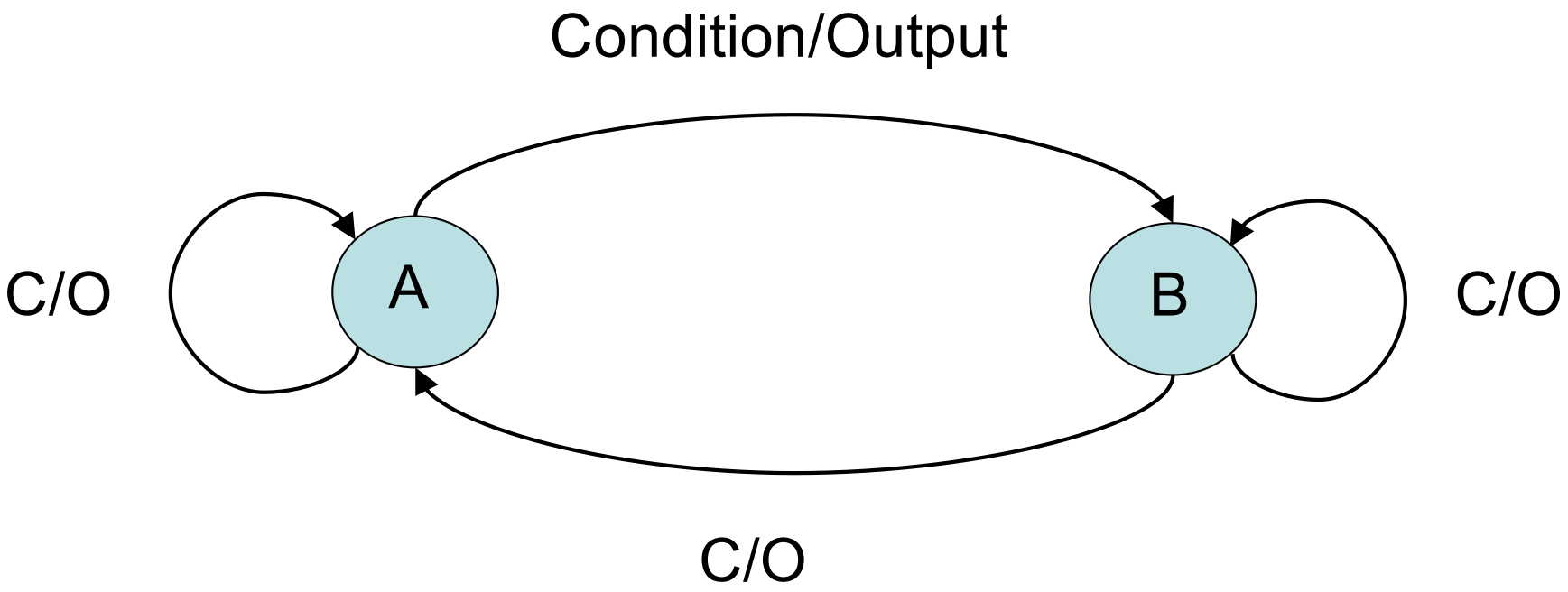
State machines (remember?)



State machine in C - simple

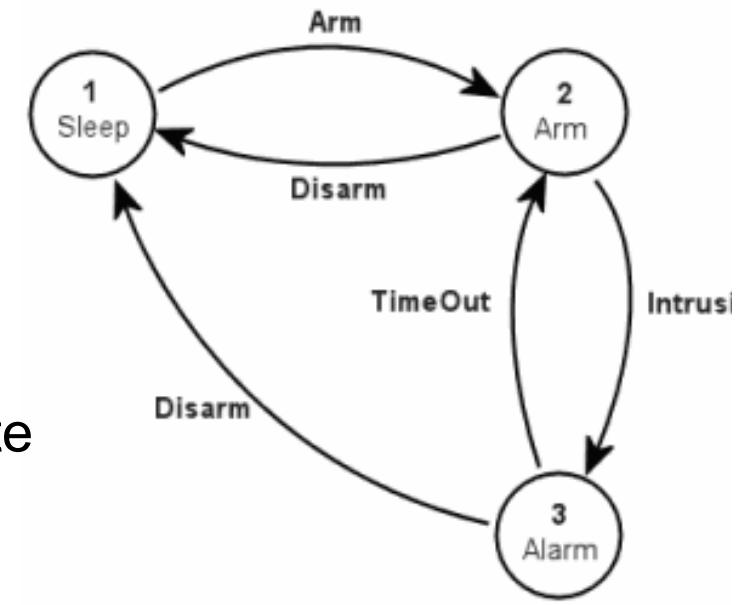
```
typedef enum {
  LED_ON,
  LED_OFF
} states;

states state = LED_OFF;                  // Initial state
while(1) {
  if(state == LED_OFF) {                 // Only new state
    led_on();                            // is important
    state = LED_ON;
  } else {
    led_off();
    state = LED_OFF;
  }
  sleep(1); // sleep for a second
}
```

Tenta 2015-01-30 (1/2)

In our example, we will have 3 states:

- **Sleep state** (You are at home)
- **Arm state** (Before leaving home, you set the alarm on)
- **Alarm state** (Something happened !)



We have two actions (outputs) in our system:

- in Arm state, an indicator will show that this state is on (or off, otherwise).
- in Alarm state, a siren will be on

Many events (inputs) can appear:

- The alarm is armed (You are leaving home)
- The alarm is disarmed (You are back)
- There is an intrusion
- The alarm time-out has expired (for instance, the siren was sounding and has to stop now)

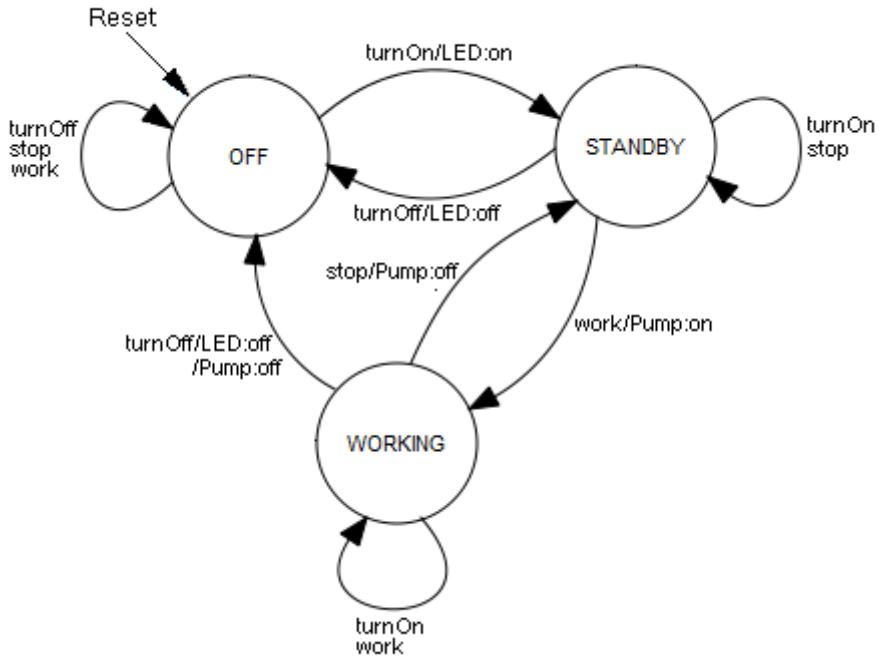
Tenta 2015-01-30 (2/2)

```
while (1) {
        Switch (current_state) {
                Case SLEEP:
                        arm_indicator (OFF); siren (OFF);
                        If (arm_system()) {

                                Next_state = ARM;
                        } break;
                Case ARM:
                        arm_indicator (ON); siren (OFF);
                        If (!arm_system()) {
                                Next_state = SLEEP;
                        } else if (intrusion ()) {
                                next_state = ALARM; starttimer();
                        } break;
                Case ALARM:
                        arm_indicator (OFF); siren (ON);
                        If (!arm_system()) {
                                Next_state = SLEEP;
                        } else if (timeOut()) {
                                next_state = ARM;
                        }break;
        }
        current_state = next_state;
}
```

Example: State machine (Mealy)

```
state ="OFF";
While(1): switch (entry):
Case ”turnOff”:
        if (state =="STANDBY”):
                state ="OFF”;
                println("LED off");
        if (state =="WORKING”):
                state ="OFF”
                println("LED and pump off'
Case ”turnOn”:
        ifstate =="OFF":
                state ="STANDBY”
                println("LED enabled")
Case ”stop”:
        if (state =="WORKING”)
                state ="STANDBY”
                println("Pump off")
Case ”work”:
        if (state =="STANDBY”)
                state ="WORKING”
                println("Pump enabled")
```



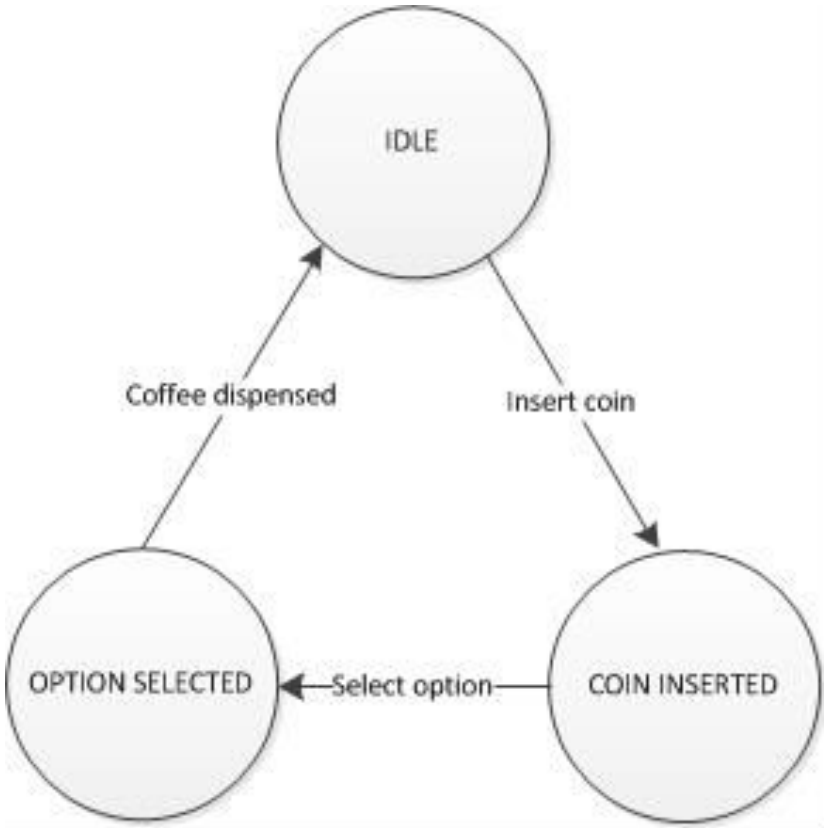
State machine in C - alternatives

- There are other ways….e.g.:
  - Tables (example from Lab 5)
    - For each combination, show next state/output

State/input	Inp=”1”	Inp=”2”	Inp=”3”	Inp=”NK”
C	C/Show C	F/Show F	CF/ ShowCF	C/Show C
F	C/Show C	F/Show F	CF/ ShowCF	F/Show F
CF	C/Show C	F/Show F	CF/ ShowCF	CF/ ShowCF

  - Function pointers
  - …

Example: Coffe table machine



This is a very simple approach.
A switch-case statement or if-else statement is used to check each state. Within each state, another conditional statement is added to check the event triggered.
A handler is then added for that state/event combination.
Alternatively, we could swap the order and check the event first and then the state. If there is a state change involved, the handler returns the new state.

```
typedef enum {
  IDLE_STATE,
  COIN_INSERTED_STATE,
  OPTION_SELECTED_STATE,
  MAX_STATES
} state_e;

typedef enum {
  INSERT_COIN_EVENT,
  SELECT_OPTION_EVENT,
  COFFEE_READY_EVENT,
  MAX_EVENTS
} event_e;

state_e state = IDLE_STATE;
state_e next_state;
event_e event;

while(1)
{
  event = read_event();
  if (state == IDLE_STATE)
  {
    if(event == INSERT_COIN_EVENT)
    {
      next_state = insert_coin_event_handler();
    }
  }
  else if(state == COIN_INSERTED_STATE)
  {
    if(event == SELECT_OPTION_EVENT)
    {
      next_state = select_option_event_handler();
    }
  }
  else if(state == OPTION_SELECTED_STATE)
  {
    if(event == COFFEE_READY_EVENT)
    {
      next_state = coffee_ready_event_handler();
    }
  }

  state = next_state;
}
```

Example: Coffe table machine (moore)

Link to place of origin in later slides…

Table-based approach - example

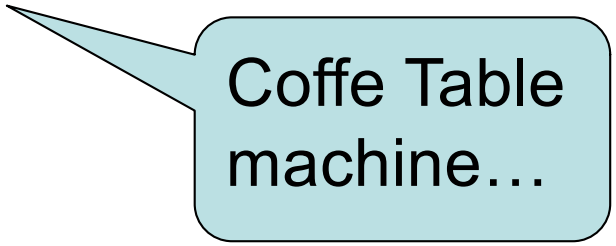
In this approach, the state machine is coded in a table with one dimension as states and the other as events. Each element in the table has the handler for the state/event combination. The table could be implemented in C using a two-dimensional array of function pointers.

```
state_e (*state_table[MAX_STATES][MAX_EVENTS])(void) = {
  {insert_coin_event_handler, error_handler, error_handler},
  {error_handler, select_option_event_handler, error_handler},
  {error_handler, error_handler, coffee_ready_event_handler},
};

while(1)
{
  event = read_event();
  if((event >= 0) && (event < MAX_EVENTS))
  {
    next_state = state_table[state][event]();
    state = next_state;
  }
}
```

Useful Links:

- http://codeandlife.com/2013/10/06/tutorial-state-machines-with-c-callbacks/
- http://www.heptapod.com/lib/statemachines/
- http://www.embedded.com/design/prototyping-and-development/4442212/Implementing-finite-state-machines-in-embedded-systems



ATmega32u4 A/D Converter

- Succesive Approximation A/D
- 10/8 Bit resolution
- 1/2 LSB Accuracy
- 65-260 uS conversion time
- 12 Multiplexed input channels
- Interrupt on AD conversion complete

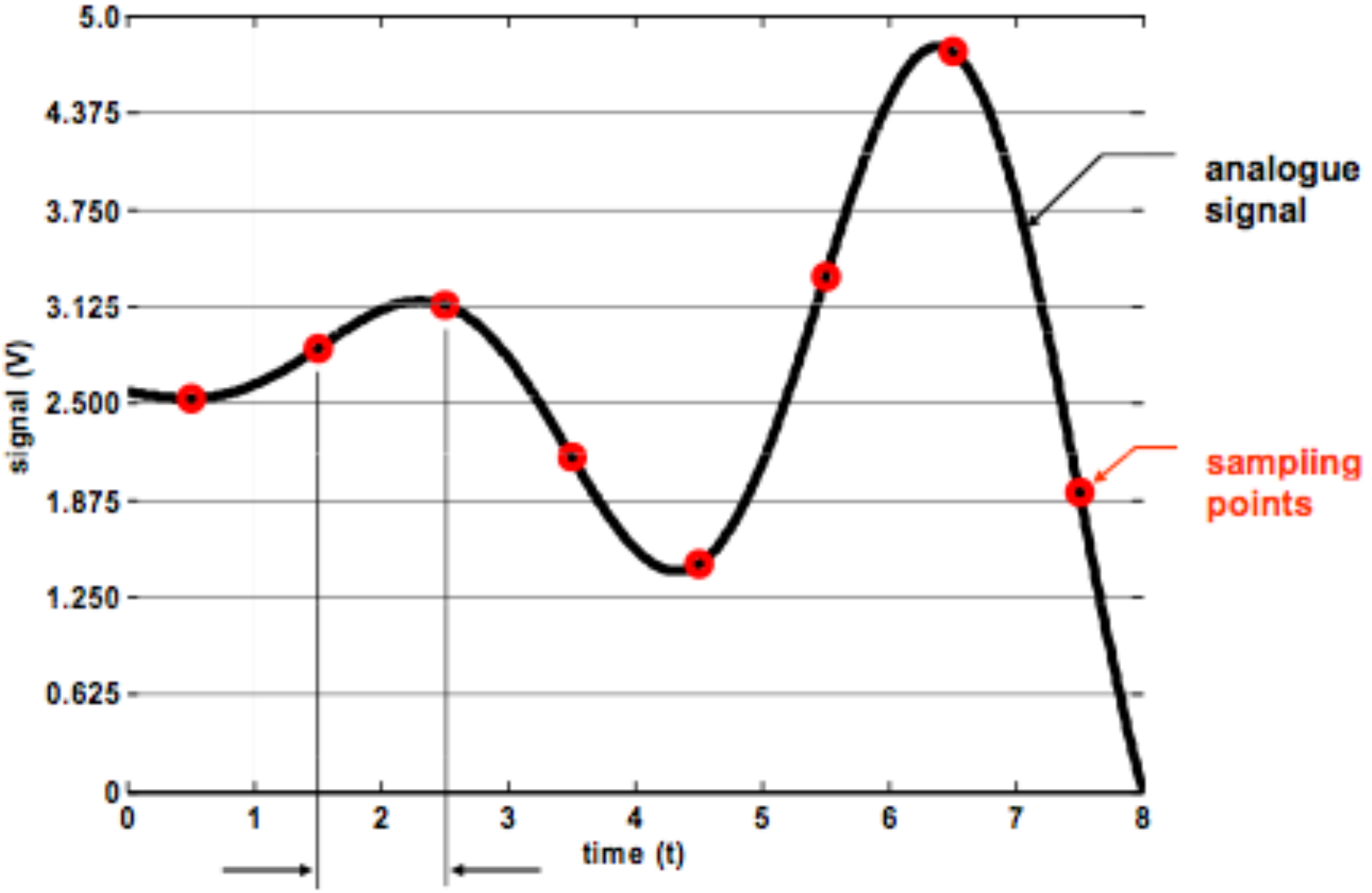
A-to-D conversion: The process

- There are two related steps in A-to-D conversion:
  - sampling,
  - quantisation.

- Sampling:
  - The analogue signal is extracted, usually at regularly-spaced time instants.
  - The samples have real values.

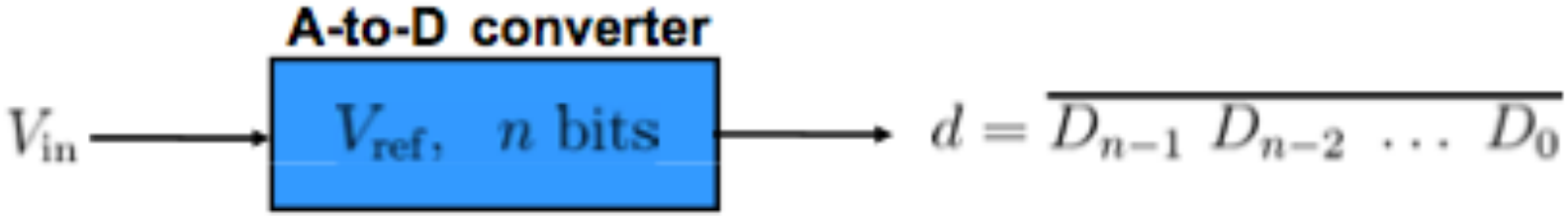
- Quantisation:
  - The samples are quantized to discrete levels.
  - Each sample is represented as a digital value.

Sampling an analogue signal



What is a suitable sampling period for a signal?

Quantising the sampled signal



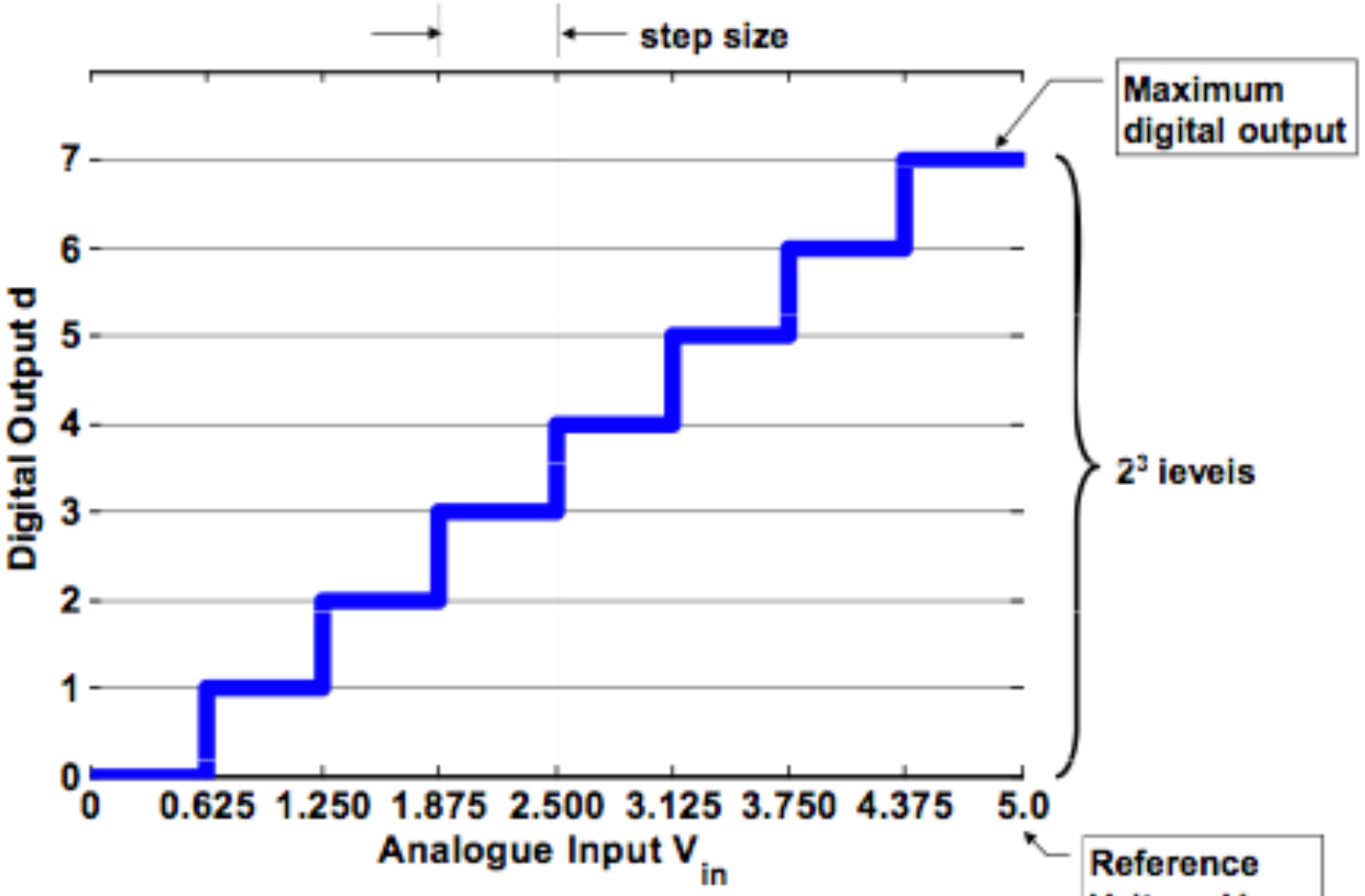
- Consider an n-bit ADC.
- Let V_{ref} be the reference voltage.
- Let V_{in} be the analog input voltage.
- Let V_{min} be the minimum allowable input voltage, usually $V_{min} = 0$.
- The ADC's digital output, $d = D_{n-1}D_{n-2} \ldots D_0$, is given as

$$d = \text{round down } [\frac{V_{in} - V_{min}}{\text{step size}}]$$

- The step size (resolution) is the smallest change in input that can be discerned by the ADC:

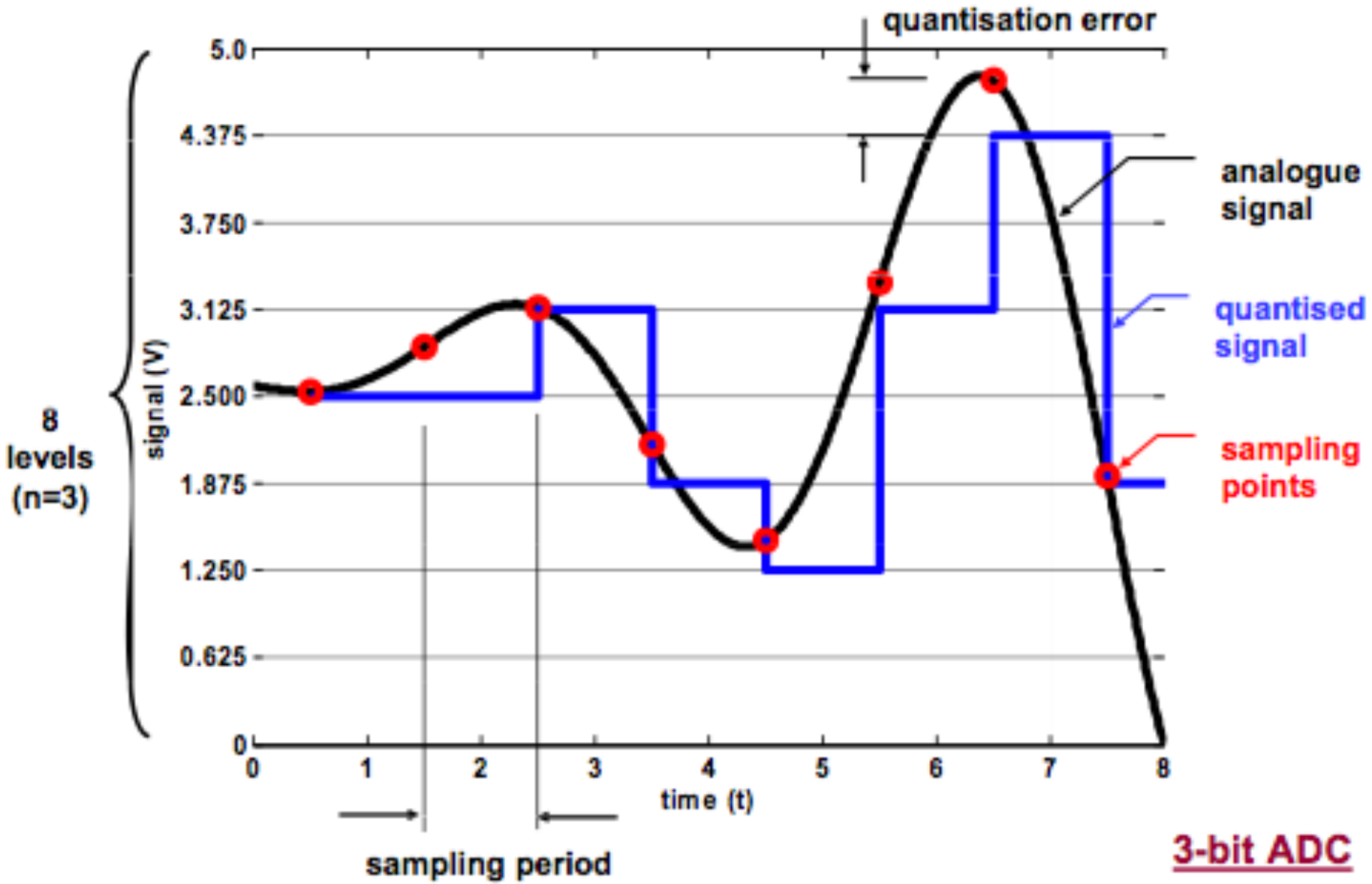
$$\text{step size} = \frac{V_{ref} - V_{min}}{2^n}$$

Quantising the sampled signal



A 3-bit A-to-D converter

Quantising the sampled signal



3-bit ADC

A-to-D converter: Parameters

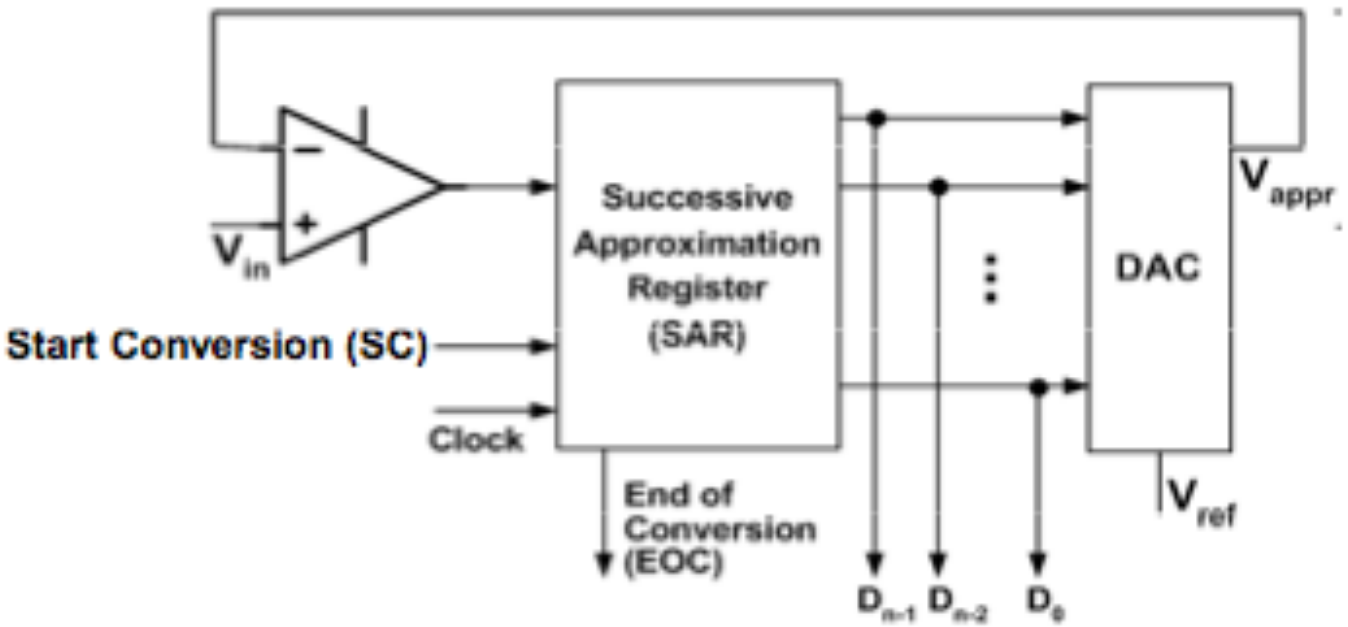
- Number of bits n: The higher is the number of bits, the more precise is the digital output.

- Quantization error E_q: The average difference between the analogue input and the quantised value. The quantisation error of an ideal ADC is half of the step size.

- Sample time T_{sample}: A sampling capacitor must be charged for a duration of T_{sample} before conversion taking place.

- Conversion time T_{conv}: Time taken to convert the voltage on the sampling capacitor to a digital output.

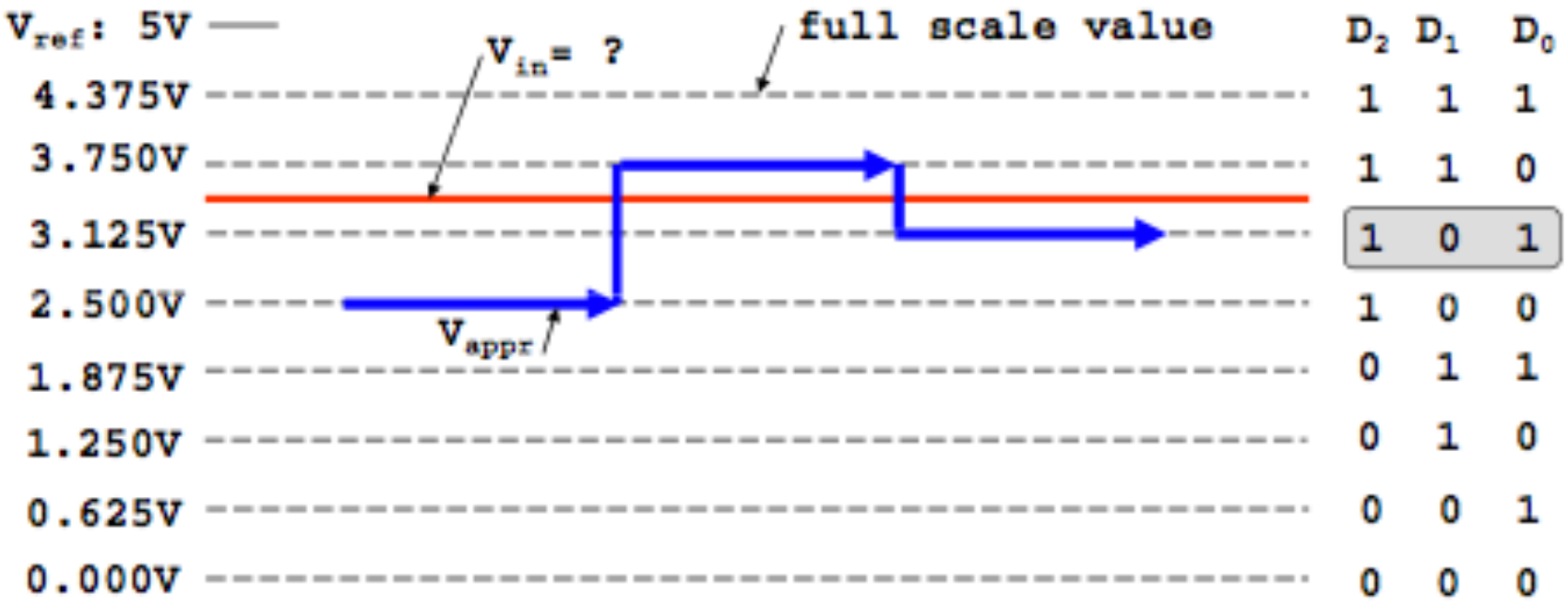
Successive-approximation ADC



- A DAC is used to generate approximations of the input voltage.
- A comparator is used to compare V_{in} and V_{appr}.
- In each cycle, SAR finds one output bit using comparator.
- To start conversion, set SC = 1. When conversion ends, EOC = 1.
- Quite fast, one of the most widely used design for ADCs.

Successive-approximation ADC

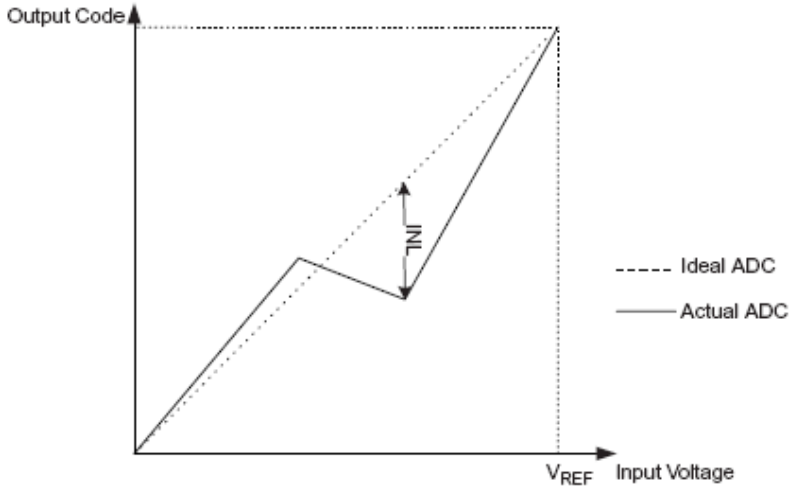
Binary search for a 3-bit ADC



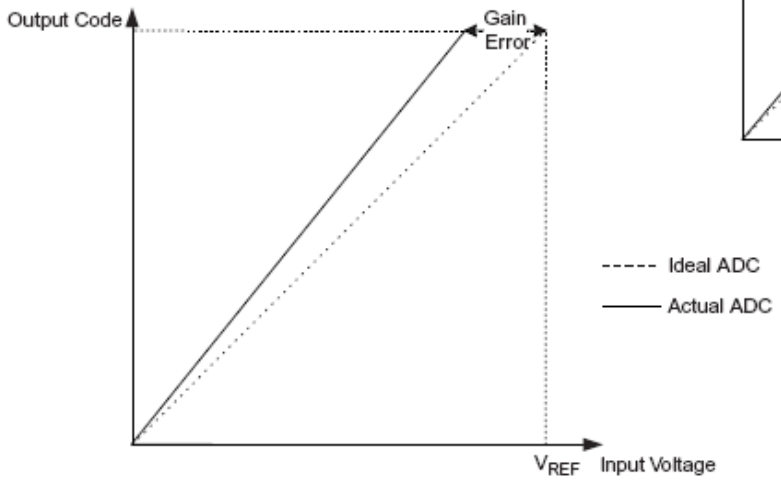
clock cycle 1	clock cycle 2	clock cycle 3
$D_2 = 1$	$D_1 = 0$	$D_0 = 1$
$[V_{in} > V_{appr}]$	$[V_{in} < V_{appr}]$	$[V_{in} > V_{appr}]$

The Analog Digital Converter

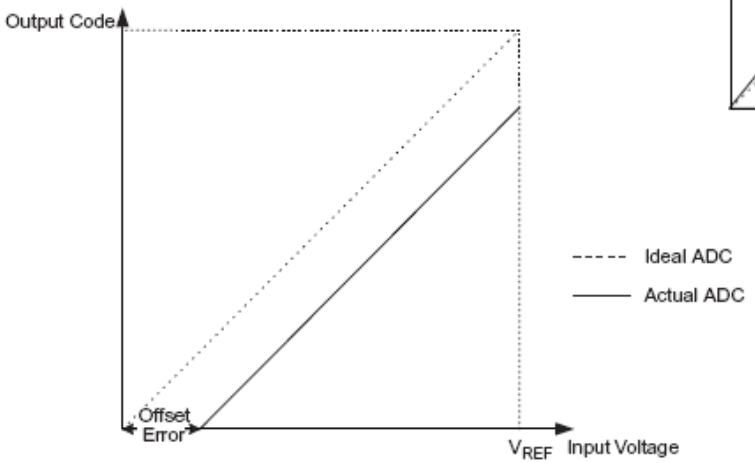
Types of Conversion Errors:



Non-linearity

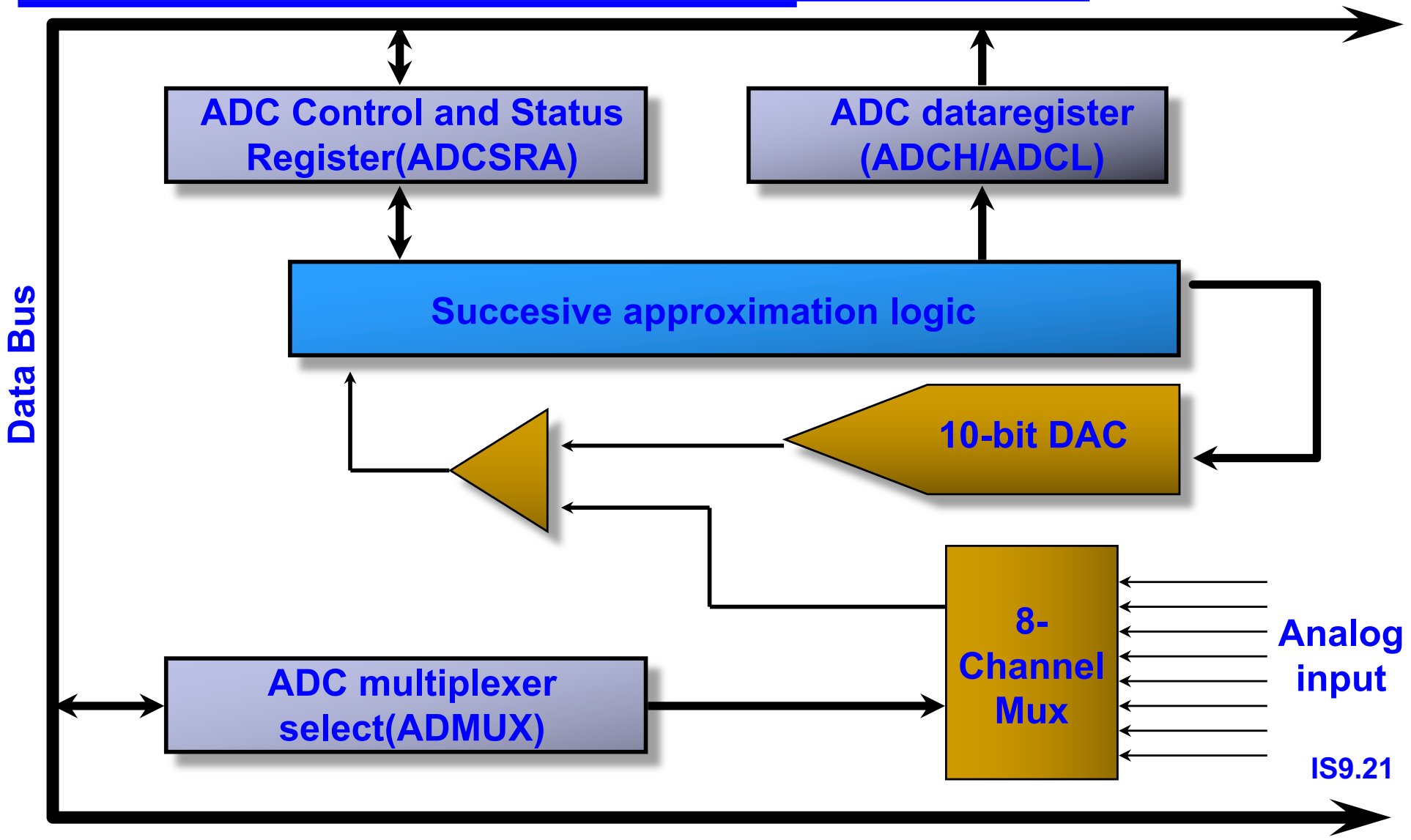


Gain error



Offset error

ATmega32U4 A/D converter



ADC

- General AVR ADC reading process:
  - Enable ADC
    - Set reference voltage
    - Set Left/Right adjust of result (10 bits in 16 bit registers)
    - Set prescale
  - Specify pin to read
    - ADMUX selection
    - Disable digital signals on same physical pins…
  - Call for conversion
  - Check completion bit (or wait for Interrupt)
  - Copy data from data register

ADC pin selection – DIDR0 (Digital Input Disable Register 0)

Analog Input0	Analog Input1	Analog Input2	Analog Input3	Analog Input4	Analog Input5
ADC7	ADC6	ADC5	ADC4	ADC1	ADC0
PF7	PF6	PF5	PF4	PF1	PF0

- We cannot use the same pins in PORTF and ADC simultaneously. It may also cause problems:
- *"an analog input will be floating all over the place, and causing the digital input to constantly toggle high and low. This creates excessive noise near the ADC, and burns extra power"*
- We can disable PF0 and PF1, when using ADC0 and ADC1. See 24.9.5 **DIDR0** register ("1" means disable)

ADMUX (1/

Table 13-4: V_{ref} source selection table

REFS1	REFS0	V Reference
0	0	AREF pin
0	1	AVCC pin
1	0	Reserved
1	1	Internal 2.56 V

REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

REFS1:0- Bit7:6 Reference Selection Bits
These bits select the voltage reference for the ADC.

ADLAR- Bit5 ADC Left Adjust Reslts
This bit dictate either the left bits or the right bits of the lt registers ADCH:ADCL are used to store the result. If we write ADLAR to one th sult will left adjusted otherwise the result is right adjusted.

MUX4:0- Bit4:0 Analog Channel and gain selection bi
The value of these bits selects the gain for the differential els and also which combination of analog inputs are connected to the ADC.

We use
AVCC = 5V

ADMUX (2/2)

- ADMUX – ADC Multiplexer
- This register is being mainly used to set the ADC source, ADC reference source and also the ADC bit alignment selector

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
0	0	0	0	0	0	0	0

Refrence Sel. Bits

00 -> AREF, Internal V_{REF} OFF

01 -> V_{REF} is equal to V_{AVCC} = 5V

10 -> Reserved

11 -> V_{REF} = 2.56V (Internal Ref V)

ADC Channel Sel. Bits

0000 -> ADC0

0001 -> ADC1

0100 -> ADC4

0101 -> ADC5

0110 -> ADC6

0111 -> ADC7

ADC input selection

Table 24-4. Input Channel and Gain Selections

MUX5..0[1]	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
000000	ADC0	N/A		
000001	ADC1			
000010	N/A			
000011				
000100	ADC4			
000101	ADC5			
000110	ADC6			
000111	ADC7			
001000	N/A	N/A	N/A	N/A
001001		ADC1	ADC0	10x
001010		N/A	N/A	N/A
001011		ADC1	ADC0	200x
001100	N/A	N/A		
001101				
001110				
001111				
010000		ADC0	ADC1	1x

Note! MUX5 is in ADCSRB!

ADCSRA (1/3)

ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN- Bit7 ADC Enable
This bit enables or disables the ADC. Writing this bit to one will enable and writing this bit to zero will disable the ADC even while a conversion is in progress.

ADSC- Bit6 ADC Start Conversion
To start each coversion you have to write this bit to one.

ADATE- Bit5 ADC Auto Trigger Enable
Auto Triggering of the ADC is enabled when you write this bit to one.

ADIF- Bit4 ADC Interrupt Flag
This bit is set when an ADC conversion completes and the Data Registers are updated

ADIE- Bit3 ADC Interrupt Enable
Writing this bit to one enables the ADC Conversion Complete Interrupt.

ADPS2:0- Bit2:0 ADC Prescaler Select Bits
These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

ADCSRA (2/3)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
0	0	0	0	0	0	0	0

ADC Prescaler Sel. Bits

000 -> 2	100 -> 16
001 -> 2	101 -> 32
010 -> 4	110 -> 64
011 -> 8	111 -> 128

ADCSRA – (3/3)

- **ADEN:** This bit is set to enable the ADC.
- **ADSC:** This bit is set to start an ADC Conversion
- **ADIF:** ADC Interrupt flag set by the hardware when a conversion is over.
- **ADIE:** ADC Interrupt Enable bit.
- **ADSPx :** ADC prescaler bit. ADC Sampling works well at a frequence range of 50kHz to 200kHz. So we need to prescale the system clock value down to a value within that frequency.

ADC Prescaler

- PreScaler Bits let us change the clock frequency of ADC
- The frequency of the computer is 16 Mhz. The slowest rate of conversion is 16000/128 = 125 kHz
- Conversion time is 13 ADC cycles ≈ 1660 CPU cycles

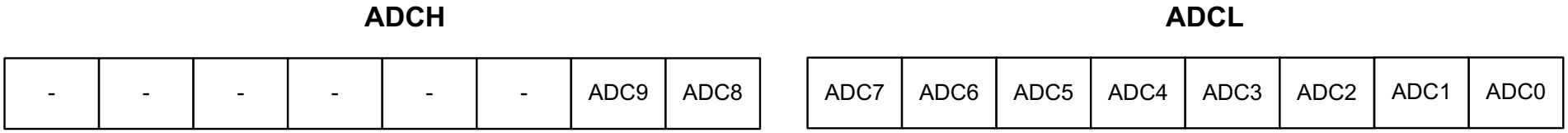
Table 24-5. ADC Prescaler Selections

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

ADCH and ADCL Data registers

- ADCH:ADCL store the results of conversion.
- The 10 bit result can be right or left justified (based on value of ADLAR bit in ADMUX:

ADLAR = 0



ADLAR =1

ADCSRB

Bit	7	6	5	4	3	2	1	0	
	ADHSM	ACME	MUX5	–	ADTS3	ADTS2	ADTS1	ADTS0	ADCSRB
Read/Write	R/W	R/W	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADHSM: ADC High Speed Mode**

Writing this bit to one enables the ADC High Speed mode. This mode enables higher cor
rate at the expense of higher power consumption.

- **Bit 5 – MUX5: Analog Channel Additional Selection Bits**

This bit make part of MUX5:0 bits of ADRCSRB and ADMUX register, that select the c
tion of analog inputs connected to the ADC (including differential amplifier configuration)

- **Bit 3:0 – ADTS3:0: ADC Auto Trigger Source**

See table 24-6. first line below…

Table 24-6. ADC Auto Trigger Source Selections

ADTS3	ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	0	Free Running mode

No need to set for this mode…

ADC Interrupt vector

Table 9-1. Reset and Interrupt Vectors (Continued)

Vector No.	Program Address(2)	Source	Interrupt Definition
30	$003A	ADC	ADC Conversion Complete
31	$003C	EE READY	EEPROM Ready
32	$003E	TIMER3 CAPT	Timer/Counter3 Capture Event
33	$0040	TIMER3 COMPA	Timer/Counter3 Compare Match A

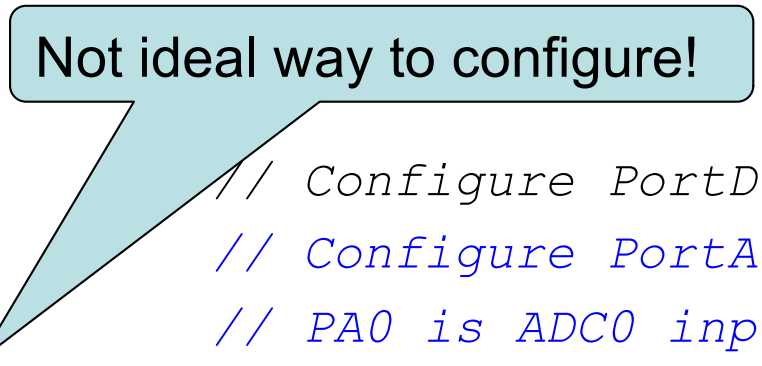
In C, the ADC interrupt vector is called “ADC_vect”

Example code

```
ISR(ADC_vect) {
        PORTD = ADCH;           // Output ADCH (only) to PortD
        ADCSRA |= 1<<ADSC;      // Start Conversion again
}

int main(void) {
        DDRD = 0xFF;            // Configure PortD as output
        DDRA = 0x00;            // Configure PortA as input ???
                                // PA0 is ADC0 input
        ADCSRA = 0x8F;// Enable the ADC and its interrupt
                      // set ACD clock pre-scalar to clk/128
        ADMUX = 0xE0; // Select int2.56V as Vref, left justify
                      // data reg, select ADC0 as input ch

        sei();                  // Enable Global Interrupts
        ADCSRA |= 1<<ADSC;      // Start Conversion
        while(1);               // Wait forever
}
```



Example code

```
ISR(ADC_vect) {
        PORTD = ADCH;          // Output ADCH (only) to PortD
        ADCSRA |= 1<<ADSC;     // Start Conversion again
}
```

This way is better!

```
int main(void) {
        DDRD = 0xFF;           // Configure PortD as output
        DDRA = 0x00;           // Configure PortA as input ???
                               // PA0 is ADC0 input
        ADCSRA = (1<<ADEN) |(1<<ADIE); // Enable ADC, interrupt
        ADCSRA|= (1<<ADPS2)|(1<<ADPS1)|(1 << ADPS0); //presc 128
        ADMUX |= (1<<REFS0)|(1<<REFS1)| (1 << ADLAR);//ref 2,56V
        sei();                 // Enable Global Interrupts
        ADCSRA |= 1<<ADSC;     // Start Conversion
        while(1);              // Wait forever
}
```

Example setup: – Used for next example

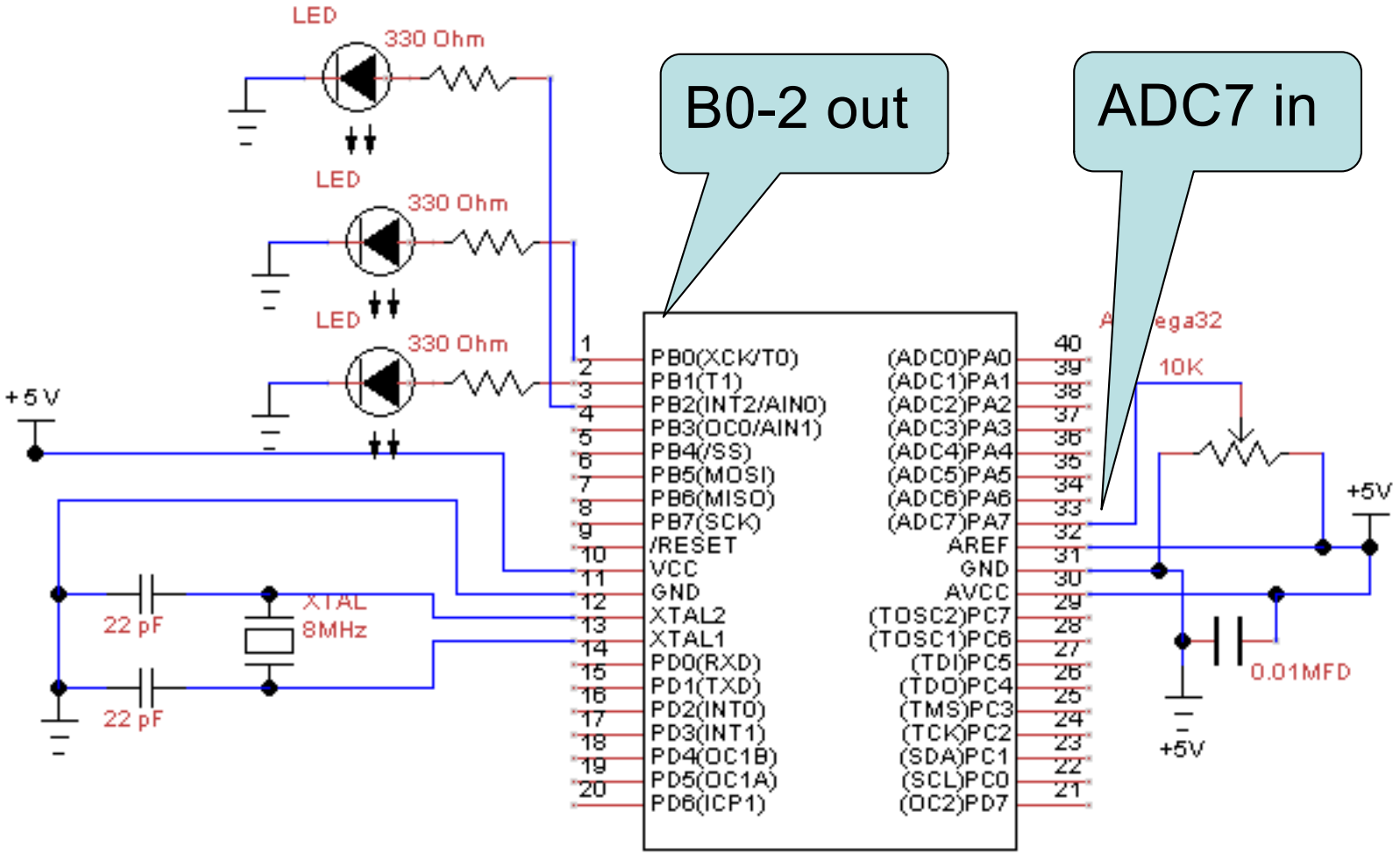


Figure 4.0

Example code (1/2):

```
#include <avr/io.h>
#include <util/delay.h>
#include<avr/interrupt.h>
void main() {
    DDRB = 0b00000111; //Grn=PB0, Yellow=PB1, Red=PB2
    PORTB = 0x00;
    ADCSRA = (1<<ADEN) | (1<<ADSC) | (1<<ADIE);
    ADCSRA|=(1<<ADPS2)|(1<<ADPS1)|(0 << ADPS0); //presc.
    ADMUX = (1<<ADLAR)|(1<<MUX0)|(1<<MUX1)|(1<<MUX2);
    sei();
    while(1) {}
}
```

First set these

Then add these

Example code (2/2):

```
ISR(ADC_vect) {
      uint8_t x = ADCH; //Using the 8-bit ADC mode
      if      (x > 128)    PORTB = 0x04; //PB2 on
      else if (x > 64)     PORTB = 0x02; //PB1 on
      else                 PORTB = 0x01; //PB0 on

      ADCSRA |= (1<<ADSC); // start conv again
}
```

Example:

Assume all bits = 0 if not set to 1!

Example: Initialisation for Interru driven AD Conversion

```
ADCSRA =  (1<<ADPS2) | (1<<ADPS1);
   // Prescaler = 64, free running mode off, interrupts off
   // prescaler = 64 (ADPS2 = 1, ADPS1 = 1, ADPS0 = 0)
   // ADCYCLE 7,3728Mhz/prescaler 115200Hz or 8.68 us/cycle
   // 14 (single conversion) cycles = 121.5 us (8230 s/sec)
   // 26 (1st conversion) cycles = 225.69 us

ADCSRA |= (1<<ADIF);      // Reset any pending ADC interrupts
ADCSRA |= (1<<ADEN);      // Enable the ADC
ADCSRA |= (1<<ADIE);      // Enable ADC interrupts

ADMUX=0;                    // Select ADC channel 0
ADCSRA |= (1<<ADSC);        // single conversion: Start ADC
```

The Analog Digital Converter

Example: ADC Interrupt Service Routine

```
ISR(ADC_vect)          // AD-conversion-complete
                       // interrupt service routine
{
    unsigned char low,high;
    unsigned int value;

    low = ADCL;        // read ADC value (low byte first!)
    high = ADCH;       // read ADC value high byte
    value = (high<<8) + low;
                       // calculate (16 bit) integer value
}
```

Video

- Good instruction videos:
- https://www.youtube.com/watch?v=sVvDiACf3yE
- https://www.youtube.com/watch?v=v0G-ddUeWpo
- https://www.youtube.com/watch?v=e7X6LGXhOss

Lab 5 base…

```
void temp_init(void)
{       // UPPGIFT: konfigurera ADC-enheten genom ställa in ADMUX och ADCSRA
        // enligt kommentarerna nedan!

        ADMUX |= 0;             // set reference voltage (internal 5V)
        ADMUX |= 0;             // select diff.amp 10x on ADC0 & ADC1
                                // right adjustment of ADC value

        ADCSRA |= 0;            // prescaler 128
        ADCSRA |= 0;            // enable Auto Trigger
        ADCSRA |= 0;            // enable Interrupt
        ADCSRA |= 0;            // enable ADC

        DIDR0 = 3;              // disable digital input on ADC0 and ADC1

        USBCON = 0;             // disable USB controller
                                // (to make interrupts possible)

        sei();                  // enable global interrupts
                                // start initial conversion
        ADCSRA |= 0;            // UPPGIFT: gör så att den initiala
                                // A/D-omvandlingen sker
}
```