

# **Sardtarkvara loomisel kasutatavad programmeerimiskeeled**

Erkki Moorits

Cybernetica AS, Navigatsioonisüsteemide osakond

# Peamised sardsüsteemides kasutatavad programmeerimiskeeled

▶▶ C

▶▶ C++

▶▶ Assembler (ASM)

▶▶ Java ja Python

▶▶ Ada

▶▶ LabVIEW

# C

- ▶ Loodud 1972 aastal Dennis Ritchie poolt asenduseks B keelele
- ▶ Algusest peale mõeldud asenduseks assemblerile
- ▶ Lisaks on pandud väga suur rõhk koodi loetavusele

```
#include <stdio.h>

int main (void)
{
    puts ("Hello world!");
    return 0;
}
```

## C keele plussid

- ▶▶ Hea kompilaatoriga sama kiire programm kui ASM'is
- ▶▶ Oluliselt lihtsam kirjutada ja debugida kui ASM'is kirjutatud programmi
- ▶▶ Piisavalt palju programmeerijaid kes valdavad C keelt
- ▶▶ Paljud teegid on kirjutatud C keeles
- ▶▶ On võimalik suhteliselt lihtsalt kirjutada porditavat koodi

## C keele miinused

- ▶▶ Programmeerimisel on võimalik väga lihtsalt vigu teha
  - Viidatüüpidega on võimalik teha selliseid vigu, mille avastamiseks võib minna mitme kuu töö
- ▶▶ Pole korralike UML'i tööriistu (vähemalt vabavara omasid)
- ▶▶ Lõpliku programmi kiirus, RAM'i kasutus ja programmi suurus sõltub täielikult kompilaatorist ja kasutatavast optimeerimisest
- ▶▶ C kompilaatoril kasutatav optimeerimine võib mõningad koodilõike välja jätta

# C keele kasutuskohad

- ▶▶ Enamusjaolt kasutatakse riistvara lähedaste programmide loomisel:
  - Kernelid ja draiverid
  - Väiksemad ilma kernelita sardsüsteemid (mikrokontrollerite programmid)
- ▶▶ Teegid mis peavad olema kiired:
  - Matemaatika funktsioonid
  - Stringitöötlus
  - Java JNI funktsioonid

## Kohad kus C'd ei kasutata

- ▶ Suured keerukad programmid mis ei ole otseselt seotud riistvaraga (keerukad graafilised kasutajaliidesed, serverid, andmebaasid, jne)
- ▶ Keerukad programmid, mille koodi ei saa kuskilt juba valmiskirjutatuna, kas lähtekoodina või teegina ning on vaja minimaalse ajaga turule saada
- ▶ Ohutuskriitilistes kohtades, välja arvatud juhud kus on kasutatud spetsiaalseid koodi korrasoleku kontrole (IEC 61508 nõue)

# **GCC eripärad**

- ▶ GCC ei sisalda endas linkurit
  - GCC kutsub linkimise staadiumis ise linkuri välja
  - Lõpliku programmi tõmmise valmistamiseks on linkuri (binutils) paketis on olemas kõik vajalikud programmid
- ▶ GCC ei sisalda libc'd
  - libc sisaldab enamust vajalikke funktsioone
- ▶ Ei ole 8 bitiste Microchip'i PIC mikrokontrollerite tuge

# Assembler (ASM)

- ▶▶ Loodud 1950'ndate paiku
- ▶▶ On sisuliselt masinkoodi „tõlge“ inimesele arusaadavamale kujule

```

section .text          ;section declaration
                        ;we must export the entry point to the ELF linker
                        ;or loader. They conventionally recognize _start
                        ;as their entry point.

global _start

section .data          ;section declaration
msg db "Hello, world!",0xa ;our dear string
len equ $-msg          ;length of our dear string

_start:                ;write our string to stdout
    mov edx,len        ;third argument: message length
    mov ecx,msg        ;second argument: pointer to message to write
    mov ebx,1          ;first argument: file handle (stdout)
    mov eax,4          ;system call number (sys_write)
    int 0x80           ;call kernel
                        ;and exit
    mov ebx,0          ;first syscall argument: exit code
    mov eax,1          ;system call number (sys_exit)
    int 0x80           ;call kernel

```

# Veel üks „Hello world!“ näide

```
; (C) Copyright 2001, 2002 Ian P. Cardenas
; <ian.cardenas@ultraviolent.com>
```

```
.data
.cstring
.align 2
msg:
    .asciz "Hello world!\n"
    len = . - msg

.text
.align 2
.globl _start
_start:
    li  r3,1           ; arg1 = stdout = 1
    lis r4,hi16(msg)   ; arg2 = address of string. loaded in two steps
    ori r4,r4,lo16(msg) ; high halfword and then low
    li  r5,len         ; arg3 = numBytes = 12
    li  r0,4           ; 4 is write syscall
    sc                  ; call write
    nop                ; need the no-op because sc skips one instr.
    li  r0,1           ; 1 is exit syscall
    li  r3,0           ; exit status 0
    sc                  ; call exit
```

# Assembleri plussid

- ▶ Saab kasutada kõiki riistvara võimalusi kasutada
- ▶ Saab kasutada sellised instruksioone mida C või mõne muu keele kompilaator ei tunne
- ▶ Programmi transleerimisel masinkoodi ei ole mingit optimeerimist vahel
- ▶ Võimalik teha kindla suurusega või väga kiireid programme

# Assembleri miinused

- ▶▶ Kood on reeglina raskesti loetav
- ▶▶ Suurt programmi väga raske ilma vigadeta kirjutada
- ▶▶ Grupitöö on raskendatud
- ▶▶ Praktiliselt võimatu on teha erinevate arhitektuuride vahel porditavat koodi
- ▶▶ Vägagi keerukas on teha taaskasutatavat koodi

# Assembleri kasutuskohad I

▶▶ Kohad mida ei ole mingil põhjusel võimalik teha muudes keeltes

- Mõndade arhitektuuride schedulerid
- Programmide osad mis kasutavad kompilaatorile tundmatuid käske
- Tsüklite „täitmine“. Näiteks, on vaja ühte lihtsat viidet:

```
/* intuitiivne variant */  
for (i = 0; i < 0x1ff; i++);
```

```
/* üks korrektsetest võimalustest */  
for (i = 0; i < 0x1ff; i++)  
    asm volatile ("nop");
```

# Assembleri kasutuskohad II

- ▶▶ Programmid mis peavad olema kindla suurusega
  - Mõningad bootloaderid, eriti nende esimesed astmed
  - Mõned viirused
- ▶▶ Väga suurt kiirust nõudvad programmilõigud (juhul kui ei ole võimalik mõnes kõrgkeeles kirjutada)
  - DSP ja matemaatika funktsioonid
  - Katkestuste teenindamine
- ▶▶ Kui pole teada täpselt kuidas C kompilaator toimib mingis väga kriitilises kohas

# Mõningad märkused assembleri kasutamise kohta I

- ▶▶ Assemblerit on mõtet ainult siis kasutada kui on täiesti kindel, et teisiti ei saa
- ▶▶ Kui on vaja assemblerit kasutada koos mõne muu keelega, siis peab olema piiratud assembleri kasutamine ainult kindlate funktsioonidega, st. peab kapseldama assembleri lõigud eraldi funktsioonidesse. Näide järgmisel slaidil.

# Assembleri lõikude kapseldamine eraldi funktsioonidesse

```

/* funktsioon 1 */
uint8_t lpm (uint16_t addr16)
{
    uint8_t result;
    asm volatile ("lpm %0, Z" "\n\t" /* Z unchanged */
                 : "=r" (result)
                 : "z" (addr16) /* Z register */);
    return result;
}

/* funktsioon 2 */
void delay (uint8_t ms)
{
    uint16_t cnt;
    uint16_t delay_count = 0x200;
    asm volatile ("L_d11%=: \n\t"
                 "    mov %A0, %A2\n\t"
                 "    mov %B0, %B2\n\t"
                 "L_d12%=: \n\t"
                 "    sbiw %A0, 1\n\t"
                 "    brne L_d12%=: \n\t"
                 "    dec %1\n\t"
                 "    brne L_d11%=: \n\t"
                 : "=&w" (cnt)
                 : "r" (ms), "r" (delay_count));
}

/* lõplik programmilõik */
delay (lpm (0x100));

```

# Mõningad märkused assembleri kasutamise kohta II

- ▶ Kui kasutatakse assemblerit mõne teise keelega, siis ei tohi ilma **väga mõjuva põhjuse**ga reserveerida assemblerile mingeid kindlaid mälualasid või registreid. Näide on järgmisel slaidil.

# Registrite ja mäluualade

## reserveerimine (mittekorrektne)

```
/* muutuja var on suunatud linkuriga mäluualale 0x100 */
static uint8_t __attribute__((section (".asm_section"))) var;

void inc_var (void)
{
    asm volatile ("ld __tmp_reg__, 0x100" "\n\t"
                  "inc __tmp_reg__" "\n\t"
                  "st 0x100, __tmp_reg__" "\n\t"
                  :::);
}

int main (void)
{
    register uint8_t counter asm ("r3");
    /* nullime registri r3 */
    asm volatile ("clr r3");
    while (1)
    {
        counter++;
        inc_var ();
    };
}
```

# Eelmise näite korrektne variant

```
static uint8_t var;

void inc_var (uint8_t *ptr)
{
    asm volatile ("ld __tmp_reg__, %a0" "\n\t"
                 "inc __tmp_reg__" "\n\t"
                 "st %a0, __tmp_reg__" "\n\t"
                 : : "e" (ptr));
}

int main (void)
{
    uint8_t counter = 0;

    while (1)
    {
        counter++;
        inc_var (&var);
    };
}
```

# C++

- ▶ Loodi vahemikus 1979...1986 Bjarne Stroustrup poolt
- ▶ Mõeldud objektorienteeritud täienduseks C keelele

```
#include <iostream>

int main (void)
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

# C++ eelised sardsüsteemides

- ▶ On olemas mitmeid UML tööriistu
- ▶ Võimalik natukene lihtsamini kui C's koodi või testi kattuvust leida
- ▶ Võimaldab mõningatel juhtudel lihtsamalt kirjutada **sama kiireid** programme kuid C's

# C++ pahupooled sardsüsteemides I

- ▶ Väiksematel sardsüsteemidel annab luua ainult staatilisi klasse
  - Dünaamiliste (tööajal loodud) klasside loomiseks on vajalik, et oleksid täidetud kaks tingimust
    - Kontroller peab suutma otse RAM'ist programmi täita
    - Kompilaator ja linker ei tohi siduda programmi koodi kindlate programm-mälu aadressidega. Üldiselt suudab sellisel juhul kompilaator ja linkur luua ka asukohast sõltumatut koodi (Position Independent Code – PIC)

# C++ pahupooled sardsüsteemides II

- ▶ Väiksematel sardsüsteemidel ei pruugi kompilaatorid korraliku koodi teha
- ▶ On mõningal määral keerukam kui C
- ▶ Suhteliselt lihtne tekitada mälulekkeid
- ▶ Tarkvara kirjutamisel 8 ja 16 bitistel kontrollritel ei anna olulisi eeliseid

# C++ eelistatud kasutuskohad

- ▶ Keerukamad graafilised kasutajaliidesed
- ▶ Sobiv suurematel kontrolleritel (32 bitistel) igasuguste rakendusprogrammide kirjutamiseks

# Ada

- ▶ Loodud Jean Ichbiah (CII Honeywell Bull) juhtimise all, DoD (United States Department of Defense) tellimusel
- ▶ On ettenähtud missiooni- ja turvakriitiliste süsteemide (sard)tarkvara loomiseks

```
with Text_IO;
```

```
procedure Hello_World is  
begin  
    Text_IO.Put_Line("Hello World!");  
end Hello_World;
```

# Ada plussid ja miinused

## Plussid

- ▶ Võrreldes teiste keeltega suhteliselt raske teha kriitilisi vigu
  - Vigu on teha raskem kui teistes keeltes kuid mitte võimatu
- ▶ Loodud spetsiaalselt sard- ja reaalaaja süsteemidele

## Miinused

- ▶ Tööajal programmi kontroll võtab palju resurssi
- ▶ Väikene kasutajate ring

# Ada peamised kasutuskohad

▶▶ Lennundus

▶▶ Kosmose süsteemides

▶▶ Relvasüsteemides

# Java

- ▶ Loodud James Gosling'i (Sun Microsystems) poolt, esimene versioon 1995 aastal
- ▶ Java moto - „Write Once, Run Anywhere“

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println ("Hello world!");
    }
}
```

# Java plussid

- ▶ Saab ühel arhitektuuril kirjutatud programmi üle viia teise arhitektuuriga seadmesse ilma, et peaks midagi ümber kompileerima
- ▶ On võimalik suhteliselt kiiresti kirjutada töötavat programmi
- ▶ Ei ole vaja eraldi spetsiaalset arenduskeskonda või riistvara

# Java miinused

- ▶ Reaalselt võiks olla Java moto – „Write Once, Debug Everywhere“
- ▶ Kuna Java programm on tarkvaraliselt interpreteeritav, siis on võrreldes mitteinterpreteeritava programmiga tunduvalt aeglasem (näiteks C's kirjutatud)

# Java kasutuskohad

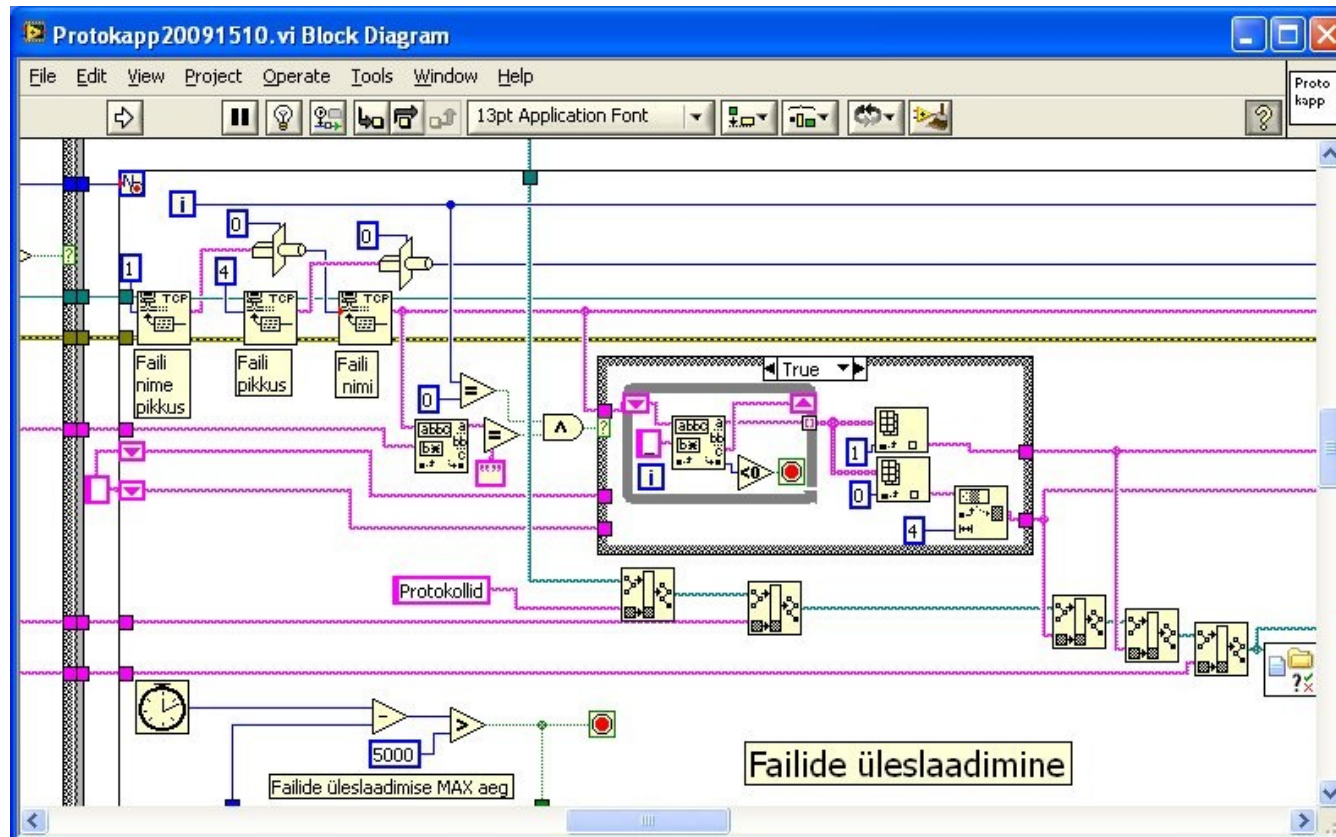
- ▶▶ Kõiksugu multiplatvormsed PC ja serverite programmid
- ▶▶ Mõned mobiiltelefonide kasutaja programmid
- ▶▶ (Üli)koolide juures mõningate robotite laborid
  - Hobikorras on tehtud ka Lego NXT'le ja AVR mikrokontrollerile sobiv Java virtuaalmasin

# Kohad kus Javat ei ole soovitatav kasutada

- ▶ Süsteemides millel on väikene arvutusvõimsus võrreldes andmete sisendvooga (väikse arvutusvõimsusega sardsüsteemid ja süsteemid mis nõuavad väga suurt andmetöötluskiirust)
- ▶ Otseselt riistvara juhtimiseks
- ▶ Range reaalaajaga süsteemid, juhul kui ei kasutata spetsiaalset reaalaja Javat
- ▶ Ohutus- või missioonikriitiliste süsteemide otseseks juhtimiseks

# LabVIEW

- ▶ Graafiline programmeerimiskeskond, mis on suhteliselt intuiitivne ja kergesti õpitav



# LabVIEW

- ▶ Viimastel aastatel on laienenud ka sardsüsteemide valdkonda. Hetkel on toetatud ainult ARM seeria mikrokontrollerid ja Nationali Instruments'i poolt välja töötatud FPGA plaadid.
- ▶ Peamine laialdast levi pärssiv faktor on seni toote hind ja platvormide nappus.
- ▶ Lingid
  - <http://www.ni.com/embedded/>
  - <http://www.ni.com/fpga/>

# Keelte võrdlus

Keel	Koodi kiirus	Mälu nõudlus (RAM ja ROM)	Keele keerukus
C	Suur/väga suur	Vähe	Keerukas
ASM	Suur/väga suur	Vähe	Keerukas
C++	Suur	Vähe	Keerukas
Ada	Keskmine	Keskmiselt	Keskmine
Java	Aeglane	Palju	Lihtne

**NB!** Koodi kiirus on pöördvõrdelises seoses energia tarbega

# Kernelites kasutatavad progarmeerimiskeeled

Keel/ Kernel	C	ASM	C++ või C#
Linux	+	+	
FreeBSD, OpenBSD, NetBSD	+	+	
Darwin	+	+	
Windows	+	+	+
NutOS	+	+	
eCos	+		+
FreeRTOS	+	+	
Contiki	+	+	

## Kokkuvõte

- ▶ Programmeerimiskeeltest on sardsüsteemides kõige paremini kasutatav C ning mõnel juhul ka C++
- ▶ Assemblerit on soovitatav kasutada nendes kohtades kus muid keeli ei saa kasutada
- ▶ Java kasutamine otse riistvara juhtimiseks on raskendatud, kuigi kasutajaliideseid on väga hea teha
- ▶ Missiooni- ja turvakriitilistes süsteemidel on soovitatav kasutada Ada't

# Küsimusi?