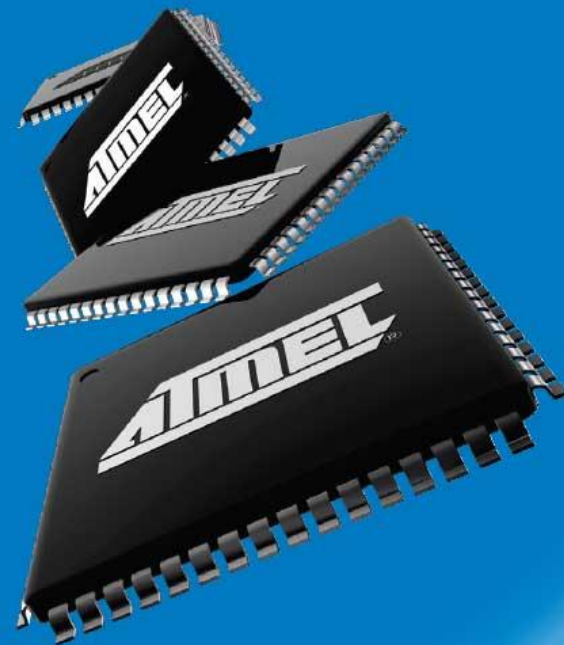
AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors



Interrupts and 16-bit Timer/Counter 1 (Normal Mode)

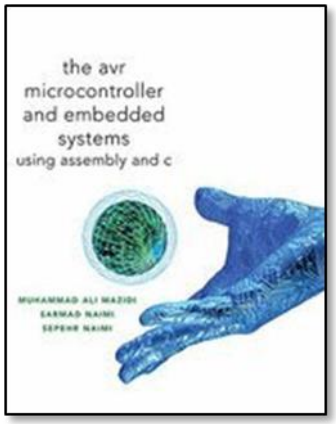
February 2009

ATMEL®

Everywhere You Are®

ATmega Interrupts

Reading



The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi

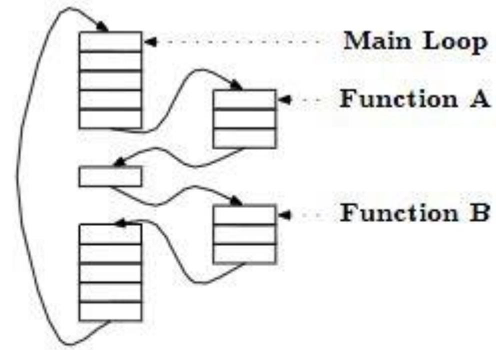
Chapter 10: AVR Interrupt Programming in Assembly and C

Section 10.1: AVR Interrupts

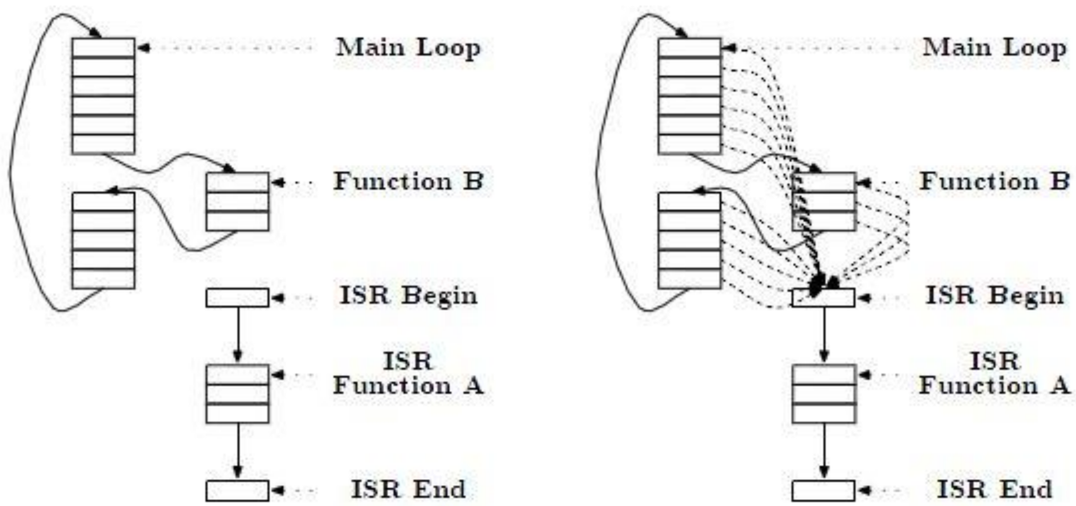
Section 10.4: Interrupt Priority in the AVR

Table of Contents

Interrupt Basics



- A microcontroller normally executes instructions in an orderly fetch-execute sequence as dictated by a user-written program.
- However, a microcontroller must also be ready to handle unscheduled, events that might occur inside or outside the microcontroller.
- The interrupt system onboard a microcontroller allows it to respond to these internally and externally generated events. By definition we do not know when these events will occur.
- When an interrupt event occurs, the microcontroller will normally complete the instruction it is currently executing and then transition program control to an Interrupt Service Routine (ISR). These ISR, which handles the interrupt.
- Once the ISR is complete, the microcontroller will resume processing where it left off before the interrupt event occurred.



(a) Program with ISR

(b) ISR called from anywhere

The Main Reasons You Might Use Interrupts[1]

- To detect pin changes (eg. rotary encoders, button presses)
- Watchdog timer (eg. if nothing happens after 8 seconds, interrupt me)
- Timer interrupts - used for comparing/overflowing timers
- SPI data transfers
- I2C data transfers
- USART data transfers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready

[1] Source: Gammon Software Solutions forum – What are interrupts?

ATmega328P Interrupt Vector Table

- The ATmega328P provides support for 25 different interrupt sources. These interrupts and the separate Reset Vector each have a separate program vector located at the lowest addresses in the Flash program memory space.
- The complete list of vectors is shown in Table 11-6 “Reset and Interrupt Vectors in ATMega328P. Each Interrupt Vector occupies two instruction words.
- The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request 0.

ATmega328P Interrupt Vector Table

Vector No	Program Address	Source	Interrupt Definition	Arduino/C++ ISR() Macro Vector Name
1	0x0000	RESET	Reset	
2	0x0002	INT0	External Interrupt Request 0 (pin D2)	(INT0_vect)
3	0x0004	INT1	External Interrupt Request 1 (pin D3)	(INT1_vect)
4	0x0006	PCINT0	Pin Change Interrupt Request 0 (pins D8 to D13)	(PCINT0_vect)
5	0x0008	PCINT1	Pin Change Interrupt Request 1 (pins A0 to A5)	(PCINT1_vect)
6	0x000A	PCINT2	Pin Change Interrupt Request 2 (pins D0 to D7)	(PCINT2_vect)
7	0x000C	WDT	Watchdog Time-out Interrupt	(WDT_vect)
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A	(TIMER2_COMPA_vect)
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B	(TIMER2_COMPB_vect)
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow	(TIMER2_OVF_vect)
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event	(TIMER1_CAPT_vect)
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A	(TIMER1_COMPA_vect)
13	0x0018	TIMER1 COMPB	Timer/Counter1 Compare Match B	(TIMER1_COMPB_vect)
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow	(TIMER1_OVF_vect)
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A	(TIMER0_COMPA_vect)
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B	(TIMER0_COMPB_vect)
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow	(TIMER0_OVF_vect)
18	0x0022	SPI, STC	SPI Serial Transfer Complete	(SPI_STC_vect)
19	0x0024	USART, RX	USART Rx Complete	(USART_RX_vect)
20	0x0026	USART, UDRE	USART, Data Register Empty	(USART_UDRE_vect)
21	0x0028	USART, TX	USART, Tx Complete	(USART_TX_vect)
22	0x002A	ADC	ADC Conversion Complete	(ADC_vect)
23	0x002C	EE READY	EEPROM Ready	(EE_READY_vect)
24	0x002E	ANALOG COMP	Analog Comparator	(ANALOG_COMP_vect)
25	0x0030	TWI	2-wire Serial Interface (I2C)	(TWI_vect)
26	0x0032	SPM READY	Store Program Memory Ready	(SPM_READY_vect)

ATMEGA328P INTERRUPT PROCESSING

- When an interrupt occurs, the microcontroller completes the current instruction and stores the address of the next instruction on the stack
- It also turns off the interrupt system to prevent further interrupts while one is in progress. This is done by clearing the SREG Global Interrupt Enable I-bit.

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	**I**	**T**	**H**	**S**	**V**	**N**	**Z**	**C**	**SREG**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- The Interrupt flag bit is cleared for Type 1 Interrupts only (see the next page for Type definitions).
- The execution of the ISR is performed by loading the beginning address of the ISR specific for that interrupt into the program counter. The AVR processor starts running the ISR.
- Execution of the ISR continues until the return from interrupt instruction (`reti`) is encountered. The SREG I-bit is automatically set when the `reti` instruction is executed (i.e., Interrupts enabled).
- When the AVR exits from an interrupt, it will always return to the interrupted program and execute one more instruction before any pending interrupt is served.
- 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.

```
push  reg_F
in    reg_F,SREG
  :
out   SREG,reg_F
pop   reg_F
```

ATMEGA328P INTERRUPT PROCESSING – TYPE 1 –

- 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.
  - The SREG I-bit is automatically set to logic one when a Return from Interrupt instruction – RETI – is executed.
- There are basically two types of interrupts…
  - The **first type (Type 1)** is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector in order to execute the interrupt handling routine, and **hardware clears the corresponding Interrupt Flag**.
    - If the same interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software (interrupt cancelled).
    - Interrupt Flag can be cleared by writing a logic one to the flag bit position(s) to be cleared.
  - If one or more interrupt conditions occur while the Global Interrupt Enable (SREG I) bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the Global Interrupt Enable bit is set on return (`reti`), and will then be executed by order of priority.

ATmega328P Interrupt Processing – Type 2 –

- The **second type (Type 2)** of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

WHEN WRITING AN INTERRUPT SERVICE ROUTINE (ISR)[2]

- As a general rule get in and out of ISRs as quickly as possible. For example do not include timing loops inside of an ISR.
- If you are writing an Arduino program
  - Don't add delay loops or use function **delay()**
  - Don't use function **Serial.print(*val*)**
  - Make variables shared with the main code **volatile**
  - Variables shared with main code may need to be protected by "critical sections" (see below)
  - Toggling interrupts off and on is not recommended. The default in the Arduino is for interrupts to be enabled. Don't disable them for long periods or things like timers won't work properly.

[2] Source: Gammon Software Solutions forum – What are interrupts?

Program Initialization and the Interrupt Vector Table (IVT)

- Start by jumping over the Interrupt Vector Table

```
RST_VECT:
   rjmp   reset
```

- Add jumps in the IVT to your ISR routines

```
 .ORG INT0addr        // 0x0002 External Interrupt 0
   jmp    INT0_ISR
.ORG OVF1addr
   jmp    TOVF1_ISR
```

- Initialize Variables, Configure I/O Registers, and Set Local Interrupt Flag Bits

```
reset:
   lds   r16, EICRA     // EICRA Memory Mapped Address 0x69
   sbr   r16, 0b00000010
   cbr   r16, 0b00000001
   sts   EICRA, r16     // ISC0=[10] (falling edge)
   sbi   EIMSK, INT0    // Enable INT0 interrupts
```

- Enable interrupts at the end of the initialization section of your code.

```
   sei                         // Global Interrupt Enable

loop:
```

THE INTERRUPT SERVICE ROUTINE (ISR)

```
; -- Interrupt Service Routine --
INT0_ISR:
  push   reg_F
  in     reg_F,SREG
  push   r16
  ; Load
  ; Do Something
  ; Store
  pop    r16
  out    SREG,reg_F
  pop    reg_F
  reti
; -------------------------------------------------------
```

Predefined Arduino IDE Interrupts

- When you push the reset button the ATmega328P automatically runs an Arduino Boot program located in a separate Boot Flash section at the top of program memory. This Boot program configures and enables two interrupts.

17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow	(TIMER0_OVF_vect)

- The `millis()` and `micros()` function calls make use of the "timer overflow" feature utilize timer 0. The ISR runs roughly 1000 times a second, and increments an internal counter which effectively becomes the `millis()` counter (see On your own question).

19	0x0024	USART, RX	USART Rx Complete	(USART_RX_vect)
21	0x0028	USART, TX	USART, Tx Complete	(USART_TX_vect)

- The hardware serial library uses interrupts to handle incoming and outgoing serial data. Your program can now be doing other things while data in an SRAM buffer is sent or received. You can check the status of the buffer by calling the `Serial.available()` function.

- **On your own**. Given that you are using 8-bit Timer/Counter 0, you have set TCCR0B bits CS02:CS01:CS00 = 0b011 ($clk_{I/O}/64$), and the system clock f_{clk} = 16 MHz, what value would you preload into the Timer/Counter Register TCNT0 to get a interrupt 1000 times a second.

Source: Gammon Software Solutions forum – this blog also covers how to work with all the interrupts in C++ and the Arduino scripting language.

Programming the Arduino to Handle External Interrupts[3]

- Variables shared between ISRs and normal functions should be declared "`volatile`". This tells the compiler that such variables might change at any time, and thus the compiler should not "optimize" the code by placing a copy of the variable in one of the general purpose processor registers (R31..R0). Specifically, the processor must reload the variable from SRAM whenever it is referenced.

```
int pin = 13;
volatile int state = LOW;
```

- Add jumps in the IVT to ISR routine, configure External Interrupt Control Register A (EICRA), and enable local and global Interrupt Flag Bits.

```
 void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}
```

[3] Read ATmega328P External Interrupts to learn more about this example.

PROGRAMMING THE ARDUINO TO HANDLE EXTERNAL INTERRUPTS - CONTINUED[4]

- Write Interrupt Service Routine (ISR)

```
void blink()
{
  state = !state;
}
```

- To disable interrupts globally (clear the I bit in SREG) call the `noInterrupts()` function. To once again enable interrupts (set the I bit in SREG) call the `interrupts()` function.

- Again – Toggling interrupts ON and OFF is not recommended. For a discussion of when you may want to turn interrupts off, read Gammon Software Solutions forum – Why disable Interrupts?

[4] Read ATmega328P External Interrupts to learn more about this example.