NTNU

Department of Engineering Cybernetics

Suggested Solution

Examination paper for TTK4155 Industrial and Embedded Computer Systems Design

Academic contact during examination: Jo Arve Alfredsen

Phone: 90945805

Examination date: Monday 2019-12-09

Examination time (from-to): 09:00 - 13:00

Permitted examination support material: D

Standard pocket calculator permitted.

Printed and handwritten material not permitted.

Other information:

Answers may be given in English or Norwegian

Concise answers are required.

Read the text carefully. Each problem may have several questions.

Exam counts 50% of final grade.

Language: English

Number of pages (front page excluded): 5

Number of pages enclosed: 0

Informasjon om trykking av eksamensoppgave

Originalen er:

1-sidig ☐ **2-sidig** ☐

sort/hvit ☐ **farger** ☐

skal ha flervalgskjema ☐

Checked by:

Date Signature

It should be noted that these are suggested solutions. Alternative solutions may exist to some of the problems.

Problem 1. (50 %)

a. Calculate the minimum and maximum frequencies of the TOV3 interrupt of Timer/Counter 3 of an AVR ATmega162 microcontroller (see Figure 1) running at 8MHz.

   (5%)

   The TOV3 interrupt is generated when TCNT3 overruns from 0xFFFF to 0x0000 in normal mode. T/C3 may be clocked directly from the system clock or by a prescaled (downscaled) version of it. No prescaler gives the maximum TOV3 frequency. The largest prescaler value is 1024, which gives the minimum TOV3 frequency.

   System clock: f_{osc} = 8000000 Hz

   $f_{TOV3\_MAX} = f_{osc}/2^{16} = \underline{\underline{122.07\text{ Hz}}}$

   $f_{TOV3\_MIN} = (f_{osc}/2^{10})/2^{16} = \underline{\underline{0.119\text{ Hz}}}$

   (T/C3 operating in normal mode was the intended "goes without saying" interpretation of this question. However, setting the timer in PWM-mode with a minimum roll-over value TOP=0x0001, a TOV3 frequency of $f_{osc}/2$ may be generated. Generating interrupts at this rate is not meaningful and any handling them would most likely lead to disaster. However, since normal mode was not explicitly stated, suggesting this solution, despite being unpractical, will not be penalized.)

b. An ultrasonic range sensor measures distance to objects in the range 0.3 – 10 m. The sensor output is made available as a pulse train where the pulse width is proportional to the measured distance with resolution 1 µs/mm.
   Write a driver in C/pseudo-code for the sensor using the microcontroller from a) and the timing mechanism provided by Timer/Counter 3 to output full resolution distance measurements. The driver should be non-blocking and contain the functions:

```
uint8_t sensor_control(uint8_t on) // on=1/0 sensor driver on/off
int16_t sensor_getrange_mm(void) // returns distance to object in mm
```

   (20%)

   The duration of the high state of the pulses (pulse width) needs to be measured continuously as long as the pulse train is present. The input capture (ICP) unit of T/C3 is ideal (and actually made) for this kind of task. The sensor output must therefore be connected to the MCU ICP3 pin and T/C3 should be set up to generate TIMER3_CAPT interrupts alternatingly on the rising and falling edges of the sensor signal. Shortest and longest pulse widths will be 300mm*1µs/mm=300µs and 10000*1µs/mm=10000µs, respectively. A prescaler of 8 on the 8MHz system clock will give a timer increment of one per microsecond, which preserves sensor resolution while avoiding timer overflow issues ($10000<2^{16}$). Making sure the timer is reset on the rising edge, the distance measurements in millimeters will then be available directly as the ICR3 falling edge timestamp in the TIMER3_CAPT ISR. As long as the driver is turned ON, new pulse width measurements will be calculated continuously in the background by the ISR, making the driver non-blocking (no busy-wait for a new measurement) and capable of immediately returning the measurement of the most recent sensor pulse when invoked.

```c
// sensor.c
// Pseudo-code can be used for details and things not fully documented in exam text
#define F_CPU 8000000UL
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>

// We need to measure the duration/pulse width of a pulse in a pulse train with a resolution of 1us
// Shortest distance: 0.3m => 300 usec pulse width
// Longest distance: 10m => 10000 usec pulse width
// Input Capture (ICP) unit of TIMER3 should be used for this purpose, with the sensor signal on the ICP3 pin
// TIMER3 clock tick should be set to 1us, which is achieved by prescaling the 8MHz system clock by 8
// The ICES bit is used to alternatingly trigger TIMER3_CAPT interrupts on rising and falling pulse edges
// The distance measurement (range) in mm is simply the ICR3 readout on the sensor pulse falling edge
// Failing to use the input capture unit is considered as a wrong solution

#define ON 1
#define OFF 0

// Driver state variables (limit scope to this file)
static volatile int16_t range = 0;
static uint8_t sensor_driver_state = 0;

// External sensor driver interface
// Function declarations sensor_control and sensor_getrange_mm in sensor.h-file
uint8_t sensor_control(uint8_t on)      // assume sensor always powered and physically connected to MCU PD3
{
      if(on) {                          // turn sensor driver on
            DDRD &= ~(1<<PD3);          // set ICP3(PD3) as input for the sensor
            PORTD |= 1<<PD3;            // enable pull-up register
            TCCR3A = 0x00;              // normal count mode and waveform (timer reset)
            TCCR3B |= (1<<ICNC3) || (1<<ICES3) || (1<<CS31); // enable input capture unit (ICP3)
            with noise canceler and capture on rising edge. Set prescaler to 8 (8/8MHz = 1usec)
            ETIFR |= 1<<ICF3;           // clear ICP flag after setting to ICES to avoid spurious interrupt
            ETIMSK |= 1<<TICIE3;        // enable TIMER3 input capture interrupt
            sensor_driver_state = ON;   // Here some validation of “sensor signal good” could be implemented
            sei();                      // enable global interrupts
      }
      else {                            // turn sensor driver off
            ETIMSK &= ~(1<<TICIE3);     // disable TIMER3 input capture interrupt
            TCCR3B = 0;                 // stop TIMER3
            sensor_driver_state = OFF;
            range = 0;
      }
      return sensor_driver_state;
}

int16_t sensor_getrange_mm(void)        // return range if driver active, sensor assures range>=300 if valid
{
      if(sensor_driver_state == ON) {
            return range;
      }
      else {
            return 0;
      }
}

//TIMER3 input capture interrupt on ICP3 pin – calculates new range continously (for every sensor pulse)
//Toggling the ICES3 bit before leaving ISR makes it fire on both rising and falling edges
//This is the “heart” of the sensor driver. Keep ISR as slim as possible...
ISR (TIMER3_CAPT_vect)
{
      TCNT3 = 0;                        // reset timer immediately to start new measurement
      if(TCCR3B & (1<<ICES3) == 0)      // check if interrupt was caused by a falling edge (ICES3 bit zero)
            range = ICR3;               // range[mm] = pulse width[us] = no. of timer ticks captured in ICR3
      TCCR3B ^= 1<<ICES3;               // toggle edge select bit
      ETIFR |= 1<<ICF3;                 // clear ICP flag after changing edge sel. by writing 1 to its bit loc.
}

// Somewhere else------------------------------------------------------------------------------------------
#include “sensor.h”

int main (void)
{
      int16_t r;

      sensor_control(ON);
      while(1) {
            // Do stuff here…
            r = sensor_getrange_mm();
            // use r for something
            // Do other stuff here…
      }
      sensor_control(OFF);
}
```

c. What is the purpose of a quadrature encoder (see Figure 3)?
   Explain how you can use the quadrature encoder together with the microcontroller from a) to create a rotational position sensor (quadrature counter).
   Suggest an implementation of the position sensor using C/pseudo-code. The sensor firmware should be able to run in the background unaffected by other code running on the MCU.

(25%)

A quadrature encoder is a sensor that measures rotational speed and direction, of e.g. motors shafts. The frequency of the signal A (or B) is proportional to rotational speed (RPM = 60*freq(A)/N), and the phase shift between A and B determines rotational direction ($\Delta\varphi(A,B) = +90°$ or $-90°$. A leads B, or B leads A by 90°). NB: an incremental encoder like the HEDS-9000 is not a rotational position sensor in itself.

The ATmega162 can in conjunction with the quadrature encoder operate as a rotational position sensor by maintaining a counter variable which is clocked by signal A (or B). The counter should be bi-directional and incremented or decremented depending on the direction of rotation.

A simple solution is to connect signal A to one of the external interrupt (INTn) pins and signal B to a GP-input pin, and let the INTn ISR increment or decrement the counter variable depending on the current state of signal B (see Figure 3).

A more elegant solution would be to connect signal A to one of the input capture interrupt (ICPn) pins instead of the external interrupt (INTn) pin, and exploit the timer capture mechanism to determine rotational speed through frequency measurement of A as well. The timer clock frequency must be set to some sensible value that maintains good resolution and matches the actual speed range of the rotating equipment. Some kind of handling of low/zero speed situations, for which the timer will saturate, must be implemented. The TOVn interrupt can conveniently be used to detect low/zero speed situations and define a suitable speed cut-off or overflow handling.

Assumptions (other parameters may be selected, but calculations should show the feasibility of implementation with the available hardware resources):

- Motor speed is in the range 0 – 600 RPM
- Encoder has a codewheel with resolution N = 500
- Using T/C1 with signal A connected to the ICP1 pin and signal B to GPIO PE1
- The ICP1 ISR maintains a counter variable ‘position’ that reflects rotational position and is incremented/decremented by one for each interrupt depending on the rotational direction ‘dir’, and the time in timer ‘ticks’ since the previous rising edge of signal A for RPM calculation.

Calculations:

- f_{Amax} = 500*600/60 = 5000Hz. T/C1 must therefore be able to measure pulses of 1/5000Hz = 200µs duration. The ICP1 interrupt will fire at 5000Hz at this speed.
- Selecting a prescaler of 64 gives a 64/8000000Hz = 8µs timer tick for T/C1, and therefore 200/8 = 25 timer ticks at the maximum rotational speed.
- T/C1 is 16 bit and will therefore overflow (and fire the TOV1 interrupt) at 2^{16} ticks. With the selected prescaler, this will happen at a signal A pulse period of $T_c >$ 2^{16}*8µs = 0.524s. Speeds down to 60/(0.524*500) = 0.23RPM can hence be measured with these timer settings. If acceptable the TOV interrupt may just set the measured speed to zero, or implement an overflow handling mechanism that allows even lower speed measurements.
- The counter variable (int_32) may contain (2^{31}-1)/500 = 4294967 revolutions in either direction before overflowing, which will happen first after almost five days

at full speed - which is regarded acceptable (but should be handled in some sensible way).

- If relevant, rotational position can easily be converted to linear position by knowing the conversion factor (e.g. millimeters per degree rotation).

```
// Pseudo-code can be used for details and things not fully documented in exam text
#define F_CPU 8000000UL
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>

#define ON 1
#define OFF 0
#define N 500                          //number codewheel slots
#define CW 0                           //clockwise rotation (B low when A rise)
#define CCW 1                          //counterclockwise rotation (B high when A rise)
#define PRESCALER 64                   //prescaler of timer clock
#define K 60*F_CPU/(PRESCALER*N)       //conversion factor from timer ticks to RPM (RPM = K/ticks)

// Sensor driver state variables
static volatile int32_t position = 0;
static volatile uint32_t ticks = 0;
static volatile uint8_t dir = CW;
static volatile uint16_t overflows = 0;
static uint8_t driver_state = OFF;

// Sensor driver interface
uint8_t quadrature_control(uint8_t command);     // Start/stop quadrature sensor driver
int32_t quadrature_getposition(void);            // Returns position as signed integer (divide by N to get revs)
float quadrature_getspeed(void);                 // Returns rotational speed in RPM

uint8_t quadrature_control(uint8_t command)
{
      if(command == ON) {              // turn quadrature driver on
            DDRD &= ~(1<<PE0);         // set ICP1(PE0) as input for encoder channel A
            DDRD &= ~(1<<PE1);         // set GPIN PE1 as input for encoder channel B
            PORTE |= (1<<PE0)||(1<<PE1);// enable pull-up register
            TCCR1A = 0x00;             // TIMER 1 normal count mode and waveform
            TCCR1B |= (1<<ICNC1)||(1<<ICES1)||(1<<CS11)||(1<<CS10); // enable input capture unit (ICP1)
            with noise canceler and capture on rising edge. Set prescaler to 64 (timer tick = 64/8MHz = 8usec)
            TIFR |= 1<<ICF1;           // clear ICP flag after setting to ICES to avoid spurious interrupt
            TIMSK |= 1<<TICIE1;        // enable TIMER1 input capture interrupt
            position = 0;              // reset state variables
            ticks = 0;
            dir = CW;
            overflows = 0;
            driver_state = ON;
            sei();                     // enable global interrupts
      }
      else {                           // turn quadrature driver off
            TIMSK &= ~(1<<TICIE1);     // disable TIMER1 input capture interrupt
            TCCR1B = 0;                // stop TIMER1
            position = 0;              // reset state variables
            ticks = 0;
            dir = CW;
            overflows = 0;
            driver_state = OFF;
      }
      return driver_state;
}

int32_t quadrature_getposition(void)   //Return position as the accumulation of positive and neg. codewheel counts
{                                      //div by N gives no. of revolutions, mod by N gives fraction of revolution
      if(driver_state == ON) {
            return position;
      }
      else {
            return 0;
      }
}

float quadrature_getspeed(void)        //Return speed in RPM
{
      if((driver_state == ON) && (ticks != 0)){
            if(dir == CW){
                  return K/ticks;
            }
            else {
                  return -K/ticks;
            }
      }
      else {
            return 0;
      }
}
// Continues...
// Continued...

ISR (TIMER1_CAPT_vect)                //Fires on every rising edge of quadrature channel A/ICP1. Keep it fast.
{
      ticks = ICR1 + overflows*0xFFFF; //capture number of timer ticks since last pulse (for RPM calculation)
      TCNT1 = 0;                       //reset timer to start the next measurement immediately
      overflows = 0;                   //reset overflow counter as this was a new rising edge on channel A
      dir = PINE & (1<<PE1);           //read current state of quadrature channel B to find direction
      if (dir == CW){                  //codewheel has rotated one slot CW
            position++;                //maybe consider some saturation logic here
      }
      else  {                          //codewheel has rotated one slot CCW
            position--;                //maybe consider some saturation logic here
      }
}

ISR (TIMER1_OVF_vect)                  //Handle situation when speed is very low/zero by catching TOV1
{
      overflows++;                     //keeps track of the number of timer overflows
}
```

Finally, it should be noted that it is perfectly possible to double and quadruple the resolution of the sensor by taking both rising and falling edges of either one or both channels into account, so-called x2 and x4 encoding (the above solution is an example of x1 encoding). The code above can be extended quite easily to x2 encoding by exploiting the ICES1 bit, and to x4 by connecting channel B to an external interrupt (either INT0 or INT1) and setting interrupt sense control to trigger on both edges.

Problem 2. (50 %)

Design case: Suppose that you are hired to design a new room occupancy sensor controller for a building automation company. The sensor should be capable of sensing the presence of people in a room, sensing ambient light, air temperature and quality, controlling room lighting and ventilation, as well as communicating with the building automation central both through wired or wireless network interfaces. The device's detailed specifications are given as follows:

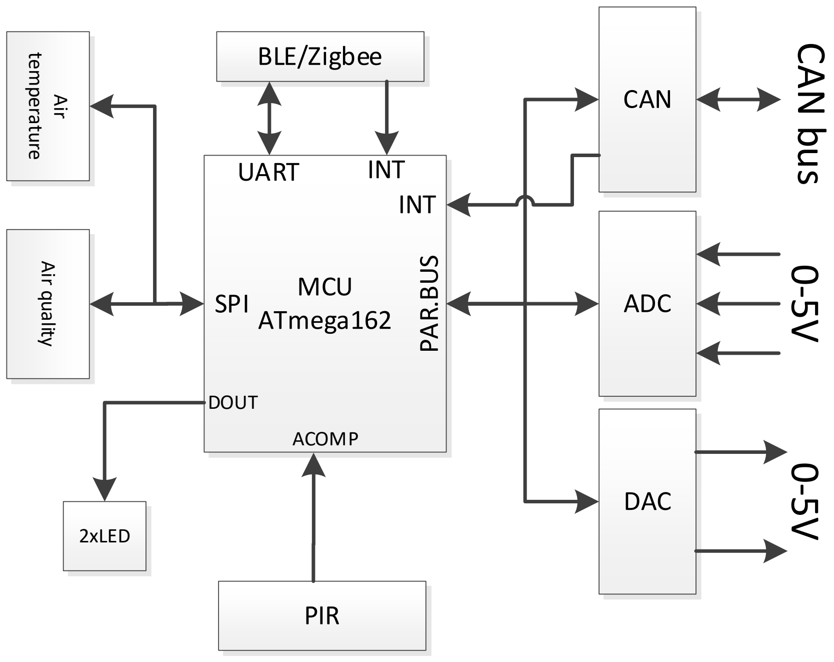
- MCU: The sensor controller should be based on a microcontroller of type AVR ATmega162 as shown in Figure 4. The MCU requires a V_{CC} = 5V power supply and an 8MHz crystal oscillator.
- Occupancy sensor: A pyroelectric passive IR (PIR) sensor with some suitable opamp signal conditioning circuitry is used for this. The output of the sensor is an analog voltage V_{PIR}, where $V_{PIR} > V_{CC}/2$ indicates personnel detection. The comparator inputs of the MCU should be used for this sensor.
- Ambient light sensor: This sensor outputs three analog voltages in the range 0 – 5V which together makes it possible to calculate the room's ambient photopic light level. This will be used to control the intensity of the room lighting fixtures.
- Air temperature and air quality sensors: These two sensors are available as integrated sensor modules each with their own SPI interface.
- Lighting and ventilation control: The room lighting fixtures and ventilation system require two analog input signals in the range 0 – 5V.
- ADC: To read sensor signals, an 8-channel AD converter with 16-bit resolution and a parallel bus interface is available. The ADC should be integrated with the MCU as a pure memory mapped I/O and requires 32 bytes memory space. The AD converter accepts voltage signals in the range 0 – 5V.
- DAC: To output control signals, a 2-channel DA converter with 8-bit resolution and a parallel bus interface is available. The DAC should be integrated with the MCU as a pure

memory mapped I/O and requires 8 bytes memory space. The DA converter generates voltage signals in the range 0 – 5V.

- Wired network interface: A CAN network interface is used as the wired connection to the building automation central. The CAN interface should be based on the SJA1000 CAN controller shown in Figure 2.
- Wireless network interface: A stand-alone wireless communication module featuring both Bluetooth LE 4.2 and IEEE 802.15.4/Zigbee is available. The module integrates the full stack of the two wireless protocols as well as a simple UART for interfacing with the host MCU.
- Two LEDs for status indication should be included.

a. Describe the system in terms of a *high-level* block diagram (a detailed circuit schematic not requested here). Read the specification above in detail, focus on identifying and drawing the individual function blocks/modules that together make up the system, and then indicate the interface between them using simple arrows and labels (no detailed bus/signal connections requested here). If deemed necessary, make your own reasonable assumptions that will make the system work as intended.

(10%)



b. Make a memory map for the computer using partial address decoding, and derive the associated decoding logic (remember that the 1280 lowest addresses (0x0000 - 0x04FF) of the ATmega162 are reserved).

(15%)

According to the specification, four ‘devices’ must be accommodated in the 64 kB (16 bit) address space of the ATmega162: CAN, ADC, DAC and AVR reserved memory space. Decoding of four separate memory ranges can be accomplished in its most simple form by using only the two MSB bits of the address bus, $A_{15}A_{14}$. Each range will cover 16kB and give (more than) sufficient space for each device:

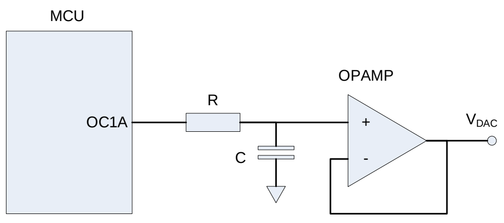
AVR reserved	0x0000 – 0x3FFF (16kB)
ADC	0x4000 – 0x7FFF (16kB)
DAC	0x8000 – 0xBFFF (16kB)
CAN	0xC000 – 0xFFFF (16kB)

Chip selects active low: $/CS_{ADC} = /A_{15}A_{14}$, $/CS_{DAC} = A_{15}/A_{14}$, $/CS_{CAN} = A_{15}A_{14}$

c. The design could be simplified by replacing the function of the DAC with internal MCU resources and some simple external signal conditioning circuitry. How?

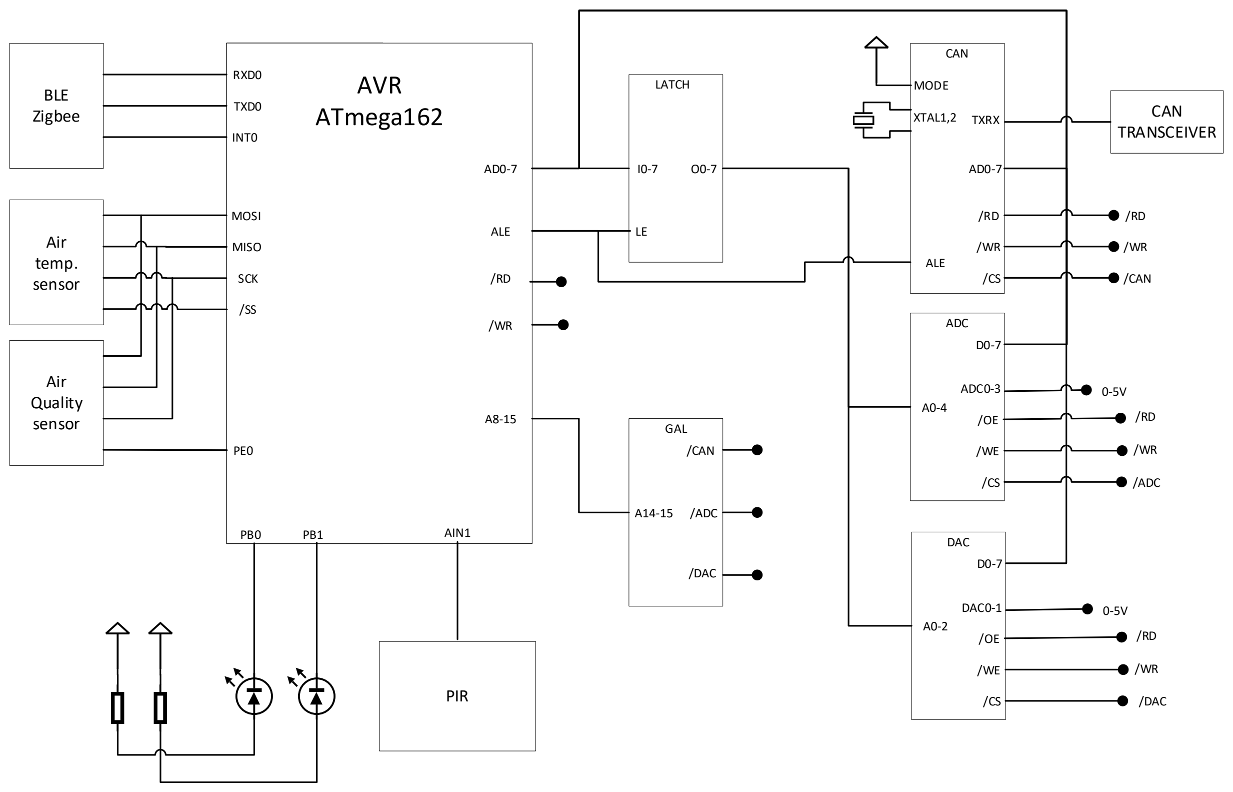
(10%)

The DAC function can be realized by setting MCU timers in PWM mode and use the waveform generation capability of the corresponding output compare pins. The PWM output should be LP-filtered and buffered to generate a low impedance analog output channel whose voltage is proportional to the PWM duty cycle. The time constant of the RC filter should be set sufficiently high compared to the PWM frequency to achieve the required averaging effect. Below an example of using T/C1 with output compare unit OCR1A/OC1A as one of the DAC channels (OCR1B/OC1B can be used similarly to realize the second DAC channel).



d. Draw and explain a circuit schematic that shows how the components of the embedded system should be connected (the details can be limited to central signal lines, but the width of all buses and which ports/bit signals are connected to must be shown). Specify the circuits and signals you find necessary to add.

(15%)



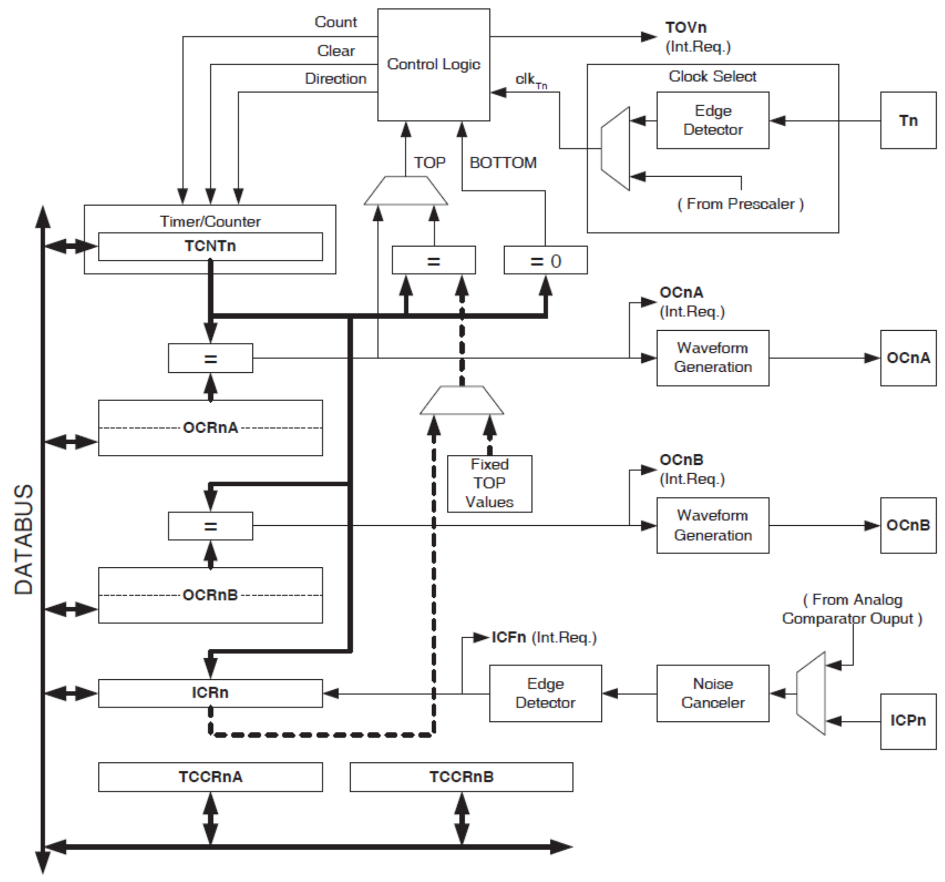


Figure 1. Diagram showing Timer/Counter 1 and 3 in the AVR ATmega162.

- In Timer/Counter n (n = 1 or 3), the TCNTn, OCRnA, OCRnB and ICRn registers are all 16 bit.
- The Timer/Counter can be driven by internal or external clock sources, either directly from the system clock ($clk_{I/O}$), via a prescaled (divided) version of the system clock, or via a clock signal on pin T1 for T/C1. The selectable prescaler values for T/C1 are 8, 64, 256 or 1024. T/C3 has in addition prescaler values 16 and 32, but cannot be clocked by an external source.
- In *Normal* mode, TCNTn increments by one for each clock pulse (0x0000→0x001→…→0xFFFF→0x0000→…). The interrupt signal TOVn (timer overflow) is generated every time the TCNTn overruns from 0xFFFF to 0x0000.
- The ICRn (input capture register) is used in normal mode to capture/timestamp external events in terms of rising and falling edges on the ICPn pin or a trigger signal from the microcontroller's internal Analog Comparator unit. When a rising or falling edge is detected on the ICPn pin or comparator output, the current value of the timer register TCNTn gets loaded immediately into ICRn and the input capture interrupt signal ICFn is generated. The ICESn bit of the timer control register TCCRnB selects which edge on the ICPn pin that is used to capture an event (0 = falling edge, 1 = rising edge).
- The OCRnA and OCRnB are two output compare registers that are used to generate the interrupt signals OCnA and OCnB as well as signal waveforms on the OCnA and OCnB pins when the timer TCNTn matches their content.
- When Timer n is set to *Clear Timer on Compare (CTC) mode*, TCNTn will be cleared to zero when a match with the content of OCRnA is detected and the interrupt signal OCnA will be generated and the OCnA pin will be toggled.
- When Timer n is set to *Fast PWM mode*, TCNTn counts from BOTTOM (0x0000) to TOP and then restarts from BOTTOM again. TOP can be set to fixed values 0x00FF, 0x01FF, 0x03FF or the content of OCRnA or ICRn. The OCnx pin is set on a compare match between TCNTn and OCRnx, and cleared at TOP. A TOVn interrupt is generated when TCNTn rolls over from TOP to BOTTOM.

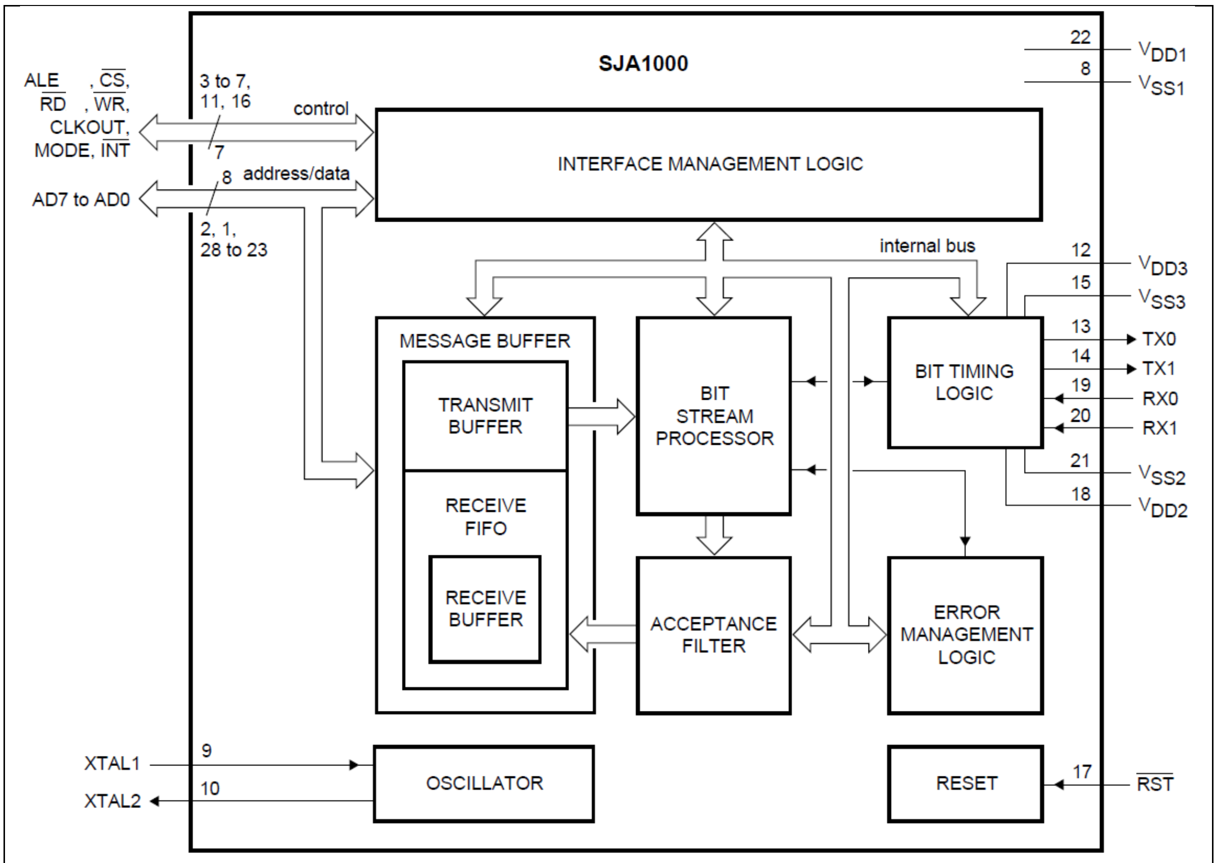


Figure 2. Diagram showing the SJA1000 CAN controller (CAN2.0B).

The SJA1000 features an 8-bit multiplexed address/data parallel bus interface (AD0-7) and a 256 byte internal address space containing CAN receive/transmit buffers as well as status and control registers. With the MODE-pin pulled high, the SJA1000 may be connected directly to the AVR ATmega162 external bus interface.

- ALE, $\overline{\text{RD}}$, $\overline{\text{WR}}$ and $\overline{\text{CS}}$ are control signals controlling bus transactions with the host microcontroller
- $\overline{\text{INT}}$ is an active low interrupt signal which is generated by the CAN controller e.g. when a new CAN message is received
- CLKOUT is a clock output signal (not used here)
- $\overline{\text{RST}}$ is an active low reset signal
- TX and RX are CAN output and input signals that should be connected to a CAN transceiver
- XTAL1 and XTAL1 are terminals for a 16 MHz crystal
- V_{DD} and V_{SS} are 5 V power and ground, respectively

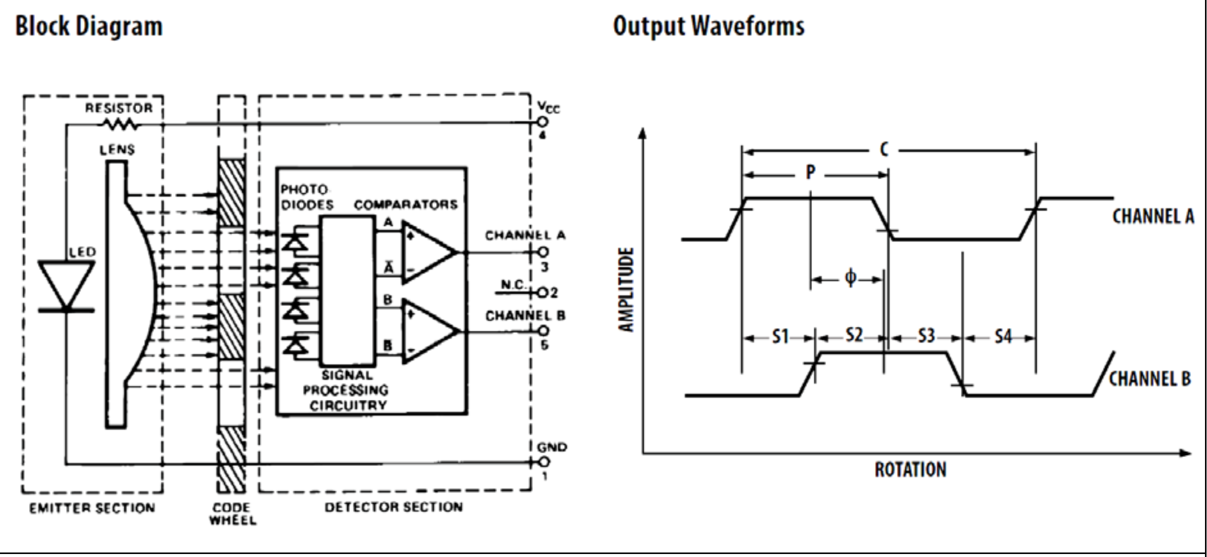


Figure 3. The HEDS-9000 quadrature encoder. The code wheel is divided into N slots and generates N cycles (C) on the output channels A and B for each rotation of the code wheel.

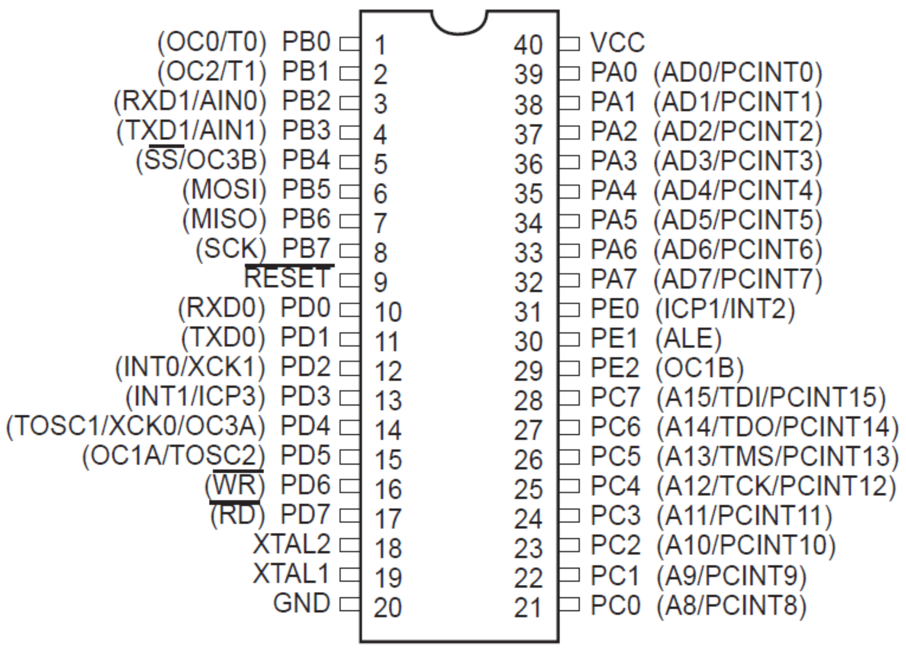


Figure 4. AVR ATmega162.