Interrupts

An *interrupt* is an exception, a change of the normal progression, or interruption in the normal flow of program execution.

An interrupt is essentially a hardware generated function call.

Interrupts are caused by both internal and external sources.

An interrupt causes the normal program execution to halt and for the interrupt service routine (ISR) to be executed.

At the conclusion of the ISR, normal program execution is resumed at the point where it was last.

Interrupts

Interrupts should be used for infrequent events (1000's of clock cycles)

- -keyboard strokes
- -1ms clock ticks
- -serial data (USART, SPI, TWI)
- -analog to digital conversion

uC response time to interrupts is very fast

- -AVR: 4 cycles max
- -80386: 59 cycles min, 104 cycles max;

(0-wait state memory; www.intel.com/design/intarch/technote/2153.htm)

If response time is really critical, a "tight" *polling loop* is used.

- -polling can be faster than interrupts.....,

but further processing is on hold!

```
//spin on SPSR bit checking for serial transfer complete
while (bit_is_clear(SPSR,SPIF)) {};
da:   77 9b           sbis    0x0e, 7 ; 1 or 2 cycles
dc:   fe cf           rjmp    .-4     ; 2 cycles
```

Interrupts

Interrupts vs. Polling

Polling uses a lot of CPU horsepower

- -checking whether the peripheral is ready or not
- -are you ready...yet?!
- -interrupts use the CPU only when work is to be done

Polled code is generally messy and unstructured

- -big loop with often multiple calls to check and see if peripheral is ready
- -necessary to keep peripheral from waiting
- -ISRs concentrate all peripheral code in one place (encapsulation)

Polled code leads to variable latency in servicing peripherals

- -whether if branches are taken or not, timing can vary
- -interrupts give highly predictable servicing latencies

Interrupts

AVR interrupt servicing

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

```
00000000 <__vectors>:
   0:   0c 94 46 00     jmp     0x8c    ; 0x8c <__ctors_end>
   4:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
   8:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
   c:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  10:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  14:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  18:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  1c:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  20:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  24:   0c 94 63 00     jmp     0xc6    ; 0xc6 <__bad_interrupt>
  28:   0c 94 b4 02     jmp     0x538   ; 0x538 <__vector_10>
```

Interrupts

AVR interrupt servicing (cont.)

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

From *.lst file:

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

compiler generated code

user code

Interrupts

AVR interrupt servicing (cont.)

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

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

compiler generated code

6. The CPU resumes executing the original instruction stream.

Interrupts

AVR interrupt vector table

-All interrupts have separate interrupt vectors in the interrupt vector table

-Interrupts have priority in accordanc with their position in the table.

-Lower interrupt vector address have higher priority.

-Reset has top priority.

Table 23. Reset and Interrupt Vectors

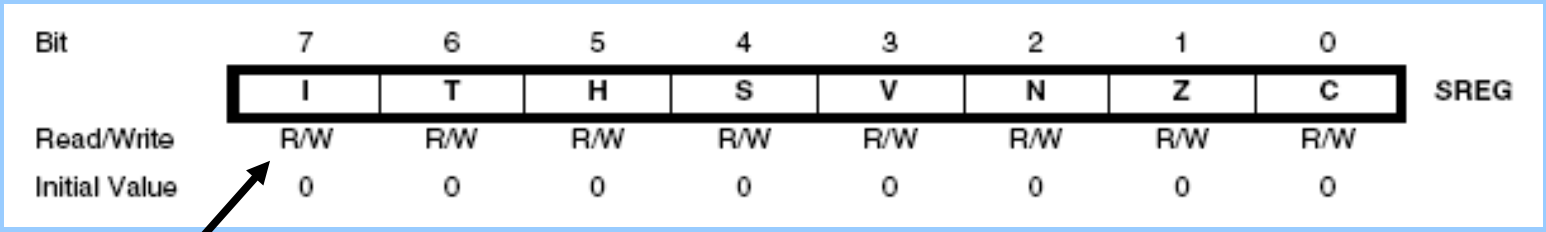
Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External 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	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow

31	$003C(3)	USART1, RX	USART1, Rx Complete
32	$003E(3)	USART1, UDRE	USART1 Data Register Empty
33	$0040(3)	USART1, TX	USART1, Tx Complete
34	$0042(3)	TWI	Two-wire Serial Interface
35	$0044(3)	SPM READY	Store Program Memory Ready

Interrupts

Enabling interrupts

1. Global Interrupt Enable (GIE) bit must be set in the status register (SREG)



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

Global Interrupt Enable (GIE):
This bit must be set for interrupts to be enabled. To set GIE:

```
sei();          //global interrupt enable
```

It is cleared by hardware after an interrupt has occurred, but may be set again by software [ISR(xxx_vec, ISR_NOBLOCK)] or manually with the `sie()` to allow nested interrupts.
It is set by the RETI instruction to enable subsequent interrupts. This is done automatically by the compiler.

Interrupts

Enabling interrupts

2. The individual interrupt enable bits must be set in the proper control register.
   -For example, for timer counter 1, the timer overflow bit (TOV)

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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	

- **Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 overflow interrupt is enabled. The corresponding interrupt vector (see "Interrupts" on page 57) is executed when the TOV1 flag, located in TIFR, is set.

TOIE1 must be set to allow the TCNT1 over flow bit to cause an interrupt

The interrupt occurs when the TOV1 flag becomes set. Once you enter the ISR, it is reset automatically by hardware.

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

- **Bit 2 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent of the WGMn3:0 bits setting. In normal and CTC modes, the TOV1 flag is set when the timer overflows. Refer to Table 61 on page 133 for the TOV1 flag behavior when using another WGMn3:0 bit setting.

TOV1 is automatically cleared when the Timer/Counter1 Overflow interrupt vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

Interrupts

Avr-libc ISR code generation *(from *.lst)*

The c code:

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

The compiler generated ISR:

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

Interrupts

ISR Signal Names

```
ISR(SPI_STC_vect){};        //SPI sertial transfer complete
ISR(TIMER0_COMP_vect){};    //TCNT0 compare match A
ISR(TIMER0_OVF_vect){};     //TCNT0 overflow
ISR(INT2_vect){};           //External Interuupt Request 2
```

These names come from:
http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Interrupts

ISR Usage

-understand how often the interrupt occurs
-understand how much time it takes to service each interrupt
-make sure there is enough time to service all interrupts and to still get work done in the main loop
-there's only 1 sec of compute time per second to get everything done!

-Keep ISRs short and simple. (short = short time, not short code length)
Do only what has to be done then RETI.
Long ISRs may preclude others from being run
Ask yourself, "Does this code really need to be here?"

-Example: alarm clock with 1Sec interrupts:
-at interrupt, just increment the "seconds counter" and return
-in the main code loop we can
-do binary to BCD conversion
-lookup display code
-write to display

Interrupts

Effecting main program flow with ISRs

ISRs are never called by the main routine. Thus,

-nothing can be passed to them (there is no "passer")

-they can return nothing (its not a real function call)

So, how can it effect the main program flow?

-use a global? yech!

-use *volatile* type modifier

Compiler has a optimizer component (let's digress a moment...)

-02 level optimization can optimize away some variables

-it does so when it sees that the variable cannot be changed within the scope of the code it is looking at

-variables changed by the ISR are outside the scope of main()

-thus, they get optimized away

Volatile tells the compiler that the variable is shared and may be subject to outside change elsewhere.

Interrupts

Example:

```
volatile uint8_t tick; //keep tick out of regs!

ISR(TIMER1_OVF_vect){
  tick++; //increment my tick count
}

main(){

while(tick == 0x00){
    bla, bla, bla...
}
```

Without the volatile modifier, -02 optimization removes *tick* because nothing in while loop can ever change *tick*.

Interrupts

Volatile Variable Example:

```
// tcnt1_volatile.c

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

volatile uint8_t external_count;  //protected against optimiaztion

/***** interrupt service routine *****/
ISR(TIMER1_OVF_vect) {external_count++;}

int main() {
  DDRB   = 0x01;                  //set all port B bit zero to output
  TCCR1A = 0x00;                  //normal mode
  TCCR1B = (1<<CS11) | (1<<CS10); //use clk/64
  TCCR1C = 0x00;                  //no forced compare
  TIMSK |= (1<<TOIE1);            //enable tcnt1 timer overflow bit

  sei();       //global interrupt enable
  while(1){
     //spin repeatedly setting PORTB to 0x01 while external_count
     //is an odd number
     while((external_count % 2) == 0x01){PORTB = 0x01;}
     PORTB = 0x00; //if even, set PORTB to all zeros
  }//while
}// main
```

Interrupts

Volatile Variable Example - from `main()`:

Case:

```
volatile uint8_t external_count
OPTIMIZE = -O2
```

```
sei();
10c:  78 94        sei                          ;interrupts turned on
 while(1){
    while((external_count % 2) != 0x00){PORTB = 0x01;}
10e:  80 91 00 01  lds    r24, 0x0100           ;load r24 with external_count
112:  80 ff        sbrs   r24, 0                ;skip next if bit 0, r24 is set
114:  06 c0        rjmp   .+12                  ;jump forwards to address 0x122
116:  91 e0        ldi    r25, 0x01             ;load r25 with 0x01
118:  98 bb        out    0x18, r25             ;send 0x01 to PORTB
11a:  80 91 00 01  lds    r24, 0x0100           ;put r24 back into SRAM
11e:  80 fd        sbrc   r24, 0                ;skip next if bit 0, r24 is cleared
120:  fb cf        rjmp   .-10                  ;jump backwards to address 0x118
    PORTB = 0x00;
122:  18 ba        out    0x18, r1              ;put value of r1 out to PORTB
124:  f4 cf        rjmp   .-24                  ;jump back to address 0x10E
```

Interrupts

Volatile Variable Example – from `main()`:

Case: `uint8_t external_count`
`OPTIMIZE = -O2`
same code only change is "volatile"

variable external_count is optimized away!

```
 sei();
10c:  78 94        sei                         ;interrupts on
10e:  81 e0        ldi    r24, 0x01            ;laod r24 with 0x01
 while(1){
    while((external_count % 2) != 0x00)
      PORTB = 0x01;
110:  88 bb        out    0x18, r24            ;write a 0x01 to PORTB
112:  88 bb        out    0x18, r24            ;do it again!
114:  fd cf        rjmp   .-6                  ;jump back to address 0x110
```

GCC optimization levels

-O0

Does not perform any optimization. Compiles the source code in the most straightforward way possible. Each line of source code is converted directly to corresponding instructions without rearrangement. Best option when debugging.

-O1

Turns on the most common forms of optimization that do not require any speed-space tradeoffs. Resulting executables should be smaller and faster than with -O0.

-O2

Turns on further optimizations such as instruction scheduling. Optimizations that do not require any speed-space tradeoffs are used, so executable should not increase in size. Provides maximum optimization without increasing the executable size. Default optimization level for GNU packages.

-O3

Turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels. The -O3 optimization level may increase the speed of the resulting executable, but can also increase its size.

-Os

Selects optimizations which reduce the size of an executable. Tries to produce the smallest possible executable, for memory constrained systems.

The benefit of optimization must be weighed against the cost. Cost of optimization includes more difficult debugging. Hint: use -O0 for debugging, and -O2 for final product.