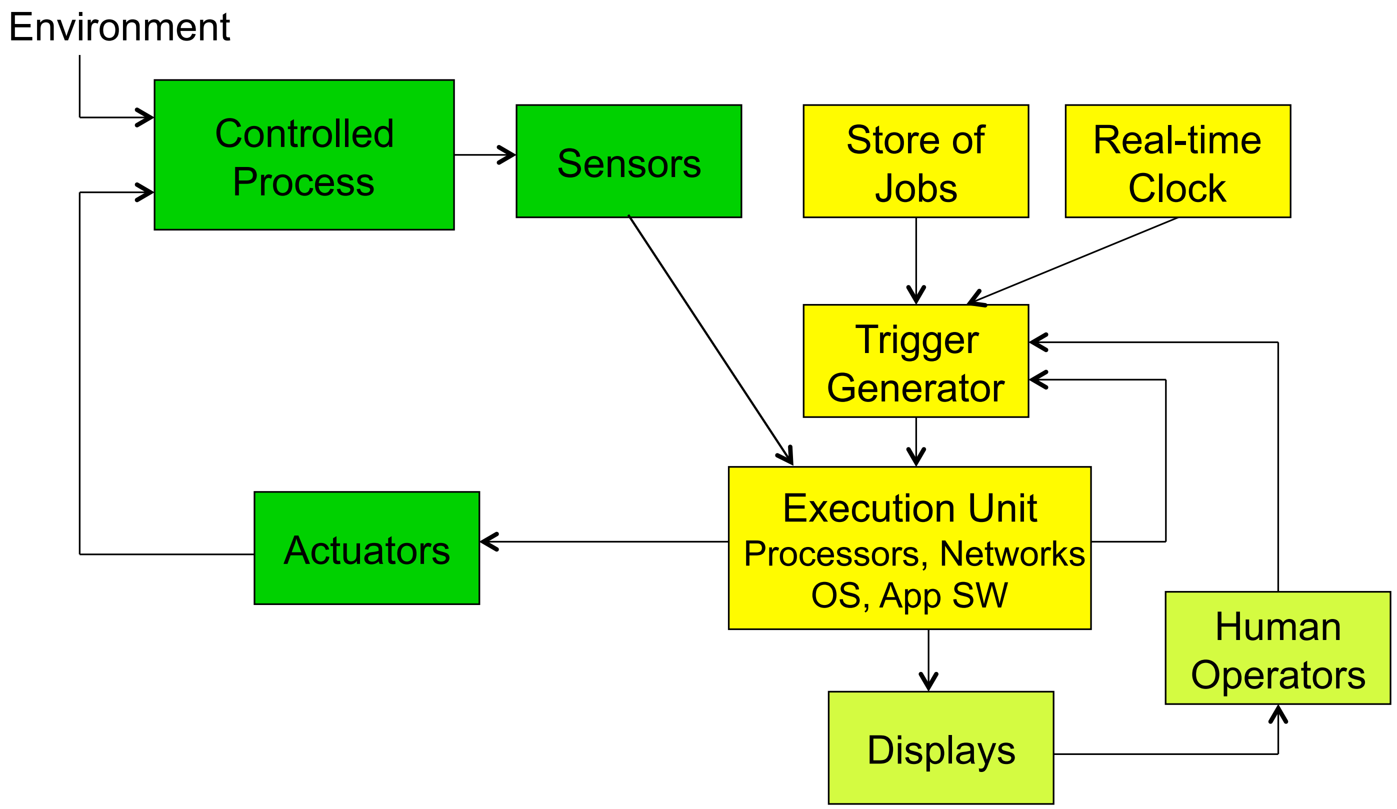
Real-time and Interrupt Programming

Baris Aksanli
Department of Electrical and Computer Engineering
San Diego State University

Outline

- Timing and event-triggered systems
  - Polling
- Interrupts
- Useful terminology about interrupts
- AVR Interrupts

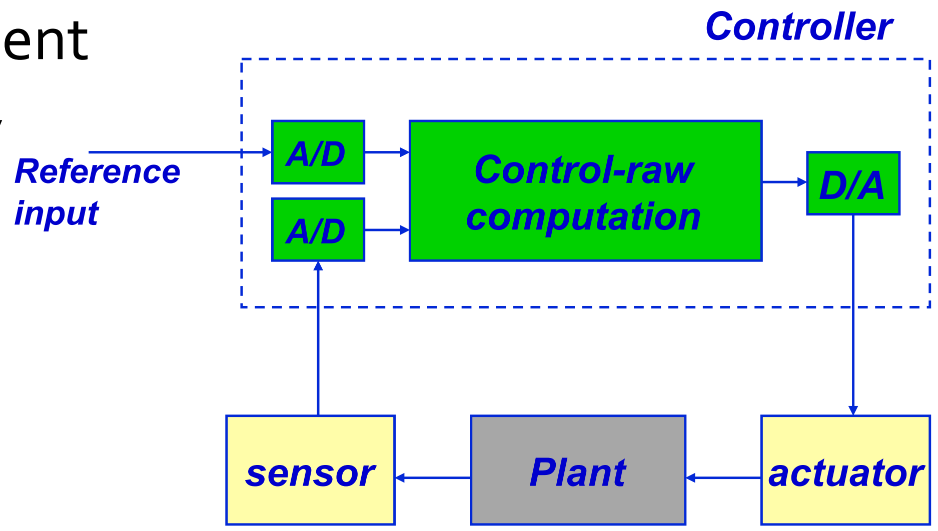
A Typical Real-time Embedded System



- Sensors/actuators: Getting data from/sending signals to the environment
- Execution units: Compute the signals
- Displays: Connection between embedded systems and humans

Real-time Embedded Systems

- **Embedded system**
  - Integration of SW and HW components
  - Integration of multiple subsystems
- **Real-time system**
  - Timely computation requirement
  - Formal modeling requirement
  - Deadlines/jitter/periodicity
  - Temporal dependency



What is “Real Time?”

- A real-time system must respond to an event and complete the related processing in a finite amount of time
  - Inputs are processed and outputs are generated at the rate which they naturally occur
  - For example, a simulation which progresses at the same rate as the entity being simulated, such as a flight simulator

Event Driven Systems

- The hardware and software together take action as a direct result of the occurrence of an event
- When an event occurs:
  - Other processing is suspended
  - The event is processed
  - Other processing is resumed
- Usually these are also real-time systems

Polling Events

- The simplest, but least efficient method of determining if an event has occurred
- Processor waits in a loop for the event
- Often used in simple I/O handlers:
  - Wait: Input buffer full?
    - No: jump to Wait
    - Yes: read input data buffer and continue
  - Processor does not do anything else
    - How efficient is this?

How did we do polling before?

- Busy wait!!
- Main idea is to wait in a busy while loop until a condition is observed
    - **while(condition is not observed){}**
- Examples:
    - while(serial transmit buffer is not empty){}
    - while(Timer0 overflow flag is not set){}
    - while(Timer1 OCRA compare flag is not set){}
- What are the tradeoffs associated with polling (busy wait)?
    - Advantages: simple to implement
    - Disadvantages: keeps processor busy, additional overhead, wasted processor time
    - What if you wanted to wait for two events??

Interrupts

What is an interrupt?

- One way to think of interrupts is that they are hardware-generated functions calls
- Internal hardware
  - When timer rolls over, then call a routine that blinks an LED
  - When built-in A/D converter is done converting, call a routine that manipulates the result
- External hardware
  - When the voltage level on a I/O pin changes, call a routine that turns a motor
  - When the USART receives a bit, call a routine that stored the bit, etc.
- The routines that are called when an interrupt occurs are called Interrupt Service Routines (ISRs)

Source: University of Iowa: ECE 55:036

Example

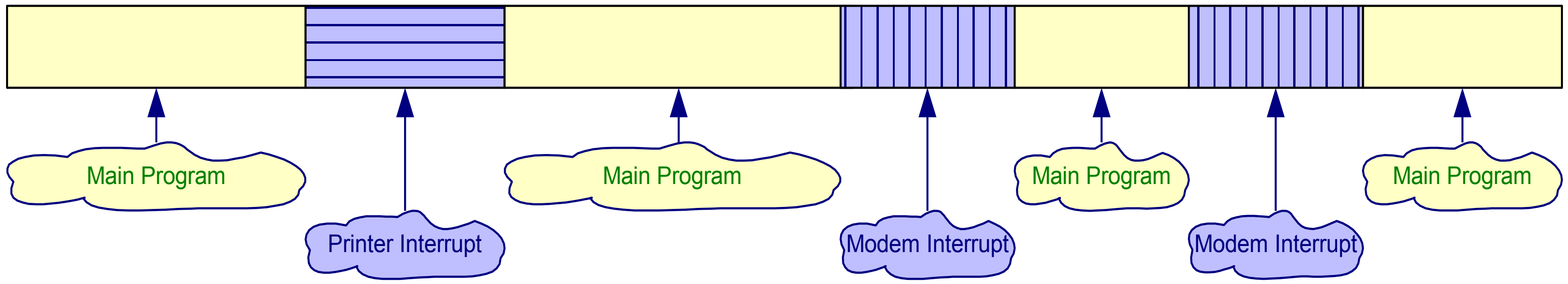
- Imagine a pushbutton that forces the CPU to call a specific program subroutine
  - When the switch is closed, it effectively CALLs a special subroutine (Interrupt Service Routine, or ISR)
  - This routine can be called independently of the instruction executing when it occurs
  - This is literally what is done for many microcontroller interrupt applications

Interrupts

- Interrupt types
  - Hardware interrupts: External event
  - Software interrupts: Internal event (Software generated)
  - Maskable and non-maskable interrupts
  - Interrupt priority
- Interrupt vectors and interrupt handlers
- Interrupt controllers

The Purpose of Interrupts

- Useful when interfacing I/O devices with low data-transfer rates
  - Like a keyboard or a mouse
  - Because polling the device wastes valuable processing time
- The peripheral interrupts the normal application execution, requesting to send or receive data
- The processor jumps to a special program called *Interrupt Service Routine* to service the peripheral
- After the processor services the peripheral, the execution of the interrupted program continues



Interrupts – Software vs. Hardware

- Software Interrupts: subroutine calls
  - Used to call routines without knowing address
  - Example: Operating System Calls
    - Allows access to standard system functions
- Hardware Interrupts: a “hardware” call
  - Used to allow response to external events
  - Minimizes wasted CPU time
  - Eliminates need to poll for an event

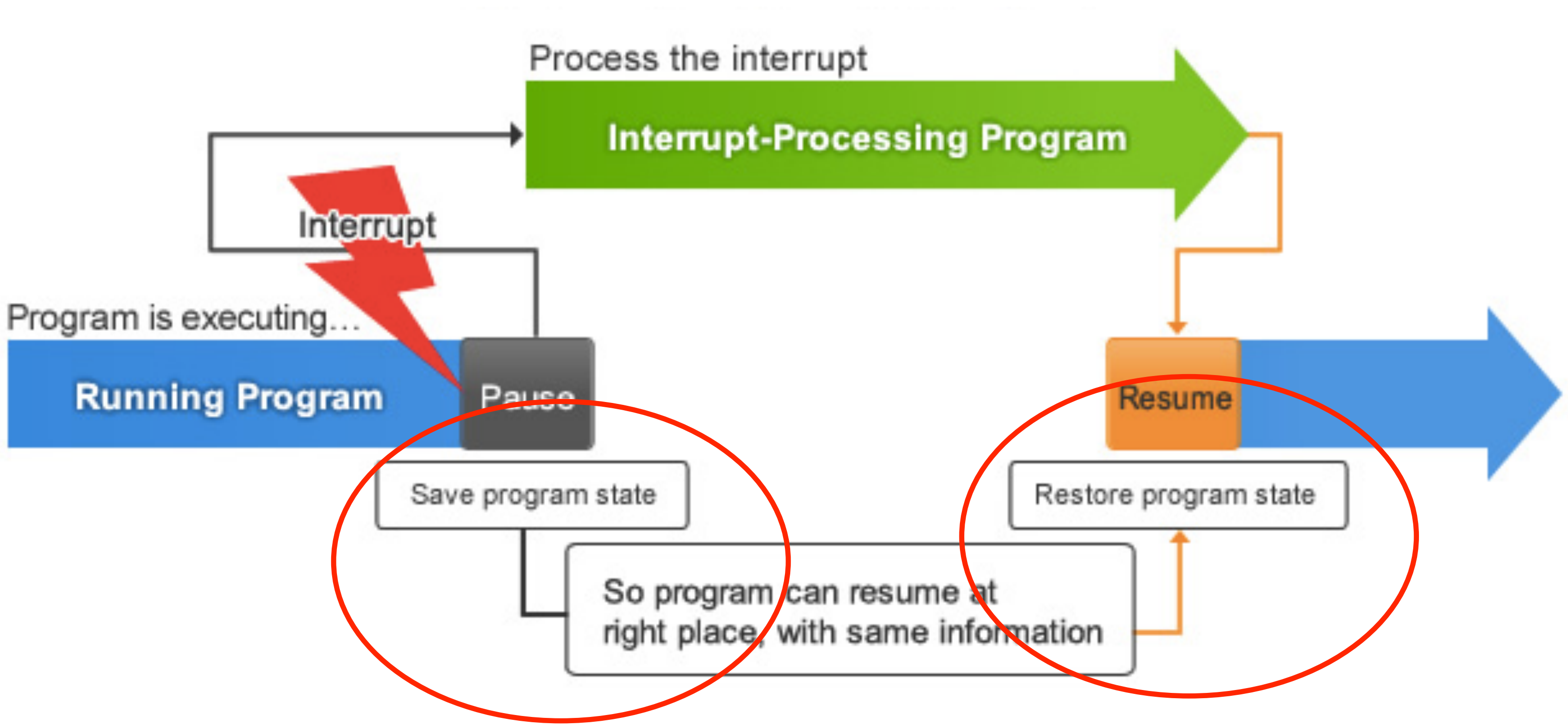
Software Interrupts

- Specialized subroutine calls
  - Always occur at the same point in a program
  - Synchronized to the program execution
  - Location Independent
  - Uses:
    - Calling a subroutine in ROM
    - Predefined O.S. Functions
    - Handling Exceptions (Overflow, divide by zero)

Hardware Interrupts

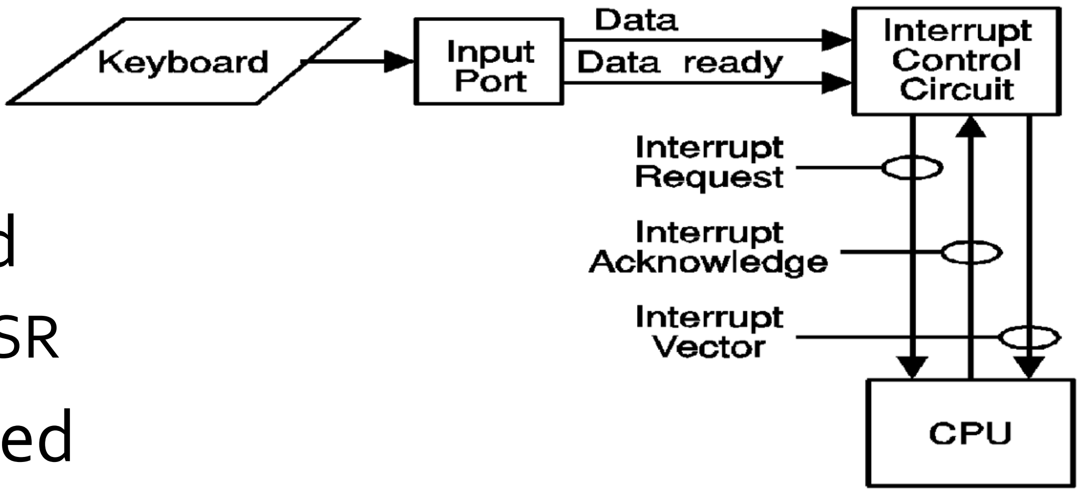
- Hardware Interrupts: a hardware call
  - Asynchronous to program
    - Can occur at any point in program execution
    - Timing is unrelated to main program execution
  - Forces the CPU to call interrupt subroutine
    - **I**nterrupt **S**ervice **R**outine (ISR)
  - Allows timely response to an event
    - Important events are processed on demand

Interrupt Processing Flow

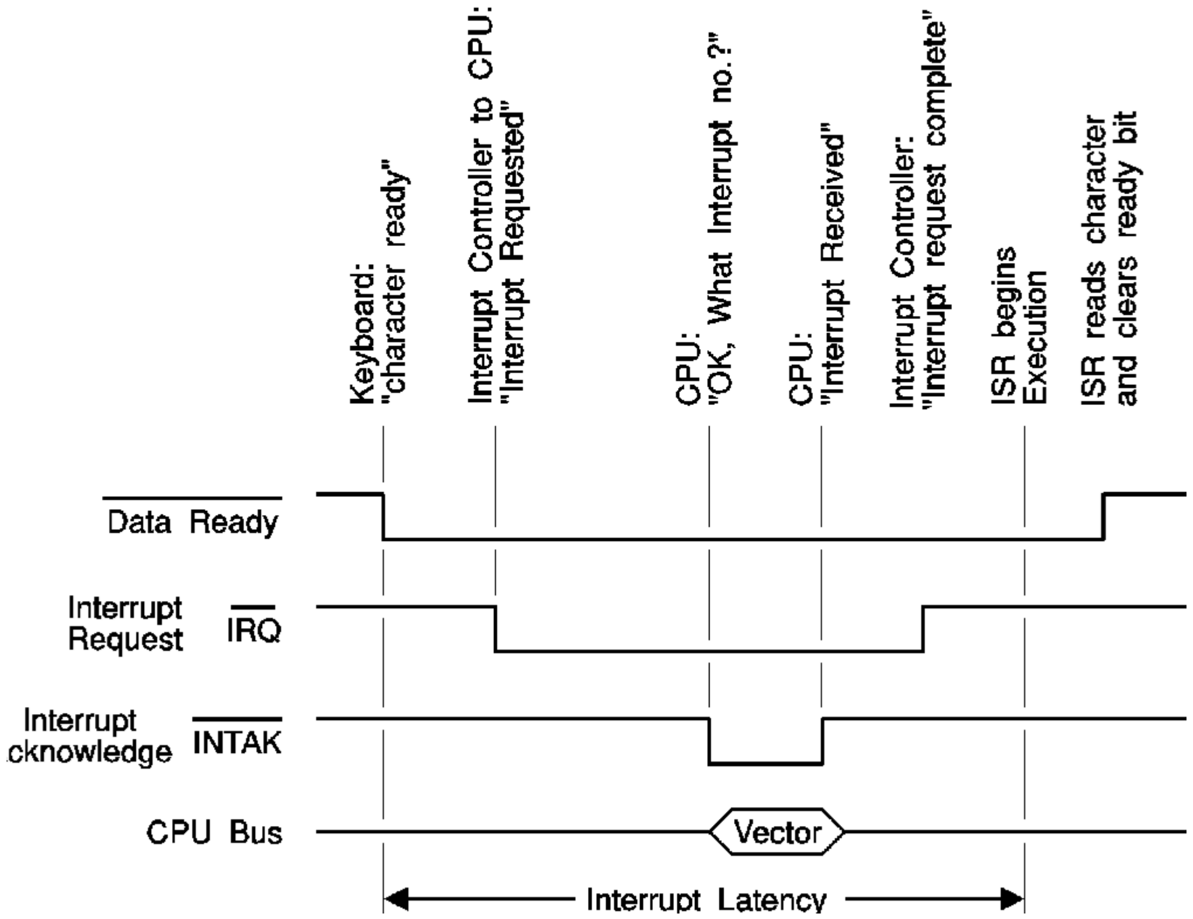


Example: Interrupt Keyboard Input

- The event
  - Pressing a key
- Interrupt generated
  - Selects keyboard ISR
- Interrupt is processed
  - CPU suspends main
  - CPU calls ISR
  - ISR stores key code
  - ISR returns to main



Example - Keyboard Interrupt Timing


Keyboard: "character ready"
Interrupt Controller to CPU: "Interrupt Requested"
CPU: "OK, What Interrupt no.?"
CPU: "Interrupt Received"
Interrupt Controller: "Interrupt request complete"
ISR begins Execution
ISR reads character and clears ready bit
Data Ready
Interrupt Request IRQ
Interrupt cknowledge INTAK
CPU Bus
Vector
Interrupt Latency

Interrupt Latency

- Interrupt latency is the CPU time it takes to recognize, and start up the ISR
- Non-useful "overhead" delay time
- We want to minimize this delay
- Determined by hardware and software
  - Hardware latency determined by chip design
  - We control the software latency

Program Structure

- Interrupt driven programs have 3 parts:
  - Initialization
    - Prepares necessary variables for interrupt programming
  - Main program (not event driven)
    - Performs non-time critical processing
  - Interrupt service routines (ISRs)
    - Processes the interrupt events

Initialization

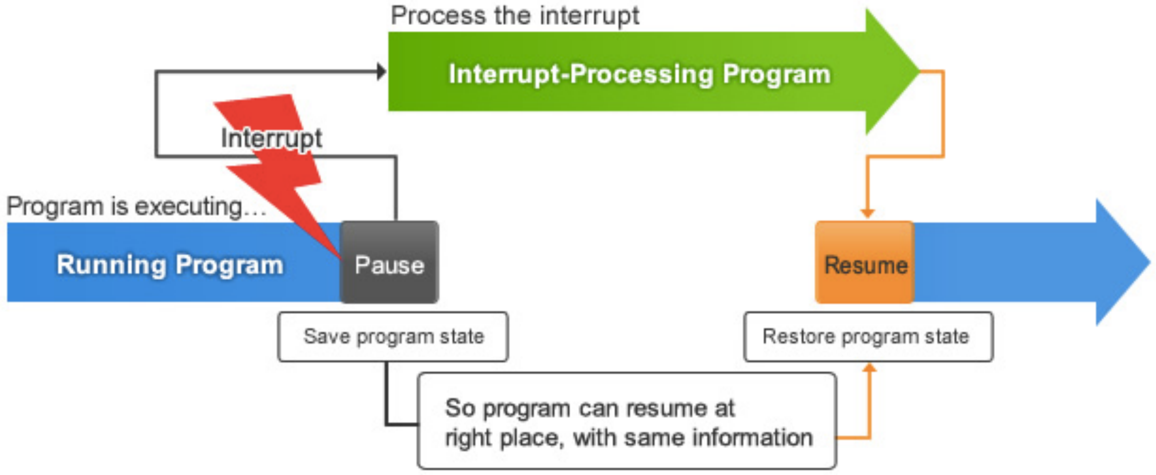
- Only happens once (unless there is an exception)
- Puts all variables, memory, in known state
  - Initialize variables, flags, I/O, IRQs
- Sets up CPU interrupt control system
  - Initialize interrupt priorities, enables
  - Clear pending interrupt requests
- Enables interrupts to occur

Main Program

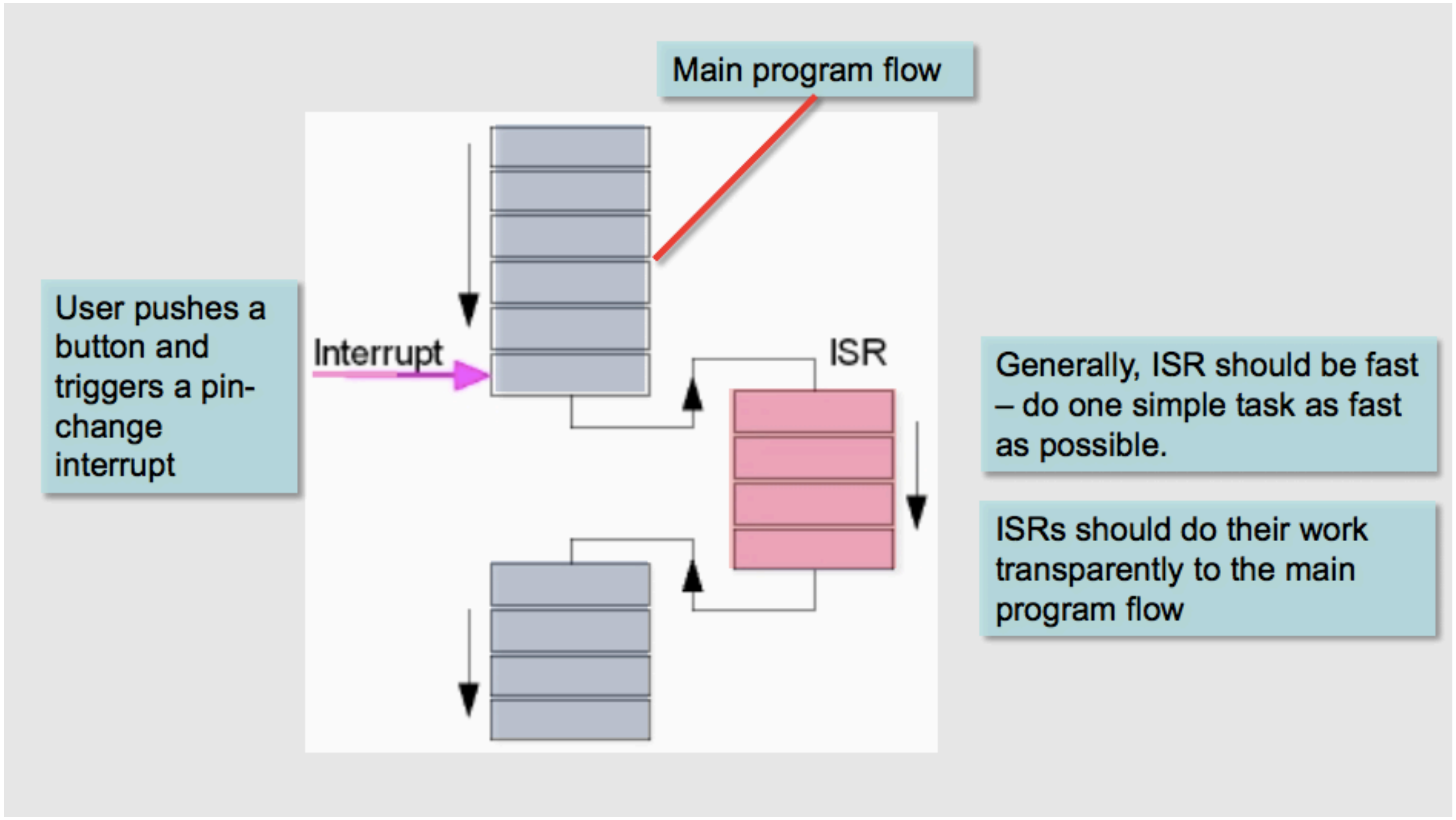
- Processes non-critical functions
  - The things that can be done at any time
- Anything which is not tied to an event
- When there’s nothing better to do:
  - May just wait in a tight loop
  - Execute software diagnostics
  - Execute hardware diagnostics

Interrupt Service Routine (ISR)

- External event causes interrupt request
  - CPU interrupts the running program
  - CPU “CALLs” the ISR
- When an interrupt is processed:
  - ISR must save the processor state
  - ISR processes the
  - ISR restores the p
  - ISR returns to the



ISR workflow



Generally, ISR should be fast – do one simple task as fast as possible.

ISRs should do their work transparently to the main program flow

Source: University of Iowa: ECE 55:036

Useful Terminology About Interrupts

Basic Interrupt Terminology

- ***Interrupt pins*****:** Set of pins used in hardware interrupts
- ***Interrupt Service Routine (ISR) or Interrupt handler*****:** code used for handling a specific interrupt
- ***Interrupt priority*****:** In systems with more than one interrupt inputs, some interrupts have a higher priority than other
- ***Interrupt vector*****:** Code loaded on the bus by the interrupting device that contains the Address (segment and offset) of specific interrupt service routine
- ***Interrupt Masking*****:** Ignoring (disabling) an interrupt
- ***Non-Maskable Interrupt*****:** Interrupt that cannot be ignored (power-down)

Re-entrant Code

- Code that can be interrupted, and called again one or more times, before completing execution
  - Must use only local resources (i.e. variables)
  - Cannot modify global resources
  - Compiler libraries are usually not re-entrant
  - Most OS functions are not re-entrant
  - Code that uses dedicated hardware is not
- Why is it important to have re-entrant code vs. not?

Critical Code Segments

- A code sequence that can't be interrupted without potential for error
  - If an interrupt occurs inside one of these segments of code, an error can occur
  - Simplest fix is to disable interrupts before critical segment and enable afterward
  - Example: two processes use printer and signal use of the printer with a flag

Critical Code Segments

- If interrupted, may result in errors
  - Actual occurrence often very low probability
- Occurs when resources are shared
- Mutually exclusive access required
- One solution: disable interrupts
  - Why is this bad?
- Other solution: use of semaphore (a.k.a. mutex) to control access to critical code segments
  - Topic of operating systems class

Type of Interrupts

- Single vs. multiple (nested)
  - Can one ISR interrupt another?
- Multiple prioritized
  - Can a higher priority ISR interrupt a lower?
- Maskable vs. non-maskable
  - Can interrupts be disabled in code?
- Level vs. edge-triggered
  - Sampled or transition Induced

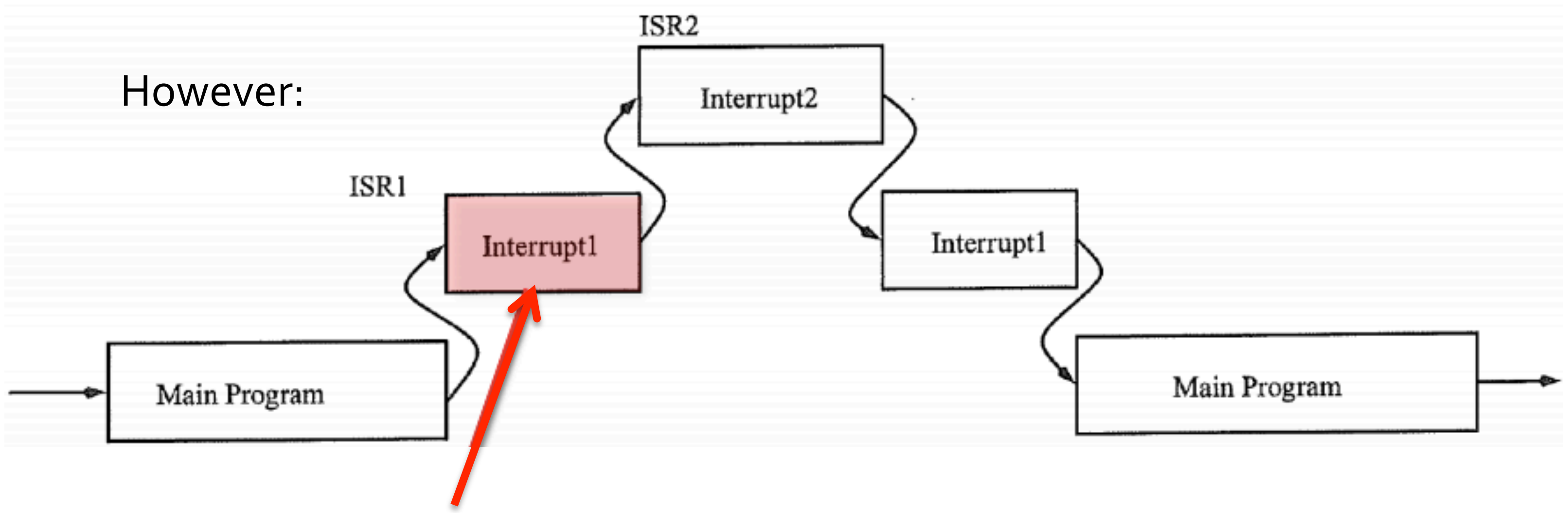
Single vs. Multiple Interrupts

- Some simple microcontrollers only allow one interrupt
- Most microcontrollers allow nested interrupts
  - ISR must re-enable interrupt(s)
  - Potential for calling ISR more than once
- In order to make good use of multiple interrupts, most processors also include some form of prioritized interrupts

Nested Interrupts

- When an ISR is invoked, the interrupts are turned off globally
- When an ISR ends, it turns on interrupts globally ("reti" instruction – slide 51)
- Thus, normally, an ISR will not be interrupted by other ISRs

However:



If this ISR enables interrupts (using "**sei**" instruction, then other ISRs can interrupt it) → We will see what this instruction is (slide 46)

Source: University of Iowa: ECE 55:036

Prioritized Interrupts

- Higher priority events supersede low
- Fixed and variable priorities

- Interrupts:
  - Fixed priority within a level (hardware)
  - Two levels (low/high) under program control

Maskable vs. Non-maskable

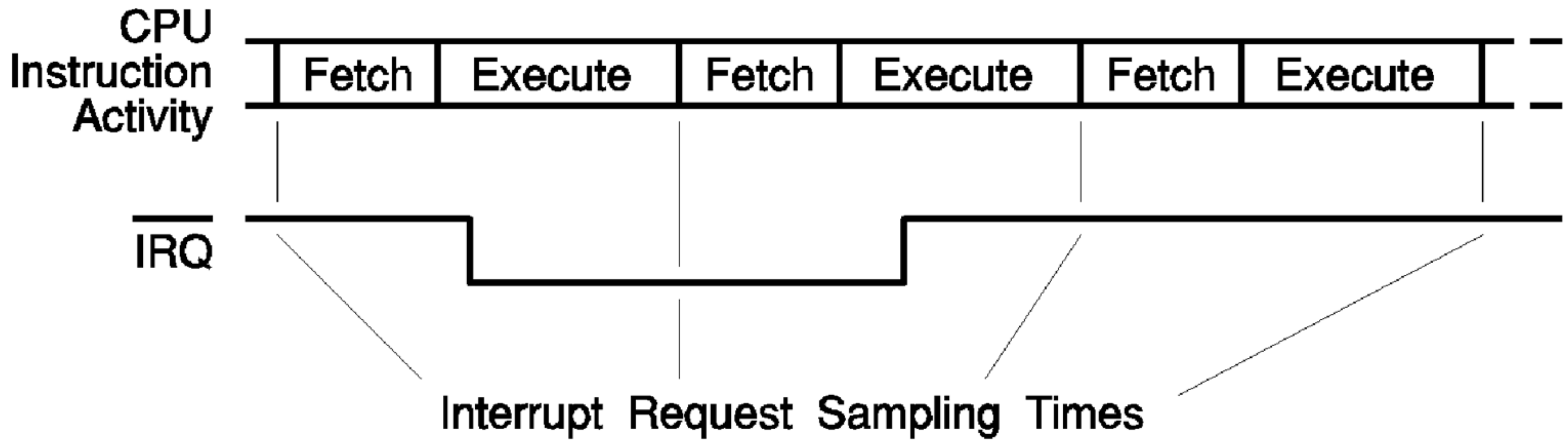
- Maskable interrupts can be disabled by code
- Global disable
  - e.g. The “Enable All” bit
- Specific, individual interrupt enables
  - e.g. Enable the Timer 1 interrupt
- Non-maskable interrupts can’t be turned of by the program

Level vs. Edge

- Level:
  - CPU samples the interrupt request (IRQ) input at the end of every instruction, and calls the appropriate ISR if the IRQ is active
- Edge:
  - Signal transition (edge) on IRQ input is held in a latch until the CPU processes the interrupt

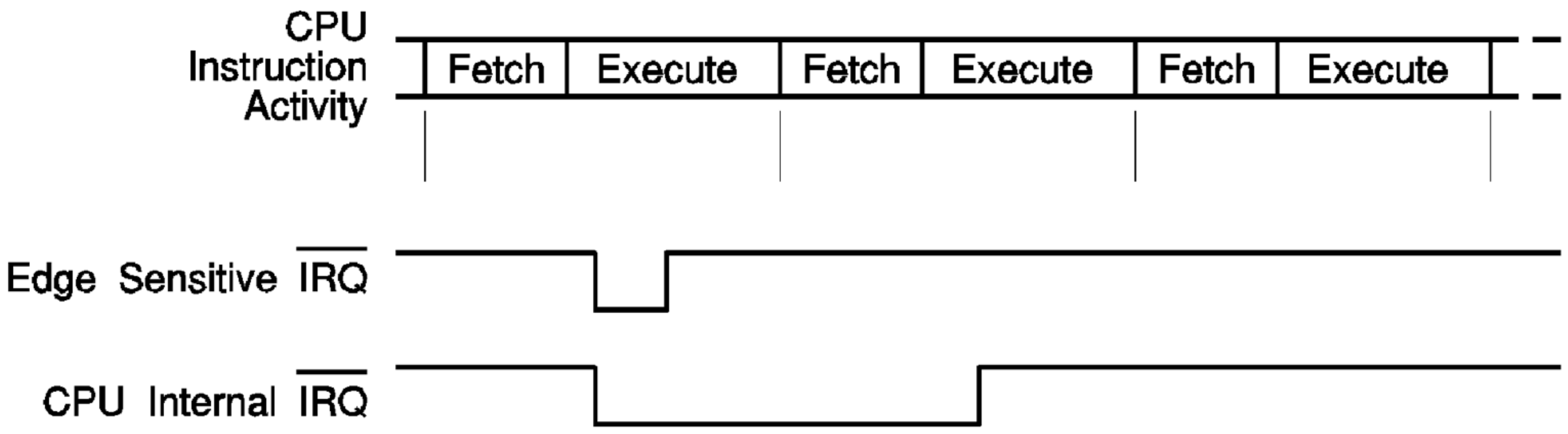
Level Sensitive Interrupt

- Interrupt input sampled by CPU at the end of every instruction, when interrupt is enabled
- Request must be active when sampled



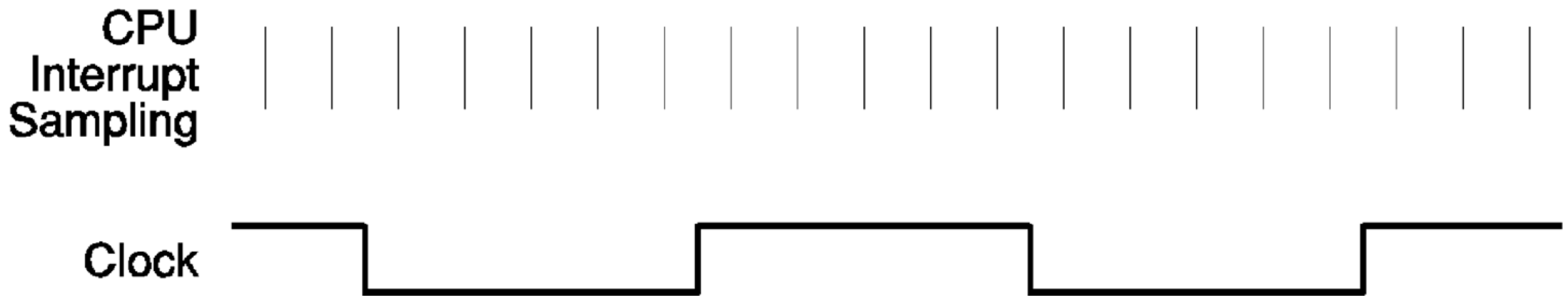
Edge Sensitive Interrupt

- Transition on input (edge) is latched internally by a flip flop in the CPU
- Internally latched version is then sampled just like a level sensitive input


CPU Instruction Activity
Fetch
Execute
Fetch
Execute
Fetch
Execute
Edge Sensitive $\overline{\text{IRQ}}$
CPU Internal $\overline{\text{IRQ}}$

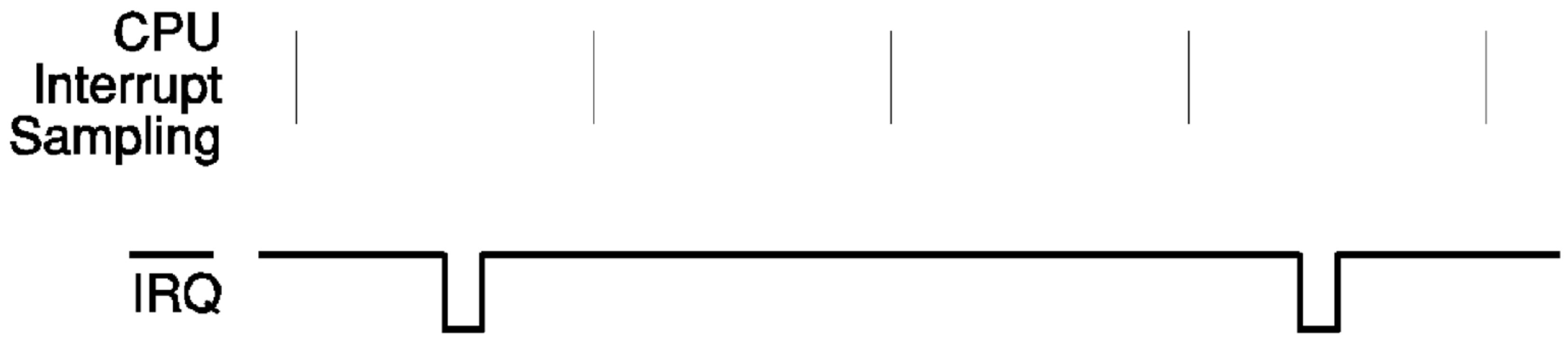
When to Use Edge Sensitive Interrupt

- When interrupt signal is too long
- e.g. 60 Hz square wave for clock
- Level sensitive would respond more than once



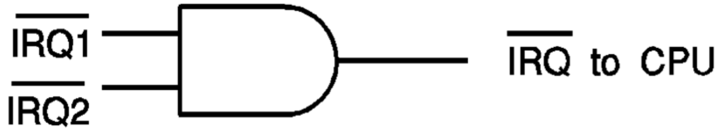
When to Use Edge Sensitive Interrupt

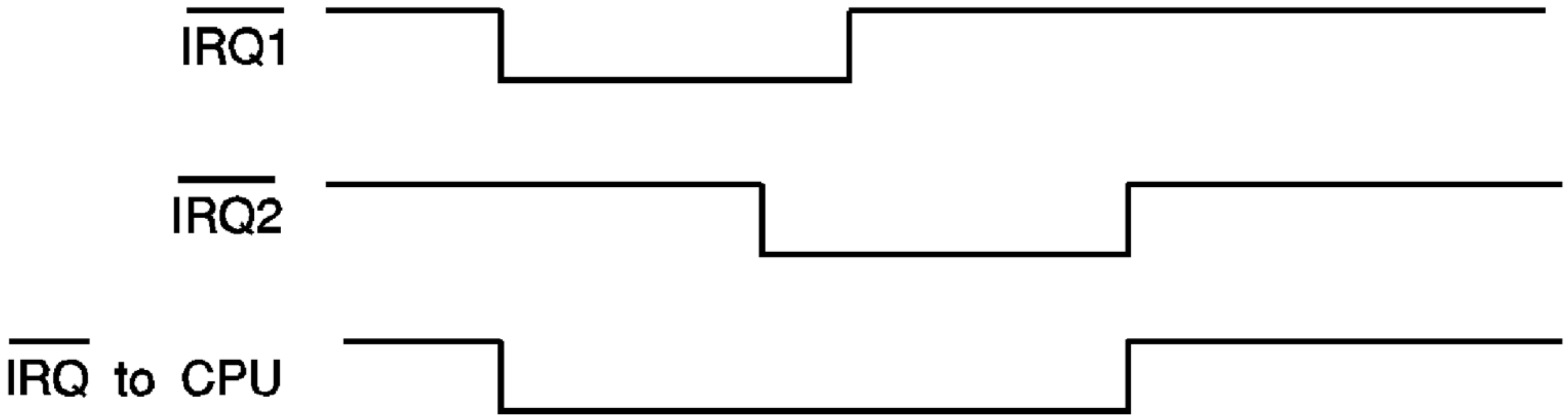
- When interrupt signal is too short
- e.g. very short pulse from a sensor
- Level sensitive would miss the event



When to Use Level Sensitive?

- Common IRQ line
- More than one event
- Only one edge on output!





Vectored vs. Non-vectored

- Non-vectored:
  - All interrupts go to the same ISR address
  - ISR must poll hardware to identify source
- Vectored:
  - Each interrupt input calls a separate ISR address
  - ISR addresses may be at a fixed address
  - Or may be stored in a table

Questions

- When is re-entrant code necessary?
- What happens if a critical code segment is interrupted?
- How does one avoid the problem?
- When must an edge sensitive interrupt be used?
- When must a level sensitive interrupt be used?

Interrupts in AVR

AVR Interrupt Table (ATmega328P)

Vector Number	Interrupt Definition	Vector Name	Vector Number	Interrupt Definition	Vector Name
1	External pin, power-on reset, brown-out reset, watchdog reset	RESET	14	Timer/Counter1 Overflow	TIMER1_OVF_vect
2	External Interrupt Request 0	INT0_vect	15	Timer/Counter0 Compare Match A	TIMER0_COMPA_vect
3	External Interrupt Request 1	INT1_vect	16	Timer/Counter0 Compare Match B	TIMER0_COMPB_vect
4	Pin Change Interrupt Request 0	PCINT0_vect	17	Timer/Counter0 Overflow	TIMER0_OVF_vect
5	Pin Change Interrupt Request 1	PCINT1_vect	18	SPI Serial Transfer Complete	SPI_STC_vect
6	Pin Change Interrupt Request 2	PCINT2_vect	19	USART Rx Complete	USART_RX_vect
7	Watchdog Time-out Interrupt	WDT_vect	20	USART Data Register Empty	USART_UDRE_vect
8	Timer/Counter2 Compare Match A	TIMER2_COMPA_vect	21	USART Tx Complete	USART_TX_vect
9	Timer/Counter2 Compare Match B	TIMER2_COMPB_vect	22	ADC Conversion Complete	ADC_vect
10	Timer/Counter2 Overflow	TIMER2_OVF_vect	23	EEPROM Ready	EE_READY_vect
11	Timer/Counter1 Capture Event	TIMER1_CAPT_vect	24	Analog Comparator	ANALOG_COMP_vect
12	Timer/Counter1 Compare Match A	TIMER1_COMPA_vect	25	Two-wire Serial Interface	TWI_vect
13	Timer/Counter1 Compare Match B	TIMER1_COMPB_vect	26	Store Program Memory Read	SPM_READY_vect

Be careful that this table changes from device to device

Page 65 (section 12.4) in the complete AVR datahseet

Coding Interrupts in AVR C

- Include interrupt macros and device vector definitions
  - #include <avr/interrupt.h>
  - #include <avr/io.h>
- Define ISR using macro and appropriate vector name: by default the compiler determines all registers that will be modified and saves them (prologue code) and restores them for you (epilog)
  - ISR(UART0_RX_vect){
  - // ISR code to execute here
  - }
- Somewhere in main or function code
  - sei(); //Enable global interrupts
- Enable specific interrupts of interest in corresponding control registers. Ex:
  - UCSRB |= (1<<RXCIE); //enable interrupt

Source: Mohsenin. C Programming and Embedded Systems, UMBC

ISR Implementation Practices

- ISRs affect the normal execution of program and can block handling of other interrupts
- Common strategy for ISR is to keep it as short as possible
- Creates stable timing and avoids system from being flooded with handling certain ISRs and not being able to service other interrupts fast enough
- Each ISR should do only what it needs to do at the time of the event
- If long ISRs are needed, consider allowing nested interrupts

Source: Mohsenin. C Programming and Embedded Systems, UMBC

AVR Interrupt Response Time

- The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum. After four clock cycles the program vector address for the actual interrupt handling routine is executed.
- During this four clock cycle period, the Program Counter is pushed onto the Stack.
- The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served.
- If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.
- A return from an interrupt handling routine takes four clock cycles.
- During these four clock cycles, the Program Counter (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.

from the complete datasheet

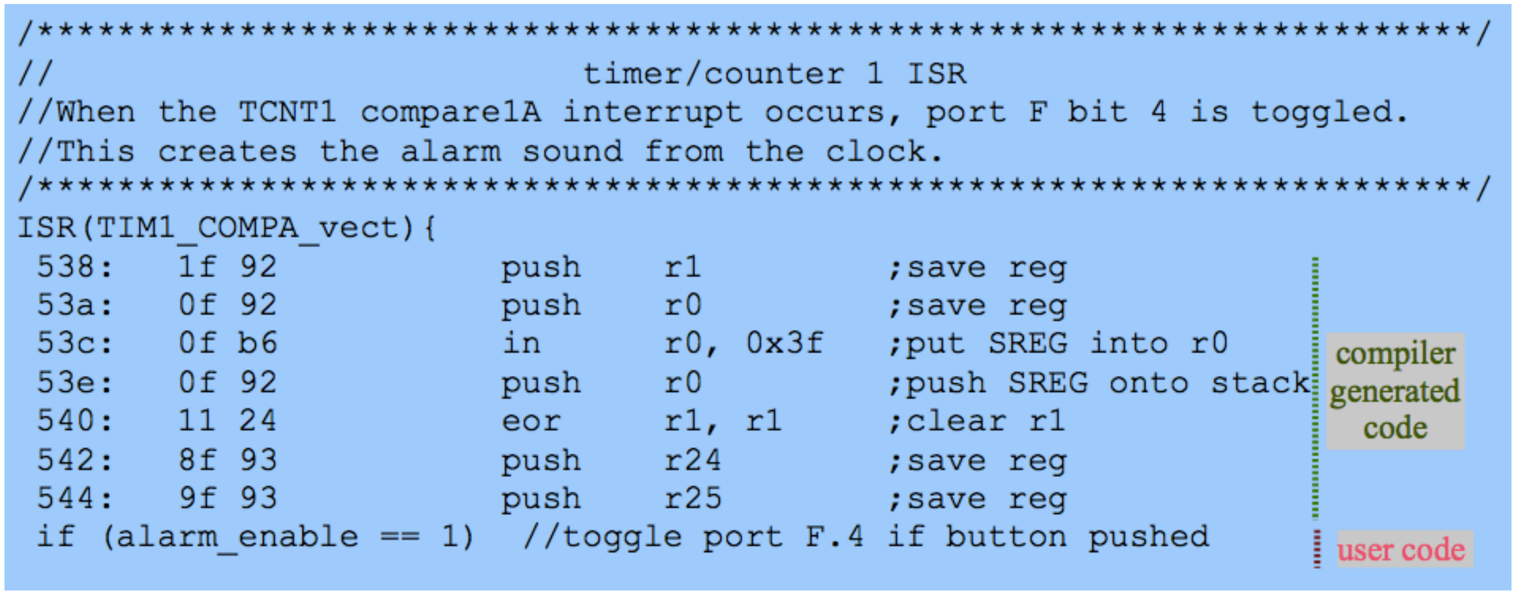
AVR Interrupt Servicing

1. In response to the interrupt, the CPU finishes any pending instructions and then ceases fetching further instructions. Global Interrupt Enable (GIE) bit is cleared.
2. Hardware pushes the program counter on the stack.
3. The CPU fetches the instruction from the interrupt vector table that corresponds to the interrupt. This instruction is usually “jmp, address”. The address is the address of the ISR.

Source: Traylor. System Design with Microcontrollers, Oregon State

AVR Interrupt Servicing

4. The CPU then begins to execute the ISR code. The first part of the ISR is compiler generated code that pushes the status register on the stack as well as any registers that will be used in the ISR.



```
/*********************************************************************/
//                          timer/counter 1 ISR
//When the TCNT1 compare1A interrupt occurs, port F bit 4 is toggled.
//This creates the alarm sound from the clock.
/*********************************************************************/
ISR(TIM1_COMPA_vect){
 538:    1f 92          push    r1          ;save reg
 53a:    0f 92          push    r0          ;save reg
 53c:    0f b6          in      r0, 0x3f    ;put SREG into r0
 53e:    0f 92          push    r0          ;push SREG onto stack
 540:    11 24          eor     r1, r1      ;clear r1
 542:    8f 93          push    r24         ;save reg
 544:    9f 93          push    r25         ;save reg
 if (alarm_enable == 1)  //toggle port F.4 if button pushed
```

AVR Interrupt Servicing

5. Just before the ISR is done, compiler generated code pops the saved registers as well as the status register. Then the RETI instruction is executed. This restores the program counter from the stack. Global Interrupt Enable bit gets set again.

```
55a:    9f 91           pop     r25   ;restore regs
55c:    8f 91           pop     r24   ;restore regs
55e:    0f 90           pop     r0    ;put SREG back into r0
560:    0f be           out     0x3f, r0 ;put r0 into SREG
562:    0f 90           pop     r0    ;restore regs
564:    1f 90           pop     r1    ;restore regs
566:    18 95           reti
```

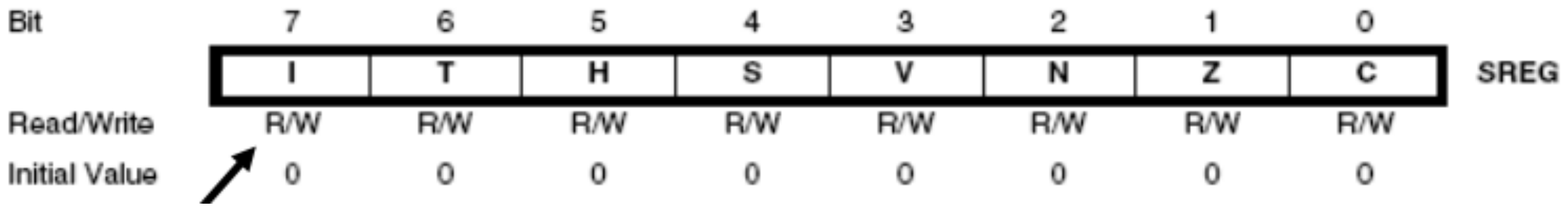
compiler generated code

6. The CPU resumes executing the original instruction stream.

Source: Traylor. System Design with Microcontrollers, Oregon State

AVR Enabling Interrupts

- Global Interrupt Enable (GIE) bit must be set in the AVR global status register (SREG)

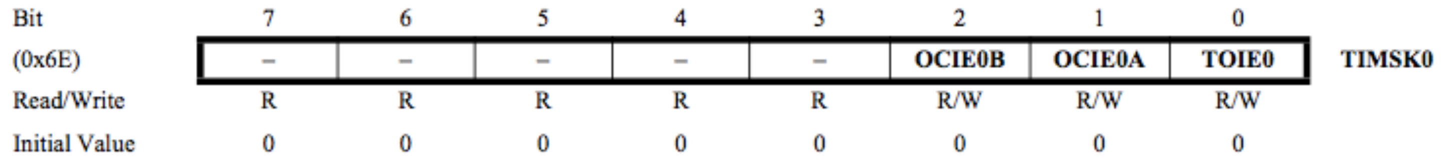


Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Global Interrupt Enable (GIE): This bit must be set for interrupts to be enabled. To set GIE:
  - `sei(); //global interrupt enable`
- It is cleared by hardware after an interrupt has occurred, but may be set again by software [ISR(xxx_vec, ISR_NOBLOCK)] or manually with the sei() to allow nested interrupts.
- It is normally set by the RETI instruction to enable subsequent interrupts. This is done automatically by the compiler.

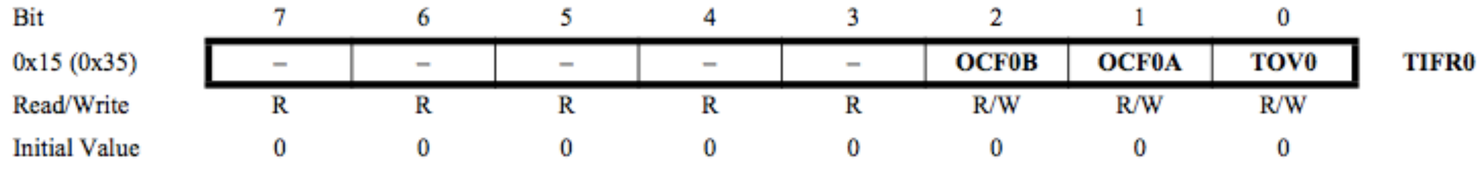
AVR Enabling Interrupts

- The individual interrupt enable bits must be set in the proper control register.
  - For example, for timer counter 0, the timer overflow bit (TOV)



Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- TOIE0 must be set to 1 to allow the TCNT0 over flow bit to cause an interrupt
  - When this bit is written to 1 and the I-flag in SREG is set, the Timer/Counter0 overflow interrupt is enabled
- The interrupt occurs when the TOV0 flag becomes set. Once you enter the ISR, it is reset automatically by hardware



Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Example Interrupt Codes

- C code:

```
ISR(TIMER1_OVF_vect){
      external_count++;
}
```

Example Interrupt Codes

- The compiler generated ISR:

```
ISR(TIMER1_OVF_vect) {
   cc: 1f 92 push r1 ##save r1
   ce: 0f 92 push r0 ##save r0
   d0: 0f b6 in r0, 0x3f ##put SREG into r0 (SREG=0x3F)
   d2: 0f 92 push r0 ##push SREG onto stack
   d4: 11 24 eor r1, r1 ##clear r1
   d6: 8f 93 push r24 ##push r24 (it is about to be used)
   external_count++;
   d8: 80 91 00 01 lds r24, 0x0100 ##load r24 with external_count
   dc: 8f 5f subi r24, 0xFF ##subtract 255 from r24 (++)
   de: 80 93 00 01 sts 0x0100, r24 ##store external_count to SRAM
   e2: 8f 91 pop r24 ##all done with r24, put it back
   e4: 0f 90 pop r0 ##pop SREG from stack into r0
   e6: 0f be out 0x3f, r0 ##put r0 contents into SREG
   e8: 0f 90 pop r0 ##restore r0
   ea: 1f 90 pop r1 ##restore r1
   ec: 18 95 reti ##return from interrupt
}
```

Source: Traylor. System Design with Microcontrollers, Oregon State

Effecting main program flow with ISRs

- ISRs are never called by the main routine. Thus,
  - Nothing can be passed to them (there is no "passer")
  - They can return nothing (its not a real function call)
- So, how can it effect the main program flow?
  - Global variable – never safe – should be careful
  - Volatile variable!!
- Compiler has an optimizer component
  - o2 level optimization can optimize away some variables
  - It does so when it sees that the variable cannot be changed within the scope of the code it is looking at
  - Variables changed by the ISR are outside the scope of main()
  - Thus, they get optimized away
- Volatile tells the compiler that the variable is shared and may be subject to outside change elsewhere

Source: Traylor. System Design with Microcontrollers, Oregon State

Example

```
volatile uint8_t tick; //do not make optimizations
ISR(TIMER1_OVF_vect){
      tick++; //increment my tick count
}
main(){
      while(tick == 0x00){
            // do something
      }
…
```

Without the volatile modifier, -o2 optimization removes tick because nothing in while loop can ever change *tick*.

Example interrupt-based code for AVR

Original version - Set Timer 0 for 1ms

```
// this code sets up a timer0 for 1ms @ 16Mhz clock cycle
// using no interrupts

#include <avr/io.h>

int main(void){
    TCCR0A |= (1 << WGM01); // Set the Timer Mode to CTC

    OCR0A = 0xF9; // Set the value that you want to count to

    // set prescaler to 64 and start the timer
    TCCR0B |= (1 << CS01) | (1 << CS00);

    while (1){

        // wait for the overflow event
        while ( (TIFR0 & (1 << OCF0A) ) == 0){}

        TIFR0 |= (1 << OCF0A); // reset the overflow flag
         //DO SOMETHING

    }
}
```

Interrupt version - Set Timer 0 for 1ms

```
// this code sets up a timer0 for 1ms @ 16Mhz clock cycle
// using interrupts

#include <avr/io.h>
#include <avr/interrupt.h>

int main(void){
    TCCR0A |= (1 << WGM01); // Set the Timer Mode to CTC

    OCR0A = 0xF9; // Set the value that you want to count to

    TIMSK0 |= (1 << OCIE0A);    //Set the ISR COMPA vect
    sei();         //enable interrupts

    // set prescaler to 64 and start the timer
    TCCR0B |= (1 << CS01) | (1 << CS00);

    while (1){}
}
ISR (TIMER0_COMPA_vect)  // timer0 Compare register A interrupt
{
    //DO SOMETHING
}
```

Original Version - Set Timer 0 for 1ms and 50% Duty Cycle

```
// this code sets up a timer0 for 1ms @ 16Mhz clock cycle and 50% duty cycle
// in order to function as a time delay at the beginning of the main loop
// using no interrupts

#include <avr/io.h>

int main(void){

    TCCR0A |= (1 << WGM01); // Set the Timer Mode to CTC

    OCR0A = 0xF9; //represents 1ms timer
    OCR0B = 0x7D; //represents 50% duty cycle

    TCCR0B |= (1 << CS01) | (1 << CS00); // set prescaler to 64 and start the timer

    while (1){
       //CREATE A LOGIC 1 SIGNAL (E.G. TURN ON LED)

        while ( (TIFR0 & (1 << OCF0B) ) == 0){} // wait for OCR0B overflow event

        TIFR0 |= (1 << OCF0B); // reset OCR0B overflow flag

        //CREATE A LOGIC 0 SIGNAL (E.G. TURN OFF LED)

        while ( (TIFR0 & (1 << OCF0A) ) == 0){} // wait for OCR0A overflow event

        TIFR0 |= (1 << OCF0A); // reset OCR0A overflow flag
    }
}
```

Interrupt Version - Set Timer 0 for 1ms and 50% Duty Cycle

```
// this code sets up a timer0 for 1ms @ 16Mhz clock cycle and 50% duty cycle
// in order to function as a time delay at the beginning of the main loop
// using interrupts

#include <avr/io.h>
#include <avr/interrupt.h>

int main(void){
    TCCR0A |= (1 << WGM01); // Set the Timer Mode to CTC

    OCR0A = 0xF9; //represents 1ms timer
    OCR0B = 0x7D; //represents 50% duty cycle

    TIMSK0 |= (1 << OCIE0A) | (1 << OCIE0B);    //Set the ISR COMPA vect and COMPB vec
    sei();         //enable interrupts

    TCCR0B |= (1 << CS01) | (1 << CS00); // set prescaler to 64 and start the timer

    while (1){}
}
ISR (TIMER0_COMPA_vect)  // timer0 Compare register A interrupt
{
    //CREATE A LOGIC 1 SIGNAL (E.G. TURN ON LED)
}
ISR (TIMER0_COMPB_vect)  // timer0 Compare register B interrupt

    //CREATE A LOGIC 0 SIGNAL (E.G. TURN OFF LED)
}
```