RĪGAS TEHNISKĀ UNIVERSITĀTE

DATORZINĀTNES UN INFORMĀCIJAS TEHNOLOĢIJAS FAKULTĀTE

DATORVADĪBAS, AUTOMĀTIKAS UN DATORTEHNIKAS INSTITŪTS

Datoru tīklu un sistēmas tehnoloģijas katedra

Mikroprocesoru tehnika

Laboratorijas darbi

Asist. R. Taranovs
Laborants G. Miezītis
Profesors V.Zagurskis

Rīga 2012

Ievads

Laboratorijas darbu apraksts paredzēts priekšmeta „Mikroprocesoru tehnika" apguvei. Galvenais mērķis ir iepazīstināt studentus mikrokontrolleru galvenajām sastāvdaļā, to programmēšanu un veidot izpratni par mikrokontrollera darbību, uzbūvi un pielietojumiem.

Laboratorijas darbi bāzējas uz mikrokontrolleri ATmega128, kas izvietots uz izstrādes CharonII (to tuvāks apskats būs pirmajā laboratorijas darbā) un tiks programmēts izmantojot AVR Studio 4. Laboratorijas darbu ietvaros tiks aplūkotas tādas pamata lietas kā mikrokontrollera ciparu ieejas un izejas, taimeri/skaitītāji, atmiņa (rādītāji), analogs cipars pārveidošana - ACP, sargtaimeris (*watchdog timer*) un komunikācijas interfeiss USART.

Kopumā ir paredzēti seši laboratorijas darbi, kas saistīti ar iepriekš uzskaitītājām mikrokontrolleru komponentēm. Pēc laboratorijas darba pabeigšanas ir nepieciešams sagatavot tā atskaiti un aizstāvēt. Aizstāvēšanas laikā uzdos teorētiskus jautājumus par attiecīgo tēmu un atkarībā no atbildēm laboratorijas darbs tiks novērtēts ar atzīmi.

Veiksmi laboratorijas darbu izpildē!

Programmēšanas valoda C un mikrokontrolleri

Programmējot mikrokontrollerus C ir viena no ērtākajām valodām, jo tai ir vienkārša sintakse un ir pieejami bezmaksas kompilatori un izstrādes vides. Programmējot mikrokontrollerus ir pieejamas visas pamata C bibliotēkas (piem., *math, stdlib, stdint*) un daudzas papildus bibliotēkas, kas ir tieši saistītas ar mikrokontrolleriem (piem., *avr/io, avr/interrupt, util/delay*). Šīs bibliotēkas kļūst pieejamas pēc tam, kad ir uzinstalēts WinAVR. Pieņemot, ka students ir pazīstams ar C, tad tālāk tiks aplūkots C īpatnības, kas saistītas ar mikrokontrolleru programmēšanu.

Mainīgo pieraksta atšķirības

Vienu un to pašu mainīgos var pierakstīt dažādos veidos, proti, pie dažādām skaitļu bāzēm. Programmējot mikrokontrollerus visbiežāk mainīgos pieraksta **binārajā** vai **heksadecimālajā** pierakstā. Ja grib pierakstīt astoņu bitu mainīgo:

Binārais pieraksts: **0bxbbbbbbbb**

Heksadecimālais pieraksts: **0xbb**

Piemērs: Skaitli **169** var pierakstīt, kā **0xA9** un **0b10101001**. Kompilatoram visi šie pieraksti nozīmē vienu un to pašu.

Bitu loģiskās operācijas

Programmējot mikrokontrollerus bieži nepieciešams izmantot bitu loģikas operatorus, jo notiek darbošanās ar bitiem.

Tabula 1 Bitu loģikas operatori

Nosaukums	Apzīmējums	C operators
NE	NOT	~
VAI	OR	|
UN	AND	&
Izslēdzošais VAI	XOR	^

Piemēri:

NOT	
0	1
1	0

OR		
0	0	0
1	0	1
0	1	1
1	1	1

AND		
0	0	0
1	0	0
0	1	0
1	1	1

XOR		
0	0	0
1	0	1
0	1	1
1	1	0

Piemērs: ja A:0b10111001 un B:0b00001111, tad:

- !A => 01000110
- A|B =>10111111
- A&B =>00001001
- A^B =>10110110

Bitu bīdes operatori

Vēl bieži tiek izmantoti bitu bīdes operatori. Tie ir divi:

- Bitu nobīdīšana pa kreisi - **<<**
  A<<2 => 11100**100**
  B<<5 => 111**00000**
- Bitu nobīdīšana pa labi - **>>**
  A>>4 => **0000**1011
  B>>2 => **00**000011

Izbīdītie bieti tiek zaudēti un iebīdītas tiek **0**.

Darbs ar reģistriem

Programmējot mikrokontrollerus ļoti bieži nāksies saskarties ar dažādu reģistru vērtību mainīšanu un pieejamo reģistru meklēšanu. Lai varētu atrast kādi reģistri ir konkrētam mikrokontrollerim ir nepieciešams internetā atrast tā saucamo *datu lapu* (angl. Datasheet). Datu lapu var atrast ražotāja mājas lapā (www.atmel.com) vai izmantojot kādu no interneta meklētājiem (www.google.lv).

Datu lapā tiek uzskaitīti vis pieejamie reģistri un to nozīme. Reģistriem piekļūst izmantojot tā nosaukumu.

Piemērs:

PORTA=0b01010100; ADCSRA=0b00011100; TCCR0 = 0b00000010;

Lielākā daļa reģistru tiek izmantoti, lai iestādītu kā mikrokontrollerim ir jādarbojas, tāpēc bieži būs tādas situācijas, ka nepieciešams mainīt tikai viena bita stāvokli nemainot pārējos bitus. Lai to varētu panākt var pielietot jau iepriekš minētās bitu operācijas:

- Lai uzstādītu reģistra TCCR0 5. bitu uz **1** izmanto **VAI**, un var rīkoties sekojoši:
  - TCCR0=TCCR0|(0b00010000);
  - TCCR0|=0b00010000;
  - TCCR0|=0x10;
  - TCCR0|=(1<< 5);
  - TCCR0|=(1<< COM01);
- Lai uzstādītu reģistra TCCR0 5. bitu uz **0** izmanto **UN**, un var rīkoties sekojoši:
  - TCCR0=TCCR0&(0b11101111);
  - TCCR0&=0b11101111;
  - TCCR0&=0xEF;
  - TCCR0&=~(1<< 5);
  - TCCR0&=~(1<< COM01);

- Lai pārslēgtu reģistra TCCR0 5. bitu uz pretējo loģisko stāvokli (**0 uz 1 vai 1 uz 0**) izmanto **Izslēdzošo VAI**, un var rīkoties sekojoši:
    - TCCR0=TCCR0^(0b00010000);
    - TCCR0^=0b00010000;
    - TCCR0^=0x10;
    - TCCR0^=(1<< 5);
    - TCCR0^=(1<< COM01);

Piezīme! COM01 ir reģistra TCCR0 5. bita nosaukums, kas ir atrasts datu lapā. Vadības reģistriem katram bitam ir piešķirts konkrēts nosaukums. *iom128.h* failā ir nodefinēts, ka COM01 ir reģistra TCCR0 5. bits. Priekšrocība tam ir tāda, ka nav jāuztraucas par to, kurš bits reģistrā tas ir, pietiek zināt tikai reģistra lauka nosaukumu.

Pārtraukumi

Viena no biežāk pielietotājām lietām mikrokontrolleru programmēšanā ir pārtraukumu izmantošana – precīzāk pārtraukuma apstrādes funkciju (ISR) aprakstīšana. No programmēšanas viedokļa ISR ir funkcija, kas izpildīsies uz kādu ārēju vai iekšēju notikumu (piemēram, pogas nospiešanas, taimera/skaitītāja vērtība sasniedz maksimumu, ADC pabeidzis pārveidot analogo vērtību, utt.).

Lai pārtraukums vispār varētu izpildīties to vispirms ir nepieciešams atļaut. Pārtraukumu atļaušana notiek divos līmeņos: globāli un lokāli mikrokontrollera komponentēm. Globāli pārtraukumus atļauj ar funkciju *sei()*, bet aizliedz ar *sei()*. Patiesībā šīs funkcijas maina reģistra SREG 7.bitu, kas globāli atļauj/aizliedz pārtraukumus.

Lai atļautu, piemēram, taimera/skaitītāja 0 pārpildes (sasniegta maksimālā skaitītāja vērtība) pārtraukumu ir jādarbojas ar reģistra TIMSK 0.bitu. Parasti 1 – pārtraukums ir atļauts, 0 - pārtraukums ir aizliegts.

Ja pārtraukums ir atļauts tad nākošais solis ir definēt ISR, proti, ko mikrokontrolleris izpildīs reaģējot uz konkrēto notikumu. Izmantojot avr-gcc C kompilatoru, taimera/skaitītāja 0 pārpildes ISR[1] tiek definēts un aprakstīts sekojoši:

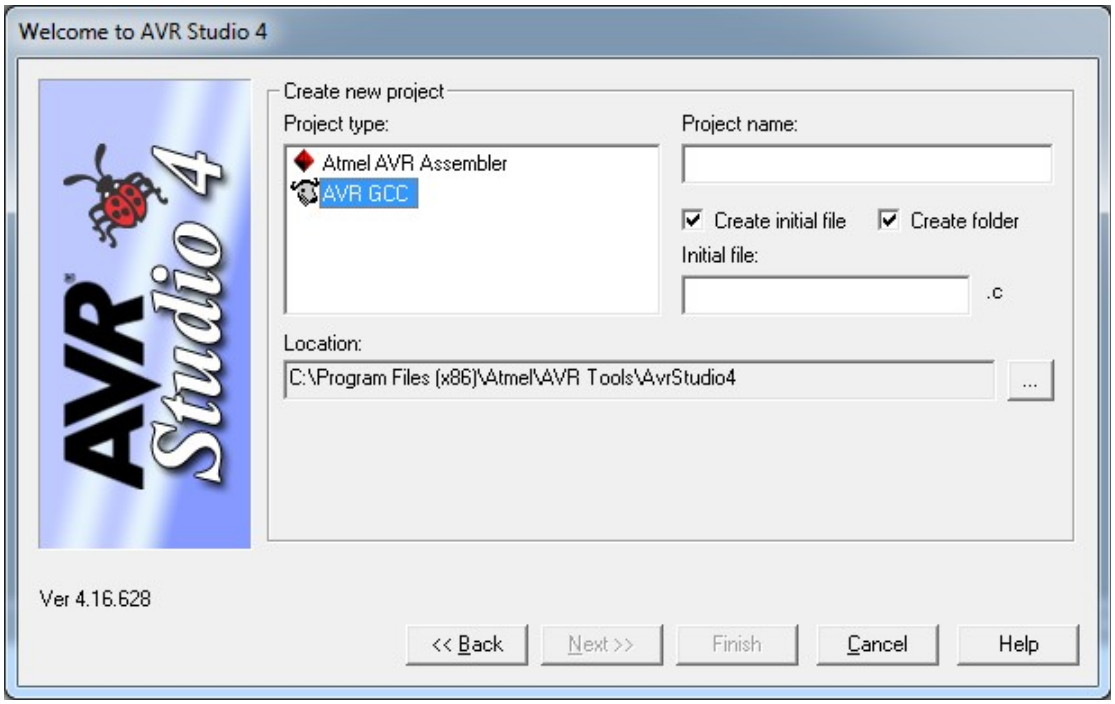
```
#include <avr/interrupt.h>

ISR(TIMER0_OVF_vect)
{
   // izpildāmais kods
}
```

Pārtraukums var darboties ar jebkuru globālo mainīgo un pārtraukumā var veikt jebkuru darbību, kas ir nepieciešama. Ja svarīga ir mikrokontrollera ātrdarbība tad ir ieteicams veidot ISR aprakstu pēc iespējas īsāku.

[1] http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Darbs ar AVR Studio 4

Palaižot lietojumprogrammu AVR Studio 4 atveras dialoga logs, kas piedāvā izveidot jaunu projektu vai atvērt jau eksistējošu. Pirmo reizi uzsākot darbu nepieciešams izvēlēties **New Project**. Ja šis dialoga logs neatveras nospiežat ***Project->New Project*** AVR Studio 4 logā. Atveras dialogs **Create new project (**Sk. Att. 1).



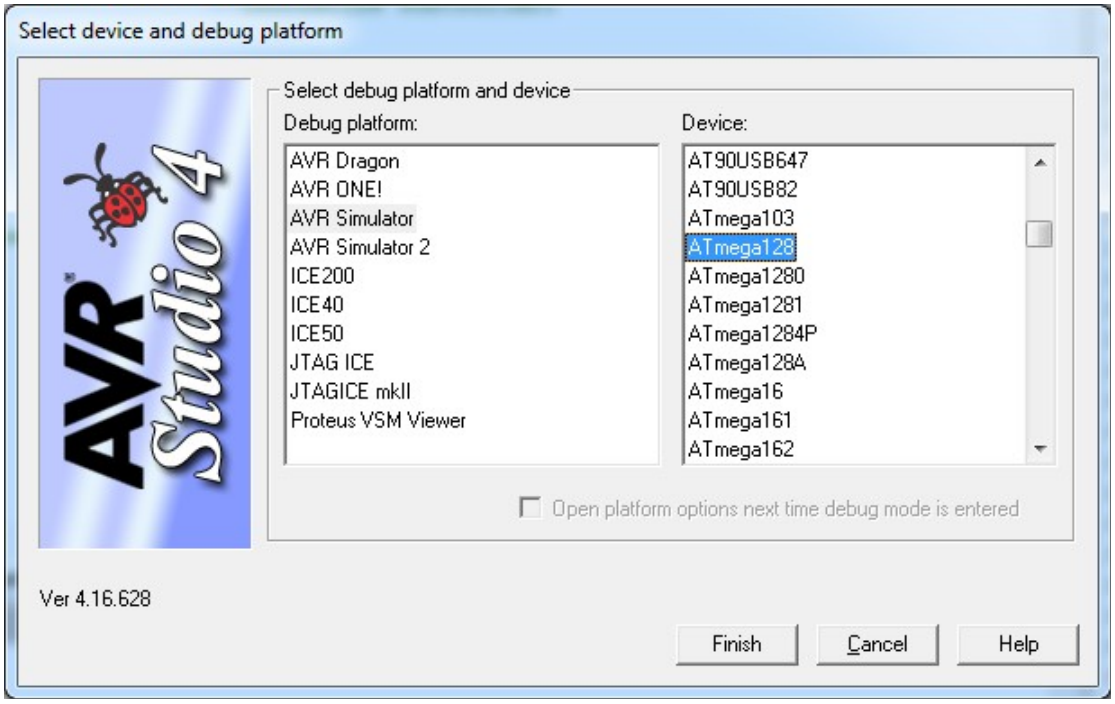
Att. 1 Jauna projekta veidošanas logs

Šajā dialogā ir jāizvēlas **AVR GCC** projekta tips, jo mikrokontrollera programmēšanā tiks izmantots *avr-gcc C-kompilators*. (Ja šāda izvēle nav iespējama, tad visticamākais, ka nav uzinstalēts WinAVR).

- Ievadiet projekta nosaukumu **Project name** laukā;
- Ievadiet sākotnējā programmas pirmteksta faila nosaukumu (pēc noklusējuma tas ir vienāds ar projekta nosaukumu);
- Izvēlieties projekta atrašanās vietu uz datora. Uz ekrāna izveidojiet mapīti ar savu vārdu un uzvārdu un to norādiet kā projekta atrašanās vietu;

Kad tas ir izdarīts tad var spiest pogu **Next**. Atveras dialoga logs, kurā jāizvēlas atkļūdošanas platforma un programmējamais mikrokontrolleris (Sk. Att. 2).

- Kā atkļūdošanas platformu jāizvēlas **AVR Simulator**;
- Kā programmējamo mikrokontrolleri ATmega128;



Att. 2 Atkļūdošanas platformas un mikrokontrollera izvēles dialogs

Līdz ar to projekta uzstādīšana ir beigusies. Un tālāk var spiest **Finish**. Tālāk ir iespējams veidot savu programmu.

Par piemēru ņemsim sekojošu programmas pirmtekstu:

```
/*****************************************************************/
#define F_CPU 14745600UL //Mikrokontrollera takts frekvences definēšana

/********* Standarta C un specialo AVR bibliotēku iekļaušana*********/
#include <avr/io.h>
#include <avr/iom128.h>
#include <avr/interrupt.h>
#include <math.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <util/delay.h>

/******************** Portu inicializācijas funkcija *****************/
void port_init(void)
{
 DDRA  = 0x00; //visas porta A līnijas uz IEvadi
 DDRB  = 0x00; //visas porta B līnijas uz IEvadi
 DDRC  = 0x00; //visas porta C līnijas uz IEvadi
 DDRD  = 0xFF; //visas porta D līnijas uz IZvadi
 DDRE  = 0x00; //visas porta E līnijas uz IEvadi
 DDRF  = 0x00; //visas porta F līnijas uz IEvadi
 DDRG  = 0x00; //visas porta G līnijas uz IEvadi
```

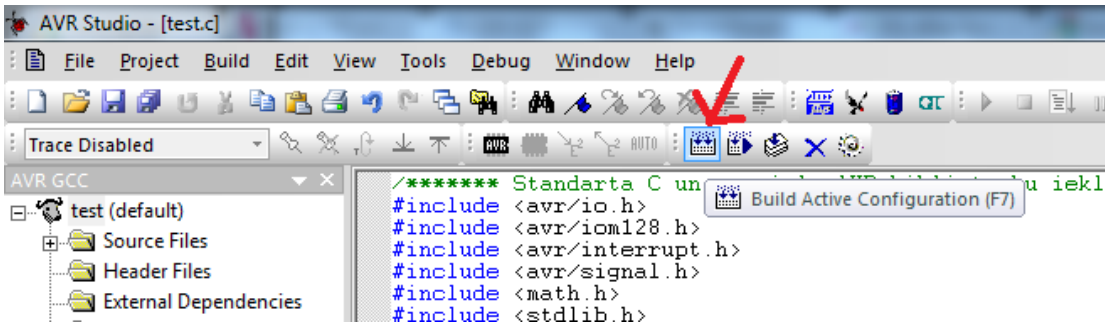
```
 PORTA  = 0x00; //porta A atsienošie rezistori pret +Vcc NEtiek izmantoti
 PORTB  = 0x00; //porta B atsienošie rezistori pret +Vcc NEtiek izmantoti
 PORTC  = 0x00; //porta C atsienošie rezistori pret +Vcc NEtiek izmantoti
 PORTD  = 0x00; //porta D izejas līniju līmeņi uz 0
 PORTE  = 0x00; //porta E atsienošie rezistori pret +Vcc NEtiek izmantoti
 PORTF  = 0x00; //porta F atsienošie rezistori pret +Vcc NEtiek izmantoti
 PORTG  = 0x00; //porta G atsienošie rezistori pret +Vcc NEtiek izmantoti
}
/*********************************************************************/

/******************** Kontrollera inicializācija**********************/
void init_devices(void)
{
cli();                          //Aizliedz visus pārtraukumus
XDIV = 0x00;                    //Takts impulsu dalītājs NEtiek izmantots
XMCRA = 0x00;                   //Ārējo atmiņu NEizmanto
MCUCR = 0x00;                   //NEtiek izmantoti nekādi enerģiju taupoši stāvokli
port_init();                    //Izsauc funkciju, kas inicializē portus
}
/*********************************************************************/

/***************************** Main funkcija *************************/
int main (void)
{
init_devices();                         //Inicializē mikrokontrolleri

for (;;)                                //Mūžīgais cikls, lai programma
                                        //nebeigtos
{
PORTD=0b00000001;                       //Datu izvade uz porta D līnijām
PORTD=0b00000010;
}
return 1;                               //Beidz main izpildi
}
/*********************************************************************/
```

Izdruka 1 Programmas piemērs

Programmas pirmkoda kompilēšana un ierakstīšana mikrokontrollerī

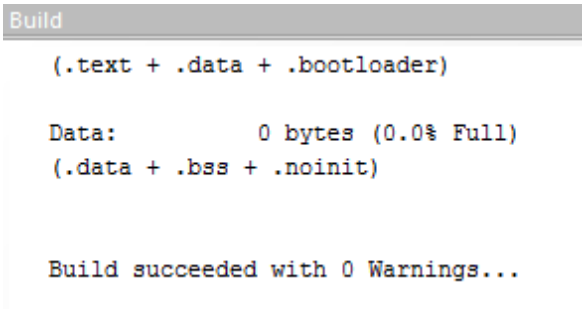
Kad esat uzrakstījis programmu tālāk to ir nepieciešams nokompilēt. To var veikt trīs dažādos veidos:

- Izvēloties **Build->Build**;
- Nospiežot taustiņu **F7;**
- Vai nospiežot uz attiecīgo piktogrammu programmas logā (Sk. Att. 3);



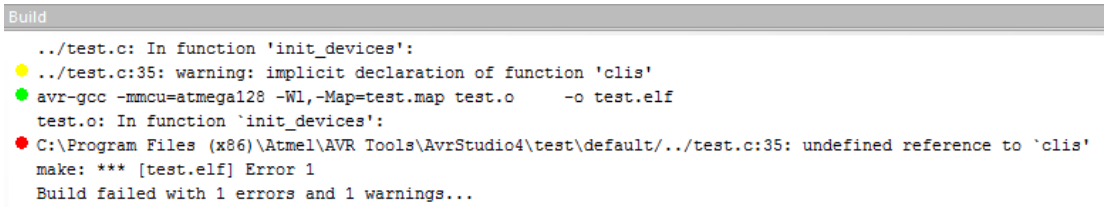
Att. 3 Programmas kompilēšana

Ja programmas kompilācija ir bijusi veiksmīga, tad **Build** logā parādīsies sekojošs ziņojums:



Att. 4 Veiksmīgi kompilēta programma

Pretējā gadījumā šajā logā parādīsies paziņojums par neveiksmīgu kompilāciju un atrasto kļūdu un/vai brīdinājumu saraksts:



Att. 5 Neveiksmīgi kompilēta programma

Ar dzeltenu tiek atzīmēti **brīdinājumi** – tos novērst nav obligāti, bet ir vēlams. Ar sarkanu ir atzīmētas **kļūdas**, kuras novērst ir obligāti. Att. 5 ir redzams viens brīdinājums un viena kļūda. Ieteicams pirmo novērst kļūdu, jo kā redzams brīdinājums izriet no kļūdas (kompilators meklē funkciju clis() un tā kā tā nav definēta pirms funkcijas izsaukuma tiek izdots brīdinājums par netieši definētu funkciju; kad kompilators sasniedz pirmkoda beigas un joprojām neatrod šis funkcijas

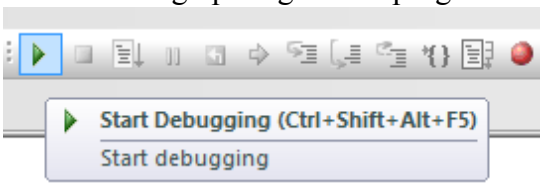
definīciju tiek izvadīts kļūdas paziņojums). Kad kļūda novērsta failu vispirms saglabā un mēģina atkārtoti kompilēt.

Tālāk ir divas iespējas: programmu var atkļūdot izmantojot iepriekš izvēlēto atkļūdošanas platformu vai programmu ielādēt mikrokontrollerī.

Programmas atkļūdošana ar AVR Simulator

Atkļūdošana ir nepieciešama, lai varētu pārbaudīt izveidotās programmas darbību to simulējot (kaut arī sintakse ir pareiza un kompilators nav izdevis kļūdas paziņojumus nav garantija, ka programmas semantika ir pareiza un to iespējams pārbaudīt ar atkļūdošanu). Programmas atkļūdošanu iespējams uzsākt trīs dažādos veidos:

- Izvēloties **Debug->Start Debugging**;
- Nospiežaut taustiņu kombināciju **Ctrl+Shift+Alt+F5**;
- Vai nospiežot uz attiecīgo piktogrammu programmas logā (Sk. Att. 6);



Att. 6 Programmas atkļūdošanas uzsākšana

Pirmkodam labajā pusē parādās dzeltena bulta, kura norāda nākošo C komandu ko izpildīs. Lai varētu to izpildīt šo un pāriet pie nākošas komandas var:

- spiest **Step Into** (jeb **F11**), kas, piemēram, ja izpildāmā komanda ir kāda funkcija ieies funkcijā un soli pa solim izpildīs arī funkciju (lai izietu no funkcijas var lietot **Step Out**);
- spiest **Step Over** (jeb **F10**), kas izpildīs visu funkciju vienā solī;

```
/****************************** Main funkcija ***************************/
int main (void)
{
init_devices();                  //Inicializē kontrolleri
for (;;)
{
PORTD=0b00000001;

PORTD=0b00000010;

}
return 1;                      //Beidz main izpildi
}
/************************************************************************/
```

Att. 7 Programmas atkļūdošana

Secīgi izpildot komandas **I/O View** logā var redzēt mikrokontrollera reģistrus, uz kuriem tiek attēlots patreizējais mikrokontrollera stāvoklis un izpildoties programmai redzams kā tie mainās. Par piemēru ņemsi Izdrukas 1 pirmkodu. Attiecīgo līniju stāvokli iespējams aplūkot nospiežot „+” pie **PORTD** (Sk. Att. 8). Sākotnēji redzams, ka visas PORTD līnijas ir loģiskajā „0” stāvoklī (balts kvadrātiņš atbilst loģiskajam „0”, bet pelēks loģiskajam „1”).

ir

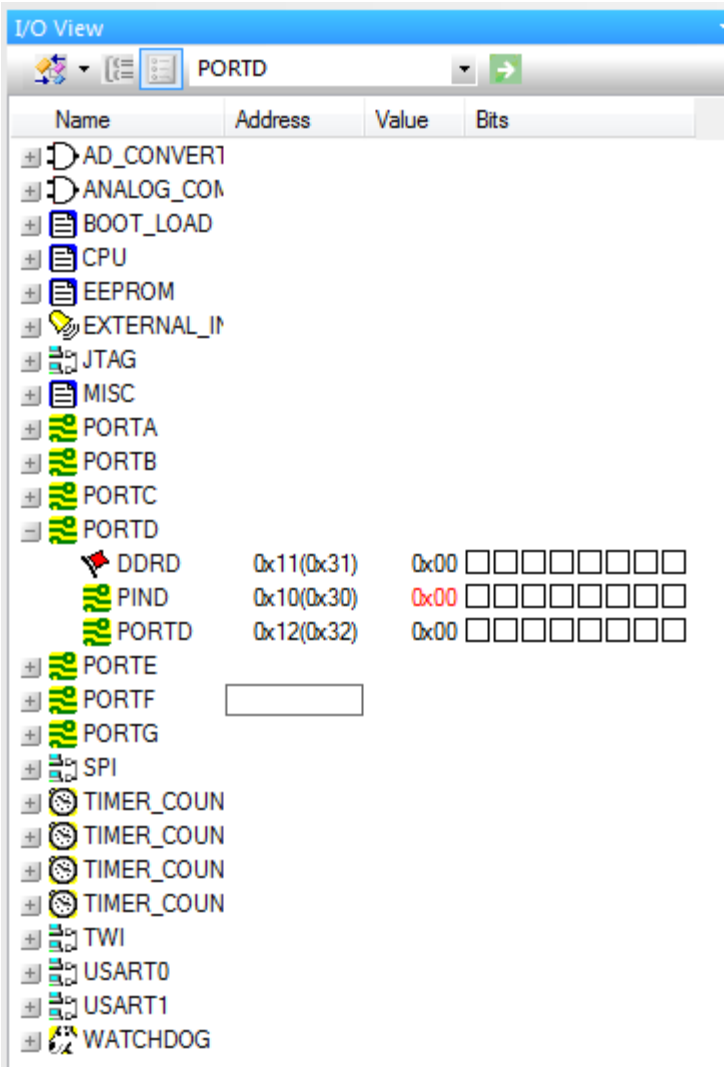
ar

uz

ar

no

uz



Ievades izvades līnijas grupētas portos (PORTA-PORTG), un katrs ports tiek vadīts trim reģistriem.

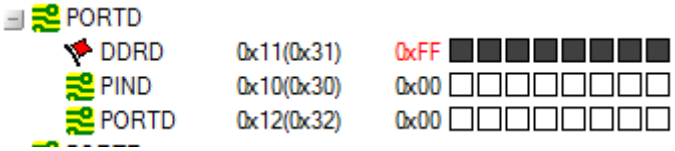
DDRx nosaka katras līnijas datu virzienu. 1 uzstāda attiecīgo porta līniju uz izvadi, bet 0 ievadi. Pēc noklusējuma visas porta līnijas ir uzstādītas uz ievadi.

PORTx ir datu reģistrs kura palīdzību datus izvada uz līnijas. Attiecīgās I/O līnijas, kuras ir uzstādītas uz izvadi atbīldīs attiecīgajām PORTx reģistra bitu vērtībām

PINx izmanto, lai nolasītu līnijas vērtību konkrētā porta un līnijām kas uzstādītas ievadi.

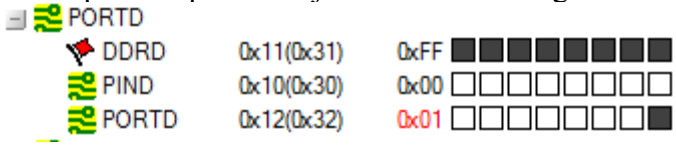
Att. 8 I/O View logs

Nospiežot **F10** tiek izpildīt mikrokontrollera inicializācijas funkcija un porta D līnijas tiek uzstādītas uz izvadi, t.i. kvadrātiņi pie **DDRD** iekrāsojas pelēki:



Piezīme! Ja šeit nospiestu **F11** secīgi soli pa solim tiktu izpildīta *init_devices()* funkcija. Ja nav nepieciešams pārbaudīt kādu vērtību, reģistru, kas atrodas funkcijā tad var izpildīt **Step Over**, pretējā gadījumā **Step Into**.

Vēlreiz nospiežot **F10** porta D pirmā līnija tiek uzstādīta augstā līmeni:

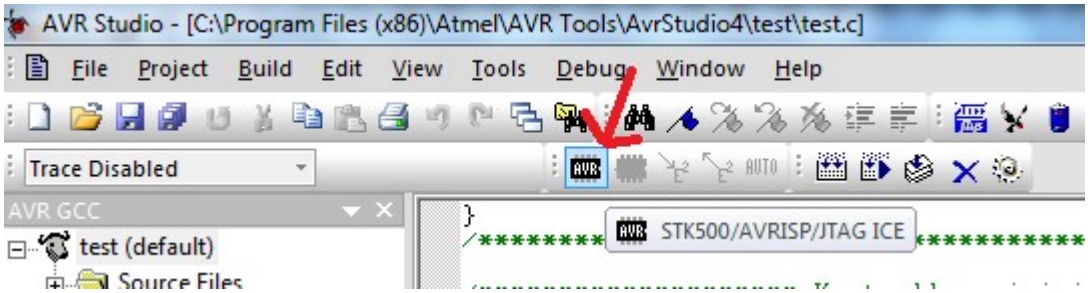


Un vēlreiz nospiež **F10:**

Redzams, ka porta D izejas mainās tā kā bija paredzēts programmā. Kad atkļūdošana ir pabeigta var pāriet pie nākošā soļa – programmas ielādes mikrokontrollerī.

Programmas ielādēšana mikrokontrollerī

Pirms programmu var ielādēt mikrokontrollerī ir nepieciešams uzstādīt AVR Studio 4. Vispirms nepieciešams uzstādīt izmantojamo programmatoru:



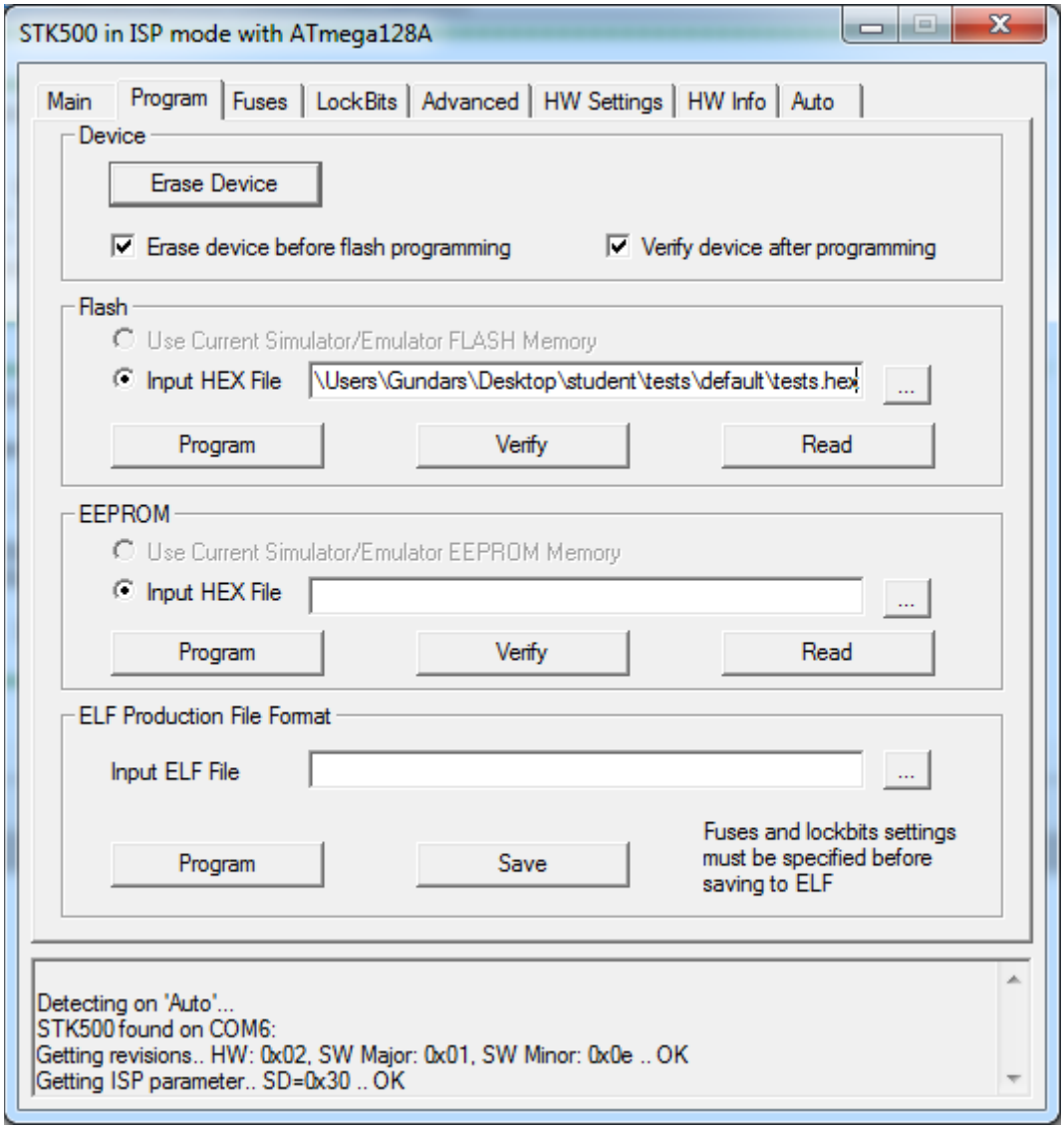
Att. 9 Programmatora uzstādīšana

Laboratorijas darbos tiks izmantots programmatora STK-500 klons – HW STK-500 ISP. To uzstāda nospiežot uz Att. 9 norādītās piktogrammas ar uzrakstu **AVR**. Ja programmators ir pievienots datoram tad atvērsies logs, kas prasa atjaunot programmatora versiju, spiediet **Cancel**. Ja atveras **Connec Failled** logs, tad jāpārbauda vai Charon II plate ir pieslēgta barošanai un vai programmators ir pieslēgts gan platei, gan datoram.

Ja visi uzstādījumi ir korekti atvērsies **STK500** programmēšanas dialogs (Sk. Att. 10). Šeit nepieciešams izmainīt dažus uzstādījumus:

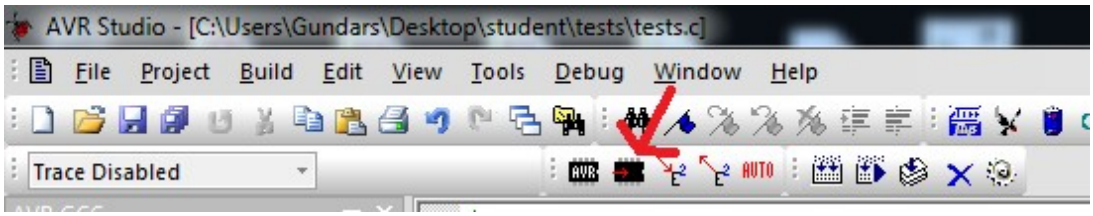
- **Program** apakš sadaļā **Flash** izvēlaties **Input HEX File** un atrodiet sevis izveidoto projekta mapi un tajā sameklējiet failu uar paplašinājumu ***.hex** (../Desktop/student/Project_folder/default/file_name.hex). Izvēlaties šo failu un nospiediet **Open**;
- Pārejiet uz **Main** sadaļu. **Programming Mode and Target Settings** izvēlaties kā **PP/HVSP mode**. **Device and Signature Bytes** apakš sadaļā izvēlaties ATmega128A. Varat arī izpildīt **Read Signature**, ar to pārbauda vai pievienots izvēlētais mikrokontrolleris. Ja viss pareizi, tad zem izvadītā rezultāta būs redzams uzraksts **Signature matches selected device**, pretējā gadījumā: **WARNING: Signature does not match selected device!**

Ar to beidzas programmatora uzstādīšana. Šo logu nedrīkst aizvērt ar **Close**, jo tad uzstādījumi būs jāveic no jauna. To novieto uz palodzes ar **Minimize**.



Att. 10 Programmēšanas dialogs

Lai ielādētu programmu AVR Studio 4 logā nepieciešams nospiest pogu **Write program memory** (Sk. Att. 11), kas kļuva aktīva pēc programmatora uzstādīšanas.



Att. 11 Mikrokontrollera programmēšana

Programmatora un programmēšanas dialoga uzstādīšanu nepieciešams veikt tikai vienu reizi. Pēc tam pietiek ar pirmkoda kompilēšanu (**Build Active Configuration** vai **F7**) un programmas atmiņas ierakstīšanu (**Write program memory**).

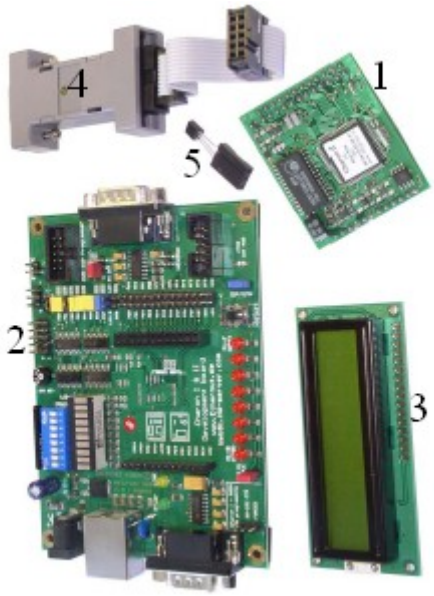
1. laboratorijas darbs

Uzdevums

1. Iepazīties ar ATmega128 un CharonII;
2. Nokompilēt piedāvātu izejas kodu un ierakstīt *.hex* programfailu mikrokontrollera atmiņā. Pārbaudīt testa programmas korektu darbību;
3. Pārrakstīt programmu tā, lai visas gaismas diodes ieslēgtos pēc kārtas, izveidot „skrejošas gaismas" efektu;

Teorētiskais apraksts

Zem katra Jūsu darba galda ir izvietoti Charon II izstrādes komplekti (Att. 12). Pie izstrādes plates ir pievienots Charon II modulis, barošanas avots, programmators un COM vads (saslēdz Charon II izstrādes plati ar datoru).



Att. 12 Charon II izstrādes komplekts

Charon II izstrādes komplekts sastāv no:

1. Charon II moduļa.
2. Izstrādes plates.
3. LCD displeja.
4. HW STK-500 ISP programmatora.
5. 1-Wire temperatūras sensora.

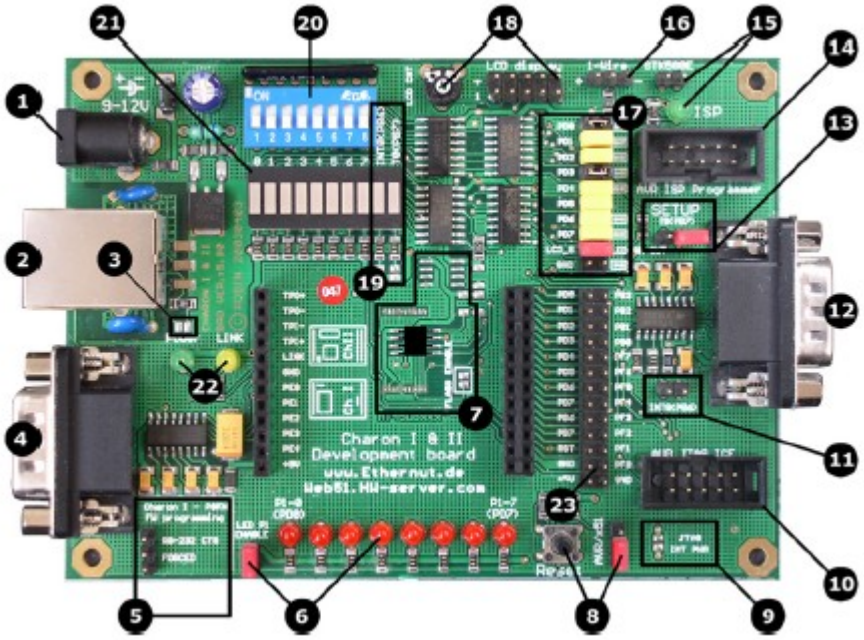
Charon II modulis sastāv no:

1. Atmega128 – tas 8. bitu RISC mikrokontrollera.
2. RTL8019AS – tas ir pilns duplekss Realtek firmas Ethernet kontrolleris.

Charon II izstrādes plate (Att. 13) sastāvā ietilpst:

1. Barošanas avota pieslēgvieta.
2. Ethernet RJ45 savienotājs.
3. Gala tiltslēgs.

4. RS-232 (seriālā pieslēgvieta 0).
5. Charon I PSEN programmaparatūras programmēšanas tiltslēgs.
6. LED ieslēgšanas/atslēgšanas tiltslēgs un astoņi LED.

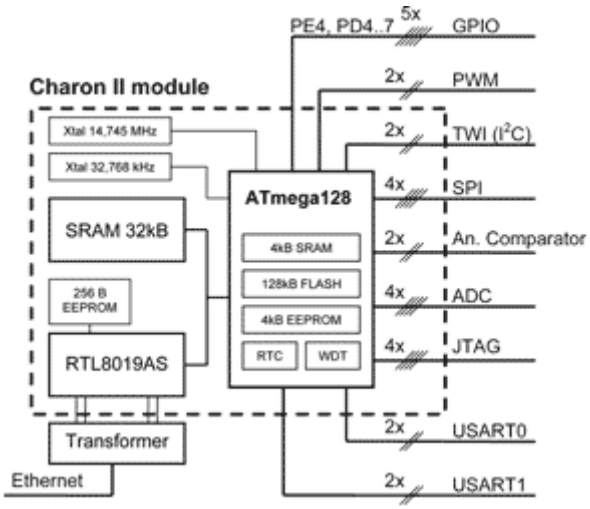


Att. 13 Charon II izstrādes plate

7. SPI Flash atmiņas laukums.
8. Atiestates slēdzis un polaritātes tiltslēgs.
9. JTAG iekšējas barošanas tiltslēgs.
10. AVR JTAG ICE savienotājs.
11. PB6/OC1B tiltslēgs.
12. RS-232 (seriālā pieslēgvieta 1).
13. PB7/SETUP režīma JMP1 tiltslēgs.
14. AVR ISP (iekš sistēmas programmators) savienotājs.
15. ISP LED un STK500 programmēšanas tiltslēgs.
16. 1-Wire kopnes savienotājs.
17. Perifērijas tiltslēgu lauks.
18. LCD savienotājs un LCD kontrasta regulators.
19. PB6 un PB7 LED indikatori.
20. Pārbīdes reģistra paralēlais ievades slēdzis.
21. Pārbīdes reģistra paralēlās izejas LED.
22. LED indikatori.

! Visus papildus nepieciešamos aprakstošos dokumentus var atrast mapē Materials.

Att. 14 ir shematiski paradīta Charon II moduļa struktūra, kā arī ATmega128 mikrokontrollera perifērijas interfeisi, kuri ir izmantoti Charon II izstrādes platē.



Att. 14 Charon II moduļa pieslēgums pie Atmega128 mikrokontrollera

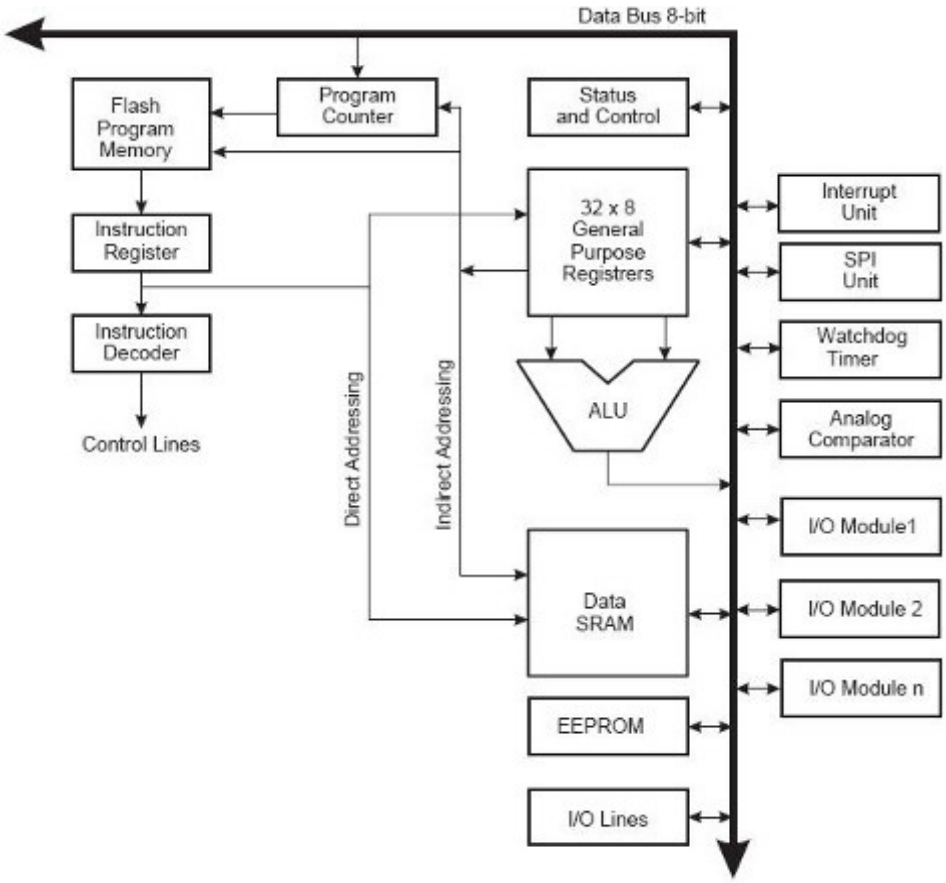
Pildot laboratorijas darbus būs nepieciešams programmēt mikrokontrolleri ATmega128. Līdz ar to ir īsumā tiks aprakstīts šis mikrokontrolleris.

Ievads AVR arhitektūrā

AVR arhitektūra apvieno sevī jaudīgo Harvardas arhitektūru ar sadalītu datu un programmas atmiņas piekļuvi un RISC procesoru (bāzes AVR komandu kopa sastāv no 120 instrukcijām), 32. vispārēja nolūka reģistrus, katrs no kuriem var strādāt kā reģistrs-akumulators, un paplašinātu komandas sistēmu ar fiksētu 16 bitu garumu (Att. 15). Komandu vairākums izpildās vienā taktī ar vienlaicīgu komandas izpildīšanu un nākamās komandas ienesi. Tas nodrošina ap 1 MIPS veiktspēju uz takts frekvenci 1 MHz.

Visiem AVR mikrokontrolleriem ir iebūvēta FLASH ROM ar iekš sistēmas programmēšanas (ISP) iespēju caur 4 līniju seriālo interfeisu. AVR mikrokontrollera perifērija sastāv no: taimeriem-skaitītājiem, PWM, ārēju pārtraukumu atbalsta, analogiem komparatoriem, 10 bitu 8. kanālu ACP, paralēliem portiem (no 3. līdz 48.ievades izvades līnijām), UART un SPI interfeisiem, sargsuņa taimera (angl.. *Watchdog timer*). Visas šīs īpašības pārvērš AVR mikrokontrollerus par jaudīgiem instrumentiem mūsdienīgu, augstražīgu, daudz nolūku sistēmu būvēšanā.

AVR mikrokontrolleris atbalsta miega režīmu, kā arī mikropatēriņa režīmu. Miega režīmā tiek apstādināts centrālā procesora kodols, tajā pat laikā reģistri, taimeri-skaitītāji, sargsuņa taimeris turpina funkcionēt. Mikropatēriņa režīmā tiek saglabāts reģistru saturs, tiek apstādināts taktu ģenerators, tiek aizliegtas visas mikrokontrollera funkcijas, līdz tam brīdim, kamēr neiestāsies ārējais pārtraukums vai atiestatīšana. Atšķirīgi AVR mikrokontrolleri strādā 2,7 – 6 V vai 4 – 6 V barošanas sprieguma diapazonā.



Att. 15 AVR arhitektūras bloku diagramma

ATmega128 īpašības

Atmega128 mikrokontrollerim piemīt visas augšā aprakstītas īpašības un sastāv elementi. Bet tomēr daži raksturlielumi atšķiras :

1. 53 programmējamās līnijas, kuras sastāv no:
   - A porta;
   - B porta;
   - C porta;
   - D porta;
   - E porta;
   - F porta;
   - G porta;
2. 128 KB programmējamas atmiņas FLASH ROM
3. 4 KB MCU SRAM
4. 4 KB programmējamas atmiņas EEPROM

Programmas izstrāde

Izveidojiet jaunu projektu, piemēram, Lab1. Iekopējiet piedāvāto programmas pirmtekstu Lab1.c failā. Šis pirmteksts ir līdzīgs kā Izdrukā 1, atšķirsies tikai *main* funkcija. Tāpēc tagad aplūkosim tikai to (Sk. Izdruka 2).

```
int main (void)
{
 unsigned char i,port_data;      //Mainīgo definēšana

 init_devices();                 //Mikrokontrollera inicializēšana

 while(1)                        //Mūžīgais cikls, lai programma nekad nebeigtos
 {
  port_data = 0b00000001;        //Bināras vērtības piešķiršana
  PORTD=~port_data;              //Izvadīt uz Porta D līnijām apgrieztu (~)      port_data
                                 //mainīga vērtību
 }
 return 1;
}
```

Izdruka 2 Lab1.c *main* funkcija

No sākuma notiek divu mainīgo definēšana, pēc tam *init_devices()* funkcijas izsaukšana, kurā notiek portu un mikrokontrollera inicializēšana. Pārējais programmas pirmkods tiek pildīts mūžīgajā ciklā *while(1)*, jo citādāk mikrokontrolleris pabeigs programmas izpildi un „apstāsies". Ciklā notiek bināras vērtības (**0b**00000001) piešķiršana *port_data* mainīgajam, un nākamajā rindā šo datu (tikai apgrieztā veidā) izvadīšana uz porta D līnijām. Dotajā piemērā, uz porta D līnijām ir izvadīta apgrieztā *port_data* mainīga vērtība, jo gaismas diodes uz Charon II izstrādes plātes iedegsies tikai, ja uz porta D līnijām būs izvadītas nulles (skat. Charon II izstrādes plates shēmu).

Nokompilējiet šo kodu un ielādējiet ATmega128 mikrokontrollerī. Ja viss ir izdarīts pareizi, tad iedegsies gaismas diode numur nulle.

Jums vajag modificēt doto kodu tā, lai uz porta D līnijām varētu izvadīt „skrejošu uguni". Piemēram, pirmajā iterācijā uz porta D tiek izvadīta vērtība ~(00000001), tad iedegsies tikai nulles gaismas diode. Pēc nelielas pauzes, uz porta D līnijām tiek izvadīta vērtība ~(00000010), tad iedegsies tikai pirmā gaismas diode. Un tā turpināsies, kamēr neiedegsies astotā gaismas diode. Pēc tam, programmai ir jāsāk no jauna, t.i. iededzināt nulles diodi, pēc tam pirmo u.t.t.

Atskaitē

Atskaitē īsumā ir jāapraksta ATmega128 mikrokontrolleris un jāveic neliels salīdzinājums ar jebkuru citu AVR mikrokontrolleri; jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto **viss** programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi

1. Aprakstīt Charon II izstrādes plati (sastāvdaļas);
2. Atšķirība starp ISP un JTAG;
3. Paskaidrot izveidotās programmas darbību;
4. Paskaidrot kā notiek informācijas ievade uz porta līnijām un informācijas nolasīšana no porta līnijām;

2. laboratorijas darbs – 8 bitu taimeris/skaitītājs

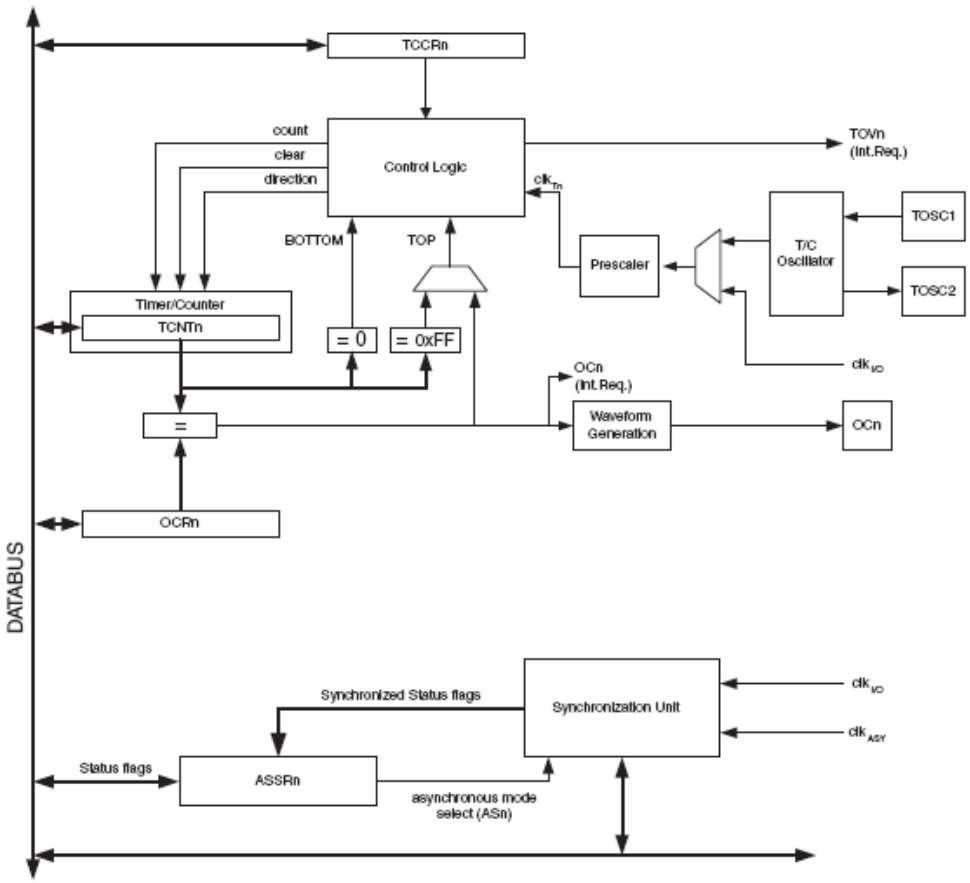
Uzdevums

Modificēt pirmā laboratorijas darba programmas pirmkodu tā lai pauze starp gaismas diožu pārslēgšanām būtu vienāda vienai sekundei, laika mērīšanai izmantojot ATmega128 Taimeri/skaitītāju 0.

Teorētiskais apraksts: ATmega128 8 bitu taimeris/skaitītājs 0.

Taimeris/Skaitītājs0 ir vienkanāla, 8-bitu Taimera/Skaitītāja modulis. Taimera/Skaitītāja 0 galvenās īpašības ir:

- vienkanāla skaitītājs;
- taimera notīrīšana pēc salīdzināšanas sakrišanas;
- fāzē korekta impulsa platuma modulators (PWM);
- frekvences ģenerators;
- 10-bitu takts priekš dalītājs;
- pārpildīšanās un salīdzināšanas sakritības pārtraukumu avoti(TOV0 un OCF0);
- Atļauj taktēšanu no ārējā 32kHz kristāla neatkarīgi no I/O takts



Att. 16 8 bitu taimera/skaitītāja bloku diagramma

Taimera/Skaitītāja (TCNT0) un izejas salīdzināšanas reģistrs (OCR0) ir 8-bitu reģistri. Pārtraukuma pieprasījuma signāls (saīsinājums *Int.Req*) ir redzams taimera pārtraukumu karodziņu reģistrā (TIFR). Visi pārtraukumi ir individuāli maskēti TIMSK (*Timer Interrupt Mask Registr*) reģistrā. TIFR un TIMSK reģistri nav paradīti Att. 16, jo tos lieto citās taimera daļās.

Taimeris/Skaitītājs 0 var būt taktēts no iekšēja oscilatora, ar priekš dalītāju vai asinhroni no TOSC1/2 līnijām. Taktēšanas izvēles loģika kontrolē kurš taktēšanas avots izraisīs taimera/skaitītāja vērtības palielināšanu (vai samazināšanu). Taimeris/Skaitītājs ir neaktīvs, kad taktēšanas avots nav izvēlēts. Charon II tiek darbināts ar iekšējo oscilatoru ar frekvenci **14.7456 MHz**, t.i. 14745600 takts impulsu sekundē.

Lai vadītu 8 bitu taimeru/skaitītāju 0 ir nepieciešams operēt ar to reģistriem lai sakonfigurētu un nolasītu taimera/skaitītāja vērtības. Sākumvērtība

Taimera/Skaitītāja vadības reģistrs – TCCR0

Bits	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Lasīt/Rakstīt	R	L/R	L/R	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	

Att. 17 Taimera/Skaitītāja 0 TCCR0 reģistrs

7. bits – FOC0 Force Output compare (piespiedu izejas salīdzināšana)
FOC0 bits ir aktīvs tikai tad, kad WGM biti ir norādīti kā ne-PWM režīms. Savietojamības dēļ šim bitam jātiek uzstādītam uz nulli, kad tiek ierakstīts TCCR0 bits, darbojoties PWM režīmā. Kad FOC0 bitā tiek ierakstīts 1, nekavējoties notiek salīdzināšanas sakritības uzspiešana impulsa platuma ģeneratoram.
FOC0 bits vienmēr ir nolasāms kā nulle.

6. un 3. bits – WGM01:0: Waveform Generation Mode (viļņa formas ģenerēšanas režīms)

Biti kontrolē skaitītāja skaitīšanas secību, skaitītāja maksimālās (TOP) vērtības avotu un viļņa formas ģenerācijas tipu.

Taimera/skaitītāja režīmi:

- Normālais režīms
- Taimera notīrīšanas uz salīdzināšanas sakritības režīms (CTC)
- Divi impulsa platuma modulācijas veidi (PWM)

Režīms	WGM01[1] (CTC0)	WGM00[1] (PWM0)	Taimeris/Skaitītājs Darbības režīms	TOP	OCR0 tiek ko-riģēts	TOV0 karodziņš
0	0	0	Normal	0xFF	**Tūlītēji**	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	**Tūlītēji**	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

5. un 4. bits – COM01:0: salīdzināšanas sakritības izejas režīms

Šie biti nosaka izejas salīdzināšanas pina (OC0) uzvedību. Ja viens vai abi biti ir iestatīti, OC0 izeja tiek izmainīta normāla izejas/ieejas porta funkcionalitāte, pie kura tas ir pievienots. Jāpiezīmē, ka DDR reģistram jābūt iestatītam, lai atļautu pinam darboties kā izejai.

COM01	COM00	Apraksts
0	0	Normāla porta darbība, OC0 atvienots
0	1	Uzstāda OC0 uz salīdzināšanas sakritības
1	0	Pārslēdz OC0 uz salīdzināšanas sakritības
1	1	Nodzēš OC0 uz salīdzināšanas sakritības

2. , 1., un 0. bits – CS02:0: takts izvēle

Trīs takts izvēles biti iestata izvēlēto takts avotu, kuru izmantos taimeris/skaitītājs 0. Mainot priekš-dalītāja vērtību iespējams palielināt laika intervālu ko tas mēra, bet tiek zaudēta precizitāte.

CS02	CS01	CS00	Apraksts
0	0	0	Nav taktēšanas avota (Taimers/Skaitītājs apturēts)
0	0	1	clk_{T0S}/ (Bez priekšdalīšanas)
0	1	0	clk_{T0S}/8 (No priekšdalītāja)
0	1	1	clk_{T0S}/32 (No priekšdalītāja)
1	0	0	clk_{T0S}/64 (No priekšdalītāja)
1	0	1	clk_{T0S}/128 (No priekšdalītāja)
1	1	0	clk_{T0S}/256 (No priekšdalītāja)
1	1	1	clk_{T0S}/1024 (No priekšdalītāja)

Taimera/skatītāja reģistrs - TCNT0

Taimera/skatītāja 0 TCNT reģistrs dod tiešu pieeju, gan pie rakstīšanas, gan pie lasīšanas operācijām, saistītām ar taimera/skatītāja tekošo vērtību.

Bits	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Lasīt/Rakstīt	L/R	L/R	L/R	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	

Att. 18 Taimera/Skaitītāja 0 TCNT0 reģistrs.

Taimera/skatītāja reģistrs - TIMSK

Bits	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Lasīt/Rakstīt	L/R	L/R	L/R	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	

Att. 19 Taimera/Skaitītāja 0 TIMSK reģistrs.

1. bits – OCIE0: Taimera/Skaitītāja0 Output Compare sakrišanas pārtraukuma ieslēgšana

Kad reģistrā OCIE0 ir ierakstīts viens, un I-bits Status Reģistrā arī ir uzstādīts (vieninieks), taimera/skaitītāja 0 sakrišanas pārtraukums ir ieslēgts.

0. bits – TOIE0: Taimera/Skaitītāja0 pārpildes pārtraukuma ieslēgšana

Kad reģistrā TOIE0 ir ierakstīts viens, un I-bits Status Reģistrā arī ir uzstādīts (vieninieks), taimera/skaitītāja0 pārpildes pārtraukums ir ieslēgts. Attiecīgs pārtraukums tiek izsaukts ja parādās taimera/skaitītāja0 pārpilde, t.i. kad TOV0 bits ir uzstādīts taimera/skaitītāja0 pārtraukumu karodziņu reģistrā – TIFR.

Atskaitē

Atskaitē īsumā ir jāapraksta ATmega128 mikrokontrollera Taimeris/Skaitītājs 0; jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto **viss** programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi

1. Kas ir Taimeris/Skaitītājs? Tā darbības principi, pielietojumi.
2. Paskaidrojiet pārtraukuma būtību.
3. Aprakstiet Taimera moduļa konfigurēšanu, kā arī signālu skaitīšanu uz shēmas (Att. 16).

3. laboratorijas darbs – Rādītāji

Uzdevums

Papildināt 2. laboratorijas darbu ar rādītāju uz datu masīvu. Masīvā glabā veselas skaitļu vērtības, piemēram, no 1 līdz 5, masīvu aizpildīt izmantojot rādītāju. Masīva vērtības nolasīt un izmantot skrejošās gaismas ilguma veidošanā. Piemēram, masīva 1. elements ir 1, tad skrejošās gaismas ilgums ir 1 sekunde, masīva 2. elements ir 2, tad skrejošās gaismas ilgums ir 2 sekundes, utt.

Teorētiskais apraksts:

Datu masīvi

Masīvs ir salikts objekts, kas tiek veidots no viena tipa elementiem jeb mainīgajiem, kas tiek apzīmēts ar vienu nosaukumu un kam piekļūst izmantojot indeksācijas numurus. Vienkāršu masīvu var deklarēt sekojoši:

```
<datu_tips> x [n1][n2]...[nk];
```

Kur *x* – masīva nosaukums, *ni* – masīva dimensija. *k* masīvs ir dēvēts par *k* – dimensiju masīvu, ar *datu_tipa* tipa elementiem.

Piemēram:

```
int page[10]; /*vienas dimensijas masīvs ar int tipa10 elementiem ,kura elementi ir
                sanumurētiem no 0 līdz 9*/
char line[81]; /*char tipa vienas dimensijas masīvs*/
float big[10][10], sales[10][5][8]; /*float tipa divu dimensiju un trīs dimensiju
                masīvi*/
```

Atsaukties uz *k dimensiju* **x** masīva elementu var sekojoši:

```
page[5]
line[i+j-1]
big[i][j]
```

Rādītāji

Rādītājs ir datu tips, kura vērtība tieši „norāda” uz atmiņas apgabalu, karā glabājas kāda mainīgā vērtība. Tātad rādītājs ir mainīgas, kas glabā adresi. Rādītāju deklarē sekojoši:

```
<datu_tips> *ptr1, *ptr2,...,*ptrn;
```

Kur *ptr1, *ptr2,...,*ptr**n** – rādītājs uz *datu_tips* tipa mainīgajiem. Šie mainīgie kalpo par atsaucēm uz objektiem ar *datu_tips* tipu. Tādu tipu sauc par mainīgo-rādītāju ***bāzes tipu.***

Piemēram:

```
int *ptr, *qi;    /* rādītāji uz veseliem objektiem */
char *c;          /* rādītājs uz teksta objektu */
```

Atsauce uz iepriekš definētiem objektiem

Deklarēts (neinicializēts) rādītās rāda uz nenoteiktu adresi, tāpēc nepieciešams izmantot atsauces. Rādītāji var nodrošināt atsauci uz iepriekš definētiem objektiem. Tāda objekta adrese var būt noteikta lietojot adresācijas operatoru & (*address of operator*). Piemēram, apskatīsim mainīgus ***i*** un ***ptr***, definētus kā:

```
int i, *ptr;
```

Lai rādītājs ***ptr*** norādītu uz objekta ***i*** adresi veic sekojošu operāciju:

```
ptr = &i;
```

Tagad rādītājs ***ptr*** norāda uz objekta ***i*** adresi. Tāpēc uz objektu ***i*** tagad arī ir iespējams atsaukties ar rādītāju ***ptr*** un to var veikt izmantojot operatoru *, sekojošā pierakstā: *ptr. Šajā gadījumā vārdus ***i*** un ****ptr*** dēvē par pseido-vārdiem.

Rādītājs uz masīvu

Masīvu gadījumā sintakse ir ļoti līdzīga:

```
int *ptr;
int array[3] = {1,2,3};
```

Lai norādītu uz masīvu:

```
ptr=&array[];
//vai
ptr=array;
```

Bet pēc šīs operācijas rādītājs rādīs tikai uz masīva pirmo elementu. Bet kā tad nolasīt nākošo masīva elementu? Šim nolūkam var izmantot rādītāju aritmētiku, proti, rādītāju saskaitīšanu un atņemšanu:

Piemēram:

```
ptr++;        //Pāriet uz masīva nākamo elementu
ptr--;        //Pāriet uz masīva iepriekšējo elementu
ptr=ptr+2;    //Pāriet par diviem elementiem masīvā uz priekšu
ptr=ptr-2;    //Pāriet par diviem elementiem masīvā atpakaļ
```

Lai mainītu vai nolasītu masīva elementu, kā iepriekš, izmanto operatoru * un rādītāju.

Atskaitē

Atskaitē īsumā ir jāapraksta ATmega128 mikrokontrollera atmiņas veidi; jāizveido īss rādītāju apraksts; jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto izveidotais programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi

1. Paskaidrot rādītāja būtību un pielietojumu.
2. ATmega128 izmantotie atmiņas veidi;

4. laboratorijas darbs – Analogs Cipars Pārveidošana (Analog-Digital Converter – ADC)

Uzdevumi:

1. Izmainīt piedāvāto 8 bitu ADC pārveidošanas kodu, lai būtu iespēja strādāt ADC 10 bitu režīmā.
2. Izmainīt mērāmo signālu uz sin().
3. Pierādīt koda pareizo darbību.

Teorētiskais apraksts:

Analogs-Cipars pārveidotājs

ATmega 128 tiek izmantots 10 bitu secīga tuvinājuma ADC (Sk.Att. 20). ADC ir pievienots multipleksoram ar 8 kanāliem. Tas nozīmē, ka ar šādu ADC var mērīt 8 dažādu analogos signālus. Trūkums ir tāds, ka vienā laika momentā ir iespējams pārveidot tikai vienas ieejas analogo signāli. Praktiski tas nozīmē, kas vispirms ir jāizvēlas kanāls, kura analogo signālu vēlas mērīt, jāpārveido šī signāls un tad ir jāpārslēdz cita ACP ieeja ar multipleksoru.

Secīga tuvinājuma ADC satur *Sample and Hold* shēmu, kas nodrošina, ka ADC ieejas spriegums tiek turēts nemainīgā līmenī, kamēr notiek pārveidošana.

ADC ir iespējami divi barošanas avoti – viens ārējs, kuru pieslēdz pie AVCC, un vien iekšējs, kurš parasti ir 2.56V un tiek nodrošināts no shēmas. To, kuru izmantot ir iepriekš jāizvēlas.

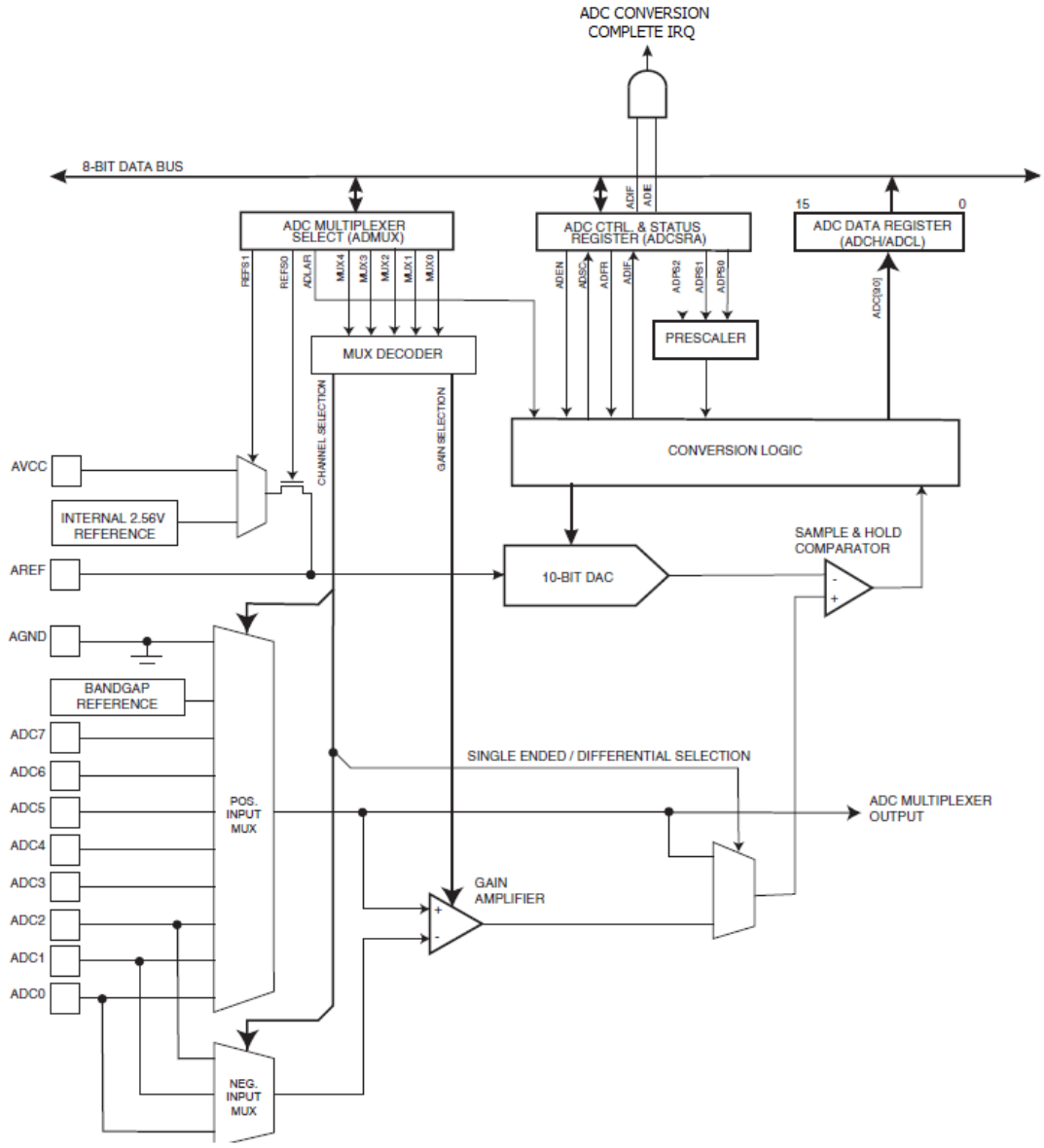
Īsumā var uzskaitīt galvenās ATmega 128 ADC īpašības:

- 10 bitu izšķirtspēja
- 0.5 LSB integrālā nelinearitātes kļūda
- ±2 LSB absolūtā precizitāte
- 13-260 µs pārveidošanas laiks
- 8 multipleksēti vienas ieejas kanāli
- Var izvēlēties rezultāta izkārtošanu – izlīdzinājums pa kreisi vai labi
- 0-VCC ADC ieejas sprieguma diapazons
- Izvēles 2.56V ADC atbalsta spriegums
- Nepārtrauktas pārveidošanas vai vienreizējas pārveidošanas režīmi
- Pārtraukums uz ADC pārveidošanas pabeigšanu
- Miega režīma trokšņu slāpētājs

ADC darbības iestādīšanai izmanto ADCSRA (ADC statusa un vadības reģistrs) un ADMUX (ADC multipleksēšanas izvēles reģistrs. Lai ieslēgtu ADC jāuzstāda ADEN bits šajā reģistrā. Ja ADEN nav uzstādīt mikrokontrolleris nepatērē enerģiju, tāpēc ieejot miega režīmā to ir vēlams atslēgt.

Pārveidošanas rezultāts tiek glabāts ADC datu reģistros – ADCH un ADCL. Pēc noklusējuma rezultāts ir izlīdzināts pa labi, bet to var izlīdzināt arī pa kreisi. ADMUX reģistrā attiecīgi izmainot bitu ADLAR (Sk. Att. 21). Ja rezultāts ir izlīdzināts pa kreisi un nav nepieciešama vairāk kā 8 bitu precizitāte, tad var nolasīt tikai ADCH

reģistru. Pretējā gadījumā vispirms jānolasa ADCL un tikai tad ADCH, lai nodrošinātu, ka rezultāts pieder vienai un tai pašai pārveidošanai. Nolasot ADCH reģistru tiek atjaunota ADC pieeja rezultāta reģistram un tas var ierakstīt jaunu rezultātu.



Att. 20 ATmega 128 ADC shēma

ADLAR = 0:

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	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Att. 21 izlīdzināšana pa labi (ADLAR=0) un pa kreisi (ADLAR=1)

ADC ir savs pārtraukums, kurš tiek izsaukts tad, kad tiek pabeigta pārveidošana. Lai aprakstītu ADC pārtraukuma apstrādes funkciju:

```
ISR(ADC_vect)
{
        //Kods
}
```

Nepārtrauktas atjaunošanas režīmā ADC nepārtraukti ciparo signālu un atjauno datu reģistrus. To izvēlas uzstādot ADFR bitu ADCSRA reģistrā. Lai uzsākto pirmo pārveidošanu ir jāuzstāda ADSC bits šajā pašā reģistrā.

ADC tiek nodrošināts arī ar pirms dalītāju, kas nodrošina ADC ar taktsignālu. To uzstāda ar ADPS bitiem ADCSRA reģistrā. To izmanto, lai palielinātu vai samazinātu rezultāta atšķirību no oriģinālā ieejas signāla. Lai pārveidotu signālu ir nepieciešami 13 ADC takts impulsi.

Multipleksora reģistrs – ADMUX

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

Att. 22 ADMUX reģistrs

7. un 6. bits – REFS1:0 - atbalsta sprieguma izvēles biti

Šie biti ir ADC atbildīgi par atbalsta sprieguma avota izvēli. Ja ieejošais ADC signāls ir ārpus šīm robežām tad tas tiek pielīdzināts vai nu maksimālajam (ja spriegums lielāks par atbalsta spriegums) vai minimālajam = 0 (pretējā gadījumā).

REFS1	REFS0	Sprieguma avota izvēle
0	0	AREF, iekšējas Vref izslēgts
0	1	AVCC ar ārējo kondensatoru pie AREF pina
1	0	Rezervēts
1	1	Iekšējais 2.56V sprieguma atbalst ar ārējo kondensatoru pie AREF pina

5. bits – ADLAR – ADC rezultāts izlīdzināts pa kreisi
Šis bits atbild par to, lai izlīdzinātu ADC rezultātu ADCH un ADCL reģistros pa kreisi (vienāds ar 1) vai pa labi (vienāds ar 0).

4.-0. bits – MUX4:0 – ieejas kanāla izvēle
Ierakstot 2-0 bitā var izvēlēties kanālu, bet 4-3 var izvēlēties vai šo signālu ir nepieciešams pastiprināt. Sīkāku paskaidrojumu skatieties ATmega 128 datu lapā 244. lpp.

Vadības un statusa reģistrs – 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	

Att. 23 ADCSRA reģistrs

7. bits – ADEN – ADC atļaušana
Uzstādot šo bitu tiek iedarbināts ADC, nodzēšot bitu tas tiek izslēgts.

6. bits – ADSC – ADC pārveidošanas uzsākšana
Vienas pārveidošanas režīmā pirms katras pārveidošanas uzsākšanas ir jāuzstāda šis bits loģiskajā 1. Nepārtrauktas pārveidošanas režīmā to nepieciešams veikt tikai vienu reizi.

5. bits – ADFR – nepārtrauktas pārveidošanas režīma izvēle
Uzstādot šo bitu tiek ieslēgts nepārtrauktas pārveidošanas režīms. Šajā režīmā nepārtraukti tiks atjaunoti datu rezultātu reģistri.

4. bits – ADIF – ADC pārtraukuma karodziņš
Šis bits tiek automātiski uzstādīts tad, kad tiek pabeigta pārveidošana un tiek atjaunoti datu reģistri. Ja ir atļaut pārtraukums tad tiek izsaukta pārtraukuma apstrādes funkcija. Tas tiek nodzēsts, kad pārtraukuma apstrādes funkcija tiek pabeigta. Alternatīvi to var nodzēst ierakstot loģisko 1 karodziņā.

3. bits – ADIE – ADC pārtraukuma atļaušana
Uzstādot šo bitu tiek atļauta ADC pārtraukuma apstrādes funkcijas izpildīšana.

2.-0. bits – ADPS2:0 – ADC pirms dalītāja izvēles biti
Ar šiem bitiem tiek uzstādīta ADC pirms dalītāja vērtība. Par pamatu tiek ņemta takts frekvence no XTAL un dalīta ar attiecīgo dalītāju.

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

Tā kā laboratorijā nav pieejami nekādi sensori vai signāla avota ģeneratori tad šo uzdevumu veiksim lietojumprogrammā *Proteus ISIS 7*, kas ir paredzēta dažādu simulāciju veikšanas, kas ne tikai balstās uz mikrokontrolleriem, bet arī uz citiem elektroniskiem elementiem, piemēram, rezistori, kondensatori, kā arī dažādi izpildmehānismi, piemēram, dzinēji, utt. Aprakstu kā darboties ar *ISIS 7* meklējiet I pielikumā.

Atskaitē

Atskaitē īsumā ir jāapraksta ATmega128 ADC; jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto izveidotais programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi:

1. Paskaidrojiet kā notiek analogs cipars pārveidošana Atmega128 mikrokontrollerī (Att. 20)?
2. Ar ko atšķiras 8 bitu ADC režīms no 10 bitu režīma?
3. Kā ir saistīta *Naikvista kritērijs* ar ADC pirms dalītāju?

5. laboratorijas darbs - Sargtaimeris (*Watchdog timer* - WDT)

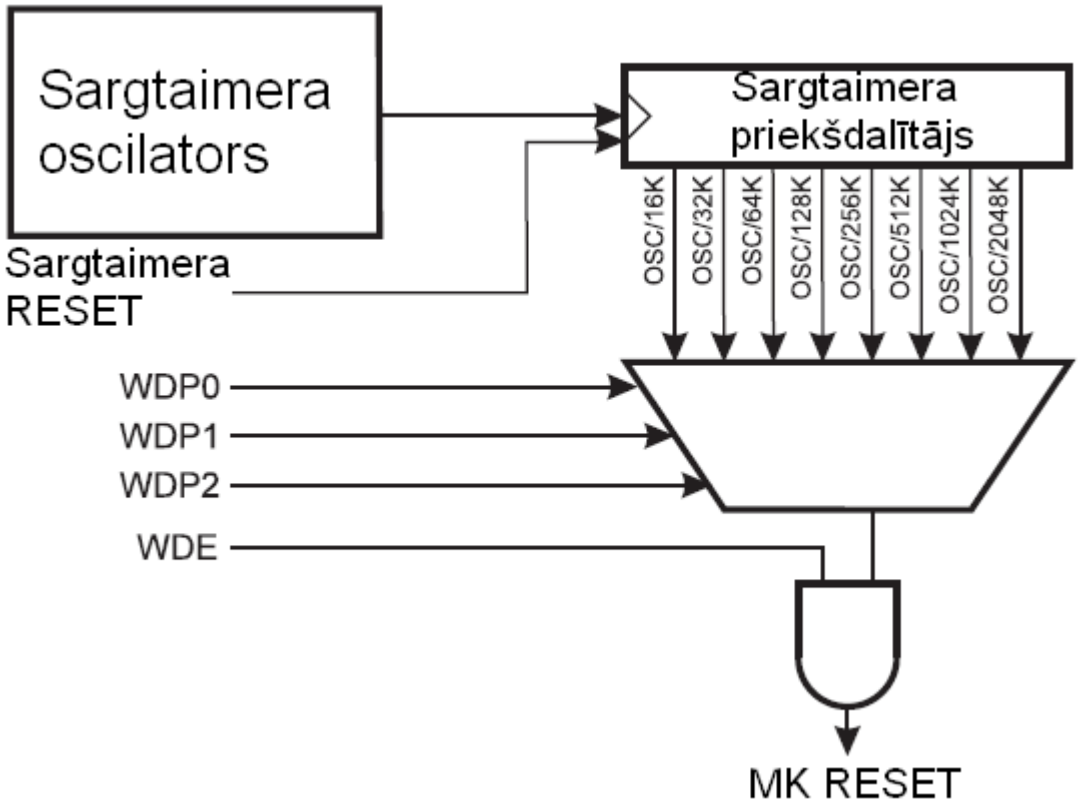
Uzdevums

Papildināt 2. laboratorijas darbu ar sargtaimeri. Uzstādīt sargtaimera taimautu/atiestatīšanas laiku vienādu ar aptuveni 2 sekundēm. Pierādīt (ar gaismas diodēm), ka sargtaimeris strādā ar izvēlēto taimauta laiku.

Teorētiskais apraksts:

Sargtaimeris

Sargtaimeris (*watchdog timer*), dažreiz saukts arī par COP (skaitļotājs darbojas pareizi), kontrolē vai programmas darbības tiek veiktas noteiktā laikā, t.i. kontrolē vai programma nav "uzkārusies". Bieži tiek lietots, lai pārbaudītu programmas izpildes rezultātus. Vienreiz to ieslēdzot taimeris sāk skaitīt atpakaļ, kad saskaita līdz 0 tiek izsaukta RESET instrukcija kura atiestata mikrokontrolleri kā arī sāk no jauna programmas izpildi. Lai šī atiestatīšana nenotiktu ir jāatslēdz sargtaimeris vai to ir jāatjauno. Ja programma novirzās no tās normālās izpildes, tad ar sargtaimeri tā tiek palaista no jauna.



Att. 24 Sargtaimera bloka diagramma

Sargtaimerim ir savs iekšējais oscilators (1 MHz), kas palielina viņa drošumu. Tā tipiskais barošanas spriegums ir V_{cc} = 5V. Kontrolējot sargtaimera priekš dalītāju var regulēt WDR (*Watchdog Timer Reset*) intervālu – laiku pēc kura sargtaimeris veiks mikrokontrollera atiestatīšanu.

WDP2	WDP1	WDP0	WDT oscilatora ciklu skaits	Tipisks taimauts pie Vcc = 3.0V	Tipisks taimauts pie Vcc = 5.0V
0	0	0	16K (16,384)	14.8 ms	14.0 ms
0	0	1	32K (32,768)	29.6 ms	28.1 ms
0	1	0	64K (65,536)	59.1 ms	56.2 ms
0	1	1	128K (131,072)	0.12 s	0.11 s
1	0	0	256K (262,144)	0.24 s	0.22 s
1	0	1	512K (524,288)	0.47 s	0.45 s
1	1	0	1,024K (1,048,576)	0.95 s	0.9 s
1	1	1	2,048K (2,097,152)	1.9 s	1.8 s

WDR – tā ir instrukcija, kas atiestata pašu sargtaimeri. Sargtaimeris arī tiek atiestatīts, kad to atslēdz un kad notiek čipa atiestatīšana (Chip Reset). Astoņi dažādi taktēšanas ciklu periodi var būt izvēlēti, lai noteiktu atiestatīšanas periodu. Ja atiestatīšanas periods beigsies un cits WDR netiek padots, ATmega128 pārlādēsies.

Lai novērstu nejaušu sargtaimera atslēgšanos kā arī sargtaimera nejaušu skaitīšanas perioda maiņu, tiek lietoti 3 dažādi drošības līmeņi, kas tiek iestādīti ar *Fuse* M103C un WDTON bitiem:

M103C	WDTON	Drošī-bas lī-menis	WDT sākuma stāvoklis	WDT izslēg-šana	Timauta maiņa
Nav ieprogrammēts	Nav ieprogrammēts	1	Izslēgts	Secība laikā	Secība laikā
Nav ieprogrammēts	Ieprogrammēts	2	Ieslēgts	Vienmēr ieslēgts	Secība laikā
Ieprogrammēts	Nav ieprogrammēts	0	Izslēgts	Secība laikā	Bez aizliegumiem
Ieprogrammēts	Ieprogrammēts	2	Ieslēgts	Vienmēr ieslēgts	Secība laikā

0- drošības līmenis. Šis stāvoklis ir savietojams ar sargtaimera darbību Atmega103. Taimeris sākumā ir izslēgts, bet var tikt ieslēgts, ierakstot WDE bitu „1” bez jebkādiem aizliegumiem. Skaitīšanas periods var tikt mainīts jebkurā laikā bez jebkādiem aizliegumiem.

1- drošības līmenis. Šajā stāvoklī sākotnēji sargtaimeris ir izslēgts, bet var tikt ieslēgts ierakstot WDE bitu „1” bez jebkādiem aizliegumiem. Ir jāizpilda noteiktu darbu secība, lai izmainītu skaitīšanas periodu, vai izslēgtu sargtaimeri:

1. Vienā operācijā jāieraksta vienineks WDCE un WDE bitos. WDE bitā jāieraksta vienineks, neatkarīgi no iepriekšējās WDE bita vērtības.
2. Nākošo četru taksu laikā vienā operācijā jāieraksta WDE un WDP bitus, izvēloties attiecīgo taimauta laiku, bet notīrot WDCE bitu.

2- Drošības līmenis. Šajā stāvoklī sargtaimeris ir vienmēr ieslēgts, un WDE bits tiks vienmēr nolasīts kā „1”. Arī šeit ir noteikta darbību secība, lai mainīt sargtaimera skaitīšanas periodu. Lai izmainītu skaitīšanas periodu, ir jāievēro sekojoša procedūra.

1. Vienas operācijas laikā jāieraksta vieninieks WDCE un WDE bitos. Lai arī WDE ir vienmēr iestatīts, WDE vienalga ir jāiestata vieniniekā, lai iesāktu laika secību.
2. Nākošo četru taksu laikā vienā operācijā jāieraksta WDP biti, kā ir iecerēts, bet notīrot WDCE bitu. WDE bitā ierakstītā informācija ir nesvarīga.

Sargtaimera vadības reģistrs – WDTCR

Bits	7	6	5	4	3	2	1	0	
	–	–	–	WDCE	WDE	WDP2	WDP1	WDP0	WDTCR
Lasīt/Rakstīt	L	L	L	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	

Att. 25 Sargtaimera vadības reģistrs WDTCR reģistrs

7., 6., 5. biti ir rezervētie biti ATmega128 un vienmēr tiek nolasīti kā nulles.

4. bits- WDCE: Watchdog Change Enable
Šis bits tiek izmantot, lai norādītu par sargtaimera uzstādījumu (taimauta laiku, ieslēgt/izslēgt sargtaimeri) mainīšanu. Ierakstot loģisko „1”, pēc četrām taktīm aparatūra to notīra.

3. bits - WDE: Watchdog Enable
Kad WDE bits ir ierakstīts kā loģiskais „1”, sargtaimeris ir ieslēgts un ja WDE ir ierakstīts kā loģiskā „0”, sargtaimeris ir izslēgts. WDE bitu var notīrīt tikai tad, ja WDCE bits ir loģiskais „1”. Lai ieslēgtu vai izslēgtu ir jāveic sekojošas procedūras:

- Vienā operācijā ieraksta loģisko vieninieku WDCE un WDE. Loģiskajam vieniniekam jābūt ierakstītam WDE, lai arī tas ir ierakstīts jau iepriekš, pirms vel atslēgšanas operācija bija sākusies.
- Nākamajās četrās taktīs jāieraksta loģiskā „0” WDE. Tas atslēgs sargtaimeri. Drošības līmenī 2, nav iespējams atslēgt sargtaimeri.

Atskaitē

Atskaitē īsumā ir jāapraksta ATmega128 mikrokontrollera atmiņas veidi; jāizveido īss rādītāju apraksts; jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto izveidotais programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi

1. Paskaidrojiet sargtaimera darbības principu un pielietojumu.
2. Uz shēmas (Att. 24) paskaidrojiet sargtaimera darbību.
3. Kur programmas pirmkodā ir vispiemērotākā vieta WDR veikšanai?
4. Paskaidrojiet kāpēc jāizmanto konkrēta darbību secība sargtaimera uzstādījumu mainīšanai.

6. laboratorijas darbs – USART modulis

Uzdevums

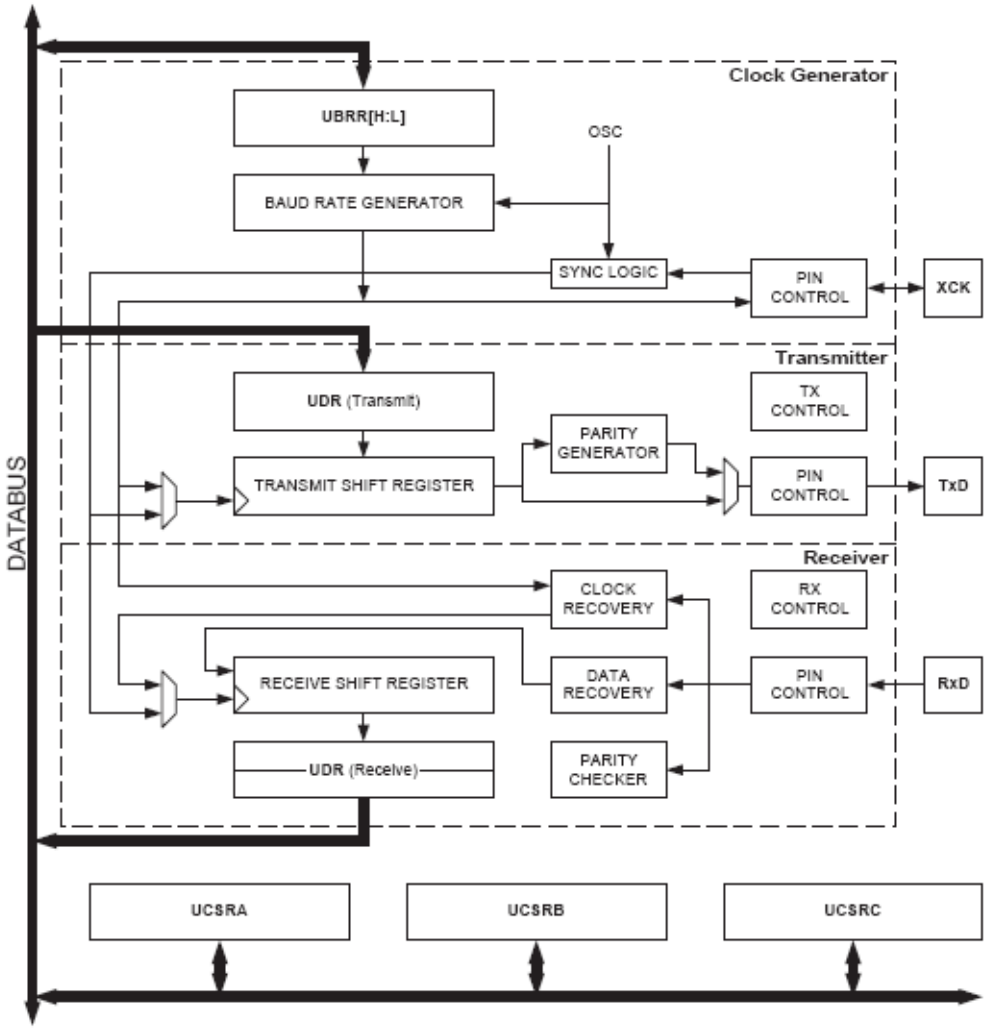
1. Izmantojot Atmega128 aprakstu nokonfigurēt UART moduli datu sūtīšanai un saņemšanai (Tx un Rx). Datu freima formāts 8N1;
2. Pārsūtīt no CharonII uz datoru frāzi, kas pieprasa ievadīt skrejošās gaismas aizkavi;
3. Saņemt no datora kādu skaitli (vēlams arī divciparu skaitļus), kas veidotu skrejošās gaismas aizkavi.

Teorētiskais apraksts:

USART modulis

USART (*Universal Synchronous asynchronous receiver/transmitter*) modulis – ir skaitļošanas tehnikas daļa, kas pārveido datus no paralēlas formas seriālajā. USART parasti tiek izmatots kopā ar citiem komunikācijas standartiem, piemēram EIA RS-232.

Parasti USART ir atsevišķs vai kā daļa no mikroshēmas, kas tiek izmantots seriāla pārraidē starp datoru un perifērijas ierīcēm caur seriālu portu (RS232). ATmega 128 mikrokontrollerī arī ir iebūvēts USART modulis (Att. 26), kurš aktivizē sazināšanas iespēju starp mikrokontrolleri un, piemēram, datoru.



Att. 26 USART modulis Atmega128 mikrokontrollerī

USART modulis ATmega128 mikrokontrollerī ir sadalīts trīs blokos: taktsignāla ģeneratora, raidītāja un saņēmēja (angl. *Clock Generator, Transmitter, Receiver*).

USART moduli veido:

- UBRR[H:L] –reģistrs, kas uzdot ātrumu. Ierakstītais tur skaitlis noteic takts frekvences dalīšanas vērtība, vajadzīgais koeficients tiek izskaitļots vai nu pēc formulas, vai (standarta kvarcam) pēc tabulas.

Bits	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Lasīt/Rakstīt	L	L	L	L	L/R	L/R	L/R	L/R	
	L/R	L/R	L/R	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

- Boda ātruma ģenerators (angl. BAUD RATE GENERATOR) – pirms dalītājs , kas ģenerē pārraidāmos impulsus ar noteiktu ātrumu, kuru var noteikt pēc formulas

$$BAUD = \frac{f_{osc}}{16(UBRR+1)}$$, kur

$\mathbf{f_{osc}}$ – mikrokontrollera takts frekvence, **UBRR** – UBRR[H:L] reģistra vērtība.

- Sinhronizēšanas loģika (angl. SYNC LOGIC) – nepieciešama sinhronai pārraidei.
- Izvada kontrole (angl. PIN CONTROL) – nodrošina informācijas izvadi uz līniju.
- Taktēšanas impulsa pārnese (angl. XCK) – šā līnija tiek izmantota tikai sinhronā režīmā taktēšanas impulsu pārraidei/saņemšanai.
- UART datu reģistrs (angl. UDR) – reģistrs, kurš saglabā saņemtos vai sūtītos datus.
- Pārraides pārbīdes reģistrs (angl. TRANSMIT SHIFT REGISTER) – speciāls reģistrs, kurš ar katru padoto taktsimpulsu izvada vienu bitu.
- Paritātes ģenerators (angl. PARITY GENERATOR) – izskaitļo pārraidāmā freima datu paritātes bitu. Ja paritātes ģenerēšana ir atļauta, tad raidītāja vadības loģika ievietos freimā paritātes bitu starp pēdējo datu bitu un stop bitu.
- Pārraides vadības mezgls (angl. TX CONTROL) – nodrošina korektu freima pārraidi.
- Pārraides līnija (angl. TxD) – līnija, pa kuru tiek pārraidīts freims.
- Saņēmēja pārbīdes reģistrs (angl. RECEIVE SHIFT REGISTER) - speciāls reģistrs, kurš ar katru padoto taktsimpulsu saglabā vienu bitu.
- Taktsimpulsu atgūšanas bloks (angl. CLOCK RECOVERY) – bloks tiek izmantots iekšējā taktsimpulsa ģeneratora un ienākošā freima sinhronizācijai.
- Datu atgūšanas bloks (angl. DATA RECOVERY) – iztver katru ienākošo bitu, kā arī satur zemo frekvenču filtru, kas nodrošina noturīgumu pret kļūdām.
- Paritātes pārbaude (angl. PARITY CHECKER) – speciāls bloks, kurš salīdzina saņemto paritātes bitu ar no freima datiem izskaitļotu.
- Statusa reģistrs UCSRA – speciāls statusa reģistrs:

Bits	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Lasīt/Rakstīt	L	L/R	L	L	L	L	L/R	L/R	
Sākumvērtība	0	0	1	0	0	0	0	0	

Att. 27 Statusa reģistrs UCSRA

  - RXCn – norāda, vai ir pabeigta datu saņemšana (0, kad datu buferis ir tukšs);
  - TXCn – norāda, kad datu pārraide ir pabeigta (t.i. pārbīdes reģistrs ir tukšs);
  - UDREn – karodziņš, kurš norāda, vai UDR reģistrs var saņemt datus;
  - FEn – norāda, ka ir notikusi freima kļūda;
  - DORn – datu pārpilde, notiek kad, saņemšanas buferis ir pilns, pārbīdes reģistrā ir jauns simbols un jauns starta bits tiek konstatēts;
  - UPEn – parāda vai ir notikusi paritātes kļūda;
  - U2Xn – divkāršots USART pārraides ātrumu asinhronajā režīmā;
  - MPCMn – ierakstot „1” iestāda vairākprocesoru komunikācijas režīmu.

- Statusa reģistrs UCSRB – speciāls statusa reģistrs:

Bits	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Lasīt/Rakstīt	L/R	L/R	L/R	L/R	L/R	L/R	L	L/R	
Sākumvērtība	0	0	0	0	0	0	0	0	

Att. 28 Statusa reģistrs UCSRB

- RXCIEn – ierakstot 1 aktivizē pārtraukumu uz datu saņemšanas pabeigšanu;
- TXCIEn – ierakstot 1 aktivizē pārtraukumus uz datu sūtīšana pabeigšanu;
- UDRIEn – ierakstot 1 aktivizē pārtraukumus uz datu reģistra iztukšošanu;
- RXENn – ierakstot 1 aktivizē USARTn uztvērēju;
- TXENn – ierakstot 1 aktivizē USARTn raidītāju;
- UCSZn2 – uzstāda datu bitu daudzumu freimā;
- RXB8n – šeit tiek saglabāts saņemtais 9. datu bits;
- TXB8n – šeit tiek saglabāts nosūtītais 9. datu bits;

- Statusa reģistrs UCSRC – speciāls statusa reģistrs:

Bits	7	6	5	4	3	2	1	0	
	–	UMSELn	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Lasīt/Rakstīt	L/R	L/R	L/R	L/R	L/R	L/R	L/R	L/R	
Sākumvērtība	0	0	0	0	0	1	1	0	

Att. 29 Statusa reģistrs UCSRC

- UMSELn – uzstāda sinhrono (1) vai asinhrono (0) USART moduļa režīmu;
- UPMn1:0 – uzstāda un aktivizē paritātes izskaitļošanu un pārbaudīšanu;
- USBSn – uzstāda stop bitu skaitu:

USBSn	Stop bits (-i)
0	1-bit
1	2-bits

- UCSZN1:0 – uzstāda datu bitu daudzumu:

UCSZn2	UCSZn1	UCSZn0	Simbola izmērs
0	0	0	5-bitu
0	0	1	6-bitu
0	1	0	7-bitu
0	1	1	8-bitu
1	0	0	Rezervēts
1	0	1	Rezervēts
1	1	0	Rezervēts
1	1	1	9-bitu

- UCPOLn – izmanto tikai sinhronajā režīmā, uzstāda attiecību starp datu izvades izmaiņām un datu ievades izmaiņām signālā un sinhronajās taktīs;

USART datu pārraides reģistrs (UBRR) un atpakaļ skaitītājs ir savienots tā ka tas funkcionē kā programmējams pirms dalītājs, jeb datu pārraides ātruma ģenerators. Atpakaļ-skaitītājs, kurš darbojas izmantojot sistēmas pulksteni (f_{osc}), tiek piepildīts ar UBRR vērtību katru reizi kad skaitītājs saskaita atpakaļ līdz nullei. Šis pulkstenis ir datu pārraides ģeneratora pulksteņa izvads. Raidītājs dala datu pārraides pulksteņa izvadi ar 2, 8, vai 16 atkarībā no izvēlētā režīma.

Darbības režīms	Datu pārraides ātruma izskaitļošanas vienādojums	UBRR vērtības izskaitļošanas vienādojums
Asinhronais normālais režīms (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asinhronais divkāršota ātruma režīms (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Sinhronais vedēja režīms	$BAUD = \frac{f_{OSC}}{2(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

BAUD – Datu pārraides ātrums (biti sekundē, bps)
f_{osc} – sistēmas oscilatora takts frekvence.
UBRR – UBRRH un UBRRL saturs

USART datu sūtītājs

USART datu pārraidītājs tiek ieslēgts iestatot pārraides atļaušanas (TXEN) bitu UCSRB reģistrā. Kad pārraidītājs ir ieslēgts, TxD pina porta darbība tiek pārvaldīta ar USART un tiek nodota funkcijai kā pārraidītāja seriālā izeja. Datu pārraides ātrums, darbošanās režīms kā arī freima formāts ir jāuzstāda pirms pārraides. Ja sinhronā pārraide netiek izmantota, takts uz XCK pina tiks pārtverta un tiks izmantots pārraides taktētājs.

Datu pārraide tiek uzsākta ielādējot pārraides buferī pārraidei domātos datus. CPU var ielādēt datus pārraides buferī ierakstot UDR reģistrā I/O vietu. Buferī ievietotie dati no pārraides bufera tiek pārvietoti uz pārbīdes reģistru, kad pārraides reģistrs ir gatavs jauna freima sūtīšanai. Pārbīdes reģistrā ielādē jaunus datus, kad tas ir bezdarbības režīmā (nav izejošu sūtījumu), vai tūlīt pēc pēdējā stop bita no iepriekšējā freima pārraides. Kad pārbīdes reģistrs tiek pielādēts ar jauniem datiem, tas pārraidīs vienu pilnu freimu reģistrā norādītajā pārraides ātrumā.

USART freima formāts tiek uzstādīts UCSRB un UCSRC izmantojot UCSZ2:0, UPM1:0 un USBS bitus. Saņēmējs un sūtītājs izmanto vienus un tos pašus uzstādījumus. Izmainot uzstādījumus jebkuram no šiem bitiem tiks pārtraukta notiekošā komunikācija starp saņēmēju un sūtītāju.

USART rakstzīmes izmēra(UCSZ2:0) bitiem tiek izvēlētas datu bitu skaits freimā. USART paritātes režīma (UPM1:0) biti ieslēdz un iestāda paritātes bitu tipu. Izvēle starp to vai būs viens vai 2 stop biti tiek izdarīta ar USART stop bita izvēles (USBS) bitu. Saņēmējs ignorē otro stop bitu. Freima kļūda (*Frame Error*) tiks atklāta gadījumos, kad pirmais stop bits ir nulle.

USART datu saņēmējs

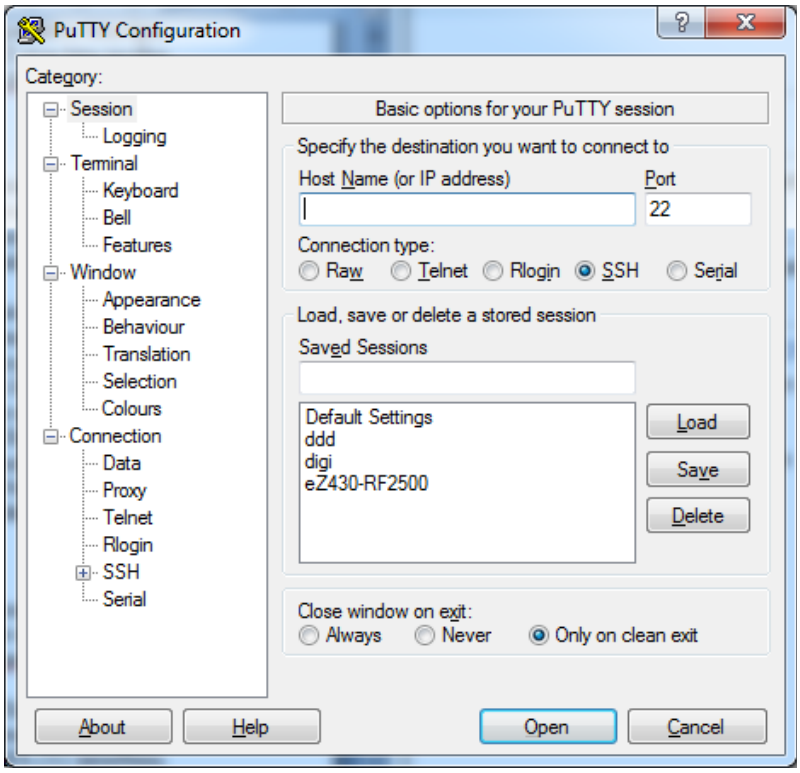
USART saņēmējs tiek ieslēgts ierakstot saņēmēja ieslēgšanas (RXEN) bitu UCSRB reģistrā “1”. Kad saņēmējs ir ieslēgts normālā RxD pina operācija tiek pārrakstīta USART un tiek iedota funkcija, kā saņēmēja seriālā ieeja. Datu pārraidīšanas ātrumu un freima formātu uzstāda pirms jebkura seriālā saņemšana ir pabeigta.

Saņēmējs sāk datu saņemšanu, kad tiek saņemts pirmais derīgais *start* bits. Katrs bits, kurš seko pēc *start* bita tiek samplēts pēc datu pārsūtīšanas ātruma un pārbīdīts uz saņēmēja pārbīdes reģistru, kamēr nav saņemts freima pirmais *stop* bits. Nākošais stop bits tiks ignorēts. Kad ir saņemts pirmais stop bits, tas nozīmē, kad viss freims ir ievietots pārbīdes reģistrā, pārbīdes reģistra saturs tiek pārvietots uz saņēmēja buferi. USART saņēmējam ir viens karogs, kas norāda saņēmēja stāvokli. Datu saņemšanas pabeigšanas (RXC) karogs norāda vai ir nenolasīti dati saņēmēja buferī. Šis karogs ir “1”, kad nenolasīti dati ir saņēmēja buferī, un nulle, kad saņēmēja buferis ir tukšs. Ja saņēmējs ir atslēgts (RXEN=0), saņēmēja buferis tiks nodzēsts un tātad RXC bits kļūs par “0”.

Seriālā komunikācija uz datora

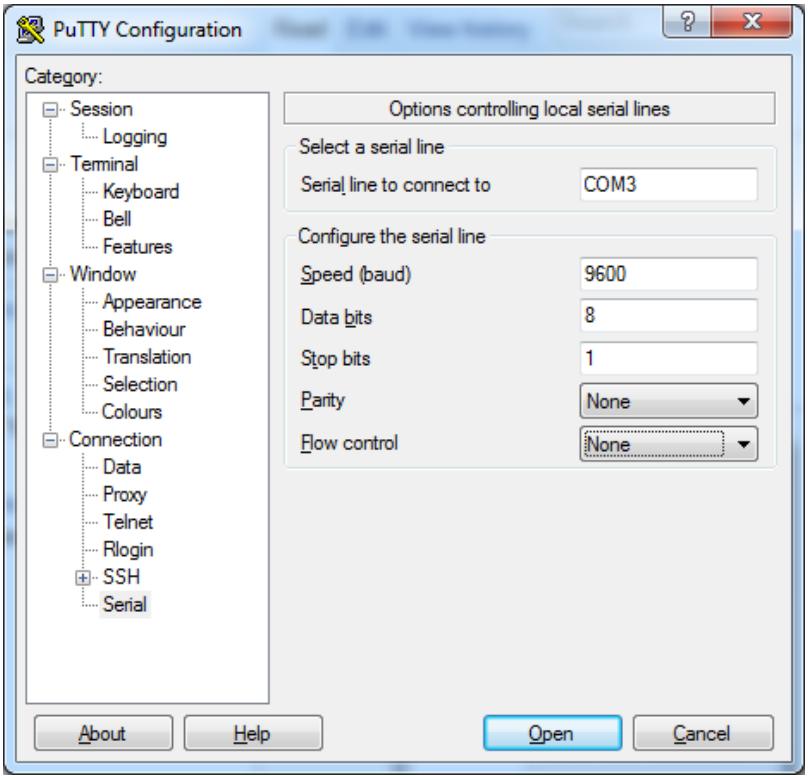
Lai būtu iespējams komunicēt mikrokontrollerim ar datoru tiem ir jābūt savienotiem ar RS-232 kabeli. Uz datora tas tiek attēlots kā seriālais savienojums jeb COM. Visus pieejamos COM interfeisus var aplūkot *Device Manager* (*Control Panel ->System->Harware->Device Manager* sadaļā *Ports*).

Uz katra datora ekrāna ir atrodama programma **PuTTY** (Sk. Att. 30). Tā ir neliela utilīta dažādu terminālu emulēšanai un interfeisu klausīšanai. Mēs to izmantosim kā datu saņēmēju un tur tiks attēlota informācija, ko sūta CharonII caur RS-232 interfeisu.



Att. 30 Utilīta PuTTY

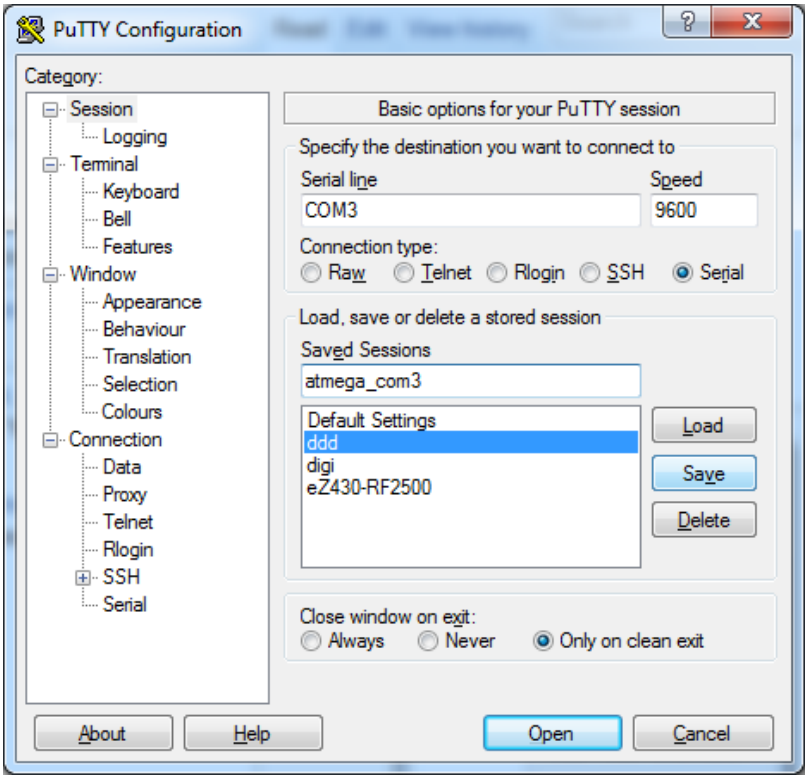
Pirmais, ko ir nepieciešams uzstādīt ir freima formāts. To dara nospiežot uz „*Serial*” *Category* logā (Sk. Att. 31).



Att. 31 Freima formāta uzstādīšana

Šeit ir jāizvēlas viens no pieejamiem COM portiem; jāievada datu pārraides ātrums - 9600, datu bitu skaits – 8, stop bitu skaits – 1 paritāte – netiek izmantota, plūsmas kontrole – netiek izmantota.

Kad tas ir aizpildīts izvēlamies „*Session*" *Category* logā un atķeksējam *Serial* pie *Connection Type* (Sk. Att. 32). Šeit tiek parādīts seriālais ports un datu pārraides ātrums, ko mēs ievadījām. Lai nebūtu katru reizi tas jāveic atkārtoti šos uzstādījumus var saglabāt ierakstot *Saved Sessions* laikā kādu nosaukumu (piemēram, atmega_com3) un nospiežot *Save*. Tagad ikreiz atverot PuTTY pietiek uzklikšķināt uz šī nosaukuma sarakstā un nospiest *Load*, lai ielādētu izvēlētos uzstādījumus.



Att. 32 Seriālā savienojuma izvēle un saglabāšana

Lai atvērtu savienojumu jānospiež poga *Open* un atvērsies seriālā porta komunikācijas logs (Sk.). Šajā logā tad arī ir jāparādās datiem ko sūta mikrokontrolleris un šeit datus ir iespējams nosūtīt mikrokontrollerim (katrs nospiestais taustiņš tiek nosūtīts caur šo interfeisu).

Atskaitē

Atskaitē kodolīgi ir jāapraksta ATmega128 USART modulis (galvenās īpašības); jāizveido programmas pirmtekstam algoritma blokshēmu; jāievieto izveidotais programmas pirmkods ar paskaidrojumiem; jāuzraksta secinājumi.

Jautājumi:

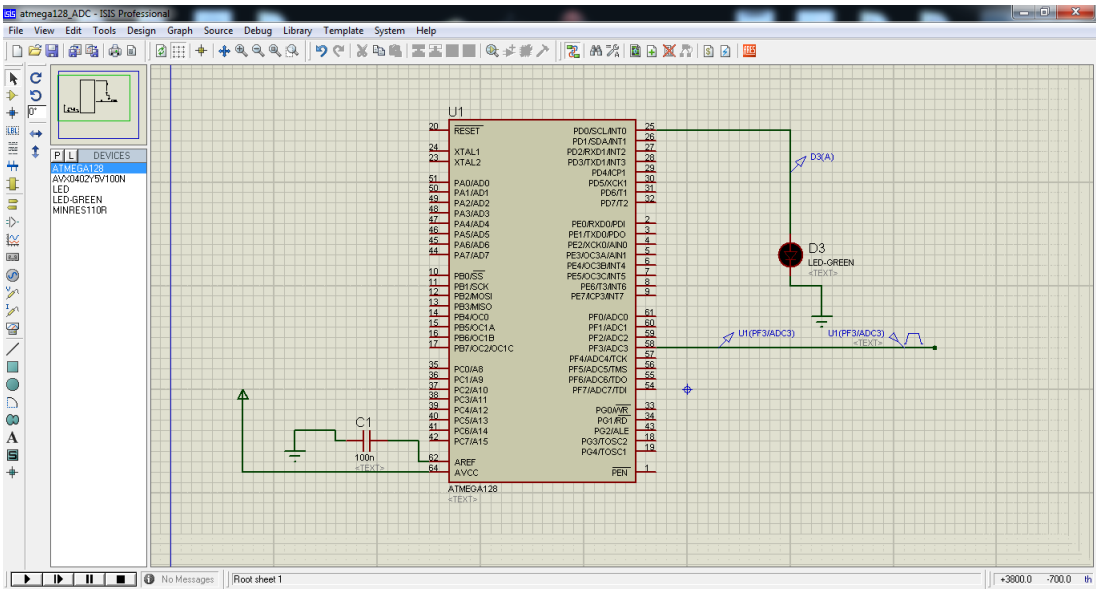
1. UART un USART moduļu starpība.
2. Ar ko raksturojas UART komunikācija?
3. Paskaidrojiet jūsu programmas kodu.

I pielikums

Darbs ar ISIS 7

Darbs ar šo lietojumprogrammu notiks 4. laboratorijas darba ietvaros – ADC programmēšana. Šajā aprakstā apskatīsim kā pievienot jaunas detaļas modelim, ielādēt un palaist programmu darba sagatavē un modificēt modeli. Programma, protams, vispirms ir jānokompilē izmantojot AVR Studio 4.

ISIS 7 darba virsma



Att. 33 ISIS 7 darba virsma

Atverot laboratorijas darba sagatavi atvērsies Att. 33 redzamā shēma. Tajā var redzēt:

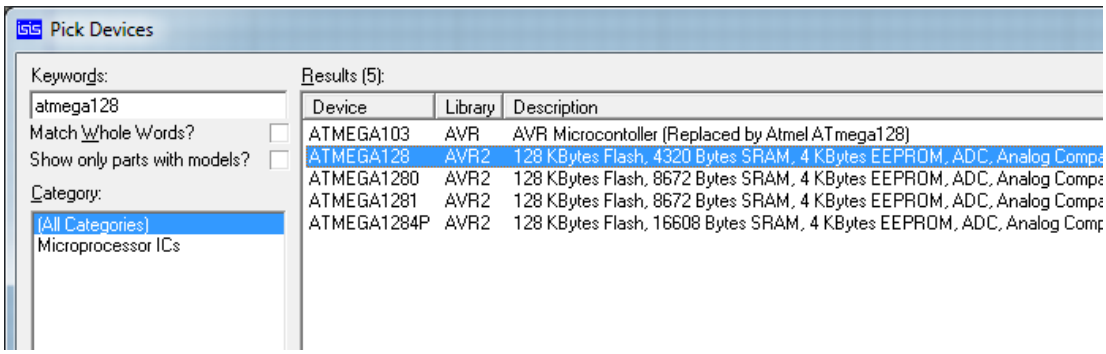
- U1 – ATmega128;
- C1 – 100 nF kondensators, kas pievienots ADC ieejai AREF;
- D3 – zaļa LED;
- U1(PF3/ADC3) – takts impulsa ģenerators, kas pievienot ADC 3. kanālam;
-  - barošanas avots (5V);
-  - zemējums;

Jaunas detaļas pievienošana

Pirms turpināt ar programmu apskatīsim, ka var pievienot kādu jaunu objektu modelim. Vispirms ir jāatver *Pick Devices*, jeb komponentu bibliotēka. To var atvērt nospiežot taustiņu **P** uz klaviatūras vai **Library -> Pick Devices/Symbol**. Laukā **Keywords** ierakstiet nepieciešamo detaļas nosaukumu, piemēram, atmega128, un logā **Results** izvēlieties jau konkrētu detaļas modeli ar dubultklikšķi (Sk. Att. 34). Šī detaļa tiek pievienota sarakstam **DEVICES** (kreisajā malā, ja nav redzams DEVICES

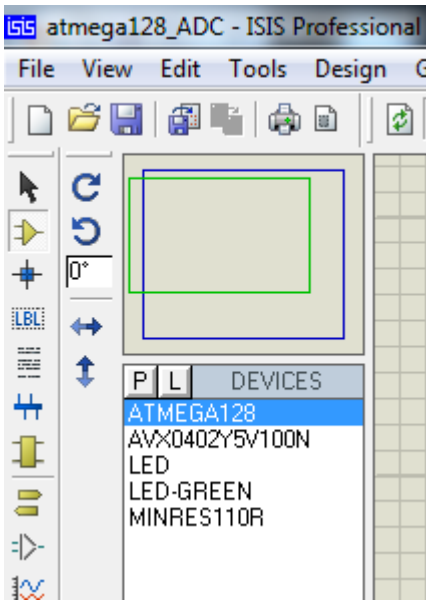
saraksts tad nepieciešams ieslēgt **Component Mode** nospiežot uz ikonas:  , kas arī atrodas kreisajā malā.

To detaļu, kuru vēlaties pievienot modelim iezīmē, vienreiz uz tās noklikšķinot. Savukārt pievieno modelim nospiežot kreiso taustiņu uz darba virsmas (iekš zilās

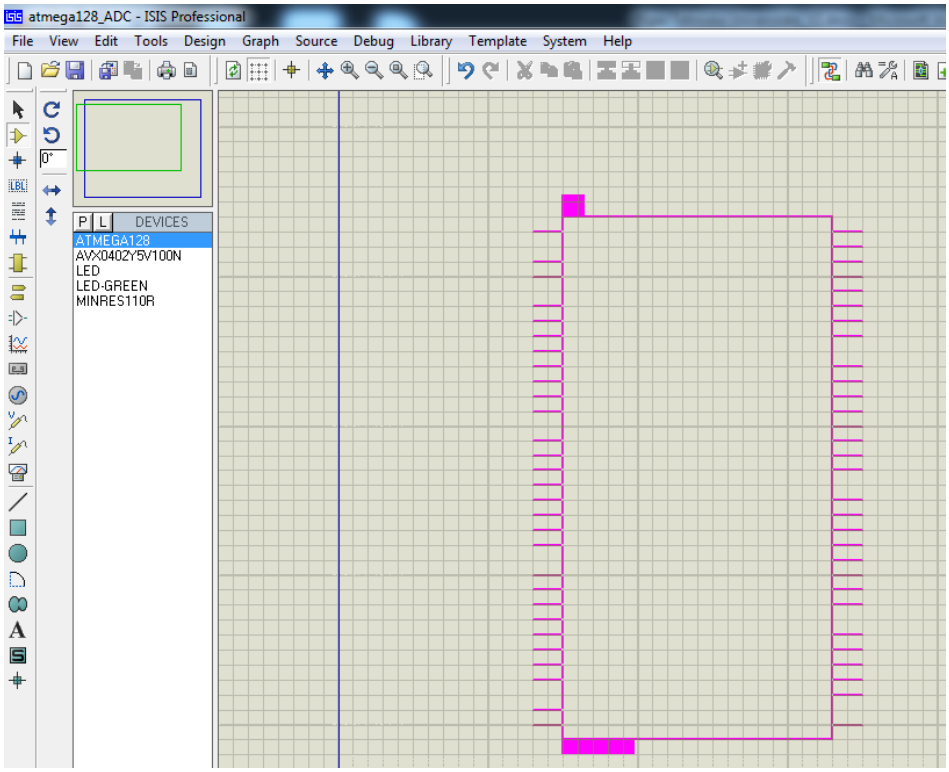


Att. 34 Detaļas izvēles piemērs

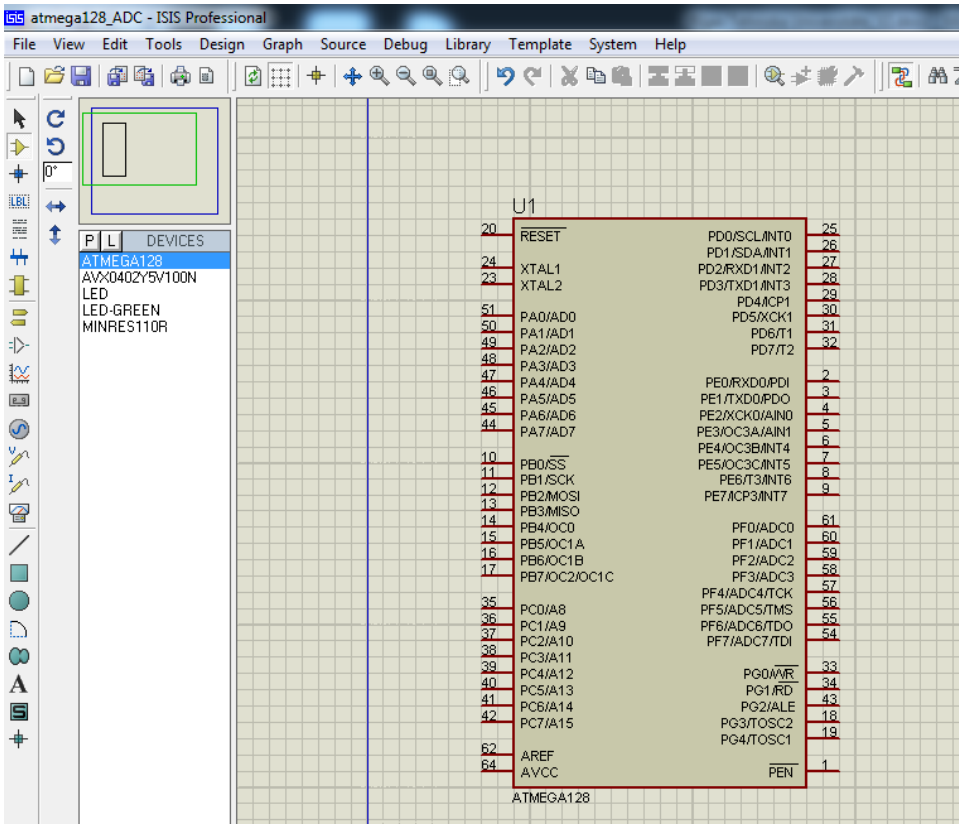
līnijas, viss, kas ir ārpus šīs līnijas tiks ignorēts), parādīsies gaiši violets detaļas modelis, noklikšķinot otrreiz detaļa tiek pievienota modelim (attiecīgi Sk. Att. 35, Att. 36 un Att. 37)



Att. 35 Detaļas izvēle



Att. 36 Pirmais klikšķis



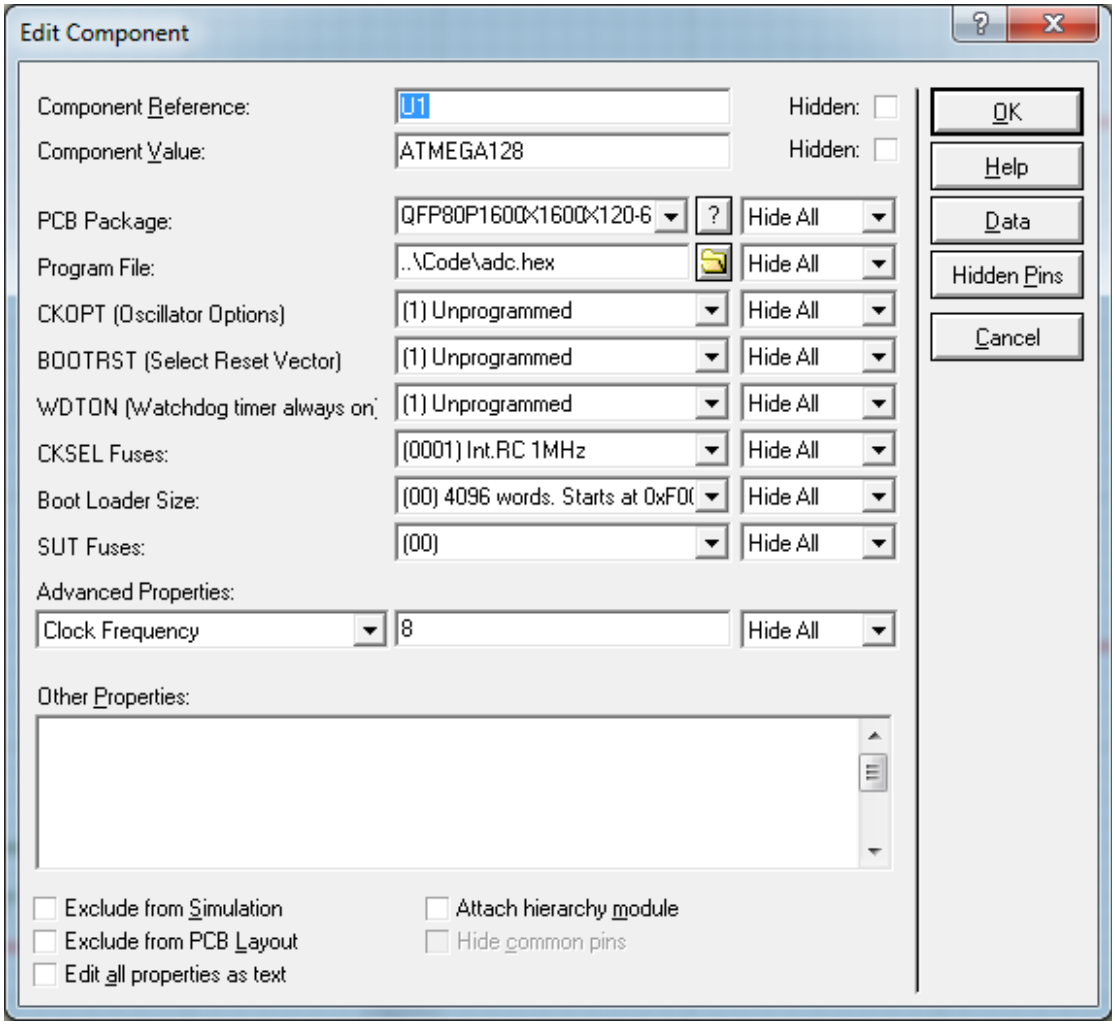
Att. 37 Otrais klikšķis

Tagad detaļa ir veiksmīgi pievienota. Līdzīgi arī rīkojas citiem elementiem, ko grib pievienota modelim.

Programmas pirmteksta pievienošanas modelim

Kā jau tika minēts programmai ir jābūt jau iepriekš nokompilētai.

Lai pievienotu programmas kodu mikrokontrollerim ir nepieciešams divreiz noklikšķināt uz mikrokontrollera. Atvērsies logs **Edit Component** (Sk. Att. 38). Logā **Program File** jānospiež uz mapītes ikonas un ir jāatrod mapīte, kurā esat saglabājis savu laboratorijas darba projektu, savukārt šajā mapē jāatrod fails ar paplašinājumu *.hex (piemēram, **adc.hex**). Kad atrasts un izvēlēts pareizais fails, aizveram logu ar OK.

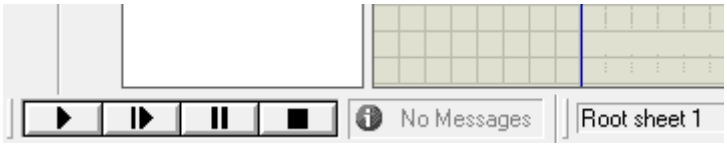


Att. 38 Edit Component logs

Simulācijas palaišana

Simulācija tiek palaista no programmas pamat loga. Kreisajā apakšējā logā ir četras pogas, kuras atbild par simulāciju (Sk. Att. 39):

- **Play** – simulācijas palaišana;
- **Step** – simulācija ar soli
- **Pause** – simulācijas nopauzēšana;
- **Stop** – simulācijas apturēšana un atgriešanās modeļa rediģēšanas režīmā;



Att. 39 Simulācijas pogas

Kad simulācija tiek palaista ar **Play**, šī poga iekrāsojas zaļā krāsā un blakus šīm pogām parādīsies ziņojumi par kļūdām, ja tādas ir, un simulācija tiks apturēta, vai arī brīdinājuma ziņojumi, vai vienkārši statusa ziņojumi (attiecīgi sarkanā, dzeltenā un zaļā krāsā tiks iekrāsot izsaukumu zīmes ikona). Ja ir bijušas kļūdas, tad jāpārbauda modelis vai arī programmas pirmkods, jo šeit var parādīties arī kļūdas saistībā ar nekorekti izveidotu pirmkodu.

Ja palaista simulācija no parauga, tad kļūdām nevajadzētu būt. Pie mikrokontrollera līnijām parādās krāsaini kvadrātiņi:

- Sarkans kvadrātiņš nozīmē, ka uz līniju tiek padots loģiskais 1;
- Zils kvadrātiņš nozīmē, ka uz līniju tiek padots loģiskais 0;
- Pelēks kvadrātiņš nozīmē, ka līnijai nav noteikts stāvoklis;

Situācijai jābūt sekojošai: ikreiz, kad uz ADC ieejas līnijas kvadrātiņš iekrāsojas sarkans, arī zaļajai diodei ir jāiedegas. Par to atbild iepriekš sarakstītā programma – tiek pārbaudīts ieejas spriegums uz ADC 3. kanāla un ikreiz, kad tas lielāks par 2,5 V tiek izvadīts loģiskais 1 uz PORTD 0. pina, kas attiecīgi pievienots LED.

Pārveidojamā signāla nomaiņa

Viens no Jūsu uzdevumiem bija nomainīt ieejas signālu no impulsu signāla uz sinusoīdu. Lai to varētu izdarīt jāveic dubultklikšķis uz U1(PF3/ADC3).

Atvērsies **Pulse Generation Properties**. Šajā logā tad arī izvēlaties sinusoīdu.