

Osnove mikroprocesorske elektronike

Vaja 6: Prekinitve

Naloge:

1. Program za serijsko komunikacijo dopolnite s prekinitvami. Ko USART modul sprejme znak, naj sproži prekinitvev, ki nato ta znak shrani v krožni medpomnilnik dolžine 100 byte-ov.
2. Napišite podprogram, ki vrne število byte-ov v medpomnilniku.
3. Napišite podprogram, ki vrne en znak iz medpomnilnika.
4. Ugotovite, katere spremenljivke beremo ali pišemo v prekinitvah in v glavnem programu in jih na kritičnih mestih zaščitite (izklopite prekinitve preden dostopate do njih in jih nato spet vklopite).
5. Podprogram za oddajo preko USART modula dopolnite z oddajnim medpomnilnikom dolžine 100 byte-ov in prekinitvenim podprogramom za oddajo posameznih znakov.
6. Tipki T1 in T2 sta priključeni na vhoda, ki lahko prožita zunanje prekinitve. Napišite prekinitveni podprogram za zunanjo prekinitvev 0, ki negira cel port B. Napišite prekinitveno past, ki vsakih 200 ms negira port B. Prekinitvena past je podprogram za primer, ko prekinitveni podprogram ni nastavljen – ISR vektor "BADISR_vect". Prekinitvenega programa za zunanjo prekinitvev 1 *ne* napišite. Vhoda za prekinitvev 0 in 1 nastavite tako, da se prekinitvev sproži, ko spustite tipko. Obe zunanji prekinitvi omogočite.

Navodila:

Osnova za vajo je rešena vaja 5.

Za delo s prekinitvami je treba vključiti datoteko `<avr/interrupt.h>`. Za dodatne pomožne funkcije boste potrebovali še `<util/atomic.h>`. Podrobnejše informacije najdete v navodilih za AVR-libc.

1. Če prejmemo več znakov zapored in prvega ne preberemo pravočasno, ga utegne naslednji povoziti. Zato je priporočljivo, da za branje USARTa uporabimo prekinitvev, ki vsak prispeli znak takoj prebere in shrani v večji medpomnilnik, ki ga lahko kasneje v miru preberemo.

- Prekinitve:

Če so prekinitve vključene, se pri izpolnjenem pogoju zgodi prekinitvev – procesor skoči v zato pripravljen podprogram, ISR - Interrupt Service Routine. Prekinitve nimajo tipa, in ne sprejmejo parametrov. Namesto tega se vedno začnejo s ključno besedo ISR, v oklepaju pa sledi ime prekinitvenega vektorja, kateremu je namenjena. Npr.: `ISR(TIMER0_COMPA_vect) {...`
Imena prekinitvenih vektorjev so napisana v navodilih za `<avr/interrupt.h>`.
Ne pozabite takoj po inicializaciji vključiti prekinitvev z ukazom `sei()`.

- Krožni medpomnilnik:

Če rabimo tak medpomnilnik, da podatke iz njega beremo v enakem vrstnem redu kot smo jih pisali (FIFO), se najbolje obnese krožni medpomnilnik. Realiziramo ga kot polje znakov (npr. `char RxBuf[10]`), in nekaj pomožnih spremenljivk. Spodnja razlaga je zelo lepo ponazorjena v videu: <https://www.youtube.com/watch?v=g9su-lnW2Ks>

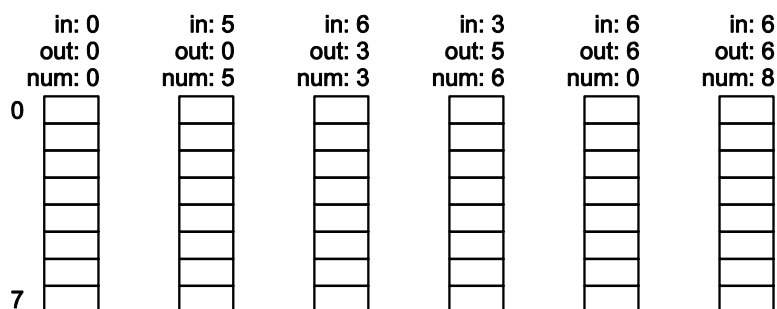
Ko pišemo v medpomnilnik, moramo vedeti katero je prvo prosto mesto. Ker lahko medpomnilnik vsebuje poljubno vrednost (tudi `0x00` je veljavna vrednost), ne moremo ločiti katera mesta so že zapolnjena in katera ne. Zato potrebujemo dodatno

spremenljivko, v katero si zapišemo, katero je prvo prosto mesto.

Do enake dileme pridemo pri branju medpomnilnika, zato potrebujemo dodatno spremenljivko, v katero si zapišemo katero je prvo zapolnjeno mesto v medpomnilniku. Praktično si je omisliti še tretjo spremenljivko, v katero si pišemo trenutno število znakov v medpomnilniku.

Do takrat, ko pri pisanju v medpomnilnik pridemo do konca medpomnilnika, smo zelo verjetno prva mesta medpomnilnika že izpraznili, zato lahko nadaljujemo na začetku. Ker tako zakrožimo iz konca na začetek, ga imenujemo krožni medpomnilnik.

Na spodnji sliki je nekaj primerov krožnega medpomnilnika z definiranimi vrednostmi pomožnih spremenljivk. Za vsak primer narišite na katera mesta kažeta indeksa »in« in »out«, ugotovite katera mesta v medpomnilniku so zasedena in jih pobarvajte.



Na koncu navodil je tudi psevdo-koda za podprograme za shranjevanje podatkov v krožni medpomnilnik in branje podatkov iz njega.

- Število byte-ov v medpomnilniku je zapisano v eni spremenljivki. Ta podprogram mora samo vrniti vrednost te spremenljivke. Čeprav se trenutno zdi pisanje tega podprograma nesmiselno, ker je tako kratek, se v praksi izkaže za koristno.
- Vsakič, ko preberemo znak iz pomnilnika moramo popraviti tudi pomožne spremenljivke, zato je praktično, če vse te operacije združimo v nov podprogram.
- Če do iste spremenljivke dostopata tako prekinitev, kot glavni program (ali podprogrami), se lahko zgodi, da prekinitev spremeni vrednost spremenljivke ravno takrat, ko jo glavni program bere. V takem primeru lahko preberemo popolnoma napačno vrednost. Da se to ne zgodi, moramo v glavnem programu dostope do vseh takšnih spremenljivk zaščititi, tako da za čas dostopa izklopimo prekinitve. To najlažje naredimo z ukazoma cli(); in sei();, vendar se lahko zgodi, da nam optimizator spremeni vrstni red izvajanja določenih ukazov in se zato prekinitve izklopijo in nazaj vklopijo na napačnih mestih v programu (glej http://www.nongnu.org/avr-libc/user-manual/optimization.html#optim_code_reorder). Namesto cli(); in sei(); je bolje uporabiti makroje ATOMIC_BLOCK(), ki bolje ščitijo pred neželenimi posledicami optimizacije. Ta makro kot parameter sprejme ATOMIC_FORCEON, ki na koncu bloka vključi prekinitve ali ATOMIC_RESTORESTATE, ki na koncu bloka vključi prekinitve samo v primeru, če so bile vključene pred začetkom bloka.
- Na enak način kot za sprejem znakov napravite še medpomnilnik za oddajo znakov. Poglavitna razlika je v tem, kdaj je prekinitev sploh vključena. Prekinitev za sprejem je vključena ves čas, prekinitev za oddajo pa samo kadar je kaj za poslati:
 - Vsakič, ko zapišemo nov znak v oddajni medpomnilnik, moramo tudi vključiti prekinitev.
 - Prekinitev mora vsakič, preden prebere znak iz medpomnilnika, preveriti ali je kak znak v medpomnilniku in če ga ni, izključiti prekinitev in končati.

Psevdo-koda podprogramov za delo s krožnim medpomnilnikom

```
char RxBuf[100];
int in;
int out;
int num;

int Koliko_podatkov_je_v_medpomnilniku()
{
    Vrni vrednost spremenljivke num
}

void Shrani_podatek(char podatek)
{
    Preveri ali je dovolj prostora v medpomnilniku, če ga ni, končaj
    Zapiši podatek v medpomnilnik
    Povečaj števec podatkov v medpomnilniku (num) za ena
    Povečaj naslov prvega prostega mesta (in) za ena
    Preveri, če naslov prvega prostega mesta (in) še ni prevelik. Če je, ga popravi, da bo kazal na
    začetek medpomnilnika (RxBuf)
}

char Preberi_podatek()
{
    Preveri ali je v medpomnilniku kakšen podatek, če ni, vrni 0
    Preberi podatek iz medpomnilnika in ga shrani v začasno spremenljivko.
    Zmanjšaj števec podatkov v medpomnilniku (num) za ena
    Povečaj naslov prvega zasedenega mesta (out) za ena
    Preveri, če naslov prvega zasedenega mesta (out) še ni prevelik. Če je, ga popravi, da bo kazal
    na začetek medpomnilnika (RxBuf)
}
```