ;Assembler will place code at Timer 0 vector address
CPL P0.1                ;Toggle Port 0 pin 1
MOV TL0,#00                     ;Must reload timer values manually in this mode
MOV TLH,#80H
RETI                            ;Return from int, pops PC and resets interrupt

;—- Main body of program. Start by configuring Timer 0
ORG 0030H                       ;Directs assembler past interrupt vector table
MAIN:
MOV TMOD, #00000001B    ;Timer 0, mode 1
MOV TL0,#00                     ;Initially load timer values
MOV TLH,#80H
MOV IEN0, #10000010B    ;Enable timer 0 interrupt
SETB TR0                ;Start the timer
IDLE: SJMP IDLE         ;Loop until interrupt(no polling of int flag!)
```

TimerInterrupt.asm

Example 2: Timer Interrupt

Purpose of the Program

The program:

- configures Timer 0 in Mode 1 (16-bit timer)
- enables Timer 0 interrupt
- starts the timer
- whenever timer overflows:
  - interrupt occurs
  - ISR toggles P0.1
- repeated toggling creates a square wave

Program Structure

The program has 3 parts:

1. Reset vector
2. Interrupt Service Routine (ISR)
3. Main program

Example 2: Timer Interrupt

```
ORG 0000H
LJMP MAIN                    ;Jump past the interrupt vector table.
```

Place the following instruction at memory address 0000H
Address 0000H is the reset address of 8051.

LJMP MAIN
Long jump to label MAIN.

Why?
Because lower memory addresses are reserved for interrupt vectors.
So the main program is placed later at address 0030H.

Example 2: Timer Interrupt

```
ORG 000BH                    ;Assembler will place code at Timer 0 vector address
```

Timer 0 Interrupt Vector

000BH is the interrupt vector address for Timer 0.
Whenever Timer 0 overflows:
PC → 000BH
The CPU automatically jumps there.

```
CPL P0.1                     ;Toggle Port 0 pin 1
```

CPL = Complement (toggle)
So:
0 → 1
1 → 0
Pin P0.1 changes state every interrupt.
Repeated toggling creates a square wave.

Example 2: Timer Interrupt

```
MOV TL0,#00                ;Must reload timer values manually in this mode
MOV TLH,#80H
```

Timer Mode 1 is a 16-bit timer.
After overflow:
- timer does NOT reload automatically
- programmer must reload values manually

The timer starts counting again from:
8000H
up to:
FFFFH
Then another overflow occurs.

RETI

RETI = Return from interrupt
1.pops return address from stack
2.restores normal program execution
3.re-enables interrupts
After RETI, CPU returns to wherever it was interrupted.

Example 2: Timer Interrupt

```
;—- Main body of program. Start by configuring Timer 0
ORG 0030H                          ;Directs assembler past interrupt vector table
```

Places the main program at address 0030H.
This avoids overwriting interrupt vector locations.

```
MAIN:
MOV TMOD, #00000001B      ;Timer 0, mode 1
```

MAIN Label

Bit	Function
Timer 0 Mode	Mode 1 = 16-bit timer.
Timer/Counter	Timer (0)

Example 2: Timer Interrupt

```
MOV TL0,#00                    ;Initially load timer values
MOV TH0,#80H
```

Loads starting value: 8000H

```
MOV IEN0, #10000010B       ;Enable timer 0 interrupt
```

Binary:
10000010B

Bit	Meaning
EA=1	Global interrupt enable
ET0=1	Enable Timer 0 interrupt

So Timer 0 interrupts are enabled.

Example 2: Timer Interrupt

```
SETB TR0                ;Start the timer
IDLE: SJMP IDLE         ;Loop until interrupt(no polling of int flag!)
```

TR0 = Timer Run control bit.
Setting it to 1 starts Timer 0 counting.

Important Concept — No Polling

Notice:

No checking of TF0 flag

The CPU does NOT continuously test the overflow flag.

Instead:

Timer overflow automatically triggers interrupt

This is interrupt-driven programming.

Example 2: Timer Interrupt

Timer Starts
↓
Timer Counts
↓
Overflow Occurs
↓
Interrupt Generated
↓
CPU Jumps to 000BH
↓
Toggle P0.1
↓
Reload Timer
↓
RETI
↓
Return to Main Program
↓
Repeat Forever

Square Wave Generation

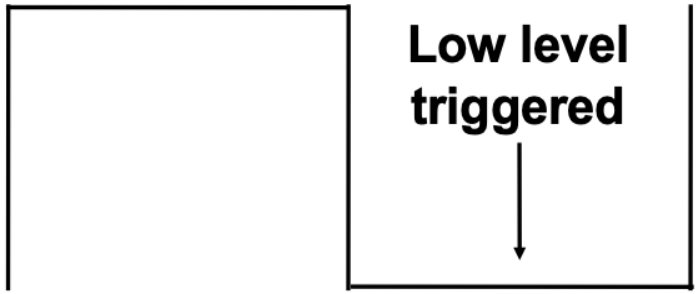
Every interrupt changes P0.1:

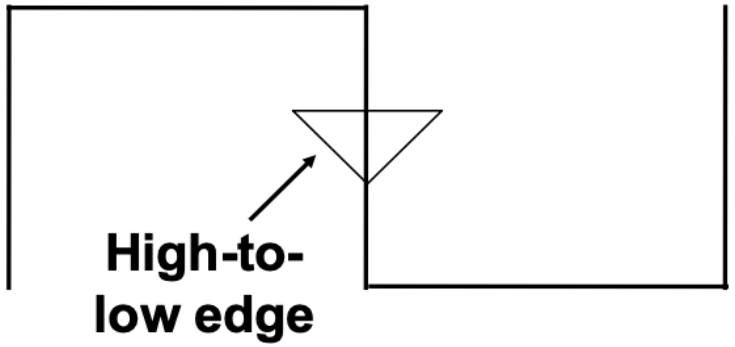
0 → 1 → 0 → 1 → 0

This produces a square wave signal.

TimerInterrupt.asm

External Interrupt



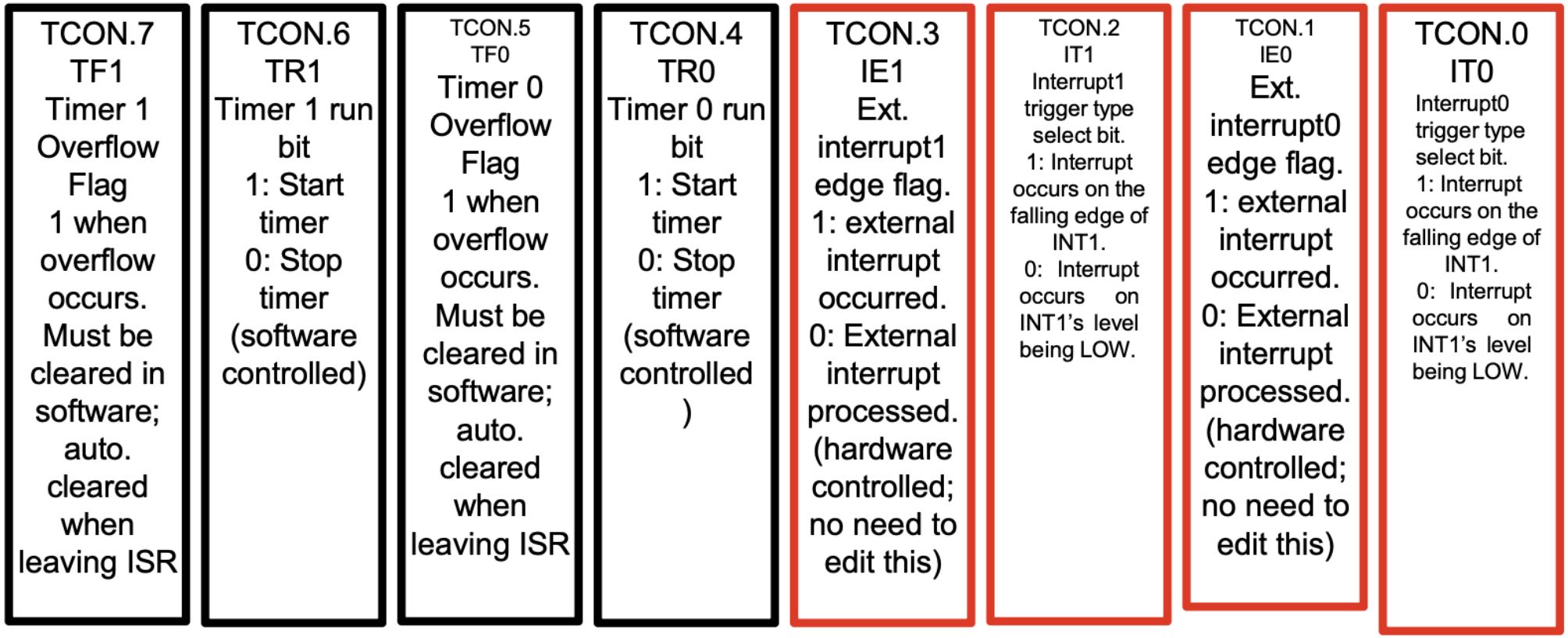


- On the 8051, interrupts can be set to be triggered in response to two different types of pin states/state changes:
  - Low-level triggered: when the external interrupt pin is LOW, the interrupt is triggered.
    - The interrupt will be re-triggered if the pin remains low after the ISR!
  - Edge-triggered interrupt: interrupt is triggered on the falling edge of the signal at the external interrupt pin.
    - Falling edge: when the pin goes from HIGH to LOW.
    - Low-level or falling edge mode: configured in TCON register.
    - Need to trigger on rising edge? Use an inverter chip (e.g., 7404)

External Interrupt Configuration

Low 4 bits of TCON register: configure/examine external interrupts.

- TCON.3 (IE1) and TCON.1 (IE0): 'external interrupt occurred' flags.
- TCON.2 (IT1) and TCON.0 (IT0): set interrupt trigger type.



Example 3: External Interrupt

Goal: Toggle an LED connected to Port 1, pin 3.
This LED will toggle when an external interrupt (INT1) occurs.

```
ORG 0000H
LJMPMAIN              ;Jump past the interrupt vector table.

;— ISR: Interrupt 1, toggles LED when new interrupt arrives
ORG 0013H             ;Location in vector table of INT1
CPL P1.3              ;Toggle Port 1 pin 3
RETI                  ;Reset PC and clear interrupt flags
;Set up interrupts at ROM location past vector table
ORG 0030H

MAIN:
SETB TCON.2           ;Interrupt is falling edge triggered
MOV IEN0 #10000100B ;Enable interrupts, INT1

IDLE:
SMJP DILE             ; Other code could go here (CPU is idle)
END
```

ExternalInterrupt.asm

Example 3: External Interrupt

The program:

- enables External Interrupt 1
- configures it as edge-triggered
- waits in an idle loop
- when INT1 occurs:
  - LED on P1.3 toggles

Whenever an interrupt signal arrives at INT1 pin:

- the CPU automatically jumps to the ISR
- the ISR toggles an LED connected to P1.3
- then returns to the main program

Example 3: External Interrupt

Program Structure

1. Reset vector
2. Interrupt vector for INT1
3. Main program

1. Reset Vector

```
ORG 0000H
LJMP MAIN              ;Jump past the interrupt vector table.
```

ORG 0000H
Places instruction at memory address: 0000H This is the reset starting address of 8051.

Long jump to the main program.
Because addresses near 0000H are reserved for interrupt vectors.
So, the real program starts later at 0030H.

Example 3: External Interrupt

2. INT1 Interrupt Vector

```
;— ISR: Interrupt 1, toggles LED when new interrupt arrives
ORG 0013H              ;Location in vector table of INT1
```

External Interrupt 1 (INT1)
When INT1 occurs:
PC → 0013H
The CPU automatically jumps there.

```
CPL P1.3               ;Toggle Port 1 pin 3
```

The LED connected to P1.3 changes state every interrupt.

```
RETI                   ;Reset PC and clear interrupt flags
```

`RETI` = Return from Interrupt
1.restores the saved Program Counter (PC)
2.returns execution to main program
3.clears interrupt servicing status
After RETI:
`CPU continues where it stopped`

Example 3: External Interrupt

2. Main Program

```
ORG 0030H
```

Places main program after interrupt vector table.

```
MAIN:
SETB TCON.2          ;Interrupt is falling edge triggered
```

TCON.2 is the: IT1 bit

IT1 controls how INT1 is triggered.
Setting it to 1 means: Falling edge triggered

So interrupt occurs when signal changes: 1 → 0
Example:
- button press
- pulse signal
- sensor output

Example 3: External Interrupt

2. Main Program

```
MOV IEN0 #10000100B ;Enable interrupts, INT1
```

So INT1 interrupts are enabled.

```
IDLE:
SMJP IDLE              ; Other code could go here (CPU is idle)
END
```

Infinite loop.
The CPU waits here doing nothing.
Meanwhile:
•hardware monitors INT1 pin
•if interrupt occurs → CPU automatically jumps to ISR

Example 3: External Interrupt

Program running
↓
INT1 signal occurs
↓
CPU finishes current instruction
↓
PC saved on stack
↓
Jump to 0013H
↓
Toggle P1.3
↓
RETI
↓
Return to IDLE loop

Real Hardware Example

Suppose:

- push button connected to INT1 pin
- LED connected to P1.3

Every button press:
LED toggles ON/OFF

Example 4: External Interrupt

Write an 8051 assembly program that uses Timer 1 interrupt to continuously toggle Port 3 pin 7 whenever Timer 1 overflows.