Interrupts

Interrupts in 8051

- Interrupt vs Polling
- Interrupt service routine / "handler" and interrupt vectors
- Interrupt priorities
- Applications: keypad, timer, serial port, counter

Disadvantages with Polling

- Polling: e.g., **while** (TF0==0) { }
    - use loop, keep testing a flag until it is set
    - Problem: Wasteful=> not useful work
- Could try polling less often
    - e.g., **while** (TF0==0) { *do some work* }
    - Problem: potentially slow response / long latency

Solution: Interrupts

- Let hardware test flag instead of software
- When the flag is set,
  automatically call a subroutine (handler)
  - This means "interrupting" (suspend) the normal software execution in handler
- Handler returns to normal software
  - Software might not "know" it happened!

Polling vs. Interrupt

polling loop

"handling"

```
setup (e.g., timer);
while (TF0 == 0) {
}       // wasted cycles!
TF0 = 0;
other code to "handle" timer
```

vs

```
setup (enable interrupt)
regular program code
```

```
ISR(for timer, UART, etc) {
        TF0 = 0;
        other code to handle..
}
```

ISR is called automatically when the interrupt condition is detected by hardware

Terminology

- Interrupt vector:
  - address of an interrupt handler
- Interrupt vector table:
  - array of interrupt vectors (i.e., addresses)
- Interrupt service routine (ISR)
  - also known as *interrupt handler*
  - called by MCU to handle an interrupt

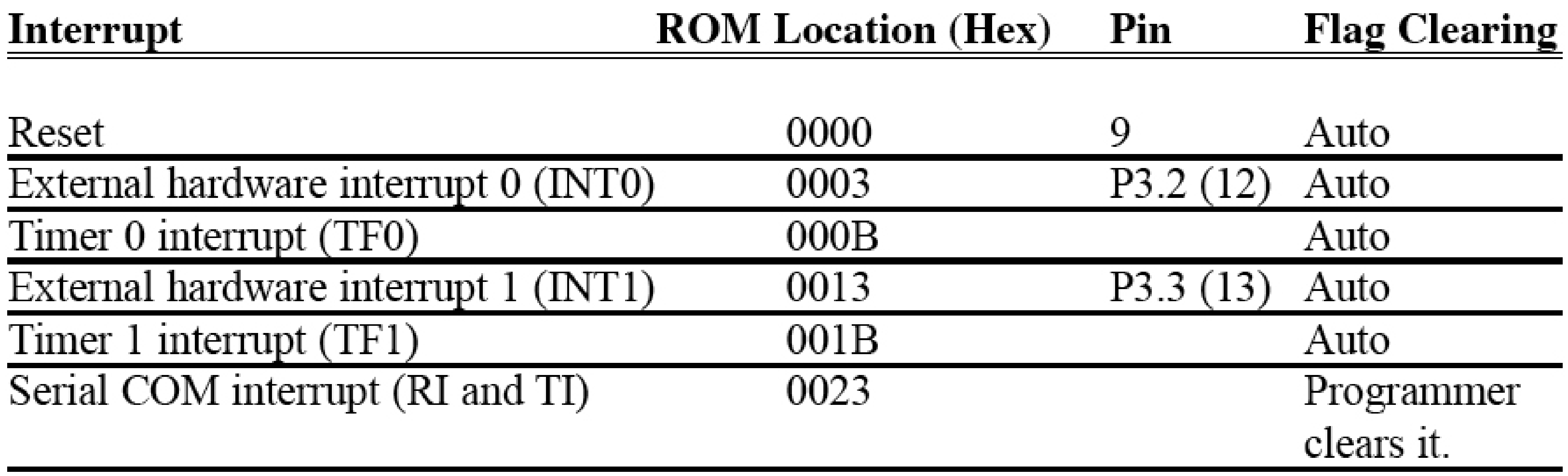
Steps in an Interrupt

- Finish current instruction
- Push next PC on stack,
  save other interrupt status in internal reg
- Jump to the interrupt vector (addr of ISR)
- Run until RETI (return from interrupt)
  => don't use RET -(for regular subroutines)
- Restore interrupt status, pop stack into PC

Interrupt types in 8051

- **Reset** - a special kind of interrupt
  - Jump to 0000H, "reset handler"
    (or: handler is at 0H), but no RETI
- Timer 0 and 1 (jump to 000BH, 001BH)
- INT0, INT1 pins (jump to 0003H, 0013H)
- Serial (both Rx and Tx): jump to 0023H

8051 Interrupt vectors



Interrupt	ROM Location (Hex)	Pin	Flag Clearing
Reset	0000	9	Auto
External hardware interrupt 0 (INT0)	0003	P3.2 (12)	Auto
Timer 0 interrupt (TF0)	000B		Auto
External hardware interrupt 1 (INT1)	0013	P3.3 (13)	Auto
Timer 1 interrupt (TF1)	001B		Auto
Serial COM interrupt (RI and TI)	0023		Programmer clears it.

- 0000H: a jump (2 or 3 bytes) to _main
    - (if you want to use interrupts)
- 0003H, 000BH, 0013H, ... (8 byte spaces)
    - Handler code (if fit in 8 bytes), or jump to handler routine if too long

Assembly code layout

```
       ORG     0       ;; reset handler
       LJMP    MAIN
       ORG     0003H ;; INT0 handler
       code for INT0 handler
       ORG     000BH
       code for INT0
       ORG     30H
MAIN:  .. main program code
```

Interrupt-enable register IE (SFR)

- IE.0 ... IE.5 (EX0, ET0, EX1, ET1, ES, ET2)
  - enable bit for each interrupt source
- IE.6: reserved (by default, write 0)
- IE.7 (EA): enable all
  - Actually, EA=1: set individual IE.0, ... IE.5; but if EA=0 then disable all interrupts

Content of IE register

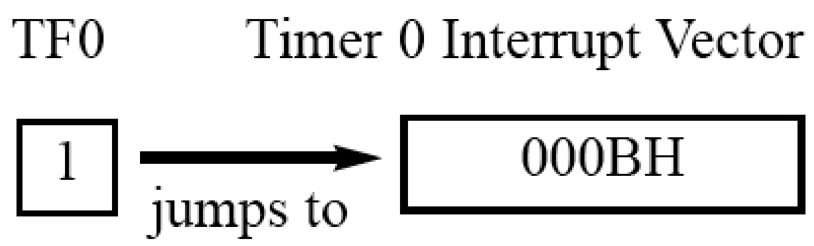
D7							D0
EA	--	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables Timer 1 overflow interrupt.
EX1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

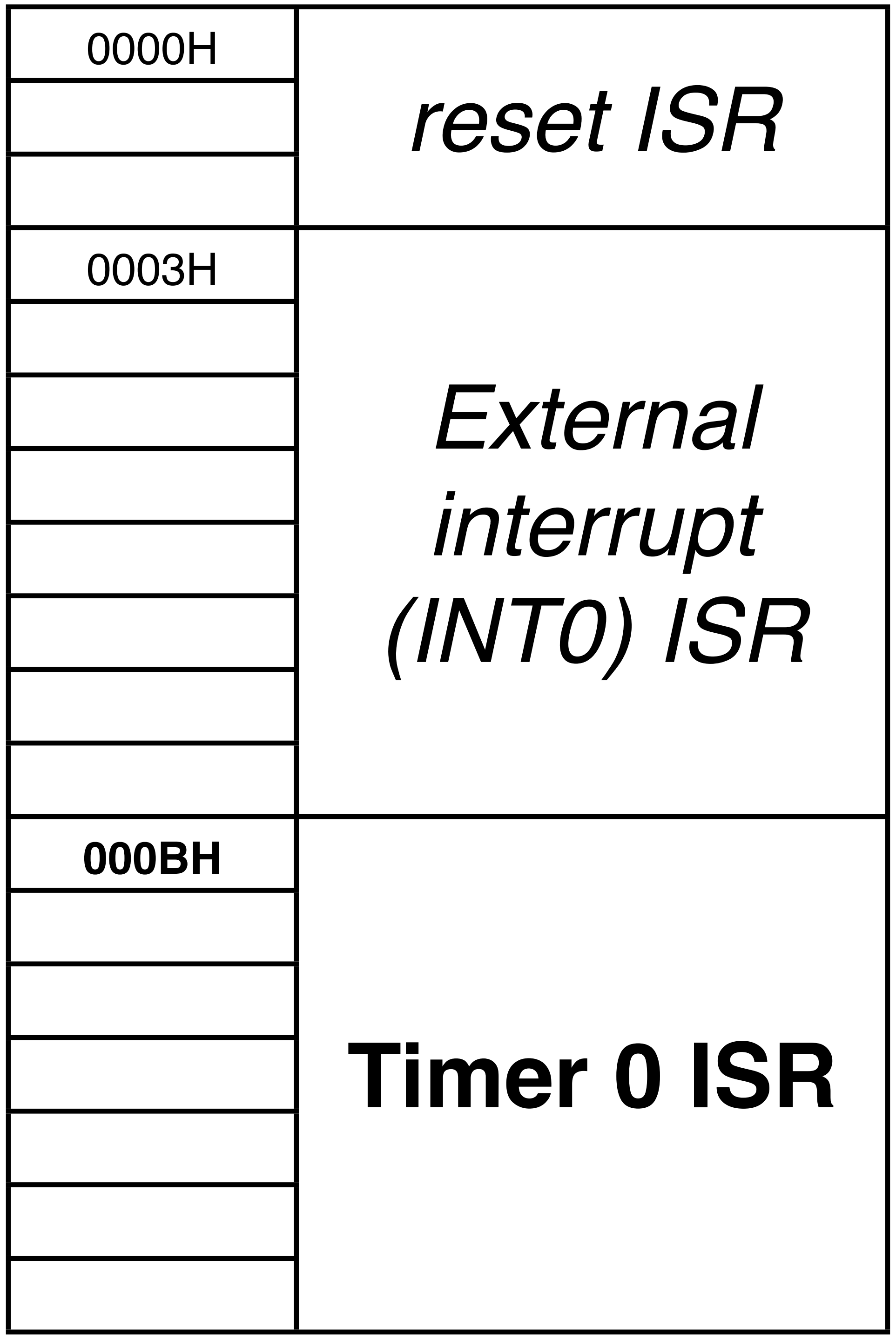
example timer roll-over

- Interrupt:



- To enable:
  MOV IE, #82H

EA	--	ET2	ES	ET1	EX1	**ET0**	EX0
1	0	0	0	0	0	1	0



Example 1

```
        ORG     0000H           ;; reset "ISR"
        LJMP    MAIN
        ORG     000BH           ;; timer 0 ISR!
        CPL     P2.1            ;; toggle pin
        RETI                    ;; total ISR fits in 8 bytes!!
        ORG     0030H           ;; space after interrupt vectors
MAIN:   MOV     TMOD, #02H      ;; timer 0 mode 2 auto reload
        MOV     P0, #0FFH       ;; input pin
        MOV     TH0, #-92       ;; auto reload value
        MOV     IE, #82H        ;; enable timer 0 interrupt!
        SETB    TR0             ;; start timer
BACK:   MOV     A, P0           ;; main loop to read P0 write P1
        MOV     P1, A           ;; interrupt can happen anywhere
        SJMP    BACK            ;; while doing "useful work"
```

Ex. 2: 50Hz square wave

```
        ORG     0000H           ;; reset "ISR"
        LJMP    MAIN

        ORG     000BH           ;; timer 0 ISR!
        CPL     P1.2            ;; toggle pin
        MOV     TL0, #00        ;; load timer value
        MOV     TH0, #0DCH      ;;
        RETI                    ;; total ISR fits in 8 bytes!!

        ORG     0030H           ;; space after interrupt vectors
MAIN:   MOV     TMOD, #01H      ;; timer 0 mode 1
        MOV     TL0, #00        ;; load timer value
        MOV     TH0, #0DCH      ;;
        MOV     IE, #82H        ;; enable timer 0 interrupt
        SETB    TR0             ;; start timer
BACK:   SJMP    BACK            ;; while doing "useful work"
        END
```

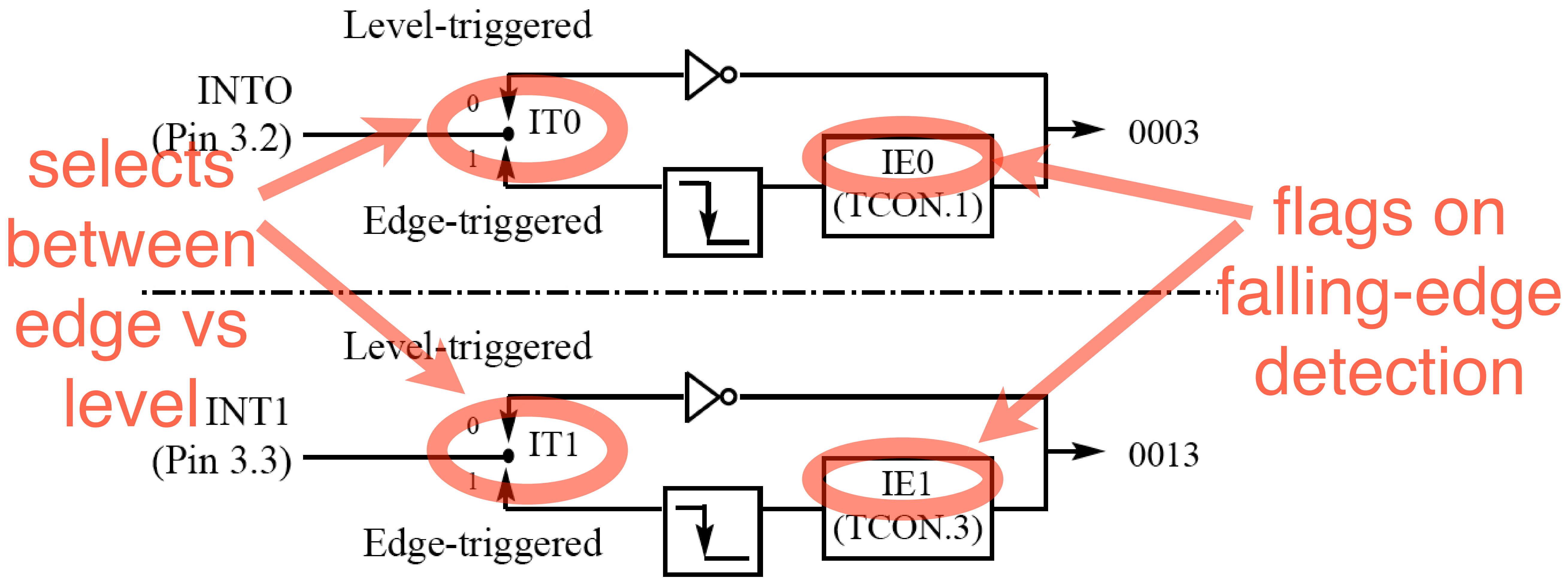
Issues with Interrupts

- External input:
  - Edge-triggered vs level sensitive
- Interrupt priority
  - Hardware vs. software defined priority
  - Nested interrupts
  - Software interrupts

External interrupts

- Input source: INT0 (P3.2) or INT1 (P3.3)
- Two ways to trigger an interrupt
    - Level-sensitive (as long as line is low)
    - Edge-triggered (on falling edge)
- Event type is programmable in TCON
    - settable via IT0, IT1 bits of TCON
    - on power up, level-sensitive

Schematic for hardware interrupt



INT0, INT1 pin configuration

- Actually, should be /INT0 and /INT1 signals
    - / denotes overbar => active low
- Should be normally high (as input)
- Outside signals could jointly pull signal low
    - Multiple devices could share an INT line!
    - ISR would query small group of devices

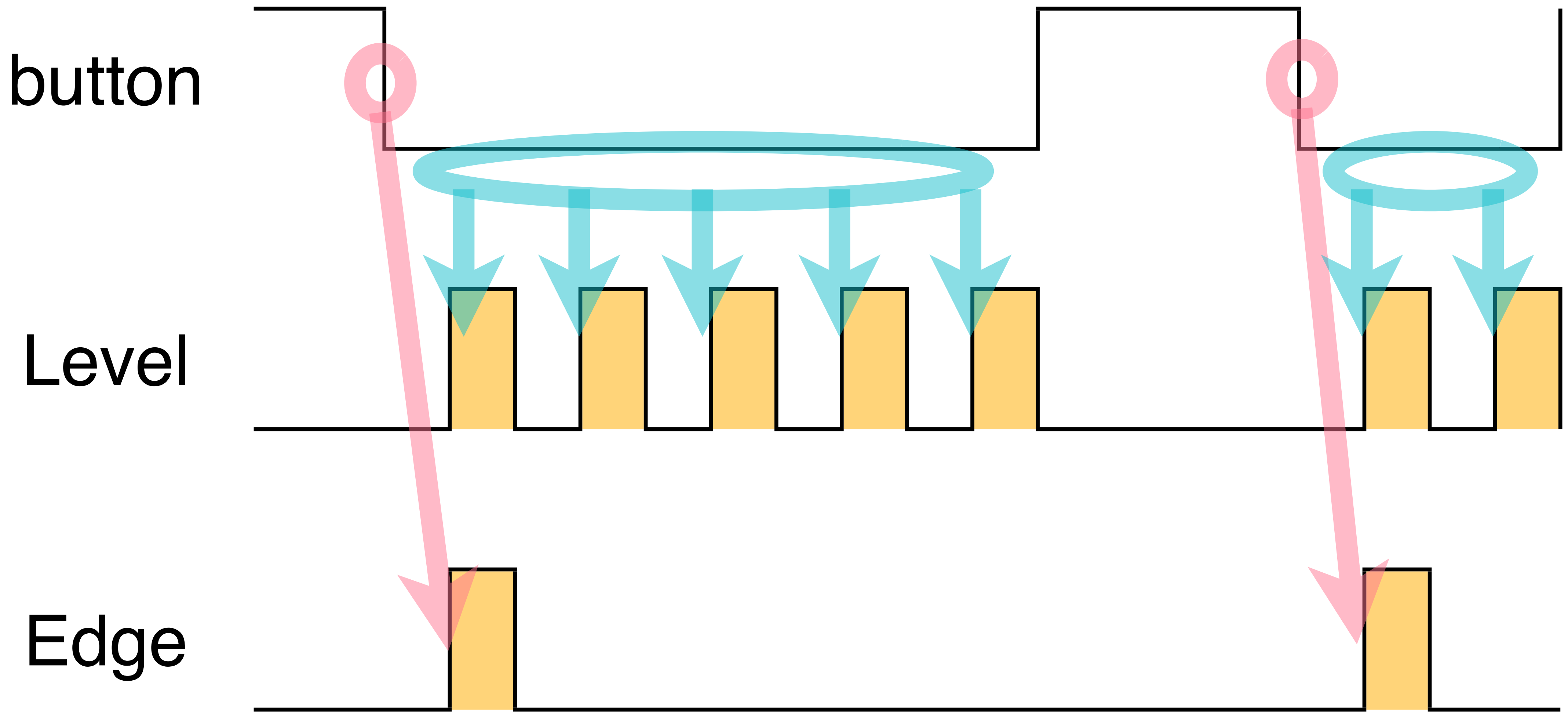
Level-sensitive external interrupts

- Required low-value minimum time of signal
  - 4 machine cycles minimum
  - input must remain stable until ISR runs
- Can get another interrupt as long as signal remains low
  - To prevent consecutive interrupt:
    => ensure signal goes high before RETI

Edge-triggered external interrupts

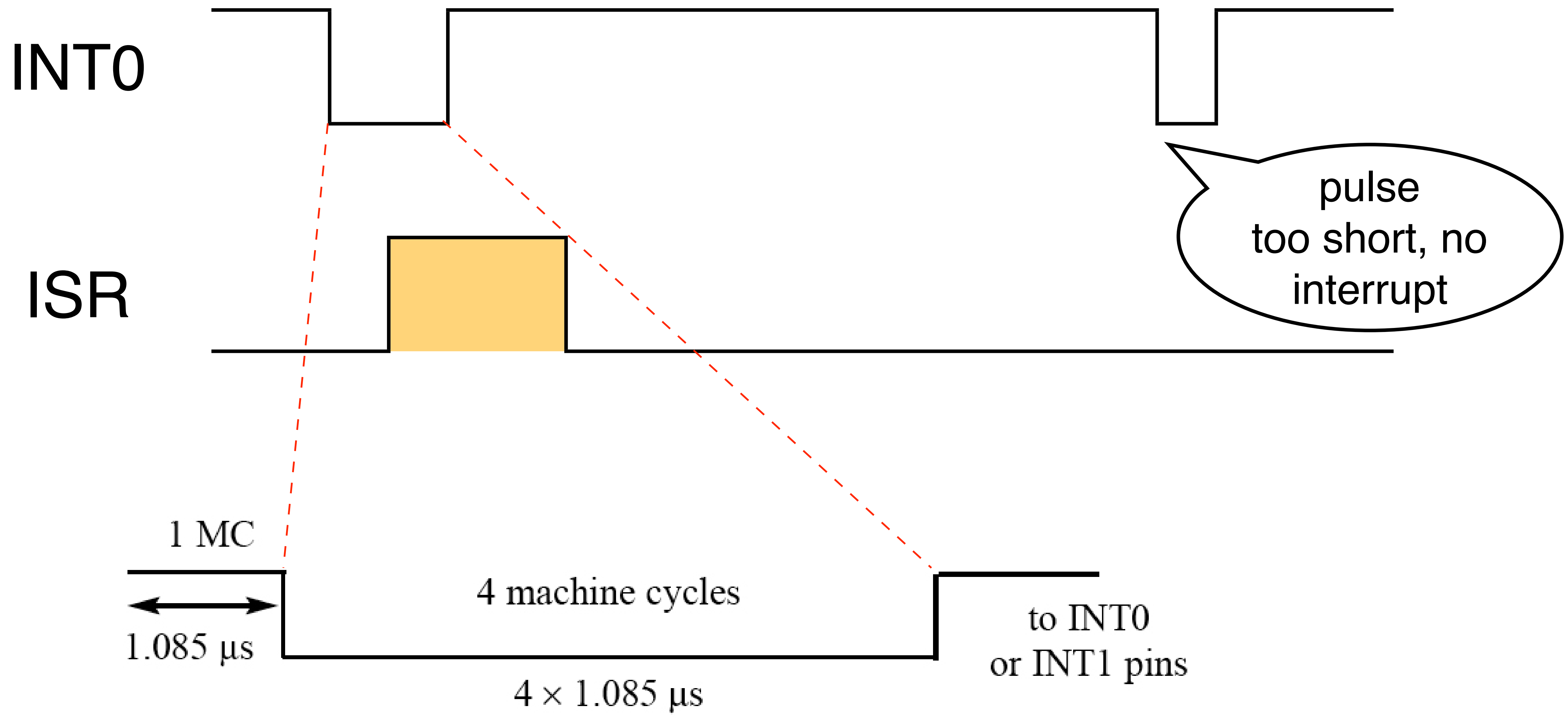
- IT0 = 1 or IT1 = 1 for edge triggered (=0 for level sensitive by default)
  - on falling edge (high-to-low transition)
  - IT0 is TCON.0, IT1 is TCON.2
- Same interrupt handler code may behave differently depending on level-sensitive or edge-triggered

Level Sensitive vs. Edge Triggered



🟨 interrupt handling

4 cycle min duration for INT0, INT1 input

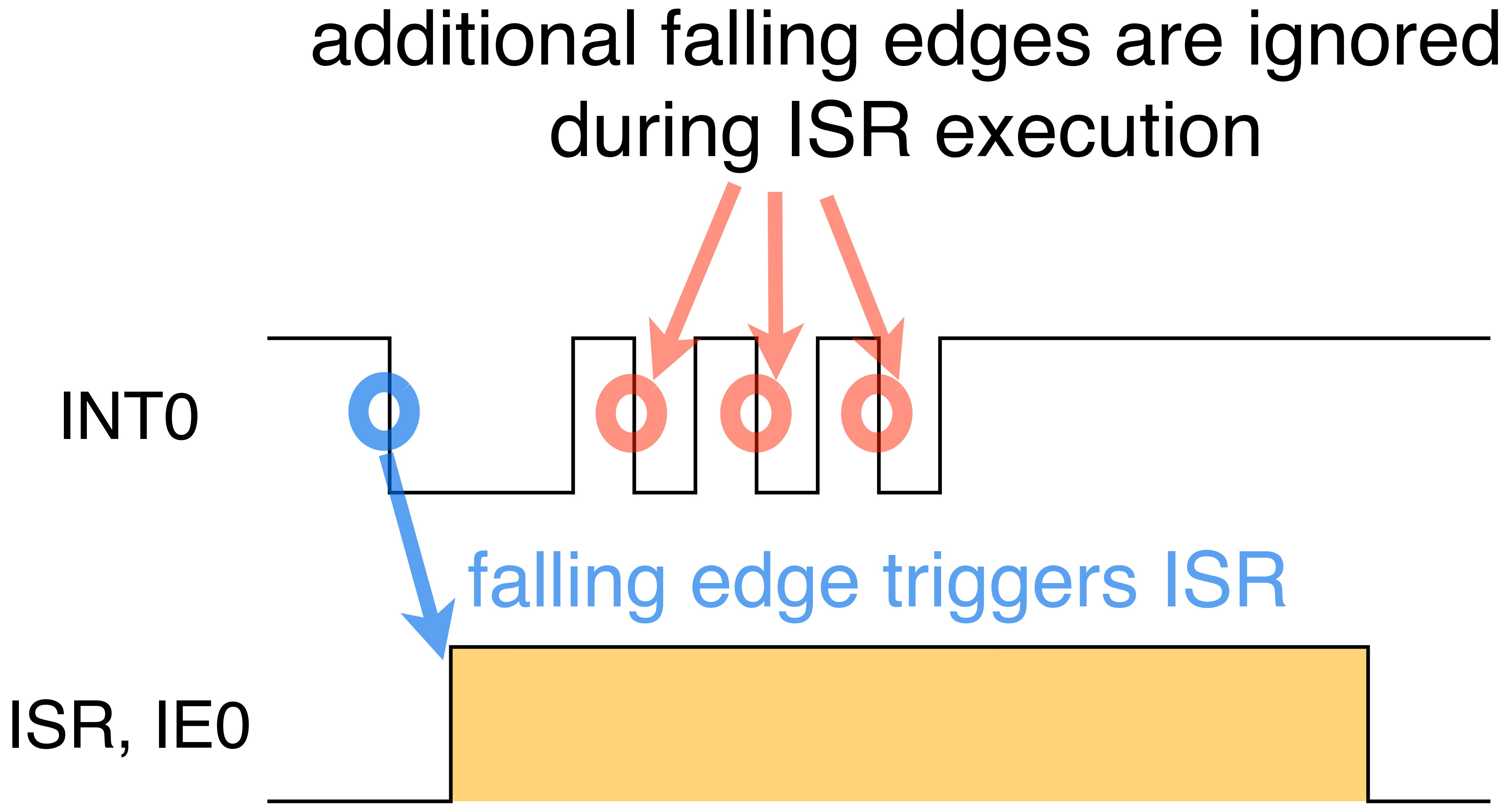


Note: On RESET, IT0 (TCON.0) and IT1 (TCON.2) are both low, making external interrupts level-triggered.

Edge-triggered external interrupt

- The falling edge is latched in TCON
  - IE0 (TCON.1) and IE1 (TCON.3) are the flags on falling edge
  - Flag not set on level-triggered signal (or, Flag is cleared automatically by ISR)
- Additional falling edges during ISR are ignored when handling edge-triggered INT

Edge-triggered: ignore edge during ISR


additional falling edges are ignored during ISR execution
INT0
falling edge triggers ISR
ISR, IE0

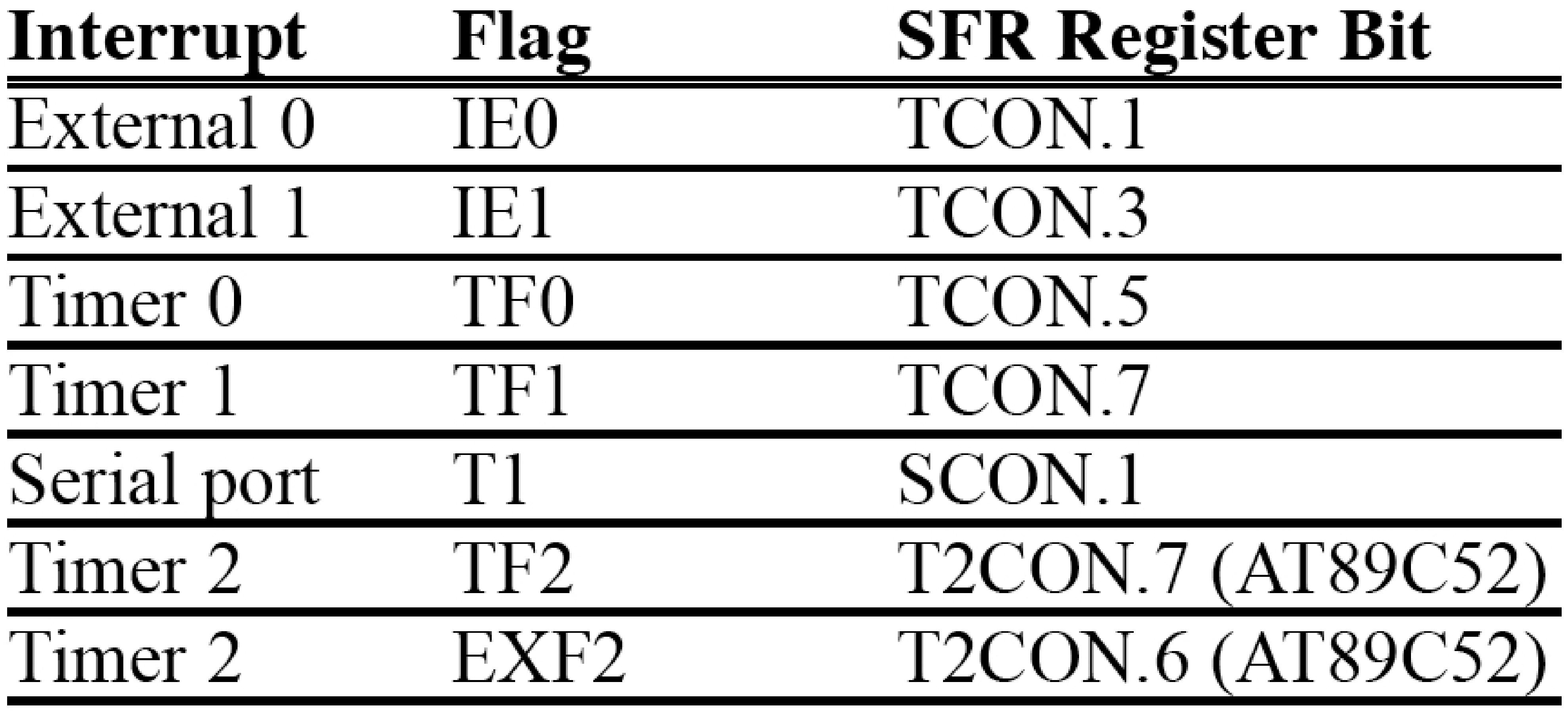
Serial port: review

- SBUF register
  - write SBUF => transmit;
    read SBUF => receive
- Flags
  - TI: => 1 when ready for next byte
  - RI => 1 when a byte has been received
- Flag could be polled or used as interrupt

8051: same vector for both Tx and Rx

- one interrupt vector for both Tx and Rx
- User must check whether TI or RI is on
  - TI on => ready to send next char
  - RI on => read char from SBUF
- User is responsible for clearing the flag!
  - Both could be set, but might handle just either Rx or Tx at a time

Summary of flags in SFRs



Interrupt	Flag	SFR Register Bit
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial port	T1	SCON.1
Timer 2	TF2	T2CON.7 (AT89C52)
Timer 2	EXF2	T2CON.6 (AT89C52)

Interrupt priorities

- Hardware-defined polling order
  - (hi) INT0, TF0, INT1, TF1, RI/TI, TF2 (lo)
- Software-programmable priority group
  - IP (interrupt priority) special-function reg
  - One bit per interrupt source
  - if set, then higher priority group

Interrupt Priority Register (IP)

- All devices w/IP bit=1 have higher *group priority* than all devices w/IP bit=0
- If two devices have same IP bit (i.e, same priority group), then their relative priorities are hardware defined
  - INT0 >TF0 > INT1 > TF1 > RI+TI > TF2

IP.7							IP.0
--	--	PT2	PS	PT1	PX1	PT0	PX0
res	res	timer2	serial	timer1	ext 1	timer0	ext 0

When interrupt priorities apply

- Multiple interrupts happen at same time
  - Can call only one ISR
  - Called: the interrupt with highest priority
- Conceptually: checked serially
- Another use: "nested interrupts"
  - Allows a higher priority one to interrupt a lower priority handler!

Nested interrupt

- A lower priority interrupt L happens
    - ISR_L gets called, but before RETI
- A higher priority interrupt H comes
    - ISR_H interrupts ISR_L
        - hardware pushes PC on stack,saves state
        - ISR_H interrupts ISR_L, RETI to resume ISR_L
- ISR_L continues handling interrupt L

Software interrupt

- SETB the flag register bit yourself!
  - e.g., SETB TF1
    triggers timer int. 1, like the real thing!
  - of course, need to follow the priority
- Why is this useful?
  - Want to "simulate" the effect of an interrupt, but can't use LCALL or ACALL

Issues with interrupts and C

- Shared data structures
  - volatile
  - atomic operations
- Code re-entrance
  - overlay, register banks
  - critical functions and statements
- See also SDCC manual Sections 3.8-3.11

Volatile variables

- **volatile** is a keyword in C
  - it means the variable can be changed by someone else other than (user) program
- **volatile** __bit clicked;

```
volatile __bit clicked;
void main(void) {
    clicked = 0;
    while (!clicked) ;
     ...do other things
```

presumably, an interrupt service routine changes click to 1 at some point, similar to an I/O port

What if you didn't declare it volatile?

- it may still work, but
- Compiler may optimize it away!
  - Why? because it's like
    if (true) { A } else { B }
    => compiler thinks it's same as just { A }
- clicked = 0;
  while (!clicked) => while(0) => eliminate!

Atomic operation

- An operation can be a *sequence* of instructions
- "Atomic": executed as if one indivisible unit (all or nothing)
  - Desirable property in user (interrupted) code, but not automatic!
  - Can be achieved by disabling interrupts (but expensive)

Example of non-atomic operations

- 16-bit and 32-bit operations on 8051
  - e.g., i++ => need a sequence of instructions to manipulate Carry bit
- Even 8-bit variables, due to limitation of addressing mode
  - counter += 8;
- External data access

```
MOV  A, counter
ADD  A, #8
MOV  counter, A
```

How to achieve atomicity

- Disable interrupts
  (EA = 0; instructions; EA = 1;)
  - 2-instruction overhead
- Explicit locking
  - does not require disabling interrupt
  - test-and-set (test-and-clear on 8051)

Reentrant Function

- A function that can be called before another instance finishes
- Requirement: each instance must have its own copy of local variables (incl. params)
  - Compile with --stack-auto flag
  - For 16/32-bit support, --int-long-reent
  - **void** foo(**void**) **__reentrant** { ... }

Overlays

- Reusing memory for functions that are never active simultaneously
  - i.e., one does not call the other
  - Overlay for locals and parameters
  - Saves memory
- Problem: not re-entrant. can disable by
  #pragma nooverlay
  in the source code

Register Banks for ISRs

- **void** ISR(**void**) __interrupt (4) __using (1)
  - uses register bank #1 (onchip addr. 08H-0FH)
- Advantages
  - Separate R0..R7 registers between ISR & user code => no need to save/restore
- Be careful with nested interrupts!
- ISR calling other functions
  => callee inherits the ISR bank, save & restore

Hot to make code interrupt-safe: __critical

- Functions
  int foo () **__critical** { ... }
- Statements
  __critical { i++; ... *more statements*}
- Effects
  - Disables interrupts; code; re-enable
  - Could also do EA = 0; ... EA = 1;

Issues: interrupt jitter vs. interrupt latency

- Jitter = Difference between shortest and longest interrupt latency
  - should be low jitter for media (e.g. audio) or else sounds terrible
- Latency
  - should be short for buffered data (e.g., serial port -- or else buffer overflow!)

Minimizing duration of disabling interrupts

- has the effect of minimizing
  - interrupt latency
  - interrupt jitter
- still need to keep ISR short
  - lower-priority ISR can't interrupt a higher
  - ISR should call few other functions

Locking implementation: JBC

- 8051: JBC *bit*, *target*
  - "jump to target if bit is set and clear bit."
  - atomic test-and-clear instruction to lock (0 => locked, 1 => unlocked)
  - To unlock, just SETB *bit*
- Other architectures commonly do atomic test-and-set (0=> unlocked, 1=>locked)

8051 Interrupt numbers used by C

Interrupt	**Name**	**Numbers used by 8051 C**
External Interrupt 0	(INT0)	0
Timer Interrupt 0	(TF0)	1
External Interrupt 1	(INT1)	2
Timer Interrupt 1	(TF1)	3
Serial Communication	(RI + TI)	4
Timer 2 (8052 only)	(TF2)	5

Serial port interrupts -- revisited

- Same ISR shared between RI and TI
  - Both RI and TI could have triggered!
  - ISR checks which of RI, TI needs servicing
- Issues
  - Use of software interrupt with TI
  - Shared data structure

ISR for Rx

```
volatile char RxData = 0;
volatile __bit receivedByte = 0;

void UART_ISR(void) __interrupt (4) {
   if (RI) {
      RxData = SBUF;  // read from Rx
      RI = 0;            // clear receive flag
      receivedByte = 1;
   }
}
// user's main program uses shared data variables
// similar to I/O ports
```

TxChar() routine and ISR for Tx

```
volatile char TxData;
volatile __bit TxBusy;
void TxChar(char c) {
   TxData = c; // "enqueue"
   if (!TxBusy) {
      TI = 1;  // SW interrupt
   } /* otherwise, we expect
   TI will come anyway */
}
/* assume TxData == 0
means empty, not valid char
*/
```

```
void UART_ISR(void) ... {
   if (TI) {
      if (TxData) { /* waiting*/
         SBUF = TxData;
         TxData = 0;
         TxBusy = 1;
      } else { /* completion */
         TxBusy = 0;
      }
      TI = 0; // clear in both
   }
}
```

Try it out: Echo example (11.059MHz)

```
void serial_handler(void)
__interrupt (4) {
  if (TI) {
    if (gotByte) {
      SBUF = b;
      gotByte = 0;
      txBusy = 1;
    } else {
      txBusy = 0;
    }
    TI = 0;
  }

  if (RI) {
    b = SBUF;
    // move received byte to A
    RI = 0;   // clear the RI flag
    gotByte = 1;
  }
}
/* complete source online:
"ser-buf-int.c" */
```

What else you have to do differently in C code

- void main(void) {
  }
- void _sdcc_gsinit_startup(void) {main();}
- void _mcs51_genRAMCLEAR(void){}
- void _mcs51_genXINIT(void){}
- void _mcs51_genXRAMCLEAR(void){}