Programmi kompileerimine ja C eripärad

Erkki Moorits
Cybernetica AS, Navigatsioonisüsteemide osakond

Teemad

- Programmi kompileerimine, paigutus mälus ja kuidas jõuab programm mällu
- C keele eripärad
  - Ohtlikud kohad
  - Koodi suurus

NB! Järgnev loeng on C keele spetsiifiline, täpsemalt GNU C kompilaatori ja linkuri

Programmi kompileerimine ja linkimine

- Kompileerimise etapid
  - Töödeltakse makrod (eelprotsesor)
  - Kontrollitakse koodi süntaksit
  - Teisendamine abstraktseteks operatsioonide jadaks
  - Optimeerimine
    - LTO
  - Objekt-failide genereerimine
- Linkimise etapid
  - Lahendatakse kõik märgendid (muutujad, funktsioonid ...)
  - Genereeritakse lõplik täidetav fail (.elf, .exe)

C eelprotsessor

- Peamiselt tegeleb:
  - Koodi lisamisega (#include)
  - Makrodega (#define)
  - Tingimusliku koodi lisamisega (#if)
- Üldiselt on ettenähtud C ja C++ jaoks, kuid on võimalik ka muude keeltega kasutada
- Eelprotsessor ei avasta koodist endast vigu
  - Eelprotsessoriga on võimalik väga halvasti leitavaid vigu tekitada

Mälujaotus kompileerimise ajal

- Kompileerimise ajal määratakse esialgne mälujaotus
  - Määratakse registrid, `.stack`, `.data` ja `.heap`, aga ei määrata nende aadresse
  - Lõpliku mälujaotuse paneb paika linukur
    - NB! objekt failidest ei saa kunagi korrektseid mälu aadresse – kõik nad näitavad konstanti 0

```
> objdump -S test.o
test.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:  55                          push   %rbp
   1:  48 89 e5                    mov    %rsp,%rbp
   4:  bf 00 00 00 00              mov    $0x0,%edi
   9:  e8 00 00 00 00              callq  e <main+0xe>
   e:  b8 00 00 00 00              mov    $0x0,%eax
  13:  c9                          leaveq
  14:  c3                          retq
```

Kompileerimine – näide

- C programm

```
#include <stdio.h>

#define output(string) puts(string)

int main (void)
{
    output ("hello world");
    return 0;
}
```

- Kompileerimine koos vahe sammude väljastusega

```
gcc -save-temps -Wall hello.c -o hello
```

Näiteprogramm – väljund peale eelprotsesorit

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 356 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 353 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 354 "/usr/include/sys/cdefs.h" 2 3 4

------------------------------------------------------------------
extern char *ctermid (char *__s) __attribute__ ((__nothrow__));
# 906 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__));
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__)) ;
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__));
# 936 "/usr/include/stdio.h" 3 4
# 2 "hello.c" 2

int main (int argc, char *argv[])
{
    puts("hello world");
    return 0;
}
```

Näiteprogramm – kompilaatori assembleri väljund

```
.LC0:
    .string "hello world"
    .text
.globl main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    movq    %rsp, %rbp
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Gentoo 4.5.3-r1 p1.0, pie-0.4.5) 4.5.3"
    .section    .note.GNU-stack,"",@progbits
```

Näiteprogramm – mälujaotus

```
[erx@erkki tmp]$ nm hello

0000000000600e40 d _DYNAMIC
0000000000600fe8 d _GLOBAL_OFFSET_TABLE_
0000000000400648 R _IO_stdin_used
                 w _Jv_RegisterClasses
0000000000600e30 D __DTOR_END__
0000000000601020 A __bss_start
0000000000601010 D __data_start
0000000000601018 D __dso_handle
                 w __gmon_start__
0000000000600e14 d __init_array_end
0000000000600e14 d __init_array_start
0000000000400560 T __libc_csu_fini
0000000000400570 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000601020 A _edata
0000000000601030 A _end
0000000000400638 T _fini
0000000000400408 T _init
0000000000400450 T _start
0000000000601010 W data_start
0000000000400534 T main
                 U puts@@GLIBC_2.2.5
```

Assembleris koodi vaatamise teine variant – objdump

- Objdump objektfailist (*.o) annab ilma aadressideta assembrleris koodi

```
> objdump -S test.o
test.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:  55                          push   %rbp
   1:  48 89 e5                    mov    %rsp,%rbp
   4:  bf 00 00 00 00              mov    $0x0,%edi
   9:  e8 00 00 00 00              callq  e <main+0xe>
   e:  b8 00 00 00 00              mov    $0x0,%eax
  13:  c9                          leaveq
  14:  c3                          retq
```

Assembleris koodi vaatamise teine variant – objdump jätk

- Objdump lõplikust programmist annb kõik täidetavad instruktsioonid
  - Ei ole mõtekas kasutada PC programmide puhul

```
led_demo:     file format elf32-avr
Disassembly of section .text:
00000000 <__vectors>:
   0:   12 c0           rjmp    .+36            ; 0x26 <__ctors_end>
   2:   2a c0           rjmp    .+84            ; 0x58 <__bad_interrupt>
   4:   29 c0           rjmp    .+82            ; 0x58 <__bad_interrupt>
   6:   28 c0           rjmp    .+80            ; 0x58 <__bad_interrupt>
--------------------------------------------------------------------------

00000026 <__ctors_end>:
  26:   11 24           eor     r1, r1
  28:   1f be           out     0x3f, r1        ; 63
  2a:   cf ed           ldi     r28, 0xDF       ; 223
  2c:   cd bf           out     0x3d, r28       ; 61

0000002e <__do_copy_data>:
  2e:   10 e0           ldi     r17, 0x00       ; 0
```

Linkimine

- Linkur seob erinevad objekt-failid, lahendab märgendite asukohad mälus ja loob programmi täidetava faili (.exe, .elf …)
  - Sardsüsteemides kasutavavaid .ihex, .srec mälutõmmise formaati teisendab objcopy
  - Täidetavaid faile loevad ka mõned bootloaderid ja programmatorid
- Staatiline linkimine
  - Kõik vajalik on täidetavas failis
- Dünaamiline linkimine
  - Vajalikud teegid loetakse töö ajal
  - Ei ole kastutav mikrokontrolleritel

Linkimine jätk

- Objekt-failid, failid millest koostatakse lõplik täidetav programm või teek
  - Objekt-fail sisaldab nii programmi koodi kui ka märgendite tabelit
    - Märgendite tabelis on märgendi nimi kui ka asukoht koodis
  - Objekt-failis on märgendid lahendamata
    - Aadressid on väärtusega 0, lahendatakse linkuri poolt
- Märgendite lahenamine linkuri poolt
  - Linkur kogub kokku kõik globaalsed märgendid
  - Otsib üles objekt-failid, kus need märgendid on
  - Staatilisel linkimisel tühi märgendi koht asendatakse konkreetse aadressiga

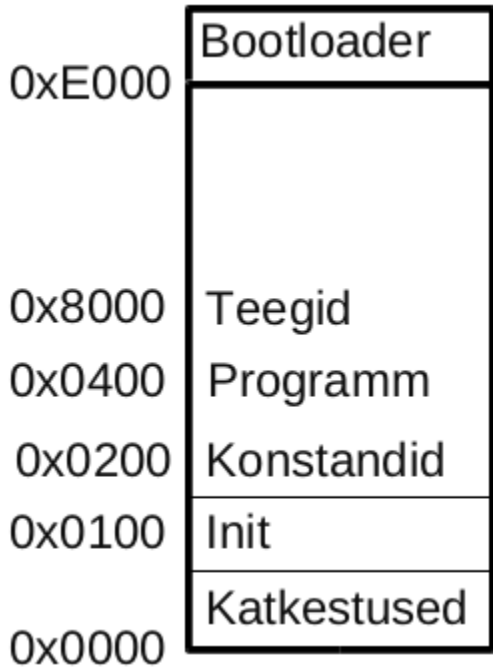
Linkimise näide

```
#include <stdio.h>

static int j = 2;

int fn_1 (int i)
{
   return (i + 1);
}

int fn_2 (void)
{
    return (j + 1);
}

int main (void)
{
    printf ("%d\n%d\n",
              fn_1 (1), fn_2 ());
    return 0;
}
```

```
0000000000600e40 d _DYNAMIC
0000000000600fe8 d _GLOBAL_OFFSET_TABLE_
0000000000400678 R _IO_stdin_used
                 w _Jv_RegisterClasses
0000000000600e30 D __DTOR_END__
0000000000601024 A __bss_start
0000000000601010 D __data_start
0000000000601018 D __dso_handle
                 w __gmon_start__
0000000000600e14 d __init_array_end
0000000000600e14 d __init_array_start
0000000000400590 T __libc_csu_fini
00000000004005a0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000601024 A _edata
0000000000601038 A _end
0000000000400668 T _fini
0000000000400408 T _init
0000000000400450 T _start
0000000000601010 W data_start
0000000000400534 T fn_1
0000000000400543 T fn_2
0000000000601020 d j
0000000000400552 T main
                 U printf@@GLIBC_2.2.5
```

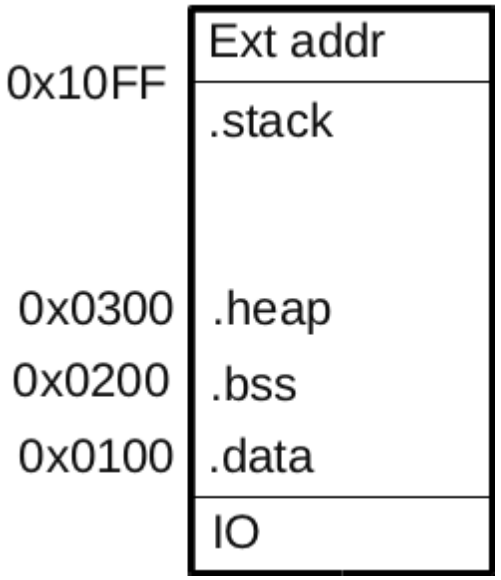
Programm mälu jaotus – AVR'i näide

- `.text` piirkond
  - `.vectors` – katkestusvektorid
  - `.init` – Intisialiseerimise kood
    - Muutujate algväärtustamine
  - Konstandid
  - Programm ise
  - Teegid
    - Programmi uuendamine?
- `.bootloader` piirkond
  - Piirkonna nimi ei ole väga rangelt määratud, oluline on see, et see piirkond oleks õigetel aadressidel



RAM'i jaotus – AVR'i näide

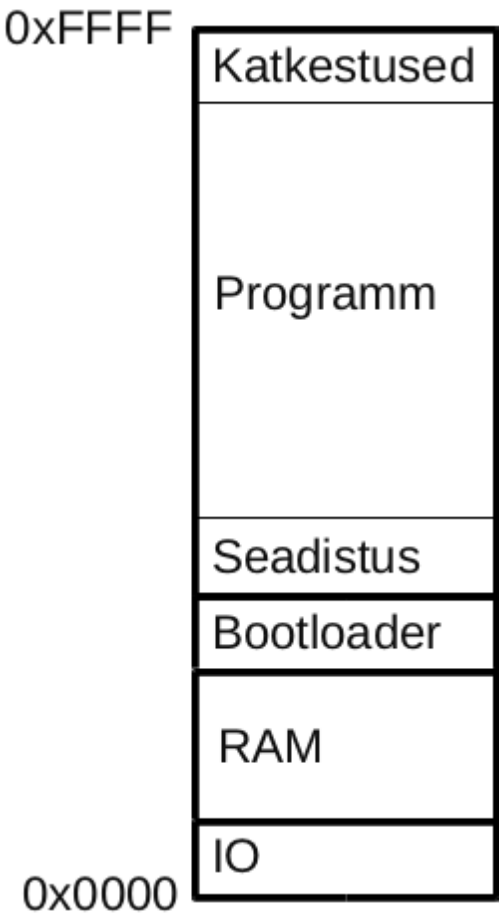
- RAM'i aadressidel olev IO
- RAM
  - `.data` osa
    - Konstandid
    - Globaalsed või staatilised muutujad
  - `.bss` osa
    - Algväärtuseta globaalsed või staatilised muutujad (null väärtusega)
    - `.noinit` osa
  - `.heap` osa, malloc'ile lubatud ala
  - `.stack` osa
- Väline aadressi väli
  - Välised seadmed, RAM



Võrdluseks MSP430 mälujaotus

- Von Neumani arhitektuuril on kõik mälud ühes aadressi väljas

- Katkestusvektorid
- Programm
- Info segmendid
- Bootloader
- RAM
- IO



Mälukujutus (memory image)

- a.out, .elf, .exe ...
  - Tavaliselt kõik sektsioonid (ROM, RAM'i konstandid, EEPROM)
    - Laadimine keerukas
  - Kui programm on kompileeritud mõnele OS'ile siis:
    - Siis programmi failiga kaasas see osa mida pole jagatud teekides
    - OS peab tagama korrektse mälujaotuse
- .hex, .srec ...
  - Tavaliselt programm mälu sisu, vaikimisi ainult `.text` piirkond
  - Ülejäänud mäluosade (sektsioonide) lisamiseks tuleb kas:
    - Määrata objcopy parameetritega
    - Teha vastav linkuri skript

Viimane samm – programmi mällu “jõudmine”

- Peale linkimist teisendatakse objcopy'ga programm mõneks laetavaks mälutõmmiseks (ihex, srec)

- Laetakse kontrolleri mällu
  - SPI programmator
  - JTAG
  - Bootloader
  - SD kaart, ...

C keele eripäerad - ohtlikud funktsioonid ja tegevused

- Tüübi muutmine (*casting*)
- Tsüklid (do, while, for)
- Viitade aritmeetika
- Kaudsed funktsioonid (*indirect functions* või *indirect calls*)
- *Local* ja *nonlocal* goto
- Tühjad tsüklid
- Väikse *stacki* puhul suured massiivid
- Sisendandmete kontroll

Olulised erand(id)

- `malloc'i` ja `free'd` ei ole vaja igal pool kasutada
  - Paljudel juhtudel reserveeritakse draiverites puhvriteks (vms.) mälu mis on kogu süsteemi töö jooksul kasutuses
  - Pidevalt töös olva draiveri puhul pole võimalik või siis mõtekas kasutada `free`'d.
    - Kui proovida sellisel draiveril teha staatilist koodi kontrolli siis leitakse peaaegu alati mälulekke.

Tüübi muutmine (typecasting)

- Muutujate tüübi muutmine väiksema maksimumväärtusega tüübilt (ühtlasi ka väiksemalt tüübilt) suuremale ei kujuta reeglina ohtu
  - Näiteks `uint16_t` tüüpi muutuja (16 bitine) muutmisel `uint32_t` tüüpi muutujaks (32 bitine) ei kujuta piisava mälu juures ohtu.
  - **NB! IO registrid!** Tavaliselt on erinevate liideste/seadmete vms registrid kõrvuti

Tüübi muutmine (typecasting) jätk

- Muutuja tüübi muutmisel suurema maksimumväärtusega tüübilt väiksema maksimum väärtusega tüübile peab **alati** tegema eelneva kontrolli
  - Erandiks on ainult juhus kus ei saa suurema maksimum väärtusega muutuja mitte mingil juhul olla väiksemast muutujast suurem, näiteks kui 32 bitine muutuja on jagatud 2^17'ga ja tulemus muudetud 16 bitiseks

Viidatüübi muutmine

- Viida tüüpide (pointerite) suuremaks muutmisel tuleb olla eriti tähelepanelik, näiteks kui muudetakse 16 bitistele andmetele viitav viit 32 bitistele andmetele viitavaks viidaks
  - C kompilaator ei anna mitte mingit vea indikatsiooni kui tüübi muutmise tagajärjel rikutakse teise funktsiooni mälu ära
  - PC'le (Linuxi) kirjutatud programm mis kasutab vigast viidateisendust pannakse suure tõenäosusega kinni, kuid mikrokontrolleritel põhjustab selline viga terve süsteemi kokkujooksmise
  - Tüübi muutmist annab teha suhteliselt ohutult massiividega, näide järgmisel slaidil

Viidatüübi muutmine massiivis

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

int main (void)
{
    uint8_t arr[] = {0x11, 0x22};
    uint16_t *ptr;

    ptr = (uint16_t*)&arr[0];
    printf ("arr: 0x%02x, 0x%02x\n", arr[0],
                                     arr[1]);
    printf ("ptr: 0x%04x\n", *ptr);

    /* vigane pointeri tüübi muutmine */
    ptr = (uint16_t*)&arr[1];
    printf ("ptr: 0x%04x\n", *ptr);

    return 0;
}

/* väljund */
arr: 0x11, 0x22
ptr: 0x1122
ptr: 0x2200
```

NB! kolmandal väljundreal on kaks viimast numbrit nullid, see tuleneb sellest, et operatsioonisüsteem on kas mälu eest ära nullinud (turvalisuse kaalutlustel) või on juhtunud lihtsalt tühjale alale. Sellist käitumist ei tasu eeldada teistel süsteemidel.

Viidatüübi muutmine

- Suvalise viidatüübi (näiteks `uint8_t *ptr1`) muutmine tühjaks viidatüübiks (`void *ptr2`) kaotab ära igasuguse tüübi informatsiooni
  - Sellist muutmist kasutatakse peamiselt mõningates draiverites või *CallBack* funktsioonides tundmatute parameetriterite edasiandmiseks
  - C puhul on tüübi informatsioon oluline ainult kompilaatorile ja programmeerijale endale, lõplik masinkoodis programm ei erine mitte mingil määral korrektsete tüüpidega programmist

Tüübi modifikaatorite muutmine

- Tüübi modifikaatori muutimisel tugevamalt piiratud tüübist vähem piiratud tüüpi on võimalik tekitada väga raskesti leitavaid vigu
  - Näites muudetakse funktsioonis fun parameeter väljaspool kriitilist sektsiooni
    - GCC puhul näitab selliseid teisendusi `-Wcast-qual` käsurea parameeter

```
volatile uint8 *data;
uint8_t fun (uint8_t *ptr)
{
   EnterCritical ();
   *ptr = 15;
   ExitCritical ();

   return *ptr;
}
```

```
int main (void)
{
   EnterCritical ();
   *data = 10;
   ExitCritical ();

   fun (data);
}
```

Tsüklid (do, while, for)

- Do, while ja for tsüklid on ühed enimlevinud potentsiaalselt ohtlikud koodilõigud
- Tüüpilised ohtlikud kohad
  - Threadide töötsüklid
  - Pollimine

Threadide töötsüklid ja pollimine

```
for (; 1;)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
};
/*******************************/
while (1)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
};
/*******************************/
do
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;
    /* ... veel mingi kood ... */
}
while (1);
```

- Threadide töötsükli juhul on võimalik jooksutada cooperative schedureriga kernel sellise koodiga kokku, seda juhul kui ei lisata mingit ajalise viitega koodi
- Pollimise juhul on võimalik, et programm ei jõua kunagi siit kaugemale

Pollimine (korrektne näide)

```
/* maksimaalne i väärtus on 2^16 */
uint16_t i;


for (i = 0; i < 0xffff; i++)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* ... veel mingi kood ... */
};

if (i == 0xffff)
    puts ("ei olnud aktiivsust");
```

Lisatud on üks 16-bitine muutuja, mille täitudes tsükkel lõppeb ja väljastatakse teade.

Pollimine (vigane näide)

```
/* maksimaalne i väärtus on 2^16 */
uint16_t i;


for (i = 0; i <= 0xffff; i++)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* ... veel mingi kood ... */
};

if (i == 0xffff)
    puts ("ei olnud aktiivsust");
```

Kuna muutuja i on 16 bitine (maksimaalväärtus on 0xFFFF), siis võrdlemine 0xFFFF 'ga annab **alati** tõese väärtuse ja tsüklist ei pruugita mitte kunagi väljuda. Kompilaator ei tarvitse seda viga tavalise hoiatuse taseme juures näidata, et näitaks tuleb gcc'l lülitada sisse **-Wall** ja **-Wextra** vigade indikatsioon.

Tsükli kasutamine lõplikus threadis (korrektne näide)

```
/* lõplik thread */

while (1)
{
    if (PORTA & (1 << PA3))
        break;
    else if (PORTB & (1 << PB4))
        break;

    /* mitteaktiivne 10 ajaühikut */
    SLEEP (10);

    /* ... veel mingi kood ... */
};

puts ("PA3 või PB4 oli aktiivne");
```

Igas tsüklis on üks 10 ajaühiku pikkune viide, selline viide annab võimaluse teistel threadidel samaaegselt oma ülesandeid täita, sh võimaluse watchdog'i taimerit nullida.

Tsükli kasutamine lõputus threadis (korrektne näide)

```
/* lõputu thread */

while (1)
{
    /* mitteaktiivne 10 ajaühikut */
    SLEEP (10);

    if (!(PORTA & (1 << PA3)))
        continue;
    else if (!(PORTB & (1 << PB4)))
        continue;

    puts ("PA3 ja PB4 olid aktiivsed");
    /* ... veel mingi kood ... */
};
```

Ajaline viide peab olema alati sellises kohas kus tsükkel käib alati läbi.

Kaudsed funktsioonid

```
#include <stdio.h>

static void fn_1(void)
{
    puts ("Hello world!");
}

void (*indirect_fn)(void) = fn_1;

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

- Kaudset funktsiooni kutsutakse välja kasutades selleks vastavat viita (pointerit), seejuures see viit on ise sisuliselt muutuja mille tüüp on funktsioon
- Peamiseks kasutuskohaks on kernelis scheduleriga seotud funktsioonid ja draiverid

Kaudsed funktsioonid II

```
#include <stdio.h>

/* main funktsioon asub aadressil
 * 0x0000 */
int main (void)
{
    puts ("Hello world!");
}

/* funktsioon bootloader
 * täidetakse alati peale
 * resetit ning ta asub aadressil
 * 0xff00 */
void bootloader (void)
{
   /* programmi mällu laadimine */
   ((void (*)())0x0000)();
}
```

- Kaudset funktsiooni saab välja kutsuda kui muuta eelnevalt antud aadress funktsiooni viidaks ning siis kutsuda välja see viit
- Peamiseks kasutuskohaks on bootloaderisse sisenemise ja lahkumise kohad

Kaudsed funktsioonid III

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

typedef struct fp_struct
{
    uint8_t num;
    uint8_t (*func) (uint8_t *p1);
    struct fp_struct *next;
} FP_STRUCT;

uint8_t fn_1 (uint8_t *p1)
{
    printf ("fn_1\t[p1 == %u]\t", *p1);
    return (*p1) + 1;
}

uint8_t fn_2 (uint8_t *p1)
{
    printf ("fn_2\t[p1 == %u]\t", *p1);
    return (*p1) + 10;
}
```

```
int main (void)
{
    uint8_t i, rc;
    FP_STRUCT *fn, *fn_last, *fn_first;

    fn = malloc (sizeof (FP_STRUCT)*2);
    fn_first = fn;
    fn->func = fn_1;
    fn->num = 0;
    fn_last = fn;
    fn++;
    fn_last->next = fn;
    fn->func = fn_2;
    fn->num = 1;
    fn->next = fn_first;

    for (i = 0; i < 5; i++)
    {
        rc = fn->func (&i);
        printf ("fn: %u; rc: %u\n",
                fn->num, rc);
        fn = fn->next;
    };
    free (fn_first);
    return 0;
}
/* väljund */
fn_2    [p1 == 0]       fn = 1; rc = 10
fn_1    [p1 == 1]       fn = 0; rc = 2
fn_2    [p1 == 2]       fn = 1; rc = 12
fn_1    [p1 == 3]       fn = 0; rc = 4
fn_2    [p1 == 4]       fn = 1; rc = 14
```

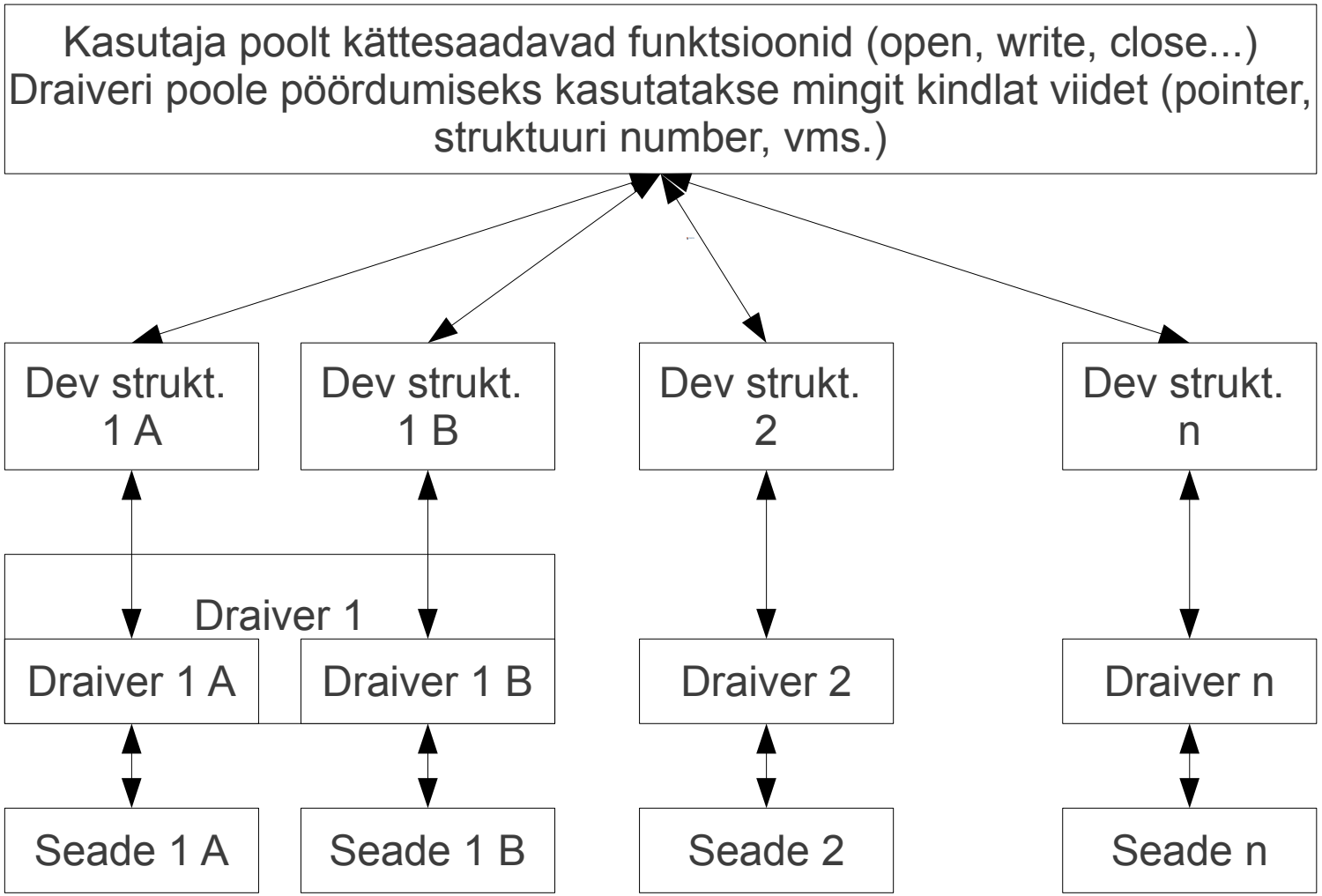
Probleemid kaudsete funktsioonidega

- Võimalik tööajal funktsiooni aadressi muuta
  - Tüüpiline olukord – üks funktsioon muudab pointeriga valet aadressi (pole vahet kas pointer on suunatud valele aadressile või on tegemist vale castinguga)
- Väga lihtsalt võimalik valesid parameetreid ette anda ja vale tagastusväärtuse saada

Kaudsete funktsioonide peamine kasutuskoht – kernel ja draiverid

- Draiverid on need kerneli osad mille ülesanneteks on ainult riistvaraga suhtlemine
  - Kuna draiverite funktsioone kutsutakse tihtipeale väga palju välja siis peaksid draiverid olema kirjutatud nii, et nad võtaksid võimalikult vähe ressurssi
- Järgnevatel slaidide on toodud NutOS'i baasil usart draiveri näide. Näites ei ole täielikku koodi välja toodud vaid on toodud olulised koodilõigud

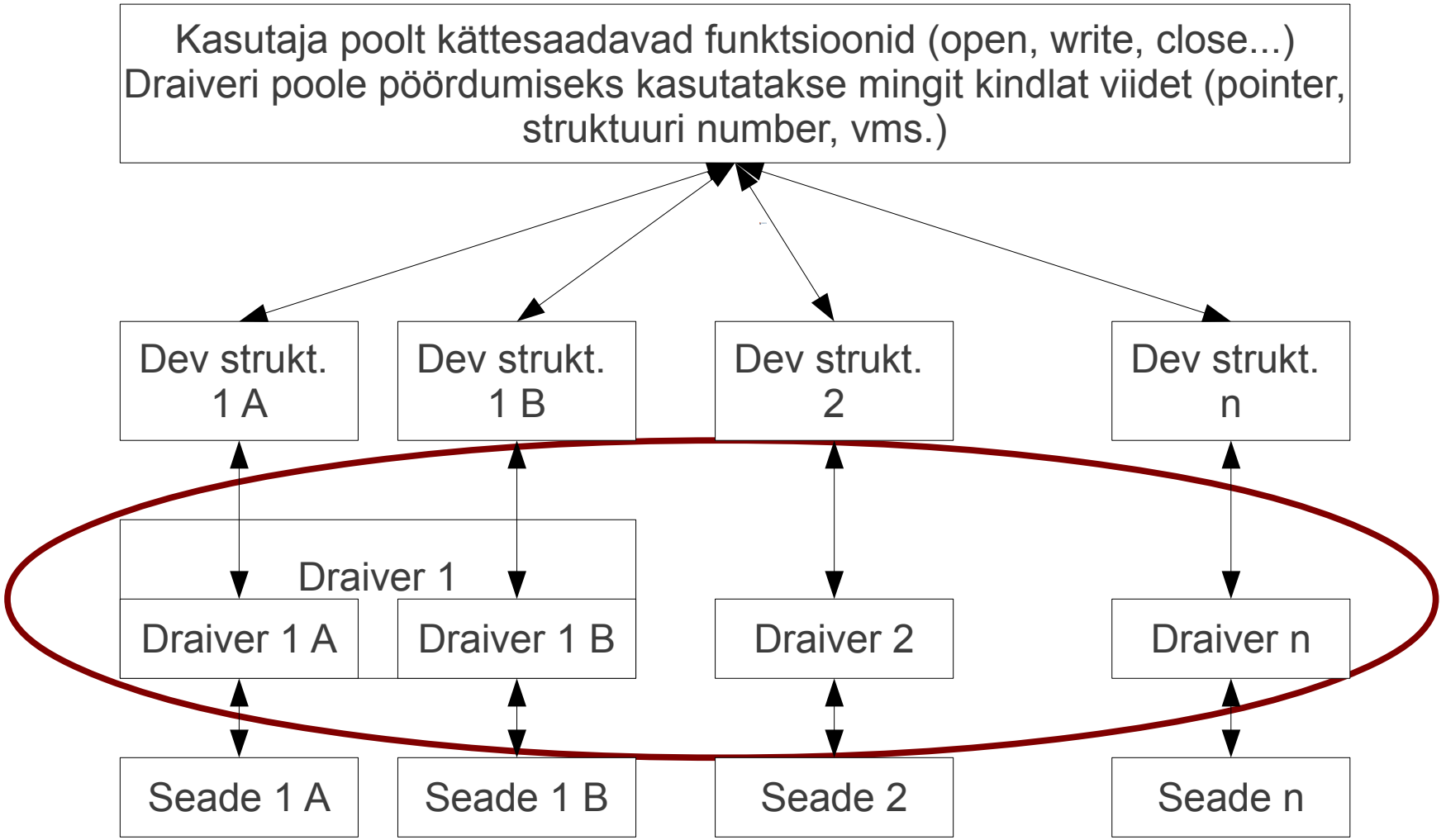
Tüüpiline suhtlus seadmega läbi draiveri



Riistvara lähedased draiveri funktsioonid – DCB struktuur

- Riistvara lähedased funktsioonid paigutatakse kõik DCB struktuuri (*device control block structure*)
  - Siia alla kuuluvad kõiksugused riistvara seadistamised ja portidest lugemised/kirjutamised.
  - DCB struktuuri puhul on tegemist kaudsete funktsioonide struktuuridega
- Sarnaselt DCB struktuurile on olemas ka ICB struktuur (*interface control block structure*)
  - ICB puhul on üldiselt programmerijal rohkem vabadust luua oma lisasid
  - ICB on mõtekas kasutada pigem lihtsate liideste puhul (CAN liides)

Riistvara lähedased funktsioonid



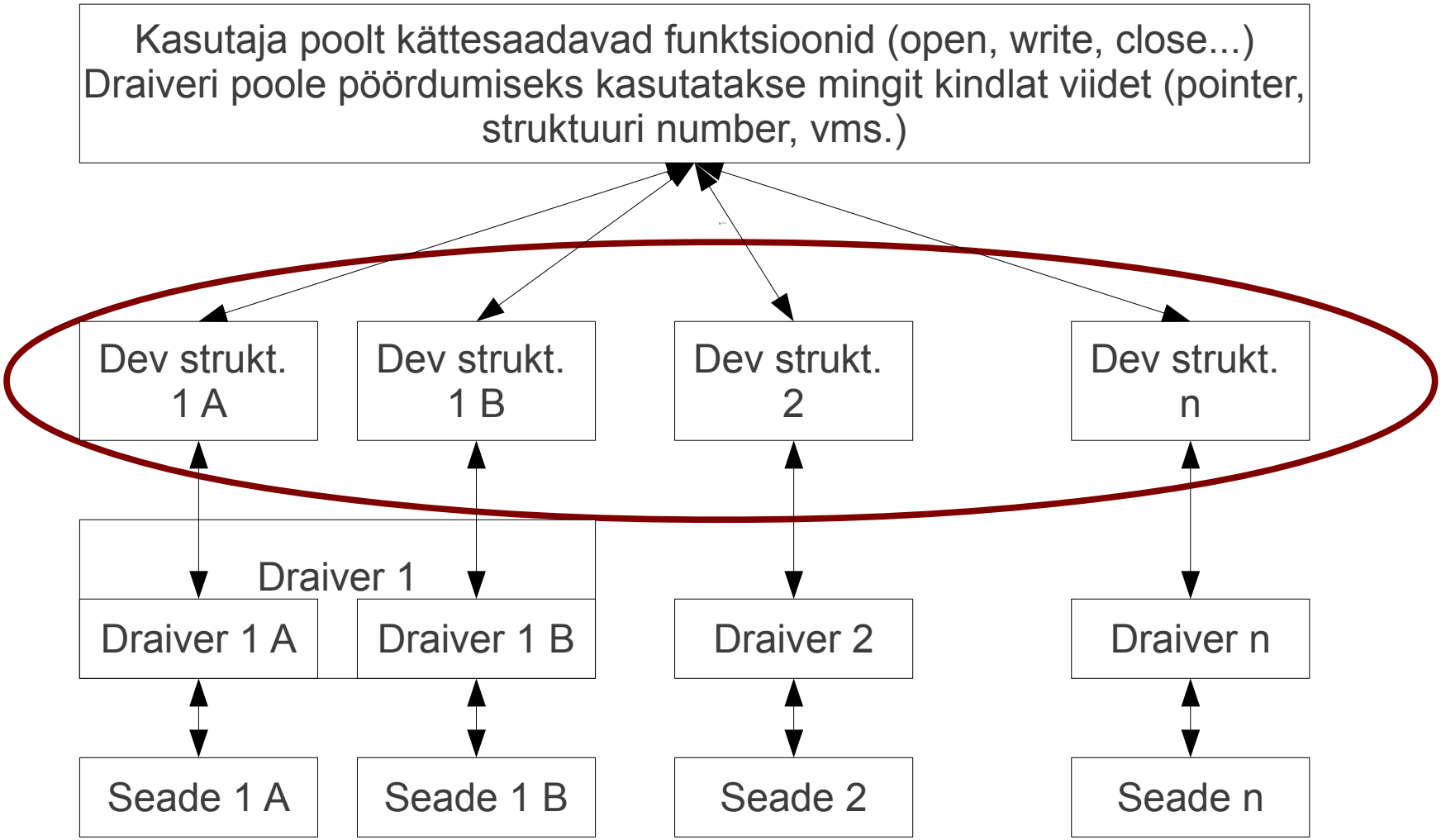
DCB struktuuri initsialiseerimine

```
static USARTDCB dcb_usart0 = {
    0,                          /* dcb_modeflags */
    0,                          /* dcb_statusflags */
    0,                          /* dcb_rtimeout */
    0,                          /* dcb_wtimeout */
    {0, 0, 0, 0, 0, 0, 0, 0},   /* dcb_tx_rbf */
    {0, 0, 0, 0, 0, 0, 0, 0},   /* dcb_rx_rbf */
    0,                          /* dbc_last_eol */
    AvrUsartInit,               /* dcb_init */
    AvrUsartDeinit,             /* dcb_deinit */
    AvrUsartTxStart,            /* dcb_tx_start */
    AvrUsartRxStart,            /* dcb_rx_start */
    AvrUsartSetFlowControl,     /* dcb_set_flow_control */
    AvrUsartGetFlowControl,     /* dcb_get_flow_control */
    AvrUsartSetSpeed,           /* dcb_set_speed */
    AvrUsartGetSpeed,           /* dcb_get_speed */
    AvrUsartSetDataBits,        /* dcb_set_data_bits */
    AvrUsartGetDataBits,        /* dcb_get_data_bits */
    AvrUsartSetParity,          /* dcb_set_parity */
    AvrUsartGetParity,          /* dcb_get_parity */
    AvrUsartSetStopBits,        /* dcb_set_stop_bits */
    AvrUsartGetStopBits,        /* dcb_get_stop_bits */
    AvrUsartSetStatus,          /* dcb_set_status */
    AvrUsartGetStatus,          /* dcb_get_status */
    AvrUsartSetClockMode,       /* dcb_set_clock_mode */
    AvrUsartGetClockMode,       /* dcb_get_clock_mode */
};
```

Device struktuur

- Device struktuur on vahelüli kasutaja funktsioonide ja riistvara lähedaste funktsioonide vahel
- Iga draiver hoiab oma kirjutamise, lugemise ja IO juhtimise funktsioone globaalses Device struktuuris. Linuxi all hoitakse sarnaselt *char device*'i funktsioone analoogses struktuuris `file_operations`

Riistvaralähedaste fun. ja kasutaja vahelised funktsioonid



Device struktuur

```
struct _NUTDEVICE {
    NUTDEVICE *dev_next;
    char dev_name[9];
    uint8_t dev_type;
    uintptr_t dev_base;
    uint8_t dev_irq;
    void *dev_icb;
    void *dev_dcb;
    int (*dev_init) (NUTDEVICE *);
    int (*dev_ioctl) (NUTDEVICE *, int, void *);
    int (*dev_read) (NUTFILE *, void *, int);
    int (*dev_write) (NUTFILE *, CONST void *, int);
#ifdef __HARVARD_ARCH__
    int (*dev_write_P) (NUTFILE *, PGM_P, int);
#endif
    NUTFILE * (*dev_open) (NUTDEVICE *, CONST char *, int, int);
    int (*dev_close) (NUTFILE *);
    long (*dev_size) (NUTFILE *);
};
```

- NB! Harvardi arhitektuuril on üks funktsioon rohkem

Device struktuuri initsialiseerimine

```
NUTDEVICE devUsartAvr0 = {
    0,                          /* Pointer to next device, dev_next. */
    {'u', 'a', 'r', 't', '0', 0, 0, 0, 0},    /* Unique device name, dev_name. */
    IFTYP_CHAR,                 /* Type of device, dev_type. */
    0,                          /* Base address, dev_base (not used). */
    0,                          /* First interrupt number, dev_irq (not used). */
    0,                          /* Interface control block, dev_icb (not used). */
    &dcb_usart0,                /* Driver control block, dev_dcb. */
    UsartInit,                  /* Driver initialization routine, dev_init. */
    UsartIOCtl,                 /* Driver specific control function, dev_ioctl. */
    UsartRead,                  /* Read from device, dev_read. */
    UsartWrite,                 /* Write to device, dev_write. */
    UsartWrite_P,               /* Write data from program space to device, dev_write_P. */
    UsartOpen,                  /* Open a device or file, dev_open. */
    UsartClose,                 /* Close a device or file, dev_close. */
    UsartSize                   /* Request file size, dev_size. */
};
```

Draiveri kõrgemataasemelised funktsioonid

```
int UsartInit(NUTDEVICE * dev)
{
    int rc;
    USARTDCB *dcb = dev->dev_dcb;

    /* Initialize the low level hardware
     * driver. */
    if ((rc = (*dcb->dcb_init) ()) == 0) {
        /* Ignore errors on initial
         * configuration. */
        (*dcb->dcb_set_speed)(USART_INITSPEED);
    }
    return rc;
}

/*************************************/

int UsartRead(NUTFILE * fp, void *buffer,
              int size)
{
    NUTDEVICE *dev = fp->nf_dev;
    USARTDCB *dcb = dev->dev_dcb;
    RINGBUF *rbf = &dcb->dcb_rx_rbf;

    /*  seadmest lugemise funktsioonid */
};
```

- Device struktuuri liikmed viitavad vastavate draiveri kõrgemataasemeliste funktsioonidele
- Draiveri kõrgemataasemelised funktsioonid võivad olla ühised mitmele madalmataasemelisele draiveri funktsioonile

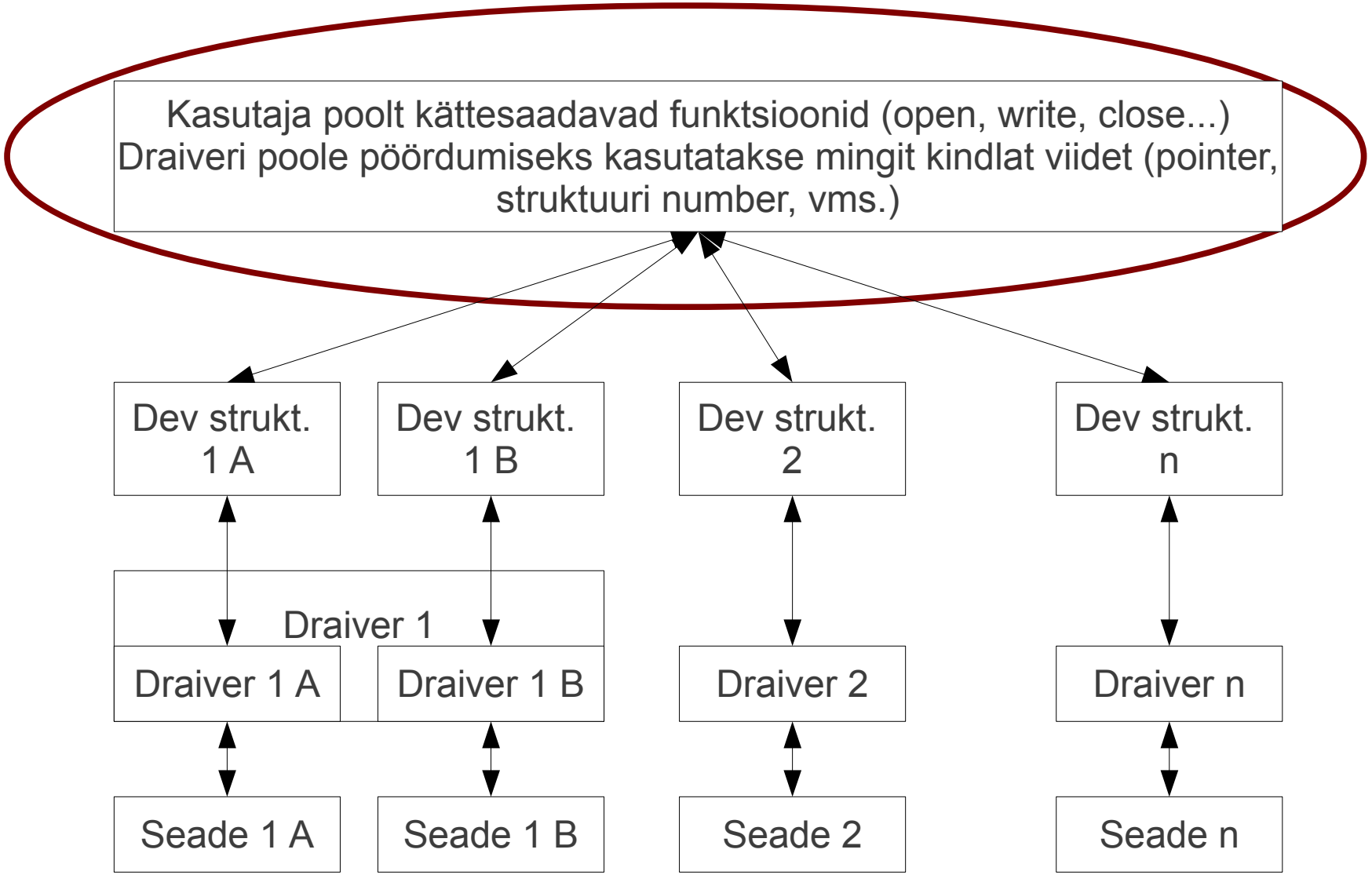
Draiveri kõrgemataasemelised funktsioonid

- Draiveri kõrgemataasemeliste funktsioonide kasutamine annab järgmisi eeliseid:
  - Nendega hoiab mälu kokku
  - Kuna draiveri kõrgemataasemelised funktsioonid peavad olema porditavad, siis peab vähem riistvaraliste funktsioonide kirjutamistel programmi kirjutama

Võrdlus Linuxiga - file_operations struktuur

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);

------- Järgnev osa ära lõigatud ---------------------------------------------
};
```

Kasutajale kättesaadavad funktsioonid



Rakendusele kättesaadavad funktsioonid

```
int _read(int fd, void *buffer,
          unsigned int count)
{
    NUTFILE *fp = (NUTFILE *) ((uintptr_t) fd);
    NUTDEVICE *dev;

    NUTASSERT(fp != NULL);
    dev = fp->nf_dev;
    if (dev == 0) {
        NUTVIRTUALDEVICE *vdv =
                        (NUTVIRTUALDEVICE *) fp;
        return (*vdv->vdv_read)
                            (fp, buffer, count);
    }
    return (*dev->dev_read) (fp, buffer, count);
}
```

- Rakendus saab kasutada device struktuuri kaudu vastavaid draiverite funktsioone
- Rakendusele kättesaadavad funktsioonid võivad olla mitmetele draiveritele ühised

Goto, adresseeritud goto ja setjmp/longjmp (non-local goto)

- Kõik goto käsud võimaldavad teha programmis suhteliselt suvalisi hüppeid kuid:
  - Local goto on piiratud ainult ühe funktsiooniga
  - Adresseeritud goto võimaldab tööd jätkata suvaliselt aadressilt
  - „Non-local goto“, ehk setjmp/longjmp võimaldavad hüpata ühest funktsioonist teisse funktsiooni
- Goto kasutamise juures tuleb silmas pidada järgnevaid kitsendusi:
  - Goto käsuga on võimalik üle hüpata muutujate algväärtustamistest, või siis stacki operatsioonidest
  - Goto käsk teeb reeglina koodi raskemini loetavaks

Goto ja adresseeritud goto

```
#include <stdlib.h>

static void f1 (void);

int main (void)
{
    uint8_t i;
    uint8_t j;
    void *label_2 = &f1;

    for (i = 0, j = 0; i < 7; i++)
    {
        printf ("i = %u, j = %u\n",i,j);

        if (i == 2)
            goto label_1;
        if (i == 5)
            goto *label_2;

        j++;
        continue;
label_1:
        puts ("label_1");
    };

    return 0;
}
```

```
static void f1 (void)
{
    puts ("function f1");
    exit (0);
}

/*************************************/
Programmi väljund

i = 0, j = 0
i = 1, j = 1
i = 2, j = 2
label_1
i = 3, j = 2
i = 4, j = 3
i = 5, j = 4
function f1
```

Adresseeritud goto bootloaderis

```
#include <stdio.h>

/* main funktsioon asub aadressil
 * 0x0000 */
int main (void)
{
    puts ("Hello world!");
}

/* funktsioon bootloader täidetakse
 * alati peale resetit ning asub
 * aadressil 0xff00 */
void bootloader (void)
{
   /* programmi mällu laadimine */
    goto *(void*)(0x0000);
}
```

- Adresseeritud goto kutsutakse välja koos vastavale aadressile suunatud funktsiooni viidaga, peale mida jätkub programmi töö viida aadressilt
- Peamiseks kasutuskohaks on bootloaderisse sisenemise ja lahkumise kohad

Goto käskude kasutuskohad

- Goto käskudest õigustab kõige paremini oma kasutamist adresseeritud goto
  - Kasutatakse ühe programmi lõppedes teise programmi käivitamiseks, näiteks bootloaderi lõppedes põhiprogrammi välja kutsumiseks või ohutuskriitilises rakenduses põhiprogrammi veaga lõppedes ohutust tagava funktsiooni käivitamiseks.
- Enamusjaolt kipuvad goto kasutamised näitama viletsat programmerimis praktikat

Tühjad tsüklid I

- Tsüklid mille eesmärk on ainult niisama teatud aja jooksul protsessori aega kulutada on nn. tühjad tsüklid
  - Kui võimalik, siis on parem neid mitte kasutada, vt. eelnevate tsüklite probleeme
  - Programmi täitmise kiirus tühjade tsüklite juures sõltub väga suurel määral kompilaatori optimeerimis tasemest
  - Üpriski tülikas on teha tühja tsüklit mis oleks alati sama kestvusega, porditav ja ei kaoks erinevate optimeerimistega ära

Tühjad tsüklid II

```
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>

#define LOOP_MAX_VAL 0xffffffff
static long int get_tdiff (struct timeval t1,
                           struct timeval t2);
int main (void)
{
    struct timeval start;
    struct timeval end;
    uint32_t i;

    gettimeofday (&start, NULL);
    /* intuitiivne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++);

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    gettimeofday (&start, NULL);

    /* korrektne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++)
        asm volatile ("nop");

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    return 0;
}
```

```
static long int get_tdiff (struct timeval t1,
                           struct timeval t2)
{
    long int v1 = (t1.tv_sec * 1000000 +
                   t1.tv_usec);
    long int v2 = (t2.tv_sec * 1000000 +
                   t2.tv_usec);

    return (v2 - v1);
}
```

Tühjad tsüklid III

```
/* Optimeerimistasemega O0 kompileeritud
 * programm */
705802 us
705802 us

/* Optimeerimistasemega O3 kompileeritud
 * programm */
1 us
138594 us
```

Optimeeritud programmi töö on ühe suurusjärgu võrra kiirem kui optimeerimata programmil, lisaks on optimeeritud programmist tühi tsükkel välja jäetud

Tühjad tsükli volatile variant

```
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>

#define LOOP_MAX_VAL 0xfffffff
static long int get_tdiff (struct timeval t1,
                           struct timeval t2);

int main (void)
{
    struct timeval start;
    struct timeval end;
    volatile uint32_t iv;
    uint32_t i;

    gettimeofday (&start, NULL);
    /* intuitiivne variant */
    for (iv = 0; iv < LOOP_MAX_VAL; iv++);

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    gettimeofday (&start, NULL);

    /* korrektne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++)
        asm volatile ("nop");

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    return 0;
}
```

```
/* Optimeerimistasemega O0 kompileeritud
 * programm */
705958 us
716922 us

/* Optimeerimistasemega O3 kompileeritud
 * programm */
716922 us
138586 us
```

Selles näites on tsükli muutuja *i* asendatud muutujaga *iv*, mis on *volatile*. Muutujat mis on märgitud kui *volatile* ei optimeerita, seega võtab sellise muutuja kasutamine rohkem programm mälu ja tõenäoliselt ka RAM'i, kuna muutujat ei hoita registrites.

Tühjad tsükli asm volatile variant

```
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>

#define LOOP_MAX_VAL 0xfffffff
static long int get_tdiff (struct timeval t1,
                           struct timeval t2);

int main (void)
{
    struct timeval start;
    struct timeval end;
    uint32_t i;

    gettimeofday (&start, NULL);
    /* intuitiivne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++);

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    gettimeofday (&start, NULL);

    /* korrektne variant */
    for (i = 0; i < LOOP_MAX_VAL; i++)
        asm volatile ("");

    gettimeofday (&end, NULL);
    printf ("%ld us\n", get_tdiff (start,end));
    return 0;
}
```

```
/* Optimeerimistasemega O0 kompileeritud
 * programm */
706277 us
705468 us

/* Optimeerimistasemega O3 kompileeritud
 * programm */
1 us
100913 us
```

Tsüklis mis peab ainult loendama, on lisatud `asm volatile` rida, mida kompilaator ei saa välja optimeerida.

LTO?

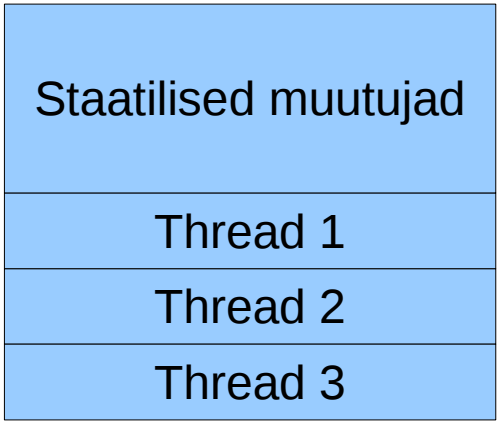
Alternatiivne tühja tsükli variant

```
/* alternatiiv variant */
uint8_t val = PORTB;

for (i = 0; i < LOOP_MAX_VAL; i++)
{
    if (val != PORTB)
        break;
};
```

Kui lasta programmil võrrelda mingit kindlat registrit mis mitte kunagi ei muutu, siis ei ole võimalik kompilaatoril seda koodilõiku välja optimeerida. Selline võiks olla ka assembleris käsu nop C's kirjutatud asendus.

NB! selline tühi tsükkel ei ole porditav

Väike stack ja suur massiiv




```
void fun (void)
{
   uint8_t array_1[1000]; /* ohtlik */
   static uint8_t array_2[1000];
   uint8_t *ptr;

   ptr = malloc (1000);
   /* ......... */
   /* kui stacki suurus on alla 1000 baidi,
    * siis nüüd rikume thread 2 stacki ära */
   array_1[999] = 0xAA;
}
```

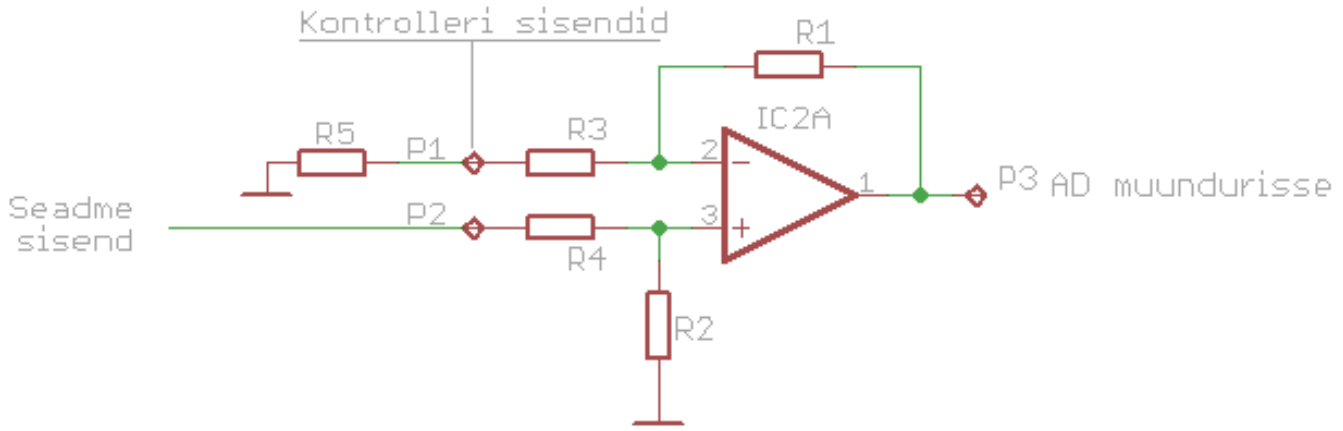
Suuri massiive ei tohi kunagi deklareerida funktsiooni sisena, nende jaoks tuleb alati võtta kas *malloc*'iga mälu või deklareerida nad staatilisena

Sisendandmete kontroll ja selle tegemata jätmine

- Kõiki sisendandmeid mille baasil võetakse vastu mingi otsus või on võimalus, et mõjutatakse süsteemi tööd tuleb alati kontrollida, näiteks:
  - Aku laadimisel aku pinge ja laadimisvoolu jälgimine
  - Süsteemidel millele on võimalik palju andmeid saata ja mis kõik pannakse enne töötlust sisendpuhvrisse, nendel tuleb kontrollida, et sisendpuhvrit üle ei täidetaks (serverite puhul üpriski tavaline kontroll)
- Alati ei ole vaja kontrollida sisendandmeid mis ei mõjuta mitte mingil moel süsteemi tööd, näiteks:
  - Kui ADC annab temperatuurianduri mõõtetulemuseks vale väärtuse ja see väärtus on ainult indikatsiooniks

Reaalne näide: digitaalne filter

AD muunduri väljund annab 10 bitise väärtuse, seda nii diferentsiaal sisendite puhul kui ka tavalise sisendi puhul (*single end*), kuid digitaalse filtri jaoks on vaja 9 bitist väärtust. Kui kasutada diferentsiaal sisendiga AD muundurit, ühendades ühe signaali maaga, siis on võimalik saada kohe otse AD muundurist 9 bitine väärtus



Eelmise näite käitumine reaalselt

- Kuna eelmsel skeemil pole näha mitte mingit sisendpingete piiramist, siis võib sisendisse sattuda ka negatiivne pinge, mis diferentsiaal sisendi puhul tekitab negatiivse tulemuse. Kui tegemist on märgita arvudega, siis teisendatakse negatiivne arv väga suureks positiivse arvuks

- Eelmise näite tulemused sõltuvad ka sellest kui suured on takistid R1, R3 ja R5 – piisavalt suurte väärtuste juures võib saada valesid tulemusi
  - Suurte väärtuste puhul on sisend müradele tundlik
  - Parasiitvoolud

Koodi optimeerimine

- Koodi optimeerimisega tegeleb peamiselt küll kompilaator kuid väga palju annab teha ära ka programmeerija poolt
- Mõned põhilisemad optimeerimise meetodid:
  - Kui võimalik, siis peaks võrdlustehete puhul kasutama nulli võrdlust
  - Korrutamised võiksid võimalusel olla kahe astmetena (2, 4, 8, jne)
  - Kui arvud on esitatud suurte muutujatega, kuid võrreldakse ainult madalamaid bitte, siis tuleks võrreldav muutuja teisendada süsteemi enda registri laiuseks muutujaks
  - Võimalikult vähe kasutada ujuvkoma arve

Koodi optimeerimine, jätk

- Kui 8 bitisel kontrolleril on üks instruktsioon ühele 8 bitisele tehtele, siis:
  - siis 16 bitine tehe võtab vähemalt kaks kuni kümme instruktsiooni
  - 32 bitine võtab vähemalt neli kuni 100 instruktsiooni

Mõned märkused 8 bitiste kontrollerite kohta

- Kasutada võimalikult väikeseid muutujatüüpe
  - Muutujad väärtusega kuni 255 kasutada uint8_t tüüpi, muutujatel väärtusega 256 kuni 65535 uint16_t, jne.
- Sõltuvalt kompilaatorist võib kompilaator kasutada vaikimisi 16 bitiseid andmetüüpe (GCC). Näide järgmisel kahel slaidil.
  - int on defineeritud 16 bitisena
  - Kõik tehted milles peab kompilaator ise muutuja tüübi määrama, määratakse alati muutuja tüübiks int

Kompilaatori poolt määratu väärtus

- Algne kood
  - ASSR on 8 bitine riistvara register
  - ASSR_BUSY on 8 bitine mask

```
while (ASSR & ASSR_BUSY);
```

- Assembleris väljund

```
a000:   80 91 b6 00     lds r24, 0x00B6
a004:   90 e0           ldi r25, 0x00   ; 0
a006:   8f 71           andi    r24, 0x1F   ; 31
a008:   90 70           andi    r25, 0x00   ; 0
a00a:   89 2b           or  r24, r25
a00c:   c9 f7           brne    .-14        ; 0xa000 <AvrCountTicks+0x14>
```

Kompilaatori poolt määratu väärtus, jätk

- Algne kood
  - Kuna int on 16 bitine aga meil on tegemist 8 bitiste väärtustega, siis salvestame maskiga väärtuse 8 bitisesse registrisse ja anname tulemuse while tsüklile kontrolliks. Vastasel juhul oleks meil tegemist 16 bitise and'iga

```
uint8_t tmp;
while ((tmp = (ASSR & ASSR_BUSY)) != 0);
```

- Assembleris väljund

```
9ffe:   80 91 b6 00     lds r24, 0x00B6
a002:   8f 71           andi    r24, 0x1F   ; 31
a004:   e1 f7           brne    .-8         ; 0x9ffe <AvrCountTicks+0x12>
```

Mõned märkused 8 bitiste kontrollerite kohta jätk

- Kui võimalik, siis kasutada võimalusel märgita tüüpe
- Kui võimalik, siis mitte kasutada *enum* tüüpe, kuna tihtipeale neid muudetakse 16 bitisteks muutujateks
- Kasutada väiksemate *switch* lausete asemel *if/else* lauseid (juhul kui see koodi loetamatuks ei tee)
  - Üldiselt on siis mõtekas kui on alla 5 case alamosa
- Pidevalt kasutuses olev mälu, näiteks puhvrid, tuleks deklareerida staatilisena

Mõned tähelepanekud optimeerimise kohta

- GCC optimeerib väikseid funktsioone paremini kui suuri
- Muutujatega tehakse tehteid ainult siis kui neid vaja läheb
- Kompilaatori poolsel optimeerimisel võib tekkida probleeme katkestustes ja muudes kriitilistes kohtades
  - Optimeerides võidakse tehteid muutujatega suhteliselt suvaliselt ümber tõsta
- Kompilaator paigutab ise enamuse muutujaid registritesse
  - Võtmesõna register on pigem programmerija jaoks

Optimeerimata funktsioon

```
/* mitteoptimaalne */
uint32_t crc32_update (uint32_t crc,
                       const uint8_t data)
{
    uint8_t i;

    crc = crc ^ data;

    for (i = 0; i < 8; i++)
    {
        if (crc & 0x01)
            crc = (crc >> 1) ^ 0xEDB88320;
        else
            crc = crc >> 1;
    };

    return crc;
}
```

Algne funktsioon, kus ei ole mingit optimeerimist tehtud

Optimeeritud funktsioon

```
/* optimaalne */
uint32_t crc32_update (uint32_t crc,
                       const uint8_t data)
{
    uint8_t i = 8;

    crc = crc ^ data;

    for (; i; i--)
    {
        if ((uint8_t)crc & 0x01)
            crc = (crc >> 1) ^ 0xEDB88320;
        else
            crc = crc >> 1;
    };

    return crc;
}
```

Eelnev funktsioon on kirjutatud optimaalsemalt ümber – tsüklis muutuja võrdlus käib nulli suhtes ja CRC võrdlusel kontrollitakse ainult nooremaid bitte

Mittetöötav optimeeritud funktsioon

```
void foo (uint8_t y)
{
    uint8_t x = (8 + 1) * (0x0f & y);

    EnterCritical();
    /* out(x) tuleb täita 4 takti jooksul*/
    out (x);
    ExitCritical();
}
```

Näidisfunktsioon `foo` ei toimi kõrgemate optimeerimise tasemete juures, kuna kompilaator liigutab kõik matemaatika tehted kriitilise sektsiooni sisse

Switch ja if/else

```
#include <stdint.h>

uint8_t test_switch (uint8_t a)
{
    switch (a)
    {
        case 'A':
            return 0;
        case 'B':
            return 1;
        default:
            return 255;
    };
}

/* if/else lausetega sama funktsioon */
uint8_t test_if (uint8_t a)
{
    if (a == 'A')
        return 0;
    else if (a == 'B')
        return 1;
    else
        return 255;
}
```

8 bitistel kontrolleritel tehakse mõnikord switch'i operatsioonid 16 bitise muutujaga, mis omakorda tekitab suurema koodi.

NB! Uuemad GCC (4.X) kompilaatorid suudavad mõlemad näited niimoodi ära optimeerida, et ei ole suuruse vahet

Switch'i default võti

```
#include <stdint.h>

uint8_t test_switch (uint8_t a)
{
    switch (a)
    {
        case 'A':
            return 0;
        case 'B':
            return 1;
        case 'C':
            return 255;
    };
}

uint8_t test_switch (uint8_t a)
{
    switch (a)
    {
        case 'A':
            return 0;
        case 'B':
            return 1;
        default:
            return 255;
    };
}
```

Iga switch lause peab sisaldama default võtit.

Switch lause puhul tuleks alati leida selline lause mis vastab kõige üldisemale olukorrale ning see ümber nimetada default võtmeks. Sellise ümbernimetamisega saab ära kaotada vähemalt ühe protsessoripoolse võrdlustehte

Staatilised ja globaalsed muutujad

- Staatilisi ja globaalseid muutujad ei ole mõtet algväärtustada nulliks
  - Programmi käivitudes nullitakse eelnevalt kõik staatilised ja globaalsed muutujad ära ning peale nullimist kopeeritakse vastavatele aadressidele algväärtused
- Kõiksugused protsessori registrid on sarnased globaalsetele muutujatele – kehtivad kõik samad reeglid
  - Ideaalis võiks IO registrite muutmisega tegeleda ainult vastavad draiverid

Makrod ja static inline funktsioonid

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

#define fun1_m(x) {x = x / 3; \
                   printf ("  x = %d\n", x);}

static inline void fun1_i (uint8_t num)
{
    num = num / 3;
    printf ("num = %d\n", num);
}

int main (void)
{
    uint8_t y = 60;
    uint8_t z = 60;

    printf ("1: y = %d; z = %d\n", y, z);
    fun1_m (y);
    fun1_i (z);
    printf ("2: y = %d; z = %d\n", y, z);

    return 0;
}

/* programmi väljund */
1: y = 60; z = 60
  x = 20
num = 20
2: y = 20; z = 60
```

Makrode asemel tuleks kasutada static inline funktsioone, kuna sellisel juhul kapseldatakse muutujad funktsiooni sisse.

Static inline funktsioonide puhul tuleks kasutada kolme käsurea võtit: -Winline, -finline-functions ja -fno-keep-inline-functions

#if ja #ifdef

- Mingi funktsionaalsuse lisamiseks on otstarbekam kasutada #if tingumust #ifdef tingimuse asemel
  - #if tingimusel parameetri puudumisel antakse viga, kuid #ifdef parameetri puudumisel viga ei anta

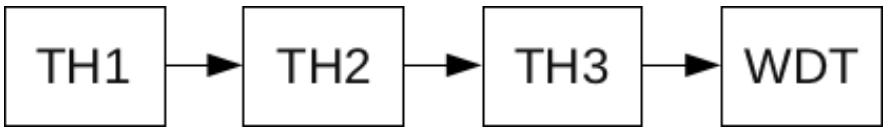
```
#if (X_ENABLED == 0)
#define foo(a) bar(a)
#else
#define foo(a) baz(a)
#endif

#ifdef (X_ENABLED)
#define foo(a) bar(a)
#else
#define foo(a) baz(a)
#endif
```

Watchdog – programmi valvekoer

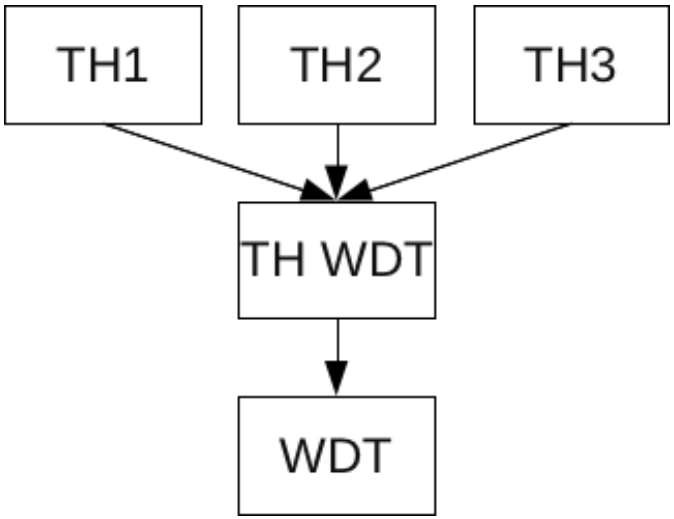
- Vaja pidevalt nullida
  - Tavaliselt pole probleem *super loop* programmis
  - Tekkib raskusi mitme haruga (threadiga) programmides
- Mitme threadiga programmides on kaks (korrektset) võimalust
  - Järjestik nullimine
  - Eraldi threadiga nullimine

Valvekoera nullimine

- Suhteliselt lihtne teha
- Aeglasem thread nullib kiiremat ja kõige kiirem nullib valvekoera



- Keerukas teha, kuid mõnikord ainukene variant
- Threadid ei sõltu üksteisest



Koodi staatiline kontroll

- Kompilaatori poolne
  - `Wall` ja `Wextra` võtmetega
- Väliste programmidega
  - Splint (lint'i edasiarendus)
  - Frama-C
  - Blast (Berkeley Lazy Abstraction Software Verification Tool)
  - Sparse (Linuxi kerneli koodi staatiline kontroll)
  - Clang (C, C++ kompilaatori front end)

Kompilaatori poolne kontroll

- Kompilaatori poolne programmi kontroll võimaldab avastada enamuse vigu mis programmerimise käigus on tehtud.
  - Missiooni ja ohutuskriitilisi programme ei ole mõtet ilma kompilaatori poolse kontrollita (seda lisaks välistele kontrollidele)
- GCC'l on mõtekas sisse lülitada nii `Wextra` kui ka `Wall` kontrollid
  - Need kontrollid küll ei katkesta (enamusjaolt) kompileerimist, kuid lisavad üpriski palju vigadest teavitamise infot. Et nende kontrollidega katkestada kompileerimist tuleks lisada veel ka `Werror` võti

Koodi kontroll Splintiga

- Toimib sarnaselt kompilaatorile koodi kontrollile, aga on tunduvalt põhjalikum, kuid:
  - Splint teeb ainult staatilist koodi kontrolli ja väljastab teateid võimalike ohtklike kohtade kohta koodis
  - Splint on võimeline leidma ainult koodisiseseid ebakooskõlasid
- Splinti peamisteks puudusteks on:
  - Splint ei ole mõeldud sardsüsteemidele ning võib tekitada sellega suurt segadust
  - Tavaliste testide ajal suhteliselt suur veateadete arv, mis omakorda nõuab iteratiivset splinti kasutamist
  - Splintis on mitmeid omavahel vastuolulisi teste

Kodeerimis stiil/standard

- Ühe projekti raames tuleks jääda kindlalt mingi kodeerimis stiili või standardi juurde
  - Kõige hullem asi mis saab ühes projektis olla on see kui on terve projekti jooksul muudetud mitu korda stiili, selline tegevus raskendab oluliselt programmi lugemist ning võib tekitada ka vigu
- Kui ei ole veel mingit välja kujunenud stiili, siis on mõtekas valida mõni tuntud stiilidest
  - Allman (ANSI)
  - K & R

„Viitsütikuga pomm“ I

```
void RunTimeInit(void)
    {
    register uint8_t temp0=0;
        wdt_reset();
        //ScanMatrix();

        for(temp0=0;temp0<60;temp0++) Kar.Karakt[temp0]=0xE4;
                    #if (TESTMODE==1)
                        for(temp0=0;temp0<=DB_KarLEN;temp0++) Kar.Karakt[temp0]=0xE4;
                        for(temp0=temp0;temp0<=(DB_KarLEN*2);temp0++)
Kar.Karakt[temp0]=0x64;
                    #endif
        Kar.Karakt[temp0]=0x00;
        Kar.Karakt[127]=0;  //Igaks juhuks, kuigi viimane peab alati null olema

        /* veel koodi */
    }
```

„Viitsütikuga pomm“ II

```
int main (void)
{
//  INITSIALISEERIMISED...
register uint8_t *reg1;
register uint8_t treg2,treg3;
    CRC_TableCalc();                                        // CRC=R17:R16
    SignatureTest();

    reg1=(uint8_t*)0x210;
    treg2=*reg1;
    reg1++;
    treg3=*reg1;

    for (reg1=(uint8_t*)0x100;(uint16_t)reg1<(uint16_t)0x900;reg1++) *reg1=0;

    reg1=(uint8_t*)0x210;
    *reg1=treg2;
    reg1++;
    *reg1=treg3;


    Status.Hoiatused_RAM=0;
    if ( *((uint16_t*)0x0210) !=( (pgm_read_byte((0x7E00<<1)-
1)<<8)+pgm_read_byte((0x7E00<<1)-2)   )) Status.Hoiatused_RAM|=(1<<CRC_Error);
    EInc2Byte((uint16_t*)&Eepr0.StartCounter);

    init();
    /* veel koodi */
}
```

Kokkuvõte

- Tüübi muutmine juures tuleb jälgida suuremast väiksemasse tüüpi muutused
- Tsüklite kasutamisel peab olema kindel, et tsüklil on lõpu tingimus või siis tsükkel ei võtaks kogu ressurssi endale
- Sisendandmeid tuleb (peaaegu) alati kontrollida
- Ühe projekti raames peab olema kindel kodeerimise stiil

Küsimusi?