Univerza v Ljubljani
Fakulteta za elektrotehniko

Uvod v mikrokrmilniške sisteme

Zgradba in programiranje

Janez Puhan
Tadej Tuma

Ljubljana, 2011

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.2/.3(075.8)

PUHAN, Janez, 1969-
Uvod v mikrokrmilniške sisteme : zgradba in programiranje / Janez Puhan, Tadej Tuma. - 2. dopolnjena izd. - Ljubljana : Založba FE in FRI, 2011

ISBN 978-961-243-166-2 (Fakulteta za elektrotehniko)
1. Tuma, Tadej
254260992

Založnik: Založba FE in FRI, Ljubljana
Izdajatelj: UL Fakulteta za elektrotehniko, Ljubljana
Urednik: mag. Peter Šega

Natisnil: UL Fakulteta za elektrotehniko
Naklada: 30 izvodov
2. dopolnjena izdaja

Kazalo

Poglavje 1

Uvod

V okviru univerzitetnega programa na Fakulteti za elektrotehniko v Ljubljani študentje smeri Elektronika v osmem in devetem semestru poslušajo predmeta Računalniško načrtovanje vezij in Mikroprocesorji v elektroniki. Prvi predmet obravnava gradnjo, drugi pa programiranje kompaktnih industrijskih računalniških sistemov. Čeprav je namen obeh predmetov v prvi vrsti podajati osnovna načela industrijskih mikrokrmilniških sistemov ne glede na proizvajalca, temeljijo vsi konkretni zgledi na Philipsovem mikrokrmilniku LPC2138 z osnovo centralnega procesnega jedra ARM7TDMI-S [3], ki poleg mikroprocesorskega jedra vsebuje še vrsto perifernih enot.

Skripta je sestavljena iz treh delov. V prvem delu so podane osnove digitalnih vezij. Razloženi so osnovni elementi digitalnega sveta, ki se pojavljajo v mikrokrmilniških sistemih. Z razumevanjem delovanja osnovnih digitalnih sklopov dobi bralec vpogled tudi v strojno zgradbo mikrokrmilnika, oziroma mikrokrmilniškega sistema.

Drugi del skripte obravnava mikrokrmilnike s programskega stališča. Podana so osnovna načela hkratnega in sprotnega programiranja. Mikrokrmilnik nastopa kot zaprta naprava, ki le izvaja napisano kodo, program. Njegova dejanska strojna zgradba nas na tem nivoju ne zanima.

Hkratno in sprotno programiranje je pogojeno z uporabo operacijskega sistema, ki teče v realnem času. Snov je podana na primeru preprostega operacijskega sistema v realnem času. Razloži nam osnovne principe takšnega pristopa, ter temu prirejen način programiranja. K razlagi je priložena tudi izvorna koda v programskem jeziku C in v zbirniškem jeziku. S poznavanjem zbirnika dobi

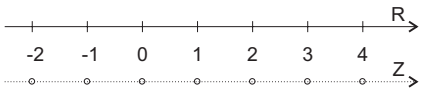
bralec vpogled, kaj se v resnici dogaja za vrsticami programske kode napisane v enem izmed višjih programskih jezikov.

Sestavni del obeh predmetov so tudi laboratorijske vaje, katerih cilj je omogočiti praktično delo z industrijskimi mikrokrmilniškimi sistemi. Delo v laboratoriju poteka na učnih razvojnih sistemih Š-arm [12], ki jih študentje v grobem spoznajo že v prvem letniku. Namen laboratorijskih vaj je poglobljeno obravnavanje delovanja mikrokrmilniškega sistema in vgrajenih perifernih enot, ter spoznavanje osnovnih načel hkratnega in sprotnega programiranja (programske opreme, ki teče v realnem času). Ob spoznavanju perifernih enot dobijo študentje vpogled v različne komunikacijske protokole, kot tudi v zgradbo in delovanje mikrokrmilnika. Snov, ki zajema ta del, je močno vezana na izbran mikrokrmilnik in je popisana v tretjem delu skripte, v dodatkih.

Poglavje 2

Zgradba mikrokrmilniškega sistema

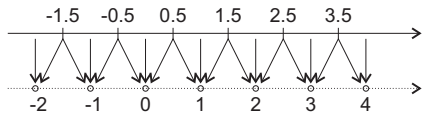
V digitalnih vezjih se navadno ne pogovarjamo o vhodnih in izhodnih napetostih, tokovih ..., kot je to navada v analognem svetu. Govorimo le še o vhodnih in izhodnih stanjih, ki jih je vedno končno mnogo. Z drugimi besedami, digitalni svet je diskretiziran. Poglejmo si razliko med zveznim in diskretnim na enostavnem primeru realnih in celih števil (slika 2.1).



Slika 2.1: Realna in cela števila

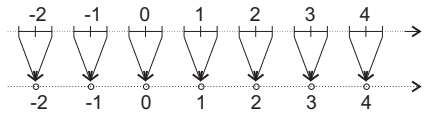
Številska premica realnih števil je neprekinjena. Realno število lahko zavzame poljubno vrednost. Enako velja za različne fizikalne veličine v našem makro svetu, na primer električno napetost. Pravimo, da so fizikalne veličine v naravi zvezne. Na drugi strani imamo cela števila, ki jih niti ne moremo več ponazoriti s premico. Od množice realnih vrednosti jih je ostalo le "nekaj", ki jih lahko celo število zavzame. Ostale vrednosti so sedaj prepovedane. Oziroma cela števila imajo lahko le diskretne vrednosti. Ker so fizikalne veličine v naravi

zvezne, se dogovorimo za diskretizacijo. Primer (slika 2.2): električna napetost med -0.5V in 0.5V pomeni 0, med 0.5V in 1.5V pomeni 1 ...



Slika 2.2: Diskretizacija zvezne fizikalne veličine

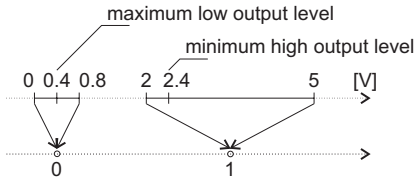
Zvezni fizikalni veličini smo z dogovorom pripisali le diskretne vrednosti. Naša električna napetost ima v resnici lahko poljubno realno vrednost, ki pa jo interpretiramo kot diskretno vrednost.



Slika 2.3: Diskretizacija s prepovedanimi področji

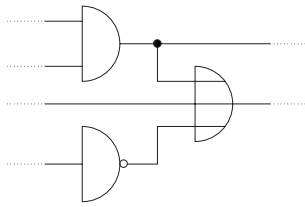
V primeru na sliki 2.2 imamo težave, ko ima napetost točno mejno vrednost, na primer 0.5V. Ali takrat to pomeni vrednost nič, ali ena. Matematično bi sicer lahko določili eksaktno preslikavo tudi na mejah, vendar bi bilo to v praksi zaradi končne natančnosti težko realizirati. Zato uvedemo prepovedana področja. Dogovorimo se na primer, da napetost med -0.3V in 0.3V pomeni 0, med 0.7V in 1.3V pomeni 1 ... Ostale vrednosti napetosti se ne preslikajo v diskretne

vrednosti (slika 2.3). Pravimo, da so prepovedane, oziroma se ne smejo nikdar pojaviti.



Slika 2.4: Napetostni nivoji TTL

Napetosti v digitalnih vezjih so v praksi omejene. Nikdar ne presežejo neke v naprej določene zgornje meje, na primer 5V, ter nikdar ne padejo pod v naprej določeno spodnjo mejo, na primer 0V. Digitalna vezja delujejo le z dvema diskretnima nivojema. Zato je potrebno napetosti med spodnjo in zgornjo mejo preslikati v le dve vrednosti ločeni s prepovedanim področjem. Slika 2.4 podaja primer takšne preslikave in predstavlja standardne nivoje TTL (angl. Transistor Transistor Logic).

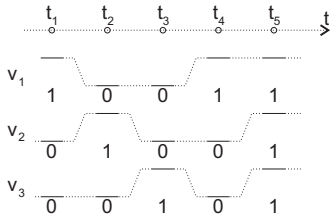


Slika 2.5: Digitalno vezje je sestavljeno iz osnovnih gradnikov

Zvezno (analogno) fizikalno veličino (napetost), smo tako diskretizirali in jo pripravili za uporabo v digitalnem svetu. Digitalna vezja so sestavljena iz osnovnih gradnikov, ki navadno opravljajo različne logične operacije (slika 2.5). Napetosti na njihovih vhodih vedno predstavljajo vrednost nič (nizko stanje) ali ena (visoko stanje). Enako velja za napetosti na izhodih. Tako se nobena izmed omenjenih napetosti ne sme nahajati v prepovedanem področju. Vsak izhod lahko predstavlja vhod naslednjega gradnika. Da napetost na izhodu zagotovo predstavlja želeno vrednost na naslednjem vhodu, sta najvišji izhodni nivo za nizko stanje in najnižji za visoko stanje za izhodne napetosti predpisana

strožje kot za vhodne (slika 2.4). Oziroma, prepovedano področje je za izhodne napetosti širše kot za vhodne.

Ker se nobena napetost nikdar ne sme nahajati v prepovedanem področju, se poraja vprašanje, kako potem sploh lahko pride do sprememb? Vemo, da so fizikalne veličine v našem makro svetu časovno zvezne. Neskončno hitrih sprememb v naravi ni, kar pomeni, da se mora napetost ob prehodu iz ene vrednosti v drugo nujno nekaj časa nahajati v prepovedanem področju. To pa ni dovoljeno. Težava izvira iz dejstva, da nismo diskretizirali časa. Čas v našem digitalnem vezju še vedno teče zvezno. Diskretiziramo ga tako, da določimo trenutke, ko je stanje v vezju regularno.



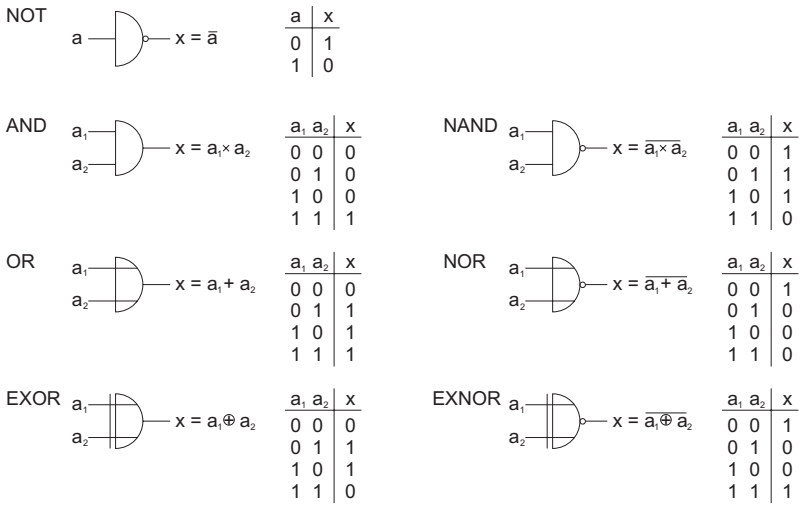
Slika 2.6: Diskretizacija časa

Na sliki 2.6 se v trenutkih t_1, t_2, ... nobena napetost ne nahaja v prepovedanem področju. Vse napetosti so znotraj dovoljenega že nekaj časa pred dogovorjenim trenutkom in še nekaj časa po njem. Spremembe se dogajajo med diskretnimi trenutki. S tem smo prišli do dveh osnovnih lastnosti digitalnih vezij, in sicer v digitalnih vezjih diskretiziramo tako napetosti, kot čas.

2.1 Osnovni gradniki digitalnih vezij, logična vrata

Najosnovnejši gradniki digitalnih vezij so logična vrata. To so digitalna vezja, ki imajo dva ali več vhodov, ter en sam izhod. Napetost na vsakem izmed vhodov predstavlja logično nizko ali visoko stanje (glej sliko 2.4). Izhodna napetost, oziroma izhodna logična vrednost je določena z logičnim izrazom, v katerem nastopajo vhodne vrednosti. Slika 2.7 prikazuje simbole, pravilnostne tabele

in logične izraze za sedem osnovnih tipov vrat. Pravilnostne tabele podajajo izhodne vrednosti za vse kombinacije vhodov.



a	x
0	1
1	0

a_1	a_2	x
0	0	0
0	1	0
1	0	0
1	1	1

a_1	a_2	x
0	0	1
0	1	1
1	0	1
1	1	0

a_1	a_2	x
0	0	0
0	1	1
1	0	1
1	1	1

a_1	a_2	x
0	0	1
0	1	0
1	0	0
1	1	0

a_1	a_2	x
0	0	0
0	1	1
1	0	1
1	1	0

a_1	a_2	x
0	0	1
0	1	0
1	0	0
1	1	1

Slika 2.7: Osnovna logična vrata

Negacija (vrata NOT) pravzaprav ne ustreza definiciji logičnih vrat, ker ima le en vhod. Izhod je vedno nasprotje vhoda, kar matematično zapišemo z izrazom (2.1).

$$x = \overline{a} \tag{2.1}$$

Prečna črta nad logično spremenljivko ali izrazom pomeni obratno vrednost.

Konjunkcija, logični in (vrata AND). Na izhodu dobimo visoko stanje le v primeru, ko vsi vhodi visoki. Operaciji konjunkcije pravimo tudi infimalna ope-

racija, kar pomeni, da izhod sledi najmanjši vhodni vrednosti. Vrata AND imajo lahko dva ali več vhodov, kar je nakazano v enačbi (2.2).

$$x = a_1 \times a_2 \times \ldots \tag{2.2}$$

Logično operacijo konjunkcije simbolično zapišemo kot množenje.

Disjunkcija, logični ali (vrata OR). Na izhodu dobimo visoko stanje v primeru, ko je vsaj eden izmed vhodov visok. Operaciji disjunkcije pravimo tudi supremalna operacija, kar pomeni, da izhod sledi največji vhodni vrednosti. Vrata OR imajo lahko dva ali več vhodov, kar je nakazano v enačbi (2.3).

$$x = a_1 + a_2 + \ldots \tag{2.3}$$

Logično operacijo disjunkcije simbolično zapišemo kot seštevanje.

Negirana konjunkcija, negirani logični in (vrata NAND). Kombinira operaciji konjunkcije in negacije. Na izhodu dobimo nizko stanje le v primeru, ko so vsi vhodi visoki. Nad vhodnimi vrednostmi se najprej izvrši operacija konjunkcije, ter nato negacija, kar opisuje enačba (2.4). Vrata NAND imajo lahko, tako kot vrata AND, dva ali več vhodov.

$$x = \overline{a_1 \times a_2 \times \ldots} \tag{2.4}$$

Negirana disjunkcija, negirani logični ali (vrata NOR). Kombinira operaciji disjunkcije in negacije. Na izhodu dobimo nizko stanje vedno, ko je vsaj eden izmed vhodov visok. Nad vhodnimi vrednostmi se najprej izvrši operacija disjunkcije, ter nato negacija, kar opisuje enačba (2.5). Vrata NOR imajo lahko, tako kot vrata OR, dva ali več vhodov.

$$x = \overline{a_1 + a_2 + \ldots} \tag{2.5}$$

Izključna disjunkcija, izključni logični ali (vrata EXOR). Na izhodu dobimo visoko stanje v primeru, ko je le eden izmed vhodov visok (enačba (2.6)). Z dru-

gimi besedami, visoko stanje se na izhodu pojavi takrat, ko sta vhodni vrednosti različni. To pomeni, da imajo vrata EXOR lahko le dva vhoda.

$$x = a_1 \oplus a_2 \tag{2.6}$$

Logično operacijo izključne disjunkcije simbolično zapišemo z obkroženim znakom za seštevanje. Izključna disjunkcija pravzaprav predstavlja enobitni seštevalnik. S tega stališča je mogoče vrata EXOR posplošiti na vrata s poljubnim številom vhodov. V tem primeru na izhodu dobimo visoko stanje takrat, kadar je liho število vhodov visokih.

Negirana izključna disjunkcija, negirani izključni logični ali (vrata EXNOR). Kombinira operaciji izključne disjunkcije in negacije. Na izhodu dobimo visoko stanje v primeru, ko sta oba vhoda enaka. Nad vhodnimi vrednostmi se najprej izvrši operacija izključne disjunkcije, ter nato negacija, kar opisuje enačba (2.7). Vrata EXNOR imajo, tako kot vrata EXOR, dva vhoda.

$$x = \overline{a_1 \oplus a_2} \tag{2.7}$$

2.2 Boolova algebra

Osnovno matematično orodje, ki ga največkrat uporabljamo pri načrtovanju digitalnih vezij, je Boolova algebra [1], katere temelje je postavil angleški matematik George Boole (1815-1864). Boolova algebra je definirana nad množico $\mathcal{X}$. Elementi množice $\mathcal{X}$ lahko zavzamejo vrednosti nič (nizko stanje) in ena (visoko stanje), med njimi pa sta definirani dve binarni operaciji konjunkcije in disjunkcije, ter ena unarna operacija negacije. Vse tri operacije smo že srečali med osnovnimi gradniki digitalnih vezij (vrata AND, OR in NOT). Vrstni red operacij je natančno določen, in sicer ima konjunkcija prednost pred disjunkcijo.

Boolova algebra temelji na šestih aksiomih, ki jih je postavil ameriški matematik Edward V. Huntington (1874-1952). Aksiomi Boolova algebre so:

1. Množica $\mathcal{X}$ vsebuje vsaj dva elementa $a_1, a_2 \in \mathcal{X}$, tako da velja $a_1 \neq a_2$.

2. *Zaprtost:* Za vsak $a_1, a_2 \in \mathcal{X}$ velja

$$\begin{array}{rcl} a_1 + a_2 & \in & \mathcal{X} \\ a_1 \times a_2 & \in & \mathcal{X} \end{array} \tag{2.8}$$

3. *Obstoj nevtralnih elementov:* Za vsak $a_1 \in \mathcal{X}$ velja

$$\begin{array}{rcl} \text{za } 0 \in \mathcal{X}, \quad a_1 + 0 & = & a_1 \\ \text{za } 1 \in \mathcal{X}, \quad a_1 \times 1 & = & a_1 \end{array} \tag{2.9}$$

4. *Komutativnost:* Za vsak $a_1, a_2 \in \mathcal{X}$ velja

$$\begin{array}{rcl} a_1 + a_2 & = & a_2 + a_1 \\ a_1 \times a_2 & = & a_2 \times a_1 \end{array} \tag{2.10}$$

5. *Distributivnost:* Za vsak $a_1, a_2, a_3 \in \mathcal{X}$ velja

$$\begin{array}{rcl} a_1 + a_2 \times a_3 & = & (a_1 + a_2) \times (a_1 + a_3) \\ a_1 \times (a_2 + a_3) & = & a_1 \times a_2 + a_1 \times a_3 \end{array} \tag{2.11}$$

6. *Komplementarnost:* Za vsak $a_1 \in \mathcal{X}$ obstaja $\overline{a_1} \in \mathcal{X}$ in velja

$$\begin{array}{rcl} a_1 + \overline{a_1} & = & 1 \\ a_1 \times \overline{a_1} & = & 0 \end{array} \tag{2.12}$$

Aksiomi so med seboj neodvisni in jih ne dokazujemo. Uporabljamo jih za dokazovanje različnih izrekov Boolove algebre, od katerih je najbolj znan DeMorganov izrek.

2.2.1 DeMorganov izrek

DeMorganov izrek je najpomembnejši izrek v svetu digitalnih vezij. Odkril ga je britanski matematik Augustus DeMorgan (1806-1871). Po DeMorganovem izreku zapišemo negacijo izraza z zamenjavo vseh spremenljivk z njihovimi nega-

cijami in zamenjavo disjunkcije s konjunkcijo in obratno. Negacija konjunkcije spremenljivk ali izrazov je disjunkcija negiranih spremenljivk ali izrazov.

$$\overline{a_1 \times a_2} = \overline{a_1} + \overline{a_2} \tag{2.13}$$

Negacija disjunkcije spremenljivk ali izrazov je konjunkcija negiranih spremenljivk ali izrazov.

$$\overline{a_1 + a_2} = \overline{a_1} \times \overline{a_2} \tag{2.14}$$

V splošnem velja izrek za $n = 2, 3, \ldots m$ spremenljivk.

$$\begin{array}{rcl} \overline{a_1 \times a_2 \times a_3 \times \ldots \times a_m} & = & \overline{a_1} + \overline{a_2} + \overline{a_3} + \ldots + \overline{a_m} \\ \overline{a_1 + a_2 + a_3 + \ldots + a_m} & = & \overline{a_1} \times \overline{a_2} \times \overline{a_3} \times \ldots \times \overline{a_m} \end{array} \tag{2.15}$$

Dokaz
Izrek dokažemo s pomočjo aksiomov 1 do 6. Da bi izrek dokazali, najprej pokažimo, da velja naslednji pomožni izrek (2.16).

Če velja $b_1 + b_2 = 1$ in $b_1 \times b_2 = 0$, potem $\overline{b_1} = b_2$.

$$b_1 + b_2 = 1 \ \text{ in } \ b_1 \times b_2 = 0 \ \Rightarrow \ \overline{b_1} = b_2 \tag{2.16}$$

Pomožni izrek pokažemo z uporabo aksiomov.

$$\begin{array}{ll} \overline{b_1} & \\ = \overline{b_1} \times 1 & \textit{obstoj nevtralnih elementov} \\ = \overline{b_1} \times (b_1 + b_2) & \textit{prvi pogoj pomožnega izreka} \\ = \overline{b_1} \times b_1 + \overline{b_1} \times b_2 & \textit{distributivnost} \\ = b_1 \times \overline{b_1} + \overline{b_1} \times b_2 & \textit{komutativnost} \\ = 0 + \overline{b_1} \times b_2 & \textit{komplementarnost} \\ = b_1 \times b_2 + \overline{b_1} \times b_2 & \textit{drugi pogoj pomožnega izreka} \\ = b_2 \times b_1 + b_2 \times \overline{b_1} & \textit{komutativnost} \\ = b_2 \times (b_1 + \overline{b_1}) & \textit{distributivnost} \\ = b_2 \times 1 & \textit{komplementarnost} \\ = b_2 & \textit{obstoj nevtralnih elementov} \end{array} \tag{2.17}$$

Naj bo $b_1 = a_1 \times a_2 \times \ldots \times a_m$ in $b_2 = \overline{a_1} + \overline{a_2} + \ldots + \overline{a_m}$. Če pokažemo, da velja $b_1 + b_2 = a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} + \overline{a_2} + \ldots + \overline{a_m} = 1$ in $b_1 \times b_2 =$

$a_1 \times a_2 \times \ldots \times a_m \times (\overline{a_1} + \overline{a_2} + \ldots + \overline{a_m}) = 0$, potem zaradi pomožnega izreka (2.16) DeMorganov izrek $\overline{a_1 \times a_2 \times \ldots \times a_m} = \overline{a_1} + \overline{a_2} + \ldots + \overline{a_m}$ velja.

$$
\begin{aligned}
& a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times 1 + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times (a_2 + \overline{a_2}) + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times a_2 + \overline{a_1} \times \overline{a_2} + 1 \times \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times a_2 + (\overline{a_1} + 1) \times \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times a_2 + 1 \times \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times a_2 + \overline{a_2} + \ldots + \overline{a_m} = \ldots \\
= \; & a_1 \times a_2 \times \ldots \times a_m + \overline{a_1} \times a_2 \times \ldots \times a_m + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & (a_1 + \overline{a_1}) \times a_2 \times \ldots \times a_m + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & 1 \times a_2 \times \ldots \times a_m + \overline{a_2} + \ldots + \overline{a_m} = \\
= \; & a_2 \times \ldots \times a_m + \overline{a_2} + \ldots + \overline{a_m} = \ldots \\
= \; & a_m + \overline{a_m} = 1
\end{aligned}
\tag{2.18}
$$

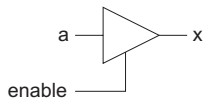
$$
\begin{aligned}
& a_1 \times a_2 \times \ldots \times a_m \times (\overline{a_1} + \overline{a_2} + \ldots + \overline{a_m}) = \\
= \; & a_1 \times a_2 \times \ldots \times a_m \times \overline{a_1} + a_1 \times a_2 \times \ldots \times a_m \times \overline{a_2} + \ldots + \\
& + a_1 \times a_2 \times \ldots \times a_{m-1} \times a_m \times \overline{a_m} = \\
= \; & a_1 \times \overline{a_1} \times a_2 \times \ldots \times a_m + a_1 \times a_2 \times \overline{a_2} \times a_3 \times \ldots \times a_m + \ldots + \\
& + a_1 \times a_2 \times \ldots \times a_{m-1} \times a_m \times \overline{a_m} = \\
= \; & 0 \times a_2 \times \ldots \times a_m + a_1 \times 0 \times a_3 \times \ldots \times a_m + \ldots + \\
& + a_1 \times a_2 \times \ldots \times a_{m-1} \times 0 = 0
\end{aligned}
\tag{2.19}
$$

Na enak način je mogoče dokazati tudi relacijo $\overline{a_1 + a_2 + \ldots + a_m} = \overline{a_1} \times \overline{a_2} \times \ldots \times \overline{a_m}$. Z dokazom DeMorganovega izreka smo pokukali v matematično zakulisje, oziroma teorijo, na kateri slonijo digitalna vezja. V nadaljevanju tega poglavja se bomo bolj kot teoriji raje posvetili praktičnim rešitvam.

2.3 Tristanjska logična vezja

Logična vrata iz poglavja 2.1, ki predstavljajo osnovne gradnike digitalnih vezij, imajo lahko le dve različni izhodni stanji, oziroma dva izhodna nivoja napetosti. To pomeni, da je posamezen izhod lahko povezan z enim ali več vhodi, nikakor pa ne z drugim izhodom. V primeru, da med seboj povezana izhoda vsilju-

jeta različni izhodni napetosti, med njima steče velik električni tok, kar lahko povzroči uničenje vrat.



enable	x
0	high-Z
1	a

Slika 2.8: Tristanjski ojačevalnik

Z uporabo vodil v mikrokrmilniških sistemih pa zahtevamo prav to. Zato poleg dveh nivojev napetosti uvedemo še tretje stanje, to je stanje visoke impedance. V ta namen se na izhodih uporabljajo tristanjski ojačevalniki (slika 2.8).

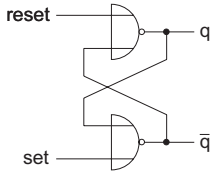
Ko je izhod omogočen ($enable = 1$) se vhod a le preslika na izhod x. V nasprotnem primeru ($enable = 0$) je izhod plavajoč. Vhod a in izhod x sta med seboj ločena (odprto stikalo). Pravimo, da je izhod x v stanju visoke impedance. Z uporabo tristanjskih ojačevalnikov je mogoče med seboj povezati več izhodov logičnih vrat. Paziti je potrebno le, da je v nekem trenutku omogočen le eden izmed njih. Vsi ostali morajo biti ob istem času onemogočeni.

2.4 Pomnilne celice

Stanje na izhodu logičnih vrat je odvisno le od trenutnega stanja na vhodih. Vsa predhodna stanja vhodov nimajo nobenega vpliva. Logična vrata sama po sebi niso sposobna pomnjenja svoje zgodovine. To omogočajo pomnilne celice (angl. flip-flop).

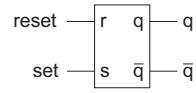
Pomnilna celica je logično vezje, ki ima poljubno število vhodov in dva izhoda. Shranjuje en bit informacije, ki se nahaja na njenem izhodu q. Na drugem izhodu $\overline{q}$ najdemo negirano različico shranjene informacije. Oba izhoda se postavita v svoji obratni vrednosti ob določenem stanju na vhodih, ter takšna

ostaneta tudi po tem, ko vhodno stanje ni več prisotno. Flip-flop ima sposobnost pomnjenja, oziroma zadržanja vsebine tudi po prenehanju vzbujanja.



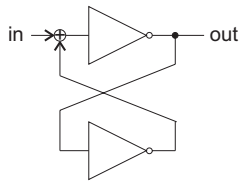
reset	set	q(t+1)	$\overline{q}$(t+1)
0	0	q(t)	$\overline{q}$(t)
0	1	1	0
1	0	0	1
1	1	0	0

≡



Slika 2.9: Pomnilna celica RS z vrati NOR

Osnovno pomnilno celico RS (angl. Reset-Set) dobimo z navzkrižno vezavo dveh vrat NOR (slika 2.9). Iz pravilnostne tabele razberemo, da visoko stanje na vhodu *reset* postavi izhod q v nizko stanje, visoko stanje na vhodu *set* pa v visoko stanje. Posebej zanimivi sta situaciji, ko sta logični vrednosti na obeh vhodih enaki. V primeru dveh nizkih stanj pomnilna celica RS zadrži prejšnje stanje. Vhoda ne vplivata na izhoda vrat NOR (aksiom 3 na strani 10). Izhod q vzdržuje stanje negiranega izhoda $\overline{q}$ in obratno. Flip-flop pomni zadnje postavljeno stanje. V primeru visokih stanj na obeh vhodih pa sta izhoda obeh vrat NOR nizka. Izhoda q in $\overline{q}$ sedaj nista več negirana, kar ni v skladu z definicijo pomnilne celice. Zato pravimo, da je to stanje prepovedano. Oziroma, če nočemo kršiti definicije, se na obeh vhodih ne smeta hkrati pojaviti dve visoki stanji.



in	out	
0	0	labile
+saturation	-saturation	stabile
-saturation	+saturation	stabile

Slika 2.10: Pozitivna povratna vezava pomnilne celice

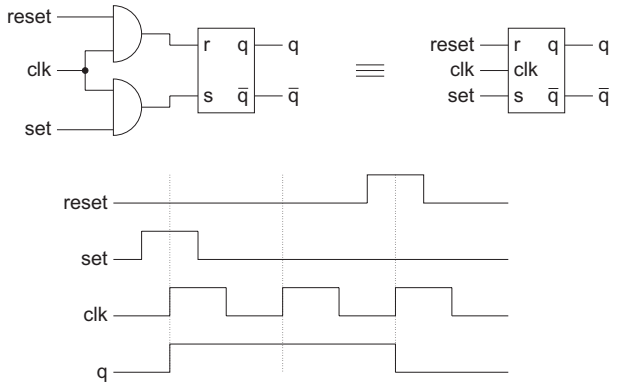
Z analognega zornega kota pomnilna celica RS v bistvu predstavlja sklop s pozitivno povratno vezavo. Če za trenutek pustimo oba vhoda ob strani, ter na vrata NOR gledamo kot na idealen invertirajoč ojačevalnik z ojačenjem večjim od ena (slika 2.10), potem povratna vezava vezje ob premiku iz labilne ničelne lege sili v eno ali drugo skrajnost. Majhna sprememba na vhodu se ojači, ter

takšna pride nazaj na vhod, se zopet ojači ... Z analognega stališča je zaradi pozitivne povratne vezave vezje nestabilno, oziroma se vedno ujame v eni ali drugi skrajni legi.

Prepovedano stanje, ko je na obeh vhodih *reset* in *set* visoko stanje, ima poleg kršitve definicije pomnilne celice še eno neprijetno lastnost. Zaradi pozitivne povratne vezave je izhodno stanje pomnilne celice po prenehanju prepovedanega stanja nepredvidljivo.

2.4.1 Sinhrone pomnilne celice

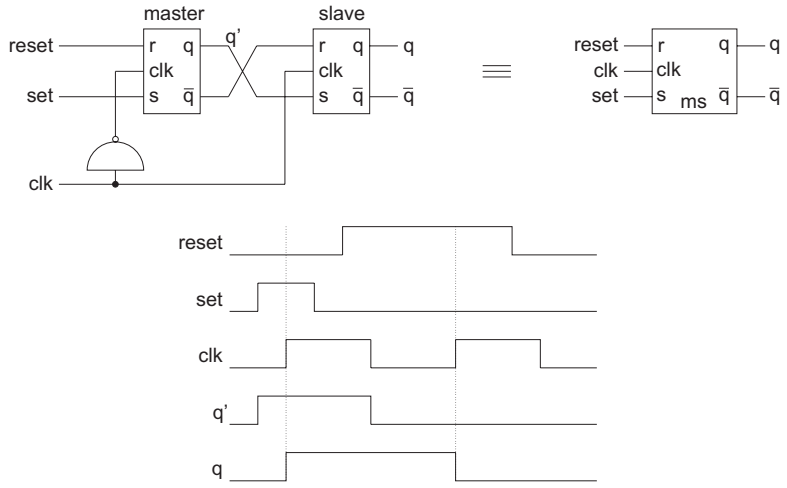
Pomnilna celica RS na sliki 2.9 spremeni svoje stanje takoj, ko je na vhodih prisotna zahteva po spremembi. Zaradi trenutnega odziva pravimo, da je asinhrona. Mikrokrmilniški sistemi pa delujejo sinhrono. Spremembe se dogajajo enakomerno ob vnaprej določenih trenutkih. Ritem narekuje pravokotni signal sistemske ure, ki s svojimi naraščajočimi, ali padajočimi frontami podaja trenutke sprememb. Sinhrono pomnilno celico RS dobimo iz asinhrone tako, da vhodna signala *reset* in *set* pripeljemo do asinhronih vhodov le ob prisotnosti urinega signala *clk* (slika 2.11). Sprememba stanja pomnilne celice se zgodi takoj, ko vhodno stanje pride do asinhronih vhodov, torej ob naraščajoči fronti urinega signala.



Slika 2.11: Sinhrona pomnilna celica RS prožena z urinim signalom

Vendar ima pomnilna celica na sliki 2.11 pomanjkljivost. Do spremembe stanja lahko pride tudi med urinim impulzom, in ne samo ob njegovi naraščajoči

fronti. To se zgodi takrat, kadar se signala *reset* in *set* spreminjata medtem, ko je urin signal visok. Naša pomnilna celica je pravzaprav prožena s stanjem urinega signala *clk*, in ne z njegovimi frontami. Da bi naredili pravo sinhrono pomnilno celico RS proženo ob frontah, moramo uporabiti dve celici proženi z nasprotnimi stanji urinega signala (slika 2.12). Stanje glavne (angl. master) celice se postavi, ko je urin signal v enem izmed stanj, ter se nato ob nasprotnem stanju prenese na delovno (angl. slave) celico.



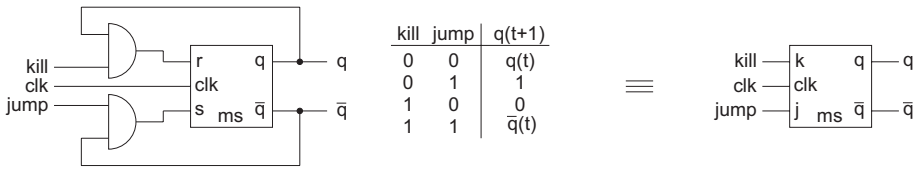
Slika 2.12: Sinhrona pomnilna celica RS s predpomnjenjem prožena z naraščajočimi frontami urinega signala

Prikazani vezavi pravimo tudi pomnilna celica s predpomnjenjem (angl. master/slave). Sinhrono pomnilno celico RS proženo s padajočimi frontami dobimo

tako, da urin signal negiramo. Obrnjen urin signal pride sedaj na delovno celico, ravno obratno kot pri celici proženi z naraščajočimi frontami.

2.4.2 Pomnilna celica JK

Pomnilna celica RS se obnaša nepredvidljivo v primeru prepovedanega vhodnega stanja. V ta namen definiramo sinhrono pomnilno celico JK (angl. Jump-Kill). Deluje naj enako kot celica RS, vhodu *reset* ustreza *kill* in vhodu *set* je ekvivalenten *jump*. Dodatno predpišemo obnašanje v primeru prepovedanega vhodnega stanja, torej ko je $jump = kill = 1$. In sicer naj pomnilna celica JK v tem primeru ob fronti urinega signala spremeni izhodno stanje q v nasprotno vrednost. Izvedbo sinhrone pomnilne celice JK s pomočjo sinhrone celice RS prikazuje slika 2.13.



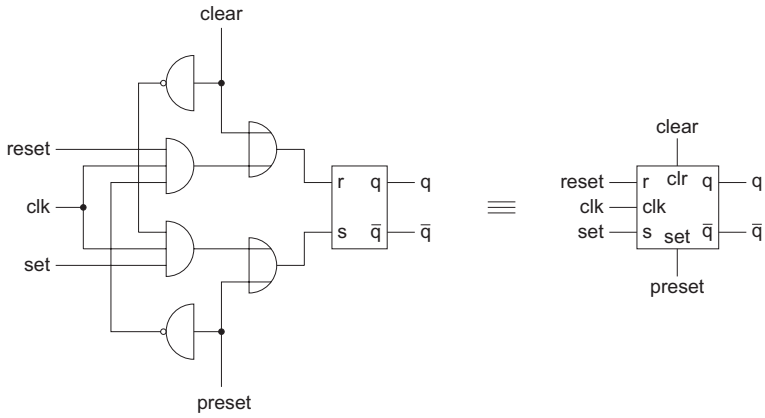
Slika 2.13: Sinhrona pomnilna celica JK prožena z naraščajočimi frontami urinega signala

Zaradi vrat AND pride signal *kill* do vhoda *reset* le v primeru, ko je pomnilna celica postavljena ($q = 1$). In obratno, signal *jump* pride do vhoda *set* le v primeru, ko pomnilna celica ni postavljena ($\overline{q} = 1$). Posledica takšne vezave je, da se prepovedano stanje na vhodih *reset* in *set* nikdar ne pojavi. Ob vhodnem stanju $jump = kill = 1$ je z vrati AND eden od signalov ustavljen, odvisno od

trenutne postavitve pomnilne celice. Zaradi splošnosti je sinhrona pomnilna celica JK največkrat uporabljana pomnilna celica v digitalnih vezjih.

2.4.3 Asinhroni vhodi

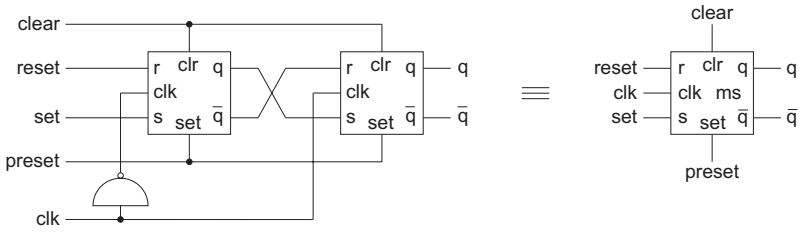
Sinhroni pomnilni celici RS proženi z urinimi signalom s slike 2.11 dodamo še asinhrona vhoda *clear* in *preset* kot prikazuje slika 2.14. Visoko stanje na vhodu *clear* začasno onemogoči urin signal *clk* in celico postavi v nizko stanje. Obratno vhod *preset* celico, zopet ne glede na urin signal, postavi v visoko stanje.



Slika 2.14: Sinhrona pomnilna celica RS z dodanima asinhronima vhodoma

Z uporabo opisane strukture pri gradnji celic s predpomnjenjem dobimo celice s sinhronimi in asinhronimi vhodi (slika 2.15). Asinhroni vhodi se upo-

rabljajo za postavitev stanja celice v želeno stanje ne glede na urin signal, kar navadno pride v poštev ob zagonu.



preset	clear	clk	set	reset	q(t+1)	$\bar{q}$(t+1)	
0	0	0 1 ↓	x	x	q(t)	$\bar{q}$(t)	
0	0	↓	0	0	q(t)	$\bar{q}$(t)	
0	0	↓	0	1	0	1	
0	0	↓	1	0	1	0	
0	0	↓	1	1	x	$\bar{x}$	(labile)
0	1	x	x	x	0	1	
1	0	x	x	x	1	0	
1	1	x	x	x	0	0	(labile)

Slika 2.15: Sinhrona pomnilna celica RS prožena z naraščajočimi frontami urinega signala z dodanima asinhronima vhodoma

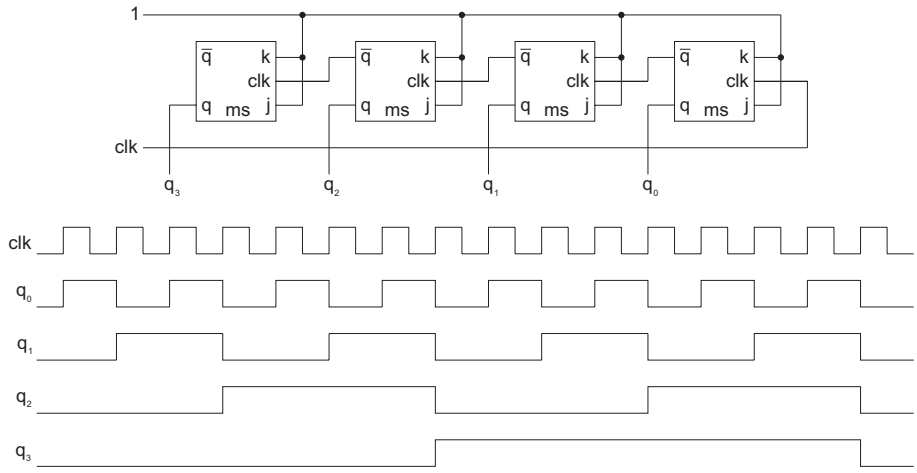
2.5 Registri

Register sestavlja skupina pomnilnih celic, ki skupaj shranjujejo več kot en bit informacije. Srečujemo jih v štiri, osem, šestnajst ... bitnih izvedbah, čeprav je njihova velikost v osnovi poljubna. Najenostavnejši register je preprost števnik (angl. counter). Poleg tega poznamo še shranjevalne (angl. buffer) in pomikalne (angl. shift) registre. Registri igrajo v mikrokrmilniških sistemih zelo pomembno vlogo. Poenostavljeno bi lahko rekli, da je mikrokrmilniški sistem

sestavljen iz množice registrov, med katerimi se pretakajo informacije v binarni obliki.

2.5.1 Števniki

Štiri bitni števnik sestavljen iz pomnilnih celic JK je prikazan na sliki 2.16. Števnik šteje urine impulze, oziroma bolje rečeno naraščajoče fronte urinega signala. Vsaka padajoča fronta na izhodu predhodne pomnilne celice (naraščajoča fronta na negiranem izhodu) povzroči preklop v naslednji celici. Števnik ima 16 različnih stanj, in sicer šteje od $q_3 = q_2 = q_1 = q_0 = 0$ do $q_3 = q_2 = q_1 = q_0 = 1$, nakar zopet prične z $q_3 = q_2 = q_1 = q_0 = 0$.



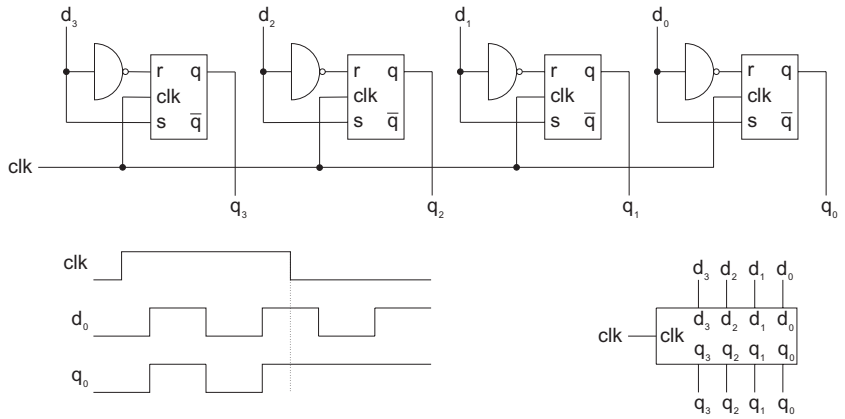
Slika 2.16: Štiri bitni števnik navzgor

Z različnimi vezavami in uporabo nekaj dodatnih logičnih vrat je možno realizirati razne vrste števnikov, ki štejejo navzgor ali navzdol, se ob določeni vrednosti postavijo na nič ... Skratka realiziramo lahko poljubno sekvenco štetja. Seveda velja splošna omejitev, in sicer ima števnik z n pomnilnimi celicami največ 2^n različnih stanj. Števnik lahko uporabimo za deljenje frekvence uri-

nega signala. Tako na sliki 2.16 na primer vidimo, da je frekvenca signala q_1 pravzaprav 1/4 frekvence urinega signala *clk*.

2.5.2 Shranjevalni registri

Shranjevalni (angl. buffer) registri binaren podatek za določen čas shranijo, oziroma ga zadržijo. Zaradi načina delovanja jih imenujemo tudi zatiči (angl. latch). So registri z vzporednim vhodom in izhodom, kar pomeni, da se v register zapišejo vsi biti binarnega podatka naenkrat, ter so tudi hkrati dostopni.



Slika 2.17: Štiri bitni zatič

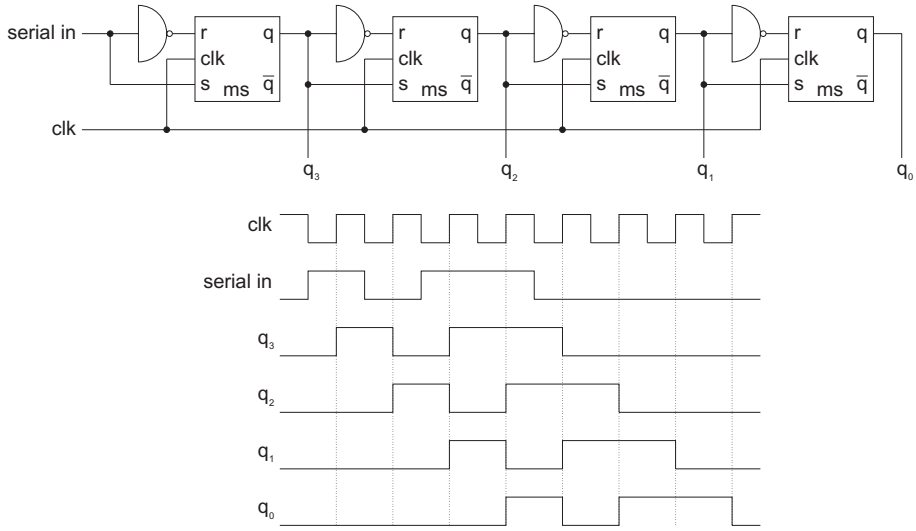
Slika 2.17 prikazuje štiri bitni zatič. Sestavljen je iz pomnilnih celic RS, ki so prožene s stanjem urinega signala. Ob visokem stanju ure se vhodno stanje $d_3 \ldots d_0$ preslika na izhod $q_3 \ldots q_0$. Vsak izhod sledi dogajanju na pripadajočem vhodu. Med vhodi in izhodi si v logičnem smislu lahko predstavljamo sklenjena stikala. Ob prehodu urinega signala v nizko stanje (padajoča fronta) se stanje izhodov zamrzne. Stikala med vhodi in izhodi se razklenejo in na izhodih ostanejo vrednosti, ki so bile na vhodih prisotne ob padajoči fronti ure.

Če bi hoteli imeti frontno prožen zatič, bi morali namesto pomnilnih celic RS proženih s stanjem ure uporabiti celice s predpomnjenjem. Izhodno stanje zatiča bi se lahko spremenilo le ob naraščajoči, ali padajoči fronti urinega signala. S

tem preprečimo neposredno preslikavo vhodov na izhode med trajanjem urinega impulza.

2.5.3 Pomikalni registri

Pomikalne (angl. shift) registre v grobem razdelimo po načinu vpisovanja in branja podatkov. Podatek vpišemo (ali ga preberemo) zaporedno ali vzporedno. Zaporeden prenos pomeni, da se posamezni biti podatka v času zvrstijo eden za drugim po eni sami povezavi. Pri vzporednem prenosu vsi biti podatka na voljo naenkrat, zato za vsak bit potrebujemo eno povezavo.

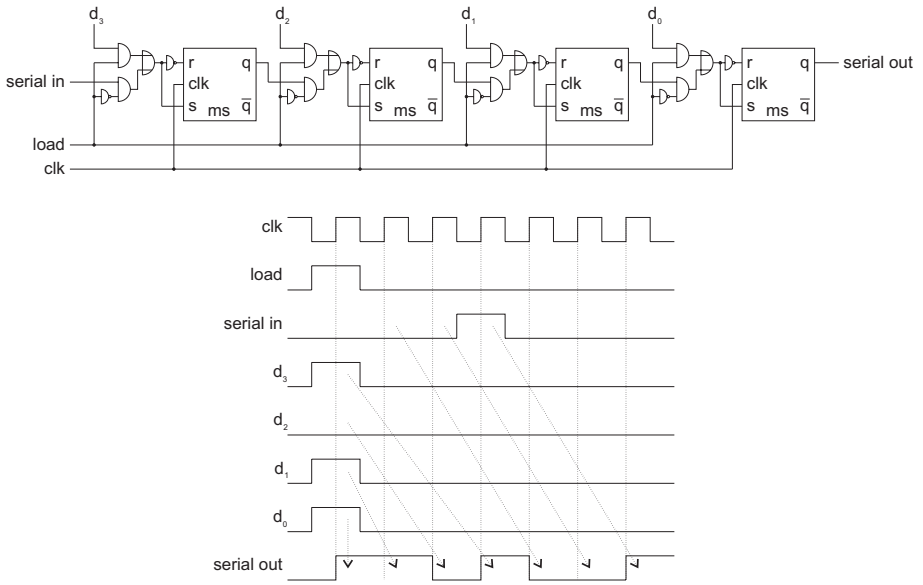


Slika 2.18: Pomikalni register z zaporednim vpisom in vzporednim branjem

Pomikalni register z zaporednim vpisom in vzporednim branjem (angl. serial-in, parallel-out) je prikazan na sliki 2.18. Njegovo delovanje je precej preprosto. Ob naraščajoči fronti urinega signala vsaka izmed pomnilnih celic prevzame stanje svoje predhodnice. Prva pomnilna celica prevzame stanje z zaporednega vhoda. Tako se vsebina registra pomakne za eno mesto. Ker so izhodi vseh po-

mnilnih celic dostopni, je mogoče celoten podatek prebrati naenkrat, oziroma vzporedno.

Pomikalni register z zaporednim vpisom in vzporednim branjem se uporablja pri pretvorbi podatkov iz zaporednega toka na eni sami povezavi v vzporedno obliko na več povezavah. Poznamo različne izvedbe, ki z nekaj dodatnimi logičnimi vrati omogočajo asinhron reset registra, dinamično določanje smeri pomika ... Omenimo še, da so pomikalni registri zaradi svoje narave frontno proženi. Proženje s stanjem ure ni primerno, saj bi biti podatka med pomnilnimi celicami prehajali z nekontrolirano hitrostjo. Zato so uporabljene pomnilne celice s predpomnjenjem.



Slika 2.19: Pomikalni register z vzporednim vpisom in zaporednim branjem

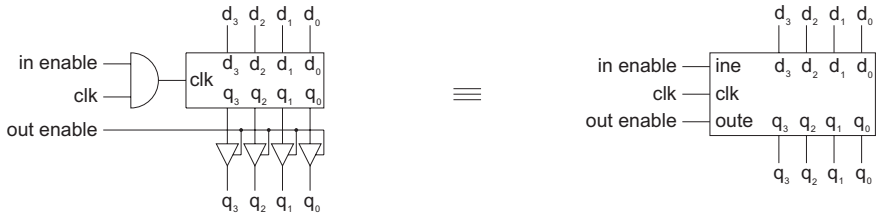
Obraten je pomikalni register z vzporednim vpisom in zaporednim branjem (angl. parallel-in, serial-out). Celoten podatek v register vpišemo naenkrat ob naraščajoči fronti urinega signala. Da se vzporeden vpis zgodi, mora biti vhod *load* visok. V nasprotnem primeru se podatek v registru pomakne. Oba načina delovanja določa vezava logičnih vrat pred vsako pomnilno celico (slika 2.19). Glede na vhod *load* se podatek v registru pomakne ($load = 0$), ali vzporedno

vpiše ($load = 1$). Vsebino registra beremo zaporedno na izhodu zadnje pomnilne celice *serial out*.

Ko je register v pomikalnem načinu delovanja ($load = 0$), je mogoče podatek vanj vpisati tudi zaporedno. V prvo pomnilno celico se vpisuje stanje na zaporednem vhodu *serial in*. V našem primeru na sliki 2.19 imamo sinhron vzporedni vpis podatka. Vpis v register se zgodi ob naraščajoči fronti urinega signala, in ne takoj, ko je omogočen ($load = 1$). Poznamo tudi izvedbe z asinhronim vpisom, ko se stanje z vhodov $d_3 \ldots d_0$ v pomnilne celice prenese ob *load* signalu, ne glede na urine impulze.

2.5.4 Tristanjski registri

Prenos podatkov iz enega v drug register navadno poteka preko vodila, na katerega je hkrati priključenih več registrov. Da se prenos od izvornega v ponorni register izvrši, morata biti na vodilo priključena le izbrana registra, ostali pa ne. Zato uporabljamo tristanjske registre, ki z izhodi v stanju visoke impedance vodilo prepustijo izvornemu registru. Na vseh izhodih so v ta namen dodani tristanjski ojačevalniki (glej opis v poglavju 2.3). Sponke *enable* so povezane skupaj v vhod *out enable* (slika 2.20).



Slika 2.20: Tristanjski štiri bitni zatič

Enako morajo vhodni signali vplivati le na ponorni register, in ne tudi na vse ostale. V ta namen je dodan vhod *in enable*, ki pravzaprav omogoča uro *clk*. Če urinega signala ni, potem vhodi ne vplivajo na stanje registra.

2.6 Vodila

Pri opisu tristanjskih logičnih vezij in registrov smo že omenili vodila (angl. bus), ki so v današnjih arhitekturah mikrokrmilniških sistemov [2] zadolžena za prenos informacij po sistemu. Glede na tip informacij, ki se po vodilu pretakajo, jih lahko razdelimo v tri skupine, in sicer:

1. podatkovno vodilo (angl. data bus), ki je namenjeno prenosu vsebine, oziroma podatkov,

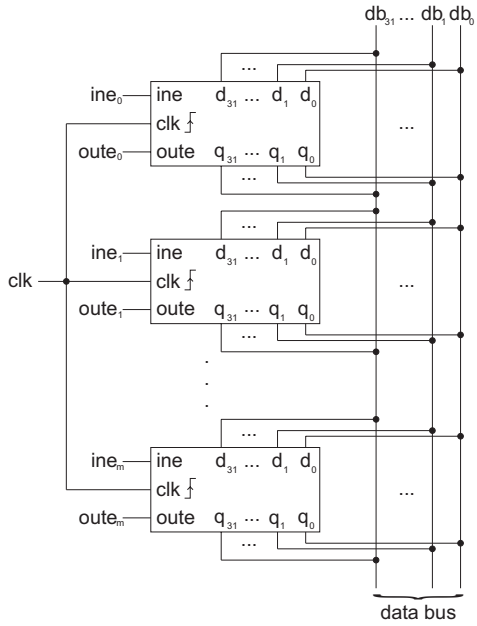
2. naslovno vodilo (angl. address bus), ki določa izvor in ponor prenosa podatka, ter

3. nadzorno vodilo (angl. control bus), ki celotno dogajanje usklajuje.

2.6.1 Podatkovno vodilo

Po podatkovnem vodilu se prenašajo podatki, kar pomeni, da vsebino enega registra prenesemo v drug register. Podatkovno vodilo je skupina povezav, kamor so priključeni vsi vhodi in vsi izhodi vseh z vodilom povezanih registrov (slika 2.21). S takšno vezavo je mogoče vsebino kateregakoli registra prenesti v

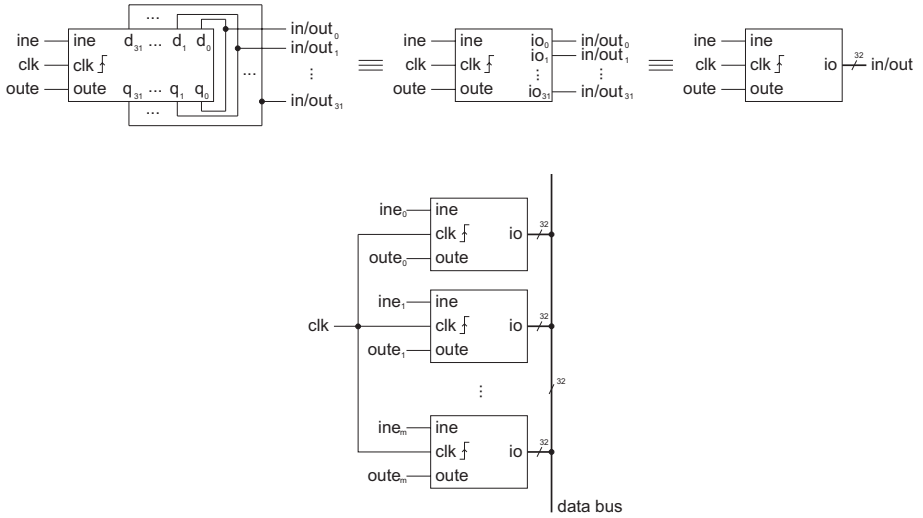
katerikoli drug register. Da se prenos izvrši, potrebujemo le ustrezne signale na vhodih (*clk*, *ine* in *oute*) posameznih registrov.



Slika 2.21: Podatkovno vodilo

Na sliki 2.21 prikazano podatkovno vodilo je sestavljeno iz 32-ih povezav in povezuje $(m + 1)$ 32 bitnih registrov. Znak $\lceil\!\downarrow$ na vhodih *clk* pomeni, da so registri proženi ob naraščajoči fronti urinega signala. Ker vodilo tvori 32 povezav, pravimo da je široko 32 bitov. Širina vodila je pravzaprav določena

z dolžino nanj priključenih registrov. Ker so naši registri 32 bitni, imamo 32 bitno podatkovno vodilo.



Slika 2.22: Poenostavljen prikaz podatkovnega vodila

Na vsakem izmed registrov je i-ti vhod d_i preko podatkovnega vodila kratko povezan z i-tim izhodom q_i. To pomeni, da lahko vhode in izhode registrov med seboj kratko sklenemo (slika 2.22). S tem prihranimo polovico povezav s podatkovnim vodilom. Prav tako postane shema vezja nepregledna, če vodilo narišemo takšno, kot v resnici je. V našem primeru je sestavljeno iz 32-ih povezav. Iz tega razloga vse povezave simbolično združimo, kar naredi shemo vezja bolj berljivo.

Prenos vsebine iz i-tega v j-ti register se izvrši ob naraščajoči fronti urinega signala. Da se to zgodi, morajo biti na vhodih *ine* in *oute* registrov na podatkovnem vodilu vrednosti podane v (2.20).

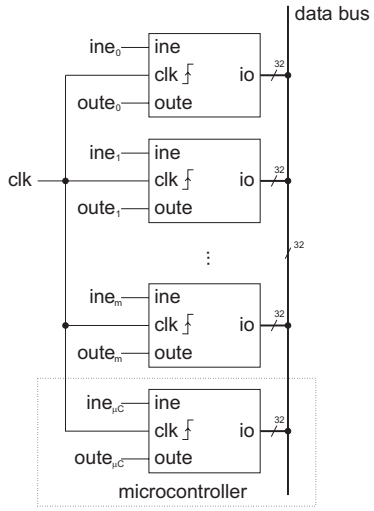
$$\begin{array}{ll} ine_i = 0 & oute_i = 1 \\ ine_j = 1 & oute_j = 0 \\ ine_k = 0 & oute_k = 0 \quad k = 0, 1 \ldots m \quad k \neq i \quad k \neq j \end{array} \tag{2.20}$$

Na vodilo je priključen izhod i-tega registra ($oute_i = 1$). Njegovo stanje določa napetosti na njem. Izhodi vseh ostalih registrov so v stanju visoke im-

pedance. Tako je stanje vodila enolično podano. Če na vodilo ni priključen nobeden izmed izhodov, potem napetosti na njem niso določene. Pravimo, da vodilo "plava". Priključitev več izhodov pa lahko pripelje do vsiljevanja različnih napetosti. Da se vsebina prenese v j-ti register, je na vodilo priključen njegov vhod ($ine_j = 1$). Ostali vhodi so onemogočeni, čeprav to za delovanje vodila ni nujno. Če bi bil omogočen še kateri izmed vhodov, bi se vsebina i-tega registra hkrati prenesla tudi v ta register.

2.6.2 Nadzorno vodilo

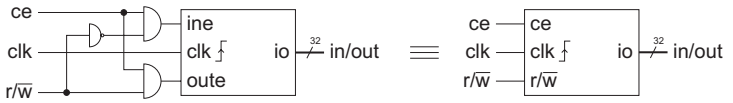
Z vezavo $(m + 1)$ registrov na sliki 2.22 je mogoče podatke po podatkovnem vodilu prenašati med poljubnima registroma. Da se to res zgodi, je potrebno poleg urinih impulzov *clk* zagotoviti še $2 \times (m+1)$ krmilnih signalov *ine* in *oute*, kar je zelo veliko.



Slika 2.23: Podatkovno vodilo z dodanim mikrokrmilnikom

Na podatkovno vodilo priključimo še poseben register. Dogovorimo se, da je novi register udeležen v vseh prenosih podatkov po podatkovnem vodilu. V posebni register lahko prenesemo vsebino poljubnega registra, vsebino posebnega registra pa je mogoče zopet prenesti v poljubni register. Prenosi med dvema

poljubnima registroma niso več dovoljeni. To lahko sedaj storimo v dveh korakih, in sicer s prenosom podatka iz izvornega v posebni register in nato od tam v ponorni register. Posebni register je pomembnejši od vseh ostalih in na nek način nadzira dogajanje na podatkovnem vodilu. V mikrokrmilniških sistemih je ta register del mikrokrmilnika. Odtod tudi indeks μC (angl. microcontroller), s katerim je register na sliki 2.23 označen.



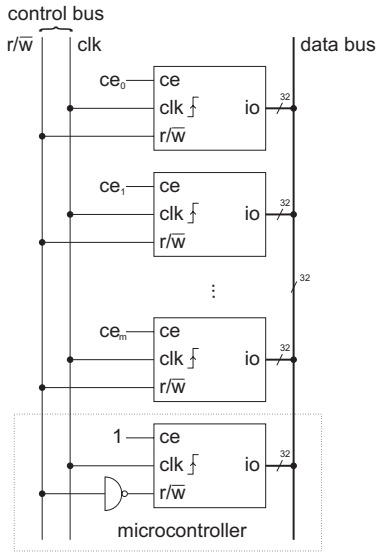
ce	ine	oute	
0	0	0	register is not involved at the transfer
1	r/w	r/w	register is data source or data drain regarding the r/w

Slika 2.24: Tristanjski 32 bitni bitni zatič in s krmilnima signaloma *ce* (angl. Chip Enable) in $r/\overline{w}$ (angl. Read/Write)

Ker vsi prenosi po podatkovnem vodilu sedaj potekajo preko mikrokrmilnika, lahko krmiljenje celotnega sistema poenostavimo. V ta namen najprej preoblikujmo registre priključene na vodilo. Namesto, da ima vsak register dva krmilna signala *ine* in *oute*, uvedemo signala *ce* (angl. Chip Enable) in $r/\overline{w}$ (angl. Read/Write). Zveza med signali *ine*, *oute*, *ce* in $r/\overline{w}$ je prikazana na sliki 2.24. Signal *ce* določa, ali izbran register pri prenosu sodeluje, ali ne. Signal $r/\overline{w}$ pa podaja smer prenosa.

Če register pri prenosu ne sodeluje ($ce = 0$), potem so njegovi izhodi v stanju visoke impedance ($oute = 0$), stanje na vhodih pa se vanj ne zapiše ($ine = 0$). Signal $r/\overline{w}$ je pomemben le v primeru, ko register pri prenosu sodeluje ($ce = 1$). Ko mikrokrmilnik iz registra bere ($r/\overline{w} = 1$), so na vodilo priključeni njegovi

izhodi ($ine = 0$, $oute = 1$). In obratno, ko mikrokrmilnik v register piše ($r/\overline{w} = 0$), se stanje na vodilu shrani ($ine = 1$, $oute = 0$).



Slika 2.25: Nadzorno in podatkovno vodilo

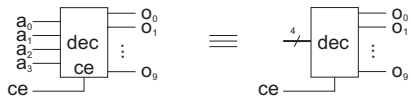
Mikrokrmilnik sodeluje v vseh prenosih po podatkovnem vodilu. Zato je njegov signal *ce* stalno v visokem stanju. Prav tako ima glede na ostale registre ravno obrnjen signal $r/\overline{w}$. Kadar bere, to pomeni zapis v njegov register. In obratno, kadar piše, to pomeni branje iz njega. Namesto $2 \times (m + 1)$ krmilnih signalov imamo sedaj pol manj signalov *ce*. Signala $r/\overline{w}$ in *clk* sta skupna vsem registrom in zato sodita v nadzorno vodilo (slika 2.25).

2.6.3 Naslovno vodilo

Da promet po podatkovnem vodilu nemoteno teče, je potrebno zagotoviti signala *clk* in $r/\overline{w}$, ter še $(m + 1)$ signalov *ce*. V mikrokrmilniških sistemih je m največkrat zelo veliko število, kar z drugimi besedami pomeni, da bi morali krmiliti zelo veliko število povezav *ce*. Ker z mikrokrmilnikom po podatkovnem vodilu komunicira le eden izmed registrov naenkrat, je v visokem stanju vedno le eden izmed signalov *ce*. Vsi ostali so v nizkem stanju. Na $(m + 1)$ povezavah

ce je torej možnih le $(m+1)$ različnih stanj. Vedno je točno ena povezava v visokem stanju. Informacijo o tem, katera izmed $(m+1)$ povezav je to, lahko podamo bolj zgoščeno. In sicer lahko z n povezavami označimo 2^n različnih stanj. Od tod sledi zveza med m in n, ki je podana z izrazoma (2.21).

$$\begin{aligned} 2^n &\geq m+1 \\ m \leq 2^n - 1 \qquad &\text{oziroma} \qquad n \geq \log_2(m+1) \end{aligned} \tag{2.21}$$



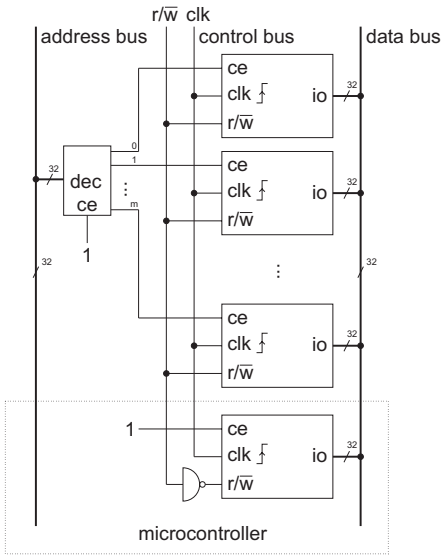
ce = 1

a_3	a_2	a_1	a_0	o_9	o_8	...	o_1	o_0
0	0	0	0	0	0	...	0	1
0	0	0	1	0	0	...	1	0
⋮				⋮			⋮	
1	0	0	0	0	1	...	0	0
1	0	0	1	1	0	...	0	0
1	0	1	0	0	0	...	0	0
⋮				⋮			⋮	
1	1	1	1	0	0	...	0	0

Slika 2.26: Dekodirnik iz $n = 4$ na $(m+1) = 10$

Števili m in n sta celi. Element, ki opravlja pretvorbo iz n na $(m+1)$ povezav, se imenuje dekodirnik. Slika 2.26 prikazuje primer BCD dekodirnika, ki ima $n = 4$ vhode $a_3 \ldots a_0$ in $(m+1) = 10$ izhodov $o_9 \ldots o_0$. Ker so vrednosti izhodov podane z logičnimi izrazi (na primer $o_4 = ce \times \overline{a_3} \times a_2 \times \overline{a_1} \times \overline{a_0}$), ga je mogoče enostavno sestaviti iz logičnih vrat. V našem primeru vhodi določajo $2^n = 16$ različnih stanj. Imamo le 10 izhodov, zato ostane 6 stanj neizkoriščenih.

Takrat so vsi izhodi nizki. Vsi izhodi so v nizkem stanju tudi v primeru, ko dekodirnik ni omogočen ($ce = 0$).



Slika 2.27: Osnovna zgradba mikrokrmilniškega sistema

Dekodirnik uporabimo v našem sistemu. Namesto $(m + 1)$ signalov *ce* moramo podati le številko registra, s katerim si mikrokrmilnik trenutno izmenjuje podatek. Ker n povezav podaja številko, oziroma naslov registra, jih imenujemo naslovno vodilo. Dobili smo osnovno zgradbo mikrokrmilniškega sistema (slika 2.27).

Prenos podatkov po podatkovnem vodilu določata nadzorno in naslovno vodilo. Signali na obeh vodilih definirajo dogajanje v sistemu, nad čemer bedi mikrokrmilnik. Na sliki 2.27 je naslovno vodilo široko 32 bitov. To pomeni, da imamo na podatkovnem vodilu lahko priključenih največ 2^{32} registrov. Oziroma

na voljo imamo 2^{32} različnih naslovov, kar določa velikost naslovnega prostora.

V vsakem registru je shranjen 32 bitni podatek, oziroma 4 bajti. To pomeni, da imamo v našem sistemu teoretično prostora za $2^{32} \times 32\text{b} = 4 \times 2^{30} \times 4\text{B} = 16\text{GB}$ podatkov. Vendar je v mikrokrmilniških sistemih z 32 bitnim podatkovnim in naslovnim vodilom navadno možno nasloviti največ 4GB podatkov. Vsak 32 bitni naslov predstavlja le en bajt, oziroma eno četrtino 32 bitnega registra. Register se tako razteza preko štirih naslovov. To z drugimi besedami pomeni, da imamo efektivno opraviti le s 30 bitnim naslovnim vodilom, zadnja dva bita naslova pa sta vedno enaka nič. Vrednosti naslovov vedno zapisujemo v šestnajstiškem zapisu, kar označimo s predpono `0x`. Primer: register na naslovu `0x40000000` se razteza od naslova `0x40000000` do vključno `0x40000003`, naslov naslednjega registra je `0x40000004`. Ker je naslovno vodilo 32 bitno le navidez, se poraja vprašanje, kaj se zgodi, če naslovimo podatek, ki ni poravnan z registrsko strukturo. Največkrat mikrokrmilnik naslov enostavno avtomatsko poravna. Primer takšne zgradbe naslovnega prostora najdemo v Philipsovem mikrokrmilniku LPC2138.

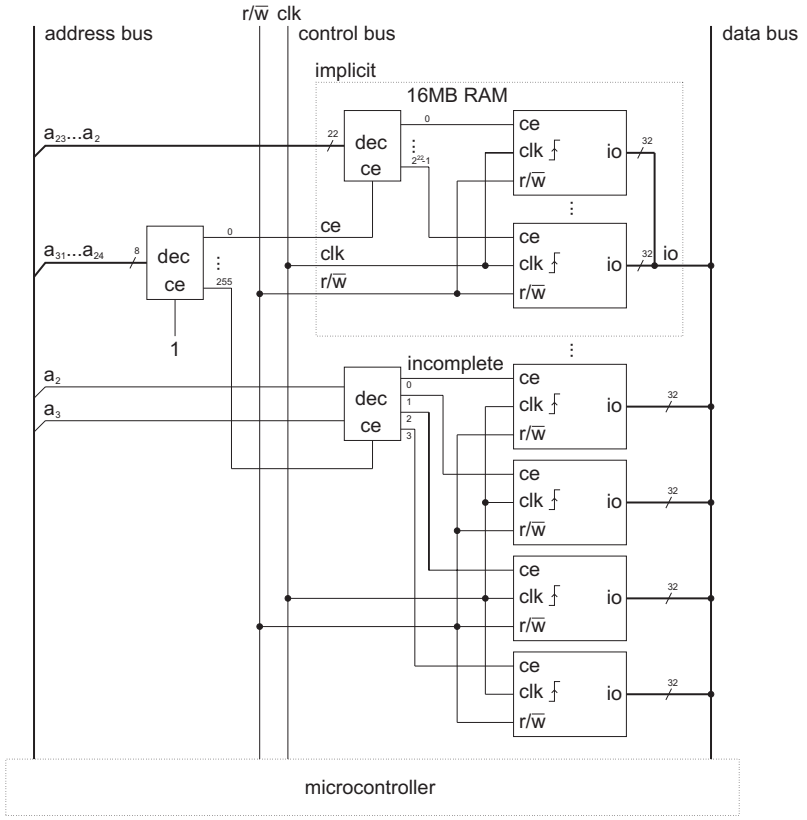
2.7 Dekodiranje

Preslikavi naslova na naslovnem vodilu na izbran register imenujemo dekodiranje. Poznamo mnogo tehnik dekodiranja, ki je lahko:

1. statično ali dinamično,
2. eksplicitno ali implicitno,
3. popolno ali nepopolno in
4. simetrično ali asimetrično.

Na sliki 2.27 je dekodiranje izvedeno z enim samim dekodirnikom. Takšno dekodiranje je statično, eksplicitno, popolno in simetrično. Statično je, ker je

dekodirno vezje nespremenljivo. Izbran naslov vedno podaja isti register. Poleg statičnega poznamo še dinamično dekodiranje, o čemer bomo govorili kasneje.



Slika 2.28: Načni dekodiranja (eksplicitno/implicitno in popolno/nepopolno dekodiranje)

Naslov vsakega izmed registrov je dekodiran eksplicitno z zunanjim dekodirnim vezjem. Implicitno dekodiranje pomeni, da ima skupina registrov svoj

interni dekodirnik. Glavni eksplicitni dekodirnik navadno dekodira le zgornji del naslova, implicitni dekodirnik pa spodnji del. Na sliki 2.28 je naslovni prostor z eksplicitnim dekodirnikom razdeljen v 2^8 odsekov po $2^{24} = 16M$ naslovov. Eksplicitni dekodirnik poskrbi za zgornjih 8 bitov naslovnega vodila, implicitni pa za spodnjih 24. Eksplicitni in implicitni dekodirnik sta zvezana v dvostopenjsko kaskado.

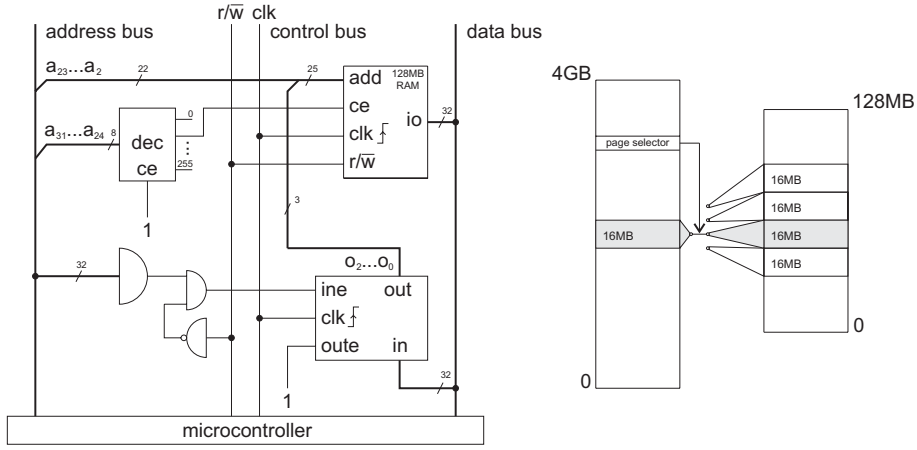
Popolno dekodiranje pomeni, da ima vsak register točno en naslov. Če več naslovov podaja isti register, imamo opravka z nepopolnim dekodiranjem. Register se preslika na več naslovov v naslovnem prostoru. To pomeni, da nekateri biti naslovnega vodila pri dekodiranju niso uporabljeni.

Poseben primer predstavlja asimetrično dekodiranje. Poljuben naslov ali skupino naslovov izločimo iz naslovnega prostora z dekodirnikom, ki zazna le točno izbran naslov. Le ta se lahko nahaja kjerkoli v naslovnem prostoru. Asimetrično dekodiranje se navadno uporablja za naslavljanje nesplošnih registrov z vnaprej določeno vlogo (na primer register analogno/digitalnega pretvornika).

2.7.1 Dinamično dekodiranje

Kot pove ime, dinamično dekodiranje pomeni s časom spremenljivo naslavljanje registrov. Nek naslov ne vodi enolično do izbranega registra. Registrov na enem naslovu je lahko več. To ni v skladu z našim dosedanjim statičnim konceptom, ko je vsakemu registru naslov enolično dodeljen (pri nepopolnem dekodiranju celo več naslovov). Kateri izmed naslovljenih registrov bo izbran, je odvisno od dodatnih informacij, ki jih na naslovnem vodilu ne najdemo. Dinamični

dekodirnik za dekodiranje poleg naslovnega vodila vedno uporablja še dodatne vire.



Slika 2.29: Dinamično dekodiranje - 128MB pomnilnik je razdeljen v 8 odsekov po 16MB

Dinamično dekodiranje uporabljamo ob posebnih prilikah. Ena izmed možnih uporab je v velikih mikroprocesorskih sistemih, ko se pojavi potreba po večjem pomnilniku, kot je na voljo naslovnega prostora. Za primer si poglejmo zgradbo sistema na sliki 2.29. Za zunanji pomnilnik naj bo na voljo le en segment z 2^{24} naslovi. Zaradi 32 bitne dolžine registrov imamo le 2^{22} poravnanih naslovov. Če hočemo priključiti več kot 2^{22} registrov, moramo uporabiti dinamično dekodiranje. V našem primeru je uporabljen 128MB pomnilnik. Za naslavljanje 32 bitnih registrov v njem potrebujemo $2^{25} = 8 \times 2^{22}$ poravnanih naslovov ($128\text{MB} = 8 \times 2^{24}\text{B} = 8 \times 2^{22} \times 32\text{b}$). Torej 8 krat več kot jih imamo na razpolago. Tri manjkajoče bite naslova naj podaja stanje posebnega registra, ki se v našem primeru nahaja na asimetrično dekodiranem naslovu. Trenutno naslovljeno 16MB stran v 128MB pomnilniku določamo programsko z vrednostjo omenjenega registra. Navadno za preklapljanje med stranmi skrbi operacijski sistem. Register ima na podatkovno vodilo priključene le vhode in pravzaprav predstavlja zatič (glej sliko 2.20). Njegovi izhodi so ves čas na voljo, saj tri izmed njih uporabljamo kot del naslova 128MB pomnilnika. Asimetrični deko-

dirnik je predstavljen simbolično z več vhodnimi vrati AND, ki iz naslovnega vodila dekodirajo točno določen naslov.

Z dinamičnim dekodiranjem lahko naslovni prostor z vnosom dodatnih informacij poljubno razširimo. Razmislite, kaj dosežemo, če pri dinamičnem dekodiranju upoštevamo na primer signal $r/\overline{w}$.

2.8 Pomnilniki

Pomnilnik dobimo, če več registrov združimo. Na sliki 2.28 je na primer uporabljen pomnilnik velikosti 16MB. Pomnilniki se med seboj razlikujejo po notranji zgradbi in načinu shranjevanja informacij, kar v veliki meri določa tudi hitrost branja, oziroma pisanja vanje. Glede na način dostopa do informacij razlikujemo med:

1. pomnilniki z naključnim dostopom (angl. random access); čas dostopa do kateregakoli registra je vedno enak, oziroma naslovi zahtevanih registrov si lahko sledijo v poljubnem vrstnem redu, ne da bi s tem vplivali na čas dostopa,

2. pomnilniki z zaporednim dostopom (angl. serial access); čas dostopa je odvisen od naslova prejšnjega dostopa do pomnilnika; preleteti je potrebno vse lokacije med prejšnjim in trenutnim naslovom, zato je čas dostopa spremenljiv, lahko tudi zelo dolg, in

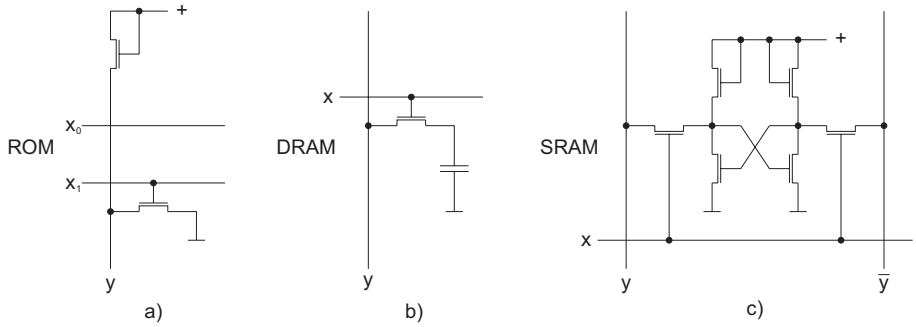
3. pomnilniki z neposrednim dostopom (angl. direct access); čas dostopa je odvisen od naslova prejšnjega dostopa do pomnilnika, vendar v mnogo manjši meri kot pri pomnilnikih z zaporednim dostopom; preleti vseh lokacij med prejšnjim in trenutnim naslovom niso potrebni.

Pogledali si bomo le nekaj vrst pomnilnikov z naključnim dostopom. Le ti v mikrokrmilniških sistemih predstavljajo delovni pomnilnik, kjer se nahajata tako

program kot podatki potrebni za delovanje. Pomnilnike z naključnim dostopom v grobem razdelimo v dve skupini, in sicer:

1. bralni pomnilniki (angl. ROM - Read Only Memory) katerih vsebine med delovanjem ne moremo spreminjati in

2. bralno-pisalni pomnilniki (angl. RAM - Random Access Memory) katerih vsebina je spremenljiva.



Slika 2.30: Izvedbe pomnilnih celic na tranzistorskem nivoju: a) celici ROM z stalno vpisanim visokim (zgoraj), oziroma nizkim stanjem (spodaj), b) dinamična celica RAM (angl. DRAM - Dynamic RAM) in c) statična celica RAM (angl. SRAM - Static RAM)

Zgradba pomnilnika z naključnim dostopom sledi iz tehnološke izvedbe pomnjenja. Na sliki 2.30 sta prikazani celici bralnega pomnilnika ROM, ter dinamična in statična izvedba celice bralno-pisalnega pomnilnika DRAM in SRAM. Vsaka celica shranjuje en bit informacije.

Posamezna celica je naslovljena z visokim nivojem napetosti na povezavi x. Podatek v njej preberemo, ali ga vanjo zapišemo preko povezave y. Iz bralnega pomnilnika ROM (slika 2.30a) lahko podatek le preberemo. Ko z naslovno linijo x_0 naslovimo zgornjo celico, dobimo na podatkovni liniji y prek bremenskega tranzistorja visok nivo napetosti. Če naslovimo spodnjo celico x_1, se pripadajoč tranzistor odpre in podatkovno linijo y sklene na maso. Prisotnost tranzistorja zapisuje nizko, odsotnost pa visoko stanje. Vsebine pomnilnika načeloma ne moremo spreminjati. Navadno so v bralnih pomnilnikih prisotni vsi tranzistorji

na vseh križiščih naslovnih in podatkovnih linij. Na mestih, kjer naj bo zapisano visoko stanje, je povezava s podatkovno linijo prekinjena. Z različnimi tehnološkimi izvedbami je mogoče prekinjene povezave zopet vzpostaviti, kar pravzaprav pomeni, da je v pomnilnik mogoče tudi pisati. Tako poleg bralnega pomnilnika ROM poznamo še:

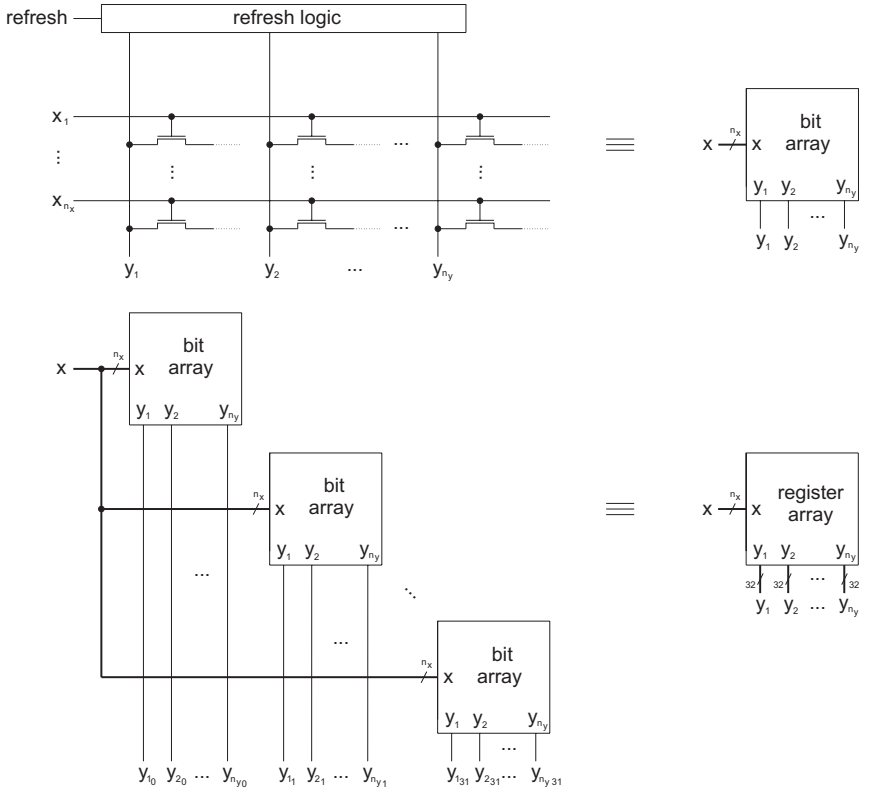
1. pomnilnik PROM (angl. Programmable ROM), v katerega je možen enkraten zapis,
2. pomnilnik EPROM (angl. Erasable PROM), katerega vsebino moramo pred novim zapisom izbrisati z ultravijolično svetlobo,
3. pomnilnik EEPROM (angl. Electrically EPROM), kjer za izbris pred ponovnim vpisom ne potrebujemo ultravijolične svetlobe, pač pa to naredimo električno za vsako celico posebej,
4. pomnilnik flash, ki je ekvivalenten pomnilniku EEPROM, le da lahko izbrišemo vse celice naenkrat ...

Za zapis v pomnilnika PROM in EPROM potrebujemo posebno strojno opremo, programator. Mikrokrmilnik lahko vsebino le bere, zato ti dve vrsti pomnilnikov upravičeno prištevamo med bralne pomnilnike. V pomnilnika EEPROM in flash lahko mikrokrmilnik tudi piše, kar pomeni, da sta to pravzaprav bralno-pisalna pomnilnika. Vendar je pisanje mnogo počasnejše kot branje, zaradi česar jih ne uporabljamo za shranjevanje tekočih podatkov, za kar se bralno-pisalni pomnilniki najpogosteje uporabljajo. Tako pomnilnika EEPROM in flash vseeno štejemo med bralne pomnilnike.

Delovanje dinamične celice RAM (slika 2.30b) je podobno delovanju celice ROM. Celica je naslovljena z napetostjo na liniji x, ki odpre tranzistor in kondenzator kratko sklene s podatkovno linijo y. Vsiljena napetost na podatkovni liniji y kondenzator nabije ali izprazni, ter tako v celico zapiše visoko ali nizko stanje. Ob branju na njej dobimo napetost, na katero je kondenzator nabit. Vendar se naboj kondenzatorja počasi izgublja, kar ima za posledico vedno nižjo napetost. Tako bi se zapisano visoko stanje sčasoma pretvorilo v nizko. Da vsebine ne izgubimo, je potrebno celice DRAM osveževati.

Statična celica RAM (slika 2.30c) je narejena z bistabilnim multivibratorjem, ki predstavlja na poseben način zvezano pomnilno celico RS. Celico zopet naslovimo z naslovno linijo x, ki odpre nanjo priključena tranzistorja. Podatkovni liniji sta tokrat dve y in $\overline{y}$. Na katerikoli izmed njiju lahko preberemo stanje, oziroma negirano stanje celice. Dve podatkovni liniji sta potrebni zaradi zapisa podatka v celico. Bistabilni multivibrator se nahaja v enem izmed

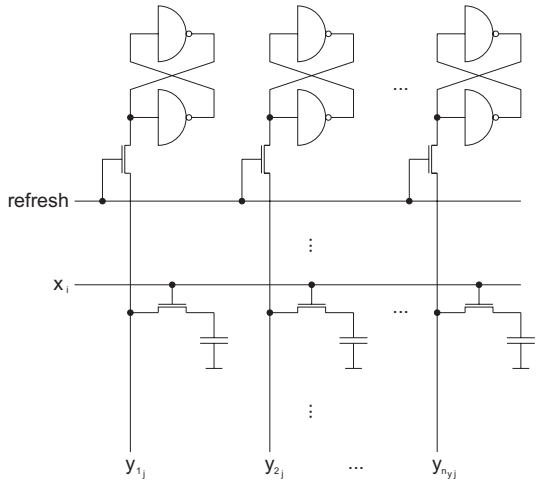
stabilnih stanj. Bremenska tranzistorja predstavljata ohmsko breme, eden izmed delovnih tranzistorjev je odprt (kratek stik), drugi zaprt (odprte sponke). Preklop povzroči vsiljevanje nizkega stanja na tisti podatkovni liniji, kjer je delovni tranzistor zaprt. Vsiljevanje visokega stanja na odprt tranzistor je mnogo težje. Ključno vlogo ob zapisu nizkega stanja tako odigra podatkovna linija y, ob zapisu visokega pa $\overline{y}$.



Slika 2.31: Zgradba pomnilnika z naključnim dostopom (pri pomnilnikih SRAM so podatkovne linije podvojene, osveževalna logika je prisotna le v pomnilnikih DRAM

Celica pomnilnika z naključnim dostopom se nahaja na križišču naslovne linije x in podatkovne linije y (pri pomnilniku SRAM še negirane podatkovne

linije $\overline{y}$). Več naslovnih in podatkovnih linij tvori pravokotno strukturo pomnilnih celic, kamor lahko zapišemo $n_x \times n_y$ bitov informacije (slika 2.31 zgoraj). Takšni pravokotni strukturi pravimo bitna ravnina. V mikrokrmilniških sistemih nas zapis, oziroma branje posameznih bitov ne zanima. Na podatkovno vodilo so priključeni registri dolžine na primer 32 bitov. To dosežemo z naslavljanjem več bitnih ravnin hkrati (slika 2.31 spodaj). Naslovna linija x_i naslovi 32 vrstic na enkrat, v vsaki bitni ravnini po eno. Istoležne podatkovne linije $y_{j0} \ldots y_{j31}$ tvorijo 32 bitni register. Ker naslovna linija x_i naslovi celo vrstico, je hkrati naslovljenih vseh n_y registrov v tej vrstici.



Slika 2.32: Princip osveževanja pomnilnika DRAM

Na sliki 2.31 je na vrhu bitne ravnine narisana osveževalna logika, ki je prisotna le v pomnilnikih DRAM. Podrobnosti osveževanja presegajo okvir te skripte, zato podajmo le princip delovanja. Pomnilne celice v pomnilniku DRAM je zaradi puščanja naboja shranjenega v kondenzatorju potrebno osveževati. Vsebino pomnilne celice preberemo in nato prebrano zapišemo nazaj, s čimer obnovimo naboj v kondenzatorju. Da shranjenih informacij ne izgubimo, celice osvežujemo dovolj hitro v enakomernih časovnih intervalih. Za osveževanje z branjem

in pisanjem v registre pomnilnika DRAM ne skrbi mikrokrmilnik, pač pa imamo za to posebno vezje prikazano na sliki 2.32.

V času, ko pomnilnika DRAM mikrokrmilnik ne uporablja, so podatkovne linije plavajoče, saj niso priključene na podatkovno vodilo. Naslovimo eno izmed vrstic v bitnih ravninah. Na podatkovnih linijah se pojavijo napetosti, na katere so nabiti kondenzatorji posameznih celic. Visoko stanje na osveževalnem vhodu na podatkovne linije priključi še bistabilne multivibratorje. Vrata NOT imajo visokoohmske izhode, zato napetost na kondenzatorju za kratek čas preglasi izhod zgornjih vrat NOT. To je dovolj, da bistabilni multivibrator preklopi v želeno stanje, ki je enako trenutnemu stanju kondenzatorja. V kolikor je bilo v celici shranjeno visoko stanje, se napetost na njem obnovi, oziroma naboj v njem se osveži.

Z naslovitvijo i-te vrstice naslovimo vrstice v vseh bitnih ravninah hkrati. Tako osvežimo n_y registrov na enkrat. Vrstice osvežujemo eno za drugo v enakomernih časovnih intervalih. Za vsako podatkovno linijo potrebujemo en multivibrator, oziroma eno celico SRAM.

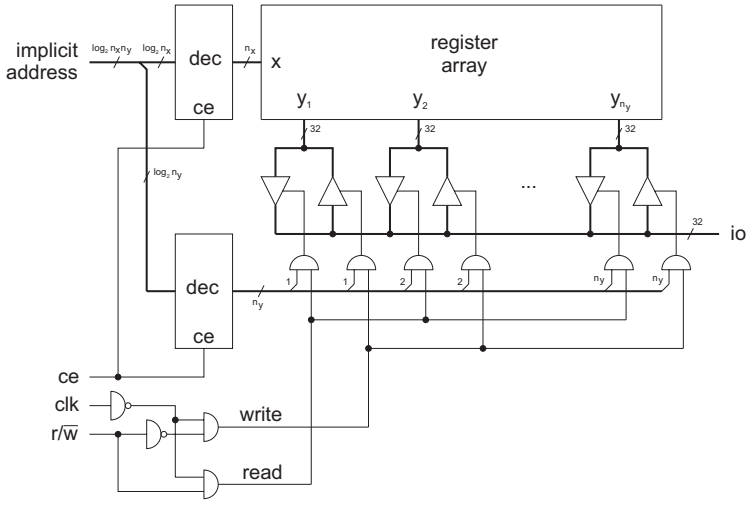
2.8.1 Krmiljenje pomnilnika z naključnim dostopom

Na sliki 2.28 je prikazan pomnilnik velikosti 16MB z internim delnim dekodirnikom. Vsak register ima svojo naslovno linijo, čemur pravimo enodimenzionalno ali linearno naslavljanje. Vendar so, kot smo spoznali v poglavju 2.8, pomnilniki z naključnim dostopom navadno sestavljeni iz pravokotnih bitnih ravnin. Število bitnih ravnin je enako dolžini registrov v pomnilniku. Če je pomnilnik organiziran v 32 bitne registre, potem imamo 32 bitnih ravnin. Zaradi pravokotne zgradbe je vedno naslovljenih n_y registrov hkrati, kar imenujemo dvodimenzionalno naslavljanje. Registri nimajo svojih naslovnih linij.

Prednost dvodimenzionalnega naslavljanja se kaže predvsem v številu naslovnih in podatkovnih linij. Medtem, ko na primer za pomnilnik velikosti 16MB organiziran v 32 bitne registre pri linearnem naslavljanju potrebujemo 2^{22} naslovnih in 32 podatkovnih linij, jih z dvodimenzionalnim naslavljanjem skupaj potrebujemo le še 33×2^{11} pri kvadratni strukturi $n_x = n_y = 2^{11}$, ali $2^{12} + 32 \times 2^{10}$ pri pravokotni strukturi $n_x = 2^{12}$ in $n_y = 2^{10}$ ... (velja za pomnilnike ROM in DRAM, pri pomnilnikih SRAM so podatkovne linije podvojene). Vsekakor mnogo manj, zaradi česar se v pomnilnikih z naključnim dostopom največkrat uporablja dvodimenzionalno naslavljanje.

Slabost dvodimenzionalnega naslavljanja so daljši dostopni časi. Pomnilniki z enodimenzionalnim naslavljanjem so hitrejši. Poleg tega pomnilnikov z dvodimenzionalnim naslavljanjem ne moremo transparentno priključiti na na-

slovno, nadzorno in podatkovno vodilo mikrokrmilniškega sistema. Potrebujemo dodatno krmilno vezje, katerega princip delovanja bomo spoznali v naslednjih odstavkih.



Slika 2.33: Krmiljenje dvodimenzionalnega polja registrov

Implicitni dekodirnik je v dvodimenzionalni zgradbi razdeljen na vrstični in stolpični del. Vrstični dekodirnik dekodira vrstico, stolpični pa stolpec, kjer se naslovljeni register nahaja. Dekodirnika dekodirata vsak svoj del implicitnega naslova, kakor prikazuje slika 2.33. Dekodiranje se zgodi le, ko je pomnilnik kot celota eksplicitno naslovljen ($ce = 1$). Na naslovnem vodilu je takrat naslov enega izmed registrov v njem.

Vrstični dekodirnik naslovi eno izmed n_x vrstic. Stolpični dekodirnik prav tako naslovi enega izmed n_y stolpcev. Podatkovne linije (stolpci) imajo vzporedno vezane vhodne in izhodne tristanjske ojačevalnike, ki omogočajo branje in pisanje po isti povezavi. Vhodni ojačevalniki naslovljenega stolpca so omogočeni, ko hočemo v register pisati. In obratno, ko hočemo iz njega brati, so omogočeni izhodni ojačevalniki naslovljenega stolpca. Vrsto dostopa, branje ali pisanje, določata signala *read* in *write*, ki se medsebojno izključujeta. Preko para naslovljenih vrat AND aktivirata vhodne, ali izhodne ojačevalnike. Signala *read* in *write* odslikavata nadzorni signal $r/\overline{w}$ ($r/\overline{w} = 1 \Rightarrow read = 1$, $write = 0$,

ter $r/\overline{w} = 0 \Rightarrow read = 0$, $write = 1$). Vendar je pogoj za visoko stanje signala *read*, ali *write*, nadzorni signal *clk*. To pomeni, da je branje iz, ali pisanje v register dovoljeno le ob nizkem stanju urinega signala *clk*. Ob prehodu v visoko stanje se branje, ali pisanje prekine. Naraščajoča fronta signala *clk* dogajanje v pomnilniku zamrzne. V tem trenutku se stanje na podatkovnem vodilu shrani v register pomnilnika (pisanje), oziroma stanje registra se preneha preslikovati na podatkovno vodilo (branje).

Poglavje 3

Preprost operacijski sistem v realnem času

Programska oprema industrijskih mikrokrmilniških sistemov se bistveno razlikuje od običajnih časovno neodvisnih programov, kot jih poznamo z osebnih računalnikov. Za programe, ki tečejo na osebnih računalnikih, si navadno želimo, da so čim hitrejši. Njihova časovna programska sled ni določena. Edino kar si želimo je, da bi računalnik vse naloge opravil kar najhitreje, še najbolje neskončno hitro, v času nič. Časovna programska sled programov v računalniških krmilnih sistemih pa je natančno opredeljena. Za takšne programe pravimo, da potekajo v realnem času. S tem mislimo predvsem na časovno uskladitev z večimi zunanjimi procesi. Za primer navedimo digitalen telefonski aparat. Mikrokrmilnik mora pri izbiranju telefonske številke proizvesti vlak impulzov, katerih širina je natanko 50ms. Na prvi pogled to ni zahtevna naloga, saj lahko v program vedno vgradimo zakasnilne zanke, ki povzročijo, da mine določen čas. Težave nastopijo v trenutku, ko se zavemo, da mora mikrokrmilnik, medtem ko pošilja impulze, še sprejemati in shranjevati vse številke, ki prihajajo s tipkovnice.

Programska oprema mora torej hkrati in sproti skrbeti za večje število opravil. Veliki industrijski računalniki so v ta namen opremljeni s posebnimi operacijskimi sistemi, ki uporabniku močno olajšajo programiranje v realnem času. Taka sistemska podpora je zelo učinkovita, vendar izredno zapletena in tako

za majhne sisteme večkrat neprimerna. V veliko primerih ne potrebujemo vse funkcionalnosti pravega prednostnega večopravilnega sistema v realnem času.

Osnovni princip je sila preprost in naraven. Opazujmo gospodinjo, ki navidezno hkrati kuha juho in golaž, lika perilo in pazi na otroka, ne da bi karkoli vedela o hkratnem in sprotnem procesiranju. Juha v loncu zahteva zelo malo pozornosti: vsake pol ure je potrebno doliti malo vode ali dodati kako začimbo. Golaž je že bolj zahteven, saj se utegne prismoditi. Zato ga mora gospodinja pomešati vsakih deset minut. Otrok se sam zabava, vendar potrebuje vsake toliko časa nekaj pozornosti in pomoči. Seveda gospodinja ne more dobesedno hkrati opravljati vseh teh opravil, pač pa je dovolj marljiva, da uspe svojo pozornost časovno smotrno razdeliti med vse procese. Tako ji celo ostanejo časovni intervali, ki jih koristno zapolni z likanjem perila. Naša gospodinja torej na osnovi svoje marljivosti in intuicije uspeva hkrati in časovno usklajeno skrbeti za štiri procese.

Mikrokrmilnik je v marsičem zelo podoben gospodinji. V primerjavi z zunanjimi enotami je po svoji strojni naravi zelo hiter (marljiv). Za inteligentno razporejanje njegove pozornosti med različnimi opravili pa poskrbi ustrezna sistemska programska oprema, ki stoji v središču naše pozornosti. V tem poglavju bomo spoznali jedro miniaturnega operacijskega sistema za hkratno in sprotno programiranje, ter tako vstopili v svet operacijskih sistemov v realnem času.

3.1 Časovno rezinjenje

Večopravilnost operacijskega sistema pomeni, da se navidezno izvaja več opravil (podprogramov) hkrati. V resnici gre za časovno rezinjenje, ki je temelj vsake sistematične obdelave večjega števila časovno vzporednih dogodkov. Časovna os se razreže na posamezne intervale, v katerih se procesor posveča različnim opravilom.

Vsako opravilo se izvaja določen čas, nakar je na vrsti naslednje opravilo. Nekatera opravila kliče operacijski sistem (sinhrona opravila), druga opravila sama opozorijo nase (asinhrona opravila ali prekinitve). Termin realni čas dobi v tem kontekstu bolj točno definicijo in pomeni, da operacijski sistem zagotavlja, da bo vsako sinhrono opravilo v določenem časovnem intervalu na vrsti najmanj enkrat. Časovni interval je lahko za posamezna sinhrona opravila različen. Poleg tega mora operacijski sistem v realnem času zagotavljati, da bo vsaka prekinitev

izvedena najkasneje v predpisanem roku. Rok je zopet lahko za vsako prekinitev drugačen.

Če za trenutek pustimo prekinitve ob strani, potem operacijski sistem kliče, oziroma izvaja opravila (podprograme). Postavi se vprašanje, kakšen naj bo kriterij, po katerem se posameznim opravilom dodeljujejo časovne rezine. Operacijski sistemi za delo v realnem času vsebujejo v svojem najožjem jedru razvrščevalnik, ki je odgovoren za logistiko časovnega rezinjenja. Naš razvrščevalnik naj bo kar se da preprost in naj opravila, ki jih ima našteta v urniku, kliče enostavno enega za drugim, brez upoštevanja morebitnih prioritetnih pravil. Urnik opravil je v našem primeru tabela z naslovi začetkov njihove kode. Vse podprograme navedemo v urniku, in operacijski sistem jih bo samodejno klical.

Razvrščevalnik bo zagotavljal, da bo vsako opravilo klicano enkrat v določenem časovnem intervalu, torej bo naš operacijski sistem deloval v realnem času. Pri načrtovanju bomo privzeli naslednje štiri poenostavitve.

1. Vse časovne rezine so natanko enako velike. Operacijski sistem tako dolžine časovnih rezin ne zna prilagojevati glede na zahtevane roke, oziroma časovne intervale opravil.

2. Vsa opravila se vedno zaključijo še pred iztekom časovne rezine. Tako se mora vsako opravilo, ki ga hočemo uvrstiti v urnik, v najslabšem primeru končati prej kot v dolžini ene časovne rezine (operacijski sistem oziroma razvrščevalnik v vsaki časovni rezini porabi nekaj časa zase, tako da se mora opravilo v resnici končati še nekaj prej). Operacijski sistem opravila ob izteku časovne rezine ne zna prekiniti, in nato z njim nadaljevati ob naslednjem klicu.

3. Urnik opravil obsega vnaprej določeno konstantno število opravil, ki se med delovanjem ne spreminja. Opravila se izvajajo ciklično. To določa časovni interval Δt v katerem je vsako opravilo točno enkrat na vrsti. Le ta je za vsa opravila enak in znaša: Δt = število opravil $\times$ dolžina časovne rezine Δt_{slice}.

4. Zunanje enote ne povzročajo prekinitev mikrokrmilnika. Operacijski sistem asinhronih opravil, oziroma prekinitev, ne pozna. Pozna le sinhrona opravila, ki jih sam kliče.

S stališča velikih sistemov so te štiri omejitve zelo radikalne, vendar na ta način pridemo do izredno kompaktnega nadzornega programa, ki je primeren tudi za najmanjše mikrokrmilniške sisteme. Naš razvrščevalnik je pregleden in

razumljiv tudi za manj izkušene (navdušene) študente. Kljub temu je praktično uporaben in v določenih primerih celo zelo učinkovit.

Sledi del izvorne kode za centralno procesno jedro ARM7, ki podaja urnik opravil in ravrščevalnik z danimi omejitvami:

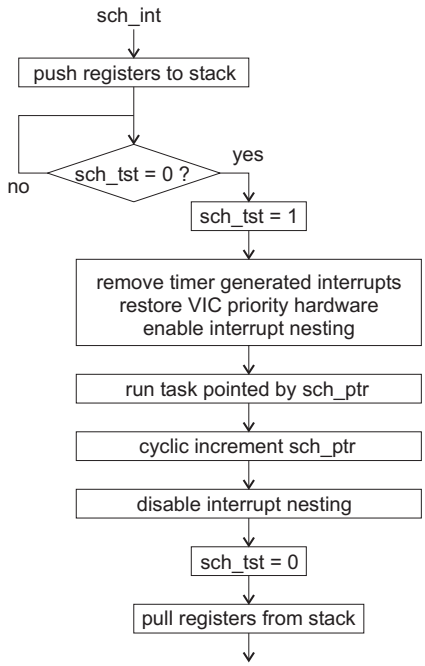
```
/* Constants */
            .equ    i,                      0x80
            .equ    t0mr0_int,              0x01
            .equ    word_len,               0x04
            .equ    rtos_active,            0x01
            .equ    rtos_inactive,          0x00
/* Registers */
            .equ    t0ir,                   0xe0004000
            .equ    vicvectaddr,            0xfffff030
/* Global symbols */
            .global task1
            .global task2
            .global task3

            .code   32

/* Uninitialised variables */
            .bss
            .lcomm  sch_tst,                4
            .lcomm  sch_ptr,                4

/* Initialised data */
            .data
sch_tab:    .long   task1
            .long   task2
sch_tab_end: .long  task3

/* Program code */
            .text
/* Real time operating system core */
sch_int:    stmfd   sp!, {r0-r5, lr}
            ldr     r0, =sch_tst
            ldr     r1, [r0]
```

```
                tst     r1, #rtos_active
                beq     sch_int_ok
sch_int_err:    b       sch_int_err
sch_int_ok:     mov     r1, #rtos_active
                str     r1, [r0]
                ldr     r1, =t0ir
                mov     r2, #t0mr0_int
                str     r2, [r1]
                ldr     r1, =vicvectaddr
                str     r0, [r1]
                mrs     r1, cpsr
                bic     r1, r1, #i
                msr     cpsr_c, r1
                ldr     lr, =task_end
                ldr     r2, =sch_ptr
                ldr     r3, [r2]
                ldr     pc, [r3]
task_end:       ldr     r4, =sch_tab_end
                teq     r3, r4
                beq     to_first
                add     r3, r3, #word_len
                b       save_ptr
to_first:       ldr     r3, =sch_tab
save_ptr:       str     r3, [r2]
                orr     r1, r1, #i
                msr     cpsr_c, r1
                mov     r1, #rtos_inactive
                str     r1, [r0]
                ldmfd   sp!, {r0-r5, lr}
                mov     pc, lr
```

Podrobnejšo razlago navodil prevajalniku je moč najti v dodatku A ali v [4], postavitev nadzornika prekinitev in časovnika v dodatkih B.4 in B.5 ali v [5], ter kratek opis registrov procesnega jedra in nabor zbirniških ukazov za arhitekturo ARM v dodatku C ali v [6].

Konstantna podatkovna struktura imenovana urnik opravil se nahaja med oznakama *sch_tab* in *sch_tab_end*, ki označujeta kazalca na prvo in zadnje opravilo. Tako je lahko v našem urniku poljubno število opravil. Urnik je sestavljen iz polja naslovov, ki označujejo začetke posameznih opravil. Opravila

sama pa so pravzaprav navadni podprogrami, oziroma funkcije, ki se podrejajo omejitvi 2.



Slika 3.1: Algoritem razvrščevalnika opravil *sch_ int*

Razvrščevalnik *sch_ int* je odgovoren za izvajanje časovnih rezin. Klican je v enakomernih časovnih intervalih kot prekinitev prožena s pomočjo časovnika.

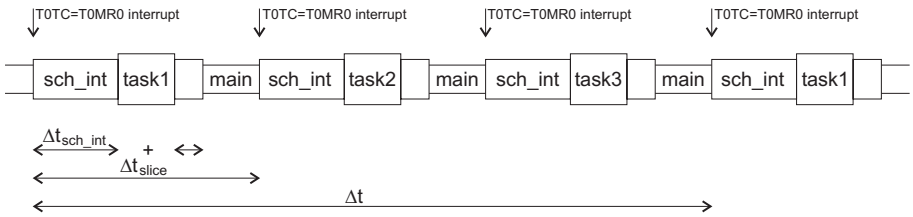
To je hkrati tudi edina prekinitev v našem sistemu in služi za poganjanje samega jedra našega preprostega operacijskega sistema. Ostale prekinitve, ki bi jih povzročal kakršenkoli drug izvor, smo v omejitvi 4 prepovedali. Načelen algoritem razvrščevalnika opravil podaja slika 3.1.

Ob prekinitvi mehanizem vektorskega nadzornika prekinitev (glej izvorno kodo na strani 109) poskrbi, da se požene razvrščevalnik *sch_int*. Razvrščevalnik najprej poskrbi za vsebino registrov, ki jih bo uporabljal. Shrani jih na sklad, tako da jih lahko pred koncem zopet postavi v prvotno stanje. Nato s pomočjo spremenljivke *sch_tst* preveri, ali morda predhodno opravilo še ni končano. Če je temu tako, se ujame v neskončni zanki *sch_int_err*, v kateri program obstane, dokler ga uporabnik ne prekine. V nasprotnem primeru v spremenljivki *sch_tst* označi pričetek novega opravila. Sledi koda, ki poskrbi za naslednjo prekinitev. In sicer je umaknjena zahteva po prekinitvi, ki jo je pravkar podal časovnik (register *T0IR*), prioritetno vezje nadzornika prekinitev pa je postavljeno v začetno stanje (register *VICVectAddr*). Ker sta ostali konstanti v registrih *T0PR* in *T0MR0* nespremenjeni, števnika časovnika *T0PC* in *T0TC* pa sta bila ob prekinitvi postavljena na nič (glej izvorno kodo na strani 116), se naslednja prekinitev sproži po vedno natanko enakem časovnem intervalu. Da se naslednja prekinitev lahko sproži tudi v primeru predolgega opravila, torej preden se trenutna prekinitev konča, je omogočeno gnezdenje prekinitev. Gnezdenje prekinitev (to je nova prekinitev znotraj trenutne prekinitve) je dovoljeno z umikom zastavice I v registru stanj (glej dodatek C.2). Nato razvrščevalnik s pomočjo kazalca *sch_ptr*, ki določa, katero opravilo v urniku je na vrsti, le to požene. Ko opravilo konča s svojim tekom, se vrne na mesto *task_end*, kjer razvrščevalnik krožno poveča kazalec *sch_ptr*. Kazalec je tako pripravljen za naslednjo časovno rezino in kaže na naslednje opravilo. Na koncu je v spremenljivki *sch_tst* označen konec opravila.

Ob vsaki prekinitvi porabi razvrščevalnik, preden pokliče tekoče opravilo, in potem, ko je opravilo končano, nekaj časa zase. To je cena za časovno rezinjenje izražena v procesorskem času, oziroma režijski stroški. Tudi naša gospodinja izgubi nekaj časa, ko odloži likalnik, naredi nekaj korakov do štedilnika in vzame kuhalnico. Šele tedaj prične z dejanskim opravilom (mešanjem golaža). Prav tako porabi nekaj časa, da se vrne h glavnemu programu (likanju). Zelo pomembno je, da je izgubljen čas kratek v primerjavi s trajanjem opravil. Čas Δt_{sch_int}, ki ga razvrščevalnik porabi zase, je konstanten, ne glede na dolžino časovne rezine. Zato je tudi odstotek časa, ki ostane na voljo, odvisen od dol-

žine časovne rezine. Daljša kot je časovna rezina Δt_{slice}, manjši je delež režije η, oziroma boljši je izkoristek.

$$\eta = \frac{\Delta t_{sch\_int}}{\Delta t_{slice}} \tag{3.1}$$



Slika 3.2: Načelo časovnega rezinjenja

Povzemimo delovanje nadzornega programa z grafično predstavitvijo na sliki 3.2. Izvajanje glavnega programa se prekine v trenutku, ko je *T0TC* enak *T0MR0*. Zažene se razvrščevalnik *sch_int*, ki potrebuje nekaj časa zase, nakar pokliče ustrezno opravilo. V našem primeru na sliki 3.2 je na vrsti opravilo *task1*. Po končanem opravilu potrebuje razvrščevalnik še nekaj procesorskega časa, nakar se nadaljuje izvajanje glavnega programa. Ob naslednji prekinitvi se vse skupaj ponovi.

Dolžino časovne rezine Δt_{slice} določamo z nastavitvami časovnika. Podrobnejšo razlago lahko bralec najde v dodatku B.5 na strani 118. Čas Δt_{sch_int}, ki ga razvrščevalnik porabi zase, pa moramo izmeriti. Več o merjenju časa bomo povedali v poglavju 3.2. Naj na tem mestu le omenimo, da za razvrščevalnik s strani 48 velja $\Delta t_{sch_int} \approx 11\mu$s pri urinem signalu $f_{clk} = 12$MHz, oziroma približno 130 strojnih ciklov.

Seveda je mogoče ekvivalenten razvrščevalnik *sch_int*, oziroma jedro operacijskega sistema, napisati tudi v programskem jeziku C [14] in [15]. Razvrščevalnik *sch_int()* postane funkcija, ki ne sprejme nobenega argumenta in tudi ničesar ne vrne. Urnik opravil je sedaj polje *sch_tab* s kazalci na funkcije, ki opravila predstavljajo. Namesto kazalca na tekoče opravilo *sch_ptr* je uporabljen indeks *sch_idx* opravila v polju *sch_tab*. V programskem jeziku C ne moremo neposredno spreminjati zastavice *I*, zato sta dodani funkciji *ena-*

ble_irq() in *disable_irq()*. Funkciji sta pravzaprav le ovojnici za nekaj vrstic zbirniške kode.

```
#define mr0_interrupt 0x00000001
// System status
#define task_completed 0
#define task_running   1
// Registers
#define T0IR        (*((volatile unsigned long *)0xe0004000))
#define VICVectAddr (*((volatile unsigned long *)0xfffff030))

typedef void (* voidfuncptr)();
// System variables
int sch_tst, sch_idx;
// Scheduler
extern void task1();
extern void task2();
extern void task3();
voidfuncptr sch_tab[] = {task1, task2, task3};

// Enable IRQ interrupts
void enable_irq() {
    asm("stmfd sp!, {r0}");
    asm("mrs   r0, cpsr");
    asm("bic   r0, r0, #0x80");
    asm("msr   cpsr_c, r0");
    asm("ldmfd sp!, {r0}");
}

// Disable IRQ interrupts
void disable_irq() {
    asm("stmfd sp!, {r0}");
    asm("mrs   r0, cpsr");
    asm("orr   r0, r0, #0x80");
    asm("msr   cpsr_c, r0");
    asm("ldmfd sp!, {r0}");
}
```

```
// Real time operating system core
void sch_int() {
    if(sch_tst == task_running) while(1);
    sch_tst = task_running;
    T0IR = mr0_interrupt;
    VICVectAddr = 0;
    enable_irq();
    (*(sch_tab[sch_idx]))();
    sch_idx = sch_idx + 1;
    if(sch_idx == sizeof(sch_tab) / sizeof(voidfuncptr))
        sch_idx = 0;
    disable_irq();
    sch_tst = task_completed;
}
```

Čas Δt_{sch_int}, ki ga razvrščevalnik porabi zase, je zopet potrebno izmeriti. Ker je razvrščevalnik sedaj napisan v programskem jeziku C, je njegova dolžina odvisna tudi od zmogljivosti in nastavitev prevajalnika. Tako lahko z različnimi prevajalniki, oziroma različnimi nastavitvami dobimo bistveno različne čase Δt_{sch_int}.

Na tem mestu moramo omeniti še zagon nadzornega programa. Načeloma pomeni zagon le začetno nastavitev spremenljivk *sch_tst* in *sch_ptr* ter registrov nadzornika prekinitev in časovnika. Vse potrebno postori podprogram *sch_on*.

```
/* Constants */
        .equ    cnt_start,          0x01
        .equ    rtos_inactive,      0x00
/* Register */
        .equ    t0tcr,              0xe0004004
/* Global symbol */
        .global task1

        .code   32

/* Uninitialised variables */
        .bss
        .lcomm  sch_tst,            4
```

```
        .lcomm  sch_ptr,          4

/* Initialised data */
        .data
sch_tab: .long   task1

/* Program code */
        .text
/* Start of real time operating system */
sch_on:  stmfd   sp!, {r0-r1, lr}
         ldr     r0, =sch_tst
         mov     r1, #rtos_inactive
         str     r1, [r0]
         ldr     r0, =sch_ptr
         ldr     r1, =sch_tab
         str     r1, [r0]
         bl      timer0_init
         bl      timer0_int
         ldr     r0, =t0tcr
         mov     r1, #cnt_start
         str     r1, [r0]
         ldmfd   sp!, {r0-r1, lr}
         mov     pc, lr
```

Podprogram *sch_on* zopet najprej poskrbi za vsebino registrov, ki jih bo uporabljal. Shrani jih na sklad, tako da jih lahko na koncu postavi v prvotno stanje. Po nastavitvi spremenljivke *sch_tst* in kazalca *sch_ptr* na njuni začetni vrednosti, inicializira še časovnik (*timer0_init* stran 116) in nadzornik prekinitev (*timer0_int* stran 109). Nato časovnik lahko prične s štetjem (umaknemo bit1 v registru *T0TCR*).

Za različico v programskem jeziku C podajmo še ekvivalentno funkcijo *sch_on()*, ki postori iste stvari. Inicializacija časovnika se izvrši v funkciji *ti-*

mer0_init() (stran 118), nadzornika prekinitev pa v funkciji *vic_int()* (stran 111).

```
#define counter_enable 0x00000001
// System status
#define task_completed 0
// Register
#define T0TCR (*((volatile unsigned long *)0xe0004004))

typedef void (* voidfuncptr)();
// System variables
int sch_tst, sch_idx;

extern void timer0_init(int, int *, int, int);
extern void vic_init(int, int, voidfuncptr *, int *,
    voidfuncptr);

// Start of real time operating system
// prescale  ... maximum prescale counter value
// match     ... match value
void sch_on(int prescale, int match) {
    int matches[4] = {match, 0, 0, 0}, interrupt[16] =
        {timer0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    voidfuncptr function[16] =
        {sch_int, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    sch_tst = task_completed;
    sch_idx = 0;
    timer0_init(prescale, matches, mr0i | mr0r, timer);
    vic_init(0, timer0, function, interrupt, 0);
    T0TCR = counter_enable;
}
```

3.2 Merjenje dolžine posameznega opravila

Zaradi omejitve 2 v poglavju 3.1 se mora posamezno opravilo zaključiti pred iztekom časovne rezine, oziroma pred pričetkom naslednje rezine. Zato je potrebno vsakemu opravilu določiti njegovo dolžino in to uskladiti z dolžino časovne rezine. V principu je to enostavna naloga. Najprej določimo najdaljšo programsko

pot skozi podprogram. To pomeni, da ob vseh vejitvah v algoritmu izberemo tisto možnost, ki ima za posledico daljšo pot do zaključitve podprograma. Nato le preštejemo strojne cikle, ki jih procesorsko jedro potrebuje za izvršitev vseh zbirniških ukazov na najdaljši programski poti. Rezultat je število strojnih ciklov, ki jih opravilo potrebuje v najslabšem primeru, oziroma dolžina opravila, ki mora biti nujno krajša od časovne rezine.

V centralnem procesnem jedru ARM7 naj bi vsak zbirniški ukaz za svojo izvršitev potreboval en strojni cikel. Tako bi moralo biti merjenje dolžine nekega podprograma še posebej enostavno. Le preštejemo zbirniške ukaze, ki se izvršijo od začetka do konca najdaljše programske poti skozi podprogram. Dobiti bi morali število ciklov, ki jih podprogram potrebuje v najslabšem primeru. Vendar temu zaradi nedeterminističnega delovanja ni vedno tako (glej dodatka C.1 in B.3). Cevovodna arhitektura optimalno deluje le na linearni programski kodi, težavo pa predstavljajo tudi relativno dolgi dostopni časi, ki jih pomnilniški pospeševalnik ne uspe zanesljivo nevtralizirati. Posledica tega je, da mora procesno jedro včasih čakati, kar seveda znižuje hitrost. Za en zbirniški ukaz en cikel ni vedno dovolj. Zaradi nedeterminističnosti tudi ni nujno, da bo isti podprogram ob različnih klicih potreboval natanko enako število ciklov. Skratka dolžino podprograma je mogoče le oceniti.

Če hočemo določiti dolžino opravila, ki je napisano kot funkcija v programskem jeziku C, imamo še dodatno težavo. In sicer ne poznamo zbirniške kode opravila, ki zopet ni enolično določena. Kako se koda, napisana v programskem jeziku C, prevede v zbirnik, je odvisno od prevajalnika. Tako bi morali za oceno dolžine funkcije le to najprej prevesti, in nato prešteti zbirniške ukaze. Vendar to ni ravno enostavna naloga, kajti zbirniška koda generirana s prevajalnikom, ni ravno lahko berljiva.

Namesto tega dolžino podprogramov, oziroma funkcij enostavno izmerimo. Enega izmed časovnikov postavimo v prosti tek (glej kodo na strani 119). Tik pred klicem podprograma časovnik sprožimo, ter ga takoj po vrnitvi ustavimo. Iz registrov časovnika je nato mogoče razbrati dolžino klicanega podprograma izraženo s številom ciklov. Pri tem moramo seveda poskrbeti, da se podprogram

odvije po najdaljši programski poti. Sledi primer zbirniške kode za centralno procesno jedro ARM7, ki uporablja časovnik Timer1.

```
/* Constants */
   .equ  cnt_start,       0x01
   .equ  cnt_stop,        0x00
/* Register */
   .equ  t1tcr,           0xe0008004

   .code 32

/* Program code */
   .text

     ...
   ldr     r0, =t1tcr
   mov     r1, #cnt_start
   mov     r2, #cnt_stop
   str     r1, [r0]
   bl      subroutine
   str     r2, [r0]
     ...
```

Koda izmeri trajanje podprograma *subroutine*. V resnici je izmerjena dolžina nekoliko daljša, saj je vštet tudi klic podprograma. Rezultat se nahaja v registrih *T1PC* in *T1TC*. In sicer je dolžina podprograma enaka $(T1PC+1)\times(T1TC+1)$

ciklov vodila VPB (glej dodatek B.2). Ekvivalentna koda bi v programskem jeziku C izgledala takole:

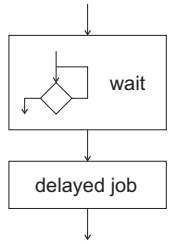
```
#define counter_start 0x00000001
#define counter_stop  0x00000000
// Register
#define T1TCR (*((volatile unsigned long *)0xe0008004))

  ...
T1TCR = counter_start;
subroutine();
T1TCR = counter_stop;
  ...
```

Na podoben način je seveda mogoče izmeriti dolžino jedra operacijskega sistema Δt_{sch_int}. S tem lahko določimo izkoristek operacijskega sistema η (enačba (3.1)) pri določeni dolžini časovne rezine Δt_{slice}.

3.3 Zakasnitve

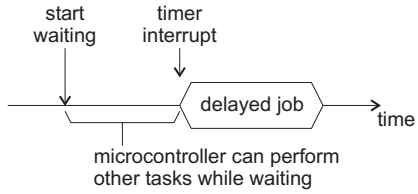
Mnogokrat pri programiranju mikrokrmilnikov naletimo na zahtevo, naj se neka naloga izvrši po preteku določenega časovnega intervala. Z drugimi besedami, na pričetek izvajanja naloge je potrebno malce počakati. Na voljo imamo dve intuitivni rešitvi. Prva najenostavnejša je, da mikrokrmilnik enostavno počaka, da zahtevan časovni interval mine, nakar nadaljuje z delom. Razmere prikazuje slika 3.3. Čakanje je navadno izvedeno v zanki, katere edini namen je, da mine

čas. Slaba lastnost takšne rešitve je v tem, da mikrokrmilnik, medtem ko čaka, ne more delati nič drugega. S čakanjem je polno zaposlen.



Slika 3.3: Zakasnitev s čakanjem

Drugi pristop je, da nas na iztek časovnega intervala opozori časovnik. Mikrokrmilnik lahko nemoteno nadaljuje z delom, saj eksplicitno čakanje v zanki ni potrebno. Ko zahtevani čas poteče, časovnik sproži prekinitev, kjer na izvršitev čaka naša naloga. Razmere poskuša v časovnem prostoru ponazoriti slika 3.4.

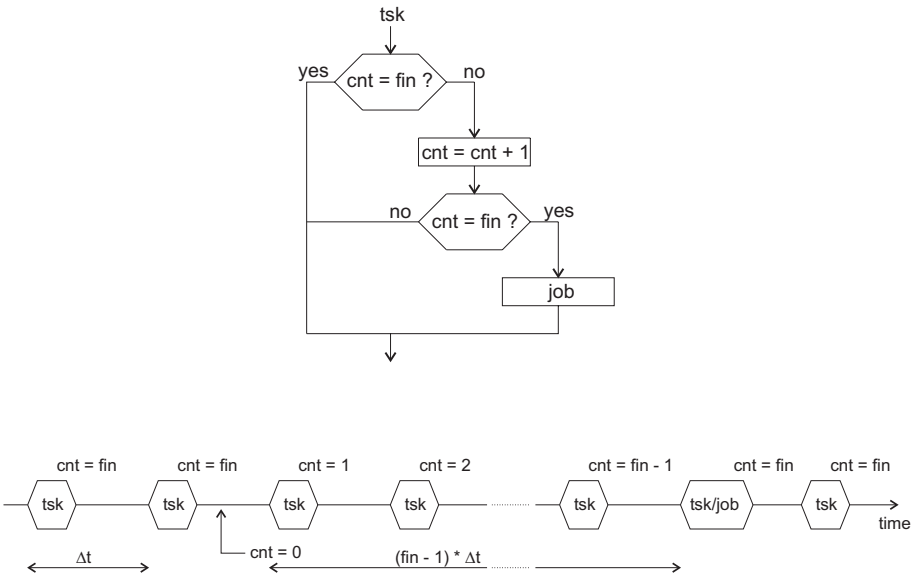


Slika 3.4: Zakasnitev s prekinitvijo

Predpostavimo, da je časovni interval, oziroma zakasnitev, mnogo večja od časovne rezine Δt_{slice} našega operacijskega sistema. Kako bi zakasnitev realizirali znotraj opravila? Niti prva, niti druga rešitev nista primerni. V prvem primeru postane zaradi čakanja opravilo predolgo, saj se ne konča, preden časovni interval ne mine. V drugem primeru pa je uporabljena prekinitev, ki jo proži časovnik, česar naš preprost operacijski sistem zopet ne dovoljuje (alinea 4 v poglavju 3.1).

Glede na nastavitve operacijskega sistema (število opravil in dolžina časovne rezine Δt_{slice}) je časovni interval Δt, v katerem je opravilo na vrsti točno enkrat,

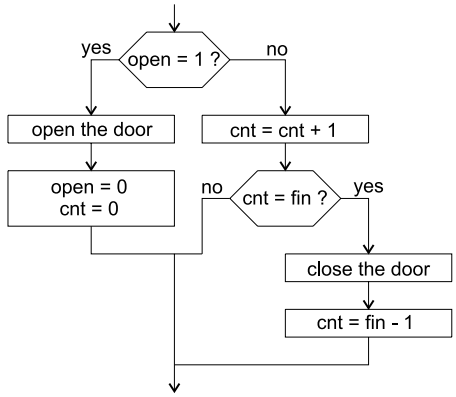
znan (alinea 3 v poglavju 3.1). Tako je tudi število klicev opravila, ki so potrebni, da zakasnitev mine, vnaprej znano. Opravilo šteje kolikokrat je bilo na vrsti, s čimer v bistvu meri čas. Ko čas zakasnitve poteče, opravilo izvrši nalogo. Zakasnitev je v tem primeru lahko realizirana na Δt natančno. Interval Δt predstavlja časovni kvant, oziroma enoto, v kateri opravilo meri čas.



Slika 3.5: Zakasnitev z opravilom

V algoritmu opravila *tsk* na sliki 3.5 nadzira čas spremenljivka *cnt*, ki ima privzeto vrednost *fin*. Na mestu, kjer se pojavi potreba po zakasnjeni izvršitvi naloge *job*, je spremenljivka *cnt* postavljena na nič, kar se zgodi zunaj opravila *tsk*. S tem opravilo začne meriti čas. Ko zakasnitev, ki je enaka najmanj $(fin + 1) \times \Delta t$ in ne več kot $fin \times \Delta t$, poteče, se naloga *job* izvrši. Opravilo do naslednje

zahteve po zakasnjeni izvršitvi naloge *job* miruje, saj ima spremenljivka *cnt* privzeto vrednost *fin*.



Slika 3.6: Opravilo za odpiranje in zapiranje vrat dvigala

Za primer si oglejmo opravilo, ki v dvigalu skrbi za odpiranje in zapiranje vrat. Algoritem opravila prikazuje slika 3.6. Znak opravilu, naj se vrata dvigala odprejo, je postavljena spremenljivka *open*. Vsakič, ko opravilo znak sprejme, prične z odpiranjem vrat in hkrati z merjenjem časa. Po preteku določenega časovnega intervala, se vrata zaprejo. Čas odpiranja je vštet v časovni interval. Če pride do nove zahteve po odpiranju vrat pred iztekom časovnega intervala, se štetje časa prične znova. Poleg tega se vrata v primeru, da pride do zahteve med zapiranjem, takoj zopet odprejo. Vrata dvigala premika motor, ki ga opravilo le požene v eno ali drugo smer. Predpostavljeno je, da za zaustavitev motorja v skrajnih legah poskrbijo končna stikala. Če končnih stikal ni, moramo za izklop motorja ob zaznavi skrajne lege poskrbeti v programski opremi.

3.4 Hkratni dostop do skupnih enot

Vedno, kadar imamo opravka s hkratnim izvajanjem večih opravil, se pojavita dve poglavitni težavi, ki neposredno sledita iz interakcij med opravili. To sta

problem hkratnega dostopa in problem usklajene komunikacije. V tem razdelku se bomo lotili prvega.

Situacijo vsi dobro poznamo iz vsakdanjega življenja. Dva olikana študenta želita v tretjem nadstropju izstopiti iz dvigala. Hkrati naredita korak proti vratom, vendar so le-ta preozka za oba. Zopet oba hkrati opazita namero drugega in drug drugemu odstopita prednost. Če nastala pat pozicija traja predolgo, se vrata zapro in dvigalo odpelje.

Dva neodvisna procesa (študenta) skušata hkrati uporabiti neko skupno napravo (vrata). Pri tem sta se prisiljena sporazumeti o tem, kdo bo šel prvi in kdo kot drugi skozi vrata. Rešitev je vedno v tem, da mora nekdo za hipec počakati!

Potrebno je torej:

1. prepoznati situacijo, ko več procesov hkrati želi dostop do neke skupne enote in

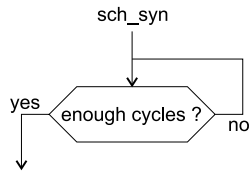
2. določiti vrstni red dostopa.

Veliki sistemi poznajo najrazličnejše arbitražne tehnike, ki so lahko tudi zelo zapletene. V našem primeru si lahko oddahnemo, saj imamo tako preprost model časovnega rezinjenja, da sploh ne more priti do zelo zapletenih konfliktov pri dostopu do skupnih naprav!

S pomočjo časovnega rezinjenja lahko v našem operacijskem sistemu navidezno hkrati teče poljubno število opravil in glavni program. Glavni program ima najnižjo prioriteto in teče le tisti čas, ko je določeno opravilo že končano, naslednje pa še ni na vrsti. Vsako opravilo in seveda tudi glavni program lahko uporablja skupne enote, ki so v mikrokrmilniškem sistemu na voljo (npr. pomnilnik RAM, vzporedna vrata ...). Če nobene skupne enote ne uporablja več kot eno opravilo, oziroma glavni program, potem so vsa opravila in glavni program med seboj neodvisni. To pomeni, da živijo povsem ločeno in med seboj ne komunicirajo. Problema hkratnega dostopa v tem primeru ni.

Takoj, ko si opravila in glavni program začno med seboj izmenjevati informacije, trčimo v najbolj preprost in tudi najbolj pogost primer hkratnega dostopa

do vsebine shranjene v pomnilniku RAM. Primer: glavni program želi prebrati tabelo desetih 32-bitnih števil. Ko jih prebere pet, ga prekine eno izmed opravil in tabelo osveži. Po koncu opravila glavni program nadaljuje z branjem drugega dela tabele, ki vsebuje osvežena števila. Tako so podatki, ki jih glavni program prebere, nekonsistentni.



Slika 3.7: Algoritem podprograma za sinhronizacijo z razvrščevalnikom

V našem primeru zaradi omejitve 2 v poglavju 3.1 do skupne enote ne moreta dostopati dve opravili hkrati, saj se nikdar ne zgodi, da bi eno opravilo prekinilo drugo. Morebitni hkratni dostop do skupne enote se lahko zgodi le v primeru, ko enoto uporablja glavni program, pri čemer ga prekine opravilo, ki prav tako uporablja isto skupno enoto. V teh primerih moramo zagotoviti, da glavni program med rabo skupne enote ne bo prekinjen. To naredimo tako, da pred kritičnim delom kode v glavnem programu pokličemo podprogram *sch_syn*, katerega algoritem prikazuje slika 3.7. Podprogram *sch_syn* ustavi izvajanje glavnega programa, dokler ni do začetka naslednje časovne rezine na voljo dovolj strojnih ciklov. Glavni program ima najnižjo prioriteto, zato čaka, da je za kritični del kode na voljo dovolj procesorskega časa. Prepoznavanje konfliktnih situacij smo preložili na programerja. Slednji mora pri načrtovanju glavnega programa predvideti mesta potencialne nevarnosti in jih ustrezno zavarovati s

klicem podprograma *sch_syn*. Sledi izvorna koda podprograma *sch_syn* za centralno procesno jedro ARM7.

```
/* Registers */
            .equ  t0tc,         0xe0004008
            .equ  t0mr0,        0xe0004018

            .code 32

/* Program code */
            .text
/* Synchronise with scheduler subroutine */
sch_syn:    stmfd sp!, {r1-r3}
            ldr   r1, =t0mr0
            ldr   r2, =t0tc
            ldr   r1, [r1]
            subs  r1, r1, r0
sch_syn_err: bls  sch_syn_err
wait:       ldr   r3, [r2]
            subs  r3, r1, r3
            bls   wait
            ldmfd sp!, {r1-r3}
            mov   pc, lr
```

Podprogram *sch_syn* je zelo preprost in učinkovit. Pokličemo ga neposredno pred vsakim kritičnim odsekom v glavnem programu. Pred klicem v delovni register *r0* vpišemo dolžino kritičnega odseka, ki jo prej seveda izmerimo (glej poglavje 3.2). V register *r0* vpišemo takšno konstanto, da je dolžina kritičnega odseka manjša od $r0 \times (T0PR + 1)$ ciklov vodila VPB (glej dodatek B.5). Ker lahko dolžino kode le ocenimo, in ker bi bilo potrebno kritičnemu odseku prišteti še del podprograma *sch_syn* od zadnjega odčitka števnika *T0TC* dalje, je nujno v registru *r0* zahtevati nekaj rezerve. Če je do začetka naslednje časovne rezine premalo časa, potem podprogram *sch_syn* bližajočo se prekinitev prepozna in počaka v zanki *wait*, da le-ta mine. Seveda se lahko zgodi, da *sch_syn* po nepotrebnem čaka prekinitev, ko po urniku sledi opravilo, ki do skupnih enot kritičnega dela kode sploh ne dostopa. Vendar so zavarovana mesta praviloma

zelo kratka v primerjavi z dolžino časovne rezine. Zato čakanje največkrat sploh ni potrebno.

Ekvivalent v programskem jeziku C predstavlja funkcija *sch_int()*. Funkcija prejme dolžino kritičnega odseka kode v argumentu *len*, zopet kot $len \times (T0PR+1)$ ciklov vodila VPB. Za funkcijo veljajo enake lastnosti, kot v zbirniški različici.

```
// Registers
#define T0TC  (*((volatile unsigned long *)0xe0004008))
#define T0MR0 (*((volatile unsigned long *)0xe0004018))

// Synchronise with scheduler
// len ... critical code length = len * prescale_val VPB cycles
void sch_syn(int len) {
    while((int)(T0MR0 - len - T0TC) <= 0);
}
```

V primeru, da je kritični del kode predolg (recimo daljši od časovne rezine), se ne more nikdar izvršiti. Glavni program zamrzne, medtem ko opravila normalno delujejo naprej. V kolikor kritični del kode v glavnem programu ni pravilno zavarovan, lahko pride do napak, ki so največkrat neponovljive in zato zelo težko odpravljive.

3.5 Usklajena komunikacija med opravili

Doslej smo obravnavali le sožitje med različnimi medsebojno neodvisnimi opravili. V tem razdelku se bomo posvetili še vprašanju njihovega medsebojnega sodelovanja. Čeprav je načeloma vsako opravilo, tako v časovnem, kot tudi v funkcijskem smislu, precej neodvisno od ostalega dogajanja v sistemu, morajo obstajati pravila medsebojne komunikacije.

Najenostavneje opravila med seboj komunicirajo preko spremenljivk. Takšna komunikacija navadno poteka časovno neusklajeno. Poglejmo si naslednji primer. Opravilo *tsk* z vrednostjo spremenljivke *stat* sporoča neko stanje (na primer trenutni odčitek temperaturnega senzorja). Vrednost spremenljivke *stat* se osveži vsakič, ko je opravilo *tsk* na vrsti, ne glede na to, ali je prejšnjo vrednost katero izmed preostalih opravil prebralo, ali ne. Opravilo *tsk* predstavlja neke vrste termometer, ki stalno sporoča trenutno temperaturo. Ostalim opravilom je podatek o temperaturi vedno na voljo. Težava nastopi, če bi hoteli izračunati povprečno temperaturo preko časovnega intervala. V tem primeru je potrebno prebrati vse odčitke temperature v tem intervalu, jih sešteti in na koncu deliti

s številom odčitkov. Nikakor se ne sme zgoditi, da opravilo *tsk* vrednost spremenljivke *stat* osveži, preden je prejšnja vrednost prebrana. Komunikacija med oddajnikom (opravilom *tsk*) in sprejemnikom mora biti časovno usklajena.

Ponazorimo si problem s primerom iz vsakodnevnega življenja. Opazujmo dogajanje na blagajni samopostrežne trgovine. Kupci prinašajo blago v košarah in vozičkih ter ga zlagajo na konec tekočega traku pred blagajno. Blagajničarka jemlje posamezne kose blaga z začetka traku in jih registrira strogo po načelu *kdor prej pride, prej melje*. Z vsakim registriranjem se tekoči trak pomakne naprej. Tako sproti nastaja prostor v zadnjem delu. Tekoči trak s čakalno vrsto je potreben zato, ker prispevanje artiklov ni in ne more biti usklajeno z njihovo obdelavo. Pri tem so možne tri različne situacije:

1. Starejša gospa tako počasi zlaga blago na trak, da ga blagajničarka uspeva sproti obdelati, torej je tekoči trak prazen - blagajničarka se dolgočasi.

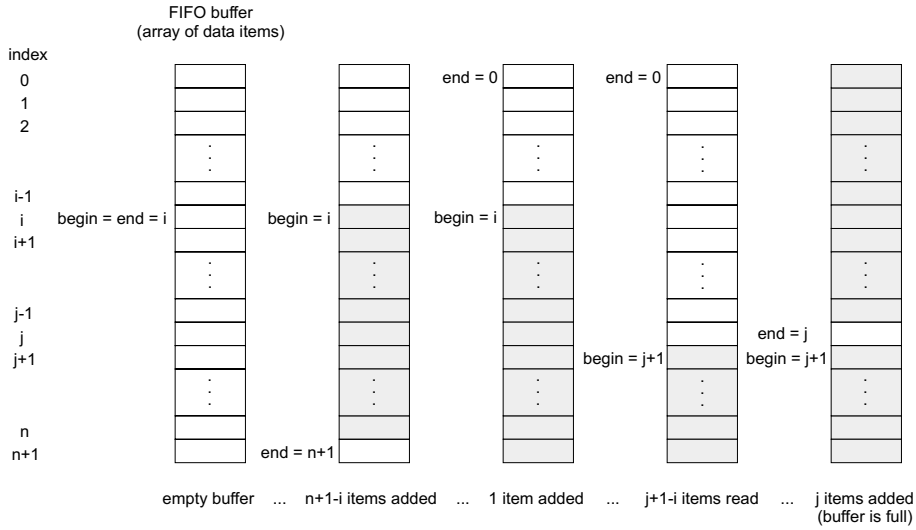
2. Živčna gospodinja pripelje voziček in ga hitro zloži na trak, saj je tam dovolj prostora za vse njeno blago. Seveda blagajničarka takoj začne z delom, čeprav je jasno, da ne more dohajati nestrpne gospodinje.

3. V času prednovoletne nakupovalne mrzlice je obupana žena mobilizirala celo svojega moža, tako da sta skupaj nabrala dva zvrhana vozička blaga. Čeprav se blagajničarka trudi, je trak do konca zapolnjen z blagom. Zakonca ne moreta več polagati blaga v vrsto, torej mora del blaga počakati v vozičku.

Prvi dve situaciji sta normalni, saj je tekoči trak dovolj velik, da sprejme blago povprečnega kupca. Zadnji primer je nenačrtovana izjema, ki pomeni preobremenjen sistem.

V programskem svetu se tak tekoči trak imenuje medpomnilnik (angl. buffer), medtem ko se je v strojnem svetu uveljavil izraz pomikalni register (angl. shift register). Kadar je programska oprema napisana tako, da deluje v realnem času, pa srečamo tudi izraz cevovod (angl. pipeline). Podatkovni tokovi med različnimi opravili so največkrat speljani preko različno velikih programskih medpomnilnikov. Medpomnilnik predstavlja začasno skladišče podatkov, ki so urejeni po času dospelosti. Zapisu v medpomnilnik ne sledi nujno takoj branje iz njega. S tem je časovna komponenta izločena, saj sinhronizacija med oddajnikom in sprejemnikom ni nujna. Zaradi končne velikosti medpomnilnika je potrebno zagotoviti le, da se jemanje iz medpomnilnika zgodi v povprečju najmanj tako pogosto, kot dajanje vanj. Ali z drugimi besedami, sprejemnik

mora biti v povprečju vsaj tako hiter, ali hitrejši, kot oddajnik, oziroma pritok novih podatkov. V našem trgovskem primeru mora biti blagajničarka v povprečju sposobna obdelati več blaga, kot ga kupci prineso. V nasprotnem bi pred blagajno čakala vedno daljša vrsta živčnih gospodinj in obupanih zakoncev, vmes pa bi našli tudi kakšno starejšo gospo. Opisana podatkovna struktura se imenuje medpomnilnik FIFO (angl. First In First Out).

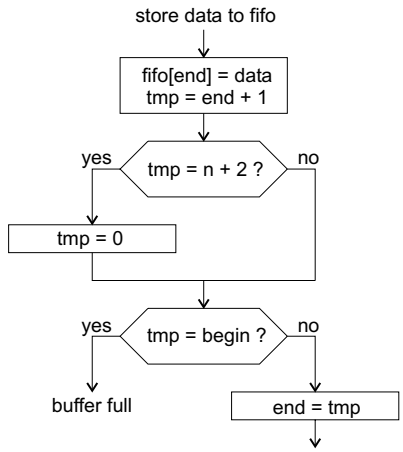


Slika 3.8: Ciklični medpomnilnik FIFO

Poznamo celo vrsto različnih pristopov k programski realizaciji medpomnilnikov. Najpreprostejša nastane, če se neposredno zgledujemo po tekočem traku, vendar je taka realizacija neučinkovita. Podatek v medpomnilnik vedno zapišemo za zadnjega, ob branju pa vzamemo prvega, ter vse ostale premaknemo za eno mesto naprej. Torej vsako branje iz medpomnilnika zahteva pomik vseh ostalih podatkov za eno mesto, kar je zelo zamudno. Poglejmo si raje nekoliko učinkovitejšo rešitev. Medpomnilnik FIFO naredimo kot statični ciklični medpomnilnik, ki ga prikazuje slika 3.8.

Statični ciklični medpomnilnik FIFO predstavlja polje z $n + 1$ elementi. V vsak element polja lahko shranimo en podatek. Stanje medpomnilnika podajata indeksa *begin* in *end*. Prvi indeks kaže na podatek, ki je trenutno na vrsti za branje, drugi indeks pa na prvo prosto mesto v medpomnilniku, kamor naj se

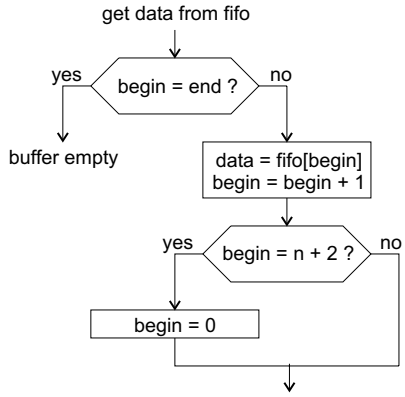
shrani naslednji na novo prispeli podatek. Lahko rečemo, da kaže indeks *begin* na mesto polnjenja, indeks *end* pa označuje mesto praznjenja. Če sta indeksa *begin* in *end* enaka, potem je medpomnilnik prazen. V njem ni nobenega podatka. Poln medpomnilnik prepoznamo tako, da je indeks *end* v cikličnem smislu za ena manjši od indeksa *begin*. Zadnjega prostega mesta ne smemo zapolniti, saj bi sicer indeksa postala enaka, kar pomeni prazen medpomnilnik. V medpomnilnik lahko torej shranimo največ n podatkov.



Slika 3.9: Algoritem dajanja (pisanja) v ciklični medpomnilnik FIFO

Iz vsega povedanega sledita tudi algoritma za dajanje v in jemanje iz statičnega cikličnega medpomnilnika FIFO (sliki 3.9 in 3.10). Algoritma preverjata stanje medpomnilnika (v poln medpomnilnik ni možno dodati novega podatka,

in obratno, iz praznega medpomnilnika podatka ni mogoče prebrati) in skrbita za ciklično povečevanje obeh indeksov.



Slika 3.10: Algoritem jemanja (branja) iz cikličnega medpomnilnika FIFO

Povrnimo se še za hip k problemu hkratnega dostopa do skupnih enot, v našem primeru do medpomnilnika. Tudi pri usklajevanju komunikacije med opravili s pomočjo medpomnilnikov lahko pride do spora pri dostopu. Denimo, da imamo medpomnilnik, kamor se vpisujejo znaki, ki naj se izpišejo na prikazovalniku LCD. Iz našega medpomnilnika bere le opravilo, ki skrbi za izpis znakov na prikazovalniku. Vanj pa lahko piše kdorkoli. Lahko se zgodi, da medtem, ko v medpomnilnik glavni program shranjuje svoje sporočilo, pridrvi opravilo, prekine glavni program sredi shranjevanja znakov, ter vrine svoje znake. Nastane nepopisna zmeda.

Premisliti velja tudi, ali lahko morda pride do težav pri vpisovanju v medpomnilnik, če je vpisovanje na kateremkoli mestu prekinjeno z branjem iz istega medpomnilnika.

Poglavje 4

Sistemski gonilniki zunanjih enot

V predhodnem poglavju smo spoznali zgradbo in princip delovanja našega preprostega operacijskega sistema. Pri tem imamo v mislih razvrščevalnik opravil realiziran v obliki prekinitvenega podprograma *sch_int* z zagonskim podprogramom *sch_on*. Razvrščevalnik opravil predstavlja jedro operacijskega sistema, ki je osnova za vse nadaljnje dodatke, kamor prištevamo tudi gonilnike zunanjih enot.

Srce mikrokrmilnika je centralno procesno jedro, ki načeloma ne zna drugega, kot izvajati strojne ukaze. Za kakršnokoli povezavo z zunanjim svetom potrebujemo zunanje, ali tako imenovane periferne enote. Nekatere izmed njih so v različnih izvedbah mikrokrmilnikov že integrirane poleg centralnega procesnega jedra. Tako je na primer v Philipsovem mikrokrmilniku LPC2138 poleg centralnega procesnega jedra ARM7TDMI-S integriranih več perifernih enot kot A/D in D/A pretvornik, sinhroni in asinhroni vmesniki (SPI, SSP, I^2C, UART), splo-

šni vhodno/izhodni priključki ... Druge zunanje enote kot prikazovalnik LCD, tipkovnica, signalne diode LED ... priključujemo na mikrokrmilnik od zunaj.

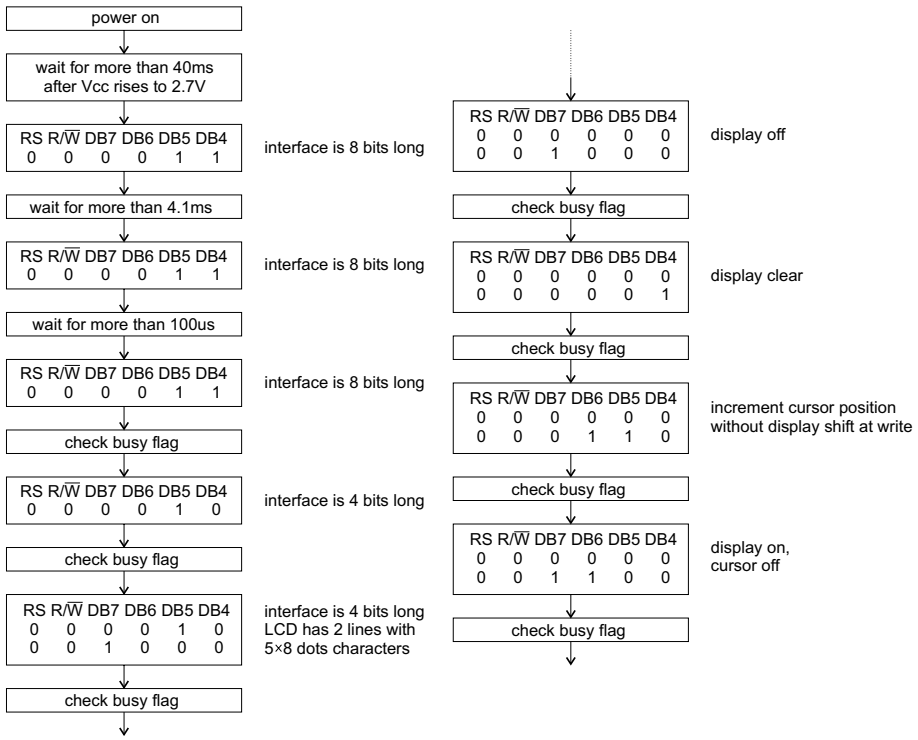
Naloga gonilnika (angl. driver) je komunikacija z zunanjo enoto na najnižjem registrskem nivoju. Vsa ostala programska nadgradnja nato z zunanjimi enotami komunicira izključno posredno preko gonilnikov. Dostopanje do zunanjih enot preko gonilnikov prinese s seboj tudi določeno stopnjo neodvisnosti programske opreme od strojne podlage. V primeru, da neko zunanjo enoto nadomestimo z drugo ekvivalentno enoto, je potrebno napisati le nov gonilnik. Vsa ostala programska oprema ostane nespremenjena.

Zgradba gonilnika je do neke mere odvisna tudi od operacijskega sistema, v katerem naj bi gonilnik deloval. Zato gonilnike štejemo za del sistema. Kot primer si bomo v tem poglavju razložili gonilnik za prikazovalnik LCD, ki bo deloval v našem preprostem operacijskem sistemu.

4.1 Primer gonilnika za prikazovalnik LCD

Poglejmo si načrtovanje gonilnika za pogosto uporabljan 16-mestni dvovrstični prikazovalnik LCD z vgrajenim krmilnikom HD44780U [13]. Prikazovalnik naj bo priključen v štiribitnem načinu na splošne vhodno/izhodne priključke mikrokrmilnika. To pomeni, da je prikazovalnik z mikrokrmilnikom povezan preko štiribitnega vodila (DB7 ... DB4), čemur so dodani še trije nadzorni biti (RS, R/$\overline{\text{W}}$ in urini impulzi E). Da prikazovalnik deluje, ga je potrebno inicializirati, kot je prikazano v algoritmu na sliki 4.1. Inicializacija prikazovalnika LCD se izvede le enkrat, navadno ob zagonu mikrokrmilniškega sistema. Tako si lahko privoščimo izvedbo zahtevanih zakasnitev kar s čakanjem, kot je prikazano na sliki 3.3. Inicializacija ni opravilo, kot bo kasneje gonilnik. Zato jo je potrebno

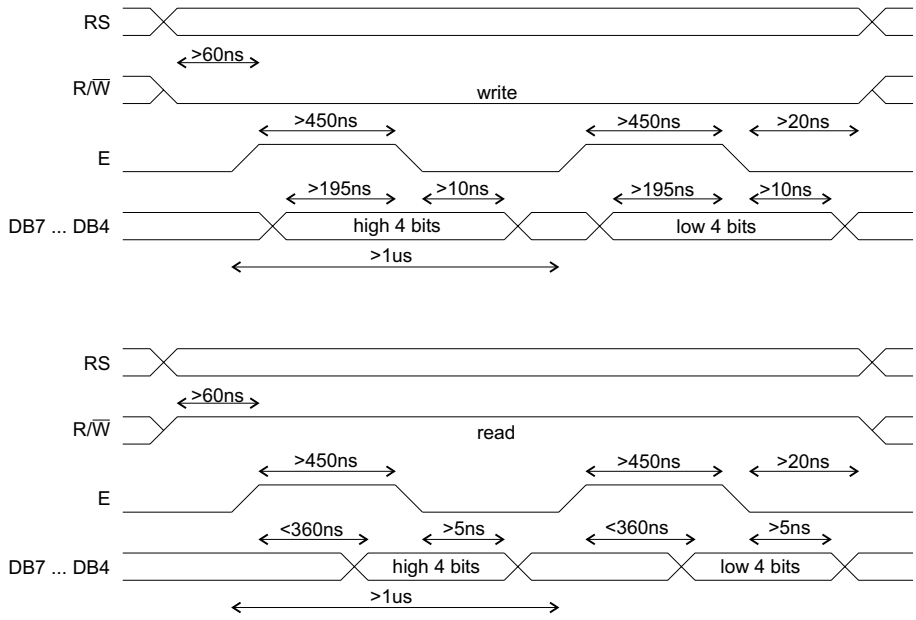
izvesti pred zagonom operacijskega sistema. Tako je gonilniku ob pričetku delovanja na voljo že delujoč prikazovalnik.



Slika 4.1: Algoritem inicializacijskega postopka prikazovalnika LCD z vgrajenim krmilnikom HD44780U v štiribitnem načinu

Mikrokrmilnik (gonilnik) bo prikazovalniku prenašal različne osembitne ukaze in podatke. Prenos osmih bitov v eno ali drugo stran po štiribitnem vodilu v

časovnem prostoru prikazuje slika 4.2. Da bo prenos potekal brez napak, mora gonilnik poskrbeti za pravilen časovni potek signalov na posameznih bitih.

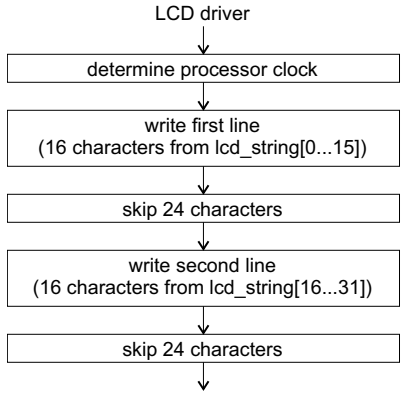


Slika 4.2: Pisanje in branje osmih bitov preko štiribitnega vodila v časovnem prostoru

Prikazovalnik HD44780U shranjuje v svojem internem pomnilniku RAM dvakrat po 40 znakov, čeprav prikazovalnik LCD prikazuje le dve vrstici po 16 znakov. Prikazani znaki pravzaprav predstavljajo okno 2×16 znakov v sicer daljših vrsticah 2×40 znakov. Okno je možno premikati, česar naš gonilnik ne bo počel. Vendar moramo pri načrtovanju gonilnika omenjeno lastnost upoštevati, saj je potrebno ob koncu prve vrstice kurzor še 24 krat premakniti, da pridemo v drugo vrstico. Enako velja ob koncu druge vrstice.

Gonilnik za prikazovalnik LCD naj bo napisan kot opravilo v operacijskem sistemu. Poskrbi naj za komunikacijo med programsko opremo in prikazovalnikom. Prikazovalnik hkrati prikazuje 32 znakov v dveh vrsticah, zato si v pomnilniku RAM rezerviramo tabelo *lcd_string* z 32-imi celicami. Gonilnik naj poskrbi, da se vsebina tabele ves čas preslikava na prikazovalnik. Oziroma

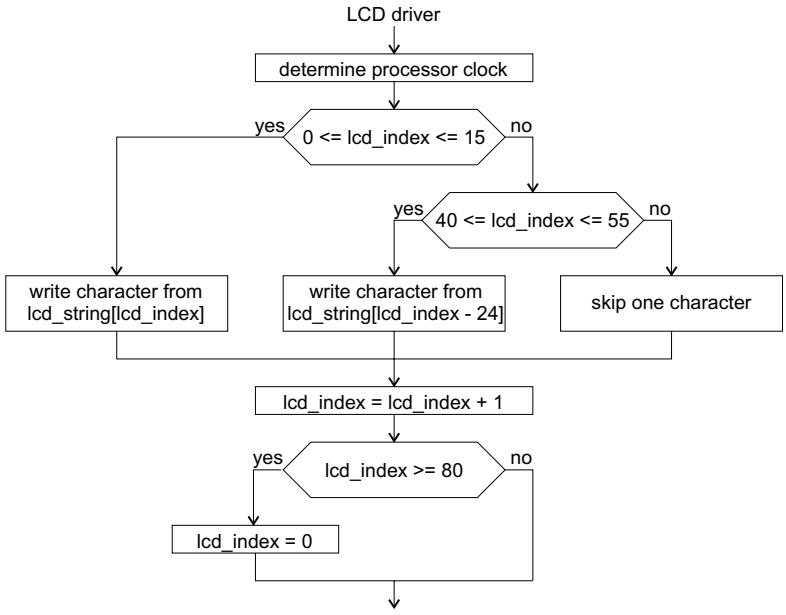
z drugimi besedami, pri pisanju programske opreme ni potrebno razmišljati o prikazovalniku, njegovi priključitvi, časovnih potekih krmilnih signalov ... Če naj se znak prikaže na določenem mestu na prikazovalniku, je potrebno njegovo ASCII kodo le vpisati na istoležno mesto v tabeli. Za dejanski prikaz je zadolžen gonilnik.



Slika 4.3: Algoritem gonilnika za prikazovalnik LCD s celotnim osveževanjem

Ker je gonilnik opravilo, se prikazovalnik osveži vsakič, ko je gonilnik na vrsti. Za osveževanje prikazovalnika pri pisanju ostale programske opreme ni potrebno skrbeti z neposrednimi klici gonilnika. Za to je zadolžen operacijski sistem, ki ga kliče v enakomernih časovnih presledkih Δt. Gonilnik lahko zasnujemo tako, da vsakič, ko je na vrsti, osveži celoten prikazovalnik, torej vseh 32 znakov. Algoritem takšne realizacije gonilnika je prikazan na sliki 4.3. Ker mora gonilnik poskrbeti za pravilen časovni potek signalov, najprej določi frekvenco urinega signala procesnega jedra, nakar osveži prvo in nato po premiku kurzorja na začetek še drugo vrstico. Pred koncem gonilnik s premikanjem postavi kurzor

zopet na začetek prve vrstice. Pristop je primeren predvsem, kadar so časovne rezine razmeroma dolge in so opravila zato relativno redko na vrsti.



Slika 4.4: Algoritem gonilnika za prikazovalnik LCD z delnim osveževanjem

Ker gonilnik vsakič osveži celoten prikazovalnik LCD, je sorazmerno dolg. Zato opisan pristop ni ustrezen v primeru, ko imamo opraviti z zelo kratkimi časovnimi rezinami. Takrat so opravila tudi pogosto na vrsti in vsakokratno osveževanje celotnega zaslona niti nima smisla. Gonilnik zato zasnujemo drugače, in sicer naj vsakič, ko je na vrsti, osveži samo en, to je naslednji, znak. Algoritem takšne realizacije gonilnika je prikazan na sliki 4.4. Zaradi pravilnih časovnih potekov krmilnih signalov zopet najprej določimo frekvenco urinega signala procesnega jedra. Kateri znak je na vrsti označuje indeks *lcd_index*, katerega začetno vrednost moramo postaviti pred zagonom operacijskega sistema, oziroma gonilnika. Upoštevati je potrebno, da je vsaka vrstica dolga pravzaprav 40 znakov, od katerih je prikazanih prvih 16.

V gonilnik bi lahko vgradili tudi nekaj inteligence. Na primer, namesto premikanja znak po znak preko 40-ih mest do začetka vrstice bi lahko kurzor tja

postavili s posebnim ukazom. Dalje bi z nekaj dodatnimi programskimi vrsticami lahko razbirali spremembe v tabeli *lcd_string*. Tako ne bi bilo potrebno osveževati vedno vseh znakov, ampak le tiste, ki so se spremenili ...

Gonilnik je mogoče napisati na več različnih načinov. Vsak način ima svoje prednosti in je primeren v določenem režimu delovanja. Seveda bi si lahko komunikacijo med programsko opremo in LCD prikazovalnikom zamislili tudi povsem drugače in ne bi uporabili tabele *lcd_string*. Temu bi morali priložiti tudi pripadajoč gonilnik.

Poglavje 5

Zbirke podprogramov in knjižnice funkcij

V predhodnih dveh poglavjih smo predstavili naš operacijski sistem. V 3. poglavju smo spoznali jedro sistema, katerega nadgradimo z gonilniki zunanjih enot. V 4. poglavju smo pokazali primer izdelave gonilnika za prikazovalnik LCD. Operacijski sistem v ožjem smislu je s tem zaključen. V širšem smislu pa so operacijski sistemi običajno opremljeni še z raznimi programskimi knjižnicami, oziroma zbirkami podprogramov. Zbirka podprogramov je nabor splošno uporabnih makrojev in podprogramov. Tukaj navadno najdemo podprograme, ki nam olajšajo delo pri pretvorbi podatkov iz ene oblike v drugo, aritmetičnih operacijah, vhodno/izhodnih medpomnilnikih ...

Pri programiranju v programskem jeziku C [14] [15] imamo na voljo obsežen nabor ANSI (American National Standards Institute) standardnih funkcij, ki nam delo močno olajšajo in pohitrijo. Po namenu uporabe so zbrane v programskih knjižnicah (angl. program library). Za primer navedimo splošno standardno knjižnico *stdlib* (standard library), kjer se med drugim nahajajo tudi funkcije za delo z znakovnimi nizi, knjižnico s funkcijami za delo s standardnim vhodom in izhodom *stdio* (standard input/output), ter knjižnico z matematičnimi funkcijami *math*. Pri uporabi standardnih knjižničnih funkcij na majhnih mikrokrmilnikih moramo biti včasih previdni. Nekatere med njimi so namenjene predvsem uporabi na večjih mikroprocesorskih sistemih, kot so osebni računalniki. Dolžina programa na takšnih sistemih dandanes ne predstavlja resne omejitve, česar za majhne mikrokrmilnike, kot je Philipsov LPC2138, ne

moremo trditi. Nekatere funkcije so napisane zelo splošno in so zato precej obsežne. Po prevajanju lahko predstavljajo tudi do nekaj 10kB programske kode, kar na majhnih sistemih hitro preseže velikost razpoložljivega pomnilnika. Takšen primer je široko uporabljana funkcija za oblikovanje znakovnih nizov *printf()*. V nekaterih primerih je namesto uporabe splošne standardne funkcije smotrno napisati novo funkcijo, v kateri implementiramo le tisto, kar v določeni aplikaciji potrebujemo. S tem smo se že dotaknili tehnik programiranja v programskem jeziku C, kar presega vsebino tega sestavka.

Dodatek A

Zbirniški prevajalnik

Vsi primeri zbirniške izvorne kode, ki so navedeni v tej skripti, so napisani za prevajalnik *as* in povezovalnik *ld*. Tukaj je podan le kratek opis nekaterih navodil prevajalniku, katerih vsaj načelno poznavanje je potrebno za razumevanje primerov. Podrobnejši opis prevajalnika *as* lahko bralec najde v [4], povezovalnika *ld* pa v [7].

GNU prevajalnik *as* predstavlja celo družino prevajalnikov za različne arhitekture oziroma različne mikrokrmilnike. Tako imamo na voljo enotno okolje za programiranje različnih arhitektur, ki imajo skupne formate objektnih datotek,

sorodno sintakso in enaka navodila prevajalniku, ki jih pogosto imenujemo tudi psevdo ukazi.

A.1 Osnovna sintaktična pravila prevajalnika *as*

Razdelek na kratko zajema sintaktična pravila prevajalnika *as*. Ob prevajanju zbirniške izvorne kode se izvede predprocesiranje ki:

- odstrani vse odvečne presledke (kakršnokoli zaporedje presledkov in tabulatorjev se nadomesti z le enim znakom za presledek),
- odstrani vse komentarje, tako da se pri tem ne spremeni koda ali oštevilčenje vrstic, ter
- znakovne konstante pretvori v numerične.

Presledek torej predstavlja eden ali več znakov za presledek ali tabulator v kakršnemkoli vrstnem redu. Presledke uporabljamo za ločevanje simbolov, ter seveda zato, da zbirniško izvorno kodo naredimo čitljivejšo.

Komentar se pri prevajanju odstrani. Označen je z začetkom `/*` in koncem `*/`. Komentarjev ne moremo gnezditi, kajti oznaka konca notranjega komentarja zaključi tudi zunanjega.

```
/* Primer komentarja, ki se ne gnezdi. */
```

Simbol je niz enega ali več znakov. Uporabljamo lahko velike in male črke angleške abecede, številke, ter znake pika `.`, podčrtaj `_` in dolar `$`. Prvi znak simbola ne sme biti številka. Simboli so občutljivi na velike in male črke, ter nimajo omejene dolžine. Vsi znaki v simbolu tvorijo njegovo ime.

Stavek največkrat predstavlja ena vrstica izvorne datoteke. Znak za novo vrstico `\n` hkrati konča tudi stavek. Redkeje se v eni vrstici nahaja več stavkov, ki so med seboj ločeni z ločilom, navadno s podpičjem `;`. Stavek se lahko razteza tudi preko več vrstic. Če je zadnji znak v vrstici vzvratna poševnica `\`, se stavek nadaljuje v naslednji vrstici.

Stavek pričenja z nič ali več labelami, katerim lahko sledi glavni simbol. Le ta pove s kakšnim stavkom imamo opraviti, ter določa nadaljnjo sintakso stavka.

Če je prvi znak v glavnem simbolu pika `.`, potem glavni simbol predstavlja takoimenovan psevdo ukaz, stavek pa je navodilo prevajalniku. V nasprotnem primeru, ko je prvi znak glavnega simbola črka, je stavek zbirniški ukaz, glavni simbol pa je navadno mnemonik tega ukaza.

Oznaka je simbol, ki mu sledi dvopičje `:`. Presledek med imenom labele in dvopičjem ni dovoljen. Nekaj ponazoritev:

```
labela:  .psevdo  ...            /* navodilo prevajalniku */
                                 /* prazen stavek         */
labela:  ukaz     operand, ...   /* zbirniski ukaz        */
         ukaz                    /* stavek brez labele    */
```

Konstanta je število, znak ali niz znakov. Njena vrednost je razvidna sama po sebi, neodvisno od zveze s simboli, ki jo obkrožajo. Naj na tem mestu podamo le nekaj osnovnih pravil pisanja konstant. Številske konstante lahko zapišemo v različnih številskih sistemih, ki jih podaja kombinacija znakov pred vrednostjo. Binaren zapis označuje prefiks `0b`, osmiški zapis označuje `0`, desetiški zapis nima posebne oznake, ter šestnajstiški zapis označuje `0x`. Tako lahko konstanto 42 zapišemo na naslednje načine: `0b101010`, `052`, `42` in `0x2a`. Znakovna konstanta je zapisana tako, da pred njo postavimo enojni narekovaj (primer: `'a`), niz znakov pa vstavimo v dvojne narekovaje (primer: `"niz"`).

$-$	unarni minus	$*$	množenje	$\|$	bitni ali
$\sim$	bitna negacija	$/$	deljenje	&	bitni in
		%	ostanek deljenja	^	bitni ekskluzivni ali
$+$	seštevanje	$<, <<$	pomik levo	!	bitni ali ne
$-$	odštevanje	$>, >>$	pomik desno		

Tabela A.1: Operacije, ki jih prevajalnik pozna

Konstanto je mogoče zapisati tudi z izrazom. Največkrat jih izračunavamo iz parametrov programa, ki so podani s psevdo ukazi `.equ` (glej dodatek A.3),

in so zbrani na enem mestu v izvorni kodi. Vrstni red računanja določimo z okroglimi oklepaji (). Operacije, ki jih prevajalnik pozna, so podane v tabeli A.1.

A.2 Odseki kode in njihova postavitev v naslovnem prostoru

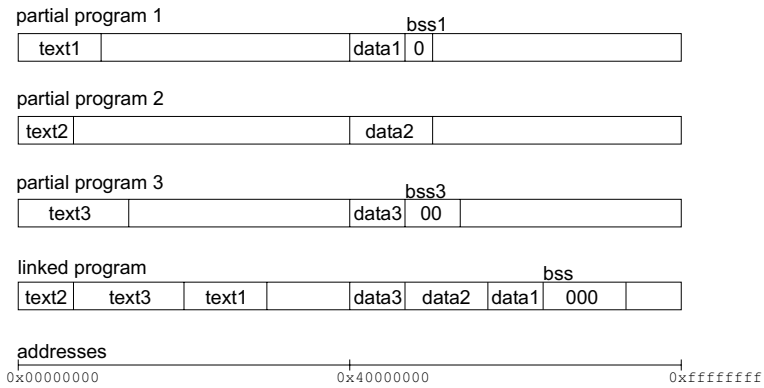
Odsek je poenostavljeno povedano neprekinjen interval v naslovnem prostoru. Vsi podatki na naslovih v takšnem odseku imajo neko skupno lastnost, na primer samo za branje.

Povezovalnik *ld* prebere eno ali več objektnih datotek, v katerih so delčki celotnega programa. Objektne datoteke naredi prevajalnik *as* iz izvornih datotek. Za delni program v vsaki od njih je privzet začetni naslov nič. Povezovalnik zato na novo določi naslove začetkov posameznih delnih programov tako, da se ne prekrivajo.

Povezovalnik premika odseke na njihove končne lokacije kot toge, nespremenljive kose kode. Dolžina in vrstni red podatkov v odseku se ne spreminjata. Poleg tega povezovalnik seveda poskrbi za preračunavanje naslovov iz relativnih v končne absolutne.

V vsaki objektni datoteki se nahajajo najmanj trije odseki, in sicer en odsek tipa *text*, en odsek tipa *data* in en odsek tipa *bss*. Katerikoli izmed odsekov je lahko tudi prazen. Program se nahaja v odsekih tipa *text* in *data*. Navadno se koda (izvršljivi ukazi) in nespremenljive konstante nahajajo v odseku tipa *text*, ki je namenjen samo branju. Spremenljive konstante se nahajajo v odseku tipa *data*. Odseki tipa *text* se pričnejo od naslova `0x00000000` dalje, odseki tipa *data* pa od naslova `0x40000000` dalje (glej dodatek B.1). Odsek tipa *bss* vsebuje ničle in je predviden za neinicializirane spremenljivke. To pomeni, da v odsek

tipa *bss* ne moremo zapisovati podatkov pred zagonom programa, oziroma v tak odsek lahko piše le program med svojim delovanjem.



Slika A.1: Povezovanje odsekov v delnih programih

Idealiziran primer postavljanja posameznih odsekov v naslovnem prostoru prikazuje slika A.1. Prikazani so trije delni programi, vsak s svojimi tremi odseki, ki so nato povezani v končni program. Naslovna os teče v horizontalni smeri.

Poenostavljeno lahko rečemo, da se preveden in povezan program na koncu nahaja le v dveh odsekih, enem tipa *text* in enem tipa *data*. Kljub temu pa imamo lahko več različnih pododsekov, v katere združujemo sorodne dele kode. Posamezen pododsek je lahko raztresen preko več delnih programov. Pododseki so oštevilčeni, kar povemo s psevdo ukazoma `.text` in `.data` (glej dodatek A.3). V končnem povezanem programu se pojavijo po vrstnem redu, in sicer najprej

tisti z najnižjimi številkami. Če pododsekov ne uporabljamo, predstavlja celoten odsek hkrati tudi ničti pododsek. Za primer si poglejmo naslednjo izvorno kodo:

```
.text 1
.long 0x00000008
.text 0
.long 0x00000000
.text 2
.long 0x00000010
.text 1
.long 0x0000000c
.text 0
.long 0x00000004
```

Ko odseke zložimo po vrsti, dobimo pravzaprav naslednje:

```
.long 0x00000000
.long 0x00000004
.long 0x00000008
.long 0x0000000c
.long 0x00000010
```

A.3 Navodila prevajalniku

Imena vseh navodil prevajalniku ali tako imenovani psevdo ukazi se začenjajo s piko `.`, ter nadaljujejo z (največkrat majhnimi) črkami angleške abecede. Tukaj je opisanih le nekaj najbolj pogosto uporabljanih psevdo ukazov in njihova sintaksa.

`.align constant1 [, constant2]`
V kodo doda toliko bajtov z vrednostjo *constant2*, da postane števnik lokacij večkratnik števila $2^{constant1}$. Če drugi argument ni podan, je privzeta vrednost za *constant2* enaka nič. Vrednosti obeh konstant sta lahko podani tudi z absolutnimi izrazi.
Na arhitekturi ARM so vsi ukazi 32-bitni (štirje bajti) in vedno začenjajo na na-

slovu, ki je večkratnik števila štiri. Zato je včasih potrebno narediti poravnavo. Primer:

```
.ascii "aaa"
.align 2
```

`.ascii "string1" [, "string2"[, "string3"[ ... ]]]`
Nizi *string1*, *string2* ... so lahko dolgi nič ali več znakov. Prevajalnik jih prevede v zaporedje ASCII kod. Vsak znak zasede en bajt. Ničti bajt na koncu vsakega niza ni dodan. Tako je psevdo ukaz

```
.ascii "string1", "string2", "string3"
```

pravzaprav enakovreden psevdo ukazu

```
.ascii "string1string2string3"
```

`.asciz "string1" [, "string2"[, "string3"[ ... ]]]`
Nizi *string1*, *string2* ... so lahko dolgi nič ali več znakov. Prevajalnik jih prevede v zaporedje ASCII kod. Vsak znak zasede en bajt. Na koncu vsakega niza je dodan še ničti bajt.

`.bss`
Označuje *bss* odsek. Psevdo ukazi (navadno `.lcomm` ukazi), ki sledijo, rezervirajo prostor za neinicializirane podatke, oziroma simbole. Ukaz ni nujno potreben, saj ostali psevdo ukazi že povedo, da gre za rezervacije prostora v *bss* odseku. Vendar ga kljub temu zaradi jasnosti in preglednosti programske kode navadno pišemo.

`.byte constant1 [, constant2[, constant3[ ... ]]]`
Prevajalnik konstante *constant1*, *constant2* ... preprosto prepiše. Vsaka konstanta se zapiše v naslednji bajt. Vrednosti konstant so lahko podane tudi z absolutnimi izrazi.

`.code 16 | 32`
Psevdo ukaz je specifičen za arhitekture procesnih jeder ARM. Podaja način

prevajanja za okleščeni 16-bitni nabor ukazov *Thumb*, oziroma za normalen 32-bitni nabor ukazov *ARM* (glej dodatek C). Uporabljata se tudi psevdo ukaza `.thumb` in `.arm`, ki imata enak pomen kot `.code 16` in `.code 32`.

`.data [subsection]`
Koda, ki sledi temu psevdo ukazu, se doda v *data* pododsek številka *subsection*. Če konstanta *subsection* ni podana, je njena privzeta vrednost enaka nič. Podana je lahko tudi z absolutnim izrazom.

`.else`
Predstavlja del *if* stavka, ki omogoča pogojno prevajanje (glej `.if` psevdo ukaz). Označuje začetek dela kode, ki se prevede, če pogoj v *if* stavku ni bil izpolnjen.

`.endif`
Predstavlja del *if* stavka, ki omogoča pogojno prevajanje (glej `.if` psevdo ukaz). Označuje konec dela kode, ki se prevaja pogojno.

`.equ symbol, constant`
Psevdo ukaz priredi konstanto *constant* simbolu *symbol*. Povsod v kodi, kjer se pojavi simbol *symbol*, se bo le ta zamenjal s konstanto *constant*, ki je lahko podana tudi z absolutnim izrazom.

`.global symbol` ali `.globl symbol`
S tem povemo prevajalniku, naj bo simbol *symbol* viden tudi povezovalniku. Če je simbol *symbol* definiran v našem delnem programu, postane s tem, ko ga deklariramo kot globalnega, viden tudi v drugih delnih programih, s katerimi je naš povezan. In obratno, če simbol *symbol* ni definiran v našem delnem programu, potem s to deklaracijo prevzamemo njegove lastnosti, ki so podane v nekem drugem delnem programu, s katerim je naš povezan.

`.hword constant1 [, constant2[, constant3[ ... ]]]`
Prevajalnik prepiše 16-bitne konstante *constant1*, *constant2* ... Vsaka konstanta se zapiše v naslednja dva bajta. Enak pomen ima psevdo ukaz `.short`. Vrednosti konstant so lahko podane tudi z absolutnimi izrazi.

`.if condition`
Označuje začetek dela kode, ki naj se prevede, če je konstanta *condition* različna

od nič. V nasprotnem primeru se ta del kode ignorira. Konec pogojnega dela kode mora biti označen s psevdo ukazom `.endif`. Med psevdo ukaza `.if` in `.endif` lahko dodamo še `.else`, s čimer uvedemo alternativo, ki se prevede, kadar je konstanta *condition* enaka nič. Primer:

```
.if condition
   koda (condition != 0)
.else
   koda (condition == 0)
.endif
```

Poleg psevdo ukaza `.if` poznamo še naslednje izpeljanke:

`.ifdef symbol` označuje začetek dela kode, ki se prevede, če je simbol *symbol* definiran (glej `.equ` psevdo ukaz).

`.ifndef symbol` ali `.ifnotdef symbol` označuje začetek dela kode, ki se prevede, če simbol *symbol* ni definiran (glej `.equ` psevdo ukaz).

Konstanta *condition* je lahko podana tudi z absolutnim izrazom.

`.include "file"`
V program vključi vsebino datoteke *file*. Tako lahko datoteko z izvorno kodo razdelimo na več manjših. Vsebina vključene datoteke se doda na mesto, kjer se nahaja `.include` psevdo ukaz.

`.lcomm symbol, length`
Rezervira *length* bajtov v odseku tipa *bss* za simbol *symbol*. Vsebina rezerviranih bajtov ni določena, oziroma ostaja neinicializirana. Simbol *symbol* ni globalen in ga kot takega povezovalnik ne vidi.

`.long constant1 [, constant2[, constant3[ ... ]]]`
Prevajalnik prepiše 32-bitne konstante *constant1*, *constant2* ... Vsaka konstanta se zapiše v naslednje štiri bajte. Enak pomen ima psevdo ukaz `.int`. Vrednosti konstant so lahko podane tudi z absolutnimi izrazi.

`.section name [, "flags"]`
Navodilo prevajalniku, naj se programska koda, ki sledi, prevede v poseben

odsek z imenom *name*. Celoten odsek *name* se na koncu ob povezovanju doda v odsek *text* ali *data*, kar je povedano v navodilih povezovalniku [7]. Kakšne vrste koda se v tem odseku nahaja opisujejo zastavice *flags*. In sicer:

- *a* ... odsek je dodeljiv (angl. Allocatable),
- *w* ... odsek je spremenljiv (angl. Writable) in
- *x* ... odsek je izvršljiv (angl. eXecutable).

`.text [subsection]`
Koda, ki sledi temu psevdo ukazu, se doda v *text* pododsek številka *subsection*. Če konstanta *subsection* ni podana, je njena privzeta vrednost enaka nič. Podana je lahko tudi z absolutnim izrazom.

Dodatek B

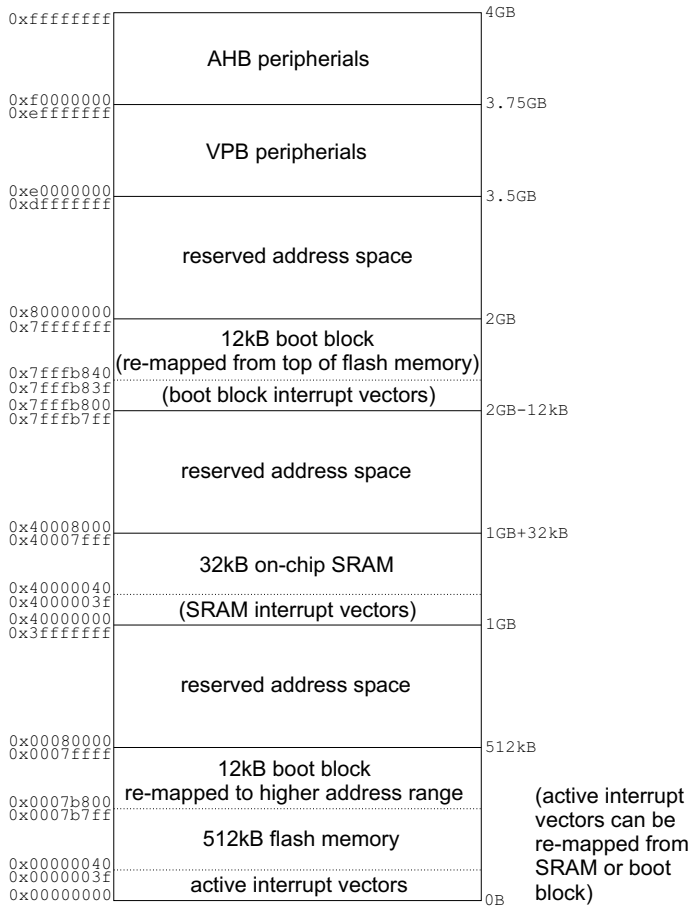
Kratek opis lastnosti mikrokrmilnika Philips LPC2138

Philipsov mikrokrmilnik LPC2138 temelji na centralnem procesnem jedru ARM7TDMI-S [3]. Popoln opis mikrokrmilnika je moč najti v [5], nekaj več razlage, ki pripomore k razumevanju tehnologije ARM pa v [8], [9] in [10]. Tukaj je navedenih le nekaj mikrokrmilnikovih glavnih značilnosti, ki so potrebne za razumevanje primerov v tej skripti.

B.1 Razdelitev naslovnega prostora

Na sliki B.1 vidimo, da ima mikrokrmilnik LPC2138 32-bitno naslovno vodilo, kar pomeni 4GB povsem linearnega naslovnega prostora. Velika večina tega prostora ostaja neizkoriščena. Mikrokrmilnik ima vgrajenih 512kB programir-

ljivega pomnilnika flash za shranjevanje izvršljive kode in konstantnih podatkov, ter 32kB statičnega pomnilnika RAM za shranjevanje spremenljivih podatkov.



Slika B.1: Razdelitev naslovnega prostora mikrokrmilnika LPC2138

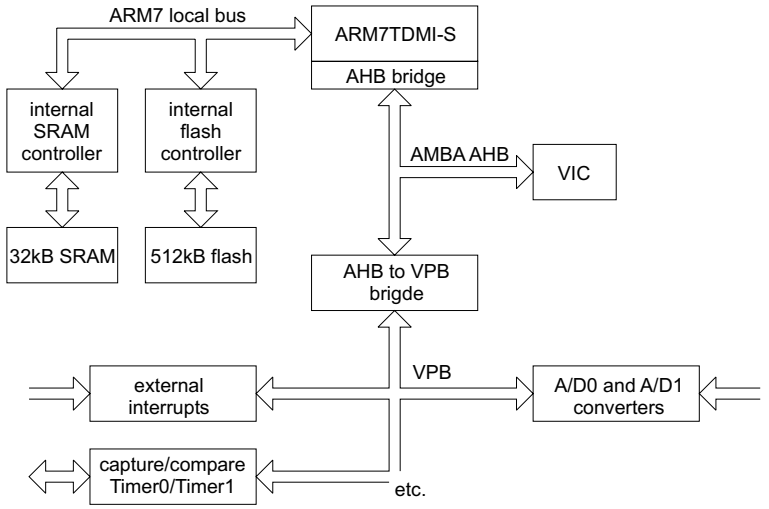
Mikrokrmilnik LPC2138 ima von Neumannovo zgradbo z enim 32-bitnim podatkovnim vodilom za prenos podatkov in ukazov. Ker je tudi podatkovno vodilo 32-bitno, so v bistvu vsakič naslovljeni po štirje bajti hkrati.

Kratici VPB in AHB pomenita VLSI Peripherial Bus in AMBA Advanced

High-performance Bus (glej dodatek B.2). Nerazporejenega naslovnega prostora ne moremo uporabiti. V primeru, da naslovimo lokacijo, ki ni razporejena, bo mikrokrmilnik generiral prekinitev.

B.2 Vodila

Centralno procesno jedro ARM7TDMI-S ima tri vodila. In sicer svoje lokalno vodilo, ki povezuje jedro s pomnilnikom, AHB, ki povezuje jedro z vektorskim nadzornikom prekinitev, ter VPB, ki skrbi za povezavo z vgrajenimi perifernimi enotami. Kratici AHB in VPB pomenita AMBA Advanced High-performance Bus in VLSI Peripherial Bus. Razmere prikazuje slika B.2.



Slika B.2: Vodila pri centralnem procesnem jedru ARM7TDMI-S

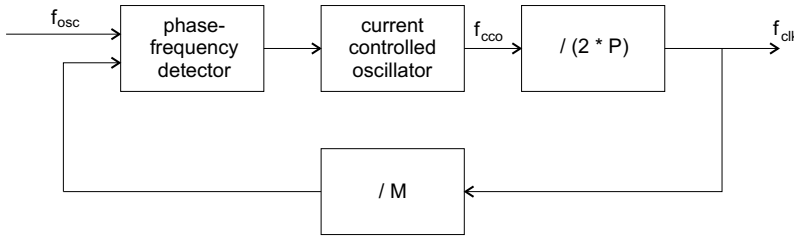
Razvijalec navzven vidi le en zvezen in linearen 32-bitni naslovni prostor. Navznoter je stvar nekoliko bolj zapletena. Jedro ARM7 je neposredno povezano le s pomnilnikom preko svojega lokalnega vodila. Z enako hitrostjo teče tudi vodilo AHB, ki je priključeno preko mosta. Kot edina periferna enota se na tem vodilu nahaja vektorski nadzornik prekinitev. Poleg tega je tu še prehod

do vodila VPB. Hitrost lokalnega in vodila AHB je določena s fazno sklenjeno zanko, ki jo bomo spoznali v nadaljevanju.

Vse ostale periferne enote so priključene na vodilo VPB. Prehod AHB/VPB vključuje delilnik frekvence, tako da lahko vodilo VPB teče počasneje kot lokalno vodilo in vodilo AHB. Pri mikrokrmilniku LPC2138 je možno faktor deljenja nastaviti tudi na ena. V tem primeru je hitrost vseh treh vodil enaka, oziroma vodilo VPB teče z enako hitrostjo kot lokalno vodilo in vodilo AHB. Nižja hitrost vodila VPB je včasih zaželena. In sicer periferne enote navadno ne potrebujejo tako velikih hitrosti za svoje delovanje, po drugi strani pa pri nižjih hitrostih potrebujejo manjši napajalni tok. Prav tako počasne periferne enote na ta način ne postanejo ozko grlo na zelo hitrem vodilu.

B.2.1 Fazno sklenjena zanka

Fazno sklenjena zanka se uporablja za generiranje signala poljubne frekvence z natančnostjo osnovnega, navadno kvarčnega oscilatorja. Razmerje med generirano in osnovno frekvenco je vedno racionalno število. V mikroprocesorskem svetu so fazno sklenjene zanke največkrat uporabljene za generiranje ure procesnega jedra, ki je višja od frekvence zunanjega oscilatorja.



Slika B.3: Fazno sklenjena zanka

Fazno sklenjena zanka na svojem vhodu sprejema signal osnovnega zunanjega oscilatorja, ki mora biti za mikrokrmilnik LPC2138 v območju med 10MHz do 25MHz. Ta signal nato pomnoži s faktorjem M in tako generira urin signal za lokalno vodilo in vodilo AHB. Urin signal mora biti za omenjeni mikrokrmilnik v območju med 10MHz do 60MHz. Iz tega sledi, da je faktor M lahko najmanj ena in največ šest. Delovanje fazno sklenjene zanke ponazarja slika B.3.

Primerjalnik faze primerja signal zunanjega oscilatorja f_{osc} z generiranim urinim signalom f_{clk} deljenim s faktorjem M. Signala bi morala biti načeloma

enaka. Izhod primerjalnika krmili tokovno krmiljen oscilator, katerega frekvenca f_{cco} mora biti v območju med 156MHz do 320MHz. Če signal f_{clk} zaostaja za f_{osc}, potem fazni primerjalnik krmiljenemu oscilatorju povečuje frekvenco, in obratno. Urin signal f_{clk} je generiran z deljenjem frekvence krmiljenega oscilatorja s faktorjem $2 \times P$.

Registri, ki definirajo delovanje fazno sklenjene zanke, so naslednji:

PLLFEED PLL FEED register (naslov: `0xe01fc08c`)
Da sprememba v *PLLCON* ali v registru *PLLCFG* postane veljavna, je potrebno v ta register zaporedoma vpisati konstantni vrednosti `0x000000aa` in `0x00000055`. Oba vpisa se morata zgoditi takoj, eden za drugim, v dveh zaporednih ukazih. Zaradi tega je potrebno medtem onemogočiti prekinitve. Prekinitev, ki bi se izvedla med obema vpisoma, bi seveda povzročila, da se ukaza ne izvedeta več eden za drugim.

PLLCON PLL CONtrol register (naslov: `0xe01fc080`)
Register služi za omogočanje in povezovanje fazno sklenjene zanke s procesnim jedrom. Ali je fazno sklenjena zanka omogočena, ali ne, pove vrednost bit0. Postavljen bit0 pomeni, da je fazno sklenjena zanka sklenjena. Zaradi svoje narave poskuša najti stabilno stanje.

Povezovanje fazno sklenjene zanke določa bit1. Postavljen bit1 povzroči, da se signal f_{clk} uporablja kot ura procesnega jedra. V nasprotnem primeru se kot ura uporablja kar signal zunanjega oscilatorja f_{osc}.

Vsaka sprememba v tem registru postane veljavna šele, ko v register *PLLFEED* zaporedoma vpišemo vnaprej določena podatka. Tako na primer postavitev bita bit0 omogoči (sklene) fazno sklenjeno zanko takoj po tem, ko register *PLLFEED* prejme predpisano zaporedje.

Ostali biti v tem registru nimajo pomena. Kombinacija bit0 = 0 in bit1 = 1, ko fazno sklenjena zanka ni sklenjena, a je vseeno povezana, je prepovedana. V

tem primeru mikrokrmilnik kot uro še naprej uporablja f_{osc}, čeprav je zahtevan f_{clk}.

bit6	bit5	P
0	0	1
0	1	2
1	0	4
1	1	8

Tabela B.1: Vrednosti konstante P

PLLCFG PLL ConFiGuration register (naslov: `0xe01fc084`)
Register določa konstanti M in P, ki sta uporabljeni za deljenje signalov f_{clk} in f_{cco} v fazno sklenjeni zanki. Najnižji trije biti, to je bit2 do bit0, podajajo vrednost konstante M zmanjšane za ena. Torej so na teh treh bitih dovoljene kombinacije od `0x00` do največ `0x05`. Bita bit3 in bit4 sta vedno enaka nič. Konstanta P je podana s kombinacijo bitov bit6 in bit5 (tabela B.1).

Konstanti M in P je potrebno določiti tako, da so vse frekvence signalov v predpisanih mejah. Oziroma veljati morajo zveze (B.1= in (B.2).

$$\begin{array}{rcccl} 10\text{MHz} & \leq & f_{osc} & \leq & 25\text{MHz} \\ 156\text{MHz} & \leq & f_{cco} & \leq & 320\text{MHz} \\ 10\text{MHz} & \leq & f_{clk} & \leq & 60\text{MHz} \end{array} \tag{B.1}$$

$$\begin{array}{rcl} f_{cco} & = & 2 \times P \times f_{clk} \\ f_{clk} & = & M \times f_{osc} \end{array} \tag{B.2}$$

Vsaka sprememba v tem registru postane veljavna šele, ko v register *PLLFEED* zaporedoma vpišemo vnaprej določena podatka.

PLLSTAT PLL STATus register (naslov: `0xe01fc088`)
Podaja trenutno stanje fazno sklenjene zanke, ki se ne ujema nujno s stanjem v registrih *PLLCON* in *PLLCFG*. Neujemanje pomeni, da po spremembi še ni

bilo predpisanega zaporednega vpisa v register *PLLFEED*, ter tako sprememba še ni veljavna. Ta register lahko le beremo. Biti v njem imajo naslednji pomen:

bit4 do **bit0** podajajo trenutno veljavno vrednost konstante M zmanjšano za ena; ker M ne more biti več kot šest, sta bit3 in bit4 vedno enaka nič

bit6 in **bit5** podajata trenutno veljavno konstanto P; vrednost konstante je določena s kombinacijo obeh bitov po tabeli B.1

bit8 pove ali je fazno sklenjena zanka trenutno sklenjena (omogočena), ali ne

bit9 pove ali je fazno sklenjena zanka trenutno povezana (procesno jedro uporablja signal f_{clk} za svojo uro), ali ne (procesno jedro uporablja f_{osc})

bit10 pove ali je fazno sklenjena zanka trenutno v stabilnem stanju; ko je zanka sklenjena (omogočena) je potrebno počakati, da se vniha, oziroma doseže stabilno stanje; šele po tem je lahko povezana s procesnim jedrom

Ostali biti v registru nimajo pomena.

Sledi primer kode, ki inicializira fazno sklenjeno zanko. Med izvajanjem podprograma *pll_ init* morajo biti prekinitve onemogočene. Tako so zapisi v register *PLLFEED* v vsakem primeru nemoteni. Prepoved prekinitev je lahko dosežena s postavitvijo zastavic I in F (glej dodatek C.2), ali pa z onemogočenjem v registru *VICIntEnClear* (glej dodatek B.4 in kodo na strani 179). Pred novo nastavitvijo je fazno sklenjena zanka razklenjena in ni povezana. Sledi postavljanje konstant M in P, ter sklenitev zanke (bit0 v *PLLCON*). Ko se zanka vniha (bit10 v *PLLSTAT*), je na vrsti povezava (bit1 v *PLLCON*).

```
/* Parameters */
          .equ  msel,                0x00
          .equ  psel,                0x03
/* Constants */
          .equ  plle,                0x01
          .equ  pllc,                0x02
          .equ  pllcon_dis,          0x00
          .equ  pll_feed_byte1,      0xaa
          .equ  pll_feed_byte2,      0x55
          .equ  msel_len,            0x05
          .equ  plock,               0x0400
```

```
/* Registers */
          .equ  pllcon,               0xe01fc080
          .equ  pllcfg,               0xe01fc084
          .equ  pllstat,              0xe01fc088
          .equ  pllfeed,              0xe01fc08c

          .code 32

/* Program code */
          .text
/* Phase locked loop (PLL) initialisation subroutine */
pll_init: stmfd sp!, {r0-r5, lr}
          ldr   r0, =pllcon
          mov   r1, #pllcon_dis
          str   r1, [r0]
          ldr   r1, =pllfeed
          mov   r2, #pll_feed_byte1
          mov   r3, #pll_feed_byte2
          str   r2, [r1]
          str   r3, [r1]
          ldr   r4, =pllcfg
          mov   r5, #psel
          mov   r5, r5, lsl #msel_len
          orr   r5, r5, #msel
          str   r5, [r4]
          mov   r4, #plle
          str   r4, [r0]
          str   r2, [r1]
          str   r3, [r1]
          ldr   r4, =pllstat
pll_lock: ldr   r5, [r4]
          ands  r5, r5, #plock
          beq   pll_lock
          mov   r4, #(pllc|plle)
          str   r4, [r0]
          str   r2, [r1]
          str   r3, [r1]
          ldmfd sp!, {r0-r5, lr}
          mov   pc, lr
```

V primeru zgoraj sta konstanti $M = 1$ in $P = 8$. Če je $f_{osc} = 12\text{MHz}$, potem je tudi $f_{clk} = 12\text{MHz}$, in $f_{cco} = 192\text{MHz}$. Pogoji iz neenačb (B.1) so tako izpolnjeni. Za razlago zbirniških ukazov glej dodatek C.

Enako je mogoče narediti v programskem jeziku C. Argument funkcije *pll_init()* je želena frekvenca ure f_{clk} v MHz. Funkcija je prirejena za frekvenco zunanjega oscilatorja $f_{osc} = 12\text{MHz}$, kar pomeni, da ima frekvenca ure f_{clk} lahko vrednosti 12MHz, 24MHz, 36MHz, 48MHz ali 60MHz. Zaporedna zapisa v register *PLLFEED* sta narejena v funkciji *pll_feed()*, ki predstavlja le ovojnico za nekaj vrstic zbirniške kode.

```
#define plle  0x00000001
#define pllc  0x00000002
#define plock 0x00000400
// Registers
#define PLLCON  (*((volatile unsigned long *)0xe01fc080))
#define PLLCFG  (*((volatile unsigned long *)0xe01fc084))
#define PLLSTAT (*((volatile unsigned long *)0xe01fc088))

// PLL feed sequence
void pll_feed() {
    asm("stmfd sp!, {r0-r2}");
    asm("ldr   r0, =0xe01fc08c");
    asm("mov   r1, #0x000000aa");
    asm("mov   r2, #0x00000055");
    asm("str   r1, [r0]");
    asm("str   r2, [r0]");
    asm("ldmfd sp!, {r0-r2}");
}

// Phase locked loop (PLL) initialisation
// clock_mhz ... clock rate in MHz [12,24,36,48,60]
void pll_init(int clock_mhz) {
    PLLCON = 0x00000000;
    pll_feed();
    switch(clock_mhz) {
    case 12:
        PLLCFG = 0x00000060;
        break;
```

```
        case 24:
            PLLCFG = 0x00000041;
            break;
        case 36:
            PLLCFG = 0x00000042;
            break;
        case 48:
            PLLCFG = 0x00000023;
            break;
        case 60:
            PLLCFG = 0x00000024;
            break;
        }
        PLLCON = plle;
        pll_feed();
        while(!(PLLSTAT & plock));
        PLLCON = PLLCON | pllc;
        pll_feed();
    }
```

B.2.2 Delilnik VPB

Delilnik VPB določa razmerje med urinim signalom lokalnega vodila, oziroma vodila AHB (f_{clk} ali f_{osc}), in urinim signalom vodila VPB (f_{vpb}), kamor so

priključene periferne enote. Nastavitev delilnika, oziroma delilno razmerje je podano v registru *VPBDIV*.

bit1	bit0	
0	0	$f_{clk} = 4 \times f_{vpb}$
0	1	$f_{clk} = f_{vpb}$
1	0	$f_{clk} = 2 \times f_{vpb}$

Tabela B.2: Delilno razmerje med urinima signaloma f_{clk} in f_{vpb}

VPBDIV VPB DIVider register (naslov: `0xe01fc100`)
Z bitoma bit0 in bit1 določa delilno razmerje $f_{clk} \div f_{vpb}$, kot ga podaja tabela B.2.

Primer izvorne kode, ki postavi delilno razmerje $f_{clk} = 4 \times f_{vpb}$. Če je $f_{clk} =$ 12MHz, potem je $f_{vpb} = 3$MHz, oziroma en cikel traja $1/3\mu$s, kar služi časovniku za merjenje časa (glej dodatek B.5).

```
/* Parameter */
             .equ  vpbdiv_val,     0x00
/* Register */
             .equ  vpbdiv,         0xe01fc100

             .code 32

/* Program code */
             .text
/* VPB divider initialisation subroutine */
vpbdiv_init: stmfd sp!, {r0-r1}
             ldr   r0, =vpbdiv
             mov   r1, #vpbdiv_val
             str   r1, [r0]
             ldmfd sp!, {r0-r1}
             mov   pc, lr
```

Za razlago zbirniških ukazov glej dodatek C.
Enako je moč napisati tudi v programskem jeziku C. Argument funkcije

set_ vpbdiv() predstavlja prihodnjo vrednost registra *VPBDIV* in je lahko enak 0, 1 ali 2.

```
// Register
#define VPBDIV (*((volatile unsigned long *)0xe01fc100))

// Set VLSI peripherial bus (VPB) divider
// div ... divider value [cclk_4,cclk_2,cclk]
void set_vpbdiv(int div) {
    VPBDIV = div;
}
```

B.3 Pomnilniški pospeševalnik

Pomnilniški pospeševalnik se nahaja med pomnilnikom flash in lokalnim ARM7 vodilom. Dostop do pomnilnika flash je počasnejši, kot je največja možna hitrost lokalnega vodila, zato bi ob branju prihajalo do čakanj procesnega jedra na podatek iz pomnilnika flash. Da se to ne bi dogajalo, poskuša poskrbeti pomnilniški pospeševalnik. To je posebno vezje, ki vnaprej bere podatke (128 bitov naenkrat) iz pomnilnika flash. V trenutku, ko centralno procesno jedro zahteva naslednji 32-bitni ukaz ali konstantni podatek, naj bi bil le-ta že pripravljen v medpomnilniku pomnilniškega pospeševalnika.

Način delovanja pomnilniškega pospeševalnika podajajo nastavitve v naslednjih dveh registrih:

MAMCR MAM Control Register (naslov: `0xe01fc000`)
Z bitoma bit1 in bit0 določa način delovanja pomnilniškega pospeševalnika.

Ostali biti v tem registru niso pomembni. Pomnilniški pospeševalnik je izklopljen, kadar sta bit1 in bit0 enaka nič `0b00`. Kombinacija `0b10` pomeni, da je pomnilniški pospeševalnik vklopljen, kombinacija `0b01` pa, da je vklopljen le deloma. Delni vklop v grobem pomeni, da se pomnilniški pospeševalnik ob vsakem branju konstantnih podatkov, ali nezaporednem branju (vejitve ali skoki) ukazov, znova napolni, zaradi česar mora centralno procesno jedro malce počakati. Ob polnem delovanju se pomnilniški pospeševalnik znova napolni le takrat, kadar je to neizogibno. Deluje podobno kot neke vrste predpomnilnik (angl. cache), kar hkrati pomeni nedeterministično delovanje (ista programska koda ob izvršitvi ne bo vedno porabila enako število strojnih ciklov). Kombinacija `0b11` je prepovedana.

MAMTIM MAM TIMing register (naslov: `0xe01fc004`)
Register z vrednostjo najnižjih treh bitov (bit2, bit1 in bit0) podaja število strojnih ciklov, ki so potrebni za eno polnjenje medpomnilnika pomnilniškega pospeševalnika. Ostali biti v tem registru nimajo pomena. Kombinacija `0b000` ni dovoljena. Torej je mogoče medpomnilnik pomnilniškega pospeševalnika napolniti v enem do sedmih strojnih ciklih. Dostopni čas do pomnilnika flash je 50ns. To pomeni, da mora biti za frekvence urinega signala med 20MHz $< f_{clk} \leq$ 40MHz število strojnih ciklov vsaj dva, ter za 40MHz $< f_{clk} \leq$ 60MHz vsaj tri. Ob spreminjanju vrednosti v tem registru mora biti pomnilniški pospeševalnik izklopljen (*MAMCR*).

Primer postavitve pomnilniškega pospeševalnika podaja izvorna koda spodaj.

Pospeševalnik je najprej izklopljen, nakar je pred končnim vklopom postavljena konstanta v registru *MAMTIM*.

```
/* Parameters */
          .equ  mam_cyc,     0x02
          .equ  mam_en,      0x02
/* Constant */
          .equ  mam_dis,     0x00
/* Registers */
          .equ  mamcr,       0xe01fc000
          .equ  mamtim,      0xe01fc004

          .code 32

/* Program code */
          .text
/* Memory acceleration module (MAM) initialisation */
mam_init: stmfd sp!, {r0-r2}
          ldr   r0, =mamcr
          mov   r1, #mam_dis
          str   r1, [r0]
          ldr   r1, =mamtim
          mov   r2, #mam_cyc
          str   r2, [r1]
          mov   r1, #mam_en
          str   r1, [r0]
          ldmfd sp!, {r0-r2}
          mov   pc, lr
```

Za razlago zbirniških ukazov v zgornjem primeru glej dodatek C.
In še ekvivalentna koda v programskem jeziku C. Funkcija *mam_init()* kot argu-

ment prejme nastavljeno frekvenco ure f_{clk} v MHz (glej dodatek B.2.1) in temu primerno postavi število ciklov enega polnjenja medpomnilnika pospeševalnika.

```
#define mam_disable 0x00
#define mam_enable  0x02
// Registers
#define MAMCR  (*((volatile unsigned long *)0xe01fc000))
#define MAMTIM (*((volatile unsigned long *)0xe01fc004))

// Memory acceleration module (MAM) initialisation
// clock_mhz ... clock rate in MHz [12,24,36,48,60]
void mam_init(int clock_mhz) {
    MAMCR = mam_disable;
    switch(clock_mhz) {
    case 12:
        MAMTIM = 1;
    case 24:
    case 36:
        MAMTIM = 2;
    case 48:
    case 60:
        MAMTIM = 3;
    }
    MAMCR = mam_enable;
}
```

B.4 Vektorski nadzornik prekinitev

To je vezje [11], ki skrbi za vrstni red izvajanja prekinitvenih programov. Vsako prekinitev razvrsti v eno od treh kategorij. In sicer *FIQ* (Fast Interrupt reQuest), *IRQ* (InteRrupt reQuest) in nevektorski *IRQ*.

Najvišjo prioriteto izvajanja imajo prekinitve *FIQ*. V primeru, da hkrati prispe več prekinitev, ki so vse opredeljene kot *FIQ*, potem o tem, katera bo prej na vrsti, odloča programska koda, oziroma načrtovalec programske opreme. Nadzornik prekinitev za prekinitve *FIQ* nima posebnega prioritetnega vezja. Zato imamo navadno le eno prekinitev označeno kot *FIQ*, ki je vedno takoj na vrsti.

Ob prekinitvi *FIQ* se izvajanje programa nadaljuje na naslovu `0x0000001c`, kjer je prekinitveni naslov *FIQ* (glej dodatek C.3).

Po prekinitvah *FIQ* so po prioriteti na vrsti vektorske prekinitve *IRQ*, za katere ima nadzornik prekinitev prioritetno vezje. Mogočih je največ 16 po prioriteti razvrščenih vektorskih prekinitev *IRQ*. Če določena prekinitev *IRQ* ni opisana kot vektorska, je prioritetno vezje ne prepozna. Takšna prekinitev ima najnižjo prioriteto in jo imenujemo nevektorska prekinitev *IRQ*. Ob prekinitvi *IRQ* se izvajanje programa nadaljuje na naslovu `0x00000018`, kjer je prekinitveni naslov *IRQ* (glej dodatek C.3).

		bit21	**bit20**
		A/D1	BOD
bit19	**bit18**	**bit17**	**bit16**
I^2C1	A/D0	EINT3	EINT2
bit15	**bit14**	**bit13**	**bit12**
EINT1	EINT0	RTC	PLL
bit11	**bit10**	**bit9**	**bit8**
SPI1	SPI0	I^2C0	PWM0
bit7	**bit6**	**bit5**	**bit4**
UART1	UART0	Timer1	Timer0
bit3	**bit2**		**bit0**
ARMcore1	ARMcore0		WDT

Tabela B.3: Pomen bitov v registrih vektorskega nadzornika prekinitev

Katera prekinitev je določena kot *FIQ*, *IRQ*, oziroma nevektorski *IRQ*, in kakšne so prioritetne relacije med vektorskimi prekinitvami *IRQ*, povemo z nastavitvami v 32-bitnih registrih, ki bodo opisani v nadaljevanju. Za vse registre, za katere ni posebej povedano drugače, velja razpored bitov v tabeli B.3.

VICSoftInt Software Interrupt register (naslov: `0xfffff018`)
Vpis enke na izbran bit v tem registru povzroči zahtevo po proženju pripadajoče prekinitve. Na ta način lahko vsako prekinitev zahtevamo programsko, ne da bi

se zares zgodila. Bite v tem registru postavljamo na nič s pisanjem v register *VICSoftIntClear*.

VICSoftIntClear Software Interrupt Clear register (naslov: `0xffffff01c`)
Vpis enke na izbran bit v tem registru povzroči vpis ničle na pripadajoč bit v registru *VICSoftInt*. Tako umaknemo zahtevo po umetnem proženju izbrane prekinitve.

VICRawIntr Raw Interrupt status register (naslov: `0xffffff008`)
Register podaja trenutno stanje zahtev po prekinitvah, ne glede na to ali je zahteva prava, ali programska (*VICSoftInt*).

VICIntEnable Interrupt Enable register (naslov: `0xffffff010`)
Vpis enke na izbran bit v tem registru omogoči izvajanje pripadajoče prekinitve, ko se pojavi zahteva po njej. Zahteva po prekinitvi je lahko prava ali programska (*VICSoftInt*). Prekinitve, katerih biti so enaki nič, so onemogočene in se tako ne bodo izvedle, čeprav je zahteva podana. Bite v tem registru postavljamo na nič s pisanjem v register *VICIntEnClear*.

VICIntEnClear Interrupt Enable Clear register (naslov: `0xffffff014`)
Vpis enke na izbran bit v tem registru povzroči vpis ničle na pripadajoč bit v registru *VICIntEnable*. Tako onemogočimo izvajanje izbrane prekinitve.

VICIntSelect Interrupt Select register (naslov: `0xffffff00c`)
Register določa vrsto posamezne prekinitve. Vpis enke na izbran bit opredeli pripadajočo prekinitev kot *FIQ*. In obratno, vpis ničle razvrsti pripadajočo prekinitev kot *IRQ*.

VICIRQStatus IRQ Status register (naslov: `0xffffff000`)
Register podaja trenutno stanje zahtev po omogočenih prekinitvah *IRQ*, torej po tistih prekinitvah *IRQ*, ki čakajo na dejansko izvršitev, glede na svojo prioriteto. Stanje v registru ne razlikuje med vektorskimi in nevektorsko prekinitvijo *IRQ*. V primeru večih nevektorskih prekinitev *IRQ* ta register podaja informacijo o tem, katera izmed njih je trenutno prispela.

VICFIQStatus FIQ Status register (naslov: `0xffffff004`)
Register podaja trenutno stanje zahtev po omogočenih prekinitvah *FIQ*, torej po

tistih prekinitvah *FIQ*, ki čakajo na dejansko izvršitev. Glede na vsebino tega registra se načrtovalec programske opreme odloča v primeru večih prekinitev *FIQ*.

VICVectCntl0-15 Vector Control registers 0 - 15
(naslovi: `0xfffff200`, `0xfffff204` ... `0xfffff23c`)
Registri podajajo posamezne vektorske prekinitve *IRQ* in njihovo prioriteto. Vektorska prekinitev *IRQ* z najvišjo prioriteto je definirana v registru *VICVectCntl0*, z najnižjo pa v *VICVectCntl15*. Biti bit0 do bit4 določajo kateremu izvoru prekinitve pripada posamezen register. Bit5 pa določa ali naj se ta izvor obravnava kot vektorska prekinitev *IRQ* ali ne. Ostali biti nimajo pomena. Če se nek izvor prekinitve ne obravnava kot vektorska prekinitev *IRQ*, potem to ne pomeni, da je ta prekinitev onemogočena, ampak le, da se bo izvajala kot nevektorska prekinitev *IRQ*. Primer: *VICVectCntl0* = `0bx...x100100`, prekinitev Timer0 (4 = `0b00100`) se obravnava kot vektorska prekinitev *IRQ* z najvišjo prioriteto.

VICVectAddr0-15 Vector Address registers 0 - 15
(naslovi: `0xfffff100`, `0xfffff104` ... `0xfffff13c`)
Vsak izmed registrov podaja naslov začetka prekinitvene kode za pripadajočo vektorsko prekinitev *IRQ*.

VICDefVectAddr Default Vector Address register (naslov: `0xfffff034`)
Podaja naslov začetka prekinitvene kode za nevektorsko prekinitev *IRQ*.

VICVectAddr Vector Address register (naslov: `0xfffff030`)
Ko se pojavi prekinitev *IRQ*, prioritetno vezje izbere tisto z najvišjo prioriteto. Naslov začetka pripadajoče prekinitvene kode, ki se nahaja v ustreznem registru *VICVectAddr*x, se prepiše v ta register. V primeru, da nobena izmed vektorskih prekinitev *IRQ* ni enaka zahtevani, se v ta register prepiše vsebina registra *VICDefVectAddr*, ki podaja začetek prekinitvene kode za nevektorsko prekinitev *IRQ*.
V ta register navadno ne pišemo, iz njega le preberemo naslov prekinitvene kode, ki jo bomo izvedli. Pisanje v register ne vpliva na njegovo vsebino, a vseeno postavi prioritetno vezje v začetno stanje, pripravljeno za izbiranje nove

prekinitve z najvišjo prioriteto. Zato je potrebno pred koncem prekinitvene kode v ta register nekaj (karkoli) zapisati.

VICProtection Protection enable register (naslov: `0xfffff020`)
Če je bit0 v tem registru postavljen na ena, potem lahko registre vektorskega nadzornika prekinitev dosežemo le iz privilegiranih načinov delovanja (glej dodatek C.3). V nasprotnem primeru, ko je bit0 enak nič, lahko registre vektorskega nadzornika prekinitev dosežemo vedno. Ostali biti v registru nimajo pomena.

Primer postavitve vektorskega nadzornika prekinitev, ki je uporabljena za proženje ravrščevalnika opravil *sch_int* na strani 48.
Podprogram *timer0_int* definira ravrščevalnik opravil *sch_int* kot vektorsko prekinitev *IRQ* z najvišjo prioriteto (v našem primeru drugih prekinitev tako ali tako ni, tako da prioriteta pravzaprav ni pomembna), ki se zgodi na zahtevo časovnika Timer0.
Ko se prekinitev *IRQ* zgodi, se izvajanje programa nadaljuje na naslovu `0x00000018`, kjer je ukaz za skok na podprogram *irq*. V njem preberemo register *VICVectAddr*, ki podaja naslov pričetka prekinitvene kode (v našem primeru podprograma *sch_int*). Postavljanje prioritetnega vezja v začetno stanje se izvrši s pisanjem v register *VICVectAddr*. Isto se ob vsaki prekinitvi zgodi tudi znotraj ravrščevalnika opravil *sch_int* (stran 48), tako da je prioritetno vezje pripravljeno na naslednjo prekinitev.

```
/* Constants */
            .equ  intselect_val,       0x00
            .equ  timer0,              0x10
            .equ  int_en,              0x20
            .equ  t0_num,              0x04
            .equ  word_len,            0x04
/* Registers */
            .equ  vicintselect,        0xfffff00c
            .equ  vicintenable,        0xfffff010
            .equ  vicvectaddr,         0xfffff030
            .equ  vicvectaddr0,        0xfffff100
            .equ  vicvectcntl0,        0xfffff200

            .code 32
```

```
/* Startup code */
            .text
            ldr   pc, =irq             /* at 0x00000018 */

/* Interrupt request interrupt service routine (ISR) */
irq:        stmfd sp!, {r0, lr}
            ldr   lr, =irq_end
            ldr   r0, =vicvectaddr
            ldr   pc, [r0]
irq_end:    ldmfd sp!, {r0, lr}
            subs  pc, lr, #word_len

/* Soubroutine sets timer0 as irq in */
/* vector interrupt controller (VIC) */
timer0_int: stmfd sp!, {r0-r1}
            ldr   r0, =vicintselect
            ldr   r1, =intselect_val
            str   r1, [r0]
            ldr   r0, =vicvectaddr0
            ldr   r1, =sch_int
            str   r1, [r0]
            ldr   r0, =vicvectcntl0
            mov   r1, #(int_en|t0_num)
            str   r1, [r0]
            ldr   r0, =vicintenable
            ldr   r1, =timer0
            str   r1, [r0]
            ldr   r0, =vicvectaddr
            str   r1, [r0]
            ldmfd sp!, {r0-r1}
            mov   pc, lr
```

Za razlago zbirniških ukazov v zgornjem primeru glej dodatek C.
Podobno je mogoče narediti tudi v programskem jeziku C. Ob prekinitvi *IRQ* se požene funkcija *irq()* [16], ki naprej pokliče funkcijo, katere naslov se nahaja v registru *VICVectAddr*. V našem primeru je to ravrščevalnik opravil *sch_int()*.

Želene nastavitve vektorskega nadzornika prekinitev opravi funkcija *vic_init()*. Argumenti funkcije določajo vrsto, prioriteto in funkcijo posamezne prekinitve.

```
typedef void (* voidfuncptr)();
// Register
#define VICIntSelect   (*((volatile unsigned long *)0xfffff00c))
#define VICIntEnable   (*((volatile unsigned long *)0xfffff010))
#define VICVectAddr    (*((volatile unsigned long *)0xfffff030))
#define VICDefVectAddr (*((volatile unsigned long *)0xfffff034))
#define VICVectAddr    (*((volatile unsigned long *)0xfffff030))
#define VICVectAddr0   (*((volatile unsigned long *)0xfffff100))
    ...
#define VICVectAddr15  (*((volatile unsigned long *)0xfffff13c))
#define VICVectCntl0   (*((volatile unsigned long *)0xfffff200))
    ...
#define VICVectCntl15  (*((volatile unsigned long *)0xfffff23c))

// Interrupt request interrupt service routine (ISR)
void irq(void) __attribute__((interrupt("IRQ")));
void irq(void) {
    (*((voidfuncptr)VICVectAddr))();
}

// Vector interrupt controller (VIC) initialisation
// fiq       ... FIQ mask determines which interrupts are FIQ
// fiq       ... IRQ mask determines which interrupts are IRQ
// function  ... array of pointers to ISRs for each sloted IRQ
// interrupt ... array of sloted IRQs
// def       ... pointer to unsloted ISR
void vic_init(int fiq, int irq, voidfuncptr *function,
    int *interrupt, voidfuncptr def) {
    int i, j;
    VICIntSelect = fiq;
    VICVectAddr0 = (int)function[0];
        ...
    VICVectAddr15 = (int)function[15];
    for(i = interrupt[0], j = -1; i; i = i >> 1, j = j + 1);
    if(j > -1) VICVectCntl0 = j | 0x00000020;
```

```
    else VICVectCntl0 = 0;
        ...
    for(i = interrupt[15], j = -1; i; i = i >> 1, j = j + 1);
    if(j > -1) VICVectCntl15 = j | 0x00000020;
    else VICVectCntl15 = 0;
    VICDefVectAddr = (int)def;
    VICIntEnable = fiq | irq;
    VICVectAddr = 0;
}
```

B.5 Časovnik

Časovnik največkrat uporabljamo, kot pove že ime, za merjenje časa. Čas merimo tako, da štejemo urine cikle na vodilu VPB (glej dodatek B.2.2). Ena perioda signala ure VPB predstavlja časovni kvant, oziroma mejo ločljivosti v časovnem prostoru. Časov manjših od časovnega kvanta ne moremo meriti.

Časovnik lahko uporabimo tako, da nam ob preteku vnaprej določenega časa, sproži nek dogodek. Primer: po preteku 10s časovnik poda zahtevo po prekinitvi; ali: po preteku 10s časovnik dvigne napetost na pripadajočem izhodnem pinu z nizkega na visok nivo - iz logične ničle v enico. Ta način delovanja precej spominja na budilko. Vnaprej nastavimo trenutek, ko želimo, da budilka zazvoni. Po preteku tega časa budilka sproži svoj dogodek, to je zvonjenje (ki nam skrajša najlepši del sna ...).

Drug način uporabe časovnika je ravno obraten. Ob nekem dogodku časovnik zabeleži čas, ko se je dogodek zgodil. Primer: ko se signal na pripadajočem vhodnem pinu dvigne z nizkega na visok nivo, časovnik shrani trenutno stanje števnika urinih ciklov na vodilu VPB. Analogijo iz vsakdanjosti sedaj predstavlja štoparica. Ko športnik doseže cilj (dogodek), štoparica zabeleži trenutek prestopa ciljne črte. Dosežen rezultat lahko tekmovalec odčita kadarkoli kasneje (ko pride do sape ...).

Časovnik je mogoče uporabljati tudi kot števnik dogodkov. Čas v tem primeru ne igra nobene vloge. Primer: štejemo prehode signala iz visokega na nizek napetostni nivo na pripadajočem vhodnem pinu.

Časovnik, oziroma števnik, ni del centralnega procesnega jedra ARM7. Je periferna enota mikrokrmilnika LPC2138. Mikrokrmilnik LPC2138 ima vgrajena dva časovnika, in sicer Timer0 in Timer1. Na tem mestu bomo na kratko opisali le Timer0, pa še tega le, ko ga uporabljamo kot budilko. Časovnik Timer1 je časovniku Timer0 identičen. Vsi registri časovnika Timer1 se nahajajo na

naslovih registrov časovnika Timer0, ki jim prištejemo konstanto `0x00004000`. Podrobnejši opis delovanja obeh časovnikov lahko bralec najde v [5].

Registri, s katerimi nastavljamo delovanje časovnika Timer0, so naslednji:

T0PR Prescale Register (naslov: `0xe000400c`)
Register podaja največjo vrednost, ki jo lahko zavzame števnik *T0PC*. Vsakič, ko je števnik *T0PC* enak konstanti zapisani v tem registru, se v naslednjem ciklu postavi na nič.

T0PC Prescale Counter register (naslov: `0xe0004010`)
Prvostopenjski prosto tekoči 32-bitni števnik, ki se poveča za ena ob vsakem urinem impulzu na vodilu VPB. Ko števnik doseže svojo končno vrednost, ki je shranjena v registru *T0PR*, se v naslednjem ciklu postavi na nič.

T0TC Timer Counter register (naslov: `0xe0004008`)
Prosto tekoči 32-bitni števnik, ki se poveča za ena vsakič, ko prvostopenjski števnik doseže svojo končno vrednost ($T0PC = T0PR$). Oziroma z drugimi besedami, prosto tekoči števnik *T0TC* se poveča za ena na vsakih $T0PR + 1$ urinih ciklov vodila VPB.

T0CTCR CounT Control Register (naslov: `0xe0004070`)
Register določa način uporabe časovnikovih števnikov *T0PC* in *T0TC*. Kot je bilo že povedano, lahko števnika štejeta čas, oziroma urine cikle na vodilu VPB, ali pa je števnik *T0TC* uporabljen kot števnik dogodkov, na primer prvih front signala na pripadajočem vhodnem pinu. Če hočemo števnika uporabiti za mer-

jenje časa, potem morata biti bit0 in bit1 tega registra postavljena na nič. Ostali biti v tem primeru niso pomembni.

T0TCR Timer Control Register (naslov: `0xe0004004`)
Odloča o tem, ali bosta števnika *T0PC* in *T0TC* štela urine cikle na VPB vodilu ali ne, oziroma bosta ustavljena. Pomembna sta le dva bita registra, in sicer:

bit0 če je bit0 enak nič, sta oba števnika ustavljena in ne štejeta urinih impulzov; v nasprotnem primeru, ko je bit0 enak ena, se štetje odvija normalno

bit1 ko je bit1 postavljen, se oba števnika ob prvi naslednji naraščajoči fronti urinega signala vodila VPB naenkrat postavita na nič; dokler je bit1 postavljen, števnika ostaneta na ničli

T0MR0, T0MR1, T0MR2 in **T0MR3** Match Registers
(naslovi: `0xe0004018`, `0xe000401c`, `0xe0004020` in `0xe0004024`)
Konstante zapisane v teh štirih registrih se ves čas primerjajo s števnikom *T0TC*. Ko je števnik enak kateri izmed konstant, se izvrši dejanje, ki je določeno v registru *T0MCR*.

T0MCR Match Control Register (naslov: `0xe0004014`)
Določa dejanje, ki naj se izvrši ob dogodku *T0TC* = *T0MR0*, ali dogodku *T0TC* = *T0MR1*, ali *T0TC* = *T0MR2*, ali *T0TC* = *T0MR3*. Ob vsakem izmed naštetih štirih dogodkov lahko podamo zahtevo po prekinitvi, ustavimo štetje, resetiramo števnik *T0TC*, ali pa ne naredimo nič. Dejanja je mogoče kombinirati. Tako

lahko na primer podamo zahtevo po prekinitvi in hkrati ustavimo štetje. Biti v registru imajo naslednje vloge:

bit0 če je postavljen, je ob dogodku *T0TC = T0MR0* zahtevana prekinitev

bit1 če je postavljen, se ob dogodku *T0TC = T0MR0* števnik *T0TC* resetira

bit2 če je postavljen, se ob dogodku *T0TC = T0MR0* ustavi štetje (bit0 registra *T0TCR* se postavi na nič)

bit3 če je postavljen, je ob dogodku *T0TC = T0MR1* zahtevana prekinitev

bit4 če je postavljen, se ob dogodku *T0TC = T0MR1* števnik *T0TC* resetira

bit5 če je postavljen, se ob dogodku *T0TC = T0MR1* ustavi štetje (bit0 registra *T0TCR* se postavi na nič)

bit6 če je postavljen, je ob dogodku *T0TC = T0MR2* zahtevana prekinitev

bit7 če je postavljen, se ob dogodku *T0TC = T0MR2* števnik *T0TC* resetira

bit8 če je postavljen, se ob dogodku *T0TC = T0MR2* ustavi štetje (bit0 registra *T0TCR* se postavi na nič)

bit9 če je postavljen, je ob dogodku *T0TC = T0MR3* zahtevana prekinitev

bit10 če je postavljen, se ob dogodku *T0TC = T0MR3* števnik *T0TC* resetira

bit11 če je postavljen, se ob dogodku *T0TC = T0MR3* ustavi štetje (bit0 registra *T0TCR* se postavi na nič)

T0IR Interrupt Register (naslov: `0xe0004000`)
Ko časovnik Timer0 zahteva prekinitev, lahko v tem registru izvemo, kateri

dogodek je temu vzrok. Zahteva po prekinitvi postavi ustrezen bit tega registra. Biti imajo naslednji pomen:

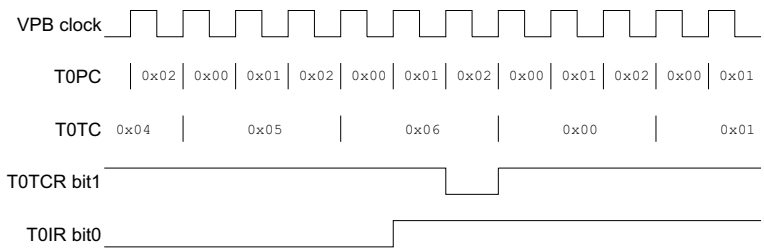
bit0 prekinitev je zahtevana, ker se je zgodil dogodek *T0TC = T0MR0*

bit1 prekinitev je zahtevana, ker se je zgodil dogodek *T0TC = T0MR1*

bit2 prekinitev je zahtevana, ker se je zgodil dogodek *T0TC = T0MR2*

bit3 prekinitev je zahtevana, ker se je zgodil dogodek *T0TC = T0MR3*

Pisanje ničel v register nima učinka. Posamezen bit postavimo na nič tako, da na njegovo mesto zapišemo enko.
Dogodek *T0TC = T0MR*x sam po sebi še ne sproži zahteve po prekinitvi. Časovnik poda zahtevo po prekinitvi le, če je tako določeno v registru *T0MCR*.



Slika B.4: Časovni potek štetja časovnika

Za primer si poglejmo delovanje časovnika, ki šteje urine cikle na vodilu VPB (slika B.4). Povečevanje števnikov se zgodi ob prvih frontah urinega signala. Konstanti sta postavljeni na *T0PR* = `0x00000002` in *T0MR0* = `0x00000006`. V *T0MCR* registru pa je ob dogodku *T0TC = T0MR0* določeno, naj se poda zahteva za prekinitev, števnika pa naj se resetirata.
Primer postavitve časovnika, ki je uporabljena za proženje ravrščevalnika opravil *sch_int* na strani 48. Za razlago zbirniških ukazov glej dodatek C.

```
/* Parameters */
            .equ  prescale0,      0x00000000
            .equ  match0,         0x002dc6bf
```

```
/* Constants */
            .equ  cnt_reset,      0x02
            .equ  t0_ints,        0xff
            .equ  t0mcr_val,      0x03
            .equ  t0ctcr_val,     0x00
/* Registers */
            .equ  t0ir,           0xe0004000
            .equ  t0tcr,          0xe0004004
            .equ  t0pr,           0xe000400c
            .equ  t0mcr,          0xe0004014
            .equ  t0mr0,          0xe0004018
            .equ  t0ctcr,         0xe0004070

            .code 32

/* Program code */
            .text
/* Reset and configure timer0 subroutine */
timer0_init: stmfd sp!, {r0-r1}
            ldr   r0, =t0tcr
            mov   r1, #cnt_reset
            str   r1, [r0]
            ldr   r0, =t0ir
            mov   r1, #t0_ints
            str   r1, [r0]
            ldr   r0, =t0pr
            ldr   r1, =prescale0
            str   r1, [r0]
            ldr   r0, =t0mr0
            ldr   r1, =match0
            str   r1, [r0]
            ldr   r0, =t0mcr
            ldr   r1, =t0mcr_val
            str   r1, [r0]
            ldr   r0, =t0ctcr
            mov   r1, #t0ctcr_val
            str   r1, [r0]
            ldmfd sp!, {r0-r1}
            mov   pc, lr
```

Podprogram *timer0_ init* najprej postavi oba števnika na nič. Nato umakne vse zahteve po prekinitvah, ki jih je časovnik morebiti izdal že prej. Nastavimo še konstanti v registrih *T0PR* in *T0MR0*, ter povemo, naj se ob dogodku *T0TC* = *T0MR0* poda zahteva po prekinitvi, števnika pa naj se postavita na nič. Časovnik meri čas, oziroma šteje urine cikle na vodilu VPB, kar dosežemo z vpisom ničel v register *T0CTCR*. Oba števnika *T0PC* in *T0TC* imata po izvršitvi podprograma *timer0_ init* vrednost nič. Štetje se prične s postavitvijo bita bit1 v registru *T0TCR* na nič, kar se zgodi v podprogramu *sch_ on* (stran 54).

V primeru sta postavljeni konstanti *T0PR* = `0x00000000` in *T0MR0* = `0x002dc6bf`. Če sta fazno sklenjena zanka in delilnik VPB nastavljena, kot je opisano v primerih v odstavkih B.2.1 in B.2.2, potem en urin cikel na vodilu VPB traja $1/3\mu$s. To pomeni, da bo razvrščevalnik klican enkrat na sekundo ($f_{vpb} \div ((T0PR + 1) \times (T0MR0 + 1)) = 1\mathrm{s}^{-1}$).

V programskem jeziku C inicializacijo časovnika Timer0 opravi funkcija *timer0_ init()*. Argumenti funkcije podajajo nastavite časovnika, kot so primerjalne vrednosti prosto tekočega in prvostopenjskega števnika, opis dogodka, ki se zgodi ob izpolnjenem pogoju, in način štetja. Po izvršitvi funkcije *timer0_ init()* se štetje še ne prične, saj je onemogočeno (register *T0TCR*). To se zgodi ob zagonu operacijskega sistema v funkciji *sch_ on()* na strani 56.

```
#define counter_reset 0x00000002
// Bit masks in T0IR register
#define mr0 0x00000001
#define mr1 0x00000002
#define mr2 0x00000004
#define mr3 0x00000008
#define cr0 0x00000010
#define cr1 0x00000020
#define cr2 0x00000040
#define cr3 0x00000080
// Registers
#define T0IR   (*((volatile unsigned long *)0xe0004000))
#define T0TCR  (*((volatile unsigned long *)0xe0004004))
#define T0PR   (*((volatile unsigned long *)0xe000400c))
#define T0MCR  (*((volatile unsigned long *)0xe0004014))
#define T0MR0  (*((volatile unsigned long *)0xe0004018))
#define T0MR1  (*((volatile unsigned long *)0xe000401c))
#define T0MR2  (*((volatile unsigned long *)0xe0004020))
#define T0MR3  (*((volatile unsigned long *)0xe0004024))
```

```
#define T0CTCR (*((volatile unsigned long *)0xe0004070))

// Reset and configure timer0
// prescale ... maximum prescale counter value
// match    ... array of match values
// control  ... match control
// count    ... count control
void timer0_init(int prescale, int *match, int control,
    int count) {
    T0TCR = counter_reset;
    T0IR = mr0 | mr1 | mr2 | mr3 | cr0 | cr1 | cr2 | cr3;
    T0PR = prescale;
    T0MR0 = match[0];
    T0MR1 = match[1];
    T0MR2 = match[2];
    T0MR3 = match[3];
    T0MCR = control;
    T0CTCR = count;
}
```

Sledi še primer postavitve prosto tekočega časovnika Timer1, ki je uporabljen v poglavju 3.2 za merjenje dolžine posameznih opravil, oziroma funkcij.

```
/* Constants */
            .equ  prescale1,      0xffffffff
            .equ  cnt_reset,      0x02
            .equ  t1_ints,        0xff
            .equ  t1mcr_val,      0x00
            .equ  t1ctcr_val,     0x00
/* Registers */
            .equ  t1ir,           0xe0008000
            .equ  t1tcr,          0xe0008004
            .equ  t1pr,           0xe000800c
            .equ  t1mcr,          0xe0008014
            .equ  t1ctcr,         0xe0008070

            .code 32
```

```
/* Program code */
            .text
/* Reset and configure timer1 as free running counter */
timer1_init: stmfd sp!, {r0-r1}
             ldr   r0, =t1tcr
             mov   r1, #cnt_reset
             str   r1, [r0]
             ldr   r0, =t1ir
             mov   r1, #t1_ints
             str   r1, [r0]
             ldr   r0, =t1pr
             ldr   r1, =prescale1
             str   r1, [r0]
             ldr   r0, =t1mcr
             ldr   r1, =t1mcr_val
             str   r1, [r0]
             ldr   r0, =t1ctcr
             mov   r1, #t1ctcr_val
             str   r1, [r0]
             ldmfd sp!, {r0-r1}
             mov   pc, lr
```

Podprogram *timer1_ init* postavi oba števnika na nič, umakne vse zahteve po prekinitvah, ter pove naj časovnik ne proži nobenih dogodkov (prekinitve, postavitev števnikov na nič ...). Zagon (in nato ustavitev) časovnika je izvršena kasneje, ob začetku in koncu merjenja časovnega intervala.

Enako je seveda mogoče doseči v programskem jeziku C. Funkcija *timer1_ init()* je povsem enaka funkciji *timer0_ init()* in služi za inicializacijo časovnika Timer1. Če želimo, da je Timer1 prosto tekoč, morajo biti argumenti funkcije naslednji: *prescale* = `0xffffffff`, *match* je poljubno polje štirih 32-bitnih števil, *control* = `0x00000000` in *count* = `0x00000000`.

```
#define counter_reset 0x00000002
// Bit masks in T1IR register
#define mr0 0x00000001
#define mr1 0x00000002
#define mr2 0x00000004
#define mr3 0x00000008
#define cr0 0x00000010
```

```
#define cr1 0x00000020
#define cr2 0x00000040
#define cr3 0x00000080
// Registers
#define T1IR   (*((volatile unsigned long *)0xe0008000))
#define T1TCR  (*((volatile unsigned long *)0xe0008004))
#define T1PR   (*((volatile unsigned long *)0xe000800c))
#define T1MCR  (*((volatile unsigned long *)0xe0008014))
#define T1MR0  (*((volatile unsigned long *)0xe0008018))
#define T1MR1  (*((volatile unsigned long *)0xe000801c))
#define T1MR2  (*((volatile unsigned long *)0xe0008020))
#define T1MR3  (*((volatile unsigned long *)0xe0008024))
#define T1CTCR (*((volatile unsigned long *)0xe0008070))

// Reset and configure timer1
// prescale ... maximum prescale counter value
// match    ... array of match values
// control  ... match control
// count    ... count control
void timer1_init(int prescale, int *match, int control,
    int count) {
    T1TCR = counter_reset;
    T1IR = mr0 | mr1 | mr2 | mr3 | cr0 | cr1 | cr2 | cr3;
    T1PR = prescale;
    T1MR0 = match[0];
    T1MR1 = match[1];
    T1MR2 = match[2];
    T1MR3 = match[3];
    T1MCR = control;
    T1CTCR = count;
}
```

B.6 Ura realnega časa

Ura realnega časa je pravzaprav poseben časovnik, ki meri čas v človeku poznanih časovnih enotah. Tako nimamo več opraviti s številom urinih ciklov na vodilu VPB, ampak s sekundami, minutami itd. Za svoje delovanje potrebuje ura realnega časa signal frekvence 32kHz, ki ga zagotovimo s posebnim zunanjim

32kHz kristalom, ali s pomočjo deljenja urinega signala na vodilu VPB. V prvem primeru ura realnega časa izpolnjuje svoj namen in meri čas povsem neodvisno od takta procesnega jedra f_{clk} oziroma takta periferije f_{vpb}. Z neprekinjenim baterijskim napajanjem lahko ob zelo nizki porabi energije ura realnega časa samostojno teče tudi, kadar mikrokrmilniku izklopimo napajanje. Za napajanje ure realnega časa je v Philipsovem mikrokrmilniku LPC2138 namenjen poseben pin V_{bat}.

Registri, s katerimi dostopamo do ure realnega časa so naslednji:

SEC SEConds counter (naslov: `0xe0024020`)
Števnik sekund. V registru se nahaja trenutno število sekund.

MIN MINutes register (naslov: `0xe0024024`)
Števnik minut. V registru se nahaja trenutno število minut.

HOUR HOURs register (naslov: `0xe0024028`)
Števnik ur. V registru se nahaja trenutno število ur.

DOM Day Of Month register (naslov: `0xe002402c`)
Števnik dni v mesecu. V registru se nahaja številka dneva v mesecu. Število dni v posameznem mesecu, kakor tudi prestopna leta med leti 1901 in 2099 se pravilno upoštevajo. Ob zapisu nove vrednosti v register se doslednost datuma ne preverja.

DOW Day Of Week register (naslov: `0xe0024030`)
Števnik dni v tednu. V registru se nahaja številka dneva v tednu med nič in šest. Ob zapisu nove vrednosti v register se doslednost datuma ne preverja.

DOY Day Of Year register (naslov: `0xe0024034`)
Števnik dni v letu. V registru se nahaja številka dneva v letu. Prestopna leta

med leti 1901 in 2099 se pravilno upoštevajo. Ob zapisu nove vrednosti v register se doslednost datuma ne preverja.

MONTH MONTHs register (naslov: `0xe0024038`)
Števnik mesecev. V registru se številka meseca. Ob zapisu nove vrednosti v register se doslednost datuma ne preverja.

YEAR YEARs register (naslov: `0xe002403c`)
Števnik let. V registru se nahaja številka leta. Ob zapisu nove vrednosti v register se doslednost datuma ne preverja.

CTIME0 Consolidated TIME register 0 (naslov: `0xe0024014`)
Register lahko le beremo in vsebuje vrednosti registrov *SEC*, *MIN*, *HOUR* in *DOW*, in sicer:

bit5 do **bit0** vrednost registra *SEC*,

bit13 do **bit8** vrednost registra *MIN*,

bit20 do **bit16** vrednost registra *HOUR* in

bit26 do **bit24** vrednost registra *DOW*.

CTIME1 Consolidated TIME register 1 (naslov: `0xe0024018`)
Register lahko le beremo in vsebuje vrednosti registrov *DOM*, *MONTH* in *YEAR*, in sicer:

bit4 do **bit0** vrednost registra *DOM*,

bit11 do **bit8** vrednost registra *MONTH* in

bit27 do **bit16** vrednost registra *YEAR*.

CTIME2 Consolidated TIME register 2 (naslov: `0xe002401c`)
Register lahko le beremo in vsebuje vrednost registra *DOY*.

CIIR Counter Increment Interrupt Register (naslov: `0xe002400c`)
Postavljeni biti v registru določajo števnike, ob katerih spremembi je podana zahteva po prekinitvi. Posamezni biti imajo naslednji pomen:

bit0 prekinitev zahtevana ob spremembi števnika *SEC*,

bit1 prekinitev zahtevana ob spremembi števnika *MIN*,

bit2 prekinitev zahtevana ob spremembi števnika *HOUR*,

bit3 prekinitev zahtevana ob spremembi števnika *DOM*,

bit4 prekinitev zahtevana ob spremembi števnika *DOW*,

bit5 prekinitev zahtevana ob spremembi števnika *DOY*,

bit6 prekinitev zahtevana ob spremembi števnika *MONTH* in

bit7 prekinitev zahtevana ob spremembi števnika *YEAR*.

ALSEC ALarm value for SEConds (naslov: `0xe0024060`)
Sekunda alarma.

ALMIN ALarm value for MINutes (naslov: `0xe0024064`)
Minuta alarma.

ALHOUR ALarm value for HOURs (naslov: `0xe0024068`)
Ura alarma.

ALDOM ALarm value for Day Of Month (naslov: `0xe002406c`)
Dan v mesecu alarma.

ALDOW ALarm value for Day Of Week (naslov: `0xe0024070`)
Dan v tednu alarma.

ALDOY ALarm value for Day Of Year (naslov: `0xe0024074`)
Dan v letu alarma.

ALMON ALarm value for MONths (naslov: `0xe0024078`)
Mesec alarma.

ALYEAR ALarm value for YEARs (naslov: `0xe002407c`)
Leto alarma.

AMR Alarm Mask Register (naslov: `0xe0024010`)
Postavljeni biti v registru določajo števnike, ki se ne primerjajo z alarmom. Zahteva po prekinitvi je podana, ko so vsi primerjani števniki enaki postavljenim vrednostim alarma. Posamezni biti imajo naslednji pomen:

bit0 če je postavljen, se števnik *SEC* ne primerja z *ALSEC*,

bit1 če je postavljen, se števnik *MIN* ne primerja z *ALMIN*,

bit2 če je postavljen, se števnik *HOUR* ne primerja z *ALHOUR*,

bit3 če je postavljen, se števnik *DOM* ne primerja z *ALDOM*,

bit4 če je postavljen, se števnik *DOW* ne primerja z *ALDOW*,

bit5 če je postavljen, se števnik *DOY* ne primerja z *ALDOY*,

bit6 če je postavljen, se števnik *MONTH* ne primerja z *ALMON* in

bit7 če je postavljen, se števnik *YEAR* ne primerja z *ALYEAR*.

Če je postavljenih vseh osem bitov, potem je alarm onemogočen in ne zahteva prekinitev.

ILR Interrupt Location Register (naslov: `0xe0024000`)
Bita v registru označujeta, kdo zahteva prekinitev. Bit bit0 se postavi, če je prekinitev zahtevana zaradi spremembe kateregakoli izmed števnikov, bit1 pa, če je prekinitev zahtevana ob primerjanju števnikov z alarmom. Postavljen bit odstranimo z zapisom enice na pripadajoče mesto. Pisanje ničle nima vpliva.

CTC Clock Tick Counter (naslov: `0xe0024004`)
To je 15-bitni števnik, ki šteje impulze signala frekvence 32kHz od nič do 32767. Ob vsakem prehodu z vrednosti 32767 na nič se spremeni vrednost registra *SEC*. Vrednost registra *CTC* lahko le beremo. Register postavimo na nič s postavitvijo ustreznega bita v registru *CCR*.

Kadar signal frekvence 32kHz (oziroma točneje 32768Hz) zagotavljamo s pomočjo deljenja urinega signala perifernega vodila f_{vpb}, je način deljenja podan v registrih *PREINT* in *PREFRAC*. Ker f_{vpb} ni nujno večkratnik frekvence

32768Hz, impulzi 32kHz signala niso nujno enako dolgi. Primer: signal frekvence f_{vpb} = 3MHz delimo tako, da 18112 period 32kHz signala traja po 92 impulzov f_{vpb}, preostalih 14656 pa po 91 impulzov f_{vpb}.

PREINT PREscaler INTeger register (naslov: `0xe0024080`)
Register se uporablja, kadar signal frekvence 32kHz zagotavljamo s pomočjo deljenja urinega signala perifernega vodila f_{vpb}. Njegova vrednost mora biti enaka $\lfloor f_{vpb} \div 32768 \rfloor - 1$, pri čemer znaka $\lfloor \rfloor$ pomenita zaokrožitev navzdol.

PREFRAC PREscaler FRACtion register (naslov: `0xe0024084`)
Register se uporablja, kadar signal frekvence 32kHz zagotavljamo s pomočjo deljenja urinega signala perifernega vodila f_{vpb}. Njegova vrednost mora biti enaka $f_{vpb} - \lfloor f_{vpb} \div 32768 \rfloor \times 32768 = f_{vpb} - (PREINT + 1) \times 32768$, pri čemer znaka $\lfloor \rfloor$ pomenita zaokrožitev navzdol.

CCR Clock Control Register (naslov: `0xe0024008`)
Biti v registru nadzorujejo štetje impulzov signala frekvence 32kHz. Posamezni biti imajo naslednji pomen:

bit0 omogoči štetje impulzov 32kHz signala,

bit1 postavi števnik CTC na nič in s tem začasno onemogoči štetje impulzov 32kHz signala in

bit4 postavljen določa uporabo zunanjega 32kHz kristala, drugače se uporabi deljen signal ure perifernega vodila f_{vpb}.

Sledi primer inicializacije ure realnega časa, ki prosto teče in ne zahteva prekinitev. Najprej onemogočimo zahteve po prekinitvah zaradi spremembe kateregakoli izmed števnikov, nato še zahtevo po prekinitvi ob primerjavi števnikov

z alarmom. Uro realnega časa omogočimo in povemo, da je signal frekvence 32kHz zagotovljen s posebnim zunanjim kristalom.

```
CIIR = 0x00000000;
AMR = 0x000000ff;
CCR = 0x00000011;
// Check date consistency
```

Podana inicializacija delovanja ure realnega časa ne zmoti, če le ta morda že teče. Zato se lahko izvede ob vsakem (ponovnem) vklopu napajanja. Datum oziroma vrednosti registrov *DOM*, *DOW*, *DOY*, *MONTH* in *YEAR* niso nujno dosledne, zato doslednost (ujemanje vrednosti registrov) eventualno preverimo.

Med zapisom novih vrednosti časa ali datuma moramo uro realnega časa začasno ustaviti. Ko zapisujemo novo vrednost časa, je smiselno postaviti na nič tudi števnik *CTC*. Razmere ponazarja sledeča koda:

```
CCR = CCR & 0xfffffffe | 0x00000002;
// Set time registers
CCR = CCR & 0xfffffffd | 0x00000001;

CCR = CCR & 0xfffffffe;
// Set date registers
CCR = CCR | 0x00000001;
```

Trenutno vrednost ure realnega časa lahko odčitamo iz števnikov *SEC*, *MIN*, *HOUR*, *DOM*, *DOW*, *DOY*, *MONTH* in *YEAR*, vendar navadno uporabljamo registre *CTIME0*, *CTIME1* in *CTIME2*. Ti nam omogočajo, da celoten datum in čas odčitamo le s tremi branji registrov. Tako je možnost, da se vrednost ure realnega časa med branjem spremeni, kar najmanjša. Da se nevarnosti spre-

membe ure realnega časa med branjem zagotovo izognemo, uporabimo večkratno branje, kot ga implementira dana koda:

```
int ctime0, ctime1[2], ctime2;
do {
    ctime1[0] = CTIME1;
    ctime0 = CTIME0;
    ctime2 = CTIME2;
    ctime1[1] = CTIME1;
} while(ctime1[0] != ctime1[1]);
```

Vrednost datuma preberemo pred in po branju časa. Če se datum med branjem časa ne spremeni, potem so prebrane vrednosti skladne.

B.7 Povezave mikrokrmilnika z zunanjimi napravami

Zunanje naprave v mikrokrmilniški sistem priključujemo preko vodil. Vendar predvsem v majhnih sistemih mikrokrmilniku pogosto dodajamo enostavne zunanje enote, kot so senzorji, tipke, prikazovalniki in podobno. Priključevanje posameznih enot na vodila ne bi bila optimalna, saj bi za vsako izmed njih potrebovali preveč dodatne elektronike (dekodirnik, tristanjski ojačevalnik ...). Zato na izbranem naslovu raje naredimo splošna vhodno izhodna vrata. Stanje vhodnih pinov vrat lahko mikrokrmilnik bere, medtem ko na izhodne pine lahko piše. Mikrokrmilniki imajo navadno ena ali več vzporednih splošnih vrat že vgrajenih, kar priključevanje enostavnih zunanjih enot še olajša. Smer posameznih pinov je navadno mogoče izbirati programsko.

Mikrokrmilnik LPC2138 ima vgrajenih dvoje splošnih vzporednih 32-bitnih vrat P0 in P1, ki pa nista popolni. Vrata P0 imajo dostopne vse pine, razen pina P0.24, pri vratih P1 pa je dostopnih le zgornjih 16 pinov od P1.16 do P1.31. Vse dostopne pine je mogoče nastaviti kot vhodne ali izhodne, razen pina P0.31, ki je vedno izhoden. Nastavitve posameznih pinov vrat P0 so podane z biti v registrih *PINSEL0* in *PINSEL1*, ki se nahajata na naslovih `0xe002c000` in `0xe002c004`. Nastavitev vrat P1 pa je določamo z registrom *PINSEL2* na

naslovu `0xe002c008`. Razlago pomena posameznih bitov v vseh treh registrih bralec najde v [5].

B.8 Zunanje prekinitve

Mnogokrat se pojavi potreba po prekinitvi ob določenem zunanjem dogodku. Na ta način zunanje naprave o dogodku obvestijo mikrokrmilnik, ki se lahko nanj odzove takoj, ko le ta nastopi. Zunanje prekinitve so lahko brezpogojne (angl. NMI - Non-Maskable Interrupt), kar pomeni, da se zunanja prekinitev ob dogodku vedno izvrši takoj. Pri milejši obliki je odločitev o izvršitvi zunanje prekinitve prepuščena mikrokrmilniku, oziroma programski opremi.

Prav tako kot časovnik tudi logično vezje, ki zahteva zunanje prekinitve, ni del centralnega procesnega jedra ARM7. Je periferna enota mikrokrmilnika LPC2138, ki ima vgrajene štiri zunanje prekinitvene vhode EINT0 ... EINT3. O tem, ali se bo zunanja prekinitev tudi v resnici zgodila, ali ne, odloča vektorski nadzornik prekinitev (glej dodatek B.4). Zato zunanje prekinitve mikrokrmilnika LPC2138 niso brezpogojne. Podrobnejši opis zunanjih prekinitev lahko bralec najde v [5].

bit3	**bit2**	**bit2**	**bit0**
EINT3	EINT2	EINT1	EINT0

Tabela B.4: Pomen bitov v registrih zunanjih prekinitev

Registri, s katerimi nastavljamo delovanje zunanjih prekinitev, so opisani v nadaljevanju. Vedno so pomembni le spodnji štirje biti registra, ki označujejo posamezne zunanje prekinitve (tabela B.4).

EXTMODE EXTernal interrupt MODE register (naslov: `0xe01fc148`)
Postavitev bita določa postavljanje zahteve po izbrani zunanji prekinitvi ob fronti vhodnega signala. V nasprotnem primeru se zahteva po zunanji prekinitvi postavi glede na stanje vhodnega signala.

EXTPOLAR EXTernal interrupt POLARity register (naslov: `0xe01fc14c`)
Postavitev bita določa postavljanje zahteve po izbrani zunanji prekinitvi ob naraščajoči fronti, oziroma visokem stanju vhodnega signala, odvisno od na-

stavitve v registru *EXTMODE*. V nasprotnem primeru se zahteva po zunanji prekinitvi postavi ob padajoči fronti, oziroma nizkem stanju vhodnega signala.

EXTINT EXTernal INTerrupt flag register (naslov: `0xe01fc140`)
Ko vhodni signal zunanje prekinitve izpolni pogoje določene v registrih *EXTMODE* in *EXTPOLAR* se v tem registru postavi pripadajoč bit. To pomeni, da je postavljena zahteva po izbrani zunanji prekinitvi. O tem, ali se bo zunanja prekinitev izvršila, ali ne, odloča vektorski nadzornik prekinitev. Zahtevo po prekinitvi umaknemo tako, da na ustrezen bit zapišemo enko, s čimer ga postavimo na nič. Pisanje ničel v register nima učinka. Zahteva po zunanji prekinitvi se ob izvršitvi ne umakne avtomatsko, ampak mora za to poskrbeti programska oprema.

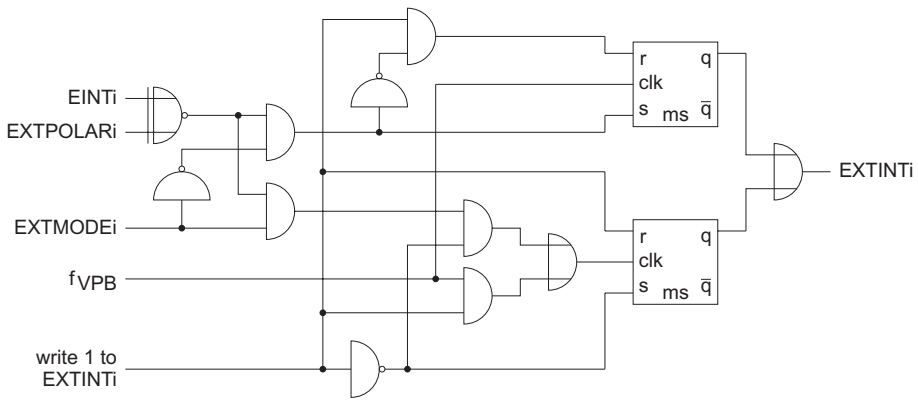
EXTWAKE EXTernal interrupt WAKEup register (naslov: `0xe01fc144`)
Postavitev bita omogoči dvig procesnega jedra iz ustavljenega stanja, ko se pojavi zahteva po pripadajoči zunanji prekinitvi. V ustavljenem stanju (angl. power-down mode) mikrokrmilnik varčuje pri porabi električne energije, tako da ustavi urine signale, oziroma oscilator.

P0.1	EINT0	*PINSEL0* = (*PINSEL0* & `0xfffffff3`) | `0x0000000c`
P0.3	EINT1	*PINSEL0* = (*PINSEL0* & `0xffffff3f`) | `0x000000c0`
P0.7	EINT2	*PINSEL0* = (*PINSEL0* & `0xffff3fff`) | `0x0000c000`
P0.9	EINT3	*PINSEL0* = (*PINSEL0* & `0xfff3ffff`) | `0x000c0000`
P0.14	EINT1	*PINSEL0* = (*PINSEL0* & `0xcfffffff`) | `0x20000000`
P0.15	EINT2	*PINSEL0* = (*PINSEL0* & `0x3fffffff`) | `0x80000000`
P0.16	EINT0	*PINSEL1* = (*PINSEL1* & `0xfffffffc`) | `0x00000001`
P0.20	EINT3	*PINSEL1* = (*PINSEL1* & `0xffffffcf`) | `0x00000300`
P0.30	EINT3	*PINSEL1* = (*PINSEL1* & `0xcfffffff`) | `0x20000000`

Tabela B.5: Pini zunanjih prekinitev

Zunanje prekinitve lahko prožijo vhodni signali le na nekaterih pinih vzporednih vrat P0. Tabela B.5 podaja pine, ki jih je mogoče definirati kot vhode zunanjih

prekinitev, ter maske registrov *PINSEL0* in *PINSEL1*, s katerimi za izbran pin dosežemo želeno nastavitev.



Slika B.5: Logika zunanje prekinitve

Princip delovanja posamezne zunanje prekinitve podaja shema na sliki B.5. Zahtevo po zunanji prekinitvi ob izbranem stanju ($EXTMODEi = 0$) vhodnega signala *EINTi* poda zgornja pomnilna celica RS. Le ta se postavi v visoko stanje takoj, ko vhodni signal *EINTi* zavzame glede na *EXTPOLARi* predpisano stanje. Preklop se zgodi ob prvem impulzu urinega signala vodila VPB. Umik zahteve medtem ni mogoč.

Zunanjo prekinitev ob izbrani fronti ($EXTMODEi = 1$) vhodnega signala *EINTi* zahteva spodnja pomnilna celica RS. Z *EXTPOLARi* predpisana fronta jo postavi v visoko stanje, kar se zgodi le, ko zahteve ne umikamo. Med umikom

zahteve po zunanji prekinitvi spodnjo pomnilno celico RS proži ura perifernega vodila VPB.

Sledi primer nastavitve zunanje prekinitve EINT1 na pinu P0.3. Prekinitev naj bo zahtevana ob padajoči fronti vhodnega signala, ter naj ne dvigne procesnega jedra iz ustavljenega stanja.

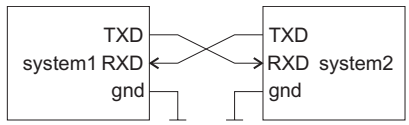
```
PINSEL0 = (PINSEL0 & 0xffffff3f) | 0x000000c0;
INTWAKE = 0x00000000;
EXTMODE = 0x00000002;
EXTPOLAR = 0x00000000;
EXTINT = 0x00000002;
```

Za posamezno zunanjo prekinitev lahko definiramo tudi več pinov hkrati, oziroma več vhodnih signalov. Če je zahteva po zunanji prekinitvi določena s stanjem, potem se poda v trenutku, ko katerikoli izmed vhodnih signalov izbrano stanje doseže. Če je zahteva po zunanji prekinitvi določena s fronto, se upošteva le vhodni signal na najnižjem pinu. Fronte vhodnih signalov na ostalih pinih se ignorirajo.

B.9 Splošni asinhroni sprejemnik in oddajnik

Med posameznimi sistemi podatke prenašamo s pomočjo različnih protokolov, ki določajo pravila prenosa. Eden izmed pogosto uporabljenih načinov je zaporedni prenos podatkov preko splošnega asinhronega sprejemnika in oddajnika (angl. Universal Asynchronous Receiver/Transmitter - UART). Smer prenosa podatka je v naprej določena in je ni mogoče spremeniti. Podatek potuje od pošiljatelja ali oddajnika (angl. Transmit Data - TXD) k prejemniku ali sprejemniku (angl. Receive Data - RXD). Ker je prenos zaporeden, potrebujemo med oddajnikom in sprejemnikom le eno povezavo. Če želimo podatke prenašati tudi v obratni smeri, je potrebno dodati še eno povezavo, kot je to prikazano na sliki B.6. Vsak izmed sistemov na sliki lahko podatke hkrati pošilja in sprejema. Imamo dva tokova podatkov, enega od prvega sistema k drugemu in drugega v obratni smeri.

Opraviti imamo s popolno dvosmerno komunikacijo med sistemoma (angl. Full DupleX - FDX), ki je sestavljena iz dveh enosmernih povezav (angl. simplex).



Slika B.6: Dvosmerni asinhron prenos podatkov med dvema sistemoma

Prenos podatkov preko splošnega asinhronega sprejemnika in oddajnika je, kot pove že ime, asinhron. To pomeni, da se sinhronizacijski signal (ura) ne prenaša. Urina signala sprejemnika in oddajnika tečeta neodvisno eden od drugega in tako nista sinhronizirana. Imata pa v naprej dogovorjeno enako frekvenco. Za časovno uskladitev sprejemnika in oddajnika so pred in po prenosu podatka dodani sinhronizacijski biti, kot je to prikazano na sliki B.7.

Slika B.7: Primer prenosa osem bitnega podatka s sodo parno kontrolo

Ko je povezava v praznem teku (podatkov ne prenašamo) je v visokem (angl. mark) stanju. Začetek prenosa oddajnik označi z nizkim (angl. space) stanjem, ki predstavlja začetni sinhronizacijski bit, ali bit start. Sprejemnik bit start zazna in sinhronizira svoj urin signal z oddajnikovim. Sledi podatek poljubne dolžine, navadno od 5 do 8 bitov. Na sliki B.7 je dodan še neobvezen bit parne kontrole, ki olajša odkrivanje napak pri prenosu. Poznamo sodo in liho parno kontrolo. Pri prvi je bit parne kontrole postavljen tako, da je število vseh visokih stanj v podatku in bitu parne kontrole sodo, pri drugi pa liho. Prenos zaključi končni sinhronizacijsi bit, ali bit stop (eden, eden in pol ali dva bita), ki povezavo zopet postavi v prazen tek.

Med prenosom dveh podatkov je povezava v praznem teku poljubno dolg čas. Zato je uporaba splošnega asinhronega sprejemnika in oddajnika posebej

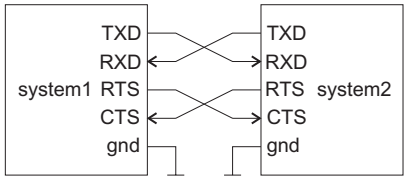
primerna, kadar je tok podatkov neenakomeren. Hitrost prenosa, število bitov podatka in parna kontrola morajo biti v naprej dogovorjeni. Frekvenci urinih signalov sprejemnika in oddajnika naj bi bili enaki, vendar prenos še vedno deluje, če se frekvenci ne razlikujeta preveč. Med prenosom enega podatka ne sme priti do zamika večjega od trajanja enega bita, kar zagotavlja visoko robustnost povezave.

Hitrost prenosa podajamo na dva načina, in sicer s številom prenesenih bitov na sekundo (angl. Bits Per Second - bps), ali pa z baudi (angl. baud rate). Čeprav se obe enoti mnogokrat uporabljata kot sinonim ena za drugo, pa ne pomenita iste stvari. Pomen števila prenesenih bitov na sekundo je nedvoumen. Hitrost izražena z baudi pa pomeni število možnih sprememb stanja signala na sekundo. V primeru splošnega asinhronega sprejemnika in oddajnika ima signal le dve možni stanji, visoko (angl. mark) in nizko (angl. space). Vsako stanje podaja en bit informacije, zato je v tem posebnem primeru hitrost izražena v baudih enaka številu prenesenih bitov na sekundo. Če lahko signal zavzame več različnih stanj, potem je za prenos enake količine bitov potrebnih manj sprememb stanja signala. Hitrost izražena v baudih je nižja od števila prenesenih bitov na sekundo. Primer: naj ima signal osem različnih diskretnih nivojev ali stanj. Vsako stanje zato podaja tri bite informacije. Če se stanje signala spremeni 100-krat na sekundo, potem v vsaki sekundi prenesemo 300 bitov informacije. Hitrost prenosa je 100 baudov, ali 300 bitov na sekundo.

Sprejemnik mora biti ves čas pripravljen na sprejem poljubne količine podatkov, ki jih lahko oddajnik pošlje kadarkoli. Neprestano pripravljenost je včasih težko zagotoviti, zato sprejemnik oddajniku pove, ali je na sprejem pripravljen ali ne. Kadar sprejemnik ni pripravljen, oddajnik s pošiljanjem podatkov počaka. Takšno sporazumevanje med oddajnikom in sprejemnikom imenujemo rokovanje (angl. handshake). Poznamo dve vrsti rokovanja, programsko in strojno. Programsko rokovanje (angl. software flow control) ne potrebuje nobenih dodatnih povezav med sistemoma. Izvaja se po obeh glavnih podatkovnih povezavah, po katerih sistema drug drugemu med podatki pošiljata še posebna kontrolna znaka DC1 (angl. Device Character 1 (Xon ali Ctrl-Q) s kodo `0x11` v tabeli ASCII) in DC3 (angl. Device Character 3 (Xoff ali Ctrl-S) s kodo `0x13` v tabeli ASCII). Znak DC1 pomeni, da je sprejemnik pripravljen, zato lahko oddajnik drugega sistema prične z oddajanjem. Znak DC3 ima nasproten pomen,

sprejemnik na sprejem trenutno ni pripravljen, zato oddajnik nasprotnega sistema z oddajo počaka.

Dobra lastnost programskega rokovanja je, da ne potrebujemo dodatne strojne opreme. Vendar je na ta način težje prenašati binarne podatke. Koda DC1 je lahko del binarnega toka podatkov, ali pa predstavljata kontrolni znak. Enako velja za DC3. Zato moramo programsko zagotoviti poseben mehanizem za razlikovanje med kontrolnim znakom (DC1 ali DC3) in binarnim podatkom z enako kodo.



Slika B.8: Asinhron prenos podatkov med dvema sistemoma s strojnim rokovanjem

Za izvedbo strojnega rokovanja (angl. hardware flow control) potrebujemo dodatno strojno opremo. Enako kot pri programskem rokovanju gre za sporazumevanje med oddajnikom in sprejemnikom, ki sedaj poteka preko dveh dodatnih povezav, kot prikazuje slika B.8. Povezava *RTS* (angl. Request To Send) nasprotnemu sistemu pove, da lahko prične z oddajo. Sprejemnik je pripravljen na sprejem. In obratno povezava *CTS* (angl. Clear To Send) pove, da je sprejemnik nasprotnega sistema pripravljen na sprejem. Lahko pričnemo z oddajo.

Do sedaj smo obravnavali asinhrono zaporedno komunikacijo med dvema enakovrednima napravama (osebni računalnik, tiskalnik, mikrokrmilniški sistem ...) in jih označujemo s kratico DTE (angl. Data Terminating Equipment). Poznamo še naprave DCE (angl. Data Communication Equipment), ki so prirejene

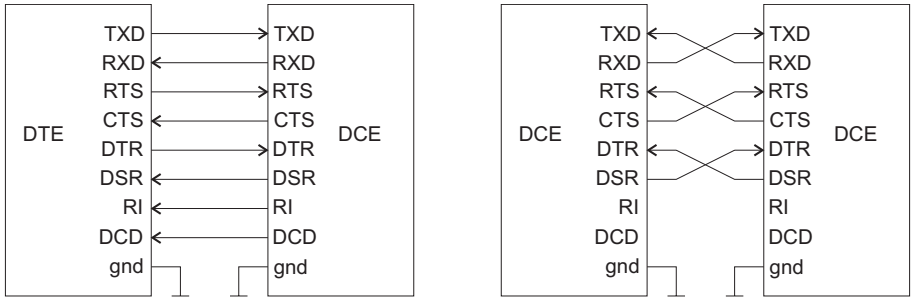
za neposreden priklop na napravo DTE, kot na primer modem. Za povezavo med napravo DTE in napravo DCE so dodane še štiri povezave, in sicer:

- *DTR* (angl. Data Terminal Ready) DTE → DCE: Sem vklopljen in pripravljen na delovanje,

- *DSR* (angl. Data Set Ready) DCE → DTE: Sem vklopljen in pripravljen na delovanje,

- *RI* (angl. Ring Indicator) DCE → DTE: Nekdo kliče in

- *DCD* (angl. Data Carrier Detect) DCE → DTE: Modem na drugi strani je pripravljen.



Slika B.9: Priklop naprave DCE na napravo DTE in naprave DCE na napravo DCE

Priklop naprave DCE prikazuje slika B.9. Pri tem naj opozorimo, da imajo naprave DCE zaradi neposrednega priklopa na naprave DTE drugače označene

povezave. Tako na primer *TXD* na napravi DTE pomeni izhodno sponko, na napravi DCE pa vhodno sponko.

Splošni asinhroni sprejemnik in oddajnik imata vgrajena notranja medpomnilnika FIFO, ki prilagajata hitrost programske opreme hitrosti prenosa. Ko hoče programska oprema podatek oddati, ga zapiše v medpomnilnik FIFO oddajnika. Dejanska oddaja podatka se ne izvrši nujno takoj. Podobno se sprejeti podatek shrani v medpomnilnik FIFO sprejemnika. Programska oprema ga lahko prebere kasneje. Medpomnilnika FIFO oddajnika in sprejemnika sta vedno majhna. V Philipsovem mikrokrmilniku LPC2138 sta za primer velika po 16 bajtov. Čemu tako majhne velikosti medpomnilnikov, saj izdelava nekaj 100 dodatnih bajtov pomnilnika RAM ne predstavlja znatnega stroška? Razlog je sporazumevanje med oddajnikom in sprejemnikom s pomočjo rokovanja. Tok podatkov od oddajnika k sprejemniku je prekinjen, ko je sprejeta programska ali strojna zahteva po zaustavitvi. V obeh primerih za prekinitev poskrbi programska oprema, in sicer tako, da oddajniku začasno ne dodeljuje novih podatkov, dokler ne prispe zahteva po ponovni vzpostavitvi povezave. Tako oddajnik kljub prekinitvi odda še vse znake, ki so že v njegovem medpomnilniku FIFO. Oziroma, ko je oddan znak DC3 (zahteva po prekinitvi prenosa), bo le ta upoštevan šele, ko ga programska oprema prebere iz medpomnilnika FIFO sprejemnika. Pred tem seveda prebere vse že prej prispele znake. Splošni asinhroni oddajnik in sprejemnik na kontrolne znake, ali signale ne reagirata samodejno, zaradi česar veliki medpomnilniki niso smiselni.

Philipsov mikrokrmilnik LPC2138 ima med perifernimi enotami vgrajena tudi dva splošna asinhrona sprejemnika in oddajnika uart0 in uart1. Prvi splošni asinhroni sprejemnik in oddajnik uart0 ima le povezavi *TXD* in *RXD*. Tako nima vgrajene podpore za strojno rokovanje in priključitev naprave DCE, oziroma modema. Drugi splošni asinhroni sprejemnik in oddajnik uart1 je polno funkcionalen. Na voljo so vse dodatne povezave potrebne pri strojnem rokovanju, kot tudi povezave do naprave DCE. Podrobnejši opis obeh enot uart0 in uart1 lahko bralec najde v [5].

Registri, s katerimi nastavljamo delovanje splošnega asinhronega sprejemnika in oddajnika uart0, so opisani v nadaljevanju. Enako velja za nastavljanje delovanja drugega splošnega asinhronega sprejemnika in oddajnika uart1, le da se

registri nahajajo na naslovih registrov enote uart0, ki jim prištejemo konstanto `0x00004000`.

bit5	bit4	parna kontrola	bit1	bit0	dolžina podatka
0	0	liha	0	0	5 bitov
0	1	soda	0	1	6 bitov
1	0	vedno ena	1	0	7 bitov
1	1	vedno nič	1	1	8 bitov

Tabela B.6: Nastavitev parne kontrole in dolžine podatka

U0LCR Line Control Register (naslov: `0xe000c00c`)
Podaja format zaporednega prenosa podatkov preko enote uart0. Posamezni biti v registru imajo naslednji pomen:

bit1 in **bit0** določata dolžino podatka, kot je podano v tabeli B.6

bit2 s postavitvijo izberemo dva bita stop (oziroma pri 5 bitnem podatku en bit stop in pol), namesto enega

bit3 omogoči bit parne kontrole

bit5 in **bit4** določata način parne kontrole, kot je podano v tabeli B.6

bit6 postavitev prekine oddajo

bit7 ali bit *DLAB*; postavitev omogoči delitvena registra *U0DLL* in *U0DLM*, ter s tem generator takta, ki določa hitrost prenosa

$$uart_{baudrate} = \frac{f_{vpb}}{16 \times (16 \times DLM + DLL)} \tag{B.3}$$

U0DLL Divisor Latch LSB register (naslov: `0xe000c000`)
Register hrani nižjih osem bitov konstante, ki po enačbi (B.3) določa hitrost

splošnega asinhronega sprejemnika in oddajnika uart0. Višjih osem bitov se nahaja v registru *U0DLM*. V primeru, da je šestnajst bitna konstanta zapisana v obeh registrih enaka `0x0000`, velja $uart_{baudrate} = f_{vpb}/16$. V register je možno pisati le v primeru, ko je bit *DLAB* registra *U0LCR* enak ena.

U0DLM Divisor Latch MSB register (naslov: `0xe000c004`)
Register hrani višjih osem bitov konstante, ki po enačbi (B.3) določa hitrost splošnega asinhronega sprejemnika in oddajnika uart0. Nižjih osem bitov se nahaja v registru *U0DLL*. V primeru, da je šestnajst bitna konstanta zapisana v obeh registrih enaka `0x0000`, velja $uart_{baudrate} = f_{vpb}/16$. V register je možno pisati le v primeru, ko je bit *DLAB* registra *U0LCR* enak ena.

bit7	bit6	število bajtov
0	0	1 bajt
0	1	4 bajti
1	0	8 bajtov
1	1	14 bajtov

Tabela B.7: Število bajtov v sprejemnem medpomnilniku FIFO, ki sprožijo zahtevo po prekinitvi

U0FCR FIFO Control Register (naslov: `0xe000c008`)
Določa stanje obeh medpomnilnikov FIFO splošnega asinhronega sprejemnika in oddajnika uart0. Posamezni biti v registru imajo naslednji pomen:

bit0 postavitev bita omogoči delovanje obeh medpomnilnikov FIFO splošnega asinhronega sprejemnika in oddajnika

bit1 postavitev bita resetira sprejemni medpomnilnik FIFO

bit2 postavitev bita resetira oddajni medpomnilnik FIFO

bit7 in **bit6** podajata število sprejetih bajtov (tabela B.7), ki se morajo nahajati v sprejemnem medpomnilniku FIFO, da splošni asinhroni sprejemnik in oddajnik uart0 poda zahtevo po prekinitvi

U0LSR Line Status Register (naslov: `0xe000c014`)
Posamezni biti v registru podajajo trenutno stanje splošnega asinhronega sprejemnika in oddajnika uart0 in imajo naslednji pomen:

bit0 pove, ali v sprejemnem registru *U0RBR* čaka prispeli podatek

bit1 se postavi v primeru, ko prispe nov podatek, vendar je sprejemni medpomnilnik FIFO poln, zaradi česar je podatek izgubljen

bit2 označuje napako pri preverjanju paritete podatka v registru *U0RBR*

bit3 označuje napako okvirja, oziroma izostanek pričakovanega bita stop

bit4 označuje prekinitev prenosa (angl. break interrupt), ki se pojavi v primeru, da je sprejemna povezava *RXD* celoten čas prenosa enega znaka (vključno z bitom stop) v nizkem (angl.space) stanju

bit5 pove, ali je oddajni register *U0THR* prazen

bit6 pove, da je oddajni register *U0THR* prazen, oddaja zadnjega podatka v tem registru pa je že končana

bit7 označuje, ali je bila za prispeli podatek v sprejemnem registru *U0RBR* zaznana katerakoli izmed napak, ki jih podajajo biti bit2 (pariteta), bit3 (okvir) in bit4 (prekinitev)

U0RBR Receiver Buffer Register (naslov: `0xe000c000`)
Register hrani najstarejši prispeli podatek, oziroma predstavlja najvišje mesto v sprejemnikovem medpomnilniku FIFO. Status podatka podajajo zastavice v registru *U0LSR*. Ker branje *U0RBR* prispeli podatek odstrani iz medpomnilnika, se ob tem spremenijo tudi zastavice v *U0LSR*. Da dobimo veljaven par, je potrebno *U0LSR* prebrati pred *U0RBR*. Register *U0RBR* je možno prebrati le v primeru, ko je bit *DLAB* registra *U0LCR* enak nič. Pisanje vanj ni mogoče.

U0THR Transmit Holding Register (naslov: `0xe000c000`)
V register zapišemo podatek, ki ga želimo oddati. Podatek se do oddaje hrani v medpomnilniku FIFO oddajnika. Na vrsto za oddajo pride, ko so oddani vsi podatki pred njim. S pisanjem v register pravzaprav polnimo oddajni medpomnilnik FIFO. Status registra, oziroma oddajnega medpomnilnika FIFO podajajo

zastavice v registru *U0LSR*. V register *U0THR* je možno pisati le v primeru, ko je bit *DLAB* registra *U0LCR* enak nič. Branje registra ni mogoče.

U0IIR Interrupt ID Register (naslov: `0xe000c008`)
Register hrani status trenutne prekinitve, ki jo zahteva splošni asinhroni sprejemnik in oddajnik uart0. Prebrati ga moramo pred zaključkom prekinitvenega podprograma, da se zahteva po prekinitvi uart0 pravilno umakne. Biti v registru povedo, za kakšne vrste prekinitev gre, in sicer:

bit0 če je bit0 enak nič, potem na izvršitev čaka vsaj ena zahteva po prekinitvi uart0

bit3, **bit2** in **bit1** povedo za katero prekinitev gre, in sicer:

- vrednost 011 označuje prekinitev *RLS* (angl. RX Line Status), ki se zgodi, kadar pride pri prenosu do napake. Vrsto napake lahko razberemo iz bitov bit1 do bit4 registra *U0LSR*. Zahteva po prekinitvi *RLS* ima med prekinitvami uart0 najvišjo prioriteto in se odstrani z branjem registra *U0LSR*.
- vrednost 010 označuje prekinitev *RDA* (angl. RX Data Available), ki se zgodi, kadar se v sprejemnem medpomnilniku FIFO nabere toliko znakov, kot je določeno z bitoma bit7 in bit6 registra *U0FCR*. Zahtevo po prekinitvi *RDA* odstranimo z branjem registra *U0RBR*.
- vrednost 110 označuje prekinitev *CTI* (angl. Character Time-out Indication), ki se zgodi, kadar se v sprejemnem medpomnilniku FIFO nahaja vsaj en neprebran znak, in že dalj časa (približno za štiri znake) ni prispel noben nov znak. Zahtevo po prekinitvi *CTI* odstranimo z branjem registra *U0RBR*.
- vrednost 001 označuje prekinitev *THRE* (angl. Transmitter Holding Register Empty), ki se zgodi, kadar je oddajni medpomnilnik FIFO prazen, kar podaja tudi bit5 registra *U0LSR*. Zahteva po prekinitvi *THRE* ima med prekinitvami uart0 najnižjo prioriteto in se odstrani s pisanjem v register *U0THR*.
- (- na splošnem asinhronem sprejemniku in oddajniku uart1 ima pomen tudi vrednost 000, ki označuje modemsko prekinitev. Zgodi se ob spremembi stanja na kateremkoli izmed modemskih vhodov *DCD*, *DSR*, ali *CTS*. Sproži jo tudi prehod iz nizkega v visoko stanje na

vhodu *RI*. Zahteva po modemski prekinitvi ima še nižjo prioriteto od zahteve *THRE* in se odstrani z branjem registra *M1MSR*.)

Pisanje v register *U0IIR* ni mogoče.

U0IER Interrupt Enable Register (naslov: `0xe000c004`)
Register določa, kateri izmed dogodkov *RLS*, *RDA*, *CTI* in *THRE* dejansko podajo zahtevo po prekinitvi zaradi splošnega asinhronega sprejemnika in oddajnika uart0. Biti v registru imajo naslednji pomen:

bit0 postavitev bita omogoči prekinitvi zaradi dogodkov *RDA* in *CTI*

bit1 postavitev bita omogoči prekinitev zaradi dogodka *THRE*

bit2 postavitev bita omogoči prekinitev zaradi dogodka *RLS*

(**bit3** postavitev bita omogoči modemsko prekinitev na splošnem asinhronem sprejemniku in oddajniku uart1)

Do registra *U0IER* lahko dostopamo le, ko je bit *DLAB* registra *U0LCR* enak nič.

U0TER Transmit Enable Register (naslov: `0xe000c030`)
V registru ima pomen le bit7, ali *TXEN*, ki omogoča izvedbo programskega (pri splošnem asinhronem sprejemniku in oddajniku uart1 pa tudi strojnega) rokovanja. Ko je bit7 postavljen, se znaki shranjeni v oddajnem medpomnilniku FIFO normalno oddajajo. Umaknitev bita ($TXEN = 0$) oddajo prekine. Znak, ki se trenutno oddaja se odda do konca.

Splošni asinhroni sprejemnik in oddajnik uart1 ima dodano še podporo za strojno rokovanje, ki je splošni asinhroni sprejemnik in oddajnik uart0 nima. Za delo z dodatnimi povezavami sta dodana še dva registra, in sicer:

U1MCR Modem Control Register (naslov: `0xe0010010`)
Najnižja bita v registru določata vrednosti izhodnih povezav strojnega rokovanja, torej izhodov *DTR* in *RTS*. Poleg tega lahko splošni asinhroni sprejemnik in oddajnik uart1 postavimo v tako imenovani "loopback" testni način delova-

nja, ko sta oddajnik in sprejemnik povezana med seboj. Biti v registru *U1MCR* imajo naslednji pomen:

bit0 stanje izhoda *DTR*

bit1 stanje izhoda *RTS*

bit4 postavitev bita omogoči "loopback" testni način delovanja

U1MSR Modem Status Register (naslov: `0xe0010018`)
Register na posameznih bitih podaja stanje vhodnih povezav strojnega rokovanja, torej vhodov *CTS*, *DSR*, *RI* in *DCD*. Biti v registru *U1MSR* imajo naslednji pomen:

bit0 postavljen označuje, da je na vhodu *CTS* prišlo do spremembe stanja; ob branju registra *U1MSR* se bit avtomatsko postavi na nič

bit1 postavljen označuje, da je na vhodu *DSR* prišlo do spremembe stanja; ob branju registra *U1MSR* se bit avtomatsko postavi na nič

bit2 postavljen označuje, da je na vhodu *RI* prišlo do spremembe stanja; ob branju registra *U1MSR* se bit avtomatsko postavi na nič

bit3 postavljen označuje, da je na vhodu *DCD* prišlo do spremembe stanja; ob branju registra *U1MSR* se bit avtomatsko postavi na nič

bit4 podaja negirano stanje vhoda *CTS*

bit5 podaja negirano stanje vhoda *DSR*

bit6 podaja negirano stanje vhoda *RI*

bit7 podaja negirano stanje vhoda *DCD*

Pisanje v register *U1MSR* ni mogoče.

Splošna asinhrona sprejemnika in oddajnika uart0 in uart1 vgrajena v Philipsov mikrokrmilnik LPC2138 sta povezana na pine vzporednih vrat P0. Tabela B.8 podaja nabor pinov, kjer se vmesnika nahajata. Da pini delujejo s splošnim asinhronim sprejemnikom in oddajnikom uart0, oziroma uart1, jih je potrebno

temu primerno nastaviti. Maske registrov *PINSEL0* in *PINSEL1*, s katerimi za izbran pin dosežemo želeno nastavitev, so podane v zadnjem stolpcu tabele B.8.

P0.0	TXD_{uart0}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xfffffffc}) \mid \texttt{0x00000001}$
P0.1	RXD_{uart0}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xfffffff3}) \mid \texttt{0x00000004}$
P0.8	TXD_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xfffcffff}) \mid \texttt{0x00010000}$
P0.9	RXD_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xfff3ffff}) \mid \texttt{0x00040000}$
P0.10	RTS_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xffcfffff}) \mid \texttt{0x00100000}$
P0.11	CTS_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xff3fffff}) \mid \texttt{0x00400000}$
P0.12	DSR_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xfcffffff}) \mid \texttt{0x01000000}$
P0.13	DTR_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xf3ffffff}) \mid \texttt{0x04000000}$
P0.14	DCD_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0xcfffffff}) \mid \texttt{0x10000000}$
P0.15	RI_{uart1}	$PINSEL0 = (PINSEL0 \;\&\; \texttt{0x3fffffff}) \mid \texttt{0x40000000}$

Tabela B.8: Pini splošnih asinhronih sprejemnikov in oddajnikov uart0 in uart1

Sledi primer inicializacije splošnega asinhronega sprejemnika in oddajnika uart0. Najprej povežemo pina P0.0 (TXD_{uart0}) in P0.1 (RXD_{uart0}), ter začasno onemogočimo oddajo. Po postavitvi hitrosti prenosa na 300 bitov na sekundo (vrednosti registrov *U0DLM* in *U0DLL* sta izračunani ob predpostavki, da je frekvenca urinega signala vodila VPB enaka f_{vpb} = 3MHz) podamo format prenosa. In sicer bo podatek dolg osem bitov s sodo parno kontrolo in enim stop

bitom. Nato izpraznimo in omogočimo oba medpomnilnika FIFO. Prekinitev naj sproži vsak prispeli podatek.

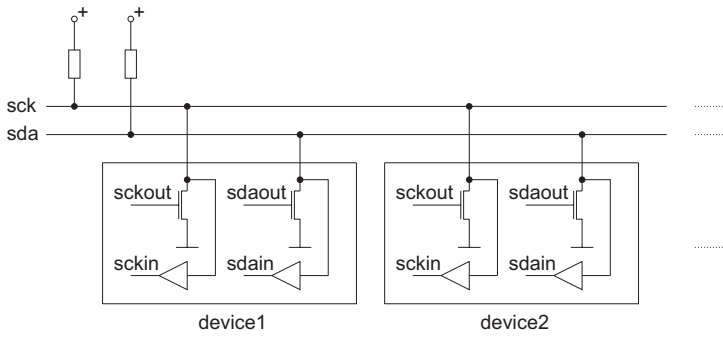
```
PINSEL0 = (PINSEL0 & 0xfffffffc) | 0x00000001;
PINSEL0 = (PINSEL0 & 0xfffffff3) | 0x00000004;
U0TER = 0x00;
U0LCR = 0x80;
U0DLL = 0x71;     // 16 * U0DLM + U0DLL = fvpb / (16 * bps)
U0DLM = 0x02;
U0LCR = 0x1b;
U0IER = 0x00;
U0FCR = 0x07;
U0IER = 0x01;
U0TER = 0x80;
```

S tem je splošni asinhroni sprejemnik in oddajnik uart0 pripravljen na delovanje brez ali s programskim rokovanjem. Za strojno rokovanje vmesnik uart0 nima vgrajene podpore. Vendar tudi programsko rokovanje ne bo steklo samo po sebi. Realizirati ga je potrebno s programsko opremo. Oddaja bo potekala avtomatsko, in sicer bo vsak podatek zapisan v register *U0THR* oddan takoj, ko bo mogoče. Vsak sprejeti podatek bo podal zahtevo po prekinitvi vrste uart0. Da do prekinitve res pride, poskrbi pravilno nastavljen vektorski nadzornik prekinitev (glej dodatek B.4).

B.10 Vodilo I^2C

Eden izmed dokaj razširjenih protokolov za komunikacijo med integriranimi vezji je protokol I^2C (angl. Inter Integrated Circuit), ki ga je razvilo podjetje Philips. S pomočjo vodila v mikrokrmilniški sistem navadno dodajamo druge standardne gradnike, kot na primer zaporedni RAM, prikazovalnike, generatorje zvoka itd. Vodilo I^2C je sestavljeno iz podatkovne povezave *sda* in povezave z urinimi impulzi *sck*. Vse enote morajo biti na obe povezavi priključene preko izhodov z odprtim kolektorjem, oziroma odprtim ponorom, kakor prikazuje slika B.10.

Zaradi uporov na napajalno napetost (angl. pull up), se obe povezavi v primeru, ko so izhodni tranzistorji vseh enot zaprti, nahajata na visokem nivoju.



Slika B.10: Konfiguracija vodila I^2C

Na vodilo I^2C je lahko priključeno večje število enot. Vsaki izmed njih protokol omogoča tako pošiljanje kot prejemanje podatkov. Ko je vodilo v praznem teku (obe povezavi *sda* in *sck* sta na visokem nivoju), lahko katerakoli izmed enot prične s prenosom podatkov. To stori s tako imenovanim stanjem start, ki označuje začetek prenosa. Stanje start podaja prehod povezave *sda* iz visokega na nizek nivo, medtem ko je povezava *sck* na visokem nivoju. Konec prenosa podatkov označuje stanje stop, oziroma prehod povezave *sda* iz nizkega na visok nivo, medtem ko je povezava *sck* na visokem nivoju. Med stanjem start in stop je vodilo I^2C zasedeno. Če hoče enota takoj po koncu prenosa začeti z novim prenosom, bi moralo stanju stop takoj slediti novo stanje start. V tem primeru je stanje stop izpuščeno. Enota nadaljuje brez stanja stop z

novim stanjem start, ki ga imenujemo ponovni start. Ponovni start je povsem enakovreden stanju start.

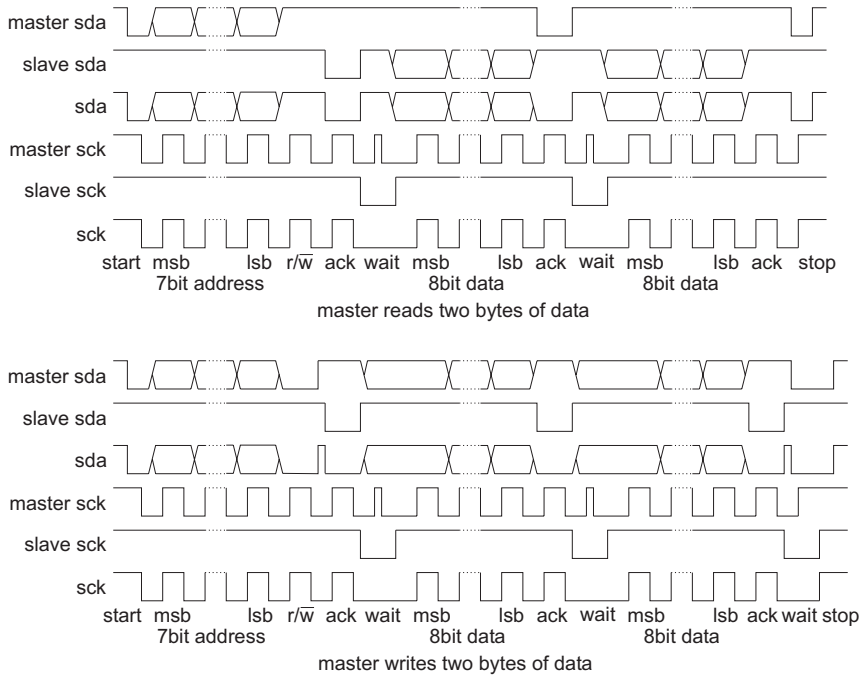
Enota, ki začne s prenosom podatkov, je do konca prenosa nadrejena (angl. master). Nadrejena enota odda stanji start in stop, ter nadzoruje povezavo z urinimi impulzi *sck*. Med prenosom je podatek na povezavi *sda* veljaven takrat, ko je povezava *sck* na visokem nivoju. Nivo na *sda* se lahko spreminja le, ko je *sck* nizek. Prenos se izvršuje po bajtih. Prvi bajt odda nadrejena enota in podaja naslov podrejene (angl. slave) enote in smer prenosa. Z drugimi besedami, nadrejena enota v prvem bajtu pove s katero enoto želi komunicirati in v kateri smeri bo prenos potekal.

Iz tega sledi, da protokol I^2C predvideva štiri načine delovanja enot na vodilu. Enota je lahko nadrejena ali podrejena in podatke pošilja, oziroma oddaja, ali sprejema. Enota tako deluje kot nadrejena in oddaja (angl. master transmitter mode), ali nadrejena in sprejema (angl. master receiver mode), ali podrejena in oddaja (angl. slave transmitter mode), ali podrejena in sprejema (angl. slave receiver mode).

Vsakemu prenesenemu bajtu sledi potrditveni bit, ki ga odda enota, ki sprejema. Če nadrejena enota podatek pošlje, podrejena enota pa sprejema ne potrdi, potem mora nadrejena enota prenos prekiniti. To stori s stanjem stop, ali s ponovnim startom. V primeru, ko nadrejena enota podatke sprejema, mora podrejeni enoti sporočiti, kdaj je prenosa konec. To stori tako, da za zadnjim sprejetim bajtom ne odda potrditvenega bita.

Če podrejena enota po sprejemu ali oddaji enega bajta ni takoj pripravljena na sprejem ali oddajo naslednjega bajta, potem lahko prenos zadrži. To stori tako, da po oddaji ali sprejemu potrditvenega bita postavi povezavo *sck* na nizek

nivo. To je znak nadrejeni enoti, ki načeloma določa nivo povezave *sck*, naj z urinimi impulzi počaka.



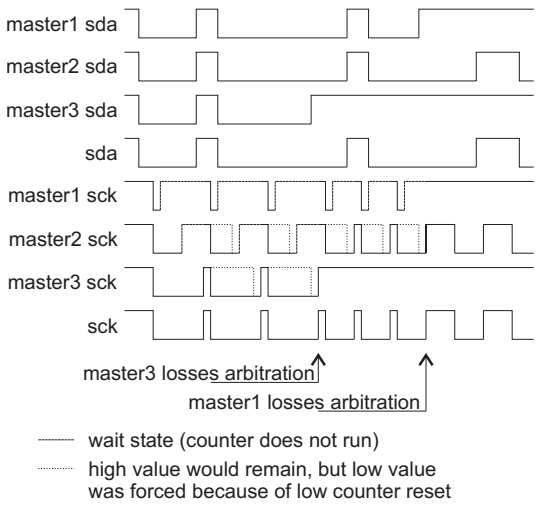
Slika B.11: Potek signalov na vodilu I^2C

Prvi bit v prenosu enega bajta je vedno najpomembnejši (angl. Most Significant Bit, MSB), zadnji pa najmanj pomemben (angl. Least Significant Bit, LSB). V prvem oddanem bajtu, ki ga vedno odda nadrejena enota, podaja prvih sedem bitov naslov podrejene enote, osmi bit pa smer prenosa (bit $R/\overline{W}$). Slika B.11 ponazarja dogajanje na povezavah *sck* in *sda* med bralnim in pisalnim prenosom.

Na vodilo I^2C je lahko priključenih več podrejenih enot in prav tako več nadrejenih enot. Neka enota lahko v enem prenosu zavzame vlogo podrejene enote, v naslednjem pa vlogo nadrejene enote. Vsaka podrejena enota mora imeti unikaten sedem bitni naslov. Kot je bilo že povedano generira urine impulze *sck* vedno nadrejena enota. Vendar, če jih je več, katera? V mirovanju, ko prenos

podatkov ne teče, je na povezavi *sck* visok nivo. Urinih impulzov ni. Prenos se začne, ko katerakoli izmed nadrejenih enot generira stanje start in s tem zasede vodilo. Druge nadrejene enote morajo počakati na konec prenosa (stanje stop), oziroma, da je vodilo zopet v mirovanju. Vodilo tako zasede tista nadrejena enota, ki prva generira stanje start.

Težava lahko nastopi v primeru, ko dve ali več nadrejenih enot generira stanje start istočasno. V tem primeru nastopi arbitražni postopek.



Slika B.12: Arbitražni postopek med nadrejenimi enotami na vodilu I^2C

Vse nadrejene enote dajejo na povezavo *sck* vsaka svoje urine impulze. Števniki, ki štejejo trajanje nizkega nivoja, se v vseh enotah resetirajo v trenutku, ko gre povezava *sck* dejansko iz visokega na nizek nivo. Podobno se števniki, ki štejejo trajanje visokega nivoja ure resetirajo v trenutku, ko gre povezava *sck* dejansko iz nizkega na visok nivo. Ker se upošteva dejansko stanje na povezavi *sck*, se vsi števniki vseh nadrejenih enot vedno resetirajo hkrati. Zaradi načina vezave, ki izvaja logično operacijo *in*, so na povezavi *sck* urini impulzi, katerih nizek nivo traja toliko časa, kolikor traja najdaljša nizka perioda, in katerih visok nivo traja toliko časa, kolikor traja najkrajša visoka perioda. S tem so urini impulzi enolično določeni. Razmere prikazuje slika B.12.

Vsaka izmed nadrejenih enot na začetku prenosa odda naslov podrejene enote in bit $R/\overline{W}$. Stanje na povezavi *sda* je veljavno takrat, kadar je povezava *sck* na

visokem nivoju. Nadrejena enota, ki želi oddati na povezavo *sda* visok nivo, in ji to ne uspe, ker druga nadrejena enota oddaja nizek nivo, se umakne in preneha s prenosom. V primeru, da je nadrejena enota, ki je prenehala s prenosom, lahko tudi podrejena enota, se mora takoj preklopiti v podrejen način delovanja. Lahko se namreč zgodi, da nadrejena enota, ki nadaljuje s prenosom, naslavlja prav to podrejeno enoto, ki je ravnokar prenehala s prenosom kot nadrejena enota.

V posebnem primeru bi se lahko dogodilo, da do arbitraže ne pride do konca prenosa. Torej dve ali več nadrejenih enot opravijo povsem enak prenos. V tem primeru moramo zagotoviti, da vse nadrejene enote prenos končajo hkrati in na enak način. Ne sme se na primer zgoditi, da neka nadrejena enota po prenosu določenega števila bajtov odda stanje stop ali ponovni start, druga nadrejena enota pa bi po prenosu istih bajtov oddajala še naprej. Prav tako se ne sme zgoditi, da bi po koncu prenosa ena nadrejena enota oddala stanje ponovni start, druga pa stanje stop.

Prenos podatkov po vodilu I^2C je zaporeden, zato je potrebna le ena podatkovna povezava *sda*. Prav tako lahko poteka le en prenos naenkrat. Sinhronizacijski signal, oziroma urini impulzi se prenašajo skupaj s podatki po povezavi *sck*. Zaradi tega je prenos podatkov po vodilu I^2C sinhron. Vodilo I^2C je tako predstavnik sinhrone zaporedne komunikacije. Hitrost prenosa določa frekvenca urinih impulzov na povezavi *sck*. Odvisna je od nadrejene enote, vendar ne večja kot 400kHz. Med dvema prenosoma podatkov po vodilu I^2C je vodilo v praznem teku poljubno dolg čas. Obe povezavi *sda* in *sck* sta na visokem nivoju. Vodilo čaka na nov prenos. Zaradi te lastnosti je vodilo I^2C primerno tudi ob neenakomernih prenosih.

Philipsov mikrokrmilnik LPC2138 ima med perifernimi enotami vgrajena dva vmesnika I^2C i2c0 in i2c1. Mikrokrmilnik je tako mogoče priključiti na dve ločeni vodili I^2C. Oba vmesnika sta povsem enakovredna, podrobnejši opis pa je mogoče najti v [5]. V nadaljevanju sledi opis registrov, ki določajo delovanje vmesnika i2c0. Za vmesnik i2c1 so na voljo enaki registri, katerih naslovi so od navedenih premaknjeni za `0x00040000`.

I2C0SCLH Serial CLock High duty cycle register (naslov: `0xe001c010`)
Register določa dolžino trajanja visokega nivoja urinega impulza na povezavi *sck* izraženo s številom urinih ciklov na vodilu VPB (glej dodatek B.2). Visok nivo na *sck* mora trajati vsaj štiri urine cikle na vodilu VPB, zato mora biti

vrednost v tem registru najmanj štiri ali več. Nastavitev v registru *I2C0SCLH* se uporabi v le primeru, ko je vmesnik i2c0 nadrejena enota.

$$f_{\mathrm{I^2C}sck} = \frac{f_{vpb}}{SCLH + SCLL} \tag{B.4}$$

I2C0SCLL Serial CLock Low duty cycle register (naslov: `0xe001c014`)
Register določa dolžino trajanja nizkega nivoja urinega impulza na povezavi *sck* izraženo s številom urinih ciklov na vodilu VPB. Nizek nivo na *sck* mora trajati vsaj štiri urine cikle na vodilu VPB, zato mora biti vrednost v tem registru najmanj štiri ali več. Nastavitev v registru *I2C0SCLL* se uporabi v le primeru, ko je vmesnik i2c0 nadrejena enota. Končna frekvenca urinega signala na povezavi *sck*, ki jo glede na nastavitve registrov *I2C0SCLH* in *I2C0SCLL* izračunamo po enačbi (B.4), mora biti manjša od 400kHz.

I2C0ADR slave ADdRess register (naslov: `0xe001c00c`)
V register zapišemo sedem bitni naslov vmesnika i2c0, ki se nahaja na bitih od bit7 do bit1. Naslov se uporabi, kadar vmesnik deluje kot podrejena enota. Bit bit0 pove, ali naj se vmesnik, poleg izbranega naslova, odziva tudi na naslov `0x00`, ki pomeni splošni poziv vsem podrejenim enotam na vodilu I^2C. Če sedem bitni naslov vmesnika i2c0 ni izbran, oziroma je enak `0x00`, potem se vmesnik na splošni poziv ne odzove. Ostali biti v registru nimajo pomena.

I2C0CONSET CONtrol SET register (naslov: `0xe001c000`)
Register nadzira delovanje vmesnika i2c0 in je komplementaren registru *I2C0CONCLR*. Vpis enke na posamezen bit odstrani postavitev pripadajočega bita v registru *I2C0CONCLR*. Vpis ničle nima učinka. Posamezni biti imajo naslednji pomen:

bit2 ali bit *AA*; omogoči oddajanje potrditvenega bita kadar je vmesnik i2c0 naslovljen, ali kadar je sprejel podatek

bit3 ali bit *SI*; zahteva po prekinitvi; postavi se avtomatsko ob vsaki spremembi stanja na vodilu, oziroma registra *I2C0STAT*; dokler je zahteva po prekinitvi prisotna, vmesnik i2c0 prenos podatkov po vodilu zadrži (povezavo *sck* drži na nizkem nivoju)

bit4 ali bit *STO*; povzroči oddajo stanja stop, če je vmesnik i2c0 nadrejen; po oddanem stanju stop se bit avtomatsko odstrani

bit5 ali bit *STA*; povzroči oddajo stanja start, vmesnik i2c0 pa postane nadrejena enota na vodilu; stanje start je oddano takoj, ko je vodilo prosto

bit6 omogoči delovanje vmesnika i2c0

Ostali biti v registru nimajo pomena. Bite v tem registru postavljamo na nič s pisanjem v register *I2C0CONCLR*.

I2C0CONCLR CONtrol CLeaR register (naslov: `0xe001c018`)
Register je komplementaren registru *I2C0CONSET*. Vpis enke na posamezen bit odstrani postavitev pripadajočega bita v registru *I2C0CONSET*. Vpis ničle nima učinka. Posamezni biti imajo naslednji pomen:

bit2 ali bit *AAC*; onemogoči oddajanje potrditvenega bita

bit3 ali bit *SIC*; odstrani zahtevo po prekinitvi

bit5 ali bit *STAC*; odstrani zahtevo po oddaji stanja start

bit6 onemogoči delovanje vmesnika i2c0

Ostali biti v registru nimajo pomena. Bite v tem registru postavljamo na nič s pisanjem v register *I2C0CONSET*.

I2C0DAT DATa register (naslov: `0xe001c008`)
Register vsebuje osem bitni podatek, ki čaka na oddajo, ali je bil ravnokar sprejet. Podatek za oddajo, ali sprejeti podatek, lahko v register zapišemo, oziroma ga iz njega preberemo le, kadar je podana zahteva po prekinitvi *SI* v registru *I2C0CONSET*. V tem času vmesnik i2c0 vsebine registra *I2C0DAT* ne spreminja. Takoj, ko je zahteva po prekinitvi z bitom *SIC* v registru *I2C0CONCLR* odstranjena, lahko vmesnik i2c0 prične z oddajo, ali sprejemom novega podatka.

stanje	opis stanja	odziv	opis odziva
`0x00`	napaka na vodilu I^2C	*STAC* *STO* *AA* če *I2C0ADR*, drugače *AAC*	zaključitev prenosa
`0x08`, `0x10`	oddano stanje start ali ponovni start	→*I2C0DAT*	oddaja naslova podrejene enote s smerjo prenosa
`0x18`, `0x28`	sprejet potrditveni bit po oddanem naslovu sprejemne enote, oziroma podatku	*STAC* →*I2C0DAT*, ali *STO*	oddaja podatka ali zaključitev prenosa
`0x20`, `0x30`	potrditveni bit po oddanem naslovu sprejemne enote, oziroma podatku ni bil sprejet	*STAC* *STO*	zaključitev prenosa
`0x38`	umik po arbitraži	*STA*	ponoven prenos, ko bo vodilo prosto
`0x40`	sprejet potrditveni bit po oddanem naslovu oddajne enote	*AA*, ali *AAC* *STAC*	določitev oddaje potrditvenega bita

`0x48`	potrditveni bit po oddanem naslovu oddajne enote ni bil sprejet	*AA* če *I2C0ADR*, drugače *AAC* *STAC* *STO*	zaključitev prenosa
`0x50`	oddan potrditveni bit po sprejetem podatku s podrejene enote	*I2C0DAT*→ *AA*, ali *AAC* *STAC*	sprejem podatka, določitev oddaje potrditvenega bita
`0x58`	potrditveni bit po sprejetem podatku s podrejene enote ni bil oddan	*I2C0DAT*→ *AA* če *I2C0ADR*, drugače *AAC* *STAC* *STO*	sprejem podatka, zaključitev prenosa
`0x80`, `0x90`	oddan potrditveni bit po sprejetem podatku z nadrejene enote	*I2C0DAT*→	sprejem podatka
`0xa8`, `0xb0`	oddan potrditveni bit po sprejetem (lastnem) naslovu podrejene oddajne enote	→*I2C0DAT*	oddaja podatka
`0xb8`	sprejet potrditveni bit z nadrejene enote po oddanem podatku	→*I2C0DAT*	oddaja podatka

Tabela B.9: Stanja vmesnika i2c0 in odzivi nanje

I2C0STAT STATus register (naslov: `0xe001c004`)
Register je namenjen le branju, pisanje vanj ni mogoče. Biti bit7 do bit3 podajajo stanje vmesnika i2c0. Ostali biti v registru nimajo pomena. Vsaka sprememba stanja vmesnika i2c0 postavi zahtevo po prekinitvi *SI* v registru *I2C0CONSET*. Podroben opis stanj vmesnika in možnih odzivov nanje se nahaja v [5]. Na tem mestu je v tabeli B.9 podan le okrnjen opis. Simbol $\rightarrow$*I2C0DAT* pomeni, da je podatek v omenjeni register zapisan, simbol *I2C0DAT*$\rightarrow$ pa, da je iz njega prebran.

Po tem, ko je stanje vmesnika i2c0 obdelano (stolpec odzivov), mora programska oprema s postavitvijo bita *SIC* v registru *I2C0CONCLR* umakniti zahtevo po prekinitvi. Za razumevanje odzivov je potrebno upoštevati, da je vrstni red stanj vmesnika deloma določen (primer: vmesnik lahko pride v stanje `0x18` le preko stanj `0x08`, ali `0x10`). Prav tako se posamezna stanja lahko pojavijo le, kadar je vmesnik v določenem načinu delovanja (primer: vmesnik ne more v stanje `0x50`, če bit *AA* v registru *I2C0CONSET* ni postavljen).

P0.2	sck_{i2c0}	*PINSEL0* = (*PINSEL0* & `0xfffffffc`) | `0x00000010`
P0.3	sda_{i2c0}	*PINSEL0* = (*PINSEL0* & `0xffffff3f`) | `0x00000040`
P0.11	sck_{i2c1}	*PINSEL0* = *PINSEL0* | `0x00c00000`
P0.14	sda_{i2c1}	*PINSEL0* = *PINSEL0* | `0x30000000`

Tabela B.10: Pini vmesnikov i2c0 in i2c1

Vmesnika do vodila I^2C i2c0 in i2c1 vgrajena v Philipsov mikrokrmilnik LPC2138 sta povezana na pine vzporednih vrat P0 (glej dodatek B.7). Tabela B.10 podaja nabor pinov, kjer se vmesnika nahajata. Da pini delujejo z vmesnikom i2c0, oziroma i2c1, jih je potrebno temu primerno nastaviti. Ma-

ske registra *PINSEL0*, s katerimi za izbran pin dosežemo želeno nastavitev, so podane v zadnjem stolpcu tabele B.10.

način	globalne spremenljivke	komentar
nadrejena oddajna ali sprejemna enota	*int i2c0_status* *int i2c0_num_of_bytes* *int i2c0_address_rw* *char *i2c0_buf*	ko je *i2c0_status* = 0 postavimo: *i2c0_status* = 1 *i2c0_num_of_bytes* = število bajtov, ki jih želimo oddati ali sprejeti *i2c0_address_rw* = 2 * naslov + r/$\overline{\text{w}}$ *i2c0_buf* = začetek oddajnega niza, oziroma sprejemnega prostora *STA* v registru *I2C0CONSET* ko je prenos končan, je: *i2c0_status* = 0 *i2c0_num_of_bytes* = število neoddanih (nesprejetih) bajtov če pride do napake je: *i2c0_status* = 2
podrejena oddajna enota	*char *i2c0_txslv*	na zahtevo se prične oddajati niz znakov, na katere kaže kazalec *i2c0_txslv*
podrejena sprejemna enota	*char i2c0_rxslv[...]* *int i2c0_rxslv_begin* *int i2c0_rxslv_end*	sprejeti znaki se shranjujejo v ciklični FIFO medpomnilnik *i2c0_rxslv*; indeksa *i2c0_rxslv_begin* in *i2c0_rxslv_end* podajata prvi znak in prvo prosto mesto v medpomnilniku; ko je medpomnilnik poln, se na novo prispeli znaki ignorirajo

Tabela B.11: Komunikacija med gonilnikom vmesnika do vodila I^2C in ostalo programsko opremo

Čeprav imamo na voljo vgrajen vmesnik i2c0, prenos podatkov po vodilu I^2C s programskega stališča ni povsem enostavna naloga. Vmesnik ob spremembi na vodilu zahteva prekinitev in označi stanje, v katerem se trenutno nahaja.

Za pravilen odziv pa mora v prekinitvenem podprogramu poskrbeti programska oprema. Prekinitveni podprogram tako predstavlja gonilnik vmesnika do vodila I^2C. Po drugi strani je gonilnik določen tudi z načinom komunikacije z ostalo programsko opremo. Možen način komunikacije podaja tabela B.11. V tem primeru ostala programska oprema z gonilnikom vmesnika i2c0 komunicira preko globalnih spremenljivk v drugem stolpcu tabele. V nadaljevanju sledi primer izvorne kode gonilnika, ki ima navedene lastnosti. Najprej moramo definirati globalne spremenljivke, ki jih gonilnik uporablja.

```
// Slave receiver mode (cyclic FIFO buffer)
int i2c0_rxslv_begin; // Index of the first received byte
int i2c0_rxslv_end;   // Index of the first empty place
char i2c0_rxslv[...]; // Buffer (define size)
// Slave transmitter mode
char *i2c0_txslv; // String of bytes to be transmitted
// Master modes
int i2c0_address_rw;   // Device address with read/write bit
int i2c0_num_of_bytes; // To be received/transmitted
int i2c0_status;       // Current master status
char *i2c0_buf;        // Receiving/transmitting buffer
```

Sledi inicializacija vmesnika i2c0. Postavimo začetne vrednosti globalnih spremenljivk in povežemo pina P0.2 (sck_{i2c0}) in P0.3 (sda_{i2c0}). Določimo frekvenco urinih impulzov na povezavi *sck* v primeru, ko je vmesnik i2c0 nadrejena enota (vrednosti registrov *I2C0SCLH* in *I2C0SCLL* sta izračunani tako, da je $f_{\mathrm{I^2C}sck}$ simetrična in enaka 300kHz za f_{vpb} = 3MHz). Nato je dan naslov vmesnika i2c0 kot podrejene enote, odziva pa naj se tudi na splošni poziv. Da se kot naslovljena podrejena enota oglasi s potrditvenim bitom, je postavljen bit

AA v registru *I2C0CONSET*. Ko vmesnik i2c0 omogočimo, ob vsaki spremembi stanja zahteva prekinitev.

```
i2c0_rxslv_begin = 0;
i2c0_rxslv_end = 0;
i2c0_txslv = ...; // Define string to be transmitted
i2c0_status = 0;
PINSEL0 = (PINSEL0 & 0xffffff0f) | 0x00000050;
I2C0SCLH = 0x00000005;
I2C0SCLL = 0x00000005;
I2C0ADR = 2 * ... + 1; // Define slave address
I2C0CONCLR = 0x0000006c;
I2C0CONSET = 0x00000044;
```

Da bo vmesnik i2c0 pravilno deloval, moramo imeti pripravljen ustrezen prekinitveni podprogram, oziroma gonilnik. Prekinitev na zahtevo vmesnika i2c0 mora biti omogočena, za kar poskrbi pravilno nastavljen vektorski nadzornik prekinitev (glej dodatek B.4). Sledi izvorna koda prekinitvene funkcije gonilnika. Funkcija ustrezno reagira na trenutno stanje vmesnika i2c0 glede na tabeli B.9 in B.11.

```
void handle_i2c0_state() {
  static int i2c0_txslv_cnt, i2c0_num_of_bytes_internal, i2c0_cnt;
  switch(I2C0STAT) {
  case 0x00000000:
    i2c0_status = 2;
    I2C0CONCLR = 0x00000020;
    I2C0CONSET = 0x00000010;
    if(I2C0ADR) I2C0CONSET = 0x00000004;
    else I2C0CONCLR = 0x00000004;
    break;
  case 0x00000008:
  case 0x00000010:
    i2c0_cnt = 0;
    I2C0DAT = i2c0_address_rw;
    break;
  case 0x00000018:
    i2c0_num_of_bytes_internal = i2c0_num_of_bytes;
  case 0x00000028:
    I2C0CONCLR = 0x00000020;
```

```
    if(i2c0_num_of_bytes_internal) {
      i2c0_num_of_bytes_internal = i2c0_num_of_bytes_internal - 1;
      I2C0DAT = i2c0_buf[i2c0_cnt];
      i2c0_cnt = i2c0_cnt + 1;
    } else {
      i2c0_status = 0;
      i2c0_num_of_bytes = 0;
      I2C0CONSET = 0x00000010;
    }
    break;
  case 0x00000030:
    i2c0_num_of_bytes = i2c0_num_of_bytes_internal + 1;
  case 0x00000020:
    i2c0_status = 0;
    I2C0CONCLR = 0x00000020;
    I2C0CONSET = 0x00000010;
    break;
  case 0x00000038:
    I2C0CONSET = 0x00000020;
    break;
  case 0x00000050:
    i2c0_buf[i2c0_cnt] = I2C0DAT;
    i2c0_cnt = i2c0_cnt + 1;
  case 0x00000040:
    i2c0_num_of_bytes = i2c0_num_of_bytes - 1;
    if(i2c0_num_of_bytes) I2C0CONSET = 0x00000004;
    else I2C0CONCLR = 0x00000004;
    I2C0CONCLR = 0x00000020;
    break;
  case 0x00000058:
    i2c0_buf[i2c0_cnt] = I2C0DAT;
  case 0x00000048:
    if(I2C0ADR) I2C0CONSET = 0x00000004;
    else I2C0CONCLR = 0x00000004;
    i2c0_status = 0;
    I2C0CONCLR = 0x00000020;
    I2C0CONSET = 0x00000010;
    break;
  case 0x00000080:
```

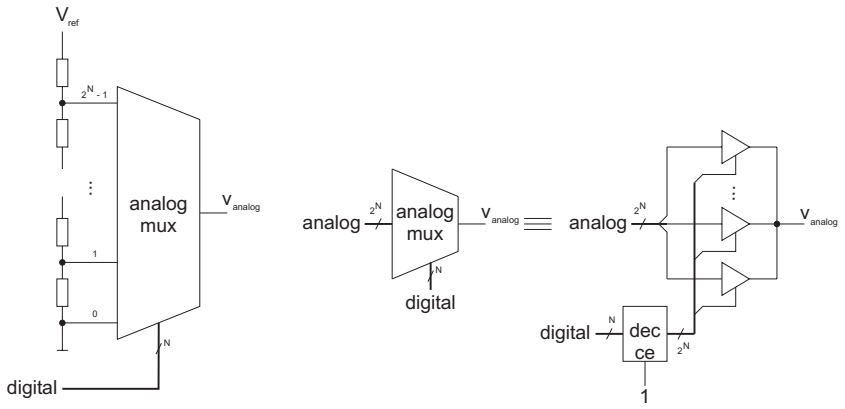
```
  case 0x00000090:
    int tmp = i2c0_rxslv_end;
    i2c0_rxslv[tmp] = I2C0DAT;
    tmp = tmp + 1;
    if(tmp == ...) tmp = 0; // Define buffer size of i2c0_rxslv
    if(tmp != i2c0_rxslv_begin) i2c0_rxslv_end = tmp;
    break;
  case 0x000000a8:
  case 0x000000b0:
    i2c0_txslv_cnt = 0;
  case 0x000000b8:
    I2C0DAT = i2c0_txslv[i2c0_txslv_cnt];
    i2c0_txslv_cnt = i2c0_txslv_cnt + 1;
    break;
  }
  I2C0CONCLR = 0x00000008;
}
```

Podobno, kot smo napisali gonilnik vmesnika i2c0, lahko napišemo tudi gonilnik vmesnika i2c1.

B.11 Digitalno analogni pretvornik

Za generiranje analogne napetosti iz digitalnega zapisa uporabljamo digitalno analogne pretvornike. Najpogosteje so narejeni z uporabo R-2R uporovne lestvice, ali pa s pomočjo uporovne verige. Digitalno analogna pretvorba z R-2R uporovno lestvico je primerna za pretvorbe 6 do 16-bitnih digitalnih števil in ima relativno malo uporov. Da z R-2R uporovno lestvico dosežemo zadovoljivo linearnost in monotonost, se morajo upori med seboj zelo dobro ujemati, kar ni trivialna naloga. Druga slaba stran R-2R uporovne lestvice so tokovni sunki

ob preklopih. Zato se v zadnjem času največkrat uporablja digitalno analogna pretvorba z uporovno verigo (slika B.13).



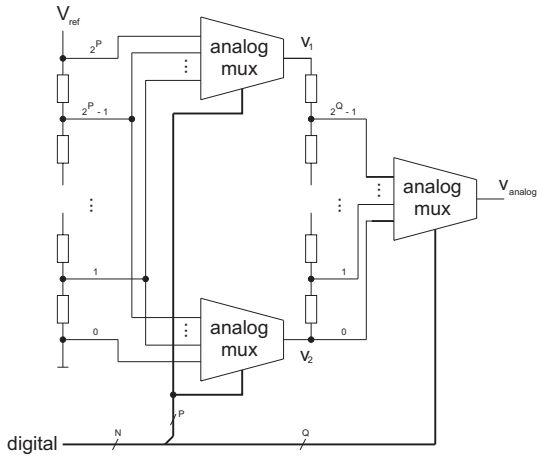
Slika B.13: Enostopenjski N-bitni digitalno analogni pretvornik z uporovno verigo

Iz slike B.13 vidimo, da je v uporovni verigi 2^N uporov. Dodan je analogni izbiralnik (angl. analog multiplexer), ki izmed 2^N nivojev napetosti izbere tistega, ki ga določa N-bitno digitalno število n. Izhodna napetost je enaka:

$$v_{analog} = \frac{n \times V_{ref}}{2^N} \tag{B.5}$$

Zaradi velikega števila uporov v verigi je enostopenjska izvedba digitalno analognega pretvornika z uporovno verigo primerna za pretvorbo največ 8-bitnih digitalnih števil. Pri hitrih pretvornikih predstavljajo težavo tudi zakasnitve v

analognem izbiralniku, ki naraščajo s številom bitov N. Enostopenjski pretvornik tako preoblikujemo v dvostopenjskega (slika B.14).



Slika B.14: Dvostopenjski N-bitni digitalno analogni pretvornik z uporovno verigo

N-bitno digitalno število n je sedaj razdeljeno na dva dela. In sicer P-bitni višji del p in Q-bitni nižji del q. Velja:

$$N = P + Q \qquad n = 2^Q \times p + q \tag{B.6}$$

Analogna izbiralnika izbirata med $2^P + 1$ nivoji napetosti in sta na prvostopenjsko uporovno verigo priključena zamaknjeno. Zgornji izbiralnik ne more izbrati nivoja mase, spodnji pa nivoja referenčne napetosti V_{ref}. Velja:

$$v_1 = \frac{(p+1) \times V_{ref}}{2^P} \qquad v_2 = \frac{p \times V_{ref}}{2^P} \tag{B.7}$$

Izhodni analogni izbiralnik izbira med 2^Q nivoji napetosti na drugostopenjski

uporovni verigi, na kateri se deli napetostna razlika $v_1 - v_2$. Izhodna analogna napetost je z upoštevanjem enačb (B.6) in (B.7) enaka:

$$v_{analog} = v_2 + \frac{q \times (v_1 - v_2)}{2^Q} = (2^Q \times p + q) \times \frac{V_{ref}}{2^N} = \frac{n \times V_{ref}}{2^N} \tag{B.8}$$

Philipsov mikrokrmilnik LPC2138 ima med perifernimi enotami vgrajen en 10-bitni digitalno analogno pretvornik, ki pri pretvorbi uporablja uporovno verigo. Njegovo delovanje podajajo biti v registru *DACR*.

DACR Digital to Analog Converter Register (naslov: `0xe006c000`)
Register podaja N=10-bitno digitalno število, ki se neprestano pretvarja v analogno napetost po enačbi (B.8), ter bit, ki določa hitrost pretvarjanja. Razporeditev bitov v registru *DACR* je enaka:

bit6 do **bit 15** 10-bitno digitalno število n

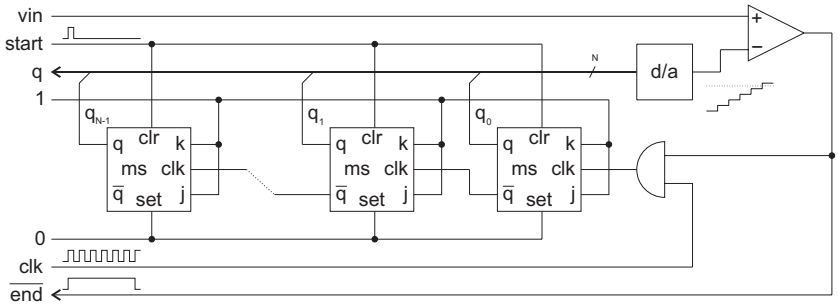
bit16 postavitev bita zmanjša porabo digitalno analognega pretvornika na največ 350μA, vendar se pri tem podaljša čas pretvorbe na 2.5μs. Normalna poraba je največ 700μA, čas pretvorbe pa 1μs. Omenjene vrednosti veljajo za bremena s kapacitivnostjo manjšo od 100pF.

Analogni izhod digitalno analognega pretvornika je na pinu P0.25. Da pin deluje kot analogni izhod, ga je potrebno nastaviti z masko *PINSEL1* = (*PINSEL1* & 0x000c0000) | 0x00080000. Razen nastavitve pina P0.25 digitalno analogni pretvornik ne potrebuje nikakršne inicializacije. Poleg pina P0.25 so pomembni še pini z napajalno in referenčno napetostjo. To sta pina V_{DDA} in V_{SSA}, ter pin V_{REF}. Napajalna napetost digitalno analognega pretvornika je enaka napajalni napetosti mikrokrmilnika (V_{DDA} = 3.3V in V_{SSA} = 0), vendar je zaradi zmanjšanja šuma in napak od nje ločena. Referenčna napetost se deli na uporovni verigi in je prav tako enaka napajalni napetosti ($V_{REF} = V_{ref}$ = 3.3V).

B.12 Analogno digitalni pretvornik

Analogno digitalni pretvornik izvaja ravno obratno funkcijo kot digitalno analogni pretvornik iz poglavja B.11. Velikost analogne napetosti prevede v digitalni zapis. Poznamo celo paleto izvedb. Najenostavneje analogno napetost pretvorimo v digitalno obliko s primerjavo z vsemi napetostnimi nivoji, ki ustrezajo

vsem digitalnim zapisom. Napetostne nivoje dobimo s pomočjo uporovne verige, kjer se v vsakem vozlišču nahaja primerjalnik (angl. comparator) za primerjavo z vhodno analogno napetostjo. Takšni analogno digitalni pretvorniki so izredno hitri, vendar potrebujejo za pretvorbo v N-bitni digitalni zapis $2^N - 1$ primerjalnikov. Zato jih v mikrokrmilnikih ne srečujemo. Zaradi velikega števila primerjalnikov na vhodu imajo visoko vhodno kapacitivnost.

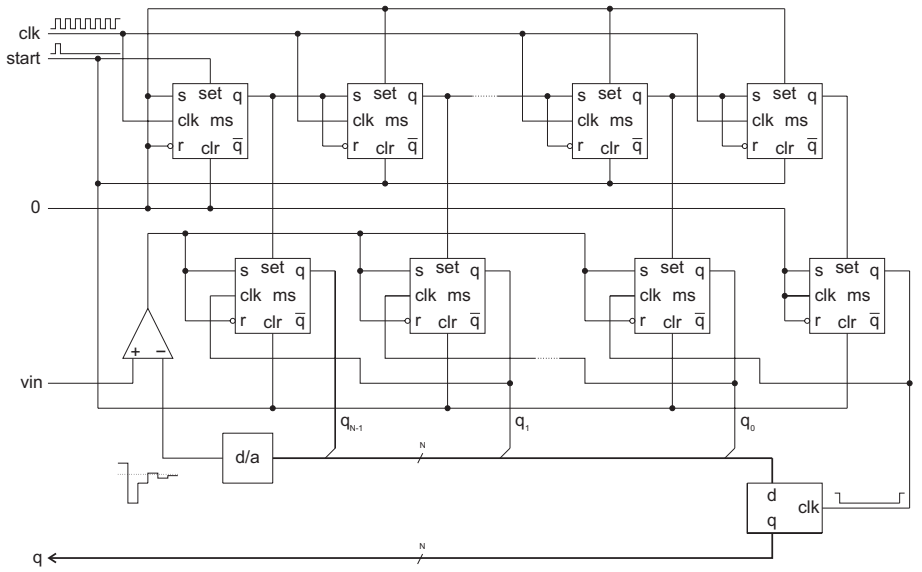


Slika B.15: Analogno digitalni pretvornik s števnikom

Drugi princip analogno digitalne pretvorbe prikazuje slika B.15. Števnik šteje impulze urinega signala. Stanje števnika se v digitalno analognem pretvorniku pretvori v analogno napetost, ki se primerja z neznano vhodno napetostjo. Ko sta napetosti enaki, števnik preneha s štetjem.

Pretvorbo sproži impulz *start*, ki stanje števnika postavi na nič. Pretvorjena napetost je tako enaka nič. Ob vhodni napetosti večji od nič je izhod primerjalnika na visokem nivoju. Števnik prične s štetjem urinih impulzov, kar

dela toliko časa, dokler pretvorjena napetost ne doseže vhodne. Nizko staje na izhodu primerjalnika ($\overline{end}$) pomeni, da je pretvorba končana.



Slika B.16: Analogno digitalni pretvornik z registrom z zaporednim približevanjem

Slaba stran analogno digitalne pretvorbe s števnikom je dolg čas pretvorbe. Za pretvorbo analogne vrednosti je v najslabšem primeru potrebnih kar $2^N - 1$ urinih impulzov. Zato se namesto števnika uporablja register z zaporednim približevanjem (angl. Successive Approximation Register - SAR) na sliki B.16. Pretvorbo zopet sproži impulz *start*, ki vse celice, razen prvih dveh, postavi v nizko stanje. Ob vsakem urinem impulzu se enica v zgornji vrsti celic pomakne za eno mesto desno. Pri tem se pripadajoča celica v drugi vrsti postavi v visoko stanje, kar povzroči, da se celica pred njo postavi tako, kot narekuje izhod primerjalnika. Biti zavzemajo pravilne vrednosti od najbolj do najmanj pomembnega. Pretvorba traja N ciklov urinega signala. Dodan je še zatič,

ki ob koncu shrani pretvorjeno vrednost. Tako je branje pretvorjene vrednosti mogoče, čeprav se je med tem že začela nova pretvorba.

Philipsov mikrokrmilnik LPC2138 ima med perifernimi enotami vgrajena dva 10-bitna analogno digitalna pretvornika adc0 in adc1 z zaporednim približevanjem. Vsak izmed njiju lahko pretvarja napetosti na največ osmih različnih analognih vhodih. Pretvorbe si lahko sledijo neprekinjeno, ali pa jih prožimo. V prvem načinu pretvorbi, ki se konča, takoj sledi nova pretvorba na naslednjem analognem vhodu. V drugem načinu pretvorbo na izbranem analognem vhodu v želenem trenutku sproži programska oprema, ali zunanji signal. Podrobnejši opis obeh enot adc0 in adc1 se nahaja v [5].

Registra, s katerima nastavljamo delovanje analogno digitalnega pretvornika adc0, sta opisana v nadaljevanju. Enaka registra za pretvornik adc1 se nahajata istih naslovih, ki jima prištejemo konstanto `0x00002c00`. Register *ADGSR* je skupen obema pretvornikoma.

$$f_{ad} = \frac{f_{vpb}}{CLKDIV + 1} \leq 4.5\text{Mhz} \tag{B.9}$$

AD0CR A/D Control Register (naslov: `0xe0034000`)
Podaja način delovanja analogno digitalnega pretvornika adc0. Posamezni biti v registru imajo naslednji pomen:

bit7 do **bit0** določajo pine, oziroma analogne vhode, ki bodo pretvorjeni. Analogni vhod AD0.0 izberemo s postavitvijo bita bit0, analogni vhod AD0.7 pa s postavitvijo bita bit7. Če analogno digitalno pretvorbo proži programska oprema, mora biti izbran le en analogni vhod. Pri neprekinjenem proženju je lahko izbranih poljubno mnogo vhodov. Način proženja določa bit16.

bit15 do **bit8** podajajo 8-bitno konstanto *CLKDIV*, ki po enačbi (B.9) določa frekvenco urinega signala f_{ad} uporabljenega v analogno digitalnem pretvorniku. Frekvenca urinega signala f_{ad} je lahko največ 4.5MHz.

bit16 s postavitvijo izberemo neprekinjeno proženje. Da se neprekinjeno proženje prične, morajo biti bit26, bit25 in bit24 enaki nič.

bit19, **bit18** in **bit17** določajo natančnost in hitrost neprekinjenih pretvorb. Posamezne vrednosti določajo naslednje natančnosti, oziroma hitrosti:

- 000 ... 10 bitov, 11 urinih impulzov f_{ad}
- 001 ... 9 bitov, 10 urinih impulzov f_{ad}
- 010 ... 8 bitov, 9 urinih impulzov f_{ad}
- 011 ... 7 bitov, 8 urinih impulzov f_{ad}
- 100 ... 6 bitov, 7 urinih impulzov f_{ad}
- 101 ... 5 bitov, 6 urinih impulzov f_{ad}
- 110 ... 4 biti, 5 urinih impulzov f_{ad}
- 111 ... 3 biti, 4 urini impulzi f_{ad}

Programsko prožene pretvorbe vedno trajajo 11 urinih impulzov f_{ad} in imajo 10-bitno natančnost.

bit21 postavitev omogoči delovanje analogno digitalnega pretvornika

bit26, **bit25** in **bit24** podajajo trenutek, ko naj se programsko prožena pretvorba prične. Vrednost 000 pomeni, naj se pretvorba ne prične. Uporabimo jo, kadar imamo opravka z neprekinjenimi pretvorbami (bit16 $\rightarrow$ 1), ali kadar analogno digitalni pretvornik izklopimo (bit21 $\rightarrow$ 0). Pretvorbo programsko sprožimo z zapisom vrednosti 001. Vrednosti 010 do 111 določajo začetek pretvorbe glede na izbrano fronto pripadajočega zunanjega signala. Tip fronte izbira bit27.

bit27 s postavitvijo izberemo padajočo fronto pripadajočega zunanjega signala

AD0DR A/D Data Register (naslov: `0xe0034004`)
V registru se nahaja pretvorjena digitalna vrednost ter informacije, ki podajajo njen status. Posamezni biti imajo naslednji pomen:

bit15 do **bit6** 10-bitno nepredznačeno število, ki predstavlja razmerje $V \div V_{ref}$ pretvorjene analogne napetosti V glede na referenčno napetost V_{ref}. Število je veljavno, ko je pretvorba končana, kar označuje bit31.

bit26, **bit25** in **bit24** podajajo številko analognega vhoda (AD0.7 do AD0.0), ki je bil pretvorjen

bit30 postavljen pomeni, da najmanj ena pretvorjena vrednost ni bila prebrana in je bila nadomeščena z novo. Zastavica se postavlja, kadar je izbrano neprekinjeno proženje (bit16 registra *AD0CR*). Zastavica se ob branju vsebine registra *AD0DR* postavi na nič.

bit31 postavljen pomeni, da je pretvorba končana. Poleg tega analogno digitalni pretvornik ob koncu pretvorbe, torej ob postavitvi tega bita, vektorskemu nadzorniku prekinitev (glej dodatek B.4) poda zahtevo po prekinitvi. Zastavica se ob branju vsebine registra *AD0DR* postavi na nič.

ADGSR A/D Global Start Register (naslov: `0xe0034008`)
Register je skupen obema analogno digitalnima pretvornikoma in lahko vanj le pišemo. Omogoča sočasen začetek pretvorb na obeh pretvornikih. Z drugimi besedami zapis v register *ADGSR* pomeni hkraten zapis bitov, ki določajo začetek pretvorbe, v kontrolna registra *AD0CR* in *AD1CR* obeh pretvornikov. Ti biti so bit27 do bit24 in bit16, ter imajo enak pomeni kot istoležni biti v registrih *AD0CR* in *AD1CR*.

Vhodi analogno digitalnih pretvornikov adc0 in adc1 vgrajenih v Philipsov mikrokrmilnik LPC2138 so povezani na pine vzporednih vrat P0. Tabela B.12 podaja nabor pinov, kjer se analogni vhodi nahajajo. Da posamezen pin deluje kot vhod analogno digitalnega pretvornika, ga je potrebno temu primerno nasta-

viti. Maske registrov *PINSEL0* in *PINSEL1*, s katerimi za izbran pin dosežemo želeno nastavitev, so podane v zadnjem stolpcu tabele B.12.

P0.4	AD0.6	*PINSEL0* = *PINSEL0* | `0x00000300`
P0.5	AD0.7	*PINSEL0* = *PINSEL0* | `0x00000c00`
P0.6	AD1.0	*PINSEL0* = *PINSEL0* | `0x00003000`
P0.8	AD1.1	*PINSEL0* = *PINSEL0* | `0x00030000`
P0.10	AD1.2	*PINSEL0* = *PINSEL0* | `0x00300000`
P0.12	AD1.3	*PINSEL0* = *PINSEL0* | `0x03000000`
P0.13	AD1.4	*PINSEL0* = *PINSEL0* | `0x0c000000`
P0.15	AD1.5	*PINSEL0* = *PINSEL0* | `0xc0000000`
P0.21	AD1.6	*PINSEL1* = (*PINSEL1* & `0xfffff3ff`) | `0x00000800`
P0.22	AD1.7	*PINSEL1* = (*PINSEL1* & `0xffffcfff`) | `0x00001000`
P0.25	AD0.4	*PINSEL1* = (*PINSEL1* & `0xfff3ffff`) | `0x00040000`
P0.26	AD0.5	*PINSEL1* = (*PINSEL1* & `0xffcfffff`) | `0x00100000`
P0.27	AD0.0	*PINSEL1* = (*PINSEL1* & `0xff3fffff`) | `0x00400000`
P0.28	AD0.1	*PINSEL1* = (*PINSEL1* & `0xfcffffff`) | `0x01000000`
P0.29	AD0.2	*PINSEL1* = (*PINSEL1* & `0xf3ffffff`) | `0x04000000`
P0.30	AD0.3	*PINSEL1* = (*PINSEL1* & `0xcfffffff`) | `0x10000000`

Tabela B.12: Pini analogno digitalnih pretvornikov adc0 in adc1

Poleg analognih vhodov so za analogno digitalna pretvornika adc0 in adc1 pomembni še pini z napajalno in referenčno napetostjo. To sta pina V_{DDA} in V_{SSA}, ter pin V_{REF}. Napajalna napetost analogno digitalnega pretvornika je enaka napajalni napetosti mikrokrmilnika (V_{DDA} = 3.3V in V_{SSA} = 0), vendar je zaradi zmanjšanja šuma in napak od nje ločena. Referenčna napetost se deli na uporovni verigi in je prav tako enaka napajalni napetosti (V_{REF} = V_{ref} = 3.3V).

Sledi primer inicializacije analogno digitalnega pretvornika adc0, ki naj pretvarja analogno napetost z vhoda AD0.0. Pin P0.27 definiramo kot analogni vhod, v registru *AD0CR* pa izberemo vhod AD0.0. Vrednost konstante *CLKDIV* je enaka nič, pri čemer predpostavljamo, da je frekvenca urinega signala

perifernega vodila VPB $f_{vpb} < 4.5$MHz. Pretvorbe bodo prožene programsko s postavitvijo bita bit24.

```
PINSEL1 = (PINSEL1 & 0xff3fffff) | 0x00400000;
AD0CR = 0x00200001;
```

Pretvorbo analogne napetosti na pinu P0.27 zahtevamo s postavitvijo bita bit24 v registru *AD0CR*. Pretvorjeno vrednost preberemo iz registra *AD0DR*, kjer konec pretvorbe označuje bit bit31. Ponovno pretvorbo zahtevamo s ponovno postavitvijo bita bit24 v registru *AD0CR*.

B.13 Zapis v pomnilnik flash

Philipsov mikrokrmilnik LPC2138 ima vgrajen pomnilnik flash velikosti 512kB, ki se nahaja na naslovih od `0x00000000` do `0x0007ffff` (glej dodatek B.1). V njem se nahaja programska koda (glej dodatek C.1). Poleg tega v pomnilnik flash zapisujemo podatke, ki jih hočemo ohraniti tudi v primeru izklopa napajanja.

O različnih vrstah pomnilnikov smo govorili v poglavju 2.8. Na tranzistorskem nivoju je izvedba pomnilne celice pomnilnika flash pravzaprav enaka izvedbi celice ROM (slika 2.30). Namesto navadnih tranzistorjev so v pomnilniku flash uporabljeni posebni tranzistorji z dodanimi plavajočimi vrati med kanalom in pravimi vrati. Ti tranzistorji se seveda nahajajo na vseh križiščih naslovnih in podatkovnih linij. Plavajoča vrata so povsem izolirana od okolice, naboj ujet na njih pa lahko izniči vpliv pravih vrat in tako prepreči prevajanje kanala. Z drugimi besedami, naboj ujet na plavajočih vratih določa, ali je na pripadajočem križišču zapisano visoko, ali nizko stanje. Plavajoča vrata nabijemo ali razelektrimo s primerno visokimi napetostmi na sponkah tranzistorja. Napetosti višje od napajalne zagotovi posebno vezje, to je nabojna črpalka. Podrobna razlaga delovanja pomnilnika flash presega domet te skripte. Obstaja več različnih izvedb pomnilnikov flash, kar tudi določa njihove bralno pisalne lastnosti.

Brisanje pomnilne celice pomnilnika flash pravzaprav pomeni nabitje plavajočih vrat, oziroma zapis visokega stanja. V izbrisanem pomnilniku flash so tako vsa plavajoča vrata nabita. Vanj podatke zapisujemo tako, da ustrezna plavajoča vrata razelektrimo. Pomnilnik flash vgrajen v Philipsov mikrokrmilnik LPC2138 lahko brišemo (plavajoča vrata nabijemo) le po sektorjih velikih 4kB ali 32kB. Posamezne celice ni mogoče izbrisati ločeno. Podatke vanj zapisujemo (ustrezna plavajoča vrata razelektrimo) v blokih velikih 256B, 512B, 1kB

ali 4kB. Zaradi opisane narave pomnilnika flash je potrebno pred vsakim novim pisanjem po že zapisanem bloku izbrisati celoten sektor, kjer se blok nahaja.

sektor	velikost	naslovni prostor
0	4kB	`0x00000000 - 0x00000fff`
1	4kB	`0x00001000 - 0x00001fff`
2	4kB	`0x00002000 - 0x00002fff`
3	4kB	`0x00003000 - 0x00003fff`
4	4kB	`0x00004000 - 0x00004fff`
5	4kB	`0x00005000 - 0x00005fff`
6	4kB	`0x00006000 - 0x00006fff`
7	4kB	`0x00007000 - 0x00007fff`
8	32kB	`0x00008000 - 0x0000ffff`
9	32kB	`0x00010000 - 0x00017fff`
10	32kB	`0x00018000 - 0x0001ffff`
11	32kB	`0x00020000 - 0x00027fff`
12	32kB	`0x00028000 - 0x0002ffff`
13	32kB	`0x00030000 - 0x00037fff`
14	32kB	`0x00038000 - 0x0003ffff`
15	32kB	`0x00040000 - 0x00047fff`
16	32kB	`0x00048000 - 0x0004ffff`
17	32kB	`0x00050000 - 0x00057fff`
18	32kB	`0x00058000 - 0x0005ffff`
19	32kB	`0x00060000 - 0x00067fff`
20	32kB	`0x00068000 - 0x0006ffff`
21	32kB	`0x00070000 - 0x00077fff`
22	4kB	`0x00078000 - 0x00078fff`
23	4kB	`0x00079000 - 0x00079fff`
24	4kB	`0x0007a000 - 0x0007afff`
25	4kB	`0x0007b000 - 0x0007bfff`
26	4kB	`0x0007c000 - 0x0007cfff`

Tabela B.13: Razdelitev pomnilnika flash po sektorjih

Razdelitev pomnilnika flash po sektorjih je prikazana v tabeli B.13. Vidimo, da je po sektorjih razdeljenih le prvih 500kB pomnilnika flash. Zadnjih 12kB od naslova `0x0007d000` do `0x0007ffff` manjka (angl. boot sector). Tu se

nahaja prednaložena programska koda, ki vsebuje tudi podprograme za brisanje in pisanje v pomnilnik flash. Torej imamo za lastno programsko kodo in podatke v resnici na voljo le 500kB. Prednaložena programska koda je narejena za tek z naslovov `0x7fffd000` do `0x7fffffff` in se takoj po zagonu mikrokrmilnika tja tudi preslika (slika B.1). Na naslovih `0x0007d000` do `0x0007ffff`, ter naslovih `0x7fffd000` do `0x7fffffff` zato vidimo enako vsebino.

V pomnilnik flash lahko podatke zapisujemo na dva oziroma tri načine. Ob zagonu Philipsov mikrokrmilnik LPC2138 preveri napetost pina P0.14. Če je na njem prisoten visok nivo, se mikrokrmilnik normalno zažene. V nasprotnem primeru (nizek nivo na P0.14) se zažene prednaložena programska koda (angl. boot loader), ki preko zaporednega splošnega asinhronega sprejemnika in oddajnika uart0 sprejme vsebino in jo zapiše v pomnilnik flash. To imenujemo način zapisovanja ISP (angl. In-System Programming).

V pomnilnik flash lahko podatke piše tudi programska koda med delovanjem po normalnem zagonu mikrokrmilnika. To naredi s klicem ustreznega prednaloženega podprograma, kar imenujemo način zapisovanja IAP (angl. In Application Programming). V tem razdelku bo v nadaljevanju na kratko opisan le način IAP. Zapis v pomnilnik flash pa je mogoč še na tretji način, in sicer preko vgrajenega zaporednega vmesnika JTAG (angl. Joint Test Action Group), ki podatke v pomnilnik pravzaprav zapisuje v načinu IAP.

Prednaloženi podprogram IAP za svoje delovanje potrebuje zadnjih 32 bajtov pomnilnika RAM od naslova `0x40007fe0` do `0x40007fff`. Programska koda, ki za pisanje v pomnilnik flash uporablja način IAP, tega prostora ne sme uporabiti za shranjevanje drugih podatkov ali za sklad. Poleg tega prednaloženi podprogram med svojim tekom potrebuje še do 128 bajtov prostora na skladu, ki narašča proti nižjim naslovom (glej dodatek C.4). Med klicem podprograma IAP pomnilnik flash ni dostopen. Koda na prekinitvenih naslovih (tabela C.3) ni dosegljiva, zato morajo biti prekinitve medtem onemogočene.

Da bi prekinitve nemoteno delovale tudi med klicem podprograma IAP je potrebno izpolniti dva pogoja. Najprej je potrebno zagotoviti, da se koda na prekinitvenih naslovih namesto s pomnilnika flash prebere s pomnilnika RAM. Preslikavo (slika B.1) naredimo tako, da v register *MEMMAP* na naslovu `0xe01fc040` zapišemo vrednost `0x02`. Na začetku pomnilnika RAM se mora seveda nahajati ustrezna koda, oziroma kopije ukazov z naslova `0x00000000`. Primer, ko za kopiranje prekinitvenih ukazov in vzpostavitev preslikave poskrbi zagonska koda, je na strani 179. Drugi pogoj je, da se tudi programska koda prekinitvenega

podprograma nahaja v pomnilniku RAM. Poleg tega prekinitvena koda seveda ne sme uporabljati pomnilnika flash.

Podprogram IAP se nahaja na naslovu `0x7ffffff0` in je napisan v okleščenem naboru ukazov *Thumb* (glej dodatek C). Koda podprograma IAP se začne izvajati ob skoku na neporavnan naslov *0x7ffffff1*, ki hkrati povzroči preklop v način *Thumb*. Argument podprograma IAP se nahaja v delovnem registru *r0*. To je kazalec na polje 32-bitnih podatkov, ki pravzaprav predstavljajo poljubno število argumentov podprograma IAP. Prvi 32-bitni podatek v polju podaja kodo ukaza, za katerega želimo, da ga podprogram IAP izvrši. Izmed ukazov, ki so na voljo, na tem mestu navedimo le tri najpomembnejše. Podrobnejši opis najde bralec v [5].

Priprava sektorjev na brisanje ali zapis
Ukaz pripravi enega ali več sektorjev pomnilnika flash na operacijo brisanja ali pisanja. Kazalec v delovnem registru *r0* kaže na polje treh 32-bitnih podatkov, in sicer:

- koda ukaza, ki je enaka `0x00000032`,
- številka prvega sektorja in
- številka zadnjega sektorja.

V kolikor želimo na brisanje ali zapis pripraviti le en sektor, sta številki prvega in zadnjega sektorja enaki.

Brisanje sektorjev
Ukaz izbriše enega ali več sektorjev v pomnilniku flash. Pred ukazom se mora

uspešno izvesti priprava izbranih sektorjev na brisanje. Kazalec v delovnem registru *r0* kaže na polje treh 32-bitnih podatkov, in sicer:

- koda ukaza, ki je enaka `0x00000034`,
- številka prvega sektorja in
- številka zadnjega sektorja.

V kolikor želimo izbrisati le en sektor, sta številki prvega in zadnjega sektorja enaki.

Zapis podatkov v pomnilnik flash iz pomnilnika RAM
Ukaz v pomnilnik flash prepiše vsebino bloka podane velikosti, ki se nahaja v pomnilniku RAM. Pred ukazom se mora uspešno izvesti priprava sektorjev, v katere se bo zapis bloka podatkov izvršil, na pisanje. Kazalec v delovnem registru *r0* kaže na polje petih 32-bitnih podatkov, in sicer:

- koda ukaza, ki je enaka `0x00000033`,
- ciljni na 256 bajtov poravnani naslov (`0x******00`) v pomnilniku flash,
- izvorni na 4 bajte poravnani naslov začetka bloka podatkov,
- dolžina bloka podatkov (256, 512, 1024 ali 4096 bajtov) in
- frekvenca urinega signala perifernega vodila f_{vpb} v kHz.

Podprogram IAP, ki izbran ukaz izvrši, pokličemo z zbirniškim ukazom `bx`. Pred tem je seveda potrebno pripraviti polje z vhodnimi podatki ukaza, v de-

lovni register *r0* pa zapisati naslov polja. Sledi odsek zbirniške kode, ki kliče podprogram IAP.

```
                    /* Prepare arguments at the address */
                    ldr r0, =address
                    mov r2, #0x7ffffff1
                    ldr lr, =return_address
                    bx  r2
return_address: /* Returned value resides in r0 */
```

koda v *r0*	
0	ukaz se je uspešno izvršil
2	neporavnan izvorni naslov
3	neporavnan ciljni naslov
4	izvorni naslov se ne nahaja v pomnilniku RAM
5	ciljni naslov se ne nahaja v pomnilniku flash
6	nepravilna dolžina bloka
7	nepravilna številka sektorja
9	nepripravljen sektor
11	vmesnik za programiranje pomnilnika flash zaseden

Tabela B.14: Vrnjene vrednosti podprograma IAP

Ko se podprogram IAP konča, v delovnem registru *r0* vrne kodo, ki pove, ali se je ukaz izvršil uspešno, oziroma kaj je narobe. Vrnjene vrednosti z razlago so zbrane v tabeli B.14.

Dodatek C

Zgradba procesnega jedra mikrokrmilnika Philips LPC2138

Primeri v tej skripti so pisani za Philipsov mikrokrmilnik LPC2138, ki temelji na centralnem procesnem jedru ARM7TDMI-S [3]. Jedro je član širše družine splošnih 32-bitnih mikroprocesorskih jeder ARM, ki temeljijo na RISC (angl. Reduced Instruction Set Computer) principih. RISC nabor zbirniških ukazov in pripadajoč dekodirni mehanizem je v primerjavi s CISC (angl. Complex Instruction Set Computer) naborom mnogo enostavnejši, kar se odraža predvsem v hitrejšem izvajanju ukazov (načeloma ni mikrokode, ukazi so ožičeni). Posledica tega je odličen odziv na prekinitve, ter enostavnejša izvedba.

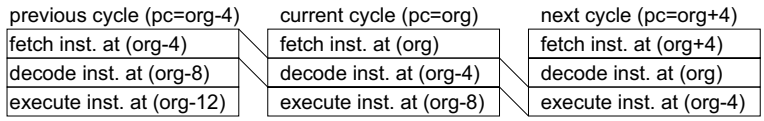
ARM7TDMI-S temelji na arhitekturi ARMv4T. Podrobnejši opis nabora zbirniških ukazov te različice je mogoče najti v [6] in [10]. Arhitektura ARMv4T lahko deluje z dvema različnima naboroma ukazov. In sicer z 32-bitnim tako imenovanim naborom *ARM*, in s 16-bitnim okleščenim naborom *Thumb*. Tukaj bomo na kratko opisali le nabor *ARM*.

C.1 Cevovodna arhitektura

Centralno procesno jedro ARM7TDMI-S ima tri stopenjsko cevovodno zgradbo

za izvajanje strojnih ukazov (angl. instruction pipeline). To pomeni, da se vsak ukaz izvrši v treh stopnjah (slika C.1), in sicer:

- v prvi stopnji je ukaz prebran iz pomnilnika,
- druga stopnja ga dekodira, nakar
- se v tretji stopnji ukaz izvrši.



Slika C.1: Cevovodna arhitektura izvajanja ukazov

Za vsako stopnjo ima centralno procesno jedro neodvisno vezje. Tako se vse tri stopnje izvajajo hkrati. Medtem, ko se nek ukaz v tretji stopnji izvršuje, se ob istem času ukaz za njim v drugi stopnji dekodira, hkrati pa procesno jedro v prvi stopnji že išče še naslednji ukaz. V vsakem trenutku so v obravnavi trije ukazi hkrati, vsak na svoji stopnji. Posledica tega je, da se ukaz navidezno izvrši v enem strojnem ciklu, kar močno poveča učinkovitost procesnega jedra. Vendar cevovodna zgradba učinkovito deluje le na linearni programski kodi, to je kodi, ki ne vsebuje skokov, oziroma vejitev. V primeru vejitve je potrebno cevovod izprazniti, ter ga na novo napolniti z ukazi, kamor je bil skok narejen, zaradi česar je izgubljenih nekaj strojnih ciklov.

Cevovodna zgradba je del centralnega procesnega jedra in je tako navzven transparentna. Programer je ne vidi. Z njegovega vidika je pomembno le, da se ukazi, razen vejitev in skokov, izvršijo v enem ciklu. Prav tako je koristno vedeti, da programski števnik v trenutku izvršitve ukaza kaže osem bajtov naprej, to je na ukaz, ki ga procesno jedro v tem trenutku bere iz pomnilnika. Efekt nazorno prikazuje naslednji ukaz, katerega koda naj se nahaja na naslovu `0x00000100`:

```
address      code
0x00000100   mov  r0, pc
```

Ukaz v delovni register *r0* naloži vsebino programskega števnika *pc* (glej dodatek C.5). Na prvi pogled se zdi, da bo v *r0* naložen naslov, kjer se ukaz nahaja,

torej `0x00000100`. Vendar je v trenutku izvajanja ukaza programski števnik že dva ukaza (osem bajtov) naprej. Tako se v delovni register *r0* naloži konstanta `0x00000108`.

Za učinkovito delovanje cevovodne zgradbe je potreben hiter dostop do pomnilnika, kjer se nahaja programska koda. Le ta je navadno v pomnilniku flash, ki pa ni dovolj hiter. Njegov dostopni čas je 50ns, kar ustreza urinemu signalu procesnega jedra $f_{clk} = 20\text{MHz}$. Pri višjih frekvencah branje naslednjega ukaza v vsakem ciklu ni več mogoče. Prav tako delovanje cevovodne arhitekture zavirajo dostopni časi do vsebine registrov perifernih enot. Zaradi tega mora procesno jedro večkrat čakati na zahtevane podatke. To dalje pomeni, da je hitrost izvajanja ukazov pravzaprav manjša, kot en ukaz na cikel. Cevovodna zgradba hitrost izvajanja ukazov sicer močno poveča, vendar idealne hitrosti en ukaz na cikel, zaradi dolgih dostopnih časov, ne doseže. Iz vsega povedanega sledi tudi, da je čas izvajanja napisane kode težko točno določiti, kajti časi trajanja posameznih ukazov niso vnaprej znani, oziroma bi bilo potrebno za vsak ukaz preveriti, s katerimi deli pomnilnika ima opravka.

Dostopne čase do ukazov programske kode je mogoče zmanjšati na dva načina. In sicer z uporabo pomnilniškega pospeševalnika (glej dodatek B.3), ali pa s tem, da kodo preselimo iz počasnega pomnilnika flash v hitri pomnilnik RAM. Ob zagonu je vsebina pomnilnika RAM naključna. Programska koda se seveda nahaja v pomnilniku flash. Zato jo je potrebno v RAM najprej prepisati, kar naredi naslednji del izvorne kode:

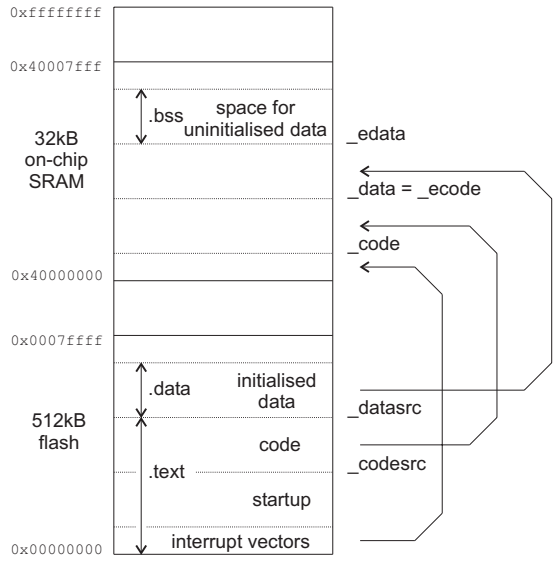
```
/* Parameters */
           .equ    irq_stack,          0x40007fe0
           .equ    usr_stack,          0x40006000
/* Constants */
           .equ    flash,              0x00000000
           .equ    ram,                0x40000000
           .equ    ram_end_table,      0x40000040
           .equ    word_len,           0x04
           .equ    irq_m,              0x12
           .equ    usr_m,              0x10
           .equ    disable_ints,       0xffffffff
           .equ    prot_val,           0x00
           .equ    remap_val,          0x02
/* Registers */
           .equ    memmap,             0xe01fc040
           .equ    vicintenclear,      0xfffff014
```

```
            .equ    vicprotection,      0xfffff020
/* Global symbols */
            .global start_up /* for C function start_up() */
            .global sch_on
            .global main_prog
            .global mam_init
            .global vpbdiv_init
            .global pll_init

            .code   32

/* Startup code */
            .text   0               /* at 0x00000000 */
            b       reset

            .text   1               /* at 0x00000040 */
/* Reset interrupt service routine */
reset:      ldr     r0, =flash
            ldr     r1, =ram
            ldr     r2, =ram_end_table
            bl      copy
            ldr     r0, =_codesrc
            ldr     r1, =_code
            ldr     r2, =_ecode
            bl      copy
            ldr     r0, =_datasrc
            ldr     r1, =_data
            ldr     r2, =_edata
            bl      copy
            mov     r0, #0x00
            ldr     r1, =_bss
            ldr     r2, =_ebss
clear_bss:  cmp     r1, r2
            strlo   r0, [r1], #word_len
            blo     clear_bss
            ldr     r0, =vicintenclear
            ldr     r1, =disable_ints
            str     r1, [r0]
            ldr     r0, =vicprotection
```

```
                mov     r1, #prot_val
                str     r1, [r0]
                ldr     r0, =memmap
                mov     r1, #remap_val
                str     r1, [r0]
                msr     cpsr_c, #irq_m
                ldr     sp, =irq_stack
                msr     cpsr_c, #usr_m
                ldr     sp, =usr_stack
                ldr     pc, =start_up

copy:           cmp     r1, r2
                ldrlo   r3, [r0], #word_len
                strlo   r3, [r1], #word_len
                blo     copy
                mov     pc, lr

/* Startup program */
start_up:       bl      init
                bl      sch_on
                b       main_prog

/* Subroutine calls MAM, VPB and PLL initialisations */
init:           stmfd   sp!, {lr}
                bl      mam_init
                bl      vpbdiv_init
                bl      pll_init
                ldmfd   sp!, {lr}
                mov     pc, lr
```

To je prva stvar, ki se ob zagonu izvrši. Na začetek pomnilnika RAM se prepiše vsebina na prekinitvenih naslovih (glej dodatek C.3). Tako so ob ustrezni postavitvi registra *MEMMAP* ukazi na prekinitvenih naslovih dostopni tudi v primeru, ko je pomnilnik flash nedosegljiv (glej dodatek B.13). Nato se vsebina pomnilnika flash iz naslova *_codesrc* dolžine (*_code* − *_ecode*) bajtov prepiše na naslov *_code* v pomnilniku RAM. Na enak način je prepisana še vsebina iz naslova *_datasrc* dolžine (*_data* − *_edata*) na naslov *_data*. Prvo kopiranje prepiše ukaze na prekinitvenih naslovih, drugo programsko kodo iz vseh pododsekov `.text`, tretje pa inicializirane podatke iz pododsekov `.data` (glej sliko

C.2). Za naslove *_codesrc*, *_code*, *_ecode*, *_datasrc*, *_data* in *_edata* poskrbi povezovalnik [7].



Slika C.2: Primer razdelitve pomnilnika po odsekih in kopiranje programske kode ob zagonu

Izvorna koda zgoraj se nahaja v posebni zagonski datoteki in se izvaja iz pomnilnika flash. V pomnilnik RAM se ne prepiše, čeprav se nahaja v enem izmed pododsekov `.text`. To je povedano v navodilih povezovalniku [7]. V RAM se prepišejo le tisti pododseki `.text`, ki se ne nahajajo v zagonski datoteki, ter vsebina prekinitvenih naslovov. Ob tem povezovalnik poskrbi tudi za pravilne vrednosti absolutnih naslovov v programski kodi, ki bo tekla iz pomnilnika RAM.

S tem, ko se programska koda izvaja iz pomnilnika RAM, se je mogoče izogniti počasnemu pomnilniku flash, vendar pa ne tudi dostopom do registrov perifernih enot.

Po končanem kopiranju programske kode so onemogočene vse prekinitve (register *VICIntEnable*, glej dodatek B.4), ki bodo definirane na novo. Nastavljanje prekinitev naj bo mogoče tudi iz neprivilegiranega uporabniškega načina delovanja (register *VICProtection*). Z ustrezno postavitvijo registra *MEMMAP*

se na prekinitvene naslove preslika začetek pomnilnika RAM (slika B.1). Pred pričetkom uvodnega dela glavnega programa *start_up* sta postavljena še dva kazalca sklada, kar je podrobneje razloženo v dodatku C.4. Uvodni program opravi razne inicializacije zbrane v podprogramu *init*, ter s klicem podprograma *sch_on* (stran 54) požene operacijski sistem, oziroma razvrščevalnik.

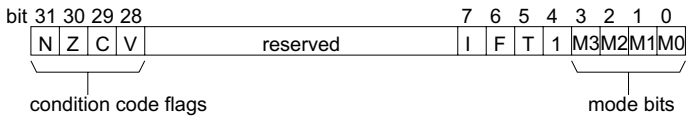
Zagonska programska koda, ki kopira kodo v pomnilnik RAM, ter postavlja začetne vrednosti kazalcev skladov, je vedno napisana v zbirniku. Uvodni program, oziroma funkcija *start_up()*, pa je že lahko prestavljena v programski jezik C. Funkcija *start_up()* kliče funkciji *init()*, kjer so zbrane inicializacije (glej dodatek B), in *sch_on()* (stran 56). Argumenti obeh funkcij predstavljajo parametre delovanja programa, ki so tako zbrani na enem mestu.

```
// timeslice = prescale_val * match_val * vpb_div / clock_rate
#define clock_rate   12
#define vpb_div      cclk_4
#define prescale_val 1
#define match_val    3000000

typedef void (* voidfuncptr)();

#define cclk_4 0x00
#define timer0 0x00000010
#define mr0i   0x00000001
#define mr0r   0x00000002
#define timer  0x00000000

extern void sch_int();
extern void mam_init(int);
extern void set_vpbdiv(int);
extern void pll_init(int);
extern void sch_on(int, int *, int, int, int, int,
    voidfuncptr *, int *, voidfuncptr);
extern void main_prog();

// Startup program
void start_up() {
    int match[4] = {match_val - 1, 0, 0, 0}, intr[16] =
        {timer0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
    voidfuncptr fnct[16] =
        {sch_int, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    init(clock_rate, vpb_div);
    sch_on(prescale_val - 1, match, mr0i | mr0r, timer,
        0x00000000, timer0, fnct, intr, 0);
    main_prog();
}

// MAM, VPB, PLL and pin initialisation
// clock_mhz ... clock rate in MHz [12,24,36,48,60]
// div       ... divider value [cclk_4,cclk_2,cclk]
void init(int clock_mhz, int div) {
    mam_init(clock_mhz);
    set_vpbdiv(div);
    pll_init(clock_mhz);
}
```

C.2 Register stanj

Register stanj *CPSR* (angl. Current Program Status Register) je 32-bitni register, pri čemer je večina bitov neizkoriščenih. Prikazan je na sliki C.3.



Slika C.3: Zgradba registra stanj *CPSR*

Biti na zgornjih štirih mestih so klasične Negative, Zero, Carry in oVerflow zastavice, ki se postavljajo glede na rezultat izvedenega ukaza. Zastavica N je enaka najvišjemu bitu (bit31) rezultata in pomeni njegov predznak v primeru, da ga interpretiramo kot število, zapisano v dvojiškem komplementu. Zastavica Z se postavi, kadar je rezultat enak nič. Zastavica C se postavi ob prenosu pri seštevanju, ali kadar ne pride do izposoje pri odštevanju. Pri seštevanju to pomeni, da je rezultat večji od obsega nepredznačenih 32-bitnih števil (rezultat > `0xffffffff`). Oziroma pri odštevanju, da je rezultat znotraj obsega nepredzna-

čenih 32-bitnih števil, kar pravzaprav pomeni, da je večji ali enak nič (rezultat $\geq$ `0x00000000`). Poleg njene osnovne vloge prenosa in izposoje se zastavica C uporablja tudi pri premikih. Zastavica V se postavi ob prekoračitvi obsega predznačenih 32-bitnih števil. Pri seštevanju to pomeni, da je rezultat večji od `0x7fffffff`, pri odštevanju pa manjši od `-0x80000000`.

M3	M2	M1	M0	način delovanja
0	0	0	0	user
0	0	0	1	*FIQ*
0	0	1	0	*IRQ*
0	0	1	1	supervisor
0	1	1	1	abort
1	0	1	1	undefined
1	1	1	1	system

Tabela C.1: Načini delovanja procesnega jedra

Najnižji bajt registra stanj *CPSR* vsebuje zastavice *I*, *F* in *T*, ter 4-bitno kodo, ki označuje način delovanja procesnega jedra. Več o načinih delovanja je povedano v nadaljevanju v dodatku C.3. Tabela C.1 podaja 4-bitne kode različnih načinov delovanja.

Poleg registra stanj *CPSR* je vgrajenih tudi več pomožnih registrov *SPSR* (angl. Saved Program Status Register). Vsak način delovanja, v katerega se procesno jedro postavi zaradi prekinitve, ima en tak register *SPSR*. Vanj se shrani vsebina registra stanj *CPSR* pred prekinitvijo, tako da je mogoče ob njenem koncu zopet vzpostaviti prvotno stanje.

Zastavici *I* in *F* onemogočata prekinitve *IRQ* in *FIQ*. Postavljena zastavica *I* pomeni, da so onemogočene prekinitve *IRQ*, oziroma postavljena zastavica *F* onemogoča prekinitve *FIQ*. Zastavica *T* podaja nabor ukazov s katerim procesno

jedro trenutno deluje. Postavljena zastavica *T* označuje okleščeni nabor ukazov *Thumb*, umaknjena pa normalni nabor *ARM*.

cond	pogoj	pomen
eq	Z=1	enak
ne	Z=0	neenak
cs	C=1	prenos
hs	C=1	večji ali enak (nepredznačeno)
cc	C=0	ni prenosa
lo	C=0	manjši (nepredznačeno)
mi	N=1	negativno
pl	N=0	pozitivno
vs	V=1	preliv
vc	V=0	ni preliva
hi	C=1 in Z=0	večji (nepredznačeno)
ls	C=0 ali Z=1	manjši ali enak (nepredznačeno)
ge	N=V	večji ali enak (predznačeno)
lt	N≠V	manjši (predznačeno)
gt	N=V in Z=0	večji (predznačeno)
le	N≠V in Z=1	manjši ali enak (predznačeno)
al	izpolnjen	vedno

Tabela C.2: Pogojne kode

Večina zbirniških ukazov se lahko izvrši pogojno. To pomeni, da se ukaz izvrši le, če je izpolnjen določen pogoj v registru stanj *CPSR*, oziroma zastavice N, Z, C in V imajo želene vrednosti. V nasprotnem primeru se ukaz ignorira. Pogojno izvajanje je podano tako, da je ukazu dodan pogoj *cond* (glej dodatek C.5). Pogoji so zbrani v tabeli C.2. Če pogoj ukazu ni dodan, se ukaz izvrši vedno, oziroma učinek je enak, kot če bi bil dodan pogoj vedno (al).

C.3 Načini delovanja in registri

Procesiranje podatkov na arhitekturi ARM se vedno dogaja v delovnih registrih. To pomeni, da je potrebno podatek pred obdelavo naložiti v enega izmed njih, ter ga po njej zopet shraniti nazaj v pomnilnik. V ta namen je na voljo 13

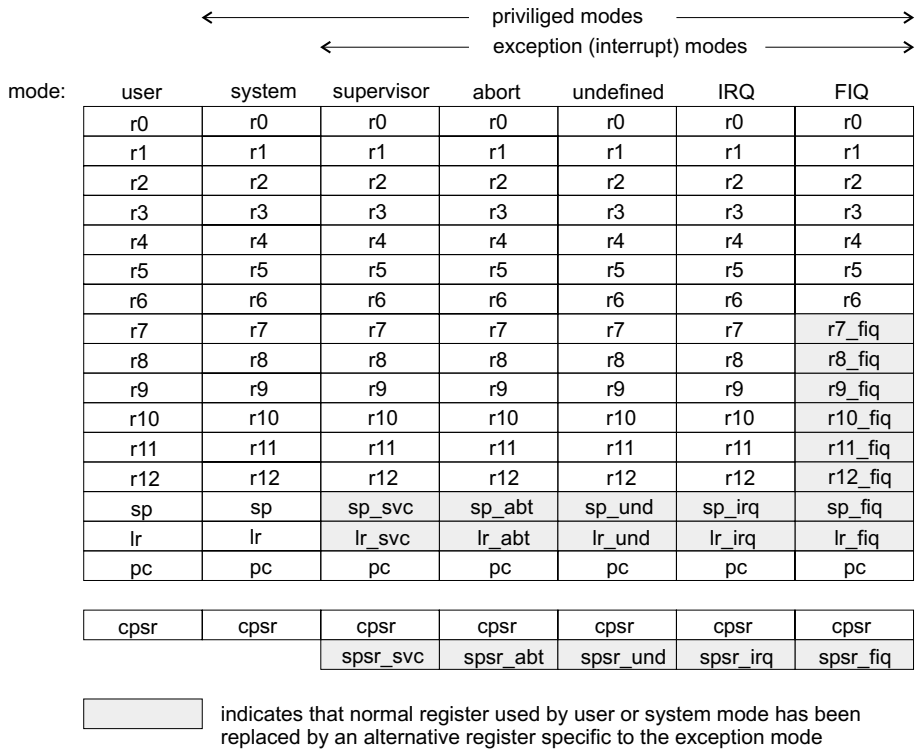
povsem splošnih 32-bitnih delovnih registrov označenih od *r0* do *r12*. Temu so dodani še trije specialni registri *r13*, *r14* in *r15*.

Register *r13* po dogovoru predstavlja kazalec sklada. Namesto oznake *r13* se navadno uporablja oznaka *sp* (angl. Stack Pointer). Ob pravilni uporabi je klasičen predstavnik svoje vrste in kaže na vrh sklada. Sklad lahko v naslovnem prostoru raste proti višjim ali nižjim naslovom, kazalec sklada pa lahko kaže na prvo prosto ali zadnje zasedeno mesto v njem. Vendar mora za sklad poskrbeti programska koda sama. Vsa odlaganja na sklad in branja iz njega se izvedejo eksplicitno, navadno z ukazoma *stm* in *ldm* (glej dodatek C.5), kar pomeni, da bi bil kot kazalec sklada lahko uporabljen pravzaprav katerikoli izmed delovnih registrov. Oziroma, če programska koda sklada ne uporablja, potem je *r13* navaden delovni register. Vzrok, da se kot kazalec sklada navadno uporablja ravno *r13*, tiči v tem, da je ta register v vsaki prekinitvi (načinu delovanja) podvojen, kar bo podrobneje razloženo v nadaljevanju. To z drugimi besedami pomeni, da ima vsaka prekinitev svoj sklad.

Sledi register *r14* ali povezovalni register. Namesto oznake *r14* se navadno uporablja oznaka *lr* (angl. Link Register). Ob klicu podprograma se vanj shrani naslov vrnitve. To omogoča hitro vračanje iz podprogramov na prvem nivoju, to je podprogramov, ki ne kličejo drugih podprogramov. V nasprotnem primeru je potrebno pred klicem podprograma na drugem ali še nižjem nivoju povezovalni register shraniti na sklad. Kadar register *lr* ni uporabljen kot povezovalni register, ga je mogoče uporabljati kot navaden delovni register.

Register *r15* je programski števnik in je navadno označen z oznako *pc* (angl. Program Counter). Zbirniški ukazi delujejo na programskem števniku povsem

enako, kot na drugih delovnih registrih. Vendar ukazi, ki spremenijo vsebino registra *pc*, pravzaprav predstavljajo skoke.



Slika C.4: Organizacija registrov (vsak stolpec predstavlja nabor registrov dostopnih v pripadajočem načinu delovanja)

Poleg opisanih registrov je tu še register stanj *CPSR* (glej dodatek C.2), ki določa način delovanja procesnega jedra (tabela C.1), in njegova senca *SPSR*. Programska koda navadno teče v uporabniškem (user) načinu. Način delovanja procesnega jedra se spremeni na primer ob prekinitvi. Vsak način delovanja, ki je posledica prekinitve, ima svoja specialna registra *sp* in *lr*, način *FIQ* pa poleg tega še delovne registre od *r8* do *r12*, kar omogoča hiter zagon prekinitve *FIQ*,

saj omenjenih registrov ni potrebno shranjevati na sklad. Registri dostopni v različnih načinih delovanja so prikazani na sliki C.4.

Centralno procesno jedro ARM7TDMI-S [3] pozna sedem vrst prekinitev, ki jih lahko povzročijo notranji ali zunanji dogodki. In sicer:

- prekinitev ob zagonu (angl. reset),
- prekinitev ob neznanem ukazu za koprocesor, ko se noben izmed morebitnih koprocesorjev ne odzove, da lahko ukaz izvrši (angl. undefined instruction),
- programsko zahtevana prekinitev z ukazom *swi* (angl. SoftWare Interrupt *SWI*),
- prekinitev ob poskusu branja ukaza iz neveljavnega naslova (angl. prefetch abort),
- prekinitev ob poskusu branja ali pisanja podatka na neveljaven naslov (angl. data abort),
- klasična prekinitev (angl. Interrupt ReQuest *IRQ*), ter
- prednostna klasična prekinitev (angl. Fast Interrupt reQuest *FIQ*).

prekinitev	način	naslov
reset	supervisor	`0x00000000`
undefined instruction	undefined	`0x00000004`
SWI	supervisor	`0x00000008`
prefetch abort	abort	`0x0000000c`
data abort	abort	`0x00000010`
IRQ	IRQ	`0x00000018`
FIQ	FIQ	`0x0000001c`

Tabela C.3: Prekinitve, pripadajoči načini delovanja, ter prekinitveni naslovi

Ob nastopu katerekoli prekinitve, se centralno procesno jedro postavi v vnaprej določen način delovanja. V programski števnik *pc* se prepiše pripadajoč prekinitveni naslov, od koder se nadaljuje izvajanje programske kode. Na tem naslovu se nahaja prvi ukaz prekinitvenega podprograma. Navadno je to skok

na mesto, kjer je njegovo jedro. Načini delovanja in prekinitveni naslovi, ki pripadajo posameznim prekinitvam, so podani v tabeli C.3. Zaradi spremembe načina delovanja, se ob prekinitvi pravzaprav spremeni nabor delovnih registrov (slika C.4).

Sistemski (system) način delovanja ne pripada nobeni izmed prekinitev. Delovni registri, dostopni v tem načinu, so enaki kot v uporabniškem (user) načinu. Način delovanja se zamenja ob prekinitvi, mogoče pa ga je spreminjati tudi ročno z zapisom določene kombinacije bitov v register stanj (glej sliko C.3 in tabelo C.1). Vendar je zapisovanje v najnižji bajt registra stanj možno le iz privilegiranih načinov delovanja. Tako v uporabniškem načinu ni mogoče zamenjati načina delovanja z ukazom, kakor tudi ni mogoče prepovedati prekinitev. Sistemski način delovanja se od uporabniškega razlikuje le po tem, da je privilegiran. Navadno uporablja za poganjanje programske kode operacijskega sistema.

Poleg opisane spremembe načina delovanja, ter nadaljevanja s prvim ukazom prekinitvenega podprograma, se ob katerikoli prekinitvi umakne zastavica *T* in postavi zastavica *I*. Pri prekinitvah reset in *FIQ* se poleg zastavice *I* postavi tudi zastavica *F*. Umik zastavice *T* pomeni, da se prekinitveni podprogram prične izvrševati z normalnim naborom ukazov *ARM*. Postavitev zastavic *I*, oziroma *F*, pa onemogoči prekinitev *IRQ*, oziroma *FIQ*, med izvajanjem samega prekinitvenega podprograma. To z drugimi besedami pomeni, da je gnezdenje prekinitev *IRQ* in *FIQ* onemogočeno, razen v primeru, ko prekinitvena koda umakne pripadajočo zastavico.

prekinitev	vrnitev
reset	–
undefined instruction	`movs pc, lr`
SWI	`movs pc, lr`
prefetch abort	`subs pc, lr, #0x04`
data abort	`subs pc, lr, #0x04` ali `subs pc, lr, #0x08`
IRQ	`subs pc, lr, #0x04`
FIQ	`subs pc, lr, #0x04`

Tabela C.4: Ukazi za zaključek prekinitev

Pred pričetkom izvajanja prekinitvene kode se trenutna vsebina registra stanj *CPSR* shrani v register *SPSR* dostopen v načinu delovanja klicane prekinitve.

V povezovalni register *lr* se, podobno kot pri klicu podprogramov, shrani naslov vrnitve, to je naslov ukaza, ki sledi ravnokar izvrševanemu ukazu. To velja za vse prekinitve, razen za prekinitev reset. Tako je mogoče ob koncu prekinitvene kode zopet vzpostaviti prvotno stanje, ter se vrniti na mesto, kjer se je prekinitev zgodila. Povezovalni register *lr* je potrebno prepisati v programski števnik *pc*, vsebino registra *SPSR* pa v register stanj *CPSR*. Ob slednjem se zamenja način delovanja procesnega jedra v način pred prekinitvijo. Vendar je kopiranje povezovalnega registra *lr* v programski števnik *pc* pravilno le v primeru, da se trenutni ukaz, med katerim je bila prekinitev zahtevana, izvrši do konca. To je res ob prekinitvah undefined instruction in *SWI*. Ostale prekinitve trenutni ukaz odložijo in ga ne dokončajo. Zato se mora le-ta izvršiti ob vrnitvi, kar pomeni, da je potrebno povezovalnemu registru *lr* odšteti štiri. Poseben primer predstavlja prekinitev data abort, ki prekine naslednji ukaz po ukazu, zaradi katerega se prekinitev zgodi (poskus branja ali pisanja podatka na neveljaven naslov). V povezovalni register *lr* se v tem primeru shrani naslov, ki je od neveljavnega ukaza oddaljen osem bajtov (dva ukaza naprej). V primeru, da prekinitvena koda odpravi vzrok prekinitve in naj se ukaz, zaradi katerega se je prekinitev zgodila, izvrši še enkrat, je potrebno povezovalnemu registru odšteti dva ukaza, torej osem. Tabela C.4 podaja ukaze, s katerimi se zaključijo posamezne prekinitve. Vsi ukazi poskrbijo tudi za register stanj *CPSR*. Za podrobnejšo razlago posameznih ukazov glej dodatek C.5.

Za morebitno gnezdenje prekinitev *IRQ*, oziroma *FIQ*, mora poskrbeti prekinitvena koda sama. In sicer mora na svoj sklad odložiti povezovalni register *lr* in register *SPSR*, ki ju na koncu od tam zopet pobere. Med delom s skladom mora biti gnezdenje prekinitev onemogočeno. V razvrščevalniku opravil *sch_int* na strani 48 je gnezdenje omogočeno, čeprav povezovalni register *lr* in register *SPSR* nista shranjena na sklad. To je možno zaradi narave razvrščevalnika. Če

se zgodi nova prekinitev, preden se trenutna konča, se razvrščevalnik ujame v neskončno zanko. Gnezdena prekinitev se nikdar ne zaključi.

prioriteta	prekinitev
najvišja	reset
	data abort
	FIQ
	IRQ
	prefetch abort
najnižja	undefined instruction and *SWI*

Tabela C.5: Razpored prekinitev po prioriteti

V primeru, da se hkrati pojavi več različnih vrst prekinitev, se najprej izvede tista z najvišjo prioriteto, nato naslednja z nižjo prioriteto, in tako dalje. Tabela C.5 podaja prioritetna razmerja med posameznimi prekinitvami.

Sledi primer z uvodnimi ukazi na prekinitvenih naslovih. Ukazi se nahajajo od naslova `0x00000000` dalje, za kar poskrbi prevajalnik. Prva ukaza prekinitev reset in *IRQ* izvedeta brezpogojna skoka na oznaki *reset*, oziroma *irq* (glej tudi kodo na straneh 179 in 109). Ostale prekinitve se v podanem primeru

pravzaprav ne smejo zgoditi. Izvajanje programa se v tem primeru ustavi, mikrokrmilnik pa pristane v mrtvi zanki.

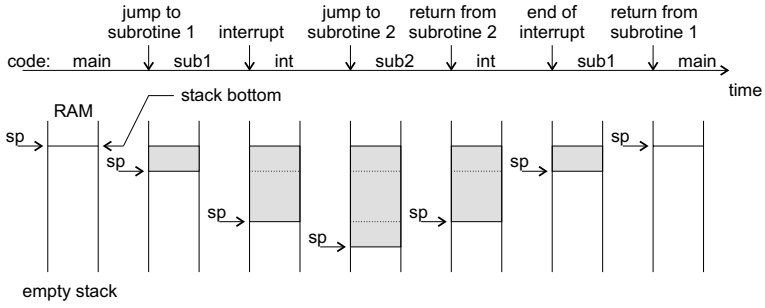
```
/* Parameter */
          .equ    user_key, 0x99600fa0
/* Global symbol */
          .global reset
          .global irq

          .code   32

/* Program code */
          .text
/* Exception or interrupt vector table */
          b       reset
und:      b       und
swi:      b       swi
pref_abt: b       pref_abt
data_abt: b       data_abt
          .long   user_key
          ldr     pc, =irq
fiq:      b       fiq
```

Skoka na oznaki *reset* in *irq* sta izvedena različno. In sicer je pri prvem uporabljen *b* ukaz, ki lahko izvrši le skoke dolge največ ±32MB. Ker se oznaka *reset* nahaja v pomnilniku flash, je to dovolj (glej dodatek C.1). Oznaka *irq* se nahaja v pomnilniku RAM, torej je približno 1GB oddaljena od mesta skoka. Zato je uporabljen ukaz *ldr*, ki v programski števnik *pc* naloži 32-bitni naslov oznake *irq*, ter tako opravi poljubno dolg skok. Zanimiva je še 32-bitna konstanta *user_key* na naslovu `0x00000014`. Mikrokrmilnik ob zagonu preveri veljavnost kode. Koda je veljavna, če je vsota prvih osmih 32-bitnih števil enaka nič, brez upoštevanja prenosa. Oziroma vsota uvodnih ukazov in konstante *user_key*

mora biti enaka nič. V nasprotnem primeru mikrokrmilnik z izvajanjem kode ne bo pričel.

C.4 Delo s skladom

V splošnem je sklad medpomnilnik LIFO (angl. Last In First Out), ki se nahaja v pomnilniku RAM. Namenjen je začasnemu shranjevanju podatkov ob klicih podprogramov in prekinitvah. Pred pričetkom izvajanja podprograma se na sklad odloži naslov vrnitve, ob prekinitvi pa vsebina vseh registrov centralnega procesnega jedra. Tako se je mogoče po koncu podprograma vrniti nazaj, po koncu prekinitve pa centralno procesno jedro ponovno postaviti v prejšnje stanje.



Slika C.5: Delovanje sklada

Kazalec sklada je register, katerega vsebina je naslov prvega prostega, ali zadnjega zasedenega prostora v skladu. Vedno kaže na vrh sklada. Njegova

vrednost se spremeni ob vsakem odlaganju na sklad, ali jemanju iz njega. Odlaganja morajo biti uravnotežena z jemanji, drugače se podatki v skladu zamaknejo, kar posledično vodi do precej nepredvidljivega obnašanja. Po zaključku podprograma se ob zamaknjenem skladu mikrokrmilnik ne vrne nazaj, ampak nadaljuje z delom na naslovu, ki ga je iz sklada dobil, kakršenkoli že ta naslov je. Prav tako se ob zamaknjenem skladu po koncu prekinitve centralno procesno jedro ne postavi v prvotno stanje. Mikrokrmilnik z delom ne nadaljuje enako, kot če prekinitve ne bi bilo. Princip delovanja sklada ponazarja slika C.5.

Delovanje sklada je mogoče primerjati z oklepaji v matematičnih izrazih. Za primer naj oglati oklepaj [pomeni klic podprograma, zaklepaj] pa vrnitev iz njega. Zavita oklepaja { in } naj označujeta začetek in konec prekinitve, okrogla oklepaja (in) pa ročno odlaganje in jemanje s sklada. Naj bo *intr* prekinitev, ki na sklad začasno odloži nek podatek, ter pokliče še nek podprogram. Po analogiji z matematičnimi izrazi bi bila prekinitev *intr* lahko predstavljena kot:

$$intr = \ldots\{\ldots(\ldots[\ldots]\ldots)\ldots\}\ldots$$

Če je dovoljeno gnezdenje prekinitev, in prekinitev *intr* prekine samo sebe, bi dogajanju s stališča sklada ustrezal na primer naslednji izraz:

$$\ldots\underbrace{\{\ldots(\ldots}\overbrace{\{\ldots(\ldots[\ldots]\ldots)\ldots\}}^{intr}\underbrace{\ldots[\ldots]\ldots)\ldots\}}\ldots$$

Podprogram *subr*, ki na sklad odloži nek podatek, se konča, nato pa podatek s sklada vzame glavni program, je primer nepravilne uporabe sklada. Podatki v

skladu se zamaknejo. Okoliščine predstavlja sintaktično nepravilen matematični izraz:

$$subr = \ldots [\ldots (\ldots] \ldots) \ldots$$

Za pisanje in jemanje s sklada veljajo enaka pravila, kot za pisanje oklepajev v matematičnih izrazih.

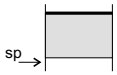
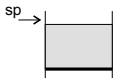
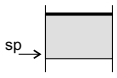
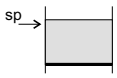
	odlaganje na sklad	jemanje s sklada
	`stmed sp!, {regs}`	`ldmed sp!, {regs}`
	`stmea sp!, {regs}`	`ldmea sp!, {regs}`
	`stmfd sp!, {regs}`	`ldmfd sp!, {regs}`
	`stmfa sp!, {regs}`	`ldmfa sp!, {regs}`

Tabela C.6: Ukazi za delo s skladom

Mikrokrmilnik LPC2138, podobno kot vsi mikrokrmilniki in mikroprocesorji arhitekture ARM, s skladom ne dela avtomatsko, kot je to navada pri mnogih

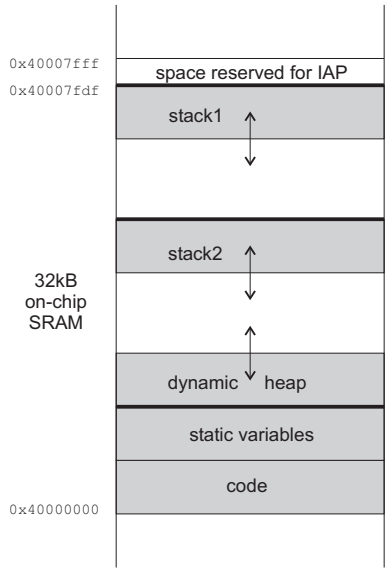
drugih. Ob prekinitvi se vsebina delovnih registrov na sklad ne shrani samodejno, kakor tudi ne naslov vrnitve ob skoku v podprogram. V prvem primeru se zamenja način delovanja, v drugem pa se naslov vrnitve shrani v povezovalni register *lr*, ki predstavlja nekakšen mini sklad (glej dodatek C.3). Za pravi sklad, oziroma medpomnilnik LIFO, ki je pri gnezdenju prekinitev, ali klicanju podprogramov na več kot enem nivoju, nujen, mora eksplicitno poskrbeti programska koda. Navadno se za odlaganje na sklad in jemanje z njega uporabljata ukaza *stm* in *ldm* (glej dodatek C.5). Ker delo s skladom ni avtomatsko, ga lahko načrtovalec programske opreme organizira po svoje. Kazalec sklada lahko kaže na prvo prosto, ali zadnje zasedeno mesto, sklad pa lahko narašča proti višjim, ali nižjim naslovom. V tabeli C.6 so zbrani ukazi za odlaganje na sklad in jemanje z njega v vseh štirih primerih.

Primer programske kode, ki postavlja začetne vrednosti kazalcev sklada, se nahaja na strani 179. Po kopiranju programske kode v pomnilnik RAM in zapisih v registra nadzornika prekinitev, se s spremembo bitov v registru *CPRS* spremeni način delovanja iz supervisor v IRQ, pri čemer prekinitve ostanejo onemogočene (glej dodatek C.2). Nato je postavljena začetna vrednost kazalca sklada za ta način delovanja. Po preklopu v uporabniški način delovanja sledi pripadajoča postavitev kazalca sklada še v tem načinu. S tem se zagonska koda mikrokrmilnika konča.

Čeprav ima vsak privilegiran način svoj kazalec, sta začetni vrednosti postavljeni le za dva sklada, to je sklada v uporabniškem (sistemskem) in načinu delovanja IRQ. Vendar bosta v konkretnem primeru, glede na prekinitvene naslove na strani 193, to edina načina delovanja, saj se v vseh ostalih prekinitvah

mikrokrmilnik znajde v neskončni zanki. Sicer ima v splošnem vsak privilegiran način delovanja svoj sklad.

Primer uporabe sklada demonstrira podprogram *init* na strani 179. Na sklad shrani povezovalni register *lr*, ki vsebuje naslov vrnitve. To mora narediti, da lahko kliče svoje podprograme *mam_ init*, *vpbdiv_ init* in *pll_ init*.



Slika C.6: Zasedenost pomnilnika RAM

Poleg skladov se v pomnilniku RAM nahaja programska koda (glej dodatek C.1), ter statične in dinamične spremenljivke. Prostor, ki ga zavzema programska koda in statične spremenljivke, je konstanten. Dinamičnim spremenljivkam je prostor dodeljen med delovanjem, zato njegova velikost ni vnaprej znana. Značilna je uporaba funkcij *malloc()* in *free()* v programskem jeziku C [14] in [15]. Prav tako ni vnaprej znana velikost skladov. Razmere v pomnilniku RAM ponazarja slika C.6.

Teoretično je možno, da bi velikost dinamičnih spremenljivk, ali katerega izmed skladov, narasla preko nezasedenega prostora. Sklad bi začel svoje podatke pisati preko nekih drugih podatkov, ki bi bili tako izgubljeni, poplavljeni. Posledica bi bila nepredvidljivo in nestabilno obnašanje mikrokrmilniškega sistema,

kar je nedopustno. Zato morajo biti nezasedena področja dovolj velika, da do poplavljanja nikdar ne pride. V primeru na sliki C.6 je sklad IRQ postavljen 32 bajtov pred koncem pomnilnika RAM (`0x40007fdf`). Zadnjih 32 bajtov uporablja prednaloženi program za pisanje v pomnilnik flash (glej dodatek B.13). Uporabniški sklad je postavljen 8kB nižje (`0x40005fff`). Tako sta sklada daleč narazen, in prav tako daleč od programske kode in statičnih spremenljivk na začetku pomnilnika RAM (`0x40000000`). Za to, da do poplavljanja ne prihaja, lahko skrbi tudi operacijski sistem.

C.5 Nabor zbirniških ukazov za mikrokrmilnik Philips LPC2138

Podan je zgoščen pregled zbirniških ukazov normalnega 32-bitnega nabora *ARM* za arhitekturo ARMv4T, na kateri temelji procesno jedro ARM7TDMI Philipsovega mikrokrmilnika LPC2138. Popolno razlago je mogoče najti v [6]. Zaviti oklepaji `{}` pomenijo neobvezno polje, ki je lahko izpuščeno. Trikotni oklepaji `<>` pomenijo uporabniško polje, ki mora biti prisotno, in določa način naslavljanja. Oklepaji niso del sintakse. V nekaterih primerih je na voljo več možnosti, od katerih je potrebno navesti natanko eno. Posamezne alternative so med seboj ločene z navpično črto `|`, ki zopet ni del sintakse. V simboličnih opisih izraz med poševnicama // pomeni kazalec, oziroma naslov. Tako */naslov/* podaja 32-bitno konstanto, ki se nahaja v štirih bajtih od naslova *naslov* dalje. Ukazi so urejeni po namenu. Na koncu so dodani seznami, ki dodatno razlagajo posamezna uporabljena polja in načine naslavljanja.

Ukazi za premikanje (angl. move)

$\underline{oprnd2} \rightarrow rd$	`mov{cond}{s} rd, <oprnd2>`
$\overline{oprnd2} \rightarrow rd$	`mvn{cond}{s} rd, <oprnd2>`
$psr \rightarrow rd$	`mrs{cond} rd, psr`
$\#32bit \rightarrow psr, rm \rightarrow psr$[1]	`msr{cond} psr{fields}, #32bit | rm`

[1] Najnižji bajt registra stanja *CPSR* je mogoče spreminjati le v privilegiranem načinu delovanja.

Aritmetični ukazi (angl. arithmetic)

$rn + oprnd2 \rightarrow rd$	`add{cond}{s} rd, rn, <oprnd2>`
$rn + oprnd2 + C \rightarrow rd$	`adc{cond}{s} rd, rn, <oprnd2>`
$rn - oprnd2 \rightarrow rd$	`sub{cond}{s} rd, rn, <oprnd2>`
$rn - oprnd2 - \overline{C} \rightarrow rd$	`sbc{cond}{s} rd, rn, <oprnd2>`
$oprnd2 - rn \rightarrow rd$	`rsb{cond}{s} rd, rn, <oprnd2>`
$oprnd2 - rn - \overline{C} \rightarrow rd$	`rsc{cond}{s} rd, rn, <oprnd2>`
$(rm \times rs)[31:0] \rightarrow rd$	`mul{cond}{s} rd, rm, rs`
$(rm \times rs + rn)[31:0] \rightarrow rd$	`mla{cond}{s} rd, rm, rs, rn`
$(rm \times rs)[31:0] \rightarrow rd$, $(rm \times rs)[63,32] \rightarrow rn$	`umull{cond}{s} rd, rn, rm, rs`
$(rm \times rs)[31:0] + rd \rightarrow rd$, $(rm \times rs)[63,32] + rn +$ carry from $((rm \times rs)[31:0] + rd) \rightarrow rn$	`umlal{cond}{s} rd, rn, rm, rs`
$(rm \times rs)[31:0] \rightarrow rd^2$, $(rm \times rs)[63,32] \rightarrow rn^2$	`smull{cond}{s} rd, rn, rm, rs`
$(rm \times rs)[31:0] + rd \rightarrow rd^2$, $(rm \times rs)[63,32] + rn +$ carry from $((rm \times rs)[31:0] + rd) \rightarrow rn^2$	`smlal{cond}{s} rd, rn, rm, rs`
$rd - oprnd2$	`cmp{cond} rd, <oprnd2>`
$rd + oprnd2$	`cmn{cond} rd, <oprnd2>`

Logični ukazi (angl. logical)

$rn \,\&\, oprnd2$	`tst{cond} rn, <oprnd2>`
$rn \oplus oprnd2$	`teq{cond} rn, <oprnd2>`
$rn \,\&\, oprnd2 \rightarrow rd$	`and{cond}{s} rd, rn, <oprnd2>`
$rn \oplus oprnd2 \rightarrow rd$	`eor{cond}{s} rd, rn, <oprnd2>`
$rn \mid oprnd2 \rightarrow rd$	`orr{cond}{s} rd, rn, <oprnd2>`
$rn \,\&\, \overline{oprnd2} \rightarrow rd$	`bic{cond}{s} rd, rn, <oprnd2>`

Vejitvena ukaza (angl. branch)

label address $\rightarrow pc$	`b{l}{cond} label`
$rn[0] \rightarrow T, rn \,\&$ `0xfffffffe` $\rightarrow pc$	`bx{cond} rn`

[2] Aritmetične operacije se izvajajo na predznačenih številih zapisanih v dvojiškem komplementu.

Bralni in pisalni ukazi (angl. load and store)

/addmod2/ → *rd*	`ldr{cond}{b}{t} rd, <addmod2>`
/addmod3/ → *rd*	`ldr{cond}s`[3]`b rd, <addmod3>`
/addmod3/ → *rd*	`ldr{cond}{s`[3]`}h rd, <addmod3>`
/rn[4]*/* → *regs*	`ldm{cond}<addmod4> rn{!}, <regs>{^}`[5]
rd → */addmod2/*	`str{cond}{b}{t} rd, <addmod2>`
rd → */addmod3/*	`str{cond}h rd, <addmod3>`
regs → */rn*[4]*/*	`stm{cond}<addmod4> rn{!}, <regs>{^}`[6]

Menjave in programske prekinitve (angl. swap, software interrupt)

/rn/ → *rd*, *rm* → */rn/*	`swp{cond}{b} rd, rm, [rn]`
next instruction address → *lr_svc*,	`swi{cond} #24bit`[7]
CPSR → *SPSR_svc*,	
`0x93` | *CPSR*[6] → *CPSR*[7 : 0],	
`0x00000008` → *pc*	

[3] Znak `s` v teh dveh ukazih pomeni razširitev predznaka (najvišjega bita 8-bitnega, oziroma 16-bitnega števila) na zgornje tri, oziroma dva bajta registra *rd*.

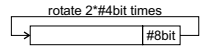
[4] Register *rn* je prvi, zadnji ... (glej način naslavnjanja *addmod4*) naslov, od koder/kamor so prebrane/zapisane vrednosti registrov *regs*.

[5] Znak ^ v ukazu `ldm` pomeni naslednje: če v seznamu registrov *regs* ni programskega števnika *pc*, potem seznam označuje registre v uporabniškem načinu delovanja. Če je med naštetimi registri tudi *pc*, potem se seznam *regs* nanaša na registre v trenutnem načinu delovanja, poleg tega se v register stanj *CPSR* prepiše vsebina registra *SPSR* (*SPSR* → *CPSR*).

[6] Znak ^ v ukazu `stm` pomeni, da se seznam registrov *regs* nanaša na registre v uporabniškem načinu delovanja. Ob uporabi znaka ^ shranjevanje končnega naslova nazaj v register *rn* (znak !) ni mogoče.

[7] Konstanta služi kot koda, katera programska prekinitev je bila zahtevana.

Polja v ukazih

`rd`, `rm`, `rn`, `rs`	delovni register *r0* ... *r15*, ali *sp*, *lr* in *pc*
`psr`	register stanj *CPSR*, ali *SPSR*
`label`	simbolični naslov
`!`	končni naslov se shrani nazaj v delovni register *rn* (*address* → *rn*).
`cond`	pogoj za izvršitev ukaza (glej tabelo C.2)
`oprnd2`	drugi operand (stran 203)
`fields`	skupine bitov v registru stanj (stran 202)
`l`	shrani naslov naslednjega ukaza v register *lr*
`s`	postavi zastavice v registru *CPSR* glede na rezultat
`b`	ukaz se izvrši le nad najnižjim bajtom
`h`	ukaz se izvrši le nad najnižjima dvema bajtoma
`t`	ne glede na trenutni način delovanja dostopa do pomnilnika v uporabniškem načinu, uporablja lahko le zadnje tri načine naslavljanja 2 (stran 203)
`addmod2`	način naslavljanja 2 (stran 203)
`addmod3`	način naslavljanja 3 (stran 204)
`addmod4`	način naslavljanja 4 (stran 204)
`#32bit`	32-bitna konstanta[8] 
`regs`	seznam delovnih registrov v zavitih oklepajih (primer: `{r0-r3,r6,lr}`)

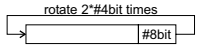
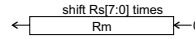
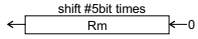
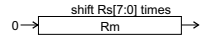
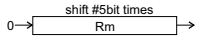
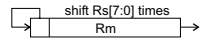
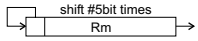
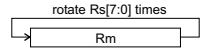
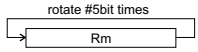
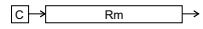
Skupine bitov *fields* v registru stanj

Polje *fields* podaja, na katere bite v registru stanj *CPSR*, oziroma *SPSR*, se ukaz nanaša. Sestavljeno je iz podčrtaja `_` in poljubne kombinacije spodnjih štirih črk (primer: `_cf`, `_csf` ...):

`c`	kontrolni biti *PSR*[0:7] (*I*, *F*, *T* in *M0* do *M3*)
`x`	*PSR*[8:15]
`s`	*PSR*[16:23]
`f`	zastavice *PSR*[24:31] (*N*, *Z*, *C* in *V*)

[8] 8-bitna konstanta sodo krat (dva krat 4-bitna konstanta) rotirana v desno.

Drugi operand *oprnd2*

`#32bit`[8]		`rm, lsl rs`[9]	
`rm, lsl #5bit`[9]		`rm, lsr rs`[9]	
`rm, lsr #5bit`[9]		`rm, asr rs`[9]	
`rm, asr #5bit`[9]		`rm, ror rs`[9]	
`rm, ror #5bit`[9]		`rm, rrx`[9]	
`rm`	*oprnd2* = *rm*		

Način naslavljanja *addmod2*

[`rn`, # ± `12bit`]{!}[10]	*address* = *rn* ± *#12bit*
[`rn`, ±`rm`]{!}[10]	*address* = *rn* ± *rm*
[`rn`, ±`rm`, `shift`[11]]{!}[10]	*address* = *rn* ± shifted *rm*
[`rn`], # ± `12bit`	*address* = *rn*, *rn* ± *#12bit* → *rn*
[`rn`], ±`rm`	*address* = *rn*, *rn* ± *rm* → *rn*
[`rn`], ±`rm`, `shift`[11]	*address* = *rn*, *rn* ± shifted *rm* → *rn*

[9] Vsebina delovnega registra *rm* ostane nespremenjena.

[10] Za pomen znaka `!` glej polja v ukazih stran 202.

[11] Oznaka *shift* je lahko `lsl #5bit`, `lsr #5bit`, `asr #5bit`, `ror #5bit`, ali `rrx`. Podaja premik delovnega registra *rm* enako kot pri drugem operandu (stran 203).

Način naslavljanja *addmod3*

$[\texttt{rn}, \# \pm \texttt{8bit}]\{!\}^{10}$	$address = rn \pm \#8bit$
$[\texttt{rn}, \pm\texttt{rm}]\{!\}^{10}$	$address = rn \pm rm$
$[\texttt{rn}], \# \pm \texttt{8bit}$	$address = rn, rn \pm \#8bit \rightarrow rn$
$[\texttt{rn}], \pm\texttt{rm}$	$address = rn, rn \pm rm \rightarrow rn$

Način naslavljanja *addmod4*

Način naslavljanja je podan s črkama v stolpcu pomnilnik (*ia*, *ib* ...), ki določata vrstni red dogodkov ob branju, oziroma pisanju. Pri delu s skladom (glej dodatek C.4) je potrebno pisanje na sklad uskladiti z branjem z njega (tabela C.6). Če podatke na sklad odložimo v načinu *db*, potem jih moramo z njega pobrati v načinu *ia*. Da bi bilo delo s skladom bolj intuitivno, so načini naslavljanja lahko podani tudi z ekvivalenti v drugem in tretjem stolpcu. Pisanje v načinu *fd* je ekvivalentno pisanju v načinu *db*. Če je bilo to pisanje na sklad, potem moramo brati z njega v načinu *ia*, čemur je ekvivalentno branje v načinu *fd*. Tako na sklad odlagamo in z njega jemljemo v načinu *fd*.

pomnilnik	sklad (branje)	sklad (pisanje)	
`ia`	`fd`	`ea`	beri/piši, nato povečaj naslov
`ib`	`ed`	`fa`	povečaj naslov, nato beri/piši
`da`	`fa`	`ed`	beri/piši, nato zmanjšaj naslov
`db`	`ea`	`fd`	zmanjšaj naslov, nato beri/piši

Literatura

[1] Arthur D. Friedman, *Fundamentals of Logic Design and Switching Theory*, Computer Science Press, Inc., New York, USA, 1986

[2] Ronald J. Tocci, Frank J. Ambrosio, *Microprocessors and Microcomputers: Hardware and Software*, Prentice Hall, Upper Saddle River, New Jersey, 2000

[3] *ARM7TDMI-S Technical Reference Manual*, ARM Limited, ARM DDI 0234A, 2001

[4] Dean Elsner, Jay Fenlason and friends, *Using as (The GNU Assembler)*, The Free Software Foundation Inc., January 1994

[5] *LPC213x User Manual*, Koninklijke Philips Electronics N. V., 24 June 2005

[6] *ARM Architecture Reference Manual*, ARM Limited, ARM DDI 0100E, 2000

[7] Steve Chamberlain, *Using ld (The GNU Linker)*, The Free Software Foundation Inc., January 1994

[8] Trevor Martin, *The Insider's Guide to the Philips ARM7-Based Microcontrollers (An Engineer's Introduction to the LPC2100 Series*, Hitex (UK) Ltd., 21 April 2005

[9] Stephen B. Furber, *ARM System-on-Chip Architecture*, Addison-Wesley Longman Publishing Co. Inc., Boston, Massachusetts, USA, 2000

[10] David Seal, *ARM Architecture Reference Manual*, Addison-Wesley Publishing Co., UK, 2000

[11] *ARM PrimeCell^{TM} Vectored Interrupt Controller*, ARM Limited, ARM DDI 0273A, 2002

[12] *http://www.s-arm.si*, Domača stran učnega razvojnega sistema Š-arm, Fakulteta za elektrotehniko, Ljubljana, 2006

[13] *HD44780U Dot Matrix Liquid Crystal Display Controller/Driver*, Hitachi, ADE-207-272(Z), 1998

[14] Franc Bratkovič, *Uvod v C*, Fakulteta za elektrotehniko, Ljubljana, 1998

[15] Herbert Schildt, *Teach Yourself C*, Osborne McGraw-Hill, Berkeley, 1997

[16] Richard M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, The Free Software Foundation Inc., 2003