Interrupts

Hardware and Software interrupts
and event-driven programming

References and Resources

- Introduction to Embedded Programming ASM and C examples
  - http://www.scriptoriumdesigns.com/embedded/interrupts.php
  - http://www.scriptoriumdesigns.com/embedded/interrupt_examples.phpinterrupt examples
- GNU C Programming:
  - http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html
- Newbie's Guide to AVR Interrupts
  - http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=89843
- Using Interrupts
  - http://www.pjrc.com/teensy/interrupts.html
- Interrupt driven USARTs
  - http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=48188>
- Atmega169P Datasheet
  - http://www.atmel.com/dyn/resources/prod_documents/doc8018.pdf
- Beginners Programming in AVR Assembler
  - http://www.avr-asm-tutorial.net/avr_en/beginner/RIGHT.html
- Code segments cseg and org
  - http://www.avr-asm-tutorial.net/avr_en/beginner/JUMP.html
- A Brief Tutorial on Programming the AVR without Arduino
  - https://www.mainframe.cx/~ckuethe/avr-c-tutorial/

What is an Interrupt?

- An interrupt is a signal (an “interrupt request”) generated by some event external to the CPU
- Causes CPU to stop currently executing code and jump to separate piece of code to deal with the event
  - Interrupt handling code often called an ISR (“Interrupt Service Routine”)
- When ISR is finished, execution returns to code running prior to interrupt

Interrupt Overview

- Interrupt Sequence
  - Interrupt Event
  - Interrupt Request
  - Interrupt Service Routine
- Varying number of ISRs may be supported
  - Typically between 1-16 depending on platform
- Multiple events may be mapped to a single routine
  - i.e. any PORTA pin change
- Interrupt events may have priorities
- ISR is implemented inside a function with no parameters and no return value (void)
- Typically keep interrupt routines shorter than 15-20 lines of code

Interrupt Sources

- Hardware Interrupts
  - Commonly used to interact with external devices or peripherals
  - Microcontroller may have peripherals on chip
- Software Interrupts
  - Triggered by software commands, usually for special operating system tasks
    - i.e. switching between user and kernel space, handling exceptions
- Common hardware interrupt sources
  - Input pin change
  - Hardware timer overflow or compare-match
  - Peripherals for serial communication
    - UART, SPI, I2C – Rx data ready, tx ready, tx complete
  - ADC conversion complete
  - Watchdog timer timeout

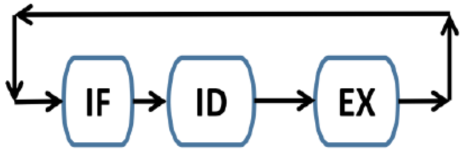
Advantage Over Software Polling

- Interrupts avoid writing software code in such a way that the processor must frequently spend time checking the status of input pins or registers
  - Lets custom hardware do that
- Polling often has a “busy-wait”
  - CPU executes instructions waiting for event

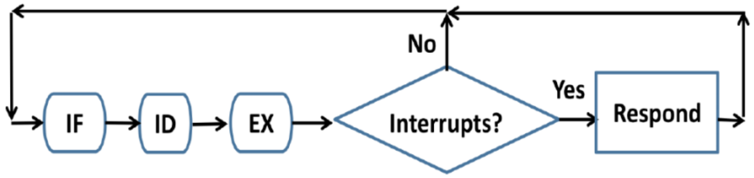
```
void USART_Transmit( unsigned char data ){
        while ( !( UCSR0A & (1<<UDRE0)) );    ← busy-wait
        UDR0 = data;
}
```

Interrupt Servicing Flow

- Normal execution flow:



- Interrupt Check Flow:



- After executing each instruction, check for any pending interrupts
  - If there is an interrupt, save PC and load address of ISR into PC
- After handling the interrupt, old PC is restored and execution of program resumes

Interrupt Vector Table

- Mapping of interrupt events/requests to functions is handled with an interrupt vector table
    - A table of function pointers
    - Interrupt system calls correct function from a table based on event that occurred
- Just like a function call, system state should be pushed onto the stack as needed to return and continue execution
- When a routine is called, it is said the interrupt has been “handled” or “serviced”

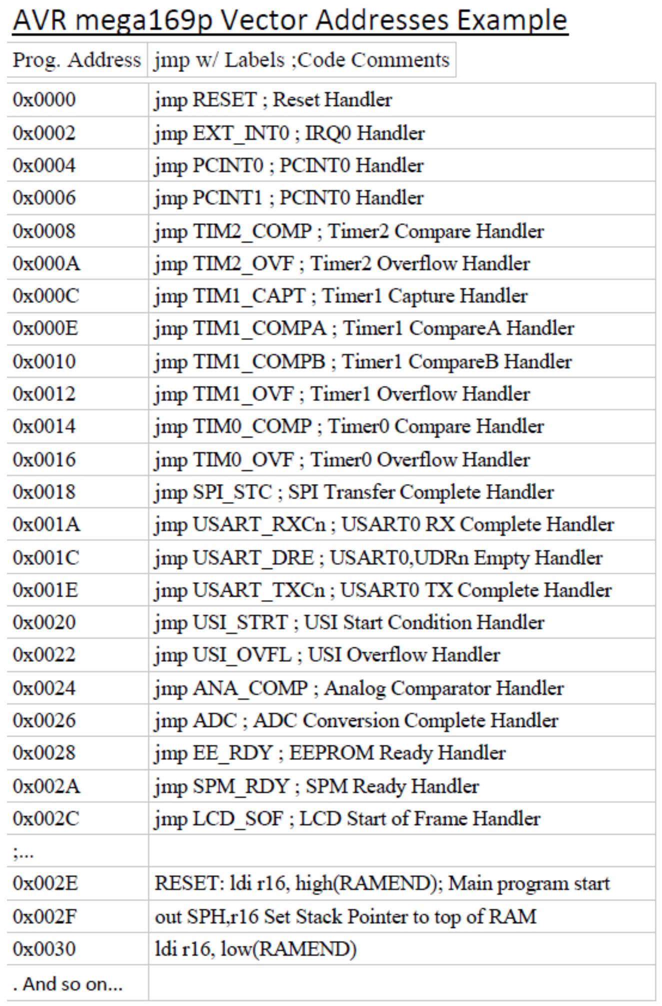
Event #	Handler Address
0	funcPtr[0]
1	funcPtr[1]
2	funcPtr[2]
3	funcPtr[3]
4	funcPtr[4]
5	funcPtr[5]

Implementations of Interrupt Vector Table

- Vector tables may be implemented as simple function addresses or instructions (typically an unconditional jump) depending on system
- Vector holds jump to ISR
  - .org This_Vector_Address
  - VN: jmp VISR ;jump to ISR
  - …
  - VISR: ISR for Interrupt N
- Vector holds address of ISR
  - .org This_Vector_Address
  - VN: VISR ;address of ISR
  - …
  - VISR: ISR for Interrupt N …

AVR Interrupt Vector Table

- On AVR, interrupt vector table is implemented at the top of program memory with jmp instructions that are listed at beginning of program memory, position determines the interrupt number



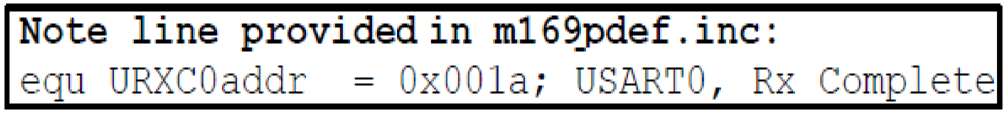
AVR mega169p Vector Addresses Example

Prog. Address	jmp w/ Labels ;Code Comments
0x0000	jmp RESET ; Reset Handler
0x0002	jmp EXT_INT0 ; IRQ0 Handler
0x0004	jmp PCINT0 ; PCINT0 Handler
0x0006	jmp PCINT1 ; PCINT0 Handler
0x0008	jmp TIM2_COMP ; Timer2 Compare Handler
0x000A	jmp TIM2_OVF ; Timer2 Overflow Handler
0x000C	jmp TIM1_CAPT ; Timer1 Capture Handler
0x000E	jmp TIM1_COMPA ; Timer1 CompareA Handler
0x0010	jmp TIM1_COMPB ; Timer1 CompareB Handler
0x0012	jmp TIM1_OVF ; Timer1 Overflow Handler
0x0014	jmp TIM0_COMP ; Timer0 Compare Handler
0x0016	jmp TIM0_OVF ; Timer0 Overflow Handler
0x0018	jmp SPI_STC ; SPI Transfer Complete Handler
0x001A	jmp USART_RXCn ; USART0 RX Complete Handler
0x001C	jmp USART_DRE ; USART0,UDRn Empty Handler
0x001E	jmp USART_TXCn ; USART0 TX Complete Handler
0x0020	jmp USI_STRT ; USI Start Condition Handler
0x0022	jmp USI_OVFL ; USI Overflow Handler
0x0024	jmp ANA_COMP ; Analog Comparator Handler
0x0026	jmp ADC ; ADC Conversion Complete Handler
0x0028	jmp EE_RDY ; EEPROM Ready Handler
0x002A	jmp SPM_RDY ; SPM Ready Handler
0x002C	jmp LCD_SOF ; LCD Start of Frame Handler
;...	
0x002E	RESET: ldi r16, high(RAMEND); Main program start
0x002F	out SPH,r16 Set Stack Pointer to top of RAM
0x0030	ldi r16, low(RAMEND)
. And so on...	

Coding Interrupts in AVR Assembly

If using interrupts

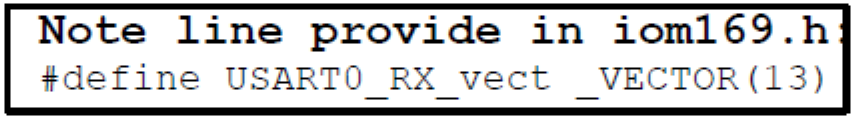
- Must provide system reset vector at .org 0x0000 with jmp to main
  - .org 0x0000
  - jmp Main
- Provide vector table entry of interest
  - .org URXCOaddr
  - jmp myISRHandler
- Provide Handler
  - myISRHandler:
  - ; ISR code to execute here
  - reti;
- Interrupts must be enabled globally in main or elsewhere
  - Main:
  - sei; Enable Global Interrupts
- Enable specific interrupts in control registers in main or elsewhere
  - in r16, UCSRB
  - ori r16, (1<<RXCIE)
  - out UCSRB, r16



```
Note line provided in m169pdef.inc:
equ URXC0addr  = 0x001a; USART0, Rx Complete
```

Coding Interrupts in AVR C

- Include interrupt macros and device vector definitions
  - #include <avr/interrupt.h>
  - #include <avr/io.h>
- Define ISR using macro and appropriate vector name: by default the compiler determines all registers that will be modified and saves them (prologue code) and restores them for you (epilog)
  - ISR(UART0_RX_vect){
  - // ISR code to execute here
  - }



```
Note line provide in iom169.h
#define USART0_RX_vect _VECTOR(13)
```

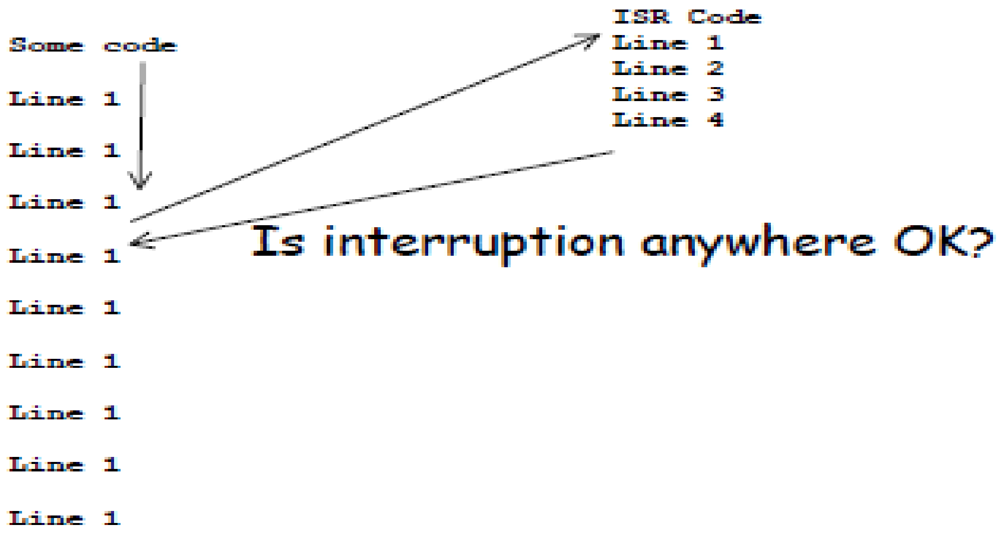
- Somewhere in main or function code
  - sei(); //Enable global interrupts
- Enable specific interrupts of interest in corresponding control registers. Ex:
  - UCSRB |= (1<<RXCIE); //enable interrupt

Keeping ISR Short

- ISRs affect the normal execution of program and can block handling of other interrupts
- Common strategy for ISR is to keep it as short as possible
- Creates stable timing and avoids system from being flooded with handling certain ISRs and not being able to service other interrupts fast enough
- Each ISR should do only what it needs to do at the time of the event
- If long ISRs are needed, consider allowing nested interrupts

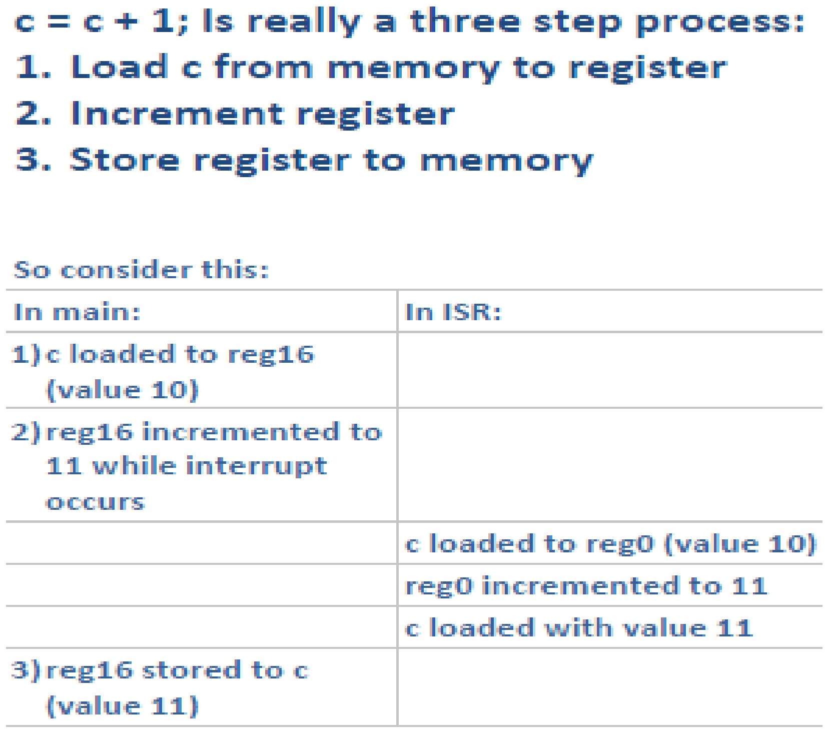
Critical Sections of Code

- A critical section is a section of code which must have complete and undisturbed access to a block of data or any other resource



Interaction of ISR and variables

- Use of volatile keyword to prevent erroneous optimization by the compiler for any shared variables
- volatile uint8_t c;
- ISR code :
  - c=c+1;
- Main:
  - c = 10;
  - c=c+1;



c = c + 1; Is really a three step process:

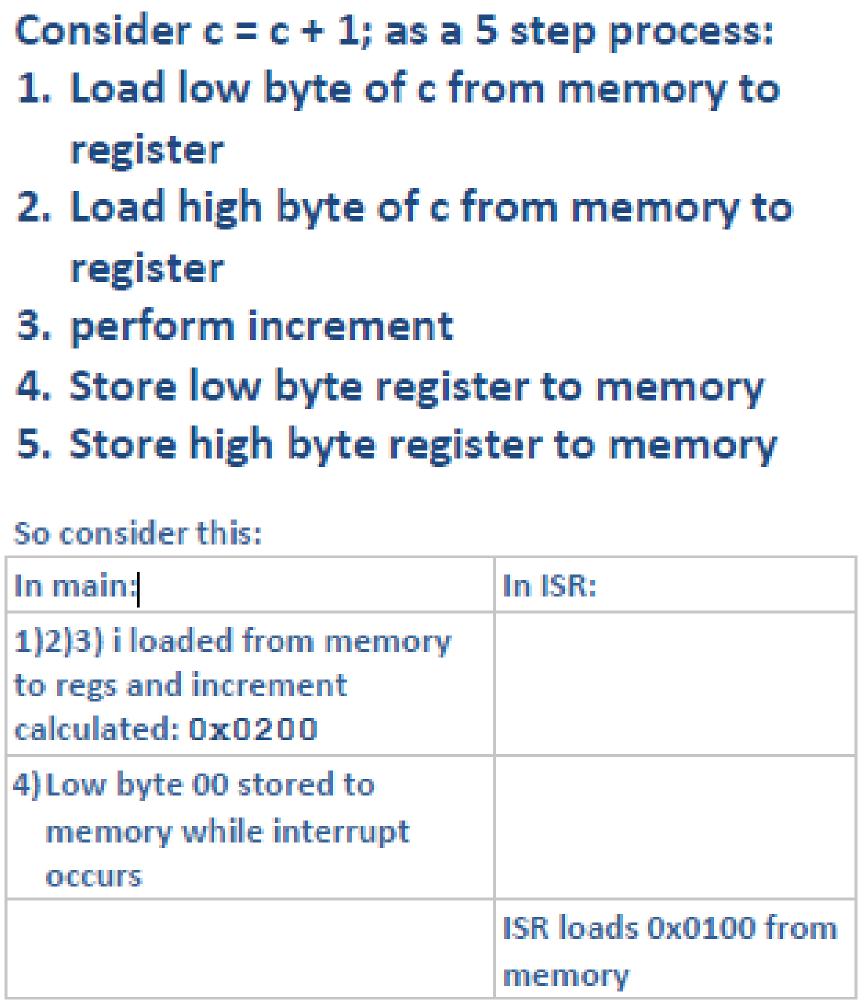
1. Load c from memory to register
2. Increment register
3. Store register to memory

So consider this:

In main:	In ISR:
1) c loaded to reg16 (value 10)	
2) reg16 incremented to 11 while interrupt occurs	
	c loaded to reg0 (value 10)
	reg0 incremented to 11
	c loaded with value 11
3) reg16 stored to c (value 11)	

ISRs and multiword code

- Global declaration
  - volatile uint16_t c;
- ISR code:
  - if(c==0x100)
  -     PORTB = 1;
- Main:
  - c = 0x01FF;
  - c= c+1;



Consider c = c + 1; as a 5 step process:

1. Load low byte of c from memory to register
2. Load high byte of c from memory to register
3. perform increment
4. Store low byte register to memory
5. Store high byte register to memory

So consider this:

In main:	In ISR:
1)2)3) i loaded from memory to regs and increment calculated: 0x0200	
4) Low byte 00 stored to memory while interrupt occurs	
	ISR loads 0x0100 from memory

Atomic Resource Access

- Once you start using interrupts you must be sensitive to code that it is not OK to interrupt
  - During execution of such code, some interfering ISRs should be blocked
- Alternatively, resources like shared variables or hardware registers may be protected form multiple interfering access by using additional code to flag access to shared resource
- Simplest way to guarantee ATOMIC access is to temporarily disable interrupts
- AVR gcc provides a macro “ATOMIC_BLOCK” to disable interrupts

Multiple Pending Interrupts in AVR

- If multiple interrupt requests are pending, the order in which they are handled is system dependent
  - Some predefine priorities based on event number
  - Others allow software defined priorities
- AVR uses lowest-addressed vector
  - Execution flow returns to main allowing at least one ASSEMBLY instruction to run before handling next IRQ

New Interrupts during ISR

- What about new interrupts during ISR execution
  - If interrupt service routines are (by default) interrupted by higher priority interrupts or at all is system-dependent. You must make yourself aware of how a given system handles interrupts
- AVR interrupts are disabled by default when ISR is called until RETI is encountered

Interrupt Enabling

- Typically, methods exist to globally disable or enable interrupts (AVR provides sei,cli in asm and C).
- Furthermore, individual Interrupts can be enabled/disabled according to the status of certain flag bits which may be modified.
- It is common to set the enable bits for all the individual interrupts that should be initially enabled and then set the global interrupt enable.
- Other interrupts can be enabled and disabled as needed.
- You may temporally disable individual interrupts as needed
- You may temporally disable all interrupts using global enable/disable commands
- Many systems disable interrupts upon invoking an ISR (or at least temporarily disable interrupts after an ISR is called to allow the coder to disable them for longer if desired) to prevent other interrupt service routines.

Interrupt Mask Registers

- Interrupt Mask Registers
  - Potential to enable or disable groups of interrupts through a masking process
- Interrupts can be disabled when
  - Not needed or used
  - Critical section of code is running
    - Can’t be interrupted because of trimming or order of operations

Clearing the Interrupt Flag (IRQ)

- When ISR is called, the corresponding interrupt flag must be cleared or the ISR would be called again
- Depending on the system, the flag may be cleared
  - Automatically by hardware
  - Require that software for an ISR must handle clearing the flag
- AVR clears the flag automatically using hardware as soon as the ISR is called
  - Means that if multiple interrupts are mapped to the same ISR there is no way to tell in the ISR itself which event triggered it

Interrupt Sequence (AVR)

- With interrupts enabled and foreground code running, an Interrupt Event occurs
  - A request is flagged by the hardware
- Current Instruction Completed (machine instruction, which is NOT same as a line of C code)
- Address of next instruction is stored on the stack
- Address of ISR is loaded into PC and Global Interrupt Enable Bit is Cleared and Specific Interrupt Flag is Cleared automatically indicating it has been handled
- Processor Executes ISR
  - If desired, interrupts should be be reenabled with sei() command to allow the ISR itself to be interrupted (if writing C, avr-gcc provide macro for this)
  - If any state registers should be saved because they will be changed, they must be explicitly saved (if using C avr-gcc provides macros that do this)
- reti is encountered (C macros take care of including this)
- PC loaded from stack and Global Interrupt Enable Bit is Set
- Foreground code execution continues

Interrupt Response Time (AVR)

- The interrupt execution response for all the enabled 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 four 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 (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.
- Pasted from datasheet

Questions for ISR coding

- How are ISRs and the interrupt vector table defined?
- Are there priorities of interrupts and how are they defined?
- Nested Interrupts: Do you want them and are they enabled by default? Do you need to enabled or disable interrupts be to allow nested interrupts?
  - In AVR, interrupts are disabled when an interrupt routine is called, so you need to explicitly call sei() in ISR if desired
- Which interrupts should be enabled?
  - Should only certain interrupts should be enabled? May be a mechanism to allow int. ISRs based on priority. Otherwise, may be able to manually enabled selected interrupts
- How is the state restored and what side effects can be caused when interrupts are called?
  - Generally, ISRs should save and restore status registers and any registers it uses for work. Can use stack or RAM for this.

Questions (continued)

- Where, if anywhere, should interrupts be disabled?
  - Main code, with atomic or critical sections of code should disable interrupts. May be related to timing or need to access a set of resources without any interruptions to avoid invalid states or just to achieve proper sequences of operations to peripheral hardware.
- How are interrupts flagged as being serviced?
  - Should interrupts do something to indicate IRQ has been handled? Usually handled automatically internally by processor, but if request is coming from external device it may need a signal that the request has been handled.
- Do interrupt service routines need to set some flag to indicate further action to be taken by main code or peripheral?