**VENTSPILS AUGSTSKOLA**

**INFORMĀCIJAS TEHNOLOĢIJU FAKULTĀTE**

**BAKALAURA DARBS**

**AVR MIKROKONTROLIERU STEKATMIŅAS IZMANTOŠANAS ANALIZATORA IZSTRĀDE**

Autors Ventspils Augstskolas

Informācijas tehnoloģiju fakultātes

bakalaura studiju programmas „Datorzinātnes”

3. kursa students

Anatolijs Koļesņevs

Matr.nr. 22020011

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Fakultātes dekāns doc. Dr.sc.comp. Vairis Caune

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Zinātniskais vadītājs \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ieņemamais amats, zinātniskais nosaukums, vārds, uzvārds)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Recenzents \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ieņemamais amats, zinātniskais nosaukums, vārds, uzvārds)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Ventspils

2025

**ANOTĀCIJA**

**Darba nosaukums:** AVR mikrokontrolieru stekatmiņas izmantošanas analizatora izstrāde.

**Darba autors:** Anatolijs Koļesņevs

**Darba vadītājs:** Mg. sc. ing. Jānis Šmēdiņš

**Darba apjoms:**

**Atslēgas vārdi:**

Bakalaura darbā ir …

**ANNOTATION**

**Title of the thesis:** Development of a stack memory usage analyser for AVR microcontrollers.

**Author:** Anatolijs Koļesņevs

**Supervisor:** Mg. sc. ing. Jānis Šmēdiņš

**Volume of the thesis:**

**Keywords:**

The bachelor's thesis …

**SATURS**

**IEVADS**

**SAĪSINĀJUMU UN NOSACĪTO APZĪMĒJUMU SARAKSTS**

RISC - samazinātas instrukciju kopas dators

ALU - aritmētiski loģiskā ierīce

….

1. **TEORĒTISKAIS PAMATOJUMS**
   1. **AVR mikrokontrolieru arhitektūras apskats**

Katra AVR mikrokontroliera "sirds" ir 8 bitu centrālais procesors, kas balstīts uz RISC (Reduced Instruction Set Computer) principiem. RISC pieeja nozīmē vienkāršākas un efektīvākas instrukcijas, kas ļauj mikroprocesoram darboties ātrāk ar mazāku elektrības patēriņu.

AVR procesora darbības pamatā ir aritmētiski loģiskā ierīce (ALU), kas izpilda visas matemātiskās un loģiskās operācijas. Šī sistēma ir izstrādāta tā, lai vienlaicīgi notiktu divas darbības – kamēr ALU izpilda pašreizējo instrukciju, programmskaitītājs (PC) jau iegūst nākamo instrukciju no programmu atmiņas. Šāda "cauruļvada" pieeja ļauj AVR mikroprocesoram sasniegt iespaidīgu veiktspēju – vienu miljonu instrukciju sekundē (MIPS) uz katru takts frekvences MHz. [1]

AVR mikrokontrolieru atšķirīga iezīme ir to reģistru organizācija - arhitektūra ietver 32 vispārējās nozīmes reģistrus (General Purpose Registers jeb GPR), kur katrs ir 8 bitu (viena baita) izmēra. Šie reģistri ir kartēti datu adresācijas telpas sākumā, adrešu diapazonā 0x00-0x1F. [7]

Šī arhitektūras īpatnība nozīmē, ka programmētājs var piekļūt šiem reģistriem gan tieši, izmantojot reģistru operācijas (kas ir ātrāk), gan arī netieši caur datu atmiņas adresācijas instrukcijām (LD/LDS/LDD un ST/STS/STD). [7]

Būtiska atšķirība starp reģistriem un operatīvo atmiņu ir to funkcionalitāte – reģistri spēj veikt daudzveidīgas operācijas (saskaitīšanu, atņemšanu, loģiskās operācijas, bitu pārbīdes utt.), kamēr operatīvajā atmiņā var tikai glabāt datus. Tāpēc efektīvās mikrokontroliera programmās cenšas pēc iespējas vairāk izmantot reģistrus, un tikai nepieciešamības gadījumā dati tiek pārvietoti uz operatīvo atmiņu. [10]

* + 1. **Atmiņas organizācija AVR mikrokontrolieros**

AVR mikrokontrolieros tiek izmantota Hārvardas arhitektūra, kuras būtiska īpatnība ir programmu un datu atmiņu fiziska un loģiska nošķiršana. Tas nozīmē, ka mikrokontrolierī ir atdalīti gan programmatūras un datu atmiņu adresēšanas apgabali, gan piekļuves kopnes. Šāda arhitektūra ļauj vienlaikus izgūt instrukciju un apstrādāt datus, tādējādi uzlabojot izpildes efektivitāti.

Datu atmiņa mikrokontrolieros parasti sastāv no divām neatkarīgām komponentēm – operatīvās atmiņas (SRAM) un pastāvīgās datu atmiņas (EEPROM), kuras katra atrodas savā adresējamā telpā. Savukārt instrukciju atmiņa jeb programatmiņa tiek realizēta ar Flash tipa ROM.

**Programmu atmiņa (Flash ROM)** ir paredzēta mikrokontroliera izpildāmo instrukciju glabāšanai. Tā ir 16 bitu organizācijas atmiņa, kuras ietilpība dažādos AVR mikrokontrolieru modeļos var svārstīties no 0.5 līdz 384 KB. Flash tehnoloģijas galvenā priekšrocība ir elektriskās pārrakstīšanas iespēja, kas ļauj vairākkārtīgi ierakstīt un dzēst informāciju.

Programmas ierakstīšana Flash atmiņā mūsdienu AVR mikrokontrolieriem ir iespējama, izmantojot vairākas dažādas programmēšanas saskarnes. Mūsdienās vairums jauno AVR mikrokontrolieru izmanto UPDI (Unified Program and Debug Interface) - vienvada protokolu ar ātrumu virs 1Mbps, kas apvieno gan programmēšanas, gan atkļūdošanas funkcijas. Vecāki AVR modeļi izmanto In-System Programming (ISP) caur SPI saskarni ar sešiem pieslēguma punktiem: MISO, MOSI, SCK, RST, VCC un GND. [8, 9]

Papildus UPDI, AVR mikrokontrolieriem var būt arī citas programmēšanas saskarnes, atkarībā no modeļa un ģimenes: JTAG (parasti 40-pinu un lielākiem čipiem), TPI (Tiny Programming Interface dažiem vecākiem ATtiny modeļiem) un PDI (Program and Debug Interface XMEGA sērijai). UPDI saskarnes ieviešana ir daļa no Microchip (kas pārņēma Atmel) stratēģijas, lai visiem jaunajiem AVR mikrokontrolieriem būtu vienota programmēšanas un atkļūdošanas metode.

Atšķirībā no vecākiem ATtiny11 un ATtiny28 modeļiem, kur programmēšana bija ierobežota, mūsdienu AVR mikrokontrolieros UPDI funkcionalitāte dod iespēju programmēt iekārtu tieši uz montāžas plates. Ja UPDI izvads tiek pārkonfigurēts kā GPIO, dažiem modeļiem programmēšanas režīmā var iekļūt, pielietojot 12V impulsu Reset izvadam, ko atbalsta specializētie programmatori. [8]

Turklāt Mega sērijas mikrokontrolieri atbalsta arī pašprogramēšanu (self-programming), tas dod tiem iespēju pašiem mainīt savu programmu atmiņas saturu izpildes laikā. Šī funkcionalitāte ļauj izstrādāt adaptīvas un elastīgas sistēmas, kas var mainīt savu darbības loģiku atkarībā no iekšējiem vai ārējiem nosacījumiem. [1]

Saskaņā ar Atmel korporācijas tehnisko dokumentāciju, AVR otrās paaudzes mikrokontrolieru Flash atmiņai tiek garantēts vismaz 10 000 pārrakstīšanas ciklu. [2]

* + 1. **RAM, Flash un EEPROM raksturojums**

AVR mikrokontrolieros datu atmiņa ir loģiski sadalīta trīs daļās: reģistru atmiņā, operatīvajā atmiņā (RAM) un neatkarīgajā datu atmiņā (EEPROM). Katra no šīm komponentēm pilda noteiktu funkciju mikrokontroliera darbībā.

**Reģistru atmiņu** veido 32 vispārējās nozīmes reģistri (GPR – General Purpose Registers), kas organizēti kā vienots reģistru fails, kā arī speciālie ievades/izvades reģistri (I/O reģistri). Lai gan šie reģistri ir adresējami tajā pašā adresēšanas telpā kā operatīvā atmiņa, fiziski tie nav tās sastāvdaļa.

Ievades/izvades reģistru zonā atrodas dažādi kontroles un stāvokļa reģistri, kas nosaka mikrokontroliera darbību. Tie ietver gan sistēmas vadības reģistrus (piemēram, statusa reģistrus), gan perifērijas ierīču kontroles reģistrus. Patiesībā mikrokontroliera vadība būtībā ir šo reģistru pārvaldība, jo tie ir atbildīgi par visa sistēmas funkcionālā aprīkojuma konfigurēšanu un darbību.

Datu ilgtermiņa glabāšanai tiek izmantota **EEPROM** (Electrically Erasable Programmable Read-Only Memory) – elektriski izdzēšama un pārrakstāma pastāvīgā atmiņa. Tā ir noderīga informācijai, kas var mainīties darbības laikā, piemēram, lietotāja iestatījumiem, konfigurācijas parametriem, sērijas numuriem, kriptogrāfiskajiem atslēgām u.c.

Visi AVR mikrokontrolieri ir aprīkoti ar EEPROM moduli, kura izmērs atkarībā no modeļa var būt no 64 baitiem līdz 4 KB. EEPROM saturs ir pieejams mikrokontroliera programmai izpildes laikā. Ierakstīšanu un nolasīšanu iespējams veikt gan programmēšanas laikā (ar ārēju programmatoru vai SPI interfeisu), gan arī darbības laikā no paša mikrokontroliera koda. EEPROM tipiski garantē vismaz 100 000 pārrakstīšanas ciklu, saglabājot datus arī pēc barošanas atslēgšanas.

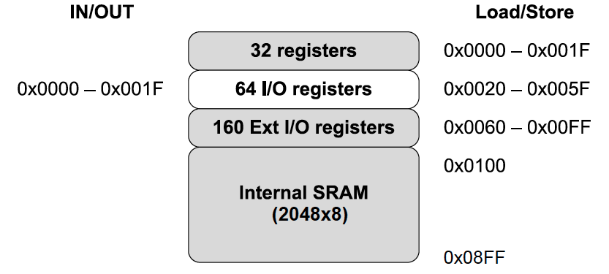
**Operatīvā atmiņa** (RAM – Random Access Memory) AVR mikrokontrolieros tiek realizēta kā statiskā RAM (SRAM), kas organizēta baitos. Tā tiek izmantota mainīgo, kaudzes (heap) un citu īslaicīgu datu glabāšanai izpildes laikā. Atmiņas apjoms dažādos mikrokontrolieru modeļos var būt no 64 baitiem līdz 16 KB.

Tā kā SRAM ir gaistoša atmiņa, tās saturs tiek zaudēts, tiklīdz tiek pārtraukta barošana. Tomēr datu nolasīšanas un ierakstīšanas ciklu skaits nav tehniski ierobežots. Dažiem mikrokontrolieriem ir iespējams pievienot arī ārējo statisko RAM līdz 64 KB, kas ievērojami paplašina iespējas lielāku datu apstrādei. [1]

* 1. **Stekatmiņas koncepcija un tās nozīme**

**Steks** (stack – grēda, krāvums) ir datu struktūra, kura darbojas pēc principa "pēdējais iekšā, pirmais ārā" (LIFO – Last In First Out). Steku raksturo divas pamatoperācijas: ievietošana (push) un izņemšana (pop). Ievietošanas operācija pievieno jaunu elementu steka augšpusē. Izņemšanas operācija izņem elementu, kurš atrodas steka augšpusē, un atgriež to lietotājam. Apskate (peek) – steka elementa pārbaude bez tā izņemšanas. [3]

AVR mikrokontrolieros steka rādītājs (SP- Stack Pointer ) ir 16-bitu reģistrs, kas norāda uz pēdējo aizņemto adresi stekā. Steka rādītāja inicializācijas vērtība ir iestatīta uz augstāko SRAM adresi.



*1.1. att.* ATmega328P mikrokontroliera SRAM atmiņa [7]

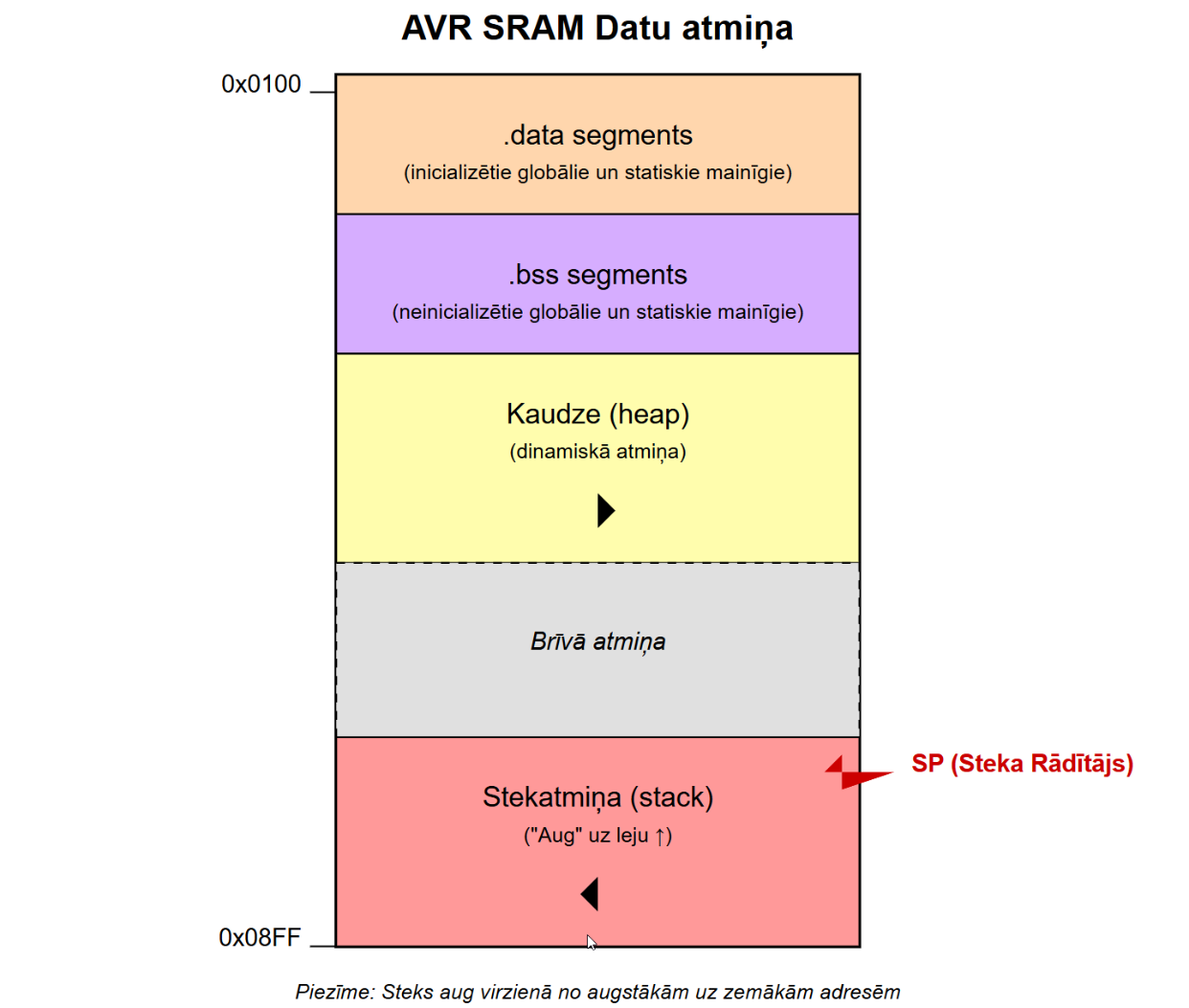
1.1. attēls attēlo ATmega328P mikrokontroliera atmiņas karti un adresēšanas struktūru. Tas parāda, kā AVR arhitektūrā ir organizēta adrešu telpa, iekļaujot reģistrus un SRAM. Attēls parāda četras galvenās komponentes, kas katra aizņem noteiktu adrešu diapazonu atkarībā no modeļa.

**32 reģistri** (0x0000 – 0x001F) - Šie ir 32 vispārējas nozīmes darba reģistri (R0-R31), kas veido reģistru failu. Tie ir ātrākā pieejamā atmiņa mikrokontrolierim un tiek izmantoti tiešām datu apstrādes operācijām. [11]

**64 I/O reģistri** (0x0020 – 0x005F) - Šie reģistri veido standarta I/O atmiņas telpu, kas paredzēta perifērijas ierīču kontrolei un citām I/O funkcijām. Šie reģistri ir īpaši, jo tiem var piekļūt divējādi. Izmantojot IN/OUT instrukcijas - šajā gadījumā tās adresē kā 0x00-0x3F un izmantojot Load/Store instrukcijas - šajā gadījumā tās adresē kā 0x20-0x5F. [11]

**160 paplašinātie I/O reģistri** (0x0060 – 0x00FF) - Šie reģistri tiek izmantoti, lai kontrolētu dažādus mikrokontroliera perifērijas ierīces un funkcijas, piemēram, portus, taimeri, ADC utt. Paplašinātajiem I/O reģistriem var piekļūt tikai ar Load/Store tipa instrukcijām (LD/LDS/LDD un ST/STS/STD), bet ne ar IN/OUT instrukcijām. Tas nozīmē, ka šiem reģistriem nav dubultas adresācijas, kā tas ir standarta I/O reģistriem. Šie reģistri nav tieši bitu-adresējami ar SBI un CBI instrukcijām, kas padara darbu ar atsevišķiem bitiem mazāk efektīvu. [11]

**Iekšējā SRAM** (0x0100 – 0x08FF) - Šī ir mikrokontroliera operatīvā atmiņa, kas AVR ATmega328P gadījumā ir 2048 baiti (2KB). Šeit glabājas programmas dati, mainīgie, un steks. Steks tipiski sākas no augšējās atmiņas adreses (0x08FF) un "aug" lejup virzienā uz zemākām adresēm. [11]



*1.2. att.* ATmega328P mikrokontroliera SRAM datu atmiņa [12]

1.2 attēls parāda AVR mikrokontrolieru SRAM datu atmiņas struktūru, kas ir būtiska, lai izprastu, kā tiek organizēti dati programmas izpildes laikā. SRAM datu atmiņa sākas pēc I/O reģistriem (aptuveni adresē 0x0100, atkarībā no konkrētā AVR modeļa). Šajā attēlā redzamas četras galvenās SRAM daļas:

1. **.data segments** – šajā daļā tiek glabāti inicializētie globālie un statiskie mainīgie. Programmas sākumā šo datu vērtības tiek kopētas no Flash atmiņas.
2. **.bss segments** – šeit atrodas neinicializētie globālie un statiskie mainīgie. Programmas sākumā šī daļa tiek aizpildīta ar nullēm.
3. **Kaudze (heap)** – dinamiskās atmiņas apgabals, kuru programma var pieprasīt izpildes laikā. Kaudze parasti "aug" virzienā no zemākām uz augstākām adresēm.
4. **Stekatmiņa (stack)** – tiek izmantota lokālajiem mainīgajiem, funkciju parametriem, atgriešanās adresēm. Steka rādītājs (SP) norāda uz pēdējo izmantoto atmiņas šūnu. AVR mikrokontrolieros steks aug virzienā no augstākām uz zemākām adresēm (SP samazinās, kad tiek pievienoti dati).

Starp kaudzi un steku parasti ir brīva atmiņa. Ja steks izaug pārāk liels un pārklājas ar kaudzi, rodas "stack overflow" kļūda.

**Funkciju izsaukums: CALL / RCALL**

Funkciju izsaukšanas laikā tiek izmantotas instrukcijas CALL (vai RCALL, ja relatīvs adresējums). Funkcijas izsaukšanas laikā notiek šādas darbības:

1. programmskaitītāja (PC) vērtība, kas norāda uz nākamo instrukciju pēc funkcijas izsaukuma, tiek ievietota stekā. AVR mikrokontrolieriem ar līdz 128 KB programmatūras atmiņu, PC ir 16 biti vai 22 biti (atkarībā no ierīces), tāpēc tiek saglabāti 2 vai 3 baiti,
2. SP tiek samazināts (jo steks aug uz zemākām adresēm),
3. vadība tiek nodota uz funkcijas sākuma adresi (CALL mērķis). [4]

**Atgriešanās no funkcijas: RET**

Instrukcija RET tiek izmantota, lai atgrieztos no funkcijas:

1. No steka tiek izņemta iepriekš saglabātā PC vērtība (2 vai 3 baiti).
2. SP tiek palielināts, atbrīvojot iepriekš izmantotās steka šūnas.
3. Vadība tiek nodota uz šo adresi – t.i., tiek atgriezta kontrole vietai, kur funkcija tika izsaukta. [4]

Lai nodrošinātu funkciju darbības izolāciju un izvairītos no datu zuduma, AVR arhitektūrā tiek plaši izmantotas instrukcijas PUSH un POP. Tās ļauj īslaicīgi saglabāt reģistru vērtības funkcijas izpildes laikā un pēc tam tās atjaunot, nodrošinot konteksta saglabāšanu.

**PUSH – Reģistra ievietošana stekā**

Instrukcija PUSH Rr veic sekojošas darbības:

1. Reģistra Rr saturs tiek ierakstīts SRAM adresē, uz kuru tagad norāda SP,
2. SP samazinās par 1, norādot uz nākamo brīvo vietu stekā. [4]

**POP – Reģistra atjaunošana no steka**

Instrukcija POP Rd darbojas šādi:

1. SP palielinās par 1, jo šī steka šūna vairs nav aizņemta,
2. no SRAM adreses, uz kuru norāda SP, tiek nolasīta vērtība,
3. šī vērtība tiek ievietota reģistrā Rd. [4]

AVR mikrokontrolieros lokālo mainīgo glabāšanai tiek izmantots steks, kas ir kritiski svarīga atmiņas pārvaldības mehānisma sastāvdaļa. Atšķirībā no globālajiem mainīgajiem, kuri tiek glabāti fiksētās SRAM adresēs, lokālie mainīgie tiek izvietoti stekā un ir pieejami tikai funkcijas izpildes laikā. Sarežģītāki datu tipi, piemēram, masīvi un struktūras, stekā tiek glabāti secīgi. [5, 100–104 lpp.]

AVR procesora instrukciju komplektā nav tiešu instrukciju darbam ar steku, izņemot PUSH un POP operācijas. Tādēļ, lai piekļūtu lokālajiem mainīgajiem, tiek izmantots netiešās adresācijas mehānisms, galvenokārt ar Y reģistru (r28). [4]

Lokālo mainīgo piekļuves process:

1. Funkcijas sākumā tiek saglabāti iepriekšējie Y reģistra saturs (ja tas tiek izmantots kā ietvara rādītājs)
2. Y reģistrs tiek iestatīts, lai norādītu uz steka ietvara sākumu
3. Piekļuve lokālajiem mainīgajiem notiek ar nobīdi attiecībā pret Y reģistru
   1. **AVR vispārīgie reģistri un operandu apzīmējumi (Rd, Rr)**

AVR arhitektūrā ir pieejami 32 vispārīgie 8 bitu reģistri, kas apzīmēti kā R0 līdz R31. Šie reģistri ir tieši pieejami instrukcijām un atrodas procesora vispārīgajā reģistru failā. Tie tiek izmantoti gan aritmētiskajām un loģiskajām operācijām, gan datu pārnesei un uzglabāšanai.

**Reģistru priekšrocības:**

* tie ir pieejami vienā ciklā (viena takts laikā), tādēļ darbības ar tiem ir ļoti ātras,
* reģistri R26–R31 var tikt apvienoti pāros (X, Y, Z) un izmantoti kā indeksētie rādītāji (LD, ST u.c. instrukcijās).

**Reģistru apzīmējumi instrukcijās: Rd un Rr**

Instrukciju sintaksē bieži tiek izmantoti saīsinājumi:

* Rd (Destination Register- mērķa reģistrs) – reģistrs, kurā tiks ievietots rezultāts,
* Rr (Source Register - avota reģistrs) – reģistrs, no kura tiek ņemta ievades vērtība (piemēram, saskaitāmais).

Šie apzīmējumi netiek lietoti programmā tieši, bet tiek izmantoti instrukciju formātos un dokumentācijā, lai apzīmētu lomu, kāda katram reģistram ir konkrētajā instrukcijā.

**Piemērs (asemblera valodā) – Aritmētiska operācija:**

ADD R16, R17 ; R16 = R16 + R17

R16 ir Rd – rezultāts tiek glabāts šeit,

R17 ir Rr – tas tiek pieskaitīts.

Ne visas instrukcijas ļauj izmantot jebkuru no 32 reģistriem. Instrukcija LDI (Load Immediate) darbojas tikai ar reģistriem R16 līdz R31, jo tai nepieciešams vairāk bitu, lai kodētu gan konstanti, gan reģistra adresi. Instrukcijas, kas izmanto rādītājus (LD, ST, LPM, utt.), izmanto reģistru pārus (R26:R27, R28:R29, R30:R31) kā adresācijas reģistrus. [4]

1.1. tabula

**AVR mikrokontroliera reģistru un instrukciju pārskats**

|  |  |  |
| --- | --- | --- |
| **Apzīmējums** | **Nozīme** | **Apraksts** |
| R0–R31 | Vispārīgie reģistri | Pieejami operācijām, glabāšanai, rādītājiem |
| Rd | Destination Register | Reģistrs, kurā nonāk rezultāts |
| Rr | Source Register | Reģistrs, no kura ņem ievades vērtību |
| LDI | Load Immediate | Atļauts tikai R16–R31 |
| PUSH/POP | Steka operācijas | Var izmantot R0–R31 |

1.1. tabulā ir iekļauta informācija par vispārīgajiem reģistriem (R0-R31), kurus var izmantot dažādām operācijām, datu glabāšanai un kā adrešu rādītājus. Papildus tam ir aprakstīti speciālie reģistru apzīmējumi, piemēram, mērķa reģistrs (Rd) un avota reģistrs (Rr), kas tiek izmantoti instrukciju pierakstā.

* 1. **Stekatmiņas pārplūdes problēmas un to sekas**

Kad programma mēģina izmantot vairāk stekatmiņas, nekā fiziski ir pieejams – to sauc par stekatmiņas pārplūdi. Tajā brīdī steka rādītāja nonāk aiz pieļaujamās zonas, izraisot nepareizu programmas izpildi vai pat pilnīgu sistēmas avāriju. AVR mikrokontrolieros šis ir īpaši kritisks jautājums šādu iemeslu dēļ.

**Neparedzama uzvedība**: Viena no nopietnākajām stekatmiņas pārplūdes sekām ir neparedzama programmas uzvedība. Kad steks pārplūst, tas sāk pārrakstīt atmiņas apgabalus, kas sākotnēji tam nebija paredzēti. Šis process izpaužas kā šķietami neizskaidrojamas mainīgo vērtību izmaiņas programmā. Piemēram, sensors, kas iepriekš rādīja korektas vērtības, pēkšņi sāk uzrādīt kļūdainas lasījumus, vai LED indikatori sāk mirgot neparedzētā secībā. Šādas uzvedības atkļūdošana ir ārkārtīgi sarežģīta, jo problēmas izpausmes var būt ļoti attālinātas no to patiesā cēloņa.

**Sistēmas avārijas**: Stekatmiņas pārplūde bieži izraisa pilnīgu sistēmas avāriju, kad mikrokontrolieris pārstāj reaģēt uz ievadi vai izpildīt programmu. Tas notiek tāpēc, ka pēc steka pārplūdes turpmākā programmas darbība var pārrakstīt atgriešanās adreses, kas nepieciešamas funkciju izsaukumu pareizai izpildei. Ja atgriešanās adrese tiek bojāta, programmas izpildes plūsma var nonākt neparedzētā koda segmentā vai adresē, kas nav derīga izpildei.

**Ierobežotie atmiņas resursi**: AVR mikrokontrolieri ir projektēti ar ļoti ierobežotiem atmiņas resursiem. Piemēram, plaši izmantotajam ATmega328P mikrokontrolierim, kas ir Arduino Uno platformas pamatā, ir tikai 2KB SRAM atmiņas. No šiem 2048 baitiem stekatmiņa ir tikai daļa, jo atmiņa tiek dalīta starp globālajiem mainīgiem, dinamisko atmiņu un steku.

Šo ierobežoto resursu kontekstā stekatmiņas pārplūde kļūst par īpaši aktuālu problēmu. Ja programma satur dziļi iegultas funkciju izsaukumu hierarhijas vai lielu daudzumu lokālo mainīgo, 2KB SRAM var ātri izrādīties nepietiekami.

**Nav atmiņas aizsardzības mehānismu**: Atšķirībā no modernām datoru operētājsistēmām, kur pastāv atmiņas aizsardzības mehānismi, AVR mikrokontrolieri darbojas bez šādām aizsardzības sistēmām. Tas nozīmē, ka pārplūdušais steks var bez jebkādiem ierobežojumiem pārrakstīt jebkuru atmiņas apgabalu, un aparatūra par to neziņos un nemēģinās novērst.

**Grūti atkļūdojamas problēmas**: Stekatmiņas pārplūdes radītās problēmas ir ārkārtīgi grūti atkļūdot vairāku iemeslu dēļ:

Pirmkārt, simptomi var parādīties tālu no patiesa cēloņa. Piemēram, funkcija, kas izraisa steka pārplūdi, var pabeigt savu izpildi bez acīmredzamām problēmām, bet tās radītie bojājumi parādīsies tikai vēlāk programmas darbībā, kad cita funkcija mēģinās piekļūt bojātajiem datiem.

Otrkārt, problēmas var neparādīties konsekventi. Tās var būt atkarīgas no konkrētiem izpildes apstākļiem vai ievades datiem, kas padara atkļūdošanu vēl sarežģītāku. Piemēram, steka pārplūde var notikt tikai tad, kad lietotājs ievada īpaši garu tekstu vai kad vairākas funkcijas tiek izsauktas specifiskā secībā.

Treškārt, tradicionālās atkļūdošanas metodes, piemēram, izdrukāšana (debug print), var nedarboties, jo pati atkļūdošanas funkcija var patērēt papildu stekatmiņu, tādējādi pasliktinot problēmu vai mainot tās izpausmi. Šo efektu dažkārt dēvē par "Heizenberga nenoteiktību atkļūdošanā" — mēģinājums novērot problēmu maina pašu problēmu. [6, 89-183 lpp.]

Ņemot vērā stekatmiņas pārplūdes nopietno raksturu un potenciālo ietekmi, iegulto sistēmu projektēšanā ir nepieciešams ievērot īpašu piesardzību. Tas ir īpaši svarīgi kritiskās lietojumprogrammās, piemēram:

1. Medicīnas ierīcēs, kur programmatūras atteice var apdraudēt pacientu dzīvību.
2. Automobiļu vadības sistēmās, kur atteice var izraisīt avāriju.
3. Industriālās vadības sistēmās, kur atteice var izraisīt ražošanas zaudējumus vai pat bojāt aprīkojumu.

Stekatmiņas pārplūdes novēršana ir galvenais uzdevums šo sistēmu drošības un uzticamības nodrošināšanā. Efektīva stekatmiņas pārvaldība ietver gan programmatūras izstrādes paņēmienus, gan aparatūras resursu plānošanu, gan testēšanas un verifikācijas metodoloģijas.

1. **AVR STEKATMIŅAS IZMANTOŠANAS ANALIZATORA IZSTRĀDE**
   1. **Izstrādes prasības un mērķi**

Galvenās prasības un mērķi izstrādājot AVR stekatmiņas analizatoru ir šādi:

1. **Precīza steka izmantojuma aprēķināšana** - nodrošināt rīku, kas var ticami prognozēt maksimālo steka izmantojumu mikrokontroliera programmai, kura ir uzrakstīta C valodā.
2. **Ērti izmantojama** - nodrošināt saprotamu un viegli izmantojamu lietotāja saskarni.
3. **Statiskā analīze bez programmas izpildes** - veikt analīzi, balstoties tikai uz pirmkodu un kompilētu asemblera kodu, bez nepieciešamības programmu reāli izpildīt.
4. **Rekursīvo funkciju atbalsts** - korekti analizēt un aprēķināt steka izmantojumu programmās ar rekursīvām funkcijām, ņemot vērā to maksimālo dziļumu.
5. **Optimizācijas atbalsts** - nodrošināt izvēli starp nepieciešamo optimizācijas līmeni.
6. **Detalizēta funkciju analīze** - nodrošināt informāciju par katras funkcijas individuālo steka izmantojumu un tās ietekmi uz kopējo steku.
7. **Izsaukumu grafika vizualizācija** - radīt vizuālu programmas funkciju izsaukumu struktūras attēlojumu ar īpašu uzsvaru uz rekursīvajām funkcijām.
8. **Atbalsts dažādiem AVR mikrokontrolieriem** - nodrošināt analizatora darbību ar dažādiem AVR sērijas mikrokontrolieriem, ņemot vērā to atmiņas īpatnības.
9. **Drošības rezerves iestrāde** - iekļaut aprēķinos drošības koeficientu, lai kompensētu neparedzētus steka izmantojuma gadījumus.
10. **Daudzvalodību atbalsts** - nodrošināt lietotāja saskarni gan latviešu, gan angļu valodā.
    1. **Izmantotās tehnoloģijas un rīki**

Autors izstrādājot steka analizatoru kā galveno programmēšanas valodu izvēlējas Python tās universāluma, vienkāršības un plašo bibliotēku dēļ. Python nodrošina:

* ērtu teksta apstrādi un regulāro izteiksmju lietošanu,
* vienkāršu integrāciju ar ārējiem rīkiem, izmantojot subprocess moduli,
* efektīvu datu struktūru apstrādi grafu un algoritmu realizācijai.

Lai veiktu nepieciešamas kompilācijas tika izmantoti AVR GCC kompilatora komplekta rīki:

* avr-gcc - C koda kompilēšanai uz AVR asemblera kodu,
* avr-objdump - ELF failu disasemblēšanai, lai iegūtu asemblera kodu analīzei,
* avr-size - programmas atmiņas sekciju izmēru noteikšanai.

Statiskās analīzes procesā tiek izmantotas regulārās izteiksmes asemblera koda parsēšanai. Analizators izvelk un apstrādā GCC kompilatora ģenerētos steka izmantojuma (.su) failus apvienojot GCC sniegtos datus ar paša aprēķināto steka lietojumu, izmantojot konservatīvo pieeju - tiek izmantota lielākā no abām vērtībām, lai nodrošinātu drošu novērtējumu.

Analizators izmanto grafu teorijas algoritmus, lai analizētu funkciju izsaukumu struktūru un noteiktu “sliktāko ceļu”.

Izsaukumu grafika vizualizācijai izmantots:

* DOT formāts grafiku specificēšanai,
* Graphviz pakotne DOT failu pārvēršanai attēlos.
  1. **Analizatora arhitektūras projektējums**

Analizatora arhitektūra veidota modulāri, ar skaidri nodalītiem atbildības līmeņiem:

1. Ievades apstrādes modulis.

* C pirmkoda parsēšana.
* Avota koda analīze rekursīvo funkciju meklēšanai.
* Kompilatora karogu apstrāde un kompilēšanas procesa vadība.

1. Kompilēšanas modulis.

* C koda kompilēšana uz ELF formātu.
* Kompilatora steka izmantojuma atskaišu (.su faili) savākšana.
* ELF faila disasemblēšana asemblera koda iegūšanai.

1. Asemblera koda analīzes modulis.

* Asemblera koda parsēšana pa funkcijām.
* Steka operāciju (push, pop, call, sbiw, adiw) atpazīšana un analīze.
* Funkciju izsaukumu grafika rekonstrukcija.

1. Rekursijas analīzes modulis.

* Rekursīvo funkciju identificēšana.
* Rekursijas dziļuma ierobežojumu noteikšana.
* Rekursīvo funkciju steka izmantojuma aprēķināšana.

1. Steka kalkulācijas modulis.

* Individuālo funkciju steka izmantojuma aprēķins.
* Maksimālā kopējā steka dziļuma aprēķins.
* Drošības rezerves piemērošana.

1. Vizualizācijas un atskaišu modulis.

* Teksta atskaišu ģenerēšana.
* JSON formāta atskaites.
* Izsaukumu grafika DOT formāta ģenerēšana.

1. Lietotāja saskarnes modulis.

* Komandrindiņas argumentu apstrāde.
* Dažādu valodu atbalsts.
* Rezultātu formatēšana un attēlošana.
  1. **AVR mikrokontrolieru steka izmantojuma analīzes metodes un algoritmi**
     1. **Statiskie steka izmantošanas aprēķini**

Statiskās analīzes metodes ļauj noteikt steka izmantojumu, analizējot programmas kodu bez tās izpildes. AVR steka analizators izmanto divas metodes: GCC steka lietojuma pārskatu analīzi un asamblera koda analīzi.

Rīks izmanto GCC kompilatora -fstack-usage opciju, kas ģenerē .su failus ar informāciju par katras funkcijas steka izmantojumu. Šī metode nodrošina precīzu steka izmantojuma informāciju par katru individuālo funkciju, ko aprēķinājis kompilators. Taču tā neņem vērā funkciju izsaukumu ķēdes vai rekursiju, kas var izraisīt steka pārplūdi.

Tātad statiskās analīzes pirmais solis ir programmas koda kompilēšana ar īpašiem parametriem, kas ļauj iegūt detalizētu informāciju par steka izmantojumu. AVR-GCC kompilatoram šim nolūkam tiek izmantoti vairāki specifiski karogi:

-fstack-usage: Liek kompilatoram ģenerēt ".su" failus, kas satur informāciju par katras funkcijas steka izmantojumu.

-O0, -O1, -O2, -O3, -Os, -Og: Iestāda optimizācijas pakāpi.

-fno-inline: Atspējo funkciju iekļaušanu (inlining), kas ir svarīgi, lai precīzi analizētu funkciju izsaukumus.

-mmcu=atmega328p: Norāda konkrēto mikrokontroliera tipu, kas ietekmē atmiņas modeli.

-g: Pievieno atkļūdošanas informāciju izpildāmajam failam, kas ļauj saistīt assemblera kodu ar pirmkodu.

Kompilēšanas komanda varētu izskatīties šādi: avr-gcc -mmcu=atmega328p -O0 -g -fstack-usage -fno-inline -fno-inline-small-functions -o program.elf program.c

Šī kompilācija nodrošinās, ka tiek izveidoti visi nepieciešamie faili tālākai analīzei, vienlaikus saglabājot koda struktūru maksimāli tuvu sākotnējam kodam.

Kompilēšana ar "-fstack-usage" karogu ģenerē ".su" paplašinājuma failus, kas satur informāciju par katras funkcijas steka izmantojumu. Šo failu saturs izskatās apmēram šādi:

program.c:10:function\_name 24 static

program.c:20:another\_function 16 dynamic

Katra rinda satur šādu informāciju:

* avota faila ceļš un rindas numurs,
* funkcijas nosaukums,
* steka izmantojums baitos,
* izmantojuma veids ("static" vai "dynamic").

"Static" nozīmē, ka steka izmantojums ir konstants un nav atkarīgs no funkcijas parametriem. "Dynamic" norāda, ka steka izmantojums var mainīties atkarībā no funkcijas parametriem, piemēram, ja funkcijā tiek izveidots masīvs ar izmēru, kas atkarīgs no parametra.

Šo failu parsēšana ir pirmais solis faktiskās steka analīzes veikšanai. Analīzes procesā tiek apkopota informācija par katras funkcijas lokālo steka izmantojumu – cik atmiņas tā aizņem saviem lokālajiem mainīgajiem un saglabātajiem reģistriem.

Papildus GCC pārskatiem, rīks disasemblē programmu, lai analizētu steka operācijas tiešā assemblera līmenī. Šī metode identificē specifiskas steka operācijas AVR asemblera instrukciju kopā, ieskaitot push, pop, call, un steka rādītāju pielāgošanu ar sbiw/adiw instrukcijām. Tā palīdz identificēt arī funkciju izsaukumu grafu - kritisku informāciju steka dziļuma aprēķināšanai.

Analizatora pirmais solis ir assemblera koda iegūšana, izmantojot avr-objdump rīku:

avr-objdump -d program.elf

Komanda atgriež ELF faila disasemblēto kodu. Parametrs -d norāda, ka nepieciešams disasemblēt tikai koda sekcijas. Rezultātā iegūstam pilnu mikrokontroliera programmas assemblera kodu, kas satur visas instrukcijas, ko izpildīs mikrokontrolieris.

Lai veiktu precīzu analīzi, vispirms assemblera kods tiek sadalīts pa funkcijām. Tas ļauj veikt analīzi katrai funkcijai atsevišķi. Analizators izmanto regulāro izteiksmi, lai identificētu funkciju sākumus assemblera kodā. **function\_pattern = re.compile(r'^[0-9a-f]+ <([^>]+)>:')** AVR assemblera kodā funkcijas sākas ar šādu struktūru: **00000084 <function\_name>:**. Pēc funkcijas identificēšanas, visas tās instrukcijas tiek apkopotas un saglabātas vārdnīcā, kur atslēga ir funkcijas nosaukums. Rezultātā iegūstam vārdnīcu functions, kur katrā ierakstā ir funkcijas nosaukums un tās pilns assemblera kods.

Pēc koda strukturēšanas seko galvenais analīzes posms – steka operāciju identificēšana un to ietekmes modelēšana.

|  |
| --- |
| # Rakstu šabloni steka operācijām  push\_pattern = re.compile(r'\s+push\s+')  pop\_pattern = re.compile(r'\s+pop\s+')  call\_pattern = re.compile(r'\s+(?:call|rcall|icall)\s+(?:0x)?([0-9a-f]+)?\s\*<?([^>\s]\*)')  sbiw\_pattern = re.compile(r'\s+sbiw\s+r(2[89]),\s\*(\d+)') # Y rādītāja pielāgošana  adiw\_pattern = re.compile(r'\s+adiw\s+r(2[89]),\s\*(\d+)') # Y rādītāja pielāgošana |

*2.1. koda fragments.* Regulāro izteiksmju definīcijas AVR asemblera steka operāciju atpazīšanai

2.1 koda fragmentā tiek definētas regulārās izteiksmes, kas atpazīs dažādas steka operācijas AVR assemblera kodā. Katra no šīm operācijām ietekmē steka izmantojumu citādi. **Push** instrukcija pievieno vienu baitu stekā, tādējādi palielinot steka izmantojumu. **Pop** instrukcija izņem vienu baitu no steka, samazinot steka izmantojumu. Funkciju izsaukuma instrukcijas **call/rcall/icall** pievieno atgriešanās adresi stekā. AVR arhitektūrā atgriešanās adrese aizņem 2 baitus. **SBIW/ADIW** instrukcijas ar Y reģistru tiek izmantotas, lai manipulētu ar steka rādītāju, rezervējot vietu lokālajiem mainīgajiem vai atbrīvojot to.

Pēc regulāro izteiksmju definēšanas steka analīzē notiek sistemātisks process, kurā katras funkcijas assemblera kods tiek analizēts rindu pa rindai, modelējot steka izmaiņas katrā instrukcijā. Šis process ir analīzes kodola daļa, kas dod iespēju noteikt maksimālo steka izmantojumu.

Pēc regulāro izteiksmju definēšanas analizators veic katras funkcijas assemblera koda pārskatīšanu, lai modelētu steka stāvokli un maksimālo izmantojumu. Šajā brīdī notiek vairāki būtiski procesi:

1. **Steka stāvokļa izsekošana** - **current\_stack** mainīgais dinamiski seko pašreizējam steka izmantojumam, reaģējot uz katru instrukciju, kas to maina. Šis ir līdzīgi kā "virtuālā steka" simulēšana.
2. **Funkciju izsaukumu identificēšana** - kad tiek atrasta call/rcall/icall instrukcija, tā:

* Identificē izsauktās funkcijas nosaukumu.
* Pievieno to funkciju izsaukumu grafikam, ja tā vēl nav pievienota.
* Palielina pašreizējo steka izmantojumu par 2 baitiem (atgriešanās adresei).

1. **Push/Pop operāciju apstrāde** - katru reizi, kad tiek atrasta push instrukcija, steka izmantojums palielinās par 1 baitu, savukārt pop instrukcija to samazina par 1 baitu.
2. **Steka rāmja pielāgojumu apstrāde** - īpaši svarīgas ir sbiw un adiw instrukcijas ar r28/r29 (Y reģistru), jo tās norāda uz vietas rezervēšanu lokālajiem mainīgajiem:

* sbiw r28, X norāda, ka tiek rezervēti X baiti lokālajiem mainīgajiem.
* adiw r28, X norāda, ka tiek atbrīvoti X baiti, kas iepriekš bija rezervēti.

1. **Maksimālā steka izmantojuma izsekošana** - vienmēr tiek saglabāta lielākā steka izmantojuma vērtība, kas novērota funkcijas analīzes laikā.

Īpaši svarīgs ir mainīgais **frame\_size**, kas uzkrāj kopējo lokālo mainīgo izmēru, kas rezervēts ar sbiw instrukcijām. Tas ir būtiski, jo lokālie mainīgie veido lielu daļu no funkcijas steka izmantojuma. Katru reizi, kad tiek atrasta sbiw instrukcija ar Y reģistru, šī vērtība tiek atjaunināta.

Kad visa funkcija ir analizēta, rīks izmanto abu metožu priekšrocības, lai iegūtu precīzāku novērtējumu. Šāda pieeja nodrošina, ka tiek ņemts vērā gan GCC konservatīvais steka izmantojuma novērtējums, gan tiešā asemblera koda analīze, izvēloties lielāko vērtību. (2.2. koda fragments)

|  |
| --- |
| # Izmanto vai nu šīs programmas aprēķināto vērtību, vai GCC vērtību, kura ir lielāka  gcc\_value = gcc\_stack\_usage.get(function\_name, 0)  calculated\_value = max\_stack\_size + frame\_size  # Saglabā lielāko no abām vērtībām  function\_stack\_usage[function\_name] = max(calculated\_value, gcc\_value) |

*2.2. koda fragments.* Konsolidētā steka izmantojuma aprēķināšana, kombinējot asamblera analīzi ar kompilatora ģenerētiem datiem

* + 1. **Rekursīvo funkciju analīze**

Rekursīvās funkcijas rada īpašu izaicinājumu steka analīzē, jo to steka izmantojums ir atkarīgs no rekursijas dziļuma. AVR steka analizators izmanto vairākas metodes, lai precīzi identificētu un analizētu rekursīvās funkcijas.

Divas metodes rekursīvo funkciju identificēšanai:

1. Pārbauda, vai funkcija tieši izsauc sevi.
2. Meklē raksturīgos rekursijas samazināšanas šablonus:

* Samazināšana par skaitlisku konstanti: if (x > 0) { func(x - N); },
* Dalīšana ar skaitli: if (x > 1) { func(x / N); },
* Bitu bīdīšana pa labi (dalīšana ar 2^N): if (x > 1) { func(x >> N); }.

Rekursīvo funkciju identificēšana ir daudz sarežģītāka nekā vienkārša funkcijas nosaukuma meklēšana kodā. AVR steka analizators izmanto vairākas stratēģijas, lai atšķirtu patiesi rekursīvas funkcijas no šķietami līdzīgām konstrukcijām.

Funkcijas konteksta analīze ir fundamentāla, jo ļauj precīzi noteikt, vai funkcijas nosaukuma parādīšanās ir patiess pašizsaukums vai tikai nosaukuma sakritība citā kontekstā. Analizators izolē katras funkcijas ķermeni, pārliecinoties, ka atrastais izsaukums atrodas tieši funkcijas koda blokā, nevis, piemēram, komentāros vai virkņu literāļos.

Nosacīto pašizsaukumu analīze identificē rekursiju pat tad, kad kompilators veic optimizāciju. Daudzās rekursīvās funkcijās ir raksturīga "bāzes gadījuma" pārbaude ar nosacījumu, kas nosaka rekursijas apstāšanos. Šādu modeļu atpazīšana, izmantojot iepriekš minētos šablonus, ir īpaši noderīga, kad tiešā asemblera koda analīze nespēj identificēt rekursiju kompilācijas optimizāciju dēļ.

Rekursijas dziļuma noteikšana ir izšķiroša precīzai steka analīzei, jo katra rekursīvā izsaukuma līmenī steks pieaug, potenciāli sasniedzot kritisku punktu. Analizators izmanto vairākas stratēģijas, lai tuvotos reālai situācijai.

Sākotnējo vērtību izsekošana balstās uz principu, ka rekursijas dziļums bieži ir tieši atkarīgs no sākotnējām vērtībām, ar kurām tiek izsaukta funkcija. Piemēram, factorial(5) izsaukums norāda uz 5 līmeņu dziļu rekursiju (plus bāzes gadījums). Analizators meklē gan tiešus izsaukumus ar skaitliskām vērtībām, gan mainīgo deklarācijas, lai noteiktu potenciālo maksimālo dziļumu.

Mainīgo vērtību dinamiskā analīze paplašina iepriekšējo stratēģiju, izsekojot ne tikai tiešos izsaukumus, bet arī mainīgo vērtības, kas tiek nodotas funkcijai. Šī pieeja atrisina problēmu, kad rekursīvā funkcija tiek izsaukta ar mainīgā, nevis literāla vērtību.

Drošības noklusējumu stratēģija nodrošina, ka pat situācijās, kad nav iespējams precīzi noteikt rekursijas dziļumu, analizators pieņem konservatīvu, bet drošu vērtību (noklusējumā 10 līmeņi). Šī pieeja atspoguļo inženierijas principu, ka labāk pārvērtēt nekā nepietiekami novērtēt potenciālo steka izmantojumu.

Īpaši interesanta problēma ir atšķirt patiesu rekursiju no bezgalīgajiem cikliem, jo īpaši main() funkcijā. Šī diferenciācija ir būtiska vairāku iemeslu dēļ.

Izpildes modeļu atšķirība starp rekursiju un bezgalīgo ciklu ir fundamentāla – rekursija ar katru izsaukumu pievieno jaunu slāni stekā, kamēr bezgalīgais cikls paliek vienā steka līmenī. Analizators identificē tipiskos bezgalīgo ciklu modeļus, piemēram, while(1), while(true) vai for(;;), kas ir izplatīti mikrokontrolieru programmēšanā.

Iegulto sistēmu konteksts nosaka, ka main() funkcija AVR mikrokontrolieros gandrīz vienmēr satur bezgalīgo ciklu, kas nepārtraukti apstrādā ievadi un izpilda programmas loģiku. Šī īpatnība ir tik izplatīta, ka analizators īpaši pārbauda main() funkciju, lai izvairītos no tās kļūdainas klasificēšanas kā rekursīvās. (2.3 koda fragments)

|  |
| --- |
| def is\_infinite\_loop\_function(self, func\_name):  """Nosaka, vai funkcija satur bezgalīgu ciklu."""  if func\_name != 'main':  return False  # Meklē funkcijas definīciju  func\_def\_pattern = re.compile(r'(?:\w+\s+)+' + re.escape(func\_name) + r'\s\*\([^)]\*\)\s\*{', re.DOTALL)  # [...]  # Meklē while(1) vai while(true) vai for(;;) modeli  infinite\_loop\_pattern = re.compile(r'while\s\*\(\s\*(1|true)\s\*\)|for\s\*\(\s\*;\s\*;\s\*\)')  if infinite\_loop\_pattern.search(func\_body):  return True  return False |

*2.3. koda fragments.* Bezgalīgā cikla noteikšanas funkcija

2.3. koda fragments identificē bezgalīgus ciklus, kas ir izplatīti iegulto sistēmu main() funkcijās. Šādi cikli netiek uzskatīti par "īstiem" rekursīviem izsaukumiem, jo tie nepalielina steka izmantojumu ar katru iterāciju.

Rekursīvo funkciju steka kalkulācija balstās uz matemātisku modeli, kas ņem vērā mikrokontrolieru arhitektūras īpatnības.

Atgriešanās adrešu uzskaite ir kritiska, jo AVR arhitektūrā katra funkcijas izsaukuma laikā stekā tiek saglabāta atgriešanās adrese (2 baiti). Rekursīvā funkcija ar dziļumu D rada D atgriešanās adreses stekā, papildus funkcijas lokālajiem mainīgajiem.

Kumulatīvā steka aprēķins apvieno visus elementus - lokālos mainīgos, saglabātos reģistrus un atgriešanās adreses - katram rekursijas līmenim. Analizators izmanto formulu (2.1.).

**Kopējais steks = Pirmā izsaukuma lokālais steks + (Dziļums - 1) × (Lokālais steks + Atgriešanās adreses izmērs),** (2.1.)

Šis aprēķins atšķiras no parastām funkcijām, kur steks ir vienkārši lokālo mainīgo un saglabāto reģistru summa.

Izsaukumu grafikā rekursīvās funkcijas rada īpašu izaicinājumu, jo tās veido ciklus, kas var radīt bezgalīgu stāvokļu skaitu. Analizators risina šo problēmu ar vairākām stratēģijām.

Rekursīvo funkciju izolācija ļauj tās apstrādāt atsevišķi no pārējā izsaukumu grafika. Kad analizators rekonstruē izsaukumu grafiku, tas identificē rekursīvos ceļus un apstrādā tos kā atsevišķus apakšgrafikus ar iepriekš aprēķinātu steka patēriņu.

Memoizācija analīzes algoritmā ļauj izvairīties no atkārtotas aprēķināšanas un potenciālām bezgalīgām cilpām pašā analīzes procesā. Izsaukumu grafika pārskatīšanas laikā analizators atceras jau apmeklētos ceļus un to aprēķinātos steka patēriņus, tādējādi efektīvi risinot ciklisko grafiku problēmu.

Šī vispusīgā pieeja rekursīvo funkciju analīzei ļauj analizatoram precīzi modelēt steka izmantojumu pat sarežģītās programmās ar dažādiem rekursijas veidiem, nodrošinot uzticamus rezultātus mikrokontrolieru programmatūras izstrādei ar ierobežotiem atmiņas resursiem.

* + 1. **Izsaukumu grafika rekonstrukcija no asemblera**

Precīza izsaukumu grafika rekonstrukcija ir būtiska maksimālā steka dziļuma aprēķināšanai. Rīks veic šo uzdevumu, analizējot gan asemblera kodu, gan pirmkodu. Izsaukumu grafika analīze tiek papildināta ar pirmkoda analīzi gadījumos, kad asemblera koda analīze sniedz nepilnīgu informāciju. Šis ir kritisks solis, jo daudzos gadījumos kompilators var optimizēt izsaukumus (piemēram, izmantojot funkciju ievietošanu), kas apgrūtina izsaukumu grafika rekonstrukciju tikai no asemblera koda.

Izsaukumu grafiks ir datu struktūra, kas attēlo, kā funkcijas programmas izpildes laikā izsauc citas funkcijas. Katrs grafika mezgls ir funkcija, bet šķautnes parāda izsaukuma attiecības starp tām. Steka analīzē šī informācija ir kritiski svarīga, jo:

* Steka pieaugums ir kumulatīvs - kad funkcija A izsauc funkciju B, funkcijai B nepieciešamā steka atmiņa tiek pievienota jau esošajam funkcijas A steka izmantojumam.
* Maksimālais steka izmantojums rodas specifiskā izpildes ceļā - tikai rekonstruējot pilnīgu izsaukumu grafiku, var identificēt tieši to programmas izpildes ceļu, kas izmanto visvairāk steka atmiņas.
* Rekursīvie izsaukumi ir īpaši nozīmīgi - grafika cikli norāda uz rekursīviem izsaukumiem, kur steka izmantojums var pieaugt proporcionāli rekursijas dziļumam.

AVR mikrokontrolieru arhitektūrā funkciju izsaukumi notiek, izmantojot specifiskas asemblera instrukcijas. Analizators atpazīst šīs instrukcijas, izmantojot regulāro izteiksmju meklēšanu disasemblētajā kodā. (2.4. koda fragments)

|  |
| --- |
| call\_pattern = re.compile(r'\s+(?:call|rcall|icall)\s+(?:0x)?([0-9a-f]+)?\s\*<?([^>\s]\*)')  for line in function\_code.split('\n'):  call\_match = call\_pattern.search(line)  if call\_match:  called\_function = call\_match.group(2)  if called\_function and called\_function not in call\_graph[function\_name] and called\_function in gcc\_stack\_usage:  call\_graph[function\_name].append(called\_function)  logger.debug(f"Found call from {function\_name} to {called\_function}") |

*2.4. koda fragments.* Funkciju izsaukumu atklāšana un grafu veidošana

2.4. koda fragments meklē trīs veidu izsaukuma instrukcijas AVR asemblerā:

* 1. **call** - absolūts izsaukums (izmanto absolūto adresi),
  2. **rcall** - relatīvs izsaukums (izmanto relatīvo nobīdi no pašreizējās adreses),
  3. **icall** - netiešs izsaukums (adresi norāda reģistrs, parasti izmanto funkciju norādēm).

Katra atpazītā izsaukuma instrukcija tiek analizēta, lai identificētu mērķa funkciju un atjauninātu izsaukumu grafiku.

Tiešo izsaukumu identificēšana (ar call un rcall instrukcijām) parasti ir vienkārša, jo mērķa funkcijas nosaukums vai adrese ir tieši norādīta instrukcijā. Sarežģītāks ir netiešo izsaukumu gadījums (ar icall instrukciju), kur mērķa funkcija nav tieši redzama. Šādos gadījumos analizators mēģina rekonstruēt izsaukuma mērķi, analizējot iepriekšējās instrukcijas, kas iestatījušas Z reģistra vērtību.

Netiešo izsaukumu identifikācija ir sarežģīta un ne vienmēr pilnībā iespējama tikai no asemblera koda, jo:

1. Z reģistra vērtība var tikt iestatīta dinamiski programmas izpildes laikā,
2. funkciju norādes var tikt glabātas tabulās vai masīvos,
3. var tikt izmantotas sarežģītas adresācijas shēmas.

Viena no lielākajām problēmām asemblera koda analīzē ir kompilatora optimizācijas, kas var būtiski mainīt sākotnējo koda struktūru:

1. **Funkciju ievietošana** – kompilators var aizstāt funkcijas izsaukumu ar pašas funkcijas kodu, pilnībā likvidējot izsaukuma instrukciju.
2. **Funkciju sapludināšana** – līdzīgas funkcijas var tikt apvienotas, lai samazinātu koda izmēru.
3. **Rekursijas transformācija** – rekursīvi izsaukumi var tikt pārveidoti par cikliem.
4. **Astes rekursijas optimizācija** – īpašs rekursijas veids, kur rekursīvais izsaukums ir funkcijas pēdējā darbība, var tikt optimizēts, lai neizmantotu papildu steka atmiņu.

Šīs optimizācijas var radīt situācijas, kur pirmkodā esošie funkciju izsaukumi asemblerā nav tieši redzami, radot nepilnīgu izsaukumu grafiku.

Lai risinātu asemblera koda analīzes ierobežojumus, analizators kombinē to ar pirmkoda (C valodas) analīzi:

|  |
| --- |
| known\_functions = list(gcc\_stack\_usage.keys())  if 'main' in call\_graph and not call\_graph['main']:  logger.warning("No calls detected from main(). Scanning source code for function calls.")  for func in known\_functions:  if func != 'main' and re.search(r'\b' + re.escape(func) + r'\s\*\(', self.source\_content):  call\_graph['main'].append(func)  logger.info(f"Found call from main to {func} in source code") |

*2.5. koda fragments.* Izsaukumu grafika papildināšana ar pirmkoda analīzi, kad asemblera analīze nesniedz rezultātus

2.5. koda fragments risina situāciju, kad asemblera koda analīze neatklāj visus 'main' funkcijas izsaukumus, parasti optimizāciju dēļ. Analizators tad pārslēdzas uz pirmkoda meklēšanu, identificējot funkciju izsaukumus, izmantojot regulārās izteiksmes.

Pirmkoda analīzē tiek izmantotas regulārās izteiksmes, lai atpazītu funkciju izsaukumus: **func\_call\_pattern = r'\b' + re.escape(func\_name) + r'\s\*\([^)]\*\)'**

Šī izteiksme atpazīst funkcijas nosaukumu kā atsevišķu vārdu (\b nodrošina vārda robežu), pēc kura seko neobligāta atstarpe (\s\*), tad atveroša iekava (\(), jebkuri simboli, izņemot aizverošo iekavu ([^)]\*) un nobeigumā aizverošā iekava (\)).

Lai pilnveidotu izsaukumu grafiku, analizators papildus meklē funkciju deklarācijas un definīcijas pirmkodā: **func\_def\_pattern = re.compile(r'(?:\w+\s+)+' + re.escape(func\_name) + r'\s\*\([^)]\*\)\s\*{', re.DOTALL).**

Šī izteiksme atpazīst funkcijas definīciju, kas sastāv no atribūtiem un atgriešanas tipa ((?:\w+\s+)+), funkcijas nosaukuma, parametru saraksta un funkcijas ķermeņa sākuma ar atverošo figūriekavu. Funkciju deklarāciju un definīciju identifikācija ļauj analizatoram izveidot pilnīgāku funkciju sarakstu, ko izmantot izsaukumu grafika rekonstrukcijā.

Lai atvieglotu analīzes rezultātu interpretāciju, analizators piedāvā izsaukumu grafika vizualizāciju DOT failā, kas var tikt pārveidots par grafisku attēlojumu, izmantojot rīkus kā Graphviz. Vizualizācija iezīmē rekursīvās funkcijas ar atšķirīgu krāsu un rekursīvos izsaukumus ar biezākām, sarkanām līnijām.

* + 1. **Maksimālā steka dziļuma aprēķināšana**

Maksimālā steka izmantojuma analīze apvieno visus iepriekšējos soļus vienotā algoritmā:

1. **Atsevišķi aprēķina rekursīvu funkciju steka izmantojumu**:

* Ņem vērā katra rekursijas līmeņa ietekmi uz steku.
* Ņem vērā atgriešanās adreses pārklājumu (2 baiti AVR arhitektūrā).
* Izmanto iepriekš noteiktos rekursijas dziļuma ierobežojumus.

1. **Izmanto dinamisko programmēšanu (memoizāciju)** steka izmantojuma aprēķināšanai:

* Saglabā jau aprēķinātos ceļus.
* Novērš atkārtotus aprēķinus sarežģītos izsaukumu grafos.
* Efektīvi apstrādā cikliskos grafus.

1. **Identificē maksimālo steka ceļu**:

* Katram funkciju izsaukumam atrod ceļu, kas izmanto visvairāk steka.
* Ņem vērā atgriešanās adreses pārklājumu.
* Apvieno vietējo steka izmantojumu ar izsaukuma ķēdes steka izmantojumu.

1. **Sāk analīzi no main() funkcijas**:

* Main() funkcija iegultajās sistēmās parasti ir izpildes steka pamatā, no kuras sākas visa programmas darbība.
* Nodrošina, ka tiek analizēti visi iespējamie izpildes ceļi.

Rīks izmanto kombinētu pieeju, iegūstot izsaukumu grafiku gan no asemblera koda, gan no pirmkoda, lai nodrošinātu maksimālu precizitāti. (2.5. koda fragments) Šī pieeja risina problēmu, kad funkciju izsaukumi tiek optimizēti kompilēšanas laikā, padarot izsaukumu grafika rekonstrukciju no asemblera koda vien nepilnīgu.

Maksimālā steka dziļuma precīza aprēķināšana ir viens no steka analizatora kritiskākajiem uzdevumiem, jo šis rādītājs nosaka, vai programma var droši izpildīties konkrētā mikrokontroliera ierobežotajā atmiņas telpā. Atšķirībā no vienkāršas visu funkciju steka patēriņu summēšanas, reālas programmas izpildes laikā steka izmantojums ir dinamiska parādība. Programma var izpildīties pa dažādiem ceļiem, un katrs no šiem ceļiem var radīt atšķirīgu steka patēriņu. Izstrādātais analizators balstās uz fundamentālu principu, kas precīzi atspoguļo steka uzvedību AVR mikrokontrolieru arhitektūrā.

Funkcijas kopējais steka patēriņš = Funkcijas vietējais steka patēriņš + Maksimālais steka patēriņš no visām izsauktajām funkcijām

Šis princips atspoguļo reālo steka uzvedību izpildes laikā:

* Kad funkcija sāk izpildīties, tā aizņem steka vietu saviem lokālajiem mainīgajiem un saglabātajiem reģistriem.
* Kad šī funkcija izsauc citu funkciju, jaunās funkcijas steka patēriņš "uzliekas" virs esošā steka.
* Steka atmiņa tiek atbrīvota tikai tad, kad funkcija pabeidz izpildi un atgriežas.

Līdz ar to, lai noteiktu maksimālo steka dziļumu visai programmai, analizatoram jāmeklē "sliktākais izpildes ceļš" – tāda funkciju izsaukumu ķēde, kas rada vislielāko kumulatīvo steka patēriņu.

Lai atrastu maksimālo steka dziļumu, analizators izmanto rekursīvu "dziļumā vispirms" (depth-first) algoritmu, kas soli pa solim izpēta visus iespējamos izpildes ceļus izsaukumu grafikā. Algoritma pamatā ir ideja, ka jāiziet cauri visiem potenciālajiem ceļiem un jāatrod tas, kas rada vislielāko steka patēriņu.

Algoritma darbību var aprakstīt šādi:

1. Sākt no programmas ieejas punkta (parasti funkcija main).
2. Katrai funkcijai aprēķināt tās vietējo steka patēriņu (lokālie mainīgie, saglabātie reģistri).
3. Identificēt visas funkcijas, kuras tā var izsaukt (izmantojot izsaukumu grafiku).
4. Rekursīvi aprēķināt steka patēriņu katram iespējamam izpildes ceļam.
5. Atrast ceļu ar vislielāko steka patēriņu.
6. Atgriezt kopējo steka patēriņu kā vietējā patēriņa un maksimālā izsaukuma patēriņa summu.

Kritiskākais solis maksimālā steka dziļuma aprēķināšanā ir dažādu izpildes ceļu salīdzināšana, lai atrastu tieši to, kas patērē visvairāk steka atmiņas. Šī ceļu salīdzināšana notiek katras funkcijas analīzes laikā un ir algoritma pamatelements.

Kad analizators sastop funkciju, kas var izsaukt vairākas citas funkcijas, tas rekursīvi izpēta katru iespējamo ceļu un salīdzina to steka patēriņu. Šo procesu var ilustrēt ar 2.6 koda fragmentu.

|  |
| --- |
| max\_call\_stack = 0  new\_path = call\_path + [func\_name]  # Pārbauda katru izsaukto funkciju  for called\_func in call\_graph.get(func\_name, []):  # Izlaiž pašizsaukumus rekursīvām funkcijām, kuras jau esam apstrādājuši atsevišķi  if called\_func == func\_name and func\_name in recursive\_functions:  continue  # Aprēķina steku šim izsaukumam  call\_overhead = 2 # Atgriešanās adrese (2 biti AVR)  # Aprēķina steku šim izsaukumam  called\_stack = get\_stack\_usage(called\_func, new\_path, depth + 1)  # Atjaunina maksimālo izsaukuma steka izmantojumu  path\_stack = call\_overhead + called\_stack  if path\_stack > max\_call\_stack:  max\_call\_stack = path\_stack |

*2.6. koda fragments.* Rekursīvā steka dziļuma aprēķināšana, izvairoties no bezgalīgām cilpām funkciju izsaukumu grafā

Šis salīdzināšanas mehānisms ir ļoti svarīgs, jo tas nodrošina, ka analizators atradīs tieši to izpildes ceļu, kas patērē visvairāk steka atmiņas. Visu iespējamo ceļu sistemātiska salīdzināšana ir garantija, ka neviens potenciāli bīstams ceļš netiks izlaists.

Nozīmīgi ir tas, ka šis process notiek rekursīvi visā izsaukumu grafikā, izpētot visus iespējamos ceļus, sākot no programmas ieejas punkta. Rezultātā tiek identificēts pilnīgs funkciju izsaukumu ceļš, kas rada maksimālo steka patēriņu – tieši šis steka patēriņa lielums ir kritiskais rādītājs, kas nosaka, vai programma var droši darboties konkrētā mikrokontrolierī.

Rekursīvās funkcijas prasa īpašu pieeju, jo to steka patēriņš ir atkarīgs no rekursijas dziļuma. Analizators vispirms identificē rekursīvās funkcijas un aprēķina to kopējo steka patēriņu. (2.7. koda fragments)

|  |
| --- |
| # Aprēķina kopējo steku rekursīvajām funkcijām  recursive\_stack = {}  for func in recursive\_functions:  local\_usage = function\_stack\_usage.get(func, 0)  depth = recursion\_limits.get(func, 10)  # Pirmais izsaukums  first\_call = local\_usage  # Sekundārie izsaukumi ar atgriešanās adresēm  subsequent\_calls = (depth - 1) \* (local\_usage + 2)  total = first\_call + subsequent\_calls  recursive\_stack[func] = total |

*2.7. koda fragments.* Rekursīvo funkciju kopējā steka aprēķināšana

Rekursijas dziļums tiek noteikts, analizējot pirmkodu un meklējot sākotnējās vērtības, ar kurām tiek izsauktas rekursīvās funkcijas. Rekursijas ierobežojumu dinamiska pielāgošanas pieeja ļauj modelēt rekursīvos izsaukumus izsaukumu grafikā, vienlaikus saglabājot iespēju ierobežot maksimālo rekursijas dziļumu, pamatojoties uz avota koda analīzi.

Steka dziļuma aprēķināšanas algoritms varētu būt skaitļošanas ziņā intensīvs, īpaši lielām programmām ar sarežģītiem izsaukumu grafikiem, kur daudzi ceļi var pārklāties. Lai uzlabotu algoritma efektivitāti, analizators izmanto memoizācijas tehniku – jau aprēķināto ceļu rezultātu saglabāšanu.

Memoizācija darbojas šādi:

1. Katru reizi, kad tiek aprēķināts steka patēriņš konkrētam funkcijas un izsaukumu ceļa pārim, rezultāts tiek saglabāts speciālā kešā.
2. Pirms jaunu aprēķinu veikšanas algoritms pārbauda, vai šāds ceļš jau ir aprēķināts.
3. Ja rezultāts jau ir zināms, tas tiek ātri atgriezts, izvairoties no atkārtotiem aprēķiniem.

Šāda pieeja būtiski uzlabo algoritma veiktspēju, īpaši programmām ar daudzām funkcijām un sarežģītu izsaukumu struktūru, kur atkārtoti ceļu aprēķini būtu ļoti neefektīvi.

Pēc maksimālā steka dziļuma aprēķināšanas analizators pievieno drošības rezervi – parasti 15% no aprēķinātā steka patēriņa. Šī drošības rezerve ir būtiska vairāku iemeslu dēļ:

* 1. **Kompensē analīzes neprecizitātes**: Statiskā analīze nevar vienmēr perfekti modelēt visas iespējamās programmas izpildes nianses, īpaši gadījumos ar sarežģītiem datu atkarīgiem ceļiem vai pārtraukumiem.
  2. **Nodrošinās pret neparedzētiem gadījumiem**: Programmas izpildes laikā var rasties situācijas, kas nav skaidri redzamas statiskajā analīzē, piemēram, dziļāka rekursija nekā sākotnēji paredzēts.
  3. **Kritiski svarīga iegultajās sistēmās**: AVR mikrokontrolieros steka pārplūde var izraisīt nopietnas sistēmas kļūdas, tādēļ papildu aizsardzība ir ļoti svarīga, īpaši kritiskās lietojumprogrammās.

Drošības rezerves piemērošana ir konservatīva pieeja, kas palīdz izvairīties no potenciālām steka pārplūdes problēmām reālās darba situācijās.

**SECINĀJUMI**

**PRIEKŠLIKUMI**

**IZMANTOTĀS LITERATŪRAS UN AVOTU SARAKSTS**

[1] <https://myrobot.ru/stepbystep/mc_architecture.php>

[2] <https://ww1.microchip.com/downloads/en/DeviceDoc/en590320.pdf>

[3] <https://www.cprogramming.com/tutorial/computersciencetheory/stack.html>

[4] <https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf>

[5] <https://www.bogotobogo.com/cplusplus/files/embed/OReilly_Programming_Embedded_Systems_Second_edition_ebook.pdf>

[6] <https://theswissbay.ch/pdf/Gentoomen%20Library/Misc/Art%20of%20Designing%20Embedded%20Systems~tqw~_darksiderg.pdf>

[7] <https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/memory/>

[8] <https://www.kanda.com/blog/microcontrollers/all-you-need-to-know-about-avr-isp-updi-jtag-pdi-and-tpi/>

[9] <https://www.arnabkumardas.com/arduino-tutorial/avr-architecture/>

[10] <https://www.embedded.com/optimizing-data-memory-utilization/>

[11 ] <https://www.arnabkumardas.com/arduino-tutorial/avr-memory-architecture/>

[12] <https://www.thecoderscorner.com/electronics/microcontrollers/efficiency/how-arduino-avr-memory-model-works/>

**PIELIKUMI**

**GALVOJUMS**

Ar šo es, *Anatolijs Koļesņevs,* galvoju, ka šis bakalaura darbs ir manis paša patstāvīgi izpildīts oriģināls darbs. Visi informācijas avoti, kā arī no tiem ņemtie dati un definējumi ir norādīti darbā. Šis darbs tādā vai citādā veidā nav iesniegts nevienai citai pārbaudījumu komisijai un nav nekur publicēts.

Esmu informēts (-a), ka mans bakalaura darbs tiks ievietots un apstrādāts Vienotajā datorizētajā plaģiāta kontroles sistēmā plaģiāta kontroles nolūkos.

202\_\_. gada \_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Es, *Anatolijs Koļesņevs*, atļauju Ventspils Augstskolai savu bakalaura darbu bez atlīdzības ievietot un uzglabāt Latvijas Nacionālās bibliotēkas pārvaldītā datortīklā Academia (www.academia.lndb.lv), kurā tie ir pieejami gan bibliotēkas lietotājiem, gan globālajā tīmeklī tādā veidā, ka ikviens tiem var piekļūt individuāli izraudzītā laikā, individuāli izraudzītā vietā.

Piekrītu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nepiekrītu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

202\_\_. gada \_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_