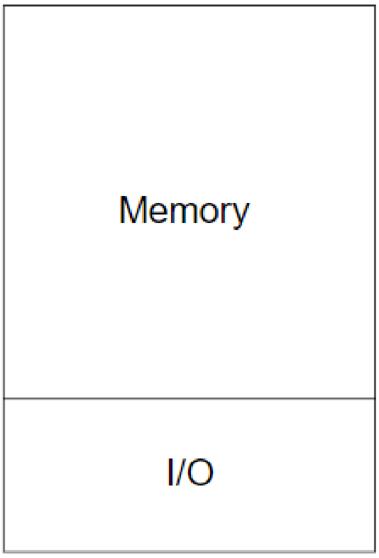
Week5

Memory Mapped I/O

- The entire memory address space contains a section for I/O registers.



AVR Memory Mapped I/O

- In AVR, 64+ I/O registers are mapped into memory space \$0020 ~ \$01FF
  - with 2-byte address
- With such memory addresses, the access to the I/O's uses memory-access type of instructions
  - E.g. *st* and *ld*

Memory Mapped I/O (cont.)

- Advantages:
  - Simple CPU design
  - No special instructions for I/O accesses
  - Scalable
- Disadvantages:
  - I/O devices reduce the amount of memory space available for application programs.
  - The address decoder needs to decode the full address bus to avoid conflict with memory addresses.

Separate I/O

- Two separate spaces for memory and I/O.
  - Less expensive address decoders than those needed for memory-mapped I/O
- Special I/O instructions are required.

Separate I/O (cont.)

- In AVR, the first 64 I/O registers can be addressed with the separate I/O addresses: $00 ~ $3F
  - 1 byte address
- With such separate addresses, the access to the I/O's uses I/O specific instructions.
  - *IN* and *OUT*

```
.equ    SREG     = 0x3f
.equ    SPL      = 0x3d
.equ    SPH      = 0x3e
.equ    EIND     = 0x3c
.equ    RAMPZ    = 0x3b
.equ    SPMCSR   = 0x37
.equ    MCUCR    = 0x35
.equ    MCUSR    = 0x34
.equ    SMCR     = 0x33
.equ    OCDR     = 0x31
```

I/O Synchronization

- CPU is typically much faster than I/O devices.
- Therefore, synchronization between CPU and I/O devices is required.
- Two synchronization approaches:
  - Software
  - Hardware
    - To be covered later

Software Synchronization

- Two basic methods:
  - Real-time synchronization
    - Uses a software delay to match CPU to the timing requirement of the I/O device.
      - The timing requirement must be known
      - Sensitive to CPU clock frequency
      - Consumes CPU time.
  - Polling I/O
    - A status register, with a DATA_READY bit, is added to the device. The software keeps reading the status register until the DATA_READY bit is set.
      - Not sensitive to CPU clock frequency
      - Still consumes CPU time, but CPU can do other tasks at the same time.

AVR PORTs

- Can be configured to receive or send data
- Include physical pins and related circuitry to enable input/output operations.
- Different AVR microcontroller devices have different port design
  - ATmega2560 has 100 pins, most of them form 11 ports for parallel input/output.
    - Port A to Port G
      - Having separate I/O addresses
        - » using *in* or *out* instructions
    - Port H to Port L
      - Only having memory-mapped addresses
    - Three I/O addresses are allocated for each port. For example, for Port *x*, the related three registers are:
      - PORTx: data register
      - DDRx: data direction register
      - PINx: input pin register

```
.equ    PORTD   = 0x0b
.equ    DDRD    = 0x0a
.equ    PIND    = 0x09
.equ    PORTC   = 0x08
.equ    DDRC    = 0x07
.equ    PINC    = 0x06
.equ    PORTB   = 0x05
.equ    DDRB    = 0x04
.equ    PINB    = 0x03
.equ    PORTA   = 0x02
.equ    DDRA    = 0x01
.equ    PINA    = 0x00
```

Load I/O Data to Register

- Syntax: ***in Rd, A***
- Operands: $0 \le d \le 31$, $0 \le A \le 63$
- Operation: Rd ← I/O(A)
- Words: 1
- Cycles: 1
- Example:

```
in r25, 0x03        ; read port B
```

Store Register Data to I/O Location

- Syntax: ***out A, Rr***
- Operands: $0 \le r \le 31$, $0 \le A \le 63$
- Operation: I/O(A) ← Rr
- Words: 1
- Cycles: 1
- Example:

```
out 0x05, r16       ; write to port B
```

How does it work? (cont.)

- When the pin is configured as an input pin, the pull-up resistor can be activated/deactivated.
- To active pull-up resistor for input pin, PORTxn needs to be written logic one.

```
.include "m2560def.inc"

        clr     r16             ; clear r16
        ser     r17             ; set r17
        out     DDRA, r17       ; set Port A for output operation

        out     PORTA, r16      ; write zeros to Port A
        nop                     ; wait (do nothing)
        out     PORTA, r17      ; write ones to Port A
```

```
.include "m2560def.inc"

        clr     r15
        out     DDRA, r15       ; set Port A for input operation

        in      r25, PINA       ; read Port A
        cpi     r25, 4          ; compare read value with constant
        breq    exit            ; branch if r25=4
        ...
exit:   nop                     ; branch destination (do nothing)
```

Example 1

- Design a simple control system that can control a set of LEDs to display a fixed pattern.

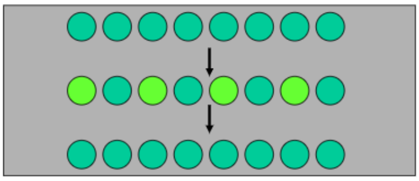
Example 1 (solution)

- The design consists of a number of steps:
  - Set a port for the output operation, one pin of the port is connected to one LED.
  - Write the pattern value to the port so that it drives the LEDs to display the related pattern.

```
.include "m2560def.inc"
        ser r16
        out DDRB, r16           ; set Port B for output

        ldi r16, 0xAA           ; write the pattern
        out PORTB, r16
end:
        rjmp end
```

Example 2

- Design a simple control system that can control a set of LEDs to display a fixed pattern for *one second and then turn the LEDs off.*



- The design consists of a number of steps:
  - Set a port for the output operation, one pin of the port is connected to one LED
  - Write the pattern value to the port so that it drives the display of LEDs
  - *Wait for one second*
  - *Write a pattern to set all LEDs off.*

- The design consists of a number of steps:
  - Set a port for the output operation, one pin of the port is connected to one LED
  - Write the pattern value to the port so that it drives the display of LEDs
  - *Wait for one second*
  - *Write a pattern to set all LEDs off.*

Counting One Second

- Basic idea:
  - Assume the clock cycle period is 1 ms (very very slow, not a real value). Then we can write a program that executes

    $$1/10^{-3} = 1 \times 10^{3}$$

    single cycle instructions.

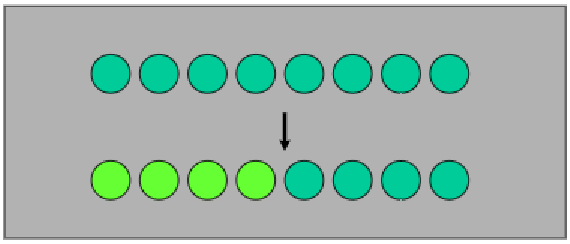
  - Execution of the code will take 1 second if each instruction in the code takes one clock cycle.
- An AVR implementation example is given in the next slide, **where the 1 ms clock cycle time is assumed.**

```
.include "m2560def.inc"
.equ loop_count = 124
.def iH = r25
.def iL = r24
.def countH = r17
.def countL = r16
.macro oneSecondDelay
            ldi countL, low(loop_count)         ; 1 cycle
            ldi countH, high(loop_count)
            clr iH                              ; 1
            clr iL
   loop:    cp iL, countL                       ; 1
            cpc iH, countH
            brsh done                           ; 1, 2 (if branch)
            adiw iH:iL, 1                       ; 2
            nop
            rjmp loop                           ; 2
   done:
.endmacro
```

```
.include "m2560def.inc"

            ser r15
            out DDRB, r15               ; set Port B for output

            ldi r15, 0xAA               ; write the pattern
            out PORTB, r15
            oneSecondDelay              ; 1 second delay
            ldi r15, 0x00
            out PORTB, r15              ; turn off the LEDs
end:
            rjmp end
```

Example 3

- Design a simple control system that can control a set of LEDs to display a fixed pattern *that is specified by the user.*
  - Assume there are switches. Each switch can provide two possible values (switch-on for logic one and switch-off for logic 0)



- Design
  – Connect the switches to the pins of a port
  – Set the port for input
  – Read the input
  – Set another port for the output operation, each pin of the ports is connected to one LED
  – Write the pattern value provided by the input switches to the port so that it drives the display of LEDs

- Execution
  – Set the switches for a desired input value
  – Start the control system

```
.include "m2560def.inc"

          clr r17
          out DDRC, r17            ; set Port C for input
          ser r17
          out PORTC, r17           ; activate the pull up

          in r17, PINC             ; read the pattern set by the user
                                   ; from the switches
          ser r16
          out DDRB, r16            ; set Port B for output

          out PORTB, r17           ; write the input pattern
end:
          rjmp end
```

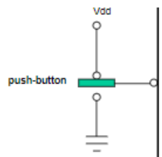
Example 4

- Design a simple control system that can control a set of LEDs to display a pattern specified by the user *during the execution*.

Example 4 (solution)

- Using polling to handle dynamic input
  – The processor continues checking if there is an input for read. If there is, the processor reads the input and goes to next task, otherwise the processor is in a waiting state for the input.

Example 4 (solution)



- Design
  – Set one port for input and connect each pin of the port to one switch
  – Set another port for the output operation, each pin of the ports is connected to one LED
  – Set a pin for input and connect the pin to the push-button,
    - When the button is pressed, it signals “Input Pattern is ready”
  – Poll the pin until “Input Pattern is ready”
  – Read the input pattern
  – Write the pattern to the port so that it drives the display of LEDs

- During execution
  – Set the switches for the input value
  – Push the button
  – The LEDs show the pattern as specified by the user.

Code for Example 4

- Set an extra input bit for signal from user when the input is ready.

```
.include "m2560def.inc"

            cbi DDRD, 7                 ; set Port D bit 7 for input
            clr r17
            out DDRC, r17               ; set Port C for input
            ser r17
            out PORTC, r17              ; activate the pull up
            ;ser r17
            out DDRB, r17               ; set Port B for output

waiting:                                ; check if that bit is clear
            sbic PIND, 7                ; if yes skip the next instruction
            rjmp waiting                ; waiting

            in r17, PINC                ; read pattern set by the user
                                        ; from the switches
            out PORTB, r17
            rjmp waiting
```

Figure 1 is a snapshot from the AVR studio simulation.

```
.include "m2560def.inc"

.macro store
ldi r16, @0
sts @1, r16
.endmacro


store 0xf0, 0x0021
store 0x01, 0x0022
store 0x53, 0x24
store 0x2c, 0x25

lds r30, 0x21
lds r31, 0x22
adiw z, 0x15
out SPL, r30
out SPH, r31
in r29, 0x04
push r29

end: rjmp end
```

Figure 1

(a) What changes will be made to the data memory when the execution comes to the breakpoint? Draw the memory map (addresses and contents) of the affected area and show the changes.

(b) When the execution comes to the end of the program from the breakpoint, what are stored in the address pointer and stack pointer? Are there any further changes to the data memory? If there are any, show the memory location(s) and the related contents.

Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Arithmetic and logic instructions
     - ser
   - Data transfer instructions
     - in, out
   - Bit operations
     - sbi, cbi
   - Program control instructions
     - sbic, sbis
   - MCU control instructions
     - nop

2. Study the following code. What is the function ?

```
.include "m2560def.inc"
.def temp =r16

.equ PATTERN1 = 0x5B
.equ PATTERN2 = 0xAA

          ser temp
          out PORTC, temp           ; Write ones to all the LEDs
          out DDRC, temp            ; PORTC is output
          out PORTA, temp           ; Enable pull-up resistors on PORTA
          clr temp
          out DDRA, temp            ; PORTA is input
switch0:
          sbic PINA, 0              ; Skip the next instruction
                                    ; if switch0 is pushed
          rjmp switch1              ; If not pushed, check the other switch
          ldi temp, PATTERN1        ; Store PATTERN1 to the LEDs
          out PORTC, temp           ; if the switch was pushed
switch1:
          sbic PINA, 1              ; Skip the next instruction
                                    ; if switch 1 is pushed
          rjmp switch0              ; If not pushed, check the other switch
          ldi temp, PATTERN2        ; Store PATTERN2 to the LEDs
          out PORTC, temp           ; if the switch was pushed
          rjmp switch0              ; Now check switch 0 again
```

3. Refer to “Introduction to Lab Board”. Study the lab board. Write the assembly code to display pattern 10110111 on the LED bar through each of the following I/O ports:
   (a) port C
   (b) port F
   (c) port L

Week6

CPU Interaction with I/O

Two typical approaches:

- Polling
  - Software queries I/O devices
  - No extra hardware needed
  - Not efficient
    - It takes processor cycles to query a device even if it does not need any service.
- Interrupt
  - I/O devices generate signals to request the services of CPU
  - Need special hardware to implement interrupt services
  - Efficient
    - A signal is generated only if the I/O device needs services from CPU.

Interrupt System

- An interrupt system implements interrupt services
- It basically performs three tasks:
  - Detecting interrupt event
  - Responding to interrupt
  - Resuming normal programmed task

Detect Interrupt Event

- Interrupt event
  - Associated with interrupt signal
    - In different forms, including signal levels and edges.
  - Can be multiple and simultaneous
    - There may be many sources to generate an interrupt;
    - A number of interrupts can be generated at the same time.
- Approaches are required to
  - Identify an interrupt event among multiple sources
  - Determine which interrupt to serve if there are multiple simultaneous interrupts

Respond to Interrupt

- Handling interrupt
  - Wait for the current instruction to finish.
  - Acknowledge the interrupting device.
  - Branch to the correct ***interrupt service routine*** (interrupt handler) to service the interrupting device.

Resume Normal Task

- Return to the interrupted program at the point it was interrupted.

Multiple Interrupt Masking

- Masking enables some interrupts and disables others
- Individual disable/enable bit is assigned to each interrupting source.

Transferring Control to Interrupt Service Routine

- Hardware needs to save the return address.
  - Most processors save the return address on the stack.
- Hardware may also save some registers such as program status register.
  - AVR does not save any register. It is the programmer's responsibility to save program status register and conflict registers.
- The delay from the time the pending IRQ is generated to the time the Interrupt Service Routine (ISR) starts to execute is called *interrupt latency*.

Interrupt Service Routine

- A section of code to be executed when the corresponding interrupt is responded by CPU.
- Interrupt service routine is a special function, therefore can be constructed with three parts:
  - Prologue:
    - Code mainly for saving conflict registers
  - Body:
    - Code for doing the required task.
  - Epilogue:
    - Code for restoring conflict registers
    - The last instruction is the return-from-interrupt instruction.

Software Interrupt

- Software interrupt is the interrupt generated by software without a hardware-generated-IRQ.
- Software interrupt is typically used to implement system calls in OS.
- Some processors have a special machine instruction to generate software interrupt.
  - SWI in ARM.
- AVR does NOT provide a software interrupt instruction.
  - Programmers can use External Interrupts to implement software interrupts.

- Reset is a special interrupt available in most processors (including AVR).
- It is non-maskable.
- Its service function mainly sets the system to the initial state (hence called reset interrupt).
  - No need to deal with conflict registers.

AVR Interrupts

- Interrupts in AVR basically can be divided into internal and external interrupts
- Each has a dedicated interrupt vector
  - To be discussed
- Hardware is used to detect interrupt
- To enable an interrupt, two control bits must be set
  - the Global Interrupt Enable bit (I bit) in the Status Register, SREG
    - Using sei instruction
  - the enable bit for that interrupt
- To disable all maskable interrupts, reset the I bit in SREG
  - Using cli instruction
- Priority of interrupts is used to handle multiple simultaneous interrupts
  - To be discussed

Interrupt Vectors

- Each interrupt has a 4-byte (2-word) interrupt vector, containing an instruction to be executed after CPU has accepted the interrupt.
- The lowest address space in the program memory is, by default, defined as the section for Interrupt Vectors.
- The priority of an interrupt is based on the position of its vector in the program memory
  - The lower the address, the higher the priority level
- RESET has the highest priority

Interrupt Vectors in Mega2560

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	PCINT1	Pin Change Interrupt Request 1
12	$0016[3]	PCINT2	Pin Change Interrupt Request 2
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	$0030	SPI, STC	SPI Serial Transfer Complete
26	$0032	USART0 RX	USART0 Rx Complete
27	$0034	USART0 UDRE	USART0 Data Register Empty
28	$0036	USART0 TX	USART0 Tx Complete
29	$0038	ANALOG COMP	Analog Comparator

30	$003A	ADC	ADC Conversion Complete
31	$003C	EE READY	EEPROM Ready
32	$003E	TIMER3 CAPT	Timer/Counter3 Capture Event
33	$0040	TIMER3 COMPA	Timer/Counter3 Compare Match A
34	$0042	TIMER3 COMPB	Timer/Counter3 Compare Match B
35	$0044	TIMER3 COMPC	Timer/Counter3 Compare Match C
36	$0046	TIMER3 OVF	Timer/Counter3 Overflow
37	$0048	USART1 RX	USART1 Rx Complete
38	$004A	USART1 UDRE	USART1 Data Register Empty
39	$004C	USART1 TX	USART1 Tx Complete
40	$004E	TWI	2-wire Serial Interface
41	$0050	SPM READY	Store Program Memory Ready
42	$0052[3]	TIMER4 CAPT	Timer/Counter4 Capture Event
43	$0054	TIMER4 COMPA	Timer/Counter4 Compare Match A
44	$0056	TIMER4 COMPB	Timer/Counter4 Compare Match B
45	$0058	TIMER4 COMPC	Timer/Counter4 Compare Match C
46	$005A	TIMER4 OVF	Timer/Counter4 Overflow
47	$005C[3]	TIMER5 CAPT	Timer/Counter5 Capture Event
48	$005E	TIMER5 COMPA	Timer/Counter5 Compare Match A
49	$0060	TIMER5 COMPB	Timer/Counter5 Compare Match B
50	$0062	TIMER5 COMPC	Timer/Counter5 Compare Match C
51	$0064	TIMER5 OVF	Timer/Counter5 Overflow
52	$0066[3]	USART2 RX	USART2 Rx Complete
53	$0068[3]	USART2 UDRE	USART2 Data Register Empty
54	$006A[3]	USART2 TX	USART2 Tx Complete
55	$006C[3]	USART3 RX	USART3 Rx Complete
56	$006E[3])	USART3 UDRE	USART3 Data Register Empty
57	$0070[3]	USART3 TX	USART3 Tx Complete

Interrupt Process

- When an interrupt service occurs,
  - the Global Interrupt Enable I-bit is cleared and
  - all interrupts are disabled.
- The user software can set the I-bit to allow nested interrupts
- The I-bit is automatically set
  - when a Return from Interrupt instruction, *reti*, is executed.
- When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.
  - The Reset interrupt is an exception

Initialization of Interrupt Vector Table (IVT) in Mega2560

- Typically an interrupt vector contains
  - a branch instruction (*jmp* or *rjmp*) that branches to the first instruction of the interrupt service routine, or
  - simply *reti* (return-from-interrupt) if you don't need to handle this interrupt.

Example of IVT Initialization in Mega2560

```
.include "m2560def.inc"
.cseg
.org 0x0000
; first vector -----
    rjmp RESET          ; Jump to the start of Reset interrupt service routine
                        ; Relative jump is used if RESET is not far
    nop                 ; to make the vector 4 bytes.
;second vector ----
    jmp IRQ0            ; Long jump is used assuming IRQ0 is very far away
; third vector -----
    reti                ; Return to the break point (No handling for this interrupt).
...
RESET:                  ; The interrupt service routine for RESET starts here.
...
IRQ0:                   ; The interrupt service routine for IRQ0 starts here.
```

Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Bit operations
     - sei, cli
     - sbi, cbi

Week7

External Interrupts

- The external interrupts are triggered through the INT7:0 pins.
  - If enabled, the interrupts can be triggered even if the INT7:0 pins are configured as outputs
    - This feature provides a way of generating a software interrupt.
  - Can be triggered by a falling or rising edge or a logic level
    - Specified in External Interrupt Control Register
      - EICRA (for INT3:0)
      - EICRB (for INT7:4)

External Interrupts (cont.)

- To enable an external interrupt, two bits must be set
  - I bit in SREG
  - INTx bit in EIMSK
- To generate an external interrupt, the following must be met:
  - The interrupt must be enabled
  - The associated external pin must have a designed signal produced.

EIMSK

- External Interrupt Mask Register
  - A bit is set to enable the related interrupt

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
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	

EICRA

- External Interrupt Control Register *A*
  - For INT0-3
  - Defines the type of signal that activates the external interrupt
    - on rising or falling edge or level sensed

Bit	7	6	5	4	3	2	1	0
(0x69)	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00
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

ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request
0	1	Any edge of INTn generates asynchronously an interrupt request
1	0	The falling edge of INTn generates asynchronously an interrupt request
1	1	The rising edge of INTn generates asynchronously an interrupt request

EICRB*

- External Interrupt Control Register B
  - For INT4-7
  - Defines the type of signals that activate the External Interrupt
    - on rising or falling edge or level sensed.

Bit	7	6	5	4	3	2	1	0
(0x6A)	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40
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

Table 15-3. Interrupt Sense Control[(1)]

ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request
0	1	Any logical change on INTn generates an interrupt request
1	0	The falling edge between two samples of INTn generates an interrupt request
1	1	The rising edge between two samples of INTn generates an interrupt request

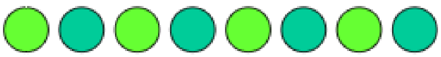
EIFR

- Interrupt flag register
  - A bit in the register is set when an edge-triggered interrupt is enabled and an event on the related INT pin happens.

Bit	7	6	5	4	3	2	1	0
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
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

Example 1

- Design a system, where the state of LEDs toggles under the control of the user, and the number of toggles is counted.



```
.include "m2560def.inc“

.def        temp = r16
.def        output = r17
.def        count = r18                 ; count number of interrupts
.equ        PATTERN = 0b01010101

                                        ; set up interrupt vectors
            jmp RESET
.org        INT0addr                    ; defined in m2560def.inc
            jmp EXT_INT0

RESET:
            ser temp                    ; set Port C as output
            out DDRC, temp
            out PORTC, temp
            ldi output, PATTERN
```

```
            ldi temp, (2 << ISC00) ; set INT0 as falling edge triggered interrupt
            sts EICRA, temp

            in temp, EIMSK                  ; enable INT0
            ori temp, (1<<INT0)
            out EIMSK, temp

            sei                             ; enable Global Interrupt
            jmp main

EXT_INT0:
            push temp                       ; save register
            in temp, SREG                   ; save SREG
            push temp

            com output                      ; flip the pattern
            out PORTC, output
            inc count

            pop temp                        ; restore SREG
            out SREG, temp
            pop temp                        ; restore register
            reti

main:
            clr count
            clr temp
loop:
            inc temp            ; a dummy task in main

            cpi temp, 0x1F      ; the following section in red
            breq reset_temp     ; shows the need to save SREG
            rjmp loop           ; in the interrupt service routine
reset_temp:
            clr temp
            rjmp loop
```

```
.equ   ISC00   = 0   .equ   INT0   = 0
.equ   ISC01   = 1   .equ   INT1   = 1
.equ   ISC10   = 2   .equ   INT2   = 2
.equ   ISC11   = 3   .equ   INT3   = 3
.equ   ISC20   = 4   .equ   INT4   = 4
.equ   ISC21   = 5   .equ   INT5   = 5
.equ   ISC30   = 6   .equ   INT6   = 6
.equ   ISC31   = 7   .equ   INT7   = 7
```

Example 2

- Based on Example 1, implement a software interrupt
  - When there is an overflow in the counter that counts LED toggles, all LEDs are turned on.

Example 2 (solution)

- Use another external interrupt as software interrupt
  - Software generates the external interrupt request
- In the main program, test if there is an overflow
  - If there is an overflow, write a value (based on the interrupt type chosen) to the pin to invoke the interrupt.

```
.include "m2560def.inc"
.include  "my_macros.inc"                 ; macros for oneSecondDelay

.def        temp =r16
.def        output = r17
.def        count = r18
.equ        PATTERN = 0b01010101
.equ        OVERFLOW = 0b11111111

                                          ; set up interrupt vectors
            rjmp RESET
.org        INT0addr
            rjmp EXT_INT0
.org        INT1addr
            jmp EXT_INT1

RESET:
```

```
        ser temp                                ; set Port C as output
        out DDRC, temp
        ldi output, PATTERN
        out PORTC, temp
        ldi temp, 0b00000010
        out DDRD, temp                          ; set Port D bit 1 as output
        out PORTD, temp


        ldi temp, (2 << ISC00) | (2 << ISC10)   ; set INT0 and INT1 as
        sts EICRA, temp                         ; falling edge sensed interrupts

        in temp, EIMSK                          ; enable INT0 and INT1
        ori temp, (1<<INT0) | (1<<INT1)
        out EIMSK, temp

        sei                                     ; enable Global interrupt
        jmp main

EXT_INT0:
        push temp                       ; save register
        in temp, SREG                   ; save SREG
        push temp

        com output                      ; flip the pattern
        out PORTC, output
        inc count

        pop temp                        ; restore SREG
        out SREG, temp
        pop temp                        ; restore register
        reti

EXT_INT1:
        push temp
        in temp, SREG
        push temp

        ldi output, OVERFLOW
        out PORTC, output
        oneSecondDelay                  ; macro for one second delay
                                        ; stored in "my_macro.inc"

        ldi output, PATTERN             ; set pattern for normal LED display
        sbi PORTD, 1                    ; set bit for INT1
        pop temp
        out SREG, temp
        pop temp
        reti
```

```
                        ; main - does nothing but increment a counter
main:
        clr count
        clr temp
loop:
        inc temp
        cpi count, 0xFF
        breq OV                 ; if overflow
        rjmp loop
OV:     cbi PORTD, 1            ; generate an INT1 request
        clr count               ; prepare for the next sw interrupt
        rjmp loop
```

Timer

- A Timer is simply a binary counter
- Can be used to
  - Measure time duration
  - Generate PWM signals
  - Schedule real-time tasks
  - etc.

Timers in AVR

- In AVR, there are 8-bit and 16-bit timers/counters.
  - Timer 0 and Timer 2
    - 8-bit counters
  - Timer 1, 3-5
    - 16-bit counters

8-bit Timer

- The counter can be initialized with
  - 0 (controlled by reset)
  - a number (controlled by *count signal*)
- Can count up or down
  - controlled by *direction signal*
- Those controlled signals are generated by hardware control logic
  - The control logic is further controlled by programmer by
    - Writing control bits into TCCRnA/TCCRnB
- Output
  - Overflow interrupt request bit
  - Output Compare interrupt request bit
  - OCn bit: Output Compare bit for waveform generation

TIMSK0

- Timer/Counter Interrupt Mask Register
  - Set TOIE0 (and I-bit in SREG) to enable the Overflow Interrupt
  - Set OCIE0 (and I bit in SREG) to enable Compare Match Interrupt

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

Control bits for Timer/Counter0

TCCR0A/B

- Timer Counter Control Register

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR0 Bit Description

- COM0xn/WGM0n/FOC0:
  - control the mode of operation
    - the behavior of the Timer/Counter and the output, is defined by the combination of the Waveform Generation mode (WGM02 WGM00) and Compare Output mode (COM0x1:0) bits.
    - The simplest mode of operation is the Normal Mode (WGM02:00 =00). In this mode the counting direction is up. The counter rolls over when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00).
- Refer to Mega2560 Data Sheet (pages 118~194) for details.

TCCR0 Bit Description (cont.)

- Bit 2:0 in TCCR0B
  – Control the clock selection

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

T_{clk}

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Example 3

- Implement a scheduler that can execute a task every one second.

Example 3 (solution)

- Use 8-bit Timer0 to count the time
  - Let's set Timer0 prescaler to /64 (i.e. the system frequency divided by 64)
    - The time-out for the setting should be
      - 256*(clock period) = 256*64/(16 MHz)
        = 1024 us
        - » Namely, we can set the Timer0 overflow interrupt that is to occur every 1024 us.
        - » Note, Clk_{TOS} = 1/16 MHz (obtained from the data sheet)
    - For one second, there are
      - 1000000/1024 =~ 1000 interrupts
- In code,
  - Set Timer0 interrupt to occur every 1024 microseconds
  - Use a counter to count to 1000 interrupts for counting 1 second
  - To observe the 1 second time period, use LEDs that toggles every 1000 interrupts (i.e. one second).

```
; This program implements a timer that counts one second using
; Timer0 interrupt

.include "m2560def.inc"

.equ PATTERN=0b11110000
.def temp=r16
.def leds = r17


; The macro clears a word (2 bytes) in a memory
; the parameter @0 is the memory address for that word
.macro Clear
        ldi YL, low(@0)         ; load the memory address to Y
        ldi YH, high(@0)
        clr temp
        st Y+, temp             ; clear the two bytes at @0 in SRAM
        st Y, temp
 .endmacro
```

```
.dseg
 SecondCounter:
      .byte 2                    ; Two-byte counter for counting seconds.
TempCounter:
      .byte 2                    ; Temporary counter. Used to determine
                                 ; if one second has passed (when TempCounter=1000)
 .cseg
 .org 0x0000
      jmp RESET
      jmp DEFAULT                ; No handling for IRQ0.
      jmp DEFAULT                ; No handling for IRQ1.
      ...
.org OVF0addr
      jmp Timer0OVF              ; Jump to the interrupt handler for Timer0 overflow.
      ...
      jmp DEFAULT                ; default service for all other interrupts.
DEFAULT:  reti                   ; no service
RESET:

          ser temp                          ; set Port C as output
          out DDRC, temp

          rjmp main

Timer0OVF:                 ; interrupt subroutine for Timer0
         in temp, SREG
         push temp         ; Prologue starts.
          push Yh          ; Save all conflict registers in the prologue.
          push YL
          push r25
          push r24         ; Prologue ends.
          ldi YL, low(TempCounter)    ; Load the address of  the temporary
          ldi YH, high(TempCounter)   ; counter.
          ld r24, Y+       ; Load the value of the temporary counter.
          ld r25, Y
          adiw r25:r24, 1  ; Increase the temporary counter by one.
```


```
Timer0OVF:                 ; interrupt subroutine for Timer0
         in temp, SREG
         push temp         ; Prologue starts.
          push Yh          ; Save all conflict registers in the prologue.
          push YL
          push r25
          push r24         ; Prologue ends.
          ldi YL, low(TempCounter)    ; Load the address of  the temporary
          ldi YH, high(TempCounter)   ; counter.
          ld r24, Y+       ; Load the value of the temporary counter.
          ld r25, Y
          adiw r25:r24, 1  ; Increase the temporary counter by one.
```

```
        st Z, r25           ; Store the value of the second counter.
        st -Z, r24
        rjmp EndIF
NotSecond:
        st Y, r25           ; Store the value of the temporary counter.
        st -Y, r24
EndIF:
        pop r24             ; Epilogue starts;
        pop r25             ; Restore all conflict registers from the stack.
        pop YL
        pop YH
        pop temp
        out SREG, temp
        reti                ; Return from the interrupt.

main:
        ldi leds, 0xff              ; Init pattern displayed
        out PORTC, leds
        ldi leds, PATTERN
        Clear TempCounter           ; Initialize the temporary counter to 0
        Clear SecondCounter         ; Initialize the second counter to 0
        ldi temp, 0b00000000
        out TCCR0A, temp
        ldi temp, 0b00000011
        out TCCR0B, temp            ; Prescaling value=64
        ldi temp,  1<<TOIE0         ; =1024 microseconds
        sts TIMSK0, temp            ; T/C0 interrupt enable
        sei                         ; Enable global interrupt
  loop: rjmp loop                   ; loop forever
```

Exercises

1. In AVR, what are the following registers used for?

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
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	

2. To enable an external interrupt, what should be done?

Exercises

3. How can an interrupt event be represented?

4. Why do we need to set values of the following register for an external interrupt?

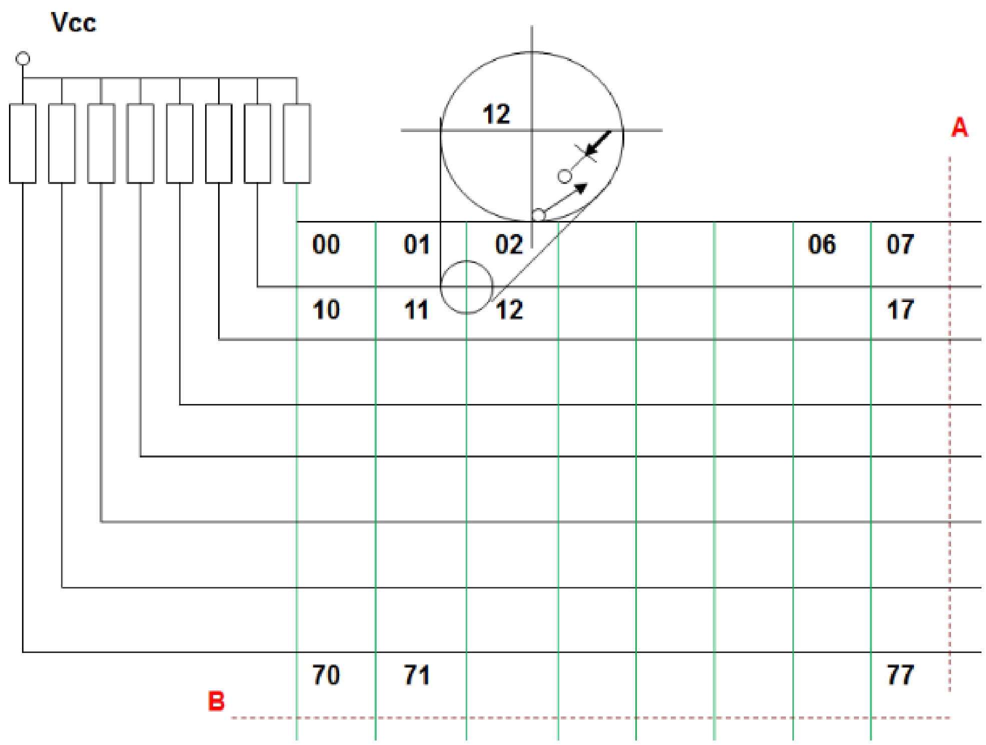
Bit	7	6	5	4	3	2	1	0
(0x69)	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Week8

Input Switch

- A switch provides different values, depending on the switch position.
- Pull-up resistor/circuit may be necessary for the switch to provide a high logic level when the switch is open.
- Problem with switch:
  - Switch bounce
    - When a switch makes contact, its mechanical springiness will cause the contact to bounce, namely contact and break, for a few milliseconds (typically 5 to 10 ms).

Keybpad



Keypad (cont.)

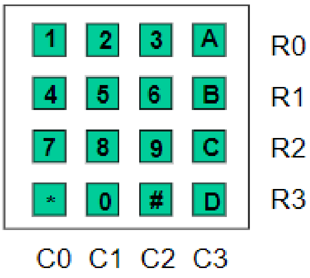
- A keypad is an array of switches arranged in a two-dimensional matrix, consisting of two layers
  - A layer of the horizontal lines
    - connected to the power supply via resistors
  - A layer of the vertical lines
    - normally disconnected to the horizontal layer
- Each intersection of the vertical and horizontal lines forms a switch
  - The switch can be operated by a key button
  - When the key is pressed, the switch connects both two lines.

Keypad (cont.)

- The 8*8 keypad can be interfaced directly to 8-bit output and input ports
  - at point *A (as input)* and *B (as output)*
- The output from each horizontal line
  - Normally is a logic high (1)
  - Becomes logic low (0) when a key is pressed and the related vertical line is set/connected to logic low (0)
- The diode prevents a problem called ghosting.

Example

- Get an input from 4x4 keypad used in our lab board



Example (solution)

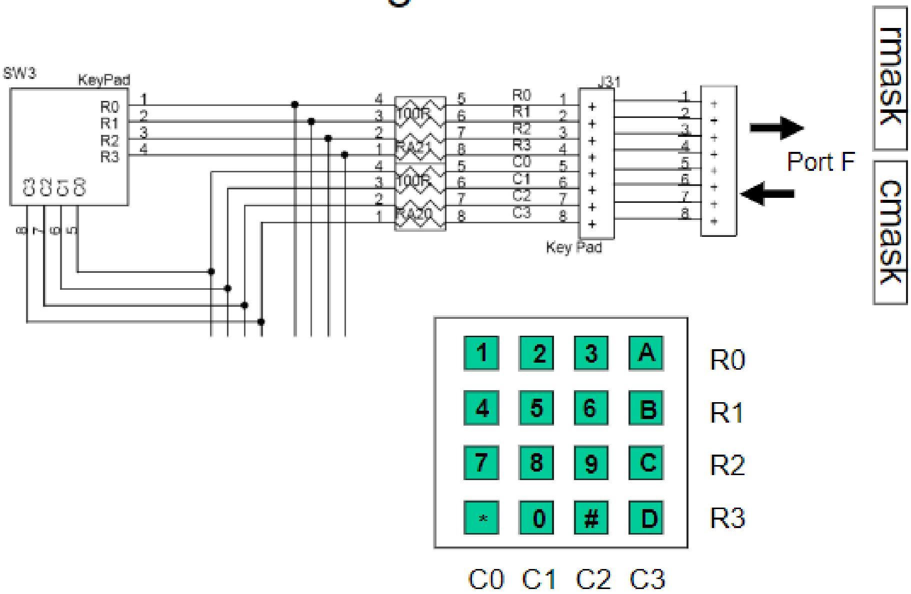
- Algorithm

```
Scan columns from left to right
          for each column, scan rows from top to bottom
                    for each key being scanned
                              if it is pressed
                                        display
                                        wait
                              endif
                    endfor
          endfor
Repeat the scan process
```

  - To select a column, set the related Cx value to 0
  - A mask is used to read one row at a time.

Example (solution)

- Hardware Interfacing



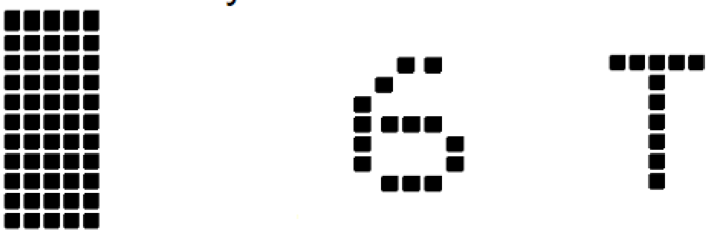

```
; The program gets input from keypad and displays its ascii value on the
; LED bar

.include "m2560def.inc"

.def  row = r16                 ; current row number
.def  col = r17                 ; current column number
.def  rmask = r18               ; mask for current row during scan
.def  cmask = r19               ; mask for current column during scan
.def  temp1 = r20
.def  temp2 = r21

.equ PORTFDIR = 0xF0            ; PF7-4: output, PF3-0, input
.equ ROWMASK  =0x0F             ; for obtaining input from Port F
.equ INITCOLMASK = 0xEF         ; scan from the leftmost column,
.equ INITROWMASK = 0x01         ; scan from the top row
```

```
RESET:

        ldi     temp1, PORTFDIR         ; PF7:4/PF3:0, out/in
        out     DDRF, temp1
        ser     temp1                   ; PORTC is output
        out     DDRC, temp1
        out     PORTC, temp1

main:
        ldi     cmask, INITCOLMASK      ; initial column mask
        clr     col                     ; initial column

colloop:
        cpi     col, 4
        breq    main                    ; if all keys are scanned, repeat.
        out     PORTF, cmask            ; otherwise, scan a column

        ldi     temp1, 0xFF             ; slow down the scan operation.
delay:  dec     temp1
        brne    delay

        in      temp1, PINF             ; read PORTF
        andi    temp1, ROWMASK          ; get the keypad output value
        cpi     temp1, 0xF              ; check if any row is low
        breq    nextcol
                                        ; if yes, find which row is low
        ldi     rmask, INITROWMASK      ; initialize for row check
        clr     row                     ;

rowloop:
        cpi     row, 4
        breq    nextcol                 ; the row scan is over.
        mov     temp2, temp1
        and     temp2, rmask            ; check un-masked bit
        breq    convert                 ; if bit is clear, the key is pressed
        inc     row                     ; else move to the next row
        lsl     rmask
        jmp     rowloop

nextcol:                                ; if row scan is over
        lsl cmask
        inc col                         ; increase column value
        jmp colloop                     ; go to the next column
```

```
convert:
        cpi     col, 3              ; If the pressed key is in col. 3
        breq    letters             ; we have a letter

                                    ; If the key is not in col. 3 and

        cpi     row, 3              ; if the key is in row3,
        breq    symbols             ; we have a symbol or 0

        mov     temp1, row          ; Otherwise we have a number in 1-9
        lsl     temp1
        add     temp1, row          ;
        add     temp1, col          ; temp1 = row*3 + col
        subi    temp1, -'1'         ; Add the value of character '1'
        jmp     convert_end
letters:
        ldi temp1, 'A'
        add temp1, row              ; Get the ASCII value for the key
        jmp convert_end

symbols:
        cpi col, 0                  ; Check if we have a star
        breq star
        cpi col, 1                  ; or if we have zero
        breq zero
        ldi temp1, '#'              ; if not we have hash
        jmp convert_end
star:
        ldi temp1, '*'              ; Set to star
        jmp convert_end
zero:
        ldi temp1, '0'              ; Set to zero

convert_end:
        out PORTC, temp1            ; Write value to PORTC
        jmp main                    ; Restart main loop
```

Dot Matrix LCD

- Characters are displayed using a dot matrix.
  - 5x7, 5x8, and 5x11
- A controller is used for communication between the LCD and other components, e.g. microprocessor unit (MPU)
- The controller has an internal character generator ROM. All display functions are controllable by instructions.



24

Pin Descriptions

Signal name	No. of Lines	Input/Output	Connected to	Function
DB4 ~ DB7	4	Input/Output	MPU	4 lines of high order data bus. Bi-directional transfer of data between MPU and module is done through these lines. Also DB_7 can be used as a busy flag. These lines are used as data in 4 bit operation.
DB0 ~ DB3	4	Input/Output	MPU	4 lines of low order data bus. Bi-directional transfer of data between MPU and module is done through these lines. In 4 bit operation, these are not used and should be grounded.
E	1	Input	MPU	Enable - Operation start signal for data read/write.
R/W	1	Input	MPU	Signal to select Read or Write "0": Write "1": Read
RS	1	Input	MPU	Register Select "0": Instruction register (Write) : Busy flag; Address counter (Read) "1": Data register (Write, Read)
Vee	1		Power Supply	Terminal for LCD drive power source.
Vcc	1		Power Supply	+5V
Vss	1		Power Supply	0V (GND)

Operations

- MPU communicates with LCD through two registers
  - Instruction Register (IR)
    - To store
      - instruction code
        - » E.g Display Clear or Cursor Shift
      - address for the Display Data RAM (DD RAM)
      - etc.
  - Data Register (DR)
    - To store
      - data to be read/written to/from the DD RAM of the display controller.

Operations (cont.)

- The register select (RS) signal determines which of these two registers is selected.

RS	R/W	Operation
0	0	IR write, internal operation (Display Clear etc.)
0	1	Busy flag (DB_7) and Address Counter ($DB_0 \sim DB_6$) read
1	0	DR Write, Internal Operation (DR ~ DD RAM or CG RAM)
1	1	DR Read, Internal Operation (DD RAM or CG RAM)

Operations (cont.)

- When the busy flag is high or “1”, the LCD is busy with the internal operation.
- The next instruction must not be written until the busy flag is low or “0”.
- For details, refer to the LCD USER’S MANUAL.

LCD Instructions

- A list of binary instructions are available for LCD operations
- Some typical ones are explained in the next slides.

Instructions

- Function Set

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	1	DL	N	F	x	x

  - Set the interface data length, the number of lines, and character font.
    - DL = "1", 8 –bits; otherwise 4 bits
    - N: Sets the number of lines
      - N = "0" : 1 line display
      - N = "1" : 2 line display
    - F: Sets character font.
      - F = "1" : 5 x 10 dots
      - F = "0" : 5 x 7 dots

Instructions

- Entry Mode Set

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	1	I/D	S

  - Set the Increment/Decrement and Shift modes
    - I/D = 1: increments the address counter by 1 for each DD RAM access (read or write); I/D = 0: decrements the address counter
    - S=0, no shift
    - S=1, shift the entire display
      - Shift to the left when I/D = 1
      - Shift to the right when I/D = 0

Instructions

- Display ON/OFF Control

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	1	D	C	B

  – Control the display ON/OFF, Cursor ON/OFF and Cursor Blink function.
    - D: The display is ON when D = 1 and OFF when D = 0.
    - C: The cursor displays when C = 1 and does not display when C = 0.
    - B: The character indicated by the cursor blinks when B = 1.

Instructions

- Clear Display

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	0	1

  – The display clears and the cursor moves to the upper left corner of the display.

Instructions

- Read Busy Flag and Address

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	1	BF	A	A	A	A	A	A	A

  – Read the busy flag (BF) and value of the address counter (AC). BF = 1 indicates that an internal operation is in progress and the next instruction will not be accepted until BF is set to “0”. If the display is written while BF = 1, abnormal operation will occur.

Instructions

- Return Home

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	1	x

  – The cursor moves to the upper left corner of the display. Text on the display remains unchanged.

Instructions

- Write Data to DD RAM

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	1	0	D	D	D	D	D	D	D	D

  – Write binary 8-bit data DDDDDDDD to the CG or DD RAM.
  – The previous designation determines whether the CG or DD RAM is to be written (CG RAM address set or DD RAM address set). After a write the entry mode will automatically increase or decrease the address by 1. Display shift will also follow the entry mode.

Instructions

- Set DD RAM Address

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	1	A	A	A	A	A	A	A

  – Sets the address counter to DD RAM.
  – The address range:
    - For 1-line display, 0x00-0x4F
    - For 2-line display,
      – 0x00-0x27 for the first line
      – 0x40-0x67 for the second line

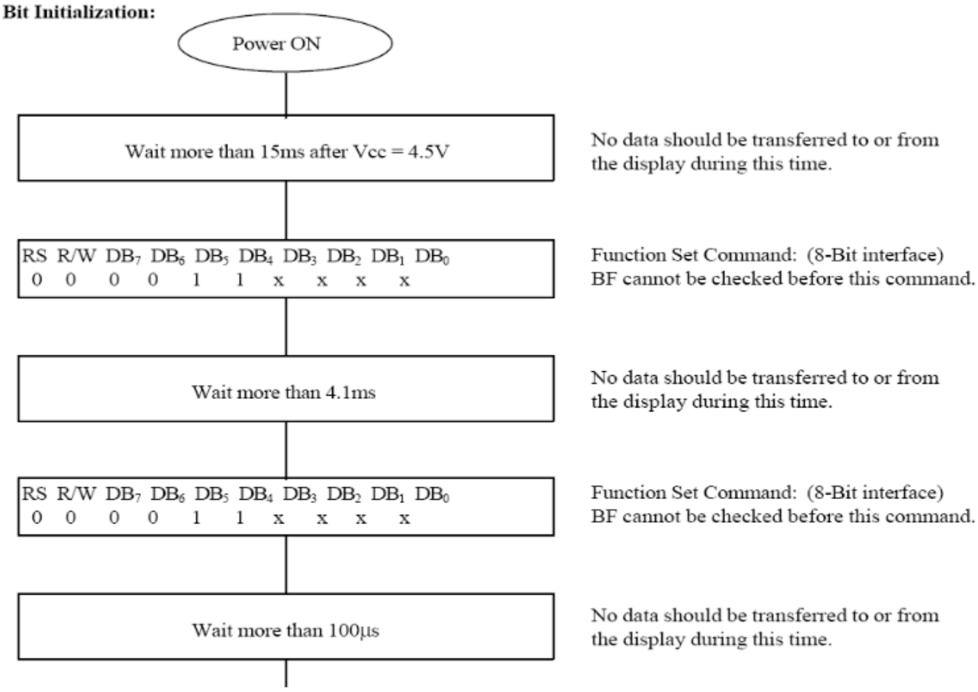
; General purpose register *data* stores value to be written to the LCD
; Port F is output and connects to LCD; Port A controls the LCD (Bit LCD_RS for RS and bit LCD_RW for RW, LCD_E for E). The character to be displayed is stored in register *data*
; Assume all labels are pre-defined.

```
.macro lcd_write_com
        out PORTF, data                 ; set the data port's value up
        ldi temp, (0<<LCD_RS)|(0<<LCD_RW)
        out PORTA, temp                 ; RS = 0, RW = 0 for a command write
        nop                             ; delay to meet timing (Set up time)
        sbi PORTA, LCD_E                ; turn on the enable pin
        nop                             ; delay to meet timing (Enable pulse width)
        nop
        nop
        cbi PORTA, LCD_E                ; turn off the enable pin
        nop                             ; delay to meet timing (Enable cycle time)
        nop
        nop
.endmacro

.macro lcd_write_data
        out PORTF, data                 ; set the data port's value up
        ldi temp, (1 << LCD_RS)|(0<<LCD_RW)
        out PORTA, temp                 ; RS = 1, RW = 0 for a data write
        nop                             ; delay to meet timing (Set up time)
        sbi PORTA, LCD_E                ; turn on the enable pin
        nop                             ; delay to meet timing (Enable pulse width)
        nop
        nop
        cbi PORTA, LCD_E                ; turn off the enable pin
        nop                             ; delay to meet timing (Enable cycle time)
        nop
        nop
.endmacro
```

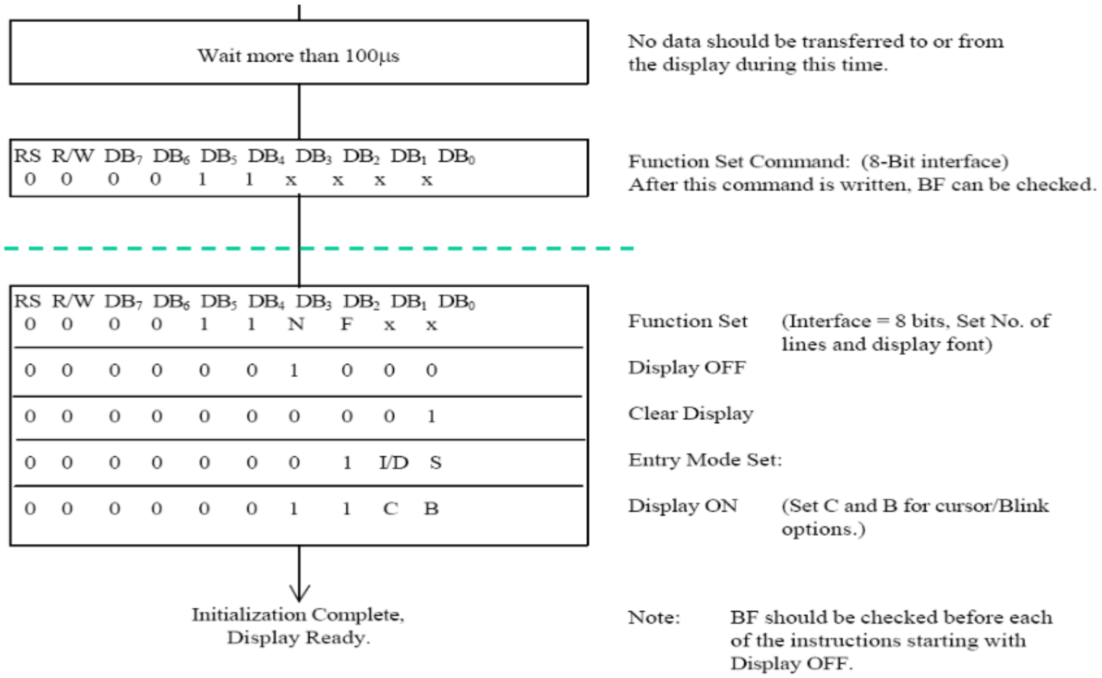
```
.macro lcd_wait_busy
            clr temp
            out DDRF, temp              ; Make PORTF be an input port for now
            out PORTF, temp
            ldi temp, 1 << LCD_RW
            out PORTA, temp             ; RS = 0, RW = 1 for a command port read
busy_loop:
            nop                         ; delay to meet set-up time
            sbi PORTA, LCD_E            ; turn on the enable pin
            nop                         ; delay to meet timing (Data delay time)
            nop
            nop
            in temp, PINF               ; read value from LCD
            cbi PORTA, LCD_E            ; turn off the enable pin
            sbrc temp, LCD_BF           ; if the busy flag is set
            rjmp busy_loop              ; repeat command read
            clr temp                    ; else
            out PORTA, temp             ; turn off read mode,
            ser temp                    ;
            out DDRF, temp              ; make PORTF an output port again
.endmacro
```

Software Initialization

8 - Bit Initialization:



Software Initialization



Homework

1. Write an assembly program to initialize LCD panel to display characters in one line with the 5x7 font.

```
; LCD init
rcall sleep_15ms
do_lcd_command 0b00111000 ; 2x5x7
rcall sleep_5ms
do_lcd_command 0b00111000 ; 2x5x7
rcall sleep_1ms
do_lcd_command 0b00111000 ; 2x5x7
do_lcd_command 0b00111000 ; 2x5x7
do_lcd_command 0b00001000 ; display off
do_lcd_command 0b00000001 ; clear display
do_lcd_command 0b00000110 ; increment, no display shift
do_lcd_command 0b00001110 ; Cursor on, bar, no blink
```