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: Saturday 2018-12-15

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

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

Answers should be concise.

Exam counts 50% of final grade.

Language: English

Number of pages (front page excluded): 7

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 Signatur

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

Problem 1. (50 %)

Tip: Read through all questions a) - d) to get an overview first.

a. Figure 1 shows the diagram of the 16 bit Timer/Counter unit (T/C) of the Atmel AVR ATmega162 microcontroller. Assume that the system clock is driven by a 4.194304 MHz external crystal oscillator and that T/C 1 is set to normal mode and clocked (clk_{T1}) by the system clock with the prescaler set to 1024.
What value must be written to the output compare register OCR1A to generate an OC1A interrupt frequency of 4 Hz?

(8%)

f_{osc} = 4194304 Hz, prescaler = 1024

$f_{OC1A} = f_{osc}/((OCR1A+1)*prescaler) = 4$ Hz

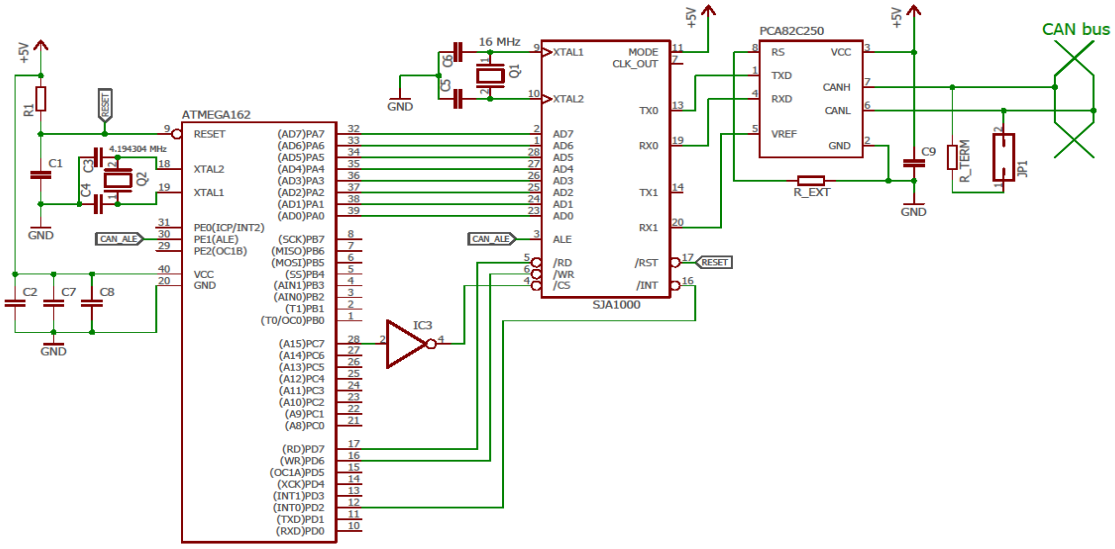
$OCR1A = 2^{22}/(2^2*2^{10}) - 1 = 2^{10} - 1 = 1023$

T/C 1 is specified to run in normal mode, meaning that TCNT1 must be reset manually in the OC1A ISR to achieve a 4 Hz OC1A interrupt frequency. The OCR1A **+ 1** comes from the fact that the OC1A interrupt is first raised at the next clock cycle after a compare match (TCNT1=OCR1A). Missing this subtlety will not be penalized as it was not stated in the included documentation (and causes just a tiny timing error here).
(Some may find CTC mode practical in this situation because it features automatic reset of TCNT1, but normal mode is explicitly stated in the specification and should be adhered to. There may be reasons why normal mode is specified, see e.g. 2.d.)

b. Figure 2 shows the SJA1000 CAN controller which features an 8-bit multiplexed address/data parallel bus interface. The SJA1000 may be connected to the external bus interface of the ATmega162 from a) as a memory-mapped I/O device using the 8-bit address/data bus and the bus control signals ALE, $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$. Assume for now that there are no other memory-mapped devices connected and hence no need for an address decoder. Assign base address 0x8000 of the ATmega162's address space to the SJA1000. Also assume that a 5 V power supply is available for the chips as well as a 16 MHz crystal for the SJA1000.
Draw a schematic showing how the chips should be connected together. Use signal names given in Figures 2 and 5.

(14%)

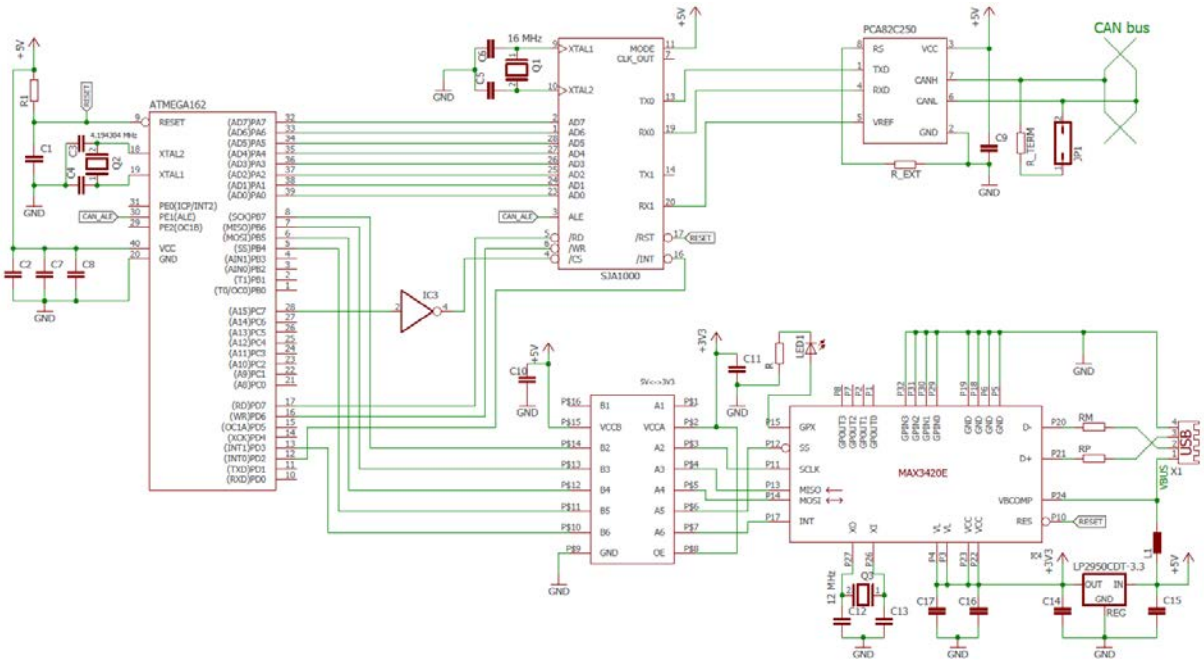
The ATmega162's external bus interface should be enabled. A simple inverter on A15 will allow reads and writes to the CAN controller from base address 0x8000, with the upper 32KB of its address space allocated to the CAN controller. The AVR and CAN controller are required in the schematic, while showing the CAN transceiver is not a requirement (but gives a good impression). Some power rails and decouplings have been left out for the sake of clarity.



c. Figure 3 shows the MAX3420E USB peripheral controller with SPI interface. The MAX3420E will be used to give the system from b) a USB interface that makes it possible to connect it to a PC.
Draw a schematic showing how the MAX3420E and the system from b) should be connected together. The complete system should get its power from the USB interface. Use signal names given in Figures 3 and 5.

(14%)

Correct connection of SPI and interrupt through suitable voltage level translator must be shown. Generation of 3.3V supply to MAX3420E from 5V USB VBUS through LDO regulator.



d. The embedded system from c) can now be employed as a CAN bus analyzer with a PC-interface, where the microcontroller monitors the CAN bus traffic and transmits the contents of CAN messages and other information to a PC application through the USB interface.
Use the T/C 1 output compare interrupt from a) to write a program in C/pseudo-code that calculates and transmits the CAN bus traffic load over the USB interface every 250 millisecond. CAN bus traffic load should be expressed as the number of CAN messages received per second. Assume that the SJA1000 generates one interrupt per received CAN message, and that you have the function *USB_SendCANLoad(float message_rate)* available to send the load value over the USB interface.

(14%)

```
// Pseudo-code can be used for details and things not fully documented in exam text
#include <stdint.h>
#include <avr/interrupt.h>
#include “timer.h” // Indicate that we have a driver for the T/C units
#include “CAN.h” // Indicate that we have a CAN driver
#include “USB.h” // Indicate that we have a USB driver

#define PERIOD 0.250f                // Measurment period (seconds)

int32_t can_count = 0;               // Holds the current count of CAN interrupts
uint8_t calulate_flag = FALSE;       // Flag for new caluclation of CAN message rate

void main {
    float can_load = 0; // CAN message load. Floating point for sake of generality

    // Initialize T/C 1
    init_Timer1(MODE=NORMAL, PRESCALER=1024, CLOCKSOURCE = SYSCLK,
                OCR3A=1023, OC3A_INTERRUPT=ENABLE);
    // Initialize USB communication
    Init_USB();
    // Initialize CAN communication. Enable ext. interrupt INT0 on received message
    Init_CAN();
    sei();

    while (TRUE){
        if calculate_flag == TRUE {
            can_load = can_count/PERIOD; // Calculate CAN messages receved per second
            can_count = 0;
            calculate_flag = FALSE;
            USB_SendCANLoad(can_load); // Send the calculated CAN load to PC over USB
        }
        // Do other things here
    }
}

ISR(INT0_vect) { // Triggered by CAN interrupt (on received message only)
    can_count++;
}

ISR(TIMER1_COMPA_vect) { // Triggers when counter1 equals 1024 (every 250 ms)
    TCNT1 = 0;           // Reset counter
    calculate_flag = TRUE;  // Flag new calculation of CAN message load
}
```

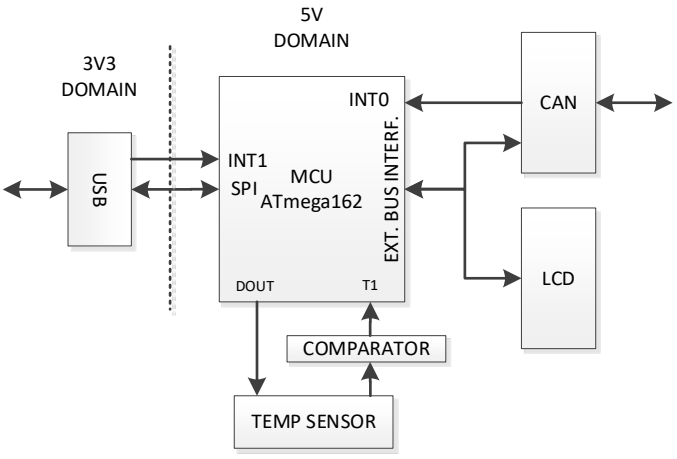
Problem 2. (50 %)

The hardware developed in Problem 1 can also serve as the basis for a CAN node operating in a distributed monitoring and control system. Assume that you are hired to design a temperature sensing CAN node that transmits CAN messages containing a temperature measurement every second. The temperature measurement is made possible by including the LMT01 temperature sensor described in Figure 4 to your design. The CAN node should also include a small embedded display. The device's detailed specifications are given below:

- The system should be built up around a microcontroller of type AVR ATmega162 as shown in Figure 5
- The system should include the CAN and USB controllers as specified in Problem 1
- The device should include LMT01 temperature sensors as described in Figure 4 that should be powered by digital output pin from the microcontroller and read by the T1 external clock input pin of T/C 1
- The display features a parallel bus interface and operates as a purely memory-mapped I/O that can be accessed by writing/reading to/from a linear 1 kilobyte memory inside the display
- A separate 5 V power supply is available in the CAN node

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. Use the T/C 3 output compare interrupts A and B to time/trigger LMT01 sensor readings and sending of temperature measurements over the CAN bus (ISR(TIMER3_COMPA_vect) and ISR(TIMER3_COMPB_vect)). Furthermore, use T/C 1 to count the pulses from the LMT01 via the microcontroller's external clock input pin T1 (see Figures 1 and 5). Assume that the pulses are converted to 0-5 V logic pulses by means

of a comparator.
Use C/pseudo-code and outline a program that reads temperatures and transmit them as CAN messages every second.

(15%)

We use T/C 3 OCR3A to time the sensor conversion and pulse reading interval (104 ms), and T/C 3 OCR3B to generate the 1 second sample and send interval. The idea is to let the OC3B interrupt turn on the sensor and schedule an OC3A interrupt 104 ms later when the sensor conversion and pulse transmission is completed. The OC3A interrupt then reads the pulse count, calculates the temperature and raises a flag that will dispatch a CAN temperature message from the main loop.

With the system clock of 4194304 Hz and prescaler 1024 given in 1 a), we get the compare register values:

OCR3A = (4194304 Hz/1024)*0.104s - 1 = 425.984 ≈ 426

OCR3B = (4194304 Hz/1024)*1s – 1 = 4095

The current pulses from the LMT01 will be converted to logic voltage pulses by an external analog comparator that drives the external clock input pin T1 of T/C 1. The sensor value (pulse count) will therefore be stored in TCNT1 after each conversion. Furthermore, Port B bit 0 is configured as a digital out pin and will power the LMT01 when the pin is set.

```
// Pseudo-code can be used for details and things not fully documented in exam text
#include <stdint.h>
#include <avr/interrupt.h>
#include “timer.h” // Indicate that we have a driver for the T/C units
#include “CAN.h” // Indicate that we have a CAN driver

volatile float temperature = 0.0f;
uint_8 can_send = FALSE;

void main {
    // Make port B bit 0 a digital output and make sure sensor is turned off
    DDRB |= (1<<DDB0);
    PORTB &= ~(1<<PORTB0);

    // Initialize CAN communication.
    Init_CAN();

    // Reset and initialize timer 1 to normal mode, no prescaler and with T1 pin
    // as clock source.
    init_Timer1(MODE=NORMAL, PRESCALER=NONE, CLOCKSOURCE=T1, INTERRUPTS=DISABLE);

    // Reset and initialize timer 3 to normal mode and the output compare registers to
    // their respective values. Interrupt enabled for compare match on B. Start timer
    init_Timer3(MODE=NORMAL, PRESCALER=1024, CLOCKSOURCE = SYSCLK, OCR3A=426, OCR3B=4095,
                OC3A_INTERRUPT=DISABLE, OC3B_INTERRUPT=ENABLE);
    sei();

    while (TRUE){
        if can_send == TRUE {
            CAN_SendTemperature(temperature); // Convert float to byte array and
                                              // transfer to CAN controller for sending
            can_send = FALSE;
        }
        // Do other stuff
    }
}

ISR(TIMER3_COMPB_vect) {            // 1 s boundary reached, prepare reading sensor…
    TCNT3 = 0;                      // Reset T/C 3, next OC3B interrupt in 1 s
    ETIMSK |= (1<<OCIE3A);          // Enable OC3A interrupt; will fire in 104 ms
    TCNT1 = 0;                      // Reset T/C 1, ready for counting pulses on T1 pin
    PORTB |= (1<<PORTB0);           // Turn on sensor
}

ISR(TIMER3_COMPA_vect) {            // 104 ms sensor conversion and reading cycle finished
    PORTB &= ~(1<<PORTB0);          // Turn off sensor
    ETIMSK &= ~(1<<OCIE3A);         // Disable OC3A interrupt (to be sure…)
    temperature = (TCNT1/4096)*256 – 50; // Calculate temperature; pulse count in TCNT1
                                    // Be cautious about float calculations in ISRs!
    can_send = TRUE;                // Let main loop send the CAN message to keep ISR lean
}
```

c. Assume that the base address of the display should be put at 0x4000 and the CAN controller at 0x8000 as before. Show how you would organize the address space of the computer in the simplest possible way, and derive the associated decoding logic (remember that the 1280 lowest addresses (0x0000 - 0x04FF) of the ATmega162 are reserved).

(10%)

According to the specification, three ‘devices’ must be accommodated in the 64 kB (16 bit) address space of the ATmega162: CAN, LCD and AVR reserved memory space. Decoding of three memory ranges requires the two MSB bits of the address bus. The CAN controller is specified to have base address 0x8000 and will be allocated the upper half of the address

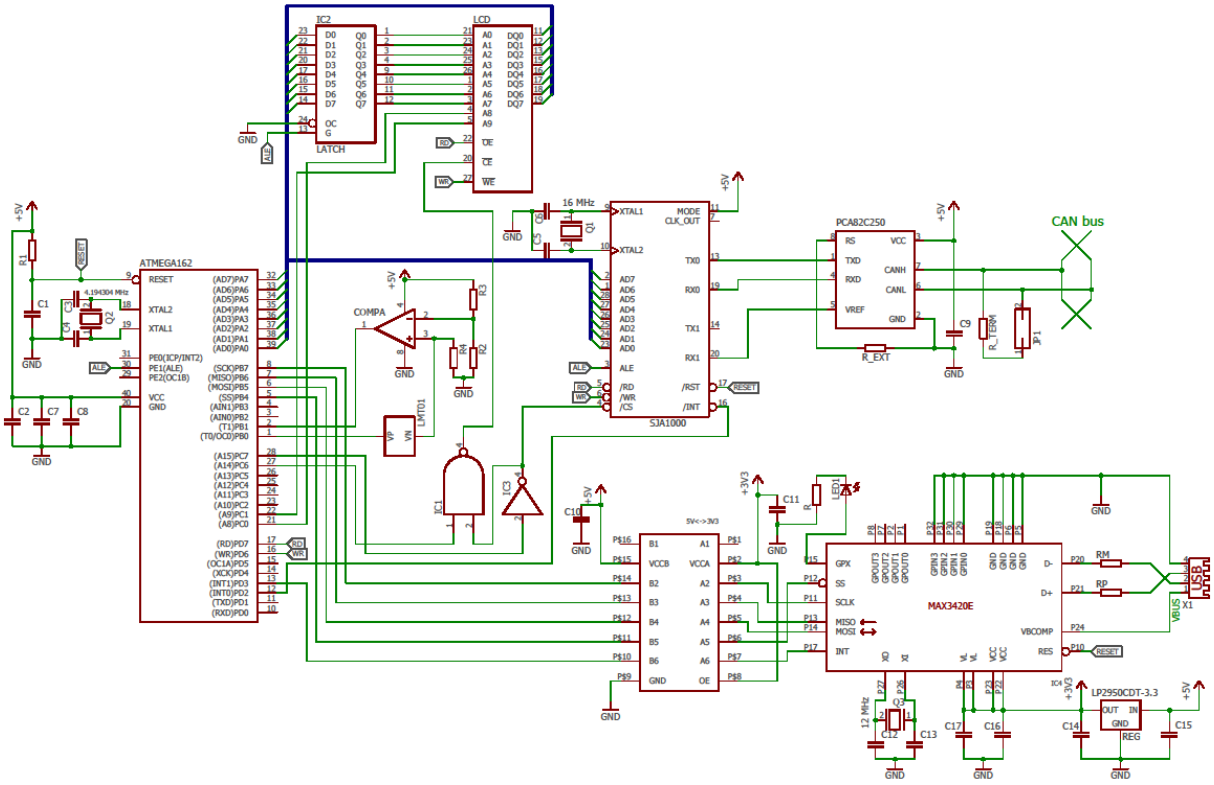
space using A_{15} (MSB) of the address bus to make decoding simple. The lower half of the address space can further be divided into two equally sized separate address ranges to accommodate for the remaining two devices by using A_{14}, with LCD at base address 0x4000 as specified.

AVR reserved	0x0000 – 0x3FFF (16kB)
LCD	0x4000 – 0x7FFF (16kB)
CAN	0x8000 – 0xFFFF (32kB)

Active low logic: $/CS_{CAN} = A_{15}$, $/CS_{LCD} = /A_{15}A_{14}$

d. Draw and explain a circuit schematic that shows how the components of the system are connected together (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%)



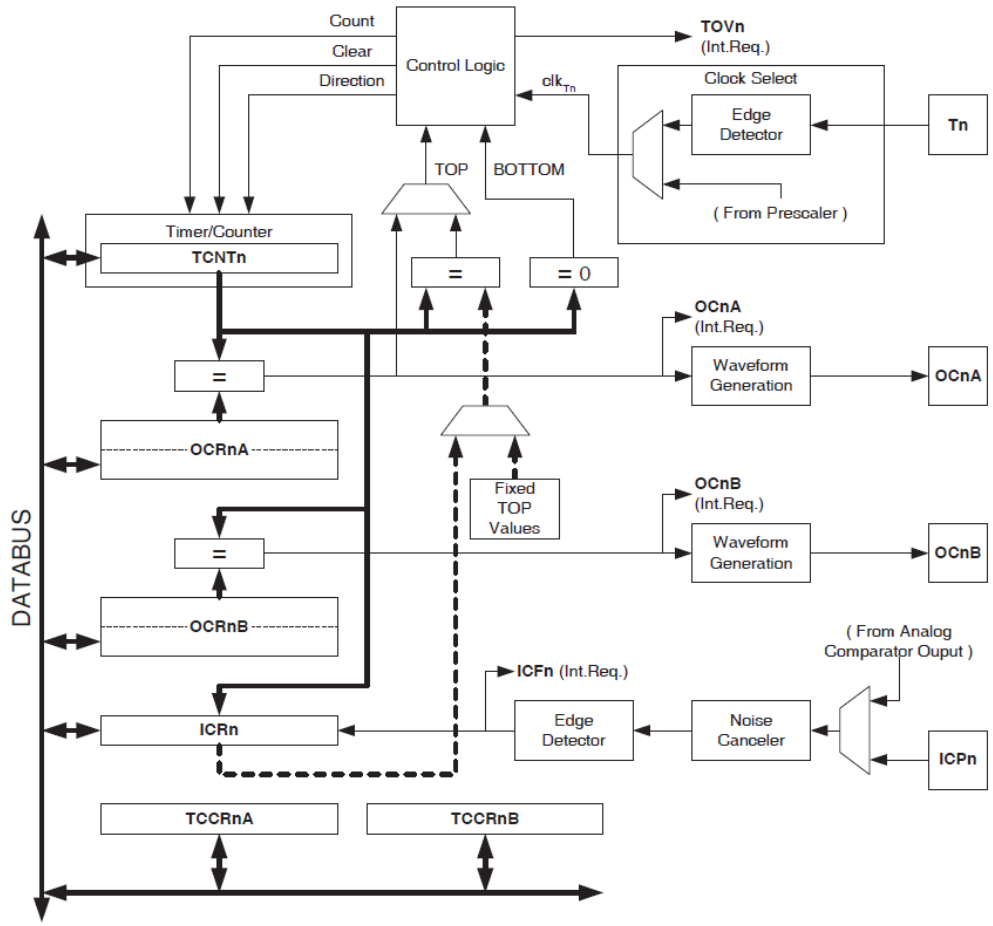


Figure 1. Diagram showing Timer/Counter 1 and 3 in the AVR ATmega162.

- 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, via a prescaled (divided) version of the system clock, or via a clock signal on pin Tn. The selectable prescaler values are 8, 64, 256 or 1024.
- 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 of the ICPn pin signal triggers the capture (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 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.

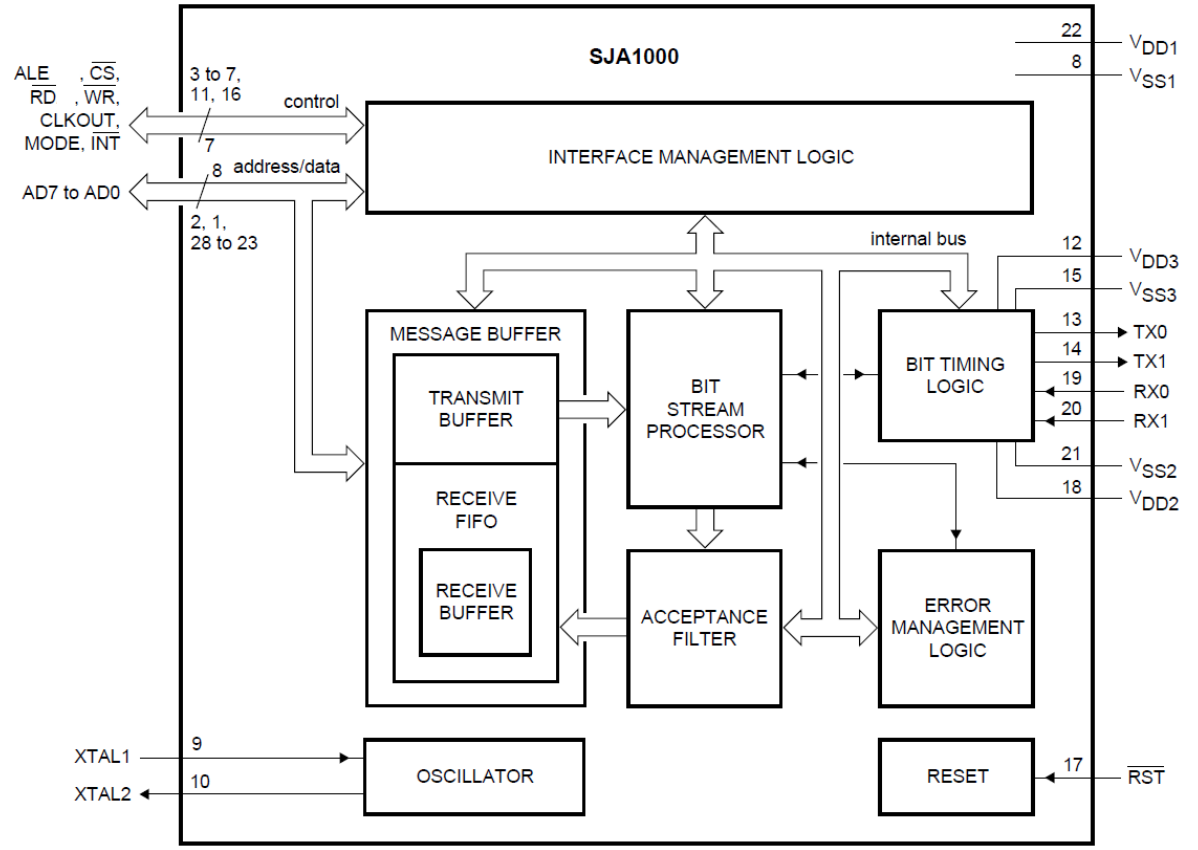


Figure 2. Diagram showing the SJA1000 CAN controller (CAN2.0B).

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, $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$ are control signals controlling bus transactions with the host microcontroller
- $\overline{INT}$ 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)
- $\overline{RST}$ 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

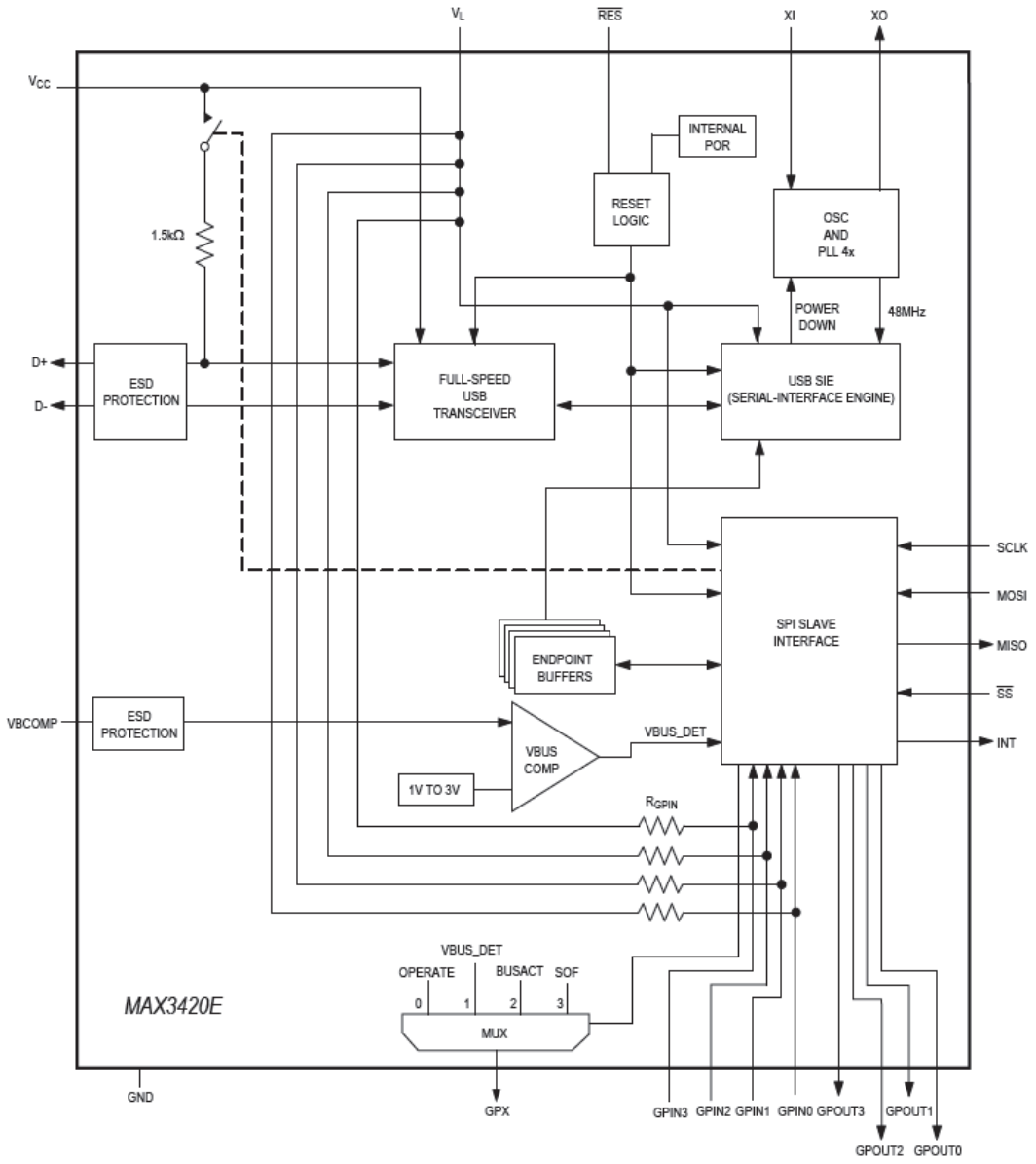


Figure 3. Diagram showing the MAX3420E USB 2.0 peripheral controller with SPI interface.

The MAX3420E includes endpoints 0-3 (EP0 IN/OUT (control), EP1 OUT (bulk or interrupt), EP2 IN (bulk or interrupt) and EP3 IN (bulk or interrupt)). The chip runs on a 3.3 V power supply and requires a voltage level converter to interface with 5 V logic devices.

- D+ and D- are the USB data signals
- MOSI, MISO, SCLK and $\overline{SS}$ comprise the SPI interface
- INT is an interrupt signal with programmable polarity generated by the USB controller
- $\overline{RES}$ is an active low reset signal
- GPIN and GPOUT are general purpose digital I/O signals
- GPX is a digital output signal indicating the USB state
- XI and XO are terminals for a 12 MHz crystal
- VBCOMP is a comparator input to detect valid USB power (V_{BUS})
- V_{CC} and GND are 3.3 V power supply terminals, and V_L are logic voltage level input (should be connect to V_{CC} here)

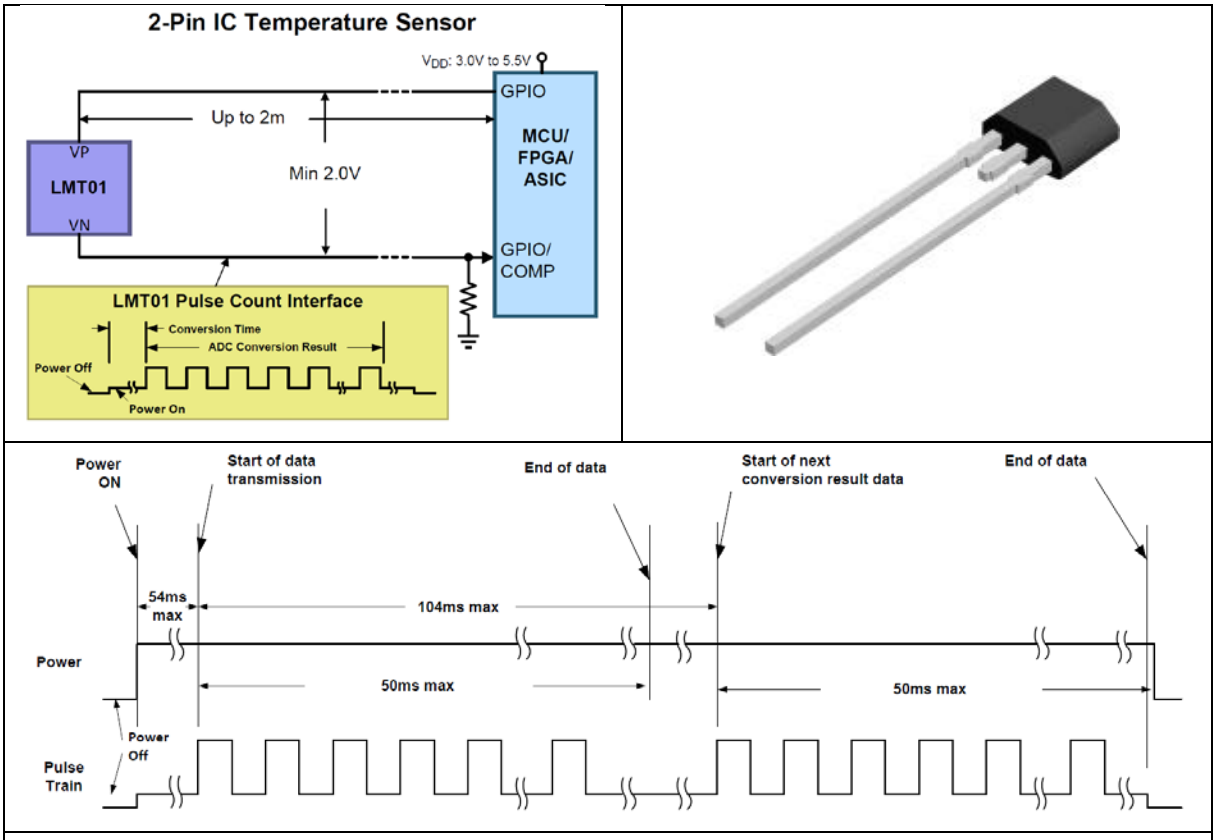


Figure 4. The LMT01 is a 2-pin digital current output temperature sensor with pulse count interface.

As can be seen from the timing diagram above, after power on is applied to the VP-pin, the sensor first enters a conversion phase that lasts maximum 54 ms. The sensor then outputs a train of current pulses on the VN-pin over the next 50 ms where the number of pulses (pulse count = *PC*) within the data transmission interval reflects the ambient temperature. The relationship between temperature and pulse count is given as:

$$T[°C] = \frac{PC}{4096} \times 256 - 50$$

After the sensor is powered up, it transmits a very low current of 34µA while the part executes the temperature to digital conversion. When the temperature-to-digital conversion is complete, the sensor starts to transmit the pulse train that toggles from the low current of 34µA to a high current level of 125µA. The output pulse frequency is typically 88kHz and the pulse duty cycle is 50%. The LMT01 outputs minimum 1 pulse and maximum 4095 pulses. Each pulse has a weight of 0.0625°C. If power is continuously applied, the conversion and pulse train transmission will repeat every 104ms.

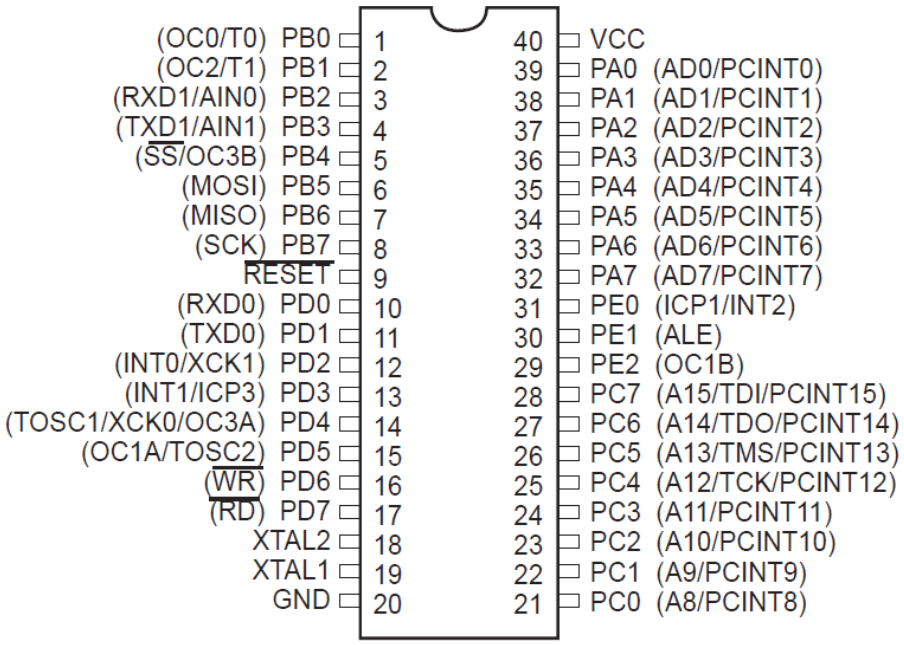


Figure 5. Atmel AVR ATmega162.