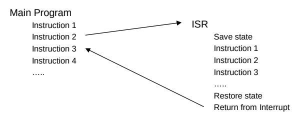
Interrupts

- Fundamental concept in computation
- Interrupt execution of a program to “handle” an event
  - Don’t have to rely on program relinquishing control
  - Can code program without worrying about others
- Issues
  - What can interrupt and when?
  - Where is the code that knows what to do?
  - How long does it take to handle interruption?
  - Can an interruption be, in turn, interrupted?
  - How does the interrupt handling code communicate its results?
  - How is data shared between interrupt handlers and programs?

What is an Interrupt?

- Reaction to something in I/O (human, comm link)
- Usually asynchronous to processor activities
- “interrupt handler” or “interrupt service routine” (ISR) invoked to take care of condition causing interrupt
  - Change value of internal variable (count)
  - Read a data value (sensor, receive)
  - Write a data value (actuator, send)



Interrupts

- Code sample that does not interrupt

```
char SPI_SlaveReceive(void)
{
/* Wait for reception complete */
while(!(SPSR & (1<<SPIF)))
;
/* Return data register */
return SPDR;
}
```

- Instead of busy waiting until a byte is received the processor can generate an interrupt when it sets SPIF

```
SIGNAL(SIG_SPI) {
    RX_Byte = SPDR
}
```

Saving and Restoring Context

- Processor and compiler dependent
- Where to find ISR code?
  - Different interrupts have separate ISRs
- Who does dispatching?
  - Direct
    - Different address for each interrupt type
    - Supported directly by processor architecture
  - Indirect
    - One top-level ISR
    - Switch statement on interrupt type
  - A mix of these two extremes?

Saving and Restoring Context

- How much context to save?
  - Registers, flags, program counter, etc.
  - Save all or part?
  - Agreement needed between ISR and program
- Where should it be saved?
  - Stack, special memory locations, shadow registers, etc.
  - How much room will be needed on the stack?
  - Nested interrupts may make stack reach its limit – what then?
- Restore context when ISR completes

Ignoring Interrupts

- Can interrupts be ignored?
  - It depends on the cause of the interrupt
  - No, for nuclear power plant temperature warning
  - Yes, for keypad on cell phone (human timescale is long)
- When servicing another interrupt
  - Ignore others until done
  - Can’t take too long – keep ISRs as short as possible
    - Just do a quick count, or read, or write – not a long computation
- Interrupt disabling
  - Will ignored interrupt “stick”?
    - Rising edge sets a flip-flop
  - Or will it be gone when you get to it?
    - Level changes again and its as if it never happened
  - Don’t forget to re-enable

Prioritizing Interrupts

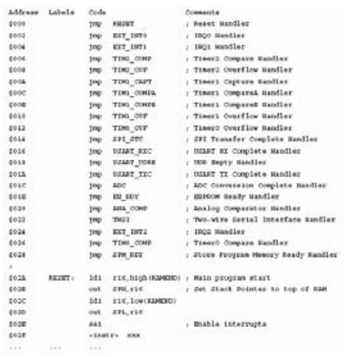
- When multiple interrupts happen simultaneously
  - Which is serviced first?
  - Fixed or flexible priority?
- Priority interrupts
  - Higher priority can interrupt
  - Lower priority can't
- Maskable interrupts
  - "don't bother me with that right now"
  - Not all interrupts are maskable, some are non-maskable

Interrupts in the ATmega16

- External interrupts
  - From I/O pins of microcontroller
- Internal interrupts
  - Timers
    - Output compare
    - Input capture
    - Overflow
  - Communication units
    - Receiving something
    - Done sending
  - ADC
    - Completed conversion

Interrupt Jump Vector Table

- Fixed location in memory to find first instruction for each type of interrupt
- Only room for one instruction
  - JMP to location of complete ISR



```
Address  Labels  Code                       Comments
$000             jmp   RESET                ; Reset Handler
$002             jmp   EXT_INT0             ; IRQ0 Handler
$004             jmp   EXT_INT1             ; IRQ1 Handler
$006             jmp   TIM2_COMP            ; Timer2 Compare Handler
$008             jmp   TIM2_OVF             ; Timer2 Overflow Handler
$00A             jmp   TIM1_CAPT            ; Timer1 Capture Handler
$00C             jmp   TIM1_COMPA           ; Timer1 CompareA Handler
$00E             jmp   TIM1_COMPB           ; Timer1 CompareB Handler
$010             jmp   TIM1_OVF             ; Timer1 Overflow Handler
$012             jmp   TIM0_OVF             ; Timer0 Overflow Handler
$014             jmp   SPI_STC              ; SPI Transfer Complete Handler
$016             jmp   USART_RXC            ; USART RX Complete Handler
$018             jmp   USART_UDRE           ; UDR Empty Handler
$01A             jmp   USART_TXC            ; USART TX Complete Handler
$01C             jmp   ADC                  ; ADC Conversion Complete Handler
$01E             jmp   EE_RDY               ; EEPROM Ready Handler
$020             jmp   ANA_COMP             ; Analog Comparator Handler
$022             jmp   TWSI                 ; Two-wire Serial Interface Handler
$024             jmp   EXT_INT2             ; IRQ2 Handler
$026             jmp   TIM0_COMP            ; Timer0 Compare Handler
$028             jmp   SPM_RDY              ; Store Program Memory Ready Handler
;
$02A     RESET:  ldi   r16,high(RAMEND)     ; Main program start
$02B             out   SPH,r16              ; Set Stack Pointer to top of RAM
$02C             ldi   r16,low(RAMEND)
$02D             out   SPL,r16
$02E             sei                        ; Enable interrupts
$02F             <instr>  xxx
...      ...     ...
```

Chain of Events on Interrupt

- Finish executing current instruction
- Disable all interrupts CLI
- Push program counter on to stack
- Jump to interrupt vector table
- Jump to start of complete ISR
- Save any context that ISR may otherwise change
  - Registers and flags must be saved within ISR and restored before it returns – **<u>this is very important!</u>**
- Re-enable interrupts if nested interrupts are ok
- Complete ISR's code
- Re-enable interrupts upon return
- Jump back to next instruction before interruption

Automatic

Compiler

SEI

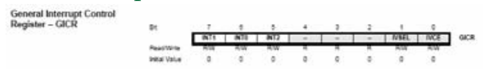
RETI

Shared Data Problem

- When you use interrupts you create the opportunity for multiple sections of code to update a variable.
- This might cause a problems in your logic if an interrupt updates a variable between two lines of code that are directly dependent on each other (e.g. if statement)
- One solution is to create critical sections where you disable the interrupts for a short period of time while you complete your logic on the shared variable

```
cli();
…..critical section code goes here…..
sei();
```

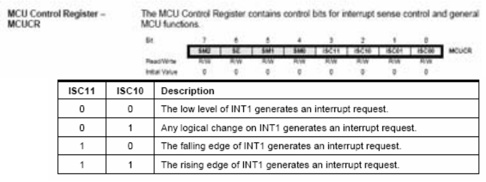
External Interrupts



General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Special pins: INT0, INT1, INT2
  - Can interrupt on edge or level
- Can interrupt even if set to be output pins
  - Implements "software interrupts" by setting output



MCU Control Register – MCUCR

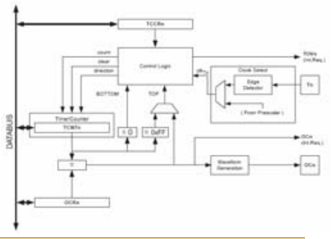
The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
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	

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.

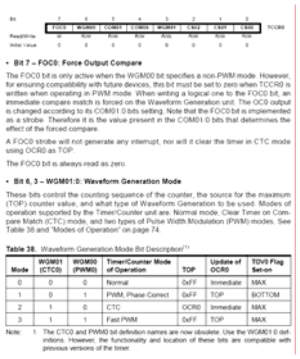
Closer Look at a Timer/Counter

- Timer0/Counter0
  - Clear timer on compare match (auto reload)
  - Prescaler (divide clock by up to 1024)
  - Overflow and compare match interrupts
  - Registers
    - Configuration
    - Count value
    - Output compare value



Timer/Counter Registers

- Timer/Counter Control Register TCCR0



Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	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 7 – FOC0: Force Output Compare**

The FOC0 bit is only active when the WGM00 bit specifies a non-PWM mode. However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0 is written when operating in PWM mode. When writing a logical one to the FOC0 bit, an immediate compare match is forced on the Waveform Generation unit. The OC0 output is changed according to its COM01:0 bits setting. Note that the FOC0 bit is implemented as a strobe. Therefore it is the value present in the COM01:0 bits that determines the effect of the forced compare.

A FOC0 strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR0 as TOP.

The FOC0 bit is always read as zero.

- **Bit 6, 3 – WGM01:0: Waveform Generation Mode**

These bits control the counting sequence of the counter, the source for the maximum (TOP) counter value, and what type of Waveform Generation to be used. Modes of operation supported by the Timer/Counter unit are: Normal mode, Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes. See Table 38 and "Modes of Operation" on page 74.

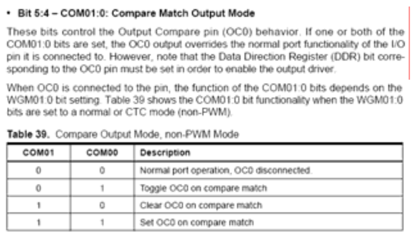
Table 38. Waveform Generation Mode Bit Description[1]

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of Operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

Note: 1. The CTC0 and PWM0 bit definition names are now obsolete. Use the WGM01:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Timer/Counter Registers (cont'd)

- Timer/Counter Control Register TCCR0



- **Bit 5:4 – COM01:0: Compare Match Output Mode**

These bits control the Output Compare pin (OC0) behavior. If one or both of the COM01:0 bits are set, the OC0 output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0 pin must be set in order to enable the output driver.

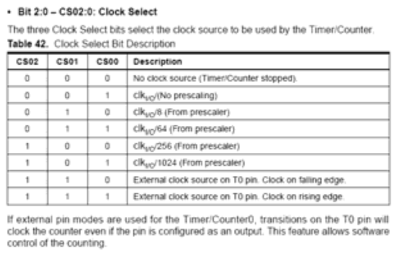
When OC0 is connected to the pin, the function of the COM01:0 bits depends on the WGM01:0 bit setting. Table 39 shows the COM01:0 bit functionality when the WGM01:0 bits are set to a normal or CTC mode (non-PWM).

Table 39. Compare Output Mode, non-PWM Mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Timer/Counter Registers (cont'd)

- Timer/Counter Control Register TCCR0



- **Bit 2:0 – CS02:0: Clock Select**

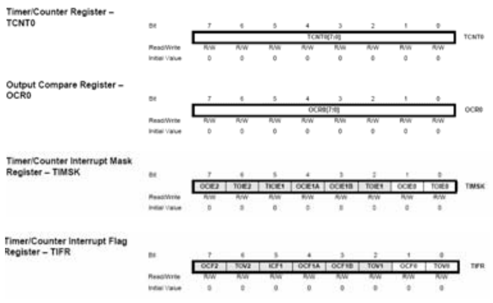
The three Clock Select bits select the clock source to be used by the Timer/Counter.

Table 42. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Timer/Counter Registers (cont'd)



Timer/Counter Register – TCNT0

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
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	

Output Compare Register – OCR0

Bit	7	6	5	4	3	2	1	0	
	OCR0[7:0]								OCR0
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	

Timer/Counter Interrupt Mask Register – TIMSK

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	

Timer/Counter Interrupt Flag Register – TIFR

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	

Setting Register Values

- Defined names for each register and bit
  - Set timer to clear on match
  - Set prescaler to 1024

```
TCCR0 = (1<<WGM01) | (1<<CS02) | (1<<CS00);
```

  - Set count value to compare against

```
OCR0 = 150;
```

  - Set timer to interrupt when it reaches count

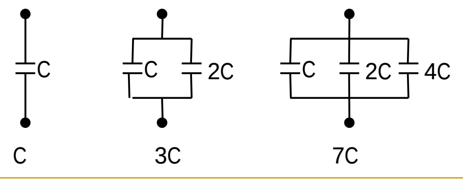
```
TIMSK = (1<<OCIE0);
```

Writing an Interrupt Handler in C

- Set and clear interrupt enable
  - sei();
  - cli();
- Interrupt handler
  - SIGNAL(SIG_OUTPUT_COMPARE0)
```
{
        i++;
}
```
- Setting I/O registers
  - TCCR0 = (1<<WGM01) | (1<<CS02) | (1<<CS00);
- Enabling specific interrupts
  - TIMSK = (1<<OCIE0);

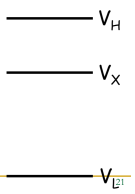
Analog to digital conversion

- Use charge-redistribution technique
  - no sample and hold circuitry needed
  - even with perfect circuits quantization error occurs
- Basic capacitors
  - sum parallel capacitance



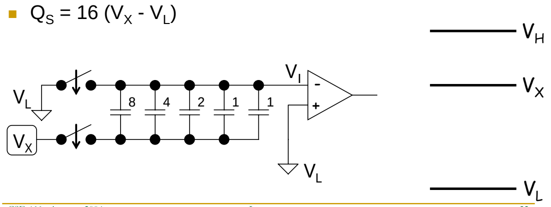
Analog to digital conversion (cont'd)

- Two reference voltage
  - mark bottom and top end of range of analog values that can be converted (V_L and V_H)
  - voltage to convert must be within these bounds (V_X)
- Successive approximation
  - most approaches to A/D conversion are based on this
  - 8 to 16 bits of accuracy
- Approach
  - sample value
  - hold it so it doesn't change
  - successively approximate
  - report closest match



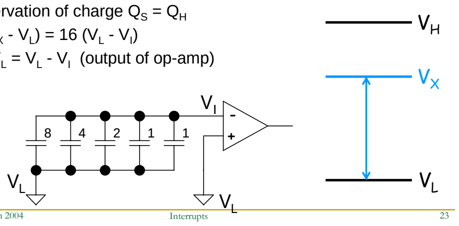
A-to-D – sample

- During the sample time the top plate of all capacitors is switched to reference low V_L
- Bottom plate is set to unknown analog input V_X
- Q = CV
- $Q_S = 16 (V_X - V_L)$



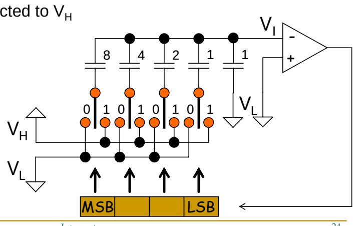
A-to-D – hold

- Hold state using logically controlled analog switches
  - Top plates disconnected from V_L
  - Bottom plates switched from V_X to V_L
- $Q_H = 16 (V_L - V_I)$
  - conservation of charge $Q_S = Q_H$
  - $16 (V_X - V_L) = 16 (V_L - V_I)$
  - $V_X - V_L = V_L - V_I$ (output of op-amp)



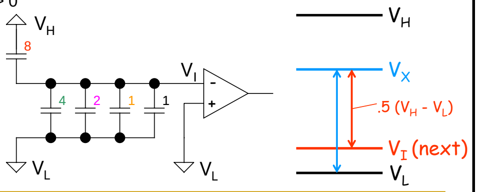
A-to-D – successive approximation

- Each capacitor successively switched from V_L to V_H
  - Largest capacitor corresponds to MSB
- Output of comparator determines bottom plate voltage of cap
  - > 0 : remain connected to V_H
  - < 0 : return to V_L



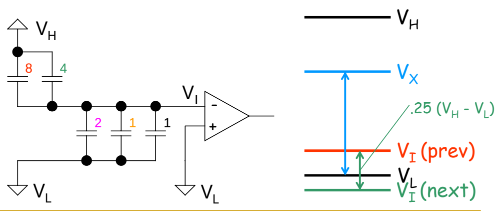
A-to-D example - MSB

- Suppose $V_X = 21/32\ (V_H - V_L)$ and already sampled
- Compare after shifting half of capacitance to V_H
  - V_I goes up by $+ 8/16\ (V_H - V_I) - 8/16\ (V_L - V_I) = + 8/16\ (V_H - V_L)$
  - original $V_L - V_I$ goes down and becomes
  - $V_L - (V_I + .5\ (V_H - V_L)) = V_L - V_I - .5\ (V_H - V_L)$
- Output > 0



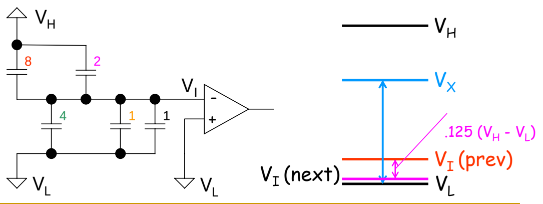
A-to-D example - (MSB-1)

- Compare after shifting another part of cap. to V_H
  - V_I goes up by $+ 4/16\ (V_H - V_I) - 4/16\ (V_L - V_I) = + 4/16\ (V_H - V_L)$
  - original $V_L - V_I$ goes down and becomes
  - $V_L - (V_I + .25\ (V_H - V_L)) = V_L - V_I - .25\ (V_H - V_L)$
- Output < 0 (went too far)



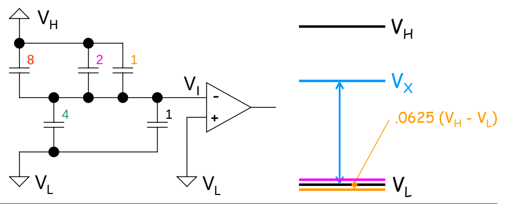
A-to-D example - (MSB-2)

- Compare after shifting another part of cap. to V_H
  - V_I goes up by $+ 2/16\ (V_H - V_I) - 2/16\ (V_L - V_I) = + 2/16\ (V_H - V_L)$
  - original $V_L - V_I$ goes down and becomes
  - $V_L - (V_I + .125\ (V_H - V_L)) = V_L - V_I - .125\ (V_H - V_L)$
- Output > 0



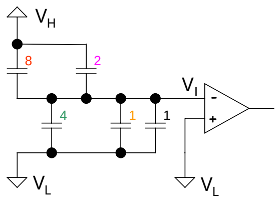
A-to-D example - LSB

- Compare after shifting another part of cap. to V_H
  - V_I goes up by $+ 1/16\ (V_H - V_I) - 1/16\ (V_L - V_I) = + 1/16\ (V_H - V_L)$
  - original $V_L - V_I$ goes down and becomes
  - $V_L - (V_I + .0625\ (V_H - V_L)) = V_L - V_I - .0625\ (V_H - V_L)$
- Output < 0 (went too far again)

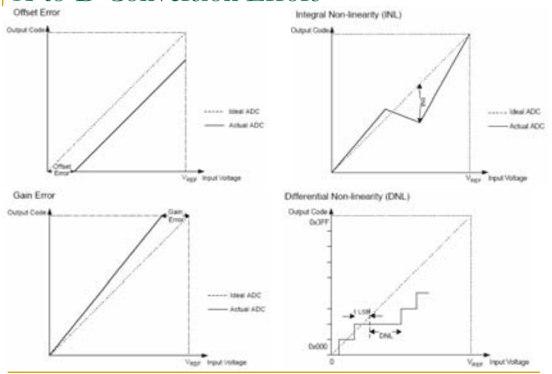


A-to-D example final result

- Input sample of 21/32
- Gives result of <u>1010</u> or 10/16 = 20/32
- 3% error

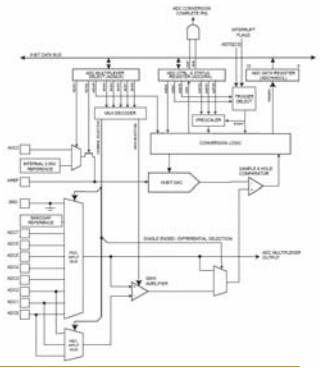


A-to-D Conversion Errors



Closer Look at A-to-D Conversion

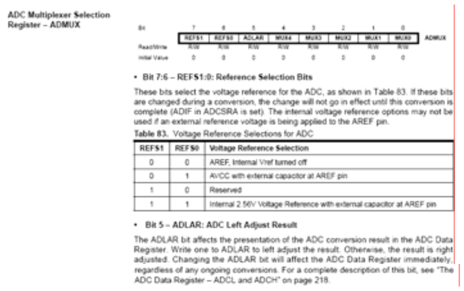
- Needs a comparator and a D-to-A converter
- Takes time to do successive approximation
- Interrupt generated when conversion is completed



A-to-D Conversion on the ATmega16

- 10-bit resolution (adjusted to 8 bits as needed)
- 65-260 usec conversion time
- 8 multiplexed input channels
- Capability to do differential conversion
  - Difference of two pins
  - Optional gain on differential signal (amplifies difference)
- Interrupt on completion of A-to-D conversion
- 0-V_{CC} input range
- 2*LSB accuracy (2 * 1/1024 = ~0.2%)
  - Susceptible to noise – special analog supply pin (AVCC) and capacitor connection for reference voltage (AREF)

A-to-D Conversion (cont’d)



ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
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 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 83. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 83. Voltage Reference Selections for ADC

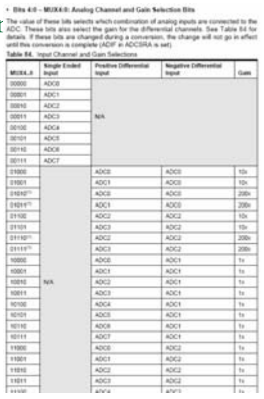
REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions. For a complete description of this bit, see “The ADC Data Register – ADCL and ADCH” on page 218.

A-to-D Conversion (cor

- Single-ended or differential
  - 1 of 8 single-ended
  - ADCx – ADC1 at 1x gain
  - ADC{0,1} – ADC0 at 10x
  - ADC{0,1} – ADC0 at 200x
  - ADC{2,3} – ADC2 at 10x
  - ADC{2,3} – ADC3 at 200x
  - ADC{0,1,2,3,4,5} – ADC2 at 1x



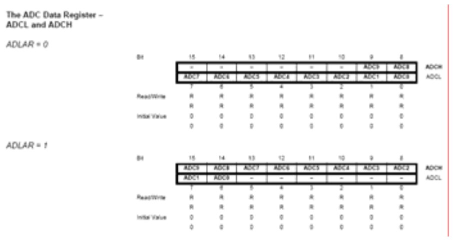
- **Bits 4:0 – MUX4:0: Analog Channel and Gain Selection Bits**

The value of these bits selects which combination of analog inputs are connected to the ADC. These bits also select the gain for the differential channels. See Table 84 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

Table 84. Input Channel and Gain Selections

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010[1]		ADC0	ADC0	200x
01011[1]		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110[1]		ADC2	ADC2	200x
01111[1]		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x

A-to-D Conversion (cont’d)



The ADC Data Register – ADCL and ADCH

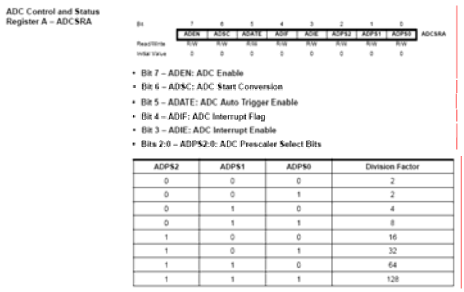
ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

A-to-D Conversion (cont’d)



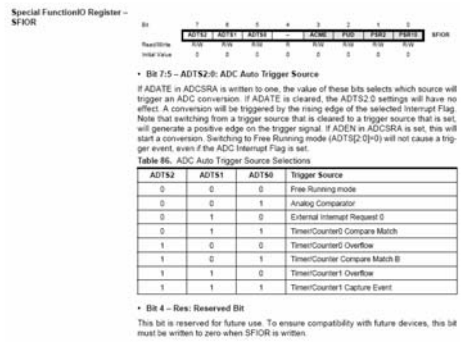
ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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 7 – ADEN: ADC Enable**
- **Bit 6 – ADSC: ADC Start Conversion**
- **Bit 5 – ADATE: ADC Auto Trigger Enable**
- **Bit 4 – ADIF: ADC Interrupt Flag**
- **Bit 3 – ADIE: ADC Interrupt Enable**
- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

A-to-D Conversion (cont'd)



Special FunctionIO Register – SFIOR

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:5 – ADTS2:0: ADC Auto Trigger Source**

If ADATE in ADCSRA is written to one, the value of these bits selects which source will trigger an ADC conversion. If ADATE is cleared, the ADTS2:0 settings will have no effect. A conversion will be triggered by the rising edge of the selected Interrupt Flag. Note that switching from a trigger source that is cleared to a trigger source that is set, will generate a positive edge on the trigger signal. If ADEN in ADCSRA is set, this will start a conversion. Switching to Free Running mode (ADTS[2:0]=0) will not cause a trigger event, even if the ADC Interrupt Flag is set.

Table 86. ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

- **Bit 4 – Res: Reserved Bit**

This bit is reserved for future use. To ensure compatibility with future devices, this bit must be written to zero when SFIOR is written.

A-to-D Conversion (cont'd)

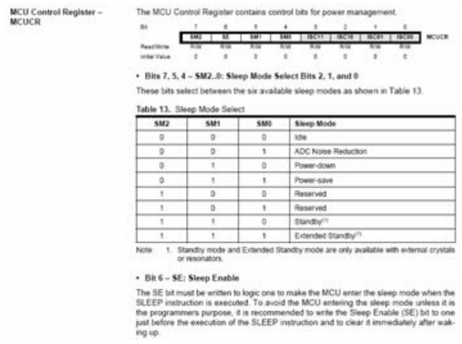
Writing an Interrupt Handler in C (again)

- Ensure main program sets up all registers
- Enable interrupts as needed
- Enable global interrupts (SEI)
- Write handler routine for each enabled interrupt
  - What if an interrupt occurs and a handler isn't defined?
- Make sure routine does not disrupt others
  - Data sharing problem
  - Save any state that might be changed (done by compiler)
- Re-enable interrupts upon return
  - done by compiler with RETI

Power modes

- Processor can go to "sleep" and save power
- Different modes put different sets of modules to sleep
  - Which one to use depends on which modules are needed to wake up processor
  - Timers, external interrupts, ADC, serial communication lines, etc.
- set_sleep_mode (mode);
- sleep_mode ();

Power modes (cont'd)



MCU Control Register – MCUCR

The MCU Control Register contains control bits for power management.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
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	

- **Bits 7, 5, 4 – SM2..0: Sleep Mode Select Bits 2, 1, and 0**

These bits select between the six available sleep modes as shown in Table 13.

Table 13. Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby(1)
1	1	1	Extended Standby(1)

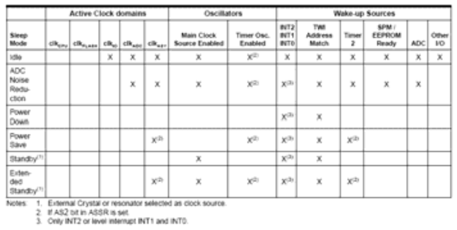
Note: 1. Standby mode and Extended Standby mode are only available with external crystals or resonators.

- **Bit 6 – SE: Sleep Enable**

The SE bit must be written to logic one to make the MCU enter the sleep mode when the SLEEP instruction is executed. To avoid the MCU entering the sleep mode unless it is the programmers purpose, it is recommended to write the Sleep Enable (SE) bit to one just before the execution of the SLEEP instruction and to clear it immediately after waking up.

Power modes (cont'd)

- Wake up sources and active clocks



	Active Clock domains					Oscillators		Wake-up Sources					
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Osc. Enabled	INT2 INT1 INT0	TWI Address Match	Timer 2	SPM / EEPROM Ready	ADC	Other I/O
Idle			X	X	X	X	X(2)	X	X	X	X	X	X
ADC Noise Reduction				X	X	X	X(2)	X(3)	X	X	X	X	
Power Down								X(3)	X				
Power Save					X(2)		X(2)	X(3)	X	X(2)			
Standby(1)						X		X(3)	X				
Extended Standby(1)					X(2)	X	X(2)	X(3)	X	X(2)			

Notes: 1. External Crystal or resonator selected as clock source.
2. If AS2 bit in ASSR is set.
3. Only INT2 or level interrupt INT1 and INT0.