UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

OSNOVE PROGRAMSKEGA JEZIKA C NA PRIMERU MIKROKRMILNIKA ARM7

**Zapiski predavanj za predmeta
Računalništvo 2 in
Računalništvo za elektrotehnike 2**

Iztok Fajfar

Ljubljana, 2006

Kazalo

1 Preden začnemo

1.1 Ali potrebujem kakšno predznanje?

Snov poletnega semestra, čemur so namenjeni tudi ti zapiski, obravnava programski jezik C na konkretnem primeru mikrokrmilnika ARM7. Da lahko zapiske študirate, se predpostavlja, da imate vsaj osnovno znanje jezika JavaScript, ki je bil snov zimskega semestra. Nekatere pomembne stvari bomo v teh zapiskih na hitro ponovili in predvsem izpostavili razlike med obema jezikoma.

1.2 Hardver

Ves semester se bo vrtel okrog učnega razvojnega sistema Š-ARM (Študentski ARM), ki je zgrajen na osnovi Philipsovega mikrokrmilnika ARM7. Sistem bomo uporabljali brez operacijskega sistema, kar bo za seboj potegnilo še nove razlike v filozofiji načrtovanja programov.

1.3 Alternativa hardveru

Tisti, ki si sistema v fizični obliki ne boste privoščili, je za vas brezplačno dostopen Microsoftov Visual C++ 5.0. V nekaterih primerih (predvsem na začetku) bodo vključene opombe, ki vam bodo govorile, kaj morate v kodi spremeniti, da jo boste lahko preskusili z okoljem Visual C++.

2 Prvi koraki

V tem poglavju bomo nekaj najosnovnejših pojmov, ki jih večino že poznamo iz jezika JavaScript, osvetlili s cejevskega vidika. Videli bomo, da je precej stvari zelo podobnih ali celo popolnoma enakih.

2.1 Funkcija `main()`

Ko se prvič lotimo pisanja programa v novem jeziku, se nam med prvimi zastavi vprašanje, kam postaviti kodo, da se bo pravilno izvršila. Ko smo pisali programe v jeziku JavaScript, smo kodo, za katero smo želeli, da se izvrši ob nalaganju strani, zapisali v element `body`.

V jeziku C pišemo programsko kodo v funkcijo `main()`. Kadarkoli bomo zagnali program, se bo ta začel izvajati na začetku te funkcije. Minimalni cejevski program (ki ne naredi absolutno ničesar) je:

```
int main(void)
{
  return 0;
}
```

Naš sistem Š-ARM za razliko od osebnega računalnika nima operacijskega sistema. To pomeni, da imamo mi, programerji, absoluten nadzor nad računalnikom in ne smemo nikoli dopustiti, da se program konča. Operacijski sistem je, poenostavljeno gledano, v resnici program, ki se nikoli

ne konča. Bolj natančno si bomo ta problem ogledali kasneje, zdaj funkciji `main()` dodamo samo ponavljalni stavek `while`, ki se nikoli ne konča:

```
int main(void)
{
  while (1);
  return 0;
}
```

Kaj pomeni enica v oklepaju, bomo izvedeli v poglavju Krmilni stavki.

2.2 Prevajanje in zagon kode

Za razliko od jezika JavaScript, ki je skriptni jezik, je C t.i. ‚pravi' programski jezik. To pomeni, da je potrebno napisano kodo najprej *prevesti* (compile). To storimo s posebnim programom, ki mu pravimo *prevajalnik* (compiler). Koda se prevede v zaporedje enic in ničel. To zaporedje digitalni računalnik razume kot niz ukazov, ki jih izvrši ob zagonu kode. Cejevski kodi, ki jo napišemo, pravimo tudi *izvorna koda* (source code), prevedeni kodi pa pravimo *izvršilna koda* (executable code).

Med postopkom prevajanja lahko pride do napak (compile-time error). Te napake niso zelo nadležne, saj nas nanje opozori prevajalnik. Te napake so predvsem sintaktične narave, na primer manjkajoč zaklepaj, podpičje ali napačno uporabljen operator.

Bolj zoprne so napake, ki se pojavijo med izvajanjem programa (run-time error). Te napake je pogosto zelo težko odkriti. Pogosto so to napake v logični zasnovi programa. Pri tem gre bodisi za drobne logične spodrsljaje bodisi za popolnoma napačno zastavljeno rešitev.

2.3 Stavek `#include`

Poleg standardnih programskih stavkov običajno v programih kličemo tudi vgrajene funkcije. Spomnimo se na primer funkcij `write()` ali `prompt()`, ki smo ju spoznali pri jeziku JavaScript. Tudi jezik C pozna vrsto uporabnih funkcij, ki pa jih ne moremo uporabiti kar takoj. Vključiti moramo ustrezno knjižnico. Katere knjižnice potrebujemo za določene funkcije, je opisano v poglavju Opisi uporabljenih funkcij, tu si oglejmo le, kako se knjižnico v program vključi. Uporabimo stavek `#include`, ki mu sledi ime datoteke, ki vsebuje knjižnico. Ime datoteke je v dvojnih navednicah, če je knjižnica v isti mapi kot naš projekt:

```
#include "io.h"
```

Če je knjižnica v posebni, sistemsko določeni mapi, pišemo ime knjižnice med znaka manjši (<) in večji (>):

```
#include <stdio.h>
```

2.4 Deklaracije spremenljivk in podatkovni tipi

Spomnimo se, da za hranjenje podatkov v programu potrebujemo spremenljivke, ki jih pred uporabo napovemo oziroma *deklariramo*. V jeziku JavaScript v trenutku deklaracije ni bilo potrebno vedeti, kakšne vrste podatek bomo v spremenljivki hranili. Še več - ena in ista spremenljivka je lahko med izvajanjem programa spreminjala svoj tip: enkrat je hranila številski podatek, potem na primer znakovni niz, pa mogoče Boolov logični podatek, itd.

V jeziku C se moramo že vnaprej odločiti, kakšnega tipa bo spremenljivka. Tip določimo ob deklaraciji in se ves čas izvajanja programa ne spremeni. Naslednja tabela povzema osnovne podatkovne tipe, ki jih pozna C.

Tip	**Opis**	**Opombe**
`char`	Znak ASCII oziroma 8-bitno predznačeno celo število.	Nekateri prevajalniki tega tipa ne poznajo. Nadomestijo ga z `unsigned char`.
`unsigned char`	Znak ASCII oziroma 8-bitno nepredznačeno celo število.	
`short`	16-bitno predznačeno celo število.	
`unsigned short`	16-bitno nepredznačeno celo število.	
`long`	32-bitno predznačeno celo število.	
`unsigned long`	32-bitno nepredznačeno celo število.	
`int`	32-bitno predznačeno celo število.	Redki prevajalniki obravnavajo ta tip kot 16-bitno celo število.
`unsigned int`	32-bitno nepredznačeno celo število.	Redki prevajalniki obravnavajo ta tip kot 16-bitno nepredznačeno celo število.
`float`	32-bitno realno število.	
`double`	64-bitno realno število.	

Spremenljivko deklariramo tako, da najprej napišemo ime tipa, potem ime spremenljivke in na koncu podpičje:

```
tip_spremenljivke ime_spremenljivke;
```

Na primer, spremenljivko tipa `char` z imenom `znak` deklariramo takole:

```
char znak;
```

Za razliko od jezika JavaScript neinicializirana spremenljivka (t.j. spremenljivka, ki ji nismo priredili nobene vrednosti) nima vrednosti `undefined`. Namesto tega dobi lokalna spremenljivka poljubno vrednost, ki ustreza podatkovnemu tipu spremenljivke. Na primer, deklarirana in neinicializirana spremenljivka realnega tipa ima poljubno realno vrednost. Neinicializirana globalna spremenljivka dobi vrednost 0.

Poglejmo si vsakega od zgoraj naštetih tipov bolj natančno.

2.4.1 Celoštevilski tip

Spremenljivka, ki jo deklariramo kot celoštevilsko, lahko hrani le vrednost, ki je celo število. Iz matematike vemo, da je celo število pač celo število. Od kod zdaj naenkrat toliko različnih tipov celih števil? Preden odgovorimo na to vprašanje, si oglejmo naslednji kratek program, ki izračuna faktorsko potenco oziroma fakulteto števila n. Fakulteta števila n je enaka produktu vseh naravnih števil do vključno n:

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$$

Tole pa je program:

```
int main(void)
{
  int fakulteta;
  int i;
  int n = 5;
  fakulteta = 1;
  for (i = 1; i <= n; i++)
  {
    fakulteta *= i;
  }
  while (1); //visual c++: odstrani to vrstico
  return 0;
}
```

Opomba: Kako se tak program prevede in zažene, si preberite v dokumentaciji, ki jo dobite na zgoščenki, ki je priložena sistemu Š-ARM. Tisti, ki boste uporabljali okolje Microsoft Visual C++, dobite navodila na spletni strani http://fides.fe.uni-lj.si/~lrnv/racunalnistvo2/msdnaa.html, kjer najdete tudi samo okolje Visual C++.

V trenutku, ko se zanka `for` v gornjem programu konča (takrat se program ujame v neskončni zanki `while (1)`), je vrednost spremenljivke *fakulteta* enaka 120. Spremenljivka n ima namreč vrednost 5, in 5! je enako 120. Vrednost spremenljivke lahko pogledamo tako, da v vrstico z neskončno zanko postavimo prekinitveno točko. Ko se program zaustavi ob tej točki, si lahko ogledamo vrednosti vseh spremenljivk. Mimogrede, vrednost spremenljivke i je v tem trenutku 6. Ali znate pojasniti, od kod ta vrednost?

Če boste program poskusili v okolju Visual C++, potem ne pozabite odstraniti neskončne zanke v predzadnji vrstici.

Do tu je šlo vse gladko. Zdaj smo že bolj pogumni in bomo želeli izračunati faktorsko potenco kakšnega večjega števila. Čaka pa nas neprijetno presenečenje, ko ugotovimo, da rezultat za števila od 13 naprej ni več pravilen. Če ste vraževerni, številka 13 ni prav nič kriva za to. Vzrok tiči drugje.

Če torej spremenimo vrstico

```
int n = 5;
```

v

```
int n = 13;
```

bomo namesto pričakovanih 6227020800, kolikor je 13 faktorsko v resnici, dobili 1932053504. Spremenimo tip spremenljivke $fakulteta$ v `short` in videli boste, da daje program zdaj napačne rezultate že za vrednosti spremenljivke n od 8 naprej. Dobimo, da je 8 faktorsko enako -25216. Zdaj se nam že počasi razkriva odgovor na vprašanje, zakaj toliko različnih tipov celih števil.

Za hranjenje podatkov v računalniku potrebujemo *prostor*. Večja so števila, več prostora potrebujemo, da jih hranimo. Če iz narave problema vemo, kako velika (oziroma majhna) števila bomo dobivali, potem lahko privarčujemo nekoliko pomnilnika tako, da izberemo ustrezno manjši podatkovni tip.

V naslednji tabeli so zbrani vsi celoštevilski podatkovni tipi z ustreznimi območji vrednosti, ki jih lahko hranijo:

Tip	**Območje vrednosti**
`short`	-32.768 do +32.767
`long`	-2.147.483.648 do +2.147.483.647
`unsigned short`	0 do +65.535
`unsigned long`	0 do +4.294.967.295

Od kod te nenavadne vrednosti, bomo videli v poglavju Kodiranje podatkov. Povejmo še to, da unsigned po angleško pomeni nepredznačen. *Nepredznačena števila* nimajo predznaka in pokrivajo zgolj območje nenegativnih vrednosti. Po drugi strani pokrivajo *predznačena števila* približno simetrično območje negativnih in pozitivnih vrednosti.

2.4.2 Znakovni tip

Digitalni računalnik hrani znake v obliki številskih vrednosti. V ta namen je bil določen mednarodni standard ASCII (American Standard Code for Information Interchange), ki vsakemu znaku enoumno priredi številsko vrednost. V naslednji tabeli so zbrani znaki s kodami med 32 in 125, ki jih zna prikazati tudi večina enostavnih prikazovalnikov LCD. Popolno tabelo bo bolj zvedav bralec brez težav našel v kateri izmed neštetih računalniških knjig ali na svetovnem spletu.

Koda	Znak	Koda	Znak	Koda	Znak	Koda	Znak
32	presledek	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	”	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	’	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	‘	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f		
55	7	79	O	103	g		

Za deklaracijo spremenljivke znakovnega tipa obstaja tip `char`, ki obstaja tudi v svoji nepredznačeni obliki `unsigned char`. Nekateri prevajalniki predznačene oblike niti ne poznajo in tolmačijo tudi tip `char` kar kot `unsigned char`. Glede na to, da so znaki shranjeni kot cela števila, imata tudi omenjena znakovna tipa svoje območje vrednosti:

Tip	Območje vrednosti
`char`	-128 do 127
`unsigned char`	0 do 255

Seveda lahko z znaki izvajamo tudi vse običajne številske operacije kot na primer seštevanje, odštevanje ali primerjava. Že v naslednjem primeru bomo to dejstvo s pridom uporabili.

V vsakdanjem življenju srečamo naprave, ki nimajo tipkovnice, a vseeno omogočajo uporabniku, da vnaša preprosta besedila. Primer je radijski ali televizijski sprejemnik, kjer lahko vtipkamo ime priljubljene postaje, ki smo jo shranili v spomin. Ime vtipkamo tako, da z dvema tipkama ”manjšamo” oziroma ”večamo” znak, s tretjo tipko pa znak potrdimo in se hkrati pomaknemo na naslednji znak. Tudi mobiteli nam omogočajo, da iz tipkovnice spravimo več črk kot je na voljo

tipk. To dosežemo tako, da eno in isto tipko pritisnemo večkrat zapored in se tako "sprehodimo" čez 3 ali 4 zaporedne črke.

Naslednji program na prikazovalniku najprej prikaže črko a, potem pa prikazani znak spremeni vsakokrat, kadar pritisnemo tipko T0 ali T1. Ob pritisku na T0 prikaže znak, ki ima za ena manjšo kodo ASCII, ob pritisku na T1 pa znak, ki ima za ena večjo kodo od trenutno prikazanega znaka.

```
#include "io.h"
//visual c++: zamenjaj gornjo vrstico z:
//#include <conio.h>

int main(void)
{
  char znak = 'a'; //znake pisemo v enojnih navednicah
  char tipka = 0;

  //visual c++: odstrani naslednje tri vrstice:

  _LCDInit(); //pripravi LCD za uporabo
  _KeyInit(); //pripravi tipke za uporabo
  _setcursortype(_NOCURSOR); //izklopi kurzor

  while (1)
  //visual c++: gornjo vrstico nadomesti z:
  //while (tipka != '3')
  {
    putch(znak);
    tipka = getch();
    switch (tipka)
    {
      case '0': znak--; break;
      case '1': znak++; break;
    }
    putch('\b');
  }
  return 0;
}
```

V neskončni zanki `while` v gornjem programu se najprej izpiše trenutna vrednost spremenljivke `znak`, ki je na začetku enaka črki a. Potem program počaka, da pritisnemo kakšno tipko. Če pritisnemo tipko T0 ali T1, potem se v stavku `switch` vrednost spremenljivke `znak` pomanjša oziroma poveča za ena. V naslednjem obhodu zanke se zato izpiše znak, ki je v tabeli ASCII eno mesto pred oziroma eno mesto za nazadnje prikazanim znakom. Da se znaki ne bi "kopičili" na prikazovalniku, čisto na koncu zanke vsakokrat izpišemo ubežno sekvenco `\b`, ki pomakne kurzor

za eno mesto v levo in izbriše nazadnje izpisani znak. Tako se bo naslednji znak izpisal na isto mesto kot znak pred njim, kar bo povzročilo učinek animacije.

Bralec naj razmisli, kako bi program dopolnil, da bi z uporabo tretje tipke (na primer T2) omogočil, da uporabnik potrdi prikazani znak in se pomakne za eno mesto naprej na vnos naslednjega znaka. V resnici ni potrebno narediti nič drugega, kot da v primeru, da pritisnemo tipko T2, zaobidemo izpis ubežne sekvence `\b`.

Razmislite tudi, kako bi program predelali, da bi bilo možno vnašati le omejen nabor znakov. Na primer samo velike črke in nekaj običajnih znakov, kot so pomišljaj, presledek, pika, okrogli oklepaj in zaklepaj ter morda še kak znak.

2.4.3 Realni tip

Jezik C pozna dva tipa realnih podatkov, ki ju povzema naslednja tabela:

Tip	**Približno območje vrednosti**	**Natančnost**
`float`	$\pm 10^{-38}$ do $\pm 10^{+38}$	prvih 7 mest (enojna natančnost)
`double`	$\pm 10^{-308}$ do $\pm 10^{+308}$	prvih 15 mest (dvojna natančnost)

Vidimo, da območje vrednosti tu ni tak problem kot pri celih številih. Zlasti je to res pri tipu `double`, saj v naravi praktično ne najdemo primerov, ki bi zahtevali tako ekstremne vrednosti. Kaj pa natančnost? Števil seveda ne moremo hraniti na poljubno mest natančno. Ne glede na to, kako velika ali majhna so števila, vedno nosi koristno informacijo le prvih 7 oziroma 15 mest, ki so različne od nič. Večinoma nas takšna natančnost zadovolji, obstajajo pa primeri, ko lahko napake zaradi zaokroževanja postanejo tako velike, da prerastejo sam rezultat, ki je tako povsem neuporaben. S takšnimi problemi se ukvarja področje numerične matematike, mi pa si za ilustracijo poglejmo le primer, kaj se lahko zgodi.

Vzemimo, da v danem okolju naselimo neko živalsko vrsto (na primer lemurje) in opazujemo, kako se količina populacije P spreminja s časom. Predpostavimo, da v okolju vladajo konstantni življenski pogoji, ki v vsakem trenutku prenesejo P_{max} osebkov. Če je osebkov preveč ($P > P_{max}$), potem bo začelo njihovo število upadati zaradi prostorske stiske, pomanjkanja hrane ipd. Če je osebkov manj kot jih prenese okolje ($P < P_{max}$), potem se bodo veselo množili in bo njihovo število naraščalo.

Vzemimo, da imamo v času n populacijo lemurjev $p_n = P_n/P_{max}$ izraženo v odstotku zasičenosti okolja. Sto odstotna zasičenost je predtsavljena z vrednostjo $p_n = 1$.

Prirastek lemurjev v času $n + 1$ bo enak

$$\frac{p_{n+1} - p_n}{p_n} \tag{1}$$

Prirastek je sorazmeren količini prostega okolja. Če je populacija manjša, kot jo prenese okolje, bo prirastek pozitiven, sicer bo negativen:

$$\frac{p_{n+1} - p_n}{p_n} \propto 1 - p_n \tag{2}$$

Uvedemo količnik rasti r in dobimo

$$\frac{p_{n+1} - p_n}{p_n} = r(1 - p_n) \tag{3}$$

In končno dobimo populacijo lemurjev v času $n + 1$:

$$p_{n+1} = p_n + p_n r(1 - p_n) \tag{4}$$

Napišimo zdaj program, ki bo računal količino populacije v različnih časovnih obdobjih. Problem je *iterativen*. To pomeni, da lahko vsako naslednjo vrednost izračunamo le tako, da poznamo vse prejšnje. Vsakemu koraku izračuna pravimo *iteracija*. Za začetek izberemo eno odstotno zasedenost okolja (0.01), količnik rasti pa naj bo $r = 0.2$. Program naj na prikazovalnik izpisuje stanje le v vsaki peti iteraciji.

```
#include "io.h"
//visual c++: zamenjaj gornjo vrstico z:
//#include <conio.h>
//#include <stdio.h>

int main(void)
{
  int n = 1;   //stevec iteracij
  float pn = 0.01, pn_1;
  float r = 0.2;

  //visual c++: odstrani naslednji dve vrstici:
  _KeyInit();
  _LCDInit();

  while (1)
  //visual c++: zamenjaj gornjo vrstico z:
  //while (n <= 100)
  {
    pn_1 = pn + pn * r * (1 - pn);
    pn = pn_1;
    n++;
    if (n % 5 == 0) //izpisemo le vsako peto iteracijo
    {
      printf("%3d: %.6f\r", n, pn);
      getch();
    }
  }

  return 0;
```

```
}
```

V gornjem programu najprej nastavimo začetne vrednosti spremenljivk. Spremenljivka n bo služila kot števec iteracij. Spremenljivka pn bo vedno hranila velikost populacije v danem trenutku n, spremenljivka pn_1 pa v naslednjem trenutku $n + 1$.

V prvi vrstici zanke `while` najprej po enačbi (4) izračunamo velikost populacije v času $n + 1$. Potem posodobimo staro vrednost (pn), ki je ne bomo več potrebovali, in povečamo števec iteracij za ena (`n++`).

Funkcija `printf()` povzroči izpis zaporedne številke iteracije in količine populacije v danem trenutku. Številka 3 v prvem formatnem določilu pomeni, da se bo zaporedna številka iteracije izpisala na treh mestih. Pred številkami, ki so krajše od treh mest, se postavi ustrezno število presledkov. Številka .6 v drugem formatnem določilu pomeni, da se bo količina populacije izpisala s šestimi decimalnimi mesti.

Ostanek deljenja spremenljivke n s 5 je enak nič (`n % 5 == 0`) le za vrednosti spremenljivke n, ki so mnogokratniki števila 5. Tako se dejansko izpišejo le stanja vsake pete iteracije. Funkcija `getch()` v našem primeru služi le temu, da program po izpisu rezultata čaka, dokler uporabnik ne pritisne katerekoli tipke in šele potem nadaljuje z računanjem. Brez klica te funkcije bi se vrednosti na prikazovalniku izpisovale tako hitro, da jih ne bi uspeli prebrati.

V naslednji tabeli vidimo, kako se relativna količina populacije počasi bliža 100% in pri tej vrednosti obmiruje. To je za pričakovati, saj se iz enačbe (2) lepo vidi, da je pri vrednosti $p_n = 1$ prirastek enak nič.

Število iteracij n	**Relativna količina populacije** p_n
5	0.020552
10	0.049861
15	0.116757
20	0.252825
25	0.471932
30	0.712113
35	0.877668
40	0.955437
45	0.984834
50	0.994967
55	0.998344
60	0.999457
65	0.999822
70	0.999942
75	0.999981
80	0.999994
85	0.999998
90	0.999999
95	1.000000
100	1.000000

Bodimo zdaj bolj pogumni in nasujmo našim lemurjem malo afrodiziaka. Ravno toliko, da se količnik rasti r poveča na 3. Naslednja tabela prikazuje izračune najprej z uporabo spremenljivk z enojno natančnostjo (`float`), potem pa še z dvojno natančnostjo (`double`).

Število iteracij n	p_n **(enojna natančnost)**	p_n **(dvojna natančnost)**
5	1.288978	1.288978
10	0.215593	0.215587
15	0.521926	0.521671
20	0.165525	0.171085
25	0.996441	0.743567
30	1.106520	1.232112
35	0.989278	0.297907
40	1.265200	0.002909

Pričakujemo, da bomo pri računanju z večjo natančnostjo dobili tudi natančnejše rezultate. Ko primerjamo rezultate v štirideseti iteraciji, pa podvomimo v pravilnost kateregakoli od obeh izračunov, saj se razlikujeta za več kot 400-krat! Ob tako velikem količniku rasti postane naš sistem kaotičen, kar pomeni, da infinitezimalna sprememba stanja v danem trenutku po dovolj dolgem času povzroči poljubno veliko spremembo. Tako je pri nas sprememba vrednosti na sedmem decimalnem mestu po 40 iteracijah že presegla vse meje dobrega okusa. V takšnem primeru ni računalnika, ki ne bi po dovolj dolgem času dal popolnoma neveljavnih rezultatov. Presenečenj še ni konec. Če isti program zaženemo v okolju Visual C++, dobimo spet čisto druge vrednosti. Ne samo sprememba natančnosti, ampak tudi drugačen način izračunavanja matematično identičnih izrazov daje drugačne rezultate.

Nauk te zgodbe je, da ne smemo slepo zaupati računalnikom in računskim rezultatom, ki nam jih ponudijo. Samo z dobrim matematičnim ali fizikalnim razumevanjem problema si lahko zagotovimo uporabne rezultate.

2.4.4 Prazen podatkovni tip `void`

Podatkovni tip `void` je podatkovni tip brez vrednosti. To si ta hip verjetno težko predstavljete. Ko govorimo o podatkovnem tipu, vedno govorimo o podatkovnem tipu izraza (spremenljivka je le poseben primer izraza). Za izraze pa smo bili doslej navajeni, da imajo vedno neko vrednost.

Prazen tip se uporablja le v povezavi s funkcijami in kazalci. Ko bomo govorili o njih, bo pomen tega tipa postal bolj jasen.

2.4.5 Pretvorba podatkovnega tipa

Jezik C je zelo strog glede podatkovnih tipov spremenljivk. Za razliko od jezika JavaScript, kjer se lahko tip spremenljivke praktično poljubno spreminja, v jeziku C to ni mogoče. Ko je spremenljivka enkrat deklarirana, je zapečaten tudi njen tip.

Kaj pa, če se v enem izrazu znajde več vrednosti različnih tipov? Potem se morajo tipi nujno prilagoditi, ampak le začasno, za potrebe računanja konkretnega izraza. Tipi spremenljivk se dejansko ne spreminjajo.

Nekatere pretvorbe tipov se izvajajo avtomatsko, nekatere pa moramo zahtevati.

Avtomatska pretvorba: Poglejmo si nekaj primerov, kjer pride do avtomatske pretvorbe podatkovnega tipa. Če, na primer, spremenljivki tipa `int` priredimo konstantno realno vrednost, potem se slednja pretvori v celoštevilsko vrednost. Pri pretvorbi se enostavno odreže del za decimalno piko:

```
int x = -7.1;
```

Spremenljivka `x` bo tako dobila vrednost -7. Ker je pretvorba avtomatska, jo prevajalnik navadno pospremi z opozorilom, da je prišlo do krnjenja (truncation) podatka in posledično do izgube dela informacije.

Čeprav malo manj očitno, kaže naslednji primer podobno situacijo:

```
float y = -7.1;
```

Tu spremenljivka y sicer dobi pravilno vrednost -7.1, vendar nas bo večina prevajalnikov vseeno opozorila na krnjenje podatka. Da to razumemo, je potrebno vedeti, da se v Ceju konstantne realne številske vrednosti obravnavajo kot vrednosti tipa `double`. Ker v gornjem primeru zapisujemo podatek dvojne natančnosti (`double`) v spremenljivko enojne natančnosti (`float`), pri tem seveda izgubljamo del informacije.

Če nas to ne moti, motijo pa nas številna nepotrebna opozorila prevajalnika, potem lahko pretvorbo zahtevamo sami. V tem primeru nas prevajalnik seveda ne bo opozoril na krnjenje podatka, saj smo ga sami eksplicitno zahtevali. Očitno vemo, kaj delamo.

Zahtevana pretvorba: Zahtevo po pretvorbi podamo tako, da pred podatek, katerega tip pretvarjamo, v oklepaje zapišemo želeni tip:

```
float y = (float) -7.1;
```

Zdaj smo pretvorbo tipa izrecno zahtevali in opozorilo prevajalnika bo izginilo.

Še en pogost primer zahteve po pretvorbi je, kadar želimo spremenljivki tipa `char` prirediti celoštevilsko vrednost. Nekateri prevajalniki tega ne bodo dovolili, če ne zahtevamo pretvorbe tipa:

```
char znak = (char) 65;
```

2.5 Operatorji

V naslednji tabeli so povzeti vsi cejevski operatorji:

Prednost	Operatorji	Vrstni red izvajanja
1	`() [] -> .`	z leve proti desni
2	`! ~ ++ -- +(predznak) -(predznak) *(indirekcija) &(naslovni operator) (pretvorba tipa) sizeof`	z desne proti levi
3	`* / %`	z leve proti desni
4	`+ -`	z leve proti desni
5	`<< >>`	z leve proti desni
6	`< <= > >=`	z leve proti desni
7	`== !=`	z leve proti desni
8	`&`	z leve proti desni
9	`^`	z leve proti desni
10	`|`	z leve proti desni
11	`&&`	z leve proti desni
12	`||`	z leve proti desni
13	`?:`	z desne proti levi
14	`= += -= *= /= %= &= ^= |= <<= >>=`	z desne proti levi
15	`,`	z leve proti desni

Kot že vemo, se v primeru, da je v enem izrazu več različnih operatorjev, izvedejo najprej tisti z nižjo zaporedno številko prednosti. Če je več operatorjev z isto prednostjo, se izvajajo po vrsti z leve proti desni ali z desne proti levi, kakor je to navedeno v desnem stolpcu.

Vidimo, da se je za razliko od jezika JavaScript pojavilo nekaj novih operatorjev, ki jih bomo spoznali, ko bo za to čas, prav tako pa je nekaj operatorjev izginilo. Za naše potrebe se v resnici ni zgodilo nič dramatičnega. Verjetno bo večina bralcev opazila, da sta izginila primerjalna operatorja `===` in `!==`. To bo imelo za posledico tudi majhno razliko v delovanju stavka `switch`. Njegovo delovanje je v jeziku JavaScript namreč slonelo ravno na operatorju identitete (`===`).

2.6 Izrazi

2.6.1 Logični in primerjalni izrazi

Zelo pomembna razlika, ki jo za nas prinaša jezik C je ta, da tu ne obstajata Boolovi vrednosti `true` in `false`. Namesto njiju se uporabljata vrednosti 0 in 1 oziroma 0 in različno od 0. Pravzaprav to za nas ni posebna novost, kvečjemu poenostavitev. Spomnimo se namreč, da se v jeziku JavaScript vrednost `false` pretvori v 0, če jo uporabimo v kakšni operaciji, ki pričakuje številsko vrednost. Podobno se vrednost `true` pretvori v 1. Obratno se vrednost 0 pretvori v vrednost `false`, kadar se pojavi na mestu, kjer se pričakuje Boolova vrednost. Vrednost različna od 0 se pretvori v `true`.

V jeziku C velja: Primerjalni operatorji vrnejo 0, če relacija, ki jo preverjajo, ne drži, in 1, če relacija drži.

Primer: če ima spremenljivka x vrednost 7, potem izraz `x < 10` vrne vrednost 1, izraz `x != 7` pa vrne vrednost 0.

Za izračun vrednosti izrazov z logičnimi operatorji uporabimo naslednje tri tabele.

Logični in (`&&`):

`x`	`y`	`x && y`
0	0	0
0	različen od 0	0
različen od 0	0	0
različen od 0	različen od 0	1

Vidimo, da operator logični in vrne 1 le v primeru, ko sta in levi in desni operand različna od nič. Sicer vrne 0.

Logični ali (`||`):

`x`	`y`	`x || y`
0	0	0
0	različen od 0	1
različen od 0	0	1
različen od 0	različen od 0	1

Operator logični ali vrne 1 v primeru, ko je vsaj eden od obeh operandov različen od nič. Sicer vrne 0.

Logična negacija (`!`):

`x`	`!x`
0	1
različen od 0	0

Operator negacije vrne 1 le v primeru, ko je njegov operand enak 0. Sicer vrne 0.

2.6.2 Celoštevilsko deljenje

Kot smo že videli, je jezik C precej strog glede podatkovnih tipov posameznih spremenljivk. Ko se enkrat odločimo, kakšnega tipa bo spremenljivka, tega ne moremo več spremeniti, razen začasno s pomočjo operatorja za zahtevano pretvorbo tipa.

V tem slogu operator deljenja v primeru, da sta in števec in imenovalec oba celoštevilskega tipa, vrne celoštevilski rezultat. Dobi ga tako, da enostavno odreže del za decimalno piko. Rezultat je celoštevilski, tudi če po matematičnih zakonitostih tega ne bi pričakovali.

V naslednjem delu programa dobi spremenljivka x vrednost 3, čeprav je realnega tipa. Izraz na desni strani priredilnega operatorja predstavlja namreč celoštevilsko deljenje:

```
float x;
int a = 10, b = 3;
x = a / b;
```

Če bi hoteli dobiti pravilen realni rezultat, bi morali vsaj eno od spremenljivk a in b deklarirati kot realno spremenljivko. Druga pot je, da izrecno zahtevamo pretvorbo katere od obeh spremenljivk:

```
x = (float) a / b;
```

Nič drugače ni, če imamo opravka s konstantami. Izraz `-10 / 3` bo vrnil vrednost -3, izraza `-10 / (float) 3` in `-10 / 3.0` pa bosta vrnila -3.333333. Konstantna vrednost 3.0 se namreč tolmači kot vrednost realnega tipa, natančneje, tipa `double`.

2.7 Krmilni stavki

Jezik C pozna enake krmilne stavke kakor JavaScript. Ker so ti stavki zelo pomembni, jih bomo na tem mestu vse ponovili. Edini razliki, ki ju bomo opazili, izhajata iz dejstva, da C ne pozna operatorja identičnosti (`===`) ter vrednosti `true` in `false`, ki sta v jeziku JavaScript vrednosti Boolovega podatkovnega tipa.

V jeziku C temeljijo vse odločitve na številskih vrednostih nič (ki ustreza vrednosti `false`) in različno od nič (ki ustreza vrednosti `true`). To v resnici ni nobena novost, če se spomnimo pretvorb podatkovnih tipov v jeziku JavaScript: kadar se na mestu, kjer se pričakuje Boolova vrednost, pojavi številska vrednost, se vrednost nič pretvori v `false`, vrednost različna od nič pa v `true`.

2.7.1 Stavka `if` in `if..else`

Sintaktični zapis stavka `if` je

```
if (p) s;
```

Če je vrednost izraza `p` različna od nič, se bo izvedel stavek `s`. Naj se spomnimo, da je stavek `s` lahko navaden stavek, sestavljen stavek (več stavkov združenih s parom zavitih oklepajev) ali krmilni stavek.

V naslednjem primeru se bo vrednost spremenljivke y na koncu postavila na 10, kajti vrednost izraza `x < 2` je 1 in potemtakem različna od nič:

```
int x = 0, y = 1;
if (x < 2) y = 10;
```

Morda nekoliko bolj nenavaden primer je naslednji:

```
int x = -2, y = 1;
if (x + 2) y = 10;
```

Tu bo y ostal enak 1, kajti stavek `y = 10` se ne bo izvršil, ker je vrednost izraza `x + 2` enaka 0.

Sintaktični zapis stavka `if..else` je

```
if (p) s1;
else s2;
```

Če je vrednost izraza `p` različna od nič, se bo izvedel stavek `s1`, sicer se bo izvedel stavek `s2`.

2.7.2 Stavek `switch`

S stavkom `switch` izbiramo med večimi različnimi možnostmi. Njegov sintaktični zapis je

```
switch (izraz)
{
  case vrednost1: stavki1;
  case vrednost2: stavki2;
  case vrednost3: stavki3;
  //...
  case vrednostN: stavkiN;
  default: ostalo;
}
```

Najprej se izračuna vrednost izraza `izraz` in se po vrsti primerja z vrednostmi od `vrednost1` do `vrednostN`. Če je `izraz` enak kateri od naštetih vrednosti (na primer vrednosti `vrednostn`), se izvedejo po vrsti vsi stavki od `stavkin` od `stavkiN` in na koncu še stavki `ostalo`. Če se nobena primerjava ne izide, potem se izvedejo le stavki `ostalo`, ki sledijo besedi `default`, ki pa ni obvezen del stavka `switch`.

Spomnimo se, da v stavek `switch` pogosto vključimo stavke `break`, s katerimi lahko prekinemo izvajanje stavka. Le tako lahko dosežemo, da v resnici izbiramo med večimi možnostmi.

Razlika med jezikoma C in JavaScript je ta, da je lahko `izraz` samo celoštevilskega ali znakovnega tipa. Primerjava, ki se izvaja med izrazom in vrednostmi, pa ni primerjava z operatorjem identičnosti (`===`), ki ga C ne pozna, ampak s primerjalnim operatorjem enakosti (`==`).

2.7.3 Stavek `while`

To je ponavljalni stavek, katerega sintaktični zapis je

```
while (p) s;
```

Najprej se izračuna vrednost izraza `p`. Če je vrednost različna od nič, potem se izvede stavek `s` in takoj za tem se ponovno izračuna vrednost izraza `p`. Če je ta še vedno različen od nič, se cikel ponovi. In tako vse, dokler ne postane izraz `p` enak nič.

S stavkom

```
while (1)
{
  //koda
}
```

tako dosežemo, da se `koda` izvaja neprestano, saj je 1 vedno različna od nič.

2.7.4 Stavek `do..while`

Še en ponavljalni stavek, katerega sintaktični zapis je

```
do s; while (p);
```

Tu se najprej izvede stavek `s` in šele na to izračuna vrednost izraza `p`. Če je ta različna od nič, se cela zgodba ponovi. In to vse dokler ne postane izraz `p` enak nič.

2.7.5 Stavek `for`

To je tretji ponavljalni stavek, katerega sintaktični zapis je

```
for (s1; p; s2) s3;
```

Tu se najprej izvede stavek `s1`. Potem se izračuna vrednost izraza `p`. Če je ta različna od nič, se po vrsti izvedeta stavka `s3` in `s2`. Potem je spet na vrsti izračun izraza `p`. Če je še vedno različen od nič, se spet izvedeta stavka `s3` in `s2`. In tako dalje, dokler izraz `p` ne postane enak nič.

Stavek `for` se najpogosteje uporablja, kadar imamo opravka s kakšnim štetjem. V stavku `s1`, ki se zgodi le enkrat, nastavimo začetno vrednost števca, z izrazom `p` preverjamo, če je števec že dosegel končno vrednost, stavek `s2` pa povzroča spreminjanje vrednosti števca.

2.8 Funkcije

Funkcije so deli programske kode, ki jih kodiramo ločeno in jih lahko po potrebi kličemo iz poljubnega dela programa. Veliko o funkcijah smo izvedeli že pri spoznavanju jezika JavaScript in veselilo nas bo, da lahko večino osvojenega znanja uporabimo tudi v Ceju.

Poglavitna razlika med funkcijami v obeh jezikih izvira iz dejstva, da jezik C zahteva, da se programer opredeli glede podatkovnih tipov. Ker je klic funkcije izraz, ne bo nobeno presenečenje, da moramo funkciji določiti tip.

Formalni parametri funkcije so lokalne spremenljivke funkcije – tudi to že vemo iz jezika JavaScript – in jim moramo prav tako določiti tipe.

Tip funkcije in njenih parametrov določimo v t.i. *prototipu*:

```
tip_funkcije ime_funkcije(tip1 param1, tip2 param2, ...);
```

Prototip služi prevajalniku v prvi fazi prevajanja, da preveri, če smo funkcijo pravilno klicali. V tem trenutku ni važno, kaj funkcija v resnici počne. V tem smislu deluje prototip funkcije kot ohišje kakšne naprave. Iz ohišja je že razvidno, kakšne vrste kablov lahko nanj priklopimo. To lahko storimo pravilno, tudi če ne vemo, kako naprava sploh deluje. Po kablih vodimo signale v napravo in iz nje, tako kot preko parametrov in klica funkcije v funkcijo in iz nje dobivamo podatke.

Pomembno je, da je lahko tip funkcije katerikoli veljaven podatkovni tip, lahko pa je tudi prazen tip `void`. Funkcija tipa `void` ne vrača nobene vrednosti. Prazen tip lahko uporabimo tudi namesto seznama formalnih parametrov. V tem primeru funkcija ne bo sprejemala nobenih parametrov.

Funkcijo, ki niti ne sprejema parametrov niti ne vrača vrednosti, deklariramo takole:

```
void ime_funkcije(void);
```

Funkcijo tipa `void` lahko kličemo samo na en način:

```
ime_funkcije(izraz1, izraz2, ...);
```

oziroma

```
ime_funkcije();
```

če je funkcija brez parametrov. Pomembno je, da takšna funkcija ne more nastopiti kot del kakšnega drugega izraza.

Funkcijo, ki ni tipa `void`, lahko kličemo na enak način, poleg tega pa lahko takšno funkcijo kličemo tudi kot del poljubnega izraza. Na primer:

```
spremenljivka = ime_funkcije(izraz1, izraz2, ...);
```

Če bi vam kdo prodal samo ohišje kakšne naprave, bi jo vi sicer lahko priklopili, a kmalu bi ugotovili, da je ohišje prazno. Tudi prevajalnik bo v drugi fazi prevajanja opazil, da mu manjka *definicija* funkcije:

```
tip_funkcije ime_funkcije(tip1 param1, tip2 param2, ...)
{
  telo funkcije
}
```

Definicija je podobna prototipu, le da smo dodali še telo funkcije. Telo je koda, ki se ob klicu funkcije izvede. To pa ni nič več novega, saj smo na podoben način definirali tudi funkcije v jeziku JavaScript.

Za konec si oglejmo še primer funkcije, ki vrne večjo od dveh podanih realnih vrednosti:

```
#include "io.h"
//visual c++: zamenjaj gornjo vrstico z:
//#include <stdio.h>

float maks(float a, float b);

int main(void)
{
  float x = 2.5, y = 4.6;
  float rez;
  _LCDInit();  //visual c++: odstrani to vrstico

  rez = maks(x, y);
  printf("%f", rez); //na prikazovalnik se izpise 4.6
  while (1);   //visual c++: odstrani to vrstico
  return 0;
}

float maks(float a, float b)
{
  return a > b ? a : b;
}
```

V programu ni ničesar posebno novega, rezen dejstva, da sedaj prevajalnik budno bedi nad tem, da se tipi vseh izrazov, ki sodelujejo pri klicu funkcije, ujemajo z napovedanimi tipi v prototipu. Vseeno ponovimo mehanizem klica funkcije.

Ob klicu se najprej vrednosti vseh dejanskih parametrov (v našem primeru sta to vrednosti spremenljivk x in y) prekopirajo v istoležne formalne parametre, ki so lokalne spremenljivke klicane funkcije. Vrednost spremenljivke x se prekopira v spremenljivko a in vrednost spremenljivke y se prekopira v spremenljivko b. Ko izvajanje funkcije doseže stavek `return`, se funkcija konča in vrne vrednost izraza, ki je za stavkom `return`. V našem primeru je to izraz `a > b ? a : b`, v katerem z odločitvenim operatorjem izberemo večjo od obeh spremenljivk.

2.8.1 Območje spremenljivke

Pojem *območje spremenljivke* že poznamo iz jezika JavaScript. Zaradi pomembnosti, in ker bomo v zvezi z njim spoznali še en drug pojem, ga tu ponovimo. Kadar govorimo o območju spremenljivke, imamo v mislih območje programa, kjer je ta spremenljivka dostopna. Vemo, da vseh spremenljivk ne moremo uporabljati povsod v programu. Glede na območje dostopa ločimo dve vrsti spremenljivk:

Lokalna spremenljivka: Spremenljivki, ki je deklarirana v telesu funkcije ali kot formalni parameter funkcije, pravimo *lokalna spremenljivka*. Takšna spremenljivka je dostopna le v

funkciji, v kateri je deklarirana. V tem smislu sta, na primer, spremenljivki x in y iz zadnjega primera lokalni spremenljivki funkcije `main()` in nista dostopni znotraj funkcije `maks()`.

Neinicializirana lokalna spremenljivka ima poljubno vrednost.

Globalna spremenljivka: Spremenljivki, ki je deklarirana zunaj katerekoli funkcije, pravimo *globalna spremenljivka*. Taka spremenljivka je dostopna kjerkoli v programu, razen v funkcijah, ki imajo kakšno lokalno spremenljivko z istim imenom.

Če je programska koda zapisana v večih datotekah, potem je globalna spremenljivka dostopna le v datoteki, v kateri je deklarirana. Kadar želimo, da postane spremenljivka dostopna tudi v kaki drugi datoteki, jo v tej datoteki ponovno deklariramo in pred deklaracijo postavimo besedo `extern`.

Neinicializirana globalna spremenljivka ima vrednost 0.

2.8.2 Obstoj spremenljivke

Pojem *obstoj spremenljivke* je tesno povezan z območjem spremenljivke. Površnemu bralcu se bo morda prvi hip zdelo, da gre za eno in isto reč, čeprav je med obema pomembna razlika. Tudi glede na obstoj ločimo dve vrsti spremenljivk:

Avtomatičen obstoj: Spremenljivka, ki ima *avtomatičen obstoj*, obstaja le v času izvajanja funkcije, v kateri je deklarirana. Avtomatičen obstoj imajo lokalne spremenljivke. V praksi to pomeni, da lokalna spremenljivka izgubi svojo vrednost, čim se izvajanje funkcije konča.

Statičen obstoj: Spremenljivka, ki ima *statičen obstoj*, obstaja ves čas izvajanja programa. Statičen obstoj imajo globalne spremenljivke in lokalne spremenljivke, če jih ob deklaraciji opremimo s ključno besedo `static`.

Bistvena razlika med pojmoma območja in obstoja spremenljivke je ta, da spremenljivka, ki obstaja, ni nujno dostopna. To se lahko zgodi v treh primerih:

1. Globalno spremenljivko prekrije lokalna spremenljivka: Globalna spremenljivka obstaja ves čas, vendar do nje nimamo dostopa iz funkcije, ki ima lokalno spremenljivko z istim imenom.

2. Lokalna statična spremenljivka ni dostopna iz drugih funkcij: Lokalna spremenljivka, ki jo deklariramo kot `static`, obstaja ves čas, vendar je dostopna le iz funkcije, v kateri je deklarirana.

3. Lokalna spremenljivka začasno ni dostopna zaradi klica druge funkcije: Funkcija (na primer `a()`) lahko kliče kakšno drugo funkcijo (na primer `b()`). Ker se izvajanje funkcije `a()` še ni končalo, v času izvajanja funkcije `b()` lokalne spremenljivke funkcije `a()` še vedno obstajajo. Dostopne pa te spremenljivke niso. Ta situacija je nastopila v zadnjem primeru, ko lokalni spremenljivki x in y funkcije `main()` nista dostopni v času izvajanja funkcije `maks()`, čeprav nedvomno obstajata, saj se funkcija `main()` še ni končala. V resnici se ta funkcija v našem primeru nikoli ne konča, zaradi neskončne zanke `while (1);`

Za konec si oglejmo še en primer funkcije, ki vrača največjo vrednost. Tokrat bo funkcija sprejela en sam celoštevilski parameter in jo bomo morali klicati večkrat. Vsakokrat bo vrnila vrednost, ki je bila od vseh podanih vrednosti do tistega trenutka največja.

Težava je v tem, da si mora funkcija na nek način zapomniti določene podatke iz svojih prejšnjih klicev. To je možno na dva načina. Uporabimo lahko globalno spremenljivko, kar pa ni priporočljivo. Preveč globalnih spremenljivk namreč poveča možnost nenamerne hkratne uporabe ene in iste spremenljivke v dva različna namena. Takšne napake je velikokrat zelo težko odkriti. Drugi način je, da uporabimo lokalno statično spremenljivko. Mi se odločimo za drugi način.

Tule je primer:

```
#include "io.h"
//visual c++: zamenjaj gornjo vrstico z:
//#include <stdio.h>
//#include <conio.h>

int maks(int x); //funkcija vrne najvecjo vrednost od vseh podanih

int main(void)
{
  int znak;

  //visual c++: odstrani spodnji dve vrstici:
  _LCDInit();
  _KeyInit();

  do
  {
    znak = getch();
    printf("Maks: %3d\r", maks(znak));
  } while (1);
  //visual c++: zamenjaj gornjo vrstico z:
  //} while (znak != 'x');

  return 0;
}

int maks(int x)
{
  static int prvic = 1;
  static int najv;

  if (prvic)
  { //prvi klic funkcije
```

```
    najv = x;
    prvic = 0;
  }
  else if (x > najv)
  { //vsi ostali klici
    najv = x;
  }
  return najv;
}
```

Program deluje tako, da neprestano prebira kode pritisnjenih tipk in vsako sproti poda funkciji `maks()`. Ta funkcija podano kodo primerja z doslej največjo, in večjo od obeh vrednosti vrne.

Da lahko to nalogo opravi kakor je treba, potrebuje dve statični spremenljivki *prvic* in *najv*. S pomočjo spremenljvke *prvic* bo funkcija vedela, ali gre za prvi klic ali ne. Ob prvem klicu nima podane vrednosti s čim primerjati in jo enostavno shrani v spremenljivko *najv* (`najv = x;`). Hkrati si mora funkcija zapomniti, da je prvi klic že opravljen, in to stori tako, da spremenljivko *prvic* postavi na 0 (`prvic = 0;`).

Ob drugem in vseh ostalih klicih funkcija enostavno primerja podano vrednost z doslej največjo. Če je podana vrednost večja, jo shrani (`najv = x;`).

Funkcijo `maks()` bi lahko krajše, vendar manj razumljivo, definirali tudi takole:

```
int maks(int x)
{
  static int prvic = 1;
  static int najv;

  if (x > najv || prvic)
  {
    najv = x;
    prvic = 0;
  }
  return najv;
}
```

2.9 Primeri izpitnih vprašanj

Opomba: Včasih boste med primeri vprašanj našli dve zelo podobni vprašanji z istim naborom odgovorov. Zaradi malenkost spremenjenega vprašanja lahko postane pravilen kateri drug odgovor. Sem in tja boste zasledili tudi dve popolnoma enaki vprašanji, kjer so zamenjani samo napačni odgovori. Takšna vprašanja vas opozarjajo na natančnost pri branju tako vprašanja kot tudi vseh ponujenih odgovorov. Velika nevarnost je, da na izpitu naletite na podobno vendar ne enako vprašanje, kakršnega ste že videli, in takrat zaradi nepazljivosti obkrožite napačen odgovor. Pomembno je tudi, da med učenjem ne ugotavljate le, kateri odgovor je pravilen in zakaj, ampak tudi, zakaj je vsak od ostalih treh napačen. Lahko se preizkusite tudi v tem, da si poskusite

zamisliti podobno vprašanje, ki bo imelo za pravilen odgovor katerega od ponujenih napačnih odgovorov.

1. V programu želimo klicati funkcijo `sin()`, ki je v knjižnici `math.h`. Kako to knjižnico vključimo v program?

   (a) `include #math.h`

   (b) `#include <math.h>`

   (c) `#include math.h`

   (d) `include <math.h>`

2. Vemo, da bodo v našem programu nastopale le vrednosti med -20000 in 0. Kateri podatkovni tip spremenljivke moramo uporabiti, da bo poraba prostora v pomnilniku čim manjša?

   (a) `short`

   (b) `unsigned short`

   (c) `long`

   (d) `unsigned long`

3. V spremenljivki *crka*, ki je znakovnega tipa, imamo shranjeno kodo male tiskane črke. Kako lahko to kodo spremenimo v kodo iste, vendar velike tiskane črke.

   (a) `crka += 'A';`

   (b) `crka -= 'A';`

   (c) `crka += 'A' - 'a';`

   (d) `crka -= 'A' - 'a';`

4. Shraniti želimo neko realno število na vsaj 10 mest natančno. Kateri podatkovni tip spremenljivke moramo uporabiti?

   (a) `long`

   (b) `unsigned short`

   (c) `float`

   (d) `double`

5. Realna spremenljivka x ima vrednost 2.5. Kakšen izpis bo na prikazovalniku povzročil klic funkcije `printf("x=%.3f", x)`?

   (a) `x= 2.5`

   (b) `x=2.50`

   (c) `x=2.500`

(d) `x=2.5`

6. Celoštevilska spremenljivka x ima vrednost 8. Kakšen izpis bo na prikazovalniku povzročil klic funkcije `printf("x=%2d", x)`?

   (a) `x=8`

   (b) `x=28`

   (c) `x= 8`

   (d) `x=%2d`

7. Celoštevilska spremenljivka x ima vrednost 15. Kakšen izpis bo na prikazovalniku povzročil klic funkcije `printf("%d,%d", x, x + 1)`?

   (a) `1516`

   (b) `15,16`

   (c) `1615`

   (d) `16,15`

8. Celoštevilska spremenljivka x ima vrednost 15. Kakšen izpis bo na prikazovalniku povzročil klic funkcije `printf("%d,%d", x, x + 1)`?

   (a) `1516`

   (b) `15,16`

   (c) `15 16`

   (d) `15, 16`

9. V kateri od naslednjih deklaracij prihaja do krnjenja podatka na desni strani priredilnega operatorja (= podatek na desni vsebuje več informacije kot jo lahko hranimo v spremenljivki na levi)?

   (a) `double x = 3.1;`

   (b) `int x = 3;`

   (c) `float x = 3.1;`

   (d) `char x = 104;`

10. Spremenljivka x je deklarirana kot celoštevilska spremenljivka. Kakšna je vrednost izraza `x || (x + 1)`?

    (a) -1

    (b) 0

    (c) 1

(d) Odvisno od vrednosti spremenljivke x.

11. Spremenljivka x je deklarirana kot celoštevilska spremenljivka. Kakšna je vrednost izraza `x && (!x)`?

    (a) -1

    (b) 0

    (c) 1

    (d) Odvisno od vrednosti spremenljivke x.

12. Spremenljivka x je deklarirana kot celoštevilska spremenljivka. Kakšna je vrednost izraza `1 < x < 10`?

    (a) -1

    (b) 0

    (c) 1

    (d) Odvisno od vrednosti spremenljivke x.

13. Kakšna je vrednost izraza `3 / y / 2`, če je spremenljivka y tipa `int` in ima vrednost 4?

    (a) 0

    (b) 0.375

    (c) 1

    (d) 1.5

14. Spremenljivka x je deklarirana kot spremenljivka tipa `int` in ima vrednost 7. Kateri od naslednjih izrazov edini nima vrednosti 7?

    (a) `x / 8 * 8`

    (b) `x / 8.0 * 8`

    (c) `(float) x / 8 * 8`

    (d) `x * 8 / 8`

15. Kakšna je vrednost izraza `(int) -3.7`?

    (a) `-3`

    (b) `-4`

    (c) `3`

    (d) `4`

16. Kako zapišemo prototip funkcije `neki()`, ki sprejme en parameter tipa `float` in ne vrača ničesar?

(a) `void neki(float x);`

(b) `float neki(void);`

(c) `neki(float x);`

(d) `neki(void, float x);`

17. Kako zapišemo prototip funkcije `neki()`, ki sprejme dva parametra tipa `float` in ne vrača ničesar?

(a) `void neki(float x, y);`

(b) `void neki(float x; y);`

(c) `void neki(float x; float y);`

(d) `void neki(float x, float y);`

18. Podan je prototip funkcije `void neki(void);` Kateri od naslednjih klicev te funkcije je sintaktično pravilen?

(a) `rez = neki();`

(b) `neki(1);`

(c) `neki();`

(d) `neki() + 1;`

19. Podan je prototip funkcije `float abs(float x);` Kateri od naslednjih klicev te funkcije je sintaktično pravilen?

(a) `abs();`

(b) `abs(-3.6);`

(c) `vredn = abs();`

(d) Noben od naštetih.

20. Podan je prototip funkcije `float abs(float x);` Kateri od naslednjih klicev te funkcije je sintaktično NAPAČEN?

(a) `abs(-3.8);`

(b) `abs(-3.6) + 4.8;`

(c) `abs(2);`

(d) `vredn = abs();;`

21. Vsaka od funkcij `f1()` in `f2()` ima svojo lokalno spremenljivko x. Katera od obeh spremenljivk obstaja v času izvajanja funkcije `f2()`, če vemo, da je to funkcijo klicala funkcija `f1()`?

(a) Obstajata obe spremenljivki.

(b) Ne obstaja nobena od obeh spremenljivk.

(c) Obstaja lokalna spremenljivka funkcije `f1()`.

(d) Obstaja lokalna spremenljivka funkcije `f2()`.

22. Vsaka od funkcij `f1()` in `f2()` ima svojo lokalno spremenljivko x. Katera od obeh spremenljivk je dostopna v času izvajanja funkcije `f2()`, če vemo, da je to funkcijo klicala funkcija `f1()`?

(a) Dostopni sta obe spremenljivki.

(b) Dostopna ni nobena od obeh spremenljivk.

(c) Dostopna je lokalna spremenljivka funkcije `f1()`.

(d) Dostopna je lokalna spremenljivka funkcije `f2()`.

3 Zbirke

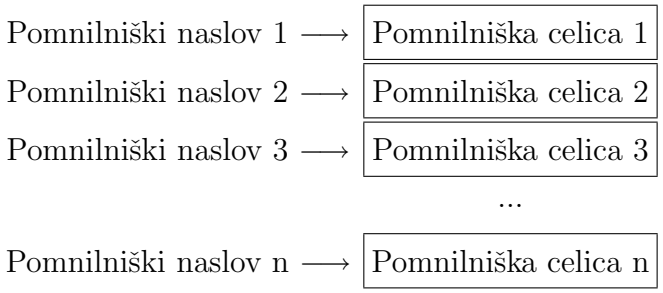
Tudi zbirke (array) smo že spoznali v jeziku JavaScript. Kljub temu, da je njihova uporaba v Ceju precej podobna, pa med obema jezikoma obstajata dve bistveni razliki:

1. V jeziku C zbirka ni objekt, temveč je skupek neodvisnih elementov. Tako brez poznavanja mehanizmov dinamične alokacije pomnilnika zbirki ni možno dodajati novih elementov. Potrebno število elementov moramo določiti že ob deklaraciji.

2. Vsi elementi zbirke morajo pripadati istemu podatkovnemu tipu.

3.1 Pomnilnik

Za razumevanje zbirk je potrebno imeti vsaj približno predstavo o tem, kako so podatki v računalniku shranjeni. Podatke hranimo v *pomnilniku*, katerega shematično zgradbo prikazuje naslednja slika.



Pomnilnik je sestavljen iz vrste *pomnilniških celic*, od katerih lahko vsaka hrani podatek velikosti 1 bajt. 1 bajt predstavlja 8 bitov, t.j. osem mestno dvojiško vrednost. Podrobneje o dvojiških zapisih podatkov bomo govorili v poglavju o kodiranju. Celice ločimo med sabo po *pomnilniških naslovih*. Pomnilniški naslov je v resnici nepredznačeno celo število in vsaka naslednja celica ima naslov, ki je za ena višji od njene predhodnice. Več o tem v poglavju o kazalcih.

3.2 Splošne zbirke

Zbirko v jeziku C deklariramo na enega od treh načinov, v vsakem pa moramo povedati, kakšnega tipa bodo elementi (vsi elementi so istega tipa), in koliko jih bo.

Takole ustvarimo zbirko n elementov tipa `tip_elementa`:

```
tip_elementa ime_zbirke[n];
```

Vseh n elementov takšne zbirke dobi poljubne vrednosti, če je zbirka deklarirana kot lokalna spremenljivka. Če gre za globalno zbirko, potem se vsi elementi postavijo na nič.

Na primer, zbirko petstotih elementov celoštevilskega tipa ustvarimo takole:

```
int zbirka1[500];
```

Elemente zbirke lahko ob deklaraciji tudi inicializiramo:

```
tip_elementa ime_zbirke[n] = {vredn_1, vredn_2, ... vredn_m};
```

Tako smo prvim m elementom zbirke po vrsti priredili vrednosti od `vredn_1` do `vredn_m`. Število vrednosti ne sme biti večje od števila elementov zbirke:

$$m \leq n$$

Elementi, ki jim vrednosti nismo priredili, bodo v vsakem primeru dobili vrednosti 0.

Takole usvarimo zbirko 500 elementov celoštevilskega tipa, kjer prvim petim elementom priredimo vrednosti od 10 do 14:

```
int zbirka2[500] = {10, 11, 12, 13, 14};
```

Ostalih 495 elemetov dobi vrednost 0.

Tretji način deklaracije zbirke je takšen:

```
tip_elementa ime_zbirke[] = {vredn_1, vredn_2, ... vredn_n};
```

Tako ustvarimo zbirko z n elementi, ki smo jim po vrsti priredili vrednosti od `vredn_1` do `vredn_n`.

Na primer, zbirko s 3 realnimi elementi, ki imajo vrednosti 3, 2 in 1, ustvarimo takole:

```
float zbirka3[] = {3, 2, 1};
```

Ko je zbirka enkrat deklarirana, jo uporabljamo na podoben način kot smo bili navajeni v jeziku JavaScript. Ime zbirke, skupaj z zaporedno številko elementa (indeksom) v oglatih oklepajih, se obnaša kot običajna spremenljivka. Paziti moramo le, da z indeksom ne gremo preko meja, ki smo jih v deklaraciji določili. Če imamo zbirko z n elementi, potem imajo lahko indeksi vrednosti med 0 in $n - 1$.

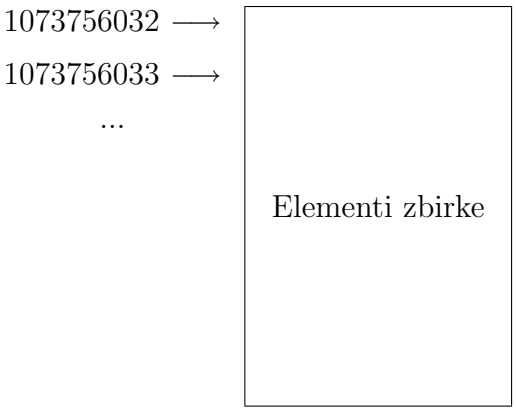
Peti element zbirke `zbirka2`, na primer, bi povečali za ena takole:

```
zbirka2[4]++;
```

Glede na to, da sama zbirka ni objekt, nastopi zanimivo vprašanje, kaj pomeni ime zbirke brez indeksa in oglatih oklepajev. V resnici predstavlja to ime pomnilniški naslov, na katerem se začne kos pomnilnika, kjer je shranjena zbirka. Poskusimo s funkcijo `printf()` izpisati vrednost izraza `zbirka2`. Ker je pomnilniški naslov nepredznačeno celo število, uporabimo formatno določilo `%u`:

```
printf("%u", zbirka2);
```

Ko sem sam napisal kratek program, v katerem sem deklariral zbirko in izpisal njen naslov na prikazovalnik, sem dobil vrednost 1073756032. To je naslov, na katerem se začne kos pomnilnika, kjer so shranjeni elementi zbirke. V pomnilniku imamo takšno stanje:



Ko boste zadevo poskusili sami, boste verjetno dobili drugačen naslov. Bistveno pri tem je, da se ta naslov, kakršenkoli že je, ves čas izvajanja programa ne spremeni.

3.3 Znakovni nizi

Znakovni niz je samo poseben primer zbirke. Zanj je predpisano, da so elementi tipa `char`, in da ima zadnji element vrednost 0. Temu zadnjemu elementu rečemo tudi *zaključni ničelni znak* (terminating null character). Njegova naloga je, da označuje konec znakovnega niza.

Znakovni niz lahko po pravilih iz prejšnjega razdelka deklariramo takole:

```
char geslo[] = {'b', 'a', 'n', 'a', 'n', 'a', 0};
```

Opozorim naj na zaključni ničelni znak, ki ni v enojnih navednicah. To namreč ni znak `'0'`, ki ima kodo 48, ampak je številska vrednost 0. Včasih boste za tak znak videli tudi zapis `'\0'`. Ker je znakovni niz v resnici besedilo, je takšen način deklaracije nekoliko neroden. V praksi se uporablja poenostavljen zapis:

```
char geslo[] = "banana";
```

Znakovni niz pišemo v dvojnih navednicah in brez vmesnih vejic. Prav tako nismo dodali zaključnega ničelnega znaka, kajti prevajalnik ga doda sam od sebe.

Če želimo izpisati besedilo na prikazovalnik, uporabimo funkcijo `printf()`, kot to že znamo:

```
printf(geslo);
```

Funkciji `printf()` v resnici podamo pomnilniški naslov, na katerem se besedilo nahaja. Funkcija potem na prikazovalnik izpiše znake enega za drugim, dokler ne pride do zaključnega ničelnega znaka.

Za izpis lahko uporabimo tudi formatno določilo `%s`:

```
printf("Geslo je: %s", geslo);
```

Čeprav funkcija za izpis besedila že obstaja, si bomo za primer napisali svojo. Funkcijo bomo poimenovali `putstring()` in bo kot parameter sprejela znakovni niz, ki ga bo potem izpisala na prikazovalnik.

Takole izgleda kompleten program:

```
#include "io.h"

void putstring(char *besedilo);

int main(void)
{
  char geslo[] = "banana";
  _LCDInit();
  putstring(geslo);
  while (1);
  return 0;
}

void putstring(char *besedilo)
{
  int i;
  for (i = 0; besedilo[i]; i++)
  {
    putch(besedilo[i]);
  }
}
```

Pred imenom parametra funkcije `putstring()` je zvezdica, kar pomeni, da je parameter pomnilniški naslov. Dokler ne bomo prišli do poglavja o kazalcih, vam več ni treba vedeti.

Glavni del funkcije `putstring()` je zanka `for`, ki poskrbi, da funkcija `putch()` na prikazovalnik enega po enega izpiše vse znake besedila. Začnemo s prvim znakom, ki ima indeks 0, zanka pa se ustavi, ko pride do zaključnega ničelnega znaka. Takrat namreč vrednost izraza `besedilo[i]` prvič ni več različna od nič.

Naslednji primer se navezuje na problem, ki smo ga v razdelku Znakovni tip delno že rešili. Ugotovili smo, kako s pomočjo dveh tipk prikazati na prikazovalnik poljuben znak. Z uporabo znanja o znakovnih nizih lahko sedaj sestavimo funkcijo, ki nam bo omogočala, da vpišemo in

tudi shranimo v pomnilnik poljubno besedilo. Funkcija bo delovala tako, da se bo ob pritisku na katero od tipk T0 ali T1 spremenil zadnji od prikazanih znakov. Ko bo uporabnik prišel do želenega znaka, bo s pritiskom na tipko T2 na koncu dodal nov znak, ki ga bo spet lahko spreminjal s tipkama T0 in T1. Pritisk na tipko T3 bo končal izvajanje funkcije. Tule je primer programa s takšno funkcijo:

```
#include "io.h"

void getstring(char *t);

int main(void)
{
  char ime[20];
  _LCDInit();
  _KeyInit();
  _setcursortype(_NOCURSOR);
  printf("Kako ti je ime?\r\n");
  getstring(ime);
  printf("\rZdravo, %s!", ime);
  while (1);
  return 0;
}

void getstring(char *t)
{
  char tipka = 0;
  char znak = 'a';
  int i = 0;

  while (tipka != '3')
  {
    t[i] = znak;
    t[i + 1] = 0;
    putch('\r');
    printf(t);
    tipka = getch();

    switch (tipka)
    {
      case '0': znak--; break;
      case '1': znak++; break;
      case '2': i++;
    }
```

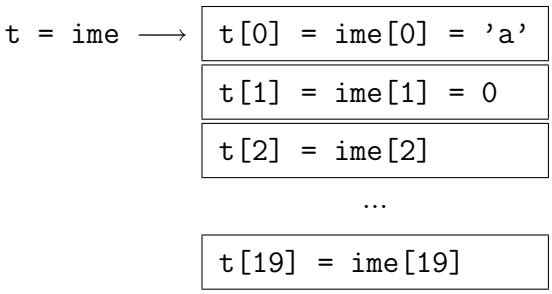
```
  }
}
```

V funkciji `main()` se ne dogaja nič posebnega. Na začetku smo deklarirali znakovni niz *ime* dolžine 20 elementov, v katerega bomo lahko shranili največ 19 znakov in zaključni ničelni znak. Po inicializaciji tipk in prikazovalnika se izpiše poziv, naj uporabnik vnese svoje ime. Funkcija `getstring()` bo tekla toliko časa, dokler ne bo uporabnik vnesel svojega imena in pritisnil tipke T3. Takoj zatem se bo na prikazovalniku pojavil pozdrav.

Funkcija `getstring()` sprejme en sam parameter, ki je pomnilniški naslov znakovnega niza. V našem primeru smo funkciji podali naslov znakovnega niza *ime*, saj želimo, da funkcija ravno v ta znakovni niz vpiše besedilo, ki ga bomo vtipkali.

Mehanizem podajanja naslovov spada v področje kazalcev, ki se ga bomo lotili proti koncu semestra. Za zdaj je pomembno le to, da razumemo, da se ob klicu funkcije `getstring()` v parameter t vpiše pomnilniški naslov znakovnega niza *ime*. Tako je spremenljivka t v našem primeru sinonim za znakovni niz *ime*.

Glavni del funkcije je zanka `while`, ki se bo izvajala, dokler ne bomo pritisnili tipke T3. Ker je *tipka* na začetku enaka 0 (torej različna od '3'), se zanka začne izvajati, in v prvih dveh vrsticah v znakovni niz vpišemo črko 'a' (vrednost spremenljivke *znak*) in zaključni ničelni znak. Ker je i enak nič, se ta dva znaka vpišeta na prvo in drugo mesto v znakovnem nizu:



Nastal je znakovni niz, ki vsebuje en sam znak, ki pa mora biti kljub temu zaključen z zaključnim ničelnim znakom. Samo tako si lahko zagotovimo, da ga bo funkcija `printf()` pravilno izpisala.

Če zdaj pritisnemo katero od tipk T0 ali T1, se znak spremeni. Ker se i ni spremenil, se na prvem mestu v nizu zdaj znajde drug znak. Ko smo z izbranim znakom zadovoljni, pritisnemo tipko T2 in i se poveča za ena. Zdaj prvega znaka ne moremo več spreminjati. Vsi nadaljnji pritiski tipk T0 in T1 bodo odslej vplivali na drugi znak v nizu.

Pomanjkljivost naše funkcije `getstring()` je ta, da v primeru, da se zmotimo, ne moremo več nazaj. Glede na to, da imamo na voljo le štiri tipke, lahko to pomanjkljivost odpravimo le z nekoliko domišljije. Ena možnost je, da se odrečemo spreminjanju znaka v obe smeri. Znake lahko spreminjamo z eno samo tipko, če funkcijo popravimo tako, da bo to spreminjanje ciklično (takoj za 'z' pride spet 'a'). Naslednji problem nastopi, če si zaželimo tudi tipke, ki bi preklapljala med velikimi in malimi črkami. Z nekoliko znanja o uri lahko ločimo enojen in dvojen pritisk tipke, podobno kot ima lahko dvoklik miške drugačen učinek kot klik.

Če nam ni škoda nekaj dodatnih tolarjev, pa si lahko ves trud prihranimo z nakupom dodatne tipke. Kako takšno tipko priklopiti na sistem, si bomo ogledali v naslednjem poglavju.

3.4 Primeri izpitnih vprašanj

1. Podana je deklaracija zbirke `int x[10]={2,3};` Koliko elementov ima ta zbirka?
   (a) 2
   (b) 3
   (c) 9
   (d) 10
2. Podana je deklaracija zbirke `int x[10]={2,3};` Kakšna je vrednost elementa `x[2]`?
   (a) 0
   (b) 2
   (c) 3
   (d) Vrednosti ne moremo določiti.
3. Podana je deklaracija zbirke `int x[10];` Kakšna je vrednost elementa `x[2]`?
   (a) 0
   (b) 2
   (c) 3
   (d) Vrednosti ne moremo določiti.
4. Podana je deklaracija zbirke `int x[]={2,3};` Koliko elementov ima ta zbirka?
   (a) 0
   (b) 2
   (c) 3
   (d) Števila elementov ne moremo določiti.
5. Podana je deklaracija `char x[]="a";` Koliko elementov ima znakovni niz `x`?
   (a) 0
   (b) 1
   (c) 2
   (d) Števila elementov ni mogoče določiti.
6. Kako deklariramo znakovni niz `x` in vanj vpišemo besedilo, ki je sestavljeno iz enega samega znaka `'a'`?

(a) `char x[1] = 'a';`

(b) `char x[] = "a";`

(c) `char x[] = 'a';`

(d) `char x[] = {'a'};`

7. Kako deklariramo znakovni niz `x` in vanj vpišemo besedilo, ki je sestavljeno iz enega samega znaka `'a'`?

(a) `char x[] = 'a';`

(b) `char x[1] = "a";`

(c) `char x[2] = {'a', '0'};`

(d) `char x[] = {'a', 0};`

8. V znakovnem nizu `test` imamo shranjeno besedilo `"abcd"`. Kaj se bo izpisalo na prikazovalniku, ko se izvede naslednji del kode?

```
test[1] = 0;
printf(test);
```

(a) `test`

(b) `abcd`

(c) `t`

(d) `a`

9. V znakovnem nizu `test` imamo shranjeno besedilo `"abcd"`. Kaj se bo izpisalo na prikazovalniku, ko se izvede naslednji del kode?

```
for (i = 0; test[i]; i++)
  if (test[i] == 'a') putch('A');
```

(a) `AAAA`

(b) `Abcd`

(c) `ABCD`

(d) `A`

10. V znakovnem nizu `test` imamo shranjeno besedilo `"abcd"`. Kaj se bo izpisalo na prikazovalniku, ko se izvede naslednji del kode?

```
for (i = 0; test[i]; i++)
  if (test[i] == 'a') test[i] = 'A';
printf(test);
```

(a) `test`

(b) `Abcd`

(c) `ABCD`

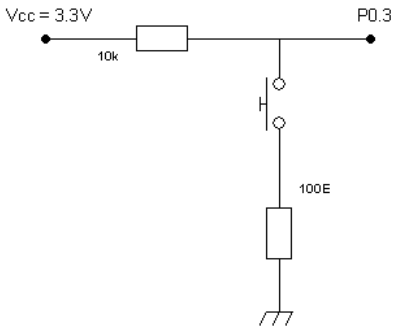
(d) `A`

4 Priklop zunanjih naprav

Mikrokrmilnik, pa naj bo še tako zmogljiv, nam nič ne koristi, če nanj ne znamo priklopiti kakšne naprave. V najosnovnejši obliki je ta naprava lahko enostavna tipka ali dioda LED. Nekaj naprav je na sistemu Š-ARM že priklopljenih in napisane so tudi knjižnice funkcij, s katerimi jih zelo enostavno izkoriščamo. Spoznali smo že prikazovalnik LCD in tipke. Čas je, da se nekoliko opogumimo in poskusimo kakšno stvar še sami priklopiti. Začeli bomo z enostavno tipko.

4.1 Tipka

Naslednja slika prikazuje električno shemo priklopa tipke na vrata (port) P0.3 našega mikrokrmilnika:



Vrata niso nič drugega kot ena sama sponka, prek katere lahko v mikrokrmilnik ali iz njega prenesemo en bit informacije, t.j. logično enko oziroma ničlo. V primeru tipke bomo informacijo prenašali iz zunanjega sveta v mikrokrmilnik. Iz gornje sheme je razvidno, da se bo v primeru, ko je tipka spuščena, na vratih pojavila napajalna napetost Vcc = 3,3V. Zanka med napajalno napetostjo in zemljo je namreč razprta in na uporu ni nobenega padca napetosti. Ko tipko pritisnemo, čez upora steče tok in na vratih se pojavi napetost zelo blizu ničle, napetost 3,3V se namreč razdeli v razmerju 1:100. Znotraj mikrokrmilnika se napetost blizu 3,3V tolmači kot logična enica, napetost blizu ničle pa kot logična ničla.

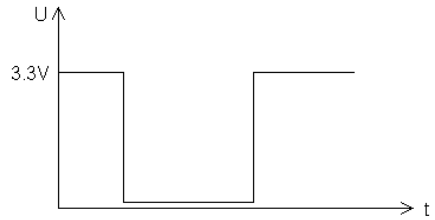
Naslednji program bo na prikazovalnik prikazoval enico, če bo tipka spuščena in ničlo, če bo tipka pritisnjena:

```
#include "io.h"

int main(void)
{
  _LCDInit();
  _setpindir(3, 0);
  _setcursortype(_NOCURSOR);

  while (1)
  {
    printf("%d\r", inportp(3));
  }
  return 0;
}
```

Program je sila preprost. Najprej moramo določiti vrata P0.3 kot vhodna. To storimo s klicem funkcije `_setpindir(3, 0)`. Prvi parameter pove zaporedno številko vrat oziroma sponke (pin), katere smer določamo. Drugi parameter pove smer pretoka informacije. Ničla predstavlja vhod podatka v mikrokrmilnik, enka pa izhod podatka iz mikrokrmilnika. Klic funkcije `inportp()` vrne vrednost, ki je trenutno navzoča na sponki. Ta vrednost je lahko 0 ali 1.

Namesto tipke lahko na sponko priklopimo nešteto drugih naprav. Na primer svetlobno tipalo, na katerega pada laserski snop. Ko se med laserjem in senzorjam pojavi kak objekt, se laserski snop prekine. Posledica je sprememba napetosti na tipalu, kar se takoj odrazi tudi v našem programu. Na ta način lahko zgradimo številne naprave, kot so na primer števci, alarmni sistemi ali merilniki hitrosti.

Ostanimo zaenkrat pri tipki in si zamislimo, kako bi napisali program, ki bo štel število pritiskov. Idealen potek napetosti na sponki, če pritisnemo in potem spet spustimo tipko, kaže naslednja slika:



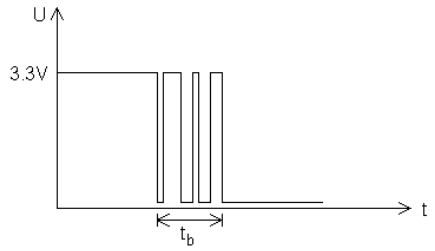
Vsaka sprememba v stanju tipke se odrazi tudi v spremembi napetosti. Ker nas zanima le število pritiskov, bomo šteli prehode iz enke v ničlo. Potrebujemo tri spremenljivke: *stevec* šteje število pritiskov, *stanje* hrani zadnje prebrano stanje tipke, in *staro* hrani prejšnje stanje tipke. To je potrebno hraniti, sicer ne bi mogli ugotoviti, ali je prišlo do spremembe. Takole izgleda program:

```
#include "io.h"
```

```
int main(void)
{
  int stanje;
  int staro = 1;
  int stevec = 0;
  _TimerInit();
  _LCDInit();
  _setpindir(3, 0);
  _setcursortype(_NOCURSOR);

  while (1)
  {
    stanje = inportp(3);
    if (staro != stanje)
    {
      staro = stanje;
      if (stanje == 0)
      {
        stevec++;
      }
      //delay(50);
    }
    if (stevec % 10 == 0)
    {
      printf("%d\r", stevec);
    }
  }
  return 0;
}
```

Za nas je zanimiva le situacija, ko se stanje spremeni (`staro != stanje`). Najprej popravimo staro stanje (stavek `staro = stanje;`), in potem, če je novo stanje enako nič, povečamo števec za ena. Namreč le v tem primeru gre za pritisk tipke. Če je novo stanje enako ena, potem smo tipko spustili, kar pa nas ne zanima. Za kontrolo na vsakih 10 pritiskov izpišemo stanje števca na prikazovalnik. Začuda, ko program poženemo, ugotovimo, da že ob veliko manj kot 10 pritiskih števec naraste na 10.

Zaman iščemo napako v programu, ker je ni. Napaka je posledica fizikalnega pojava, ki se pripeti zaradi neidealnosti električnega kontakta tipke. Še preden se kontakt popolnoma sklene, pride do nekajkratnega preskoka napetosti, čemur strokovno rečemo poskakovanje (bouncing). V visokonapetostnih stikalih se ta pojav kaže kot iskrenje. Približen potek napetosti med poskakovanjem kaže naslednja slika:

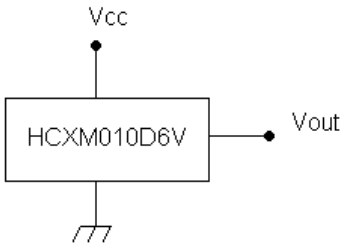


Čas poskakovanja ob pritisku tipke označimo s t_b. Ta čas je odvisen od kvalitete tipke in je pri kakovostnih tipkah lahko zelo majhen. Za tipko, ki sem jo uporabil sam, se je izkazalo, da je ta čas veliko manjši od 50ms. V programu sem moral zato ob vsaki spremembi stanja počakati vsaj 50ms, da se je poskakovanje zagotovo končalo. Čakanje lahko dosežemo s funkcijo `delay()`, ki zakasni delovanje programa za podano število milisekund. V gornjem primeru je klic začasno onemogočen z dvema poševnicama, da je možno opazovati učinek poskakovanja.

4.2 Tipalo za tlak

Gumb je naprava iz katere dobimo zgolj binarno informacijo v obliki dveh stanj, kot na primer nič in ena, izklopljen in vklopljen, napačen in pravilen, nizek in visok, in podobno. Takšne naprave priklopimo na vrata, ki so prav tako sposobna prenašati le binarne informacije. Kaj pa, če imamo opravka s celim spektrom različnih vrednosti, ki bi jih želeli prenašati med računalnikom in zunanjim svetom?

Vzemimo na primer tipalo za tlak, ki na eni strani tipa tlak, na drugi strani pa oddaja napetost, ki je neposredna funkcija izmerjenega tlaka. Uporabili bomo tipalo z oznako HCXM010D6V, katerega merilno območje je od 0 do 10 milibarov. Izmerjeni tlak tipalo pretvori v napetost med 0 in 3,3 V. To napetost izmerimo na sponki Vout. Na naslednji sliki vidimo simbolni prikaz našega tipala.



Tipalu moramo zagotoviti ustrezno napajanje (na sliki vezje, ki skrbi za to, ni prikazano) in že lahko na sponki Vout odčitavamo napetost in s tem posredno tlak. Kako zdaj to napetost, ki ima lahko kakršnokoli vrednost med 0 in 3,3V prenesemo v računalnik? Za to poskrbi tako imenovani analogno digitalni pretvornik (analogue-digital converter - ADC), ki je del mikrokrmilnika ARM7. Ta pretvornik pretvarja (analogno) napetost med 0 in Vcc (= 3,3V) v celoštevilsko vrednost med 0 in 1023. Vhod analogno digitalnega pretvornika je priključen na sponko P0.27 in na to sponko priključimo sponko Vout tipala za tlak. Če nimamo pri roki nobenega tipala, lahko na sistem Š-ARM nataknemo jumper J20, s čimer na vhod analogno digitalnega pretvornika priklopimo

potenciometer, ki je na plošči. Vrednost, ki se pojavi na izhodu pretvornika, odčitamo s funkcijo `_adconvert()`.

Merjenje tlaka poteka torej v dveh korakih. Najprej tipalo pretvori tlak v zvezno napetost, potem pa pretvornik AD to napetost pretvori v diskretno celoštevilsko vrednost, primerno za obdelavo v digitalnem računalniku.

Naslednji program na prikazovalnik prikazuje vrednosti med 0 in 102, odvisno od trenutne napetosti na vhodni sponki pretvornika AD:

```
#include "io.h"

int main(void)
{
  _LCDInit();
  _ADCInit();
  _setcursortype(_NOCURSOR);

  while(1)
  {
    printf("\r%4d", _adconvert() / 10);
  }
  return 0;
}
```

Program verjetno ne potrebuje kakšne posebne razlage. Opozorim naj le na to, da je potrebno analogno digitalni pretvornik pred uporabo inicializirati (`_ADCInit()`). Zaradi različnih neidealnosti zadnja cifra vrednosti, ki jo odčitamo na pretvorniku, včasih poskakuje, čeprav se merjeni tlak v resnici ne spreminja. Deljenje z 10 to poskakovanje nekoliko omili.

Gornji program lahko uporabimo za merjenje nivoja tekočine v rezervoarju. Glede na to, da ima uporabljeni senzor merilno območje do 10 milibarov, lahko merimo nivoje do 10 cm. Bolj kot nivo tekočine, pa nas običajno zanima volumen tekočine v rezervoarju. Če je rezervoar pravilen kvader, ali pokonci postavljen valj, ali kakšno drugo telo, ki se mu površina vodoravnega prereza z višino ne spreminja, potem je pretvorba iz tlaka v volumen stvar preprostega množenja z določeno konstanto. Če pa temu ni tako, moramo vrednosti tabelirati. Gornji primer programa nam na prikazovalniku kaže vrednosti med 0 in 102, kar predstavlja 103 različne odčitke. Za vsakega od teh možnih odčitkov enostavno vnesemo volumen tekočine, ki temu odčitku ustreza. Posamezne volumne vnesemo v zbirko, pri čemer predstavlja indeks elementa vrednost odčitka na tipalu, vrednost elementa pa volumen tekočine, ki ustreza temu odčitku:

```
float ml[103] = {0, 2.3, 4.7, 7.1, 9.6, ..., 500}; //polna posoda ima pol litra
```

Zdaj lahko volumen v mililitrih izpišemo takole:

```
printf("\r%.1fml", ml[_adconvert() / 10]);
```

Vzemimo za primer, da je nivo tekočine takšen, da funkcija `_adconvert()` vrne vrednost 21. To delimo z 10 in ker gre za celoštevilsko deljenje, dobimo 2. Na prikazovalnik se torej izpiše vrednost izraza `ml[2]`, ki je 4.7, z dodano besedico "ml":

```
4.7ml
```

4.3 Primeri izpitnih vprašanj

1. Vrata smo konfigurirali kot vhodna. Kakšno vrednost preberemo z njih, če je na vhodni sponki napetost 3V?

   (a) 0

   (b) 1

   (c) 3

   (d) 3,3

2. Imamo analogno digitalni pretvornik, ki območje vhodnih napetosti med 0 in 3,3V pretvarja v celoštevilske vrednosti med 0 in 1023. V kakšno vrednost se bo pretvorila napetost 1V?

   (a) 1

   (b) 10

   (c) 100

   (d) 310

3. Imamo analogno digitalni pretvornik, ki območje vhodnih napetosti med 0 in 3,3V pretvarja v celoštevilske vrednosti med 0 in 1023. Pretvorjeno vrednost preberemo s klicem funkcije `_adconvert()`. Kateri od naslednjih izrazov bo vrnil vrednost 0, dokler je napetost manjša ali enaka 3V, in vrednost 1, ko napetost preseže 3V?

   (a) `_adconvert() > 3 ?  0 :  1`

   (b) `_adconvert() > 930 ?  0 :  1`

   (c) `_adconvert() > 3 ?  1 :  0`

   (d) `_adconvert() > 930 ?  1 :  0`

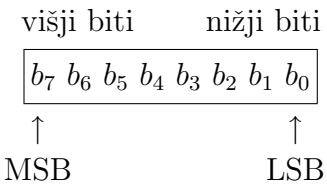
5 Kodiranje podatkov

Kadar govorimo o kodiranju podatkov, običajno mislimo na zapisovanje informacije na načine, ki se razlikujejo od nam naravnega. Za nas je naraven način zapisovanja črk latinica, medtem ko za števila uporabljamo arabske cifre in desetiški sistem. V različnih okoliščinah so ti za nas naravni zapisi neprimerni, zato smo si ljudje izmislili različne načine kodiranja. Kot primer lahko vzamemo črtno kodo, ki jo najdemo na vseh prodajnih izdelkih, kjer s kombinacijami različno

debelih črt kodiramo desetiške cifre. Tak način kodiranja je zelo prikladen, ker omogoča, da je izdelek v trenutku odčitavanja kode v različnih legah. Primer kodiranja je tudi barvno zapisovanje številskih vrednosti, uporabljeno za označevanje upornosti in toleranc električnih uporov. Vsi poznamo Morsejevo kodo, kjer so črke, cifre in še nekateri posebni znaki kodirani z nizi pik in črt. Tak način kodiranja je primeren za prenos sporočil s preprostimi zvočnimi in svetlobnimi napravami.

5.1 Dvojiško kodiranje

Digitalni računalniki hranijo in obdelujejo podatke v binarni oziroma dvojiški obliki, predstavljeni z visokimi in nizkimi napetostnimi nivoji. Zaradi tega moramo podatke, preden bodo lahko ti kakorkoli uporabni za digitalni računalnik, prestaviti v dvojiški zapis, t.j. v nize enic, ki predstavljajo visok, in ničel, ki predstavljajo nizek napetostni nivo. Takšni predstavitvi pravimo *pozitivna logika*. V primeru *negativne logike* je enica predstavljena z nizkim in ničla z visokim nivojem. Najmanjši enoti dvojiškega zapisa pravimo *bit*. Bit ima lahko vrednost 1 ali 0. Pravimo, da je bit, ki ima vrednost 1, *postavljen* (set), bit, ki ima vrednost 0, pa *pobrisan* (clear). Iz praktičnih razlogov, vezanih na zgradbo večine digitalnih računalnikov, združujemo bite v skupine po osem in takšno skupino poimenujemo *bajt* (byte). Pogosto imamo opravka tudi s podatki dolžine dveh bajtov. Dvema bajtoma skupaj pravimo *beseda* (angl. word). Posamezne bite znotraj bajta ali besede označimo z zaporednimi številkami, pri čemer številka predstavlja *težo* oziroma *pomembnost* bita, ki narašča od desne proti levi. Tako bitom proti levi pravimo tudi *višji*, bitom proti desni pa *nižji biti*. Skrajno levemu bitu pravimo najpomembnejši oziroma MSB (Most Significant Bit) bit, skrajno desnemu pa najmanj pomemben oziroma LSB (Least Significant Bit) bit. Naslednja slika prikazuje te oznake za primer bajta. Vidimo, da bitu LSB ustreza bit b_0, bit MSB pa bitu b_7.



5.2 Pozicijski številski sistemi

Tradicionalen številski sistem, ki smo se ga učili v šoli in ga uporabljamo v vsakodnevnem življenju, spada med *pozicijske številske sisteme*. V takšnem sistemu predstavimo število kot niz cifer, pri čemer položajem cifer pripišemo ustrezne *uteži*. Vrednost števila je utežna vsota posameznih cifer, na primer:

$$103,93 = 1 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0 + 9 \cdot 10^{-1} + 3 \cdot 10^{-2}$$

Vsaka utež je potenca števila 10, ki ustreza položaju cifre. Cifre na levi strani decimalne vejice imajo uteži z nenegativnimi potencami, cifre na desni strani pa imajo uteži z negativnimi

potencami.

V splošnem zapišemo število D, ki ima p cifer na levi in n cifer na desni strani decimalne vejice, v obliki

$$d_{p-1}d_{p-2}\cdots d_0, d_{-1}d_{-2}\cdots d_{-n}$$

njegovo vrednost pa izračunamo kot vsoto

$$D = \sum_{i=-n}^{p-1} d_i r^i \tag{5}$$

Pri tem je r osnova številskega sistema, d_i pa predstavlja posamezno cifro (digit) številskega zapisa.

5.3 Dvojiški zapis nepredznačenih celih števil

Vzemimo splošno enačbo (5) ter postavimo za osnovo $r = 2$ in število mest za decimalno vejico $n = 0$. Namesto cifer vzememo bite in jih označimo z b_i. Dobimo enačbo, po kateri izračunamo vrednost nepredznačenega (unsigned) celega števila v dvojiškem zapisu:

$$B = \sum_{i=0}^{p-1} b_i 2^i \tag{6}$$

Vzemimo za primer dvojiško vrednost 10011. Po enačbi (6) izračunamo pripadajočo desetiško vrednost

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

5.3.1 Območja vrednosti

V poglavju Deklaracije spremenljivk in podatkovni tipi smo že omenjali, da so vrednosti, ki jih s celoštevilskimi podatkovnimi tipi lahko zapisujemo, po velikosti omejene. Omejitve so bile podane z nenavadnimi vrednostmi, za katere pa imamo zdaj dovolj znanja, da jih lahko pojasnimo.

Spomnimo se, da se v digitalnem računalniku podatki hranijo v pomnilniku. Za vsak podatkovni tip je točno določeno, koliko prostora v pomnilniku zasede:

Podatkovni tip	**Odmerjen prostor v pomnilniku (v bitih)**
`char`	8
`short`	16
`int (long)`	32

Čim trčimo ob fizično omejitev, ki nam omejuje število bitov, ki so na voljo za zapis vrednosti, se soočimo tudi z omejitvijo vrednosti, ki jih lahko tako zapišemo. Zelo nazorno lahko to vidimo na primeru avtomobilskega števca kilometrov. V vseh praktičnih primerih je seveda 6-mesten števec dovolj. Če pa bi nekdo z avtomobilom prevozil milijon kilometrov ali več, števec ne bi več kazal pravilno. Vsi naslednji odbirki bi bili točno za milijon premajhni. Največja vrednost, ki jo lahko zapišemo kot 6-mestno desetiško število, je $10^6 - 1$ oziroma 999999.

Prestavimo to razmišljanje v naš dvojiški svet in po krajšem razmisleku ugotovimo, da je največja vrednost, ki jo lahko zapišemo z n biti, enaka $2^n - 1$. Od tod torej pridejo največje možne vrednosti za podatkovne tipe `unsigned char`, `unsigned short` in `unsigned long`, ki so 255, 65535 in 4294967295.

Razmišljanje o avtomobilskem števcu nas pripelje tudi do naslednjih dveh pomembnih zaključkov v zvezi s seštevanjem in odštevanjem dveh n-bitnih nepredznačenih vrednosti:

1. Če je pravilna vsota večja ali enaka 2^n, potem bo dejanska vsota za 2^n premajhna.
2. Če je pravilna razlika manjša od nič, potem bo dejanska razlika za 2^n prevelika.

Oglejmo si dva primera, ki sprašujeta, kolikšna je dejanska vsota (razlika) dveh števil tipa `unsigned short`.

1. **60000 + 10000** = 70000 - 65536 = **4464**
2. **3 - 5** = -2 + 65536 = **65534**

V prvem primeru je vsota 70000 presegla zgornjo mejo, ki jo lahko zapišemo s 16 biti. Zato je dejanski rezultat za $2^{16} = 65536$ premajhen.

V drugem primeru je vsota -2 presegla spodnjo mejo, ki jo lahko zapišemo s 16 biti. Zato je dejanski rezultat za $2^{16} = 65536$ prevelik.

Oba primera lahko preizkusite tudi z računalnikom:

```
//...
unsigned short x = 60000;
x += 10000;
printf("%u\r\n", x); //izpise 4464
x = 3;
x -= 5;
printf("%u", x); //izpise 65534
//...
```

5.4 Dvojiški zapis predznačenih celih števil

Predznačena cela števila zajemajo območje tako pozitivnih kot tudi negativnih vrednosti. Ker so negativne vrednosti zapisane v dvojiškem komplementu, si oglejmo najprej pojma *eniškega in dvojiškega komplementa*.

5.4.1 Eniški in dvojiški komplement

Vzemimo za primer osembitno vrednost 01011000 in negirajmo njene bite. Negacija bita napravi iz ničle enico in iz enice ničlo. Dobimo vrednost 10100111. Operaciji negacije posameznih bitov pravimo tudi *eniški komplement*. Zdaj dobljenemu eniškemu komplementu prištejmo 1 in dobimo 10101000, kar je *dvojiški komplement*. Dvojiški komplement ima zanimivo lastnost. Če ga

seštejemo z originalno vrednostjo, dobimo vsoto, ki je enaka nič:

$$\begin{array}{r} 01011000 \\ +\ 10101000 \\ \hline 1\ 00000000 \end{array}$$

Pri tolmačenju vsote moramo upoštevati dejstvo, da je število bitov, ki so v pomnilniku na voljo za določen podatek, omejeno. Če sta oba sumanda osembitna, potem seveda tudi rezultat ne more biti več kot osembiten. Dobljena enica v vsoti predstavlja že deveti bit, ki ga v resnici nimamo.

Matematično gledano je vsota dveh podatkov enaka nič, če sta nasprotnega predznaka. V našem svetu omejenih vrednosti predstavlja dvojiški komplement torej množenje z -1. Dogovorimo se še, da bit MSB predstavlja predznak podatka. Če je MSB enak nič, je vrednost nenegativna, sicer je negativna.

5.4.2 Območja vrednosti

Zdaj lahko na podoben način kot smo to storili za nepredznačena števila tudi za predznačena števila določimo območje vrednosti, ki jih lahko zapišemo z n biti. Največjo pozitivno vrednost dobimo z upoštevanjem dejstva, da mora biti bit MSB za nenegativna števila vedno enak nič. Za zapis nenegativnih vrednosti nam ostane torej $n-1$ bitov. Iz razdelka o nepredznačenih številih vemo, da je največja vrednost, ki jo lahko zapišemo z $n-1$ biti, enaka $2^{n-1}-1$. Ker moramo šteti tudi ničlo, ugotovimo, da lahko zapišemo vsega skupaj 2^{n-1} različnih nenegativnih vrednosti. Ker lahko z n biti zapišemo 2^n različnih kombinacij, 2^{n-1} pa smo jih že porabili za nenegativne vrednosti, nam za negativne vrednosti ostane še 2^{n-1} kombinacij.

Območje vrednosti za predznačena n-bitna cela števila je torej od -2^{n-1} do $2^{n-1}-1$.

Podobno kot pri nepredznačenih vrednostih tudi v zvezi s seštevanjem in odštevanjem dveh n-bitnih predznačenih vrednosti ugotovimo naslednje:

1. Če je pravilna vsota/razlika večja ali enaka 2^{n-1}, potem bo dejanska vsota/razlika za 2^n premajhna.
2. Če je pravilna vsota/razlika manjša od -2^{n-1}, potem bo dejanska vsota/razlika za 2^n prevelika.

Oglejmo si še dva primera, ki sprašujeta, kolikšna je dejanska vsota (razlika) dveh števil tipa `short`.

1. **32767 + 1** = 32768 - 65536 = **-32768**
2. **-32767 - 3** = -32770 + 65536 = **32766**

V prvem primeru je vsota 32768 presegla zgornjo mejo, ki jo lahko zapišemo s 16 biti. Zato je dejanski rezultat za $2^{16} = 65536$ premajhen.

V drugem primeru je vsota -32770 presegla spodnjo mejo, ki jo lahko zapišemo s 16 biti. Zato je dejanski rezultat za $2^{16} = 65536$ prevelik.

Preizkusite oba primera tudi z računalnikom.

5.5 Dvojiški zapis realnih števil

Čeprav so realna števila zapisana v *zapisu s plavajočo vejico* (floating point format), si bomo ogledali le *zapis z nepremično vejico* (fixed point format). Zapis s plavajočo vejico je namreč precej zapleten in presega okvire našega predmeta. Pojav, ki ga želim pokazati v tem razdelku, se da videti tudi na precej enostavnejšem primeru zapisa z nepremično vejico.

Zapis je v resnici običajen pozicijski zapis s p biti pred in n biti za vejico:

$$b_{p-1}b_{p-2}\cdots b_1b_0, b_{-1}b_{-2}\cdots b_{-n}$$

Za izračun desetiške vrednosti izhajamo iz enačbe (5), v kateri cifre (d_i) nadomestimo z biti (b_i), osnova pa je $r = 2$:

$$B = \sum_{i=-n}^{p-1} b_i 2^i \tag{7}$$

Za primer izračunajmo desetiško vrednost dvojiškega števila 10,011:

$$1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 2,375$$

Poglejmo si še obratno pot pretvorbe – iz desetiškega v dvojiški zapis. Pri tem se bomo osredotočili le na del, ki je za nas zanimiv, to pa je del desno od vejice. Zato v enačbi (7) postavimo p, ki predstavlja število mest pred vejico, na 0 in dobimo

$$B = \sum_{i=-n}^{-1} b_i 2^i, \tag{8}$$

kar lahko zapišemo tudi v gnezdeni obliki

$$B = (((b_{-n} \cdot 2^{-1} + b_{-(n-1)}) \cdot 2^{-1} + b_{-(n-2)}) \cdot 2^{-1} + \cdots b_{-1}) \cdot 2^{-1} \tag{9}$$

Če enačbo (9) množimo z dve, dobimo rezultat, katerega celi del je enak vrednosti bita b_{-1}, za vejico pa nam ostane vrednost

$$U = (((b_{-n} \cdot 2^{-1} + b_{-(n-1)}) \cdot 2^{-1} + b_{-(n-2)}) \cdot 2^{-1} + \cdots b_{-2}) \cdot 2^{-1} \tag{10}$$

To vrednost še naprej množimo z dve in tako se nam po vrsti razkrivajo biti $b_{-2}, b_{-3}, \cdots b_{-n}$.

Za primer pretvorimo desetiško vrednost 0,2 v dvojiški zapis z nepremično vejico:

$$\begin{aligned}
0{,}2 \cdot 2 &= 0 + 0{,}4 \\
0{,}4 \cdot 2 &= 0 + 0{,}8 \\
0{,}8 \cdot 2 &= 1 + 0{,}6 \\
0{,}6 \cdot 2 &= 1 + 0{,}2 \\
0{,}2 \cdot 2 &= 0 + 0{,}4 \\
0{,}4 \cdot 2 &= 0 + 0{,}8 \\
0{,}8 \cdot 2 &= 1 + 0{,}6 \\
0{,}6 \cdot 2 &= 1 + 0{,}2
\end{aligned}$$

Opazimo, da se vzorec ponavlja. Dobili smo *približek* desetiške vrednosti v dvojiškem zapisu, ki je 0,00110011. Desetiške vrednosti 0,2 torej ni mogoče pretvoriti v končni dvojiški zapis, ne da bi pri tem napravili napako. Posledica tega dejstva se kaže v nenavadnem obnašanju naslednjega dela programske kode:

```
//...
float x = 0.2;
if (x == 0.2)
{
  printf("x je enak 0.2");
}
else
{
  printf("x ni enak 0.2");
}
//...
```

Na prikazovalnik se izpiše sporočilo "x ni enak 0.2", kar je na prvi pogled čudno. Zdaj pa se spomnimo, da v jeziku C konstantna realna vrednost pripada tipu `double`. Ker so vrednosti tipa `double` zapisane z večimi decimalnimi mesti kot vrednosti tipa `float`, smo v gornji kodi primerjali dva različno dobra približka vrednosti 0,2, ki sta seveda različna.

5.6 Šestnajstiški zapis

Digitalni računalniki obdelujejo in hranijo podatke, ki jih je potrebno zapisati v dvojiškem zapisu. Tak zapis je zaradi velike količine ničel in enk za človeka pogosto nepregleden. Desetiški zapis je bolj strnjen in pregleden, ampak samo, če zapisujemo številske vrednosti. V ostalih primerih je postopek pretvarjanja med obema sistemoma prezapleten, da bi upravičil zapis z desetiško vrednostjo. Potebujemo nek sistem z osnovo, ki je potenca števila 2. Uveljavil se je šestnajstiški sistem.

Šestnajstiški sistem pozna šestnajst cifer, ki so desetiške cifre od 0 do 9 in črke med A in F. Vsaka od teh cifer nadomesti štiribitno dvojiško vrednost, kot to kaže naslednja tabela.

Dvojiško	**Šestnajstiško**	**Dvojiško**	**Šestnajstiško**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Pretvarjanje iz enega sistema v drugega je zgolj stvar nadomeščanja skupin štirih bitov z ustrezno cifro in obratno. Tako se, na primer, dvojiška vrednost 1000111101010000 v šestnajstiškem zapisu glasi 8F50.

V jeziku C zapisujemo konstantne šestnajstiške vrednosti v obliki `0xhhhh`, kjer je `h` poljubna šestnajstiška (hexadecimal) cifra. Za izpis šestnajstiških vrednosti na prikazovalnik uporabimo formatno določilo `%X` oziroma `%x`, če želimo v izpisu male tiskane črke med a in f.

Za primer si oglejmo enostaven program, ki povzroči, da sta diodi LD3 in LD2 ugasnjeni, dioda LD0 je prižgana in dioda LD1 utripa. Za prižiganje diod uporabimo funkcijo `_setleds()`. Funkcija sprejme štiribitno vrednost in prižge diode, ki ustrezajo položajem enic v podani vrednosti. Ostale diode pusti pri miru. Za ugašanje diod uporabimo funkcijo `_clrleds()`. Tudi ta funkcija sprejme štiribitno vrednost in uganse diode, ki ustrezajo položajem enic v podani vrednosti. Ostale diode pusti pri miru. Tule je program:

```
  #include "io.h"

  int main(void)
  {
    _LEDInit();
    _TimerInit();

    _clrleds(0xC); //ugasne prvi dve diodi (0xC = 1100)
    _setleds(0x3); //prizge zadnji dve diodi (0x3 = 0011)

    while (1)
    {
      _clrleds(0x2); //ugasne LD1 (0x2 = 0010)
      delay(500);
      _setleds(0x2); //prizge LD1 (0x2 = 0010)
      delay(500);
    }
    return 0;
}
```

5.7 Primeri izpitnih vprašanj

1. Katera je največja desetiška vrednost, ki jo lahko zapišemo kot 10-bitno nepredznačeno število?

   (a) 511

   (b) 512

   (c) 1023

   (d) 1024

2. Katera je največja desetiška vrednost, ki jo lahko zapišemo kot 10-bitno predznačeno število?

   (a) 511

   (b) 512

   (c) 1023

   (d) 1024

3. Katera je najmanjša desetiška vrednost, ki jo lahko zapišemo kot 10-bitno predznačeno število?

   (a) -511

   (b) -512

   (c) -1023

   (d) -1024

4. Spremenljivka z je tipa `unsigned char` in ima vrednost 255. Kolikšna bo njena vrednost, ko se izvede stavek `z++;`?

   (a) 0

   (b) 1

   (c) 255

   (d) 256

5. Spremenljivka z je tipa `unsigned char` in ima vrednost 0. Kolikšna bo njena vrednost, ko se izvede stavek `z--;`?

   (a) -1

   (b) 0

   (c) 255

   (d) 256

6. Spremenljivka z je tipa `char` in ima vrednost -128. Kolikšna bo njena vrednost, ko se izvede stavek `z--;`?

   (a) -129

   (b) -128

   (c) 127

   (d) 128

7. Spremenljivka z je tipa `unsigned short` in ima vrednost 3. Kolikšna bo njena vrednost, ko se izvede stavek `z -= 4;`?

(a) -1

(b) 0

(c) 65534

(d) 65535

8. Spremenljivke x, y in z so tipa `unsigned short` in spremenljivka x ima vrednost 60000. Ko se izvede stavek `z = x + y;` dobi spremenljivka z vrednost 10. Kolikšna je vrednost spremenljivke y?

   (a) -59990

   (b) 10

   (c) 5545

   (d) 5546

9. Spremenljivke x, y in z so tipa `short` in spremenljivka x ima vrednost 30000. Ko se izvede stavek `z = x + y;` dobi spremenljivka z vrednost -30000. Kolikšna je vrednost spremenljivke y?

   (a) -60000

   (b) 5535

   (c) 5536

   (d) 60000

10. Spremenljivke x, y in z so tipa `unsigned short` in spremenljivka x ima vrednost 5. Ko se izvede stavek `z = x - y;` dobi spremenljivka z vrednost 6. Kolikšna je vrednost spremenljivke y?

    (a) -1

    (b) 1

    (c) 65535

    (d) 65536

11. Spremenljivka x je tipa float in ima vrednost 0,4. Kateri od naslednjih izrazov ima vrednost 1?

    (a) `x == 0.4`

    (b) `0.4 == x`

    (c) `(double) x == 0.4`

    (d) `x == (float) 0.4`

12. Spremenljivka x je tipa float in ima vrednost 0,4. Kateri od naslednjih izrazov ima vrednost 1?

(a) `x == 0.4`

(b) `0.4 == x`

(c) `(double) x == 0.4`

(d) `0.3 < x && x < 0.5`

13. V spremenljivko x želimo shraniti vrednost, katere dvojiški zapis je 11000001. Kako to naredimo?

    (a) `x = 0x11000001;`

    (b) `x = 11000001;`

    (c) `x = 0xC1;`

    (d) `x = C1;`

14. Spremenljivka a tipa `unsigned int` ima desetiško vrednost 17. Kaj se na prikazovalnik izpiše ob klicu `printf("%x", a);`?

    (a) 17

    (b) 11

    (c) 10001

    (d) Nič od naštetega.

6 Registri

Register je posebne vrste pomnilniška celica in sicer 32-bitna pomnilniška celica. S stališča programerja praktično ni razlike med branjem oziroma pisanjem podatka v običajen pomnilnik, kjer hranimo spremenljivke, in branjem in pisanjem podatkov v registre.

S stališča funkcionalnosti pa med običajnim pomnilnikom in registri obstajata dve bistveni razliki:

1. Spremenljivkam se v običajnem pomnilniku dodeli poljuben naslov. Nobene razlike ni, če se, na primer, spremenljivka x nahaja na naslovu 0x40003F4C ali na naslovu 0x40001B20 Po drugi strani so registrom naslovi točno določeni. Registru IO0PIN, na primer, je določen naslov 0xE0028000. Na tem naslovu ne more biti nič drugega, kakor tudi register IO0PIN ne more biti na nobenem drugem naslovu. Zakaj je temu tako, bo delno pojasnila naslednja točka.

2. Kadar zapišemo podatek v običajen pomnilnik, storimo to izključno z namenom, da bi podatek shranili za kasnejšo uporabo. Kadar zapišemo podatek v register, storimo to z namenom, da bi s tem posredno povzročili nek dogodek. Na primer, vpis enice v bit b_8 registra IO0PIN povzroči, da se na vratih P0.8 pojavi napetost 3,3V, če so vrata določena kot izhodna. Velja tudi obratno. Če so vrata določena kot vhodna, potem se ustrezen bit v registru IO0PIN postavi na vrednost, ki ustreza napetosti na vratih.

Pojasnimo razliko med običajnimi pomnilniškimi celicami in registri z bolj vsakdanjim primerom. Vzemimo, da imamo neko pismo, ki ga želimo shraniti za kasnejšo rabo. To pismo lahko shranimo v katerikoli predal običajne omare. Važno je le, da je v predalu kaj prostora in da si zapomnimo, kje je pismo, da ga lahko kasneje najdemo. Kadar pa želimo, da se bo s pismom kaj zgodilo – na primer, da se bo odposlalo – ni več vseeno, kam ga shranimo. Shraniti ga moramo v nabiralnik. Vendar spet ne v katerikoli nabiralnik, ampak v uradni poštni nabiralnik Pošte Slovenije.

V primeru, ki smo ga pravkar opisali, lahko gledamo na predale kot pomnilniške celice, na poštni nabiralnik pa kot register.

6.1 Makroji

V knjižnici `io.h` so naslovi registrov definirani z makroji, zato verjetno ne bo odveč, če rečemo besedo, dve o njih. Definicija makroja v jeziku C ima naslednjo obliko:

```
#define IME_MAKROJA zaporedje_znakov
```

Makro deluje na zelo preprost način. Preden prevajalnik začne s prevajanjem kode, makro povsod v datoteki nadomesti `IME_MAKROJA` z zaporedjem znakov, ki imenu sledijo. V tem smislu lahko razumemo makro kot ukaz "Išči in zamenjaj" (Find & Replace). Na primer, če želimo v programu uporabiti konstanto Π, definiramo takšen makro:

```
#define PI 3,14159265359
```

Nenapisano pravilo je, da imena makrojev pišemo z velikimi črkami, da se na ta način ločijo od običajnih spremenljivk.

V knjižnici `io.h` boste našli naslednjo definicijo registra IO0PIN:

```
#define IO0PIN  (*((unsigned int *)0xe0028000))
```

Kaj zapis na desni strani dejansko pomeni, bomo zvedeli v poglavju o kazalcih. Zaenkrat nas bo zadovoljila razlaga, da se `IO0PIN` obnaša kot 32-bitna spremenljivka na naslovu 0xe0028000. Ker je na tem naslovu ravno register, ki skrbi za prenos podatkov med mikrokrmilnikom in vrati od P0.0 do P0.31, bo branje in pisanje v to spremenljivko v resnici pomenilo branje in pisanje v register `IO0PIN`.

6.2 Branje tipk

Kot že vemo, so na sistemu Š-ARM med drugim tudi 4 tipke. Z visokonivojsko funkcijo `getch()` lahko beremo, katera tipka je bila pritisnjena. Kmalu spoznamo, da nas uporaba funkcije `getch()` precej omejuje. Prvič, s to funkcijo ne moremo samo preveriti, če je kakšna tipka pritisnjena. Funkcija trmasto čaka, dokler kakšne tipke dejansko ne pritisnemo. Drugič, s to funkcijo ne moremo ugotoviti, če sta slučajno hkrati pritisnjeni dve tipki. V takšnih primerih moramo uporabiti register `IO0PIN`.

Vse štiri tipke s pomočjo jumperja J16 priklopimo na vrata od P0.15 (tipka T3) do P0.12 (tipka T0). Stanje napetosti na teh vratih se kaže v obliki enic oziroma ničel v bitih b_{15} do b_{12} registra `IO0PIN`.

Naslednji program na prikazovalnik ves čas prikazuje vsebino registra `IO0PIN`. Sam program ni nič posebnega, le na začetku boste opazili uporabo makroja `KEY_INIT`, ki je definiran v knjižnici `io.h`. Makro poskrbi za to, da se vsa štiri vrata, na katerih imamo priklopljene tipke, nastavijo kot vhodna.

```
#include "io.h"

int main(void)
{
    KEY_INIT; //makro, ki nastavi smeri pinov
    _LCDInit();
    _setcursortype(_NOCURSOR);

    while (1)
    {
      printf("%08x\r", IO0PIN);
    }
    return 0;
  }
```

Ko program zaženemo, se pokaže na prikazovalniku vsebina registra:

```
7680f3f3
```

Na četrtem mestu z desne opazimo šestnajstiško cifro f. To pomeni, da so biti od b_{15} do b_{12}, ki ustrezajo našim štirim tipkam, vsi enaki 1. Vse tipke so namreč spuščene. Vrednosti ostalih bitov nas ta hip ne zanimajo. Naslednja slika podrobneje prikazuje stanje v registru `IO0PIN` v tem hipu.

$$\boxed{0111\ 0110\ 1000\ 0000\ b_{15} = 1 b_{14} = 1 b_{13} = 1 b_{12} = 1\ 0011\ 1111\ 0011}$$

Če, na primer, zdaj pritisnemo in držimo tipki T2 in T0, se na vratih P0.14 in P0.12 pojavita nizka napetostna nivoja, kar povzroči, da se bita b_{14} in b_{12} v registru `IO0PIN` pobrišeta. V registru imamo zdaj takšno stanje:

$$\boxed{0111\ 0110\ 1000\ 0000\ b_{15} = 1 b_{14} = 0 b_{13} = 1 b_{12} = 0\ 0011\ 1111\ 0011}$$

Na prikazovalniku se zaradi tega prikaže vrednost

```
7680a3f3
```

7 Bitne operacije

Na koncu prejšnjega razdelka smo videli, kako s pritiskom na tipko povzročimo spremembo stanja ustreznega bita v registru `IOOPIN`. Hitro se nam zastavi vprašanje, kako lahko računalnik iz 32-bitne vrednosti, ki jo hrani register, izlušči informacijo o stanju čisto konkretne tipke. V tem poglavju bomo osvojili znanje, ki ga za to potrebujemo.

7.1 Pomik bitov

S posebnima operandoma lahko pomikamo bite v določeni spremenljivki levo ali desno. Pomik za n bitov v levo dosežemo z operandom `<<n` in za n bitov v desno z operandom `>>n`. Pri pomiku v levo se z desne dodajajo ničle, pri pomiku v desno pa se ničle dodajajo z leve.

Vzemimo za primer 16-bitno spremenljivko x, ki ima šestnajstiško vrednost 00C1:

```
unsigned short x = 0x00C1;
```

Če to vrednost zapišemo v dvojiškem zapisu, dobimo

```
0000000011000001
```

Pomaknimo zdaj bite spremenljivke x za 3 mesta v desno:

```
x = x >> 3;
```

Če želimo spremeniti vrednost spremenljivke x, moramo uporabiti priredilni operator. Izraz na desni strani priredilnega operatorja zgolj vrne vrednost, ki ima bite pomaknjene za 3 mesta v desno, in ničesar ne spremeni.

Premaknjena vrednost v dvojiškem zapisu je

```
0000000000011000
```

kar je v šestnajstiškem zapisu enako 0018. Vidimo, da je enka na skrajni desni pri pomikanju preprosto izpadla, z leve pa so v vrednost vstopile 3 ničle.

Če bi isto vrednost (00C1) pomaknili, na primer, za dve mesti v levo:

```
x = x << 2;
```

bi dobili dvojiško vrednost 0000001100000100, kar je v šestnajstiškem zapisu 0304.

7.2 Eniški komplement

Operacijo eniškega komplementa, ki pomeni negacijo posameznih bitov, naredimo z operatorjem `~`.

Vzemimo za primer spet 16-bitno spremenljivko x, ki ima šestnajstiško vrednost 003F. V tem primeru je vrednost izraza `~x` enaka FFC0. Vrednost spremenljivke x se pri tem ne spremeni. Če želimo dejansko negirati bite v spremenljivki x, moramo uporabiti priredilni operator:

```
x = ~x;
```

7.3 Bitna logična operacija ALI in postavljanje bitov

O logičnih operatorjih smo že govorili v razdelku Logični in primerjalni izrazi. Logični operatorji, ki smo jih spoznali tam, prijemajo nad celimi izrazi. Zanima jih le, ali so izrazi, ki nastopajo v operacijah, enaki ali različni od nič, rezultat operacije pa je lahko le 0 ali 1.

Bitni logični operatorji delujejo nad posameznimi biti. Naslednja tabela prikazuje štiri možne kombinacije vrednosti dveh bitov a in b ter ustrezne rezultate bitne logične operacije ALI. Operator, ki izvede to operacijo, je pokončna črta (|).

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Operator vrne 1 v primeru, ko je vsaj en od operandov enak 1. Iz tabele razberemo dve zelo pomembni lastnosti bitne logične operacije ALI:

$$a \mid 0 = a \tag{11}$$

$$a \mid 1 = 1 \tag{12}$$

Operacija z ničlo ohranja vrednost bita, medtem ko operacija z enko postavi bit na 1. Ti dve lastnosti sta izjemno koristni, kadar želimo postaviti enega ali več bitov, ne da bi pokvarili vrednosti preostalih bitov.

Vzemimo za primer, da želimo v registru `IOOPIN` postaviti bit b_{11}. Napraviti moramo naslednjo operacijo:

$$\begin{array}{cccccccccccc} & b_{31} & b_{30} & \cdots & b_{12} & b_{11} & b_{10} & \cdots & b_1 & b_0 & \\ \mid & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 & \text{(maska)} \\ \hline & b_{31} & b_{30} & \cdots & b_{12} & 1 & b_{10} & \cdots & b_1 & b_0 & \end{array}$$

Drugi vrednosti, ki nastopa v gornji operaciji, pravimo *maska*. Z njo določimo, kateri biti bodo ostali nedotaknjeni in kateri biti se bodo postavili. Ker logična operacija ALI z ničlo ohranja vrednost bita, postavimo v maski ničle na mesta bitov, ki jih želimo ohraniti. Na mesta bitov, ki jih želimo postaviti, damo v maski enico. Gornjo operacijo zapišemo v jeziku C takole:

```
IOOPIN |= 0x00000800;
```

S tem stavkom postavimo bit b_{11} v registru `IOOPIN`.

7.4 Bitna logična operacija IN in brisanje bitov

Naslednja tabela prikazuje štiri možne kombinacije vrednosti dveh bitov a in b ter ustrezne rezultate bitne logične operacije IN. Operator, ki izvede to operacijo, zapišemo z znakom `&`.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Operator vrne 1 le v primeru, ko sta oba operanda enaka 1. Iz tabele razberemo naslednji dve lastnosti bitne logične operacije IN:

$$a\&0 = 0 \tag{13}$$

$$a\&1 = a \tag{14}$$

Vidimo, da tokrat operacija z enico ohranja vrednost bita, medtem ko operacija z ničlo bit pobriše. Ti dve lastnosti sta koristni, kadar želimo pobrisati enega ali več bitov, ne da bi pokvarili vrednosti preostalih bitov.

Vzemimo za primer, da želimo v registru `IOOPIN` pobrisati bit b_4. Napraviti moramo naslednjo operacijo:

	b_{31}	b_{30}	$\cdots$	b_5	b_4	b_3	b_2	b_1	b_0	
&	1	1	$\cdots$	1	0	1	1	1	1	(maska)
	b_{31}	b_{30}	$\cdots$	b_5	0	b_3	b_2	b_1	b_0	

V jeziku C to zapišemo s stavkom

```
IOOPIN &= 0xFFFFFFEF;
```

S tem stavkom pobrišemo bit b_4 v registru `IOOPIN`.

7.5 Bitna logična operacija izključno ALI in negacija bitov

Naslednja tabela prikazuje štiri možne kombinacije vrednosti dveh bitov a in b ter ustrezne rezultate bitne logične operacije izključno ALI (Exclusive OR, XOR). Operator, ki izvede to operacijo, zapišemo z znakom `^`.

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Operacijo izključno ALI lahko tolmačimo kot dvojiško seštevanje brez prenosa. Nič plus nič je nič, nič plus ena je ena in ena plus ena je spet nič. Iz tabele razberemo naslednji dve lastnosti bitne logične operacije izključno ALI:

$$a \hat{} 0 = a \tag{15}$$

$$a \hat{} 1 = \tilde{} a \tag{16}$$

Operacija z ničlo ohranja vrednost bita, medtem ko operacija z enico bit negira. Ti dve lastnosti sta koristni, kadar želimo negirati vrednost enega ali več bitov, ne da bi pokvarili vrednosti preostalih bitov.

Vzemimo za primer, da želimo v registru `IO0PIN` negirati bit b_2. Napraviti moramo naslednjo operacijo:

	b_{31}	b_{30}	$\cdots$	b_3	b_2	b_1	b_0	
$\hat{}$	0	0	$\cdots$	0	1	0	0	(maska)
	b_{31}	b_{30}	$\cdots$	b_3	$\tilde{} b_2$	b_1	b_0	

V jeziku C to zapišemo s stavkom

```
IO0PIN ^= 0x00000004;
```

S tem stavkom negiramo bit b_2 v registru `IO0PIN`.

7.6 Ugotavljanje vrednosti bitov

Bitne logične operacije smo doslej s pridom uporabili za spreminjanje vrednosti posameznih bitov. Na začetku poglavja pa smo si zastavili drugačen problem: kako *ugotoviti* vrednost posameznega bita.

Vzemimo za primer, da nas zanima, kakšno vrednost ima bit b_{13} registra `IO0PIN`. Sestavimo 32-bitno masko, ki ima na mestu tega bita enko, na vseh ostalih mestih pa ničle: 0x00002000. Če napravimo bitno logično operacijo IN med to masko in registrom, dobimo vrednost, ki ima vse bite, razen bita b_{13}, enake nič. Bit b_{13} obdrži svojo originalno vrednost:

	b_{31}	b_{30}	$\cdots$	b_{14}	b_{13}	b_{12}	$\cdots$	b_1	b_0	
&	0	0	$\cdots$	0	1	0	$\cdots$	0	0	(maska)
	0	0	$\cdots$	0	b_{13}	0	$\cdots$	0	0	

Jasno je, da bo celotna vrednost enaka 0, če je bit b_{13} enak 0, če pa je bit b_{13} enak 1, bo celotna vrednost različna od nič. Natančneje, enaka bo 0x00002000.

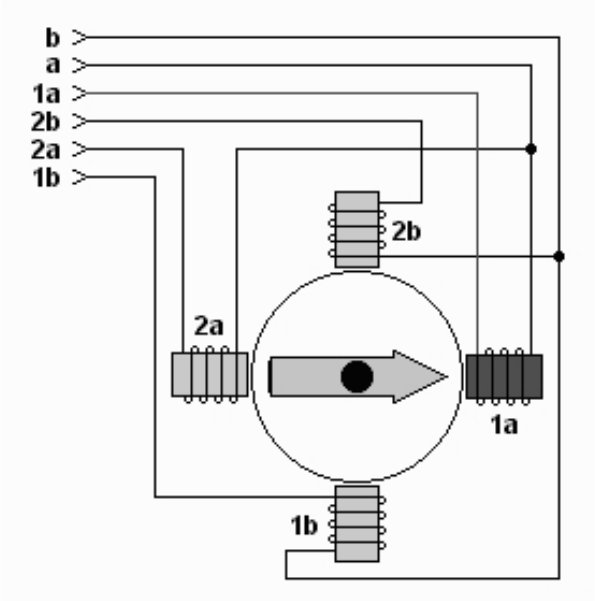
Naslednji del kode prikazuje, kako v jeziku C zapišemo pogoj, ki bo odvisen izključno od vrednosti bita b_{13} registra `IO0PIN`:

```
if ((IO0PIN & 0x00002000) != 0) //Notranji oklepaji so zato, ker ima
                                //primerjalni operator != sicer
                                //prednost pred bitno operacijo IN.
{
  //ta koda se bo izvedla, ce je bit b13 postavljen
}
else
{
  //ta koda se bo izvedla, ce je bit b13 pobrisan
}
```

7.7 Primer s koračnim motorjem

Z namenom, da nekoliko utrdimo, kar smo izvedeli v tem poglavju, si oglejmo primer enostavnega krmiljenja koračnega motorja (stepper motor).

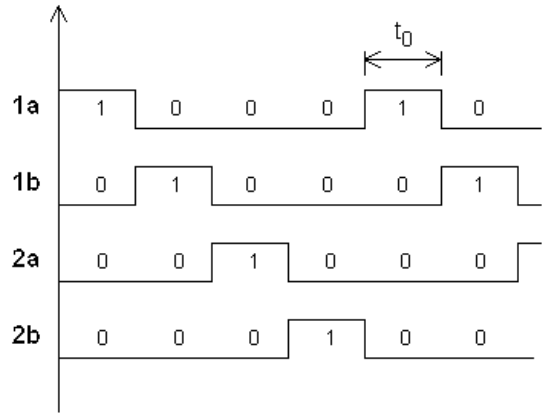
Naslednja slika prikazuje model koračnega motorja.



Motor je sestavljen iz štirih nepremičnih tuljav (stator) in vrtljivega trajnega magneta (rotor). Rotor je obrnjen proti tuljavi, skozi katero v določenem trenutku teče tok (na sliki je to tuljava

1a). Magnet namreč privlači magnetno polje tuljave. V trenutku, ko preklopimo tok na katero od sosednjih tuljav, na primer 1b, se magnet obrne proti tej tuljavi. Vidimo, da se motor ne vrti zvezno, ampak v drobnih korakih. Odtod ime koračni motor.

Vrtenje motorja v smeri urinega kazalca dosežemo s cikličnim spreminjanjem napetosti na vseh štirih tuljavah, kakor kaže naslednji diagram:



Iz diagrama lahko razberemo, kako se v trenutku, ko tuljava 1a izgubi napetost (prehod iz visokega v nizek napetostni nivo), visok nivo pojavi na tuljavi 1b. Prav tako se po določenem času (t_0) visok napetostni nivo prenese iz tuljave 1b na 2a ter spet po času t_0 naprej na 2b. Za tuljavo 2b je spet na vrsti tuljava 1a.

Hitrost vrtenja motorja je pogojena s časom t_0, toliko časa se namreč magnet zadržuje na eni tuljavi. Seveda obstaja gornja meja hitrosti, ki nam jo postavlja fizika same konstrukcije motorja.

Koračni motor smo priklopili na naš sistem tako, da smo vezali sponko 1a motorja na sponko P0.7 mikrokrmilniškega sistema, sponko 1b smo vezali na sponko P0.6, sponko 2a na P0.5 in sponko 2b na P0.4. Ker sponke P0.4 do P0.7 ustrezajo bitom b_4 do b_7 registra `IOOPIN`, bomo vrtenje v smeri urinega kazalca dosegli s pisanjem enk in ničel v ustrezne bite v naslednjem zaporedju:

b_7 **(1a)**	b_6 **(1b)**	b_5 **(2a)**	b_4 **(2b)**
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0
0	1	0	0
...	...	...	...

Za vrtenje v nasprotno smer samo obrnemo vrstni red pisanja:

b_7 **(1a)**	b_6 **(1b)**	b_5 **(2a)**	b_4 **(2b)**
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
...	...	...	...

Program za krmiljenje našega koračnega motorja bomo naredili tako, da se bo motor vrtel v smeri urinega kazalca, če bomo držali tipko T0 in v nasprotni smeri, če bomo držali tipko T1. Spomnimo se, da je tipka T0 prek jumperja J16 priključena na sponko P0.12 in tipka T1 na sponko P0.13.

Krmiljenje bomo realizirali izključno na nivoju registrov. Zato moramo, preden začnemo, spoznati še dva registra.

Prek registra PINSEL0 določimo, katere priključne sponke se bodo obnašale kot vrata (porti). Kot vemo, ni nujno, da vsaka sponka predstavlja vhodna ali izhodna vrata. Na sponko P0.27 smo, na primer, priklopili vhod pretvornika AD.

Register PINSEL0 je 32-biten in z njim lahko sponkam med P0.0 do P0.15 določimo, ali se bodo obnašale kot vrata. Bita b_0 in b_1 sta namenjena sponki P0.0, bita b_2 in b_3 sta namenjena sponki P0.1 in tako vse do bitov b_{30} in b_{31}, ki sta namenjena sponki P0.15. V oba bita moramo vpisati vrednost 0, da bo sponka delovala kot vrata. Ker mi potrebujemo vrata P0.4 do P0.7 za motor ter vrata P0.12 in P0.13 za dve tipki, moramo vpisati ničle v bite od b_8 do b_{15} za motor in v bite b_{24} do b_{27} za tipki. Ostale bite moramo pustiti pri miru. Vse to dosežemo z bitno logično operacijo IN in z masko

```
11110000111111110000000011111111,
```

ki je v šestnajstiškem zapisu enaka 0xF0FF00FF.

Drugi register, ki ga bomo rabili, in ga še ne poznamo, je register IO0DIR. S tem 32-bitnim registrom določimo smeri vrat med P0.0 do P0.31, ki ustrezajo bitom b_0 do b_{31}. Če je vrednost bita 0, potem so vrata vhodna, sicer so izhodna. V našem primeru bodo vrata P0.12 in P0.13 vhodna, vrata od P0.4 do P0.7 pa izhodna. Z eno samo operacijo ni možno pobrisati enih in hkrati nastaviti nekih drugih bitov. To storimo v dveh korakih:

1. Z bitno logično operacijo IN in masko 0xFFFFCFFF pobrišemo bita b_{12} in b_{13}.

2. Z bitno logično operacijo ALI in masko 0x000000F0 postavimo bite od b_4 in b_7.

Končno imamo dovolj podatkov, da lahko zapišemo program za krmiljenje motorja:

```
#include "io.h"

unsigned int korak(unsigned int zaporedje, int smer);
```

```
int main()
{
  unsigned int polozaj = 0x8;
  PINSEL0 &= 0xF0FF00FF; //pini P0.13, P0.12 in P0.7 do P0.4 bodo vrata
  IO0DIR |= 0x000000F0;  //motor - izhodna vrata
  IO0DIR &= 0xFFFFCFFF;  //tipki - vhodna vrata
  _TimerInit();

  while (1)
  {
    //Ce je pritisnjena katero od tipk T0 ali T1:
    if ((IO0PIN & 0x00003000) != 0x00003000)
    {
      //Ce je pritisnjena tipka T0:
      if ((IO0PIN & 0x00001000) == 0)
      {
        polozaj = korak(polozaj, 1);
      }
      //Ce je pritisnjena tipka T1:
      if ((IO0PIN & 0x00002000) == 0)
      {
        polozaj = korak(polozaj, -1);
      }
      /*Naslednja kombinacija bitov iz zaporedja za
      krmiljenje koracnega motorja se zapise v register IO0PIN:*/
      IO0PIN = (IO0PIN & 0xFFFFFF0F) | (polozaj << 4);
      delay(30);
    }
  }
  return 0;
}

unsigned int korak(unsigned int zaporedje, int smer)
{
  if (smer == 1) zaporedje >>= 1;
  else zaporedje <<= 1;
  if (zaporedje == 0x10) zaporedje = 0x01;
  if (zaporedje == 0x00) zaporedje = 0x08;
  return zaporedje;
}
```

V funkciji `main()` je pred stavkom `while` poleg nastavitev vrat in inicializacije ure še deklaracija spremenljivke *polozaj*, ki jo na začetku nastavimo na 0x8 (dvojiško 1000). Ta spremenljivka bo

ves čas hranila enega od štirih možnih položajev motorja (1000, 0100, 0010 ali 0001).

Takoj po vstopu v zanko `while` najprej pogledamo, če je pritisnjena katera od obeh tipk T0 ali T1. To storimo tako, da z bitno logično operacijo IN in masko 0x00003000 pobrišemo vse bite razen bitov b_{12} in b_{13}. Če ni pritisnjena nobena tipka, potem imata oba bita vrednost 1 in rezultat operacije IN med registrom `IOOPIN` in masko 0x00003000 je 0x00003000. Če je katera od obeh tipk pritisnjena, je rezultat različen od 0x00003000.

Na podoben način v obeh naslednjih stavkih `if` ugotovimo, če je pritisnjena tipka T0 oziroma T1 in zavrtimo motor za korak v ustrezno smer.

Kakšno je naslednje zaporedje štirih bitov, ki jih moramo posredovati motorju, določi funkcija `korak()`. Funkciji podamo trenutno zaporedje bitov, kakršnega imamo na izhodnih štirih sponkah, in smer vtenja. Če je smer enaka 1, se biti pomaknejo v desno, kar bo povzročilo obrat motorja v smeri urinega kazalca. Sicer bomo dobili obrat v nasprotni smeri urinega kazalca. Zadnja dva stavka `if` v funkciji `korak()` poskrbita za pravilen krožen pomik bitov. Če je bit že preveč levo oziroma desno, se postavi nazaj v izhodiščen položaj.

Ko nam funkcija `korak()` vrne pravilno zaporedje bitov, je potrebno to zaporedje vpisati na ustrezno mesto v register `IOOPIN`. Z bitno operacijo IN in masko 0xFFFFFF0F pobrišemo bite od b_4 do b_7, z bitno operacijo ALI z zaporedjem bitov, ki so zapisani v spremenljivki *polozaj*, pa postavimo na ena ustrezen bit. Pred tem moramo vsebino spremenljivke *polozaj* pomakniti za 4 mesta v levo, da dobimo bit na pravo mesto.

Na koncu kličemo še funkcijo `delay()`, ki poskrbi za 30 milisekndno zakasnitev (t_0) do naslednje spremembe, ki pomeni naslednji korak motorja.

7.8 Primeri izpitnih vprašanj

1. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0xABCD. Kakšna je vrednost izraza `x << 2`?

   (a) `0xCD00`

   (b) `0xCDAB`

   (c) `0xAF34`

   (d) `0xAF36`

2. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0x0006. Kakšna je vrednost izraza `x >> 3`?

   (a) `0x0000`

   (b) `0x0001`

   (c) `0x0003`

   (d) `0x0018`

3. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0xE0F0. Kakšna je vrednost izraza `~x`?

(a) `0x1F0F`

(b) `0x1F10`

(c) `0xe0f0`

(d) `0x0E0F`

4. Imamo spremenljivko x tipa `unsigned short`. Kakšna je vrednost izraza `~x + x`?

   (a) `0x0000`

   (b) `0x1111`

   (c) `0xFFFF`

   (d) Odvisno od vrednosti spremenljivke x.

5. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0x000C. Kakšna je vrednost izraza `x | 0x0101`?

   (a) `0x0000`

   (b) `0x010D`

   (c) `0x000C`

   (d) `0x010E`

6. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0x000C. Kakšna je vrednost izraza `x & 0x0001`?

   (a) `0x0000`

   (b) `0x000D`

   (c) `0x000C`

   (d) `0x000E`

7. Imamo spremenljivko x tipa `unsigned short`, ki ima vrednost 0x000C. Kakšna je vrednost izraza `x ^ 0x0008`?

   (a) `0x0000`

   (b) `0x0004`

   (c) `0x000C`

   (d) `0x0008`

8. Kako pobrišemo bita b_0 in b_2 v registru `IO0PIN`, ne da bi pri tem spremenili vrednosti preostalih 30 bitov?

   (a) `IO0PIN |= 0x00000005;`

   (b) `IO0PIN |= 0xFFFFFFFA;`

(c) `IO0PIN &= 0x00000005;`

(d) `IO0PIN &= 0xFFFFFFFA;`

9. Kako postavimo bita b_4 in b_8 v registru `IO0PIN`, ne da bi pri tem spremenili vrednosti preostalih 30 bitov?

   (a) `IO0PIN |= 0x00000110;`

   (b) `IO0PIN |= 0xFFFFFEEF;`

   (c) `IO0PIN &= 0x00000110;`

   (d) `IO0PIN &= 0xFFFFFEEF;`

10. Kako negiramo bit b_{11} v registru `IO0PIN`, ne da bi pri tem spremenili vrednosti preostalih 31 bitov?

    (a) `IO0PIN |= 0x00000800;`

    (b) `IO0PIN |= 0xFFFFF7FF;`

    (c) `IO0PIN ^= 0x00000800;`

    (d) `IO0PIN ^= 0xFFFFF7FF;`

11. Kako v registru `IO0PIN` hkrati postavimo bita b_0 in b_1 ter pobrišemo bit b_2, ne da bi pri tem spremenili vrednosti preostalih 29 bitov?

    (a) `IO0PIN = (IO0PIN | 0x00000003) & 0xFFFFFFFB;`

    (b) `IO0PIN = (IO0PIN | 0x00000004) & 0xFFFFFFFC;`

    (c) `IO0PIN = (IO0PIN & 0x00000003) | 0xFFFFFFFB;`

    (d) `IO0PIN = (IO0PIN & 0x00000004) | 0xFFFFFFFC;`

12. Kako dosežemo, da se v spremenljivko x vpiše vrednost 1, če je bit b_{12} v registru `IO0PIN` pobrisan?

    (a) `if (IO0PIN & 0x00001000) x = 1;`

    (b) `if (IO0PIN | 0x00001000) x = 1;`

    (c) `if ((IO0PIN & 0x00001000) == 0) x = 1;`

    (d) `if ((IO0PIN | 0x00001000) == 0) x = 1;`

13. Biti b_{12} do b_{15} v registru `IO0PIN` odražajo stanje štirih tipk. Ko je tipka spuščena, je ustrezen bit postavljen, ko je tipka pritisnjena, je bit pobrisan. Kako dosežemo, da se v spremenljivko *tipka* vpiše vrednost 1, če je pritisnjena vsaj ena tipka?

    (a) `if ((IO0PIN & 0x0000F000) == 0x0000F000) tipka = 1;`

    (b) `if ((IO0PIN & 0x0000F000) != 0x0000F000) tipka = 1;`

(c) `if ((IO0PIN & 0xFFFF0FFF) == 0) tipka = 1;`

(d) `if ((IO0PIN & 0xFFFF0FFF) != 0) tipka = 1;`

14. Biti b_{12} do b_{15} v registru `IO0PIN` odražajo stanje štirih tipk. Ko je tipka spuščena, je ustrezen bit postavljen, ko je tipka pritisnjena, je bit pobrisan. Kako dosežemo, da se v spremenljivko *tipka* vpiše vrednost 0, če ni pritisnjena nobena tipka?

(a) `if ((IO0PIN & 0x0000F000) == 0x0000F000) tipka = 0;`

(b) `if ((IO0PIN & 0x0000F000) != 0x0000F000) tipka = 0;`

(c) `if ((IO0PIN & 0xFFFF0FFF) == 0) tipka = 0;`

(d) `if ((IO0PIN & 0xFFFF0FFF) != 0) tipka = 0;`

8 Sistemi v realnem času

Pri problemih, ki smo jih reševali doslej, nas čas ni preveč skrbel. V primeru s koračnim motorjem smo z uporabo funkcije `delay()` poskrbeli, da je določeno stanje na izhodnih sponkah zdržalo *vsaj* 30 milisekund. S tem smo določili zgornjo mejo hitrosti vrtenja motorja. Če bi v program dodali nova opravila, na primer zaznavanje položaja motorja prek zunanjih stikal ali krmiljenje dodatnih motorjev, bi se hitrost vrtenja upočasnila. Hitrost niti ne bi bila več konstantna, odvisna bi bila od tega, v kakšni meri bi bil računalnik zaposlen z ostalimi opravili. To pa je v veliko primerih nedopustno.

Kadar pravimo, da nek sistem deluje v *realnem času*, s tem mislimo, da se *odziva v točno predpisanih časovnih okvirih*. Če določeno opravilo izvede prezgodaj ali prepozno, lahko to za celoten sistem pomeni enako težavo, kot če bi opravilo izvedel narobe.

Vzemimo primer iz vsakdanjega življenja. Franci pride v restavracijo na kosilo in se vsede za mizo. V restavraciji je že precej gostov. Nekateri so pravkar pojedli juho, drugi se mastijo s sladico, spet tretji čakajo na račun. Kljub temu Franci pričakuje, da bo v *določenem* času natakar obiskal tudi njegovo mizo. Čeprav je zelo potrpežljiv, bo po eni uri čakanja restavracijo gotovo zapustil.

Dober natakar ves čas budno spremlja vse mize in v najkrajšem možnem času ustreže sleherni želji slehernega gosta. Nikakor ne čaka, da bo prvi gost končal s kosilom in plačal račun, preden se bo lotil naslednjega. To bi bilo prav tako narobe, kot če bi vsakega gosta polil z vročo juho.

8.1 Povpraševanje

Najenostavnejši način delovanja sistema v realnem času, ki ga bomo tudi edinega spoznali, se imenuje *sistem s povpraševanjem* (polling). Deluje natanko tako kot naš natakar. Po vrsti pregleduje vsa opravila, za katera mora skrbeti, in če zazna potrebo po ukrepanju, tudi ukrepa. Pri tem bomo predpostavili, da so časi posameznih ukrepanj dovolj kratki, da zadovoljijo časovne zahteve, ki jih postavimo na sistem. Tudi natakar se pri vsakem gostu pomudi le toliko, kolikor je potrebno, da ustreže njegovi trenutni želji. Če bi se z njim ukvarjal predolgo, bi ostali gostje postali nestrpni.

8.2 Primer z uro

Ogledali si bomo preprost primer ure. Ob zagonu sistema bo ura pričela teči, točen čas pa bomo nastavljali z vsemi štirimi tipkami. S tipkama T0 in T1 bomo nastavljali minute, s tipkama T2 in T3 pa ure.

Tule je kompleten program:

```
#include "io.h"

void pokazi(unsigned long sek, long popr);

int main(void)
{
  unsigned long sekunde = 0;
  long popravek = 0;

  _TimerInit();
  _LCDInit();
  _KeyInit();
  _setcursortype(_NOCURSOR);
  pokazi(sekunde, popravek);

  while(1)
  {
    if (kbhit()) //Ali je pritisnjena kaksna tipka?
    {
      switch (getch())
      {
        case '0': popravek -= 60; break;
        case '1': popravek += 60; break;
        case '2': popravek -= 3600; break;
        case '3': popravek += 3600;
      }
      pokazi(sekunde, popravek);
    }
    if (clock() / 1000 > sekunde) //Poglej, ce je ze minila sekunda.
    {
      sekunde = clock() / 1000;
      pokazi(sekunde, popravek);
    }
  }
  return 0;
}
```

```
void pokazi(unsigned long sek, long popr)
{
  int sekunde, minute, ure;

  sek += popr;
  sekunde = sek % 60;
  minute = (sek / 60) % 60;
  ure = (sek / 3600) % 24;

  printf("%2d:%02d:%02d\r", ure, minute, sekunde);
}
```

Glavno dogajanje se odvija v funkciji `main()` v neskončni zanki `while`. Tam izmenično preverjamo, če je pritisnjena kakšna tipka, ali če je že pretekla sekunda od zadnje spremembe prikaza na prikazovalniku. Kadarkoli se zgodi kateri od obeh dogodkov, ustrezno ukrepamo.

Ali je tipka pritisnjena, ugotovimo s klicem funkcije `kbhit()`. Ta funkcija v primeru, ko tipka ni pritisnjena, vrne 0, sicer vrne vrednost različno od nič. Pomembno je, da funkcija na pritisk tipke ne čaka, ampak gre v vsakem primeru naprej. Če je tipka pritisnjena, jo preberemo s funkcijo `getch()` in v stavku `switch` ustrezno spremenimo vrednost spremenljivke *popravek*.

Da bomo razumeli vlogo te spremenljivke in spremenljivke *sekunde*, si oglejmo najprej definicijo funkcije `pokazi()`. Ta funkcija sprejme dva parametra. Prvi parameter predstavlja število sekund, ki so pretekle od zagona sistema. Iz tega podatka lahko izračunamo, koliko je preteklo ur, minut in sekund:

```
sekunde = sek % 60;
minute = (sek / 60) % 60;
ure = (sek / 3600) % 24;
```

Ker pa nas ne zanima, koliko časa je preteklo od zagona sistema, ampak, bi radi na prikazovalniku prikazovali točen čas, moramo dejansko pretečenim sekundam prišteti nek popravek. Tega dobimo prek drugega parametra funkcije `pokazi()`.

Vrnimo se k stavku `switch` v funkcijo `main()`. Zdaj razumemo, kaj se v njem dogaja. Če smo pritisnili tipko T0 oziroma T1, se *popravek* zmanjša oziroma poveča za 60. S tem v resnici povzročimo, da se prikazana ura pomakne za minuto nazaj oziroma naprej. Če smo pritisnili tipki T2 oziroma T3, se *popravek* zmanjša oziroma poveča za 3600, kar pomeni premik ure za eno uro nazaj oziroma naprej.

Takoj po stavku `switch` moramo poklicati funkcijo `pokazi()`, da se sprememba pokaže tudi na prikazovalniku.

Dejansko pretečeni čas vseskozi ugotavljamo s klicem funkcije `clock()`. Ta funkcija nam vrne število milisekund, ki so minile od klica funkcije `_TimerInit()`. Kadarkoli preteče vsaj ena sekunda, je čas, da popravimo vrednost spremenljivke *sekunde* in spremembo prikažemo na prikazovalniku:

```
if (clock() / 1000 > sekunde)
```

```
{
  sekunde = clock() / 1000;
  pokazi(sekunde, popravek);
}
```

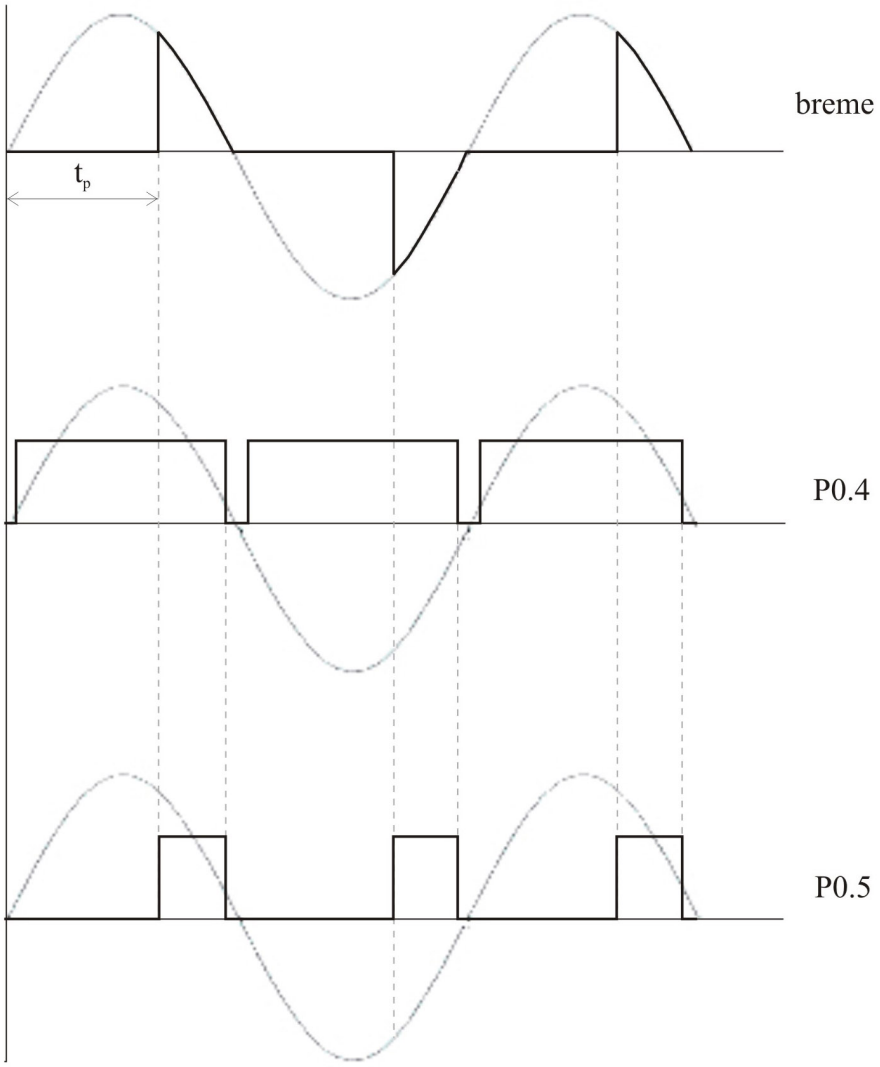
Za konec naj bralec razmisli, kaj se zgodi, ko število sekund preseže območje vrednosti, ki jih lahko zapišemo v spremenljivko tipa `unsigned long`. Tega tipa je namreč spremenljivka *sekunde*.

8.3 Stikalo za zatemnitev

Stikalo za zatemnitev (dimmer) je naprava, ki omogoča zvezno nastavljanje moči porabnika (npr. žarnice). Načrt vezja, ki ga bomo uporabili v ta namen, je prikazan v dodatku B. Podrobna razlaga vezja presega okvire našega predmeta, pojasnili bomo le osnovne principe, ki jih moramo razumeti, da lahko napišemo program.

Za krmiljenje moči uporabimo polprevodniški element triak (na načrtu označen z V2). Triak deluje tako, da ob prehodu izmenične napetosti skozi ničlo preneha prevajati. Ponovno začne prevajati šele, ko ga prožimo.

Odebeljena črta na prvem od treh grafov na spodnji sliki prikazuje spreminjanje napetosti na bremenu, ki je priključen na triak. Ob prehodu napetosti skozi ničlo triak breme odklopi. Triak se spet vklopi po času t_p, ki predstavlja čas proženja. Večji je ta čas, manj moči je deležno breme.



Da dosežemo prikazan potek napetosti, mora naš računalniški sistem poskrbeti za dve reči:

1. zaznati mora prehod napajalne (omrežne) napetosti skozi ničlo (zero cross detection) in
2. ko od prehoda skozi ničlo preteče čas t_p, mora proizvesti prožilni pulz.

Za potrebe zaznavanja prehodov skozi ničlo vezje iz dodatka B generira uskladitveni signal, ki ga prikazuje odebeljena črta na drugem grafu na gornji sliki. Signal je ves čas na visokem logičnem nivoju, le malo pred prehodom omrežne napetosti skozi ničlo pade na nizek nivo. Tam ostane še malo časa po prehodu ničle. Uskladitveni signal pripeljemo na sponko P0.4 našega mikrokrmilnika. To sponko bomo konfigurirali kot vhodna vrata.

Odebeljena črta na zadnjem od gornjih treh grafov prikazuje prožilni signal, ki ga mora proizvesti naš mikrokrmilniški sistem. Signal dvignemo na visok logičen nivo, ko preteče čas t_p od spusta uskladitvenega signala. Takoj ob naslednjem spustu uskladitvenega signala prožilni

signal spet spustimo na nizek nivo. Prožilni signal speljemo prek sponke P0.5 in to sponko bomo konfigurirali kot izhodna vrata.

Poleg zaznavanja prehoda skozi ničlo in generiranja prožilnega signala mora naš sistem skrbeti tudi za nastavljanje časa proženja t_p. Program bomo napisali tako, da bo pritisk na tipko T0 povzročil temnenje (daljšanje časa proženja), pritisk na tipko T1 svetlenje (krajšanje časa proženja), pritisk na tipko T3 pa bo zamrznil trenutno stanje. Če bomo torej želeli nekoliko zatemniti luč, bomo pritisnili najprej tipko T0, in ko bo svetloba dosegla želeno jakost, bomo temnenje ustavili s tipko T3.

Rešiti moramo še težavo s časovno ločljivostjo našega sistema. Funkcija `clock()` ob vsakem klicu vrne število milisekund, ki so pretekle od inicializacije ure (`_TimerInit()`). Glede na to, da imamo opravka z omrežno napetostjo frekvence 50Hz, se polovica napetostnega vala zgodi v 10ms. Teoretično imamo tako na voljo le 10 različnih stopenj svetilnosti, kar je bolj uboga regulacija.

Ločljivost sistemske ure lahko z 1ms (1kHz) povečamo do $1/15\mu$s (15MHz). Ločljivost se nastavi v funkciji `_TimerInit()` in sicer se vpiše v register T0PR. Vrednost v tem registru lahko po klicu funkcije `_TimerInit()` popravimo in tako spremenimo ločljivost ure. Mi se bomo zadovoljili z ločljivostjo 10μs, torej bomo vrednost v tem registru delili s 100.

Tukaj je kompleten program za krmiljenje stikala za zatemnitev:

```
#include "io.h"

#define TEMNITEV 500         //Cas med posameznimi stopnjami
#define LOCLJIVOST 100
#define TPMAKS (9*LOCLJIVOST)//Najvecji cas prozenja bo 9ms.
#define TPMIN LOCLJIVOST     //Najmanjsi cas prozenja bo 1ms.

unsigned int prehodnicle(unsigned int nicla);
void prozi(unsigned int cas);
int temnitev(int tp);        //Nastavljanje casa prozenja.

int main()
{
  unsigned int zadnjaNicla = 0;    //Hrani trenutek zadnjega prehoda skozi niclo.
  int tp = TPMIN;                 //Hrani trenutni cas prozenja.

  _TimerInit();              //Privzeta locljivost je 1ms.
  T0PR /= LOCLJIVOST;        //Locljivost povecamo.
  _KeyInit();
  _setpindir(4, 0);          //Vhod (uskladitveni signal)
  _setpindir(5, 1);          //Izhod (prozilni signal)

  while (1)
  {
    zadnjaNicla = prehodnicle(zadnjaNicla);
    prozi(zadnjaNicla + tp);
```

```
    tp = temnitev(tp);
  }
  return 0;
}

unsigned int prehodnicle(unsigned int nicla)
{
  int stanje;
  int static staro = 0;

  stanje = inportp(4);
  if (stanje != staro)
  {
    staro = stanje;
    if (stanje == 0)  //Ce zaznamo padec uskladitvenega signala,
    {
      nicla = clock();//si zapomnimo, kdaj se je to zgodilo in
      outportp(5, 0); //spustimo prozilni signal.
    }
  }
  return nicla;
}

void prozi(unsigned int cas)
{
  if (clock() > cas) //Ce je pravi cas,
  {
    outportp(5, 1);  //postavimo prozilni signal na 1.
  }
}

int temnitev(int tp)
{
  static int dt = 0;
  static int zadnjaspremememba = 0;
  if (kbhit()) //Ali je pritisnjena kaksna tipka?
  {
    switch(getch())
    {                          //Odslej se bo prozilni cas:
      case '0': dt = 1; break; //vecal
      case '1': dt = -1; break;//manjsal
      case '3': dt = 0; break; //se ne bo spreminjal
    }
```

```
  }
  //Ce je minilo dovolj casa od zadnje spremembe casa prozenja
  //(konstanta 500 je izbrana tako, da se tp spremeni od ene do
  //druge mejne vrednosti v 4 sekundah):
  if (clock() - zadnjaspremememba > TEMNITEV)
  {
    zadnjaspremememba += TEMNITEV;
    tp += dt;
    if (tp > TPMAKS) tp = TPMAKS;
    if (tp < TPMIN) tp = TPMIN;
  }
  return tp;
}
```

8.4 Primeri izpitnih vprašanj

1. Kakšno zahtevo mora izpolnjevati sistem v ralnem času?

   (a) Biti mora strašansko hiter.

   (b) Odzivati se mora v predpisanih časovnih okvirih.

   (c) Znati mora prikazovati točen čas.

   (d) Sposoben mora biti opravljati več reči hkrati.

2. Podan je del kode:

   ```
   odpri();
   start = clock();
   ```

   Kako lahko dosežemo, da se funkcija `zapri()` ne bo klicala prej kot v 15 sekundah?

   (a) `if (clock() - start < 15000) zapri();`

   (b) `if (clock() - start > 15000) zapri();`

   (c) `if (start - clock() > 15000) zapri();`

   (d) `if (start - clock() < 15000) zapri();`

9 Kazalci v treh enostavnih korakih

Kazalci (pointers) predstavljajo precej zahtevno poglavje vsakega programskega jezika, ki dopušča neposredno delo z njimi. Ko jih enkrat dobro razumemo in obvladamo, nam kazalci v zameno nudijo ogromno moč manipulacije s podatki in kodo. Čas nam žal ne dovoli, da bi se v okviru našega predmeta poglobljeno ukvarjali s kazalci, tematike se bomo le dotaknili.

9.1 Prvi korak k učenju kazalcev

Če želimo razumeti kazalce, moramo najprej razumeti vsa zakulisna dogajanja v naslednjem trivialnem programu:

```
#include "io.h"

int main(void)
{
  int x;
  _LCDInit();
  x = 42;
  printf("%d", x);

  while (1);
  return 0;
}
```

Vse dogajanje v programu je strnjeno v prvih štirih vrsticah funkcije `main()`. V prvi vrstici deklariramo celoštevilsko spremenljivko x. S tem v pomnilniku rezerviramo štiri pomnilniške celice (= 32 bitov), ki bodo služile izključno za hranjenje vrednosti spremenljivke x. V tretji vrstici v te štiri rezervirane celice vpišemo vrednost 42 (v obliki 32-bitnega predznačenega števila). V četrti vrstici s pomočjo funkcije `printf()` na prikazovalnik izpišemo vrednost, ki se nahaja v pomnilniku, rezerviranem za spremenljivko x. Ko program zaženemo, dobimo izpis:

```
42
```

S tem enostavnim primerom sem želel opozoriti na dejstvo, da, kadarkoli v programski kodi naletimo na ime kakšne spremenljivke, v resnici beremo ali pišemo v del pomnilnika, ki je rezerviran izključno za to spremenljivko.

Kje natančno je v pomnilniku rezerviran prostor za določeno spremenljivko, nas programerje velikokrat ne zanima. Na srečo lahko zaupamo prevajalniku, da bo za spremenljivko rezerviral kos še neuporabljenega pomnilnika, in s tem smo zadovoljni.

Včasih pa naletimo na operacije, ki na tak ali drugačen način potrebujejo informacijo o tem, kje v pomnilniku se določena spremenljivka nahaja. Z drugimi besedami, potrebujemo informacijo o njenem *naslovu*. Do naslova spremenljivke pridemo s pomočjo *naslovnega operatorja* (`&`), ki ga postavimo pred ime spremenljivke, kot kaže naslednji primer.

```
#include "io.h"

int main(void)
{
  int x;
  _LCDInit();
```

```
    x = 42;
    printf("%u", &x);

    while (1);
    return 0;
}
```

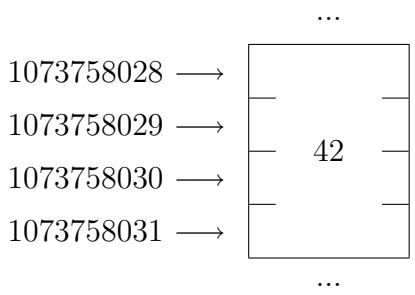
Primer se od prejšnjega razlikuje le v četrti vrstici, ki namesto vrednosti spremenljivke x izpiše njen naslov. Kot že vemo, je naslov nepredznačeno celo število, zato moramo za izpis uporabiti formatno dolo čilo `%u`.

Ko sem gornji program zagnal, je izpisal

```
1073758028
```

To pomeni, da je vrednost spremenljivke x shranjena v celicah z naslovi od 1073758028 do 1073758031. Dejanska vrednost, ki jo vrne izraz `&x`, je v resnici odvisna od konfiguracije sistema, na katerem zaganjamo program, vedno pa predstavlja pomnilniški naslov spremenljivke x.

Naslednja slika prikazuje stanje v pomnilniku tik pred klicem funkcije `printf()` v gornjem programu:



9.2 Drugi korak k učenju kazalcev

V poglavju o zbirkah smo omenili, da predstavlja ime zbirke brez indeksa naslov kosa pomnilnika, v katerem je zbirka shranjena. V resnici je to naslov prvega elementa zbirke, t.j. elementa z indeksom 0. Zdaj, ko smo uradno spoznali naslovni operator (address-of operator), lahko to dejstvo zapišemo tudi po cejevsko.

Vzemimo konkreten primer deklaracije zbirke 161 znakov:

```
char sms[161];
```

Potem je izraz

```
sms
```

popolnoma enakovreden izrazu

```
&sms[0]
```

Oba izraza predstavljata naslov prvega elementa zbirke.

Brez naslovnega operatorja lahko neposredno dobimo le naslov prvega elementa zbirke. V primeru znakovnega niza, ki je le poseben primer zbirke, ta naslov pogosto podajamo različnim funkcijam. Na primer, če želimo na prikazovalnik izpisati vsebino znakovnega niza *sms*, naredimo to takole:

```
printf(sms);
```

Funkcija `printf()` kot parameter dobi pomnilniški naslov prvega znaka v nizu. Nato od tega naslova dalje iz pomnilnika po vrsti jemlje znake in jih enega za drugim prikaže na prikazovalnik, dokler ne naleti na zaključni ničelni znak.

S tem znanjem in s pomočjo naslovnega operatorja, ki smo ga pred kratkim spoznali, si lahko privoščimo drobno vragolijo:

```
#include "io.h"

int main(void)
{
  _LCDInit();
  char sms[161] = "Niti pod razno ne priznam, da sem zabluzil!";
  printf(&sms[18]);
  while (1);
  return 0;
}
```

Funkciji `printf()` smo podali naslov devetnajstega znaka znakovnega niza *sms*. Funkcija seveda ne ve, od kod je dobila naslov in se tudi ne sprašuje, ali je naslov res naslov začetka kakšnega znakovnega niza. Funkcija začne z delom enostavno na naslovu, ki ga je dobila, in konča pri zaključnem ničelnem znaku.

Ko program zaženemo, bo izpisal tole:

```
priznam, da sem zabluzil!
```

Na takšen način se da preslepiti katerokoli funkcijo, ki kot parameter sprejema naslov. Za primer vzemimo funkcijo `strlen()`, ki vrne dolžino znakovnega niza. Vrednost izraza

```
strlen(sms)
```

je enaka 43. Toliko je v nizu namreč vseh znakov, dokler ne naletimo na zaključni ničelni znak. Izraz

```
strlen(&sms[39])
```

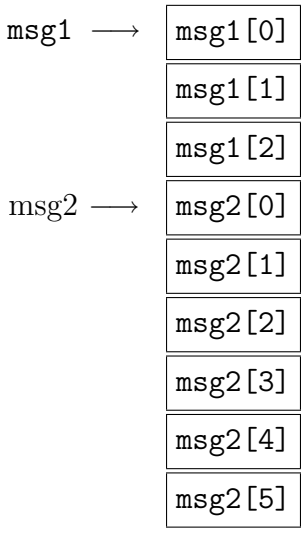
pa vrne vrednost 4. Funkcija začne namreč šteti pri štiridesetem znaku, katerega naslov je prejela kot argument.

To je vse zelo zabavno. Vendar se nam začne vedno pogosteje zastavljati vprašanje, kaj se zgodi, če po nesreči z indeksom presežemo meje, ki smo jih rezervirali za konkretno zbirko. Kaj takšnega se nam v primeru običajnih spremenljivk ne more zgoditi.

Da bi odgovorili na to vprašanje, si oglejmo naslednji program:

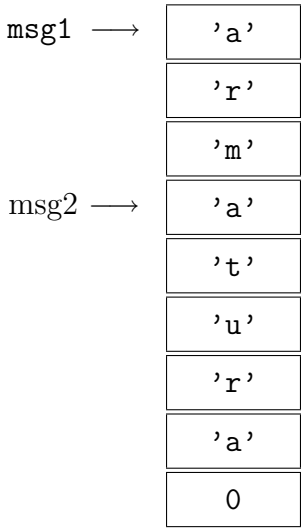
```
#include "io.h"
#include <string.h> //knjiznica, v kateri je funkcija strcpy()

char msg1[3], msg2[6];

int main(void)
{
  _LCDInit();
  strcpy(msg1, "armatura"); //Ups!
  strcpy(msg2, "7");
  printf(msg1);
  while (1);
  return 0;
}
```

V programu smo rezervirali prostor za dva znakovna niza, prvega dolžine 3 in drugega 6 znakov. Izkazalo se je, da je naš prevajalnik v pomnilniku izbral prostor za znake drugega niza takoj za prvim nizem, kar prikazuje naslednja slika:

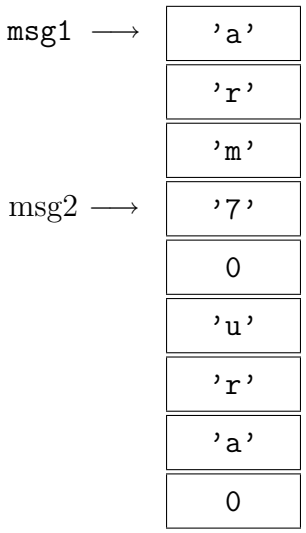


V niz *msg*1 v drugi vrstici funkcije `main()` vpišemo niz `"armatura"`. To storimo z uporabo knjižnične funkcije `strcpy()`. Ta funkcija kot parametra sprejme dva znakovna niza in znake

desnega niza prekopira v levega. Funkcija ne prejme nobene druge informacije razen naslovov začetkov obeh nizev. Kopiranje izvaja tako, da prepiše vsebino prve celice desnega niza v prvo celico levega niza, vsebino druge celice desnega niza v drugo celico levega niza in tako naprej vse do zaključnega ničelnega znaka, ki ga prepiše zadnjega. Ko funkcija `strcpy()` zaključi z delom, imamo v pomnilniku stanje, kot ga vidimo na naslednji sliki:



Za zadnjih 5 znakov niza `"armatura"` in zaključni ničelni znak je v pomnilniku, rezerviranem za niz *msg*1, zmanjkalo prostora. Vendar se funkcija `strcpy()` zaradi tega prav nič ne sekira, znake preprosto kopira naprej v pomnilnik, tako kot je naučena. Drugi klic funkcije `strcpy()` na podoben način v znakovni niz *msg*2 vpiše niz `"7"`. Začne seveda na naslovu `msg2`. Ko zaključi z delom, imamo v pomnilniku takšno stanje:



Nič čudnega, da program izpiše tole:

```
arm7
```

Klic funkcije `printf()` namreč na prikazovalnik prenese vse znake od naslova `msg1` do prvega zaključnega ničelnega znaka.

Razmislite, kakšen izpis bo povzročil klic

```
printf(msg2);
```

in kakšnega klic

```
printf(&msg2[1]);
```

Kaj bi se zgodilo, če bi med seboj zamenjali vrstni red obeh klicev funkcije `strcpy()`? Takole:

```
strcpy(msg2, "7");
strcpy(msg1, "armatura");
```

V primeru, ki smo ga pravkar videli, smo znake po nesreči zapisovali v pomnilnik, ki ni bil rezerviran za zbirko, ki smo jo uporabljali. Ko boste primer skušali zagnati sami, ni nujno, da boste zasledili enako obnašanje. V splošnem so posledice takšnih napak zelo nepredvidljive, od tega, da deluje vse tako, kot je treba, pa do tega, da se sesuje sistem.

Vse težave so v resnici posledica dejstva, da lahko pridemo z nepazljivo uporabo zbirk do ene in iste pomnilniške celice na več načinov. Tako na primer izraz `msg1[3]` predstavlja isto pomnilniško celico kot izraz `msg2[0]`.

Kar pa je za neukega težava, je lahko za izkušenega in spretnega programerja prednost. Kazalci nam omogočajo dodatne načine dostopanja do delov pomnilnika in z ustreznim znanjem postanejo izjemno prefinjeno orodje v rokah programerja.

9.3 Tretji korak k učenju kazalcev

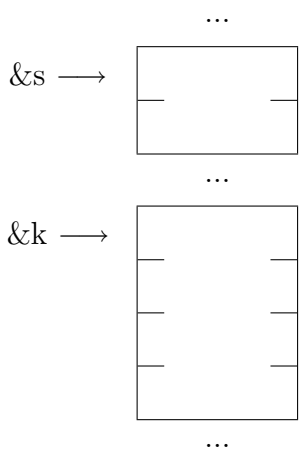
Kazalec je *spremenljivka, katere vrednost je pomnilniški naslov*. To je zelo preprosta definicija, ki pa ima za posledico precej kompleksen nabor različnih načinov upravljanja s kazalci. Mnogo prekompleksen za začetnika. Vendar brez strahu. Dotaknili se bomo le najosnovnejših pojmov.

Za izhodišče razprave vzemimo nasleden primer:

```
#include "io.h"

int main(void)
{
  short s, *k;
  _LCDInit();
  s = 3;
  k = &s;
  printf("%d %d\n\r", s, *k);
```

```
    *k = 10;
    printf("%d", s);
    while (1);
    return 0;
}
```

V prvi vrstici funkcije `main()` smo deklarirali dve spremenljivki. Spremenljivka s je običajna predznačena 16-bitna celoštevilska spremenljivka (`short`). Spremenljivka k je kazalec. Velja naslednje pravilo: *kadar pri deklaraciji pred ime spremenljivke postavimo zvezdico (*), je ta spremenljivka kazalec.* Ob deklaraciji omenjenih dveh spremenljivk se v pomnilniku rezervira nekaj prostora, kakor vidimo na naslednji sliki:



Ker je spremenljivka s 16-biten podatek, sta se zanjo rezervirali dve pomnilniški celici. Spremenljivka k je kazalec, ki hrani 32-biten pomnilniški naslov, zato zanjo potrebujemo 4 pomnilniške celice.

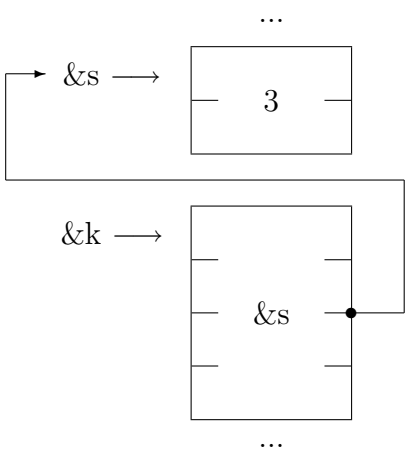
V tretji vrstici vpišemo vrednost 3 v celici, ki sta namenjeni spremenljivki s:

```
s = 3;
```

Vrstico za tem vpišemo naslov spremenljivke s v 4 celice, rezervirane za spremenljivko k:

```
k = &s;
```

Posledica je naslednje stanje v pomnilniku:



Puščica, speljana od kazalca k proti spremenljivki s shematično nakazuje dejstvo, da v kazalcu hranimo naslov spremenljivke s. Pravimo, da kazalec *kaže* na spremenljivko s.

V peti vrstici funkcije `main()` izpišemo vrednost spremenljivke s in vrednost izraza `*k`:

```
printf("%d %d\n\r", s, *k);
```

Zvezdica (`*`) pred kazalcem k igra v tem primeru vlogo *operatorja indirekcije* (indirection operator). Kadar pred ime kazalca v programu postavimo ta operator, se namesto kazalca uporabi spremenljivka, na katero kazalec kaže.

V našem primeru bo zato gornji klic funkcije `printf()` izpisal:

```
3 3
```

Kazalec k namreč kaže na spremenljivko s. Da je temu res tako, se še dodatno prepričamo v naslednjih dveh vrsticah programa:

```
*k = 10;
printf("%d", s);
```

Desetico vpišemo v spremenljivko, na katero kaže kazalec k. Ker je to spremenljivka s, bo funkcija `printf()` izpisala vrednost 10.

Odgovorimo na koncu še na vprašanje, ki si ga je morda kdo izmed vas, pazljivih bralcev zastavil: Rekli smo, da kazalec vedno hrani pomnilniški naslov, ki je 32-bitno nepredznačeno celo število. Zakaj moramo potem pri deklaraciji podati tip kazalca? In zakaj ta tip ni 32-bitno nepredznačeno celo število? V resnici pomeni tip kazalca tip podatka, na katerega kazalec kaže. To je zelo pomembna informacija. Sicer ne bi mogli uporabiti operatorja indirekcije, saj ne bi bilo jasno, koliko pomnilniških celic je potrebno nasloviti.

9.4 Še nekaj ‚fint'

Poglejmo si še nekaj osnovnih reči, ki jih je v zvezi s kazalci dobro vedeti.

9.4.1 Neinicializiran kazalec

Kadar uporabljamo kazalec, da prek njega dostopamo do podatkov v pomnilniku, se moramo vedno prepričati, če vemo, kam ta kazalec kaže. Naslednji kos programa ne bo povzročil nobenih težav pri prevajanju. Če pa program s takšno kodo zaženemo, se lahko zgodijo različne nepredvidene reči:

```
 ...
short *k;
*k = 42;
 ...
```

Vrednost 42 pišemo v spremenljivko, na katero kaže kazalec. Toda kam kazalec kaže? Nekam že, a nimamo niti najmanjšega pojma, kam. Vrednost se bo vpisala v nek naključen del pomnilnika in posledic ne moremo predvideti. Lahko, da bo vse v redu, lahko program ne bo dal pravilnih rezultatov, lahko pa se sesuje sistem.

Podoben problem imamo tu:

```
 ...
char *k;
strcpy(k, "UPS!");
 ...
```

9.4.2 Kazalci in zbirke

Kazalci in zbirke so med sabo tesno povezani. Ugotovili smo že, da predstavlja ime zbirke naslov prvega elementa v zbirki. Do posameznih elementov dostopamo z zaporedno številko, ki jo postavimo v oglate oklepaje. Do spremenljivke, na katero kaže kazalec, dostopamo s pomočjo operatorja indirekcije.

Vzemimo za primer del kode:

```
 ...
char t[] = "Uf, to pa je."
char *k;
k = t;
 ...
```

V tretji vrstici smo usmerili kazalec k na prvi element znakovnega niza t. Z znanjem, ki smo ga osvojili doslej, znamo priti do tega prvega elementa na dva načina. Izraza `*k` in `t[0]` imata oba vrednost `'U'`. Indeks v oglatih oklepajih pa lahko uporabimo tudi s kazalcem. Tako za kazalec k velja:

$$*k \equiv k[0]$$

in bolj splošno:

$$*(k+n) \equiv k[n]$$

Če kazalcu prištejemo vrednost n in nad vsoto napravimo operacijo indirekcije, pridemo, do $n+1$-vega elementa v zbirki. Glede na to, da v splošnem posamezni elementi zasedejo več kot eno pomnilniško celico, mora očitno veljati naslednje:

Če kazalcu prištejemo celoštevilsko vrednost n, se vrednost kazalca poveča za n, pomnožen s številom celic, ki jih zaseda podatek, na katerega kazalec kaže.

Vzemimo za primer kazalec, ki kaže na spremenljivko tipa `float`, ki v pomnilniku zaseda 4 celice. Če temu kazalcu prištejemo vrednost 5, se bo njegova vrednost povečala za $5 * 4 = 20$.

9.4.3 Konstanten znakovni niz

Konstante v jeziku C so izrazi, izrazi pa imajo številsko vrednost. Vemo že, da ima izraz `5` vrednost 5, in izraz `'5'` vrednost 53 (koda ASCII znaka 5). Vprašanje je, kakšno vrednost ima izraz `"5"` (konstanten znakovni niz)? Vemo, da ta izraz povzroči, da se v pomnilnik shranita dve vrednosti: 53 (koda ASCII znaka 5) in 0 (zaključni ničelni znak). Ampak to sta dve vrednosti. Izraz ne more imeti dveh vrednosti hkrati.

V jeziku C konstanten znakovni niz vrne pomnilniški naslov, na katerem je ta niz shranjen. V naslednjem delu programa nastopata dva konstantna znakovna niza, ki sta enaka, vendar očitno shranjena vsak v svojem delu pomnilnika:

```
 ...
char *k;
k = "programiranje";

if (k == "programiranje")
{
 printf("Enako.");
}
else
{
  printf("Ni enako.");
}
 ...
```

Izpisalo se bo sporočilo `"Ni enako"`. V kazalec k smo namreč v drugi vrstici shranili naslov prvega niza `"programiranje"`, potem pa smo njegovo vrednost primerjali z naslovom drugega niza `"programiranje"`. Ta naslov je seveda drugačen.

Če želimo med seboj primerjati dva znakovna niza, moramo primerjati posamezne znake obeh nizev. To za nas naredi funkcija `strcmp()`. Ta funkcija vrne 0 v primeru, da sta niza, ki ju dobi kot parametra, enaka:

```
 ...
char *k;
k = "programiranje";
```

```
if (strcmp(k, "programiranje") == 0)
{
  printf("Enako."); //to se bo izvrsilo
}
 ...
```

Bolj zahtevni bralci boste verjetno želeli razmisliti, zakaj koda, ki smo jo zasledili v razdelku o neinicializiranih kazalcih, povzroči nepredvideno obnašanje programa (pisanje v naključen del pomnilnika):

```
 ...
char *k;
strcpy(k, "UPS!");
 ...
```

In zakaj je naslednja koda povsem legitimna in ne povzroči nobenih problemov.

```
 ...
char *k;
k = "programiranje";
 ...
```

9.5 Praktična uporaba kazalcev

V dveh primerih smo kazalce s pridom uporabili, še preden smo se jih učili. Poglejmo si zdaj ta dva primera v novi luči.

9.5.1 Uporaba registrov

Kadar želimo brati ali pisati vrednosti, ki se nahajajo v hardverskih registrih, moramo to nujno početi prek naslovov. Vsak register se namreč nahaja na točno določenem naslovu. Prevajalnik teh naslovov ne pozna, priskrbeti jih mora programer.

Kot primer vzemimo register IO0PIN, ki je na šestnajstiškem naslovu e0028000. Za dostop do njega imamo v knjižnici `io.h` zapisan makro

```
#define IO0PIN  (*((unsigned int *)0xE0028000))
```

Izraz na desni strani si razložimo na naslednji način. Najprej moramo zahtevati pretvorbo tipa konstantne 32-bitne vrednosti 0xE0028000 v kazalec na podatek tipa `unsigned int`:

```
(unsigned int *)0xE0028000
```

Zdaj imamo kazalec, ki kaže na podatek na naslovu e0028000. Z operatorjem indirekcije pridemo do tega podatka, t.j. registra IO0PIN:

```
*((unsigned int *)0xE0028000)
```

9.5.2 Podajanje parametrov funkcijam

Kadar želimo funkciji kot parameter podati zbirko, to vedno storimo tako, da podamo naslov začetka zbirke. Če gre za poljubno zbirko, moramo navadno podati vsaj še en parameter, ki pove, koliko elementov ima zbirka. V primeru znakovnega niza to ni potrebno, saj je konec niza označen z zaključnim ničelnim znakom.

Kadar funkciji podamo naslov, pravimo, da podajamo parameter po *referenci*. Sicer ga podajamo po *vrednosti*.

Oglejmo si to idejo na konkretnem zgledu funkcije, za katero želimo, da med seboj zamenja vrednosti dveh parametrov. Tule je definicija funkcije:

```
void menjaj(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

Če to funkcijo kličemo z namenom, da bi med sabo zamenjali vrednosti spremenljivk x in y, ne bo uspeha:

```
 ...
int x = 10, y = 20;
menjaj(x, y);
//tu je x se vedno 10 in y se vedno 20
 ...
```

Funkciji `menjaj()` smo podali le *vrednosti* obeh spremenljivk, ki sta se prekopirali v parametra a in b. Med seboj smo zamenjali vrednosti teh dveh parametrov, originalni vrednosti spremenljivk x in y sta ostali nedotaknjeni.

Popravimo nekoliko definicijo funkcije `menjaj()`:

```
void menjaj(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

Popraviti moramo tudi klic, saj funkcija zdaj za oba parametra zahteva naslova spremenljivk:

```
 ...
int x = 10, y = 20;
menjaj(&x, &y);
//zdaj je x enak 20 in y enak 10
 ...
```

Tokrat nam je uspelo. Funkciji smo podali naslova spremenljivk x in y. Ta naslova sta se zapisala v kazalca a in b. Kazalec a zdaj kaže na spremenljivko x in kazalec b na spremenljivko y. Z operatorji indirekcije smo prek teh dveh kazalcev prišli do originalnih spremenljivk x in y ter ju tudi spremenili.

Kadarkoli torej želimo dopustiti funkciji možnost, da trajno spremeni vrednosti podanih parametrov, moramo podati *naslove* teh parametrov. Pravimo, da podamo parametre po referenci. Zbirke moramo v vsakem primeru podati po referenci, saj lahko le na ta način z enim samim parametrom podamo celotno zbirko. Prav to smo mi ves čas počeli z znakovnimi nizi. Funkcijam smo vedno podajali naslov začetka znakovnega niza.

9.6 Primeri izpitnih vprašanj

1. Spremenljivka x v pomnilniku zasede 4 pomnilniške celice. Nahaja se na naslovu 2000456 in ima vrednost 42. Kakšna je vrednost izraza `&x`?

   (a) 2000456

   (b) 42

   (c) 4

   (d) Nobena od naštetih.

2. Podana je deklaracija `char t[10] = "abcd";` Kakšna bo vrednost izraza `strlen(&t[1])`

   (a) 3

   (b) 4

   (c) 9

   (d) 10

3. Podan je del kode

   ```
   char t[10];
   strcpy(t, "abcd");
   ```

   Kakšna bo vrednost izraza `strlen(&t[5])`

   (a) 4

   (b) 5

   (c) 10

   (d) Vrednosti ni mogoče določiti.

4. Podan je del kode

```
float f = 15, *k;
k = &f;
*k = 10;
```

Kakšna bo vrednost spremenljivke f, ko se koda izvede?

(a) 0

(b) 10

(c) 15

(d) Vrednosti ni mogoče določiti.

5. Podan je del kode

```
float f = 15, *k;
*k = 10;
k = &f;
```

Kakšna bo vrednost spremenljivke f, ko se koda izvede?

(a) 0

(b) 10

(c) 15

(d) Vrednosti ni mogoče določiti.

6. Podan je del kode

```
float f = 15, *k;
//tukaj
*k = 10;
```

Kaj moramo dodati na mesto komentarja (tukaj), da bo imela na koncu spremenljivka f vrednost 10?

(a) `k = &f;`

(b) `&k = f;`

(c) `f = &k`

(d) `&f = k`

7. Imamo kazalec k, ki kaže na spremenljivko x tipa `long`. Kaj se zgodi, ko se izvede stavek `k = k + 1;`?

(a) Vrednost kazalca k se poveča za ena.

(b) Vrednost kazalca k se poveča za štiri.

(c) Vrednost spremenljivke x se poveča za ena.

(d) Vrednost spremenljivke x se poveča za štiri.

8. Imamo kazalec k, ki kaže na spremenljivko x tipa `long`. Kaj se zgodi, ko se izvede stavek `*k = *k + 1;`?

   (a) Vrednost kazalca k se poveča za ena.

   (b) Vrednost kazalca k se poveča za štiri.

   (c) Vrednost spremenljivke x se poveča za ena.

   (d) Vrednost spremenljivke x se poveča za štiri.

9. Podan je del kode

```
int z[] = {4, 2}, *k;
k = z;
*k = z[0];
```

   Kakšna bo vrednost zbirke z, ko se koda izvede?

   (a) {2, 2}

   (b) {2, 4}

   (c) {4, 2}

   (d) {4, 4}

10. Podan je del kode

```
int z[] = {4, 2, 1}, *k;
k = z;
*(k + 2) = (*k) + 2;
```

    Kakšna bo vrednost elementa $z[2]$, ko se koda izvede?

    (a) 1

    (b) 3

    (c) 4

    (d) 6

11. Podan je del kode

```
int z[10], *k;
k = z;
```

Kako s pomočjo kazalca k postavimo drugi element zbirke z na 5?

(a) `&(k + 1) = 5;`

(b) `*(k + 1) = 5;`

(c) `(k + 1) = 5;`

(d) `(*k) + 1 = 5;`

12. Kakšna je vrednost izraza `"arm7" == "arm7"`?

(a) 0

(b) 1

(c) `"arm7"`

(d) `'a'`

13. Podan je prototip funkcije `void spremeni(int *x);` Kako to funkcijo pravilno kličemo, če imamo v programu deklarirano spremenljivko z, ki je tipa `int`?

(a) `spremeni(&z);`

(b) `spremeni(*z);`

(c) `spremeni(z);`

(d) Noben od naštetih klicev ni pravilen.

14. Kakšen mora biti prototip funkcije, ki sprejme kot parameter znakovni niz?

(a) `void f(char niz);`

(b) `char f(void);`

(c) `void f(char *niz);`

(d) `void f(char &niz);`

A Opisi uporabljenih funkcij

V tem poglavju si bomo ogledali funkcije, ki jih bomo uporabljali pri predmetu. Bralec, ki ga zanimajo tudi druge funkcije jezika C (ANSI) ali katerekoli njegove različice, bo to informacijo našel v ustrezni literaturi.

A.1 ANSI C

To so standardne funkcije, ki naj bi jih podpiral sleherni prevajalnik za C. Nekatere so nekoliko prilagojene za naš sistem.

A.1.1 `unsigned long clock(void);`

Knjižnice: `"io.h" (Š-ARM), <time.h> (Microsoft Visual C++)`
Inicializacija: `_TimerInit()`

Opis: Vrne število tisočink sekunde, ki so pretekle od klica funkcije `_TimerInit()`. V okolju Visual C++ funkcija `clock()` vrne število tisočink sekunde, ki so pretekle od zagona programa.

A.1.2 `int printf(const char *format [, argument, ...]);`

Knjižnice: `"io.h" (Š-ARM), <stdio.h> (Microsoft Visual C++)`
Inicializacija: `_LCDInit()`

Opis: Funkcija `printf()` na prikazovalniku izpiše formatirano besedilo. Pri tem poleg običajnega besedila uporablja tudi tako imenovane *ubežne sekvence* in *formatna določila*. Formatno določilo opravlja podobno funkcijo kot prazno mesto v formularju, ki ga je potrebno izpolniti naknadno. Formatna določila se ne prikažejo na prikazovalniku, ampak se na njihovih mestih izpišejo vrednosti ustreznih izrazov, kakor to prikazuje spodnji primer. Formatno določilo je sestavljeno iz znaka % in ustrezne črke, odvisno od tega, kakšen tip podatka želimo pisati. Med % in črko lahko po želji vstavimo tudi številko, ki ima različne pomene, odvisno od samega formatnega določila.

Nekaj formatnih določil je zbranih v naslednji tabeli. Črki `m` in `n` predstavljata poljubni številski vrednosti.

Formatno določilo	**Podatkovni tip**
`%d %i`	Predznačeno celo število.
`%u`	Nepredznačeno celo število.
`%x`	Nepredznačeno šestnajstiško število. Cifre z vrednostjo nad 9 se izpisujejo z malimi črkami.
`%X`	Nepredznačeno šestnajstiško število. Cifre z vrednostjo nad 9 se izpisujejo z velikimi črkami.

Formatno določilo	Podatkovni tip
%nd %ni %nu %nx %nX	Celo število izpisano na n mestih. Po potrebi se z leve dodajajo presledki.
%0nd %0ni %0nu %0nx %0nX	Celo število izpisano na n mestih. Po potrebi se z leve dodajajo ničle.
%f	Realno število.
%m.nf	Realno število izpisano na m mestih, od česar je n decimalnih mest.
%c	Znak.
%s	Znakovni niz.

Ubežne sekvence ne povzročijo nobenega posebnega izpisa. Izpis le oblikujejo ali prestavijo kurzor. Funkcija `printf()` pozna iste ubežne sekvence kot funkcija `putch()`:

Znak	Pomen
\n	Prehod v novo vrstico (newline).
\r	Pomik na začetek vrstice (return).
\b	Pomik za eno mesto nazaj in izbris znaka (backspace).

Primer:

```
#include "io.h"

int main(void)
{
  int stevilka = 2;
  char znak = 'w';
  _LCDInit();
  printf("%d znaki %c pomenijo svetovni splet.", stevilka + 1, znak);
  while (1);
  return 0;
}
```

V gornjem programu funkcija `printf()` na prikazovalniku izpiše besedilo, ki je v narekovajih. Pri tem se vsako od obeh formatnih določil nadomseti z vrednostjo enega od izrazov, ki sledijo. Izraz `stevilka + 1` ima vrednost 3, ki se izpiše na mestu prvega formatnega določila, izraz `znak` pa ima vrednost 'w', ki se izpiše na mestu drugega formatnega določila.

Na prikazovalniku se tako prikaže besedilo "3 znaki w pomenijo svetovni splet."

A.1.3 `void putch(int ch);`

Knjižnice: `"io.h"` (Š-ARM), `<conio.h>` (Microsoft Visual C++)
Inicializacija: `_LCDInit()`

Opis: Funkcija `putch()` izpiše na zaslon na mesto kurzorja znak, katerega kodo (ASCII) ji podamo kot argument. Po končanem izpisu prestavi kurzor za eno mesto naprej. Na sistemu Š-ARM deluje ta funkcija za znake s kodami med vključno 32 (presledek) in 125 (znak `'}'`). Poleg tega razpozna tudi tako imenovane *ubežne sekvence* `'\n'` - prehod v novo vrstico (newline), `'\r'` - pomik na začetek vrstice (return) in `'\b'` - pomik za eno mesto nazaj in izbris znaka (backspace).

Primer:

```
#include "io.h"

int main(void)
{
  char znak = (char) 97;
  _LCDInit();
  putch(znak);
  putch('a');
  putch('\n');
  putch(97);
  while (1);
  return 0;
}
```

Primer na prikazovalnik izpiše tri a-je (črka 'a' ima kodo ASCII 97). Tretjega izpiše v drugi vrstici, ker se pred tem pomaknemo eno vrstico niže.

Izpis na prikazovalniku bo takšen:

```
aa
  a
```

A.1.4 `int puts(const char *s);`

Knjižnice: `"io.h" (Š-ARM), <stdio.h> (Microsoft Visual C++)`
Inicializacija: `_LCDInit()`

Opis: Funkcija `puts()` znakovni niz, ki ji ga podamo kot argument, izpiše na zaslon. Na koncu izpisa se kurzor pomakne v začetek nove vrstice. Različica za Š-ARM obdrži kurzor tam, kjer se je izpis končal in ne gre v novo vrstico.

Primer:

```
#include "io.h"

int main(void)
{
  char tekst[] = "Prva vrstica.";
```

```
  _LCDInit();
  puts(tekst);
  puts("\r\nDruga vrstica.");
  while (1);
  return 0;
}
```

Tukaj prvi klic funkcije `puts()` na zaslon najprej izpiše besedilo, ki je shranjeno v znakovnem nizu `tekst`. Drugi klic funkcije `puts()` najprej povzroči pomik kurzorja na začetek vrstice (znak `'\r'`), potem v novo vrstico (znak `'\n'`), na koncu pa se izpiše še preostalo besedilo, ki smo ga funkciji podali kot konstanten znakovni niz.

Končen izpis na prikazovalniku je torej:

```
Prva vrstica.
Druga vrstica.
```

A.1.5 `int rand(void);`

Knjižnice: `<stdlib.h>`

Opis: Funkcija `rand()` vrne naključno vrednost med 0 in `RAND_MAX`. `RAND_MAX` je enaka največji možni vrednosti, ki jo lahko ima podatek tipa `int`, kar je običajno 32767. Generator deluje tako, da po nekem algoritmu vsako naslednjo naključno vrednost izračuna iz prejšnje. Problem je s prvo vrednostjo, ki jo navadno dobimo tako, da pred prvim klicem te funkcije generator naključnih števil inicializiramo s funkcijo `srand()`.

Primer:

```
#include "io.h"
#include <stdlib.h>

int main(void)
{
  int i;
  _TimerInit();
  _KeyInit();
  _LCDInit();
  printf("Pritisni tipko.\r");
  getch();
  srand(clock()); //Stevilo milisekund, ki jih vrne clock()
                  //je odvisno od tega, kdaj pritisnemo tipko,
                  //in je prakticno nakljucno. Tako
                  //bomo imeli vsakokrat, ko zazenemo
                  //program, drugo zacetno vrednost.
```

```
  for (i = 0; i < 5; i++)
  {
   printf("%3d", rand() % 10);
  }
  while (1);
  return 0;
}
```

Gornji primer po pritisku na poljubno tipko na prikazovalnik izpiše pet naključnih vrednosti med 0 in 9.

A.1.6 `void srand(unsigned int seed);`

Knjižnice: `<stdlib.h>`

Opis: Funkcija `srand()` nastavi začetno vrednost generatorja naključnih števil.

A.1.7 `int strcmp(const char *s1, const char *s2);`

Knjižnice: `<string.h>`

Opis: Funkcija `strcmp()` primerja dva znakovna niza. Primerja najprej prvi znak niza $s1$ s prvim znakom niza $s2$. Potem primerja drugi znak niza $s1$ z drugim znakom niza $s2$ in tako dalje. Konča takoj, ko se kodi ASCII dveh znakov med seboj razlikujeta, ali ko doseže zaključni ničelni znak.

Če je $s1$ manjši od $s2$, funkcija vrne vrednost, manjšo od nič, če sta niza enaka, vrne nič, in če je niz $s1$ večji od niza $s2$, vrne vrednost, večjo od nič.

Primer:

```
#include "io.h"
#include <string.h>

int main(void)
{
  char ime[10] = "MIHA";
  _LCDInit();

  if (strcmp(ime, "Miha") == 0)
  {
    printf("Imeni sta enaki");
  }
  else
  {
```

```
    //Program bo izpisal tole, ker
    //imajo male crke drugacne kode
    //ASCII kot velike crke:
    printf("Imeni sta razlicni");
  }
  while (1);
  return 0;
}
```

A.1.8 `char *strcpy(char *dest, const char *src);`

Knjižnice: `<string.h>`

Opis: Funkcija `strcpy()` prekopira znakovni niz *src* v niz *dest*. Kopiranje konča takoj, ko prekopira zaključni ničelni znak.

Primer:

```
#include "io.h"
#include <string.h>

int main(void)
{
  char s[10];
  _LCDInit();

  strcpy(s, "Yo!");
  printf(s); //izpise Yo!
  while (1);
  return 0;
}
```

A.1.9 `size_t strlen(const char *s);`

Knjižnice: `<string.h>`

Opis: Funkcija `strlen()` prešteje število znakov v znakovnem nizu *s*, pri čemer ne šteje zaključnega ničelnega znaka.

Primer:

```
#include "io.h"
#include <string.h>
```

```
int main(void)
{
  char s[10] = "ARM7";
  _LCDInit();

  printf("%d", strlen(s)); //izpise 4
  while (1);
  return 0;
}
```

A.2 Funkcije, ki se zgledujejo po funkcijah DOS

Te funkcije niso del standarda ANSI. Poznajo jih različni prevajalniki kot del orodij za pisanje konzolnih aplikacij. To so programi, ki z uporabnikom komunicirajo izključno v tekstovnem načinu.

A.2.1 `void _setcursortype(int cur_t);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_LCDInit()`

Opis: Spremeni izgled kurzorja. Sprejme enega od treh možnih argumentov:

`_NOCURSOR`	izklopi kurzor
`_NORMALCURSOR`	črtica spodaj (_)
`_SOLIDCURSOR`	utripajoč pravokotnik

A.2.2 `void clrscr(void);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_LCDInit()`

Opis: Pobriše prikazovalnik in postavi kurzor v gornji levi kot.

A.2.3 `void delay(unsigned int milliseconds);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_TimerInit()`

Opis: Zaustavi delovanje programa za podano število milisekund.

Primer:

```
#include "io.h"

int main(void)
```

```
{
  char mlincek[] = "|/-\\";
  int i = 0;
  _LCDInit();
  _TimerInit();
  _setcursortype(_NOCURSOR);
  while(1)
  {
    printf("%c\r", mlincek[i]);
    delay(300);
    i++;
    i = i % 4;
  }
  return 0;
}
```

Primer ciklično prikazuje štiri znake, shranjene v znakovnem nizu `mlincek`: pokončno črto, poševnico, pomišljaj in poševnico nazaj. Poševnico nazaj moramo zapisati z dvema poševnicama, ker ena sama predstavlja začetek ubežne sekvece. Dobimo animacijo, ki daje videz vrteče se palice. Med posameznimi izpisi kličemo funkcijo `delay()`, ki povzroči, da program vsakokrat obstane za 300 milisekund. Če tega ne bi storili, bi se znaki prehitro menjavali in ne bi videli ničesar drugega kot migetajočo packo.

A.2.4 `int getch(void);`

Knjižnice: `"io.h" (Š-ARM), <conio.h> (Microsoft Visual C++)`
Inicializacija: `_KeyInit()`

Opis: Funkcija `getch()` zaustavi izvajanje programa, dokler ne pritisnemo kakšne tipke. Potem vrne kodo (ASCII) tipke, ki smo jo pritisnili. V sistemu Š-ARM imamo le tipke T0 do t3, ki vračajo kode med 48 in 51, kar ustreza znakom med `'0'` in `'3'`. Da bo funkcija delovala, morate natakniti jumper J16, ki priključi tipke na ustrezna vrata mikrokrmilnika.

Primer:

```
#include "io.h"

int main(void)
{
  char tipka;
  _LCDInit();
  _KeyInit();
  while (1)
  {
```

```
    tipka = getch();
    putch(tipka + 1);
  }
  return 0;
}
```

Primer v neskončni zanki vsakič najprej prebere kodo pritisnjene tipke, potem pa izpiše znak, ki ima za eno večjo kodo od pritisnjene tipke. Tako se izpisujejo znaki med `'1'` in `'4'`.

A.2.5 `void gotoxy(int x, int y);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_LCDInit()`

Opis: Funkcija postavi kurzor v stolpec x in vrstico y prikazovalnika. Na primer v levi gornji kot prikazovalnika postavimo kurzor s klicem `gotoxy(1,1);`

A.2.6 `int inportp(unsigned int pinid);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_setpindir()`

Opis: Funkcija prebere stanje vrat, ki morajo biti določena kot vhod. S parametrom *pinid*, ki ima lahko vrednosti med 0 in 15, izberemo želena vrata med P0.0 do P0.15.

Primer:

```
_setpindir(3, 0); //vrata P0.3 nastavimo kot vhod
vredn = inportp(3); //preberemo stanje na sponki
```

A.2.7 `int kbhit(void);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_KeyInit()`

Opis: Funkcija `kbhit()` v primeru, da je bila pritisnjena kakšna tipka, vrne vrednost različno od nič, sicer vrne 0. Klic funkcije se v vsakem primeru takoj zaključi in ne čaka na pritisk tipke. Funkcija je uporabna, kadar želimo, da program med čakanjem na pritisk tipke počne še kaj drugega. Šele ko se prepričamo, da je bila tipka pritisnjena, njeno kodo preberemo s funkcijo `getch()`.

Primer:

```
while (1)
{
```

```
    if (kbhit())
    {
       tipka = getch();
    }
    //tu pocnemo druge reci
}
```

A.2.8 `void outportp(unsigned int pinid, int value);`

Knjižnice: `"io.h"` (Š-ARM)
Inicializacija: `_setpindir()`

Opis: Funkcija prebere stanje vrat, ki morajo biti določena kot vhod. S parametrom *pinid*, ki ima lahko vrednosti med 0 in 15, izberemo želena vrata med P0.0 do P0.15.

Primer:

```
_setpindir(3, 1); //vrata P0.3 nastavimo kot izhod
outportp(3, 0); //na sponki se bo pojavila nula
//...
outportp(3, 1); //na sponki se bo pojavila enka
delay(10); //cakamo 10ms
outportp(3, 0); //na sponki se bo pojavila nula
```

Gornji primer povzroči na sponki P0.3 pravokoten impulz širime 10ms.

A.3 Razni primitivi

A.3.1 `int _adconvert(void);`

Knjižnice: `"io.h"` (Š-ARM)
Inicializacija: `_ADCInit()`

Opis: Vrne 10-bitno nepredznačeno vrednost (med 0 in 1023), ki se v trenutku klica nahaja na izhodu analogno digitalnega pretvornika, ki zajema podatke z vhoda AD0.0 (sponka P0.27). Na ta vhod lahko z jumperjem J20 priključimo potenciometer, ki je pritrjen na sistemski plošči.

Primer:

```
#include "io.h"

int main(void)
{
  int vrednost;
  _LCDInit();
```

```
  _ADCInit();
  while (1)
  {
    vrednost = _adcconvert();
    printf("Napetost:%4d\r", vrednost);
  }
  return 0;
}
```

Primer v neskončni zanki bere vrednost z izhoda analogno digitalnega pretvornika in jo izpisuje na prikazovalnik. Če imate na sistemu Š-ARM nataknjen jumper J20, potem boste z obračanjem potenciometra lahko opazovali, kako se izpisana vrednost spreminja od 0 do 1023.

A.3.2 `int _adconvertExt(int input);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: ni potrebna

Opis: Vrne 10-bitno nepredznačeno vrednost (med 0 in 1023), ki se v trenutku klica nahaja na izhodu izbranega analogno digitalnega pretvornika. Vrednost parametra `input` je lahko med 0 in 3 za vhode med AD0.0 in AD0.3 (sponke P0.27 - P0.31).

Funkcija je razširitev funkcije `_adconvert()`, zato je klic `_adconvertExt(0)` ekvivalenten klicu `_adconvert()`. Razlika med obema klicema je ta, da pred klicem funkcije `_adconvertExt()` ni potrebno inicializirati AD pretvornika.

A.3.3 `void _clrleds(int state);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_LEDInit()`

Opis: Ugasne diode LED, ki ustrezajo položajem enic vrednosti parametra *state*, zapisani v dvojiškem zapisu.

Primer:

```
#include "io.h"

int main(void)
{
  _LEDInit();
  _clrleds(9);
  while (1);
```

```
  return 0;
}
```

Če zaženemo gornji primer, bosta ugasnili (če nista že ugasnjeni) diodi LD0 in LD3. Vrednost 9 v dvojiškem zapisu je namreč 1001. Diodi LD1 in LD2 svojega stanja ne bosta spremenili. Da bo primer deloval, morate natakniti jumper J15, ki omogoči diode LED.

A.3.4 `void _setleds(int state);`

Knjižnice: `"io.h" (Š-ARM)`
Inicializacija: `_LEDInit()`

Opis: Prižge diode LED, ki ustrezajo položajem enic vrednosti parametra *state*, zapisani v dvojiškem zapisu.

Primer:

```
#include "io.h"

int main(void)
{
  _LEDInit();
  _setleds(9);
  while (1);
  return 0;
}
```

Če zaženemo gornji primer, se bosta prižgali (če nista že prižgani) diodi LD0 in LD3. Vrednost 9 v dvojiškem zapisu je namreč 1001. Diodi LD1 in LD2 svojega stanja ne bosta spremenili. Da bo primer deloval, morate natakniti jumper J15, ki omogoči diode LED.

A.4 Inicializacija hardvera

Vhodno izhodne naprave, ki jih želimo uporabiti, moramo najprej primerno pripraviti (inicializirati). Vsako napravo, ki jo bomo potrebovali, na začetku programa inicializiramo z ustrezno funkcijo.

A.4.1 `void _ADCInit(void);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Pripravi analogno digitalni pretvornik za uporabo.

A.4.2 `void _KeyInit(void);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Pripravi tipke za uporabo.

A.4.3 `void _LCDInit(void);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Pripravi prikazovalnik LCD za uporabo.

A.4.4 `void _LEDInit(void);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Pripravi diode LED za uporabo.

A.4.5 `void _setpindir(unsigned int pinid, int dir);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Nastavi smer vrat. Parameter *pinid* ima lahko vrednost med 0 in 15, in z njim izberemo ena od vrat P0.0 do P0.15. S parametrom *dir* določimo smer. 0 pomeni vhod in 1 izhod.

A.4.6 `void _TimerInit(void);`

Knjižnice: `"io.h" (Š-ARM)`

Opis: Pripravi časovnik (timer) za uporabo.

B Električni načrt stikala za zatemnitev