RTOS2

Lastnosti izpopolnjenega večopravilnega operacijskega sistema v realnem času so naslednje:

- opravila se razvrščajo na podlagi prioritet,
- časovni interval, v katerem je opravilo enkrat na vrsti, se določa za vsako opravilo posebej,
- izvajanje opravila ni časovno omejeno, čeprav morajo biti opravila končno dolga,
- prekinitve delujejo neovirano, razen prekinitve, ki jo uporablja operacijski sistem (timer0), in
- dinamično dodajanje in odstranjevanje opravil in podatkovnih cevi, ter dinamično dodeljevanje pomnilnika.

Z operacijskim sistemom upravljamo s klici njegovih funkcij. Za njihovo uporabo je potrebno vključiti datoteko rtos2.h, kjer se nahajajo deklaracije.

Operacijski sistem inicializiramo s klicem funkcije *rtos2init()*. Deklaracija funkcije je naslednja:

```
void rtos2init(char *memory, unsigned int size, unsigned int slice);
```

-*memory*...kazalec na začetek sistemskega dinamičnega pomnilnika,
-*size* ...velikost sistemskega dinamičnega pomnilnika v bajtih in
-*slice* ...dolžina časovne rezine v mikrosekundah.

Sistemski dinamični pomnilnik je kos RAM-a, ki ga uporablja operacijski sistem. Njegova velikost je odvisna od števila ustvarjenih opravil, podatkovnih cevi in njihove velikosti. Definiramo ga kot globalno polje.

Po izvršitvi funkcije *rtos2init()* operacijski sistem še ne teče. Dvignemo ga s klicem funkcije *enable_os()*. Podobno je mogoče operacijski sistem tudi ustaviti s klicem funkcije *disable_os()*. Medtem ko je operacijski sistem ustavljen, ga lahko tudi ponovno inicializiramo.

```
void enable_os();
void disable_os();
```

Med inicializacijo se nastavi prekinitev časovnika timer0 in jo operacijski sistem potrebuje za svoje delovanje. Zato po inicializaciji ne smemo več klicati funkcije *vic_init()*. Oziroma ostale prekinitve je potrebno nastaviti pred klicem funkcije *rtos2init()*. Prekinitev timer0 je vrste IRQ z najvišjo prioriteto. To pomeni, da ob klicu funkcije *vic_init()* vektorske prekinitve nič ne moremo uporabiti.

Primer uporabe:

```
char system_memory[500];                    // Globalno polje s 500 bajti sistemskega
                                            pomnilnika.
  ...
vic_init(...);    // Nastavitev prekinitev. Razen vektorske IRQ z najvišjo prioriteto in
prekinitve timer0.
rtos2init(system_memory, 500, 10000);       // Inicializacija operacijskega sistema s
                                            časovno rezino 10ms.
enable_os();                                // Dvig operacijskega sistema.
  ...
disable_os();                               // Ustavitev operacijskega sistema.
```

Ko je operacijski sistem inicializiran, ustvarimo najprej podatkovne cevi za pretok podatkov med posameznimi opravili in nato sama opravila. Podatkovno cev ustvarimo s klicem funkcije *rtos2pipe_create()*, ki vrne kazalec na cev. Deklaracija funkcije je naslednja:

```
struct rtos2pipe *rtos2pipe_create(unsigned int size);
```

-*size*...velikost medpomnilnika podatkovne cevi v bajtih.

V podatkovno cev pišemo s funkcijo *rtos2pipe_write()*. Iz nje beremo z *rtos2pipe_read()*. Obe funkciji vračata število uspešno zapisanih, oziroma prebranih bajtov. Podatek se v cev ne more zapisati v primeru, ko v njej ni več prostora. Oziroma ga iz nje ne moremo prebrati, če je cev prazna.

```
unsigned int rtos2pipe_write(struct rtos2pipe *pipe, char *data, unsigned int size);
```

-*pipe*...kazalec na cev v katero želimo pisati,
-*data*...kazalec na začetek podatkov, ki jih želimo zapisati, in
-*size* ...število bajtov v *data*, ki jih želimo zapisati.

```
unsigned int rtos2pipe_read(struct rtos2pipe *pipe, char *buffer, unsigned int size);
```

-*pipe* ...kazalec na cev iz katere želimo brati,
-*buffer*...kazalec na medpomnilnik, kamor se bodo prebrani podatki shranili, in
-*size* ...število bajtov, ki jih želimo prebrati v *buffer*.

Podatkovno cev, ki je ne potrebujemo več, uničimo s funkcijo *rtos2pipe_delete()*.

```
void rtos2pipe_delete(struct rtos2pipe *pipe);
```

-*pipe*...kazalec na cev, ki jo želimo uničiti.

Primer uporabe:

```
char buf[10];
unsigned int n;
struct rtos2pipe *transfer;
  ...
transfer = rtos2pipe_create(100);        // Ustvarimo cev z medpomnilnikom velikosti 100 bajtov.
  ...
n = rtos2pipe_write(transfer, buf, 10);  // Poskus zapisa 10-ih bajtov iz buf.
  ...
n = rtos2pipe_read(transfer, buf, 10);   // Poskus branja 10-ih bajtov v buf.
  ...
rtos2pipe_delete(transfer);              // Uničimo cev.
```

Cevi lahko uporabljamo v glavnem programu in v opravilih, ne moremo pa jih uporabljati v prekinitvah IRQ, oziroma FIQ. Da cevi lahko uporabljamo tudi v prekinitvah, je potrebno v funkcijah *rtos2pipe_...()* (glej datoteko rtos2pipe.c) začasno onemogočiti prekinitve. To naredimo s postavitvijo (ob vstopu v funkcijo) in umikom (ob izstopu iz funkcije) zastavice I, oziroma F.

Opravila so funkcije tipa *rtos2taskptr*, oziroma funkcije, ki ničesar ne vračajo, sprejmejo pa kazalec na poljuben tip. Opravilo ustvarimo s klicem funkcije *rtos2task_create()*, uničimo pa ga s funkcijo *rtos2task_delete()*.

```
typedef void (* rtos2taskptr)(void *);
void rtos2task_create(rtos2taskptr function, void *arg, unsigned int priority, unsigned int interval);
```

-*function*...funkcija opravila,
-*arg* ...argument s katerim je opravilo klicano,
-*priority* ...prioriteta opravila (nižja vrednost pomeni višjo prioriteto) in
-*interval* ...število časovnih rezin, ki sestavljajo interval v katerem je opravilo enkrat na vrsti.

Zadnji argument *interval* podaja pogostnost klicanja opravila. In sicer bo opravilo na vrsti enkrat na časovni interval *t = interval * dolžina časovne rezine*. Dolžino časovne rezine smo določili ob inicializaciji operacijskega sistema z argumentom *slice* funkcije *rtos2init()*.

```
void rtos2task_delete(rtos2taskptr function);
```

-*function*...funkcija opravila, ki ga želimo odstraniti.

Za vsako opravilo smo določili, na koliko časovnih rezin naj bo na vrsti. Če sta hkrati na vrsti dve ali več opravil, potem se požene tisto z najvišjo prioriteto. Če trenutno teče opravilo z nižjo prioriteto, na vrsti pa je opravilo z višjo prioriteto, potem opravilo z višjo prioriteto prekine opravilo z nižjo prioriteto. Po končanju opravila z višjo prioriteto se nadaljuje izvajanje opravila z nižjo prioriteto. Tako opravila z nižjo prioriteto čakajo na izvajanje takrat, ko ni zahtev za zagon opravil z višjo prioriteto. Zato se lahko zgodi, da opravilo z nižjo

prioriteto ne bo na vrsti točno ob poteku časovnega intervala, ampak nekaj časovnih rezin kasneje. Vendar se zaradi tega trenutek naslednje razvrstitve opravila ne zamakne. Frekvenca klicanja ostaja konstantna enkrat na časovni interval.

Primer uporabe:

```
char arg[10];
void task(void *data);
  ...
rtos2task_create(task, arg, 5, 100);// Opravilo task(arg) bo na vrsti v vsaki 100-ti rezini.
  ...
rtos2task_delete(task);                  // Uničimo opravilo.
```

Za dinamične potrebe po pomnilniku si lahko definiramo kos pomnilnika RAM. To naredimo s funkcijo *rtos2mem_create()*. Definirati je mogoče več med seboj neodvisnih kosov pomnilnika.

```
void rtos2mem_create(struct rtos2mem *region, char *memory, unsigned int size);
```

-*region* ...kazalec na kos pomnilnika,
-*memory*...kazalec na začetek dinamičnega pomnilnika (pomnilnik, namenjen dinamičnemu dodeljevanju, definiramo kot globalno polje) in
-*size* ...velikost dinamičnega pomnilnika v bajtih.

Znotraj izbranega kosa pomnilnik dinamično dodeljujemo in sproščamo s funkcijama *rtos2mem_allocate()* in *rtos2mem_free()*. Funkcija *rtos2mem_allocate()* vrne kazalec na dodeljen pomnilnik. V primeru, da dodelitev ne uspe, vrne vrednost 0x00000000.

```
void *rtos2mem_allocate(struct rtos2mem *region, unsigned int size);
```

-*region*...kazalec na kos pomnilnika, ker želimo dodelitev, in
-*size* ...število bajtov, ki jih želimo dobiti.

```
void rtos2mem_free(struct rtos2mem *region, void *pointer);
```

-*region* ...kazalec na kos pomnilnika, kjer želimo sprostiti dodeljen pomnilnik, in
-*pointer*...kazalec na dodeljen pomnilnik.

Do posameznega kosa pomnilnika naj načeloma dostopa le eno opravilo (glavni program, prekinitev IRQ ali FIQ). V nasprotnem primeru trčimo ob težavo hkratnega dostopa. Zato moramo biti v takšnem primeru še posebej pazljivi. Potrebno je zagotoviti, da do določenega kosa pomnilnika dostopa le eno opravilo (glavni program, prekinitev) naenkrat. Opravilo (glavni program, prekinitev) med dostopom do kosa pomnilnika (dodeljevanje, uporaba ali sproščanje pomnilnika) ne sme biti prekinjeno z drugim opravilom (prekinitvijo), ki dostopa do istega kosa pomnilnika. Težavo lahko rešimo z zaklepanjem, kar pomeni, da moramo kritične dele kode napisati na poseben način. Lahko pa si pomagamo tudi s funkcijama *disable_os()* in *enable_os()* s katerima med kritičnimi deli kode začasno izklapljamo operacijski sistem (preprečimo prekinitev s strani drugega opravila), in s postavljanjem

zastavic I in F s katerima začasno onemogočimo prekinitve IRQ in FIQ (preprečimo prekinitev zaradi nove zahteve po prekinitvi).

Operacijski sistem ima za svoje potrebe določen svoj kos pomnilnika.

Primer uporabe:

```
void *ptr;
char chunk[1000];
struct rtos2mem piece;
  ...
rtos2mem_create(&piece, chunk, 1000);      // Ustvarimo kos piece velik 1000 bajtov.
  ...
ptr = rtos2mem_allocate(&piece, 100);      // V kosu piece rezerviramo 100 bajtov, na
                                           katere kaže ptr.
  ...
rtos2mem_free(&piece, ptr);                // V kosu piece sprostimo pomnilnik, na
                                           katerega kaže ptr.
```