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: Tuesday 6th December 2016

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 question may have several parts.

Answers should be concise.

Exam counts 60% of final grade.

Language: English

Number of pages (front page excluded): 4

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

Problem 1. (30 %)

a. Figure 1 shows the diagram of the 16 bit Timer/Counter unit of the Atmel AVR ATmega162 microcontroller. Suppose that the system clock is driven by an external crystal oscillator at the standard frequency 6.5536MHz.
Assume that the timer is set to normal mode and clocked directly by the system clock (no prescaler). Show that the frequency of the overflow interrupt (TOV1) is 100Hz.

(5%)

TCNT1 is 16 bit and will therefore overflow after 2^{16} counts. No prescaler. clk_{T1} = 6553600Hz = $100*2^{16}$ Hz.
We get: $f_{TOV1} = clk_{T1}/2^{16} = 100*2^{16}/2^{16} = 100$Hz

b. Suppose that your application is required to run two periodical tasks, PID_update() and LCD_update() at 12.5Hz and 50Hz, respectively. Use the setup from a) and show using C/pseudo-code how the TOV1 interrupt handler and main() should be structured. The tasks should run as unaffected as possible from other application code.

(10%)

Period of TOV1 interrupt = 1/100Hz = 10 ms
Period of PID_update() = 1/12.5Hz = 80 ms
Period of LCD_update() = 1/50Hz = 20 ms

This means that the TOV1 interrupt needs to invoke LCD_update() and PID_update() every 2nd and 8th time it gets triggered, respectively.

```
uchar8_t pid_counter = 0;
uchar8_t lcd_counter = 0;
void main {
     init_Timer1(NORMAL, NO_PRESCALER);
     init_PID();
     init_LCD();
     while (TRUE){
             // Do other stuff
     }
}
```

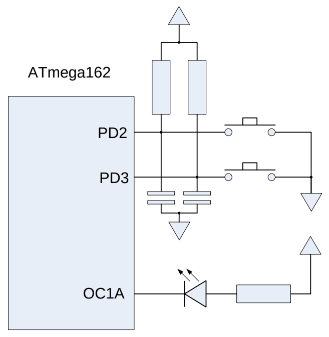
```
ISR(TIMER1_OVF) {

     lcd_counter++;
     if(lcd_counter >= 2) {
             LCD_update();
             lcd_counter = 0;
     }

     pid_counter++;
     if(pid_counter >= 8) {
             PID_update();
             pid_counter = 0;
     }
}
```

Placing the update tasks in the main-loop triggered by flags from the ISR will violate the requirement that the tasks should run as unaffected as possible from other application code. However, it should be commented that the max execution time of the tasks must never exceed 10 ms with some margin.

c. The timer in Figure 1 can be used in PWM mode to control dimming of LEDs and other light sources. Sketch a simple circuit for dimming a LED up and down using an AVR ATmega162 together with two push buttons (see Figure 3 for the pinout of ATmega162). Let TOP equal 0x00FF (fixed), let prescaler equal 1024, and use the same crystal oscillator as in a). Let OCR1A determine the PWM duty cycle.
Calculate the frequency of the PWM square wave generated on the OC1A pin.
Write a simple program in C/pseudo-code implementing the LED dimmer. The program only needs to show the main structure of main() and the interrupt handler.

(15%)

We can simply connect the two push buttons to two of the MCU's general purpose digital input pins and let them increment/decrement a dimmer variable that is transferred to the output compare match register OCR1A. A capacitor to ground will smooth the button signals. OCR1A determines the PWM duty cycle of the OC1A pin that drives the LED (ds = 100*OCR1A/TOP %). Reading of the buttons can be put in the main loop while the TOV1 interrupt updates the OCR1A with the current value of the dimmer variable. Interrupt driven input pins and direct manipulation of OCR1A also give nice solution here.



With prescaler = 2^{10}, $clk_{T1} = 100*2^{16}/2^{10}$ Hz = 6400Hz. With TOP = 0x00FF, the frequency of the PWM square wave gets 6400Hz/256 = 25 Hz.

```
#define TOP 0x00FF
uint16_t dimmer = TOP/2;
void main {
    init_timer1(PWM, OC1A, NON_INV, PRESCALER_1024, TOP);
    while (TRUE){
            if ((PD2 == LOW) && (dimmer < TOP)) dimmer++;
            if ((PD3 == LOW) && (dimmer > 0)) dimmer--;
            // Limit button bouncing a bit
            delay_ms(10);
    }
}
```

```
ISR(TIMER1_OVF) {
    // Update between
    // PWM cycles
    OCR1A = dimmer;
}
```

```
#define TOP 0x00FF
void main {
    init_timer1(PWM, OC1A, NON_INV, PRESCALER_1024, TOP);
    while (TRUE){
            // Do other stuff
    }
}
```

```
ISR(INT0) {
    if (OCR1A > 0)
            OCR1A--;
}

ISR(INT1) {
    if (OCR1A < 0XFF)
            OCR1A++;
}
```

<u>Problem 2.</u> (30 %)

a. Why is the capacitor shown in the circuit in the upper left panel of Figure 2 needed?

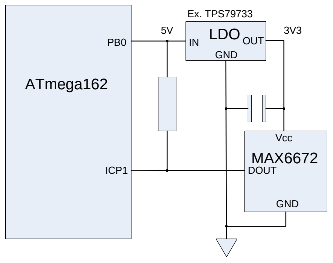
(5%)

It is a decoupling capacitor and it is needed to suppress noise on the supply input of the sensor caused by stray inductance in the supply wires in combination with current transients generated by switching transistors in the digital circuit.

b. The sensor given in Figure 2 should be interfaced to and powered by an ATmega162 running on 5V. Assume that the sensor requires a 3.3V voltage supply. Show how this can be done using a voltage regulator and the open drain output of the sensor. Explain what type of regulator you would use.

(10%)

The sensor is low power and can easily be powered by a general purpose digital output pin on the MCU. However, the 5V output must be down-regulated to 3.3V before it can be applied to the sensor supply pin. A simple and safe solution is to use a linear voltage regulator for this. A switched mode buck regulator would be a less ideal solution ("overkill" and more noisy). The small 1.7V voltage difference between the two circuits suggests that a regulator of the LDO type should be used to avoid problems with the rather high dropout voltage of a standard regulator. The sensor features an open drain output which can be pulled up conveniently to 5V before connected to the input pin of the MCU.



c. To what pin should the sensor be connected on the ATmega162 in order to use timer/counter 1 for reading the sensor?
Assume you have the same crystal oscillator as in 1.a). What is the largest prescaler value you can use in order to get at least 9 bit time resolution over the PWM period (t_1+t_2)?
Write a simple interrupt handler for the timer using C/pseudo-code that continuously calculates the temperature.

(15%)

The sensor output should be connected to the input capture pin (ICP1) of the MCU. Timer1 can then be used to timestamp the rising and falling edges of the sensor signal and thereby measuring t_1 and t_2 needed to calculate the temperature. Note that the measurement should exploit the ratiometric nature of the sensor, meaning that the temperature calculation should not depend on the frequency of the PWM signal (which may be inaccurate), just the ratio of t_1 and t_2.

The sensor generates a PWM square wave of 1.4kHz, giving a PWM period of t = 1/1400Hz = 714.3us. Minimum 9 bit resolution implies that the timer clock cycle must at least be $714.3/2^9$ = 1.395us. That gives a prescaler value $p < 1.395\text{us} * 6.5536\text{MHz} \approx 9$. The closest possible prescaler giving the required resolution is therefore 8 (giving 9.2 bit resolution).

The Timer1 input capture register ICR1 and the input capture interrupt ICI should be employed to make the measurement. The ISR manipulates the edge selection control bit ICES in the TCCR1B register to alternatingly trigger on the rising and falling edge of the sensor signal. Floating point calculations should be avoided in the ISR and the calculation is therefore delegated to the main loop using a flag.

```
ISR(TIMER1_CAPT) {
    if(ICES == 1) {                 // ICP1 rising edge
        if(t2 > 0) {                // Only calculate if t2 already measured
            t1 = ICR1 – t2;         // Capture t1
            start_calc = TRUE;      // Tell main loop to calculate temperature
        }
        TCNT1 = 0;                        // Let timer start at 0 for each new PWM cycle
        ICES = 0;                         // Next capture interrupt on falling edge
    }
    else if(ICES == 0) {                  // ICP1 falling edge
        t2 = ICR1;                        // Capture t2
        ICES = 1;                         // Next capture interrupt on rising edge
    }
}
```

```
uint16_t t1,t2 = 0;
uint8_t start_calc = FALSE;
float t;
void main {
    // Start timer 1 in normal mode w/ prescaler = 8 and init capture interrupt w/first
    // interrupt on rising edge (ICES = 1)
    init_timer1(NORMAL, PRESCALER_8, CAPTURE_RISING);
    poweron_sensor();
    while (TRUE){
        if(start_calc) {
            t = calc_temp(t1, t2);
            display(t);
            start_calc = FALSE;
        }
    }
}
```

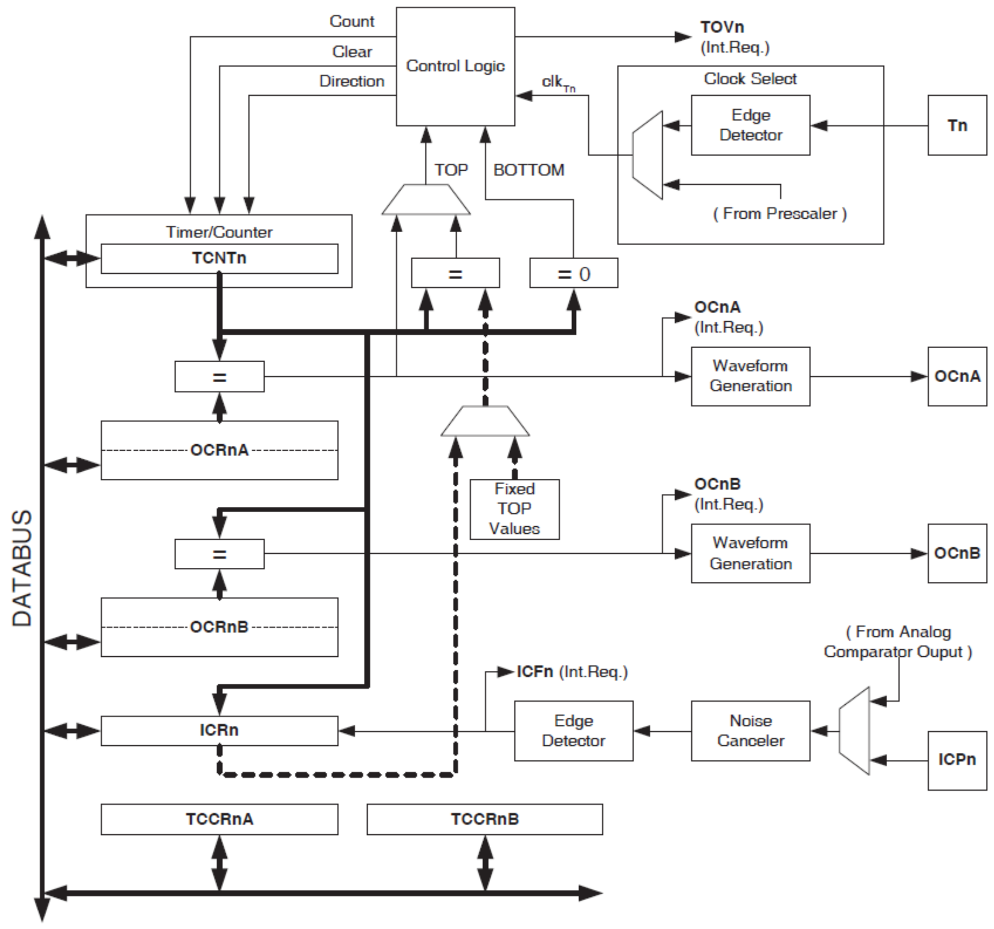


Figure 1. Diagram showing Timer n in the AVR ATmega162.

- In Timer 1 (n = 1), the TCNT1, OCR1A, OCR1B and ICR1 registers are all 16 bit.
- The Timer 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 T1. The selectable prescaler values are 8, 64, 256 or 1024.

- In *Normal* mode, TCNT1 increments by one for each clock pulse (0x0000→0x001→…→0xFFFF→0x0000→…). The interrupt signal TOV1 (timer overflow) is generated every time the TCNT1 overruns from 0xFFFF to 0x0000.
- The ICR1 (input capture register) is used in normal mode to capture/timestamp external events in terms of rising and falling edges on the ICP1 pin. When a rising or falling edge is detected on the ICP1 pin, the current value of the timer register TCNT1 gets loaded immediately into ICR1 and the input capture interrupt signal ICF1 is generated. The ICES1 bit of the timer control register TCCR1B selects which edge on the ICP1 pin that will trigger a capture (0 = falling edge, 1 = rising edge).
- The OCR1A and OCR1B are two output compare registers used to generate the interrupt signals OC1A and OC1B as well as waveforms on the OC1A and OC1B pins when the timer TCNT1 matches their content.
- When Timer 1 is set to *Clear Timer on Compare (CTC) mode*, TCNT1 will be cleared to zero when a match with the content of OCR1A is detected and the interrupt signal OC1A will be generated and the OC1A pin will be toggled.
- When Timer 1 is set to *Fast PWM mode*, TCNT1 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 OCR1A or ICR1. The OC1x pin is set on a compare match between TCNT1 and OCR1x, and cleared at TOP. A TOV1 interrupt is generated when TCNT1 rolls over from TOP to BOTTOM.

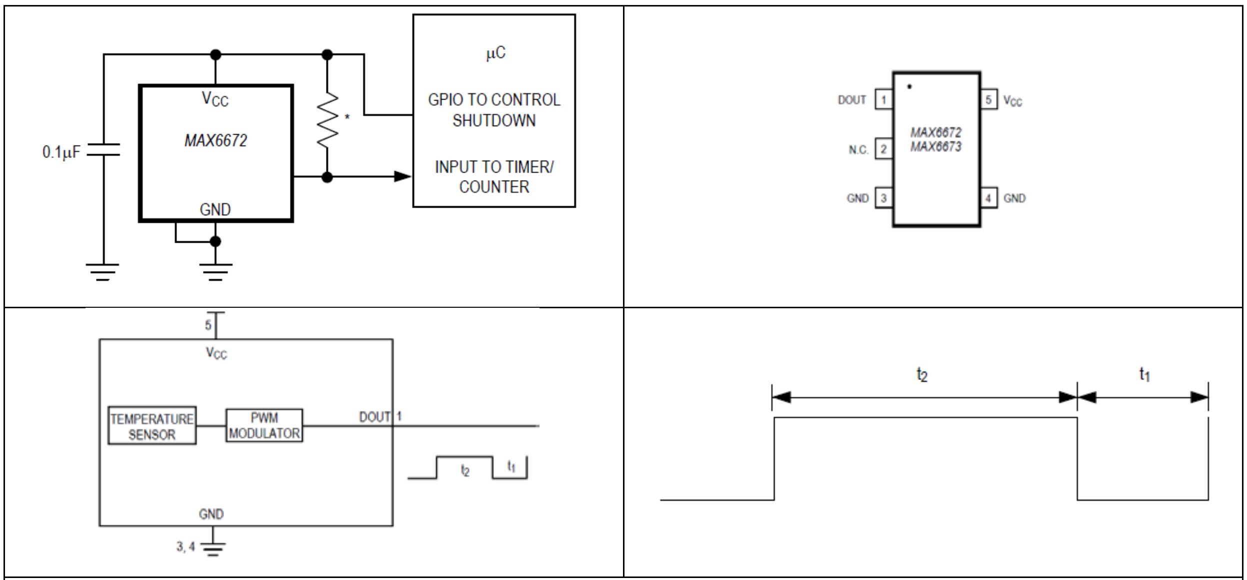


Figure 2. MAX6672 is a 5-pin low-current temperature sensor with a single wire output. The sensor converts the ambient temperature into a ratiometric 1.4kHz PWM output signal. The temperature is obtained with the formula:

$$Temperature\ (°C) = -200\left(0.85 - \frac{t_1}{t_2}\right) + 425\frac{t_1}{t_2} - 273$$

, where t_1 and t_2 represent the off- and on-times of the PWM square wave.

The example circuit in the upper left panel shows a situation where the sensor is powered directly from an MCU digital output pin, enabling the MCU to switch the sensor on and off to save power. Note that the sensor output is open drain.

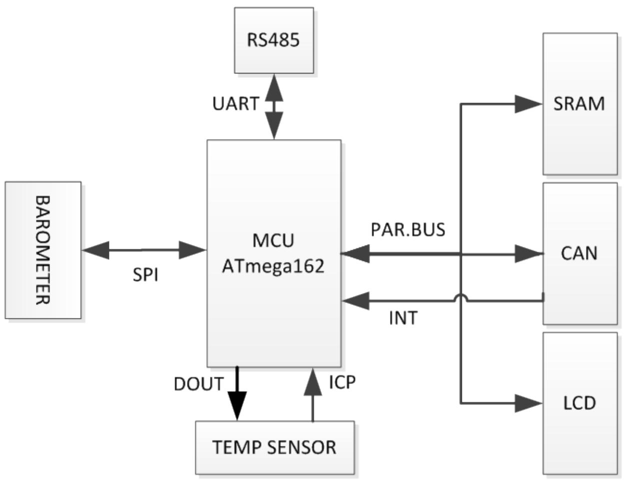
Problem 3. (40 %)

An embedded computer that implements a sensor instrument for monitoring of temperature and barometric pressure in a factory environment shall be designed. The instrument shall be based on the temperature sensor given in Figure 2, an ATmega162 microcontroller and some additional components. In addition to the barometric sensor, the instrument features a small display and communication interfaces for CAN and RS-485. The instrument's detailed specification is given below:

- The system should be built up around a microcontroller of type AVR ATmega162 as shown in Figure 3.
- ATmega162 features a relatively small amount of internal SRAM and requires inclusion of an extra external 32 kByte SRAM.
- The sensor given in Figure 2 should be included to measure temperature.
- The barometric sensor features an SPI interface.
- The CAN interface should be implemented using a stand-alone CAN controller circuit featuring a parallel bus interface, meaning that it should be connected to the microcontroller's external data- and address bus as memory mapped I/O. The CAN controller's internal register file (message buffers + control and status registers) requires 512 addresses in the address space. The CAN controller must be able interrupt the microcontroller.
- An RS-485 transceiver should be connected to one of the MCU's UARTs.
- The display has 4 lines x 40 characters and features a parallel bus interface. The display operates as a pure memory mapped I/O and can be accessed by writing/reading to/from a linear 160 bytes memory inside the display where the addresses corresponds to the character positions on the display (first character on the first line has address 0, first character on the second line has address 40, etc.). Moreover, the display features ten 8-bit data-, status- and control registers, requiring a total of 170 bytes of the address space.

a. If necessary, make your own reasonable assumptions that will make the system function properly, and describe it in terms of a high-level block diagram (not a detailed circuit schematic) with some textual comments or explanations when required. 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 labels and simple arrows (not detailed bus/signal connections here).

   (10%)



b. Assume that the base address of the display should put at 0x2000. Explain 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).

(15%)

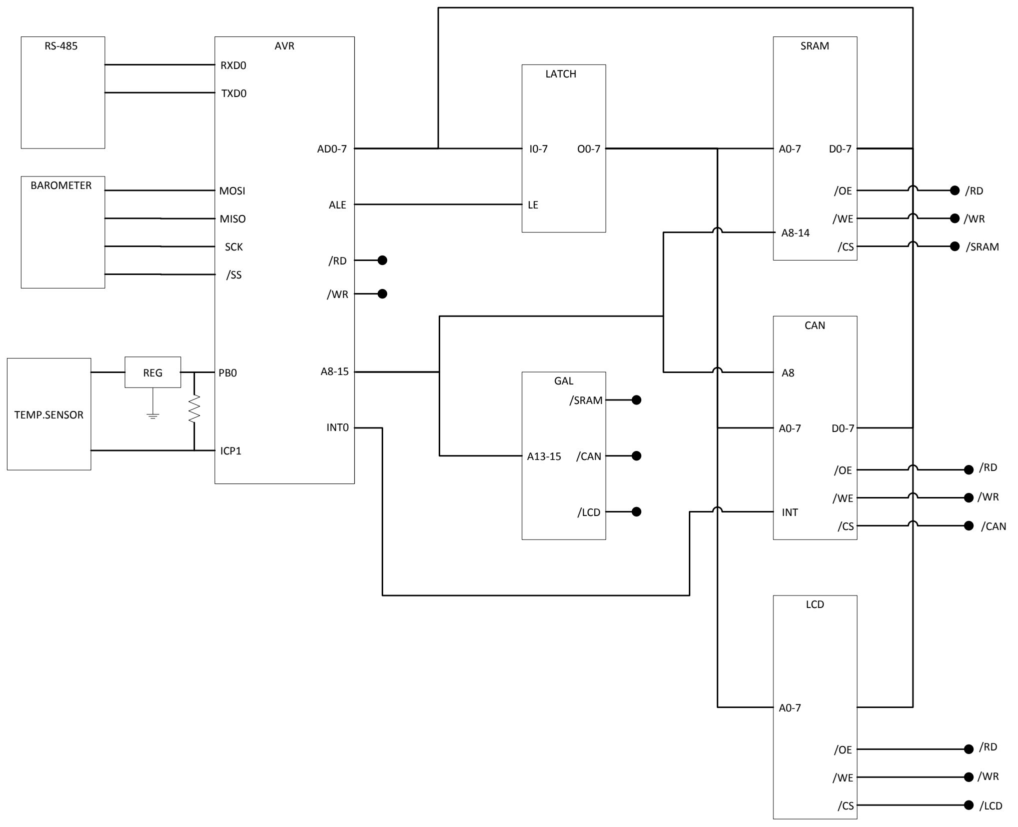
According to the specification, four devices must be made room for in the 64 kB (16 bit) address space of the MCU: SRAM, CAN, LCD and AVR (reserved addresses). The SRAM, taking half of the memory space, can be located to the upper half of the address space. Furthermore, according to the partial decoding approach, the lower half of the address space can be divided into four equally sized separate address ranges to accommodate for the remaining three devices. Decoding: Let A_{15} MSB of the address bus select SRAM with base address 0x8000, and the two next bits $A_{14}A_{13}$ select one of the four lower ranges. 0x0000 is reserved for the AVR, making 0x2000, 0x4000 and 0x6000 possible base addresses for the LCD and CAN. LCD is specified to base address 0x2000, making either 0x4000 and 0x6000 available base address for CAN (CAN could alternatively be decoded by using only $A_{15}A_{14}$ giving it 16 kB in the range 0x4000-0x7FFF).

Reserved AVR 8 kB (0x0000 – 0x1FFF)
LCD 8 kB (0x2000 – 0x3FFF)
CAN 8 kB (0x4000 – 0x5FFF)
Free 8 kB (0x6000 – 0x7FFF)
SRAM 32 kB (0x8000 – 0xFFFF)

Active low logic: $CS_{RAM} = A_{15}$, $CS_{LCD} = /A_{15}/A_{14}A_{13}$, $CS_{CAN} = /A_{15}A_{14}/A_{13}$

c. 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%)



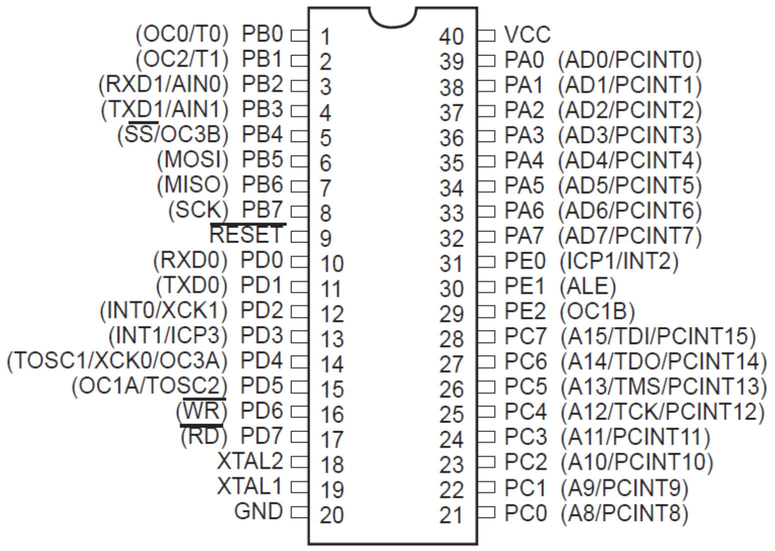


Figure 3. Atmel AVR ATmega162.