AVR-specific methods and C (part 2) Video Lecture

ENGN3213/6213
Digital Systems and Microprocessors

What is in this lecture

- Some remarks about compilers
- Timers
- Analog to digital conversion
- AVR ATmega8 analog to digital conversion specifics

Resources

- Chapter 2 of the book:
  Barnett, O'Cull, Cox, *Embedded C Programming and the Atmel AVR, 2nd Edition*, Thomson Delmar 2007
  - There is one copy of the book at Hancock library (consultation only)
  - Chapter 2 is on Wattle

- The ATmega8 data sheet (on Wattle)

Notes about compilers

- The textbook chapter I have provided you provides coding rules and examples written in C for the CodeVisionAVR compiler.
- Atmel Studio uses a different compiler: AVR-GCC
- The C language standard, which defines the so called ANSI C, (ANSI = American National Standards Institute), does not extend to the microcontroller-specific routines, so different compilers require different keywords and syntax.
- Some examples shown in the book *will not work* in the lab unless you make some changes.

Notes about compilers (2)

- Some notable coding style differences between CodeVisionAVR and AVR-GCC:

CodeVisionAVR	**AVR-GCC**
Interrupt service routine declarations	
`interrupt [ ] void my_isr()`	`ISR( )`
Interrupt vector names, e.g.,	
`EXT_INT0`	`INT0_vect`
Enable global interrupt bit (assembly SEI)	
`#asm("sei")`	`asm("sei");`
Bit-access to I/O port	
`PORTB.0=1;`	`PORTB=0x01;`

- So if you find that something you tried does not work, this may be why. You will get some practice in the lab.

Timer/Counters

- Very commonly used, versatile peripherals
- They are simple up-counters:
  - When operating as *counters,* up-counting is triggered by external events
  - When operating as *timers,* up-counting is dictated by the clock
- The rollover of counters is associated with an interrupt
- The ATmega8 has two 8-bit timers (Timer0 and Timer2) and one 16-bit timer (Timer1)
- Key configurable aspects of the timer/counters are:
  - Input selection
  - Timer prescaler
- Configuration is done via the Clock Select (CSx) bits of the Timer Counter Control Register (TCCRx)

Timer 0

- An 8-bit counter (counts up to 256)
- Normally used to generate precise timing (*tick*) for a program. A tick is usually generated by:
  - Setting the counter to a meaningful start value
  - Counting up and picking up an event as an interrupt when the counter overflows

- TCCR0 configuration:

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Timer 0 (2)

- 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.

- The slowest tick that can be achieved on a 1MHz clock is approximately 10^{-6}*1024*256 = 262 ms. If slower ticks are needed interrupt-routine-driven global variables can be used
- The current count can be set/read from 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	

Timer 0 (3)

```
static unsigned int time_count;    //0.5sec counter

ISR(TIMER0_OVF_vect)
{
    TCNT0 = 207; //each count to 256 takes 50ms
    time_count++;
    if (time_count==10) {
       PORTC = (PORTC ^ 0x01); //switch logic level on pin
       time_count = 0;
    }
}

void main ()
{
    DDRC = 0x01;    //set lsb of port C for output
    TCCR0 = 0x05;   //set prescaler to clk/1024
    TCNT0=0;        //set count to 0

    TIMSK=0x01;     //unmask timer0 overflow interrupt

    asm("sei");     ;//enable interrupts

    while(1)
       ;            //do nothing
}
```

Timer 1

- A 16-bit timer. Has additional features, in particular:
  - ***Input capture***, generally used for measuring pulse widths of digital inputs or capturing times
  - ***Output compare***, generally used to produce frequencies or pulses at one of the microcontroller's output pins
- Timer 1 has the following registers:
  - A count register TCNT1 (16-bit: TCNT1H + TCNT1L)
  - 2 output compare register (16-bit each: OCR1A and OCR1B)
  - An input Capture Register ICR1 (16-bit: ICR1H + ICR1L)
  - A control register TCCR1 (16-bit: TCCR1A + TCCR1B):
    - TCCR1A controls the output compare mode
    - TCCR1B controls the input capture and the prescaler

Timer 1 – Input capture

- Description of TCCR1B (*Input capture* settings)

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

  - ICNC1: Input capture noise canceler (1 to enable)
  - ICES1: Input capture edge select (1 for rising edge, 0 for falling edge)
  - WGM13/12: Output waveform control. (more in TCCR1A)
  - CSx: Clock select bits. They are the same as for Timer 0

- *Input capture mode*
  - Rather than starting/stopping a timer (as in Timer 0), in input capture mode, counts are assigned to events by a "free running" counter. The difference between counts gives the duration of an event

Timer 1 – Input capture (2)

- The Input capture register (ICR1) captures the count of Timer 1 when a required signal is asserted at the input capture pin (ICP1).
- When a new value is captured, an interrupt is generated, allowing a routine to deal with the newly stored value (so that it is not overwritten by a further capture event). That same routine can also interpret meaningfully interpret capture events.
- Timer 1 has also a noise cancelling function. If active, an active level change is recognised only I persistent for at least 4 clock cycles (prevents short-lived noise spikes from initiating a capture events)
- The next slide shows an example of C code to measure the period of a square wave read from pin ICP1 and output the number as a word output on port C.

Timer 1 – Input capture (3)

```
global char ov_counter //counter for amt of times overflowed
global unsigned int start_edge, end_edge //storage for times
global unsigned long int ticks

ISR(timer1_ovf_vec) {
    ov_counter++;
}

ISR(timer1_capt_vec) {
    end_edge = 256*ICR1H + ICR1L; //store end time
    ticks = (unsigned long)end_edge + (unsigned long)ov_counter*65536
             - (unsigned long)start_edge;  //note typecasting
    PORTC = (ticks / 125); // 125 ticks to a ms at CK/8 prescaling
    start_edge = end_edge;
    ov_counter=0;
}

void main (void) {
    DDRC = 0xFF; //set port C to all-bit output
    TCCR1A = 0x00; //disable waveform functions
    TCCR1B = 0xc2; //prescaler set to CK/8, enable input capture
    TIMSK = 0x24;  //unmask timer 1 overflow and capture interrupts
    asm("sei");
    while(1) ; //do nothing
}
```

Timer 1 – Output compare

- In output compare mode, a pre-defined value is stored in the output compare register and when the value of the timer matches that in the register, an interrupt is generated.
- The comparison operation can also be used to toggle an output pin on the processor, according to the settings of register TCCR1A:

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	

Effect of COM1xy settings on oputput of pins OC1A/OC1B upon compare match

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on Compare Match
1	0	Clear OC1A/OC1B on Compare Match (Set output to low level)
1	1	Set OC1A/OC1B on Compare Match (Set output to high level)

Timer 1 – Output compare (2)

- C code example to generate a 1/234 clock division:

```
ISR (timer1_compa_vect)
{
   OCR1A=OCR1A+117; //set next comparison point
}


void main (void)
{
   DDRB = 0x02; //set OC1A bit for output (same as PORTB.1)

   TCCR1A = 0x40; //enable toggle OC1A pin on match
   TCCR1B = 0x00; //prescaler not set (CK)

   TIMSK = 0x10;  //unmask timer compare interrupts for register A

   asm("sei");

   while(1)
      ;       //do nothing
}
```

Timer 1 – PWM mode

- The output compare mode can be used to generate square waves with varying duty cycles. A device that can generate square waves of rapidly varying duty cycle is called a Pulse Width Modulator (PWM).
- A (digital) PWM output can be filtered with a low pass filter to generate an output of varying average amplitude (a simple form of Digital-to-Analog Conversion)
- A PWM wave could be created with minor changes to the last code example. There is, however, a dedicated PWM mode for timer 1:
  - In PWM mode, the counter counts up to a *top value* called the *PWM resolution* and then back to 0. This determines the frequency of the PWM wave.
  - The value in the output compare register is still used to toggle the OC pin(s). Depending on where this value is set, one can achieve varying duty cycles as desired.

Timer 1 – PWM mode (2)

- The configuration of the PWM mode is done through the bits WGM10-13 of register TCCR1A. There are several possible operating modes. Refer to the data sheet for further details (from page 88 onwards).

```
void main (void)
{
   DDRB = 0x02; //set OC1A bit for output (same as PORTB.1)

   TCCR1A = 0x91; //set for non-inverting PWM mode and 8-bit resolution
   TCCR1B = 0x02; //prescaler at CK/8

   TIMSK = 0x10;  //unmask timer compare interrupts for register A

   OCR1A = 25; //10% duty cycle desired

   while(1)
      ;       //do nothing
}
```

Timer 2

- Timer 2 is an 8-bit counter with output compare capability and PWM features similar to Timer 1.
- Timer 2 can operate on the basis of an external clock source (can run asynchronously of the system clock if required).
- Configuration is done via the TCCR2 register:

Bit	7	6	5	4	3	2	1	0	
	FOC2	**WGM20**	**COM21**	**COM20**	**WGM21**	**CS22**	**CS21**	**CS20**	**TCCR2**
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	

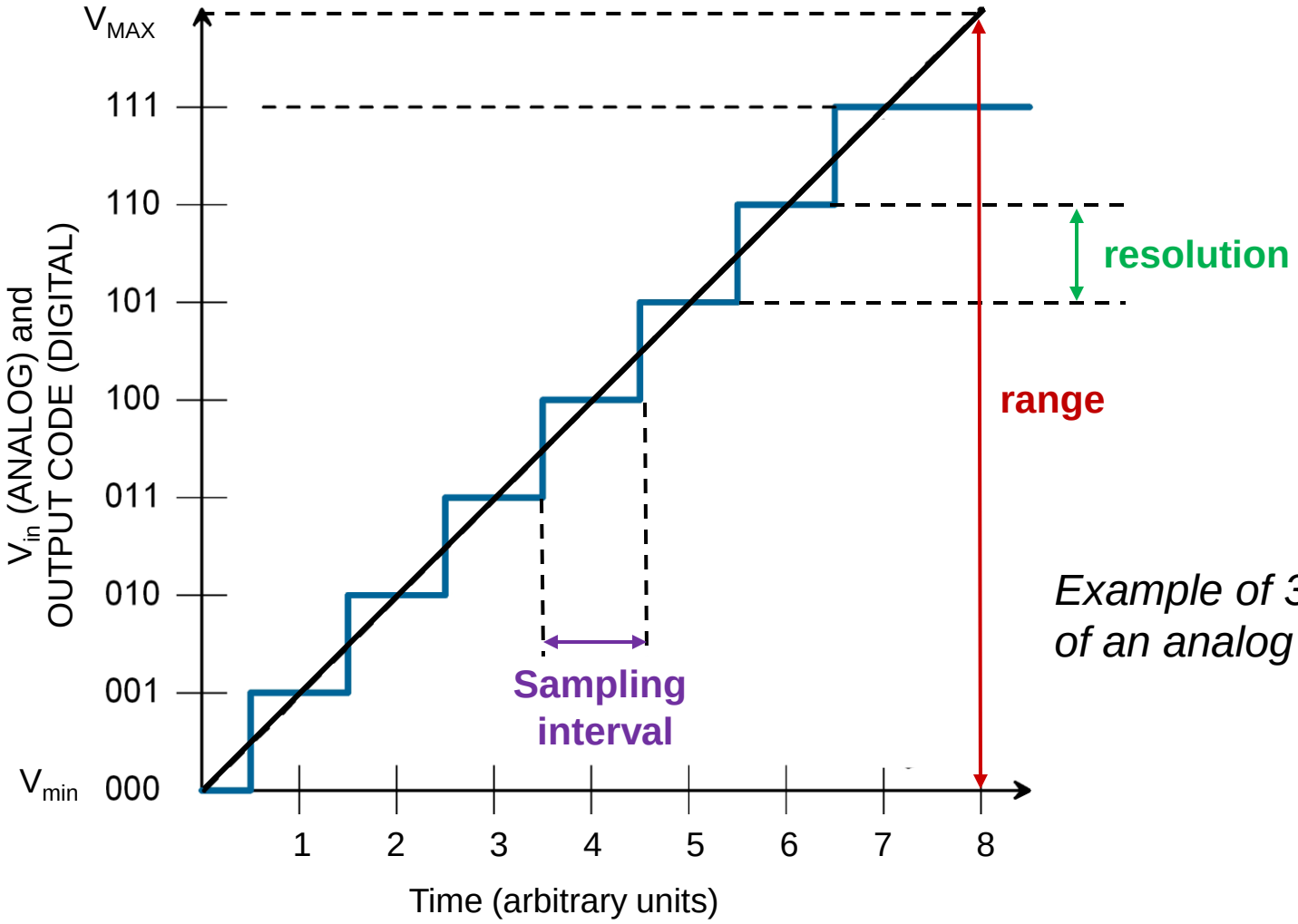
  - FOC2: forces a compare match in non-PWM mode
  - WGM20: enables the PWM function in Timer 2
  - COM2x: set the compare mode (same as in Timer 1)
  - WGM21: enables counter clear on compare match
  - CS2x: prescaler selection as per the other counters

Analog to digital conversion

- An essential process if we want to deal with physical-world information using a digital system.

- Converting a signal from analogue to digital means associating its value (Voltage) to a word of a certain length.
  - Range: the maximum voltage that the signal can span
  - Resolution: the minimum difference in signal which can be rendered accurately from the analogue to the digital signal. It is a function of the number of bits
  - Sampling interval: the time between successive digital samples of a (continuous) analogue signal

Analog to digital conversion (2)



Example of 3-bit sampling of an analog signal

Analog to digital conversion (2)

- An Analog to Digital Converter (ADC) determines the proportion of the voltage range covered by the input signal at the time of sampling and associates a bit value to that sample.

- Given a range, the resolution is easily calculated as

$$V_{resolution} = \frac{V_{range}}{2^n - 1}$$

- So for example in a 3-bit conversion of 5V each bit would correspond to approximately 714mV

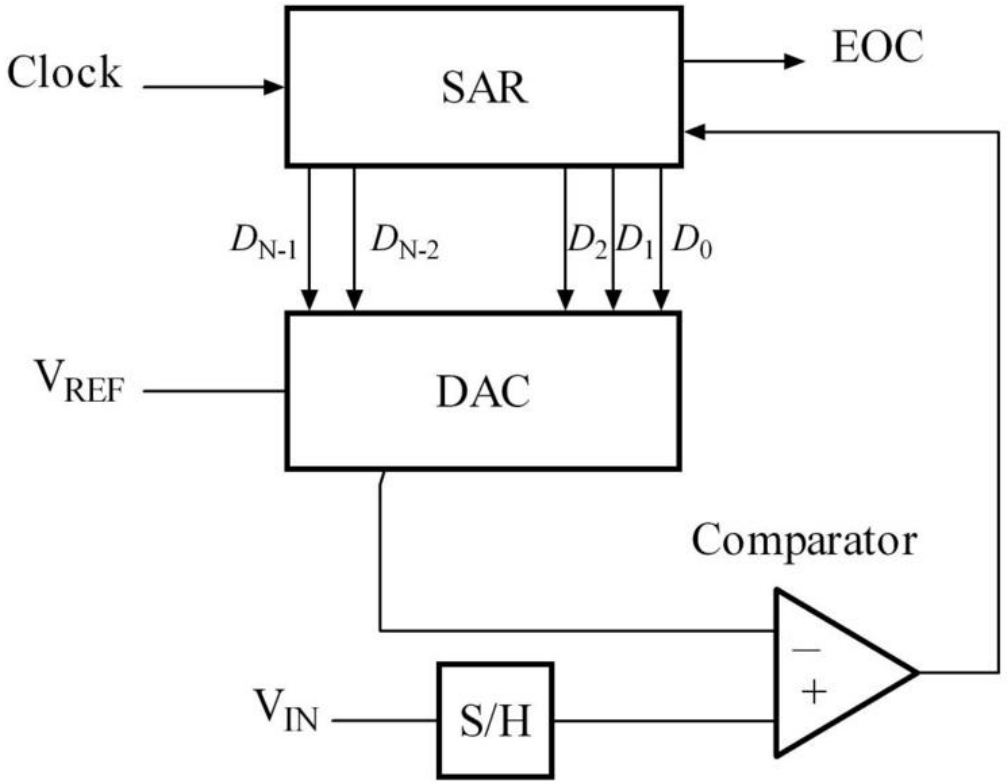
Analog to digital conversion (3)

- The analog to digital conversion which the Atmega8 carries out is based on the *successive approximation* method.
  - The input signal is sampled (Sample and Hold) and compared with reference voltages generated by an internal digital-to-analog converter, using a comparator.
  - By comparing it with a series of voltages, the circuit can determine multiple of the ADC resolution closest to the input voltage → corresponding bit word.

Analog to digital conversion (4)

- Here is a block diagram of the system. SAR is the successive approximation register (imagine an n-bit register). The system operates as follows:

1. Set the MSB of the SAR to 1, generate the corresponding voltage with the DAC
2. If the signal is greater than the voltage, retain 1 otherwise change the bit to 0
3. Move onto the next bit and repeat the operation
4. Once all bits in the SAR have been processed the SAR contains the best digital approximation of V_{IN}

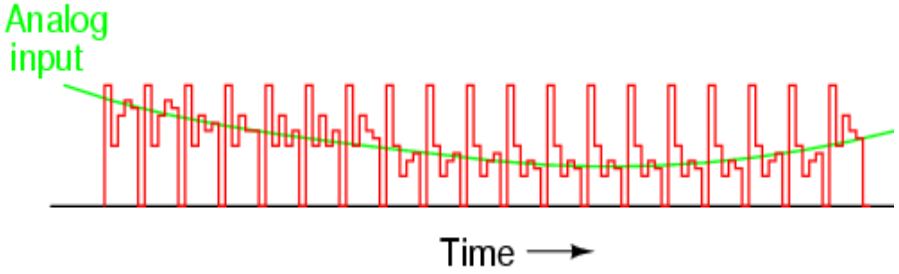


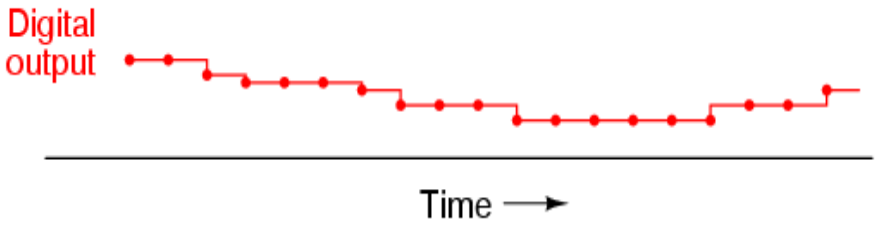
Analog to digital conversion (5)

- Let's see an example. V_{ref}=5V and V_{in}=3V, 4-bit ADC

SAR	Voltage	Comp result
1000	2.500	1
1100	3.750	0
1010	3.125	0
1001	2.813	1

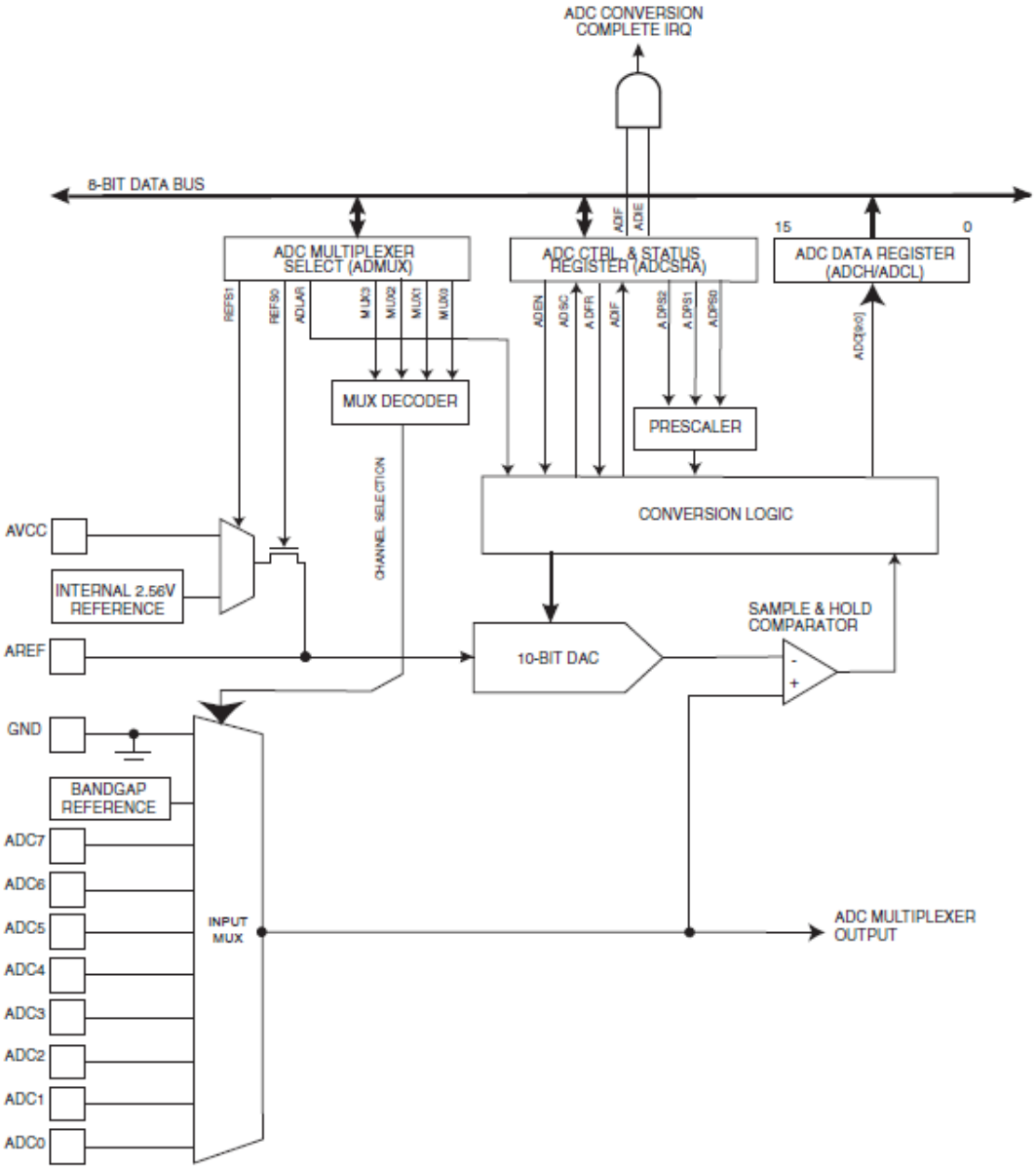
- So, 1001 is the final result. Some error between the analog voltage and the digital equivalent is always present. It is called *quantisation error*.





ADC in the AVR

- This is the schematic from the Atmega8
  - It seems more complica-ted but works in the same way
- We can notice:
  - Multiple ADC inputs (multiplexed conversions)
  - Some control registers
  - An ADC conversion IRQ (interrupt query)



ADC in the AVR (2)

- The Atmega8 has a 10-bit ADC on-board and can convert in the range 0-5V at a maximum rate of 15kSPS (thousands of Samples per Second)
  - We have already pointed out the multiplexed input
  - The ADC has an internal clock to time the conversion process and it can work at full resolutions for clock speeds in the range of 50kHz to 200kHz.
    (Can someone figure out why it can't convert at full resolution if too fast or too slow?)

- Two registers control the ADC
  - The ADC control and status register (ADCSRA)
  - The ADC multiplexer select register (ADCMUX)

ADC in the AVR (3)

- The ADC control and status register (ADCSRA)

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	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	

ADEN	ADC Enable bit, set to enable ADC
ADSC	ADC Start Conversion. Set to 1 to start a conversion
ADFR	ADC Free Running bit. Set to place in free running mode
ADIF	ADC interrupt flag. Set by hardware when conversion is done
ADIE	ADC interrupt mask bit. Set to allow ADC interrupts
ADPSx	ADC Prescaler select bits. Configure to set ADC clock to correct division of system clock (available settings on page 208 of the datasheet)

ADC in the AVR (4)

- The ADC multiplexer select register (ADCMUX)

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
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	

REFSx	ADC reference settings.
ADLAR	ADC Left-Adjust Result. This bit determines whether the conversion result is left or right-adjusted in the result register
MUXx	Channel selection bits. Select ADC input pin source

- The results register (ADCH/L) – *right adjusted*

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	

ADC in the AVR (5)

- The ADC can run in **two modes**
  - ***Single conversion***, when a conversion is enabled and an interrupt is usually generated once it is complete
  - ***Free running mode***, when conversion is carried out continually. Generally when using free-running mode ADC interrupts should be disabled and the program should pause/resume the ongoing conversion each time it accesses the results register to avoid reading data while the ADC is updating it

- Usually the following **steps** are involved in **single-conversion mode**
  1. Set the prescaler to a suitable division
     (divide system clock by 200kHz and choose next highest division factor)
  2. Set ADIE high to enable interrupts
  3. Set ADEN high to enable the ADC
  4. Set ADSC to start a conversion

ADC example

- Example of interrupt-mode use of ADC conversion

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

ISR(ADC_vect)
{
    uint16_t adc_data; //look at this data type for unsigned int!
    adc_data=ADCW; //ADCW is a short-hand code way to read the full 10 bit
                   //result from ADCH/L
    if (adc_data > 1024*2/3)
       PORTC = 0b11111011; //LED on 3rd pin of port C goes on (active low)
    else if (adc_data > 1024/3)
       PORTC = 0b11111101; //LED on 2nd pin of port C goes on (active low)
    else PORTC = 0b11111110; //LED on 1st pin of port C goes on (active low)

    ADCSRA = ADCSRA | 0x40; //set ADSC bit to 1 again to star next conv.
}

void main ()
{
    DDRC = 0x07;    //set bottom 3 bits of port C for output
    ADMUX = 0x3;    //select to read only from ADC3
    ADCSRA = 0xCD; //ADC enabled,/8 divisor, interrupt unmask, conv started

    asm ("sei");    //enable interrupts
    while(1)
       ;            //do nothing (ADC interrupt does it all)
}
```

ADC example (2)

- Example of free-running use of ADC conversion

```
int main (void)
{
   DDRC |= (1 << 2); // Cool new syntax. Set bit 2 of DDRC to 1.
   DDRC |= (1 << 0); // Cool new syntax. Set bit 0 of DDRC to 1.

   ADCSRA |= (1 << ADPS1) | (1 << ADPS0);
   // Set ADC prescaler to 128 - 125KHz sample rate @ 1MHz clock

   ADMUX |= (1 << ADLAR); /* Left adjust ADC result. This means that
      reading ADCH alone gives result with 8-bit precision*/

   // No MUX values needed to be changed to use ADC0

   ADCSRA |= (1 << ADFR);  // Set ADC to Free-Running Mode
   ADCSRA |= (1 << ADEN);  // Enable ADC
   ADCSRA |= (1 << ADSC);  // Start A2D Conversions

   while(1)
   {
      if(ADCH < 128)
         PORTC = 0b00000100; //LED on pin 2 off, 0 on (active low)
      else
         PORTC = 0b00000001; //LED on pin 0 off, 1 on (active low)
   }
 }
```

ADC example (3)

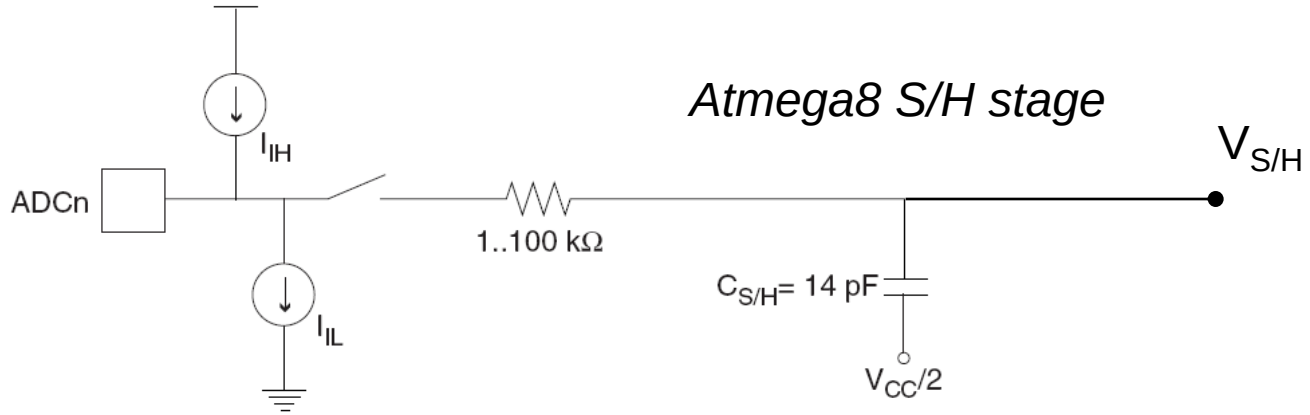
- Example of use of ADC conversion over multiple channels

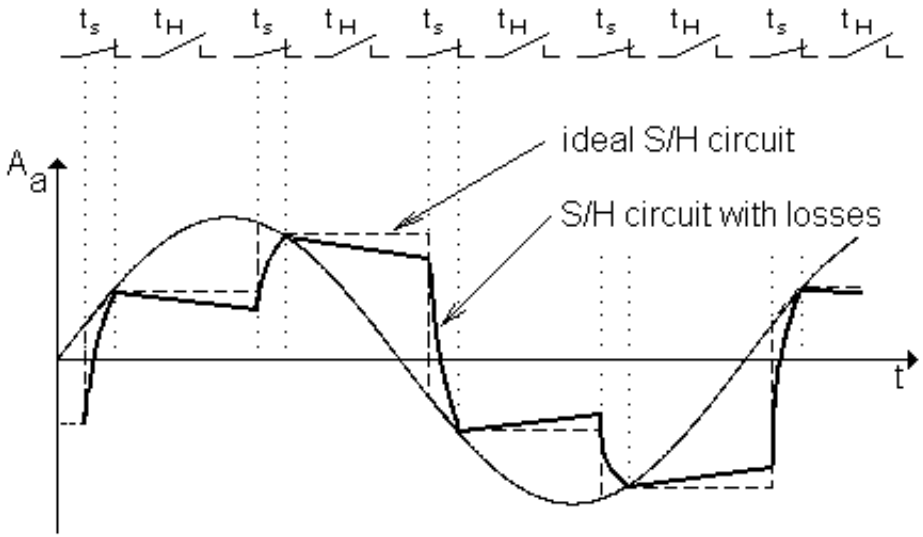
```
void main(void)
{
    ADCSRA |= (1 << ADPS1) | (1 << ADPS0);        //set clock prescaler
    ADCSRA |= 1<<ADIE; //enable ADC interrupt
    ADCSRA |= 1<<ADEN; //enable ADC
    DDRC=0x03;         //set output on 2 lsb pins of port C
    sei(); //an alternative to asm("sei");
    ADCSRA |= 1<<ADSC; //start conversion

    while (1) ;
}

ISR(ADC_vect)
{
    uint16_t adc_data=ADCW;
    switch (ADMUX)
    {
      case 0xC0:
          if (adc_data>512) PORTC|=0x01;
          else PORTC&=0x02;
          ADMUX = 0xC1; //at next iteration use different MUX settings (next case)
          break;
      case 0xC1:
          if (adc_data>512) PORTC|=0x02;
          else PORTC&=0x01;
          ADMUX = 0xC0; //at next iteration use different MUX settings (previous case)
          break;
      default: break;
    }
    ADCSRA |= 1<<ADSC; //set start conversion bit again
}
```

Summing up

- Timer/counters
  - Prescalers and interrupts
  - Input capture and output compare
  - Relevant registers
- ADC conversion in general
  - Sample/Hold
  - Successive approximation conversion
- ADC on the Atmega8
  - Relevant registers
  - Single conversion (interrupt-based) and free-running mode
  - Multiplexing
- Some new ways of writing embedded C pertinent to the AVR and the AVR GCC compiler

Additional slide. Sample&Hold





What happens when sampling with a nonideal S/H