**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 Mg.sc.ing. Jānis Šmēdiņš

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

(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:** 52 lpp., 1 tabula, 7 attēli, 15 bibliogrāfiskie avoti, 1 pielikums.

**Atslēgas vārdi:** AVR MIKROKONTROLIERI, STEKATMIŅA, STATISKĀ ANALĪZE, REKURSĪVĀS FUNKCIJAS, IZSAUKUMU GRAFS.

Bakalaura darbā ir izstrādāts stekatmiņas izmantošanas analizators AVR mikrokontrolieriem, kas risina kritisko problēmu saistībā ar ierobežotajiem atmiņas resursiem iegultajās sistēmās. Darbā detalizēti aplūkota AVR arhitektūra, atmiņas organizācija un stekatmiņas darbības principi, kā arī analizētas steka pārplūdes problēmas un to sekas.

Darba praktiskajā daļā izstrādāts inovatīvs analizatora rīks, kas veic statisko C koda un asemblera analīzi bez programmas izpildes. Analizators spēj ar augstu precizitāti aprēķināt maksimālo steka patēriņu, atbalsta rekursīvās funkcijas, identificē sarežģītus izsaukumu ceļus un darbojas ar dažādiem GCC optimizācijas līmeņiem. Īpaša uzmanība pievērsta rekursīvo funkciju analīzei un izsaukumu grafa rekonstrukcijai no asemblera koda.

Izstrādātais risinājums nodrošina detalizētas atskaites par katras funkcijas steka izmantojumu, vizualizē kritiskos izpildes ceļus un pievieno 10% drošības rezervi aprēķiniem. Analizators ir pielāgojams dažādiem AVR mikrokontrolieru modeļiem un piedāvā elastīgu konfigurāciju.

Darba mērķauditorija ir iegulto sistēmu izstrādātāji, AVR programmētāji, kā arī studenti un pētnieki, kas strādā ar mikrokontrolieru programmēšanu un atmiņas optimizāciju. Izstrādātais rīks ir publicēts GitHub repozitorijā un pieejams atvērtā pirmkoda formātā.

**ABSTRACT**

**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:** 52 pages, 1 table, 7 figures, 15 bibliographic references, 1 appendix.

**Keywords:** AVR MICROCONTROLLERS, STACK MEMORY, STATIC ANALYSIS, RECURSIVE FUNCTIONS, CALL GRAPH.

The bachelor's thesis presents the development of a stack memory usage analyzer for AVR microcontrollers, addressing the critical issue of limited memory resources in embedded systems. The work provides a comprehensive examination of AVR architecture, memory organization, and stack operation principles, along with an analysis of stack overflow problems and their consequences.

In the practical part, an innovative analyzer tool has been developed that performs static analysis of C code and assembly without program execution. The analyzer can calculate maximum stack consumption with high accuracy, supports recursive functions, identifies complex call paths, and works with various GCC optimization levels. Special attention is given to recursive function analysis and call graph reconstruction from assembly code.

The developed solution provides detailed reports on each function's stack usage, visualizes critical execution paths, and adds a 10% safety margin to calculations. The analyzer is adaptable to different AVR microcontroller models and offers flexible configuration options.

The target audience includes embedded systems developers, AVR programmers, as well as students and researchers working with microcontroller programming and memory optimization. The developed tool is published on GitHub repository and available as open-source software.

**SATURS**

[Ievads 5](#_Toc199122521)

[Saīsinājumu un nosacīto apzīmējumu saraksts 7](#_Toc199122522)

[1. Teorētiskais pamatojums 10](#_Toc199122523)

[1.1. AVR mikrokontrolieru arhitektūras apskats 10](#_Toc199122524)

[1.1.1. Atmiņas organizācija AVR mikrokontrolieros 11](#_Toc199122525)

[1.1.2. RAM, Flash un EEPROM raksturojums 12](#_Toc199122526)

[1.2. Stekatmiņas koncepcija un tās nozīme 13](#_Toc199122527)

[1.3. AVR vispārīgie reģistri un operandu apzīmējumi (Rd, Rr) 17](#_Toc199122528)

[1.4. Stekatmiņas pārplūdes problēmas un to sekas 19](#_Toc199122529)

[2. AVR stekatmiņas izmantošanas analizatora izstrāde 22](#_Toc199122530)

[2.1. Izstrādes prasības un mērķi 22](#_Toc199122531)

[2.2. Izmantotās tehnoloģijas un rīki 23](#_Toc199122532)

[2.3. Analizatora arhitektūras projektējums 24](#_Toc199122533)

[2.4. Steka izmantojuma analīzes metodes un algoritmi 27](#_Toc199122534)

[2.4.1. Statiskie steka izmantošanas aprēķini 27](#_Toc199122535)

[2.4.2. Rekursīvo funkciju analīze 33](#_Toc199122536)

[2.4.3. Izsaukumu grafa rekonstrukcija no asemblera 38](#_Toc199122537)

[2.4.4. Maksimālā steka dziļuma aprēķināšana 41](#_Toc199122538)

[2.5. Analizatora testēšana 44](#_Toc199122539)

[Secinājumi un priekšlikumi 47](#_Toc199122540)

[Izmantotās literatūras un avotu saraksts 49](#_Toc199122541)

[Pielikumi 51](#_Toc199122542)

[Galvojums 52](#_Toc199122543)

# IEVADS

Mūsdienu iegultajās sistēmās mikrokontrolieru izmantošana ir kļuvusi par neatņemamu sastāvdaļu dažādās jomās – no sadzīves tehnikas līdz kritiskām medicīnas un industriālajām iekārtām. AVR mikrokontrolieru popularitāte ir saistīta ar to vienkāršību, zemo enerģijas patēriņu un plašo pielietojumu spektru. Tomēr šo ierīču ierobežotie atmiņas resursi rada būtiskus izaicinājumus programmētājiem, īpaši stekatmiņas pārvaldības kontekstā. Stekatmiņas pārplūde var izraisīt neparedzamu sistēmas uzvedību vai pilnīgu atteici, kas kritiskās lietojumprogrammās var radīt nopietnas sekas. Neskatoties uz šo problēmu aktualitāti, esošie rīki bieži nesniedz pietiekami precīzu un detalizētu informāciju par steka izmantojumu, īpaši sarežģītās programmās ar rekursīvām funkcijām un netiešajiem izsaukumiem.

Darba mērķis ir izstrādāt efektīvu un precīzu stekatmiņas izmantošanas analizatoru AVR mikrokontrolieriem, kas spēj veikt statisko koda analīzi un sniegt detalizētu informāciju par maksimālo steka patēriņu dažādos programmas izpildes scenārijos.

Darba mērķa sasniegšanai izvirzīti šādi uzdevumi:

1. izpētīt AVR mikrokontrolieru arhitektūru, atmiņas organizāciju un stekatmiņas darbības principus;
2. analizēt esošos risinājumus steka izmantojuma analīzei un identificēt to trūkumus;
3. izstrādāt algoritmus statiskai steka izmantojuma analīzei, iekļaujot atbalstu rekursīvām funkcijām;
4. implementēt izsaukumu grafa rekonstrukcijas algoritmu no asemblera koda;
5. izveidot pilnvērtīgu analizatora rīku ar elastīgu konfigurāciju dažādiem AVR modeļiem;
6. veikt izstrādātā risinājuma testēšanu un validāciju ar reāliem programmu piemēriem.

Esošo risinājumu pārskats rāda, ka pieejamie rīki galvenokārt balstās uz vienkāršotu analīzi, kas neņem vērā sarežģītus programmas izpildes ceļus, rekursiju un kompilatora optimizācijas. GCC kompilators piedāvā -fstack-usage opciju, bet tā sniedz tikai lokālo funkciju steka informāciju bez konteksta par izsaukumu hierarhiju. Citi komerciāli risinājumi bieži ir specifiski konkrētām izstrādes vidēm un nenodrošina pietiekamu elastību vai precizitāti.

Darba struktūra ietver divas galvenās daļas. Pirmajā nodaļā tiek aplūkots teorētiskais pamatojums, iekļaujot AVR arhitektūras apskatu, atmiņas organizāciju, stekatmiņas koncepciju un pārplūdes problēmas analīzi. Otrajā nodaļā detalizēti aprakstīta analizatora izstrāde, sākot no prasību definēšanas līdz konkrētu algoritmu implementācijai – statiskai analīzei, rekursijas atpazīšanai, izsaukumu grafa rekonstrukcijai un maksimālā steka dziļuma aprēķināšanai.

Pētījuma realizācijas metodes ietver literatūras analīzi AVR arhitektūras izpratnei, algoritmu izstrādi statiskai koda analīzei, programmatūras inženierijas principu pielietošanu modulāras arhitektūras izveidei, kā arī eksperimentālu validāciju ar testa programmām. Analizators tiek implementēts Python programmēšanas valodā, izmantojot AVR-GCC rīku komplektu asemblera koda ģenerēšanai un analīzei.

# SAĪSINĀJUMU UN NOSACĪTO APZĪMĒJUMU SARAKSTS

**Aparatūras un arhitektūras apzīmējumi:**

ALU – aritmētiski loģiskā ierīce (Arithmetic Logic Unit);

AVR – Atmel RISC procesora arhitektūra 8 bitu mikrokontrolieriem;

CPU – centrālais procesors (Central Processing Unit);

MCU – mikrokontrolieris (Microcontroller Unit);

PC – programmskaitītājs (Program Counter);

RISC – samazinātas instrukciju kopas dators (Reduced Instruction Set Computer);

SP – steka rādītājs (Stack Pointer);

SPH – steka rādītāja augstākais baits (Stack Pointer High);

SPL – steka rādītāja zemākais baits (Stack Pointer Low).

**Atmiņas apzīmējumi:**

EEPROM – elektriski dzēšama un pārprogrammējama pastāvīgā atmiņa (Electrically Erasable Programmable Read-Only Memory);

Flash – pārprogrammējamā pastāvīgā atmiņa;

Kaudze (Heap) – dinamiski piešķirta atmiņas apgabals, kurā programma var pieprasīt un atbrīvot atmiņu izpildes laikā;

Programmatmiņa – atmiņas apgabals, kurā tiek glabāts programmas kods (instrukcijas), kas tiek izpildīts;

RAM – brīvpiekļuves atmiņa (Random Access Memory);

ROM – tikai lasāmā atmiņa (Read Only Memory);

SRAM – statiskā brīvpiekļuves atmiņa (Static RAM);

Steks (Stack) – LIFO (Last In, First Out) atmiņas struktūra, kas tiek izmantota funkciju izsaukumu adrešu un lokālo mainīgo glabāšana.

**Programmēšanas saskarnes:**

GND – zemējums (Ground), nulles potenciāla atskaites punkts;

ISP – sistēmā programmēšanas saskarne (In-System Programming);

JTAG – kopējās testēšanas darbības grupa (programmēšanas standarts) (Joint Test Action Group);

MISO – galvenais iekšā, pakārtotais ārā (Master In Slave Out), datu līnija no pakārtotās ierīces uz galveno;

MOSI – galvenais ārā, pakārtotais iekšā (Master Out Slave In), datu līnija no galvenās ierīces uz pakārtoto;

PDI – programmēšanas un atkļūdošanas saskarne (Programming and Debugging Interface);

RST – atiestatīšanas signāls (Reset), signāls mikrokontrollera restartēšanai;

SCK – seriālais takts signāls (Serial Clock), sinhronizācijas signāls SPI saziņai;

SPI – seriālā perifērijas saskarne (Serial Peripheral Interface);

TPI – maza programmēšanas saskarne (Tiny Programming Interface);

UPDI – vienotā programmēšanas un atkļūdošanas saskarne (Unified Program and Debug Interface);

VCC – barošanas spriegums (Voltage Common Collector), pozitīvā barošanas līnija.

**Programmatūras un kompilācijas apzīmējumi:**

.bss – globālie un statiskie mainīgie, kas tieši vai netieši inicializēti ar nulli;

.data – globālie un statiskie mainīgie, kas tieši vai netieši inicializēti ar nulli;

ELF – izpildāmais un saistāmais formāts (Executable and Linkable Format);

GCC – GNU kompilatoru kolekcija;

.su – steka izmantojuma failu paplašinājums.

**Reģistru un instrukciju apzīmējumi:**

GPR – vispārējās nozīmes reģistri (General Purpose Registers);

I/O – ievades/izvades reģistri;

Rd – mērķa reģistrs (Destination Register);

Rr – avota reģistrs (Source Register).

**Instrukciju apzīmējumi** [1]**:**

ADIW – pievienot tūlītēju vērtību vārdam (Add Immediate to Word);

CALL – funkcijas izsaukums (Call Subroutine);

ICALL – netiešais izsaukums (Indirect Call);

Instrukcija – asemblera komanda;

LD – ielādēt no atmiņas, izmantojot netiešo adresāciju (Load Indirect);

LDD – ielādēt no atmiņas ar nobīdi (Load with Displacement);

LDI – ielādēt tūlītēju vērtību (Load Immediate);

LDS – ielādēt tieši no datu atmiņas (Load Direct from SRAM);

POP – izņemt no steka;

PUSH – ievietot stekā;

RCALL – relatīvais funkcijas izsaukums (Relative Call);

RET – atgriezties no funkcijas (Return from Subroutine);

SBIW – atņemt tūlītēju vērtību no vārda (Subtract Immediate from Word);

ST – saglabāt atmiņā, izmantojot netiešo adresāciju (Store Indirect);

STD – saglabāt atmiņā ar nobīdi (Store with Displacement);

STS – saglabāt tieši datu atmiņā (Store Direct to SRAM).

**Citi apzīmējumi:**

ADC – analogciparu pārveidotājs (Analog-to-Digital Converter);

DFS – meklēšana dziļumā (Depth-First Search);

GPIO – vispārējās nozīmes ievades/izvades (General Purpose I/O);

LIFO – pēdējais iekšā, pirmais ārā (Last In First Out);

MIPS – miljoni instrukciju sekundē;

PWM – impulsa platuma modulācija (Pulse Width Modulation) - tehnika, kas ļauj kontrolēt analoga signāla vidējo vērtību, mainot digitāla impulsa ilgumu attiecībā pret periodu.

**Mērvienības:**

KB – kilobaiti (1024 baiti);

MB – megabaiti (1024 KB);

Mbps – megabiti sekundē;

MHz – megaherci.

# TEORĒTISKAIS PAMATOJUMS

Šajā nodaļā tiek aplūkoti teorētiskie aspekti, kas nepieciešami AVR mikrokontrolieru stekatmiņas izmantošanas analizatora izstrādei. Nodaļā tiek sniegts pārskats par AVR mikrokontrolieru arhitektūru, atmiņas organizāciju, stekatmiņas koncepciju un tās nozīmi iegulto sistēmu programmēšanā. Tiek analizētas stekatmiņas pārplūdes problēmas un to sekas, kas ir fundamentāli svarīgi, lai izprastu analizatora nozīmi un tā darbības principus. Teorētiskais pamatojums veido nepieciešamo zināšanu bāzi turpmākajai praktiskās daļas izstrādei un analīzei.

## AVR mikrokontrolieru arhitektūras apskats

Katra AVR mikrokontroliera "sirds" ir 8 bitu centrālais procesors, kas balstīts uz RISC (Reduced Instruction Set Computer) principiem. RISC pieeja izmanto vienkāršākas, fiksēta garuma instrukcijas, kas tipiski izpildās vienā takts ciklā, nodrošinot efektīvu darbību un optimizētu resursu izmantošanu AVR mikrokontrolieros. [2]

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. [2]

AVR mikrokontrolieri izmanto Hārvardas arhitektūru, kas nozīmē, ka programmatmiņa (Flash) ir fiziski nošķirta no datu (SRAM) atmiņas, ar atsevišķām adresācijas telpām un datu maģistrālēm. Šī arhitektūra ļauj iegūt instrukciju un veikt datu piekļuvi vienlaicīgi, palielinot kopējo sistēmas veiktspēju. [3]

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. [4]

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). [4]

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. [5]

Lai atvieglotu darbu ar lielākām datu vienībām, daži reģistru pāri var tikt apvienoti 16-bitu operācijām. Piemēram, X reģistrs (R26:R27), Y reģistrs (R28:R29) un Z reģistrs (R30:R31) ir speciālie 16-bitu reģistru pāri, kas kalpo kā rādītāji netiešai adresācijai, atvieglojot darbu ar datu struktūrām. [6]

### 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. [6]

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 programmatmiņa tiek realizēta ar Flash tipa ROM. [4][7]

**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. [8][9] Flash tehnoloģijas galvenā priekšrocība ir elektriskās pārrakstīšanas iespēja, kas ļauj vairākkārtīgi ierakstīt un dzēst informāciju. [2][7]

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. [10][6]

Papildus UPDI, AVR mikrokontrolieriem var būt arī citas programmēšanas saskarnes, atkarībā no modeļa un sērijas: JTAG (parasti 40 kontakttapu 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. [10][11]

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. [10]

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. [2]

Saskaņā ar Atmel korporācijas tehnisko dokumentāciju, AVR otrās paaudzes mikrokontrolieru Flash atmiņai tiek garantēts vismaz 10 000 pārrakstīšanas ciklu. [11]

### 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. [4]

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. [6]

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. [12]

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 saskarni), gan arī darbības laikā no paša mikrokontroliera koda. EEPROM tipiski garantē vismaz 100 000 pārrakstīšanas ciklu, saglabājot datus arī pēc barošanas atslēgšanas. [4]

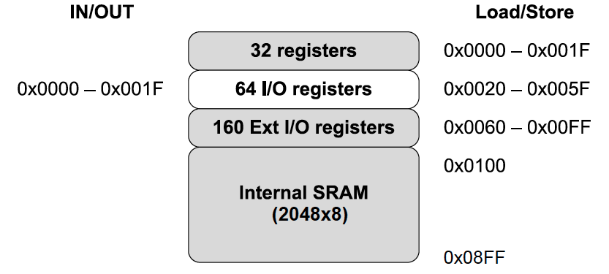
**Operatīvā atmiņa** (RAM – Random Access Memory) AVR mikrokontrolieros tiek realizēta kā statiskā RAM (SRAM), kas organizēta baitos. Tā tiek izmantota mainīgo, kaudzes (heap) un citu īslaicīgu datu glabāšanai izpildes laikā. Atmiņas apjoms dažādos mikrokontrolieru modeļos var būt no 64 baitiem līdz 16 KB. [7]

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. [2]

## 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. [12]

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. [4]



*1.1. att.* ATmega328P mikrokontroliera SRAM atmiņa [4]

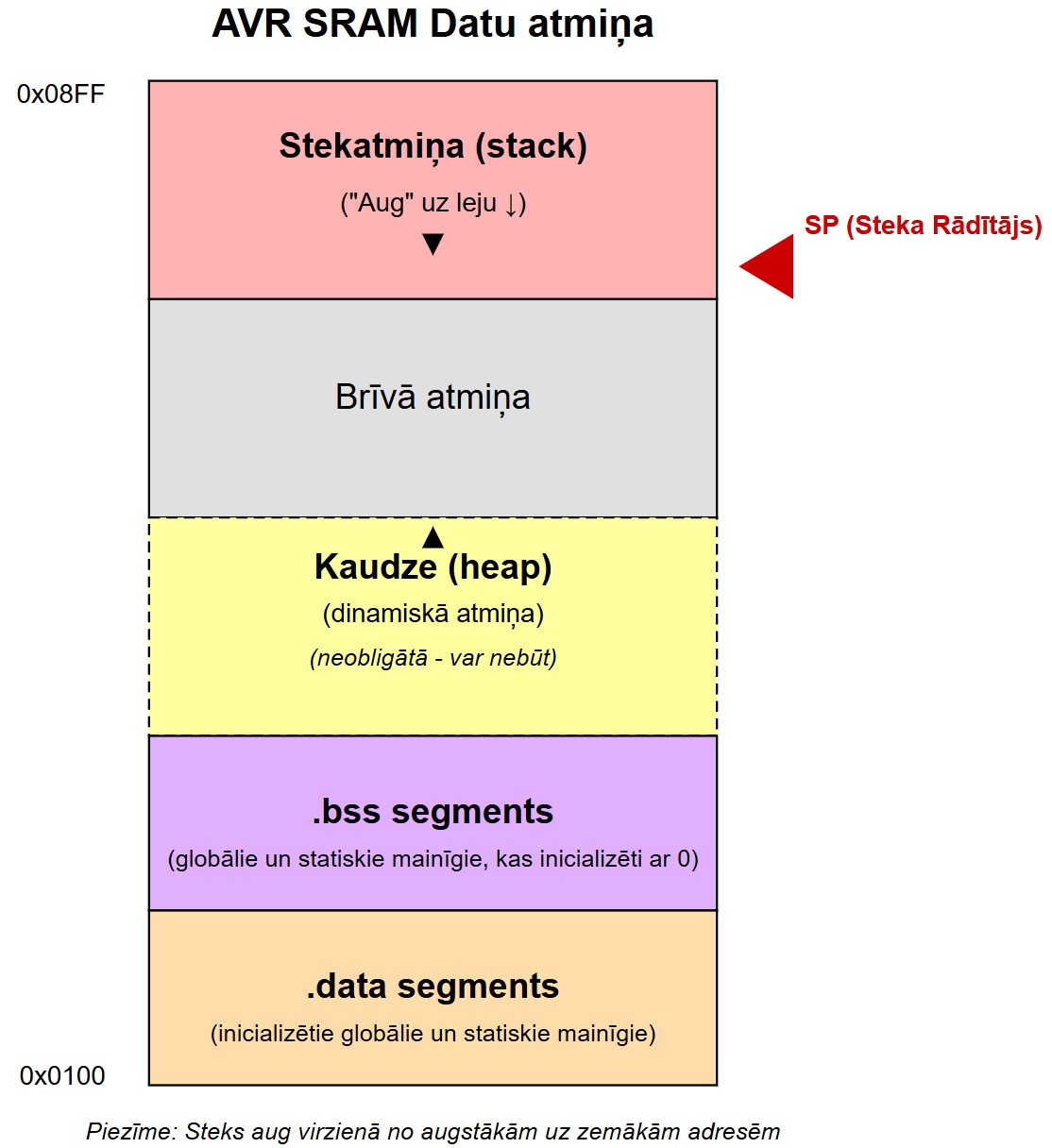
1.1. attēls attēlo ATmega328P mikrokontroliera atmiņas karti un adresēšanas struktūru. Tas parāda, kā AVR arhitektūrā ir organizēta adrešu telpa, iekļaujot reģistrus un SRAM. Attēls parāda četras galvenās komponentes, kas katra aizņem noteiktu adrešu diapazonu atkarībā no modeļa.

**32 vispārējās nozīmes reģistri** (0x0000–0x001F) ir darba reģistri (R0–R31), kas veido reģistru failu. Tie ir ātrākā pieejamā atmiņa mikrokontrolierim un tiek izmantoti tiešām datu apstrādes operācijām. [7]

**64 I/O reģistri** (0x0020–0x005F) 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. [7]

**160 paplašinātie I/O reģistri** (0x0060–0x00FF) 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. [7]

**Iekšējā SRAM** (0x0100–0x08FF) 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. [7]



*1.2. att.* ATmega328P mikrokontroliera SRAM datu atmiņa [7][3]

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 globālie un statiskie mainīgie, kas tieši vai netieši inicializēti ar nulli. Šie mainīgie aizņem vietu tikai SRAM atmiņā, jo tiem nav nepieciešams saglabāt sākotnējās vērtības Flash atmiņā;
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 .data/.bss segmentiem un steku parasti ir brīva atmiņa, kas var ietvert arī kaudzi, ja programmā tiek izmantota dinamiskā atmiņas izdalīšana. Ja steks izaug pārāk liels, tas var pārklāties ar kaudzi vai tiešā veidā ar .data/.bss segmentiem, ja kaudze netiek izmantota, abos gadījumos izraisot "stack overflow" kļūdu, kas noved pie nepareizas programmas izpildes.

**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). [1]

**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. [1]

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. [1]

**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. [1]

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. [13, 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). [1]

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.

## 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. [6]

**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). [1]

**Reģistru apzīmējumi instrukcijās: Rd un Rr**

Instrukciju sintaksē bieži tiek izmantoti saīsinājumi:

* Rd (Destination Register – mērķa reģistrs) ir reģistrs, kurā tiks ievietots rezultāts;
* Rr (Source Register – avota reģistrs) ir reģistrs, no kura tiek ņemta ievades vērtība (piemēram, saskaitāmais). [1]

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 – kas īstenībā notiek.

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. [1]

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.

## 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. [14]

**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. [14]

**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. [14]

**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. [7]

**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. [11]

**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. [15, 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.

# AVR STEKATMIŅAS IZMANTOŠANAS ANALIZATORA IZSTRĀDE

Šajā nodaļā tiek aprakstīta AVR mikrokontrolieru stekatmiņas analizatora izstrāde, sākot no sākotnējām prasībām un mērķiem līdz detalizētai algoritmu realizācijai. Nodaļā tiek apskatītas izmantotās tehnoloģijas un rīki, analizatora arhitektūras projektējums, kā arī detalizēti izklāstītas dažādas analīzes metodes un algoritmi, kas nodrošina precīzu steka izmantojuma novērtējumu dažādos mikrokontrolieru programmēšanas scenārijos.

## Izstrādes prasības un mērķi

Izstrādājot AVR stekatmiņas analizatoru, tika definēti precīzi mērķi un prasības, kas atspoguļo mikrokontrolieru programmēšanas specifisko vidi un tās ierobežojumus. Šie mērķi veidoja pamatu tehnoloģiju izvēlei un arhitektūras projektējumam:

1. **Kvalitatīva statiskā analīze** ir analizatora galvenais mērķis. Analizatoram jāspēj ar augstu precizitāti aprēķināt maksimālo steka izmantojumu, analizējot C valodas pirmkodu bez programmas izpildes.
2. **Rekursīvo funkciju atbalsts** ir īpaši sarežģīts izaicinājums, jo rekursijas dziļums var būt mainīgs un atkarīgs no ievades datiem. Analizatoram jāspēj ne tikai identificēt rekursīvās funkcijas, bet arī precīzi aprēķināt to maksimālo iespējamo steka patēriņu, analizējot rekursijas samazināšanas modeļus pirmkodā.
3. **Optimizācijas līmeņu atbalsts** atspoguļo reālu izstrādes procesu, kur kompilācijas optimizācijas var būtiski ietekmēt gan steka izmantojumu, gan funkciju izsaukumu struktūru. Analizatoram jādemonstrē stabilitāte visos GCC optimizācijas līmeņos.
4. **Detalizēta steka analīze** nozīmē, ka analizators nodrošina detalizētu informāciju par katras funkcijas individuālo steka patēriņu, veido izsaukumu grafu un identificē maksimālā steka izmantojuma ceļus. Šī informācija ir neaizmirstama efektīvai steka problēmu diagnosticēšanai un optimizācijai.
5. **Drošības rezerves īstenojums** atspoguļo konservatīvu inženierijas pieeju. Analizators pieliek 10% drošības rezervi aprēķinātajam steka izmantojumam, lai kompensētu neparedzētas situācijas un analīzes potenciālās neprecizitātes.
6. **Dažādu MCU modeļu atbalsts** nodrošina analizatora pielāgojamību. Analizatoram jādarbojas ar dažādiem AVR mikrokontrolieru modeļiem, ņemot vērā to atšķirīgos atmiņas izmērus un organizāciju.
7. **Elastīga konfigurācija** ir nepieciešama, lai analizators varētu pielāgoties dažādiem AVR mikrokontrolieru modeļiem un projektiem ar atšķirīgām prasībām. Analizatoram jāpiedāvā iespēja:

* norādīt mikrokontroliera tipu (-m vai --mcu);
* norādīt pielāgotu RAM izmēru (-r vai --ram);
* norādīt GCC optimizācijas līmeni (-o vai --optimization);
* pievienot specifiskus kompilatora karogus (-c vai --compiler-flags);
* izvēlēties žurnalēšanas detalizācijas līmeni (-l vai --log-level).

1. **Pilna ceļu analīze** ir nepieciešama, lai iegūtu pārskatu par visiem funkciju izsaukumu scenārijiem un to steka patēriņu. Analizatoram jāapstrādā visi iespējamie funkciju izsaukumu ceļi, jāatrod tas ar lielāko steka patēriņu un jārāda pilnā izsaukumu ķēde.

## Izmantotās tehnoloģijas un rīki

Autors izstrādājot steka analizatoru kā galveno programmēšanas valodu izvēlējās 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;
* iebūvēto žurnalēšanas sistēmu programmas darbības pilnvērtīgai uzraudzībai;
* pārnesamību starp dažādām operētājsistēmām.

Analizatora pilnvērtīgai darbībai tiek izmantoti šādi Python moduļi:

* **subprocess** – ārējo komandu (avr-gcc, avr-objdump, avr-size) izpildei;
* **re** – regulāro izteiksmju apstrādei koda analīzē;
* **os** un **shutil** – failu un direktoriju operācijām;
* **argparse** – komandrindas argumentu apstrādei;
* **tempfile** – pagaidu darba direktoriju pārvaldībai;
* **logging** – detalizētai programmas darbības žurnalēšanai.

Lai veiktu nepieciešamās kompilācijas un analīzes darbības, tiek izmantoti AVR GCC kompilatora komplekta rīki:

* **avr-gcc** – C koda kompilēšanai uz AVR asemblera kodu, ar īpašu -fstack-usage karogu steka izmantojuma informācijas iegūšanai,
* **avr-objdump** – ELF failu disasemblēšanai, lai iegūtu asemblera kodu analīzei,
* **avr-size** – programmas atmiņas sekciju izmēru noteikšanai.

Analizators atbalsta dažādus GCC optimizācijas līmeņus (O0, O1, O2, O3, Os, Og, Ofast, Oz), kas ļauj pielāgot analīzi reālai izstrādes videi, kur optimizācijas ir neatņemama koda kompilācijas daļa.

Statiskās analīzes procesā tiek izmantotas regulārās izteiksmes asemblera koda parsēšanai. Analizators apstrādā GCC kompilatora ģenerētos steka izmantojuma (.su) failus un veic pats savu asemblera koda analīzi, lai pilnīgāk rekonstruētu funkciju izsaukumu grafu un aprēķinātu steka izmantojumu.

Grafu algoritmi tiek izmantoti maksimālā steka ceļa identificēšanai. Analizators izmanto algoritmu meklēšana dziļumā (DFS – Depth First Search) ar memoizāciju, lai efektīvi pārlūkotu visus iespējamos funkciju izsaukumu ceļus un atrastu to, kas rada lielāko steka patēriņu.

Analizators ietver arī funkcijas rekursīvo funkciju atklāšanai un to dziļuma noteikšanai, analizējot gan disasemblēto kodu, gan sākotnējo C kodu. Tas spēj identificēt dažādus rekursijas samazināšanas modeļus (atskaitīšana, dalīšana, bitu nobīde) un precīzi aprēķināt to steka ietekmi.

Drošības nolūkos analizators pievieno 10% rezervi aprēķinātajam steka izmantojumam, lai kompensētu neparedzētas situācijas vai analīzes neprecizitātes.

## Analizatora arhitektūras projektējums

AVR steka analizatora arhitektūra ir projektēta kā modulāra sistēma, kas sastāv no vairākām savstarpēji saistītām komponentēm. Sistēmas pamatā ir AVRCStackAnalyzer klase, kas darbojas kā centrālais koordinators visām analīzes operācijām.

Analizatora inicializācija sākas ar būtisku parametru validāciju un sistēmas gatavības pārbaudi. Konstruktors \_\_init\_\_ uzstāda nepieciešamās konfigurācijas vērtības, tostarp mikrokontroliera tipu, RAM izmēru, optimizācijas līmeni, žurnalēšanas detalizācijas līmeni un papildu kompilatora karogus. Šajā posmā tiek izveidota pagaidu direktorija kompilācijas artefaktu glabāšanai, kas nodrošina tīru darba vidi katrai analīzes sesijai. Destruktors \_\_del\_\_ nodrošina pagaidu failu automātisku tīrīšanu, kas novērš sistēmas resursu noplūdi un uztur failu sistēmas tīrību pat negaidītu kļūdu gadījumā.

Kompilācijas modulis compile\_c\_code ir atbildīgs par C pirmkoda pārvēršanu AVR mašīnkodā. Šī komponente veido avr-gcc komandu ar definētiem karogiem. Īpaša uzmanība pievērsta optimizācijas līmeņa konfigurācijai un funkciju ievietošanas (inlining) atspējošanai, kas ir kritiska precīzai izsaukumu grafa analīzei.

Kompilācijas procesā tiek aktivizēts -fstack-usage karogs, kas liek GCC ģenerēt .su failus ar detalizētu informāciju par katras funkcijas lokālo steka izmantojumu. Šī informācija vēlāk tiek izmantota kā pamats kopējā steka aprēķinam. Funkcija collect\_stack\_usage\_reports īsteno algoritmu GCC ģenerēto .su failu atrašanai un parsēšanai.

Disasemblēšanas komponente disassemble\_avr transformē kompilēto ELF failu lasāmā asemblera kodā, izmantojot avr-objdump utilītu. Iegūtais asemblera kods kalpo kā pamats dziļākai steka operāciju analīzei un izsaukumu attiecību noteikšanai.

Asemblera koda analīze tiek veikta vairākos līmeņos. Pirmkārt, tiek identificētas funkciju robežas un to savstarpējās izsaukumu attiecības. Otrkārt, tiek analizētas steka manipulācijas instrukcijas, lai precīzi aprēķinātu katras funkcijas faktisko steka patēriņu.

Rekursijas noteikšana ir viena no sarežģītākajām arhitektūras daļām. Funkcija detect\_recursion\_from\_assembly īsteno daudzpakāpju algoritmu. Tas kombinē funkciju adrešu kartēšanu no asemblera koda un izsaukumu instrukciju analīzi, kas atpazīst gan tiešos izsaukumus (call, rcall), gan netiešos izsaukumus (icall, eicall).

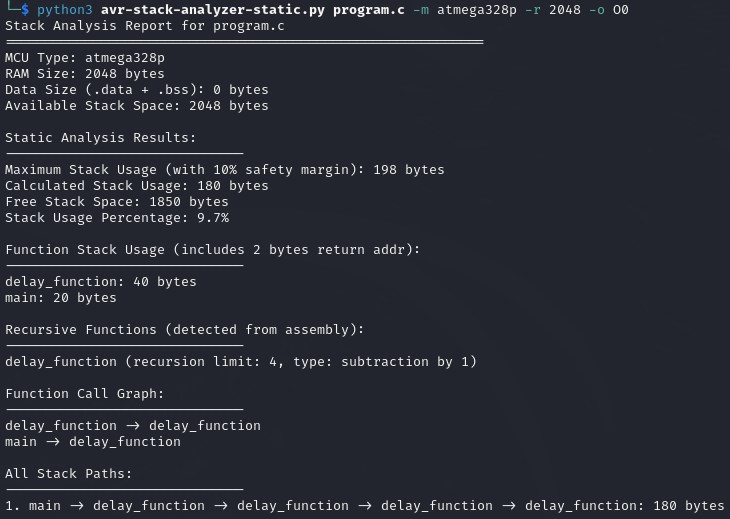
Funkcija analyze\_recursion\_depth īsteno algoritmu rekursijas dziļuma noteikšanai, analizējot pirmkoda struktūru. Algoritms meklē iniciāļos parametru piešķīrumus un izseko to transformācijas caur funkciju izsaukumu ķēdēm.

Izsaukumu grafa konstrukcija build\_call\_graph veido sistēmas datu struktūru, kas reprezentē funkciju savstarpējās izsaukumu attiecības. Šī komponente apstrādā gan vienkāršos tiešos izsaukumus, gan sarežģītos netiešos izsaukumus caur funkciju rādītājiem.

Centrālā aprēķinu komponente calculate\_max\_stack\_usage īsteno rekursīvu algoritmu, kas iziet caur visiem iespējamajiem funkciju izsaukumu ceļiem un nosaka maksimālo steka patēriņu. Algoritms izmanto memoizācijas tehniku, lai izvairītos no atkārtotas aprēķināšanas un uzlabotu veiktspēju. Rekursīvo funkciju apstrāde tiek veikta īpaši, izmantojot iepriekš aprēķinātus rekursijas dziļumus.

Funkcija analyze\_function\_stack\_usage\_from\_asm veic detalizētu AVR asemblera instrukciju analīzi, lai precīzi aprēķinātu katras funkcijas steka izmantojumu.

Pārskatu ģenerēšanas sistēma generate\_report apkopo visus analīzes rezultātus strukturētā, lasāmā formātā. Komponente aprēķina pieejamo steka vietu, ņemot vērā datu sekciju izmērus, un sniedz detalizētu informāciju par katras funkcijas steka patēriņu. Pārskats ietver drošības rezervi (10% no aprēķinātā maksimuma), kas kompensē iespējamās analīzes neprecizitātes un neparedzētus steka izmantojuma scenārijus (sk. 2.1. attēlu).



*2.1. att.* AVR steka analizatora ģenerētais pārskats program.c failam (sk. 1. pielikumu)

**Arhitektūras projektēšanas principi:**

* Modularitātes princips tiek īstenots, sadalot analīzes procesu loģiskās komponentēs, katru ar skaidri definētu atbildību. Šāda pieeja atvieglo sistēmas testēšanu, atkļūdošanu un turpmāku attīstību.
* Kļūdu apstrādes stratēģija paredz detalizētu validāciju katram analīzes posmam ar informatīviem kļūdu ziņojumiem.
* Žurnalēšanas sistēma nodrošina detalizētu informāciju par analīzes procesu dažādos detalizācijas līmeņos, sākot no pamata brīdinājumiem līdz dziļai atkļūdošanas informācijai.
* Resursu pārvaldība tiek automatizēta, izmantojot Python konteksta pārvaldniekus un destruktorus, kas nodrošina, ka sistēmas resursi tiek pareizi atbrīvoti arī izņēmumu gadījumos.

Pilns analizatora implementācijas pirmkods ir pieejams 1. pielikumā.

## Steka izmantojuma analīzes metodes un algoritmi

Šajā nodaļā tiek detalizēti aprakstītas metodes un algoritmi, kas izmantoti AVR mikrokontrolieru steka izmantojuma analizatorā. Analizatora pamatā ir vairāki savstarpēji saistīti algoritmi, kas kopā veido visaptverošu analīzes sistēmu. Nodaļa apskata četrus galvenos analīzes blokus: statiskos steka izmantošanas aprēķinus, rekursīvo funkciju analīzi, izsaukumu grafa rekonstrukciju no asemblera koda, un maksimālā steka dziļuma aprēķināšanu. Katram no šiem algoritmiem ir būtiska loma kopējā steka izmantojuma precīzā prognozēšanā, īpaši ņemot vērā AVR mikrokontrolieru ierobežotos atmiņas resursus un funkciju izsaukumu specifiku šajā arhitektūrā.

### Statiskie steka izmantošanas aprēķini

Statiskā analīze ir pamata metode, kas ļauj noteikt programmas steka izmantojumu, nekad neizpildot pašu programmu. Analizators izmanto kombinētu pieeju, kas apvieno GCC kompilatora ziņojumus ar detalizētu asemblera koda analīzi, lai iegūtu maksimāli precīzus un uzticamus rezultātus.

Lai patiesi izprastu statiskās analīzes vērtību, jāsaprot tās fundamentālā priekšrocība. Atšķirībā no dinamiskās analīzes, kas pēta programmas uzvedību izpildes laikā, statiskā analīze spēj identificēt visus iespējamos izpildes ceļus un to steka patēriņu bez nepieciešamības palaist programmu ar visiem iespējamiem ievades datiem. Tas ir īpaši svarīgi iegultajās sistēmās, kur ir nepieciešama garantija, ka programma nekad nepārsniegs pieejamo atmiņas apjomu.

Statiskās analīzes procesu var salīdzināt ar detektīva darbu, kurā jārekonstruē notikumi, pamatojoties uz pēdām un pierādījumiem. Šajā gadījumā "pierādījumi" ir kompilētais kods un asemblera instrukcijas, kas precīzi atspoguļo, kā programma manipulēs ar steku izpildes laikā.

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, Ofast, Oz**: 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 asemblera 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 (sk. 2.2. attēlu).



*2.2. att.* .su faila saturs priekš programmas program.c (sk. 1. pielikumu)

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 otrais 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.

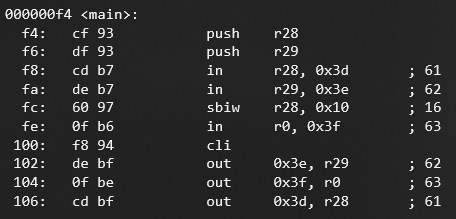
Papildus GCC pārskatiem, rīks disasemblē programmu, lai analizētu steka operācijas tiešā asemblera līmenī. Šī metode identificē specifiskas steka operācijas AVR asemblera instrukciju kopā, ieskaitot push, pop, call, rcall, icall, eicall un steka rādītāju pielāgošanu ar sbiw/adiw instrukcijām. Tā palīdz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

Pēc GCC ziņojumu apstrādes analizators veic asemblera koda disasemblēšanu, 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 asemblera kodu, kas satur visas instrukcijas, ko izpildīs mikrokontrolieris.

Pirmais svarīgākais solis ir sadalīt lielo asemblera koda masīvu atsevišķās funkcijās. AVR asemblerā katras funkcijas sākums ir skaidri iezīmēts ar struktūru, piemēram, kā 000000f4 <main>:, kur skaitlis ir funkcijas adrese heksadecimālā formātā (sk. 2.3. attēlu).



*2.3. att.* Disamblēta koda fragments no program.c (sk. 1. pielikumu)

Tomēr analizatoram jāspēj atšķirt īstas funkcijas no kompilatora ģenerētām tehniskām atzīmēm, kas var izskatīties līdzīgi.

Analizators izmanto divkāršu filtrēšanas procesu. Vispirms tas meklē visas atzīmes, kas atbilst funkciju šablonam, bet tad to filtrē, ignorējot atzīmes, kas sākas ar punktu vai satur specifiskus simbolus, jo šīs parasti ir kompilatora ģenerētas. To var darīt, jo funkcijas ir rakstītas secīgi viena pēc otras. Šāda pieeja nodrošina, ka analīze koncentrējas tikai uz reālām funkcijām, kas ir rakstītas programmētāja kodā.

Katrai identificētajai funkcijai analizators precīzi nosaka tās robežas – no funkcijas sākuma līdz nākamās funkcijas sākumam. Tas ļauj izolēt katras funkcijas asemblera kodu un analizēt to neatkarīgi, nepiejaucot citu funkciju instrukcijas.

Pēc koda strukturēšanas seko galvenais analīzes posms – steka operāciju identificēšana un to ietekmes modelēšana. Katras funkcijas analīze notiek sistemātiski, rindu pa rindai ejot caur tās asemblera kodu. Analizators uztur vairākus skaitītājus: push instrukciju skaitu, lokālo mainīgo rezervāciju, izsaukumu instrukciju skaitu un steka rāmja manipulācijas. Šis process ir līdzīgs virtuālās steka simulācijai, kur analizators seko līdzi tam, kā steka izmērs mainītos funkcijas izpildes laikā.

|  |
| --- |
| # Y reģistra steka rāmja šabloni  sbiw\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+sbiw\s+r28,\s+0x([0-9a-f]+)')  adiw\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+adiw\s+r28,\s+0x([0-9a-f]+)')  # SPL/SPH tiešās manipulācijas šabloni  spl\_in\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+in\s+r\d+,\s\*0x3d')  sph\_in\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+in\s+r\d+,\s\*0x3e')  spl\_out\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+out\s+0x3d,\s\*r\d+')  sph\_out\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+out\s+0x3e,\s\*r\d+')  # Funkciju izsaukumu šabloni  call\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+(?:call)\s+')  rcall\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+(?:rcall)\s+')  rcall\_relative\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+rcall\s+\.([+-]\d+)')  icall\_pattern = re.compile(r'^\s\*[0-9a-f]+:\s+[0-9a-f ]+\s+(?:icall|eicall)\s\*') |

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

2.1 koda fragmentā tiek definētas regulārās izteiksmes, kas atpazīs dažādas steka operācijas AVR asemblera kodā. Katra no šīm operācijām ietekmē steka izmantojumu citādi. Push un pop instrukcijas ir vienkāršākās (tiem neizmanto regulāras izteiksmes) – katrs push pievieno vienu baitu stekā, saglabājot reģistra vērtību, kamēr pop atjauno reģistra vērtību no steka. Analizators uzskaita šīs instrukcijas, lai noteiktu, cik reģistru funkcija saglabā un atjauno.

Sarežģītākas ir steka rāmja manipulācijas, kas ietver Y reģistra izmantošanu. Instrukcija sbiw r28, 0x10 (sk. 2.3. attēlu) nozīmē, ka tiek rezervēti 16 baiti lokālajiem mainīgajiem, samazinot Y reģistra vērtību par 16. Pretēji, adiw r28, 0x10 atbrīvo iepriekš rezervētos baitus. Analizators uzskaita visas šādas operācijas, lai aprēķinātu kopējo lokālo mainīgo izmēru.

Lai gan Y reģistrs ir GCC kompilatora standarta izvēle steka rāmja pārvaldībai, uz to nevar paļauties visos gadījumos. Augstākos optimizācijas līmeņos kompilators var izvēlēties neizmantot steka rāmi vispār, ja lokālie mainīgie efektīvi ietilpst reģistru failā. Manuāli rakstīts asemblera kods vai funkciju iekļautie (inline) asemblera fragmenti var izmantot Y reģistru citiem nolūkiem. Specifiskās situācijās kompilators var izvēlēties alternatīvas steka pārvaldības stratēģijas. Šādos gadījumos steka rezervēšana var notikt caur tiešu SP manipulāciju, izmantojot SPL (Stack Pointer Low, adrese 0x3d) un SPH (Stack Pointer High, adrese 0x3e) reģistrus. Šī pieeja ietver steka rādītāja nolasīšanu reģistros, tā samazināšanu par nepieciešamo baitu skaitu un jauno vērtību ierakstīšanu atpakaļ.

Lai gan šāda analīze ir teorētiski iespējama, praksē tā sastopas ar būtiskiem izaicinājumiem. Pirmkārt, reģistru izsekošana caur daudzām instrukcijām ir sarežģīta, jo reģistri var tikt izmantoti citiem nolūkiem starp nolasīšanu un ierakstīšanu. Otrkārt, kompilators var ievietot citas instrukcijas starp steka manipulācijas operācijām, kas sarežģī modeļu atpazīšanu. Treškārt, ir grūti noteikt, vai steka rādītāja manipulācija patiešām rezervē vietu lokālajiem mainīgajiem vai arī tā ir daļa no kādas citas operācijas. Piemēram, steka rādītājs var tikt pielāgots pagaidu nolūkos un pēc tam atjaunots.

Apzinoties šīs sarežģītības, analizators izmanto pragmatisku stratēģiju SPL/SPH manipulāciju apstrādē. Tas identificē tiešu steka rādītāja manipulāciju, meklējot in un out instrukcijas ar SPL (0x3d) un SPH (0x3e) adresēm, bet neanalizē to detalizēti. Tā vietā analizators izmet brīdinājuma ziņojumu tikai gadījumos, kad funkcija izmanto SPL/SPH manipulācijas bez standarta Y reģistra steka rāmja organizācijas. Ja funkcija satur gan Y reģistra SBIW/ADIW instrukcijas, gan SPL/SPH manipulācijas, analizators uzskata, ka SPL/SPH operācijas ir daļa no sarežģītākas, bet standartizētas steka organizācijas, un brīdinājums netiek rādīts. Tomēr, ja funkcija izmanto tikai SPL/SPH manipulācijas bez jebkādām Y reģistra operācijām, tiek parādīts brīdinājums, kas informē lietotāju, ka steka analīze var būt nepilnīga un ieteic pārbaudīt GCC steka ziņojumus papildu verifikācijai. Šāda selektīvā brīdinājumu stratēģija nodrošina, ka lietotāji tiek informēti par potenciālām analīzes neprecizitātēm, vienlaikus izvairīties no nevajadzīgiem brīdinājumiem standarta situācijās.

Īpaša uzmanība tiek pievērsta AVR-specifiskai rcall .+0 konstrukcijai, kas ir neparasta tehnika steka rāmja izveidei. Šī instrukcija faktiski izsauc nākamo instrukciju, radot divu baitu atgriešanās adreses ierakstu stekā, kas tiek izmantots kā papildu steka vieta.

Svarīgs kvalitātes kontroles elements ir steka bilances pārbaude. Analizators pārbauda, vai push un pop instrukciju skaits ir sabalansēts, kā arī vai steka rāmja rezervēšana un atbrīvošana notiek pareizā proporcijā. Ja tiek atklātas nesakritības, tas var norādīt uz kompilācijas optimizācijām vai potenciālām problēmām koda struktūrā.

Pēc visas funkcijas instrukciju analīzes analizators aprēķina kopējo steka izmantojumu, izmantojot matemātisku modeli. Formula ir vienkārša, bet tās pamatā ir AVR arhitektūras izpratne: **Kopējais steka izmantojums = Push instrukcijas + Steka rāmja rezervācija + Atgriešanās adrese**.

Push instrukcijas reprezentē saglabātos reģistrus, steka rāmja rezervācija ietver lokālos mainīgos un pagaidu datus, bet atgriešanās adrese ir konstanta divu baitu liela vērtība, kas nepieciešama katras funkcijas izsaukšanai AVR arhitektūrā.

Katras funkcijas analīzes noslēgumā analizators saglabā ne tikai galīgo steka izmantojuma skaitli, bet arī detalizētu sadalījumu par to, cik baiti tiek izmantoti katram mērķim. Šī informācija ir nenovērtējama funkciju optimizācijā un steka problēmu diagnosticēšanā, jo tā ļauj precīzi identificēt steka patēriņa avotus un potenciālās optimizācijas iespējas.

Papildus steka aprēķinam analizators arī klasificē funkciju izsaukumus, šķirojot call, rcall un icall instrukcijas. Šī informācija ir būtiska izsaukumu grafa rekonstrukcijā un palīdz identificēt funkcijas lomu programmas arhitektūrā – funkcijas ar daudziem izsaukumiem parasti ir centrālas programmas loģikā, kamēr funkcijas bez izsaukumiem ir lapas funkcijas, kas veic specifiskus uzdevumus.

Pēc individuālo funkciju analīzes pabeigšanas analizators veic rezultātu konsolidāciju, apvienojot asemblera analīzes rezultātus ar GCC kompilatora ziņojumiem. Šis process nodrošina maksimālu precizitāti, izmantojot abu metožu priekšrocības.

Analizators piešķir prioritāti asemblera analīzes rezultātiem, jo tie atspoguļo faktisko instrukciju ietekmi uz steku. Ja asemblera analīze kādu funkciju nav atradusi vai aprēķinājusi, tiek izmantoti GCC kompilatora ziņojumi kā rezerves variants. Šāda hierarhiska pieeja nodrošina, ka katrai funkcijai tiek piešķirta visticamākā un precīzākā steka izmantojuma vērtība.

Ja funkcija nav atrodama nevienā no avotiem, analizators izmet kļūdu, jo bez šīs informācijas nav iespējams veikt uzticamu kopējo steka analīzi. Šī stingrā pārbaude novērš klusas kļūdas, kas varētu radīt neprecīzus rezultātus.

### Rekursīvo funkciju analīze

Rekursīvo funkciju analīze veido vienu no sarežģītākajām un tehnisko izaicinājumu bagātākajām steka izmantojuma analīzes komponentēm. Atšķirībā no parastajām funkcijām, kuru steka patēriņš ir statisks un prognozējams, rekursīvo funkciju steka izmantojums ir dinamisks un tieši atkarīgs no rekursijas dziļuma, kas savukārt ir nosakāms no ievades parametriem. Šī fundamentālā atšķirība rada nepieciešamību pēc specializētām analīzes metodēm, kas spēj ne tikai identificēt rekursīvās funkcijas, bet arī precīzi novērtēt to maksimālo steka patēriņu dažādos izpildes scenārijos.

Rekursīvo funkciju steka analīzes sarežģītību var ilustrēt ar vienkāršu faktoriāla funkcijas piemēru. Ja faktoriāla funkcija tiek izsaukta ar parametru 3, tā veiks trīs rekursīvos izsaukumus un patērēs relatīvi mazu steka daudzumu. Tomēr, ja tā pati funkcija tiek izsaukta ar parametru 1000, steka patēriņš palielināsies tūkstošreiz. Šī dinamiskā daba padara tradicionālās statiskās analīzes metodes nepietiekamas un prasa jaunu pieeju izstrādi.

Izaicinājumi kļūst vēl sarežģītāki, pārejot no C pirmkoda uz asemblera līmeni. Kamēr pirmkodā rekursiju ir salīdzinoši viegli identificēt, meklējot funkcijas, kas izsauc pašas sevi, asemblera līmenī šis uzdevums kļūst ievērojami sarežģītāks. Kompilators var veikt dažādas optimizācijas, kas maina rekursīvo izsaukumu struktūru. Piemēram, astes rekursija var tikt pārveidota par iteratīvu ciklu, vai arī funkcijas var tikt integrētas izsaukuma vietā, radikāli mainot to asemblera reprezentāciju.

Papildus tam AVR arhitektūra piedāvā vairākus funkciju izsaukuma veidus. Tiešie izsaukumi izmanto call instrukciju ar absolūtu adresi, relatīvie izsaukumi izmanto rcall instrukciju ar relatīvu nobīdi, bet netiešie izsaukumi izmanto icall vai eicall instrukcijas ar adresi, kas glabājas reģistros. Katrs no šiem izsaukuma veidiem prasa specifiskas analīzes tehnikas, lai precīzi identificētu rekursīvos modeļus.

Izstrādātais rekursijas identificēšanas algoritms darbojas divos posmos: vispirms izveidojot funkciju adrešu kartējumu un pēc tam analizējot izsaukuma instrukcijas. Funkciju adrešu kartēšanas posms ir kritiski svarīgs, jo AVR arhitektūrā dažādas instrukcijas var atsaukties uz vienu un to pašu funkciju, izmantojot atšķirīgus adrešu formātus.

Algoritms izveido visaptverošu adrešu kartējumu, kurā katra funkcija tiek saglabāta vairākos formātos. Oriģinālais heksadecimālais formāts saglabā adresi tieši tā, kā tā parādās disasemblētajā kodā. Formāts bez sākuma nullēm nodrošina saderību ar optimizētām adrešu reprezentācijām. Baitu adreses un vārdu adreses tiek saglabātas atsevišķi, jo AVR arhitektūra izmanto gan baitu, gan vārdu adresēšanu atkarībā no instrukcijas tipa. Vārdu adresēšana ir AVR mikrokontrolieru specifiska īpašība, kur programmas atmiņā (Flash) katra instrukcija aizņem vienu "vārdu" – 16 bitu jeb 2 baitu bloku. Tāpēc programmas koda adreses tiek izteiktas vārdos, nevis baitos. Piemēram, ja funkcija atrodas Flash atmiņas baitu adresē 0x0200 (512 decimālā), tad tās vārdu adrese būs 0x0100 (256 decimālā), jo 512 / 2 = 256. Kad AVR procesors izpilda izsaukuma instrukcijas kā call vai rcall, tas izmanto vārdu adreses, jo katrs programmas solis ir viens vārds. Savukārt, kad disasemblē kodu, tas parasti norāda baitu adreses, lai būtu skaidrs precīzs atmiņas izvietojums. Šī daudzu formātu pieeja nodrošina, ka neatkarīgi no tā, kādā veidā kompilators ģenerē izsaukuma instrukciju, algoritms spēj pareizi identificēt mērķa funkciju.

Otrais posms ietver sistemātisku asemblera koda skenēšanu, meklējot izsaukuma instrukcijas. Algoritms izmanto specializētus regulāro izteiksmju šablonus katram izsaukuma tipam. Tiešajiem izsaukumiem tiek meklētas call instrukcijas ar heksadecimālām adresēm. Relatīvajiem izsaukumiem tiek analizētas rcall instrukcijas, īpašu uzmanību pievēršot rcall instrukcijām ar nulles nobīdi, kas bieži tiek izmantotas steka rāmja izveidei, nevis funkciju izsaukšanai.

Netiešo izsaukumu analīze ir īpaši sarežģīta un reprezentē vienu no lielākajiem statiskās analīzes izaicinājumiem AVR arhitektūrā. Lai saprastu šo sarežģītību, vispirms jāizprot, ka netiešie izsaukumi izmanto Z reģistru (r30:r31 pāri) kā rādītāju uz funkcijas adresi. Analizators mēģina izsekot šī reģistra saturu, taču reālajā praksē tas ir daudz sarežģītāk nekā šķiet.

Vienkāršākajos gadījumos algoritms var izsekot tiešas vērtību ielādes Z reģistrā, izmantojot ldi instrukcijas, kas ielādē konstantes r30 un r31 reģistros. Taču šāda pieeja darbojas tikai gadījumos, kad funkcijas adrese ir zināma kompilācijas laikā un tiek ielādēta tieši pirms izsaukuma. Reālajā programmēšanas praksē netiešie izsaukumi biežāk izmanto dinamiskus risinājumus – funkciju adreses tiek ielādētas no funkciju rādītāju masīviem RAM atmiņā, aprēķinātas, balstoties uz mainīgiem indeksiem, vai noteiktas izpildlaika laikā atkarībā no programmas stāvokļa.

Šādos sarežģītākos gadījumos, kad adreses tiek "pagrābtas" no RAM atmiņas caur komplicētām adresācijas shēmām, statiskā analīze saskaras ar fundamentāliem ierobežojumiem. Analizators nevar paredzēt, kādas būs mainīgo vērtības izpildlaika laikā, tāpēc tam jāpieņem konservatīvas pieņemšanas. Kad icall vai eicall instrukcijas tiek sastaptas bez skaidri izsekojamas Z reģistra ielādes, algoritms mēģina identificēt iespējamos kandidātus, analizējot kontekstu un meklējot tipiskus funkciju rādītāju masīvu piekļuves šablonus, taču šāda analīze nav pilnīgi ticama un var novest gan pie nepilnīgiem, gan pārāk konservatīviem rezultātiem.

Rekursija tiek identificēta, salīdzinot katra izsaukuma mērķa adresi ar pašreizējās funkcijas adresi. Ja šīs adreses sakrīt, funkcija tiek atzīmēta kā rekursīva. Algoritms īpašu uzmanību pievērš kompilatora ģenerētajām atzīmēm un iekšējām funkcijām, lai izvairītos no viltus pozitīvo rezultātu ģenerēšanas.

Rekursijas dziļuma noteikšana ir kritiskā analīzes komponente, jo tā nosaka rekursīvās funkcijas kopējo steka patēriņu. Šis process prasa ne tikai asemblera koda analīzi, bet arī semantisko izpratni par programmas darbību izpildīšanas laikā. Algoritms izpilda daudzpakāpju stratēģiju, kas pakāpeniski mēģina identificēt rekursijas sākotnējos parametrus, izmantojot arvien sarežģītākas analīzes metodes.

Pirmais un vienkāršākais līmenis meklē tiešos literāļu izsaukumus pirmkodā. Ja algoritms atrod konstrukcijas kā factorial(5) vai fibonacci(10), tas var tieši noteikt rekursijas maksimālo dziļumu. Šī metode darbojas efektīvi vienkāršos gadījumos, kur rekursīvās funkcijas tiek izsauktas ar konstantēm.

Otrais līmenis koncentrējas uz main funkcijas analīzi, meklējot mainīgo definīcijas un piešķīrumus. Algoritms izmanto regulārās izteiksmes, lai identificētu mainīgo deklarācijas ar sākotnējām vērtībām un pēc tam izsekotu šo mainīgo izmantošanu rekursīvo funkciju izsaukumos. Piemēram, ja kods satur int *n* = 15; factorial(*n*); konstrukciju, algoritms spēj noteikt, ka rekursijas dziļums būs 15 līmeņi.

Trešais un vissarežģītākais līmenis ietver parametru vērtību izsekošanu cauri funkciju izsaukumu ķēdēm. Šis process darbojas rekursīvi, analizējot katru funkciju hierarhijas līmeni. Algoritms identificē visas funkcijas, kas izsauc mērķa rekursīvo funkciju, un pēc tam analizē, kā parametri tiek nodoti cauri šīm starpniecības funkcijām. Ja rekursīvā funkcija tiek izsaukta netieši caur citu funkciju, algoritms izseko parametru vērtības cauri visai izsaukumu ķēdei. Parametru izsekošanas algoritms izmanto regulārās izteiksmes, lai identificētu funkciju definīcijas un to izsaukuma vietas. Tas analizē parametru nosaukumus un pārbauda, vai tie tiek nodoti tieši vai caur mainīgiem. Šī pieeja ļauj analizēt arī sarežģītākus gadījumus, kur rekursijas dziļums nav tieši redzams main funkcijā.

Rekursīvo funkciju efektīvai analīzei ir nepieciešams izprast to matemātisko dabu. Algoritms klasificē rekursīvās funkcijas divos galvenajos tipos, balstoties uz to parametru samazināšanas modeļiem. Šī klasifikācija ir fundamentāli svarīga, jo katram tipam ir atšķirīgs matemātisks modelis rekursijas dziļuma aprēķināšanai.

Atskaitīšanas rekursija seko lineāram samazināšanas modelim, kur katrs rekursīvais izsaukums samazina parametru par konstanti. Algoritms identificē šo tipu, meklējot konstrukcijas, kurās rekursīvais izsaukums satur atņemšanas operāciju. Regulārā izteiksme meklē situācijas, kur funkcijas izsaukumā no parametra tiek atņemts kāds pozitīvs skaitlis. Rekursijas dziļuma aprēķins šajā gadījumā ir tiešs: ja sākotnējā vērtība ir *n* un katrs solis samazina to par *k*, tad rekursijas dziļums ir **floor(*n*/*k*) + 1** (floor – noapaļošana uz leju).

Dalīšanas rekursija darbojas pēc eksponenciāla samazināšanas modeļa, kur katrs rekursīvais izsaukums dala parametru ar konstanti. Šis tips ir raksturīgs algoritmiem kā binārās ievietošanas algoritms (binary search) vai saliešanas algoritms (merge sort). Algoritms identificē dalīšanas rekursiju, meklējot konstrukcijas ar dalīšanas operatoru vai bitu nobīdes operācijām. Bitu nobīdes gadījumā algoritms saprot, ka *n* >> *k* ir ekvivalents *n* / 2^*k*, un attiecīgi aprēķina dalītāju.

Rekursijas dziļuma aprēķins dalīšanas gadījumā izmanto logaritmisko funkciju. Ja sākotnējā vērtība ir *n* un dalītājs ir *d*, tad rekursijas dziļums ir **ceil(log\_*d*(*n*)) + 1** (ceil – noapaļošana uz augšu). Algoritms izmanto matemātisku bibliotēku, lai veiktu šos aprēķinus ar pietiekamu precizitāti.

Kopējā rekursīvā steka izmantojuma aprēķins ir sarežģītāks par vienkāršu vietējā steka izmantojuma reizināšanu ar rekursijas dziļumu. Katrs rekursīvais izsaukums pievieno stekam ne tikai funkcijas lokālos mainīgos, bet arī atgriešanās adresi, reģistru saglabāšanas telpu un funkcijas parametrus.

AVR arhitektūrā atgriešanās adrese vienmēr aizņem divus baitus, un algoritms to konsekventi ņem vērā visos aprēķinos. Reģistru saglabāšana ir atkarīga no konkrētās funkcijas izpildes, un tā tiek noteikta, analizējot push un pop instrukcijas asemblera kodā. Funkcijas parametri var tikt nodoti caur reģistriem vai steku, atkarībā no to skaita un tipa.

Algoritms aprēķina kopējo rekursīvo steka izmantojumu, izmantojot formulu (2.2.).

***S = (L + A + P) × D*** (2.2.)

kur

*S* ir kopējais rekursīvais steks (visa atmiņa, ko patērē rekursīva funkcija),

*L* ir vietējais steka izmantojums (atmiņa lokālajiem mainīgajiem katrā funkcijas izsaukumā),

*A* ir atgriešanās adrese (atmiņa, lai saglabātu, kur atgriezties pēc funkcijas izpildes),

*P* ir parametru izmantojums (atmiņa, lai saglabātu funkcijas parametrus),

*D* ir rekursijas dziļums (cik reizes funkcija izsauc sevi).

(2.2.) formula nodrošina, ka tiek ņemta vērā visa steka telpa, kas nepieciešama pilnai rekursīvajai ķēdei.

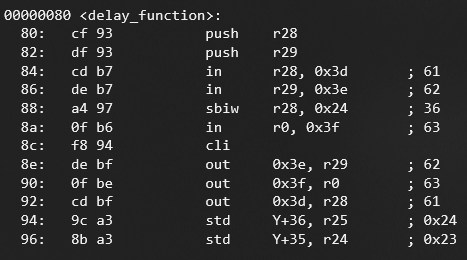
Īpaša uzmanība tiek pievērsta gadījumiem, kur rekursīvās funkcijas ir daļa no lielākas izsaukumu hierarhijas. Ja rekursīvā funkcija tiek izsaukta no main funkcijas caur vairākām starpniecības funkcijām, kopējais steka izmantojums iekļauj gan hierarhijas steku, gan pilno rekursīvo steku.

Pašreizējais īstenojums efektīvi apstrādā tiešos rekursijas gadījumus, bet tai ir daži ierobežojumi, kas rada iespējas turpmākai attīstībai. Algoritms nevar analizēt savstarpējo rekursiju, kur *A* funkcija izsauc *B* funkciju, kas izsauc *A* funkciju. Tas arī nevar precīzi analizēt gadījumus, kur rekursijas dziļums ir atkarīgs no sarežģītiem nosacījumiem vai dinamiskiem datiem.

### Izsaukumu grafa rekonstrukcija no asemblera

Izsaukumu grafa rekonstrukcija ir fundamentāls process, kurā no kompilēta asemblera koda tiek atjaunota oriģinālā funkciju izsaukumu struktūra. Šis uzdevums ir kritiski svarīgs steka analīzē, jo bez precīzas informācijas par to, kura funkcija izsauc kuru, nav iespējams aprēķināt maksimālo steka izmantojumu.

Kad C kods tiek kompilēts, augsta līmeņa konstrukcijas tiek transformētas zemā līmeņa mašīnkoda instrukcijās. Funkcijas izsaukums C kodā, piemēram, delay\_function(*a*) no program.c (sk. 2.4. attēlu), kļūst par vairākām asemblera instrukcijām: parametru ielādē reģistros, izsaukuma instrukciju uz konkrētu atmiņas adresi, un rezultāta apstrādi. Mūsu uzdevums ir pretējais process – no šīm atsevišķajām instrukcijām rekonstruēt oriģinālo programmas loģisko struktūru.



*2.4. att.* Disamblēta delay\_function(a) no program.c (sk. 1. pielikumu)

Šis uzdevums sastopas ar vairākiem būtiskiem izaicinājumiem. Pirmkārt, kompilācijas procesā zūd funkciju nosaukumi (kļūst par adresēm), mainīgo nozīme (kļūst par reģistra vērtībām) un parametru saistība ar funkcijām, kas apgrūtina rekursīvo funkciju identificēšanu un parametru vērtību izsekošanu. Otrkārt, kompilatora optimizācijas var kardināli izmainīt koda struktūru. Treškārt, dažas konstrukcijas, kā funkciju rādītāji, rada netiešos izsaukumus, kuru mērķi nav statiskās analīzes līmenī nosakāmi.

Analizators risina šo problēmu ar secīgu vairāku soļu pieeju. Algoritms iziet cauri asemblera kodam vairākās fāzēs, katru reizi vācot specifisku informāciju un pakāpeniski veidojot pilnu izsaukumu grafu. Pirmā fāze koncentrējas uz funkciju identifikāciju un to adrešu kartēšanu. Otra fāze analizē izsaukuma instrukcijas. AVR arhitektūrā galvenokārt ir trīs izsaukumu tipi: CALL (tiešais izsaukums ar pilnu adresi), RCALL (relatīvais izsaukums ar nobīdi), un ICALL (netiešais izsaukums caur reģistru). Katrs tips prasa atšķirīgu apstrādes pieeju.

Tiešie izsaukumi ir relatīvi vienkārši gadījumi. CALL instrukcijas satur pilnu mērķa adresi, kas tieši norāda uz izsaucamo funkciju. RCALL instrukcijas izmanto relatīvu nobīdi no pašreizējās pozīcijas. Analizators identificē šīs instrukcijas ar regulāriem izteicieniem un izņem mērķa adreses. Tomēr arī šeit pastāv nianses. Ne visas RCALL instrukcijas ir funkcionāli izsaukumi. Instrukcija rcall .+0 ir GCC kompilatora izmantots tehnisks paņēmiens steka ietvara izveidošanai, nevis īsts funkcijas izsaukums. Analizators atpazīst šo īpašo gadījumu un klasificē to kā steka manipulāciju, nevis funkcionālu izsaukumu.

Netiešie izsaukumi rada būtiski lielākus izaicinājumus. Tie rodas, kad C kodā tiek izmantoti funkciju rādītāji vai atgriezeniskās izsaukšanas (callback) mehānismi. Asemblera kodā šādi izsaukumi parādās kā ICALL vai EICALL instrukcijas, bet mērķa funkcija nav tieši norādīta instrukcijā. Tā vietā mērķa adrese tiek glabāta Z reģistrā (r31:r30 reģistru pāris). Lai noteiktu izsaukuma mērķi, analizatoram jāizseko, kādas vērtības tiek ielādētas šajā reģistrā pirms ICALL instrukcijas izpildes.

Analizators ievieš Z reģistra izsekošanas mehānismu. Tas atpazīst LDI (load immediate) instrukcijas, kas ielādē konstantes vērtības r30 un r31 reģistros. Kad pēc tam tiek konstatēta ICALL instrukcija, analizators rekonstruē pilno adresi no iepriekš saglabātajām reģistra vērtībām. Šis process prasa precīzu izsekošanu katrai funkcijai atsevišķi, jo reģistra saturs var mainīties starp dažādām funkcijām. Analizators uztur atsevišķas reģistra vērtību tabulas katrai funkcijai.

Vēl sarežģītāks scenārijs rodas ar masīva bāzētiem funkciju rādītājiem. Šādā gadījumā Z reģistrs tiek ielādēts nevis ar konstanti, bet ar vērtību, kas nolasīta no masīva elementa. C kodā tas var izskatīties kā funkciju rādītāju masīvs, kur konkrētā funkcija tiek izvēlēta, pamatojoties uz indeksa vērtību. Asemblera kodā šis process parādās kā instrukciju secība, kur vispirms tiek ielādēta masīva bāzes adrese, tad pievienots indekss, un visbeidzot no aprēķinātās adreses nolasīta funkcijas adrese. Šādu instrukciju secību analizators atpazīst kā masīva piekļuvi. Šajā situācijā precīza mērķa funkcijas noteikšana statiskās analīzes līmenī nav iespējama, jo indeksa vērtība ir zināma tikai izpildlaikā. Analizators izvēlas konservatīvu risinājumu – tas pievieno visas potenciālās mērķa funkcijas izsaukumu grafam, izslēdzot tikai acīmredzami nepiemērotas funkcijas, kā main funkciju vai utilītas funkcijas.

Rekursīvo funkciju atklāšana notiek paralēli izsaukumu grafa veidošanai. Analizators meklē situācijas, kur funkcija satur izsaukumu uz sevi pašu (sk. 2.4.2. apakšnodaļu).

Mūsdienu kompilatori ģenerē daudz palīgelementu, kas var maldināt analīzi. Asemblera kodā bieži parādās atzīmes ar nosaukumiem kā .L15, .Loc. Šīs nav īstas funkcijas lietotāja izpratnē, bet gan kompilatora iekšējās optimizācijas vai atkļūdošanas palīgelements. Analizators atpazīst šīs konstrukcijas pēc to raksturīgajiem nosaukumu modeļiem un neiekļauj tās izsaukumu grafā kā atsevišķas funkcijas. Tas ir svarīgi, jo pretējā gadījumā grafs būtu pārpildīts ar neeksistējošiem savienojumiem.

Kompilatora optimizācijas var arī pilnībā izdzēst funkcijas no asemblera koda. Inline optimizācija aizstāj funkcijas izsaukumus ar funkcijas koda kopiju izsaukuma vietā. Šādas funkcijas vairs nav redzamas asemblera kodā, un to izsaukumi netiek iekļauti grafā. Parasti tā nav problēma steka analīzei, jo inline kods neveido jaunu steka ietvaru.

Rekonstruētais izsaukumu grafs tiek reprezentēts kā blakusesošo virsotņu saraksts (adjacency list), kur katra funkcija ir saistīta ar to funkciju sarakstu, kuras tā izsauc. Šī reprezentācija ir piemērota turpmākajiem algoritmiskiem pielietojumiem, jo ļauj efektīvi veikt grafa šķērsošanu un ceļu analīzi.

Analizators veic vairākus validācijas soļus. Tas pārbauda, vai visām funkcijām ar netiešajiem izsaukumiem ir identificēti iespējamie mērķi. Tas arī brīdina par situācijām, kur izsaukuma mērķi nevarēja noteikt, vai kur konstatēti neparasti izsaukumu modeļi.

Izsaukumu grafa rekonstrukcijas kvalitāte tieši ietekmē visas turpmākās steka analīzes precizitāti. Nepilnīgs grafs var novest pie pārāk optimistisku steka novērtējumu, kas ir bīstami. Pārāk konservatīvs grafs rada drošus, bet potenciāli neefektīvus risinājumus.

Algoritma ierobežojumi izriet no statiskās analīzes fundamentālajām robežām. Daudzas programmas īpašības, īpaši tās, kas atkarīgas no izpildlaika vērtībām, nav statiskās analīzes līmenī nosakāmas. Analizators sniedz labāko iespējamo novērtējumu, pamatojoties uz pieejamo informāciju, bet gala lēmumi par sistēmas dizainu prasa inženieru ekspertīzi.

Rekonstruētais izsaukumu grafs kalpo par pamatu maksimālā steka ceļa aprēķinam un rekursijas dziļuma analīzei. Tā precizitāte nosaka visa steka analīzes procesa kvalitāti un rezultātu uzticamību.

### Maksimālā steka dziļuma aprēķināšana

Maksimālā steka izmantojuma aprēķināšana ir analizatora sarežģītākā un kritiskākā komponente, kas apvieno visus iepriekšējos analīzes soļus vienotā algoritmā. Šis process nevis vienkārši summē visu funkciju steka patēriņu, bet gan precīzi modelē programmas izpildes dinamiku, identificējot tieši to funkciju izsaukumu secību, kas rada vislielāko steka patēriņu.

Analizatora pamatā ir izpratne, ka steka izmantojums programmas izpildes laikā ir kumulatīvs process. Kad funkcija *A* izsauc funkciju *B*, funkcijas *B* steka patēriņš tiek pievienots funkcijas *A* jau esošajam steka izmantojumam. Šī "steka kaudze" turpina augt ar katru jaunu funkcijas izsaukumu un samazinās tikai tad, kad funkcijas pabeidz savu izpildi un atgriežas.

Pirms maksimālā steka izmantojuma aprēķināšanas analizators veic būtisku priekšdarbu – nosaka, cik daudz no mikrokontroliera kopējās SRAM atmiņas patiešām ir pieejams steka vajadzībām.

Analizators izmanto get\_memory\_sections metodi, kas ar avr-size rīka palīdzību analizē kompilētā ELF faila atmiņas sekcijas. Šis process atklāj divus kritiski svarīgus atmiņas patērētājus, kas "konkurē" ar steku par SRAM resursu piešķiršanu .data un .bss sekcijas. Analizators aprēķina pieejamo steka telpu, izmantojot vienkāršu formulu:

**Pieejamā steka atmiņa = Kopējā SRAM - (.data izmērs + .bss izmērs)**.

Maksimālā steka aprēķināšanas algoritms balstās uz rekursīvu "dziļumā vispirms" (depth-first search) pieeju, kas sistemātiski izpēta visus iespējamos funkciju izsaukumu ceļus programmas izsaukumu grafā. Algoritma pamatideja ir vienkārša: sākot no programmas ieejas punkta (main funkcija), rekursīvi aprēķināt steka patēriņu katram iespējamam izpildes ceļam un identificēt to, kas rada vislielāko kumulatīvo steka izmantojumu.

Pirms galvenās analīzes sākšanas analizators veic rekursīvo funkciju priekšapstrādi, jo šīm funkcijām nepieciešama īpaša pieeја steka aprēķināšanā. Katrai rekursīvajai funkcijai tiek aprēķināts kopējais steka patēriņš, ņemot vērā iepriekš noteikto rekursijas dziļuma ierobežojumu.

Lai izvairītos no atkārtotiem aprēķiniem sarežģītos izsaukumu grafos, analizators izmanto memoizācijas tehniku. Katram funkcijas un izsaukumu ceļa pārim tiek saglabāts aprēķinātais rezultāts. Pirms jauna aprēķina veikšanas algoritms pārbauda, vai šāds ceļš jau ir analizēts, un vajadzības gadījumā atgriež saglabāto rezultātu.

Memoizācijas atslēga sastāv no funkcijas nosaukuma un pašreizējā izsaukumu ceļa, kas nodrošina, ka tiek ņemts vērā konteksts, kurā funkcija tiek izsaukta. Šī pieeja ir īpaši svarīga programmās ar sarežģītiem izsaukumu grafiem, kur daudzi ceļi var pārklāties vai kur funkcijas tiek izsauktas no vairākām vietām.

Galvenā analīzes funkcija get\_stack\_usage darbojas rekursīvi, sākot no norādītās funkcijas un dziļumā izpētot visus tās iespējamos izsaukumu ceļus. Algoritma darbību var aprakstīt vairākos soļos:

* Vispirms tiek pārbaudīts, vai pašreizējā funkcija jau atrodas izsaukumu ceļā, kas norādītu uz ciklisko atkarību. Šādi cikli tiek apstrādāti īpaši – ja funkcija nav rekursīva, cikls tiek uzskatīts par kļūdu un funkcija atgriež nulli. Rekursīvām funkcijām šī pārbaude ļauj izvairīties no bezgalīgām cilpām analīzes procesā.
* Ja funkcija ir rekursīva un tā pirmo reizi parādās pašreizējā ceļā, tiek izmantots iepriekš aprēķinātais rekursīvais steka patēriņš. Tas nozīmē, ka rekursīvā funkcija tiek izvērsta pilnā dziļumā vienā solī, nevis analizēta soli pa solim, kas varētu radīt bezgalīgas cilpas.
* Ja funkcija ir lapas funkcija (tā neizsauc citas funkcijas), tās kopējais steka patēriņš ir vienāds ar tās vietējo patēriņu, un šis ceļš tiek reģistrēts kā pilns izpildes ceļš analīzes noslēguma ziņojumā.

Visinteresantākā un sarežģītākā daļa ir situācija, kad funkcija var izsaukt vairākas citas funkcijas. Šajā gadījumā analizators rekursīvi aprēķina steka patēriņu katram iespējamam turpmākam ceļam un identificē to, kas rada vislielāko patēriņu.

Algoritms salīdzina visus iespējamos ceļus un saglabā maksimālo steka patēriņu, kas reprezentē "sliktāko gadījumu" šīs funkcijas kontekstā. Šī maksimālā vērtība tiek pievienota funkcijas vietējam steka patēriņam, veidojot kopējo steka izmantojumu šim funkcijas izsaukumam.

Papildus maksimālā steka patēriņa aprēķinam analizators veic arī visu pilno izpildes ceļu kartēšanu. Katru reizi, kad tiek sasniegta lapas funkcija, pašreizējais ceļš tiek reģistrēts kopā ar tā kopējo steka patēriņu. Šī informācija ir nenovērtējama steka optimizācijas nolūkos, jo tā ļauj identificēt ne tikai maksimālo patēriņu, bet arī precīzi redzēt, kuri funkciju izsaukumu ceļi rada vislielāko steka slogu.

Visi reģistrētie ceļi tiek kārtoti pēc to steka patēriņa dilstošā secībā, ļaujot viegli identificēt problemātiskākos izpildes scenārijus. Šī informācija tiek iekļauta analizatora gala ziņojumā, sniedzot detalizētu pārskatu par programmas steka izmantojuma dinamiku.

Lai nodrošinātu pilnvērtīgu analīzes ziņojumu, analizators izmanto papildu algoritmu find\_max\_stack\_path, kas rekonstruē pilnu funkciju izsaukumu secību, kura rada maksimālo steka patēriņu. Šis algoritms darbojas līdzīgi galvenajai analīzes funkcijai, bet tā mērķis ir nevis aprēķināt steka patēriņu, bet gan identificēt konkrēto ceļu, kas rada šo patēriņu.

Maksimālā ceļa rekonstrukcija izmanto to pašu rekursīvo pieeju, bet katram funkcijas izsaukuma punktam tā izvēlas to turpmāko ceļu, kas rada vislielāko steka patēriņu. Rezultātā tiek izveidots lineārs saraksts ar funkciju nosaukumiem, kas reprezentē tieši to izsaukumu secību, kura patērē visvairāk steka atmiņas.

Pēc precīzā maksimālā steka patēriņa aprēķināšanas analizators pielieto desmit procentu drošības rezervi. Tā kompensē vairākus faktorus: statiskās analīzes potenciālās neprecizitātes, kompilatora optimizāciju neparedzētās sekas, kā arī iespējamās situācijas, kad programmas izpildes laikā var rasties dziļāka rekursija vai lielāks steka patēriņš nekā statiskajā analīzē prognozēts. Desmit procentu rezerve ir uzskatāma par konservatīvu, bet drošu pieeju, kas palīdz izvairīties no steka pārplūdes problēmām reālā ekspluatācijā.

Analīzes noslēgumā analizators konsolidē visus aprēķinātos rezultātus vienotā datu struktūrā. Tiek saglabāts gan tīrais maksimālais steka patēriņš, gan patēriņš ar drošības rezervi, individuālo funkciju steka izmantojums, pilnais izsaukumu grafs, rekursīvo funkciju saraksts ar to dziļuma ierobežojumiem, kā arī visi identificētie izpildes ceļi kārtoti pēc to steka patēriņa.

Šī visaptverošā informācija veido pamatu detalizētam analīzes ziņojumam, kas ne tikai sniedz galīgo atbildi par maksimālo steka izmantojumu, bet arī piedāvā dziļu ieskatu programmas steka izmantojuma dinamikā. Ziņojums ietver soli pa solim aprēķinu maksimālajam ceļam, ļaujot izstrādātājiem precīzi saprast, kur un kāpēc rodas vislielākais steka patēriņš, kas ir nenovērtējami svarīgi programmas optimizācijai un atkļūdošanai.

## Analizatora testēšana

Izstrādātā AVR stekatmiņas analizatora testēšana tika veikta, izmantojot "melnās kastes" (black box) testēšanas pieeju ar speciāli izveidotu testa programmu komplektu, kas ietver dažādus programmatūras grūtības līmeņus un specifiskus AVR arhitektūras scenārijus. Šī testēšanas metodika pārbauda analizatora funkcionalitāti, fokusējoties uz ievades un izvades atbilstību sagaidāmajiem rezultātiem. Testēšanas mērķis bija pārbaudīt analizatora precizitāti, identificēt tā ierobežojumus un novērtēt praktisko pielietojamību reālās programmēšanas situācijās.

Testēšana tika organizēta sistemātiski, izmantojot automatizētu testa skriptu (test.py), kas veic analizatora analīzi visiem testa failiem un apkopo rezultātus vienotā, salīdzināmā formātā. Testēšanas skripts test.py un visi testa faili ir pieejami 1. pielikumā. Katram testa failam tika iepriekš aprēķināts un komentāros norādīts sagaidāmais maksimālais steka patēriņš, izmantojot manuālu .su failu analīzi un funkciju izsaukumu ceļu aprēķinus, kā arī tika noteikts .data un .bss sekciju kopējais izmērs, izmantojot avr-size utilītu.

**Testēšanas konfigurācija:**

Mikrokontrolieris: ATmega328P

RAM izmērs: 2048 baiti

Optimizācijas līmenis: O0 (nav optimizācijas)

avr-button-led.c (4 baiti steks un 0 baiti .data un .bss) – Vienkāršākā testa programma, kas implementē pamata funkcionalitāti bez funkciju izsaukumiem. Programma nolasa pogas stāvokli un kontrolē LED. Šis tests pārbauda analizatora spēju apstrādāt minimālas steka prasības un pamata I/O operācijas.

avr-adc-pwm.c (12 baiti steks un 0 baiti .data un .bss) – Pamata sensoru nolasīšanas un PWM (Pulse Width Modulation) kontroles programma ar vairākām funkcijām. Tests pārbauda analizatora spēju analizēt funkciju izsaukumu ķēdes ar lokālajiem mainīgajiem un parametru nodošanu.

r\_and\_icall.c (15 baiti steks un 4 baiti .data un .bss) – Specializēts tests netiešo izsaukumu (ICALL) un relatīvo izsaukumu (RCALL) analīzei. Programma izmanto funkciju rādītāju masīvu, kas rada netiešos izsaukumus asemblera līmenī. Šis tests pārbauda analizatora spēju rekonstruēt izsaukumu grafu sarežģītos gadījumos.

data\_and\_bss.c (68 baiti steks un 356 baiti .data un .bss) – Kompleksa programma ar globālajiem mainīgajiem, statiskie mainīgie, un pārtraukumu apstrādes funkcijām. Tests pārbauda analizatora spēju aprēķināt pieejamo steka telpu, ņemot vērā .data un .bss sekciju izmērus.

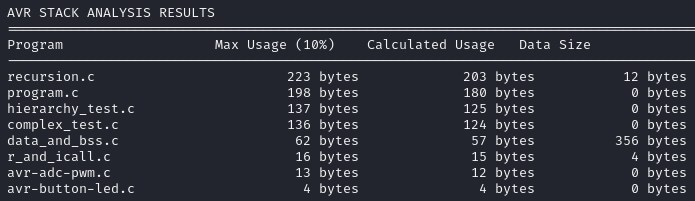
hierarchy\_test.c (125 baiti steks un 0 baiti .data un .bss) – Daudzlīmeņu funkciju hierarhijas tests ar četriem izsaukumu līmeņiem. Programma demonstrē dziļu funkciju izsaukumu ķēdi ar lokālajiem masīviem katrā līmenī. Tests pārbauda maksimālā steka ceļa aprēķināšanas precizitāti.

complex\_test.c (114 baiti steks un 0 baiti .data un .bss) – Sarežģīts tests ar rekursīvām funkcijām, vairākiem funkciju izsaukumu ceļiem un lieliem lokālajiem masīviem. Ietver faktoriāla aprēķinu ar rekursiju un vairākas palīgfunkcijas ar atšķirīgiem steka patēriņiem.

program.c (180 baiti steks un 0 baiti .data un .bss) – Rekursīvas delay\_function tests ar lokālo masīvu un parametru atkarīgu rekursijas dziļumu. Tests pārbauda analizatora spēju identificēt rekursijas dziļumu un aprēķināt kopējo rekursīvo steka patēriņu.

recursion.c (203 baiti steks un 12 baiti .data un .bss) – Visaptverošs rekursijas testa komplekts ar sešām dažādām rekursīvām funkcijām, kas implementē dažādus samazināšanas modeļus: atskaitīšanu (n-1, n-3), dalīšanu (n/2, n/4), un bitu nobīdes (n>>1, n>>3). Tests pārbauda analizatora spēju atpazīt un pareizi klasificēt dažādus rekursijas tipus.

Testa rezultāti (sk. 2.5. attēlu) parāda analizatora spēju ar augstu precizitāti aprēķināt gan steka izmantojumu, gan datu sekciju izmērus:



*2.5. att.* Analizatora testēšanas (test.py) (sk. 1. pielikumu) rezultātu tabula

2.5. attēla iegūtos rezultātus salīdzinot ar manuāli aprēķinātajām etalona vērtībām, rezultāti atklāj gan analizatora stiprās puses, gan nepilnības. Vienkāršākajos gadījumos (avr-button-led.c, avr-adc-pwm.c) precizitāte sasniedz 100%. Sarežģītākos scenārijos, piemēram, complex\_test.c gadījumā, analizators aprēķina 124 baitu steka patēriņu faktisko 114 baitu vietā, kas norāda uz rekursijas dziļuma neprecīzu noteikšanu. Līdzīgi data\_and\_bss.c testā analizators aprēķina 57 baitu steka izmantojumu faktisko 68 baitu vietā, kas atspoguļo pārtraukumu apstrādes funkciju analīzes trūkumus.

Testēšanas procesā tika identificētas divas būtiskas analizatora ierobežojuma jomas:

* Rekursijas dziļuma noteikšanas problēmas: complex\_test.c testā analizators nepareizi aprēķina factorial(0) gadījumu, jo nesaprot sarežģītākus beigu nosacījumus kā if (n <= 1) { return 1;}. Algoritms meklē tikai vienkāršus lineārus samazināšanas modeļus un nespēj analizēt kombinētus nosacījumus vai loģiskos operatorus rekursijas beigu noteikšanā.
* Pārtraukumu apstrādes funkciju analīzes trūkumi: data\_and\_bss.c testā analizators nevar pilnībā apstrādāt pārtraukumu apstrādes funkcijas (piemēram, \_\_vector\_21). Pārtraukumi var radīt papildu steka patēriņu, kas nav iekļauts statiskajā analīzē, īpaši gadījumos, kad pārtraukums notiek dziļā funkciju izsaukumu ķēdē.

Testēšanas rezultāti, salīdzinot analizatora aprēķinus ar etalona vērtībām:

* vienkārši gadījumi: 95–100% precizitāte;
* vidēji sarežģīti gadījumi: 85–95% precizitāte;
* sarežģīti rekursīvi gadījumi: 80–90% precizitāte;
* datu sekciju analīze: 100% precizitāte.

Analizators demonstrē augstu uzticamību gan steka izmantojuma aprēķinos, gan pieejamās atmiņas noteikšanā. Bez drošības rezerves analizators spēj diezgan precīzi noteikt faktisko steka patēriņu, tomēr 10% drošības rezerve padara rezultātus konservatīvākus, nodrošinot papildu aizsardzību pret neparedzētām situācijām. Šī pieeja, lai arī rada lielāku aprēķināto steka patēriņu nekā faktiskais, ir pamatota iegulto sistēmu kontekstā, kur steka pārplūde var izraisīt kritiskas sekas. Testēšanas rezultāti apstiprina, ka izstrādātais analizators ir efektīvs un uzticams rīks AVR mikrokontrolieru programmēšanā.

# SECINĀJUMI UN PRIEKŠLIKUMI

**Secinājumi**

Bakalaura darbā izstrādātais AVR mikrokontrolieru stekatmiņas izmantošanas analizators veiksmīgi sasniedz izvirzīto mērķi, nodrošinot augstas precizitātes statisko koda analīzi bez programmas izpildes un sniedzot detalizētu informāciju par maksimālo steka patēriņu dažādos programmas izpildes scenārijos.

Analizators efektīvi risina AVR mikrokontrolieru ierobežoto atmiņas resursu problēmu, piedāvājot rīku, kas ļauj izstrādātājiem identificēt potenciālās steka pārplūdes problēmas pirms programmas ievietošanas mikrokontrolierī, tādējādi novēršot sistēmas atteices kritiskās lietojumprogrammās.

Izstrādātais rekursīvo funkciju analīzes algoritms spēj ne tikai identificēt rekursīvās funkcijas asemblera kodā, bet arī noteikt to dziļuma ierobežojumus, analizējot parametru vērtības caur funkciju izsaukumu ķēdēm un atpazīstot dažādus rekursijas samazināšanas modeļus (atskaitīšana, dalīšana, bitu nobīde).

Izsaukumu grafa rekonstrukcijas algoritms veiksmīgi apstrādā gan tiešos (CALL), gan relatīvos (RCALL), gan netiešos (ICALL/EICALL) izsaukumus, izmantojot Z reģistra izsekošanu un vairāku adrešu formātu kartēšanu, kas nodrošina precīzu funkciju savstarpējo atkarību noteikšanu.

Analizatora modulārā arhitektūra ar Python valodas izmantošanu nodrošina augstu paplašināmību un pārnesamību starp dažādām operētājsistēmām, vienlaikus izmantojot AVR-GCC rīku komplekta priekšrocības kompilācijas un disasemblēšanas procesā.

Pētījums atklāj būtiskus ierobežojumus esošajās steka analīzes metodēs, jo GCC -fstack-usage opcija sniedz tikai lokālo funkciju informāciju bez konteksta par izsaukumu hierarhiju, kas padara to nepietiekamu sarežģītu programmu analīzei ar dziļām izsaukumu ķēdēm.

Analizatora testēšana ar reālām programmām atklāja papildu izaicinājumus, kas saistīti ar kompilatora ģenerētajām palīgfunkcijām (piemēram, \_\_udivmodsi4, \_\_floatunsisf), kuras netiek iekļautas standarta analīzē, bet var būtiski ietekmēt kopējo steka patēriņu.

Implementētā 10% drošības rezerve aprēķinātajam steka izmantojumam nodrošina konservatīvu, bet drošu pieeju, kas kompensē statiskās analīzes potenciālās neprecizitātes un neparedzētas situācijas programmas izpildes laikā.

**Priekšlikumi**

Kompilatora palīgfunkciju bibliotēkas izveide – Izveidot visaptverošu datu bāzi ar AVR-GCC kompilatora palīgfunkciju steka izmantojuma profiliem, kas ļautu analizatoram precīzi aprēķināt steka patēriņu arī gadījumos, kad tiek izmantotas peldošā komata operācijas, dalīšana vai citas aparātu riski neatbalstītas funkcijas.

Pārtraukumu apstrādes analīzes modulis – Paplašināt analizatoru ar specializētu moduli pārtraukumu apstrādes funkciju (\_\_vector\_XX) analīzei, kas ņemtu vērā pārtraukumu prioritātes, iespējamos pārtraukumu pārtraukumus un kritisko sekciju analīzi, kur pārtraukumi ir atspējoti.

Dinamiskās analīzes integrācija – Papildināt statisko analīzi ar simulatora vai emulatora integrāciju, kas ļautu verificēt statiskās analīzes rezultātus un identificēt gadījumus, kur statiskā analīze nav pietiekama (piemēram, dinamiski funkciju rādītāji).

Grafiskās vizualizācijas rīks – Izstrādāt grafisko saskarni izsaukumu grafa un steka izmantojuma vizualizācijai, izmantojot rīkus kā Graphviz, kas atvieglotu sarežģītu programmu struktūras izpratni un kritisku ceļu identificēšanu.

IDE integrācija – Izveidot spraudņus populārām izstrādes vidēm (Arduino IDE, PlatformIO, Atmel Studio), kas ļautu veikt steka analīzi tieši no izstrādes vides un saņemt brīdinājumus reāllaikā par potenciālām steka problēmām.

Savstarpējās rekursijas atbalsts – Paplašināt rekursijas analīzes algoritmu, lai atbalstītu savstarpējās rekursijas gadījumus, kur funkcija *A* izsauc funkciju *B*, kas savukārt izsauc funkciju *A*, izmantojot ciklu detektēšanas algoritmus izsaukumu grafā.

Konfigurējamu drošības rezervju sistēma – Ieviest elastīgu drošības rezervju sistēmu, kas ļautu lietotājiem definēt dažādus drošības līmeņus atkarībā no lietojumprogrammas kritiskuma (piemēram, 5% sadzīves ierīcēm, 20% medicīnas iekārtām).

Atvērtā koda kopienas veidošana – Aktīvi popularizēt izstrādāto rīku AVR izstrādātāju kopienā, veidojot dokumentāciju, piemērus un apmācību materiālus, lai veicinātu rīka plašāku izmantošanu un kopienas ieguldījumu tā attīstībā.

# IZMANTOTĀS LITERATŪRAS UN AVOTU SARAKSTS

[1] Microchip Technology Inc. (2021). *AVR® Instruction Set Manual*. [Tiešsaistē]. Pieejams: <https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf> [Skatīts Maijs 21, 2025]

[2] myrobot.ru. *УСТРОЙСТВО МИКРОКОНТРОЛЛЕРОВ AVR*. [Tiešsaistē]. Pieejams: <https://myrobot.ru/stepbystep/mc_architecture.php> [Skatīts Maijs 21, 2025]

[3] Dave Cherry (2018, Sept. 10). *How Arduino AVR memory model works - for AVR*. [Tiešsaistē]. Pieejams: <https://www.thecoderscorner.com/electronics/microcontrollers/efficiency/how-arduino-avr-memory-model-works/> [Skatīts Maijs 21, 2025]

[4] Microchip Technology Inc. (2023, Nov. 9). *AVR® Memory*. [Tiešsaistē]. Pieejams: <https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/memory/> [Skatīts Maijs 21, 2025]

[5] embedded.com. (2015, Janv. 12). *Optimizing data memory utilization*. [Tiešsaistē]. Pieejams: <https://www.embedded.com/optimizing-data-memory-utilization/> [Skatīts Maijs 21, 2025]

[6] A. K. Das. (2021, Apr. 4). *AVR Architecture : Arduino / ATmega328p*. [Tiešsaistē]. Pieejams: <https://www.arnabkumardas.com/arduino-tutorial/avr-architecture/> [Skatīts Maijs 21, 2025]

[7] A. K. Das. (2021, Apr. 22). *AVR Memory Architecture : Arduino / ATmega328p*. [Tiešsaistē]. Pieejams: <https://www.arnabkumardas.com/arduino-tutorial/avr-memory-architecture/> [Skatīts Maijs 21, 2025]

[8] Microchip Technology Inc. (2018). *tinyAVR® Data Sheet*. [Tiešsaistē]. Pieejams: <https://ww1.microchip.com/downloads/en/DeviceDoc/ATtiny4-5-9-10-Data-Sheet-DS40002060A.pdf> [Skatīts Maijs 21, 2025]

[9] Atmel Corporation. (2008). *AVR XMEGA*. [Tiešsaistē]. Pieejams: <https://ww1.microchip.com/downloads/en/DeviceDoc/doc7925.pdf> [Skatīts Maijs 21, 2025]

[10] Kanda.com. (2023. Marts 21). *All You Need to Know About AVR ISP, UPDI, JTAG, PDI and TPI*. [Tiešsaistē]. Pieejams: <https://www.kanda.com/blog/microcontrollers/all-you-need-to-know-about-avr-isp-updi-jtag-pdi-and-tpi/> [Skatīts Maijs 21, 2025]

[11] Microchip Technology Inc. (2017). *AVR 8-Bit Microcontroller*. [Tiešsaistē]. Pieejams: <https://ww1.microchip.com/downloads/en/DeviceDoc/en590320.pdf> [Skatīts Maijs 21, 2025]

[12] A. Allain. *The Stack Data Structure in C and C++*. [Tiešsaistē]. Pieejams: <https://www.cprogramming.com/tutorial/computersciencetheory/stack.html> [Skatīts Maijs 21, 2025]

[13] M. Barr and A. Massa, *Programming Embedded Systems, with C and GNU Development Tools*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2006. [Tiešsaistē]. Pieejams: <https://www.bogotobogo.com/cplusplus/files/embed/OReilly_Programming_Embedded_Systems_Second_edition_ebook.pdf> [Skatīts Maijs 21, 2025]

[14] Scott Thornton (2016, Nov. 14). *Avoiding stack overflow in embedded processors*. [Tiešsaistē]. Pieejams: <https://www.microcontrollertips.com/faq-avoiding-stack-overflow-embedded-processors/> [Skatīts Maijs 21, 2025]

[15] J. Ganssle, *The Art of Designing Embedded Systems*, Boston: Newnes, 1999. [Tiešsaistē]. Pieejams: <https://theswissbay.ch/pdf/Gentoomen%20Library/Misc/Art%20of%20Designing%20Embedded%20Systems~tqw~_darksiderg.pdf> [Skatīts Maijs 21, 2025]

# PIELIKUMI

1. **pielikums**

AVR stekatmiņas izmantošanas analizatora pirmkods un testa faili.

Repozitorija adrese:   
<https://github.com/MrAnaKol/avr-static-stack-analyzer-Bakalaura-Darbs>

# 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, 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 \_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_