EE2016 Microprocessor Lab & Theory Jul - Nov 2024

EE Department, IIT, Madras.

Experiment 4: Interrupts in Atmel AVR Atmega through Assembly Programming

Contents

1 Aim

Using Atmel AVR assembly language programming, implement interrupts and DIP switches control in Atmel Atmega microprocessor. Aims of this experiment are:

(i) Generate an external (logical) hardware interrupt using an emulation of a push button switch.

(ii) Write an ISR (Interrupt Service Routine) to switch ON an LED for a few seconds (10 secs) and then switch OFF. (The lighting of the LED could be verified by monitoring the signal to switch it ON).

(iii) If there is time, you could try this also: Use the 16 bit timer (via interrupt) to make an LED blink with a duration of 1 second.

Also, one needs to implement all of the above, in AVR assembly.

2 Equipments Required

Since this is a hardware based experiment, we need not only the AVR microcontroller, but also the corresponding IDE or SDK which includes a PC with the following softwares including, the Microchip / Atmel studio simulation software. The exhaustive list of equipments, software, components required are:

1. Atmel Atmega8 Microcontroller chip, USBASP programmer,
2. Bread board with hardware components, data / power cables, LED, mini push buttons etc
3. A PC with Microchip/ Studio simulation software loaded and AVR Burn-o-mat software

3 Background Material

In order to interface a peripheral, there are two steps: (i) pick or select or address a particular peripheral out of 'n' peripherals and (ii) after choosing the intended one, the operational procedure by which the peripheral is controlled or read or written into or to execute a command specific to the peripheral.

The interrupt is a solution to the 2nd step [For first step, the solution is addressing and there are two ways: memory mapped I/O or port mapped I/O. We saw in the class MMIO is the one which very common in microcontrollers, RISC processors]. The other solutions to the 2nd step are polling & DMA (Ofcourse DMA largely deals with memory interfacing only).

In 'interrupt', scheme, the peripheral initiates an 'interrupt' signal to the CPU, which forces the PC to point to a specific pre-designated address in program memory (after the CPU completing the current instruction). The list of such 'fixed' pre-designated addresses (corresponding to multiple interrupts each with a priority level) from where the *Interrupt Service Routine (ISR)* would start is called interrupt vector table. ISR is the program sought by the peripheral device, the last statement is RET or RETI, which returns to the mother program. Many memory management functions are to be carried out when the Program Counter (PC) gets into and out of ISR, which are described in the sections below. Also, the assembly instructions which are related to interrupt and hardware architecture related to implementation of interrupt are given below.

3.1 General Microprocessor Interrupt Concept

Upon activation of an interrupt (eg. through a specific external hardware interrupt pin of a microcontroller, by giving an high input), the microprocessor goes through the following steps:

1. **Interrupt Request (IRQ) Signal:** An external device or an internal event triggers an interrupt by sending an interrupt request (IRQ) signal to the microprocessor.
2. **Current Instruction Completion:** The microprocessor completes the execution of the current instruction in its pipeline or instruction queue.
3. **Interrupt Acknowledgment:** The microprocessor acknowledges the interrupt request by sending an acknowledgment signal, which may involve sending an acknowledgment code to the interrupting device.
4. **Interrupt Vector:** The microprocessor identifies the source of the interrupt by determining its interrupt vector. The interrupt vector is essentially an address in memory where the microprocessor can find the interrupt service routine (ISR) specific to the interrupting device or event.
5. **Context Saving:** Before jumping to the ISR, the microprocessor often needs to save the current context. This typically involves saving the values of program counter, registers, and any other relevant state information to the stack or memory. This step ensures that the microprocessor can later resume the interrupted program from where it left off.
6. **Jump to ISR:** The microprocessor performs a jump or branch to the address specified by the interrupt vector, which is the beginning of the ISR.

7. **Execution of ISR:** The ISR code is executed. The ISR typically handles the specific task associated with the interrupt, such as servicing a hardware device, handling an exception, or responding to a software-triggered interrupt.

8. **Context Restoration:** After the ISR completes its task, it may need to restore the saved context. This involves retrieving the saved values of the program counter, registers, and other state information from the stack or memory.

9. **Return from Interrupt (RETI):** The microprocessor executes an instruction (e.g., "return from interrupt" instruction) to return to the interrupted program. This instruction typically pops the saved program counter and registers from the stack or memory and continues execution from where it was interrupted.

10. **Continuation of Execution:** The microprocessor continues executing the (mother) program that was interrupted, with all registers and state restored.

These are the fundamental steps involved when a microprocessor handles an interrupt. The specific details and instructions used may vary depending on the microprocessor architecture and the interrupt handling mechanism implemented in the system.

4 Interrupts Implementation in Atmega8 AVR Family of Processors

4.1 Types of Interrupts in AVR

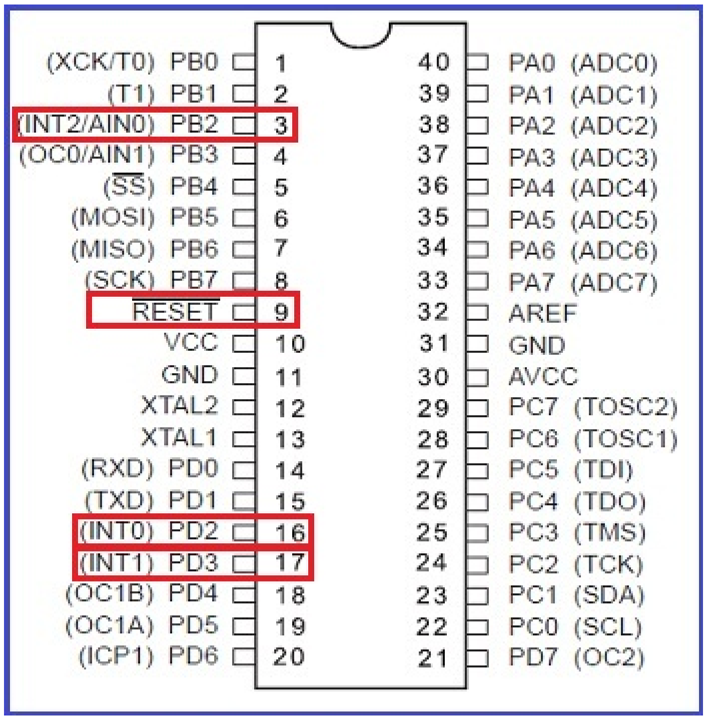
The Atmel ® AVR ® supports several different interrupt sources, depending on the list of peripherals connected to it. The following are some of the widely used sources of interrupts in the AVR

1. Two interrupts for each of the timers, one for overflow and another for compare
2. Three interrupts for external hardware interrupts: Pins PD2 (PORTD.2), PD3 (PORTD.3) and PB2 (PORTB.2) are for the external hardware interrupts INT0, INT1 and INT2 respectively.
3. USART has three interrupts one for receive and two for transmit
4. SPI interrupts
5. The ADC interrupts

In this experiment, we are concerned with the external interrupts only.

4.2 Hardware Details Regarding Interrupts in AVR ATmega32

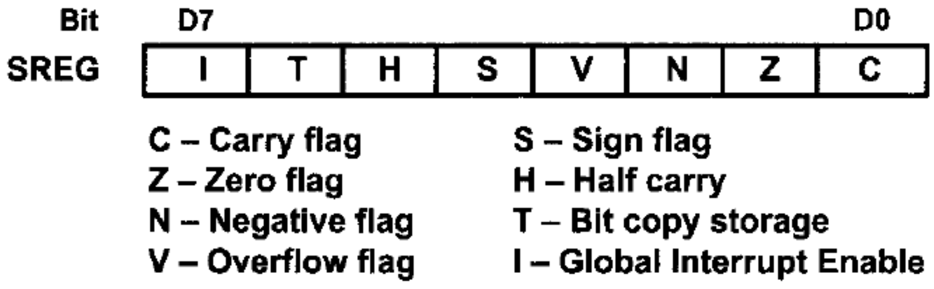
There are three external hardware interrupts in ATmega32: INT0, INT1 and INT2 (corresponding to hardware pins of PD2, PD3 and PB2, respectively.



Special Function Registers (SFRs) which implement the interrupt are SREG, GICR, MCUCR apart from PC (Program Counter), and SP (Stack Pointer). The important bits are I-bit in SREG, bits INT0, INT1 & INT2, in GICR and ISC01 & ISC00 bits in MCUCU.

4.3 Interrupt Initiation and Configuration (Enabling and Disabling Interrupts)

Upon AVR reset, all interrupt enabling bits are reset to 0. Interrupts must be enabled by software, so that microprocessor can respond to them. The D bit of the SREG (Status Register) register is responsible for enabling and disabling the interrupts globally.



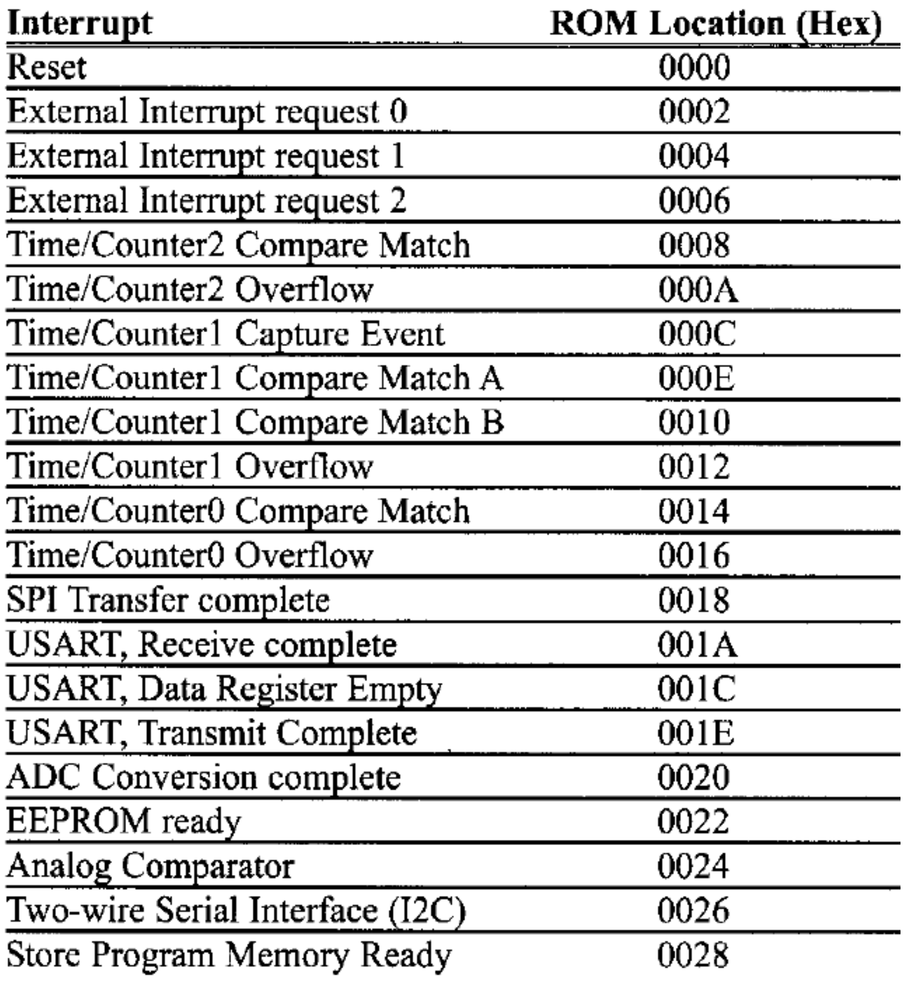
See Fig for SREG, in which if I=0, then it disables all the interrupts in the AVR microcontroller during crisis.

Steps in enabling an interrupt

1. Set bit D7 (I bit) of the SREG to logical 1. Use SEI (Set interrupt) OP-Code
2. Set the individual interrupt to HIGH, say external interrupt PD.2

4.4 Interrupt Vector Table

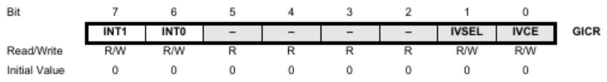
When an interrupt is invoked, the microprocessor goes to a location (implemented through PC) of the first line in the ISR. Generally, in most microprocessors, for every interrupt there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called *interrupt vector table* and is shown in Table below.



Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt request 0	0002
External Interrupt request 1	0004
External Interrupt request 2	0006
Time/Counter2 Compare Match	0008
Time/Counter2 Overflow	000A
Time/Counter1 Capture Event	000C
Time/Counter1 Compare Match A	000E
Time/Counter1 Compare Match B	0010
Time/Counter1 Overflow	0012
Time/Counter0 Compare Match	0014
Time/Counter0 Overflow	0016
SPI Transfer complete	0018
USART, Receive complete	001A
USART, Data Register Empty	001C
USART, Transmit Complete	001E
ADC Conversion complete	0020
EEPROM ready	0022
Analog Comparator	0024
Two-wire Serial Interface (I2C)	0026
Store Program Memory Ready	0028

The interrupt vector table locations 0x0002, 0x0004, 0x0006 are set aside for INT0, INT1 and INT2 respectively. The hardware interrupts must be enabled before they can take effect. This is done using the INTx bit located in General Interrupt Controller Register (GICR). INT0 is a low-level-trigerred interrupt by default.

The General Interrupt Controller Register (GICR)



Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	–	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

For example the following instructions enable INT0

```
LDI R20,0x40
OUT GICR,R20
```

4.5 Memory Management During ISR

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine (allowing 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.

Note that the Status Register is not automatically stored when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software.

When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction. The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

4.6 Interrupt Response Time

The interrupt execution response for all the enabled Atmel ® AVR ® interrupts is four clock cycles minimum. After four clock cycles, the Program Vector address for the actual interrupt handling routine is executed. During this 4-clock cycle period, the Program Counter is pushed onto the Stack. The Vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served. If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode. A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (2 bytes) is popped back from the Stack, the Stack Pointer is incremented by 2, and the I-bit in SREG is set.

4.7 Atmel AVR Instructions Related to Interrupts

The following are the instructions which are closely related to interrupt.

`SEI` **; Set Global Interrupt Flag**

I<—1

Sets the Global Interrupt Flag (I) in SREG. The instruction following SEI will be executed before any pending interrupts. Cycles: 1

`CLI` **; Clear Global Interrupt Flag (I <— 0)**

Clears the Global Interrupt flag (I) in SREG. The interrupts will be immediately disabled. No interrupt will be executed after the CLI instructions, even if it occurs simultaneously with the CLI instruction.

Flags: I <— 0. Cycles: 1

`RET`**; Return from Subroutine, ISR**

Returns from subroutine. The return address is loaded from the stack. The stack pointers uses a pre-incrment scheme during RET. Cycles: 4

`RETI` **; Return from interrupt**

Returns from the interrupt. The return address is loaded from the stack and the I bit bit is set to 1. Note that the status register is not automatically stored when entering an ISR and it is not restored when returning from an ISR. This must be handled by the application prgram. The stck pointer uses a pre-increment scheme during RTI. See example 10-2 in page 371 in Mazidi which illustrates the difference between RET and RETI OP-Codes.

`PUSH Rr` **; Push Register on Stack**

0<=d<=31, STACK<— Rr

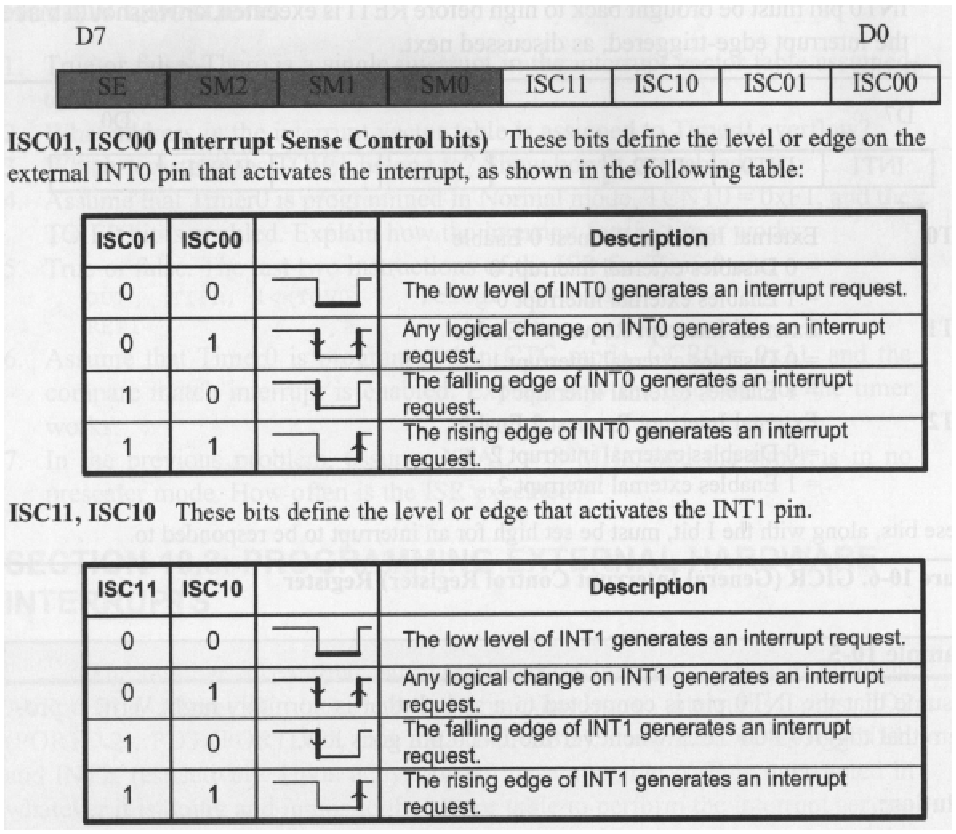
This instruction stores the contents of register Rr on the STACK. The stack pointer is post-decremented by 1 after the PUSH. Cycles: 2

`POP Rd` **; Pop register from stack**

0<=d<=31, Rd<— STACK

4.8 External Interrupts

One of the issues in external interrupts is that one needs to choose the mode of edge triggered or level triggered interrupt. Raising edge triggered (or falling edge triggered) interrupt could be configured by setting the right bits in MCUCU.



D7							D0
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00

ISC01, ISC00 (Interrupt Sense Control bits) These bits define the level or edge on the external INT0 pin that activates the interrupt, as shown in the following table:

ISC01	ISC00		Description
0	0		The low level of INT0 generates an interrupt request.
0	1		Any logical change on INT0 generates an interrupt request.
1	0		The falling edge of INT0 generates an interrupt request.
1	1		The rising edge of INT0 generates an interrupt request.

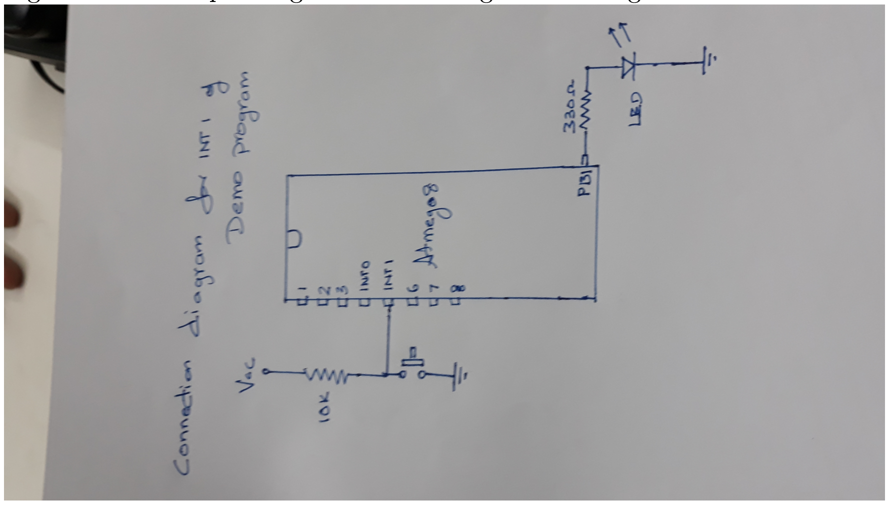
ISC11, ISC10 These bits define the level or edge that activates the INT1 pin.

ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

By setting the right combination of ISC01, ISC00 (Interrupt Sense Control) bits and ISC11 & ISC10, one can choose the rising or falling edge triggered interrupts on the INT0 & INT1 respectively.

4.9 Demo Programs

Interrupt demo program (in assembly) has been uploaded in moodle. There are blanks to be filled in. These are to ensure understanding by the student. You may refer to the manual and then fill them in. There, some lines (in assembly code) are commented for easy understanding. This program implements int1 in AVR Atmega8. The corresponding connection diagram is also given below



5 Your Tasks

You are given the file EE2016F23Exp9int1.asm which is an assembly program which implements interrupt using int1.

1. Fill in the blanks in the assembly code.
2. Use int0 to redo the same in the demo program (duely filled in). Once the switch is pressed the LED should blink 10 times (ON (or OFF) - 1 sec, duty cycle could be 50 %). Demonstrate both the cases.
3. Demonstrate the working using assembly program.

5.1 Procedure

As in previous experiment.

6 Results and Report

Demonstrate it to TAs before you leave the lab.

1. Include the flow chart if any (which is aprt from the handout uploaded by us in moodle). In this experiment you need not.
2. Include the asm code.
3. Give the gist of asm code above (explaining what are the registers involved and to their values).
4. Take the snapshot of the LED blinking using your mob and include it in your report

7 Evaluation Scheme

1. Regular lab session marks given by your caretaker TA.
2. Report (for this experiment) after submission would be evaluated by TAs.

8 References

1. Mazidi, "The AVR microcontroller and Embedded Systems using assembly and C", PHI Chapter 10 [Chapter 4 for I/O registers]