Interrupt Vector Table

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

- The mapping of interrupt events/requests to functions is handled with an interrupt vector table, which is basically a table of function pointers (each of type (void *) void)
- The interrupt system calls the correct function from a table based on event that occurred.
- Just like a function call, the system state should be pushed onto the stack as needed (depends on what is modified by the routines code) to return and continue execution.
- When a routine is called, it is said the interrupt has been handled or serviced
- The call interrupts the execution of the main sequence of code; execution returns to the main sequence when the interrupt routine is finished.

Implementations of interrupt vector table

- Vector tables may be implemented as simple function addresses or instructions (typically an unconditional jump) depended on the 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
…
```

- Pasted from http://www.scriptoriumdesigns.com/embedded/interrupts.php

AVR Hardware Implementation of Interrupt Vector Table

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

Defining the interrupt vector table

- Each platform (hardware+software+dev.tools) has some method of defining the interrupt vector table.

Prog. Addr:	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 Ex:

```
.org URXC0   add
jmp MyISRHandler
```

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

  - Provide Handler

```
MyISRHandler:
  ; ISR code to execute here
  reti
```

  - Interrupts must be enabled globally in main or elsewhere

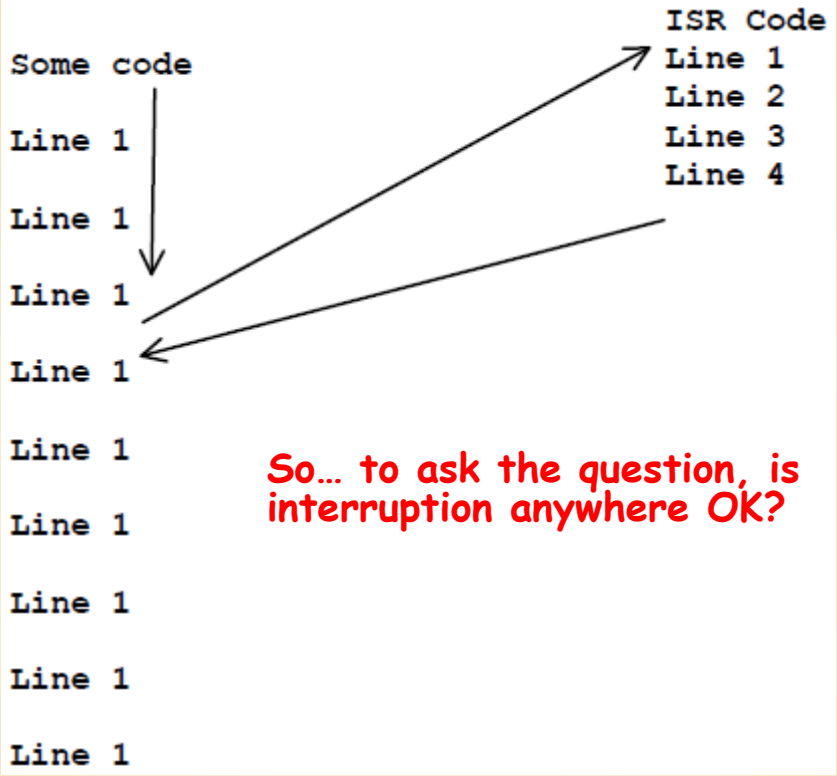
```
Main:
sei ; Enable Global Interrupts
```

  - Enable specific interrupts in control registers in main or elsewhere Ex: USART Receive Complete

```
in r16, UCSRB
ori r16, (1 << RXCIE)
out UCSRB, r16
```

http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=89843

Coding Interrupts in AVR Assembly



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

http://www.scriptoriumdesigns.com/embedded/interrupts.php

Interaction of read-modify-write code

- global declaration:
  - volatile required for any variables that ISRs share with other ISRs or standard code to prevent erroneous optimization by the compiler

```
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
reg16 stored to c (value 11)	

Interaction of code multiword code

- global declaration:

```
volatile uint16_t i;
```

- ISR code :

```
if (i==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

Multi-word access can represent a critical section of code

Atomic Resource Access

- Once you start using interrupts, you must be sensitive to code that is not OK to interrupt, and during execution of such code some interfering ISRs should be blocked.
- Alternatively, resources like the shared variables or hardware registers may be protected from multiple interfering access by using additional code to flag access to a shared resources allowing undisturbed access (ATOMIC access).
- The simplest way to guarantee ATOMIC access is to temporally disable interrupts.
- AVR gcc provides a macro `ATOMIC_BLOCK` to disable interrupts. See: http://www. nongnu.org/avr-libc/user-manual/group__util__atomic.html

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>
volatile uint16_t ctr;
ISR(TIMER1_OVF_vect) {
  ctr--;
}

...
int
main(void) {
  ...
  ctr = 0x200;
  start_timer();
  sei();
  uint16_t ctr_copy;
  do {
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
      ctr_copy = ctr;
    }
  }
  while (ctr_copy != 0);
  ...
}
```

- Pasted and modified from http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html#gaaaea265b31dabcfb3098bec7685c39e4

Multiple Pending Interrupts in AVR (Interrupt Priorities)

- If multiple interrupts requests are pending the order in which they are handled is system dependent. Some predefined have defined priorities based on the event number, while others allow the software to define the priorities.
- In the AVR, if multiple IRQ are pending the priority goes to lowest-addressed vector. Execution flow returns to main code, allowing at least one ASSEMBLY instruction to run before handling next IRQ.

New Interrupts During ISR

- What about new interrupts during an already executing ISR?
  - If interrupt service routines are (by default) themselves interrupted by higher priority interrupts or at all is also system-dependent. You must make yourself aware of how a given system handles interrupts.
- On the AVR, interrupts are globally disabled by default when an ISR is called until RETI is encountered.
- On the AVR, if a new interrupt request is pending during an ISR the one assembly instruction of foreground code is allowed to execute before the next ISR is called

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

- Sometimes, groups or individual interrupts can be enabled/disabled through a masking process using **interrupt mask registers**
- Interrupts can be disabled when:
  1) not needed or used
  2) critical section of code is running (can't be interrupted because of trimming or order of operations)

Clearing of the interrupt flag (IRQ)

- When an ISR is called, the corresponding interrupt flag must be cleared before the completion of the ISR (otherwise the ISR would called again).

- Depending on the system, the flag may be cleared
  1) automatically by hardware or
  2) it may be required that the software for an ISR must handle clearing the flag

- On the AVR, the flag is cleared automatically in hardware as soon as the ISR is called.
  - Unfortunately, this 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 on AVR

1) With interrupts enabled and foreground code running, an Interrupt Event occurs
2) A request is flagged by the hardware
3) Current Instruction Completed **(machine instruction, which is *NOT* same as a line of C code)**
4) Address of next instruction is stored on the stack
5) 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
6) Processor Executes ISR
7) 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)
8) 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)
9) reti is encountered (C macros take care of including this)
10) PC loaded from stack and Global Interrupt Enable Bit is Set
11) Foreground code execution continues

Interrupt Response Time

- 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 the AVR datasheet

Questions and considerations for ISR coding on a given system and project:

- 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.
- 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?

Timers/Counters:

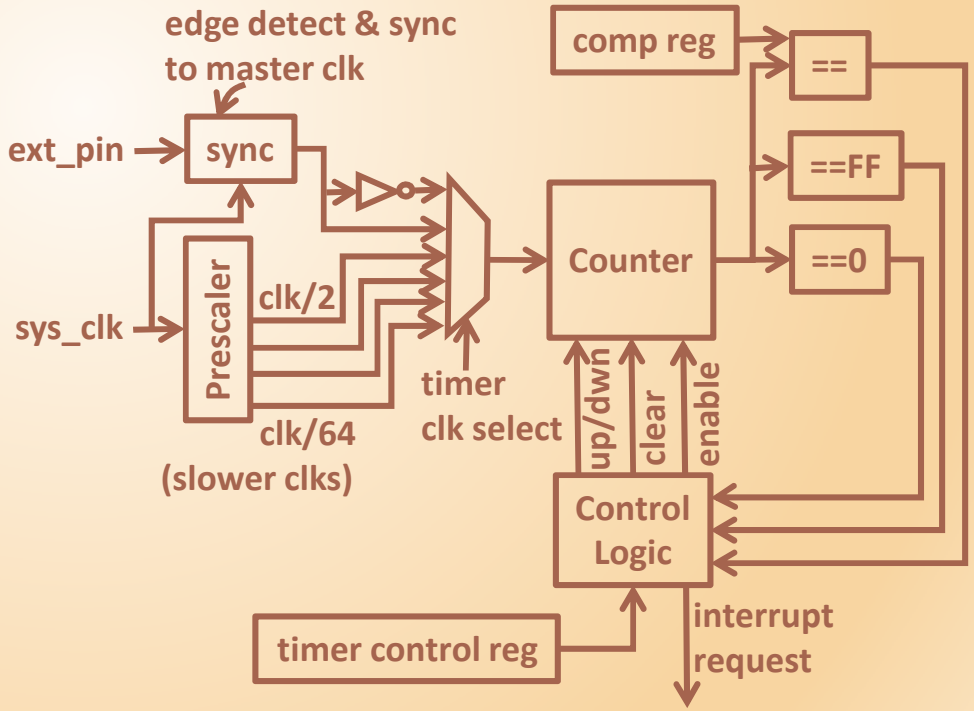
- Timers and counters are hardware provided on microcontrollers to handing timing or counting tasks or waveform generation. This may be needed be because the software is not fast or precise enough or just because it frees the microprocessor to perform other more useful work while simple timing-based tasks are handled by dedicated hardware. Typically, a microcontroller has multiple timers so it can handle multiple timing-based tasks.

- Reading Resource

http://www.scriptoriumdesigns.com/embedded/timers.php

TimersHardware:

- Basic Timer/ Counter Components:
  - Clock Prescaler (clk divider)
  - Clock Source Selection (internal clk, external clk, prescaler, etc..)
  - N-bit timer/counter itself (note: it is common for N to be larger than the native architecture word size)
  - Compare Registers (for hardware detection of specified counter value)
  - Capture Registers (for precise hardware capture of time of event)
  - Control Logic
  - Status/Control Registers to define behavior



The Core of a Timer is a Counter

- You may find the terms counter and timer used interchangeably, because the underlying hardware is the same.
- A timer is just a counter that increments or decrements with some fixed period.
- So, a counter the increments or decrements based on a clock is a timer
- If it increments based an irregular input event, it is not a timer and is acting as an event counter

Prescalar

- Consider the following scenarios:
  1) You would like to measure time for up to 2 seconds, with a sub 1-ms precision. How many bits of resolution are required?

  2) You have a an 8MHz clock available, you would like a timer that measures time up to 2 seconds. How large does the counter/timer register need to be?

     1) Ans: ceil(log2(NUMBER OF LEVELS)) = ceil(log2 (2/.001))

     2) Ans: ceil(log2(NUMBER OF CYCLES)) = ceil(log2(2*8e6))

- A clock prescalar modifies the rate at which the timer changes. This allows the timer/counter register to cover a larger range.
- A clock prescaler modifies the rate at which the timer or counter changes. This allows the timer/ counter to have a larger range of time or event counts.
- For a timer: log2(Range * clock freq.) is a lower bound on the number of counter bits required
- But you may not need to read all of them (lesser precision)
- Range AND Precision define the # of bits required
- #bits required = ceil(log2(range/precision))
- Prescaler values available may be, arbitrary values in a range, powers of two in some range, some predefined set (usually small powers of two).

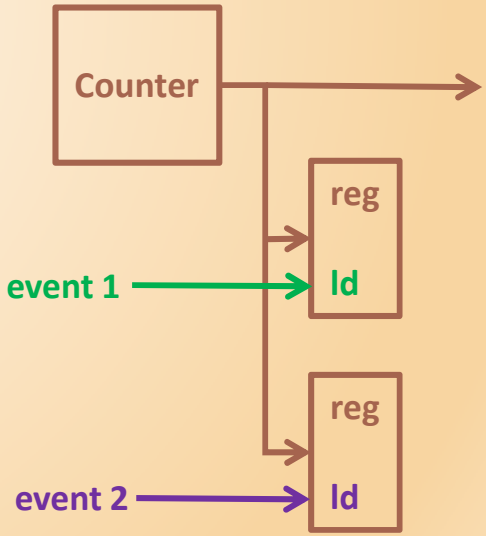
Capture Registers: High-Speed

- Consider this scenario: you are building a circuit to store the time an event occurred (indicated by a rising edge on a wire). How would you make the circuit?

- Now, do the same with a microcontroller (indicated by a rising edge on an external pin).

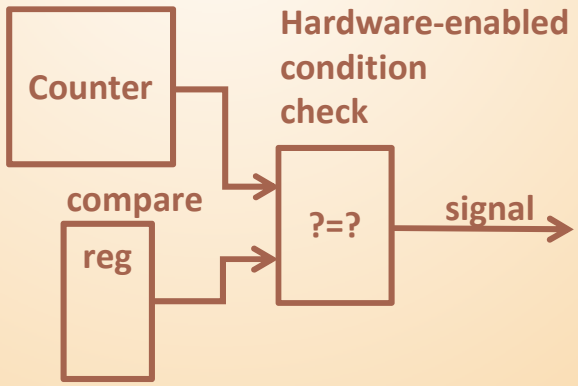
- Based on what you have learned how would you do this:
  - Software polling (poll pin than query and save timer/counter register)
    - Suffers delay in polling
      - Minimized by tight loop with processesor dedicated to detecting pin change
    - Delay in querying timer/counter register
  - Pin-Change Interrupt Approach
    - Frees the processor to do other work
    - Suffer delay for starting ISR and getting to query timer

- Dedicated hardware solution is available on some microcontrollers: a capture register. The output of the counter/timer register is connected to a storage register that is trigger to save by an event signal (typically an external pin). To allow processing of the results, the save can also trigger an ISR to process the saved value. Thus, delay is incurred in processing the data, but not in the saved result.
- If at least two capture registers are provided, precise intervals between two events can be captured.



Capture Registers: High-Speed

- Compare-Match Registers represent additional custom hardware.
- Dedicated hardware is provided to monitor the value of the timer/courter and signal some other event to occur
- Can trigger:
  - a flag to be set for software to detect through polling
  - an interrupt
  - a pin to change through hardware
  - A reset of the timer/counter register



Timer/Counter Polling

```
// poll for a value in a memory location
while ( COUNTER_REG != VALUE ); // wait for this location
                                // to contain specified value
```

- Note, there is a danger of missing the desired value for rapidly incrementing counters (such as with a fast timer) . Consider using COUNTER_REG >= VALUE and make sure the counter won't overflow before you check it (checking overflow flag can buy yourself more slack).

Timer/Counter Polling for Delay

- Get easily predictable accuracy and the range of the hardware timer
  1) Reset Timer
  2) Wait for value to be reached or exceeded

Timer/Counter Polling for Delay with other stuff to do:

- Useful when response time doesn't need to be precise and other work must be done while waiting. Check and response time limited by maximum time needed for other stuff
  1) Reset Timer
  2) Do other stuff (may not need the same stuff every time)
  3) Check if value is be reached or exceeded

Polling for a Timer Flag

- In the previous slide, there are cases where the checks can miss values. For instance, consider a 16-bit counter where we are trying to detect 2^16-1. If we miss the value, the next value is 0 and our comparison will fail. Using flags which must be explicitly cleared avoid this problem:

- Overflow Flag:
  - Clear Overflow Flag
  - Reset Timer
  - Wait for Flag to be set (and maybe do other stuff)

- Compare-Match Flag
  - Set Compare Register Value
  - Clear Compare-Match Flag
  - Reset Timer
  - Wait for Flag to be Set (and maybe do other stuff)

- Don't forget that the flags must be cleared before use and must be re-cleared to detect subsequent events.

Timer adl:

- A timer can be set up to regularly do one or more of the following:
  1) trigger an interrupt,
  2) cause the hardware to make some pin value change,
  3) reset the timer itself
- Upon one or more of the following:
  1) Overflow
  2) Compare-match events

- This hardware-based ISR triggering and self-resetting allows for precise timing and very **regular triggering** of ISR calls without relying on any software timing or intermittent and often **irregular polling** (software doesn't always take the same time in every loop iteration if conditional statements exist). Note that when an interrupt is enabled the corresponding flag is automatically cleared in hardware.

- Automatic Pin Changes allow for precise and flexible digital waveform generation.

Getting the desired timing

- Getting the frequency you want can require some thought.
  - Resulting frequency = input clock /prescaler / (max count+1)
    - Max count may be defined by a compare register or the maximum value of the counter (2^N-1)
    - Clock division = prescaler * (max count+1)
    - You must choose the prescaler and max count to get the clock scaling you want, but remember that prescaler is typically limited to a few select values (such as powers of two)

- What if your input (system) clock is 1Mhz and you need a 3 kHz period from an 8-bit timer? Assume your prescalers can divide by powers of two.
  - Generally, to get the most precision, you should choose the smallest prescaler possible as use the compare register for the rest, unless the division factor is a factor of two.
  - Here, we want division factor of:
    1M/3k = 333.3333….
  - Simply setting the compare register to 333 can't be done (8 bits only allow for 255)
  - So, we can set the prescaler to 2 and set the compare register to:
    (333.33../2) = 167 (-1 for the register entry)
  - In this case, we get a resulting period of:
    (1M/167)/2=2.994 kHz
  - Alternatively prescaler 4 and a max count of 83 (-1 for the register entry) gives:
    (1M/83)/4 = 3.012 kHz
  - Though the prescaler of 4 has an error that is larger, choosing which error is better (small freq. overshoot or larger freq. undershoot) is application-dependent.

Simultaneous Timers

- If multiple irregular and perhaps simultaneous timing events are needed, here is one approach.
- "leapfrogging" http://www.scriptoriumdesigns.com/embedded/timers.php moves the compare value in software instead of resetting the counter in hardware.
- This can be useful if multiple and/or irregular timers are to be implemented with a limited set of hardware timers. For example if the total time is to be preserved:
  1) Set compare register for time of first trigger event and setup ISR
  2) Reset and start counter
  3) Upon first call of ISR, decide timer/counter value for next event and set compare match register
  4) Repeat