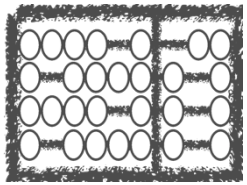
INF-741: Embedded systems programming for IoT

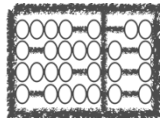
Prof. Edson Borin



Institute of Computing

University of Campinas

Interrupt-driven I/O



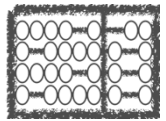
Agenda

I/O Example and Pooling

Interrupt-driven I/O

Interrupt Handling on AVR and Arduino

Lab Activities



Input/Output Example - Elevator

Two peripheral: 1 Input and 1 Output

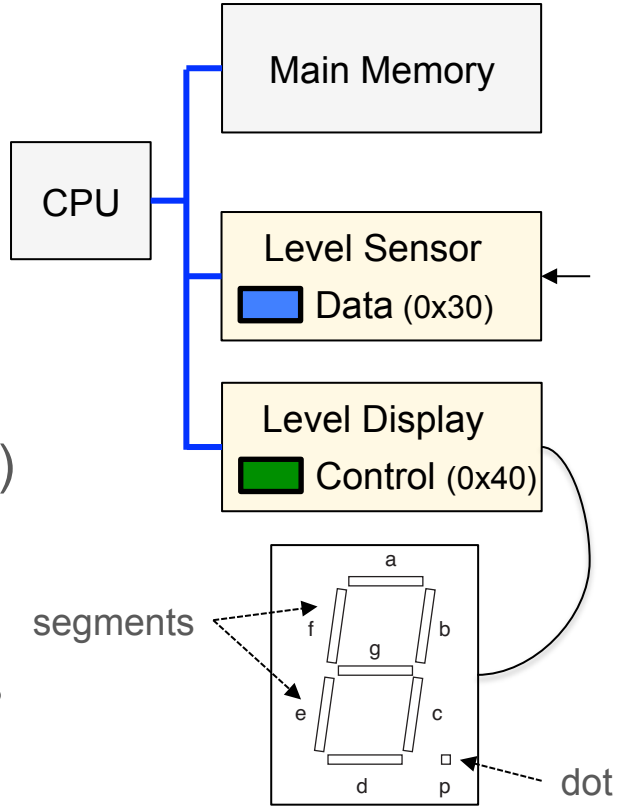
Input: Level sensor

- data register mapped to address 0x30
- contains the level the elevator is located at (0-9)

Output: Level display

- control register mapped to address 0x40
- 8 bits, each one controls one of the 7 segments the dot.

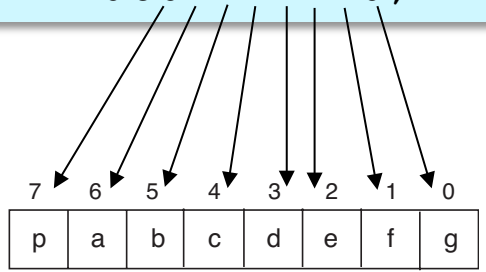
7	6	5	4	3	2	1	0
p	a	b	c	d	e	f	g

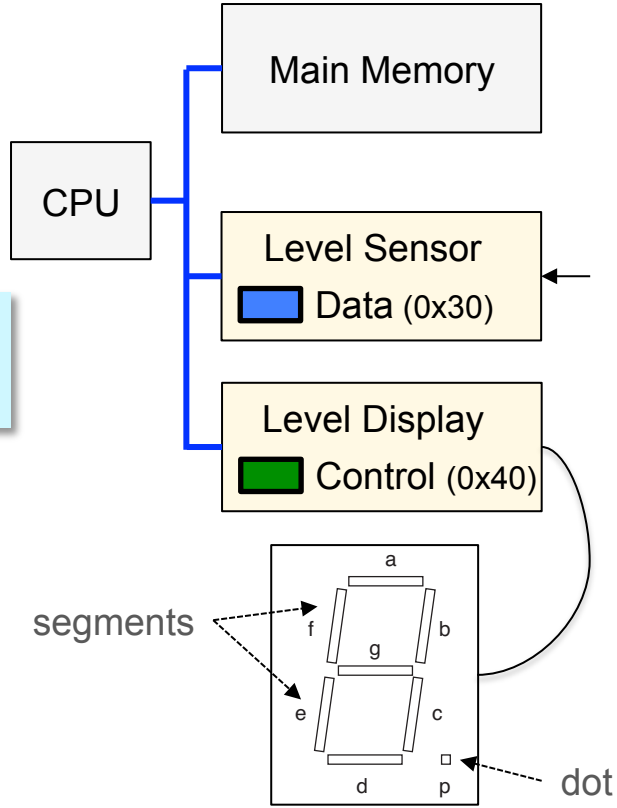


Input/Output Example - Elevator

Code to show value 0 on Level Display

```
volatile char* p = (char*) 0x40;
*p = 0b01111110;
```



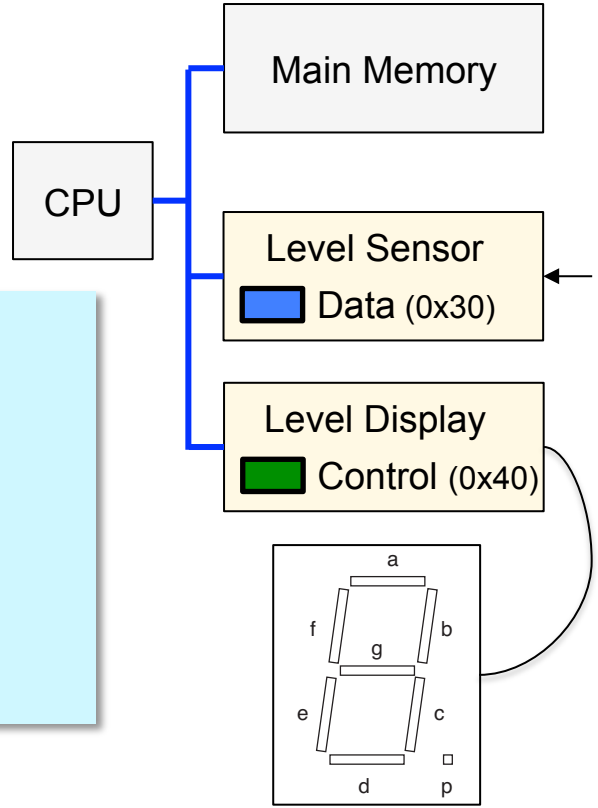


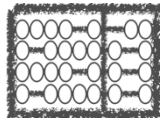
Input/Output Example - Elevator

Function to show a number on Level Display

```
char table[] = {0x7e,0x30,0x6d,0x79,0x33,
                0x5b,0x5f,0x70,0x7f,0x7b};

void show_number(int n)
{
  volatile char* p = (char*) 0x40;
  *p = table[n];
}
```

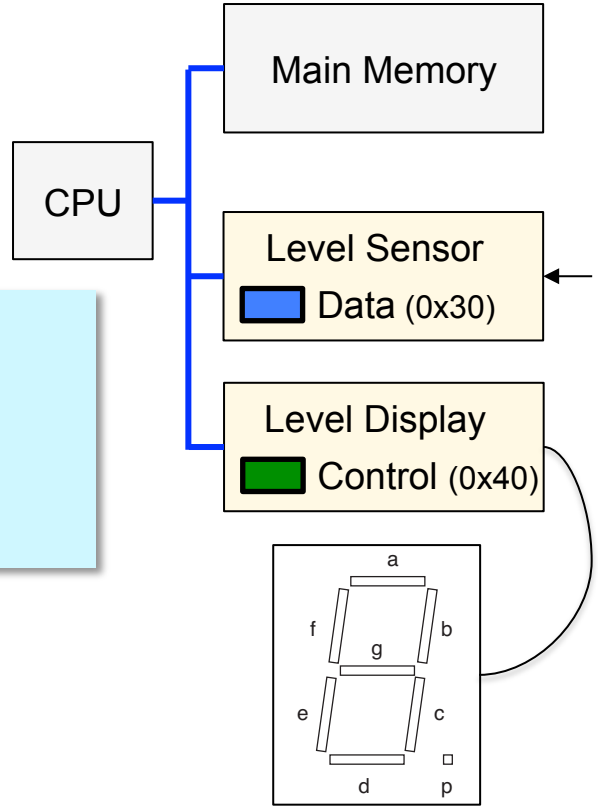


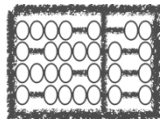


Input/Output Example - Elevator

Function to read level from Level Sensor

```
int read_level()
{
  volatile char* p = (char*) 0x30;
  return *p;
}
```

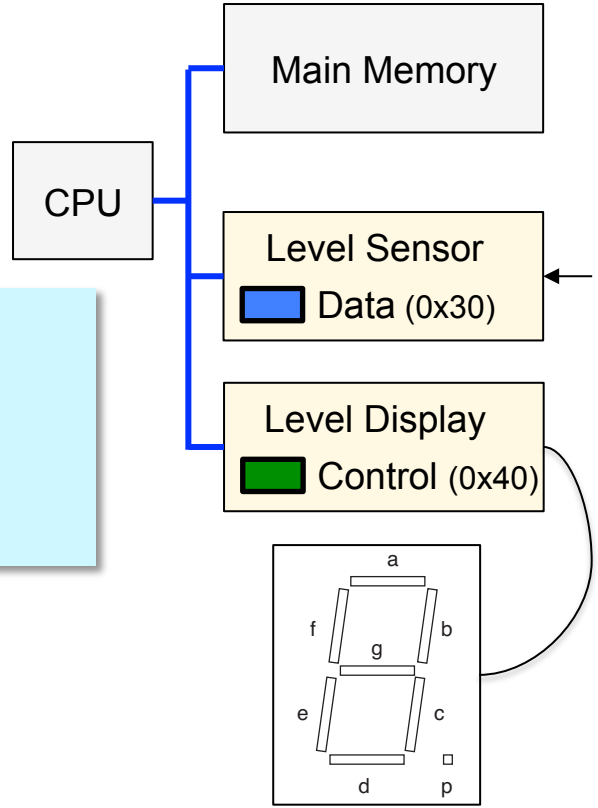


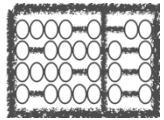


Input/Output Example - Elevator

Code to update Level Display

```
void update_level_display()
{
  int level = read_level();
  show_number(level);
}
```

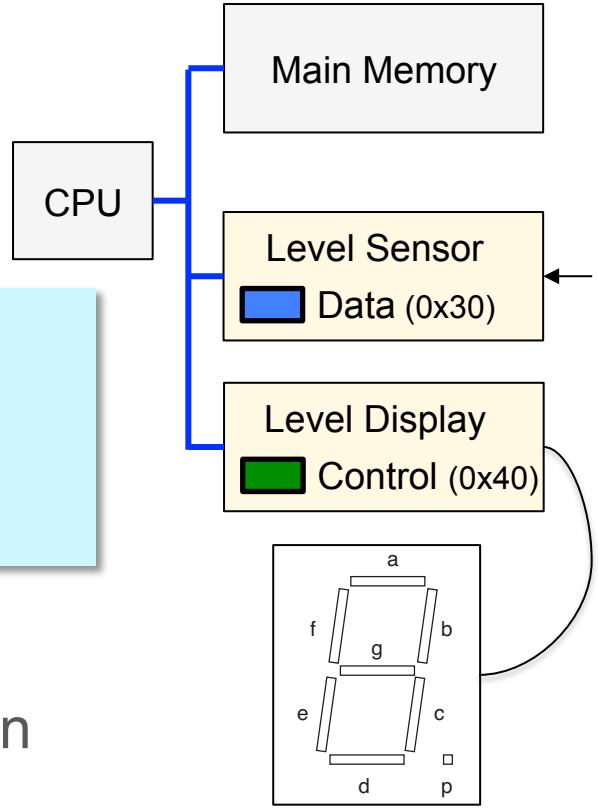




Input/Output Example - Elevator

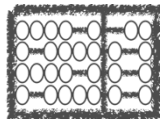
Code to update Level Display

```
void update_level_display()
{
  int level = read_level();
  show_number(level);
}
```



Assume the elevator goes up 8 levels.

How many times should we invoke the function update_level_display?

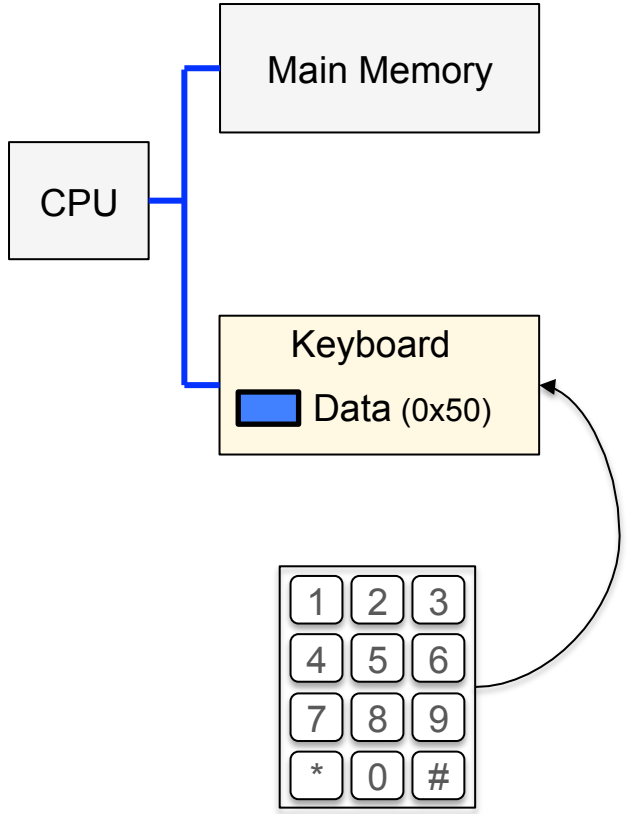


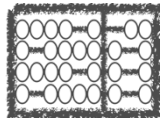
Input/Output Example - Keypad

1 Input peripheral

Data register mapped to address 0x50

- contains the code of the last key pressed





Input/Output Example - Keypad

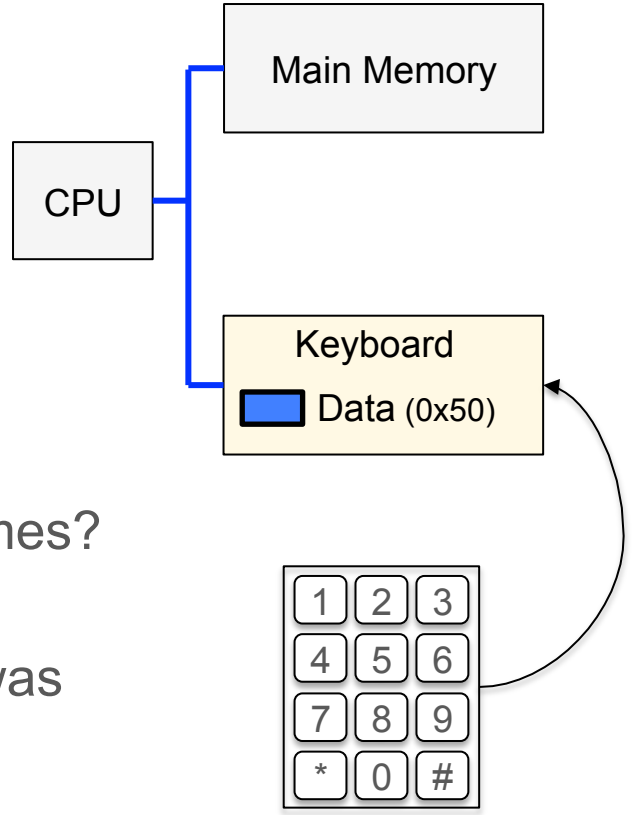
1 Input peripheral

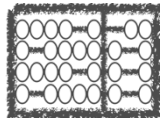
Data register mapped to address 0x50

- contains the code of the last key pressed

What happens if the keypad is pressed multiple times?

How to know the information at the Data register was already read?





Input/Output Example - Keypad

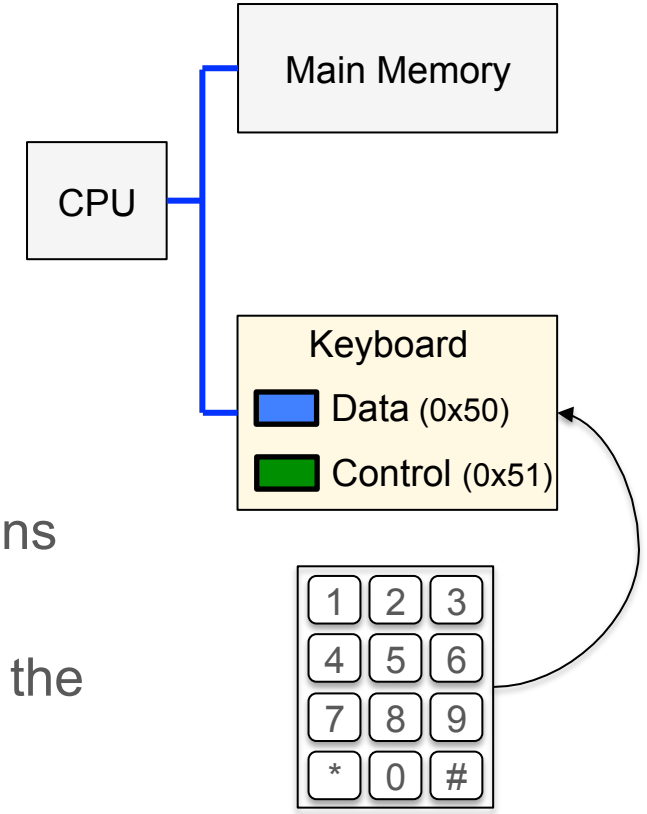
1 Input peripheral

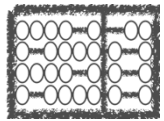
Data register mapped to address 0x50

- contains the code of the last key pressed

Control register mapped to address 0x51

- Bit 0 indicates whether the Data register contains new information - READY
- Bit 1 indicates whether a key was pressed and the Data register contained unread data - OVRN.

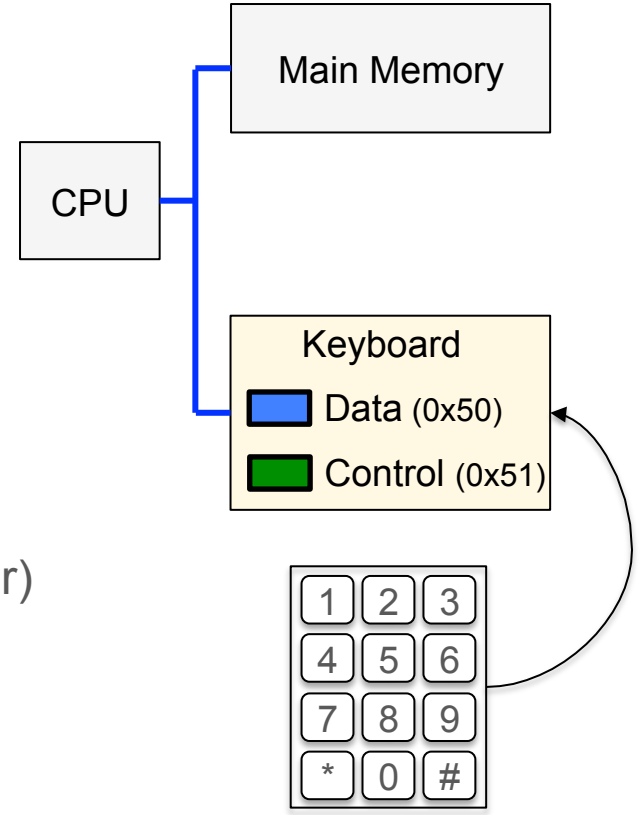


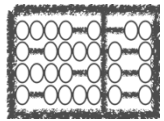


Input/Output Example - Keypad

Code to read keypad

1) Read status from control register
2) If there is no new data, then try again (busy waiting)
3) If there was a data loss (OVRN)
   - then handle error
4) Return the last key pressed (value at the data register)

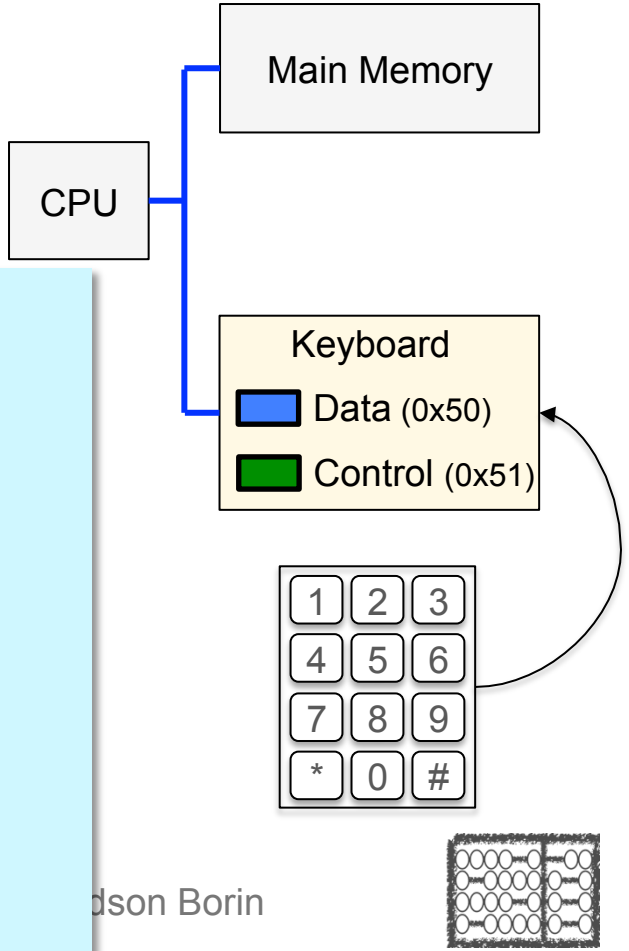




Input/Output Example - Keypad

Code to read keypad

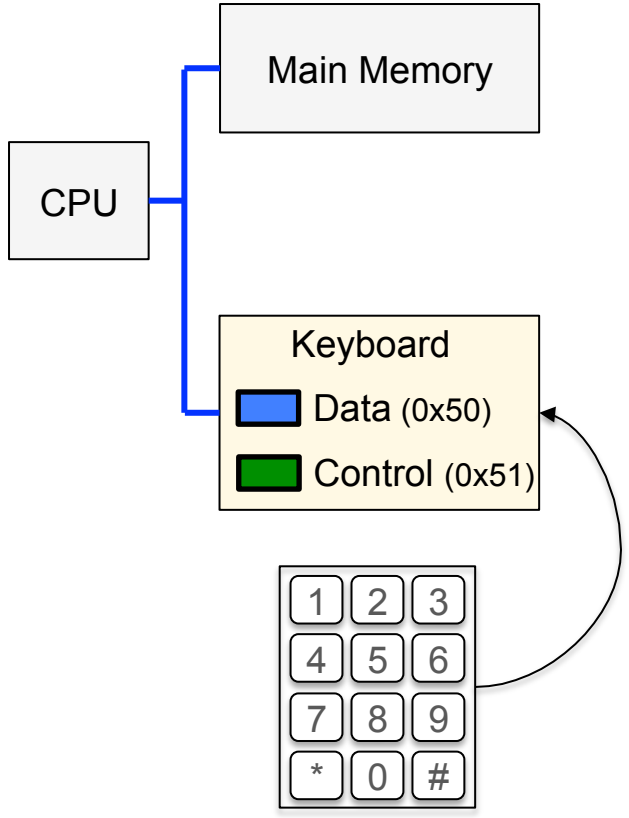
```
char read_key()
{
  char ctrl;
  volatile char* ctrl_reg = 0x51;
  volatile char* data_reg = 0x50;
  do {
    ctrl = *ctrl_reg;
  } while ( (ctrl & 0b00000001) == 0);
  if ( (ctrl & 0b00000010) != 0) {
    // Handle data overrun...
  }
  return *data_req;
}
```

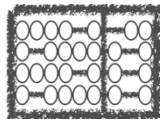


Input/Output Example - Keypad

Assume the user takes too long to press a key.

What does the processor do?





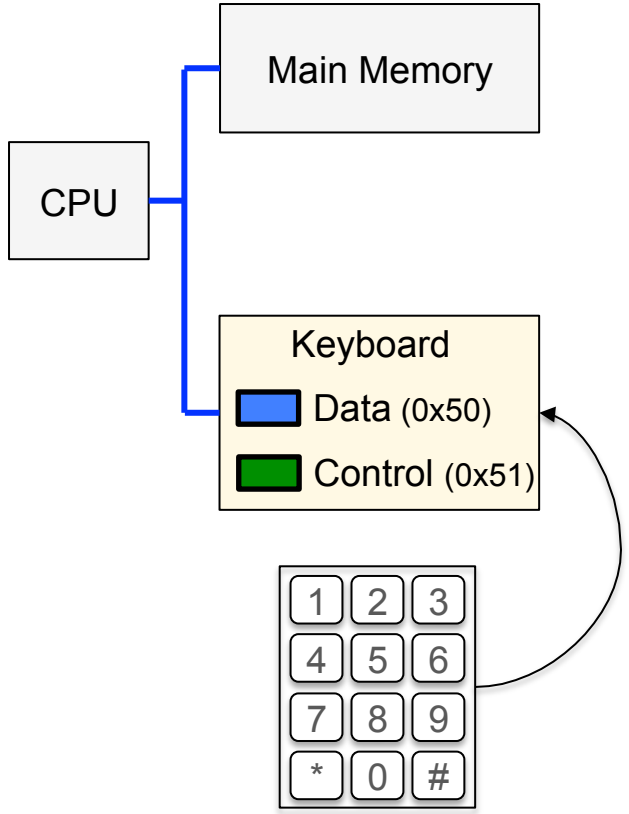
Input/Output Example - Keypad

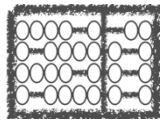
Assume the user takes too long to press a key.

What does the processor do?

How to improve it?

- Check the keypad from time to time and do something useful between checks. (Pooling)





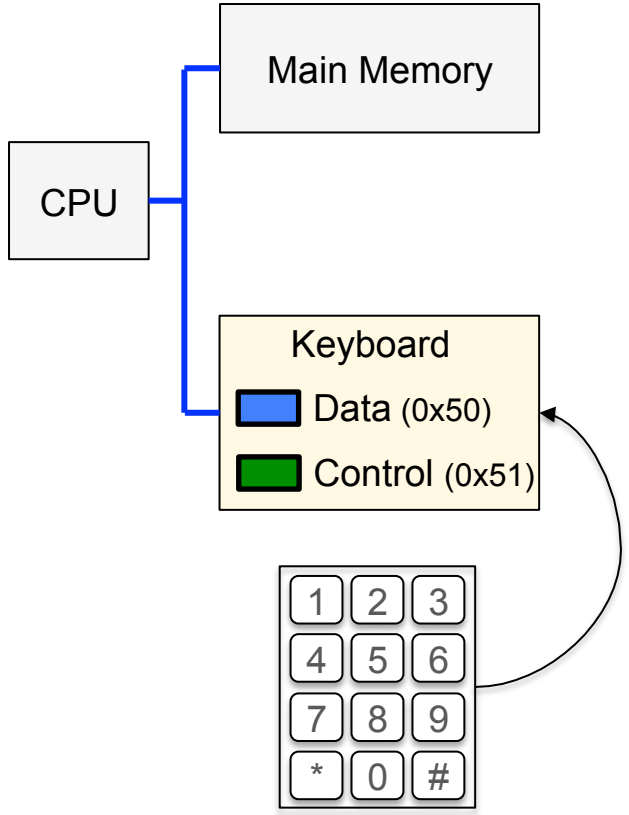
Input/Output Example - Keypad

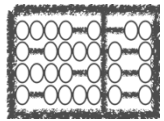
Assume the user takes too long to press a key.

What does the processor do?

How to improve it?

- Check the keypad from time to time and do something useful between checks. (Pooling)
  - User may press the keypad multiple times while the processor does something else. User may not be that fast, but what if we are programming a network device instead of a keypad?





Input/Output Example - Keypad

Assume the user takes too long to press a key.

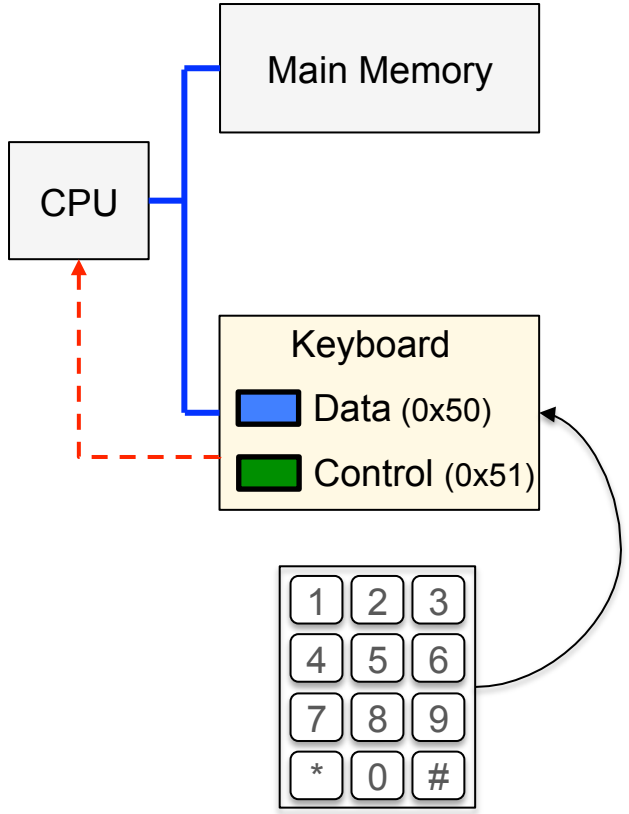
What does the processor do?

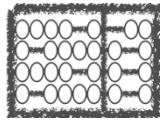
How to imp

- Check t
  someth
  - Use
    proc
    but w
    a keypad?

Interrupt-driven I/O

Peripheral informs the CPU when there is an event to be handled!





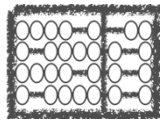
Agenda

I/O Example and Pooling

Interrupt-driven I/O

Interrupt Handling on AVR and Arduino

Lab Activities

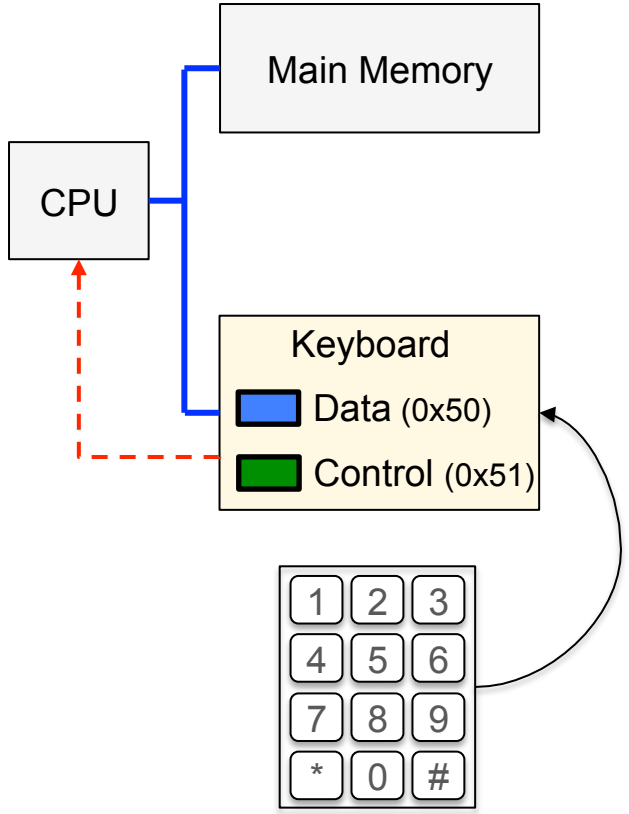


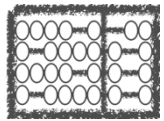
Interrupt-driven I/O

Peripheral informs the CPU when there is an event to be handled (Peripheral **interrupts** the CPU)

Example:

- When the keyboard is pressed the peripheral interrupts the CPU.
- The CPU stops whatever it was doing and invokes a routine to handle the keyboard interrupt.
- After handling, the CPU continues doing whatever it was doing before the interrupt.





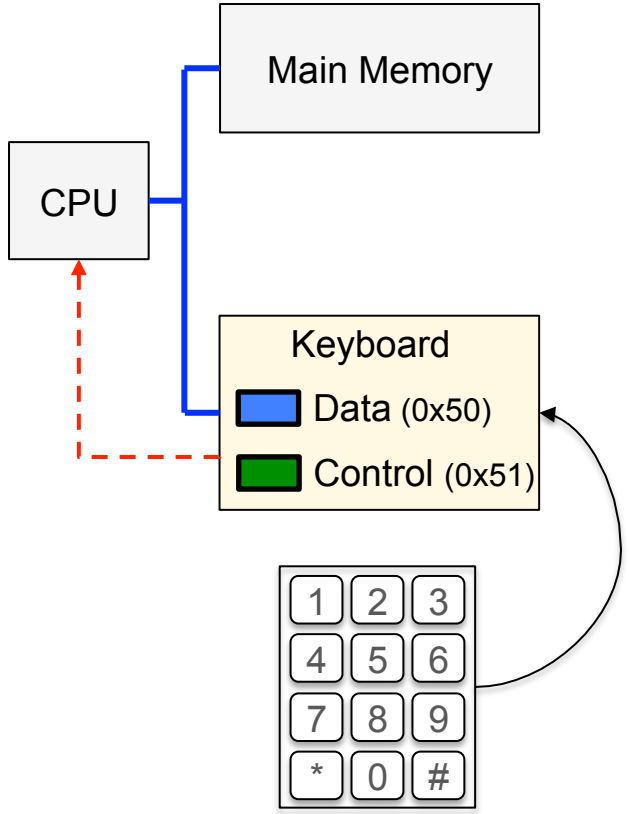
Interrupt-driven I/O

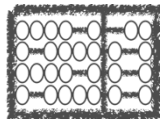
Periph is an event to be har PU)

Exampl

- When the oard is pressed the peripheral interrupts CPU.
- The CPU stops whatever it was doing and invokes a routine to handle the keyboard interrupt.
- After handling, the CPU continues doing whatever it was doing before the interrupt.

ISR: Interrupt Service Routine

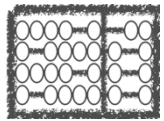




Interrupt-driven I/O

The CPU stops whatever it was doing and invokes a routine to handle the keyboard interrupt.

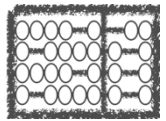
What happens to the program that was being executed by the processor?



Interrupt-driven I/O

```
@ Program that performs something useful 1000 times

main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
```

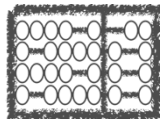


Interrupt-driven I/O

```
@ Program that performs something useful 1000 times

main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
```

Interrupt

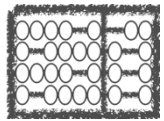


Interrupt-driven I/O

The CPU stops whatever it was doing and invokes a routine to handle the keyboard interrupt.

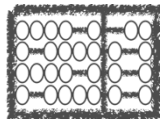
What happens to the program that was being executed by the processor?

- **Before handling the interrupt, it is important to save the "context" of the program that was being executed.** The context include:
  - CPU registers;
  - Memory used by the process;
- In order to preserve the context, usually:
  - Value of CPU registers are copied to memory.
  - Memory words used by the running process are not modified by the code that handles the interrupt.



Interrupt-driven I/O

```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```



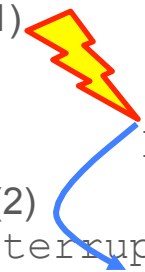
Interrupt-driven I/O

```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
 (1)    sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```

(1) Interrupt generated

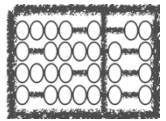
Interrupt-driven I/O

```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```



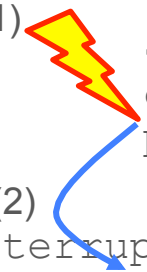
(1) Interrupt generated

(2) Execution flow is changed



Interrupt-driven I/O

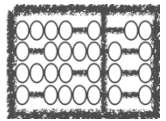
```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```



(1) Interrupt generated

(2) Execution flow is changed

(3) Context is saved



Interrupt-driven I/O

```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```

(1)

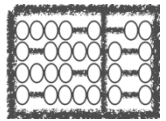
(2)

(1) Interrupt generated

(2) Execution flow is changed

(3) Context is saved

(4) Interrupt is handled



Interrupt-driven I/O

```
@ Program that performs something useful 1000 times
main:
        mov r4, #1000
loop:
        call    something_useful
        sub     r4, r4, #1
        cmp     r4, #0
        bne     loop
        ...
interrupt_sevice_routine:
        @ save context
        ...
        @ handle interrupt
        ...
        @ recover context
        ...
```

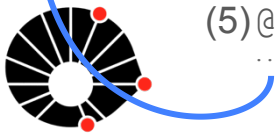
(1)

(2)

(3)

(4)

(5)



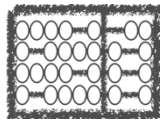
(1) Interrupt generated

(2) Execution flow is changed

(3) Context is saved

(4) Interrupt is handled

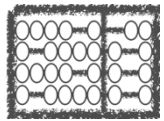
(5) Context is recovered



Interrupt-driven I/O

When an interrupt happens, the CPU changes the execution flow to execute the interrupt service routine (ISR).

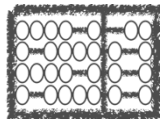
- To which address should it change the execution flow?



Interrupt-driven I/O

When an interrupt happens, the CPU changes the execution flow to execute the interrupt service routine (ISR).

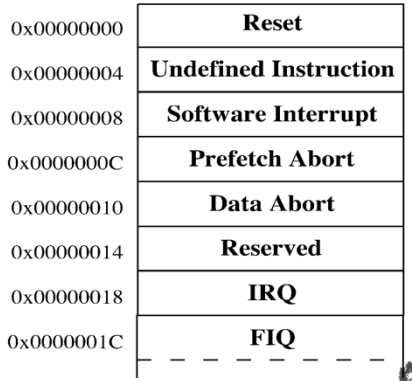
- To which address should it change the execution flow?
- Usually it is determined by an entry on a “**interrupt address table**”.

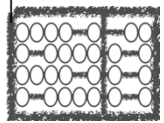


Interrupt-driven I/O

When an interrupt happens, the CPU changes the execution flow to execute the interrupt service routine (ISR).

- To which address should it change the execution flow?
- Usually it is determined by an entry on a "**interrupt address table**".
- The interrupt address table contains the address of multiple ISRs. E.g. One for each kind of interrupt
- Example: ARM interrupt address table

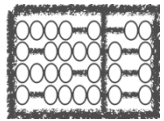




Interrupt-driven I/O

Chapter 6.4 of Vahid's book contains more information and examples of interrupt-driven I/O.

- Frank Vahid e Tony Givargis. Embedded System Design: A Unified Hardware/ Software Introduction. Wiley, 2002.



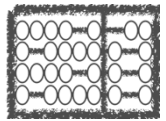
Agenda

I/O Example and Pooling

Interrupt-driven I/O

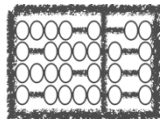
Interrupt Handling on ATMega328 and Arduino

Lab Activities



Interrupt Handling on ATMega328

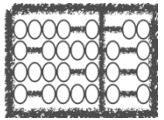
- AVR ATMega328 contains multiple peripheral that may generate interrupts.
- On a interrupt, the AVR jumps to the correct ISR using a interrupt address table.
  - Each table entry contains one instruction, which is expected to be a jump to the ISR.



Interrupt Handling on ATMega328

ATMega328 Interrupt Address Table

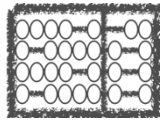
Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Coutner2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Coutner0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART Data Register Empty
21	0x0028	USART_TX	USART Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface (I2C)
26	0x0032	SPM READY	Store Program Memory Ready



Interrupt Handling on ATMega328

The interrupt address table is stored on memory and should be initialized with jumps to the correct ISRs before enabling interrupts.

```
Address   Labels    Code                                  Comments
0x0000              jmp       RESET                       ; Reset
0x0002              jmp       INT0                        ; IRQ0
0x0004              jmp       INT1                        ; IRQ1
0x0006              jmp       PCINT0                      ; PCINT0
0x0008              jmp       PCINT1                      ; PCINT1
0x000A              jmp       PCINT2                      ; PCINT2
0x000C              jmp       WDT                         ; Watchdog Timeout
0x000E              jmp       TIM2_COMPA                  ; Timer2 CompareA
0x0010              jmp       TIM2_COMPB                  ; Timer2 CompareB
0x0012              jmp       TIM2_OVF                    ; Timer2 Overflow
0x0014              jmp       TIM1_CAPT                   ; Timer1 Capture
0x0016              jmp       TIM1_COMPA                  ; Timer1 CompareA
0x0018              jmp       TIM1_COMPB                  ; Timer1 CompareB
0x001A              jmp       TIM1_OVF                    ; Timer1 Overflow
0x001C              jmp       TIM0_COMPA                  ; Timer0 CompareA
0x001E              jmp       TIM0_COMPB                  ; Timer0 CompareB
0x0020              jmp       TIM0_OVF                    ; Timer0 Overflow
```



Interrupt Handling on ATMega328

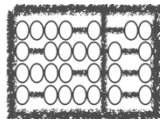
By default interrupts are disabled. After initializing the interrupt address table the program must enable them.

SREG register contains a bit (I) that enables/disables interrupts.

- Bit 7: Global Interrupt Enable
  - 0 => Interrupts disabled
  - 1=> Interrupts enabled

Name: SREG
Offset: 0x5F
Reset: 0x00

Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

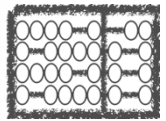


Interrupt Handling on ATMega328

The interrupt address table is stored on memory and shoul
jumps to the correct ISRs before enabling interrupts.

ISR addresses to handle “External Interrupts”

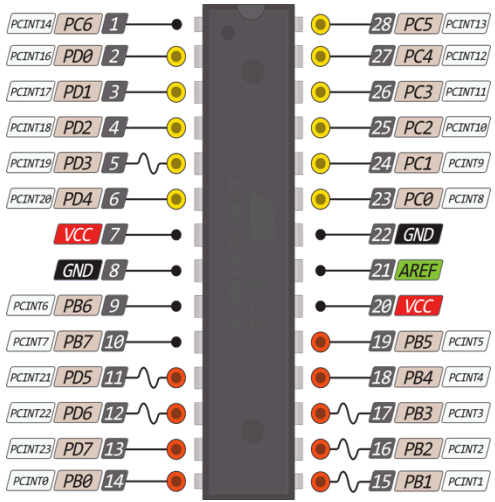
```
Address   Labels   Code                          Comments
0x0000             jmp     RESET                 ; Reset
0x0002             jmp     INT0                  ; IRQ0
0x0004             jmp     INT1                  ; IRQ1
0x0006             jmp     PCINT0                ; PCINT0
0x0008             jmp     PCINT1                ; PCINT1
0x000A             jmp     PCINT2                ; PCINT2
0x000C             jmp     WDT                   ; Watchdog Timeout
0x000E             jmp     TIM2_COMPA            ; Timer2 CompareA
0x0010             jmp     TIM2_COMPB            ; Timer2 CompareB
0x0012             jmp     TIM2_OVF              ; Timer2 Overflow
0x0014             jmp     TIM1_CAPT             ; Timer1 Capture
0x0016             jmp     TIM1_COMPA            ; Timer1 CompareA
0x0018             jmp     TIM1_COMPB            ; Timer1 CompareB
0x001A             jmp     TIM1_OVF              ; Timer1 Overflow
0x001C             jmp     TIM0_COMPA            ; Timer0 CompareA
0x001E             jmp     TIM0_COMPB            ; Timer0 CompareB
0x0020             jmp     TIM0_OVF              ; Timer0 Overflow
```

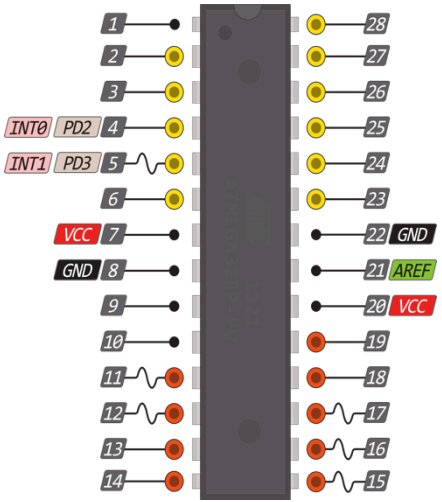


Interrupt Handling on ATMega328 – External Interrupts

External Interrupts are interrupts triggered by “external” pins.

They may be triggered by PCINT or INT pins.





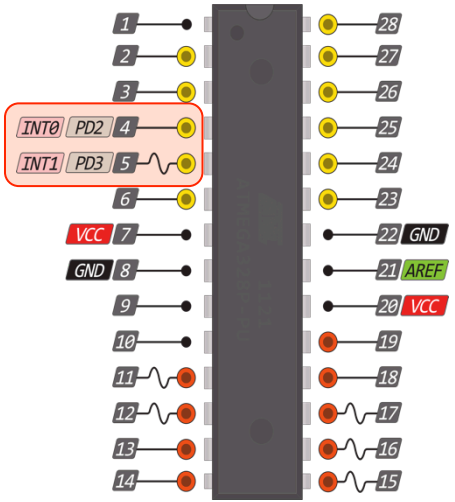
Interrupt Handling on ATMega328 – External Interrupts

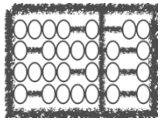
Interrupts generated by INT pins - PD2 and PD3

System can be configured to trigger interrupts on:

- low level: input = LOW;
- falling edge: input changes from HIGH to LOW;
- rising edge: input changes from LOW to HIGH;
- any change.

External Interrupt Control Register A (EICRA) configures the interrupts of INT pins.



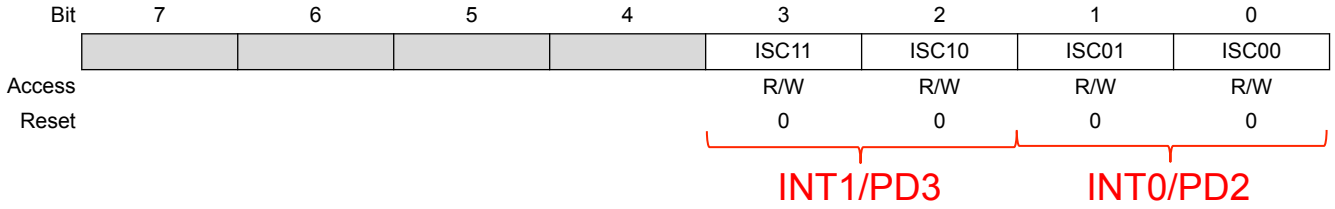


Interrupt Handling on ATMega328 – External Interrupts

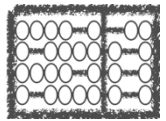
Interrupts generated by INT pins - PD2 and PD3

External Interrupt Control Register A (EICRA) configures the interrupts of INT pins.

Name: EICRA
Offset: 0x69
Reset: 0x00



Value	Description
00	The low level of INT1 generates an interrupt request.
01	Any logical change on INT1 generates an interrupt request.
10	The falling edge of INT1 generates an interrupt request.
11	The rising edge of INT1 generates an interrupt request.



Interrupt Handling on ATMega328 – External Interrupts

Interrupts generated by INT pins - PD2 and PD3

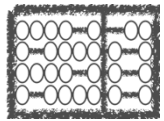
External Interrupt Mask Register (EIMSK) masks interrupts of INT pins.

- 0 => mask interrupts
- 1 => allow interrupts

Interrupts are only handled by the processor if both the I-bit in the SREG register and the INT bit in the EIMSK register are enabled.

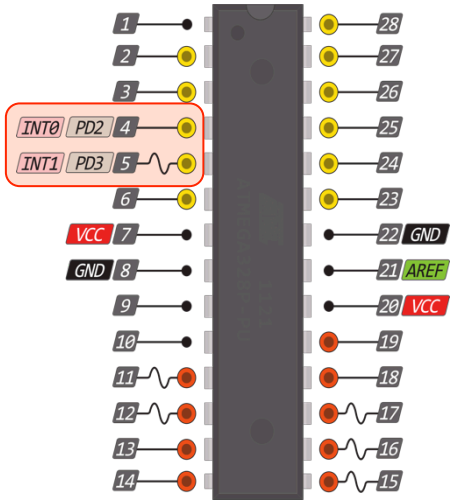
Name: EIMSK
Offset: 0x3D
Reset: 0x00

Bit	7	6	5	4	3	2	1	0
							INT1	INT0
Access							R/W	R/W
Reset							0	0

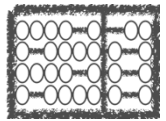


Interrupt Handling on ATMega328 – External Interrupts

```
#include <avr/io.h>
#include <avr/interrupt.h>
int main(void){
  DDRD  &= ~(1 << DDD2);     // PD2 (PCINT0 pin) is now an input
  PORTD |=  (1 << PORTD2);   // PD2 is now with pull-up enabled
  EICRA &= ~(1 << ISC00);    // set INT0 to trigger on
  EICRA |=  (1 << ISC01);    // FALLING edge (ISC0 = 01)
  EIMSK |=  (1 << INT0);     // Allow INT0 interrupts
  sei();                     // Enable Global Interrupts
  while(1) {
      /*main program loops here */
  }
}
/* Interrupt Service Routine */
ISR (INT0_vect){
}
```



Code for AVR



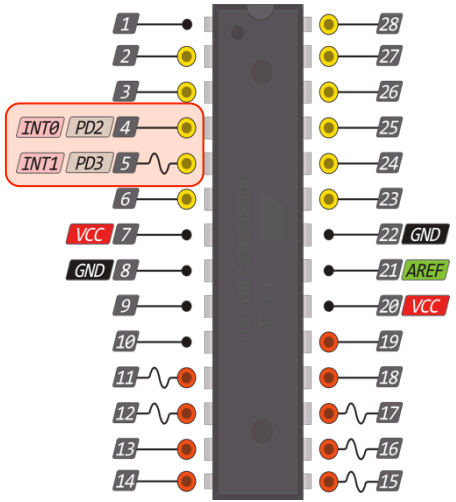
Interrupt Handling on Arduino – External Interrupts

```
const byte ledPin = 13;         // LED pin
const byte interruptPin = 2;    // Interrupt pin = PD2
volatile byte state = LOW;      // Current LED state

void setup() {
  pinMode(ledPin, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, FALLING);
}

// Main program loop
void loop() {
  digitalWrite(ledPin, state);
}

// Interrupt Service Routine
void blink() {
  state = !state;
}
```



Code for Arduino

ATMega328 Pin Change Interrupts

ATMega328 can handle interrupts from any digital pin, not only PD2 and PD3.

Programmer must configure:

- Pin Change Interrupt Control Register (PCICR)
- Pin Change Mask Register 1/2/3 (PCMSKn)

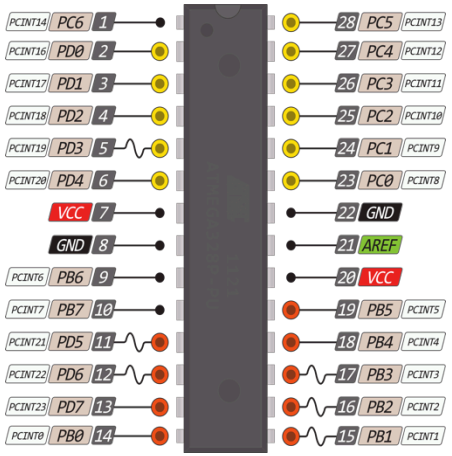
Interrupt address table register one ISR for each set of pins.

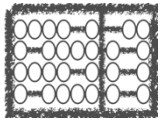
- Each set represents 7 or 8 pins.

User ISR must find out which pin generated the interrupt.

See ATMega328P datasheet Chapter 17 for more information.

http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_datasheet.pdf





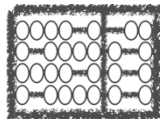
Agenda

I/O Example and Pooling

Interrupt-driven I/O

Interrupt Handling on ATMega328 and Arduino

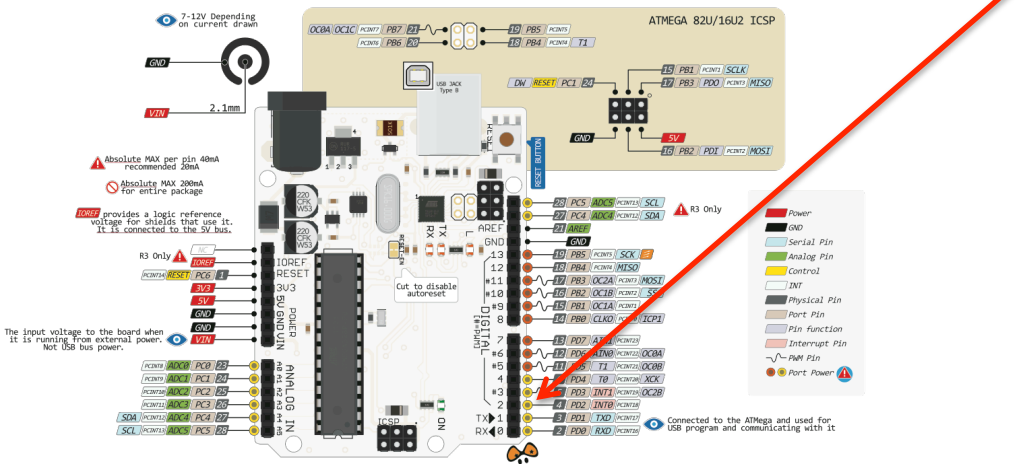
Lab Activities



Lab 5: Trigger the LED when the push button is released

Write an Arduino Sketch that turns the on-board LED (pin 13) ON/OFF when the push button is released!

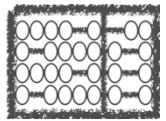
Hint: You must connect the push button to pin 2.



Lab 6: Trigger the LED when the push button is pressed

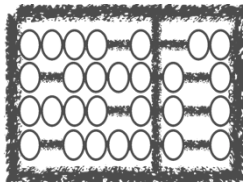
Change your Arduino Sketch to:

- turn the LED when the push button is pressed instead of pushed.



INF-741: Embedded systems programming for IoT

Prof. Edson Borin



Institute of Computing

University of Campinas