Универзитет у Београду

Математички факултет

Владимир Кузмановић

µC/OS2 оперативни систем за рад у реалном времену за ARM Cortex M3 микроархитектуру и његова примена у контроли дигиталне црне кутије

Мастер рад

Септембар 2016.

Ментор:	др Мирослав Марић, Математички факултет, Универзитет у Београду
Чланови комисије:	др Миодраг Живковић, Математички факултет, Универзитет у Београду
	др Филип Марић, Математички факултет, Универзитет у Београду
Датум одбране:	______________________________

Садржај

1. Увод 1
2. Опис изабране архитектуре хардвера 2
   2.1. NXP LPC1768FBD144 2
   2.2. Cypress Perform FM25H20 2Mb Serial 3V F-RAM Memory 3
3. Опис комуникационих протокола 4
   3.1. Универзални Асинхрони Рисивер/Трансмитер (UART) 4
      3.1.1. Пренос података 4
      3.1.2. Формирање пакета 5
      3.1.3. Рисивер 5
      3.1.4. Трансмитер 6
      3.1.5. Употреба 6
      3.1.6. Структура 6
   3.2. Директан приступ меморији (DMA) 7
      3.2.1. Принцип рада 7
      3.2.2. Режими рада 8
   3.3. Серијски периферни интерфејс (SPI) 9
      3.3.1. Принцип рада 10
      3.3.2. Поларност и фаза сата 11
   3.4. Општи улаз/излаз (GPIO) 12
4. Оперативни систем на платформи ARM Cortex M3 13
   4.1. Заштита критичних секција 13
   4.2. Прекиди 15
      4.2.1. Функција за обраду ресет прекида 16
   4.3. Бројање циклуса 17
      4.3.1. Прекид системског сата 18
      4.3.2. Промена контекста 20
      4.3.3. Функција за обраду софтверског прекида 20
   4.4. Екстерни прекиди 23
      4.4.1. Функција за обраду екстерних прекида 26
   4.5. Асемблерски старт фајл 26
5. Струткура језгра оперативног система 27
   5.1. Послови 27

5.1.1. Контролни блок посла ... 27
5.2. Креирање послова ... 28
5.3. Стања послова ... 28
5.4. Листа спремних послова ... 30
5.5. Распоређивање послова ... 30
5.6. Мерење времена ... 31
6. Синхронзација ... 32
6.1. Семафори ... 32
6.1.1. Креирање семафора ... 32
6.1.2. Брисање семафора ... 32
6.1.3. Чекање на семафор ... 32
6.1.4. Сигнализирање семафора ... 33
6.2. Катанци ... 33
6.2.1. Креирање катанца ... 34
6.2.2. Уклањање катанца ... 35
6.2.3. Чекања на катанац ... 35
6.2.4. Сигнализирање катанца ... 35
6.3. Критичне секције ... 36
7. Покретање оперативног система ... 38
8. Развој програма ... 39
8.1. Проверавање хардвера ... 39
8.1.1. Општа структура угнежденог програма ... 40
8.1.2. Први програм ... 41
8.2. UART периферијски уређај ... 42
8.2.1. UART ехо сервер ... 42
8.2.2. Синхронизација послова ... 47
8.2.3. Команде за конфигурисање серисјке везе ... 49
8.2.4. Динамичко конфигурисање дужине UART пакета ... 52
8.3. Меморија ... 53
8.3.1. Протокол за комуникацију са меморијом ... 55
8.3.2. Имплементација ... 58
8.4. Надгледање напајања ... 70
8.4.1. Пакет за упис у меморију ... 71

8.4.2. Прекид или прозивање уређаја 72
8.5. Структура, организација и синхронизација послова 74
8.5.1. Покретање оперативног система 74
8.5.2. Дефиниција свих послова 75
8.5.3. Синхронизација послова и контрола извршавања 79
9. Употреба 80
10. Резултати из екперимената 82
11. Идеје за унапређивање система 90
12. Закључак 91
13. Додатак А 93
13.1. Тестирање комуникације 93
13.2. Конфигурисање уређаја 93
13.3. Тестирање меморије 94
13.4. Подешавање напајања 95
13.5. Тумачење примљених података 95
13.6. Статистички подаци 96
Литература 98

1. Увод

Сваком развојном бироу који се приликом развоја уређаја бави и његовим активним испитивањем, потребан је независни уређај који ће истраживачима омогућити да сниме све неопходне податке са сензора у реалном времену. Понекад је овај захтев врло лако испунити уколико уређај који се тестира има неки једноставан интерфејс за комуникацију са спољним светом или уколико нису потребни специјални услови за извођење експеримента. Међутим, уколико се на пример испитује аеродинамика дизајнираног уређаја у аеро-тунелу или понашање пројектила у лету или уколико је у питању било који други изузетно скуп експеримент јасно је да ослањање на стандардне комуникационе и контролне механизме није довољно. У тим ситуацијама јавља се потреба за уређајем, тј. црном кутијом која ће бити део самог екперимента и која ће имати улогу да независно од спољног света забележи и трајно запамти све податке о променама које се током испитивања измере мерним уређајима унутар самог испитиваног објекта. Памћење података о променама директно са сензора је кључно, јер омогућава истраживачима да накнадно рекреирају и симулацијама провере сваки сегмент извршеног експеримента, што значајно смањује развојне трошкове. Тачније, у терминима авио-индустрије, наведеном типу развојних тимова је потребна својеврсна црна кутија опште намене.

Потребно је направити једноставну, јефтину и прилагодљиву црну кутију која је довољно робусна да учествује у свакој врсти испитивања и која је способна да у сваком тренутку и сваком експерименту запамти све оно што је истраживачима потребно. Црна кутија на себи мора да има неки облик трајне меморије довољне величине да запамти комплетан експеримент, као и могућност контроле читања и уписа података у сопствену меморију. Црна кутија мора да има и могућност контроле и тестирања уз помоћ рачунара, као и одређену дозу отпорности на уништење. Поред тога, у случају да се тестирани објекат састоји из више одвојених контролних целина потребно је да црна кутија у сваком тренутку памти информације о напајању целог система да би тумачење добијених резултата било лакше.

Имајући у виду наведене захтеве, потребно је дизајнирати хардвер који то све испуњава али и развити одговарајући софтвер за саму црну кутију, контролни софтвер за десктоп или лаптоп рачунар којим би се тестирала црна кутија и интерфејс за графичко приказивање добијених резултата. С обзиром да је цена један од приоритета, ARM Cortex M микроархитектура је изабрана као хардверско решење, као и најмање две одвојене меморијске јединице да би се у што већој мери сниљеним подацима осигурало преживљавање експеримента и добијање употребљивих података за каснију анализу. Предмет овог мастер рада је прилагођавање µC/OS2 оперативног система већ дизајнираној црној кутији која је заснована на ARM Cortex M3 микроархитектури, затим писање потребних управљачких програма и контролног програма за црну кутију. Додатно, у раду ће бити приказани резултати из реалне примене црне кутије, као и десктоп апликација која се користи за тестирање црне кутије и визуелизацију снимљених података.

2. Опис изабране архитектуре хардвера

ARM Cortex M је серија функционално прилагодљивих (енг. scalable), међусобно компатибилних, енергетски ефикасних микроконтролера лаких за употребу који су дизајнирани са циљем да омогуће програмерима да што лакше одговоре на изазове које пред њих постављају паметне и повезане угнеждене апликације како данашњице тако и у будућности. Неки од изазова укључују испоруку више функција по нижој цени, повећање повезивости, повећање искористивости изворног кода као и унапређење енергетске ефикасности [10].

ARM Cortex M породица процесора је оптимизована за употребу микроконтролера у условима у којима су примарни цена и потрошња енергије, као и у условима помешаних сигнала попут интернета ствари (енг. Internet of Things), контроле мотора, паметних мерења, уређаја за интеракцију са људима, аутомобилској и наменској индустрији, кућним и медицинским уређајима, као и уређајима широке потрошње.

С обзиром на осетљиву природу података и потребу за великом брзином уписа и великом поузданошћу, за трајну меморију је изабран FRAM (енг. ferroelectric RAM). FRAM је меморија са случајним приступом која је данас највећи конкурент стандардној флеш меморији у домену трајног складиштења података. Предности FRAM-а у односу на флеш меморије су мања потрошња енергије, знатно веће брзине уписа и многоструко већи број циклуса уписа и читања. Мане FRAM-а су много мања густина записа података у односу на флеш уређаје, ограничен максимални капацитет, као и већа цена [7].

За комуникацију са спољним светом изабрана је серијска RS-232 веза која је у ARM Cortex M микроархитектури реализована кроз UART периферијски уређај. Својства изабране меморије налажу потребу за серијском комуникацијом помоћу SPI (енг. Serial Peripheral Interface) протокола, а захтев за надгледањем напајања система налаже барем десетак слободних општих улазно/излазних пинова. Мањи број пинова не би задовољио захтев прилагодљивости самог система. У складу са наведеним захтевима изабран је ARM Cortex M3 процесор као лако доступан и као процесор који има велику базу корисника и произвођача.

ARM Cortex M3 је водећи 32-битни процесор за високо детерминистичке (енг. higly deterministic) апликације у реалном времену, тј. за апликације којима је кључно конзистентно извршавање операција у унапред задатом времену. Процесор је развијен са циљем да се добију јефтине платформе високих перформанси за широк дијапазон уређаја попут микроконтролера, сензора, разних система у аутомобилској и наменској индустрији, као и уређаја за бежично умрежавање. ARM Cortex M3 је процесор који има харвардску архитектуру са одвојеним магистралама за локалне инструкције и податке, као и додатну трећу магистралу за периферијске уређаје. ARM Cortex M3 је RISC процесор са уграђеним хардверским 32-битним множачем са 32-битним или 64-битним резултатом, као и 32-битним хардверским делитељем. Такође процесор има уграђену FPU (енг. Floating Point Unit) јединицу, као и максималних 240 физичких прекида (енг. interrupt) са или без опције маскирања. Опционо, процесор има и јединицу за заштиту меморије (енг. memory protection unit, MMU). Детаљније информације потражити у [11].

2.1. NXP LPC1768FBD144

У мору различитих верзија ARM Cortex M3 процесора изабрана је NXP LPC1768FBD144 верзија као најраспрострањенија и на тржишту најдоступнија. NXP LPC17xx серија микроконтролера је базирана

на ARM Cortex M3 архитектури и намењена је угнежденим (енг. embedded) апликацијама које захтевају висок ниво интеграције уређаја као и малу потрошњу енергије, тј. минималну дисипацију топлоте.

Периферијски уређаји који су уграђени у ову верзију ARM Cortex M3 процесора обухватају 512kB програмске флеш меморије, до 64kB меморије за податке, Ethernet MAC, USB интерфејс, 8-канални универзални DMA контролер, 4 UART периферијска уређаја, 2 CAN канала, 2 SSP контролера и подршку за SPI интерфејс, 4 тајмера опште намене као и до 70 општих улазно/излазних пинова. Очигледно ова верзија микроконтролера испуњава све наведене захтеве. За детаље погледати [8].

2.2. Cypress Perform FM25H20 2Mb Serial 3V F-RAM Memory

FM25H20 је трајна меморија капацитета 2Mb базирана на FRAM технологији [6]. FRAM је трајна меморија која извршава читање и писање на исти начин као што то ради и стандардни RAM. FRAM омогућава поуздано чување података у трајању до 10 година, при чему елиминише комплексности и проблеме са поузданошћу на системском нивоу које прате серијске флеш и остале сличне трајне меморије.

За разлику од серисјке флеш меморије, FM25H20 врши упис у меморију брзином једнаком брзини магистрале. Не постоји кашњење приликом уписа. Подаци се уписују у меморију одмах након што су пренети магистралом. Следећи циклус може да почне без икакве потребе за прозивањем уређаја (енг. polling). Наведена својста чине FM25H20 идеалним за употребу у условима у којима су неопходни чести и брзи уписи у меморију као и ниска потрошња енергије. Примери овога иду од скупљања података са сензора где број циклуса уписа постаје критичан, па све до захтевних индустријских контролних уређаја где дугачко време уписа стандардних серијских флеш меморија може да доведе до губитка података. FM25H20 користи врло брзи серијски SPI интерфејс за пренос података у меморију и из ње.

Меморија је организована као матрица димензија 262,144 x 8 и приступа јој се уз помоћ серијског SPI протокола (енг. Serial Peripheral Interface). Приликом приступа меморији, корисник адресира 262,144 адресе при чему свака има 8 битова. Ови битови се серијски шифтују. Адресама се приступа уз помоћ серијског SPI протокола, који укључује и линију за селектовање подређених уређаја (енг. chip/slave select) чиме се омогућава присуство већег броја меморијских уређаја на магистрали којима се командује операционим кодом и тробајтном адресом. Комплетна адреса од 18 битова једнозначно одређује сваку меморијску ћелију у меморији.

Највећи део функционалности меморије су контролисане или уз помоћ серијског SPI протокола или су аутоматски контролисане од стране уграђених електричних кола у самој меморији. Време приступа меморијским операцијама је практично нула, тј. меморија се чита или се у меморију уписује брзином саме SPI магистрале. Не постоји потреба за прозивањем уређаја до испуњења услова спремности, јер се уписи дешавају брзином на којој ради магистрала. Стога, до времена када је могуће избацити нову трансакцију преко магистрале, уписивање у меморију ће бити завршено. Поред очигледних добитака у брзини, мање очигледан је и добитак у робусности у самом процесу преноса података. У условима високог шума, велика брзина уписа података у меморију смањује могућност корупције података. Више детаља потражити у [6].

3. Опис комуникационих протокола

Комуникација са спољним светом омогућена је уз помоћ серијске RS-232 везе која је омогућена кроз UART периферијски уређај у самом процесору. Серијски SPI интерфејс се користи као механизам комуникације са трајном меморијом уграђеном у црну кутију. Додатни облик комуникације у систему је и надгледање напајања омогућено кроз опште улазно/излазне пинове (енг. GPIO – General Purpose Input Output). У наредних неколико редова биће описани употребљени протоколи.

3.1. Универзални Асинхрони Рисивер/Трансмитер (UART)

Универзални асинхрони рисивер/трансмитер, скраћено UART је део рачунарског хардвера који преводи податке између паралелене и серијске форме. UART се најчешће користи у комбинацији са неким комуникацијским стандардном као што је, на пример, RS-232. Ознака универзални означава да су формат података и брзина преноса конфигурабилни.

UART је обично одвојено интегрисано коло које се користи за комуникацију кроз серијски порт било рачунара било периферијског уређаја. Данас UART-и често налазе примену у свету микроконтролера.

3.1.1. Пренос података

На једној страни, UART узима бајтове и преноси одвојене битове секвенцијално. На другом крају, други UART периферијски уређај склапа те битове у комплетне бајтове. Сваки UART уређај садржи шифт регистар, који је основна метода конверзије између серијске и паралелне форме. Серијски пренос битова кроз једну жицу или неки други медијум је јефтинији него паралелни пренос кроз више жица [4].

Комуникација може да буде једносмерна (енг. simplex), двосмерна при чему оба UART уређаја примају и шаљу податке истовремено (енг. full duplex) или двосмерна али при чему уређаји наизменично шаљу и примају податке (енг. half duplex). Основна јединица трансфера података за UART уређај је увек један карактер, независно од логичке организације података. Логички, подаци који се шаљу серијским путем су увек организовани у пакете, тј. најчешће се зна шта је почетак а шта крај пакета, односно распоред и значење бајтова су унапред договорени. Својство UART периферисјког уређаја да ради само са карактерима, а не комплетним пакетима, омогућава да комуникација почне од било ког бајта, па је на програмеру да софтверски осигура синхроност података у оквиру пакета.

Брзина комуникације помоћу UART уређаја изражава се бодовном брзином. Бодовне брзине (енг. baud rate) које се користе за комуникацију два уређаја се најчешће крећу од неколико хиљада до неколико стотина хиљада бодова. Бод (енг. baud) у електроници представља број симбола (пулсева) који се могу пренети у једној секунди. Што је већа удаљеност између два уређаја користи се мања бодовна брзина. На пример, приликом комуникације са GPS (енг. Global Positioning system, систем за глобално позиционирање) антенама најчешће се користе брзине од 9600 бодова, док се за комуникацију два процесора серијским путем који су на истој штампаној плочи могу користити брзине и преко милион бодова уколико је штампана плоча задовољавајућег квалитета.

3.1.2. Формирање пакета

Када је линија празна, односно када нема података, она је увек под напоном [4]. Овакав приступ је преузет из телеграфије, где се линија увек држи под напоном да би се показало да су и линија и трансмитер неоштећени.

Свако слање карактера почиње са логичком нулом као почетни битом (енг. start bit), праћеном са 5 до 9 битова података, опционим битом парности уколико број битова података није 9 и на крају са једним или више завршних битова (логичких јединица). Број битова података је конфигурабилан и најчешће износи 8. Подаци се углавном шаљу LSB first методом, тј. најпре се шаље бит најмање тежине, редом прећен осталим битовима (слика 1.).

Редни број бита	1	2	3	4	5	6	7	8	9	10	11	12
Значење	старт бит	5-8 битова података								Податак или бит парности	стоп битови	
Вредност	Логичка нула	П0	П1	П2	П3	П4	П5	П6	П7	П8 или бит парности	Логичка јединица	

Слика 1. Изглед пакета приликом слања UART серијском везом

Почетни бит сигнализира рисиверу да нови карактер долази. Следећих 5 до 9 битова у зависности од изабране кодне шеме представљају сам карактер. Ако се користи бит парности, он долази након свих битова података. На крају, долазе један или два завршна бита (енг. stop bit) који су увек логичка јединица. Они сигнализирају рисиверу да је пренос карактера окончан. С обзиром на то да је почетни бит увек логичка нула, а завршни бит увек логичка јединица, то осигурава барем две промене сигнала између карактера. Уколико је линија у стању логичке нуле дуже од времена потребног за пренос једног карактера, UART уређај то може да открије и та ситуација се назива услов прекида (енг. break condition).

3.1.3. Рисивер

Све операције UART периферијског уређаја су контролисане сигналом сата који је умножак брзине преноса података. Најчешће је осам пута бржи. На сваком пулсу сата рисивер тестира стање долазног сигнала у потрази за почетним битом. Уколико почетни бит траје барем половину времена потребног за пренос једног бита, онда се прихвата као валидан и пренос карактера може да почне. У супротном, сматра се лажним пулсом и игнорише се. Након што сачека још једну дужину времена потребну за пренос бита, UART уређај поново узоркује (енг. sample) линију и прочитани ниво уписује у шифт регистар. Након проласка захтеваног броја периода за пренос битова који одговарају дужини карактера, садржај шифт регистра постаје доступан рисиверу у паралелном облику. UART уређај поставља заставицу (енг. flag) којом сигнализира рисиверу да је нови податак доступан, а може чак и да генерише прекид процесору чиме од њега захтева да изврши трансфер примљених података.

UART уређаји који међусобно комуницирају углавном немају заједничке сатове, осим сата комуникационог сигнала. Обично, UART уређаји ресинхронишу своје интерне сатове приликом сваке промене на линији података која се не сматра лажним пулсом. Ово омогућава поуздано примање података чак и у случајевима када трансмитер шаље податке на мало већој или мало мањој брзини од захтеване. Једноставни UART уређаји не раде овако, већ врше ресинхронизацију искључиво на падајућој ивици почетног бита, а затим читају центар (ниво) сваког бита податка

понаособ. Овакви системи раде уколико је брзина трансфера трансмитера довољно прецизна за поуздано узорковање (енг. sampling) почетног бита.

Стандардна карактеристика UART уређаја је да памте последњи примљени карактер док примају тренутни. Овај облик двоструког баферисања омогућава примајућем уређају да дохвати примљени карактер током читавог времена примања тренутног карактера. Многи UART уређаји имају мали FIFO бафер између шифт регистра рисивера и микроконтролера, чиме микроконтролер добија много више времена за дохватање и процесирање примљених података без ризика од губитка података на великим брзинама преноса.

3.1.4. Трансмитер

Слање података је много једноставнија операција од примања, јер уједначавање сигнала сата (енг. timing) не мора да се утврђује на основу стања линије нити је подешавање сигнала сата (енг. timing) везано за неке унапред фиксиране интервале. Чим систем који шаље упише податак у шифт регистар, након што се заврши слање претходног карактера, UART уређај генерише почетни бит, редом шифтује битове карактера, генерише и пошаље бит парности ако се користи и пошаље завршне битове. С обзиром на то да слање једног карактера може да траје јако дуго у поређењу са процесорским временом, UART уређај држи заставицу (енг. flag) којом процесору сигнализира заузетост како систем не би поставио нови податак у регистар све док се тренутни не пошаље. Спремност за слање наредног карактера може да се сигнализира процесору и механизмом прекида. Двосмерна комуникација захтева да се карактери истовремено шаљу и примају, па због тога UART уређаји имају одвојене шифт регистре за слање и примање карактера.

3.1.5. Употреба

UART уређај који шаље податке и UART уређај који прима податке морају да буду подешени за рад на истој бодовној брзини, истој дужини карактера, истом броју завршних битова као и евентуалној употреби битова парности да би уопште могли коректно да раде. У супротном, UART уређај који прима податке може да детектује поремећај у подешавањима и да постави грешку формирања пакета (енг. framing error) на самом систему.

Типични серијски портови на PC рачунарима који су повезани са модемима користе 8 битова за податке, без бита парности и један завршни бит. У овој конфигурацији, број ASCII карактера који се могу пренети у секунди је једнак брзини преноса подељеној са 10.

Многи јефтини PC рачунари и угнеждени системи не користе UART периферијске уређаје уопште, већ се ослањају на процерски сат да би узорковали (енг. sample) стање улазног порта и директно манипулишу са излазним портом да би вршили трансмисију. Иако је овај приступ изузетно процесорски захтеван, омогућава избацивање UART чипа чиме се смањује цена и површина штампаних плоча. Ова техника се назива bit-banging.

3.1.6. Структура

Из свега наведеног, очигледно је да сваки UART уређај мора да има следећу структуру:

- Генератор сата – најчешће умножак брзине преноса да би се омогућило узорковање нивоа.
- Улазни и излазни шифт регистар.

- Контрола примања/слања.
- Контрола читања/писања по регистрима.
- Бафери за слање/примање (опционо).
- Паралелна магистрала за податке (опционо).
- FIFO бафер меморија (опционо).

3.2. Директан приступ меморији (DMA)

Као што је напоменуто у опису функционисања UART уређаја, приликом слања или примања сваког карактера неопходна је интервенција процесора, што значајно смањује ефикасност целог система. Повећање ефикасности система може да се оствари уколико се непосредна контрола над процесом UART комуникације препусти DMA контролеру и тиме растерети сам централни процесор.

Директан приступ меморији (енг. Direct Memory Access, DMA) је могућност рачунарског система да дозволи одређеним хардверским подсистемима да директно приступају системској (РАМ) меморији независно од централног процесора [4]. Када процесор користи програмирани улаз/излаз без DMA, најчешће је у потпуности заузет све док трају операције читања или писања и због тога није у могућности да извршава друге послове (енг. task). Уколико се користи DMA, процесор прво иницијализује трансфер, и онда извршава неке друге послове све док трансфер траје, и када је трансфер коначно завршен DMA контролер механизмом прекида обавештава процесор о томе. Наведено својство DMA контролера је врло корисно када процесор не може да испрати брзину трансфера или када процесор треба да изврши неко израчунавање док чека да се заврши релативно спора улазно/излазна операција. DMA контролер најчешће може да контролише трансфере у три случаја:

- Меморија ка периферијском уређају (енг. Memory to peripheral) – означава слање података од процесора ка периферијском уређају, односно ка спољном свету.
- Периферијски уређај ка меморији (енг. Peripheral to memory) – означава примање података од периферијског уређаја, односно из спољног света.
- Меморија ка меморији (енг. Memory to memory) – користи се за брзо копирање блокова унутар саме РАМ меморије.

Напреднији DMA контролери имају и могућност рада у режиму Периферијски уређај ка периферијском уређају (енг. Peripheral to peripheral), чиме се додатно смањује потреба за укључивањем централног процесора у комуникационе процесе и тиме директно повећава искористивост процесора у друге сврхе. Више детаља потражити у [4].

3.2.1. Принцип рада

DMA контролер је у стању да генерише меморијске адресе као и да отпочне циклусе писања и читања. Сам контролер садржи неколико регистара којима централни процесор може да приступа, односно може у њих да уписује команде и да из њих чита стања контролера. Регистри обухватају меморијски адресни регистар, бројач бајтова, и један или више контролних регистара. Контролни регистри одређују који ће се улазно/излазни порт користити приликом комуникације, сам смер комуникације као и број бајтова који се преносе у једном налету (енг. burst).

Да би се извршио било какав трансфер уз помоћ DMA контролера, централни процесор мора да иницијализује сам контролер бројем речи које се шаљу као и меморијским адресама које ће се користити приликом трансфера. Централни процесор затим шаље команду периферним уређајима да би комуникација отпочела. DMA контролер онда самостално обезбеђује адресне и контролне линије за читање/писање по системској меморији. Сваки пут када је бајт спреман за трансфер између периферног уређаја и меморије, DMA контролер увећава свој интерни адресни регистар све док комплетан блок података не буде пренесен.

DMA трансфери могу да се догађају бајт по бајт или комлетан пакет одједном у режиму налета (енг. burst mode). Ако се ради о приступу бајт по бајт, онда је процесор у могућности да наизменично приступа системској меморији и тада се ради о крађи циклуса (енг. cycle stealing), јер се процесор и DMA контролер надмећу да би добили приступ меморији. У режиму налета, процесор може јако дуго да чека док се не оконча DMA трансфер и све док комплетан блок не буде примљен. Када су меморијски циклуси знатно бржи од процесорских, тада може да се користи и испреплетени DMA (енг. Interleaved DMA) у коме DMA контролер користи меморију док процесор није у могућности.

У bus-mastering системима, и процесору и периферијском уређају може да се додели контрола над меморијском магистралом. У случају да периферијски уређај добије контролу над меморијском магистралом, тада може директно да пише по меморији без икакве умешаности централног процесора, уколико обезбеди потребне меморијске адресе и контролне сигнале. У овом случају потребне су разне мере да би се процесор спречио да приступи магистрали да не би дошло до трке за ресурсима.

3.2.2. Режими рада

Режим налета (енг. burst mode) представља преношење комплетног блока података као један непрекидни низ. Када се DMA контролеру додели приступ системској магистрали од стране процесора, тада ће се сви бајтови у блоку пренети без враћања контроле процесору све до окончања комплетног трансфера. Иако се овај режим користи као механизам за заустављање нежељених података, доводи до дугачког чекања процесора у неактивном стању.

Режим крађе циклуса (енг. cycle stealing) се користи у системима у којима процесор не треба да буде неактиван током комплетног временског периода потребног за пренос блока података. У овом режиму, DMA контролер добија контролу над меморијском магистралом на исти начин као и у режиму налета, употребом BR (енг. Bus Request) и BG (енг. Bus Grant) сигнала. Ова два сигнала контролишу интерфејс између процесора и DMA контролера. Након преноса једног бајта података, контрола меморијске магистрале се враћа процесору уз помоћ BG сигнала. DMA контролер стално тражи контролу над магистралом уз помоћ BR сигнала и преноси само један бајт пре него што опет препусти контролу процесору. Овај процес се наставља све док се не пренесе комплетан блок података. Стално добијајући и препуштајући контролу над магистралом, DMA контролер ефективно преплиће трансфере инструкција и података. Процесор изврши инструкцију, онда DMA контролер пренесе један бајт податка, и тако редом. Са једне стране, блок података се не преноси толико брзо као у режиму налета, али са друге стране процесор у овом случају није толико беспослен. Режим крађе циклуса је врло користан за контролере који надгледају податке у реалном времену.

Транспарентни режим захтева највише времена за преношење блока података, али је ипак најефикаснији када су у питању укупне перформансе система. DMA контролер преноси податке

само онда када процесор извршава послове који не захтевају системску магистралу. Примарна предност овог приступа је у томе што процесор никада не прекида извршавање, а DMA контролер је слободан у контексту времена. Недостатак транспарентног режима је у томе што хардвер мора да утврди када процесор не користи магистралу, а то може да буде врло комплексно.

С обзиром да је серијска веза уз помоћ UART уређаја спора и прилично процесорски захтевна, природно је остварити комуникацију са спољним светом уз помоћ DMA контролера. Овим се добија знатна уштеда на процесорском времену потребном за комуникацију са спољним светом, али и могућност фокусирања на поуздано складиштење примљених података у екстерној меморији.

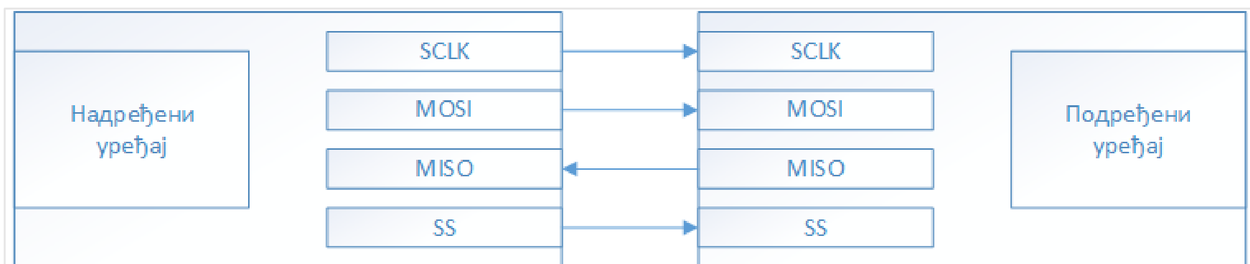
3.3. Серијски периферни интерфејс (SPI)

Серијски периферни интерфејс (енг. Serial Peripheral Interface, SPI) је синхрони серијски комуникациони интерфејс који се користи за комуникацију на малим растојањима, пре свега у свету угнеждених система (енг. embedded systems). Интерфејс је развила Моторола и углавном се користи за читање сензора, СД картица и контролу ЛЦД монитора.

SPI уређаји комуницарају у full-duplex режиму користећи надређени/подређени (енг. master-slave) архитектуру са једним надређеним. Надређени уређај генерише пакет за читање и писање, као и сигнал сата уз помоћ кога се синхронише сама комуникација. Вишеструки подређени уређаји су подржани кроз селекцију индивидуалним линијама за селекцију подређених уређаја (енг. clave select).

SPI интерфејс захтева четири логичка сигнала:

- SCLK – излаз из надређеног, серијски сигнал сата (енг. Serial clock).
- MOSI – излаз из надређеног, улаз у подређеног (енг. Master Out Slave In).
- MISO – излаз из подређеног, улаз у надређеног (енг. Master In Slave Out).
- SS – линија за селекцију подређеног, при чему је логичка нула активно стање (енг. Slave Select).



Слика 2. SPI интерфејс са једним надређеним и једним подређеним уређајем

Постоје различити називи и конвенције за ове сигнале, али у даљем тексту ће се користити искључиво наведени називи. MOSI/MISO конвенција захтева да се излаз из надређеног привеже на улаз подређеног и обратно. SS линија се користи уместо концепта адресирања. Логичка јединица се ретко када користи као ознака активног стања подређеног.

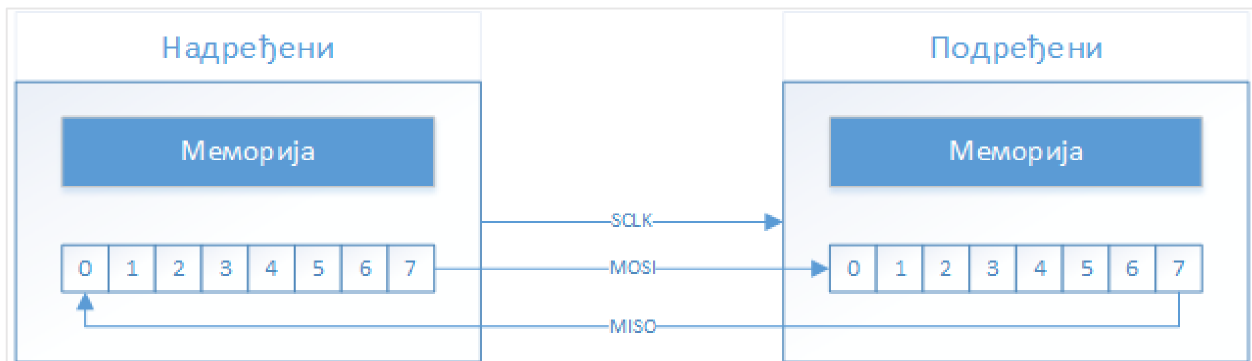
3.3.1. Принцип рада

SPI интерфејс може да функционише са једним надређеним и једним или више подређених уређаја. У случају да се ради о систему са већим бројем подређених уређаја, сваком уређају треба обезбедити одвојену SS линију.

Да би комуникација отпочела, надређени уређај конфигурише сат користећи фреквенцију подржану од стране подређеног уређаја, која уобичајено износи неколико мегахерца. Тада надређени уређај врши селекцију подређеног уређаја постављајући одговарајућу SS линију на логичку нулу. Уколико је захтеван неки период чекања, као када се ради о AD (енг. Analog to Digital) конверзији, надређени мора да сачека тај период пре него што крене да генерише сигнал сата на SCLK линији.

Током сваког циклуса сата, дешава се двосмерна комуникација. Надређени шаље бит преко MOSI линије и подређени га чита. Истовремено, подређени шаље бит преко MISO линије и надређени га чита. Овај начин рада се задржава чак и у случају једносмерне комуникације.

Трансмисија обично укључује два шифт регистра који су унапред задате величине, 8 битова на пример, један у надређеном уређају и други у подређеном уређају који су међусобно повезани топологијом виртуелног прстена (слика 3.).

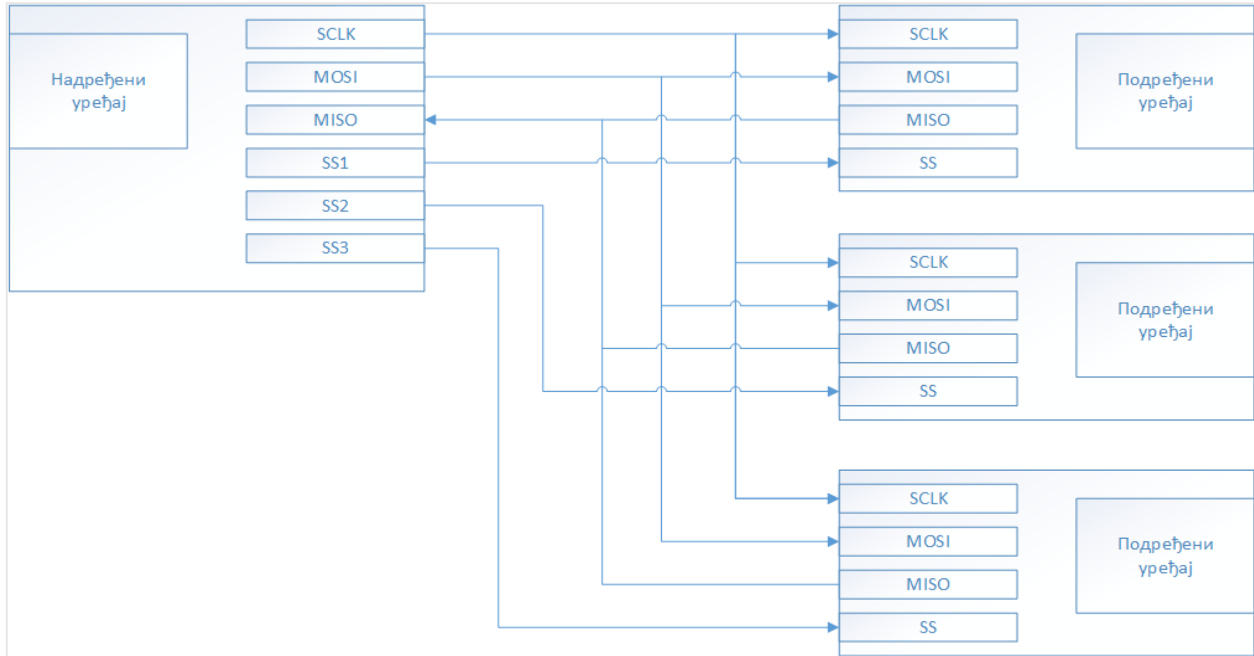


Слика 3. Шифт регистри повезани топологијом виртуелног прстена

Када је пренос података у питању, цифра највеће тежине се прва изшифтава док се истовремено у исти регистар ушифтава нова цифра на позицију најмање тежине. Када се комплетан регистар изшифтује, тада су надређени и подређени разменили вредности. Уколико постоји потреба за разменом веће количине података, регистри се поново пуне и поступак се понавља. Пренос података може да траје произвољан број циклуса сата. Када је пренос завршен, надређени престаје да генерише сигнал сата и поново подиже SS линију на логичку јединицу, чиме деселектује подређеног.

Пренос података се обично реализује речима дужине 8 бита, али нису ни остале величине реткост. Речи дужине 16 бита се користе за контролере екрана осетљивих на додир, а речи дужине 12 бита користе многи AD (енг. Analog to Digital) и DA (енг. Digital to Analog) конвертери.

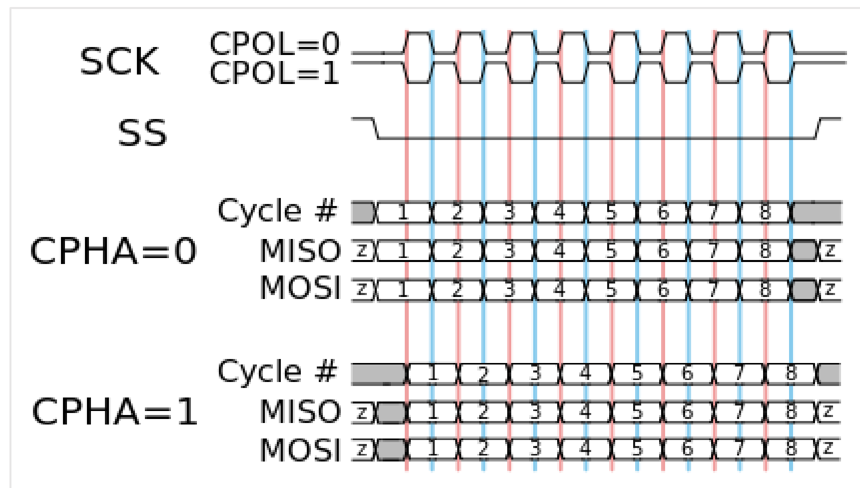
Сваки подређени уређај који није активиран употребом одговарајуће SS линије мора да игнорише улазни сигнал сата и податке на MOSI линији и не сме да шаље податке на MISO линију. У сваком тренутку највише један подређени уређај сме да буде активиран.



Слика 4. Повезивање већег броја подређених уређаја

3.3.2. Поларност и фаза сата

Поред одређивања фреквенције сата, надређени мора да конфигурише пол и фазу сата. Ове две опције се најчешће означавају као CPOL (енг. Clock Polarity) и CPHA (енг. Clock Phase). Дијаграм на слици 5 приказује однос између два наведена параметра.



Слика 5. Фаза и пол сата и утицај на комуникацију

Приказани дијаграм се односи и на подређени и на надређени уређај.

- Када је CPOL = 0, тада је основна вредност сата нула:
  a. Када је CPHA = 0, подаци се узоркују на растућој ивици сата, а пропагирају се на падајућој ивици сата.

    b. Када је CPHA = 1, подаци се узоркују на падајућој ивици сата, а пропагирају се на растућој ивици сата.
- Када је CPOL = 1, тада је основна вредност сата јединица (инверзија у односу на CPOL = 0):
    a. Када је CPHA = 0, подаци се узоркују на падајућој ивици сата, а пропагирају се на растућој ивици сата.
    b. Када је CPHA = 1, подаци се узоркују на растућој ивици сата, а пропагирају се на падајућој ивици сата.

Украткко, CPHA = 0 значи да се подаци узоркују на водећој (првој) ивици сата, док CPHA = 1 значи да се подаци узоркују на пратећој (другој) ивици сата, независно од тога да ли се ради о падајућој или растућој ивици. У случају да се ради о CPHA = 0, подаци морају да буду стабилни пола циклуса сата пре прве транзиције сата.

3.4. Општи улаз/излаз (GPIO)

Општи улаз/излаз (енг. General-Purpose Input Output) представља генеричке пинове на интегрисаном колу чије понашање корисник може да контролише током извршавања, независно од тога да ли се ради о улазним или излазним пиновима.

GPIO пинови немају дефинисану никакву специјалну улогу. Понекад систем интеграторима може да буде корисно да при руци имају неколико додатних дигиталних контролних линија, а с обзиром на то да постоје слободни и неискоришћени GPIO пинови на процесору, не јавља се потреба за увођењем додатног хардвера који би баш то омогућавао.

GPIO пинови могу током извршавања да се дефинишу као улазни или излазни, могу да се укључују и искључују по потреби, уколико су дефинисани као улазни могу се читати улазне вредности, а ако су излазни могу се постављати њихове вредности. Такође, у случају да су подешени као улазни, општи улазно излазни пинови могу да се користе и као извор екстерних прекида, на пример као сигнал за буђење уређаја.

Имплементације GPIO периферијског уређаја варирају од врло једноставних до прилично комплексних. GPIO периферијски уређај се понекад реализује само као група пинова који могу да се подешавају и користе искључиво као група, или су сви улазни или су сви излазни, док се у неким другим случајевима сваки појединачни пин може конфигурисати одвојено од осталих. GPIO порт представља одређени број GPIO пинова организованих у групу и који најчешће могу да се користе и контролишу само као група. Такве групе најчешће садрже осам пинова.

4. Оперативни систем на платформи ARM Cortex M3

Да би се систем прилагодио платформи ARM Cortex M3 неопходно је креирати све функционалности које омогућавају правилно искоришћавање ресурса које процесор поседује, тј све оне функционалности оперативног система које су уско везане за изабрану платформу, односно сам хардвер. Дакле, потребно је написати функције које омогућавају и онемогућавају прекиде као основни механизам заштите критичних секција. Оперативном систему је потребно омогућити приступ вектору прекида да би корисник могао лако у свом програму да користи све периферијске уређаје које су уграђене у процесор, али и да би могао да дефинише и сопствене софтверске прекиде.

Затим, оперативном систему мора да се омогући приступ бројачу циклуса јер је то једини начин којим време може да се мери поуздано и у еквидистантним корацима. Корисник нема никакву контролу над увећавањем бројача циклуса и управо због тога је то једини начин којим мерење протеклог времена може да се контролише безбедно и поуздано. Један циклус увек има фиксно временско трајање и након укључења увек креће од нуле.

Такође, неопходно је написати и драјвер за контролу системског сата (енг. SysTick) да би корисник могао да паузира послове на унапред дефинисано време ако за тим постоји потреба, али и да би сам оперативни систем могао правилно да извршава распоређивање постојећих послова. Направити разлику између бројача циклуса и системског сата је изузетно важно. Док бројач циклуса дозвољава читање укупног броја циклуса који су прошли без могућности прекидања и потпуно независно га увећава, основни механизам рада са системским сатом је управо заснован на механизму прекидања и потпуној контроли од стране корисника.

Да би се имлементирале наведене функционалности неопходно је познавање саме архитектуре као и распореда битова у контролним регистрима процесора. Основне функционалности ће бити написане у двопролазном armasm асемблеру док ће сложенији конструкти бити написани у C-у. Развојно окружење које ће се користити приликом израде програма за црну кутију је KEIL µVision v4.7.

4.1. Заштита критичних секција

Метод који је изабран за заштиту критичних секција је гашење прекида и чување тренутне вредности статус регистра у локалној променљивој, односно на стеку. Да би се описани поступак извео неопходне су четири помоћне асемблерске функције: `M3_SR_Save()`, `M3_SR_Restore()`, `M3_IntEn()` и `M3_IntDis()`.

Функције `M3_IntEn()` и `M3_IntDis()` редом омогућавају и онемогућавају све прекиде на процесору. Овакво понашање није нешто што одговара реалној употреби. Корисник након уласка и изласка из критичне секције очекује да активност и стања прекида које је изабрао остану непромењени. Проблем не настаје уласком у критичну секцију када се позива функција `M3_IntDis()` којом се гасе сви прекиди, већ приликом изласка из критичне секције позивом функције `M3_IntEn()` када ће сви могући прекиди на процесору бити омогућени. Овакав исход након критичне секције је недопустив и као решење се намеће чување вредности статус регистра приликом уласка у критичну секцију на стеку, пре онемогућавања свих прекида. Приликом изласка из критичне секције и пре омогућавања свих прекида, упамћено стање статус регистра пре уласка у критичну секцију се мора поново уписати у статусни регистар да би стање система пре уласка и

након изласка из критичне секције остало непромењено. Управо због свега наведеног дефинишу се још две асемблеркасе функције `M3_SR_Save()` и `M3_SR_Restore()` које користе PRIMASK регистар језгра процесора који садржи маску конфигурабилних прекида.

Након што се позове `M3_SR_Save()`, као повратна вредност се добија тренутна вредност у статус регистру. Позивом фунцкије `M3_SR_Restore()` се претходно упамћена вредност статус регистра на стеку уписује назад у статус регистар, чиме се ефективно враћа претходно стање прекида које је корисник дефинисао.

```
M3_IntDis
        CPSID   I             ; onemogucavaju se svi prekidi
        BX      LR            ; kontrola se vraca pozivajucoj funkciji

M3_IntEn
        CPSIE   I             ; omogucavaju se svi prekidi
        BX      LR            ; kontrola se vraca pozivajucoj funkciji

M3_SR_Save
        MRS     R0, PRIMASK ; vrednost prioritetne maske prekida se
                            ; kopira u registar R0, tj. postavlja se kao
                            ; povratna vrednost funkcije
        CPSID   I             ; onemogucavaju se svi prekidi
        BX      LR            ; kontrola se vraca pozivajucoj funkciji


M3_SR_Restore
        MSR     PRIMASK, R0 ; prosledjeni parametar se upisuje kao vrednost
                            ; prioritetne maske prekida
        BX      LR            ; kontrola se vraca pozivajucoj funkciji
```

Слика 6. Асембелрске функције за контролу прекида

У ARM асемблеру инструкција CPS (енг. Change Processor State) се користи за промену стања процесора, тј. врши измене у процесерском регистрру стања (CPSR). Користи се у комбинацији са једним од два ефекта, ID или IE. ID (енг. Interrupt Disable) означава гашење прекида, док IE (енг. Interrupt Enable) означава укључивање прекида. Аргумент I означава да се ради о прекидима. Постоје још аргументи F и A, који озаначавају да се ради о брзим прекидима и о прекидима изазваним грешком. BX (енг. Branch and Exchange) инструкција извршава скок на адресу уписану у аргуметну. LR је ознака за регистар у који је уписана адреса на коју треба да се врати извршавање након завршетка подпрограма, односно представља скраћеницу за линк регистар. MRS инструкција копира садржај изабраног статусног регистра у регистар опште намене. Регистар опште намене је први аргумент, а регистар процесора је други аргумент инструкције. MSR инструкција копира садржај регистра опште намене у изабрани статусни регистар. Статусни регистар је први аргумент инструкције, а регистар опште намене је други аргумент.

Да би корисник могао ефикасно да користи наведене асембелрске фунцкије дефинишу се следећи макрои:

```
/* makro inicijalizuje promenljivu za cuvanje
 * vrednosti statusnog registra na steku
 */
#define M3_SR_ALLOC()                M3_SR m3_sr = (M3_SR)0
/* makro onemogucava prekide i pamti stanje
 * prioritetne maske prekida na steku
 */
#define M3_INT_DIS()                 do { m3_sr = M3_SR_Save(); } while (0)
/* makro omogucava prekide i vraca zapamceno stanje
 * na steku u prioritetnu masku prekida
 */
#define M3_INT_EN()                  do {M3_SR_Restore(m3_sr); } while (0)
/* makro oznacava ulazak u kriticnu sekciju */
#define M3_CRITICAL_ENTER()          do ( M3_INT_DIS(); ) while (0)
/* makro oznacava izlazak iz kriticne sekcije */
#define M3_CRITICAL_EXIT()           do ( M3_INT_EN();   )   while (0)
```

Слика 7. Макрои за контролу прекида из програмског језика C

Уколико корисник жели да означи део кода као критичну секцију на почетку своје функције мора да алоцира место за памћење вредности статусног регистра на стеку позивом M3_SR_ALLOC(), и да жељени део кода уоквири позивима `M3_CRITICAL_ENTER()` и `M3_CRITICAL_EXIT()`. Макрои су уоквирени do-while блоком да би могли да се третирају и користе као обичне функције програмског језика C.

4.2. Прекиди

ARM Cortex M3 процесор поседује велики број уграђених периферијских уређаја који су доступни кориснику и за чије је успешно коришћење неопходан приступ вектору прекида. Готово сваки уграђени периферијски уређај представља посебан контролер или модул у оквиру самог процесора којем је доведено напајање и одговарајући сигнал сата и као такав својим радом ни на који начин не блокира процесор у извршавању других радњи. Сви уграђени периферијски уређаји су посвећени комуникацији са спољним светом и као такви вишеструко су спорији од самог процесора и у потпуности је неефикасно дозволити процесору да чека периферијски уређај да заврши са радом. Независно од тога да ли су протоколи синхрони или асинхрони, најчешћи модел употребе периферијских уређаја је следећи:

1. Сви подаци који се шаљу или примају се унапред припреме.
2. Омогући се прекидање за изабрани периферијски уређај.
3. Периферијски уређај се активира у неблокирајућем режиму.
4. По завршетку комуникације, периферијски уређај прекида процесор и обавештава га о завршетку слања или о пријему новог пакета.

Да би се ово постигло, оперативном систему се мора омогућити приступ вектору прекида да би прекиди за периферијске уређаје могли независно да се омогућавају и онемогућавају.

ARM Cortex M3 поседује вектор прекида који има 50 места [5]. Првих 16 су резервисани за системска стања, док осталих 34 представљају екстерне прекиде и посвећени су периферијским уређајима. Када су у питању резервисани односно интерни прекиди, потребна су нам само три. Прекид на позицији 2 је прекид који се активира након ресета процесора и његова једина улога је да покрене главни програм. Преостала два прекида су прекид системског сата који је на позицији 16 и прекид

на позицији 15 који ћемо користити за пребацивање контекста. Остали резервисани прекиди су везани за детекцију и обраду хардверских грешака и имплементирани су као бесконачне петље.

Свим екстерним прекидима је додељена само једна фунцкија за обраду у оквиру оперативног система да би се кориснику олакшао механизам употребе прекида. Олакшање се огледа у томе што корисник само треба да омогући или онемогући жељене прекиде, дефинише приоритет прекида, као и да очисти подигнуте заставице након сервисирања прекида без икаквог даљег удубљивања у начин функционисања прекида на нижем нивоу.

Распоред прекида у вектору се дефинише приликом компилације оперативног система у оквиру стартног фајла који мора бити испрограмиран на асемблеру. Стартни фајл контролише величину и место у меморији које ће бити додељено стеку и хипу, затим контролише распоред функција за обраду прекида у вектору прекида и дефинише функције за обраду интерних прекида. Стартни фајл је доступан уз сам процесор и само треба дефинисати распоред функција за обраду прекида.

4.2.1. Функција за обраду ресет прекида

Под ресетом процесора подразумева се покретање или изненадни губитак напајања. У оба наведена случаја једино што функција за обраду прекида треба да уради је да покрене извршавање main функције. Функција за обраду ресет прекида мора да буде написана у асемблеру.

```
; Reset Handler

Reset_Handler   PROC
                ; ime funkcije koje ce se koristiti prilikom linkovanje.
                ; [WEAK] oznacava da se funkcija nece koristiti
                ; ukoliko na sistemu postoji jos neka definicija funkcije
                ; Reset_Handler
                EXPORT  Reset_Handler [WEAK]
                ; uvozi se referenca na funkciju __main kojom se
                ; inicijalizuje procesor i pokrece izvrsavanje
                IMPORT  __main
                ; ucitava se adresa funkcije main u registar R0
                LDR     R0, =__main
                ; skace se na adresu upisanu u registar R0
                BX      R0
                ENDP
```

Слика 8. Функција за обраду ресет прекида

Након што се изврши функција за обраду ресет прекида и неопходна иницијализација процесора, контрола се препушта main функцији и оперативни систем креће са радом.

PROC директива у асемблеру се користи за дефинисање процедуре која у овом случају има назив `Reset_Handler`. EXPORT директива у асемблеру се користи за решавање референци на симболе у различитим објектним и библиотечким фајловима приликом линковања, тј. синоним је за GLOBAL директиву. Квалификатор `[WEAK]` означава да ће се ова имплементација користити приликом линковања само у случају да не постоји још нека имплементација функције `Reset_Handler` у систему. Директива `IMPORT` се користи приликом линковања симбола и референци који нису дефинисани у тренутном фајлу, тј. синоним је за EXTERN. LDR инструкција се користи за учитавање вредности у регистар. `__main` је библиотечки дефинисана функција која иницијализује извршавање

програма на процесору. Функција је задужена за копирање извршног кода из ЕЕПРОМ меморије процесора у радну меморију процесора, за нуловање меморијских региона додељених стеку и хипу, затим иницијализовање стека и хипа и препуштање контроле кориснички дефинисаној `main()` функцији. `__main` процедура је доступна у оквиру KEIL µVision развојног окружења и написана је од стране произвођача хардвера. Директива ENDP се користи као ознака краја имплементације процедуре.

4.3. Бројање циклуса

Да би се бројали циклуси приликом извршавања неопходно је активирати јединицу за надгледање података и контролу извршавања. Активирање поменуте јединице се врши уз помоћ следећег модула:

```
/* registar za kontrolu jedinice za nadgledanje podataka*/
#define M3_BSP_REG_DEMCR          (*(M3_REG32*)0xE000EDFC)
/* registar za resetovanje jedinice za nadgledanje podataka*/
#define M3_BSP_REG_DWT_CR         (*(M3_REG32 *)0xE0001000)
/* registar koji sadrzi broj ciklusa */
#define M3_BSP_REG_DWT_CYCCNT     (*(M3_REG32 *)0xE0001004)

#if (M3_CFG_TS_TMR_EN == DEF_ENABLED)
/* funkcija inicijalizuje brojac ciklusa */
void M3_TS_TmrInit(void) {

    M3_INT32U fclk_freq;    /* frekvencija procesora */

    /* cita se frekvencija na kojoj procesor radi */
    fclk_freq = CSP_PM_M3_ClkFreqGet();
    /* ukljucuje se jedinica za nadgledanje podataka */
    M3_BSP_REG_DEMCR |= DEF_BIT_24;
    /* resetuje se brojac ciklusa */
    M3_BSP_REG_DWT_CR = 0;
    /* ukljucuje se brojac ciklusa */
    M3_BSP_REG_DWT_CR |= DEF_BIT_00;
    /* inicijalizuje se frekvencija tajmera */
    M3_TS_TmrFreqSet((M3_TS_TMR_FREQ)fclk_freq);
}
#endif

#if (M3_CFG_TS_TMR_EN == DEF_ENABLED)
/* funkcija cita broj proteklih ciklusa */
M3_TS_TMR  M3_TS_TmrRd (void){

    M3_TS_TMR  ts_tmr_cnts; /* ukupan broj ciklusa */

    /* iz registra se cita broj ciklusa */
    ts_tmr_cnts  = (M3_TS_TMR)M3_BSP_REG_DWT_CYCCNT;

    /* broj ciklusa se vraca pozivajucoj funkciji */
    return (ts_tmr_cnts);
}
#endif
```

Слика 9. Модул за бројање циклуса

Приликом покретања оперативног система мора да се покрене и бројач циклуса позивом функције `M3_TS_TmrInit()` којом се отпочиње бројање циклуса које је кључно за мерење протеклог времена у оквиру оперативног система. Уколико корисник жели да одложи активирање посла на неко време неопходно је на неки начин мерити протекло време, јер у супротном није могуће поставити границу до које посао треба да чека. Најједноставнији начин за мерење протока времена од укључења је бројање циклуса процесора, које је прецизна мера протока времена и увек је еквидистантно.

Уколико корисник жели да паузира посао на неко време довољно је да прочита тренутни број циклуса, на тај број да дода онолико циклуса колико жели да сачека, да успава посао и да по истеку задатог броја циклуса поново активира посао.

4.3.1. Прекид системског сата

Пре дефинисања саме функције за обраду прекида неопходно је иницијализовати системски сат. Системски сат је најобичнији бројач који броји уназад заједно са бројачем циклуса. Након унапред задатог броја циклуса, тј. када вредност у регистру буде 0, системски сат подиже заставицу за прекид чиме захтева покретање функције за обраду прекида, поново поставља свој бројач на унапред дефинисану вредност и наставља са бројањем. Системски сат поседује уграђен бројач дужине 24 бита. За детаље погледати [5].

Да би се извршило подешавања системског сата, неопходни су следећи регистри:

```
/* kontorlni registar sistemskog sata */

#define  M3_REG_NVIC_ST_CTRL        (*((M3_REG32 *)(0xE000E010)))
/* registar odredjuje vrednost od koje krece brojanje unazad
 * upisana vrednost se ucitava u brojacki registar nakon
 * svakog prekida
 */
#define  M3_REG_NVIC_ST_RELOAD      (*((M3_REG32 *)(0xE000E014)))
/* registar sadrzi trenutnu vrednost u brojackom registru */
#define  M3_REG_NVIC_ST_CURRENT     (*((M3_REG32 *)(0xE000E018)))
/* registar sa fabrickim podacima o kalibraciji sistemskog
 * sata
 */
#define  M3_REG_NVIC_ST_CAL         (*((M3_REG32 *)(0xE000E01C)))
/* registar za podesavanje prioriteta prekida sistemskog sata
 * i prioriteta softverskog prekida
 */
#define  M3_REG_NVIC_SHPRI3         (*((M3_REG32 *)(0xE000ED20)))
```

Слика 10. Регистри за контролу системског сата

За подешавање функционисања системског сата користе се само контролни регистар, бројачки регистар и регистар за подешавање приоритета прекида. Прво се у бројачки регистар уписује број циклуса након којих се извршава прекид. Број циклуса се израчунава као целобројни количник фреквенције процесора и унапред дефинисане фреквенције којом корисник жели да му се јавља прекид сата. Добијени количник се умањује за 1 и уписује у регистар.

Затим, неопходно је дефинисати приоритет прекида системског сата. Подразумевани приоритет прекида је 0, тј. додељује му се највећи приоритет у систему дефинисан макроом

OS_CPU_CFG_SYSTICK_PRIO. Приоритет се подешава уз помоћ битова 24:31. Остали битови не смеју бити промењени. На крају се подешавају напајање и осцилатор за системски сат и активира се прекидање.

Паметно дефинисање фреквенције са којом се извршава функција за прекид системског сата је кључно за ефикасан рад оперативног система. Превисока фреквенција доводи до честих промена контекста, а премала фреквенција доводи до мале одзивности самог система [2].

```
/* Funkcija inicijalizuje i startuje sistemski sat */
void  OS_CPU_SysTickInit (M3_INT32U  cnts) {

    M3_INT32U  prio;     /* lokalana promenljiva u kojoj se
                          * cuva sadrzaj registra za podesavanje
                          * prioriteta
                          */

    /* broj ciklusa umanjen za 1 se upisuje
     * u brojacki registar
     */
    M3_REG_NVIC_ST_RELOAD = cnts - 1u;

    /* cita se trenutni prioritet iz registra */
    prio  = M3_REG_NVIC_SHPRI3;
    /* brise se prioritet prekida sistemskog sata */
    prio &= 0x00FFFFFF;
    /* upisuje se novi prioritet sistemskog sata */
    prio |= OS_CPU_CFG_SYSTICK_PRIO << 24;
    /* prioritet se upisuje u registar */
    M3_REG_NVIC_SHPRI3 = prio;

    /* kao oscilator se koristi klok procesora */
    M3_REG_NVIC_ST_CTRL |= 0x04;
    /* omogucava se prekid */
    M3_REG_NVIC_ST_CTRL |= 0x02;
    /* aktivira se sistemski sat */
    M3_REG_NVIC_ST_CTRL |= 0x01;
}
```

Слика 11. Функција за иницијализацију системског сата

Функција за обраду прекида системског сата није асемблерска функција, већ је написана у С-у. Функција се позива приликом сваког прекида системског сата и чита тренутни број циклуса процесора. Поред памћења системског времена, тј. укупног броја откуцаних циклуса, функција покреће и промену контекста уколико на систему постоји посао вишег приоритета у листи спремних послова. У случају да не постоји посао вишег приоритета извршавање прекинутог посла се неометано наставља без промене контекста.

Остаје још само да се дефинише активирање софтверског прекида. Као што је раније речено софтверском прекиду је додељено место 15 у вектору прекида. У оквиру регистра за контролу и стање прекида, софтверском прекиду је додељен бит на позицији 28. Постављање наведеног бита на 1, активираће се софтверски прекид и затражиће се извршавање функције за обраду прекида. Детаљније информације потражити у [5].

```
/* registar za kontrolu i stanje prekida */
#define NVIC_INT_CTRL   *((M3_REG32 *)0xE000ED04)
/* bit koji treba postaviti da bi se aktivirao
 * softverski prekid
 */
#define NVIC_PENDSVSET  0x10000000
/* makro koji aktivira softverski prekid na nivou posla */
#define OS_TASK_SW()    NVIC_INT_CTRL = NVIC_PENDSVSET
/* makro koji aktivira softverski prekid iz drugog prekida */
#define OSIntCtxSw()    NVIC_INT_CTRL = NVIC_PENDSVSET
```

Слика 12. Регистар за контролу прекида и активирање промене контекста

4.3.2. Промена контекста

Промена контекста и сама имплементација промене контекста је директно зависна од хардвера на којем се оперативни систем извршава. ARM Cortex M3 је процесор са ARM архитектуром што значи да има 16 регистара опште намене. Регистри имају имена R0 до R15 помоћу којих им се и приступа у асемблеру.

Регистар R0 се користи за враћање вредности из функција. Уколико је повратна вредност дужине 64 бита, виших 32 бита се враћају позивајућој функцији уз помоћ регистра R1. Регистри R4 до R11 припадају позивајућој функцији. Вредности наведених регистара се морају сачувати на стеку пре употребе и након употребе се морају вратити на почетне вредности. Регистри R0 – R3 и регистар R12 припадају позваној функцији и не морају се чувати пре употребе, јер их процесор сам чува на стеку аутоматски. Регистар R13 има специјалну улогу јер показује на врх стек оквира функције и као такав у систему не може имати другу улогу. Регистар R14 се најчешће назива линк регистар, јер је задужен за чување повратне адресе из подпрограма. Уколико се користи у програму, неопходно га је прво сачувати на стеку и вратити му упамћену вредност након коришћења, јер у супротном не постоји начин да се врати извршавање позивајућој функцији. Ако у програму постоји потреба за позивањем подпрограма, неопходно је сачувати вредност регистра R14 пре позива подпрограма и вратити му упамћену вредност након извршења подпрограма.

Промену контекста је могуће извршити само из оперативног система покретањем софтверског прекида. Прекид се покреће тако што се из софтвера вештачки подигне бит у контролном регистру који одговара позицији 15 у вектору прекида. Подизањем бита аутоматски ће се покренути извршавање функције за обраду прекида, а тиме и промена контекста.

4.3.3. Функција за обраду софтверског прекида

Извршавање функције за промену контекста има два независна случаја. Први случај се јавља само на укључењу, односно приликом прве промене контекста и тада нема потребе за памћењем претходног стања регистара. Други случај је свака следећа промена контекста када је неопходно упамтити и претходно стање регистара.

Псеудо код функције за обраду прекида је следећи:

1. Ако показивач стека има вредност 0, прескаче се чување стања регистара.
2. Чувају се регистри R4-R11 на стеку.

3. Показивач на врх стека се чува у оквиру тренутног посла
4. Уколико је корисник дефинисао функцију за проширење функционалности промене контекста, она се извршава.
5. Учитава се унапред одређени највиши приоритет.
6. Учитава се унапред одређени посао са највишим приоритетом.
7. Учитава се показивач на врх стека посла са највишим приоритетом.
8. Враћају се упамћене вредности регистара R4-R11 са учитаног стека.
9. Извршава се скок на учитани стек оквир којим ће се аутоматски вратити претходно стање осталих регистара.

Невођење рачуна о регистрима који нису поменути у псеудо коду је могуће, јер процесор аутоматски на стеку памти регистре R0-R3, R12 и R13-R15 приликом сервисирања било ког прекида, тј. пре уласка у фунцкију за обраду прекида. Тек по повратку контроле на прекинути контекст, вратиће се стања свих аутоматски упамћених регистара.

Функција за обраду прекида приликом промене контекста има најнижи приоритет у оквиру система да би се осигурало да је промена контекста могућа само онда када ниједан други прекид није активан, тј. да се промена контекста дешава само са једног оквира посла на други. Прекиди не користи стек оквире послова, већ системски стек у привилегованом режиму и покушај да се послу додели привелеговани стек оквир би довео до хардверске грешке и тиме заглављивања самог система.

```
OS_CPU_PendSVHandler
        CPSID   I                       ; onemogucavaju se svi prekidi
        MRS     R0, PSP                 ; sadrzaj pokazivaca na vrh steka
                                        ; se pamti u registru. ukoliko je
                                        ; vrednost PSP 0 radi se o sistemskom
                                        ; steku, tj. sistem je tek ukljucen i
                                        ; nema aktivnih poslova
        CBZ     R0, OS_CPU_PendSVHandler_nosave ; ukoliko je vrednost PSP 0
                                        ; skokom na
                                        ; labelu OS_CPU_PendSVHandler_nosave
                                        ; preskace se cuvanje sadrzaja registara
                                        ; na steku

        SUBS    R0, R0, #0x20           ; ukoliko PSP nije nula, vrh steka
                                        ; se umanje za 32
        STM     R0, {R4-R11}            ; pocevsi od adrese u R0 na steku, redom
                                        ; se cuvaju registri R4-R11
                                        ; nakon svakog postavljanja na stek
                                        ; adresa se automatski uvecava

        LDR     R1, =OSTCBCurPtr        ; u R1 se upisuje adresa promenljive
                                        ; iz C programa, tj. pokazivaca na
                                        ; trenutno aktivni posao
        LDR     R1, [R1]                ; u R1 se ucitava vrednost sa adrese
                                        ; koja je u R1 registru, tj. adresa
                                        ; okvira steka trenutno aktivnog posla
        STR     R0, [R1]                ; adresa iz R0 se upisuje kao vrednost
                                        ; na adresi koja je u R1, tj. pamti se
                                        ; trenutni okvir steka
```

```
OS_CPU_PendSVHandler_nosave

        PUSH    {R14}                ; na stek se postavlja vrednost
                                     ; link registra
        LDR     R0, =OSTaskSwHook    ; u registar R0 se upisuje adresa
                                     ; korisnicki definisane funkcije
        BLX     R0                   ; pokrece se izvrsavanje korisnikove
                                     ; funkcije za prosirivanje
                                     ; funkcionalnosti promene konteksta
        POP     {R14}                ; po povratku iz funkcije se vraca
                                     ; stara vrednost u link registar

        LDR     R0, =OSPrioCur       ; u registar R0 se ucitava adresa
                                     ; prioriteta trenutno aktivnog posla
        LDR     R1, =OSPrioHighRdy   ; u registar R1 se ucitava adresa
                                     ; prioriteta posla sa najvisim
                                     ; prioritetom
        LDRB    R2, [R1]             ; u R2 se ucitava vrednost sa adrese u
                                     ; R1
        STRB    R2, [R0]             ; na adresu iz registra R0 se upisuje
                                     ; trenutno najvisi prioritet

        LDR     R0, =OSTCBCurPtr     ; u R0 se ucitava adresa kontrolnog
                                     ; bloka trenutno aktivnog posla
        LDR     R1, =OSTCBHighRdyPtr ; u R1 se ucitava adresa kontrolnog
                                     ; bloka posla sa najvisim
                                     ; prioritetom
        LDR     R2, [R1]             ; u R2 se ucitava sadrzaj sa adrese u
                                     ; R1
        STR     R2, [R0]             ; u R2 se upisuje pokazivac na okvir
                                     ; steka kontrolnog bloka posla
                                     ; sa najvisim prioritetima

        LDR     R0, [R2]             ; u R0 se upisuje vrednost sa adrese
                                     ; koja je u R2
        LDM     R0, {R4-R11}         ; pocevsi od adrese na steku, redom
                                     ; se popunjavaju registri R4-R11
                                     ; tj. vraca se kontekst posla koji se
                                     ; aktivira
        ADDS    R0, R0, #0x20        ; vrednost u R0 se uvecava za 32
                                     ; tj. vrednost okvira steka
        MSR     PSP, R0              ; vrednost iz R0 se upisuje kao adresa
                                     ; okvira steka
        ORR     LR, LR, #0x04        ; osigurava se da posao ne koristi
                                     ; sistemski stek okvir
        CPSIE   I                    ; omogucavaju se prekidi
        BX      LR                   ; kontrola se vraca pozivajucoj
                                     ; funkciji

        END
```

Слика 13. Асемблерска функција за промену контекста

PSP у асемблеру означава показивач на врх стека тренутног стек оквира посла. CBZ (енг. Compare and Branch on Zero) инструкција означава условни скок на адресу записану у другом аргументу уколико је вредност првог аргумента нула. SUBS инстукција одузима вредност трећег аргумента од

вредности другог аргумента и резултат операције поставља у први аргумент уз освежавање заставица. ADDS ради на исти начин као SUBS инструкција, само је у питању сабирање. STM (енг. Store Multiple Registers) инструкција поставља листу регистара на стек. Први аргумент је адреса од које креће снимање, а други аргумент је листа регистара који се постављају на стек. LDM инстукција учитава вредности у листу регистра почевши од задате адресе. Адреса је први аргумент, а листа регистара је други аргумент инструкције. STR инструкција снима садржај регистра који је први аргумент на адресу у меморији која је други аргумент. LDR инструкција ради обрнуто, садржај из меморије снима у регистар. Уколико се на инструкције LDR и STR дода опција B, тада се уместо са 32-битним подацима користе 8-битни подаци. PUSH инструкција поставља вредности на стек, а POP инструкција скида вредности са стека.

Фунцкија за обраду софтверског прекида који изазива промену контекста се назива `OS_CPU_PendSVHandler()` и мора да се налази на месту 15 у вектору прекида.

4.4. Екстерни прекиди

ARM Cortex M3 поседује укупно 34 места за екстерне прекиде, при чему је свако место додељено тачно једном периферијском уређају [5]. По дефиницији, асемблерски фајл за распоређивање функција у оквиру вектора прекида дефинише одвојену функцију за сваки појединачни периферијски уређај. Такав приступ отежава и спушта програмирање на нижи ниво и захтева од корисника да у детаље познаје платформу на којој ради. Да би се толика захтевност избегла у оквиру оперативног система дефинише се само једна функција која ће се активирати приликом сваког прекида.

Да би овакво решење уопште имало смисла, неопходно је омогућити кориснику да сам дефинише своје функције за обраду прекида, да дефинише приоритета прекида као и да их по својој вољи омогућава или онемогућава. Сваки прекид може да има само једну фунцкију за обраду прекида и у сваком тренутку може да има само један приоритет. Током извршавања могуће је мењати приоритете прекида као и функције за обраду истих, али само у случају када је тај прекид онемогућен.

Очигледно, потребно је кориснику омогућити приступ регистрима процесора који контролишу прекидање као и измапирати комплетан вектор прекида у кориснику разумљива имена. У ту сврху користе се регистри на следећим адресама:

- 0xE000E100 – адреса првог регистра за омогућавање прекида,
- 0xE000E180 – адреса првог регистра за онемогућавање прекида,
- 0xE000E200 – адреса првог регистра за означавање да се прекид догодио,
- 0xE000E280 – адреса првог регистра за чишћење заставица који се подижу приликом прекидања,
- 0xE000E300 – адреса првог регистра који говори да ли се прекид тренутно догађа,
- 0xE000E400 – адреса првог регистра за подешавање приоритета прекида.

Сви регистри за интеракцију са прекидима долазе у паровима и 32-битни су. Први регистар је на наведеној адреси и свих 32 бита су му активни, док је други регистар на адреси већој за 4 у односу на први и активна су му само три најнижа бита, а остали битови су резервисани и недоступни кориснику. Да би се олакшала интеракција са регистрима сви прекиди су измапирани у

одговарајуће вредности и у коду се могу реферисати само именом. Мапа прекида се налази у `csp_grp.h` фајлу.

Број који је придружен прекиду одговара његовом положају у вектору прекида. Системски прекиди су индексирани негативном бројевима, док је првом екстерном прекиду придружен индекс 0. Припадност прекида одговарајућем регистру се лако утврђује целобројним дељењем индекса бројем 32. Уколико је резултат нула онда прекид припада првом регистру, у супротном ради се о припадности другом регистру. Бит који треба поставити или спустити се рачуна дељењем индекса са остатком бројем 32.

Сви битови у регистрима су активни када имају вредност један. Постављањем бита у сваком од наведених регистара промениће се стање прекида на одговарајући начин. Постављањем бита на нулу, неће се догодити ништа. Ако се у регистру на адреси 0xE000E100 постави бит на позицији 26 на један омогућава се прекидање DMA контролеру. Уколико је потребно искључивање прекидања на DMA контролеру, неопходно је поставити бит на позицији 26 у регистру на адреси 0xE000E180 на 1. На исти начин се понашају и регистри за означавање да се прекид догодио и регистри за чишћење заставица.

Регистар за праћење активности прекида дозвољава само читање вредности, никако упис. Покушај уписа у регистар доводи до недефинисаног понашања микроконтролера. Уколико је неки бит у регистру постављен на 1, то значи да је тај прекид тренутно активан.

Подешавање приоритета прекида је организовано уз помоћ девет 32 битних регистара, при чему је у сваком регистру могуће подесити приоритете за четири прекида. Сваком регистру су додељена четири прекида, а сваком приоритету 5 битова, што значи да је максимални приоритет нула, а минимални приоритет 31. Више детаља потражити у [5].

Свакој од наведених операција је придружен одговарајући макро ниског нивоа која оперативном систему омогућава интеракцију са вектором прекида. Макрои су следећи:

```
/* makro pristupa registru za omogucavanje prekida*/
#define  M3_REG_NVIC_SETEN(n)       (*((M3_REG32 *)(0xE000E100 + (n) * 4u)))
/* makro pristupa registru za onemogucavanje prekida */
#define  M3_REG_NVIC_CLREN(n)       (*((M3_REG32 *)(0xE000E180 + (n) * 4u)))
/* makro pristupa registru za aktiviranje prekida */
#define  M3_REG_NVIC_SETPEND(n)     (*((M3_REG32 *)(0xE000E200 + (n) * 4u)))
/* makro pristupa registru za resetovanje flagova prekida */
#define  M3_REG_NVIC_CLRPEND(n)     (*((M3_REG32 *)(0xE000E280 + (n) * 4u)))
/* makro pristupa registru za proveru aktivnosti prekida */
#define  M3_REG_NVIC_ACTIVE(n)      (*((M3_REG32 *)(0xE000E300 + (n) * 4u)))
/* makro pristupa registru za podesavanje prioriteta prekida*/
#define  M3_REG_NVIC_PRIO(n)        (*((M3_REG32 *)(0xE000E400 + (n) * 4u)))
```

Слика 14. Макрои за приступ регистрима за контролу екстерних прекида

Макрои се користе приликом дефинисања функција ниског нивоа за подешавање прекида на специфичном хардверу. Функције које оперативни систем може да користи су следеће:

```
/* funkcija onemogucava prekidanje periferijskom uredjaju
 * na poziciji pos u vektoru prekida
 */
void M3_IntSrcDis(M3_INT08U pos);
/* funkcija omogucava prekidanje periferijskom uredjaju
 * na poziciji pos u vektoru prekida
 */
void M3_IntSrcEn(M3_INT08U  pos);
/* funkcija resetuje flag koji signalizira da se prekid
 * dogodio
 */
void M3_IntSrcPendClr(M3_INT08U pos);
/* funkcija dodeljuje prioritet prekida prio prekidu
 * na poziciji pos u vektoru prekida
 */
M3_INT16S M3_IntSrcPrioGet(M3_INT08U pos);
/* funkcija vraca dodeljeni prioritet na poziciji pos
 * u vektoru prekida
 */
void M3_IntSrcPrioSet(M3_INT08U pos, M3_INT08U prio);
```

Слика 15. Фунцкије за контролу прекида на процесору

Као што је раније речено, у оквиру система постоји само једна глобална функција за обраду свих екстерних прекида, а сам процесор има више од 30 уграђених периферијских уређаја који могу да прекидају извршавање. Логично би било омогућити кориснику да за сваки од постојећих периферијских уређаја може самостално да дефинише функцију за обраду прекида, коју ће глобална рутина за обраду екстерних прекида у оквиру оперативног система позвати само онда када одговарајући периферијски уређај направи прекид.

Да би се то омогућило, у оквиру оперативног система се креира низ од 35 показивача на функције типа `void (*func)(void*)` које корисник сам може да иницијализује сопственим функцијама и на тај начин контролише обраду прекида. Приликом дефинисања функције за обраду прекида, корисник је дужан да чисти заставице које прекиди подижу.

У оквиру оперативног система дефинишу се омотачи за функције ниског нивоа уз неколико додатака. Поред основних функција за омогућавање и онемогућавање прекида дефинишу се и функције за иницијализацију низа показивача на функције, као и функције за регистровање и брисање кориснички дефинисаних функција из низа показивача.

Функције за омогућавање и онемогућавање прекида, брисање постављених заставица и подешавање приоритета заставица само позивају функције нижег нивоа и о њима неће бити речи.

Функција `CSP_IntInit()` иницијализује прекиде на систему. Псеудо код функције је следећи:

1. Онемогућити све екстерне прекиде.
2. Ресетовати заставице свих прекида.
3. Поставити све показиваче на функције за обраду прекида на NULL.

Функција `CSP_IntVectReg(CSP_DEV_NBR src_nbr, M3_FUNC_PTR isr_fnct)` поставља показивач на функцију `isr_fnct` у низ показивача на функције за обраду прекида на место `src_nbr`, чиме омогућава извршавање корисничке функције када се догоди прекид на периферијском уређају са индексом src_nbr у вектору прекида.

Функција `CSP_IntVectUnreg (CSP_DEV_NBR int_ctrl, CSP_DEV_NBR src_nbr)` брише показивач на функцију из низа показивача на функције за обраду прекида и тиме онемогућава извршавање корисникове функције када се прекид догоди.

4.4.1. Функција за обраду екстерних прекида

Приликом сваког прекида функција `CSP_IntHandler()` дефинисана у оквиру оперативног система као глобална функција за обраду свих екстерних прекида, мора да позове тачно ону функцију коју је корисник дефинисао као рутину за обраду прекида.

Функција за обраду екстерних прекида је најобичнија C функција која се позива када год се региструје неки екстерни прекид. Њена једина улога је да по регистровању прекида из таблице показивача на функције за обраду прекида дерефенцира одговарајућу функцију и изврши је.

4.5. Асемблерски старт фајл

Све до сада наведено није изводљиво уколико се не дефинише одговарајући старт фајл који ће повезати све наведене функције за обраду прекида са њиховим физичким местом у вектору прекида. Поред тога потребно је дефинисати и величину стека као и величину хипа који ће се користити приликом извршавања програма. Код за покретање процесора је наведен у `cstartup.s` фајлу.

5. Структура језгра оперативног система

Све функције које су до сада дефинисане су зависне од платформе на којој се извршавају и нису преносиве на друге архитектуре. Све што се дефинише у даљем току користи концепте који су платформски независни и омогућавају преносивост кода под условом да се за сваку нову платформу дефинишу све функције које су до сада дефинисане.

Језгро оперативног система мора да омогући подршку за постојање већег броја послова који се конкурентно извршавају као и механизме за њихову синхронизацију. Језгро оперативног система мора да омогући ефикасне механизме за [2]:

- Креирање и брисање послова (процеса),
- Управљање пословима (процесима),
- Комуникацију између послова (процеса),
- Синхронизацију послова (процеса).

5.1. Послови

Послови (процеси, енг. Task) су најчешће имплементирани у бесконачној петљи у оквиру неке функције, и представљају основну јединицу извршавања у оквиру оперативног система. У случају да се ради о послу који треба да се изврши једнократно, по извршењу корисник је дужан да тај процес избрише из листе процеса.

5.1.1. Контролни блок посла

Приликом креирања посла додељује му се контролни блок посла (енг. Task Control Block, TCB). Контролни блок посла је структура коју оперативни систем користи да би одржавао стање посла приликом распоређивања, тј. да би водио прецизну евиденцију о пословима, олакшао мултипрограмирање, чувао информације о свим покренутим пословима и омогућио идентификацију и управљање пословима [2]. Када посао поврати контролу над процесором, контролни блок посла омогућава да се извршавање настави управо тамо где је заустављено. Сви контролни блокови послова се налазе у RAM меморији процесора. Контролни блок посла има следећу структуру:

Табела 1. Опис контролног блока посла

Поље	Значење
StkPtr	Показивач на тренутни врх стек оквира
StkLimitPtr	Показивач на максималну величину стека
StkBasePtr	Показивач на почетну адресу стека
NextPtr	Показивач на следећи стек оквир у листи спремних
PrevPTr	Показивач на претходни стек оквир у листи спремних
TaskEntryAddr	Почетна адреса посла
TaskEntryArg	Адреса аргумента са којим се креира посао
PendDataTblPtr	Листа објеката на које посао чека
PendOn	Индикатор на шта посао чека
PendStatus	Статус чекања

PendDataTblEntries	Број објеката на које посао чека
TaskState	Тренутно стање у коме се посао налази
Prio	Приоритет посла
StkSize	Тренутна величина стека (број елемената)
Opt	Опције приликом креирања посла
TS	Временски печат
SemCtr	Бројачки семафор самог посла
TickCtrPRev	Претходно време када је посао био спреман
TickCtrMatch	Време када ће посао поново бити спреман
TickRemain	Преостало време до поновне спремности посла

Приликом иницијализације контролног блока посла, сва поља се иницијализују на нулу. Тек по креирању посла, поља у структури ће бити попуњена одговарајућим вредностима. Када се креира нови посао, неопходно је прво алоцирати меморију за контролни блок посла, затим иницијализовати поља контролног блока и тек тада је дозвољено креирање новог посла коме ће бити придружен алоцирани контролни блок. Приоритет посла је уједно и његов идентификатор.

5.2. Креирање послова

Оперативни систем подржава највише 18 различитих приоритета конкурентних послова. Сваком послу неопходно је доделити приоритет који је уједно и његов идентификатор. Мањи број означава већи приоритет и обрнуто, већи број означава мањи приоритет. Да би се посао креирао корисник мора да позове функцију `OSTaskCreate()` и да јој проследи све потребне параметре.

Псеудо код функције за креирање посла `OSTaskCreate()`:

1. Брише се комплетан садржај контролног блока посла.
2. Уколико је изабрана опција, брише се комплетан садржај у стек оквиру.
3. Одређује се граница до које стек може да расте у складу са смером раста стека.
4. Иницијализује се стек оквир посла и попуњавају се сва поља контролног блока посла.
5. Уписује се приоритет новокреираног посла у таблицу приоритета.
6. Увећава се глобални бројач послова.
7. Ако је оперативни систем у стању извршавања позива се распоређивач и врши се промена контекста.

Функција за креирање посла очекује од корисника да унапред алоцира потребан меморијски простор за складиштење контролног блока посла. Алоцирање може да буде статичко или динамичко. Детаљније информације се налазе у [3].

Приоритет посла треба одвојити од приоритета прекида. Приоретети посла су конструкти оперативног система и немају никакве везе са хардвером, док су приоритети прекида системска ствар и уско су везани за сам хардвер.

5.3. Стања послова

У оквиру оперативног система дефинише се пет могућих стања за сваки посао (енг. Task States): `TASK_DORMANT` (нови), `TASK_READY` (спреман), `TASK_RUNNING` (извршавање), `TASK_WAITING` (чекање) и `TASK_DELETED` (завршен) . За детаље погледати [2].

TASK_DORMANT стање имају послови који се налазе у програмској меморији процесора али нису доступни оперативном систему. Посао постаје доступан оперативном систему позивом функције OSTaskCreate() којом се оперативном систему саопштава почетна адреса изабраног посла. По завршетку извршавања ове функције, посао прелази у стање TASK_READY. Послови могу да се креирају пре почетка мултитаскинга или динамички од стране других послова приликом извршавања. Уколико се посао креира динамички и додели му се већи приоритет од креаторовог, новокреираном послу се моментално додељује контрола над процесором. Посао може да врати самог себе или било који други процес у стање TASK_DORMANT позивом функције OSTaskDel().

Псеудо код функције за брисање посла OSTaskDel():

1. Изабрани посао се уклања из листе којој је припадао, листе активних процеса или листе чекајућих.
2. Умањује се глобални бројач послова.
3. Брише се комплетан садржај контролног блока.
4. Стање посла се пребацује у TASK_DELETED.
5. Позива се распоређивач и врши се промена контекста.

Мултитаскинг се започиње позивом функције OSStart() која се позива само једном на почетку извршавања и покреће посао највишег приоритета који је креиран током иницијализације. Посао највишег приоритета ће бити у стању TASK_RUNNING. Само један процес може да буде у стању TASK_RUNNING у сваком тренутку извршавања. Посао из реда спремних неће прећи у стање извршавања све док сви послови вишег приоритета не буду премештени у стање чекања или обрисани.

Псеудокод функције OSStart() :

1. Проналази се и памти највиши приоритет посла из листе спремних послова.
2. Из листе спремних послова узима се посао са највишим приоритетом.
3. Тренутни контролни блок посла се поставља на блок посла са највишим приоритетом.
4. Стање оперативног система се пребацује у стање извршавања.
5. Покреће се посао са највишим приоритетом.

Посао који се тренутно извршава може да се нађе у стању чекања уколико треба да сачека да се неки ресурс ослободи позивом функција OSSemPend() или OSMutexPend(). У случају да се догађај није десио, посао који позове једну од ове две функције ће бити пребачен у стање чекања и остаће у том стању све док се догођај не деси. Када посао чека на неки догађај, послу са највећим приоритетом из реда спремних се моментално даје контрола над процесором. Посао прелази из стања чекања у стање спреман оног тренутка када се догађај од интереса деси. Испуњење услова може да буде сигнализирано од стране другог посла или од стране рутине за обраду прекида.

Посао који се извршава може да буде прекинут у сваком тренутку, сем у случају када сам посао или оперативни систем намерно искључе прекиде. У том случају посао улази у стање извршавања ISR_RUNNING. Када се догоди прекид, извршавање посла се зауставља и рутина за обраду прекида преузима контролу над процесором. Сама рутина може да пребаци један или више послова у стање спреман слањем сигнала пословима. У овом случају, пре него што се врати из рутине за обраду

прекида оперативни систем одређује посао са највишим приоритетом из реда спремних и њему препушта контролу над процесором.

Када су сви процеси у стању чекања, оперативни систем извршава празан (енг. idle) посао. Више информација потражити у [3].

5.4. Листа спремних послова

Спремни послови се не организују у једну листу, већ у низ редова. Низ је величине 18, јер је на систему дозвољено максимално 18 различитих приоритета послова. Сваком од приоритета, односно сваком месту у низу је додељен ред спремних послова. Поред ове листе у систему је дефинисана и таблица приоритета послова у којој се памти приоритет посла. У питању је таблица битова, а не бајтова. Таблица има битовске димензије (дужина кодне речи)x(((максимални број приоритета-1)/ дужина кодне речи) + 1). У овом случају величина таблице је један 32 битни податак.

Приликом дефинисања посла наводи се и његов приоритет који се уписује у таблицу приоритета на следећи начин:

1. Одређује се индекс положаја приоритета у таблици тако што се подели дужином кодне речи.
2. Одређује се број приоритета у таблици тако што се изврши битовска конјункција дужине кодне речи и изабраног приоритета.
3. Затим се одређује положај бита у таблици тако што се од дужине кодне речи одузме број приоритета из корака 2 увећан за један.
4. Приоритет се уписује у таблицу тако што се изврши битовска дисјункција броја из корака 3 и вредности из таблице на индексу из корака 1.

Брисање приоритета из таблице приоритета се врши само заменом корака 4 у односу на упис. Уместо дисјункције, ради се конјункција са инверзним бројем.

Одређивање највишег приоритета се ради врло лако. Поступак је следећи:

1. Приоритет се иницијализује на нулу.
2. Креће се од почетка таблице и итеративно се тражи први ненула елемент.
3. Након преласка на наредни елемент приоритет се увећава за 32.
4. Када се пронађе први ненула елемент, преброје се водеће нуле и додају се већ израчунатом приоритету.

Вредност добијена у кораку четири је индекс у низу реда послова на коме се налази ред послова највишег приооритета. Из реда послова на добијеном индексу се узима први посао, пребацује се у стање извршавања и уклања се из листе спремних.

5.5. Распоређивање послова

У сваком тренутку се извршава посао највишег приоритета. Одлучивање који посао има највећи приоритет, односно који ће се следећи извршавати ради распоређивач. Распоређивање на нивоу послова се ради уз помоћ функције `OSSched()`. Распоређивање на нивоу прекида се ради уз помоћ функције `OSIntExit()` о којој ће бити речи касније. Време распоређивања послова је константно, независно од броја послова у систему.

Псеудо код функције за распоређивање је следећи `OSSched()`:

1. Уколико се распоређивач позива из прекида, функција се враћа без промене контекста.
2. Уколико се распоређивач позива док је распоређивање угашено, функција се враћа без промене контекста.
3. Онемогућавају се сви прекиди.
4. Проналази се највиши приоритет посла.
5. Из реда спремних послова, извлачи се први посао са приоритетом из корака 4.
6. Уколико су тренутни посао и посао са највећим приоритетом исти, излази се из функције без промене контекста.
7. Уколико нису исти послови у питању, поставља се заставица за промену контекста.
8. Омогућавају се прекиди.

Као што промену контекста може да позове било који посао, на исти начин може да позове и услугу распоређивача и пребаци контролу послу са највишим приоритетом. Уколико је потребно позивање распоређивача након сервисирања прекида, тада се позива функција `OSIntExit()` која има идентичну функционалност као и функција `OSSched()`, али са детаљнијом провером угнежђености прекида.

Промена контекста се извршава тако што се прво на стеку посла који се суспендује упамте стања свих регистара, након чега се враћа претходно стање свих регистара посла са вишим приоритетом са припадајућег стека. У оквиру оперативног система, стек оквир спремног посла увек изгледа као да се прекид управо догодио и као да су сви регистри упамћени на њему.

У сваком прекиду системског сата се позива распоређивач да би процеси највишег приоритета увек имали контролу над процесором и да би се повећала одзивност система. Детаље потражити у [3].

5.6. Мерење времена

Као што је речено раније системски сат је кључан за правилно мерење времена у оквиру оперативног система. У сваком откуцају системског сата, односно приликом сервисирања прекида системског сата у листи послова који чекају да прође одлагање избацују се они чије је време чекања истекло. Избачени послови се проглашавају спремним и у случају да неки од спремних послова има виши приоритет од тренутно активног посла, извршава се промена контекста и препушта му се контрола над процесором.

6. Синхронзација

6.1. Семафори

Семафор [1] је механизам за синхронизацију више послова који користе исти ресурс. У оквиру оперативног ситема семафор је структура која има најмање два елемента, 16 битни неозначени цео број који представља бројач и листу послова који чекају да бројач семафора буде већи од 0. Оперативни систем подржава следеће четири операције са семафорима: `OSSemCreate()`, `OSSemPend()`, `OSSemPost()` и `OSSemDel()`.

6.1.1. Креирање семафора

Пре употребе неопходно је да се креира семафор позивом функције `OSSemCreate()` и да се проследи број којим се иницијализује бројач. Уколико се семафор користи као сигнал да се неки догођај десио бројач треба да се иницијализује на 0. У случају да се семафор користи за синхронизацију употребе неког од n доступних идентичних ресурса, бројач је неопходно иницијализовати бројем n и тада се ради о бројачком семафору.

Структура која представља семафор у оквиру оперативног система има следећа поља:

Табела 2. Опис структуре која представља семафор

Поље	Значење
Type	Тип објекта у оквиру оперативног система
NamePtr	Показивач на име семафора
PendList	Листа котролних блокова послова који чекају на семафору
Ctr	Бројач
TS	Временски печат последње промене

Приликом креирања семафора увећава се глобални бројач семафора у оквиру оперативног система.

6.1.2. Брисање семафора

Приликом брисања семафора, неопходно је да се претходно обришу сви послови који зависе од наведеног семафора како ниједан посао не би могао случајно да покуша да приступи непостојећем семафору.

Псеудо код функције за брисање семафора је следећи:

1. Уколико број контролних блокова послова у листи чекања није 0, пријављује се грешка.
2. У супротном умањује се глобални бројач семафора.
3. Брише се комплетан садржај семафора, тј. сва поља се постављају на нуле.

6.1.3. Чекање на семафор

Оперативни систем омогућава само блокирајуће чекање на семофору.

Псеудо код функције за чекања не семафору је следећи:

1. Уколико је бројач већи од нуле, умањује се за један и послу се дозвољава коришћење ресурса који је захтевао.
2. У супротном, посао се пребацује у стање блокиран и његов контролни блок се убацује у листу контролних блокова послова који чекају на семафор.
3. Позива се распоређивач и подиже се захтев за променом контекста.
4. Памти се тренутно време у систему да би се знало када је посао блокиран.

6.1.4. Сигнализирање семафора

Једини начин да посао који је блокиран чекајући на семафору пређе у стање спреман је ако му неки други посао или функција за обраду прекида пошаљу сигнал на том семафору. Један посао може да сигнализира већем броју семафора.

Псеудо код функције за сигнализирање семафору је следећи:

1. Чита се системско време.
2. Проверава се да ли постоје послови који чекају на семафор. Уколико је листа контролних блокова послова који чекају на семафор празна, увећава се бројач семафора, памти се тренутно системско време као временски печат и контрола се враћа позивајућем послу.
3. У случају да листа није празна, итеративно се пролази кроз листу и обавештавају се сви послови који чекају.
4. Ако је изабрана опција да се обавести само један посао, итерација се прекида након првог корака.
5. У случају да је изабрана опција без позивања распоређивача, тај корак се прескаче, а иначе се позива распоређивач и контрола се препушта послу са највишим приоритетом.

6.2. Катанци

Катанац (енг. mutex) је специјални конструкт у оквиру оперативног система који пословима омогућава екслузиван приступ ресурсима. Катанци су бинарни семафори са додатним својствима које обични семафори не пружају.

Катанци се користе да би се смањио проблем инверзије приотета. Инверзија приоритета се дешава када посао ниског приоритета држи ресурс који је потребан послу вишег приоритета. Да би се ова појава смањила језгро оперативног система може да подигне приоритет посла са нижим приотетом на приоритет посла са вишим приоритетом све док посао нижег приоритета не заврши са употребом ресурса [3].

Да би се катанци користили на прави начин оперативни систем мора да подржава већи број послова са истим приоритетом и као решење се намеће могућност да се послу који је узео катанац приоритет краткотрајно подигне на већи од свих осталих процеса и тиме избегне преемпција и осигура коректан завршетак процеса.

Претпоставимо да имамо три посла различитих приоритета који се извршавају на систему. Нека су то послови П1 са приотетеом 10, П2 са приоритетом 15 и П3 са приоритетом 20, и нека сваки од њих дели један заједнички ресурс осигуран једним катанцем. Приоритет са бројем 9 ће аутоматски бити резервисан и користиће се за приоритетно наслеђивање приоритета.

Након неког времена, могуће је да посао П3 приступи заједничком ресурсу и тиме добије катанац у своје власништво. Посао П3 може да се извршава неко време, након чега може да се догоди промена контекста и контрола се препусти послу П1. Као и осталима и послу П1 је потребан заједнички ресурс и покушаће да му приступи присвајањем катанца позивом функције `OSMutexPend()`. У овом случају, позвана функција примећује да је посао вишег приоритета затражио катанац који већ држи посао нижег приоритета и стога ће привремено подићи приоритет посла П3 на 9, чиме ће на силу извести промену контекста и вратити контролу послу П3. Након што посао П3 заврши са заједничким ресурсом, позива OSMutexPost() да би ослободио катанац. OSMutexPost() примећује да је власник катанца био посао коме је приоритет вештачки подигнут и враћа га назад на његов оригинални приоритет. Такође, функција примећује да процес са вишим приоритетом жели да користи заједнички ресурс, препушта му га, извршава промену контекста и препушта контролу послу П1.

Катанци у оквиру оперативног система су представљени структурама које имају најмање три елемента. Први елемент је заставица која сигнализира да ли је катанац слободан или не, приоритет који се додељује послу који је власник катанца у случају да посао већег приоритета жели приступ заједничком ресурсу, као и листу послова који чекају да се катанац ослободи .

Оперативни систем корисницима пружа следеће могућности за рад са катанцима: `OSMutexCreate()`, `OSMutexPend()`, `OSMutexPost()` и `OSMutexDel()`.

У складу са описаним, структура која описује катанац у оквиру оперативног система има следећа поља:

Табела 3. Опис структуре која представља катанац

Поље	**Значење**
Type	Тип објекта у оквиру оперативног система
NamePtr	Показивач на име катанца
PendList	Листа контролних блокова посла који чекају на катанцу
OwnerTCBPtr	Показивач на контролни блок посла који је власник катанца
OwnerOriginalPrio	Оригинални приоритет посла који је власник катанца
OwnerNestingCtr	Бројач
TS	Временски печат последње промене

6.2.1. Креирање катанца

Пре употребе катанца неопходно га је креирати. Креирање се постиже позивом функције `OSMutexCreate()`. Почетна вредност катанца је увек 1, чиме се сигнализира да је ресурс слободан.

Псеудо код функције за креирање катанца је следећи:

1. Тип објекта се поставља на катанац.
2. Листа контролних блокова се иницијализује на NULL.
3. Бројач се поставља на нулу.

4. Приоритет се поставља на минимални приоритет посла.
5. Временски печат се поставља на нулу.
6. Увећава се глобални бројач катанаца у систему.

6.2.2. Уклањање катанца

Уклањање катанца је потенцијално опасна операција, јер различити послови могу да приступе већ обрисаном катанцу. Да се ово не би догађало, пре брисања катанца треба да се избришу сви послови који тај катанац користе.

Псеудо код функције за уклањање катанца из система је следећи:

1. Провера се број послова који чекају на катанац.
2. Уколико тај број није нула, пријављује се грешка.
3. У супртоном смањује се глобални бројач катанаца и брише се комплетан садржај катанца, односно сва поља се постављају на нулу.

6.2.3. Чекања на катанац

Оперативни систем омогућава само блокирајуће чекање на катанац. Чекање на катанац није активно. Већи број послова може да чека на истом катанцу, али један посао може да буде блокиран само на једном катанцу.

Функција за чекање на катанцу има следећи псеудо код:

1. Ако је катанц слободан, попуњавају се поља у структури и послу се дозвољава коришћење ресурса враћањем из функције.
2. Проверава се да ли је посао који је позвао функцију исти као и посао који је власник катанца. Ако јесте, пријављује се грешка и излази се из функције.
3. Ако је приоритет посла који је власник катанца мањи од приоритета тренутно активног посла , провера се стање власника катанца.
4. Ако је власник катанца у листи спремних послова, уклања се из ње, увећава му се приоритет, увећани приоритет се уписује у таблицу приоритета и посао се убацује на почетак листе спремних послова.
5. Ако је власник катанца у стању чекања или блокиран, само му се увећава приоритет.
6. Посао који је позвао функцију се пребацује у стање блокиран и његов контролни блок се ставља у листу контролних блокова који чекају на катанац.
7. Позива се распоређивач и подиже се захтев за променом контекста.
8. Памти се тренутно време у систему да би се знало када је посао блокиран.

6.2.4. Сигнализирање катанца

Једини начин да посао који је блокиран чекајући на катанцу пређе у стање спреман је ако му неки други посао пошаље сигнал на том катанцу и тиме ослободи ресурс.

Псеудо код функције за сигнализирање катанцу је следећи:

1. Проверава се који посао жели да изврши сигнализирање катанцу. Ако се ради о послу који није власник катанца, пријављује се грешка и излази се из функције.

2. Чита се системско време, памти се временски печат у оквиру катанца и умањује се бројач.
3. Уколико је бројач већи од нула, пријављује се грешка и излази се из функције.
4. Уколико у листи послова који чекају није било контролних блокова, ресетује се катанац и излази се из функције.
5. У супротном, проверава се да ли је дошло до промене приоритета посла који је власник катанца.
6. Ако је дошло до промене приоритета, власник се уклања из листе спремних послова, приоритет му се враћа на ориганални што се памти у таблици приоритета. Посао се убацује на крај листе спремних послова и у глобални приоритет посла се уписује оригинални приоритет власника катанца.
7. Мења се власник катанца. Нови власник постаје први посао у листи послова који чекају на катанцу. Новом власнику катанца се шаље сигнал и пребацује се из листе блокираних послова у листу спремних.
8. Позива се распоређивач и диже се заставица за промену контекста, сем у случају када је изабрана опција без позивања распоређивача.

6.3. Критичне секције

С обзиром на то да је већ дефинисан изабрани модел заштите критичних секција уз помоћ хардвера, сада само треба дефинисати макрое у оквиру оперативног система који користе већ дефинисане макрое нижег нивоа. Критична секција у оквиру оперативног система није исто што и критична секција на хардверу. У оквиру оперативног система улазак у критичну секцију представља блокирање рада распоређивача, односно промену контекста, док излазак из критичне секције представља омогућавање рада распоређивача.

С обзиром да рад распоређивача може да се блокира, у оквиру оперативног система мора да се уведе додатни бројач који ће бити задужен за бројање угнеждених позива распоређивачу. Увођење бројача делује бесмислено, али није тешко замислити ситуацију у којој је неопходан. Распоређивач може да се позива из послова али и из функција за обраду прекида. Проблем настаје приликом позивања распоређивача из функција за обраду прекида.

Претпоставимо да имамо две функције за обраду прекида различитих приоритета, и да свака од њих на крају сервисирања прекида позива распоређивач. Врло лако је доћи у ситуацију у којој функција за обраду прекида вишег приоритета прекида функцију за обраду прекида нижег приоритета. Приликом завршетка извршавања функције вишег приоритета биће позван распоређивач и контрола враћена функцији нижег приоритета која поново позива распоређивач и поново подиже заставица за промену контекста. Очигледно распоређивач ће бити позван два пута без промене контекста што је беспотребно трошење времена. Бесмисленост се огледа у томе што је функција за промену контекста прекид најнижег приоритета и мора да сачека да се сви прекиди сервисирају пре него што она сама дође на ред. У замишљеном случају, довољно је позвати распоређивач једном, тј. приликом изласка из функције за обраду приоритета најнижег нивоа.

Решење проблема је врло једноставно. При иницијализацији оперативног система, бројач угнеждених позива распоређивачу се поставља на нулу. Сваким уласком у критичну секцију оперативног система, бројач се увећава за један, а приликом изласка из критичне секције се умањује за један. Распоређивач је могуће позвати само приликом изласка из критичне секције

оперативног система под условом да је бројач нула, у сваком другом случају позив распоређивачу ће бити прескочен.

У овом тренутку битно је направити разлику између критичне секције у оквиру оперативног система и критичне секције у оквиру процесора. Критична секција у оквиру процесора гарантује да ће се све наредбе у оквиру критичне секције третирати као једна атомична операција коју нико не може да прекине, док критичне секције оперативног система само онемогућавају позиве распоређивачу, а тиме и промену контекста. Кориснику су доступни макрои уз помоћ којих се користе и један и други тип критичних секција.

7. Покретање оперативног система

Да би извршавање програма било коректно и понашање оперативног система очекивано, неопходно је извршити следеће кораке у оквиру main функције:

1. Иницијализовати сам процесор и подесити одговарајућу фреквенцију процесора.
2. Онемогућити све екстерне прекиде.
3. Алоцирати и иницијализовати све контролне блокове послова који су потребни.
4. Креирати све корисничке послове који су неопходни на почетку извршавања.
5. Позвати функцију `OSInit()` којом се иницијализују све интерне променљиве самог оперативног система (листе послова, таблице приоритета, семафори, катанци, бројач угнежђености прекида, глобални бројачи семафора и катанаца...).
6. Иницијализовати системски сат (енг. SysTick) и омогућити прекидање системског сата.
7. Покренути сам оперативни систем позивом функције `OSStart()`.

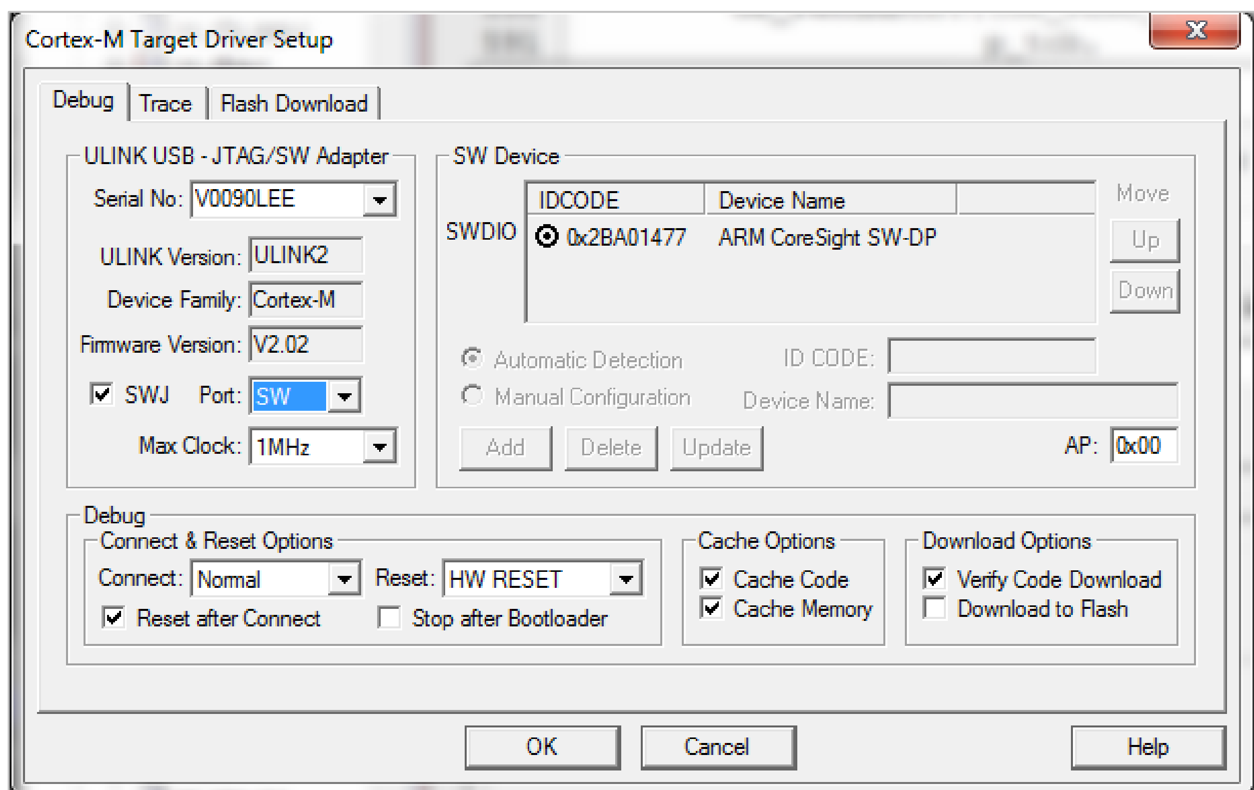
Неопходно је испоштовати наведену процедуру да би се осигурало коректно извршавање програма и распоређивање корисничких послова.

8. Развој програма

Када се ради о развоју програма за PC рачунаре, дебаговање истих је олакшано присуством многих алата и графичког подсистема, док у свету микроконтролера свега тога нема. Једини алати за дебаговање су осцилоскоп и елементарни дебагер уколико га развојно окружење поседује. Управо из тих разлога, развојни пут је дугачак и спор. Као што је раније речено, приликом развоја програма користиће се KEIL µVision развојно окружење.

8.1. Проверавање хардвера

Први корак у развоју система је уверавање да је сам хардвер исправан и да програматором може да се приступи процесору. Видљивост процесора у развојном окружењу је лако проверљива. Довољно је повезати програматор са процесором и довести напајање хардверу. Уколико је све у реду, процесор је видљив у развојном окружењу.



Слика 16. Приказ видљивости процесора у развојном систему

Када се утврди видљивост процесора, неопходно је проверити да ли процесор може да се програмира, тј. да ли су све линије ка процесорском ЕЕПРОМ-у лепо повезане. Није довољно само да се процесор испрограмира, већ након програмирања мора да крене са извршавањем програма снимљеног у ЕЕПРОМ.

Очигледно, неопходно је прво написти неки једноставни програмчић којим ће се испитати да ли процесор уопште може да се програмира, али и да тако испрограмиран заиста извршава оно што је у програму написано. Овај корак је испао тежак, јер приликом дизајнирања хардвера нигде није остављено место за диоде којима би се сигнализирало да је хардвер под напоном и да процесор

извршава неки програм. Да постоји диода једноставно би могла кроз софтвер да се пали и гаси уз помоћ процесора тако што би периодично била под напоном и без напона.

Ово се заобилази тако што се неки некоришћени GPIO пин иницијализује као излазни и наизменично се поставља на логичку јединицу, односну логичку нулу. Логичка јединица представља напон од 3.3V, док логичка нула представља напон од 0V.

8.1.1. Општа структура угнежденог програма

Да би се ово остварило неопходно је неколико предрадњи. Прво, ради се о микроконтролерима, дакле тренутно нема оперативног система који за нас одради све предрадње и ми само покренемо апликацију. Овде корисник сам мора о свему томе да мисли. Када се пише било који програм за микроконтролере, општа структура изгледа овако:

1. Подешавање фреквенције процесорског сата и иницијализација система – већина микроконтролера може да ради на различитим фреквенцијама у зависности од изабраног множиоца кристалног осцилатора. Најчешће се уграђује кристални осцилатор са фреквенцијом од 12 или 20 мегахерца. Подешавањем одговарајућих односа могуће је постићи велики број фреквенција на којима микроконтролер може да ради. У свету микроконтролера се спецификује максимална радна фреквенција, а све остале које су ниже од максималне су такође употребљиве.
2. Подешавање периферијских уређаја и приоритета прекида – сам процесор као такав нема никакав интерфејс ка спољном свету. Повезивање са спољним светом омогућено је разним периферним интерфејсима и додатним контролерима који су у процесор уграђени. Да би се ти контролери користили неопходно је прво их подесити у складу са упутством које доставља произвођач микроконтролера. Коришћење периферијских уређаја без претходног подешавања може да доведе до оштећења самог микроконтролера. Уколико периферијски уређаји користе механизам прекида за интеракцију са процесором, неопходно је подесити приоритете сваког прекида понаособ, као и испрограмирати функције (послове) које обрађују саме прекиде. Прескакање овог корака доводи до недефинисаног понашања система, трке за ресурсима и практично немогућег дебаговања. Такође, неопходно је навести меморијске адресе на којима се налазе функције за обраду прекида у самом вектору прекида.
3. Главна петља програма – програми написани за микроконтролере се најчешће врте у бесконачној петљи. У тој петљи је комплетно извршавање програма. Током извршавања могуће је мењати подешавања периферијских уређаја, искључивати их и укључивати, при чему неки микроконтролери дозвољавају чак и промену радне фреквенције централног процесора током извршавања.
4. Прецизно дефинисање адреса у ЕЕПРОМ-у на које треба снимити програм, податке, и функције за обраду прекида
5. Прецизно дефинисање адреса у РАМ-у које се користе за инструкције, податке, хип и стек.
6. Снимање програма у ЕЕПРОМ и почетак тестирања.

Кораци 4. и 5. нису увек неопходни и то најчешће зависи од верзије компилатора, верзије развојног окружења и верзије самог процесора. Фајл који се добија у кораку 4. се најчешће назива

иницијализационим фајлом, и углавном је довољно написати га једном јер се најчешће наводи само почетна адреса. Већина компилатора и развојних окружења аутоматски генерише овај фајл на основу сазнања о верзији микроконтролера која се користи.

Фајл из корака 5. се назива растурајући фајл (енг. scatter file) и служи за правилан распоред инструкција и података приликом извршавања програма. Најчешће инструкције иду на најниже адресе, праћене подацима и хипом који расте ка вишим адресама. Стек се најчешће смешта на највише адресе и расте према нижим адресама. Неки компилатори аутоматски генеришу растурајуће фајлове, док је код неких тај процес препуштен програмеру.

8.1.2. Први програм

Циљ је да се тестира хардвер, и написани програм само треба да мења стање на једном унапред изабраном пину. Пратећи општу структуру угнежденог програма прво треба да се подеси фреквенцију сата. Изабрана верзија ARM Cortex M3 процесора, NXP LPC1768FBD144 када се пробуди увек ради на максималној фреквенцији, тако да овај корак практично може да се прескочимо. Када се буде користио развијени оперативни систем, експлицитно треба подесити процесор да ради на максималној фреквенцији.

Остаје само да се изабере слободан пин и да му се мења стање. С обзиром на то да се софтверски утиче на стање на пину, очигледно је да пин мора да се дефинише као излазни и да GPIO периферијски уређај треба да се подеси у складу са тим. GPIO периферијски уређај је на изабраној верзији ARM Cortex M3 организована у три порта, али је омогућен независан приступ сваком појединачном пину на процесору. Организација по портовима је врло корисна када се користи раније поменути bit-banging. За детаље погледати [5].

Главна петља у програму обухвата постављање пина на логичку јединицу, чекање неког времена и спуштање на логичку нулу, затим чекање неког времена и опет испочетка. Дубљим размишљањем, у свакој итерацији треба да се промени стање пина и да се сачека неко предефинисано време. Чекање се једноставно реализује *for* петљом у којој се само инкрементира бројач (активно чекање).

Кораци 4. и 5. нису потребни, јер развојно окружење KEIL μVision генерише све потребне фајлове. За имплементацију свих угнеждених програма изабран је програмски језик C. Дакле, дијаграм првог програма изгледа овако:

Слика 17. Дијаграм програма за тест хардвера

Сам изворни код програма такође није ништа специјално комликован, јер се ради о једноставној угнежденој апликацији без оперативног система:

```c
int main(void) {


    /* struktura za podesavanje GPIO pinova */
    PINSEL_CFG_Type PinCfg;

    /* pinu P0.16 se dodeljuje GPIO uloga */
    PinCfg.Portnum = 0;
    PinCfg.Pinnum = 16;
    PinCfg,Funcnum = 0;
    PINSEL_ConfigPin(&PinCfg);

    /* smer pina se postavlja kao izlazni */
    GPIO_SetDir(0, 1<<16, 1);
    /* pocetna vrednost na pinu je logicka jedinica */
    GPIO_SetValue(0, 1<<16);

    /* pin se naizmenicno spusta i podize u beskonacnoj petlji */
    while (1) {

        /* vrednost pina se postavlja na logicku nulu */
        GPIO_ClearValue(0, 1<<16);

        /* vrednost pina se postavlja na logicku jedinicu */
        GPIO_SetValue(0, 1<<16);
    }

    return 0;
}
```

Слика 18. Приказ кода првог програма

Након што се преведе програм и испрограмира процесор, остаје само да се провери да ли процесор заиста извршава оно што му је задато. Једино како то може да се провери је уз помоћ осцилоскопа и да се визуелно увери да слика сигнала одговара ономе што се очекује.

Након ових тестова утврђује се да хардвер заиста ради и тек сада може да се пређе на имплементирање самог оперативног система и свих осталих потребних програма. Прво што треба да се уради је оживљавање комуникације са спољним светом.

8.2. UART периферијски уређај

Да би дебаговање програма било лако и брзо, паралелно са програмом за ARM развијаће се и .NET PC апликација уз чију помоћ ће лако моћи да се уочи да ли комуникација ради или не. С обзиром на то да је један од циљева максимално брза комуникацију са спољним светом, комуникација ће се развијати уз помоћ DMA контролера.

8.2.1. UART ехо сервер

Да би уопште постојала комуникација са спољним светом, у оквиру програма мора да се подеси и покрене сам UART периферијски уређај који је уграђена у процесор. Прво што треба да се уради је да се дефинишу који пинови на процесору се користе као улазне и излазне линије за сам периферијски уређај. Улазни пин представља рисивер линију, а излазни пин се користи као трансмитер линија. Уз помоћ ова два пина реализује се серијска комуникација. Додатно, мора да

се одреди бодовна брзину на којој ће се комуникација одвијати, број битова података, број завршних битова као и да ли се користи бит парности или не. У комуникацији ће се користити 8 битова података, 1 завршни бит и неће се користити бит парности. Бодовна брзина која се користи износи 115200 бодова. Бод у електроници представља број симбола (пулсева) који се могу пренети у секунди. Дакле, један бит траје

$$\frac{1}{115200} = 8.68\ \mu s.$$

Узимајући у обзир да се користи један почетни бит, осам битова података и један завршни бит, за један бајт треба 10 битова, што значи да за пренос једног бајта података треба

$$10 \cdot 8.68\mu s = 86.8\mu s.$$

Одавде лако може да се израчуна колико бајтова може максимално да се пренесе у једној секунди. Укупна количина бајтова износи

$$\frac{1}{86.8 * 10^{-6}} \sim 11\ 520\ Bps\ .$$

Овај податак ће бити врло важан приликом одређивања брзине за комуникацију са екстерном меморијом. UART периферијски уређај по дефиницији не ради у DMA режиму, и то је последња ствар коју корисник мора да подеси пре покретања самог периферијског уређаја.

```
/* UART konfiguraciona struktura      */
UART_CFG_Type UARTConfigStruct;
/* UART FIFO konfiguraciona struktura */
UART_FIFO_CFG_Type UARTFIFOConfigStruct;

PINSEL_CFG_Type PinCfg;  /* Struktura za konfigurisanje pinova  */
OS_ERR os_err;           /* poruka o gresci u okviru operativnog sistema */

/* prvi korak je podesavanja pinova u skladu
 * sa dizajnom hardvera
 */
PinCfg.Funcnum   = 1;
PinCfg.OpenDrain = 0;
PinCfg.Pinmode   = 0;
PinCfg.Pinnum    = 2;
PinCfg.Portnum   = 0;
PINSEL_ConfigPin(&PinCfg);
PinCfg.Pinnum    = 3;
PINSEL_ConfigPin(&PinCfg);

/* konfiguraciona struktura se inicijalizuje na podrazumevane vrednosti
 * 8 bitova za podatke, 1 stop bit, bez parnosti, 9600 bodovna brzina
 */
UART_ConfigStructInit(&UARTConfigStruct);
/* bodovna brzina se menja na unapred dogovorenu */
UARTConfigStruct.Baud_rate = 115200;
```

```
/* UART0 serijska periferija se inicijalizuje uz pomoc konfiguracione
 * strukture
 */
UART_Init((LPC_UART_TypeDef *)LPC_UART0, &UARTConfigStruct);

/* FIFOConfigStruct se inicijalizuje na podrazumevane vrednosti:
 *               - FIFO_DMAMode = DISABLE
 *               - FIFO_Level = UART_FIFO_TRGLEV0
 *               - FIFO_ResetRxBuf = ENABLE
 *               - FIFO_ResetTxBuf = ENABLE
 *               - FIFO_State = ENABLE
 */
UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
/* omogucava se upotreba DMA kontrolera */
UARTFIFOConfigStruct.FIFO_DMAMode = ENABLE;
/* inicijalizuje se FIFO za UART0 periferijski uredjaj */
UART_FIFOConfig((LPC_UART_TypeDef *)LPC_UART0, &UARTFIFOConfigStruct);
/* omogucava se UART komunikacija */
UART_TxCmd((LPC_UART_TypeDef *)LPC_UART0, ENABLE);
```

Слика 19. Подешавање UART периферијског уређаја

Остаје још само да се подеси DMA контролер. DMA контролер уграђен у ARM Cortex M3 има осам независних канала. Сваки од канала може да се користи или за улаз или за излаз. Дакле, потребна су два DMA канала, један за рисивер линију и други за трансмитер линију. Подешавање самих канала је изузетно лако. Једино што треба навести су адресе бафера, тип трансфера и дужина пакета који се шаље.

```
/* Inicijalizuje se GPDMA kontroler */
GPDMA_Init();

/* ukljucuje se prekidanje za DMA kontroler sa prioritetom 3 */
CSP_IntSrcCfg(CSP_INT_CTRL_NBR_MAIN, CSP_INT_SRC_NBR_DMA_00, 3, \
              (CSP_OPT)0);
/* registruje se funkcija za obradu DMA prekida */
CSP_IntVectReg(CSP_INT_CTRL_NBR_MAIN, CSP_INT_SRC_NBR_DMA_00, \
               DMA_IRQHandler, (void*)0);

/* kreira se semafor za sinhronizaciju upotrebe serijskog kontrolera */
OSSemCreate(&uart_semaphore, (M3_CHAR*)((void*)"UARTSem"), 1, &os_err);

/********** Podesavanje DMA kanala 0 **********/
GPDMACfg_Tx.ChannelNum = 0;      /* Kanal 0: Transmiter */
/* Memorija iz koje se salje */
GPDMACfg_Tx.SrcMemAddr    = (uint32_t) &send_data;
/* Duzina paketa */
GPDMACfg_Tx.TransferSize  = T_DATA_CONF;
/* Memorija u koju se upisuje - nebitna  */
GPDMACfg_Tx.DstMemAddr    = 0;
/* Sirina transfera - nebitna */
GPDMACfg_Tx.TransferWidth = 0;
/* Tip transfera: memorija -> periferija */
GPDMACfg_Tx.TransferType  = GPDMA_TRANSFERTYPE_M2P;
```

```
/* Periferija sa koje se prima - nebitna */
GPDMACfg_Tx.SrcConn        = 0;
/* periferija na koju se salje */
GPDMACfg_Tx.DstConn        = GPDMA_CONN_UART0_Tx;
/* Povezana lista za DMA transfer - nebitna */
GPDMACfg_Tx.DMALLI         = 0;
/* podesavanje kanala 0 u okviru samog DMA kontrolera */
GPDMA_Setup(&GPDMACfg_Tx);

/********** Podesavanje DMA kanala 1 **********/
GPDMACfg_Rx.ChannelNum = 1;     /* Kanal 1: Risiver */
/* Memorija iz koje se salje - nebitna */
GPDMACfg_Rx.SrcMemAddr     = 0;
/* Memorija u koju se upisuje */
GPDMACfg_Rx.DstMemAddr     = (uint32_t) &rx_buf;
/* Duzina paketa */
GPDMACfg_Rx.TransferSize   = R_DATA_CONF;
/* Sirina transfera - nebitna */
GPDMACfg_Rx.TransferWidth = 0;
/* Tip transfera: periferija -> memorija */
GPDMACfg_Rx.TransferType   = GPDMA_TRANSFERTYPE_P2M;
/* Periferija sa koje se prima */
GPDMACfg_Rx.SrcConn        = GPDMA_CONN_UART0_Rx;
/* Periferija na koju se salje - nebitna */
GPDMACfg_Rx.DstConn        = 0;
/* Povezana lista za DMA transfer - nebitna */
GPDMACfg_Rx.DMALLI         = 0;
/* podesavanje kanala 1 u okvira samog DMA kontrolera */
GPDMA_Setup(&GPDMACfg_Rx);

/* resetuju se brojaci prekida na odlaznom kanalu */
Channel0_TC  = 0;
Channel0_Err = 0;
/* resetuju se brojaci prekida na prijemnom kanalu */
Channel1_TC  = 0;
Channel1_Err = 0;

/* Aktivira se prekidanje na DMA kontroleru */
CSP_IntEn(CSP_INT_CTRL_NBR_MAIN,CSP_INT_SRC_NBR_DMA_00);
NVIC_EnableIRQ(DMA_IRQn);

/* ukljucuje se kanal za primanje  putem serijske veze*/
GPDMA_ChannelCmd(1, ENABLE);
```

Слика 20. Подешавање DMA контролера

Природно, DMA контролер ради у режиму прекида. Након што се покрену контролер и периферијски уређај, комуникација ће се одвијати у позадини, независно од тренутног посла. Када се пакет прими или пошаље, DMA контролер ће прекинути процесор. У том тренутку процесор је дужан да обради прекид.

Кључни корак приликом дизајнирања система је дефинисање приоритета прекида. Сви кориснички дефинисани прекиди по дефиницији имају приоритет 4, и идентичан субприоритет [5]. Уколико се за сваки прекид прецизно не наведе приоритет и субприоритет долази до потенцијалног конфликта

и заглављивања система. Ако се истовремено догоде два прекида са идентичним приоритетом и субприоритетом, долази до заглављивања система јер ће се међусобно блокирати. С обзиром на то да је DMA прекид тренутно једини кориснички дефинисани прекид, додељује му се приоритет 3. Додељени приоритет је хардверски приоритет, а не приоритет посла у оквиру оперативног система.

Рутине које обрађују прекиде морају да буду максимално брзе, јер је периферијски уређај блокиран све време док се налази у прекиду. У случају слања поруке, једино што треба да се догоди у прекиду је гашење DMA контролера, јер је порука већ послата. Када се ради о прекиду који узрокује примљена порука, постоје две могућности.

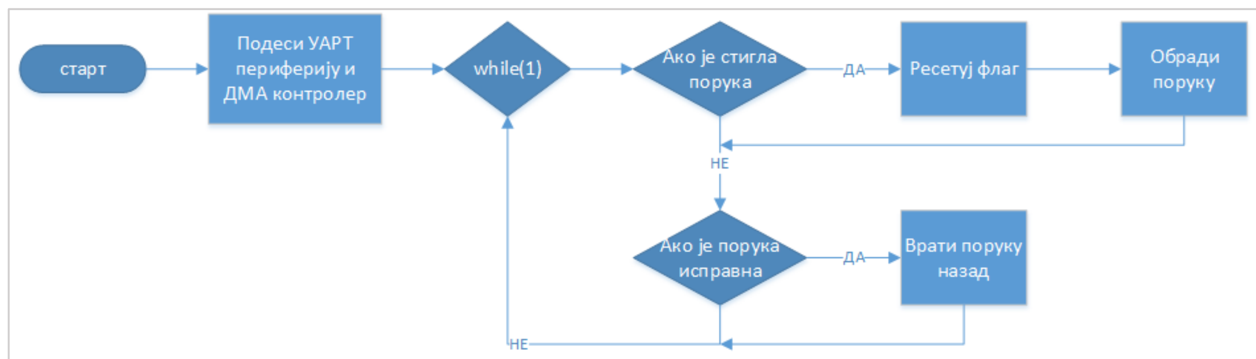
Једна могућност је обрада комплетне примљене поруке у самом прекиду и поновно покретање контролера. Иако једноставно и наизглед логично, ово решење је потпуно погрешно. Разлог за то је време које би процесор морао да проведе у стању прекида које је мало дуже од саме обраде поруке која сама по себи није занемарљива. Још један проблем је чињеница да процесор који је већ у прекиду не може још једном да буде прекинут. Циљ је скраћивање трајања времена прекида. Ово се најлакше постиже тако што се обрада саме поруке делегира другом послу. У прекиду је довољно поставити заставицу која ће обавестити главни процес о приспелој поруци, а главни процес је тај који ће одлучити када ће се порука обрадити. Ово је могуће јер је серијска комуникација са спољним светом неколико хиљада пута спорија од самог централног процесора.

```
void DMA_IRQHandler (void* p_arg)

{
    OS_ERR os_err;       /* poruka o gresci */

    /* ukoliko je DMA kanal za slanje uzrok prekida */
    if (GPDMA_IntGetStatus(GPDMA_STAT_INT, 0)) {
        /* ako je razlog prekida kraj prenosa */
        if(GPDMA_IntGetStatus(GPDMA_STAT_INTTC, 0)) {
            /* brise se podignuti flag za prekid */
            GPDMA_ClearIntPending (GPDMA_STATCLR_INTTC, 0);
            /* uvecava se brojac uspesnih slanja */
            Channel0_TC++;
            /* gasi se DMA kanal za slanje */
            GPDMA_ChannelCmd(0, DISABLE);
            /* signaliziranjem semaforu se oslobadja resurs */
            OSSemPost(&uart_semaphore, OS_OPT_POST_1, &os_err);
        }
    }
    /* ukoliko je DMA kanal za primanje uzrok prekida */
    if (GPDMA_IntGetStatus(GPDMA_STAT_INT, 1)) {
        /* ako je razlog prekida kraj prenosa */
        if(GPDMA_IntGetStatus(GPDMA_STAT_INTTC, 1)) {
            /* brise se podignuti flag za prekid */
            GPDMA_ClearIntPending (GPDMA_STATCLR_INTTC, 1);
            /* uvecava se brojac uspesnih primanja */
            Channel1_TC++;
            /* gasi se DMA kanal za primanje */
            GPDMA_ChannelCmd(1, DISABLE);
```

```
            /* postavlja se flag za primanje */
            ready = 1;
            /* signaliziranjem semaforu, obavestava se proces za
             * kontrolu serijske komunikacije da je uspesno
             * primljena poruka putem serijske veze
             */
            OSSemPost(&action_semaphore, OS_OPT_POST_1, &os_err);
        }
    }
}
```

Слика 21. Функција за обраду DMA прекида

Главни процес за сада само треба да покрене и подеси UART периферијски уређај, и у главној петљи да чека информацију о приспелој поруци, затим треба да обради приспелу поруку и врати је назад преко трансмит линије. Паралелно са главним послом, комуникација се одвија независно уз помоћ DMA контролера.



Слика 22. Дијаграм UART ехо сервера

8.2.2. Синхронизација послова

Приступ који је до сада описан има једну озбиљну ману. Прва мана је у томе што не постоји никакав механизам синхронизације у случају да већи број послова жели да користи један једини серијски периферијски уређај који је доступан. Није тешко замислити ситуацију у којој један посао покрене слање поруке и активира DMA канал, али баш у том тренутку се догоди преемпција и контрола се препусти другом процесу који хоће нешто ново да пошаље серијском везом. Чињеница да први процес више није у стању извршавања нема никакав утицај на покренути DMA канал и само слање, јер он ради независно од процесора и баш зато је неопходна заштита приликом употребе серијског контролера. С обзиром да нема заштите, други посао може да прегази тренутно слање и да крене да шаље своју поруку, без да је порука првог посла у потпуности послата. Овакав след догађаја би довео до грешке на DMA контролеру и недефинисаног понашања микроконтролера.

Да би се ефикасно користио доступан периферијски уређај неопходно је увести бројачки семафор који ће штитити употребу серијског контролера. Пре коришћена серијског контролера посао мора да пита да ли је семафор слободан позивом функције `OSSemPend()`. У случају да јесте, посао добија контролу над периферијским уређајем и може да пошаље поруку. По завршетку слања DMA контролер подиже заставицу и захтева сервисирање прекида. У функцији за обраду прекида треба обавестити први посао који потенцијално чека на серијски контролер да је контролер ослобођен и

да може да пошаље своју поруку. Обавештавање се ради позивом функције `OSSemPost()` на бројачком семафору. На овај начин, серијски контролер ће бити заштићен и комуникација ће бити коректна.

Сваки посао пре коришћења серијског контролера треба да провери да ли је контролер слободан и по поптреби сачека да се ослободи да би наставио са извршавањем. Коришћењем семафора за синхронизацију послова избегава се замка активног чекања и на тај начин се оперативни систем растерећује.

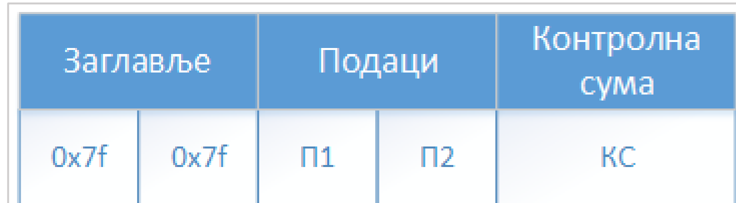
Све команде ка црној кутији пристижу серијским путем. Очигледно намеће се потреба за још једним семафором који ће овог пута бити бинарни и служиће као сигнал да је нова порука пристигла. Неопходно је дефинисати посао који ће само чекати долазак нове команде и у складу са примљеном командом будити остале послове и на тај начин контролисати рад црне кутије. По пријему поруке на DMA контролеру подиже се заставица и тражи се сервисирање прекида. У функцији за обраду прекида потребно је позвати функцију `OSSemPost()` и на тај начин обавестити главни посао да је стигла порука коју треба да обради. Главни посао је посао најмањег приоритета од свих кориснички дефинисаних послова у оквиру оперативног система и служи само као посредник који на основу примљених команди буди остале послове. Главни посао је бесконачна петља која као прву наредбу има проверу бинарног семафора и уколико није било активности чека све док не буде пробуђен пристиглом поруком. Опет, кључна предност семафора је избегавање активног чекања и оптерећивања система.

```
static void App_SerialTask(void* p_arg){


  OS_ERR err;               /* promenljiva u kojoj se cuvaju greske */
  int status = UNDEFINED;   /* status primljene poruke */

  /* pokrece se UART periferijski uredjaj */
  UART0_DMAInit();

  /* u beskonacnoj petlji */
  while (DEF_TRUE) {

    /* ceka se sve dok ne stigne nova poruka */
    OSSemPend(&action_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, &err);

    /* proverava se pristigla poruku */
    status = check_message();

    /* ako je poruka odgovarajuce strukture */
    if (status == ECHO) {
      /* ceka se dok se ne oslobodi UART uredjaj */
      OSSemPend(&uart_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, &err);
      /* primljena poruka se vraca nazad */
      echo();
    }
  }
}
```

Слика 23. Посао за контролу серијске везе.

8.2.3. Команде за конфигурисање серисјке везе

Да би све ово имало смисла, потребно је дефинисати комуникациони протокол да би долазећи и одлазећи пакети могли да се синхронизују. Изабрани протокол на почетку има два бајта облика $0x7f$, док је последњи бајт контролна сума која се рачуна као сума бајтова података маскирана са $0xff$. Бајтови између заглавља и контролне суме су бајтови података. Иницијално, дужина пакета је 5 бајтова, тј. два бајта податка, два бајта заглавља и контролна сума.



Заглавље		Подаци		Контролна сума
0x7f	0x7f	П1	П2	КС

Слика 24. Структура UART пакета

Након дефинисања структуре комуникационог протокола (пакета), неопходно је дефинисати функције који шаљу и примају поруке уз помоћ UART периферијског уређаја. Очигледно је шта треба да ради функција која шаље поруку UART-ом. Функција мора да конфигурише DMA канал за слање тако што ће у структури поставити адресу у меморији из које се чита порука, дужину поруке, тип трансфера и периферијски уређај на који се порука шаље. Након ових подешавања, функција још само треба експлицитно да укључи DMA контролер. Креирање поруке која се шаље делегира се пословима који желе да користе UART периферијски уређај.

```
void DMA1_Tx(void)

{
    /* resetuje se brojac paketa za kanal 0 */
    Channel0_TC   = 0;
    /* resetuje se brojac gresaka za kanal 0 */
    Channel0_Err = 0;

    /* Kanal 0: Transmiter */
    GPDMACfg_Tx.ChannelNum = 0;

    /* Memorija iz koje se salje */
    GPDMACfg_Tx.SrcMemAddr    = (uint32_t) &send_data;
    /* Duzina paketa */
    GPDMACfg_Tx.TransferSize   = configured == 1 ? T_DATA : T_DATA_CONF;;
    /* Memorija u koju se upisuje - nebitna */
    GPDMACfg_Tx.DstMemAddr        = 0;
    /* Sirina transfera - nebitna */
    GPDMACfg_Tx.TransferWidth = 0;
    /* Tip transfera: memorija -> periferija */
    GPDMACfg_Tx.TransferType   = GPDMA_TRANSFERTYPE_M2P;
    /* Periferija sa koje se prima - nebitna */
    GPDMACfg_Tx.SrcConn              = 0;
```

```
    /* periferija na koju se salje */
    GPDMACfg_Tx.DstConn              = GPDMA_CONN_UART0_Tx;
    /* Povezana lista za DMA transfer - nebitna */
    GPDMACfg_Tx.DMALLI             = 0;
    /* podesavanje kanala 0 u okviru samog DMA kontrolera */
    GPDMA_Setup(&GPDMACfg_Tx);

    /* Uljucuje se DMA kanal za slanje putem serijske veze  */
    GPDMA_ChannelCmd(0, ENABLE);
}
```

Слика 25.Функција за слање пакета UART-ом уз помоћ DMA контролера

Када је примање поруке у питању, посао је дужан да на исти начин конфигурише DMA канал за примање нове поруке и пре него што активира канал за ново примање мора да обради пристиглу поруку. У случају да је обрађена порука исправна, функција је дужна да постави заставицу којом ће обавестити главни посао да је пристигла порука исправна. У случају да порука није исправна, игнорише се. Након обраде поруке, DMA канал за примање се поново покреће.

```
void DMA1_Rx(void)

{
    int idx = 0, i;     /* brojaci u petljama */
    int limit = configured == 1 ? R_DATA : R_DATA_CONF; /*velicina bafera*/

    /* resetovanje brojaca paketa na prijemnom kanalu */
    Channel1_TC                      = 0;
    /* resetovanje brojaca gresaka na prijemnom kanalu */
    Channel1_Err                     = 0;

    /* podesava se prijemni kanal */
    GPDMACfg_Rx.ChannelNum   = 1;

    /* memorija iz koje se salje - nebitno */
    GPDMACfg_Rx.SrcMemAddr   = 0;
    /* memorija u koju se upisuje */
    GPDMACfg_Rx.DstMemAddr       = (uint32_t)&rx_buf;
    /* duzina paketa */
    GPDMACfg_Rx.TransferSize  = configured == 1 ? R_DATA : R_DATA_CONF;
    /* sirina transfera - nebitna */
    GPDMACfg_Rx.TransferWidth = 0;
    /* tip transfera: periferija -> memorija */
    GPDMACfg_Rx.TransferType  = GPDMA_TRANSFERTYPE_P2M;
    /* periferija sa koje se prima */
    GPDMACfg_Rx.SrcConn              = GPDMA_CONN_UART0_Rx;
    /* periferija na koju se salje */
    GPDMACfg_Rx.DstConn              = 0;
    /* povezana lista za DMA transfer - nebitno */
    GPDMACfg_Rx.DMALLI             = 0;
    /* podasavanje samog kanala u okviru DMA kontrolera */
    GPDMA_Setup(&GPDMACfg_Rx);
```

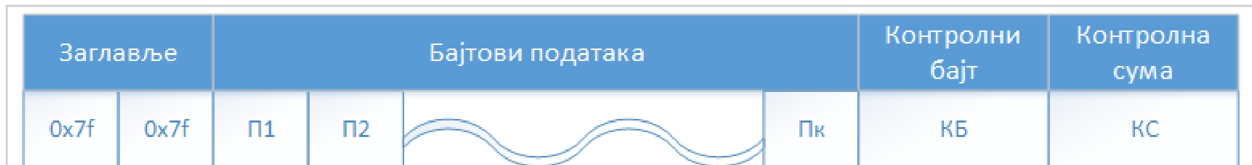
```
    /* izvlaci se podatak iz prijemnog bafera */
    while (idx < RX_BUF_SIZE) {

        /* izvlaci se podatak iz prijemnog bafera */
        rx_buf_BB[counter_BB] = rx_buf[idx];
        /* u zavisnosti od slobodnog mesta u lokalnom baferu */
        switch (counter_BB) {
            case 0:
                if(rx_buf[idx] == 0x7F) {
                    counter_BB++;
                }
                break;
            case 1:
                if(rx_buf[idx] == 0x7F) {
                    counter_BB++;
                }
                else {
                    counter_BB = 0;
                }
                break;
            default:
                if(counter_BB > 1)
                    counter_BB++;
                break;
        }

        if(counter_BB == limit) {
            /* resetuje se brojac u lokalnom baferu */
            counter_BB = 0;
            /* postavlja se flag za glavni program da je nova poruka
             * primljena
             */
            received_all = 1;

            /* podaci se kopiraju iz lokalnog bafera u bafer glavnog
             * programa
             */
            for (i = 0; i < RX_BUF_SIZE; i++)
                res_data[i] = rx_buf_BB[i];
        }
        /* uvecava se brojac */
        idx++;
    }
    /* Na kraju obrade podataka, cisti se bafer DMA kontrolera */
    for (idx = 0; idx < RX_BUF_SIZE; idx++) {

            rx_buf[idx] = 0;
    }

    /* Ukljucuje se prijemni kanal DMA kontrolera */
    GPDMA_ChannelCmd(1, ENABLE);
}
```

Слика 26. Функција за примање пакета UART-ом уз помоћ DMA контролера

8.2.4. Динамичко конфигурисање дужине UART пакета

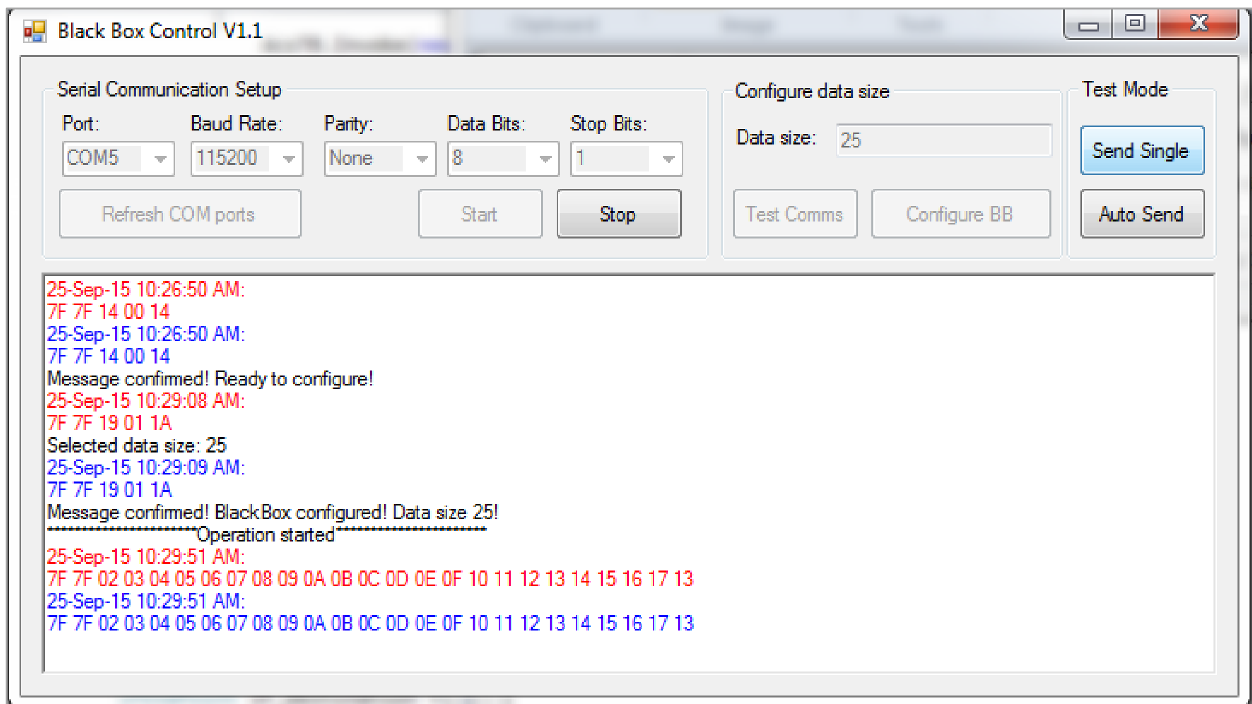
Серијски интерфејс који се прави треба да буде универзални за све црне кутије са истом архитектуром, што значи да сам пакет који се шаље серијском везом зависи од области примене саме црне кутије. Решење које се природно намеће је да се кориснику остави могућност да он сам динамички може да подеси дужину пакета у зависности од својих потреба.

Да би се ово омогућило, оперативни систем се увек буди у конфигурационом режиму. У овом режиму корисник може да тестира саму комуникацију и да подешава дужину пакета. Конфигурациона порука је дужине 5 бајтова, тј. два бајта заглавља, два бајта података и контролна сума. Први бајт података представља дужину пакета од 1 - 255 бајтова, док други бајт представља тип поруке. Тип поруке може бити 0 или 1. Нула представља тестирање комуникације и наредбу црној кутији да врати примљену поруку, док јединица представља конфигурациону поруку и подешава црну кутију за дужину пакета која је прослеђена као први бајт података. Након успешног подешавања, црна кутија ће радити у ехо режиму са изабраном дужином пакета.



Слика 27. Општа структура UART пакета

Сада када је кориснику омогућено да подешава комуникацију према својим потребама, остаје само да се оживи и меморија на самом хардверу.



Слика 28. Конфигурисање дужине пакета у серијској вези

Промена величине пакета који се шаље или прима путем серијске везе, суштински нема никакве везе са UART периферијским уређајем, већ искључиво са DMA контролером. Промена дужине пакета се остварује на следећи начин:

1. По пријему поруке која има адекватну структуру, гасе се канал за примање као и канал за слање порука серијском везом.
2. Онемогућава се прекидање за DMA контролер.
3. Величине бафера за примање и слање се постављају на примљене вредности.
4. DMA канали за примање и слање порука путем серијске везе се поново иницијализују са новим величинама бафера.
5. Омогућава се прекидање за DMA контролер.
6. Укључује се канал за примање порука путем серијске везе.

8.3. Меморија

Као што је већ речено, процесор користи FRAM меморију која користи SPI прокол као интерфејс за комуникацију. Прво што треба да се уради је да се подеси SPI периферијски уређај на изабраном процесору према упутствима произвођача меморије, а затим да се испрограмира сам интерфејс за комуникацију са меморијом.

Уграђена меморија ради у подређеном режиму и може да ради у режимима 0 (CPOL = 0, CPHA = 0) и 3 (CPOL = 1, CPHA = 1). Дужина пакета ка и од меморије је увек 8 битова. Меморија може да ради на фреквенцијама до 40MHz [6].

Да би се подесио SPI периферијски уређај, прво треба пиновима на процесору да се доделе одговарајуће улоге. За SPI протокол потребне су три комуникационе линије, MISO, MOSI и SCLK, као и број линија за избор подређених уређаја једнак укупном броју уграђених меморијских чипова, тј. подређених уређаја. Линије за избор подређених уређаја су обични GPIO пинови који су дефинисани као излазни и који иницијално морају бити постављени на логичку јединицу. У сваком тренутку највише једна линија може бити постављена на логичку нулу, тј. највише један подређени уређај сме да буде активан у сваком тренутку.

```
PINSEL_CFG_Type PinCfg;  /* struktura za podesavanje pinova */


/* Inicijalizacija SPI pinova:
 * P0.15 - SCK1
 * P0.17 - MISO1
 * P0.18 - MOSI1
 */
PinCfg.Funcnum = 2;
PinCfg.OpenDrain = 0;
PinCfg.Pinmode = PINSEL_PINMODE_PULLUP;
PinCfg.Portnum = 0;
PinCfg.Pinnum = 15;                  /* SCK  */
PINSEL_ConfigPin(&PinCfg);
PinCfg.Pinnum = 17;                  /* MISO */
PINSEL_ConfigPin(&PinCfg);
```

```
PinCfg.Pinnum = 18;                  /* MOSI */
PINSEL_ConfigPin(&PinCfg);

/* P0.16 - GPIO
 * selekt linija za memorijski cip 1
 */
PinCfg.Pinnum = 16;
PinCfg.OpenDrain = 0;
PinCfg.Funcnum = 0;
PINSEL_ConfigPin(&PinCfg);

/* P0.19 - GPIO
 * selekt linija za memorijski cip 2
 */
PinCfg.Pinnum = 19;
PinCfg.OpenDrain = 0;
PinCfg.Funcnum = 0;
PINSEL_ConfigPin(&PinCfg);

/* selekt linija za memorijski cip 1 je izlazni pin */
GPIO_SetDir(0, 1<<16, 1);
/* postavlja se vrednost pina na logicku jedinicu */
GPIO_SetValue(0, 1<<16);

/* selekt linija za memorijski cip 2 je izlazni pin */
GPIO_SetDir(0, 1<<19, 1);
/* postavlja se vrednost pina na logicku jedinicu */
GPIO_SetValue(0, 1<<19);
```

Слика 29. Подешавање пинова за SPI периферијски уређај

Након подешавања пинова, потребно је подесити и сам SPI периферијски уређај. SPI периферијски уређај мора да ради у надређеном режиму, број битова података мора бити 8, поларитет сата је 0, фаза сата је исто 0, а изабрана фреквенција је 10 мегахерца. С обзиром на то да је потребно осам битова за пренос податка у меморију, изабрана фреквенција омогућава пренос од преко милион бајтова у секунди. Управо ова чињеница дозвољава обрађивање поруке добијене преко UART периферијског уређаја у главном програму, а не у прекиду.

```
/* Podesavanje SPI komunikacije */

/* inicijalizuje se SPI komunikaciona struktura */
SSP_ConfigStructInit(&SSP_ConfigStruct);
/* 8 bitova se koristi za prenos podataka*/
SSP_ConfigStruct.Databit = SSP_DATABIT_8;
/* procesor je nadredjeni                           */
SSP_ConfigStruct.Mode = SSP_MASTER_MODE;
/* aktivna vrednost sata je logicka jedinica */
SSP_ConfigStruct.CPHA = 0;
/* uzorkovanje se vrsi na prvoj ivici */
SSP_ConfigStruct.CPOL = 0;
/* frekvencija magistrale je 10Mhz */
SSP_ConfigStruct.ClockRate = 10000000;
```

```
/* Inicijalizuje se SPI periferijski uredjaj */
SSP_Init(LPC_SSP0, &SSP_ConfigStruct);
/* Ukljucuje se SPI periferijski uredjaj */
SSP_Cmd(LPC_SSP0, ENABLE);
```

Слика 30. Подешавање и покретање SPI периферијског уређаја

Када се SPI периферијски уређај подеси према упутствима произвођача меморије, остаје још само да се развије интерфејс за комуникацију са меморијом.

8.3.1. Протокол за комуникацију са меморијом

Уграђена меморија ради у реалном времену и кориснику омогућава једноставан интерфејс за комуникацију. Комуникација преко SPI периферијског уређаја контролише се уз помоћ операционих кодова. Операциони кодови представљају команде самој меморији. Операциони кодови су команде дужине 8 бита. Након селектовања подређеног уређаја, први бајт који шаље надређени уређај мора бити операцини код. Након операционог кода, шаљу се пратећи адресни бајтови и бајтови података. Поједини операциони кодови нису праћени адресним бајтовима и бајтовима података. Приликом сваког селектовања подређеног уређаја, могуће је послати искључиво један операциони код. Линија за селектовање мора да постане неактивна (логичка јединица), па затим поново активна (логичка нула) ако желимо да пошаљемо још један операциони код. Детаљније информације потражити у [6].

Након укључивања меморије, неопходно је сачекати 1ms пре него што се пошаље прва команда. Сви трансфери података ка и од меморије иду у групама од по осам битова, синхрони су са сатом и увек се прво преноси бит највеће тежине који је праћен осталим битовима (енг. MSB first).

8.3.1.1. Операциони кодови

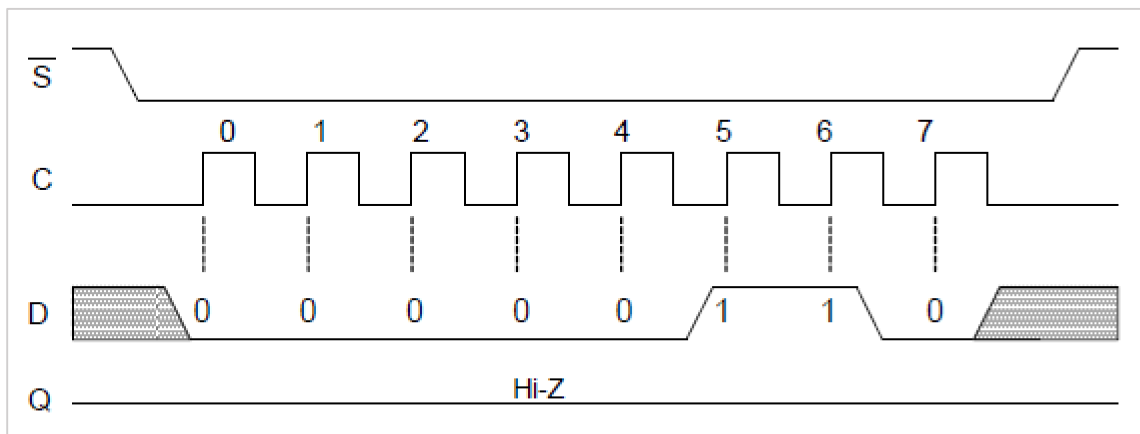
Постоји шест операционих кодова којима се контролише рад меморије. Операциони кодови се могу поделити у три групе. У првој групи су команде које нису праћене адресним бајтовима и бајтовима података. Ови операциони кодови извршавају само једну команду, попут омогућавања операције уписа у меморију. Другој групи припадају команде праћене једним улазним или излазним бајтом. Ове команде служе за интеракцију са статусним регистром унутар меморије. Трећој групи припадају трансакционе команде које поред операционог кода садрже и адресне бајтове који су праћени са једним или више бајтова података.

Табела 4. SPIсак операционих кодова и њихово занчење

Назив	Опис	Вредност
WREN	Омогућава се упис	0000 0110b
WRDI	Онемогућава се упис	0000 0100b
RDSR	Читање статусног регистра	0000 0101b
WRSR	Уписивање у статусни регистар	0000 0001b
READ	Читање података из меморије	0000 0011b
WRITE	Уписивање података у меморију	0000 0010b
SLEEP	Улажење у sleep mode	1011 1001b

Након укључења меморије могућност уписа је увек искључена. Неопходно је послати WREN команду пре било каквог уписа у меморију. Након слања ове команде, корисник је у могућности да шаље нове команде за упис било у меморију (WRITE) било у статусни регистар (WRSR).

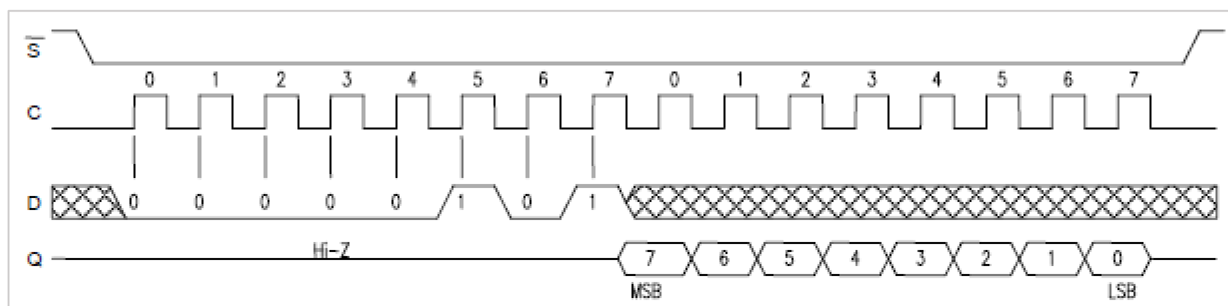
Слање WREN операционог кода изазива постављање WEL бита у статусном регистру на један. Овај бит означава да ли је упис у меморију дозвољен или не. Само постављање WEL бита на један без слања WREN команде нема никаквог ефекта на могућност уписа у меморију. WEL бит се аутоматски ресетује на нулу након подизања селект линије на логичку јединицу. Овакво понашање меморије онемогућава упис у меморију без поновног слања WREN команде.



Слика 31. Изглед SPI линија приликом слања WREN команде

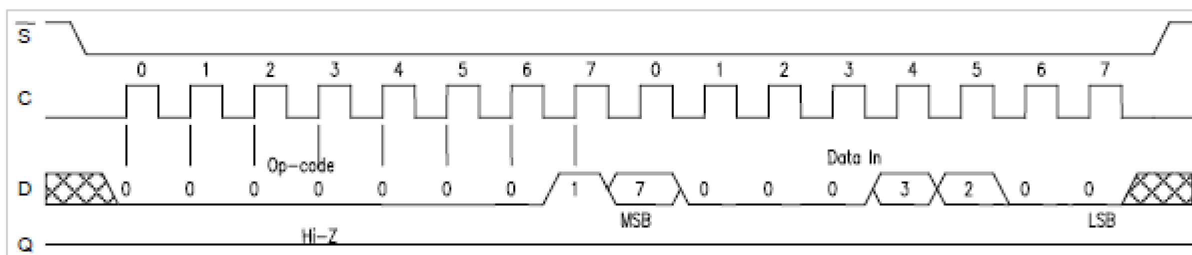
Упис се онемогућује слањем WRDI операционог кода. Након примања ове команде, WEL бит се аутоматски ресетује на нулу.

RDSR команда дозвољава кориснику да провери садржај статусног регистра. Читањем статусног регистра добијају се информације о тренутном стању заштите од писања. Након операционог кода, меморија ће вратити још један бајт са садржајем статусног регистра.



Слика 32. Изглед SPI линија приликом читања статусног регистра

WRSR команда омогућава кориснику да у статусном регистру обележи одређене делове меморије и да на њима укључи заштиту од писања. WRSR команди увек мора да претходи WREN команда да би се писање омогућило.



Слика 33. ИЗглед SPI линија приликом писања по статусном регистру

Бит	7	6	5	4	3	2	1	0
Вредност	0/1	1	0	0	0/1	0/1	0/1	0
Назив	WPEN	/	/	/	BP1	BP0	WEL	/

Слика 34. Структура статусног регистра и опис битова

Заштита од писања се може поставити на сам статусни регистар, али и на делове саме меморије. WPEN бит у статусном регистру контролише блокирање уписа у статусни регистар. Ако је WPEN бит постављен на јединицу, онда је забрањено мењање садржаја статусног регистра. Забрану уписа у делове меморије контролишу битови за заштиту блокова (BP) у оквиру статусног регистра. Као што је раније речено, промена WPEN бита уписом у статусни регистар нема никакав утицај на упис података у меморију, већ ту операцију контролише искључиво WREN команда. Комбинацијама битова BP1 и BP0 постиже се заштита од уписа на делу меморије или на укупној меморије.

Табела 2. Контрола заштите меморије

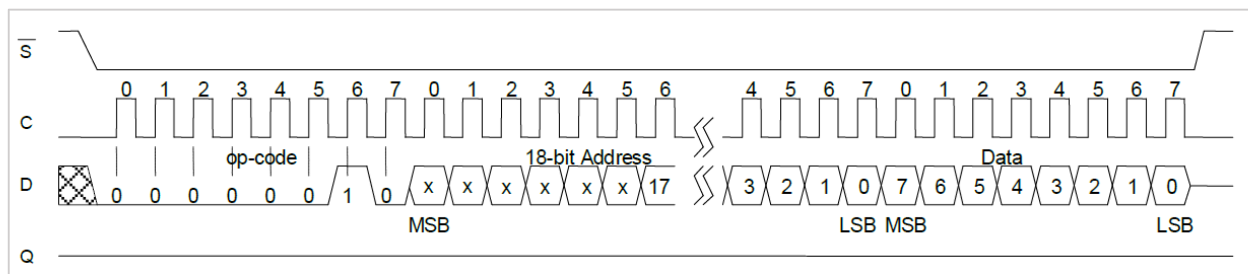
BP1	BP0	Заштићена меморија
0	0	Без заштите
0	1	Горња четвртина
1	0	Горња половина
1	1	Комплетна меморија

8.3.1.2. Операције писања и читања меморије

SPI протокол може да ради на високој фреквенцији, што само наглашава могућности FRAM-а и чињенице да се операције читања и уписа одвијају у реалном времену. Употреба бафера није потребна, јер се све операције одвијају на фреквенцији самог протокола.

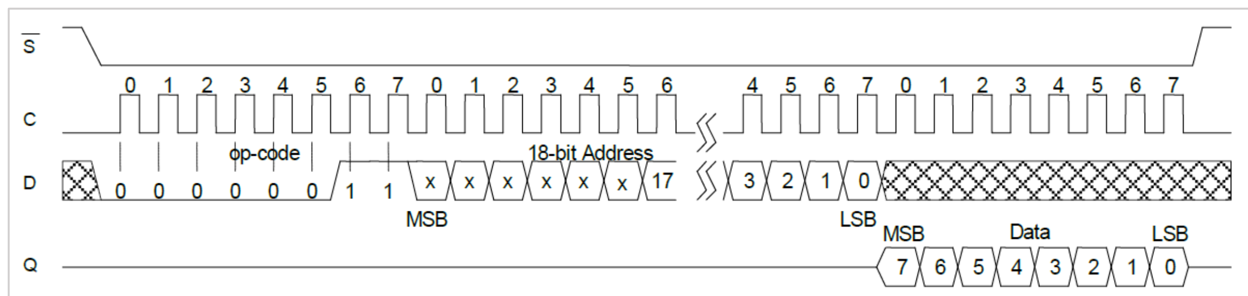
Сви уписи у меморију почињу слањем WREN команде. Следећи операциони код који се шаље је WRITE команда. WRITE команду прати тробајтна адреса на коју ће бити уписан први бајт података. Највиших шест битова у бајту највеће тежине у меморијској адреси се игноришу. Пратећи бајтови су бајтови адресе, и они се уписују секвенцијално. Адресе се интерно инкрементирају све време док надређени уређај генерише сигнал сата. Уколико се достигне највећа адреса, бројач адреса ће се аутоматски ресетовати на нулу.

За разлику од серијског флеша, неограничен број бајтова може да се уписује секвенцијално и сваки бајт се уписује у меморију након сваког осмог откуцаја сата. Растућа ивица линије за селектовање прекида операцију уписа.



Слика 35. Приказ SPI линија приликом уписа података

Након падајуће ивице линије за селектовање подређеног уређаја, надређени може да изда READ команду. Након операционог кода, неопходно је послати тробајтну адресу првог податка који се чита из меморије. Након што се пошаљу операциони код и адреса, меморија игнорише MOSI линију и почиње слање података надређеном преко MISO линије. На сваки откуцај сата, меморија шаље бит података. Након осам битова, адресе се аутоматски интерно инкрементирају. Уколико се достигне највећа адреса, бројач адреса ће се аутоматски ресетовати на нулу. Растућа ивица линије за селектовање прекида операцију читања.



Слика 36. Слика SPI линија приликом читања меморије

Сада остаје само да се испрограмирају све наведене команде и тиме кориснику направи интефејс за интеракцију са меморијом.

8.3.2. Имплементација

Први корак приликом имплементације софтвера за рад са меморијом треба да буде сазнавање укупне доступне количине меморије. Овде се јавља неколико проблема. Први проблем је променљива величина меморијских чипова који се уграђују, што онемогућава тврдо кодирање укупне количине меморије. Други је немогућност промене садржаја меморије и на тај начин је искључено динамичко сазнавања укупе количине меморије. Трећи проблем је непостајање регистра у меморији у коме је произвођач уписао величину чипа. Решење свих ових проблема је договорног типа. Прва четири бајта у меморији ће садржати величину меморије. Приликом уградње чипа на адресе 0, 1, 2 и 3 ће бити уписана количина меморије на уграђеном чипу MSB редоследом. Пети бајт на адреси 4 у меморији садржаће заставицу која говори о томе да ли је у

меморију нешто уписано или не, док ће шести бајт на адреси 5 у меморији садржати величину пакета који је коришћен у претходном упису.

Из свега описаног, јасно је који су кораци неопходни при покретању црне кутије. Први корак је иницијализација UART периферијског уређаја, други корак је иницијализација SPI периферијског уређаја и трећи корак је дохватање информација о уграђеној меморији и статусу уписа из саме меморије.

Да би се добиле информације о меморији, неопходно је да се прочитају меморијске ћелије на адресама 0 – 5. Управо због тога неопходно је да се развију функције које то раде. Општи облик функција за читање меморије гласи:

1. Поставити одговарајући операциони код (READ) у бафер.
2. Поставити меморијску адресу првог бајта податка који се чита у бафер.
3. Креирати структуру за читање и писање.
4. Прочитати податке из сваког уграђеног меморијског чипа..

SPI протокол је синхрони и full duplex, што значи да се подаци истовремено шаљу и примају. Неопходно је да се дефинише улазни и излазни бафер као и дужину пакета који се шаље. Улазни и излазни бафер морају бити исте дужине, иако можда подаци који се шаљу и примају нису једнако дугачки. У случају да је неки од бафера дужи него што је потребно, сав вишак се једноставно игнорише у примљеној/послатој поруци.

Неопходно је послати операциони код дужине једног бајта, затим тробајтну адресу првог податка и тек онда могу да се очекују подаци којих има укупно шест бајтова. Укупно је потребно десет бајтова, што ће на крају и бити дужина бафера.

Комуникација са подређеним уређајима се одвија на следећи начин: прво је потребно спустити линију за селектовање подређеног на логичку нулу, затим покренути периферијски уређај, и на крају подићи линију за селектовање на логичку јединицу. Сама интеракција са меморијом је дефинисана као блокирајућа операција у систему. Једном када крене комуникација са меморијом, немогуће је прекинути је све док се не заврши. Оваквим приступом осигурава се поузданост података која је примарна карактеристика црне кутије.

```
void SPI_GetMemSize(void) {

    /* postavlja se operacioni kod   */
    Tx_Buf[0] = READ_MEMORY;

    /* kreira se adresa prvog podatka koji se cita */
    Tx_Buf[1] = (0 >> 16) & 0xff;
    Tx_Buf[2] = (0 >> 8) & 0xff;
    Tx_Buf[3] = 0 & 0xff;

    /* popunjava se komunikaciona struktura */
    xferConfigStruct.tx_data = Tx_Buf;  /* bafer za slanje  */
    xferConfigStruct.rx_data = Rx_Buf;  /* bafer za primane */
    xferConfigStruct.length = 10;       /* duzina paketa    */
```

```
    /* spusta se selekt linija za prvi memorijski cip */
    GPIO_ClearValue(0, 1<<16);
    /* citaju se podaci iz prvog memorijskog cipa */
    SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
    /* podize se selekt linija za prvi memorijski cip */
    GPIO_SetValue(0,1<<16);

    /* rekonstruisanje primljene poruke. MSB first */
    mem_size_bank1 = (Rx_Buf[7]) | (Rx_Buf[6]<<8) | \
                     (Rx_Buf[5] << 16) | (Rx_Buf[4] << 24);
    mem_size_bank1 -= 1;
    empty_bank1 = Rx_Buf[8];
    package_size1 = Rx_Buf[9];
    cekanje_ms(200);

    /* postavlja se operacioni kod   */
    Tx_Buf[0] = READ_MEMORY;

    /* kreira se adresa prvog podatka koji se cita */
    Tx_Buf[1] = (0 >> 16) & 0xff;
    Tx_Buf[2] = (0 >> 8) & 0xff;
    Tx_Buf[3] = 0 & 0xff;

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;  /* bafer za slanje  */
    xferConfigStruct.rx_data = Rx_Buf;  /* bafer za primane */
    xferConfigStruct.length = 10;       /* duzina paketa    */

    /* spusta se selekt linija za drugi memorijski cip */
    GPIO_ClearValue(0, 1<<19);
    /* citaju se podaci iz drugog memorijskog cipa */
    SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
    /* podize se selekt linija za drugi memorijski cip */
    GPIO_SetValue(0,1<<19);

    /* rekonstruisanje primljene poruke. MSB first */
    mem_size_bank2 = (Rx_Buf[7]) | (Rx_Buf[6]<<8) | \
                     (Rx_Buf[5] << 16) | (Rx_Buf[4] << 24);
    mem_size_bank2 -=1;
    empty_bank2 = Rx_Buf[8];
    package_size2 = Rx_Buf[9];
}
```

Слика 37. Функција за читање статуса меморије

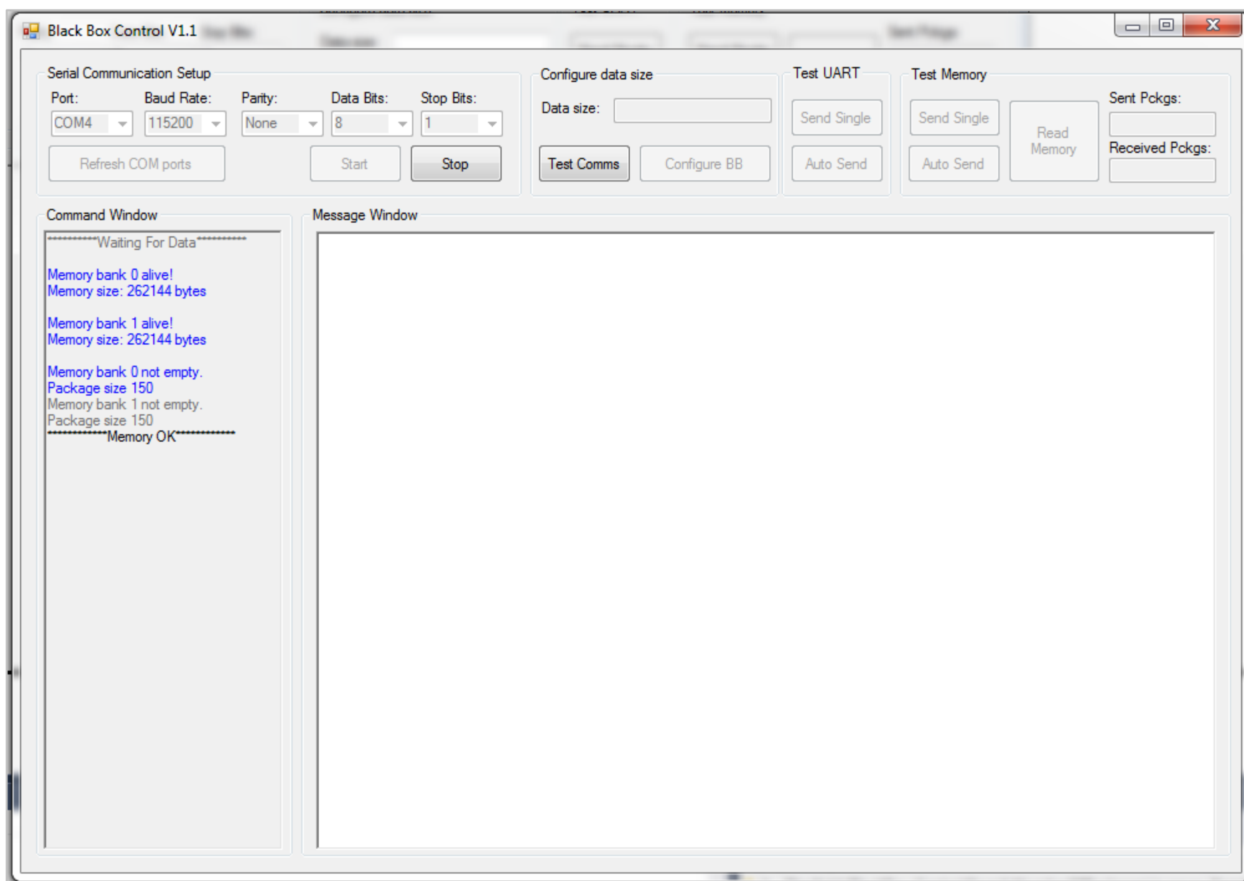
Након што се уз помоћ функције SPI_`GetMemSize()` сазнају сви подаци о стању уграђене меморије, неопходно је обавестити и корисника о томе. Корисник се обавештава тако што му се шаљу три поруке путем серијске везе. Поруке су дужине 10 бајтова, при чему су прва два бајта бајтови заглавља а последњи бајт је контролна сума. Бајт на позицији два се користи као идентификатор поруке и може да буде 0, 1 или 2. Порука са идентификатором 0 саопштава кориснику капацитет првог меморијског чипа, порука са идентификаторм 1 саопштава кориснику капацитет другог меморијског чипа, док порука са идентификатором 2 саопштава кориснику да ли је било претходног уписа у црну кутију и колика је величина пакета у том случају. Формирање и слање порука се

имплементира у функцији са именом `get_msize()`. У складу са наведеним, главни посао сада има следећи изглед:

```
static void App_SerialTask(void* p_arg){

    OS_ERR err;                 /* promenljiva u kojoj se cuvaju greske */
    int status = UNDEFINED;     /* status primljene poruke */

    /* pokrece se UART periferijski uredjaj */
    UART0_DMAInit();

    /* podesava se SPI periferija za komunikaciju sa memorijom */
    SPI_Setup();

    /* posao se pauzira na 1ms */
    OSTimeDlyHMSM(0, 0, 0, 999, OS_OPT_TIME_DLY | OS_OPT_TIME_HMSM_STRICT,\
                  &err);

    /* ceka se dok se ne oslobodi UART periferijski uredjaj */
    OSSemPend(&uart_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, &err);
    /* korisniku se salje status memorije putem serijski veze */
    get_msize();

    /* u beskonacnoj petlji */
    while (DEF_TRUE) {

        /* ceka se sve dok ne stigne nova poruka */
        OSSemPend(&action_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0,\
                  &err);

        /* proverava se pristigla poruku */
        status = check_message();

        /* ako je poruka odgovarajuce strukture */
        if (status == ECHO) {
            /* ceka se dok se ne oslobodi UART uredjaj */
            OSSemPend(&uart_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0,\
                      &err);
            /* primljena poruka se vraca nazad */
            echo();
        }
        else if (status == CONFIG_CMD) {
            /* ako je u pitanju konfiguraciona poruka, menjanju se
             * podesavanja crne kutije
             */
             configure_black_box();
        }
    }
}
```

Слика 38. Читање статусних информација из меморије

Повезивањем црне кутије са рачунаром и покретањем апликације, добијају се следеће информације:



Слика 39. Информације које су досупне по укључењу црне кутије

Наредна ствар коју треба омогућити у систему је ажурирање заставице о статусу уписа у црну кутију као и ажурирање дужине пакета у меморији. Да би се ово постигло прво треба да се дефинише општи облик функција за упис у меморију:

1. Послати WREN команду и омогућити упис.
2. Поставити одговарајући операциони код (WRITE) у бафер.
3. Поставити у бафер меморијску адресу на коју ће се уписати први бајт података.
4. Поставити у бафер комплетан пакет података.
5. Креирати структуру за читање и писање.
6. Уписати податке у сваки уграђени меморијски чип.

За сваки уграђени меморијски чип морају да се изврше свих шест корака наведеним редоследом..

Слање WREN команде је врло једноставно. Довољно је само да у бафер за слање као први бајт поставити WREN команду, подесити бафере у структури за читање и писање и подесити дужину бафера на један. Када се то уради, неопходно је спустити линију за селектовање подређеног, затим активирати периферијски уређај и након слања подићи линију за селектовање. WREN команда се шаље алтернативно прво једном, па другом меморијском чипу.

```
void SPI_Send_Command(int command){


    /* aktivira se memorijski cip koji je na redu */
    if (ind_wr == 0){
        GPIO_ClearValue(0, 1<<16);
    }
    else {
        GPIO_ClearValue(0, 1<<19);
    }

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;
    xferConfigStruct.rx_data = Rx_Buf;
    xferConfigStruct.length = 0x1;

    /* kreira se komanda koja se salje */
    Tx_Buf[0] = command & 0xff;

    /* vrsi se slanje putem SPI */
    SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);

    /* deaktivira se odgovarajuci memorijski cip */
    if (ind_wr == 0){
        GPIO_SetValue(0, 1<<16);
    }
    else {
        GPIO_SetValue(0, 1<<19);
    }
}
```

Слика 40. Функција за слање команди меморији

Након WREN команде шаље се команда за упис. У бафер за слање као први бајт поставља се WRITE команда, коју прати тробајтна адреса за којом иду бајтови података. У овом случају, уписују се само два податка, заставица и величина пакета. Адресе на које се уписују су четири и пет. Као адреса податка поставља се адреса првог бајта који се уписује, тј. поставља се вредност 4 као адреса. Први податак је заставица, а други податак је величина пакета, што значи да је дужина нашег трансфера 6. Када се све ово подеси, довољно је да се спусти селект линија, активира периферијски уређај и након слања подигне селект линија, што ће резултовати променама података у меморији.

```
void SPI_SetEmptyFlag(int flag, int pkg_size) {

    /* prvo se salje komanda za omogucavanje upisa */
    SPI_Send_Command(WRITE_ENABLE);

    /* postavlja se operacioni kod */
    Tx_Buf[0] = WRITE_MEMORY;

    /* kreira se adresa za upis prvog podatka */
    Tx_Buf[1] = (0 >> 16) & 0xff;
    Tx_Buf[2] = (0 >> 8) & 0xff;
```

```
    Tx_Buf[3] = 4 & 0xff;

    /* upisuje se prosledjeni flag */
    Tx_Buf[4] = flag;
    /* upisuje se prosledjena velicina paketa */
    Tx_Buf[5] = pkg_size;

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;  /* bafer za slanje   */
    xferConfigStruct.rx_data = Rx_Buf;  /* bafer za primanje */
    xferConfigStruct.length = 6;        /* duzina paketa     */

    /* spusta se selekt linija za prvi memorijski cip */
    GPIO_ClearValue(0, 1<<16);
    /* podaci se upisuju u memoriju */
    SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
    /* podize se selekt linija za prvi memorijski cip */
    GPIO_SetValue(0, 1<<16);

    /* drugi memorijski cip se postavlja kao aktivni */
    ind_wr = !ind_wr;

    /* prvo se salje komanda za omogucavanje upisa */
    SPI_Send_Command(WRITE_ENABLE);

    /* postavlja se operacioni kod */
    Tx_Buf[0] = WRITE_MEMORY;

    /* kreira se adresa za upis prvog podatka */
    Tx_Buf[1] = (0 >> 16) & 0xff;
    Tx_Buf[2] = (0 >> 8) & 0xff;
    Tx_Buf[3] = 4 & 0xff;

    /* upisuje se prosledjeni flag */
    Tx_Buf[4] = flag;
    /* upisuje se prosledjena velicina paketa */
    Tx_Buf[5] = pkg_size;

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;  /* bafer za slanje   */
    xferConfigStruct.rx_data = Rx_Buf;  /* bafer za primanje */
    xferConfigStruct.length = 6;        /* duzina paketa     */

    /* spusta se selekt linija za drugi memorijski cip */
    GPIO_ClearValue(0, 1<<19);
    /* podaci se upisuju u memoriju */
    SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
    /* podize se selekt linija za drugi memorijski cip */
    GPIO_SetValue(0, 1<<19);

    /* prvi memorijski cip se postavlja kao aktivni */
    ind_wr = !ind_wr;
}
```

Слика 41. Функција за ажурирање заставица у меморији

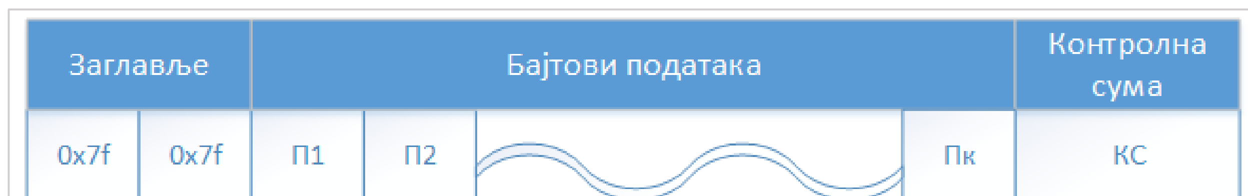
Сада када је кориснику омогућена елементарна интеракција са меморијом на ред долазе читање и писање стварних података. Приликом укључења црне кутије, црна кутија ће кориснику послати тренутни статус. Тренутни статус обухвата број уграђених меморијских чипова, њихов капацитет, заставицу који говори о томе да ли је било уписа или не, као и величину пакета из претходне конфигурације. Ове информације се шаљу кориснику, да не би несвесно обрисао већ постојећи садржај у меморији.

Након почетног конфигурисања црне кутије и избора величине пакета, кориснику треба омогућити упис, читање и тестирање комуникационог канала. Да би се ово омогућило на већ усвојено заглавље и контролну суму у пакету додаје се још један контролни бајт. Контролни бајт је претпоследњи бајт у пакету и има улогу у контроли уписа у црну кутију. Контролни бајт може да има следећа значења:

Табела 3. Контролни бајт и значења битова

Вредност	Значење
0	Тест комуникације. Ехо сервис
3	Упиши пакет у меморију
5	Ресетуј црну кутију
9	Обриши црну кутију (без ресета)
17	Прочитај један податак из меморије
33	Прочитај комплетну меморију
Било који ненаведени број	Одбацивање свих порука, грешка у систему

Тест комуникације је могућ независно од стања напајања у систему, док је за било коју другу операцију неопходно да бит напајања буде један. У супротном, сваки облик интеракције са црном кутијом је онемогућен.



Слика 42. Општа структура UART пакета

Из структуре пакета се види да дужина пакета који се шаље UART-ом мора да буде сума броја бајтова који се снимају (Б), бајтова заглавља (2), контролног бајта (1) и контролне суме (1). Дакле, укупна дужина пакета је број бајтова који се снимају плус 4.

Након конфигурисања црне кутије, свака пристигла порука изазива прекид главног посла и постављање одговарајуће заставице. Главни посао одлучује у ком тренутку ће обрадити пристиглу поруку. Када се прими исправна порука у зависности од контролне заставице, контрола над системом се препушта одговарајућем послу. У случају да контролни бајт има неку вредност која није наведена у таблици, одбацују се све поруке, јер примљене вредност сигнализира грешку у систему.

Први корак је имплементација уписа и читања једног пакета у меморији. С обзиром на то да на систему најчешће постоји већи број меморијских чипова, два приступа су могућа. Један приступ је уписивање сваког исправног пакета у сваки меморијски чип или уписивање пакета у чипове алтернативно, при чему сваки пакет иде у само један чип. Прва опција делује боља са стране поузданости података, али знатно смањује уграђени капацитет меморије, а тиме и укупно време снимања. Друга опција смањује поузданост података, али знатно повећава време снимања. Предност прве опције је то што је отпорна на евентуална оштећења чипова, јер подаци имају најмање једну резервну копију. Иако није апсолутна отпорна на оштећења чипова, разне примене црних кутија могу да компензују наведену слабост другог приступа. Снимања са сензора се углавном раде са великом учестаношћу, најчешће много већом него што природне појаве то могу да испрате, па губитак једног чипа није страшан јер интерполацијом и даље могу да се добију више него употребљиви подаци. Кориснику се оставља могућност приликом конфигурисања црне кутије да одлучи како ће се снимати подаци. Иницијално, биће развијен систем који уписује податке алтернативно.

8.3.2.1. Функција за упис пакета у меморију

У оквиру самог система сваки меморијски чип има одвојену променљиву која означава адресу на коју треба уписати пакет. Ова променљива се користи да би се избегло незгодно својство меморије да аутоматски ресетује свој интерни бројач када се дође до највеће адресе. Приликом сваког уписа пакета проверава се да ли пакет може да стане у меморију, тј. да ли се стигло до краја меморије. У случају да не може, уписивање ће бити онемогућено за тај меморијски чип.

Када пристигне исправна порука са контролним бајтом који наређује упис у меморију, контрола се препушта послу који извршава упис. Посао прво прилагођава поруку упису, тј. из пакета се избацују заглавље и контролна сума који се не памте у меморији. Посао затим шаље WREN команду одговарајућем меморијском чипу, и подешава пакет за упис. Прво се постави WRITE команда, затим адреса првог бајта у меморији коју прате бајтови података. Очигледно, укупна дужина пакета који се шаље SPI протоколом је команда (1) + адреса(3) +подаци(Б -3). Након слања пакета, инкрементира се системска меморијска адреса чипа у који је извршен упис, заставица се поставља за упис у наредни чип, и враћа се контрола главном процесу. Посао који врши упис у меморију је блокирајући и није га могуће прекинути, чиме се осигурава поузданост уписа података. Упис у меморију је могућ једино ако је вредност контролног бајта 0x3, чиме је кориснику омогућена апсолутна контрола над радом саме црне кутије.

```
int SPI_TransmitData(unsigned char* res_data) {

    int i;  /* brojac u petlji */

    /* prvo se salje komanda za omogucavanje upisa */
    SPI_Send_Command(WRITE_ENABLE);

    /* postavlja se odgovarajuci operacioni kod */
    Tx_Buf[0] = WRITE_MEMORY;
    /* kreiranje memorijske adrese od koje krece upis podataka */
    if (ind_wr == 0) {
```

```
    /* ako u memoriji nema vise mesta, kontrola se vraca glavnom programu*/
        if (memaddr_bank1 + T_DATA_SPI > mem_size_bank1)
            /* vraca se nula kao signal za neuspeh */
            return 0;
        /* u suprotnom, adresa se upisuje u paket */
        Tx_Buf[1] = (memaddr_bank1 >> 16) & 0xff;
        Tx_Buf[2] = (memaddr_bank1 >> 8) & 0xff;
        Tx_Buf[3] = memaddr_bank1 & 0xff;
        /* pokazivac se postavlja na sledecu slobodnu adresu za upis */
        memaddr_bank1 += T_DATA_SPI;
    }
    else {
    /* ako u memoriji nema vise mesta, kontrola se vraca glavnom programu*/
        if (memaddr_bank2 + T_DATA_SPI > mem_size_bank2)
            /* vraca se nula kao signal za neuspeh */
            return 0;
        /* u suprotnom, adresa se upisuje u paket */
        Tx_Buf[1] = (memaddr_bank2 >> 16) & 0xff;
        Tx_Buf[2] = (memaddr_bank2 >> 8) & 0xff;
        Tx_Buf[3] = memaddr_bank2 & 0xff;
        /* pokazivac se postavlja na sledecu slobodnu adresu za upis */
        memaddr_bank2 += T_DATA_SPI;
    }
  /* podaci se kopiraju iz bafera glavnog programa u SPI bafer za slanje */
    for (i = 0; i < T_DATA_SPI; i++)
        Tx_Buf[i+4] = res_data[i];

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;              /* bafer za slanje   */
    xferConfigStruct.rx_data = Rx_Buf;              /* bafer za primanje */
    xferConfigStruct.length = SPI_BUF_SIZE_TX2;     /* duzina paketa
*/

    /* podaci se upisuju u odgovarajuci memorijski cip */
    if (ind_wr == 0) {
        /* spusta se selekt linija za prvi memorijski cip */
        GPIO_ClearValue(0, 1<<16);
        /* podaci se upisuju u memoriju */
        SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
        /* podize se selekt linija za prvi memorijski cip */
        GPIO_SetValue(0, 1<<16);
    }
    else {
        /* spusta se selekt linija za drugi memorijski cip */
        GPIO_ClearValue(0, 1<<19);
        /* podaci se upisuju u memoriju */
        SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
        /* podize se selekt linija za drugi memorijski cip */
        GPIO_SetValue(0, 1<<19);
    }
    /* naredni memorijski cip se postavlja kao aktivni */
    ind_wr = !ind_wr;
    /* povratna vrednost koja signalizira uspesan upis */
    return 1;
}
```

Слика 43. Функција за упис пакета у меморију

8.3.2.2. Функција за читање једног пакета из меморије

Поред променљивих за чување тренутне адресе на коју се уписује пакет, у оквиру оперативног система за сваки меморисјки чип постоји и променљива која чува меморијску адресу са које се чита пакет. Променљива је уведена из истих разлога као и она за памћење адреса за упис. Приликом сваког читања, проверава се да ли се стигло до краја меморије и да ли може да се прочита нови пакет. У случају да се стигло до краја меморије, онемогућава се даље читање тог меморијског чипа. У супротном, пакет се чита и инкрементира се променљива која садржи вредност адресе са које је прочитан пакет.

Када пристигне исправна порука са контролним бајтом који наређује читање једног пакета из меморије, контрола се препушта послу који извршава читање. Посао прво подешава пакет за комуникацију са меморијом, тако што као први бајт поставља READ команду, затим тробајтну адресу првог податка који се чита из меморије, и на крају постави одговарајућу дужину пакета. Очигледно, укупна дужина пакета који се чита SPI протоколом је команда (1) + адреса(3) +подаци(Б -3). Након примања пакета, инкрементира се системска меморијска адреса чипа из кога је извршено читање, заставица се поставља за читање из наредног чипа, и враћа се контрола главном послу. Посао који врши читање из меморије је блокирајући и није га могуће прекинути, чиме се осигурава поузданост читања података.

```
void SPI_GetData(void) {

    /* postavljanje operacionog koda */
    Tx_Buf[0] = READ_MEMORY;

    /* kreira se adresa sa koje treba procitati podatak */
    if (ind_rd == 0) {
        Tx_Buf[1] = (memread_bank1 >> 16) & 0xff;
        Tx_Buf[2] = (memread_bank1 >> 8) & 0xff;
        Tx_Buf[3] = memread_bank1 & 0xff;
        /* pokazivac se postavlja na naredni slobodan paket u cipu */
        memread_bank1 += T_DATA_SPI;
    }
    else {
        Tx_Buf[1] = (memread_bank2 >> 16) & 0xff;
        Tx_Buf[2] = (memread_bank2 >> 8) & 0xff;
        Tx_Buf[3] = memread_bank2 & 0xff;
        /* pokazivac se postavlja na naredni slobodan paket u cipu */
        memread_bank2 += T_DATA_SPI;
    }

    /* popunjava se struktura za slanje putem SPI */
    xferConfigStruct.tx_data = Tx_Buf;          /* bafer za slanje   */
    xferConfigStruct.rx_data = Rx_Buf;          /* bafer za primanje */
    xferConfigStruct.length = SPI_BUF_SIZE_TX2; /* duzina paketa     */

    /* podaci se citaju iz odgovarajuceg memorijskog cipa */
    if (ind_rd == 0) {
```

```
        /* spusta se selekt linija za prvi memorijski cip */
        GPIO_ClearValue(0, 1<<16);
        /* podaci se citaju iz memorije */
        SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
        /* podize se selekt linija za prvi memorijski cip */
        GPIO_SetValue(0, 1<<16);
    }
    else {
        /* spusta se selekt linija za drugi memorijski cip */
        GPIO_ClearValue(0, 1<<19);
        /* podaci se citaju iz memorije */
        SSP_ReadWrite(LPC_SSP0, &xferConfigStruct, SSP_TRANSFER_POLLING);
        /* podize se selekt linija za drugi memorijski cip */
        GPIO_SetValue(0, 1<<19);
    }
    /* naredni memorijski cip se postavlja kao aktivni */
    ind_rd = !ind_rd;
}
```

Слика 44. Фунцкија за читање пакета из меморије

Након што су подаци прочитани и контрола враћена главном послу, примљени пакет мора да се прилагоди пакету за слање UART периферијским уређајем. Примљеном пакету се додаје заглавље и контролна сума која се претходно израчуна. Тако модификовани пакет се шаље UART-ом уз употребу DMA контролера.

Операција уписа у меморију ће се одвијати све док постоји слободно место за упис у неком од меморијских чипова. С обзиром на то да је операција уписа контролисана од стране корисника, дефинисањем функција за упис у меморију кориснику је омогућен интерфејс за памћење података.

8.3.2.3. Посао за аутоматско читање меморије

Читање је још увек изузетно напорно, јер корисник мора да пошаље нову команду за сваки пакет који жели да прочита. Природно је увођење аутоматске операције читања меморије, при чему је довољно да корисник пошаље само једну команду и да покрене читање комплетне црне кутије. Да би се то остварило довољно је да корисник пошаље исправну поруку у којој контролни бајт има вредност 33.

Када оперативни систем прими поруку којој контролни бајт има вредност 33, контрола се препушта послу за аутоматско читање црне кутију. Прво се покреће тајмер, који на сваких 40ms поставља заставицу и обавештава посао да је време да прочита и пошаље нову поруку. Аутоматско читање се извршава све док може да се прочита пакет из било ког меморијског чипа. Аутоматско читање се реализује секвенцијалним покретањем процеса за читање једног пакета у унапред задатим временским интервалима које одређује тајмер. Када се пакет прочита, прво се прилагођава пакету за слање преко UART периферијског уређаја. Пакету се додају заглавље, израчунава се контролна сума и тек онда се пакет шаље преко UART периферијског уређаја. Тајмер је неопходан, јер је SPI комуникација много бржа од UART комуникације, па би у супротном дошло до затрпавања UART периферијског уређаја новим пакетима пре него што је успео да пошаље стари, што коначно доводи до заглављивања система. По окончању слања тајмер се гаси, и вредности променљивих које чувају адресе за читање се ресетују на почетак да би се омогућило ново читање црне кутије.

Посао за аутоматско читање црне кутије је блокирајући, али га је могуће прекинути. Тајмер прекида процес чим прође 40ms и поставља му заставицу, чиме га обавештава да може да прочита и пошаље нови пакет. Једини део процеса који не може да се прекине је сама операција читања меморије, чиме се осигурава интегритет података.

8.3.2.4. Посао за брисање меморије

Приликом покретања меморије, корисник би требало да има могућност брисања садржаја меморије да би јасно одвојио снимљене податке од осталог садржаја меморије. Очигледно, брисање меморије треба омогућити и у случају када је црна кутија конфигурисана и када није. Брисање меморије најлакше може да се уради тако што се системске адресе за упис ресетују на почетне вредности. Међутим, овај приступ није добар, јер не врши физичко брисање меморије већ само означава да су ћелије слободне за упис.

Празна меморија по договору на сваком бајту има уписану вредност $0xff$, па се брисање меморије своди на уписивање тих вредности на сваком бајту. Као ознака празне меморије може да се искористи било која константа. Разлог за то је врло једноставан. Не постоје констатни сигнали у природи, па било која константна вредност може да сигнализира празну меморију.

Уколико пристигла порука у контролном бајту садржи постављен бит на позицији три, контролу треба препустити послу за брисање меморије. Посао за брисање прво ресетује меморијске адресе за упис у меморију и затим секвенцијално креће са уписом вредности $0xff$ на сваком бајту у меморији. Упис се извршава све док се не стигне до краја меморије. Када се стигне до краја меморије, адресе за упис у меморију се поново ресетује на почетак и контрола се враћа главном послу. По добијању контроле, главни процес кориснику шаље поруку о успешно извршеној операцији брисања.

Процес за брисање меморије је блокирајући, али га је могуће прекинути. Једини део процеса који је немогуће прекинути је упис вредности у меморију. Упис у меморију се извршава секвенцијално без пауза, јер меморија ради у реалном времену и на фреквенцији на којој ради сам SPI периферијски уређај.

8.3.2.5. Ресетовање црне кутије

Све наведене могућности система подразумевају да црна кутија може да се конфигурише једино приликом паљења, што није добро у случајевима када се црна кутија користи у системима који не дозвољавају гашење. У том случају неопходно је да систем има могућност поновног конфигурисања без гашења. Уколико контролни бајт има постављен бит на позицији два, црна кутија ће се вратити назад на конфигурациони режим, тј. дужина UART поруке ће се вратити на подразумеваних 10 бајтова и оперативни систем ће очекивати команду за успостављење нових комуникационих пакета. Фунцкија за ресетовање је прекидајућа и нема никакву интеракцију са меморијом, већ само ресетује дужине пакета на подразумеване и поново подешава DMA контролер и UART периферијски уређај на почетне вредности.

8.4. Надгледање напајања

До сада развијени систем има једну озбиљну ману када је његова употреба у питању. Сама црна кутија подразумева да има сопствено напајање и подразумева да је централни контролни уређај већ испитан и у потпуности функционалан, као и сви остали уређаји у систему. Очигледно, мана је управо потреба за потпуном испитаношћу комплетног уређаја. Прва ствар која мора да се испита

приликом дизајнирања било ког уређаја је стабилност напајања свих виталних компоненти. Управо зато црна кутија има могућност да снима стање напајања 12 независних компоненти система, као и стање сопственог напајања. Такође, сама црна кутија може да има два извора напајања. Црна кутија може да има своје независно напајање издвојено од остатка система, али може и да добија напајање директно од централног контролног уређаја.

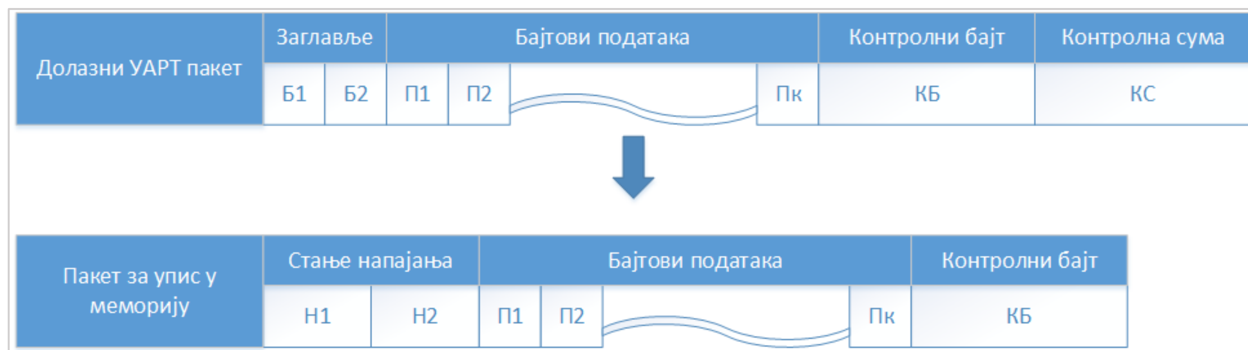
Добијање напајања од другог уређаја иницијално делује бесмислено, али је врло корисно уколико је саставни део експеримента испитивање поузданости самог контролног уређаја. Уколико се током експеримента догоди квар на контролном уређају и он изгуби напајање, престаће и слање пакета црној кутији као и напајање црне кутије. У том тренутку црна кутија би требало да буде свесна да је напајање престало и то би требало да упише у меморију. Да би се овакво понашање осигурало, црна кутија на себи има уграђен кондензатор великог капацитета који ради као батерија приликом губитка напајања.

8.4.1. Пакет за упис у меморију

Црна кутија константно чита стања напајања 12 независних уређаја као и тринаесто, тј. своје напајање. Читања напајања се једноставно омогућава употребом 13 GPIO пинова. Да би се пинови користили на овај начин морају при паљењу система да буду дефинисани као улазни пинови и неопходно је да унапред буде дефинисано подразумевано стање на пину. Пинови који раде надгледање напајања компоненти у систему имају подразумевану вредност један, ако нема напајања. У случају да напајања има, вредност на пину ће бити нула. На пину који ради надгледање напајања црне кутије, ситуација је другачија. У случају да црна кутија има напајање вредност на пину ће бити један, а у супротном нула.

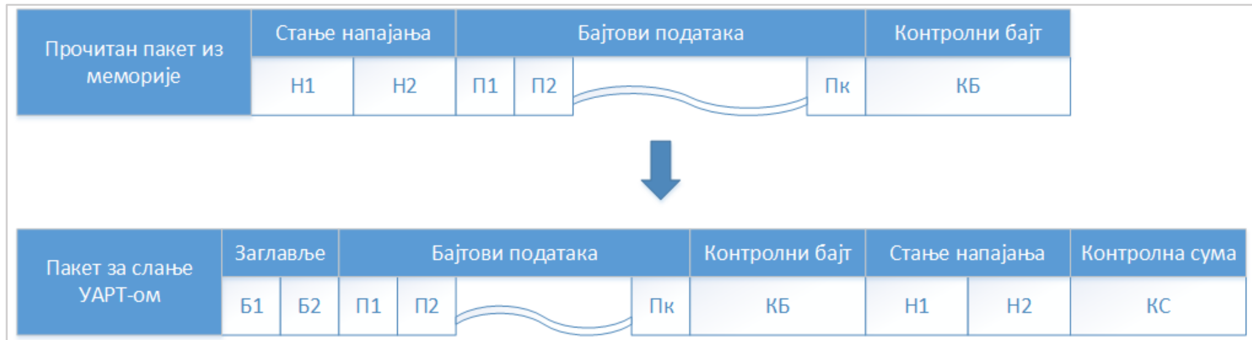
С обзиром на то да напајања може или да буде или да не буде, довољан је по један бит за сваки од праћених уређаја. Да би се искодирала стања потребно су два бајта, при чему постоје три бита вишка, које ће се једноставно игнорисати. До сада су се из UART пакета избацивали заглавље и контролна сума, и такав скраћени пакет се памтио у меморији. Сада поред поменутих бајтова података морају да се памте и стање напајања. Ово се најлакше постиже тако што се уместо избацивања два бајта заглавља на њихово место поставе подаци о напајањима.

Дакле, пакет за упис у меморију се креира тако што се прва два бајта из пакета пристиглог UART поруком замене са подацима о стању напајања, која прате сами подаци из пакета. Очигледно, пакет који се уписује у меморију је за један бајт краћи од пакета пристиглог UART поруком, јер се не памти само контролна сума.



Слика 45. Трансформација UART пакета приликом уписа у меморију

Наравно, сада се намеће питање како да се подаци о напајању проследе кориснику приликом читања меморије. Решење је очигледно. Прочитани садржај пакета из меморије се модификује тако што се подаци о напајању пребаце на крај поруке. На почетак поруке се враћају два бајта заглавља, праћених бајтовима података и контролним бајтом, за којом иду два бајта са стањем напајања. На крају, прерачунава се контролна сума која се додаје на крај пакета, тј. иза бајтова напајања. Пакет који се шаље кориснику је за два бајта дужи од оног пакета који је примљен.



Слика 46. Трансформаицја причитаног пакета из меморије приликом слања UART-ом

Све наведене измене не захтевају никакву структурну промену система, већ само промену у начину прилагођавања пакета за упис у меморију и пакета за слање кориснику приликом читања меморије. Упис у меморију се и даље догађа сама када корисник пошаље поруку која у контролном бајту садржи вредност 0x3.

8.4.2. Прекид или прозивање уређаја

Остало је само да се реши шта се дешава приликом губитка напајања у систему. Као што је речено, губитак напајања некако мора да буде записан у меморију. Комплетан упис у меморију мора да буде најкраћи могућ, јер кондензатор омогућава функционисање црне кутије још само делић секунде након престанка регуларног напајања. Због тога, у меморију се уписује само један бајт. По договору, бајт ће се уписивати на највишу адресу у меморији. Вредност која се уписује приликом престанка напајања је $0xAA$. Након што се заврши конфигурација црне кутије, на највишу адресу меморије ће бити уписана вредност $0x55$ као сигнал да напајање у систему постоји.

Сада постоји комплетно разрађен поступак који треба да се одвије након престанка напајања, али и даље није дефинисан начин откривања да је напајање престало. Очигледно, окидач за наведене операције треба да буде пад вредности на пину на који је доведено напајање црне кутије. Природно, прво решење је подешавање прекида који треба да се окине на падајућој ивици сигнала напајања на одговарајућем пину. У самој функцији која обрађује прекид неопходно је да се изврши упис у меморију. Овај приступ одступа од досадашње праксе максимално кратких функција за обраду прекида, јер је овде преостало време за извршавање изузетно кратко и зато што је ово последња операција која треба да се изврши у систему.

```
/* aktiviranje prekida na pinu 0.22 (napajanje crne kutije) */

/* brise se prekid */
GPIO_ClearInt(0, 1<<22);
```

```
/* ukljucuje se prekid na padajucoj ivici */
GPIO_IntCmd(0, 1<<22, 1);
/* brise se prekid */
GPIO_ClearInt(0, 1<<22);

/* aktivira se prekidanje na pinu u vektoru prekida */
CSP_IntSrcCfg(CSP_INT_CTRL_NBR_MAIN, CSP_INT_SRC_NBR_EINT_03, 0,
(CSP_OPT)0);
/* registruje se funkcija za obradu prekida na pinu */
CSP_IntVectReg(CSP_INT_CTRL_NBR_MAIN, CSP_INT_SRC_NBR_EINT_03,
EINT3_IRQHandler, (void*)0);
/* omogucava se prekidanje na pinu za nadgledanje napajanja */
CSP_IntEn(CSP_INT_CTRL_NBR_MAIN, CSP_INT_SRC_NBR_EINT_03);
```

Слика 47. Подешавање прекидања на падајућој ивици пина за напајање

```
void EINT3_IRQHandler(void) {

    /* ako se desio prekid napajanja */
    if (GPIO_GetIntStatus(0, 22, 1)) {

        /* brise se flag */
        GPIO_ClearInt(0, 1<<22);

        /* upisuje se gubitak napajanja u memoriju */
        SPI_SetFlag(AP_POWER_LOST);

        /* upisuje se indikator greske u privremenu memoriju */
        *AP_power = 0xAAAAAAAA;
    }
}
```

Слика 48. Функција за обраду прекида на општем пину

Након програмирања и тестирања прекида напајања, испоставља се да ово решење ипак не ради. Проблем је у томе што самом контролеру треба до 125μs за обраду прекида. Под обрадом прекида подразумева се детектовање падајуће ивице, промена контекста, и препуштање контроле функцији за обраду прекида. Додатни проблем је и нечистоћа самог улазног сигнала, па је ниво који је потребан за прекид знатно већи, што коначно доводи до тога да процесор не стигне да упише података о престанку напајања у меморију.

Решење које је изнуђено је стална провера стања пина у празном послу. Пин за напајање црне кутије се константно прозива након у празном послу. Прозивање пина у тајмерском прекиду или било ком другом прекиду доводи до истих проблема као када се постави прекид на падајућој ивици самог пина. Додатни проблем тајмерског прекида је и чињеница да такви прекиди функционишу у еквидистантним интервалима, што доводи до још веће могућности за пропуштање тренутка губитка напајања.

Прозивање пина након сваке наредбе, повећава искоришћеност процесора, јер је празан посао тај који највише времена проводи као активан. Тестови су показали да је време реакције на губитак напајања неупоредиво брже него када се постави прекид на падајућој ивици сигнала на пину. У

случају када се користи прозивање црна кутија стиже да упише заставицу о губитку напајања на предвиђено место у меморији.

```
void OS_IdleTask(void* p_arg) {

    M3_SR_ALLOC();  /* alocira se mesto na steku za
                     * statusni registar
                     */
    /* u petlji */
    while (DEF_ON) {
        /* u kriticnoj sekciji */
        M3_CRITICAL_ENTER();
        /* uvecava se brojac prolazaka kroz prazan posao */
        OSIdleTaskCtr++;
        /* napusta se kriticna sekcija */
        M3_CRITICAL_EXIT();

        /* ukoliko je doslo do gubitka napajanja */
        if ((GPIO_ReadValue(0) & (1<<22)) == 0) {

            /* upisuje se gubitak napajanja u memoriju */
            SPI_SetFlag(AP_POWER_LOST);

            /* blokira se dalje izvrsavanje */
            while (1);
        }
    }
}
```

Слика 49. Прозивање пина за напајање у празном послу

8.5. Структура, организација и синхронизација послова

8.5.1. Покретање оперативног система

Да би извршавање програма било коректно и понашање оперативног система очекивано, неопходно је извршити следеће кораке у оквиру main функције:

1. Иницијализовати сам процесор и подесити одговарајућу фреквенцију процесора.
2. Онемогућити све екстерне прекиде.
3. Алоцирати и иницијализовати све контролне блокове послова који су потребни.
4. Креирати све послове корисничке послове који су неопходни на почетку извршавања.
5. Позвати функцију `OSInit()` којом се иницијализују све интерне променљиве самог оперативног система (листе послова, таблице приоритета, семафори, катанци, бројач угнежђености прекида, глобални бројачи семафора и катанаца...).
6. Иницијализовати системски сат (енг. SysTick) и омогућити прекидање системског сата.
7. Покренути сам оперативни систем позивом функције `OSStart()`.

Неопходно је испоштовати наведену процедуру да би се осигурало коректно извршавање програма и распоређивање корисничких послова.

8.5.2. Дефиниција свих послова

Оперативни систем дефинише два посла која су кључна за правилно извршавање програма. Један од тих послова је већ поменути празни посао (`OS_IdleTask`), а други је посао који се покреће са сваким прекидом системског сата (`OS_TickTask`). `OS_TickTask` чека блокиран на семафору све док му функција за обраду прекида системског сата не пошаље сигнал и пусти га да прође кроз семафор. У случају да је оперативни систем покренут, посао ће проћи кроз листу послова који чекају и пребациће у листу спремних послова све послове којима је истекло време чекања. У случају да оперативни систем није у стању извршавања, неће се догодити ништа.

Поред ова два системски дефинисана посла, дефинише се још шест послова. Први посао који се дефинише је посао за обраду серијске комуникације и контролу црне кутије. Приликом покретања посла, иницијализују се серијска комуникација и DMA контролер, као и SPI периферијског уређаја за комуникацију са меморијом и тајмер за контролу слања пакета. У главној петљи, посао чека на семафору све док му функција за обраду прекида DMA контролера не сигнализира да је стигла нова порука. По пријему нове поруке, посао прелази у листу спремних и по добијању контроле над процесором проверава структуру примљене поруке. У складу са структуром примљене поруке посао препушта контролу одговарајућим пословима за комуникацију са меморијом или извршава подешавања саме црне кутије. Посао за котролу серијске комуникације има приоритет 10, и као такав има најнижи приоритет од свих кориснички дефинисаних послова, чиме се осигурава да не може да прекине ни један од послова који користе меморију.

```
while (DEF_TRUE) {

    OSSemPend(&action_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, &err);

    status = check_message();

    if (status == ECHO_CMD) {
        OSSemPost(&uart_semaphore, OS_OPT_1, &err);;
        echo();
    }
    else if (status == WRITE_PKG_CMD) {
        OSSemPost(&write_pkg_semaphore, OS_OPT_1, &err);
    }
    else if (status == READ_ALL_CMD) {
        OSSemPost(&read_all_semaphore, OS_OPT_1, &err);
    }
    else if (status == READ_SINGLE_CMD) {
        OSSemPost(&read_pkg_semaphore, OS_OPT_1, &err);
    }
    else if (status == RESET_CMD) {
        reset_black_box();
    }
    else if (status == DELETE_CMD) {
        OSSemPost(&delete_semaphore, OS_OPT_POST_1, &err);
    }
    else if (status == CONFIG_CMD) {
        configure_black_box();
    }
```

```
    else if (status == CURR_CONFIG_CMD) {
        OSSemPend(&uart_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, \
                  &err);
        get_msize();
    }
    else if (status == BIT_CMD) {
        OSSemPost(&bit_semaphore, OS_OPT_POST_1, &err);
    }
    else if (status == CHKSUM_FAILED) {
        OSSemPend(&uart_semaphore, 0, OS_OPT_PEND_BLOCKING, (M3_TS*)0, \
                  &err);
        send_message(CHKSUM_FAILED);
    }
}
```

Слика 50. Главна петља посла за контролу серијске везе

Други посао који се креира је посао за упис само једног пакета у меморију. Посао чека на семафору у главној петљи, све док му посао за контролу серијске комуникације не пошаље сигнал да је пристигла порука одговарајуће садржине. По проласку кроз семафор, посао уписује примљени пакет у меморију коришћењем SPI периферијског уређаја и поново прелази у стање чекања. Посао за упис података у меморију има приоритет 3.

```
static void App_WritePkgTask(void* p_arg) {

    OS_ERR err;

    while (DEF_TRUE) {
        OSSemPend(&write_pkg_semaphore, 0, OS_OPT_PEND_BLOCKING, \
                  (M3_TS*)0, &err);

        write_data_to_memory();
    }
}
```

Слика 51. Посао за упис једног пакета у меморију

Трећи посао који се дефинише је посао за читање једног пакета из меморије. Посао чека на семафору у главној петљи, све док му посао за контролу серијске комуникације не пошаље сигнал да је пристигла порука одговарајуће садржине. По проласку кроз семафор, посао чита само један пакет из меморије коришћењем SPI периферијског уређаја, прилагођава прочитани пакет за слање серијском везом и након што се UART периферијски уређај ослободи, шаље прочитани пакет путем серијске везе. Посао за читање података из меморију има приоритет 5.

```
static void App_ReadPkgTask(void* p_arg) {

    OS_ERR err;

    while (DEF_TRUE) {
        OSSemPend(&read_pkg_semaphore, 0, OS_OPT_PEND_BLOCKING, \
                  (M3_TS*)0, &err);

        read_single_package_from_memory();
    }
}
```

Слика 52. Посао за читање једног пакета из меморије

Четврти посао који се дефинише је посао за читање целокупног садржаја меморије. Посао чека на семафору у главној петљи, све док му посао за контролу серијске комуникације не пошаље сигнал да је пристигла порука одговарајуће садржине. По проласку кроз семафор, посао у петљи чита редом све пакете из меморије. Приликом сваког уласка у петљу, посао чека на семафору да га тајмер обавести да је протекло довољно времена између два слања и да нови пакет може да се пошаље. Након што прочита комплетан садржај меморије посао излази из петље и на семафору чека нови сигнал од посла за контролу серијске везе. Посао за читање свих пакета из меморије има приоритет 4.

```
static void App_ReadAllTask(void* p_arg) {

    OS_ERR err;

    while (DEF_TRUE) {
        OSSemPend(&read_all_semaphore, 0, OS_OPT_PEND_BLOCKING, \
                  (M3_TS*)0, &err);

        read_all_packages_from_memory();
    }
}
```

Слика 53. Посао за читање свих пакета из меморије

Пети посао који се дефинише је посао за брисање целокупног садржаја меморије. Посао чека на семафору у главној петљи, све док му посао за контролу серијске комуникације не пошаље сигнал да је пристигла порука одговарајуће садржине. По проласку кроз семафор, посао ће на свакој меморијској ћелији уписати вредност $0xff$, којом се означава празна ћелија. Након што се заврши брисање посао прелази у стање чекања. Посао за брисање целокупне меморије има приоритет 6.

```
static void App_DeleteTask(void* p_arg) {

    OS_ERR err;

    while (DEF_TRUE) {
        OSSemPend(&delete_semaphore, 0, OS_OPT_PEND_BLOCKING, \
                  (M3_TS*)0, &err);

        delete_memory_content();
    }
}
```

Слика 54. Посао за брисање целокупног садржаја меморије

Шести посао који се дефинише је посао за самостално тестирање црне кутије. Посао чека на семафору у главној петљи, све док му посао за контролу серијске комуникације не пошаље сигнал да је пристигла порука одговарајуће садржине. По проласку кроз семафор, у свако кораку петље посао ће на меморијску локацију уписати неки бајт који ће затим одмах прочитати. Уколико се поклапају уписани и прочитани бајтови на свим ћелијску локацијама тест се проглашава успешним и црна кутија исправном, у супротном тест се проглашава неуспешним. Након проласка кроз све меморијске ћелије и утврђивања исправности самог уређаја, посао ће кориснику послати поруку о утврђеном серијским путем и прећи у стање чекања.

```
static void App_BITTestTask(void* p_arg) {

    OS_ERR err;

    while (DEF_TRUE) {
        OSSemPend(&bit_semaphore, 0, OS_OPT_PEND_BLOCKING, \
                  (M3_TS*)0, &err);

        built_in_test();
    }
}
```

Слика 55. Посао за самостални тест црне кутије

8.5.3. Синхронизација послова и контрола извршавања

Из свега раније наведеног, види се да су комплетно извршавање и контрола црне кутије контролисани споља, тј. све зависи од команди које шаље централна јединица. У случају да нема команди једини посао који је активан је празан посао, па ће се све време само испитивати напајање црне кутије. OS_TickTask ће бити активиран са сваким прекидом који изазове системски сат, али његово извршавање неће имати никаквог смисла јер нема послова који чекају да истекне неко време. Тек по пријему поруке одговарајуће садржине од централне јединице може да крене извршавање кориснички дефинисаних процеса.

Након пријема поруке путем серијске везе, прво се покреће посао за обраду серијске комуникације који ће уколико има потребе покренути остале послове. Овакав начин извршавања је могућ, јер су сви послови синхронизовани уз помоћ семафора којима се контролише и гарантује коректно извршавање. Поред семафора, правилно извршавање послова је осигурано и одговарајућим избором приоритета. Послови за рад са меморијом имају већи приоритет од посла за рад са

серијском комуникацијом, чиме се осигурава да ни у једном случају долазак нове команде не може да прекине или поремети извршавање претходне. Такође, посао за упис података у меморију је посао са највећим приоритетом, чиме не постоји шанса да дође до пребацивања контекста приликом уписа у меморију. Упис података у меморију је најзначајнија операција у целом систему, и коректност уписа мора да буде гарантована.

Након пријема нове поруке серијским путем, примљени подаци се смештају у бафер главног програма који касније користи посао за упис података у меморију. С обзиром да је овај бафер дељени ресурс, приступ се осигурава катанцем (енг. mutex). Сваки посао који жели да приступи овом баферу, прво мора да сачека на катанцу да се ослободи. Након употребе бафера, посао је дужан да ослободи катанац и омогући другим пословима несметано коришћење ресурса. На исти начин, катанцем се штити и бафер у који се смештају подаци прочитани из меморије. Након читања података из меморије, ти подаци морају да се сместе у бафер за слање серијском везом. С обзиром да неки други посао може да користи серијску везу независно од посла који чита пакете из меморије, неопходно је заштити бафер за слање катанцем да би се осигурао интегритет података.

9. Употреба

Црна кутија се увек приликом паљења налази у конфигурационом режиму. Након подешавања свих периферијских уређаја, црна кутија UART-ом шаље стање претходне конфигурације. Поруком се шаље количина меморије у сваком уграђеном меморијском чипу, заставица да ли је црна кутија празна или не, као и величина пакета приликом претходног уписа.

Након примања информација од црне кутије, кориснику су на располагању пет команди. Прва команда је команда за поновно слање конфигурационих параметара, друга је команда за тестирање комуникације, односно ехо сервис, трећа команда је команда за брисање садржаја у меморији, четврта је за читање претходног садржаја у случају да црна кутија није празна, а пета команда а представља конфигурациону поруку за конфигурисање црне кутије за предстојеће експерименте.

У случају да корисник пошаље команду за брисање, по завршетку операције брисања црна кутија ће ресетовати све потребне заставице и послати кориснику повратну поруку са потврдом о завршетку операције брисања. Док траје операција брисања, све поруке од стране корисника ће бити игнорисане. Нормалан рад се наставља тек након потврде о завршетку брисања.

Ако је у питању команда за читање, црна кутија ће подесити периферијске уређаје за дужину пакета која је претходно уписана у меморији. Након подешавања, комплетна меморија ће бити ишчитана секвенцијалано и пакети ће редом бити слани UART-ом. По завршетку читања, црна кутија ће вратити дужину пакета на конфигурациону и послати кориснику поруку о завршетку операције читања. Док траје операција читања све долазне поруке од корисника ће бити игнорисане. Нормалан рад се наставља тек након потврде о завршетку читања.

Корисник је дужан да црној кутији пошаље конфигурациону поруку пре него што крене са уписом. Конфигурациона порука садржи дужину пакета у предстојећем експерименту и режим рада, алтернирајући или клон режим. Након успешног пријема конфигурационе поруке, црна кутија ће послати повратну поруку са потврдом пријема, и подесиће све периферијске уређаје за рад са изабраном дужином пакета. Након слања повратне поруке, црна кутија је конфигурисана за снимање предстојећег експеримента.

Корисник на располагању има неколико команди. Прва команда је команда за проверу комуникације, односно ехо сервис. Друга команда је команда за упис у меморију. Приликом пријема првог пакета за упис, црна кутија ће ажурирати заставице и означити да је било уписа у тренутној сесији. Корисник мора да води рачуна о учестаности слања. Максимални проток је 11520 бајтова у секунди, и у складу са тим корисник мора да подеси своје слање. Уколико се шаље са већом учестаношћу, долази до губитка података услед недовољне брзине комуникационог канала. Четврта команда је команда за брисање меморије. Након успешног брисања меморије црна кутија ће послати повратну поруку и обавестити корисника да може да настави са нормалним радом. Током брисања, сви долазни пакети се игноришу. Читање црне кутије је омогућено уз помоћ две команде. Једна команда наређује црној кутији да из меморије прочита први следећи пакет и тиме омогућава читање меморије пакет по пакет, док друга команда наређује црној кутији да прочита комплетну меморију. Док траје читање сви долазни пакети се игноришу. Последња команда је команда за ресетовање црне кутије, тј. враћање црне кутије у конфигурациони режим. По пријему те поруке црна кутија ће ресетовати све потребне заставице, бафере и меморијске адресе, поново

ће вратити сва подешавања периферијских уређаја на подразумевана и послаће кориснику повратну поруку са потврдом успешног ресетовања.

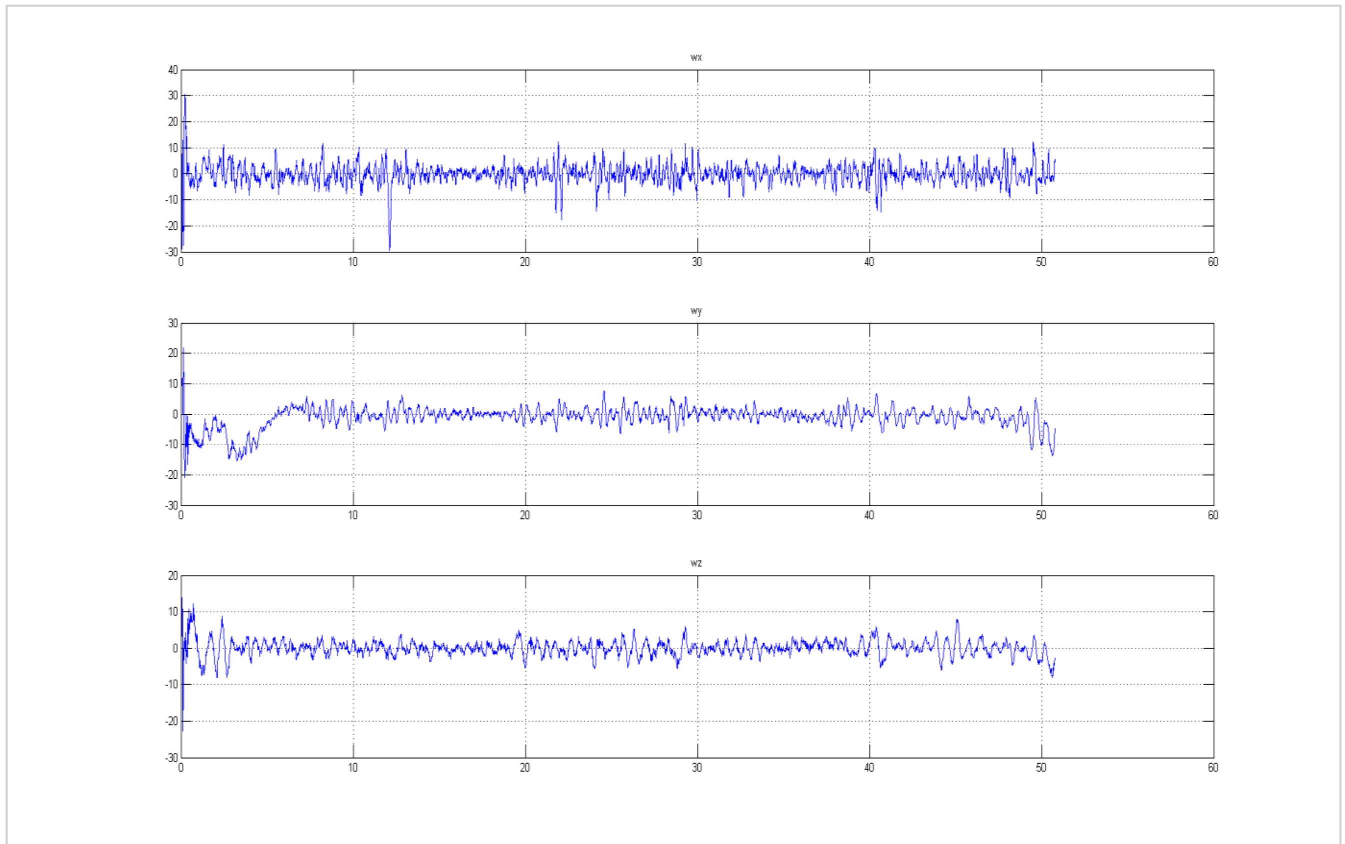
Приликом употребе црне кутије корисник би требало негде у пакету да има бројач пакета или протекло време, да би се омогућило лакше тумачење снимљених података. На следећој слици приказано је место црне кутије у систему као и општи начин повезивања.

10. Резултати из екперимената

Црна кутија је тестирана у великом броју експеримената, што у лабораторијским што у реалним условима експлоатације. На наредних неколико графика биће приказани подаци снимљени у стварној примени црне кутије. Биће приказани подаци снимљени током полигонског испитивања две INS (енг. Inertial Navigation System) навођене ракете кратког домета у које су биле уграђене црне кутије. Такође, биће приказан и снимак у лабораторијским условима којим је тестиран комуникациони канал између црне кутије и контролне јединице навигационог уређаја. На свим графицима су приказани подаци снимљени у меморији црне кутије, а за чије мерење и израчунавање су одговорни уграђени сензори и контролна јединица. Све недостајуће информације приликом описа графика изостављене су из разлога њихове поверљивости.

На наредним сликама приказани су подаци снимљени током лета INS навођене ракете. Црна кутија је преживела удар и накнадно је прочитана. На свим графицима x оса представља протекло време, а y оса представља измерене вредности. У пакету који се снима у црну кутију поред измерених вредности памти се и протекло време.

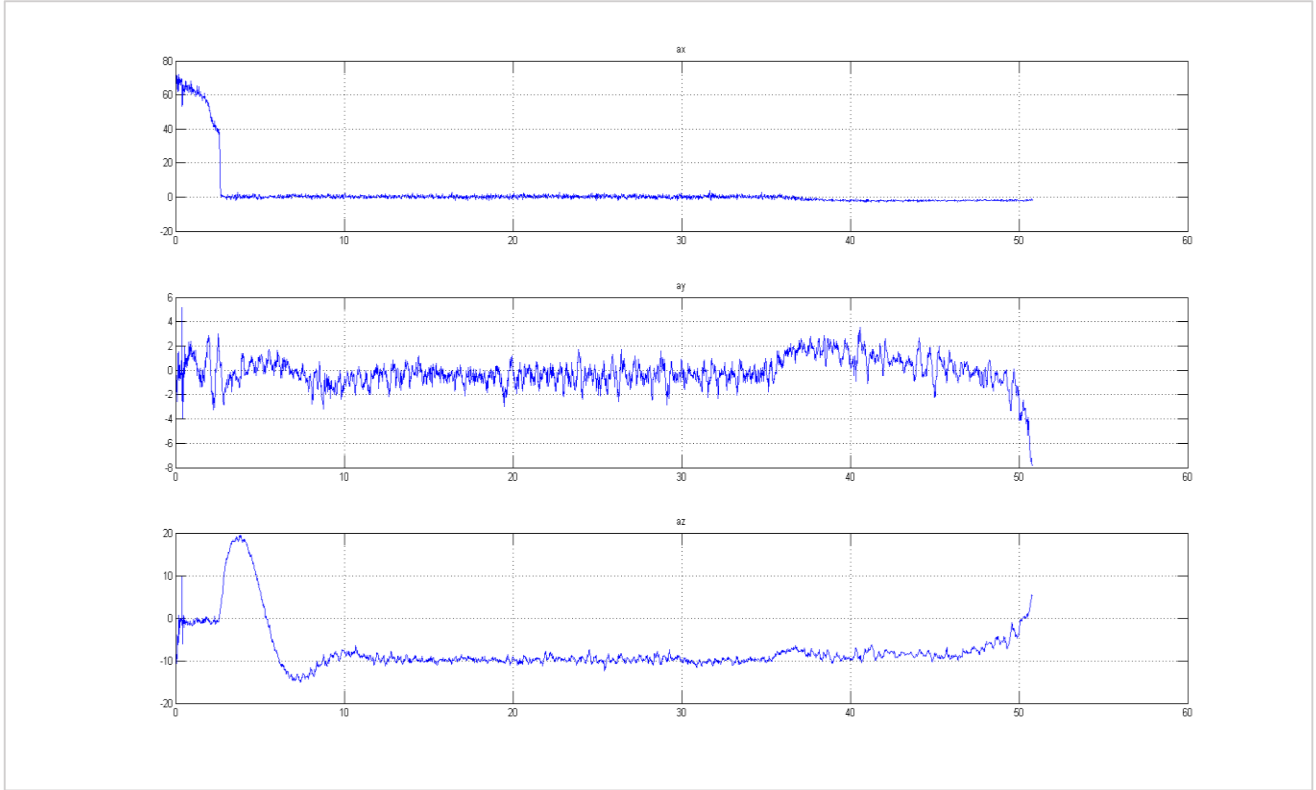
На слици 56. приказане су вредности угаоних брзина измерених уграђеним троосним жироскопом. Приказани графици редом представљају вредности измерене око прве, друге и треће осе сензора.



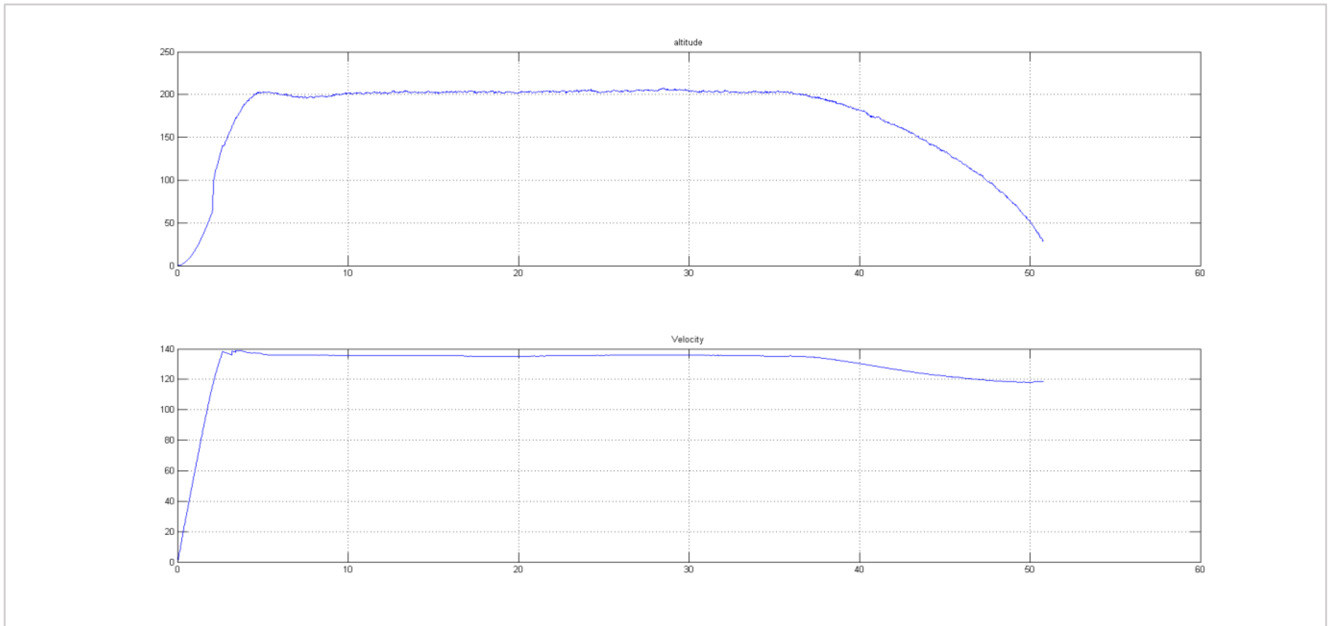
Слика 56. Угаоне брзине измерене уграђеним жироскопима

На слици 57. приказане су вредности угаоних убрзања измерених уграђеним троосним акцелерометром. Приказани графици редом представљају вредности убрзања измерених око прве,

друге и треће осе сензора. Није тешко закључити да измерена угаона убрзања представљају маневре које ракета извршава приликом стабилизације.



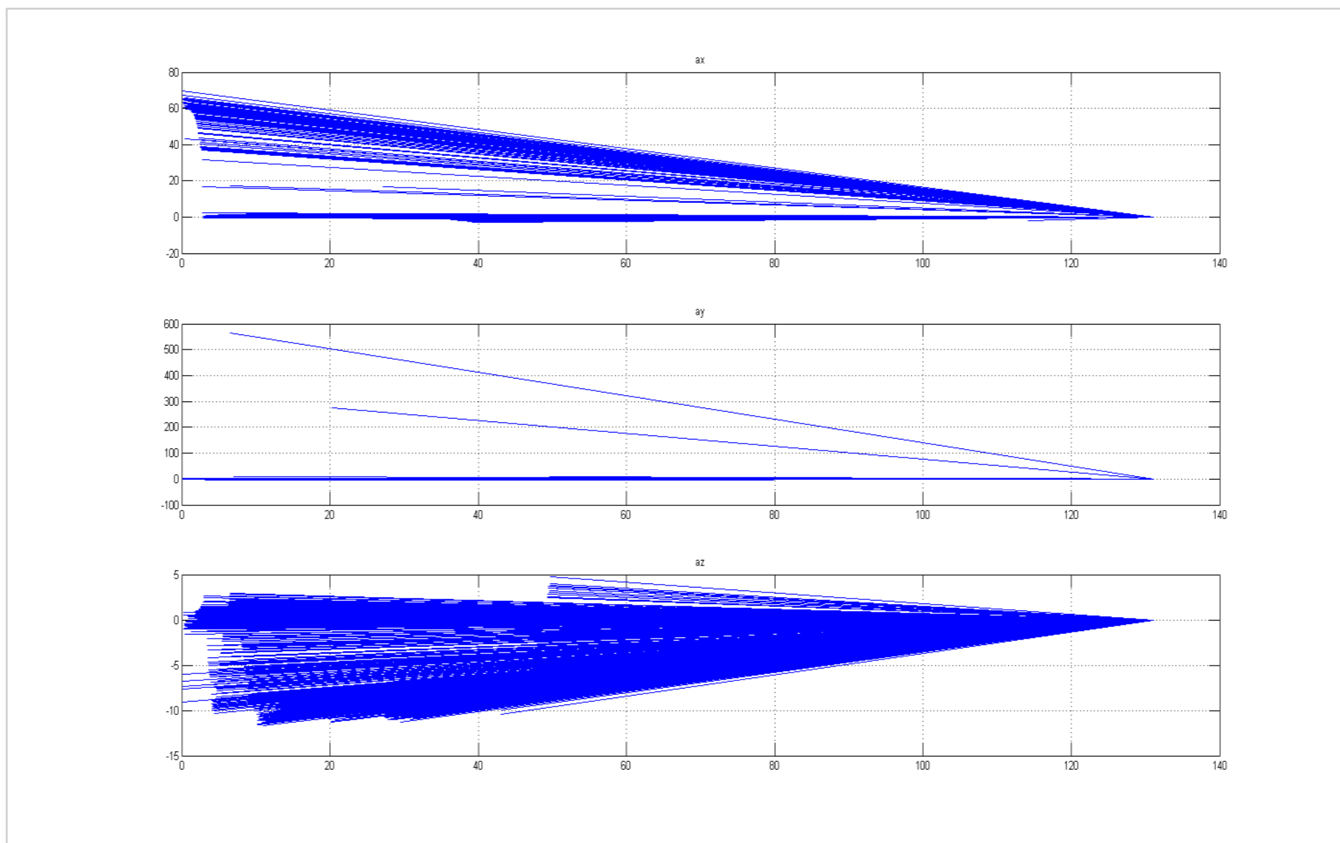
Слика 57. Угаона убрзања измерена уграђеним акцелерометрима



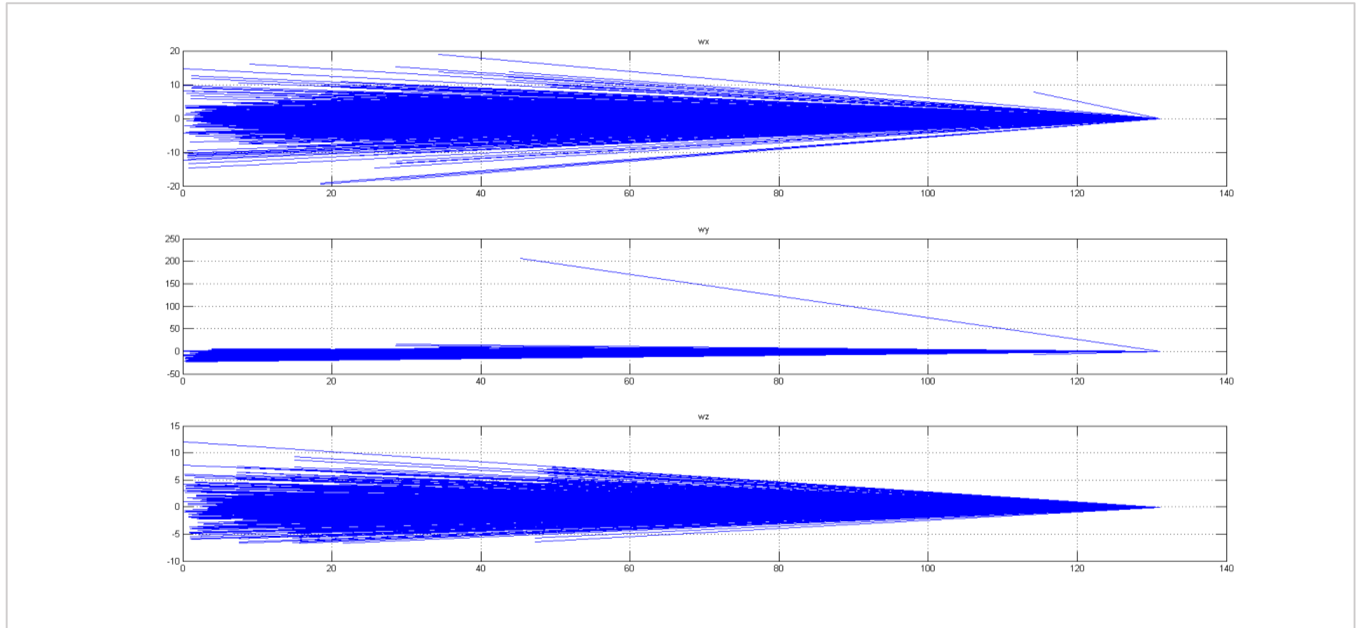
Слика 58. Синтетисане висина и брзина објекта на основу вредности са мерних уређаја

На слици 58. приказану су два графика чије се вредности израчунавају у реалном времену на основу вредности које сензори измере током лета. Први график представља висину ракете у сваком тренутку лета, а други график представља израчунату брзину ракете. Из добијених снимака се лако уочава да је пројектил прилично стабилан током лета, да је брзина прилично уједначена током целог лета као и да је упадни угао прилично велики.

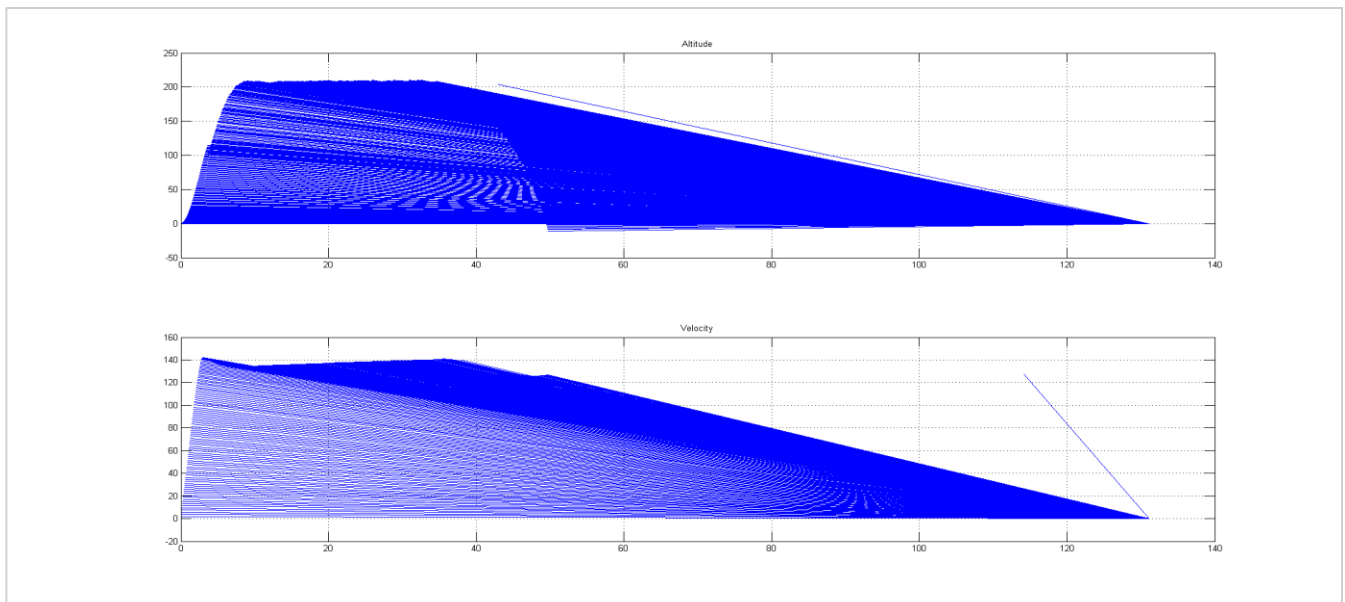
Следећи снимак је такође настао приликом полигонског испитивања ракета, али црна кутија је приликом удара у земљу била оштећена. Након читања података, добијени су следећи дијаграми:



Слика 59. Угаоно убрзања итмерена уграђеним акцелерометрима



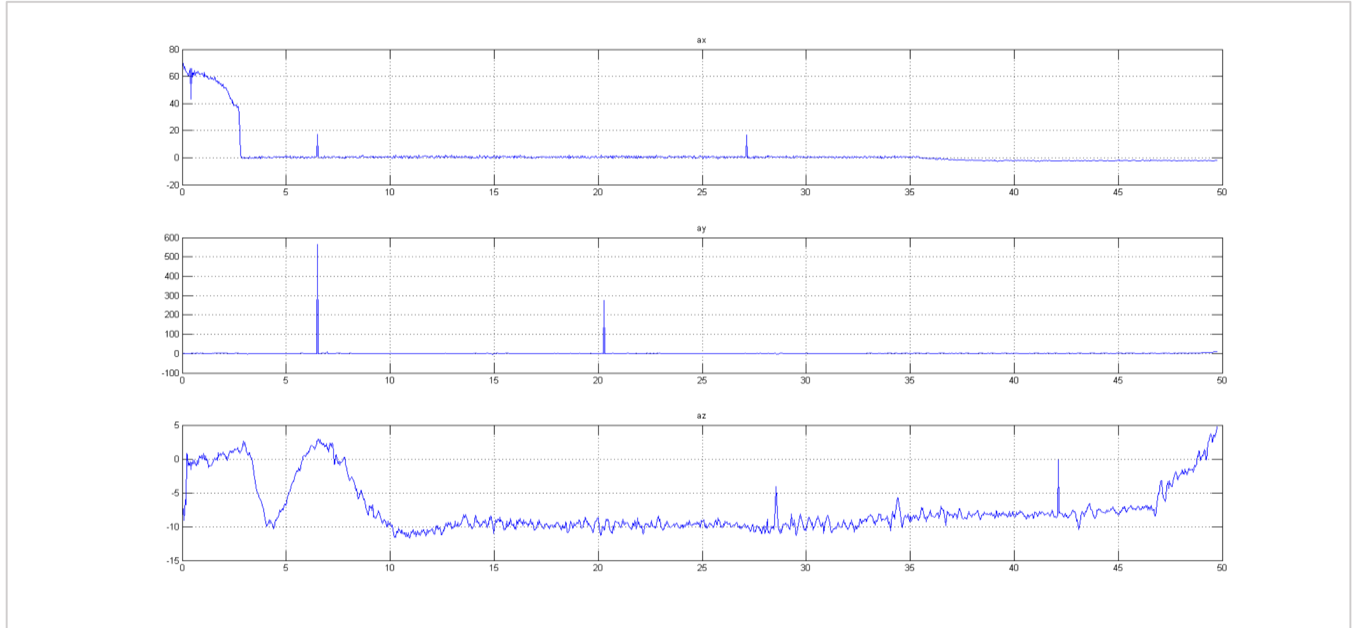
Слика 60. Угаоне брзине измерене уграђеним жироскопима



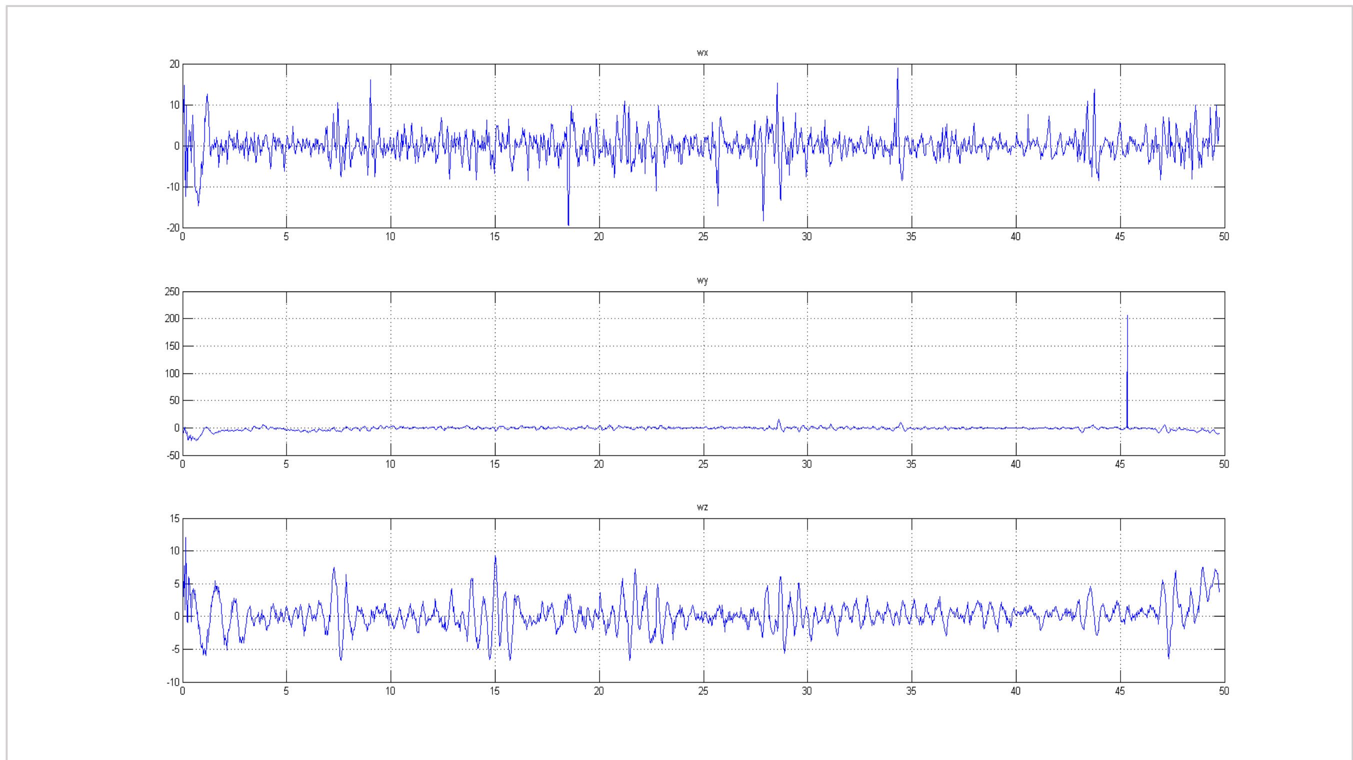
Слика 61. Синтетисане висина и брзина објекта на основу вредности са мерних уређаја

Гледајући добијене дијаграме, очигледно је да су бесмислени. Овакви подаци нису могући у реалном свету. Прва ствар која је могућа јесте да се догодила грешка у комуникацији, али систем провере самих пакета и команди је прилично сложен, па је ова опција мало вероватна. Друга могућност је да су чипови приликом удара оштећени, што је врло вероватно с обзиром на то да је црна кутија била сломљена. Детаљнијом анализом, уочава се да прве половине графика брзине и висине делује врло блиско реалности, тако да је могуће да је оштећен само један чип, а да други ипак ради. Да би се тестирала исправност првог чипа, приликом читања црне кутије избачени су сви парни пакети, јер они одговарају другом меморијском чипу. Након читања, добијени су следећи

графици. На слици 62. приказане су вредности угаоних убрзања измерених уграђеним троосним акцелерометром. Приказани графици редом представљају вредности убрзања измерених око прве, друге и треће осе сензора. Није тешко закључити да измерена угаона убрзања представљају маневре које ракета извршава приликом стабилизације.

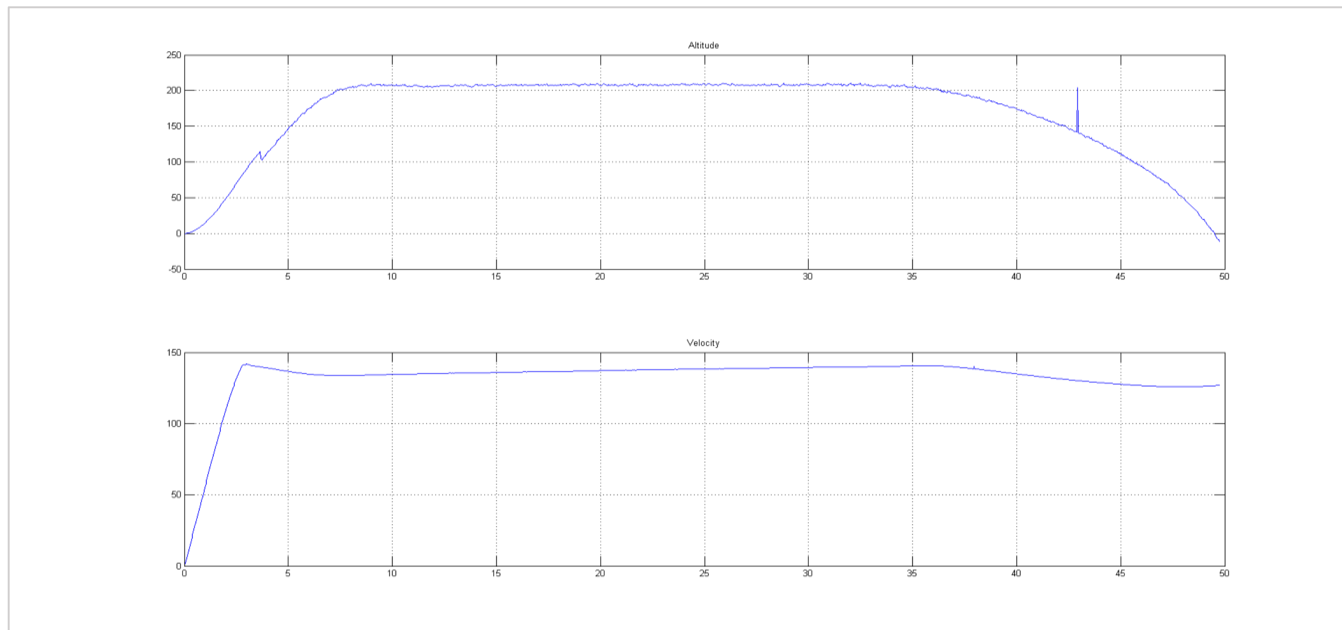


Слика 62. Угаона убрзања измерена уграђеним акцелерометрима



Слика 63. Угаоне брзине измерене уграђеним жироскопима

На слици 63. приказане су вредности угаоних брзина измерених уграђеним троосним жироскопом. Приказани графици редом представљају вредности измерене око прве, друге и треће осе сензора.



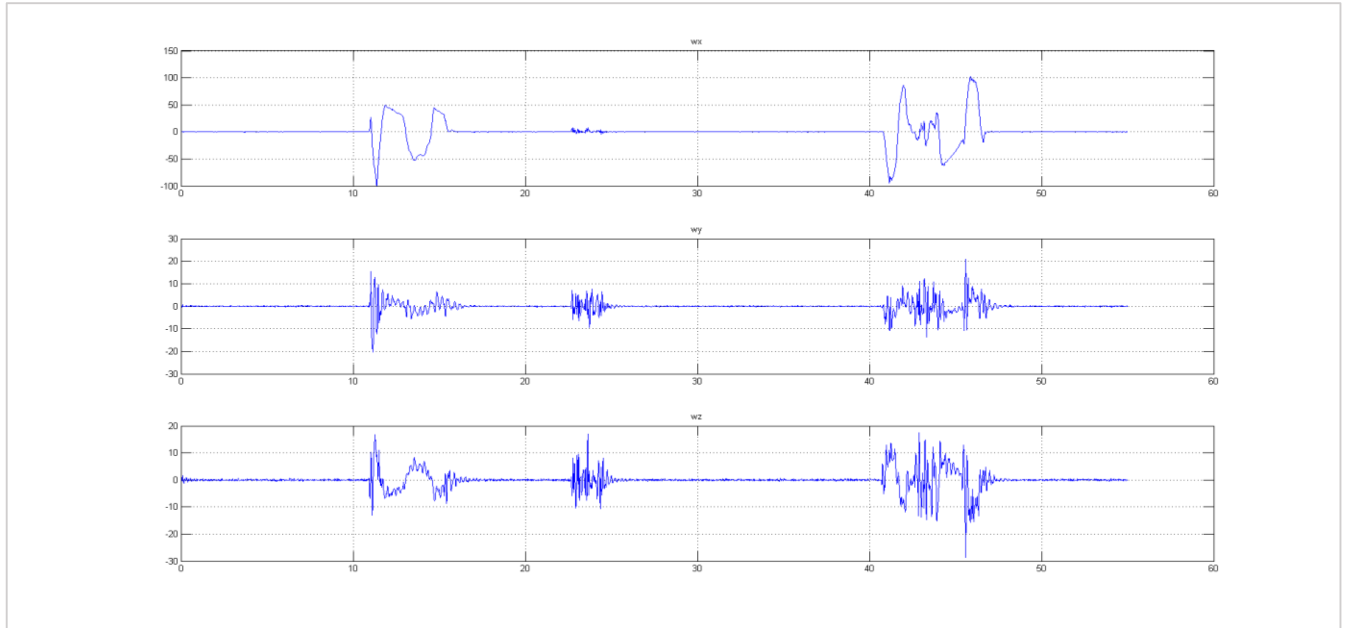
Слика 64. Синтетисане висина и брзина објекта на основу вредности са мерних уређаја

На слици 64. приказану су два графика чије се вредности израчунавају у реалном времену на основу вредности које сензори измере током лета. Први график представља висину ракете у сваком тренутку лета, а други график представља израчунату брзину ракете. Са добијених графика је очигледно да је други чип био оштећен и да је својим подацима реметио исправне податке који се налазе на првом чипу. Са графика се јасно види да је пројектил био мало немирнији у поређењу са оним из првог снимка, као и да има мало већу брзину и мањи упадни угао.

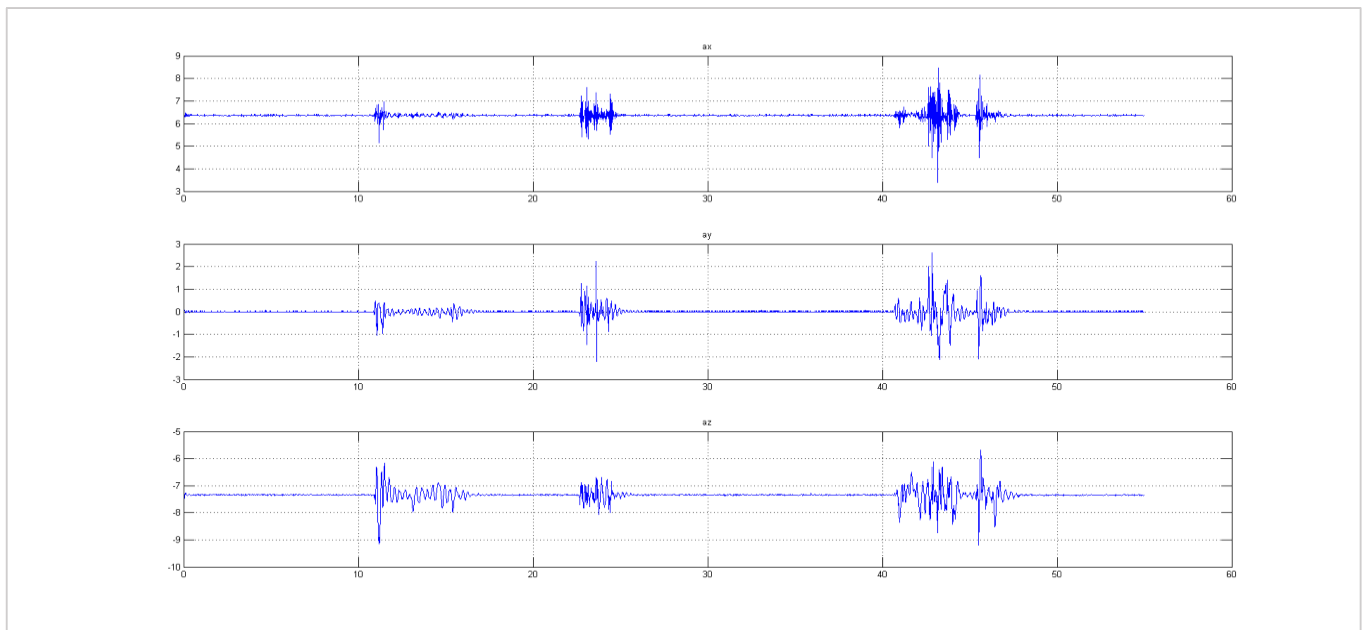
Способност црне кутије да уписује пакете алтернативно у меморијске чипове долази до изражаја у ситуацијама када долази до оштећења неког од чипова. Иако је у овом тесту дошло до отказа једног меморијског чипа, подаци снимљени на другом чипу су сасвим довољни за прецизну реконструкцију појава током самог експеримента. Једина разлика је дупло мања количина употребљивих података и дупло мања учестаност снимања, али то је нешто о чему корисник треба да води рачуна. На кориснику је да прецизно израчуна потребну дужину пакета и потребну учестаност снимања да би добио употребљиве податке и у случају отказа једног од чипова.

Два приказана теста припадају групи тестова из реалног света, док наредни тест припада групи лабораторијских тестова. Тест преставља снимање података са сензора приликом мировања навигационог уређаја у лабораторији. У одређеним тренуцима навигациони уређај је изведен из равнотежног положаја, и очекује се да се ти помераји виде на снимљеним подацима. Подаци очитани из меморије црне кутије су приказани на наредним графицима.

На слици 65. приказане су вредности угаоних убрзања измерених уграђеним троосним акцелерометром. Приказани графици редом представљају вредности убрзања измерених око прве, друге и треће осе сензора. Са графика је очигледно да је уређај три пута померан у различитим правцима.

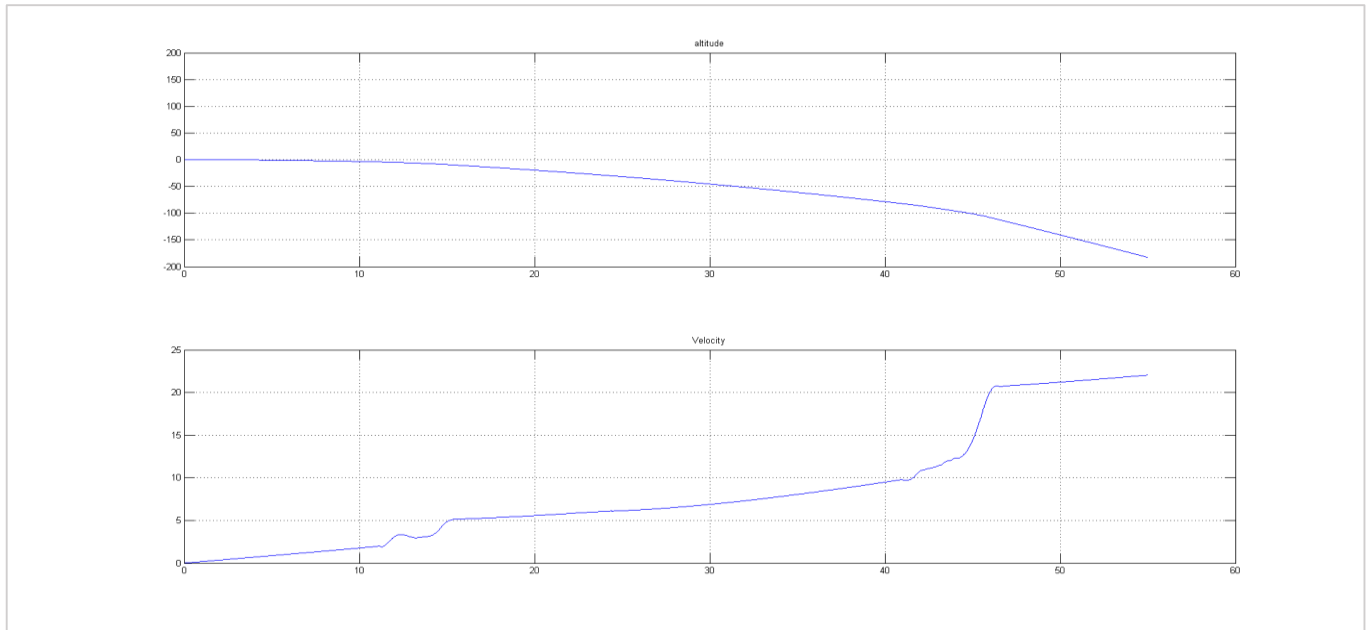


Слика 65. Угаоне брзине измерене уграђеним жироскопима



Слика 66. Угаона убрзања измерена уграђеним жироскопима

На слици 66. приказане су вредности угаоних брзина измерених уграђеним троосним жироскопом. Приказани графици редом представљају вредности измерене око прве, друге и треће осе сензора. Као и у случају вредности које су измерили акцелерометри и овде је очигледно да је уређај три пута померен из стања равнотеже.



Слика 67. Синтетисане висина и брзина објекта на основу вредности са мерних уређаја

Као што јасно видимо са графика угаоних брзина и убрзања уређај је три пута изведен из стања равнотеже и то је снимљено на самој црној кутији. На графицима се види и велика акумулација грешке на синтетисаним вредностима брзине и висине (слика 67). Оволика грешка није последица нагомилавања нумеричке грешке, већ својства жироскопа да слободно клизе када су у стању мировања.

11. Идеје за унапређивање система

Тренутно највећи проблем који систем поседује је провера присуства црне кутије у самом систему. Једино како је то могуће тренутно је слањем поруке и провером да ли је ехо сервис у функцији. Овакав вид провере оптерећује комуникациони канал и није изводљив у реалном времену. Провера функционалности црне кутије се ради само приликом конфигурисања система за предстојећи експеримент. Након тога пакети се шаљу секвенцијално све до краја експеримента, при чему не постоји никаква потврда о пријему пакета или о исправност саме црне кутије.

Решење овог проблема је увођење CAN (енг. Controller Area Network) протокола у систем. Током рада црна кутија би периодично слала CAN поруке централном уређају у систему и тиме обавештавала сам систем да је и даље у функцији. Изостанак поруке од црне кутије би централном контролном уређају сигнализирао да је у систему дошло до проблема. Такође уз помоћ CAN порука би корисник у реалном времену могао да добија информације о тренутној попуњености меморије, као и о статусу напајања свих уређаја у систему. Тиме би црна кутија поред пасивног уређаја за складиштење података постала и активни део самог система. Такође, UART протокол би могао да се користи искључиво за интеракцију са меморијом, док би сва подешавања могла да се извршавају уз помоћ CAN порука.

Поред ових унапређења, очигледан корак напред би представљала и употреба квалитетнијих вишеслојних штампаних плоча, чиме би се знатно смањили паразитски шумови у самом систему и чиме би се вероватно омогућила употреба прекида на пиновима за напајања, уместо досадашњег приступа са константним прозивањем уређаја.

12. Закључак

У овом раду приказано је прилагођавање µC/OS2 оперативног система микроархитектури ARM Cortex M3, као и његова примена у контроли дигиталне црне кутије. Описани су услови који су диктирали избор хардверских решења, као и сам хардвер који је коришћен. У складу са изабраним хардвером, детаљно су приказани сви неопходни протоколи за остваривање комуникације са спољним светом и комуникације са меморијом. Поред описа предности које се добијају употребом DMA контролера у комуникацији са спољним светом помоћу UART перифејског уређаја, у раду су описане предности које поседује FRAM меморија у комбинацији са брзим SPI протоколом у односу на стандардне флеш меморије приликом примене у условима где је велика брзина снимања главни захтев. Такође, описан је механизам надгледања напајања употребом општих улазно/излазних пинова.

У циљу прилагођавања µC/OS2 оперативног система микроархитектури ARM Cortex M3, детаљно цу описани заштита критичних секција, промена контекста, контрола интерних и екстерних прекида, контрола системског сата и јединице за дебаговање, тј. бројача инструкција уз чију помоћ се поуздано води рачуна о протоку времена током извршаввња оперативног система. У раду је објашњен начин памћења, креирања, организације и распоређивања послова у оквиру оперативног система. Посебна пажња посвећена је синхронизацији употребе дељених ресурса помоћу семафора, али и контроле синхронизације приликом приступа дељеним променљивама уз помоћ катанца. Проблем синхронизације дељених променљивих и потреба за увођењем катанаца су илустровани проблемом инверзије приоритета.

Након прилагођавања оперативног система, описан је поступак којим се врши испитивања исправности хардвера и редослед писања управљачких програма. Прво је написан управљачки програм за UART периферијски уређај којим се остварује комуникација са спољним светом. Том приликом дизајниран је комуникациони протокол којим се омогућава лако проверавање исправности пристиглих порука, чиме се подиже поузданост саме комуникације. Затим, детаљно је представљен протокол којим се комуницира са уграђеном FRAM трајном меморијом око кога су креиране све функције за једноставан упис података и читање података из меморије. Уз помоћ написаних управљачких програма, развијен је контролни програм који омогућава памћење примљених података у трајној меморији и њихово касније читање. Упоредо са развојем контролног програма, развијана је и апликација за PC рачунар уз чију помоћ је црна кутија дебагована. На крају, уведено је и надгледање напајања уз помоћ општих улазно/излазних пинова. Објашњене су предности и мане прозивања пина за напајање, као и предности и мане рада са прекидима када је у питању пин за напајање црне кутије. Постојање оперативног система и управљачких програма за периферијске уређаје који се користе омогућава корисницима који не познају свет угнежденог програмирања да лако и брзо прилагоде црну кутију својим потребама уколико уграђени контролни програм не задовољава њихове потребе.

Затим, приказани су резултати практичне примене црне кутије у експериментима. Приказани су резултати три експеримента. Два снимка из експеримената представљају полигонско испитивање ракете кратког домета. У првом експерименту црна кутија је преживела експеримент неоштећена, док је у другом била сломљена. Након оба испитивања црне кутије су прочитене и подаци добијени из прве су одговарали очекиваним резултатима, док су подаци из друге били неупотребљиви. Испоставило се да је један од меморијских чипова био оштећен и избацивањем података који

одговарају том чипу, добијени су подаци који одговарају реалним резултатима. Управо у ситуацијама у којима долази до оштећења чипова, постојање два одвојена FRAM меморијска модула долази до изражаја, јер су и поред оштећена чипа могли да се добију употребљиви подаци. Трећи експеримент је лабораторијски тест којим је само испитиван комуникациони канал између црне кутије и контролне јединице навигационог уређаја. Успешност трећег теста се огледала у томе што су снимљени подаци одговарали променама којима је навигациони уређај био подвргнут.

На крају, изложене су идеје којима се црна кутија може унапредити у смислу комуникације са спољним светом и у смислу поузданости. Главни циљеви унапређења су отклањање недостатака у начину комуникације црне кутије са спољним светом и повећање поузданости система, чији је коначан циљ лакше и брже откривање грешака када и уколико до њих дође.

13. Додатак А

Као што је у уводу наведено, поред оперативног система за црну кутију развијена је и комплетна апликација за тестирање црне кутије уз помоћ PC рачунара. Апликација омогућава тестирање сваке појединачне компоненте и функције црне кутије. Као развојна платформа изабрана је .NET платформа. Апликација комуницира са црном кутијом употребом COM порта на PC рачунару. За тестирање црне кутије потребно је да PC који се користи поседује COM порт или да постоји одговарајући RS-232 на USB конвертер.

Приликом покретања апликације, од корисника се очекује да изабере одговарајући COM порт, да одреди бодовну брзину и да дефинише употребу бита парности, као и број завршних битова. Након подешавања и покретања COM порта, омогућава се тестирање функционалности црне кутије. Приликом тестирања црне кутије, неопходно је прво покренути апликацију и подесити комуникациони канал, па тек онда укључити црну кутију. Наведени редослед је неопходан, јер црна кутија по укључивању шаље информације о стању меморије, тј. шаље се величина сваког од уграђених чипова, величина пакета из претходног експеримента као и да ли је било уписа или не. На основу примљених параметара апликација се аутоматски подешава, обавештава корисника о примљеним подацима и омогућава му тестирање и рад са UART периферијским уређајем.

13.1. Тестирање комуникације

Независно од статуса меморије, кориснику је омогућено тестирање UART периферијског уређаја помоћу ехо сервиса. Корисник има могућност да тестира комуникацију слањем једне поруке или може да покрене аутоматско тестирање којим се различита тест порука шаље сваке секунде. Када се корисник увери да комуникација ради, кликом на дугме може да прекине аутоматски тест. Такође, кориснику је омогућено брисање комплетне меморије кликом на дугме. Приликом слања команде за брисање црна кутија ће кориснику послати потврду о пријему команде и започети брисање свих ћелија у меморији. По пријему поруке о успешном завршетку процеса брисања, апликација ће обавестити корисника о окончању брисања и омогућити му даљи рад. Последња неконфигурациона команда коју корисник може да пошаље црној кутији је команда за читање већ уписаних података из претходног експеримента. Операција читања ће се извршити само у случају да су постављене заставице у меморији који означавају да постоји неки запис. У супротном, команда ће се игнорисати. По завршетку процеса читања, апликација обавештава корисника о завршетку процеса и креира бинарни фајл са прочитаним подацима.

13.2. Конфигурисање уређаја

Последњи тип поруке која се може послати серијском везом док је уређај у конфигурационом режиму је управо конфигурациона порука. Конфигурациона порука садржи величину пакета која ће се корисити у експерименту, режим у коме црна кутија треба да ради (алтернирајући или клон режим) као и информацију да ли треба да се води рачуна о напајањима у систему или не.

Да би се уређај конфигурисао, неопходно је прво тестирати UART комуникацију. Тест се једноставно врши ехо сервисом. Уколико је послата порука идентична примљеној, онда се кориснику омогућава да унесе конфигурационе параметре. Након уношења и слања конфигурационих параметара, апликација чека потврду од црне кутије да је конфигурациона порука примљена. По пријему потврде, апликација кориснику омогућава тестирање уграђене меморије. Приликом употребе црне кутије у експерименталним условима, након потврде о пријему конфигурационе поруке корисник

је дужан да сачека $500\mu s$ пре него што црној кутији пошаље било какву нову поруку. Ово време чекања је неопходно да би црна кутија могла да подеси DMA контролер за нову дужину пакета, као и да активира и подеси све потребне пинове и прекиде уколико се води рачуна о напајањима у систему.

13.3. Тестирање меморије

Након успешне конфигурације уређаја кориснику се омогућава тестирање уграђене меморије. Корисник може да шаље појединачне пакете црној кутији, као и да чита појединачне пакете из црне кутије. Такође, корисник има могућност аутоматског тестирања уписа у црну кутију. Кликом на дугме, апликација ће прерачунати максимални број пакета који могу да стану у црну кутију и у позадинском процесу ће их црној кутији послати. Сваки пакет који се пошаље, уписује се у бинарни фајл да би касније могао да се упореди са садржајем који ће из црне кутије бити прочитан. По окончању процеса уписа пакета у меморију, корисник кликом на дугме покреће читање црне кутије. По пријему сваког коректног пакета, исти ће бити уписан у бинарни фајл који ће се упоређивати са фајлом који је креиран приликом уписа у меморију. Након пријема потврде о крају слања прочитаних пакета, апликација аутоматски покреће процес за упоређивање два генерисана фајла. Када се упоређивање заврши кориснику ће бити приказани резултати. У случају да је дошло до грешке, односно до губитка пакета, фајлови ће бити различити и кориснику ће бити одштампана порука о грешци. У супротном, корисник ће добити поруку о успешно окончаном тесту. Уколико корисник жели само да прочита меморију, довољно је да пошаље команду за аутоматско читање, и по пријему потврде о завршетку читања апликација ће креирати бинарни фајл који корисник може даље да користи.

Кориснику је доступна и опција брисања меморије. Након пријема потврде о завршеном брисању од стране црне кутије, корисник може да настави са радом и тестирањем уређаја. Приликом слања команде за брисање меморије сам уређај се не ресетује, тако да изабрани режим рада и дужина пакета остају непромењени.

Поред тестирања саме меморије, кориснику је и даље доступно тестирање UART периферијског уређаја са новоизабраном величином пакета. Тестирање UART периферијског уређаја се и даље извршава уз помоћ ехо сервиса, било у ручном било у аутоматском режиму. Поред тестирања самог хардвера омогућено је и тестирање логике у самом оперативном систему црне кутије, тј. у оквиру PC апликације је тврдо кодирана порука са погрешном структуром чије би свако слање црној кутији морало да буде игнорисано.

Корисник има и опцију ресетовања црне кутије и поновног конфигурисања уколико жели да промени дужину пакета или режим рада. Након пријема потврде о успешном ресету, апликација се враћа назад у конфигурациони режим.

У сваком тренутку кориснику се приказује број послатих, као и број примељених пакета. Поред овога, корисник може да види број бајтова и број пакета који се шаљу у једној секунди, и на тај начин може да види да ли има простора за повећање учестаности слања или повећање дужине пакета. На левом панелу кориснику се приказују статусне информације, као и историја команди које су слате црној кутији, али и историја одговора црне кутије на примљене команде.

13.4. Подешавање напајања

У случају да је надгледање напајања укључено, апликација ће на панелу приказивати која су напајања активна, а која не. Информација о напајањима се налази на два претпоследња бајта у пакету. Поред приказивања стања на пиновима за напајања, апликација има и режим за подешавање волтаже на којој ће се пинови гасити.

Корисник има могућност да црној кутији пошаље команду за прелазак у режим који се користи за тестирање напајања. У том режиму црна кутија у реалном времену шаље информације које чита са пинова на којима се надгледају напајања, а апликација их приказује кориснику на одговарајућем панелу. Након подешавања самих напајањам корисник је дужан да црној кутији пошаље команду за престанак рада у режиму за подешавање напајања.

Подешавање волтаже на којој се гасе пинови је кључни корак у надгледању напајања, јер се вредности напајања не доводе директно на пинове процесора, већ пролазе кроз одређене компоненте на штампаној плочи. Компоненте су универзалне и отпорницима морају да се подесе тако да прекидају свој рад на унапред задатим волтажама, тј. морају исправно да сигнализирају губитак напајања на надгледаној компоненти.

Након проласка свих тестова и подешавања, можемо да будемо сигурни да је црна кутија исправна и спремна за учествовање у експериментима.

13.5. Тумачење примљених података

Након експеримента, једино што је потребно да се уради је читање саме меморије и тумачење добијених података. Да би се тумачење олакшало, апликација је интегрисана са Matlab-ом, чиме је корисницима омогућено лако и брзо визуелизовање снимљених података. Цртање података је могуће независно од постојања комуникације са црном кутијом. Довољно је да постоји прочитани бинарни фајл да би цртање било могуће. Поред бинарног фајла, у систему мора да постоји и конфигурациони .xml фајл. Конфигурациони фајл садржи опис пакета који се налази у меморији.

Аутоматски креирани xml фајл има следећу структуру:

```
<xml>
<!-- Duzina paketa u bajtovima -->
<PackageSize>84</PackageSize>
<!-- Broj promenljivih koje se koriste iz paketa -->
<Count>2</Count>

<!-- Opis promenljivih -->
<Variables>
    <!-- Promenljiva 0 -->
    <Var0>
        <!-- Naziv promenljive u generisanom fajlu -->
        <Name>a</Name>
        <!-- Tip podataka -->
        <Type>uint8</Type>
        <!-- Bajt od koga pocinje promenljiva u paketu -->
        <From>1</From>
```

```
        <!-- Bajt kojim se zavrsava promenljiva u paketu -->
        <To>1</To>
        <!-- Faktor skaliranja -->
        <Scale>1</Scale>
    </Var0>
    <!-- Promenljiva 0 -->
    <Var1>
        <!-- Naziv promenljive u generisanom fajlu -->
        <Name>t</Name>
        <!-- Tip podataka -->
        <Type>uint8</Type>
        <!-- Bajt od koga pocinje promenljiva u paketu -->
        <From>2</From>
        <!-- Bajt kojim se zavrsava promenljiva u paketu -->
        <To>2</To>
        <!-- Faktor skaliranja -->
        <Scale>1</Scale>
    </Var1>
</Variables>
<!-- Opis grafika koji se crtaju -->
<Plots>
    <!-- Ukupan broj grafika koji se crtaju -->
    <PlotCount>1</PlotCount>
    <!-- Redni broj grafika -->
    <Plot0>
        <!-- Zavisno promenljiva -->
        <Name>a</Name>
        <!-- Nezavisno promenljiva -->
        <Against>t</Against>
    </Plot0>
</Plots>
</xml>
```

Слика 68. Струткура xml фајла

Да би корисник визеулизовао снимљене податке, дужан је да на одговарајући начин опише пакет који се налази у меморији црне кутије. Корисник у првом кораку мора да унесе величину пакета у бајтовима као и број променљивих које се у пакету налазе. Након тога, корисник сваку променљиву мора да именује, да унесе редне бројеве бајтова на којима променљиве почињу и на којима се променљива завршавају, типове променљивих, затим податак да ли се променљива црта или не, као и променљиву од које је зависна. На основу ових вредности апликација креира конфигурациони .xml фајл који корисник може поново да користи. Приликом цртања, на основу података из конфигурационог фајла, апликација ће аутоматски креирати матлаб скрипт и покренути његово извршавање у оквиру већ покренуте матлаб инстанце.

13.6. Статистички подаци

Током испитивања различитих производа испоставило се да је анализа резултата експеримената најтежи део посла, јер су постојећи механизми памћења података у реалном времену били недовољно поуздани или су били исувише спори. Тада се дошло на идеју да се направи црна кутија која би учествовала у експериментима и која би била способна да снима податке у реалном времену са довољно високом учестаношћу. Укупно је било потребно мало више од четири месеца да би се од идеје стигло до реализације, не рачунајући време потребно за дизајн и производњу

штампаних плоча. Првих месец дана потрошени су на дебаговање хардвера и решавање свих проблема приликом производње штампаних плоча. Током наредних три месеца постепено је развијан програм описан у овом раду. Паралелно су развијани угнеждени програм за микроархитектуру ARM Cortex M3 и десктоп апликација за тестирање црне кутије. Након што је црна кутија у потпуности оспособљена, контролној апликацији за десктоп рачунар је додата могућност цртања снимљених података помоћу Matlab-а.

Угнеждена апликација је подељена у укупно 84 датотеке, од чега 44 .c датотеке, 37 .h датотека и 3 асемблерске датотеке. Наведена статистика укључује и датотеке које користе написани управљачки програми, а које су развијене од стране произвођача процесора. Контролна апликација за десктоп рачунар је подељена у 8 датотека и укупно има 3208 линија кода.

Литература

[1] Edsger W. Dijkstra, "Een algorithme ter voorkoming van de dodelijke omarming," http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[2] Miroslav Marić: Operativni sistemi, Matematički fakultet, 2015

[3] Jean J. Labrosse: MicroC/OS-II The Real-Time Kernel, Second Edition, CRC Press 2002

[4] Adam Osborne: An Introduction to microcomputers, Volume I Basic Concepts

[5] UM10360 LPC176x/5x User Manual Rev3.1 April 2014: http://www.nxp.com/documents/user_manual/UM10360.pdf

[6] Cypress Perform FM25H20 2Mb Serial 3v F-RAM Memory datasheet: http://www.farnell.com/datasheets/1929460.pdf

[7] FRAM – New Generation of Non-Volatile Memory: http://www.ti.com/lit/ml/szzt014a/szzt014a.pdf

[8] LPC1769/68/67/66/65/64/63 Rev. 9.6 August 2015 Product data sheet: http://www.nxp.com/documents/data_sheet/LPC1769_68_67_66_65_64_63.pdf

[9] Zilog Product specification Z8440/1/2/4, Z84C40/1/2/3/4. Serial input/output controller

[10] Детаљније о микроархитектури ARM Cortex M: http://www.arm.com/products/processors/cortex-m/index.php?tab=Resources

[11] Детаљније о микроархитектури ARM Cortex M3: http://www.arm.com/products/processors/cortex-m/cortex-m3.php