Rīgas Tehniska Universitāte

Datorvadības, automātikas un datortehnikas institūts

Datoru tīklu un sistēmas tehnoloģijas katedra

Mikroprocesoru tehnika
Laboratorijas darbs Nr. 3

Asist. R. Taranovs
Profesors V.Zagurskis
Students
3.kurss 1.grupa

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];
```

Rādītāji

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

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

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=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.

ATmega128 atmiņa

128 KB sistēma paprogrammējamas Flash atmiņas

Ierakstu/dzesēšanas ciklu skaits: 10,000

Energoatkarīga

Ir organizēta ka 64K x 16 (tā kā AVR instrukcijas ir 16/32 bitu garas)

Ir sadalīta divas sadaļas: Ielādēs(Boot) un programmas(Application)

Var būt programmēta SPI, JTAG vai Parallel programming režīmā.

Boot Loader atbalsts sniedz Read-While-Write pašprogrammēšanas mehānismu, kas savukārt ļauj pašam MCU lejuplādēt un augšupielādēt programmas kodu

Ir savs reģistrs: Programatmiņas Kontroles un Statusa Reģistrs SPMCSR , RAMPZ (RAM Page Z Select Register)

4KB EEPROM

Ierakstu/dzesēšanas ciklu skaits: 100,000

Energoneatkarīga

Var būt programmēta SPI, JTAG vai Parallel programming režīmā. NOTE: nevar būt programmēta, kad CPU rakstā FLASH atmiņā

Kād no tās nolasa informāciju CPU tiek apturēts uz 4 taktīm, bet, kad notiek ierakstīšanā uz 2 taktīm, pirms nākama instrukcija bus izpildīta

Ir savs reģistrs: EEARH un EEARL - adrešu, EEDR - datu, EECR - vadības,

4KB iekšējas SRAM

Gaidīšanas režīmā apstājas CPU vienlaikus ļaujot SRAM, pārtraukumu sistēmai, TCNT, SPI portam turpināt darbību.

Steks atrodas datu SRAM un tā izmērs ir cieši saistīts ar SRAM izmēru un tā izmantošanu.

Atmega neatbalsta vairāk par 64KB SRAM

Pointeri

Lai saktu izmantot pointerus man vajadzēja no sakuma atcerieties kā tos vispār izmantot. Kods, ko es uztaisīju uz datora un veiksmīgi nokompilēju ir šāds:

```
#define SIZE 10
typedef struct ptr_object{
	int* head;
	int* value;
	int size;
} ptr_object;
int main(void){

	int i = 0;

	ptr_object array1;
	array1.size = SIZE;
/*
mikrokontrollerī šo daļu var izlaist, jo tur tas neko nedos
	array1.head = (int*) malloc (array1.size * sizeof(*array1.head));
	if(!array1.head) return 0; //parbaude vai vieta ir izbriveta
*/

	array1.value = array1.head;

	for(i = 0; i < array1.size; *(array1.value++) = i++);
	for(i = 0;; (i < array1.size) ? (i++, (int)array1.value++) : (i = 1,
(int)(array1.value = array1.head) )){
		printf("%d\n", *array1.value);
	}
/* tas pats
	free(array1.head);
	return 0;
*/
}
```

Tajā ir divi pointeri head un value, kur head ir sakuma adrese un value sakuma adrese + novirze, kur novirze ir mazāka par SIZE. Pointeri ir efektīvāki par masīviem, tāpēc ka zemākajā līmeņi parēja no elementa uz elementu masīvā notiek izmantojot sākuma adresi un nobīdi, jeb *(masīva_adrese+nobīde)

Vēl viens variants, kas ir mazliet vieglāks ir izmantot jau gatavu masīvu un pointeri, kuram nodosim masīva sakuma adresi.

```
typedef struct array_object{
	int head[SIZE];
	int* value;
	int size;
} array_object;
....
	array_object array;
	array.value = array.head;
	array.size = SIZE;
```

```
	for(i = 0; i < array.size; *(array.value++) = i++);

	array.value = array.head;

	for(i = 0;; (i < array.size) ? (i++, (int)array.value++) : (i = 1, (int)
(array.value = array.head) )){
		printf("%d\n", *array.value);
	}
```
.....

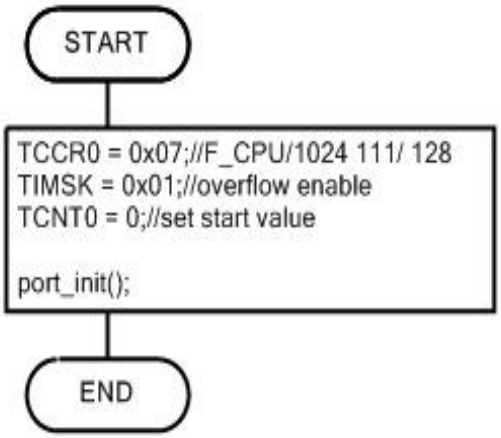
Praktiski nekas nemainās, tikai tas, ka dinamiski mēs nevarēsim mainīt masīva izmēru un mums nevajadzēs uztraukties par atmiņas izdalīšanu/atbrīvošanu, ka arī citu iespējamo kļūdu āvotus.

Programmas kods

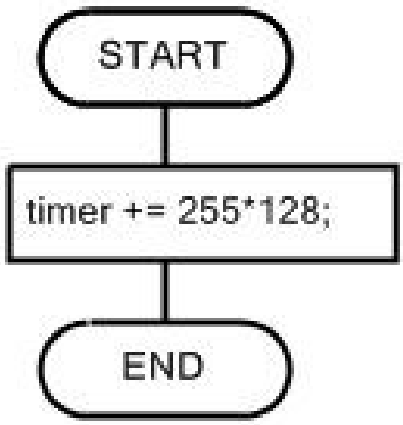
```
/*********************************************************************/
#define F_CPU 14745600UL     //cycles per second; repeats F_CPU times per
second

/********* Standarta C un specialo AVR bibliot?ku iek?au?ana***********/
#include <avr/io.h>
#include <avr/interrupt.h>

/******************** Portu inicializ?cijas funkcija *****************/
void port_init(void){
      DDRD  = 0xFF; //visas porta D l?nijas uz IZvadi
      PORTD  = ~(0x00); //porta D izejas l?niju l?me?i uz 0

      sei();
}

/******************** Rezimu[sakuma vertiibu] uzstadisana *************/
void init_devices(void){
      //timer0 setup
      TCCR0 = 0x07;     //F_CPU/1024 111/ 128
      TIMSK = 0x01;     //overflow enable
      TCNT0 = 0;        //set start value

      port_init();
}

/*****FUNCTION PROTOTYPES*********************************************/
unsigned char _rotl(const unsigned char value, unsigned char shift);

/******************** Globalie mainigie*******************************/
volatile unsigned long timer = 0;//no copy's in reg,
                        //no code opt - as compilator doesnt know-interrupt
can change it

/******************** Partraukumi ************************************/
ISR(TIMER0_OVF_vect) {
      timer += 255*128;
}

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

      //timer variables
      unsigned char port_data = 0x80, //LED pin
                              delay_value[8],   //delay array
                              *delay_head,      //delay array head
                              i,j;  //temp variables

      //save array head
      delay_head = delay_value;

      //fill delay array
      for((i=1,j=9); i < j; *(delay_head++) = i++);

      //go to array head
      delay_head = delay_value;

      while (1){
            if(timer > F_CPU*(*delay_head)){
                  port_data = _rotl(port_data,1);
```

```
                PORTD = ~port_data;
                if(*delay_value) delay_value++;
                else delay_value = delay_head;
                timer = 0;
        }

        return 1;
}

/*****************************************************************/
/*** CIRCULAR SHIFT ****/
unsigned char _rotl(const unsigned char value, unsigned char shift) {
    if ((shift &= sizeof(value) * 8 - 1) == 0) return value;
    return (value << shift) | (value >> (sizeof(value)*8 - shift));
    //return (value >> shift) | (value << (sizeof(value)*8 - shift)); //right
}
```

Blokshēma

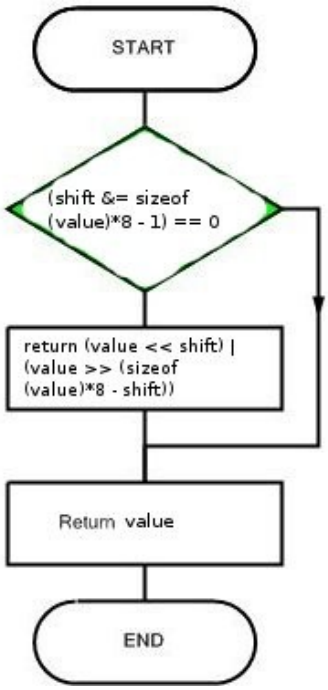


Attēls 1: init_devices()

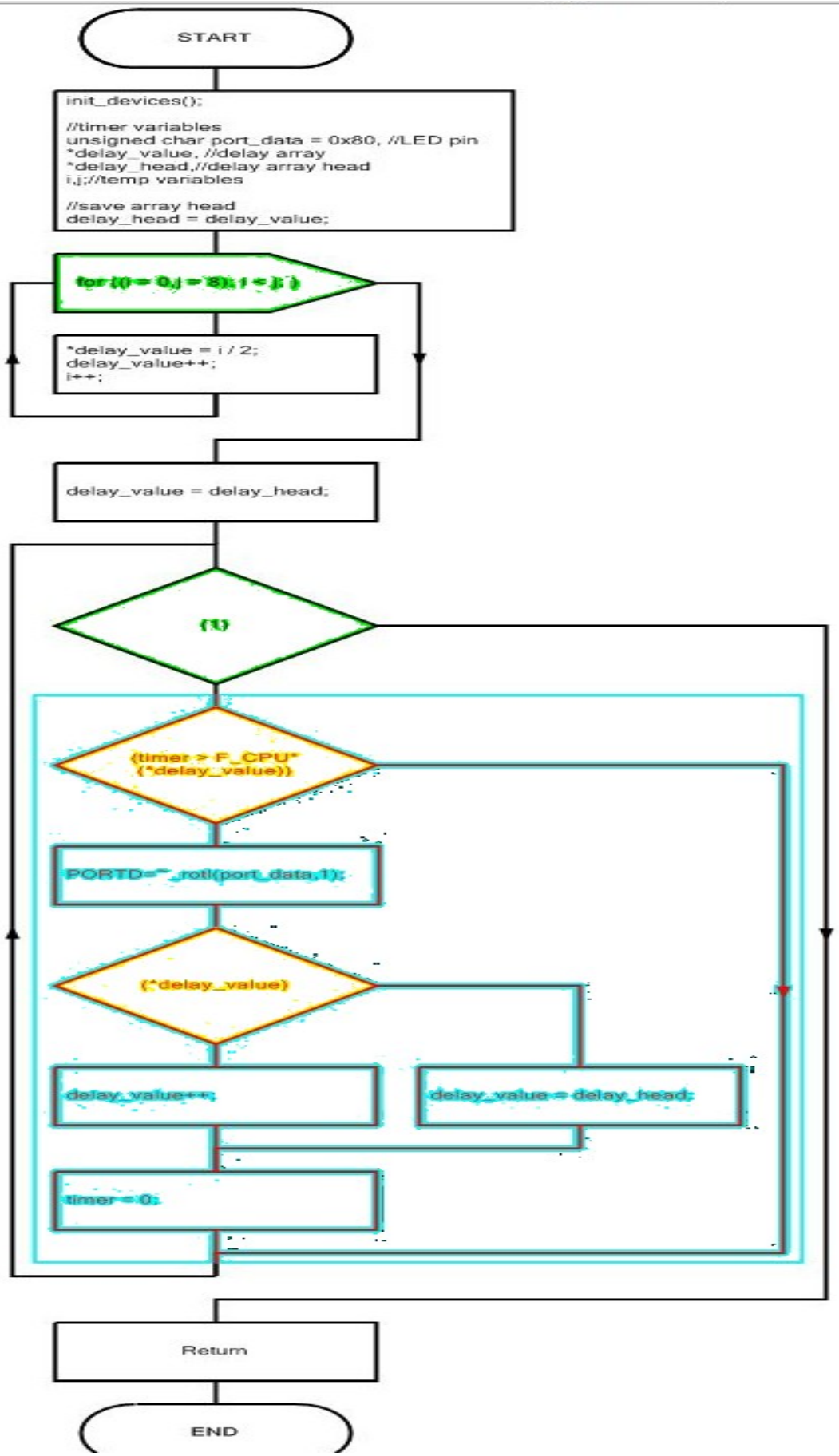


Attēls 2: ISR(TIMER_OVF)

Attēls 3: port_init()



Attēls 4: _rotl(value, shift)



Attēls 5: main(void)

Secinājumi

Lai veiksmīgi izpildītu šo darbu vajadzēja atkārtot pointeru lietojumu programmēšanas valoda C. Datoros, kuros ir milzīgs atmiņas daudzums salīdzinājuma ar mikrokontrolleriem, pointeru lietojums ir daudz vieglāks un caurskatāmāks. Kaut arī intuitīvi ir redzams, kur tieši var rasties kļūdas, praktiski to ir diezgan grūti izlabot. Problēma, kas man radusies bija acīmredzama, taču reāli bija grūti atrast tādu piemēru, kurš norādītu uz kļūdām un tajā paša laika nenovestu no pareiza ceļa. Es mēģināju veidot dinamisku masīvu, no 2 pointeriem, kur viens ir radītājs uz sākumu, bet otrais, domāts lai caurskatītu/ierakstītu masīva elementus, bet tas diez ko labi nesanāca, vajadzēs vēl paeksperimentēt ar to, un es paliku pie prastāka risinājuma. Rezultāta es saprātu, ka dinamiskie masīvi ir diezgan drošs kļūdu avots, ja izmanto tos nepareizi. PC datora šādas kļūdas var būt diezgan grūti pamanīt, jo resursu ir daudz un kamēr tie visi būs aizņemti programma droši vien jau beigs savu izpildi.

ATmega128 ir trīs atmiņas veidi: FLASH, EEPROM un SRAM, kur katrai ir savas funkcijas un priekšrocības. EEPROM un FLASH ir energoneatkarīgas atmiņas, bet SRAM ir lielāks piekļuves ātrums, tāpēc EEPROM un FLASHā var glabāt programmas kodu un konstantes, bet SRAMā mainīgos. FLASH atmiņa ir vislielākā apjoma, jo tās ir vislētākais un labākais risinājums datu glabāšanai. Tai ir liels nolasīšanas ātrums, daudz ierakstīšanās/dzēšanas ciklu. Atšķirība starp EEPROM un FLASH ir vienīgi tas, ka EEPROMā datus var dzēst pa baitiem, bet FLASHā pa sektoriem, kur viena sektora izmērs var variēties no 256 B līdz 16 KB. SRAM vēl viens ieguvums ir tas ka tai ir bezgalīgi daudz ierakstīšanās/dzēšanas ciklu. EEPROM ir dārgāks par FLASH, bet tas atmaksājas, jo FLASH atmiņai ir daudz mazāk ierakstīšanās/dzēšanas ciklu neieskaitot iepriekš minēto.